Advanced Operating Systems (labs)

Vittorio Zaccaria | Politecnico di Milano | '24/25

System-wide tracing

- Sometimes, as a developer of system modules, you will need to understand the trace execution flow or profile of the entire system
- The kernel already includes a large number of tracepoints that can be recorded **as events** using specific tools.
- New tracepoints can also be created statically or dynamically using various mechanisms.

Tracepoints

Tracepoint: explicit kernel code line used to conditionally run a function.

- Created with `trace_<subsys>_eventname` ; essentially introduces a static variable `enable` and a conditional branch to a function `callback`

```
if(enable && callback) {
  (*callback)(....);
}
```

- `callback` can be set later through:
  - `trace_event` : captures relevant variables a formatted event into a global ring buffer
  - `register_trace_<subsys>_eventname(N)` : append a generic callback
- Example: `sched_switch`
  - `trace_sched_switch`
  - `trace_event(sched_switch)`
  - `register_trace_sched_switch` (`probe_sched_switch`)

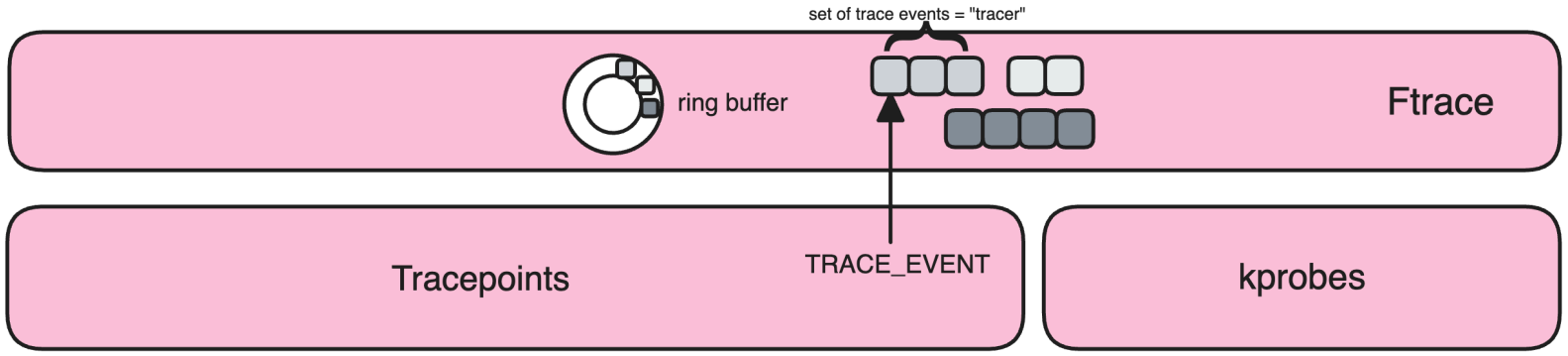
Kprobes

- Kernel build can even additional instrumentation at **function enter/exit** for you and dump the arguments
- Kernel must be configured with **CONFIG_KPROBES**=y
- **Does not use the tracepoint** infra, instead it uses code **patching** to modify `.text` code to insert calls to specific handlers

> You also find kretprobes which are pairs of `(entry, exit)` handlers attached to kernel functions

Ring buffer

- **Both kprobes and tracepoints** write into the "Ftrace" ring buffer
- Tracers enable/disable certain group of events (both tracepoints and kprobes).



> The ftrace ring buffer collects events from both probing mechanisms

Tracers

Available tracers:

- `nop` : Trace nothing, used to disable all tracing.
- `function` : Trace all kernel functions that are called.
- `function_graph` : Similar to function but traces both entry and exit.
- `hwlat` : Trace hardware latency.
- `irqsoff` : Trace sections where interrupts are disabled.
- `branch` : Trace likely()/unlikely() prediction errors.
- `mmiotrace` : Trace all accesses to the hardware (read[bwlq]/write[bwlq]).

