Linux Debug Training

Module 5

Tracing in Linux: Using Trace Tools to analyze Userspace and Kernel Activity

Introduction

- Tracing involves monitoring and recording the behavior and execution flow of software by capturing relevant data points for analysis.
- Tracing helps gain insights into both userspace and kernel activities, making it indispensable for system administrators, developers, and security professionals.
- Tracing assists in Debugging, Performance analysis and Security auditing.

What can we do with trace?

Trace allows us analyze many things:

- Library calls
- System calls (eg: open/read/write)
- Linux kernel function calls (eg: TCP/UDP path, kernel_clone)
- Userspace function calls (eg: malloc)
- Custom “events” that you’ve defined either in userspace or in the kernel.

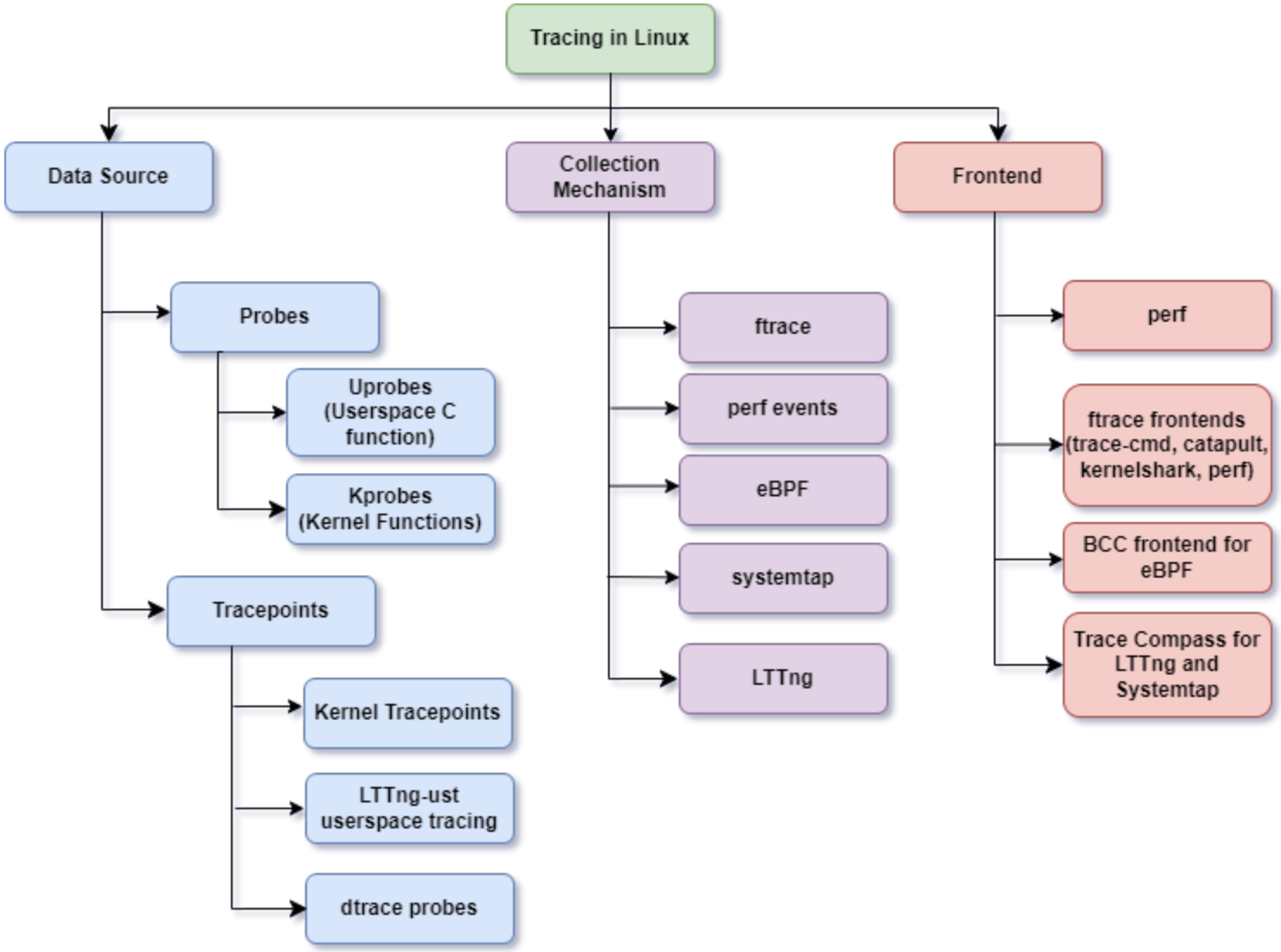
User space Tracing tools

- Strace
- Ltrace
- Uprobe
- Perf

Kernel Space Tracing tools

- Kprobe
- Perf
- ftrace
- eBPF
- LTTng
- Systemtap

Tracing landscape in Linux



Userspace application tracing

Strace

- strace is a widely used tracing tool used for tracing system calls and signals made by a process.
- It provides insights into how a program interacts with the kernel.

Usage

- Stracing a Program: `strace <Program Name>`
- Stracing a Running Process: `strace -p <pid>`
- Counting System Calls: `strace -c <Program Name>`
- Filtering System Calls: `strace -e trace=<System Call> <Program Name>`
- Using Time Format: `strace -T <Program Name>`
- Tracing Child Processes: `strace -f <Program Name>`

Strace example

```
manas@sandbox:~/work/Examples/helloworld$ strace ./hello_world
execve("./hello_world", ["./hello_world"], 0x7ffc6b5b8250 /* 26 vars */) = 0
brk(NULL)                               = 0x55f3833c5000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd5a367550) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7feeed160000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=77851, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 77851, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7feeed14c000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0i8\235HZ\227\223\333\350s\360\352,\223\340."..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=2216304, ...}, AT_EMPTY_PATH) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2260560, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7feeecf24000
mmap(0x7feeecf4c000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7feeecf4c000
mmap(0x7feeed0e1000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7feeed0e1000
mmap(0x7feeed139000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x214000) = 0x7feeed139000
mmap(0x7feeed13f000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7feeed13f000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7feeecf21000
arch_prctl(ARCH_SET_FS, 0x7feeecf21740) = 0
set_tid_address(0x7feeecf21a10)         = 24136
set_robust_list(0x7feeecf21a20, 24)     = 0
rseq(0x7feeecf220e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7feeed139000, 16384, PROT_READ) = 0
mprotect(0x55f383339000, 4096, PROT_READ) = 0
mprotect(0x7feeed19a000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7feeed14c000, 77851)           = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x3), ...}, AT_EMPTY_PATH) = 0
getrandom("\xe0\x0b\xb0\xa4\xdd\x35\x74\x16", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x55f3833c5000
brk(0x55f3833e6000)                     = 0x55f3833e6000
write(1, "\n", 1
)                       = 1
write(1, "Hello world!\n", 13Hello world!
)          = 13
exit_group(0)                           = ?
+++ exited with 0 +++
```

ltrace

- ltrace is a tracing tool used in Linux to trace library calls made by a program.
- It intercepts and displays the dynamic library calls (e.g., function calls from shared libraries) made during program execution.

Usage

- Tracing Specific Libraries: `ltrace -l <library> command`
- Filtering Functions: `ltrace -e <function> command`
- Counting Calls: `ltrace -c command`
- Tracing a Running Process: `ltrace -p <pid>`
- Please refer to the following code, which was used in the ltrace demonstration.

User-Level probe (uprobe)

- Uprobe is a dynamic tracing feature that allows the attachment of probes to user-level processes.
    - A probe is inserted into the executable binary (.text section), and the instruction at the specified offset is copied and replaced by a breakpoint.
    - When the breakpoint is encountered by a running process, the displaced instruction is executed, and control returns to the instruction following the breakpoint.
- Uprobes are typically employed for debugging, profiling, and performance monitoring of user-level applications.
- Uprobe serves as a data source and is frequently utilized by other tools like perf, ftrace, and bcc.
- Refer to uprobe documentation for more details.

Uprobe example

- Utilize the `nm` command to locate the memory offset of "first_function" function in memory example.

```
root@sandbox:/home/ubuntu/Examples/memory# nm memory | grep -i first_function
0000000000001292 T first_function
```

- Add and enable uprobe to address 0x1292, which corresponds to "first_function". (You may need to be root *sudo su*)

```
root@sandbox:/sys/kernel/debug/tracing# echo 'p /home/ubuntu/Examples/memory/memory:0x1292' > /sys/kernel/tracing/uprobe_events
root@sandbox:/sys/kernel/debug/tracing# echo 1 > /sys/kernel/tracing/events/uprobes/p_memory_0x1292/enable
```

- Activate tracing, and then run the program.

```
root@sandbox:/sys/kernel/debug/tracing# echo 1 > /sys/kernel/tracing/tracing_on
root@sandbox:/sys/kernel/debug/tracing# /home/ubuntu/Examples/memory/memory
The address of the main function is 0x56272af8c169
...
...
root@sandbox:/sys/kernel/debug/tracing# echo 0 > tracing_on
```

Uprobe example

- Confirm the uprobe hit in the trace file.

```
root@sandbox:/sys/kernel/debug/tracing# cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
# entries-in-buffer/entries-written: 3/3   #P:4
#
#                                _-----=> irqs-off/BH-disabled
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              ||| / _-=> migrate-disable
#                              |||| /     delay
#           TASK-PID     CPU#  |||||  TIMESTAMP  FUNCTION
#              | |         |   |||||     |         |
          memory-5595    [003] DNZff  2526.215701: p_memory_0x1292: (0x5566d246c292)
          memory-5599    [002] DNZff  2558.753812: p_memory_0x1292: (0x564a1490a292)
          memory-5600    [000] DNZff  2561.100325: p_memory_0x1292: (0x56272af8c292)
```

```
Disable the probe
echo 0 > /sys/kernel/debug/tracing/events/uprobes/p_memory_0x1292/enable
```

perf tool

- perf is a powerful performance analysis and tracing tool for Linux systems.
- It is a part of Linux kernel and provides deep insights into various aspects of system performance.
- The standard Linux perf tool package offers basic support. Following these instructions, building your own version is recommended.

