Linux Debug Training

Part-2

Presenters: John O'Sullivan and Manas Marawaha

Document source: https://github.com/SpecialistLinuxTraining/linux-debug-training/tree/main/Slides

Part-1 Demonstrations: https://www.udemy.com/course/linux-debug-training-part-1/?referralCode=6534D858AEF9555AB23F

Part-2 Demonstrations: https://www.udemy.com/course/linux-debug-training-part-2/?referralCode=1FAF95060F47194A895F

Presenters Introduction

John O'Sullivan

John O'Sullivan is a software architect with over 30 years of experience, specializing in the development and optimization of Linux-based embedded systems. He has extensive hands-on expertise in hardware-software integration and product development across a wide range of industries.

Email LinkedIn

Manas Marawaha

Manas Marawaha is a Principal Engineer with over 13 years of experience in embedded Linux software development. With a strong understanding of Linux internals, kernel programming, and device drivers, he brings extensive hands-on expertise in product development across audio, video, and networking domains.

Email LinkedIn

License Information

You are free to:

- **Share** - copy and redistribute the material in any medium or format for any purpose, even commercially.
- **Adapt** - remix, transform, and build upon the material for any purpose, even commercially.
- The licensor cannot revoke these freedoms as long as you follow the license terms.

License Information (Cont.)

Under the following terms:

- **Attribution** - You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **ShareAlike** - If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- **No additional restrictions** - You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Course information

- Example code demonstrated in the course is available on the GitHub.
- For **document source**, please refer to this link.
- For video demonstration, please refer to the following links.
    - **Linux Debug Training (Part-1)**
    - **Linux Debug Training (Part-2)**
- Demonstrations in the course use **Ubuntu (OS 22.04)**, **Arch Linux (2025.08.01)** and a Buildroot based installation on the **Raspberry-Pi**.
- Please review the instructions for building the course examples before proceeding.
- We value your **feedback and suggestions**! Please share them via email at specialistlinuxtraining@gmail.com.
- For corrections on the training material, feel free to raise issues and pull request in the GitHub repository.

Course Outline

Linux Debug Training (Part-1)

Module 1: Linux Operating System Architecture

Module 2: Basic Linux Analysis and Observability Tools

Module 3: Application Debugging

Module 4: Memory Issues in Linux Applications

Linux Debug Training (Part-2)

Module 5: Tracing in Linux

Module 6: Profiling in Linux

Module 7: Linux Kernel Debugging

Part-1 Synopsis

- Module 1 began with the Linux OS system itself, discussing the fundamental architecture and structure of the Linux operating system.
- In the second module of this course, we explored the Basic Linux Analysis and Observability Tools. We demonstrated how these foundational tools served as the first level of debugging and how they provided insights into the system's state and behavior.
- The third module of this course looked at how we could dissect application binaries with tools like binutils and how we could employ powerful debugging applications like GDB for more in-depth analysis.
- The fourth module was dedicated to addressing common memory issues in user-space applications, along with exploring related tools such as valgrind and sanitizers which are designed to detect and resolve these issues before the code is deployed.

Part-2 Synopsis

- Module five takes a look at tracing in Linux. We cover userspace tools like strace, ltrace, uprobe and perf. We also look at kernel space tools like Kprobe, Perf, ftrace, eBPF, and LTTng.
- In the sixth module of this course, we will explore profiling in Linux looking at tools like massif, heaptrack and memusage to profile memory. We will investigate the use of callgrind, cachegrind and perf-stat for CPU and hardware profiling. We will also look at stacktrace profiling using eBPF profile, gprof and sysprof. And, we will look at how data visualization tools can complement our analysis.
- In the seventh and final module we take a comprehensive look at Kernel debugging, investigating Kernel OOPS, reviewing logging and SysRq, and using tools like KGDB. We will show Kernel recovery using Kexec and Kdump. And we will also examine many of the new tools that have emerged in recent years like: UBSAN, KCSAN and KASAN.

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.

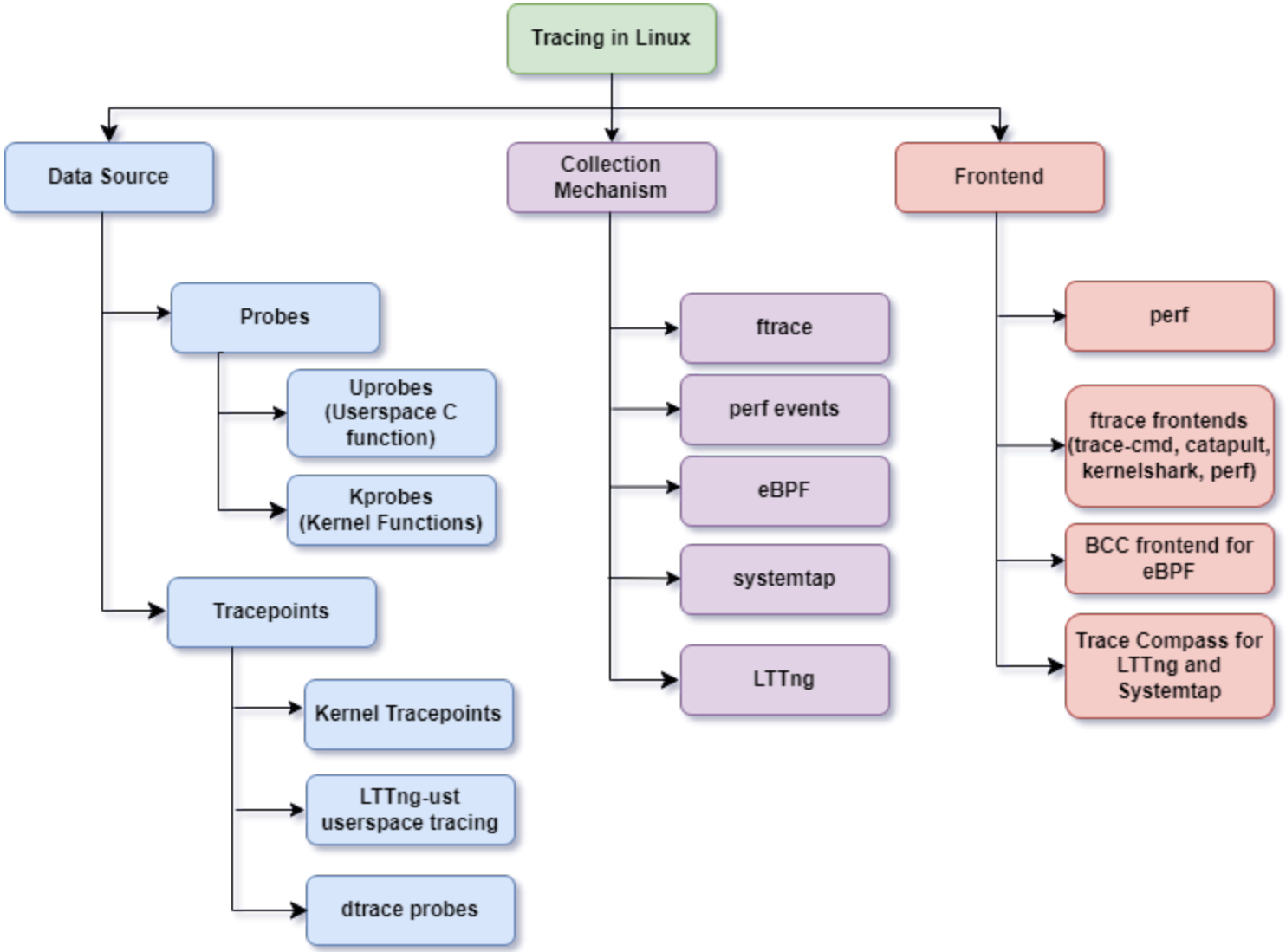
User space Tracing tools

- Strace
- Ltrace
- Uprobe
- Perf

Kernel Space Tracing tools

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

Tracing landscape in Linux



Linux tracing tools classification based on Julia Evans blog post.

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.

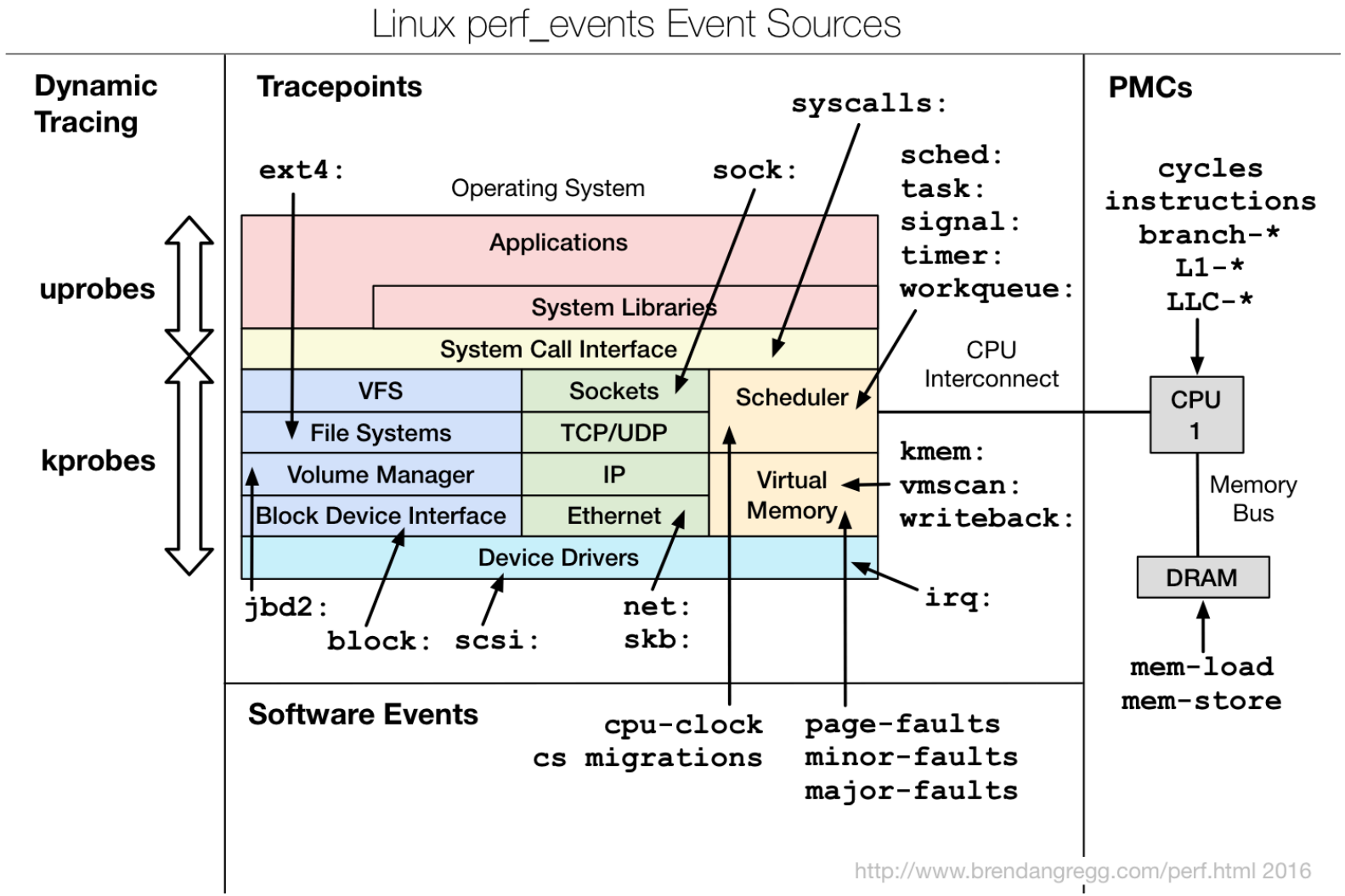
Perf Events Tracing

- 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

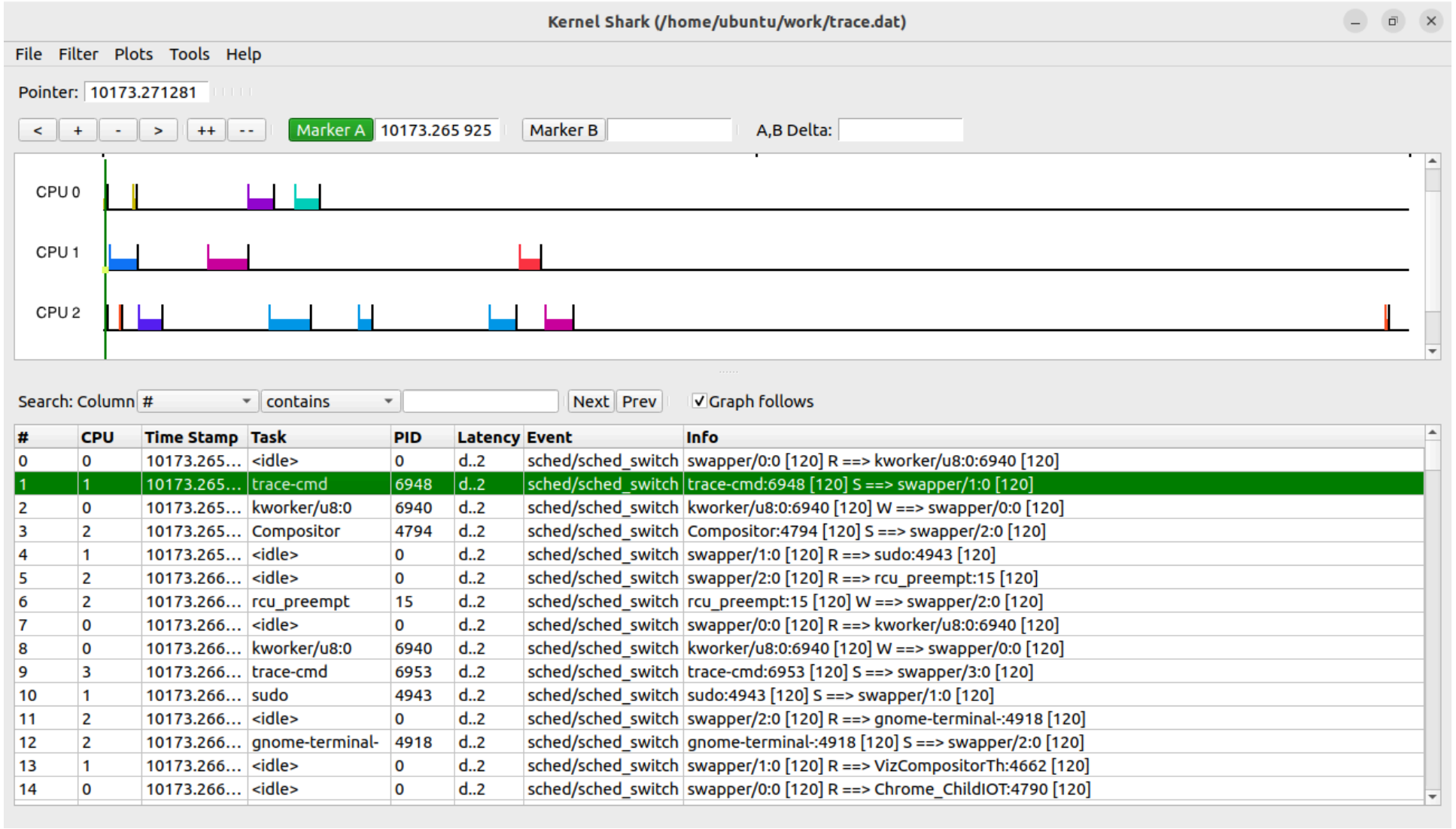
- To remove all tracers and reset ftrace buffers.
  ```
  trace-cmd reset
  ```
- To display the trace contents we can extract them to a file and then display them in human readable form.
  ```
  trace-cmd extract -o trace.dat
  trace-cmd report -i trace.dat
  ```
- Or to directly display the contents 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