Tracefs > tracepoints

```
# in the container
/repo/stage/start-qemu.sh --arch amd64

# in the aos-mini-linux vm

mount -t tracefs nodev /sys/kernel/tracing
cd /sys/kernel/tracing/
cat current_tracer          # should be "nop"
echo 1 > events/task/enable # enable tracepoints related to task creation
echo 1 > tracing_on         # start tracing
ls
echo 0 > tracing_on         # stop tracing
cat trace                   # show trace
```

Tracefs > kprobes

Inspecting the call to do_sys_open

```
# in the container
/repo/stage/start-qemu.sh --arch amd64

# in the aos-mini-linux vm
mount -t tracefs nodev /sys/kernel/tracing
cd /sys/kernel/tracing/

# must know where the parameters of the `open` are.
# Here we know path is in %si
echo "p:myprobe do_sys_open path=+u0(%si):string" > kprobe_events
echo 1 > events/kprobes/myprobe/enable
echo 1 > tracing_on; cat /init; echo 0 > tracing_on
echo 0 > events/kprobes/myprobe/enable
cat trace
```

Tracefs > function graphs

```
# in the container
/repo/stage/start-qemu.sh --arch amd64

# in the aos-mini-linux vm
echo function > current_tracer # takes a while
echo 1 > tracing_on; sleep 1; echo 0 > tracing_on
cat trace | head -50

echo function_graph > current_tracer # takes a while
echo 1 > tracing_on; sleep 1; echo 0 > tracing_on
cat trace | head -50
```

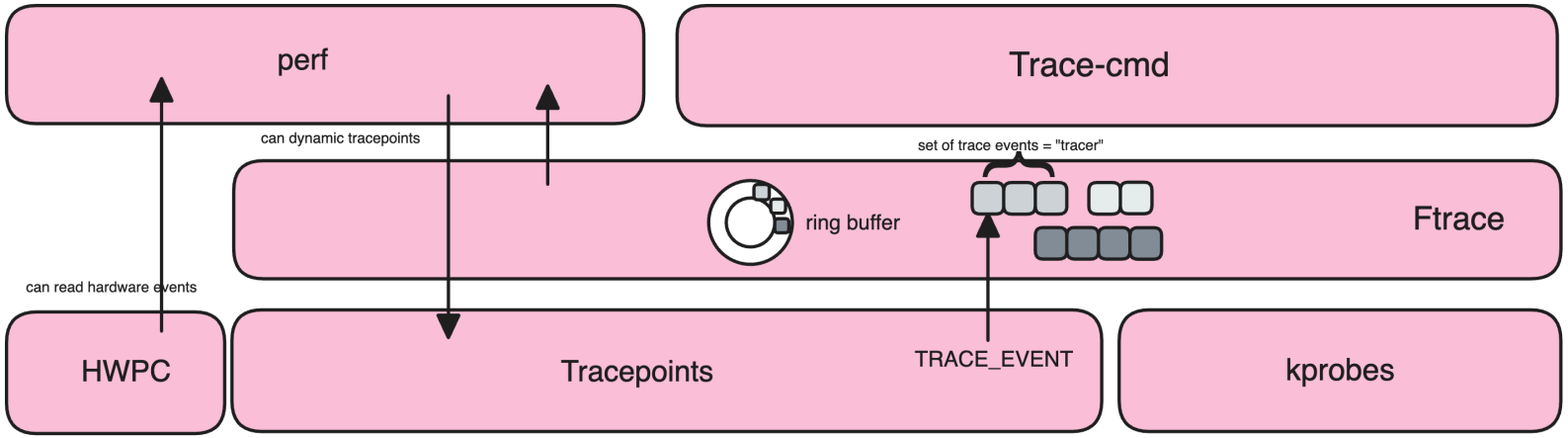
Tracefs > `trace_printk`

- Nostalgia for old printf debugging? You're set!
- `trace_printk()` allows to emit strings in the trace buffer

```
#include <linux/ftrace.h>
void read_hw() {
  if (condition)
    trace_printk("Condition is true!\n"); }
```