Key Capabilities:

- **Profiler**: Gather detailed performance data for CPU, memory, and more.
- **Tracing**: Trace system calls, interrupts, and events for in-depth analysis.
- **Hardware Performance Monitoring**: Utilize hardware counters for low-level performance metrics.
- **Event-based Profiling**: Customize profiling with specific events of interest.
- **Call-graph Profiling**: Generate call-graphs to identify function-level performance bottlenecks.

perf_event (event tracing)

Hardware Events: Monitoring CPU performance counters for metrics like cycles, cache hits, and branch instructions.

Software Events: Tracing low-level events based on kernel counters, such as CPU migrations, minor and major faults.

Kernel Tracepoint Events: Utilizing static kernel-level instrumentation points strategically placed in the kernel code.

User Statically-Defined Tracing (USDT):Defining static tracepoints for user-level programs and applications.

Dynamic Tracing: Dynamically instrumenting software using frameworks like kprobes for the kernel and uprobes for user-level software.

Timed Profiling: Collecting snapshots at a specified frequency, often used for CPU usage profiling, employing custom timed interrupt events with perf record -FHz.

User-level tracing using perf

- With dynamic tracing feature, we can trace user-level applications by adding tracepoints using the `perf probe` command. (Demonstration code)

```
# Add a tracepoint for the user-level malloc() function from libc:
perf probe -x /usr/lib/x86_64-linux-gnu/libc.so.6 --add malloc

# Add a tracepoint for the user-level malloc() function with argument to determine allocation size on an x86_64 system.
perf probe -x /usr/lib/x86_64-linux-gnu/libc.so.6 --add='malloc allocated=%di:u64'

# Add a tracepoint for myfunc() return, and include the retval as a string:
perf probe 'myfunc%return +0($retval):string'

# Add a return probe to my_func(). On X86_64 register rax is used for return value.
perf probe -x app my_func%return ret=%ax

# Add a probe on my_variable defined in my_func.
perf probe -x app my_func:3 my_var
```

- `perf record` command captures performance events specified by the user, collecting data for later analysis with tools like `perf report`.

```
# Tracing malloc event system-wide.
perf record -e probe_libc:malloc -a
```

User-level tracing using perf

- Analyze the perf.data using `perf report` command.

```
root@sandbox:/home/ubuntu# perf report -n

Samples: 20K of event 'probe_libc:malloc', Event count (approx.): 20091
Overhead       Samples  Command          Shared Object  Symbol
  46.22%          9286  gjs              libc.so.6      [.] malloc
  13.23%          2659  gdbus            libc.so.6      [.] malloc
   7.65%          1536  teamviewerd      libc.so.6      [.] malloc
   7.59%          1524  tracker-miner-f  libc.so.6      [.] malloc
   4.92%           989  tracker-extract  libc.so.6      [.] malloc
   4.81%           967  pool-tracker-mi  libc.so.6      [.] malloc
   4.56%           917  gnome-shell      libc.so.6      [.] malloc
   2.55%           512  gnome-terminal-  libc.so.6      [.] malloc
   2.45%           492  systemd-oomd     libc.so.6      [.] malloc
   2.25%           453  dbus-daemon      libc.so.6      [.] malloc
   1.12%           226  ibus-daemon      libc.so.6      [.] malloc
   0.71%           143  Qt bearer threa  libc.so.6      [.] malloc
   0.50%           100  ibus-engine-sim  libc.so.6      [.] malloc
   0.44%            89  systemd-resolve  libc.so.6      [.] malloc
   0.28%            57  Monitor thread   libc.so.6      [.] malloc
   0.26%            53  irqbalance       libc.so.6      [.] malloc
   0.24%            49  thermald         libc.so.6      [.] malloc
   0.14%            28  gmain            libc.so.6      [.] malloc
   0.05%            11  pool-tracker-ex  libc.so.6      [.] malloc
```

Kernel space tracing

Kernel Probes

- Kprobes enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively.
- Two type of Kprobes:
  - kprobes: A kprobe can be inserted on virtually any instruction in the kernel.
  - kretprobes: Also called return probes, fires when a specified function returns.
- Th following Kernel config option should be enabled.
  - CONFIG_KPROBES
  - CONFIG_MODULES
  - CONFIG_MODULE_UNLOAD
  - CONFIG_KALLSYMS
  - CONFIG_DEBUG_INFO
- Refer to Krpobe documentation for more details.

Registering kprobe and Kretprobe

- Dynamic registration of kprobes and kretprobes involves loading a kernel module.
- For kprobe, register a struct kprobe with register_kprobe(). The probe should be unregistered at module exit using unregister_kprobe().

```
struct kprobe kp = {
.symbol_name = "kernel_clone",
.pre_handler = pre_handler;
.post_handler = post_handler;
};
```

- For kretprobe, register a struct kretprobe with register_kretprobe(). The probe should be unregistered at module exit using unregister_kretprobe().

```
struct kretprobe probe = {
.kp.symbol_name = "kernel_clone",
.entry_handler = kret_entry;
.handler = kret_exit;
};
```

- Refer to the samples/kprobes/ sub-directory for kprobe examples.

Kprobe events

- Kprobe event-based mechanism allows for dynamic tracing of the Linux kernel without the need for recompiling or reloading kernel modules.
- The provided example registers a custom kprobe and kretprobe on the "kernel_clone" symbol.

```
echo 'p:kernelclone kernel_clone' >> /sys/kernel/debug/tracing/kprobe_events
echo 'r:cloneretprobe kernel_clone $retval' >> /sys/kernel/debug/tracing/kprobe_events
```

- Enable the custom kprobe events

```
echo 1 > /sys/kernel/debug/tracing/events/kprobes/kernelclone/enable
echo 1 > /sys/kernel/debug/tracing/events/kprobes/cloneretprobe/enable
```

- Verify the successful registration of the kprobe event.

```
root@sandbox:/sys/kernel/debug/tracing# cat set_event
kprobes:cloneretprobe
kprobes:kernelclone
```

Kprobe events

- Trace the kprobe event whenever there is a new process executed.

```
root@sandbox:/sys/kernel/tracing# cat trace
#           TASK-PID     CPU#  |||||  TIMESTAMP  FUNCTION
            bash-4976    [003] .....  1019.951644: kernelclone: (kernel_clone+0x0/0x3e0)
            bash-4976    [003] .....  1019.952073: cloneretprobe: (__do_sys_clone+0x66/0xa0 <- kernel_clone) arg1=0x14aa
            bash-4976    [003] .....  1024.671996: kernelclone: (kernel_clone+0x0/0x3e0)
            bash-4976    [003] .....  1024.672413: cloneretprobe: (__do_sys_clone+0x66/0xa0 <- kernel_clone) arg1=0x14ab
            bash-5002    [003] .....  1030.818426: kernelclone: (kernel_clone+0x0/0x3e0)
            bash-5002    [003] .....  1030.818836: cloneretprobe: (__do_sys_clone+0x66/0xa0 <- kernel_clone) arg1=0x14ac
            bash-4976    [002] .....  1034.867396: kernelclone: (kernel_clone+0x0/0x3e0)
            bash-4976    [002] .....  1034.867795: cloneretprobe: (__do_sys_clone+0x66/0xa0 <- kernel_clone) arg1=0x14ad
```

- In the above example, arg1 of kretprobe signifies the return value, which is the pid in the kernel_clone() call.
- Specify function names, and the kernel dynamically resolves their addresses, accommodating changes like Kernel Address Space Layout Randomization (KASLR).
- `/proc/kallsyms` provides all the kernel symbols.
- Please refer to the following code, used in the kprobe demonstration.