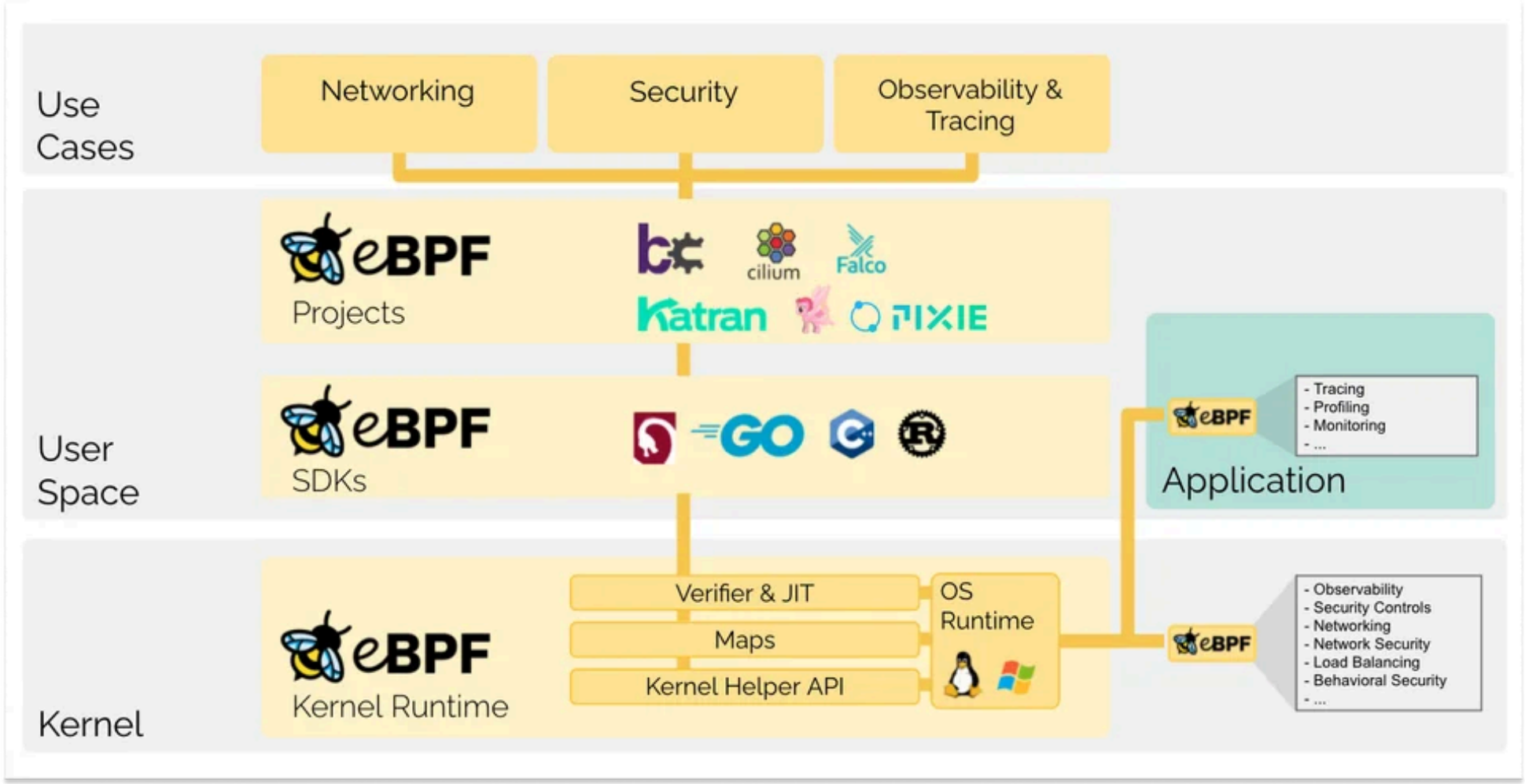


Image Credit: https://ebpf.io/

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

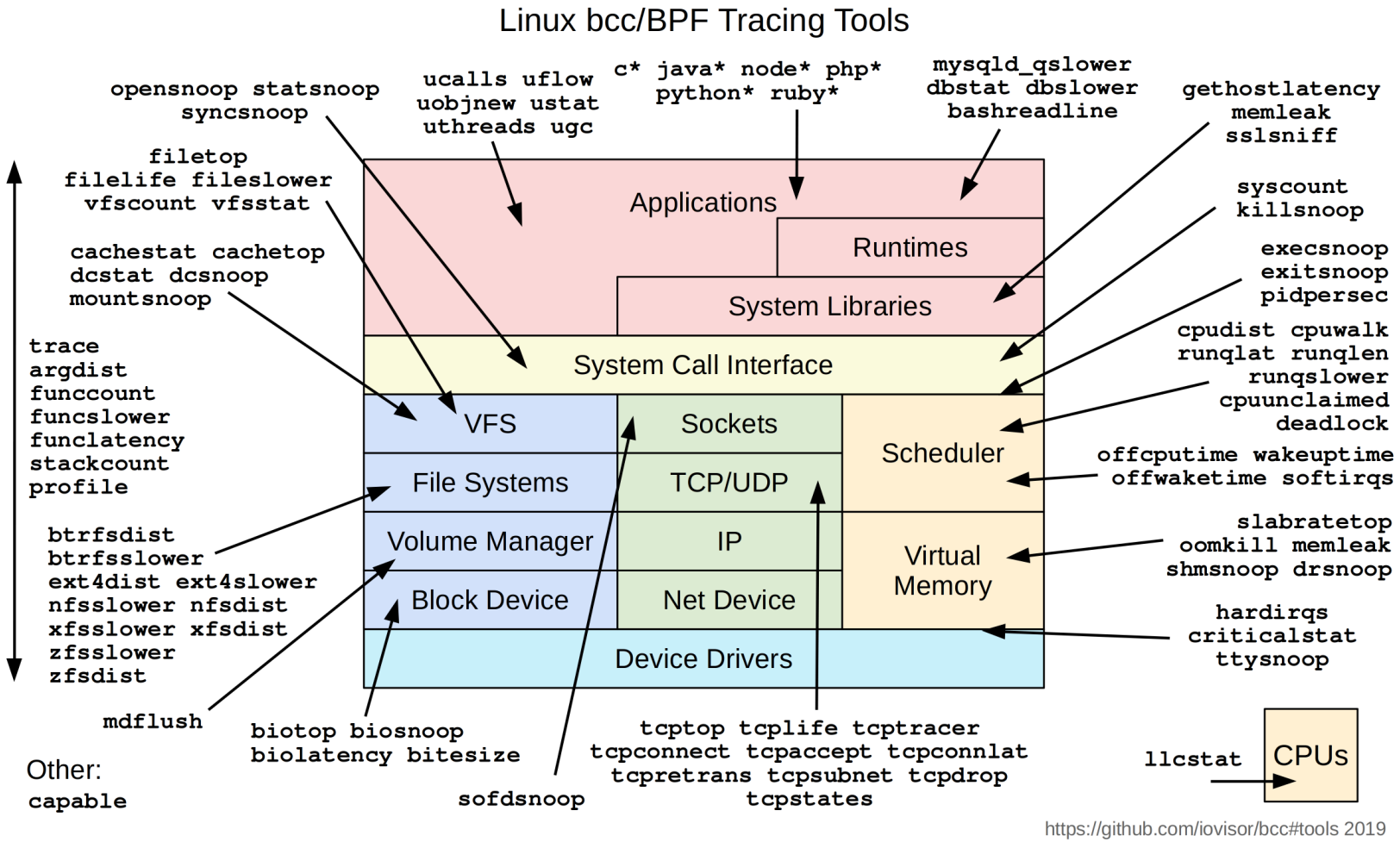


Image Credit: https://ebpf.io/

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

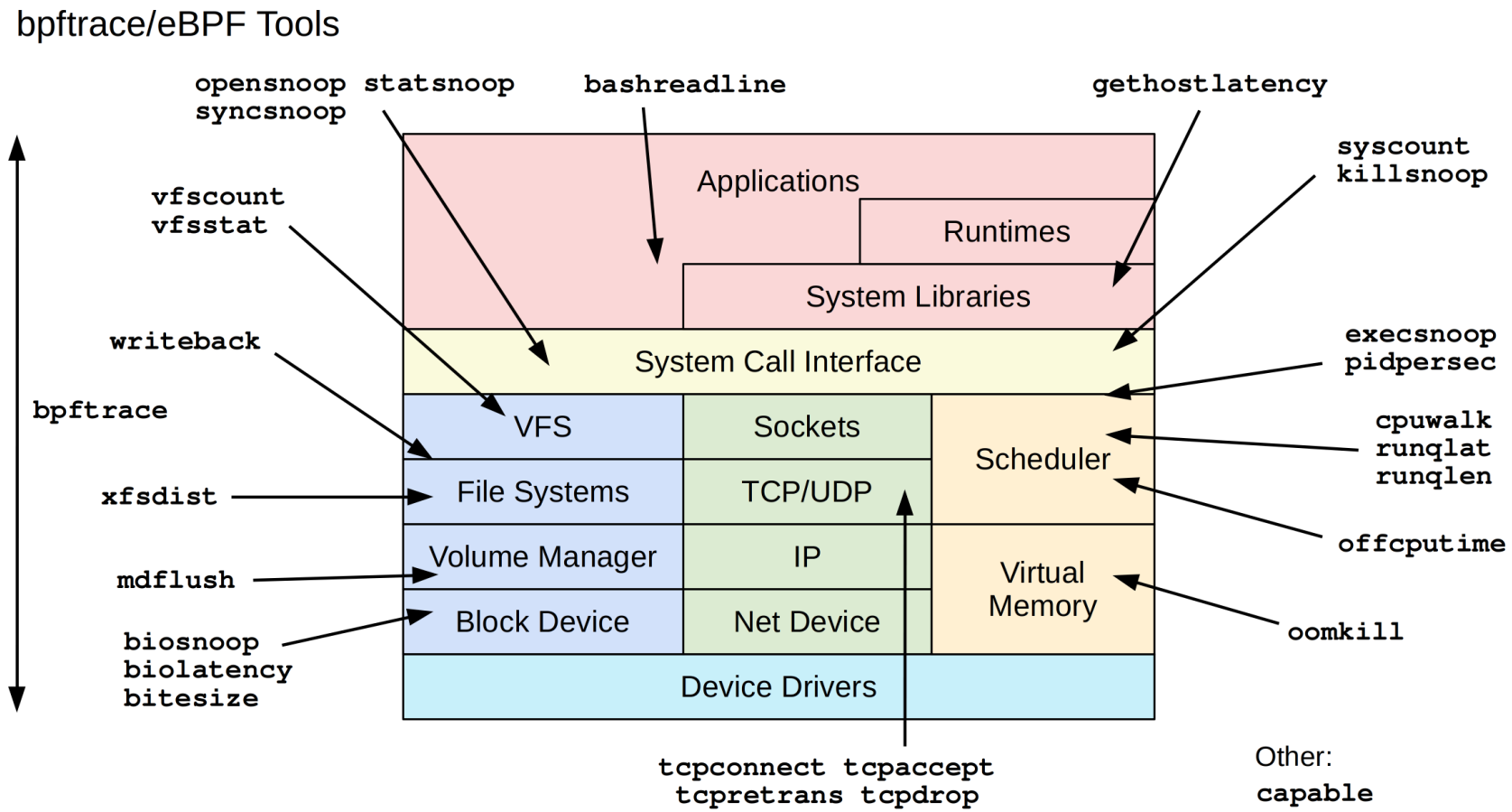


Diagram by Brendan Gregg, early 2019. https://github.com/iovisor/bpftrace

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

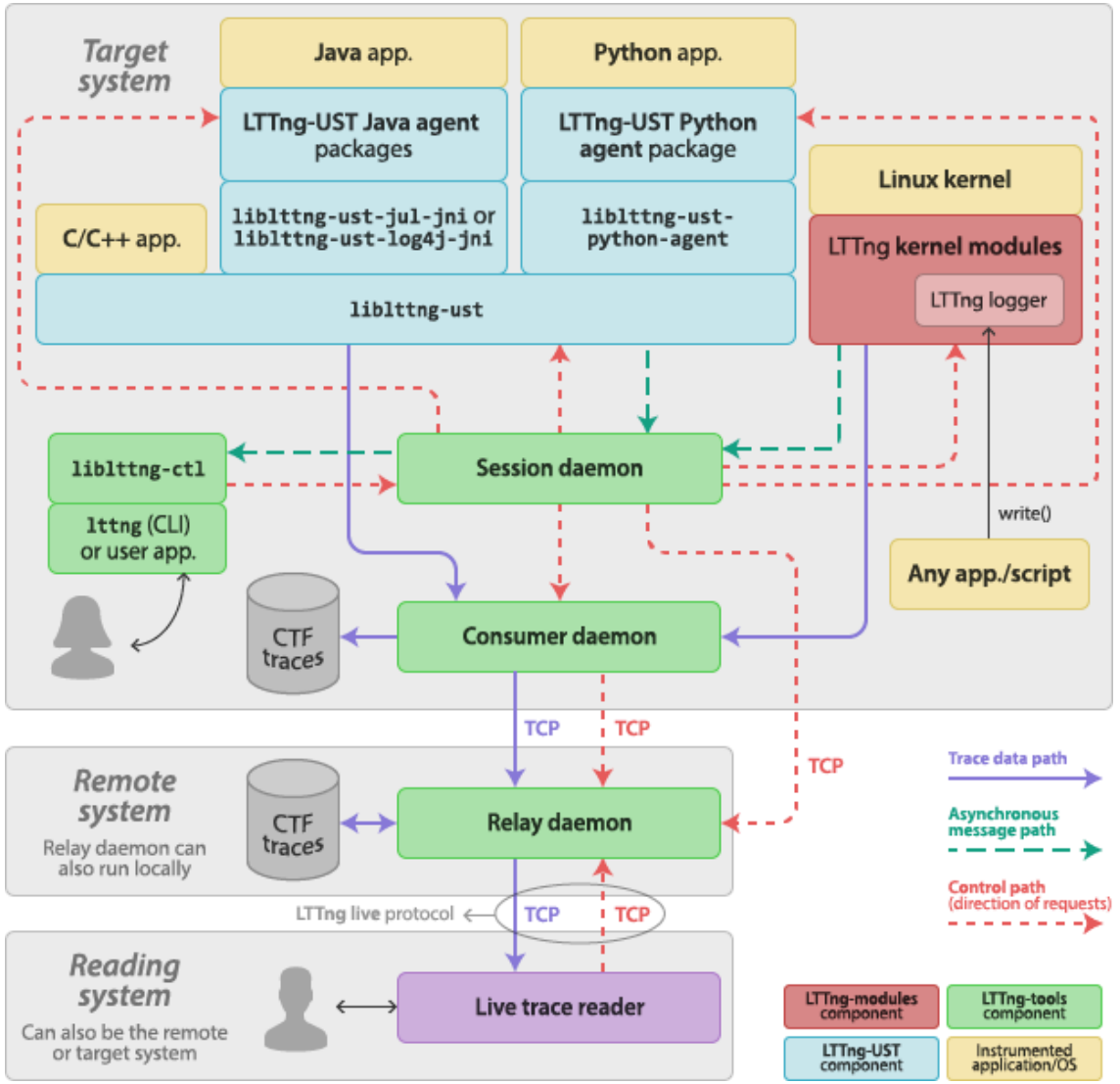


Image Credit: https://lttng.org/

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

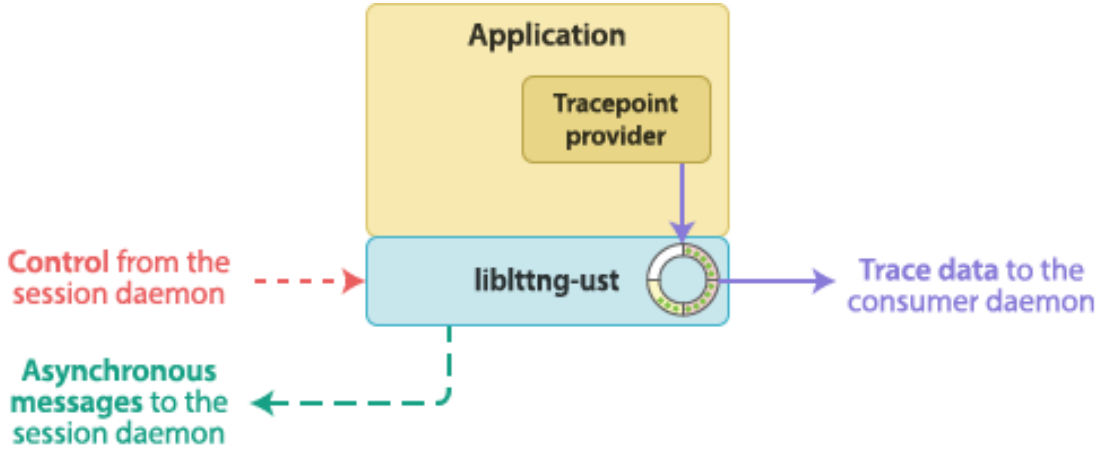
- LTTng kernel space tracepoint: A statically defined point in the source code of the kernel image or of a kernel module using the LTTng-modules macros.
- LTTng user space tracepoint: A statically defined point in the source code of a C/C++ application/library using the LTTng-UST macros.
- In userspace `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