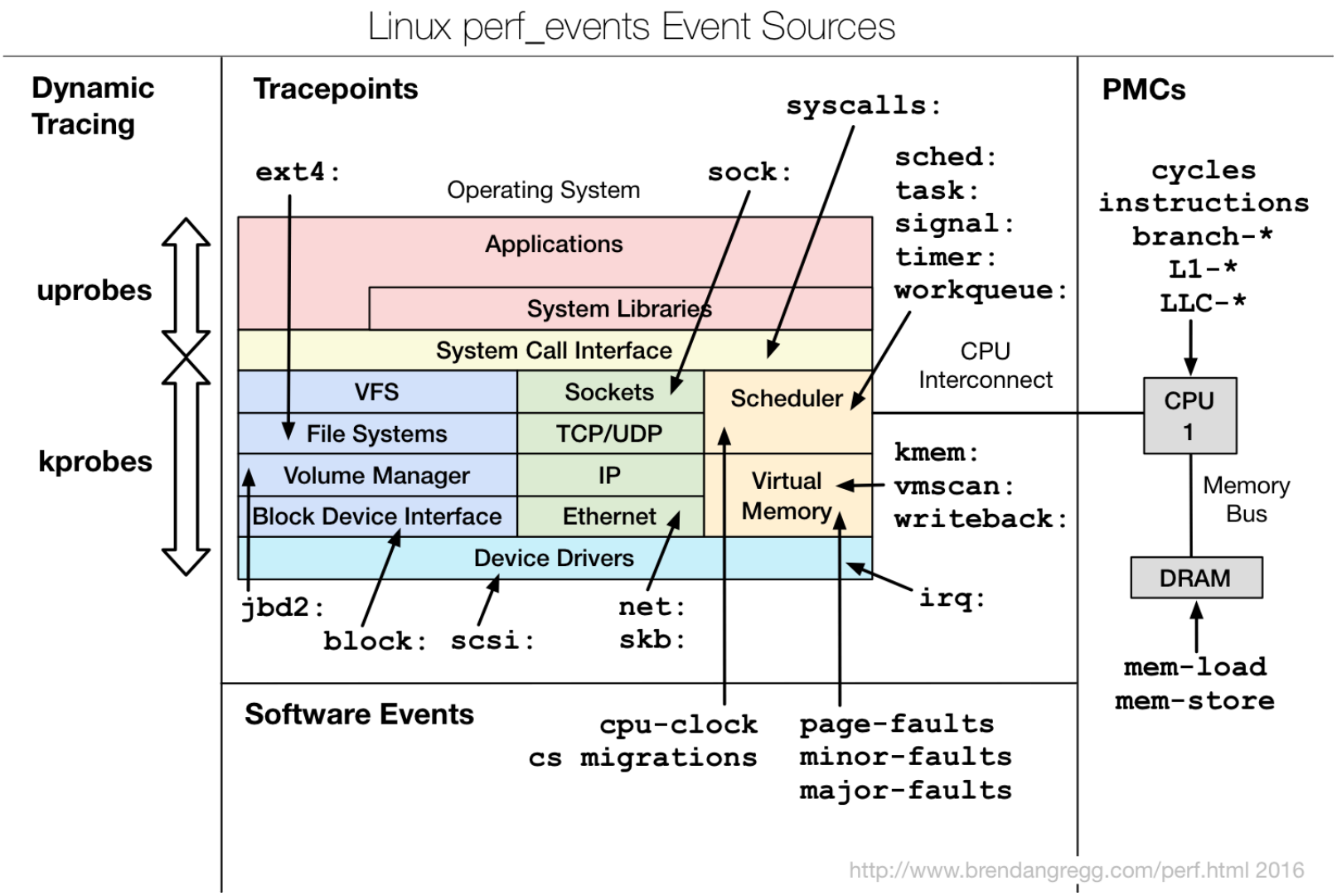
High-level tools

Perf



> Perf is one of the frontends to ftrace

Perf


Linux perf_events Event Sources
Dynamic Tracing
uprobes
kprobes
Tracepoints
ext4:
Operating System
sock:
syscalls:
sched:
task:
signal:
timer:
workqueue:
Applications
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Block Device Interface
Ethernet
Virtual Memory
Device Drivers
kmem:
vmscan:
writeback:
irq:
jbd2:
block:
scsi:
net:
skb:
CPU Interconnect
PMCs
cycles
instructions
branch-*
L1-*
LLC-*
CPU 1
Memory Bus
DRAM
mem-load
mem-store
Software Events
cpu-clock
cs
migrations
page-faults
minor-faults
major-faults
http://www.brendangregg.com/perf.html 2016