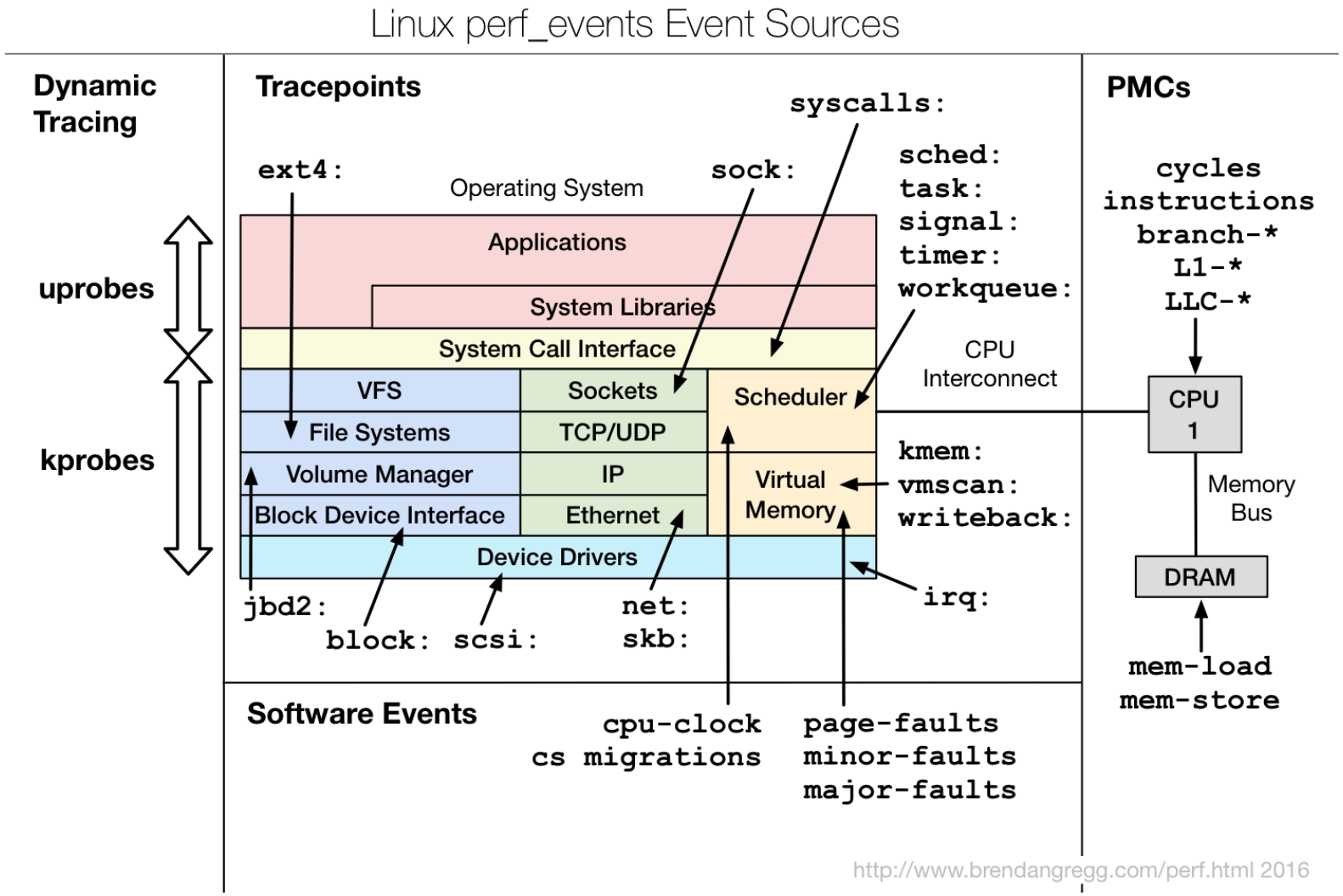
Perf Events Tracing

- In the kernel space, perf supports both static and dynamic tracing.
- Static tracing involves pre-defined tracepoints in the Linux kernel.
- `perf list` provides the list of events and tracepoints available in the kernel.
  - `perf list hw cache` : Hardware events.
  - `perf list sw` : Software events.
  - `perf list tracepoint` : All Static tracepoints.
  - `perf list 'sched:*'` : Static tracepoints related to schedule class.
  - `perf list 'syscalls:*'` : Static tracepoints related to sysclass.
- Dynamic tracing includes tracepoints created using kprobes, allowing on-the-fly instrumentation of functions for detailed analysis.

Perf Events Tracing

- Dynamic tracing relies on kernel probes (kprobes) and therefore requires CONFIG_KPROBE to be enabled in the kernel for probe insertion.
- Without debug info, perf enables dynamic creation of tracepoints on all symbols and registers.
- For tracing functions and recording variable content by name, perf requires a kernel compiled with CONFIG_DEBUG_INFO.
- In perf, the vmlinux file is essential for symbol resolution, call stack unwinding, dynamic probing, and accurate flame graph generation. Users can specify the vmlinux path using the `-k <vmlinux_file>` option.
- Please refer to these instructions to download vmlinux file on the Ubuntu 22.04 Operating System.

Perf Events Source



Perf Static Tracepoints

- The following examples demonstrate static tracing with perf.
  - `perf record -e sched:sched_process_exec -a` : Trace new processes, until ctrl+c.
  - `perf record -e block:block_rq_issue -ag` : Trace disk I/O with call graph, until ctrl+c.
  - `perf record -e page-faults -ag` : Sample page faults with stack traces, until ctrl+c.
  - `perf record -e syscalls:sys_enter_openat md5sum /bin/ls` : Record all syscalls:sys_enter_openat events for md5sum command.

- Brendan Gregg's post on perf, available here, offers extensive insights into the usage of perf.

Perf Static Tracepoint Example

- `perf record` captures and records performance events, system calls, or custom probes for later analysis into a file named perf.data.
- Trace new processes, until ctrl+c is pressed.

```
root@sandbox:/home/ubuntu/work/linux-kernel/linux-6.2/tools/perf# ./perf record -e sched:sched_process_exec -a
```

- The above command will record the instances of `sched:sched_process_exec` into perf.data file.
- Display the collected samples using `perf script` command

```
root@sandbox:/home/ubuntu/work/linux-kernel/linux-6.2/tools/perf# ./perf script
             ls  5377 [000]  1295.514952: sched:sched_process_exec: filename=/usr/bin/ls pid=5377 old_pid=5377
             cp  5378 [001]  1298.221744: sched:sched_process_exec: filename=/usr/bin/cp pid=5378 old_pid=5378
             mv  5379 [002]  1301.121680: sched:sched_process_exec: filename=/usr/bin/mv pid=5379 old_pid=5379
          touch  5381 [001]  1309.541083: sched:sched_process_exec: filename=/usr/bin/touch pid=5381 old_pid=5381
            cat  5382 [002]  1315.407542: sched:sched_process_exec: filename=/usr/bin/cat pid=5382 old_pid=5382
```

Perf Dynamic Tracepoint

- `perf probe` enables dynamic tracing in the Linux kernel by setting up probes on specified functions.
- `perf probe --funcs` lists all the kernel symbols that can be probed.
- The following examples demonstrate dynamic tracing with perf.
  - `perf probe tcp_sendmsg` : Add a tracepoint for the kernel tcp_sendmsg() function entry
  - `perf probe 'tcp_sendmsg%return $retval'` : Add a tracepoint for tcp_sendmsg() return, and capture the return value.
  - `perf probe -V do_sys_open` : Show available variables for do_sys_open().
  - `perf probe 'do_sys_open filename:string'` : Add a tracepoint for do_sys_open() with the filename as a string.

Perf Dynamic Tracepoint

- `perf probe -l` lists all the available dynamic probes in the Linux Kernel.

```
root@sandbox:/home/ubuntu/work/linux-kernel/linux-6.2/tools/perf# ./perf probe -l
...
  probe:do_sys_open    (on do_sys_open@linux-hwe-6.2-6.2.0/fs/open.c with filename_string)
  probe:tcp_sendmsg    (on tcp_sendmsg@net/ipv4/tcp.c)
  probe:tcp_sendmsg__return (on tcp_sendmsg%return@net/ipv4/tcp.c with arg1)
```

- Probe entries are available in `/sys/kernel/debug/tracing/events/probe` directory.

```
root@sandbox:/sys/kernel/debug/tracing/events/probe# ls
do_sys_open  enable  filter  tcp_sendmsg  tcp_sendmsg__return
```

- Use `perf probe -d` to delete an existing probe.

```
perf probe -d probe:tcp_sendmsg
```

Perf Dynamic Trace Example

- Add a new tracepoint on do_sys_open() with filename as a variable to print.
  `perf probe 'do_sys_open filename:string'`
- Capture the tracepoint using perf record.
  `perf record -e probe:do_sys_open -aR`
- Display recorded tracepoint using perf script.

```
root@sandbox:/home/ubuntu/work/linux-kernel/linux-6.2/tools/perf# ./perf script
     teamviewerd  1940 [001]  8778.753234: probe:do_sys_open: (ffffffffaa07aba9) filename_string="/dev/fb0"
     systemd-oomd   795 [001]  8778.792533: probe:do_sys_open: (ffffffffaa07aba9) filename_string="/proc/meminfo"
     vim  7724 [003]  8785.669457: probe:do_sys_open: (ffffffffaa07aba9) filename_string="/home/ubuntu/.viminfo"
     vim  7724 [003]  8785.670449: probe:do_sys_open: (ffffffffaa07aba9) filename_string="/etc/passwd"
     vim  7724 [003]  8785.670654: probe:do_sys_open: (ffffffffaa07aba9) filename_string="manas"
...
```

Perf Trace

- `perf trace` provides a real-time trace of system events, including system calls, signals, and other kernel or user-space events.
- Example 1: Trace all sys_enter_openat() events during the execution of the command `sha1sum perf.data`.

```
./perf trace -e 'syscalls:sys_enter_openat' sha1sum perf.data
     0.000 sha1sum/8460 syscalls:sys_enter_openat(dfd: CWD, filename: "", flags: RDONLY|CLOEXEC)
     0.046 sha1sum/8460 syscalls:sys_enter_openat(dfd: CWD, filename: "", flags: RDONLY|CLOEXEC)
     0.317 sha1sum/8460 syscalls:sys_enter_openat(dfd: CWD, filename: "", flags: RDONLY|CLOEXEC)
     0.400 sha1sum/8460 syscalls:sys_enter_openat(dfd: CWD, filename: "")
41b46b3efadab5256723e7bd6ad0c0fe1f632cdd  perf.data
```

- Example 2: Trace all network events while using the `ping` command.

```
./perf trace -e "net:*" ping -c 1 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
     0.000 ping/8497 net:net_dev_queue(skbaddr: 0xffff90d9726a5600, len: 98, name: "wlp2s0")
     0.011 ping/8497 net:net_dev_start_xmit(name: "wlp2s0", skbaddr: 0xffff90d9726a5600, protocol: 2048, len: 98, network_offset: 14, transport_offset_valid: 1, transport_offset: 34)
     0.035 ping/8497 net:net_dev_xmit(skbaddr: 0xffff90d9726a5600, len: 98, name: "wlp2s0")
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=2.11 ms

--- 192.168.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.108/2.108/2.108/0.000 ms
```

ftrace

- ftrace is a built-in Linux kernel tracing framework designed for performance analysis, tracing and debugging.
- It provides a wide range of tracing features, including function tracing, function graph tracing, event tracing, and more.
- ftrace uses tracefs filesystem and allows configuration and control through simple file system operations.
    - `mount -t tracefs nodev /sys/kernel/tracing`
- ftrace utilizes a ring buffer mechanism for efficient data storage.
- trace-cmd CLI and the Kernelshark GUI facilitates a more streamlined process for recording and visualizing tracing data.
- Please refer to the following code, used in the ftrace demonstration.