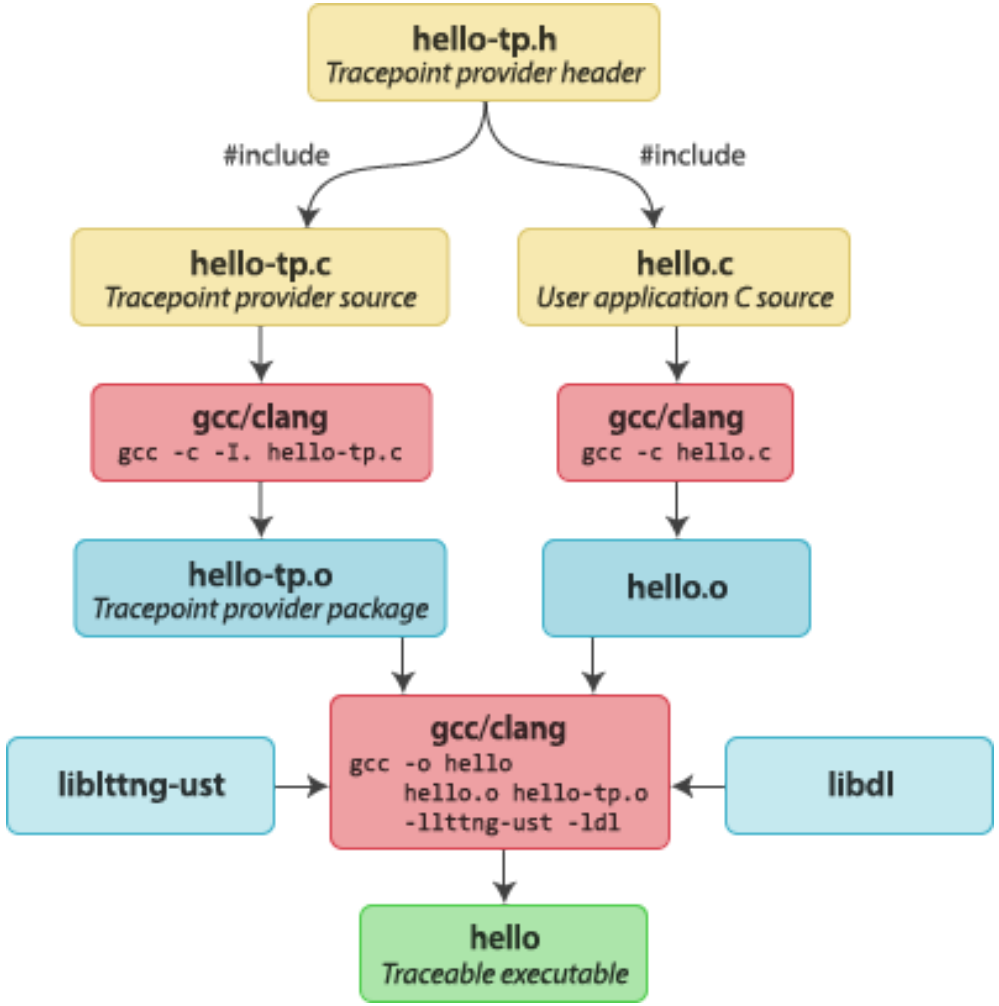


Image Credit: https://lttng.org/

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

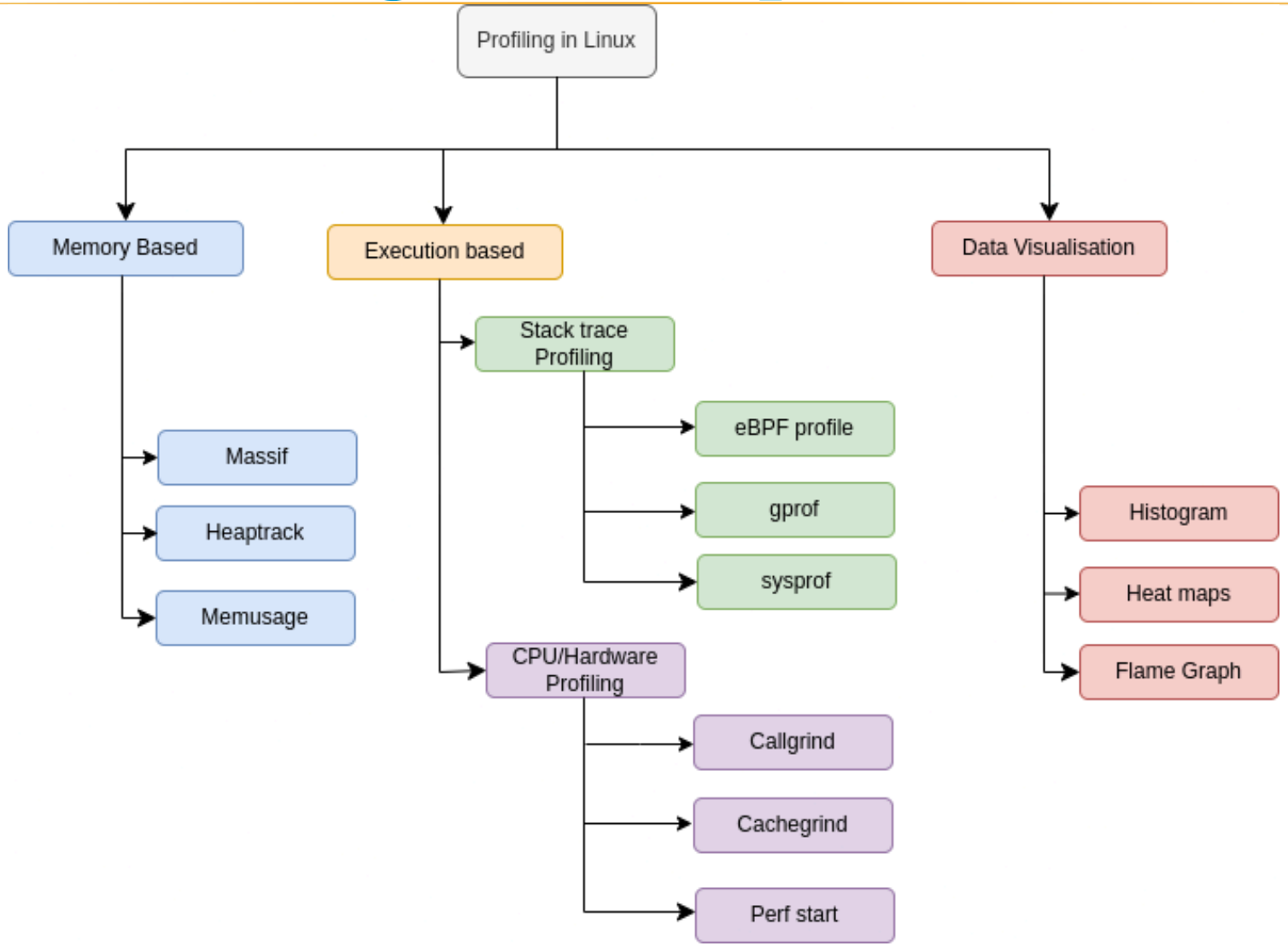
- Brendan Greg's blogs on various topics related to tracing in Linux.
  - https://www.brendangregg.com/blog/index.html
- Julia Evans blog on Linux tracing system.
  - https://jvns.ca/blog/2017/07/05/linux-tracing-systems/
- Perf tool building instructions - Manas Marawaha.
  - https://medium.com/@manas.marwah/building-perf-tool-fc838f084f71
- 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/

Module 6
Profiling in Linux

Introduction

- Profiling is the process of collecting and analyzing performance data from user applications or system-wide to understand their behavior and identify performance bottlenecks.
- A profiling tool uses sampling techniques to collect data at defined intervals.
- Profiling provides an aggregated overview of system performance to identify bottlenecks, while tracing captures detailed, sequential event data to analyze execution flow and debug specific issues.
- Profiling can be done on different aspects of the system such as:
    - Function exection time
    - Function call count
    - Memory usage
    - Hardware and Software events

Profiling landscape in Linux



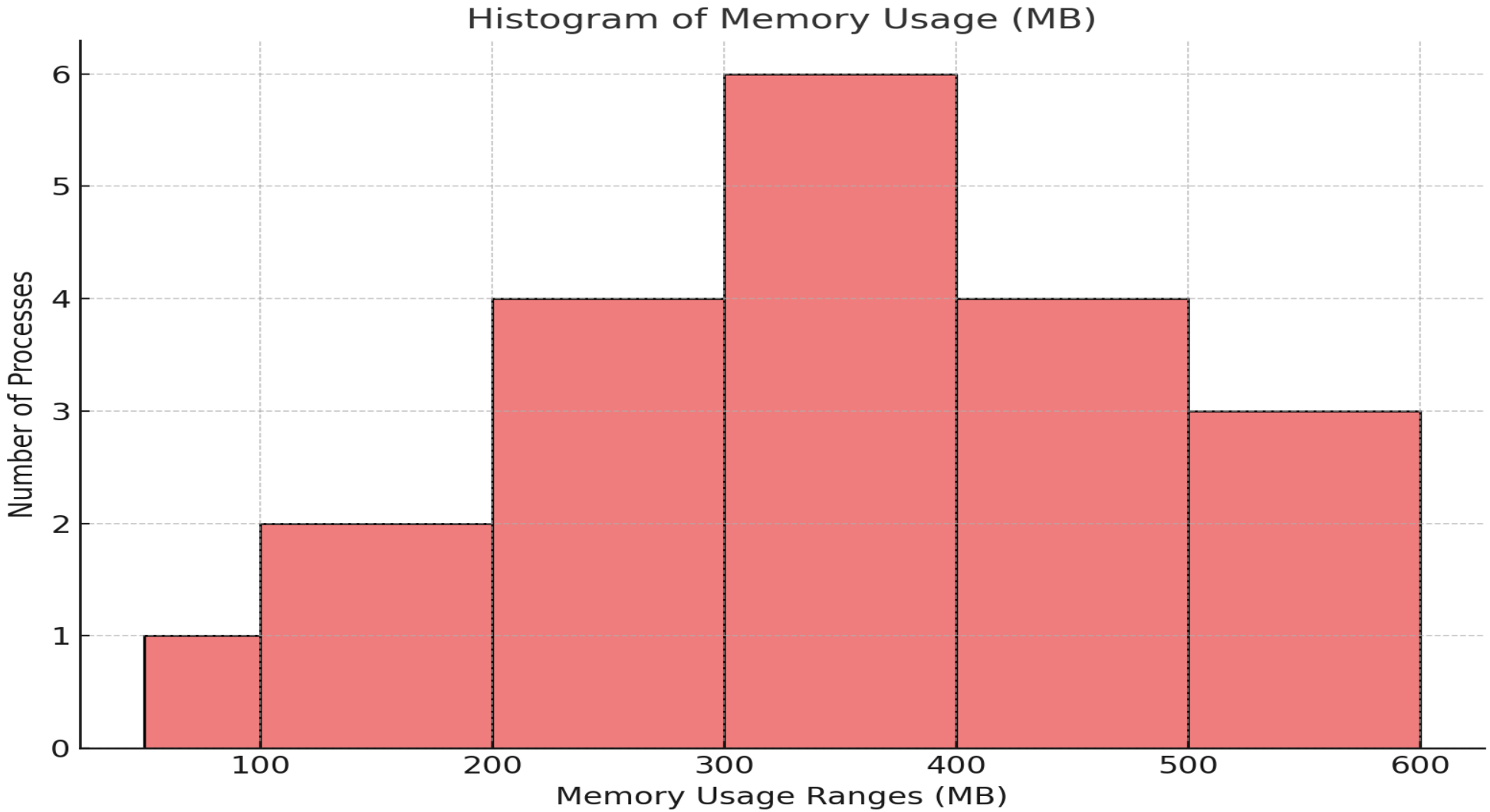
Data Visualization

Histogram

- A histogram is a type of graph that shows how data is distributed by grouping it into intervals or "bins."
- It helps understand the following:
  - Frequency of data points within specific ranges.
  - Understand patterns or trends in data.
  - Identify the shape of the data distribution (e.g., normal, skewed).
  - Spot outliers or unusual data points.
- It can be used to visualize the performance data collected by profiling tools, providing insights into resource utilization and identifying areas for optimization.
- **Example**: Analyzing the memory usage of processes running on a Linux server over a certain period.

```
Memory usage (in MB) for 20 processes:
[50, 120, 180, 200, 210, 230, 250, 300, 320, 330, 350, 370, 390, 410, 430, 450, 470, 500, 520, 550]
```

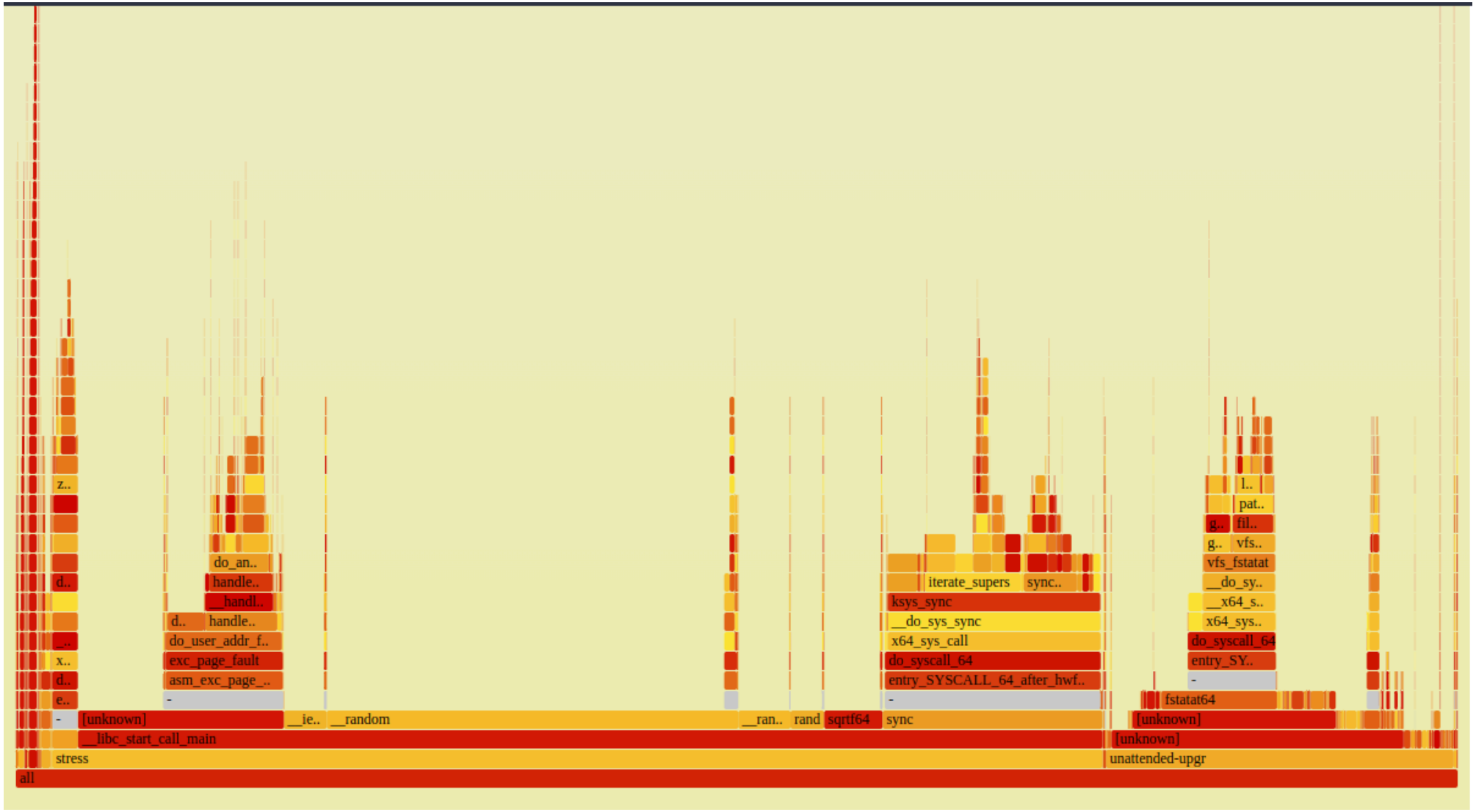
Histogram Visualization



FlameGraph

- Flame graphs are a type of visualization that represents sampled stack traces of a program's execution.
- Flame graphs are commonly used in performance analysis to identify performance bottlenecks, CPU hotspots, and inefficient code paths.
- The colors often represent different functions or function categories, aiding in quick identification.
- The vertical axis (Y-axis) represents the stack depth, showing the sequence of function calls counting from zero at the bottom.
- The horizontal axis (X-axis) represents stack profile population, sorted alphabetically. Each stack frame is represented by a horizontal bar, with wider bars indicating functions that consume more CPU time.
- Please refer to the instructions for flame graph generation.

FlameGraph Visualization



Memory profiling

- Memory profiling tools highlight inefficient memory usage patterns such as excessive allocations, unnecessary copying of data, and redundant memory accesses.
- Helps in finding and fixing memory leaks by tracking all memory allocations and deallocations, ensuring that memory is properly released.
- Enhances cache utilization by promoting better memory locality, allowing data to be accessed more quickly from the CPU cache.
- Reduces the need for frequent page swaps by maintaining a smaller working set in memory, thus minimizing paging and disk I/O.

Massif

- Massif is a memory profiler tool in Valgrind, used to analyze heap allocations, stack usage, and overall memory consumption of a program.
- It provides detailed analysis of heap memory usage, aiding in the identification of memory leaks and inefficient memory usage patterns.
- Massif tool can be invoked using following command line.

```
valgrind --tool=massif --time-unit=B <Application Program>
```

- Upon completion, Valgrind does not print summary statistics; instead, all of Massif's profiling data is written to a file, typically named massif.out.{pid} (filename can be changed using --massif-out-file option).
- You can then use the `ms_print` tool to visualize a heap allocation graph.

```
ms_print massif.out.5464
```

Massif report (1/2)

```
$ ms_print massif.out.5464
--------------------------------------------------------------------------------
Command:            ./massif
Massif arguments:   --detailed-freq=2 --time-unit=B
ms_print arguments: massif.out.5464
--------------------------------------------------------------------------------


    KB
10.58^                                                                    ####
     |                                                                    #
     |                                                                    #   :
     |                                                                    #   :
     |                                                                    #   :
     |                                                                    #   :
     |                                                                    #   :
     |                                                                    #   :
     |                                          ::::::::::::::::::::::::  #   :
     |                                          :                        #   :
     |                                          :                        #   :
     |                                          :                        #   :
     |                                          :                        #   :
     |                                          :                        #   :
     |                                          :                        #   :
     |                    @@@@@@@@@@@@@@@@@@@@@@:                        #   :
     |                    @                     :                        #   :
     |                    @                     :                        #   :
     |       @@           @                     :                        #   :
     |       @ ::::::::::@                     :                        #   :
   0 +-------------------------------------------------------------------------->KB
     0                                                                    12.16

Number of snapshots: 10
 Detailed snapshots: [1, 3, 5, 7, 8 (peak)]
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0              0            0            0
  1            408              408            400            8            0
98.04% (400B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->98.04% (400B) 0x1091AD: main (massif.c:15)
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  2          1,216            1,216          1,200           16            0
  3          1,216            1,216          1,200           16            0
98.68% (1,200B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->65.79% (800B) 0x1091BB: main (massif.c:16)
|
->32.89% (400B) 0x1091AD: main (massif.c:15)
```

