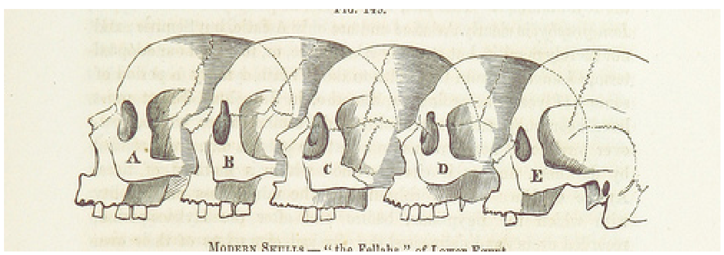


Viller Hsiao <villerhsiao@gmail.com>

Aug. 21, 2016

COSCUP 2016 – Linux Kernel Tracing

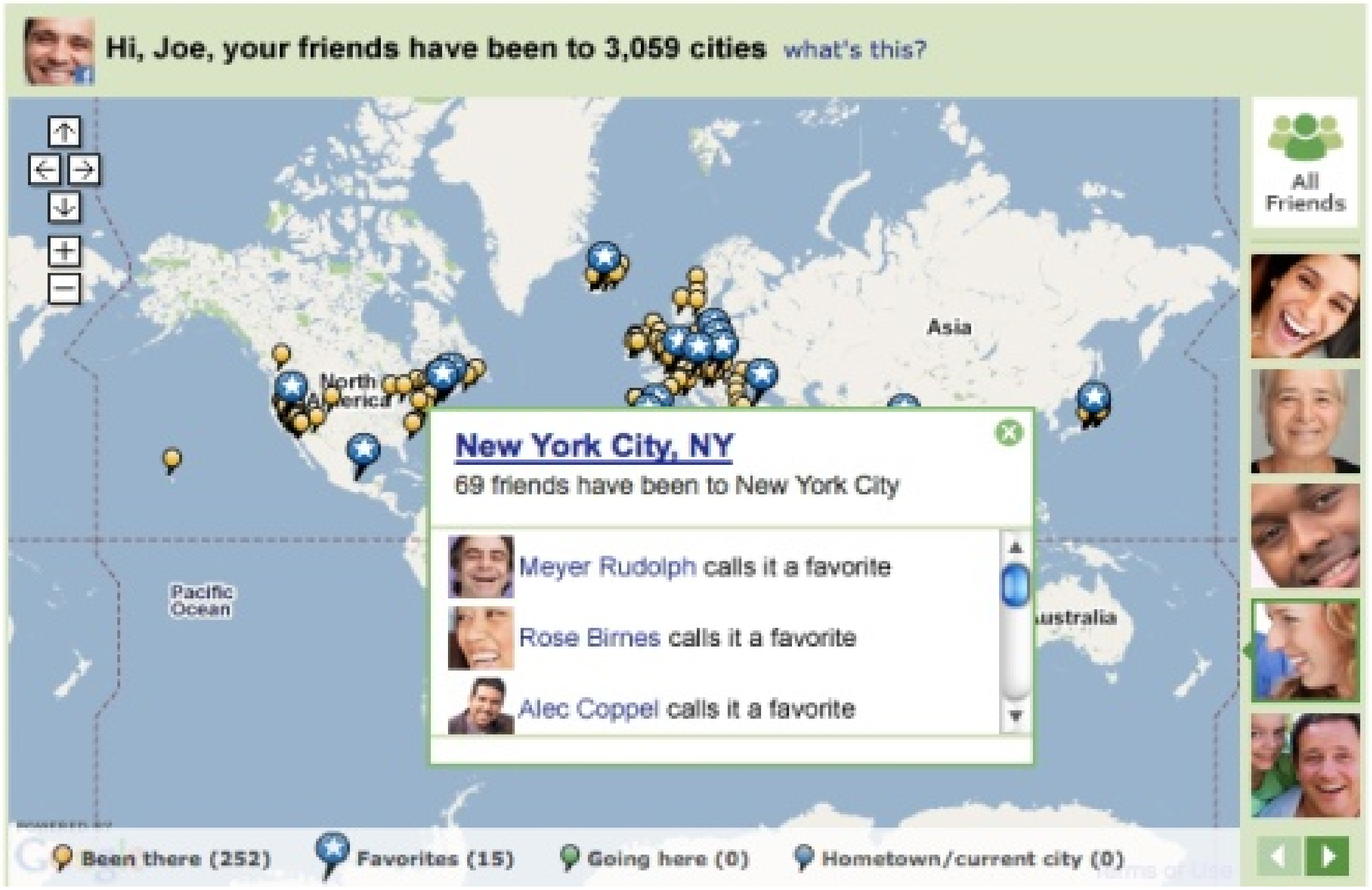
Who am I ?