ftrace controls

- ftrace output and its controls are accessible through files in `/sys/kernel/tracing`. Access to these files requires root user access.
  - available_tracers: Lists the tracers compiled into the kernel.
  - current_tracer: Currently active tracer.
  - tracing_enabled: Represents the status of tracing (enable/disable).
  - trace: Displays the acquired trace data.
  - set_ftrace_filter: Tracing on selected functions with a tailored filter.
  - set_graph_function: Graph only the specified functions.
  - trace_pipe: Same as trace. Every read from trace_pipe is consumed. This means that subsequent reads will be different. The trace is live.
  - trace_entries: Number of trace entries available or being used

ftrace tracers

- Tracers in ftrace offers diverse methods to capture, analyze, and visualize kernel events.
    - nop: Disables tracing, temporary suspension of tracing operations.
    - function: Traces function calls, useful in tracing the execution flow.
    - function_graph: Graphically represents function call relationships for a detailed execution overview.
    - latency: Specialized tracer for measuring system and task latencies.
    - hwlat: Monitors hardware latency.
    - mmiotrace: Traces memory-mapped I/O operations, aiding in analyzing interactions between the kernel and hardware devices.
    - irqsoff: Captures events with interrupts disabled in the kernel.
- Please refer to the trace/ftrace kernel documentation for more details.

function and function_graph tracer

- function tracer captures function call events in the kernel. It provides a straightforward trace of executed functions.
- function_graph tracer graphically depicts hierarchical relationships among function calls, providing a visual representation of the execution flow in the Linux kernel.

```
# tracer: function_graph
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |

 0)               |  sys_open() {
 0)               |    do_sys_open() {
 0)               |      getname() {
 0)               |        kmem_cache_alloc() {
 0)   1.382 us    |          __might_sleep();
 0)   2.478 us    |        }
 0)               |        strncpy_from_user() {
 0)               |          might_fault() {
 0)   1.389 us    |            __might_sleep();
 0)   2.553 us    |          }
 0)   3.807 us    |        }
 0)   7.876 us    |      }
```

irqsoff tracer

- The irqsoff tracer tracks the time for which interrupts are disabled.
- It is crucial for identifying and pinpointing potential bottlenecks related to latency-sensitive issues arising when interrupts are disabled in the Linux kernel.
- `IRQSOFF_TRACER` is necessary to enable the irqsoff tracer.
- Refer to the trace/ftrace kernel documentation for detailed information on available tracer.

irqsoff tracer example

```
 # cat trace
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.8.0-test+
# --------------------------------------------------------------------
# latency: 16 us, #4/4, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: swapper/0-0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: run_timer_softirq
#  => ended at:   run_timer_softirq
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
  <idle>-0       0d.s2    0us+: _raw_spin_lock_irq <-run_timer_softirq
  <idle>-0       0dNs3   17us : _raw_spin_unlock_irq <-run_timer_softirq
  <idle>-0       0dNs3   17us+: trace_hardirqs_on <-run_timer_softirq
  <idle>-0       0dNs3   25us : <stack trace>
...
```

trace-cmd

- ftrace controls can be accessed through the files available in `/sys/kernel/tracing` .
- trace-cmd is an another alternative command line method to interact with ftrace infrastructure.
- trace-cmd enhances usability with higher-level commands for simplified interaction with Ftrace features.
- It offers a command-line interface for recording, extracting, and analyzing trace data.
- It can be installed on Ubuntu 22.04 using the following command.
  `sudo apt-get install trace-cmd`

trace-cmd usage

- List available tracer: `trace-cmd list -t`

```
root@sandbox:~# trace-cmd list -t
timerlat osnoise hwlat blk mmiotrace function_graph wakeup_dl wakeup_rt wakeup function nop
```

- List available function to trace: `trace-cmd list -f`

```
root@sandbox:~# trace-cmd list -f
__traceiter_initcall_level
__traceiter_initcall_start
...
root@sandbox:~# trace-cmd list -f | wc -l
76608
```

- List available events to trace: `trace-cmd list -e`

```
root@sandbox:~# trace-cmd list -e
...
syscalls:sys_exit_sysinfo
syscalls:sys_enter_sysinfo
...
```

trace-cmd usage

- Trace all functions using function tracer.
  ```
  trace-cmd start -p function
  ```
- Trace all functions using function_graph tracer.
  ```
  trace-cmd start -p function_graph
  ```
- Trace using the function_graph tracer with a maximum function depth of 4.
  ```
  trace-cmd start -p function_graph --max-graph-depth 4
  ```
- Trace a specific function (for example: kfree()).
  ```
  trace-cmd record -l kfree -p function_graph
  ```
- Trace a specific PID.
  ```
  trace-cmd record -P {PID} -p function_graph
  ```
- Trace specified command.
  ```
  trace-cmd record -p function_graph sha256sum ~/.bashrc
  ```

trace-cmd usage

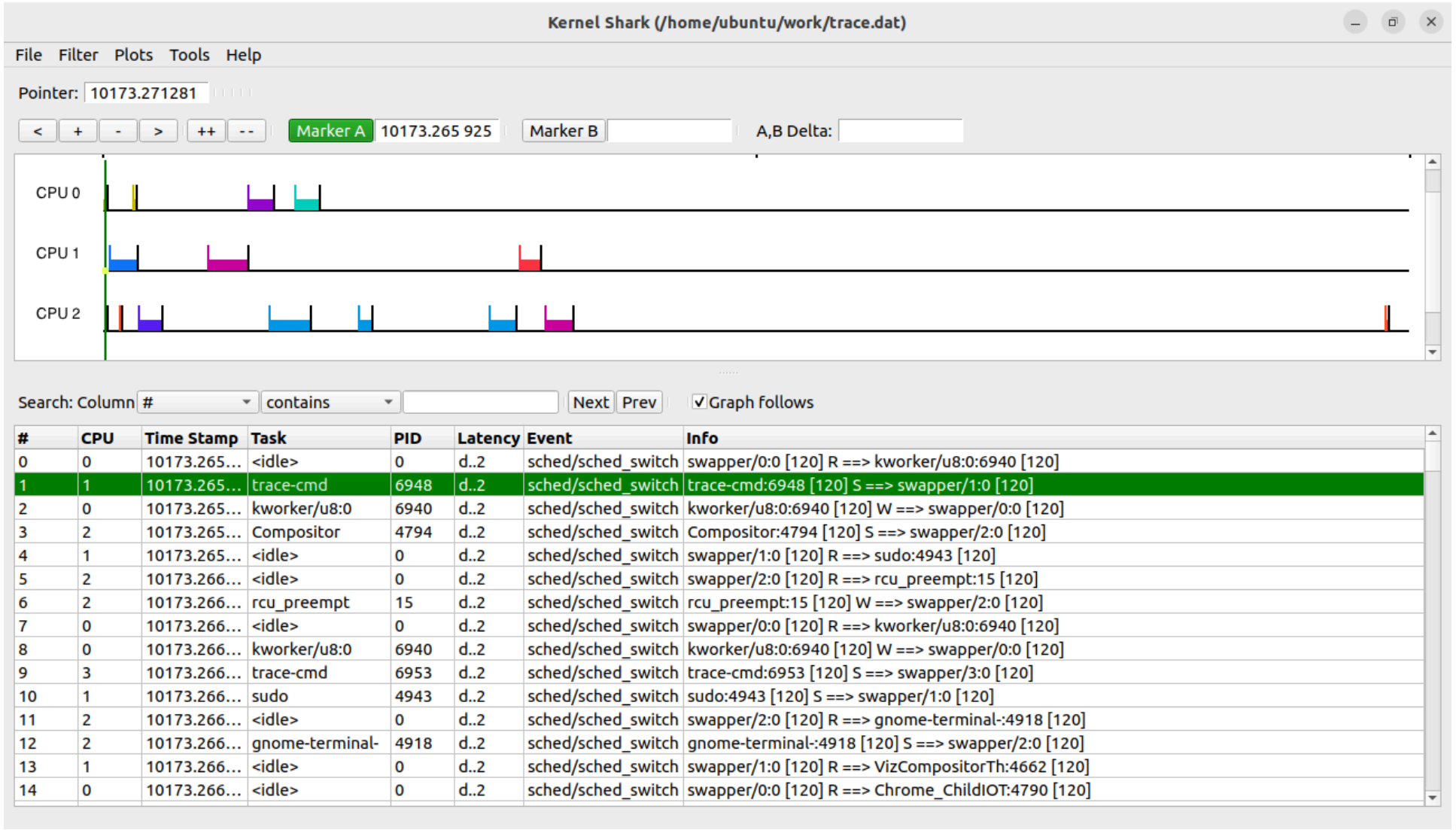
- To remove all tracers and reset ftrace buffers.
  ```
  trace-cmd reset
  ```
- To display the trace contents we can extract them to a file and then display them in human readable form.
  ```
  trace-cmd extract -o trace.dat
  trace-cmd report -i trace.dat
  ```
- Or to directly display the contents of the kernel tracing buffer
  ```
  trace-cmd show
  ```
- To record a specific trace event
  ```
  trace-cmd record -e sched:sched_switch
  ```