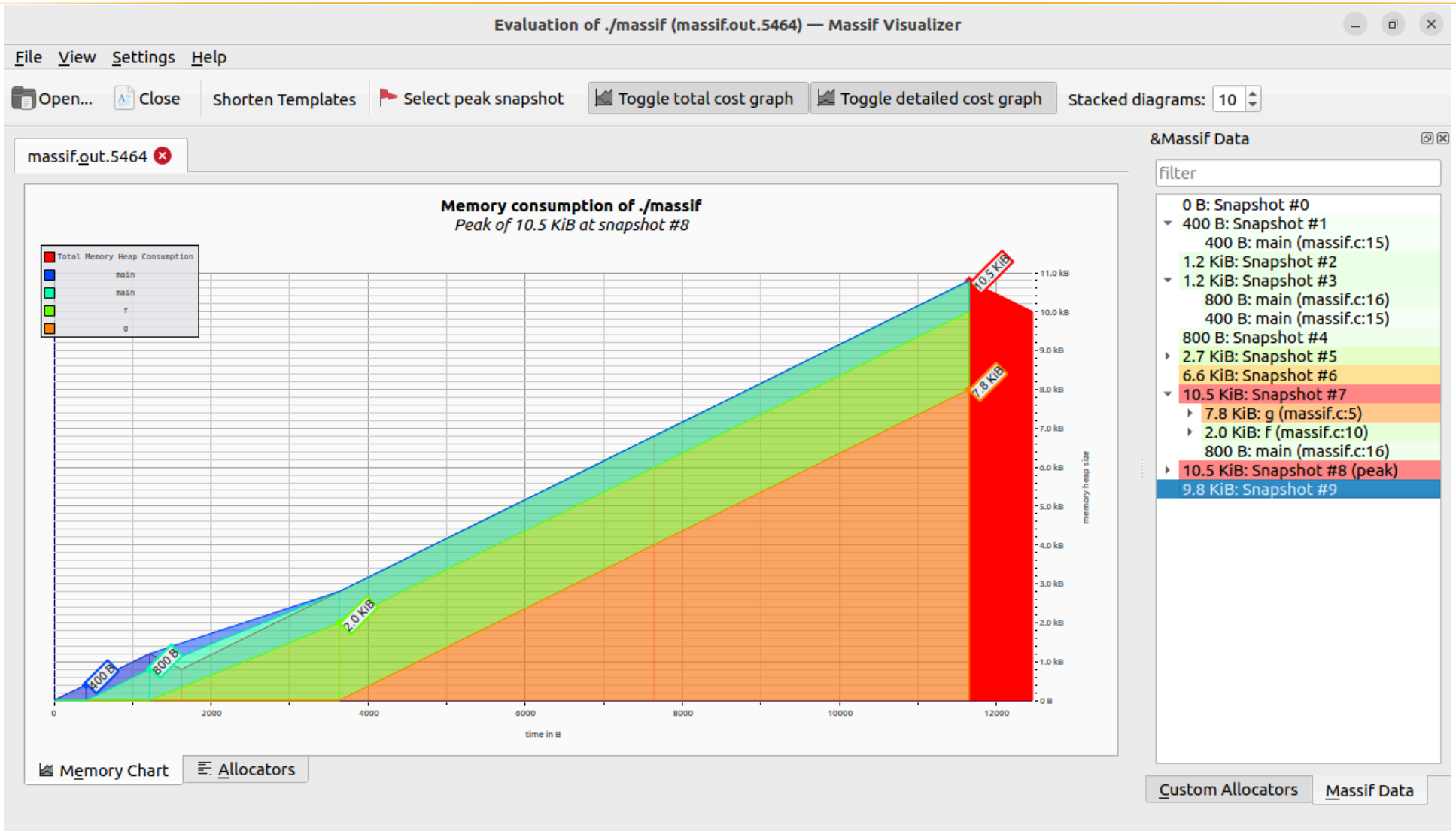
Massif report (2/2)

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  4          1,624              808              800            8            0
  5          3,632            2,816            2,800           16            0
99.43% (2,800B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->71.02% (2,000B) 0x10918F: f (massif.c:10)
| ->71.02% (2,000B) 0x109204: main (massif.c:26)
|
->28.41% (800B) 0x1091BB: main (massif.c:16)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)

--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  6          7,640            6,824            6,800           24            0
  7         11,648           10,832           10,800           32            0
99.70% (10,800B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->73.86% (8,000B) 0x10917A: g (massif.c:5)
| ->36.93% (4,000B) 0x109194: f (massif.c:11)
| | ->36.93% (4,000B) 0x109204: main (massif.c:26)
| |
| ->36.93% (4,000B) 0x109209: main (massif.c:28)
|
->18.46% (2,000B) 0x10918F: f (massif.c:10)
| ->18.46% (2,000B) 0x109204: main (massif.c:26)
|
->07.39% (800B) 0x1091BB: main (massif.c:16)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)

--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  8         11,648           10,832           10,800           32            0
99.70% (10,800B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->73.86% (8,000B) 0x10917A: g (massif.c:5)
| ->36.93% (4,000B) 0x109194: f (massif.c:11)
| | ->36.93% (4,000B) 0x109204: main (massif.c:26)
| |
| ->36.93% (4,000B) 0x109209: main (massif.c:28)
|
->18.46% (2,000B) 0x10918F: f (massif.c:10)
| ->18.46% (2,000B) 0x109204: main (massif.c:26)
|
->07.39% (800B) 0x1091BB: main (massif.c:16)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)

--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  9         12,456           10,024           10,000           24            0
```

Massif result analysis

- We are using massif sample program for this demonstration.
- Each vertical bar represents a snapshot, indicating the memory usage at a specific point in time.
- The columns are further divided into:
    - **Normal snapshots**, which record basic information and are represented using the ':' character.
    - **Detailed snapshots**, denoted by the '@' character, represent information about where allocations occurred for these snapshots.
    - **Peak snapshots**, represented by the '#' character, record the point where memory consumption was greatest.
- The graph is followed by the information for each snapshot.
- Normal snapshots provide basic memory usage counts, while detailed snapshots include an allocation tree indicating code sections responsible for heap memory allocation.

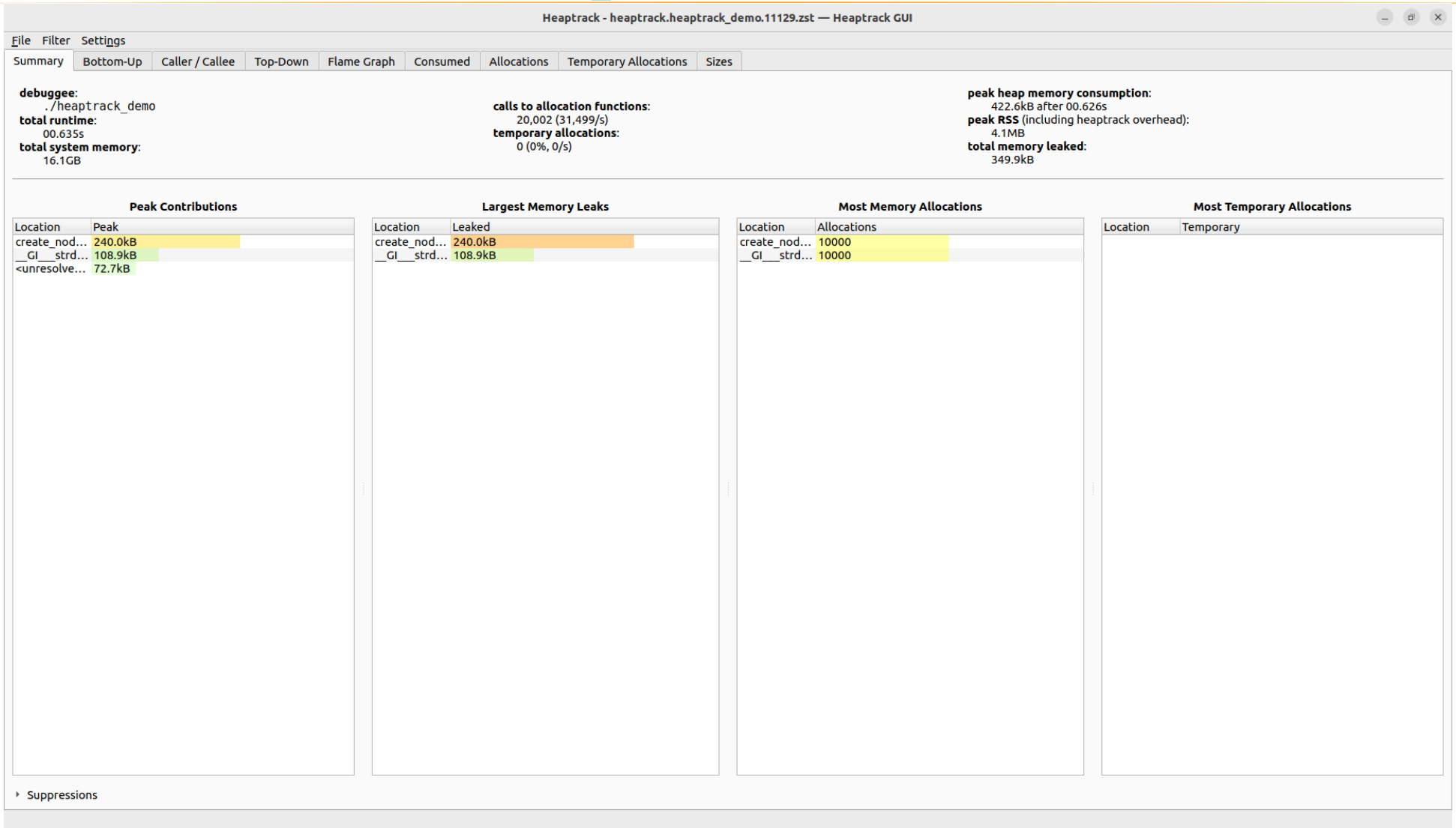
Massif Visualizer



Heaptrack

- Heaptrack traces all memory allocations and annotates them with stack traces.
- Uses `LD_PRELOAD` to intercept memory allocators.
- Analysis tools help identify hotspots for optimization, memory leaks, allocation hotspots, and temporary allocations.
- Heaptrack offers more detailed allocation context compared to Massif and incurs low overhead.
- Execute using `heaptrack {program}`.
- After execution, heaptrack generates a heaptrack.{APP}.{PID}.zst file that can be analyzed using heaptrack_print or heaptrack_gui.
- We are using heaptrack sample program for the demonstration.

Heaptrack GUI



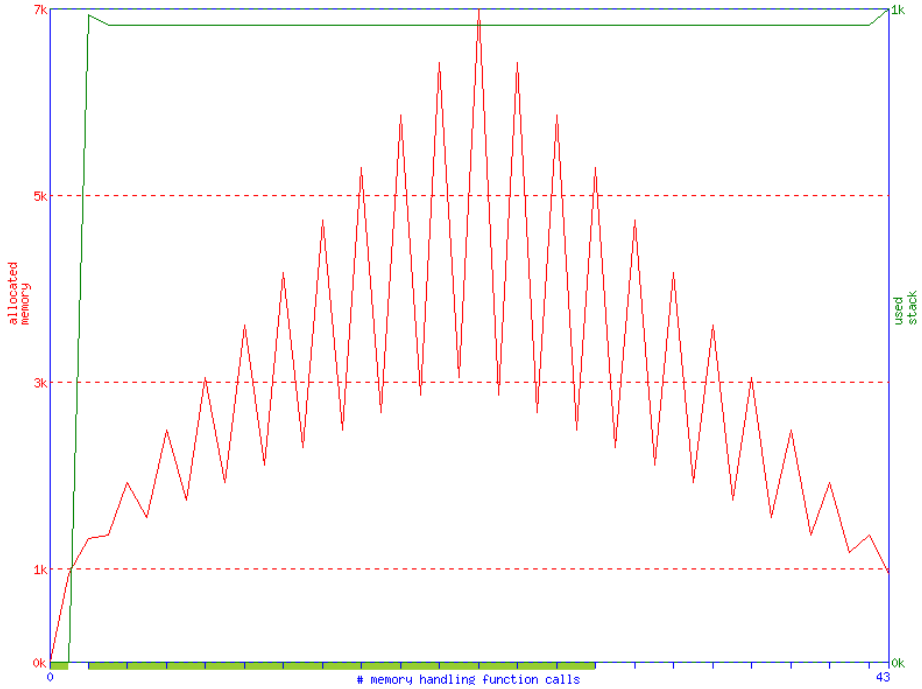
Memusage

- memusage profiles the memory usage of user-space applications.
- It utilizes the `libmemusage.so` library by preloading it into the caller's environment via the `LD_PRELOAD` environment variable.
- libmemusage.so library traces memory allocation by intercepting memory allocators like `malloc`, `calloc`, `free`, `realloc`, `mmap`, `mremap` and `munmap`.
- The output data can be collected in a textual form, or in a PNG file for graphical representation.
- memusage is a part of `libc-devtools` package on Ubuntu.
- Demonstration uses the following memusage example.

Memusage example

```
manas@sandbox:~/work/memusage$ memusage --data=memusage.dat ./userapp
---
Memory usage summary: heap total: 45464, heap peak: 7464, stack peak: 1968
         total calls   total memory   failed calls
 malloc|           2           1424              0
realloc|          40          44040              0  (nomove:37, dec:19, free:0)
 calloc|           0              0              0
   free|           1            440
Histogram for block sizes:
  240-255             1   2% ============
  400-415             1   2% ============
  432-447             3   7% ====================================
  640-655             2   4% ========================
  832-847             2   4% ========================
 1024-1039            1   2% ============
 1040-1055            4   9% ================================================
 1232-1247            2   4% ========================
 1440-1455            2   4% ========================
 1632-1647            4   9% ================================================
 1840-1855            2   4% ========================
 2032-2047            2   4% ========================
 2240-2255            3   7% ====================================
 2832-2847            2   4% ========================
 3440-3455            2   4% ========================
 4032-4047            2   4% ========================
 4640-4655            2   4% ========================
 5232-5247            2   4% ========================
 5840-5855            2   4% ========================
 6432-6447            1   2% ============
```

Memusage graphical representation



- Convert the collected data into a PNG format

```
manas@sandbox:~/work/memusage$ memusagestat memusage.dat memusage.png
```

Execution based profiling

eBPF profiler

- profile-bpfcc is an eBPF-based CPU profiler that captures stack trace samples at regular intervals.
- It helps analyze CPU usage by identifying which code is executing and measuring its impact, covering both user-space and kernel execution.
- Compared to other profilers, eBPF is efficient and low-overhead. It counts stack trace frequencies in the kernel and passes only unique stacks and their counts to user space, reducing kernel-to-user transfers.
- By default, it samples at 49 Hz across all CPUs which is adjustable via a command-line option. Frequencies like 49 Hz or 99 Hz help avoid lock-step sampling issues compared to 50 or 100 Hz.
- The profiler is a part of `bpfcc-tools` package.

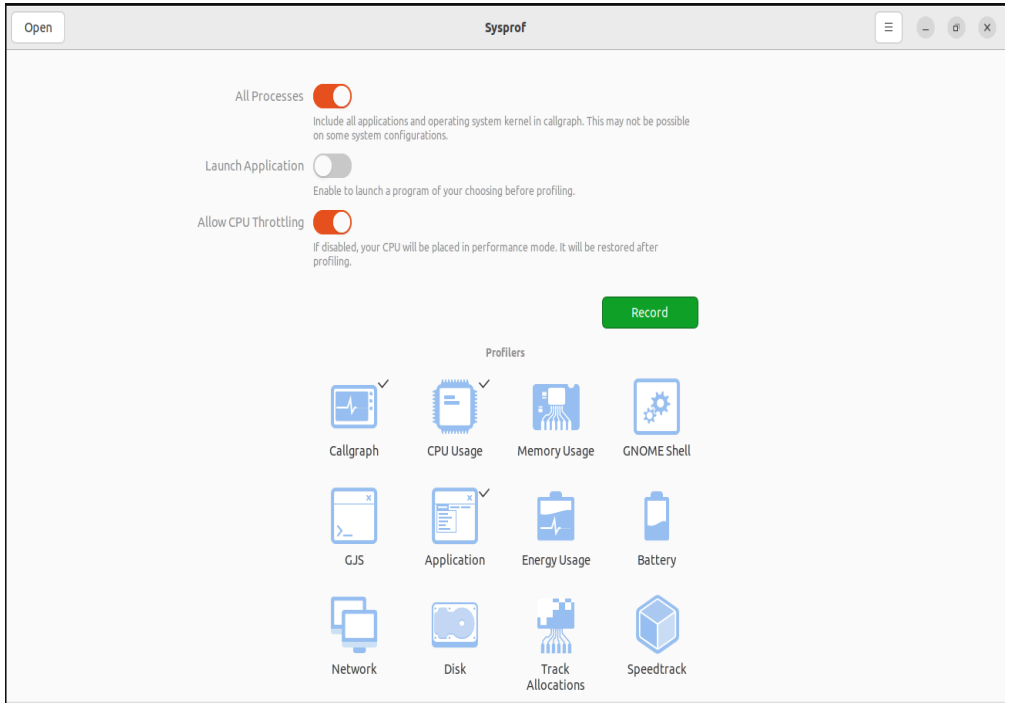
eBPF profiler usage

- The profiler can provide output in folded format directly. We can further pipe the output to `flamegraph.pl` directly to skip the intermediate file.
- The command samples at 49 Hz (-F 49), includes kernel annotations (-a), separates user and kernel stacks with a delimiter (-d), outputs in folded format (-f), and runs for 30 seconds.