Perf > build

Be sure to run the container with --privileged

```
# this must be compiled only onnce
cd /sources/linux/tools/perf
LDFLAGS=-static NO_LIBPYTHON=1 make
cp perf /staging/initramfs/fs/bin
cp perf /repo/stage
/sources/bootstrap.sh

/repo/stage/start-qemu.sh --arch amd64
```

Perf > list/record/script/report

Note, you must enable BPF_SYSCALL on kernel build

```
/repo/stage/start-qemu.sh --arch amd64

# in the aos-mini-linux vm

perf list syscalls:*
perf list # all events traceable

perf record -e syscalls:sys_enter_read sha256sum /bin/busybox

# List all events from perf.data
perf script
```

Perf > record > script

Record and print individual specific events, e.g., switching from one process to the next

```
perf record -e sched:sched_switch -e sched:sched_migrate_task -a ls
perf script # see switching from one process to the next
# ...
#                                                  goes to sleep ------+
#                                                                      ↓
# perf    74 [000]    32.116587:    ...prev_comm=perf prev_pid=74 prev_state=S ==
```

Perf > record > report

Print events' cumulative count

```
perf report
# Samples: 68  of event 'sched:sched_switch'
# Event count (approx.): 68
#
# Overhead  Command      Shared Object      Symbol
# ........  ............  ..................  ................
#            + helper thread for RCU
#            ↓
#    36.76%  rcu_preempt  [kernel.kallsyms]  [k] __schedule
#    25.00%  perf         [kernel.kallsyms]  [k] __schedule
#    25.00%  perf-exec    [kernel.kallsyms]  [k] __schedule
#    10.29%  ls           [kernel.kallsyms]  [k] __schedule
#     1.47%  kcompactd0   [kernel.kallsyms]  [k] __schedule
#     1.47%  ksoftirqd/0  [kernel.kallsyms]  [k] __schedule
```

Record tracing data

```
# Sample context-switches with stack traces, for two secnds:
perf record -e context-switches -ag -- sleep 2

# Sample page faults with stack traces for two secnds:
perf record -e page-faults -ag -- sleep 2
```

Perf > record > flamegraphs

Flamegraphs (this is going to work in lab 5, don't look at it now)

```
# install the serialaos driver (see lab 5)

perf record -F 99 -ag -- sleep 2
perf script > perf.txt
cat perf.txt > /dev/serialaos

# On the host
nc localhost 8080 > perf.txt # wait a bit
# Use [speedscope](https://www.speedscope.app/)
```

Perf > record > fixed frequency

Record profiling data with a given frequency

```
# Sample CPU stack traces (via frame pointers) for the specified PID, at 99 Hertz, for 10 seconds:
#                            ↓ stack traces
perf record -F 99 -p _PID_ -g -- sleep 10

# Sample CPU stack traces for the entire system, at 99 Hertz, for 10 seconds (< Linux 4.11):
perf record -F 99 -ag -- sleep 10

# Sample stack traces corresponding to block:block_rq_issue
perf record -e block:block_rq_issue -ag
```

Perf > stat

Just count events and access hardware performance counters

```
perf stat ls
# Performance counter stats for 'ls':
#
#             49.45 msec task-clock                #    0.534 CPUs utilized
#                 7      context-switches          #  141.549 /sec
#                 0      cpu-migrations            #    0.000 /sec
#                44      page-faults               #  889.734 /sec
#   <not supported>      cycles
#   <not supported>      instructions
#   <not supported>      branches
#   <not supported>      branch-misses
```