Kernelshark

- Kernelshark is a GUI tool for visualizing and analyzing trace data captured by ftrace.
- It provides an interactive timeline view of trace events for easy navigation.
- Allows users to filter, search, and zoom into specific events for detailed analysis.
- Displays information about function calls, latencies, and other kernel activities.
- KernelShark can analyse trace data files generated by trace-cmd; however, one must use a version of trace-cmd that is compatible with Kernelshark. We would recommend trace-cmd-stable-v2.9 and Kernelshark 2.3.1. The required versions can be downloaded from the links below:

```
https://git.kernel.org/pub/scm/utils/trace-cmd/trace-cmd.git/
https://git.kernel.org/pub/scm/utils/trace-cmd/kernel-shark.git/
```

Kernelshark GUI



#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	0	10173.265...	<idle>	0	d..2	sched/sched_switch	swapper/0:0 [120] R ==> kworker/u8:0:6940 [120]
1	1	10173.265...	trace-cmd	6948	d..2	sched/sched_switch	trace-cmd:6948 [120] S ==> swapper/1:0 [120]
2	0	10173.265...	kworker/u8:0	6940	d..2	sched/sched_switch	kworker/u8:0:6940 [120] W ==> swapper/0:0 [120]
3	2	10173.265...	Compositor	4794	d..2	sched/sched_switch	Compositor:4794 [120] S ==> swapper/2:0 [120]
4	1	10173.265...	<idle>	0	d..2	sched/sched_switch	swapper/1:0 [120] R ==> sudo:4943 [120]
5	2	10173.266...	<idle>	0	d..2	sched/sched_switch	swapper/2:0 [120] R ==> rcu_preempt:15 [120]
6	2	10173.266...	rcu_preempt	15	d..2	sched/sched_switch	rcu_preempt:15 [120] W ==> swapper/2:0 [120]
7	0	10173.266...	<idle>	0	d..2	sched/sched_switch	swapper/0:0 [120] R ==> kworker/u8:0:6940 [120]
8	0	10173.266...	kworker/u8:0	6940	d..2	sched/sched_switch	kworker/u8:0:6940 [120] W ==> swapper/0:0 [120]
9	3	10173.266...	trace-cmd	6953	d..2	sched/sched_switch	trace-cmd:6953 [120] S ==> swapper/3:0 [120]
10	1	10173.266...	sudo	4943	d..2	sched/sched_switch	sudo:4943 [120] S ==> swapper/1:0 [120]
11	2	10173.266...	<idle>	0	d..2	sched/sched_switch	swapper/2:0 [120] R ==> gnome-terminal-:4918 [120]
12	2	10173.266...	gnome-terminal-	4918	d..2	sched/sched_switch	gnome-terminal-:4918 [120] S ==> swapper/2:0 [120]
13	1	10173.266...	<idle>	0	d..2	sched/sched_switch	swapper/1:0 [120] R ==> VizCompositorTh:4662 [120]
14	0	10173.266...	<idle>	0	d..2	sched/sched_switch	swapper/0:0 [120] R ==> Chrome_ChildIOT:4790 [120]

trace_printk()

- trace_printk() is a kernel facility for printing debug messages in the trace buffer without causing deadlocks.
- It is a part of the dynamic debugging infrastructure, allowing dynamic control over printed messages.
- trace_prink() uses format specifiers similar to printk for customizable output.

```
#include <linux/ftrace.h>
void foofunc()
{
  trace_printk("I am a comment!\n");
}
```

- The trace buffer outputs the following when using the function_graph tracer.

```
 1)               |              foofunc() {
 1)               |                 /* I am a comment! */
 1)   1.345 us    |              }
```

eBPF (Extended Berkeley Packet Filter)

- eBPF is a framework in Linux kernel which enables the execution of sandboxed programs within the kernel's privileged context.
- It allows the secure and efficient extension of kernel capabilities without the need to modify kernel source code or load kernel modules.
- eBPF programs are event-driven and are run when the kernel or an application passes a certain hook point including system calls, function entry/exit, kernel tracepoints, network events.
- eBPF programs can also be attached with kprobe, uprobe or tracepoints.
- Widely used in high-performance networking, load-balancing, helping application developers trace applications, providing insights for performance troubleshooting and much more.

eBPF overview

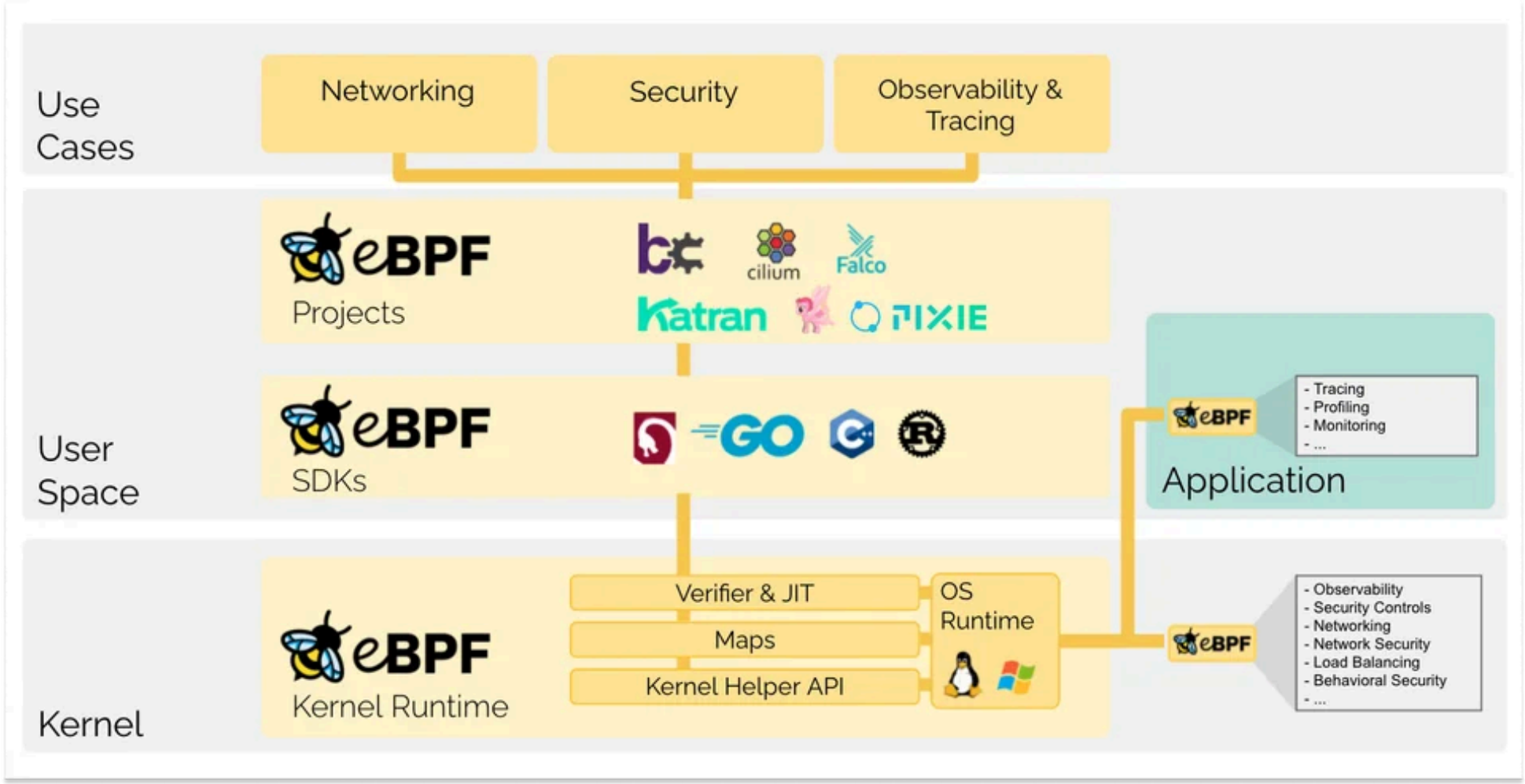


Image Credit: https://ebpf.io/

eBPF mechanism

- The eBPF program can be loaded into the kernel using the bpf system call.
- Before attaching the program to the hook, the eBPF verifier in the Linux kernel verifies it for the following.
    - The process loading the eBPF program holds the required privileges.
    - The program does not crash.
    - The program always runs to completion (doesn't loop forever).
- JIT compilation optimizes eBPF programs for efficient execution, translating bytecode into machine-specific instructions, akin to natively compiled kernel code or loaded kernel modules.
- eBPF programs utilize eBPF maps to share and store data, enabling interaction with a variety of data structures such as Hash tables, Arrays, Ring Buffer, Stack Trace which is accessible between user space, kernel space and eBPF programs.

eBPF usage

Primary eBPF frontends, listed in order of increasing complexity.

BCC Tool

- BCC tools is a toolbox of ready-to-use tools written in BPF. It consists of both a single purpose and a generic tool which simplify monitoring, analysis, and troubleshooting on Linux systems.

bpftrace

- bpftrace is more dynamic and useful for running custom one-liners and short scripts.

BCC programming

- Writing your own custom eBPF program using supported infrastructure.

BCC (BPF Compiler Collection) Tools

- BCC tools is a recommeded frontend for eBPF tracing.
- It offers over 70 ready-to-use tools written in BPF, catering to various purposes such as performance monitoring, debugging, and network analysis.
- To install BCC tools on Ubuntu, execute `sudo apt-get install bpfcc-tools`
- Below mentioned is a generic checklist of BCC tools for performance investigations.