```
# git clone https://github.com/brendangregg/FlameGraph

# sudo apt-get install bpfcc-tools

# cd FlameGraph

# Profiler output piped into flamegraph.pl
# profile-bpfcc -F 49 -adf 30 | ./flamegraph.pl > profile.svg

# firefox profile.svg
```

Sysprof (1/2)

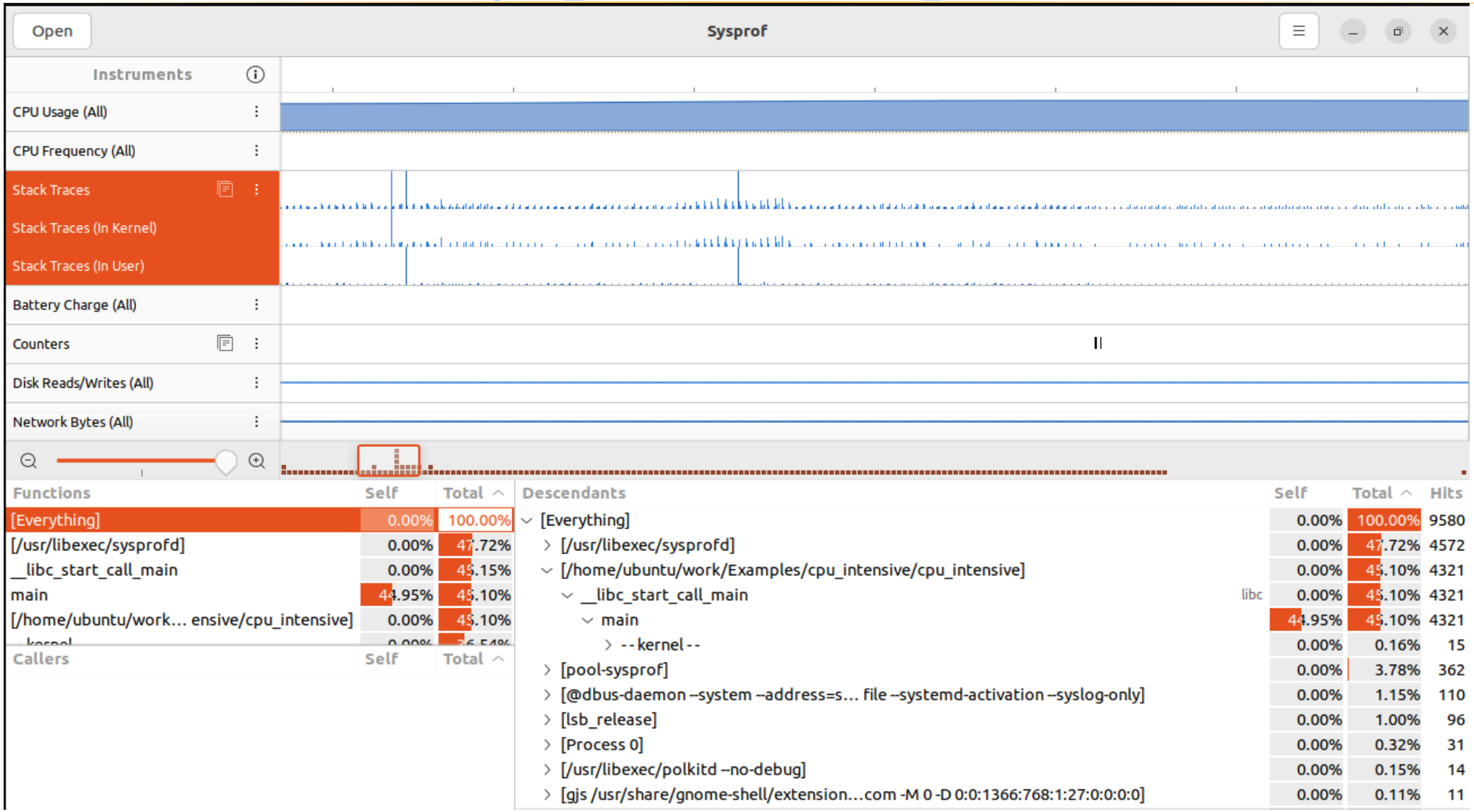
- Sysprof is a Linux sampling CPU profiler which records a stack trace of a process's activity multiple times per second.
- Sysprof is a system-wide Linux profiler that includes the kernel and all userspace applications.
- On Ubuntu platform, install it using `sudo apt install sysprof`.



Sysprof (2/2)

- Sysprof supports profiling of entire systems or specific processes. It also provides the flexibility to launch an application before recording the profile.
- Sysprof analysis includes CPU usage, thread execution, and function call stacks, helping identify performance bottlenecks and optimize applications for better performance.
- It uses frame pointer unwinding to record the stack trace. Ensure to use the `-fno-omit-frame-pointer` compiler option to instruct the compiler to store the stack frame pointer in a register.
- Following links provide detailed insight on frame pointers and working of profiler: Link-1, Link-2 and Link-3.

Sysprof example



GNU Profiler (gprof) [1/2]

- gprof is a profiling tool used to analyze the performance of user space application in terms of function call frequencies and execution times.
- It is typically used with programs written in C or C++ compiled with the `-pg` flag to generate profiling information.

```
$ gcc -o memory main.c -g -pg
```

- After compiling the program with `-pg`, execute it to generate a `gmon.out` file. Then, run gprof with the executable and gmon.out as arguments.

```
# Call graph output
$ gprof memory --graph gmon.out

# Flat Profile output
$ gprof memory -z gmon.out
```

- It produces a report showing the time spent in each function and the number of times each function was called.

GNU Profiler (gprof) [2/2]

- The generated report includes a flat profile (summary of each function) and a call graph (visual representation of function calls).
- Gprof helps identify bottlenecks and areas for optimization in the program's execution.
- It may not be suitable for multithreaded or heavily optimized programs; other tools like perf may be more appropriate in such cases.
- Refer to the gprof manual for detail on usage and limitation.

Callgrind (1/2)

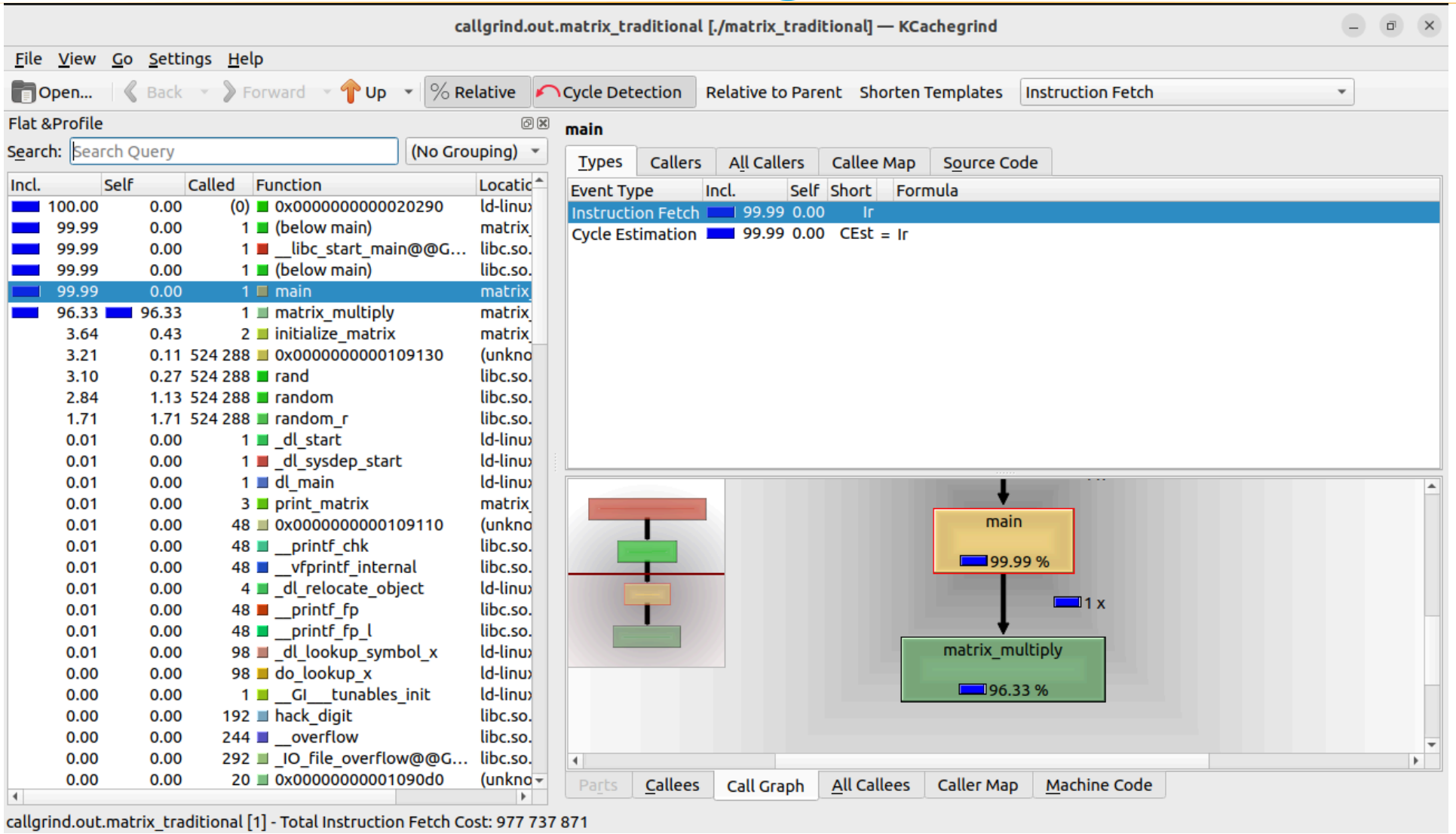
- callgrind is a part of the Valgrind tool suite which records the call history among functions in a program's run as a call-graph.
- The default collected data includes the number of executed instructions, their association with source lines, the caller-callee relationship between functions, and the counts of such calls.
- Callgrind propogates costs across function boundaries.
  - e.g: function foo calls bar, the costs from bar are added into foo's costs.
  - Provides an inclusive costs of entire program, where the cost of each function includes the costs of all functions it called, directly or indirectly.
- Callgrind usage: `valgrind --tool=callgrind [callgrind options] your-program [program options]`

Callgrind (2/2)

```
ubuntu@sandbox:~/work/Examples/matrix/$ valgrind --tool=callgrind ./matrix_traditional
==7474== Callgrind, a call-graph generating cache profiler
==7474== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==7474== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==7474== Command: ./matrix_traditional
...
==7474==
==7474== Events    : Ir
==7474== Collected : 977737730
==7474==
==7474== I   refs:      977,737,730
```

- We are using the matrix multiplication example for the illustration given above.
- The detailed profiling data is written to a file named `callgrind.out.{pid}`
- `callgrind_annotate` is the command line tool which reads in the profile data, and prints a sorted lists of functions, optionally with source annotation.
- `kcachegrind` can provide the callgrind report in GUI form.
- For more details on advance usage, please refer to callgrind manual.

KCachegrind



Cachegrind (1/2)

- Cachegrind is a cache profiler tool included in the Valgrind suite and it simulates program interaction with the CPU cache hierarchy.
- Cachegrind collects precise and reproducible profiling data.
    - **Precise**: Cachegrind precisely counts program instructions and provides detailed data at file, function, and line levels.
    - **Reproducible**: Instruction counts are reliably reproducible, often perfectly so. This enables precise measurement of even minor program changes.
- The cache simulator,simulates a machine with split (Independent instruction and data cache) L1 cache with a unified L2 cache.
- For the accurate results, the program should be compiled with debugging symbols and optimization turned ON.

Cachegrind (2/2)

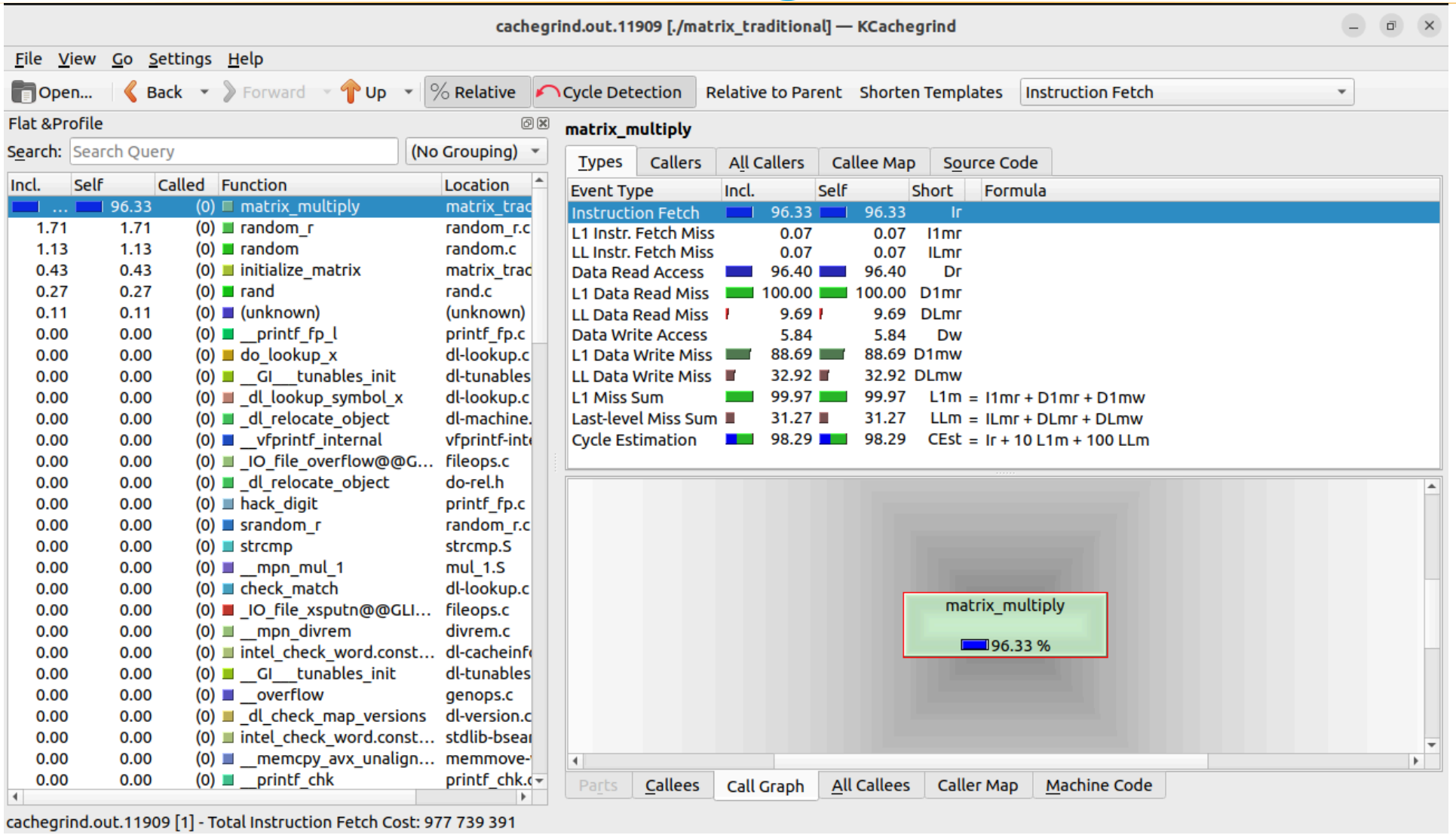
- Usage: `$ valgrind --tool=cachegrind ./program`
- The detailed profiling data is written to a file named `cachegrind.out.{pid}`
- Cachegrind output details cache misses, hit rates, and access patterns, aiding optimization with visualized hotspots.
- `cg_annotate` is a command line tool which summarizes cache-related information per function and annotates source code with color-coded markers for cache events.
- `kcachegrind` can provide the cachegrind report in GUI form.
- For more details on advance usage, please refer to cachegrind manual.

Cachegrind usage

- We are using the matrix multiplication example for the illustration given below.

```
ubuntu@sandbox:~/work/Examples/matrix$ valgrind --tool=cachegrind ./matrix_traditional
==11909== Cachegrind, a cache and branch-prediction profiler
==11909== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==11909== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==11909== Command: ./matrix_traditional
==11909==
...
==11909==
==11909== I   refs:      977,739,391
==11909== I1  misses:          1,532
==11909== LLi misses:          1,525
==11909== I1  miss rate:        0.00%
==11909== LLi miss rate:        0.00%
==11909==
==11909== D   refs:      282,943,530  (278,458,337 rd   + 4,485,193 wr)
==11909== D1  misses:    134,808,940  (134,513,355 rd   +   295,585 wr)
==11909== LLd misses:         51,364  (      1,590 rd   +    49,774 wr)
==11909== D1  miss rate:       47.6% (       48.3%     +       6.6%  )
==11909== LLd miss rate:        0.0% (        0.0%     +       1.1%  )
==11909==
==11909== LL refs:       134,810,472  (134,514,887 rd   +   295,585 wr)
==11909== LL misses:          52,889  (      3,115 rd   +    49,774 wr)
==11909== LL miss rate:         0.0% (        0.0%     +       1.1%  )
```

KCachegrind



Perf stat (1/2)

- perf stat is a Linux performance measurement tool that collects and reports detailed hardware and software performance counters such as CPU cycles, instructions, cache accesses, and branch mispredictions.
- By collecting and displaying detailed performance metrics, perf stat aids in the performance tuning and optimization of applications.
- Use the `perf list` command to view all available hardware and software events that can be monitored.

```
ubuntu@sandbox:~/work/linux-kernel/linux-6.2/tools/perf$ ./perf list
List of pre-defined events (to be used in -e or -M):
  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
...
  cgroup-switches                                    [Software event]
  context-switches OR cs                             [Software event]
...
  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-loads                                    [Hardware cache event]
...
   cache-misses OR cpu/cache-misses/                  [Kernel PMU event]
```

Perf stat (2/2)

- `perf stat` provides options to capture specific events for a command.