Perf > trace

Trace events as they happen

```
perf trace -e "syscalls:*" ls
#   0.000 ls/108 syscalls:sys_exit_execve(__syscall_nr: 59)
#   165.014 ls/108 syscalls:sys_enter_arch_prctl(option: 0x3001, arg2: 0x7fff5d462bf0)
#   167.742 ls/108 syscalls:sys_exit_arch_prctl(__syscall_nr: 158, ret: 0xffffffea)
#   294.278 ls/108 syscalls:sys_enter_brk(option: 0x3001, arg2: 0x7fff5d462bf0)
#   371.028 ls/108 syscalls:sys_exit_brk(__syscall_nr: 12, ret: 0x1346000)
#   431.930 ls/108 syscalls:sys_enter_brk(brk: 0x1347200)
#   498.830 ls/108 syscalls:sys_exit_brk(__syscall_nr: 12, ret: 0x1347200)
```

Perf > top

- perf top allows to do a live analysis of the running kernel
- It will sample all function calls and display them ordered by most time consuming one.
- This allows to profile the whole system usage

```
# PerfTop:    2159 irqs/sec  kernel:62.3%  exact:  0.0% lost: 0/0 drop: 0/18251 [4000Hz cpu-clock:ppp],  (all, 1 CPU)
# ------------------------------------------------------------------------------
#
#     51.76%  [kernel]       [k] __cpuidle_text_start
#      4.33%  perf           [.] hists__findnew_entry
#      3.70%  perf           [.] __strcmp_sse2
#      2.15%  perf           [.] evsel__parse_sample
#      1.82%  [kernel]       [k] _raw_spin_unlock_irqrestore
#      1.80%  perf           [.] perf_hpp__is_dynamic_entry
#      1.66%  perf           [.] sort__sym_cmp
#      1.65%  perf           [.] dso__find_symbol
#      1.24%  perf           [.] _int_free
#      1.16%  perf           [.] down_read
#      1.13%  perf           [.] sort__dso_cmp
#      1.12%  perf           [.] __hists__add_entry.constprop.0
#      1.10%  perf           [.] deliver_event
#      0.92%  perf           [.] _init
#      0.83%  perf           [.] hpp__nop_cmp
#      0.82%  perf           [.] hist_entry_iter__add
#      0.81%  perf           [.] __sort__hpp_cmp
#      0.80%  perf           [.] up_read
#      0.78%  [kernel]       [k] finish_task_switch
#      0.74%  perf           [.] down_write
```

Perf > top

```
# Sample CPUs at 49 Hertz
perf top -F 49

# Sample CPUs, show top process names and segments
perf top -ns comm,dso

# Count system calls by process
perf top -e raw_syscalls:sys_enter -ns comm -d 1
```

Perf > probe

Adds new tracepoints

```
# Add a tracepoint for the kernel tcp_sendmsg() function entry ("--add" is optional):
perf probe --add tcp_sendmsg
# Remove the tcp_sendmsg() tracepoint (or use "--del"):
perf probe -d tcp_sendmsg
# Add a tracepoint for the kernel tcp_sendmsg() function return:
perf probe 'tcp_sendmsg%return'
# Show available variables for the kernel tcp_sendmsg() function (needs debuginfo):
perf probe -V tcp_sendmsg
# Show available variables for the kernel tcp_sendmsg() function, plus external vars (needs debuginfo):
perf probe -V tcp_sendmsg --externs
# Show available line probes for tcp_sendmsg() (needs debuginfo):
perf probe -L tcp_sendmsg
# Show available variables for tcp_sendmsg() at line number 81 (needs debuginfo):
perf probe -V tcp_sendmsg:81
# Add a tracepoint for tcp_sendmsg(), with three entry argument registers (platform specific):
perf probe 'tcp_sendmsg %ax %dx %cx'
# Add a tracepoint for tcp_sendmsg(), with an alias ("bytes") for the %cx register (platform specific):
perf probe 'tcp_sendmsg bytes=%cx'
# Trace previously created probe when the bytes (alias) variable is greater than 100:
perf record -e probe:tcp_sendmsg --filter 'bytes > 100'
```

Linkography

- bootlin.com/doc/training/debugging/debugging-slides.pdf
- Linux perf Examples