- execsnoop
- opensnoop
- ext4slower (or btrfs*, xfs*, zfs*)
- biolatency
- biosnoop
- cachestat
- tcpconnect
- tcpaccept
- tcpretrans
- runqlat
- profile

BCC tools

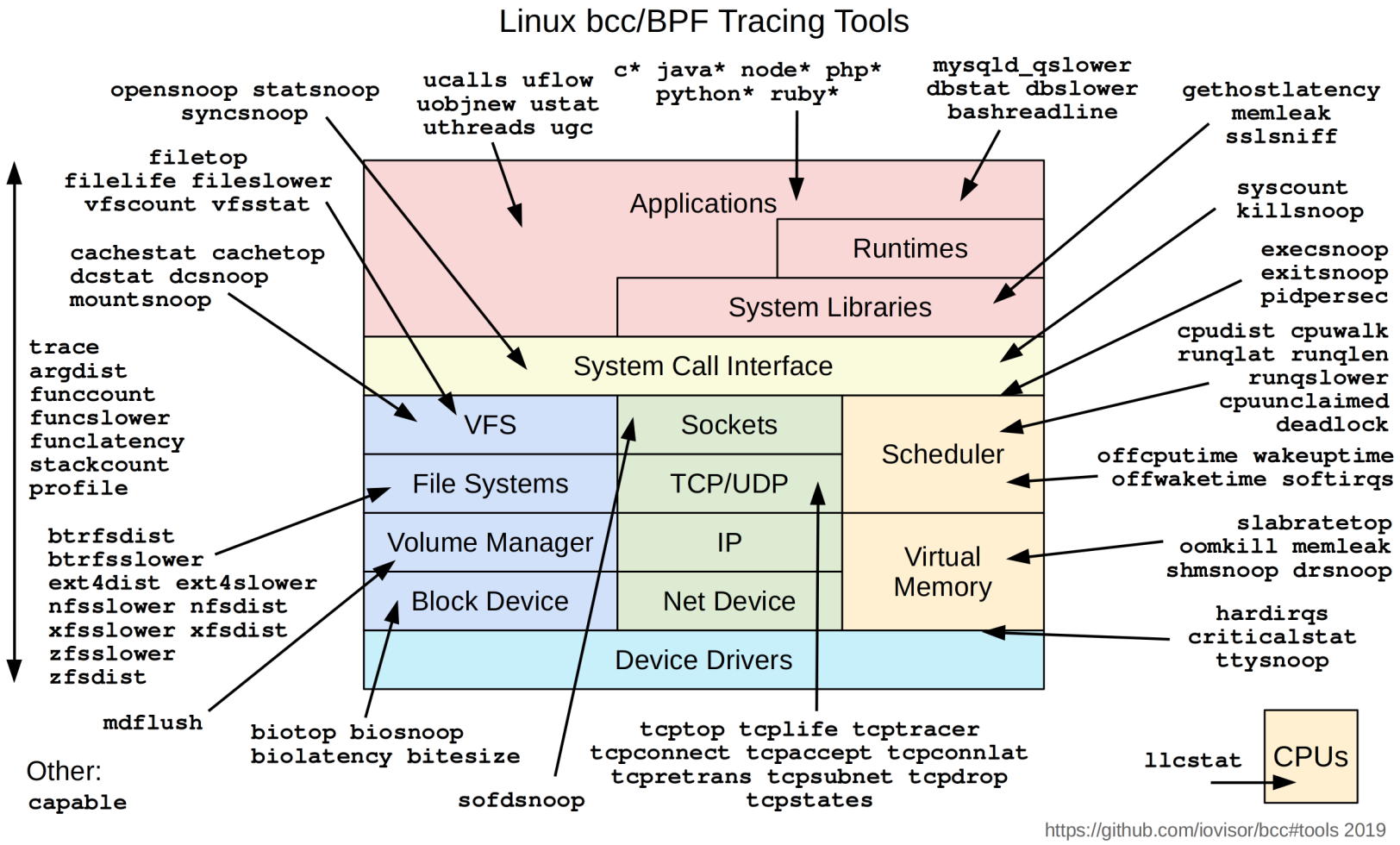


Image Credit: https://ebpf.io/

BCC tools example

- opensnoop: opensnoop prints one line of output for each open() syscall, including details.

```
root@sandbox:~$ opensnoop-bpfcc
PID    COMM               FD ERR PATH
752    systemd-oomd        7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/memory.pressure
1      systemd            41   0 /proc/330/cgroup
330    systemd-udevd      -1   2 /run/udev/queue
1668   upowerd            1668   upowerd            10   0 /sys/devices/LNXSYSTM:00/LNXSYBUS:00/PNP0C0A:00/power_supply/BAT0/present
```

- tcpconnect: It prints one line of output for every active TCP connection (eg, via connect()), with details including source and destination addresses.

```
root@sandbox:~# tcpconnect-bpfcc
Tracing connect ... Hit Ctrl-C to end
PID    COMM         IP SADDR            DADDR            DPORT
19464  ssh          4  192.168.1.109    192.168.1.109    22
19465  wget         6  2401:4900:1c74:7031:738d:9ef7:dc1b:22d7 2404:6800:4002:819::200e 80
19465  wget         6  2401:4900:1c74:7031:738d:9ef7:dc1b:22d7 2404:6800:4002:80e::2004 80
```

- Please refer to the BCC tool tutorial for more insights.

bpftrace

- bpftrace is a high-level front-end for BPF tracing, which uses libraries from bcc.
- bpftrace is best suited for spontaneous instrumentation through powerful custom one-liners and short scripts, whereas bcc is more suitable for developing complex tools and daemons.
- To install bpftrace tools on Ubuntu, execute `sudo apt-get install bpftrace`.
- Please refer to the bpftrace documentation for details on syntax and usage.

bpftrace tool

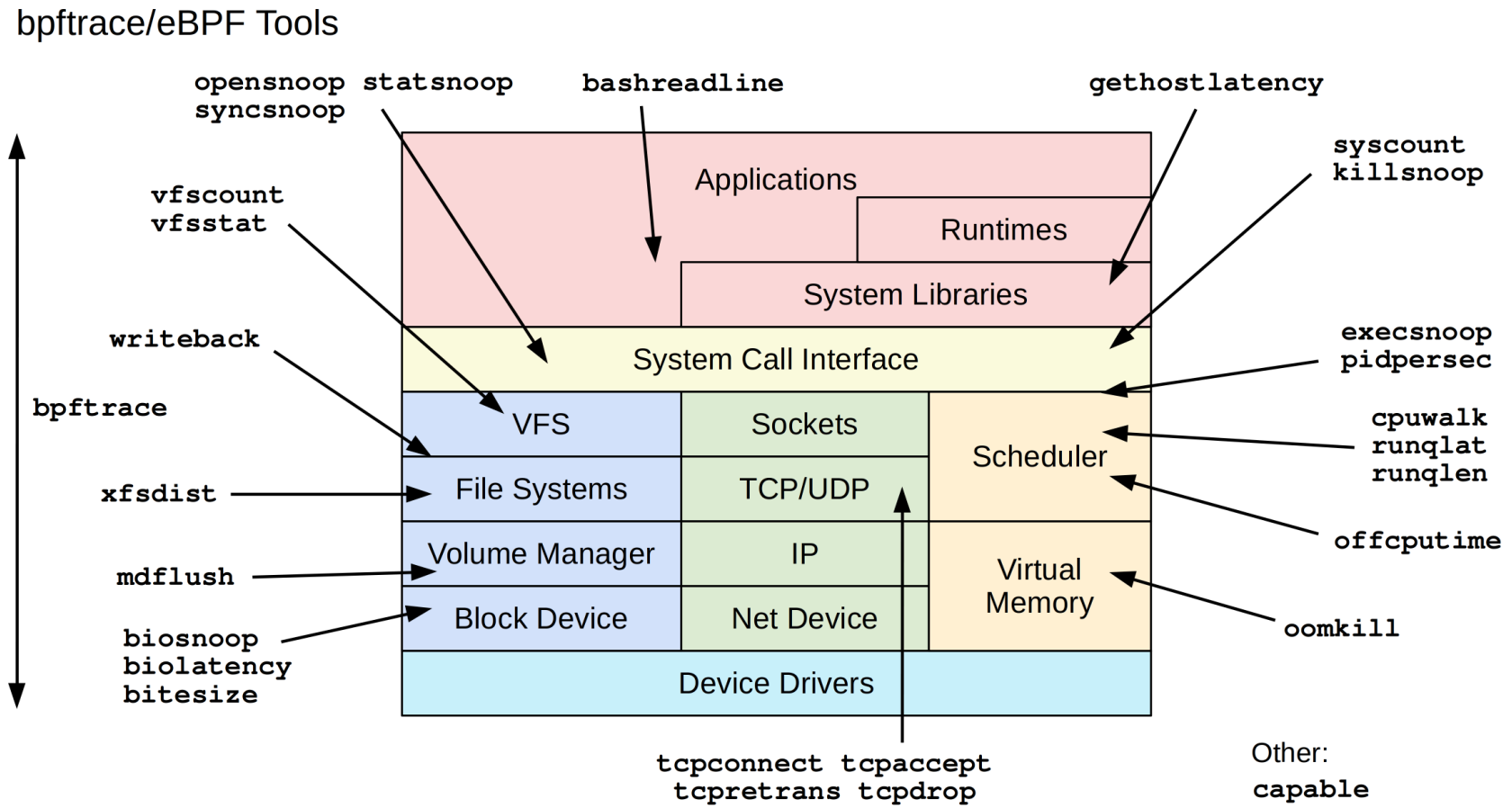


Diagram by Brendan Gregg, early 2019. https://github.com/iovisor/bpftrace

Image Credit: https://ebpf.io/

bpftrace examples

- Tracing open files:

```
# bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm, str(args->filename)); }'
```

- Counting syscalls by process name:

```
# bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'
```

- Distribution of read bytes:

```
# bpftrace -e 'tracepoint:syscalls:sys_exit_read /pid == 18644/ { @bytes = hist(args->ret); }'
```