```
perf stat -e cycles:uk dd if=/dev/zero of=/dev/null count=100000
```

- We can also use modifiers to count events in userspace (:u) or kernel space (:k).
- If there are more events than counters, the kernel uses multiplexing to give each event a chance to access the monitoring hardware.
- This leads the tool to scale the count and provide an estimate based on total time enabled vs time running.
- Reduce the number of events with each run to avoid scaling.
- Refer to perf wiki for more details.
- The standard Linux perf tool package offers basic support. Following these instructions, building your own version is recommended.

Perf stat example

```
root@sandbox:/home/ubuntu/work/linux-kernel/linux-6.2/tools/perf# ./perf stat -B dd if=/dev/zero of=/dev/null count=1000000
1000000+0 records in
1000000+0 records out
512000000 bytes (512 MB, 488 MiB) copied, 1.76965 s, 289 MB/s

 Performance counter stats for 'dd if=/dev/zero of=/dev/null count=1000000':

          1,770.13 msec task-clock                       #    0.998 CPUs utilized
                14      context-switches                 #    7.909 /sec
                 5      cpu-migrations                   #    2.825 /sec
                80      page-faults                      #   45.194 /sec
     5,47,18,32,553      cycles                           #    3.091 GHz
     2,93,68,65,191      instructions                     #    0.54  insn per cycle
       50,06,69,465      branches                         #  282.843 M/sec
          61,40,189      branch-misses                    #    1.23% of all branches

       1.774383620 seconds time elapsed

       0.781372000 seconds user
       0.989737000 seconds sys
```

- In the example above, `perf stat` captures software events like context switches and generic hardware events like CPU cycles.
- The derived metrics such as "Instruction per cycle" are presented after the '#' sign.

CPU Frequency Profiling

Understanding CPU Frequency Scaling

- Modern CPUs dynamically adjust frequency to balance power and performance.
- Frequency is scaled based on workload demands and system policy — higher frequency = more performance, lower = more power savings.
- Linux uses scaling governors: powersave, performance and schedutil. The OS governor's job is to provide hints or requests to the underlying hardware based scalar.
- Hardware scalar (HWP) offload frequency decisions to the CPU itself. Intel Speed Shift is a prime example of HWP.
- CPU frequency impacts:
    - Response time
    - Instruction throughput
    - Power and thermal envelope

Tools for CPU Frequency Observation

- cpupower: A CLI tool to view and set CPU scaling governors or frequency limits system-wide, such as performance or powersave.
- cpufreq-info: Displays per-core scaling governor, available frequencies, driver (e.g., intel_pstate), and current policy. It is useful to verify CPU capabilities and scaling behavior.
- s-tui: An interactive terminal-based UI that monitors real-time CPU frequency, utilization, temperature, and power. It is ideal for visualizing the impact of workload on CPU scaling.
- turbostat: Provides per-core CPU MHz, Bzy_MHz (active cycles), package power (PkgWatt), and temperature (PkgTemp). It's excellent for measuring fine-grained hardware-level CPU performance under load.
- perf stat: Part of the Linux perf toolset. It tracks instructions, cycles, IPC, user/sys time, and cache events. Best suited for profiling workload efficiency and execution cost.

Demonstration: Powersave vs Performance

- **Workload**: Synthetic CPU stress test using sysbench or stress-ng.

Metric	Expectation in Powersave	Expectation in Performance
CPU Frequency	Low, dynamic	High, locked at max
Instructions	Similar	Similar
Cycles	Higher (slower)	Lower (faster)
IPC	Lower	Higher
Total CPU Time	Longer	Shorter
Power/Temp	Lower	Higher

- Set scaling governer:
  - **Performance**: `sudo cpupower frequency-set -g performance`
  - **Powersave**: `sudo cpupower frequency-set -g powersave`
- Refer to this helper script to dump the CPU frequency results.

Demonstration: Key observation

- Higher and more stable frequencies in performance mode.
- Lower CPU time, higher IPC in perf stat.
- Higher Avg_MHz, Bzy_MHz and higher temparature in turbostat.
- CPU governor has a significant impact on workload efficiency.
  - `performance` mode reduces runtime by keeping frequency high
  - `powersave` mode may hurt latency-sensitive tasks
- Use frequency profiling to:
  - Optimize performance vs power trade-offs
  - Debug unexpected slowdowns in real-world deployments

References (1/2)

- Histogram - Statquest
  - https://www.youtube.com/watch?v=qBigTkBLU6g
- Flamegraph -Brendan Gregg
  - https://www.youtube.com/watch?v=VMpTU15rIZY
- eBPF - Brendan Gregg
  - https://www.youtube.com/watch?v=HKQR7wVapgk
- sysprof
  - https://fedoramagazine.org/performance-profiling-in-fedora-linux/
- Debugging with Frame pointers
  - https://developers.redhat.com/articles/2023/07/31/frame-pointers-untangling-unwinding#
  - https://blogs.oracle.com/linux/post/unwinding-stack-frame-pointers-and-orc

References (2/2)

- Perf tool building instructions - Manas Marawaha.
  - https://medium.com/@manas.marwah/building-perf-tool-fc838f084f71
- SIMD and AVX2 instruction set
  - https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX2
  - https://en.ittrip.xyz/c-language/c-matrix-simd-optimization
  - https://v0dro.in/blog/2018/05/01/building-a-fast-matrix-multiplication-algorithm/

Module 7
Linux Kernel Debugging

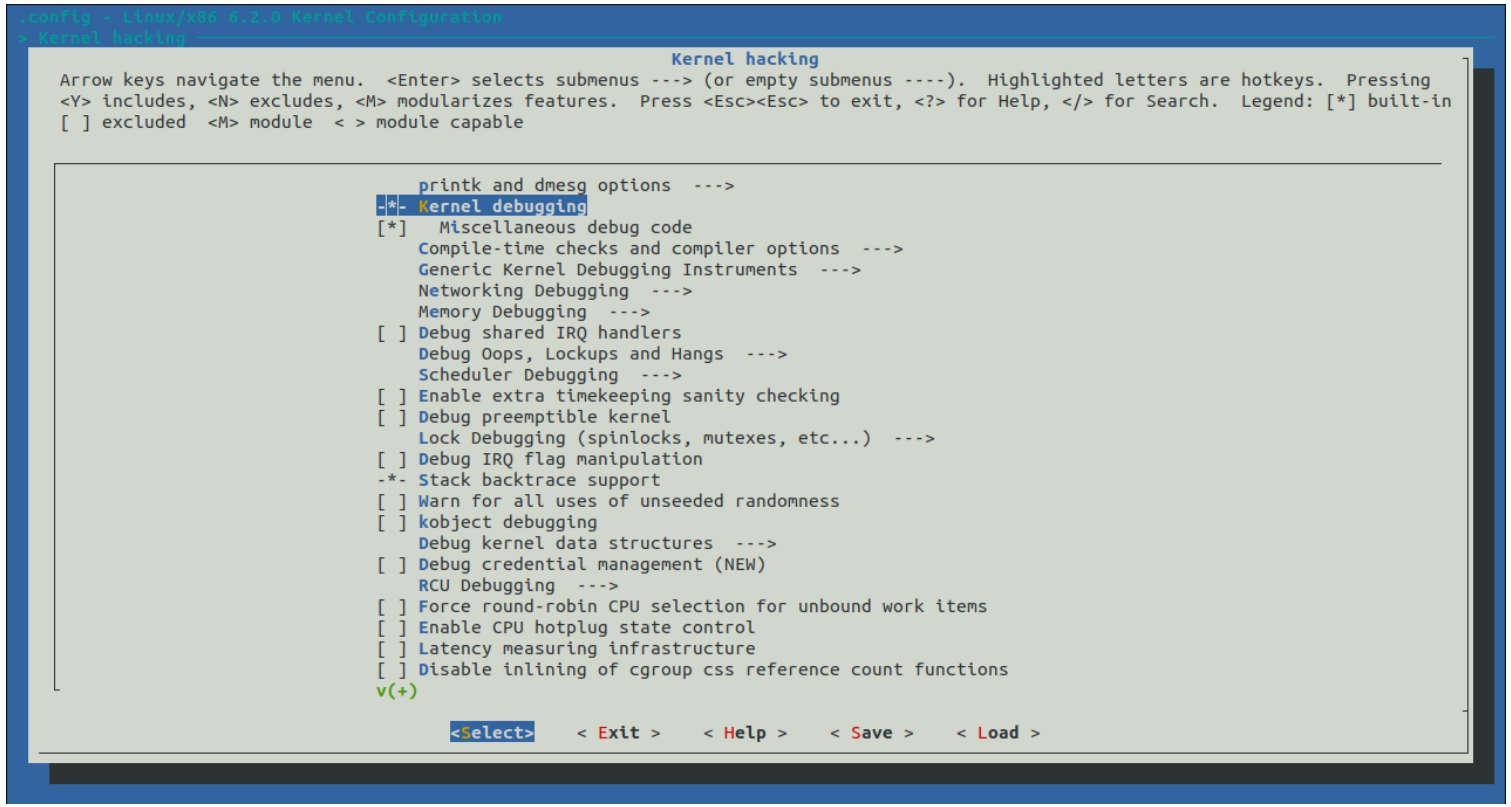
Introduction

- Debugging the Linux kernel is crucial for ensuring system stability, performance, and security, but it is more complex than userspace debugging due to its low-level operations and limited tools.
- Linux kernel provides a range of debugging tools that can be enabled through kernel configuration to aid in the identification and resolution of issues.

Kernel configuration (Kconfig)

- Kconfig is the configuration system used in the Linux kernel to enable or disable features and modules.
- It generates a `.config` file in the kernel source directory which contains all the selected configuration options.
- `make menuconfig` provides a text-based menu interface to configure kernel options through Kconfig.

Kernel Configuration



- Kernel debug features and tools are available in `Kernel hacking -> Kernel debugging` menuconfig entry.
- Enable `CONFIG_DEBUG_KERNEL`, as all other kernel debug options depend on it.

Kernel debugging support

vmlinux

- vmlinux is a uncompressed, statically linked kernel image contains all the debugging symbols and information needed for in-depth analysis.
- It is generally used with debuggers like GDB and crash utilities.

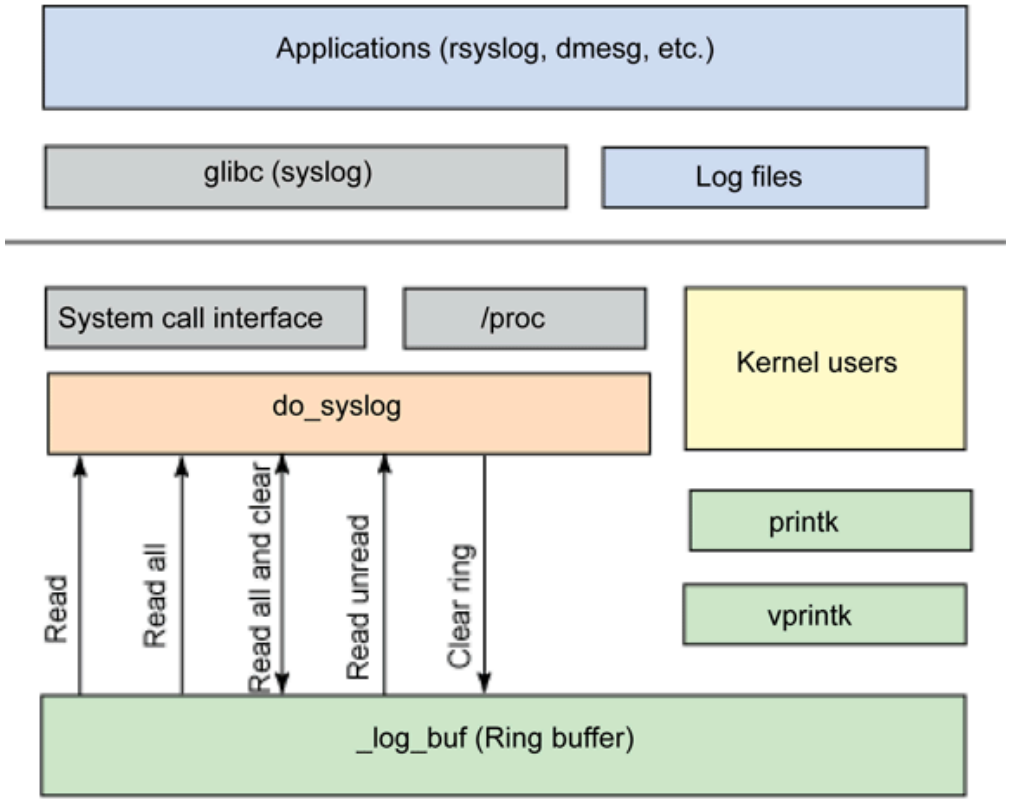
System.map

- System.map is a file containing linux kernel symbol name and its corresponding address.
- It is essential for interpreting kernel oops messages and stack traces.

GDB Scripts

- These are the scripts generated to assist with debugging the kernel using GDB. It automates and simplifies common debugging tasks and procedures.

Kernel logging system overview



Credit: IBM Developerworks

- Kernel code calls printk(), it logs a message that gets stored in a log buffer.
- The log level of the message determines whether it will be printed on the console or sent to user-space logging daemons.
- User-space logging daemons can access kernel logs using `/dev/kmsg`.

Kernel logging utilities (1/4)

printk() and pr_*() family

- printk() formats and print messages into kernel log buffer.
- `printk()` works similarly to printf() and uses a LOG MESSAGE level as a prefix string.

```
printk(KERN_ERR "%s: irq %u value %x status %d\n", __func__, irq, value, status);
```

- The pr_*() family of functions, namely pr_emerg(), pr_alert(), pr_crit(), pr_err(), pr_warning(), pr_notice(), pr_info(), pr_debug() and pr_cont(), are the recommended logging functions over printk.

```
pr_info("Ftrace startup test is disabled due to %s\n", reason);
```

- The pr_*() family of functions are macros that wrap `printk` calls with specific log levels and are recommended for use in the kernel.

Kernel logging utilities (2/4)

dev_*() family

- The dev_*() family of functions, such as dev_emerg(), dev_alert(), dev_crit(), dev_err(), dev_warn(), dev_notice(), dev_info() and dev_dbg() are the recommended logging functions for use in device drivers.

Format specifier for kernel logging

- Refer to the printk-format for the full list of format specifiers.
  - %p: Display the hashed value of pointer by default.
  - %pe: Display the string corresponding to the error number.
  - %pS: Symbol pointer (versatile_init+0x0/0x110).
  - %pa: Display physical address.
  - %pK: Kernel pointer to display hashed pointer (depends on kptr_restrict sysctl value).
  - %pOF: Device-tree node format specifier.

Kernel logging utilities (3/4)

kptr_restrict

- kptr_restrict controls the restrictions placed on exposing kernel addresses via /proc and other interfaces.
- It is accessed through `/proc/sys/kernel/kptr_restrict` interface.
  - kptr_restrict = 0, No restriction (full address is printed either hashed or unhashed depending on the format specifier)
  - kptr_restrict = 1, Restricted visibility, root users can see the pointers.
  - kptr_restrict = 2, Kernel pointers are completely masked in all contexts, including for root users and replaced with Zeros such as 0x00000000.

Kernel logging utilities (4/4)

dmesg

- Print or control the kernel log buffer.
  - `dmesg` : Print the kernel log buffer
  - `dmesg -C` : Clear the kernel log buffer
  - `dmesg --level=err,warn` : Print error and warning message only.
  - `dmesg -n 5` : Set console logging filter to KERN_WARNING or more.

Demonstration

- Please refer to this link for sample code related to the demonstration of kernel logging utilities.

Kernel dynamic debugging (1/5)

- pr_debug and dev_debug macros are defined in the kernel with the lowest message level 7 - KERN_DEBUG.
- These functions are used to print debug messages only when the `DEBUG` compiler macro is defined, which can be enabled by either of the following methods:
    - Enable debug options in Kernel config.
    - Add -DDEBUG in Makefile ccflags.
    - Add "#define DEBUG" in source code.
- Every change in a DEBUG message requires rebuilding the kernel or kernel module.
- The kernel's dynamic debugging feature allows enabling or disabling debug messages at runtime without the need to rebuild the kernel or kernel module.
- Enable `CONFIG_DYNAMIC_DEBUG=y` in the kernel config to turn ON dynamic debug feature in the kernel.

Kernel dynamic debugging (2/5)

- Dynamic debug can be controlled through `/sys/kernel/debug/dynamic_debug/control` interface.
- Mount debugfs to access dynamic debugging feature.

```
root@sandbox:~# mount -t debugfs none /sys/kernel/debug/
```

- The feature allows enabling/disabling debug prints by matching any combination of:
  - Source filename
  - Function name
  - Line number (including ranges of line numbers)
  - Module name
  - Format string

Kernel dynamic debugging (3/5)

- Dynamic debugging syntax and usage.

```
echo "<matches> <ops><flags>" > <debugfs>/dynamic_debug/control
```

```
root@sandbox:~# cd /sys/kernel/debug/dynamic_debug/
# Enable dynamic debugging based on file.
echo "file svcsock.c +p" > control

# Enable dynamic debugging based on multiple file.
echo "file drivers/usb/core/* +p" > control

# Enable dynamic debugging based on line number.
echo "file svcsock.c line 1603 +p" > control

# Enable dynamic debugging based on function name.
echo "func svc_tcp_accept +p" > control

# Enable dynamic debugging based on kernel module name.
echo "module nfsd +p" > control

# Disable dynamic debugging based on file.
echo "file svcsock.c -p" > control

# view the currently configured behaviour of all the debug statements.
cat control
```

Kernel dynamic debugging (4/5)

- When using dynamic debugging in kernel modules there are certain things we need to be aware of.
  - The module must be inserted before we can enable dynamic debug.
  - This means we will not see debug statements in the init section of a module.
  - We can confirm that a specific line has been enabled by observing the output of `cat /sys/kernel/debug/dynamic_debug/control` and looking for the `=p` indicator. For example:
  - `dynamic_debug_example.c:85 [dynamic_debug_example]dynamic_debug_function_two =p "dynamic_debug_example: In dynamic_debug_function_two, called with message: %s\n"`
  - This indicates that dynamic debug has been enabled
  - We can then observe the output in dmesg by using a suitable filter: `dmesg | grep dynamic_debug`

Kernel dynamic debugging (5/5)

- Dynamic debug can be enabled from U-boot to allow debugging kernel core code and built-in module during boot phase.
  - use dyndbg="QUERY" for kernel code.
  - use module.dyndbg="QUERY" for built-in module.

```
dyndbg="file ec.c +p"
```

- print_hex_dump_debug() and print_hex_dump_bytes() calls can be enabled dynamically.
- If neither `DEBUG` nor `CONFIG_DYNAMIC_DEBUG` are enabled, these messages are not compiled into the kernel.

Dynamic debuggging case study

- The code used in this example is available at: https://github.com/SpecialistLinuxTraining/linux-debug-training/blob/main/Examples/dynamic_debug

Kernel crash debugging

Kernel crash debugging

- The Linux kernel identifies and logs a "kernel oops" when it encounters a critical,non-fatal error, such as:
  - Invalid memory access (Null pointer dereference, out of bound access).
  - Deadlock detection.
  - Incorrect execution mode (sleeping in atomic context, Enabling Preemption in Critical Sections).
  - Voluntary oops (using WARN_ON() or WARN_ON_ONCE())
- Depending on the severity of the crash, the kernel may either continue running in a degraded state or completely hang.
- The oops message provides detailed information such as an error summary, CPU register states, stack dump, and backtrace, which helps developers in performing a "post-mortem analysis" of the kernel.

Kernel OOPS (1/2)

- The kernel oops message is displayed on the console and logged in dmesg.
- Below is an annotated kernel OOPS snapshot generated using a kernel module example.

```
manas@sandbox:~/work/Examples/crash_kernel_module$ sudo insmod crash-module.ko

========================== ERROR SUMMARY ==========================
[  234.823615] BUG: kernel NULL pointer dereference, address: 0000000000000000
[  234.823637] #PF: supervisor read access in kernel mode
[  234.823646] #PF: error_code(0x0000) - not-present page
[  234.823655] PGD 0 P4D 0
[  234.823663] Oops: Oops: 0000 [#1] PREEMPT SMP PTI

============== CPU# / PID# / KERNEL / Hardware Name ===============
[  234.823674] CPU: 0 UID: 0 PID: 457 Comm: insmod Tainted: G           OE      6.13.8-arch1-1 #1 456fce73f556a6ef8edc444db474316a8147249d
[  234.823700] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006

========================== CPU REGISTERS ==========================
[  234.823710] RIP: 0010:crash_module_init+0x15/0x30 [crash_module]
[  234.823722] Code: Unable to access opcode bytes at 0xffffffffc0a06ffb.
[  234.823730] RSP: 0018:ffffa32e812cbc48 EFLAGS: 00010246
[  234.823739] RAX: 0000000000000029 RBX: ffffffffc0a07010 RCX: 0000000000000027
[  234.823748] RDX: 0000000000000000 RSI: 0000000000000001 RDI: ffff89c19bc218c0
[  234.823756] RBP: 0000000000000000 R08: 0000000000000000 R09: ffffa32e812cbad8
[  234.823764] R10: ffffffffa46b5448 R11: 0000000000000003 R12: 000062a47eb11a1b
[  234.823773] R13: ffffa32e812cbc50 R14: ffff89c184d20cc0 R15: ffff89c1822399b8
[  234.823782] FS:  000070886016c740(0000) GS:ffff89c19bc00000(0000) knlGS:0000000000000000
[  234.823792] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  234.823800] CR2: ffffffffc0a06ffb CR3: 000000010a74a000 CR4: 00000000000506f0
```

Kernel OOPS (2/2)

```
====================  BACK TRACE  ====================

[  234.823813] Call Trace:
[  234.823819]  <TASK>
[  234.823824]  ? __die_body.cold+0x19/0x27
[  234.823834]  ? page_fault_oops+0x15c/0x2e0
[  234.823842]  ? search_bpf_extables+0x5f/0x80
[  234.823851]  ? exc_page_fault+0x81/0x190
[  234.823864]  ? asm_exc_page_fault+0x26/0x30
[  234.823872]  ? __pfx_crash_module_init+0x10/0x10 [crash_module 11a11b817c8e53686efb40be3365f6d893cbe205]
[  234.823884]  ? crash_module_init+0x15/0x30 [crash_module 11a11b817c8e53686efb40be3365f6d893cbe205]
[  234.823895]  ? crash_module_init+0x15/0x30 [crash_module 11a11b817c8e53686efb40be3365f6d893cbe205]
[  234.823905]  do_one_initcall+0x5b/0x310
[  234.823915]  do_init_module+0x60/0x230
[  234.823923]  init_module_from_file+0x89/0xe0
[  234.823932]  idempotent_init_module+0x115/0x310
[  234.823940]  __x64_sys_finit_module+0x65/0xc0
[  234.823948]  do_syscall_64+0x82/0x190
[  234.823956]  ? switch_fpu_return+0x4e/0xd0
[  234.823964]  ? arch_exit_to_user_mode_prepare.isra.0+0x79/0x90
[  234.823974]  ? syscall_exit_to_user_mode+0x37/0x1c0
[  234.823982]  ? do_syscall_64+0x8e/0x190
```

- The structure and content of the kernel oops message vary depending on the architecture used.

Kernel OOPS Analysis (1/4)

- Enable CONFIG_KALLSYMS to embed symbols in the kernel image and display them in backtraces using the following format.

```
<SYMBOL NAME>+<HEX OFFSET>/<SIZE>
```

- The program counter register (named RIP, IP, or EIP depending on the architecture) indicates the exact location in the code where the crash occurred.

```
[  234.823710] RIP: 0010:crash_module_init+0x15/0x30 [crash_module]
```

- There are tools available to analyze kernel oops and identify the exact fault location in the code.

Kernel OOPS Analysis (2/4)

Linux Debug Training

GDB

```
:~/Examples/crash_kernel_module$ gdb crash-module.ko
Reading symbols from crash-module.ko...
(gdb) list *(crash_module_init+0x15)
0x25 is in crash_module_init (crash-module.c:10).
5       static int crash_module_init(void)
6       {
7           // Method 1: Explicitly assign NULL
8           int *p = NULL;
9           printk(KERN_INFO "Attempting to dereference NULL pointer...\n");
10          printk(KERN_INFO "Value at address p: %d\n", *p); // This line will cause the oops!
11          return 0;                                         // This line will likely not be reached
12      }
13
14      static void crash_module_exit(void)
```

- The `gdb` tool can be used to display the source code at the location indicated by the symbol in the program counter (PC) register.
- In the example above, line number 10 attempts to dereference a pointer with an invalid address.

Kernel OOPS Analysis (3/4)

addr2line and nm

- Identify the starting address of the `crash_module_init()` function in the kernel module file (crash-module.ko). The nm command can be used to list the symbols within a module.

```
ubuntu@sandbox:~/crash_kernel_module$  nm crash-module.ko | grep -w crash_module_init
0000000000000010 t crash_module_init
```

- Add the offset (0x15: Shown in crash report) to the address retrieved using nm (0x10: crash_module_init()). Then, use the addr2line tool to map this address to the corresponding line in the source file.

```
ubuntu@sandbox:~/crash_kernel_module$  addr2line -fe crash-module.ko 0x25 crash_module_init
crash_module_init
/home/developer/Examples/crash_kernel_module/crash-module.c:10 (discriminator 1)
__pfx_crash_module_init
crash-module.c:?
```

Kernel OOPS Analysis (4/4)

objdump

- Use objdump to disassemble the object file.

```
ubuntu@sandbox:~/work objdump -r -S -l --disassemble  crash-module.ko
...
crash_module_init():
/home/developer/Examples/crash_kernel_module/crash-module.c:6
static int crash_module_init(void) {
  10:   f3 0f 1e fa             endbr64
  14:   e8 00 00 00 00          call   19 <init_module+0x9>
  ...
  19:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi / 1c: R_X86_64_32S        .rodata.str1.8
  20:   e8 00 00 00 00          call   25 <init_module+0x15>
                        21: R_X86_64_PLT32      _printk-0x4
/home/developer/Examples/crash_kernel_module/crash-module.c:10 (discriminator 1)
    printk(KERN_INFO "Value at address p: %d\n", *p); // This line will cause the oops!
  25:   8b 34 25 00 00 00 00    mov    0x0,%esi
  2c:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi
```

- The `crash_module_init()` function starts at address 0x10, and adding an offset of 0x15 brings us to the `printk` statement located at address 0x25.

Kernel Crash Analysis Using Sysrq

Kernel Crash Analysis [SysRq] (1/3)

- SysRq is a key combination (typically Alt + SysRq + , where SysRq is often the Print Screen key) that allows users to send commands directly to the kernel, bypassing the usual user-space layers.
- SysRq provides a way to interact with the kernel even when the system is mostly unresponsive. This is particularly valuable for diagnosing system hangs.
- The previous snapshot represents a **Kernel Panic**, which was intentionally triggered by crashing the kernel using the magic SysRq key.
- Please refer to the further slides or this document for more information about the Magic SysRq key.
- The call trace indicates that the kernel panic occurred after the `sysrq_handle_crash+0x1a/0x20` function was called.

Kernel Crash Analysis [SysRq] (2/3)

GDB

- Load the `vmlinux` file into GDB. The vmlinux file may have prebuilt paths that need to be replaced with your own Linux source code path using the `set substitute-path` command in GDB.
- Use the list command in GDB to identify the exact source line causing the panic.

```
root@sandbox:~# gdb /usr/lib/debug/boot/vmlinux-6.5.0-44-generic
Reading symbols from /usr/lib/debug/boot/vmlinux-6.5.0-44-generic...
(gdb) set substitute-path /build/linux-hwe-6.5-QkbMh4/linux-hwe-6.5-6.5.0/ /home/ubuntu/work/linux-kernel/linux-6.5/
(gdb)  l *(sysrq_handle_crash+0x1a)
0xffffffff81a8a4fa is at /build/linux-hwe-6.5-QkbMh4/linux-hwe-6.5-6.5.0/drivers/tty/sysrq.c:155.
150     static void sysrq_handle_crash(int key)
151     {
152             /* release the RCU read lock before crashing */
153             rcu_read_unlock();
154
155             panic("sysrq triggered crash\n");
156     }
```

Kernel Crash Analysis [SysRq] (3/3)

addr2line and nm

- Identify the starting address of the `sysrq_handle_crash()` function in the vmlinux file using nm command

```
root@sandbox:~# nm /usr/lib/debug/boot/vmlinux-6.5.0-44-generic | grep -wi sysrq_handle_crash
ffffffff81a8a4e0 t sysrq_handle_crash
```

- Add the offset (0x1a: Shown in crash message) to the address retrieved using nm (0xffffffff81a8a4e0: sysrq_handle_crash()). Then, use the addr2line tool to map this address to the corresponding line in the source file.

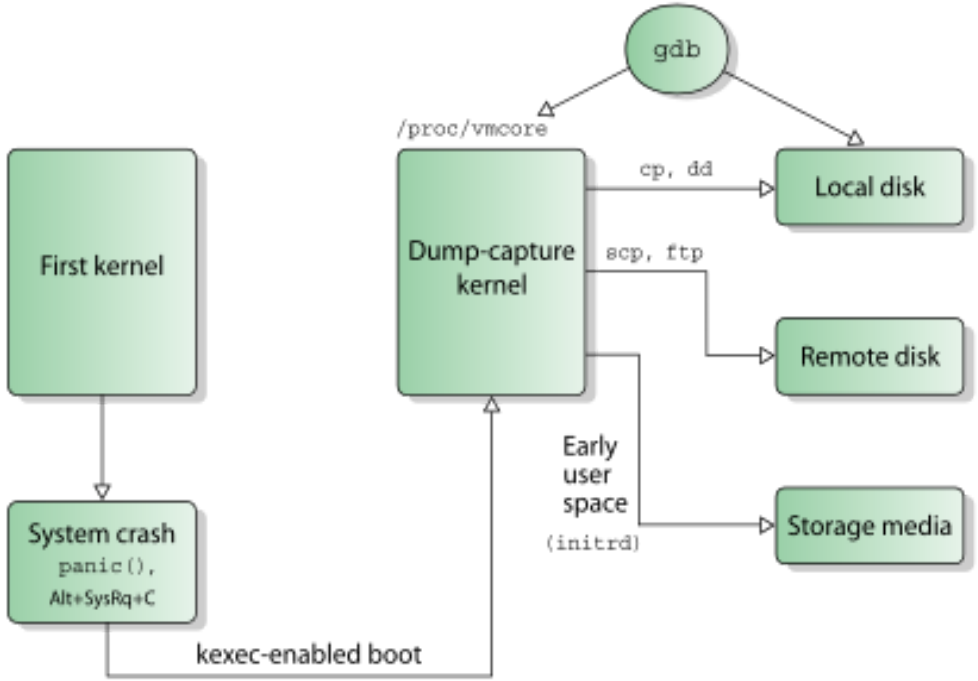
```
root@sandbox:~# addr2line -fe /usr/lib/debug/boot/vmlinux-6.5.0-44-generic 0xffffffff81a8a4fa sysrq_handle_crash
sysrq_handle_crash
/build/linux-hwe-6.5-QkbMh4/linux-hwe-6.5-6.5.0/drivers/tty/sysrq.c:155
```

```
155             panic("sysrq triggered crash\n");
```

Kexec and Kdump (1/3)

- Kexec and Kdump are mechanisms in the Linux kernel used to capture and recover the kernel memory snapshot (/proc/vmcore) after a kernel panic, without requiring a full system reboot.

- It enables the preservation and retrieval of the system kernel's memory image, which can later be used for detailed post-mortem analysis to diagnose the cause of the crash.



Kexec and Kdump (2/3)

- On panic, the kernel's kexec support allows the execution of a "dump-capture" kernel directly from the crashed kernel.
    - The "dump-capture" or Kdump kernel includes only the essential device drivers and features required to capture the crash dump.
- Kexec reserves a small section of memory in the RAM for the dump-capture kernel.
    - Use `crashkernel` kernel boot parameter to specify the memory region for dump-capture kernel.
- The `kexec -p` command loads the dump-capture kernel into the reserved memory.
- After booting from the "dump-capture" kernel, the kernel coredump found at `/proc/vmcore` can be stored as a file on the local filesystem or sent remotely over NFS or SSH.

Kexec and Kdump (3/3)

- You must ensure that the following kernel configuration options have been set in order to use the kexec and kdump functionality.

```
CONFIG_KEXEC_CORE=y (Core kexec functionality)
CONFIG_KEXEC_FILE=y (Newer kexec_file_load syscall)
CONFIG_CRASH_DUMP=y (Enables kdump functionality)
CONFIG_PROC_VMCORE=y (Exposes crash dump via /proc/vmcore)
CONFIG_DEBU_INFO=y (CRUCIAL for debugging symbols)
```

- Please refer to these instructions on configuring kdump and kexec on Ubuntu 22.04 and these instructions for configuring the same utilities on Arch Linux.

Crash Utility

- The crash tool is used to analyze state of the live kernel or memory dump (vmcore) collected from a crashed Linux system.
- The crash tool integrates with GDB, combining the kernel-specific functionality of the crash utility with GDB's source-level debugging capabilities to provide a "kernel-aware" debugging tool.
- It enables interactive post-mortem analysis of kernel data, including CPU registers, process states, memory, kernel symbols, backtraces, and loaded modules from the memory dump.
- The command set of crash utility generaly falls into four category.
  - Kernel data display: struct, union, sym, dis, etc.
  - System state: bt, dev, ps sys, task, vm, etc.
  - Utility functions: ascii, btop, ptob, eval, rd, wr, etc.
  - Session Control Commands: set, alias, exit, foreach etc.

Crash utility usage

- Refer to the instructions on installing the `vmlinux` file on Ubuntu 22.04, which is required by the crash utility to analyze core dumps. For more details on available command set consult the crash dump manual.

```
root@sandbox:/var/crash/202409061559# crash /usr/lib/debug/boot/vmlinux-6.8.0-40-generic dump.202409061559
...
      KERNEL: /usr/lib/debug/boot/vmlinux-6.8.0-40-generic
    DUMPFILE: dump.202409061559  [PARTIAL DUMP]
        CPUS: 32
        DATE: Fri Sep  6 15:59:05 IST 2024
      UPTIME: 00:34:52
LOAD AVERAGE: 0.20, 0.43, 0.36
       TASKS: 947
    NODENAME: sandbox
     RELEASE: 6.8.0-40-generic
     VERSION: #40~22.04.3-Ubuntu SMP PREEMPT_DYNAMIC Tue Jul 30 17:30:19 UTC 2
     MACHINE: x86_64  (2000 Mhz)
      MEMORY: 15.7 GB
       PANIC: "Kernel panic - not syncing: sysrq triggered crash"
         PID: 2711
     COMMAND: "bash"
        TASK: ffff952455685200  [THREAD_INFO: ffff952455685200]
         CPU: 14
       STATE: TASK_RUNNING (PANIC)
crash>
```

Magic SysRq

- Magic SysRq is a key combination used for low-level kernel commands to allow direct interaction with the kernel even when the system is unresponsive.
  - Enable it using `/proc/sys/kernel/sysrq`, where specific functionalities can be allowed or restricted.
- It trigger kernel functions like reboot, crash dump capture, process killing, or syncing disks.
  - On PCs: Trigger it using `ALT + SysRq + <command key>` key combination.
  - On Embedded Systems: Send a break character in the console using `([Ctrl] + a followed by [Ctrl] + \)`, then press command key.
  - Using command line: `echo <command character> > /proc/sysrq-trigger`.

Magic SysRq commands

- Here are some example commands available for use. For more details, please refer to the official documentation.
  - `h` : help
  - `c` : Perform a system crash and a crashdump will be taken if configured.
  - `e` : Send a SIGTERM to all processes, except for init.
  - `g` : Used by kgdb (kernel debugger).
  - `i` : Send a SIGKILL to all processes, except for init.
  - `l` : Shows a stack backtrace for all active CPUs.
  - `p` : Will dump the current registers and flags to your console.
  - `w` : Dumps tasks that are in uninterruptible (blocked) state.
- The functionality also offers an option to register custom commands, which can be triggered using SysRQ.

Kernel lock debugging

Lockdep (1/2)

- Lockdep (Lock Dependency Validator) is a built-in Linux kernel tool used for lock debugging. It detects and report potential deadlocks, lock order violations, and improper lock dependencies.
- Lockdep tracks lock acquisitions, releases, and their order during runtime to ensure safe locking practices. Information for debugging lock issues can be found in the lock debugging file located in `/proc`.