- Counting process level events:

```
# bpftrace -e 'tracepoint:sched:sched* { @[probe] = count(); } interval:s:5 { exit(); }'
```

- Please refer to the bpftrace tutorial for more examples and insight.

eBPF development infrastrucutre

- eBPF development infrastructure supports variety of libraries and compilers to aid in developing custom eBPF program.
- The LLVM compiler infrastructure contains the eBPF backend required to translate programs written in a C-like syntax to eBPF instructions.
- Support for writing eBPF Programs:
    - C: libbpf is a C/C++ based library which is maintained as part of the upstream Linux kernel.
    - Go: libbpfgo is a Go wrapper around libbpf. It supports BPF CO-RE and its goal is to be a complete implementation of libbpf APIs.
    - Rust: libbpf-rs is a safe, idiomatic, and opinionated wrapper API around libbpf written in Rust.

eBPF development infrastructure

- libbpf-bootstrap simplifies BPF program setup for beginners, enabling them to start writing and tinkering with programs without worrying about the complexity of initial setup.
- libbpf-bootstrap relies on libbpf and utilizes a simple Makefile.
- The libbpf introduction and libbpf-bootstrap introduction provides a comprehensive explanation.
- The minimal and bootstrap demo applications provided in libbpf-bootstrap serves as a good starting points.

```
root@sandbox:/home/ubuntu/work/libbpf-bootstrap/examples/c# ./bootstrap

TIME     EVENT COMM             PID     PPID    FILENAME/EXIT CODE
09:22:16 EXEC  ls               10848   10841   /usr/bin/ls
09:22:16 EXIT  ls               10848   10841   [0] (4ms)
09:22:23 EXIT  bash             10849   10841   [1]
09:22:23 EXIT  bash             10850   10841   [0]
09:22:24 EXEC  cat              10851   10841   /usr/bin/cat
09:22:24 EXIT  cat              10851   10841   [0] (2ms)
```

LTTng Introduction

- The Linux Trace Toolkit next generation is a tracing framework for Linux system used for correlated tracing of the Linux kernel, user applications, and user libraries.
- LTTng facilitates tracing interactions between the Linux kernel and user applications (C/C++, Java, Python) by capturing event information from both kernel and user spaces.
- It traces the following instrumentation points:
    - LTTng kernel space tracepoints
    - LTTng user space tracepoints
    - krpobes and kretprobes
    - uprobes and uretprobes
    - system call

LTTng Introduction

- LTTng employs CTF (Common Trace Format) as its trace format, resulting in very compact event record data.
- The `lttng` command-line tool is the standard user interface to control LTTng recording sessions. It is linked with `liblttng-ctl` to communicate with one or more session daemons behind the scenes.
- LTTng is included in Linux repositories and can be installed using the standard method. For example in Ubuntu 22.04:

```
# apt-get install lttng-tools
# apt-get install lttng-modules-dkms
# apt-get install liblttng-ust-dev
```

Components of LTTng

LTTng Tools: These are libraries and a command-line interface to control recording sessions for example: session daemon, consumer daemon, relay daemon, tracing control library, tracing control.

LTTng-UST: These are libraries and packages to instrument and trace user applications for example: C, C++ and Java application using Apache log4j 1.2 and python applications using logging.

LTTng-modules: These are Linux kernel modules to instrument and trace the kernel for example: LTTng kernel tracer module, Recording ring buffer kernel modules, Probe kernel modules and LTTng logger kernel module.

LTTng Architecture

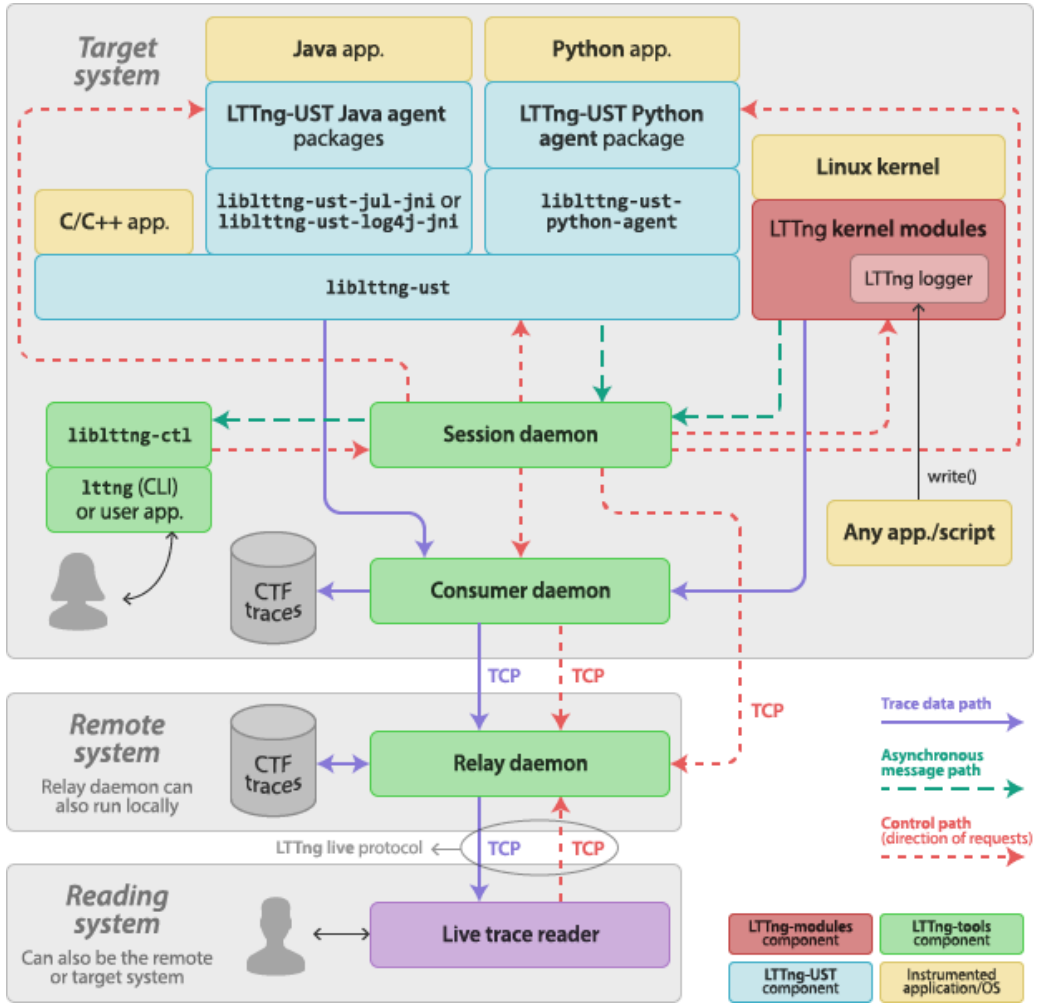


Image Credit: https://lttng.org/

LTTng core concepts

- Session: A session in LTTng is a trace collection period with specific configuration settings, including which events to trace, where to store the trace data, and how to handle overflow conditions.
- Channel: A channel organizes trace data streams within a session, enabling segregation based on criteria like event type or source.
- Ring Buffer: A circular memory buffer in LTTng, ensuring continuous tracing by overwriting old data with new data when full.
- Buffering Scheme: Determines how LTTng handles trace data when buffers reach capacity, such as overwriting, discarding, or using multiple buffers for seamless tracing.
- For detailed insight into the LTTng refer to the official LTTng documentation.

LTTng Tracepoints

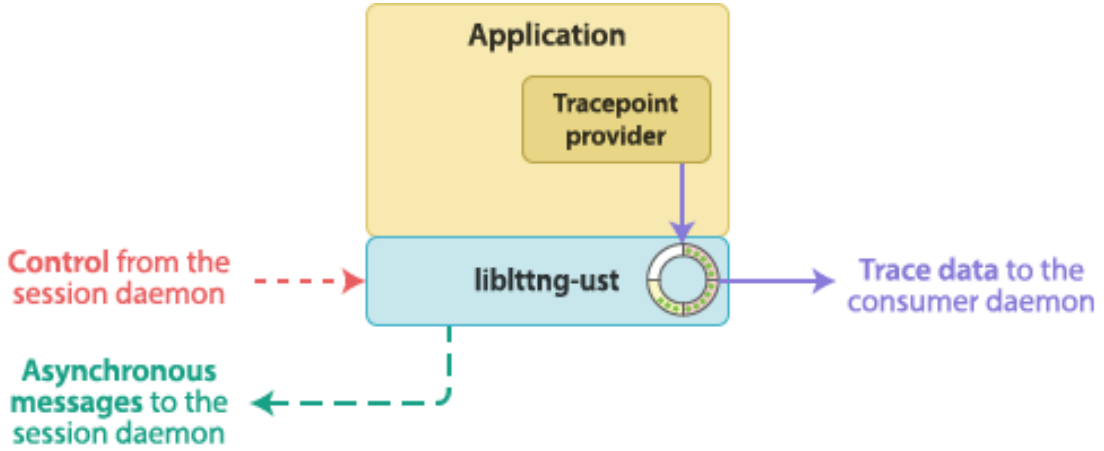
- LTTng kernel space tracepoint: A statically defined point in the source code of the kernel image or of a kernel module using the LTTng-modules macros.
- LTTng user space tracepoint: A statically defined point in the source code of a C/C++ application/library using the LTTng-UST macros.
- In userspace `LTTNG_UST_TRACEPOINT_EVENT()` macro defines a tracepoint definition for a given tracepoint. It includes the following characteristics:
  - Tracepoint provider name
  - Tracepoint identifier name
  - Input arguments: The lttng_ust_tracepoint() macro in the user application source code accepts these parameters.
  - Output event field: Describes the display format of tracepoint output.