```
/proc/lockdep
/proc/lockdep_chains
/proc/lockdep_stat
/proc/locks
/proc/lock_stats
```

- Use `grep "lock-classes" /proc/lockdep_stats` to track the number of lock classes in use. If the count increases over time, it indicates a possible leak.
- Run `grep "BD" /proc/lockdep`, save the output, and compare it later to spot leaking lock classes or missing runtime lock initialization.

Lockdep (2/2)

- Enable the following kernel configurations for lock debugging to catch issues related to locking problems. Please refer to this demonstration code.

```
 Kernel hacking > Lock Debugging (spinlocks, mutexes, etc...)

[*] Lock debugging: prove locking correctness
[ ]   Enable raw_spinlock - spinlock nesting checks (NEW)
[*] Lock usage statistics
-*- RT Mutex debugging, deadlock detection
-*- Spinlock and rw-lock debugging: basic checks
-*- Mutex debugging: basic checks
-*- Wait/wound mutex debugging: Slowpath testing
-*- RW Semaphore debugging: basic checks
-*- Lock debugging: detect incorrect freeing of live locks
(15) Bitsize for MAX_LOCKDEP_ENTRIES (NEW)
(16) Bitsize for MAX_LOCKDEP_CHAINS (NEW)
(19) Bitsize for MAX_STACK_TRACE_ENTRIES (NEW)
(14) Bitsize for STACK_TRACE_HASH_SIZE (NEW)
(12) Bitsize for elements in circular_queue struct (NEW)
[*] Lock dependency engine debugging
[*] Sleep inside atomic section checking
[ ] Locking API boot-time self-tests
<M> torture tests for locking
<M> Wait/wound mutex selftests
< > torture tests for smp_call_function*()
[ ] Debugging for csd_lock_wait(), called from smp_call_function*()
```

Kernel Concurrency Sanitizer (KCSAN)

- KCSAN is a dynamic race condition detector for the Linux kernel that identifies data races in concurrent code, where multiple threads access shared data without proper synchronization.
    - Suitable for production environments due to low overhead.
- KCSAN uses compiler instrumentation to add watchpoints to memory locations accessed by different threads.
- It monitors these accesses and flags any unsynchronized or conflicting memory operations as potential race conditions.
- To enable KCSAN configure the kernel with CONFIG_KCSAN=y Kconfig option.
    - Use `/sys/kernel/debug/kcsan` interface to configure KCSAN.
- When a race condition is detected, KCSAN logs the memory location, conflicting threads, and a stack trace for debugging.
- Please refer to the KCSAN demonstration code

Kernel Memory debugging

Kmemleak (1/3)

- Kmemleak is a memory leak detector for the Linux kernel, designed to identify possible memory leaks that are no longer referenced but has not been freed.
    - Only suitable for debugging purposes (it should not be used on production systems).
- It uses an rbtree (red-black tree) data structure to efficiently scan and track memory allocations and deallocations in the kernel.
- Kmemleak can be enabled via:
    - CONFIG_DEBUG_KMEMLEAK Kconfig in "kernel hacking" option.
    - `kmemleak=on` kernel command line parameter.
- It can be controlled through debugfs `/sys/kernel/debug/kmemleak` .
- A kernel thread scans the memory every 10 minutes (by default) and prints the number of new unreferenced objects found.

Kmemleak (2/3)

- To display the details of all the possible scanned memory leaks:

```
cat /sys/kernel/debug/kmemleak
```

- To trigger an intermediate memory scan:

```
echo scan > /sys/kernel/debug/kmemleak
```

- To clear the list of all current possible memory leaks:

```
echo clear > /sys/kernel/debug/kmemleak
```

- New leaks will then come up upon reading `/sys/kernel/debug/kmemleak` again.
- Please refer to the Kmemleak demonstration code.

Kmemleak (3/3)

```
echo scan > /sys/kernel/debug/kmemleak
[   62.888171] kmemleak: 1 new suspected memory leak (see /sys/kernel/debug/kmemleak)

cat /sys/kernel/debug/kmemleak

unreferenced object 0x83f92800 (size 1024):
  comm "insmod", pid 155, jiffies 4294939410
  hex dump (first 32 bytes):
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  backtrace (crc 9c453d41):
    kmemleak_alloc+0x44/0x4c
    __kmalloc_cache_noprof+0x2d4/0x310
    0x7f08b024
    do_one_initcall+0x50/0x214
    do_init_module+0x5c/0x228
    init_module_from_file+0x9c/0xd8
    sys_finit_module+0x1ac/0x304
    ret_fast_syscall+0x0/0x54
```

- Kmemleak found a memory object at address 0x83f92800 of size 1024 bytes that is not reachable by any known pointer - a likely memory leak.
- The hex dump shows the first 32 bytes of the leaked memory region, all zeroed; this is typical for an allocated but unused block.

Kernel Address Sanitizer (KASAN)

- KASAN is a dynamic memory error detector for the Linux kernel which is designed to find issues like use-after-free, out-of-bounds memory accesses.
    - Only suitable for debugging purposes (it should not be used on production systems).
- KASAN uses shadow memory (metadata) to track the status of each byte in the kernel's memory to indicating whether it is accessible or has been poisoned.
- Compile-time instrumentation allows the insertion of shadow memory checks before each memory access.
- KASAN can be enabled using:
    - CONFIG_KASAN=y kconfig option
    - `kasan=on` Kernel command line parameter.
- When a memory error is detected, KASAN outputs a detailed report to the kernel log, including the memory location, error type, and call trace.
- Please refer to the KASAN demonstration code.

Kernel Electric-Fence (KFENCE) (1/2)

- KFENCE is a low-overhead sampling-based memory error detector, aimed at detecting heap out-of-bounds access, use-after-free, and invalid-free errors.
    - Suitable for production system due to near zero performance overhead.
- KFENCE works by surrounding specific memory allocations with guard pages (unmapped memory) which serve as protective barriers around the allocated memory to detect out-of-bounds access.
- Any access beyond the allocated memory triggers a page fault upon hitting a guard page.
- KFENCE intercepts this page fault and logs detailed debugging information, including the fault location, allocation size, and access pattern.
- It can be enabled using CONFIG_KFENCE=y Kconfig option.

Kernel Electric-Fence (KFENCE) (2/2)

- KFENCE uses a sampling-based approach to randomly choose memory allocations that it will monitor.
    - Only a subset of allocations will be protected by guard pages at any given time.
- The sampling-based approach significantly reduces performance overhead compared to tools like KASAN, making KFENCE suitable for production systems.
- Compared to KASAN, KFENCE prioritizes performance over precision, relying on extended uptime to catch bugs in less frequently exercised code paths that may be missed by non-production testing.
    - KASAN and KFENCE are complementary, with different target environments.
- Please refer to the KFENCE demonstration code.

Undefined Behavior Sanitizer (UBSAN)

- UBSAN is a runtime error detection tool that identifies undefined behavior in both userspace and kernel space code.
    - Signed integer overflow.
    - Invalid Shift operations for example, shifting by a negative or too large number.
    - Dereferencing misaligned or null pointers.
    - Type mismatch or invalid casts between different types.
    - Refer to the exhaustive list of undefined behaviors in this document.
- UBSAN instruments the code at compile-time by adding checks to detect undefined behavior at runtime.
- Enabled by configuring the kernel with CONFIG_UBSAN and the CONFIG_UBSAN* family for various specific checks.
- When an issue is detected, it logs a detailed message with the type of undefined behavior, source code location, and a backtrace.
- Please refer to the UBSAN demonstration code.

Kernel Debugger

Kernel GNU Debugger (KGDB)

- KGDB is a source-level debugger for the Linux kernel, used with GDB (GNU Debugger) to "break into" the kernel and analyze kernel behavior in real-time.
- Developers can place breakpoints in the kernel code and perform limited execution stepping to trace through kernel execution.
- KGDB runs within the kernel context and halts the kernel execution when a breakpoint is reached. It enables developers to inspect kernel memory, variables, and the call stack, similar to how GDB is used for debugging user-space programs.
- It requires another machine (host) to act as the debugger while the target machine (running the kernel) is being debugged.

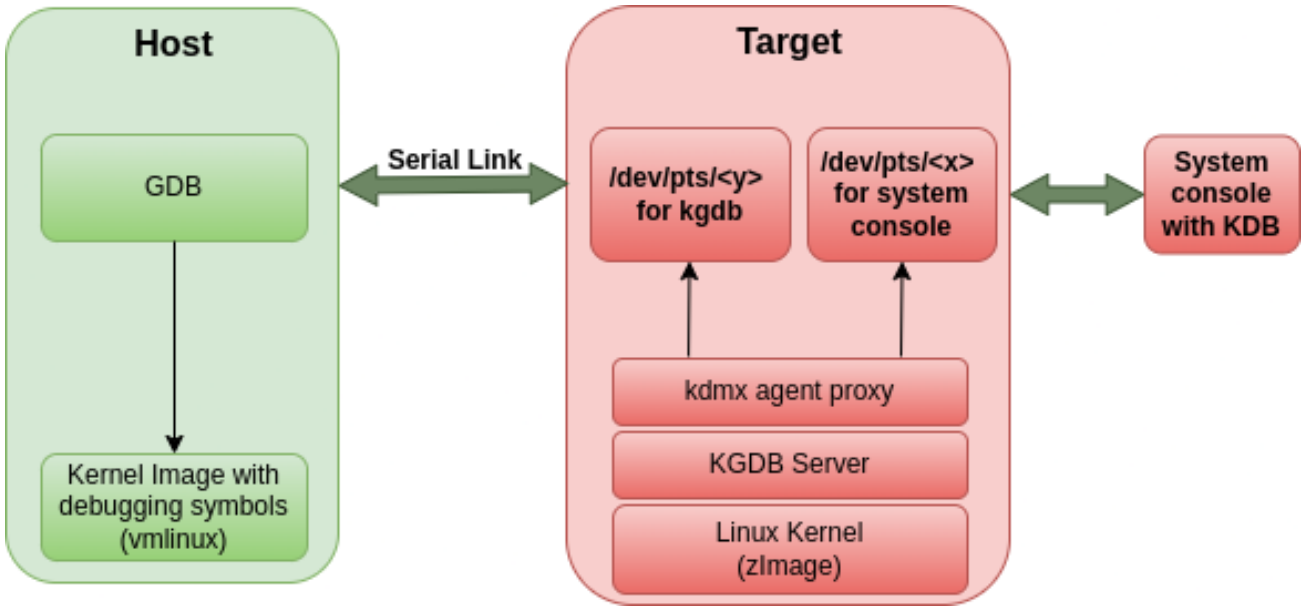
KGDB kernel config

- CONFIG_KGDB=y Enable KGDB support.
- CONFIG_DEBUG_KERNEL=y Enable Kernel debugging support.
- CONFIG_DEBUG_INFO=y Compile kernel with debugging symbols
- CONFIG_KGDB_SERIAL_CONSOLE=y Enable KGDB support over serial.
- CONFIG_KGDB_KDB=y Enable kdb frontend for kgdb.
- CONFIG_GDB_SCRIPTS=y Enable kernel GDB python scripts.
- CONFIG_MAGIC_SYSRQ=y Enable Magic SysRq support.
- CONFIG_PANIC_TIMEOUT=0 Disable reboot after panic
- CONFIG_STRICT_KERNEL_RWX=n Disable memory protection on the code section to allow setting breakpoints.
- CONFIG_FRAME_POINTER=y For reliable stacktraces.
- CONFIG_RANDOMIZE_BASE=n Disable KASLR.
- CONFIG_WATCHDOG=n Disable watchdog.

Kernel Debugger (KDB)

- KDB is a simpler, in-kernel debugger for performing basic debugging tasks without needing an external debugger like GDB.
- KDB offers basic debugging commands like memory examination, register checks, backtracing, and variable modification, but is more limited than KGDB, lacking source-level debugging and detailed inspection capabilities.
- It can be used directly on the target machine via a console (such as the kernel console or a serial console) and is useful for simple in-place debugging.
- KDB and KGDB can work together, allowing developers to switch between using KGDB for advanced debugging tasks and KDB for simpler tasks.
    - Use the `kgdb` command in KDB to enter kgdb mode.
    - Send a maintenance packet from gdb using `maintenance packet 3` to switch from kgdb to KDB mode.

KDMX (1/2)



KDMX (2/2)

- When a system has only a single serial port, it cannot use both KGDB and the console simultaneously because only one program can access the port at a time.
- KDMX is a user-space helper program that acts as a KGDB proxy/multiplexer.
- KDMX solves the issue of using both KGDB and the console simultaneously by splitting GDB packets and console traffic from the serial line into two separate pseudo-terminals (/dev/pts/x), allowing both KGDB and console output to function on a single serial port.

Setup and Configuration (1/4)

KDMX Configuration

- On the target side, ensure that any open connections to the serial port are closed. Then, run the following command to multiplex the serial port.

```
./kdmx -n -d -p/dev/ttyACM0 -b115200

serial port: /dev/ttyACM0
Initalizing the serial port to 115200 8n1
/dev/pts/4 is slave pty for terminal emulator
/dev/pts/5 is slave pty for gdb

Use <ctrl>C to terminate program
```

- Connect with target's serial console using `picocom -b 115200 /dev/pts/4`.
- Connect with GDB from host machine using `gdb <vmlinux File> -ex "set remotebaud 115200" -ex "target remote /dev/pts/5"`

Setup and Configuration (2/4)

- On the target device, enable kgdb over serial console `kgdboc` using CONFIG_KGDB_SERIAL_CONSOLE.
- Configure kgdboc at boot time by passing to the kernel:
  - kgdboc={tty-device},{bauds}.
  - For example: `console=ttyACM0,115200n8 oops=panic panic=0 kgdboc=ttyACM0`
- Or at runtime using sysfs:
  - echo ttyACM0 > /sys/module/kgdboc/parameters/kgdboc
  - If the console lacks polling support, an error will be generated when this command is executed.
- Add the `kgdbwait` parameter to the kernel to ensure it pauses and waits for a debugger connection.

Setup and Configuration (3/4)

Drop into the debugger

- Use Magic SysRq:
  - External keyboard: Alt-PrintScr-G
  - Command line shell: 'echo g > /proc/sysrq-trigger'
  - Send "BREAK-G" over serial port.
- Hardcode a breakpoint into your code: `kgdb_breakpoint()`.
- Cause an oops / panic.
- Create a custom debug trigger by inserting the `kgdb_breakpoint()` function into an IRQ handler which will force the kernel to break into the debugger when the interrupt is handled.

Setup and Configuration (4/4)

Connect with GDB

- On your host machine, start gdb as follows:
  `gdb {vmlinux File} -ex "set remotebaud 115200" -ex "target remote /dev/pts/5"`

KGDB scripts

- GDB provides a powerful Python scripting interface with helper scripts in the Linux kernel to simplify common debugging tasks.
- Enable CONFIG_GDB_SCRIPTS during kernel build and load the scripts into GDB via the `.gdbinit` file.

```
$ cat ~/.gdbinit
add-auto-load-safe-path /path/to/linux-build
<gdb> source vmlinux-gdb.py
```

- `apropos lx` List all GDB commands that contain "lx"
- `lx-lsmod` : Lists loaded kernel modules.
- `lx-symbols` : Automatically loads kernel symbols and module symbols.
- `lx-ps` : Displays the list of processes (`task_struct`) running in the kernel
- `lx-cpus` : Lists available CPUs.
- `$lx_task(pid)` : Fetches the `task_struct` for the Process ID (pid).
- `$lx_current()` : Returns the pointer to the current task in the kernel.

KGDB Demo

- Please refer to the KGDB demonstration code.

References

- https://elixir.bootlin.com/ : To navigate the kernel source code.
- Using Serial kdb / kgdb to Debug the Linux Kernel
- Raspberrypi-and-Buildroot
- Kernel Debugging How to debug the Linux kernel on a VM hosted on VirtualBox.
- Kexec and Kdump on Ubuntu 22.04
- Kexec and Kdump on Arch Linux

Part-2 Summary

- In module five we took a look at tracing in Linux. We covered userspace tools like strace, ltrace, uprobe and perf. We also looked at kernel space tools like Kprobe, Perf, ftrace, eBPF, and LTTng.
- In module six, we explored profiling in Linux looking at tools like massif, heaptrack and memusage to profile memory. We looked at data visualization tools like flame graphs and histograms. We investigated the use of callgrind, cachegrind and perf-stat for CPU and hardware profiling. We will also looked at stacktrace profiling using eBPF profile, gprof and sysprof.
- In the seventh and final module we took a detailed look at Kernel debugging. We reviewed the various debugging related Kernel configuration options. We investigated Kernel OOPS, reviewed logging and SysRq, and demonstrated the use of tools like KGDB. We showed Kernel recovery using Kexec and Kdump. And we also examined many of the new suite of sanitizer tools that have emerged in recent years like: UBSAN, KCSAN and KASAN.

Linux Debug Training

Part-2

Feedback: specialistlinuxtraining@gmail.com