Tracepoint provider package

- A tracepoint provider consists of compiled functions that supply tracepoints to an application.

- A tracepoint provider package, typically an object file (.o) or a shared library (.so), contains one or more tracepoint providers and comprises:
  - Tracepoint header (.h).
  - Tracepoint source (.c).

- A tracepoint provider package is dynamically linked with liblttng-ust, the LTTng user space tracer, at runtime.



Tracepoint definition

- Header file defining the tracepoint (sample_application_tp.h)

```
#undef LTTNG_UST_TRACEPOINT_PROVIDER
#define LTTNG_UST_TRACEPOINT_PROVIDER sample_application
#undef LTTNG_UST_TRACEPOINT_INCLUDE
#define LTTNG_UST_TRACEPOINT_INCLUDE "./sample_application_tp.h"
#if !defined(_SAMPLE_TP_H) || defined(LTTNG_UST_TRACEPOINT_HEADER_MULTI_READ)
#define _SAMPLE_TP_H

#include <lttng/tracepoint.h>
/* Tracepoint Definition */
LTTNG_UST_TRACEPOINT_EVENT(
    sample_application,           /* Provider Name */
    information_tracepoint,       /* Identifier Name */
    LTTNG_UST_TP_ARGS(            /* Input Argument */
        int, sample_integer_arg,
        char*, sample_string_arg
    ),
    LTTNG_UST_TP_FIELDS(          /* Output Argument */
        lttng_ust_field_string(sample_string_field, sample_string_arg)
        lttng_ust_field_integer(int, sample_integer_field, sample_integer_arg)
    )
)
#endif /* _SAMPLE_TP_H */
#include <lttng/tracepoint-event.h>
```

Tracepoint definition

- Tracepoint provider package source file (sample_application_tp.c)

```
#define LTTNG_UST_TRACEPOINT_CREATE_PROBES
#define LTTNG_UST_TRACEPOINT_DEFINE

#include "sample_application_tp.h"
```

- Please refer to this tracepoint definition with the `lttng_ust_tracepoint()` macro in the application source code.

```
#include <stdio.h>
#include "sample_application_tp.h"

int main(int argc, char* argv[]) {
    lttng_ust_tracepoint(sample_application, information_tracepoint, 123, "Test tracepoint");
    return 0;
}
```

- Compiling application with tracepoint to generate "Traceable application".

```
$ sudo gcc sample_application.c sample_application-tp.c -I. -llttng-ust -o sample_application
```

Tracepoint build mechanism

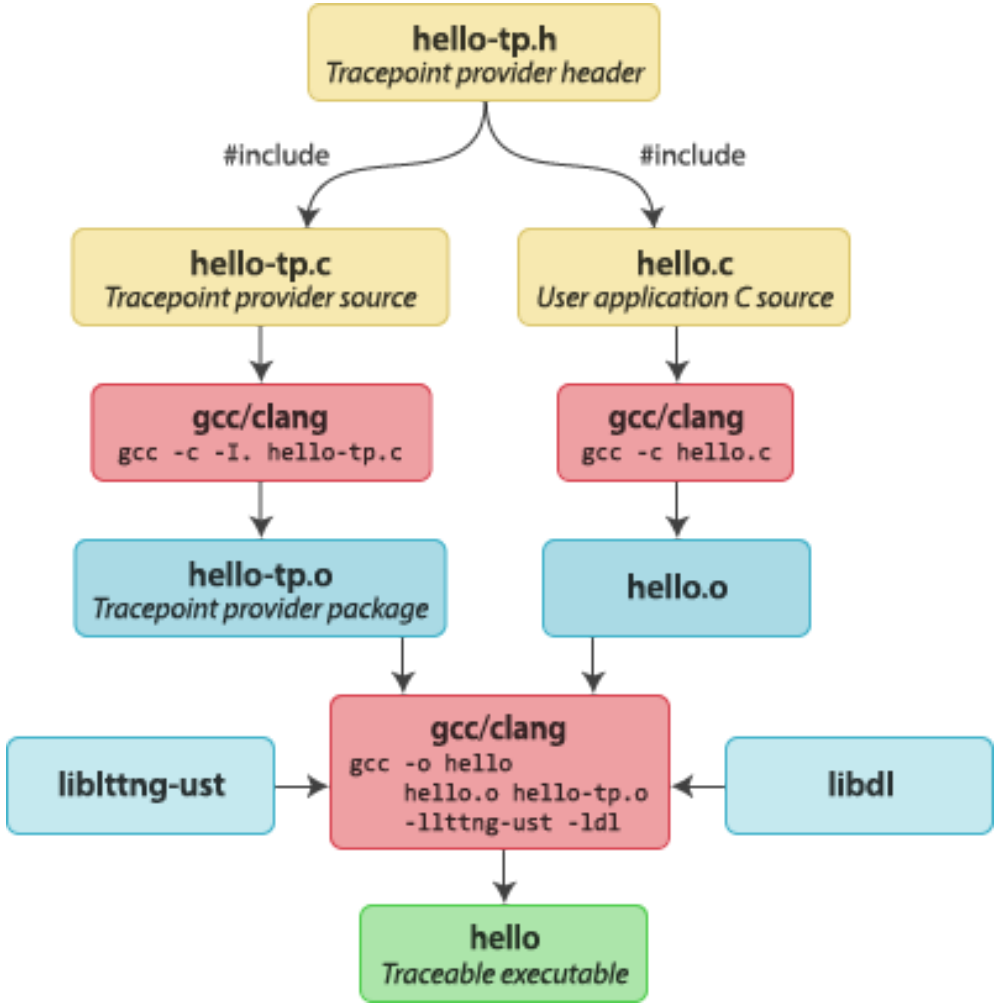


Image Credit: https://lttng.org/

LTTng user-space tracing

- Tracing user defined tracepoint in sample_application

```
# Create lttng session
ubuntu@sandbox:~/work/Examples/lttng$ lttng create my-tracing-session --output=./
Spawning a session daemon
Session my-tracing-session created.
Traces will be output to /home/ubuntu/work/Examples/lttng/

# Enable tracepoint event
ubuntu@sandbox:~/work/Examples/lttng$ lttng enable-event --userspace sample_application:information_tracepoint
ust event sample_application:information_tracepoint created in channel channel0

# Start Lttng tracing
ubuntu@sandbox:~/work/Examples/lttng$ lttng start
Tracing started for session my-tracing-session

# Run application
ubuntu@sandbox:~/work/Examples/lttng$ ./sample_application

# Destroy session
ubuntu@sandbox:~/work/Examples/lttng$ lttng destroy
Destroying session my-tracing-session..
Session my-tracing-session destroyed

# Display trace information using babeltrace
ubuntu@sandbox:~/work/Examples/lttng$ babeltrace ./ust/
[17:21:04.622451452] (+?.??????????) sandbox sample_application:information_tracepoint: { cpu_id = 0 }, { sample_string_field = "Test tracepoint", sample_integer_field = 123 }
```

LTTng kernel-space tracing

- Trace Kernel events using LTTng

```
# List traceable events in the kernel
$ lttng list --kernel
# Trace "sched_switch" event
$ lttng create test_session --output=./test_trace
$ lttng enable-event -k sched_switch
$ lttng start
$ sleep 1
$ lttng destroy
# View "sched_switch" event in trace file
$ babeltrace ./test_trace | grep sleep
```

- Trace system call using LTTng

```
$ lttng create kernel-session --output=./kernel_trace
# Trace open, close and write syscall
$ lttng enable-event --kernel --syscall open,close,write
$ lttng start
# /* Perform file operations */
$ lttng destroy
$ babeltrace ./kernel_trace
```

LTTng preloaded helpers

- The LTTng-UST package offers several helper libraries. These libraries are provided as preloadable shared objects, which automatically instrument system functions and calls.
  - liblttng-ust-libc-wrapper.so
  - liblttng-ust-pthread-wrapper.so
  - liblttng-ust-cyg-profile
  - liblttng-ust-cyg-profile-fast
  - liblttng-ust-dl
- To utilize a user space tracing helper library with any user application, preload the helper shared object during application startup.

```
$ LD_PRELOAD=liblttng-ust-libc-wrapper.so:liblttng-ust-dl.so my-app
```

Remote tracing with LTTng

- LTTng has the capability to transmit the recorded trace data from a recording session to a remote system via the network instead of saving it locally on the file system.
- Execute `lttng-relayd` deamon on the remote system.
- Create LTTng session on the target system configured to send data trace over the network

```
# Replace "remote-system" with IP address of remote system
$ lttng create my-session --set-url=net://remote-system
```

- Execute the `lttng` command line tool to initiate tracing on the target system. The trace data will be transmitted to the remote system over the network instead of being stored locally on the target system.
- Remote tracing is useful for contrained devices with limited storage capacity.

References

- Brendan Greg's blogs on various topics related to tracing in Linux.
  - https://www.brendangregg.com/blog/index.html
- Julia Evans blog on Linux tracing system.
  - https://jvns.ca/blog/2017/07/05/linux-tracing-systems/
- Perf tool building instructions - Manas Marawaha.
  - https://medium.com/@manas.marwah/building-perf-tool-fc838f084f71
- Instructions for downloading vmlinux file in Ubuntu 22.04.
  - https://github.com/SpecialistLinuxTraining/linux-debug-training/blob/main/References/vmlinux-ubuntu-22.04.md
- Understanding the Linux Kernel via Ftrace - Steven Rostedt
  - https://www.youtube.com/watch?v=2ff-7UTg5rE
- Getting Started with eBPF - Liz Rice, Isovalent
  - https://www.youtube.com/watch?v=TJgxjVTZtfw
- LTTng Documentation
  - https://lttng.org/docs/