Viller Hsiao

Embedded Linux / RTOS engineer

http://image.dfdaily.com/2012/5/4/634716931128751250504b050c1_nEO_IMG.jpg

What's Tracing



New York City, NY

69 friends have been to New York City

Meyer Rudolph calls it a favorite

Rose Birnes calls it a favorite

Alec Coppel calls it a favorite

https://www.tnooz.com/wp-content/uploads/2010/12/tripadvisor-facebook-rampup1.jpg

What's Tracing

- Famous way in C: printf()

```
void myfunc(int type)
{
        if (type > 20) {
             /* do some things */
             printf (“I like it goes here!\n”);

        } else if (type < 100) {
            /* do other things */
             printf (“But it goes here!\n”);

       } else {
            /* error handling */
             printf (“Oh! I hate it's here! Wrong type is %d\n”, type);
       }
}
```

What's tracing data used for?

Observe program behavior

What's tracing data used for?

Observe program behavior
Debug program

What's tracing data used for?

Observe program behavior
Debug program
Profile and get statistics
and so on

Well-known tool in kernel: printk()

printk() is intuitive, but

Issue of printk()

High overhead

"using printk(), especially when writing to the serial console, may take several milliseconds per write." ~ [1]

Issue of printk()

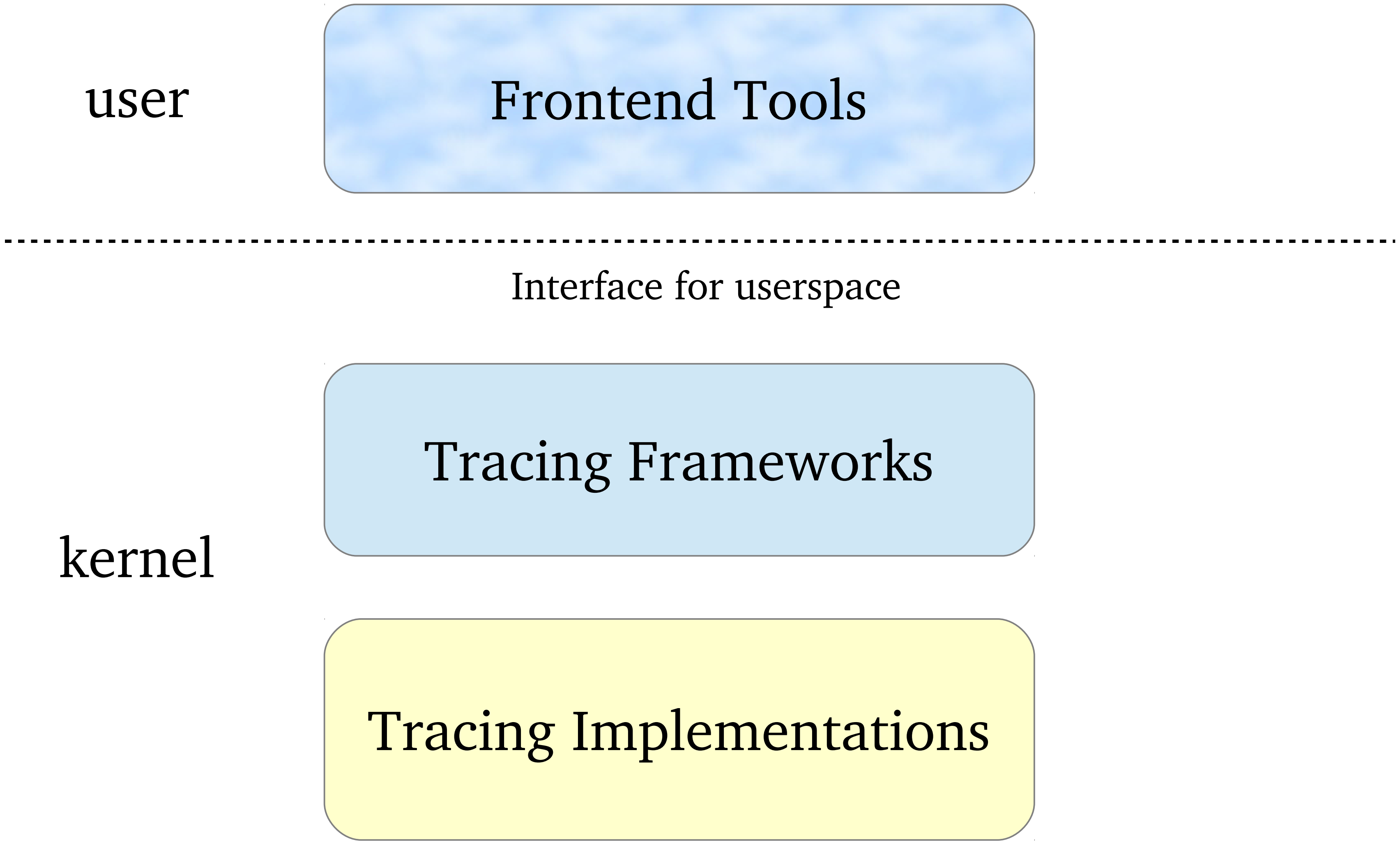
High overhead
Lack of flexibility

Topic today

Systematic tracing mechanisms in Linux kernel

How kernel exhausts compiler and CPU tricks to implement flexible and low overhead system tracing

Tracing in Linux

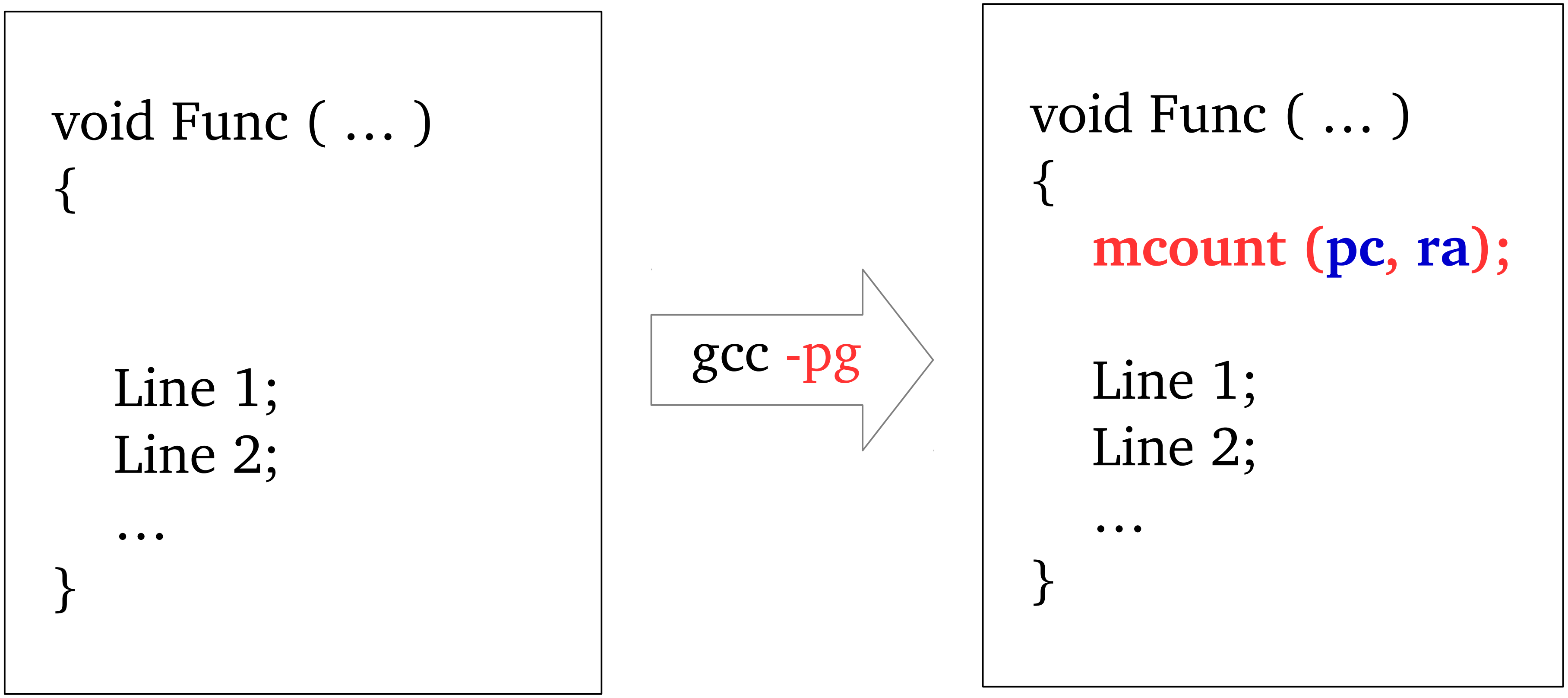


ftrace

ftrace

- Linux-2.6.27
- Linux kernel internal tracer framework
  - Function tracer
  - Tracing data output
  - Tracepoint
  - hist triggers

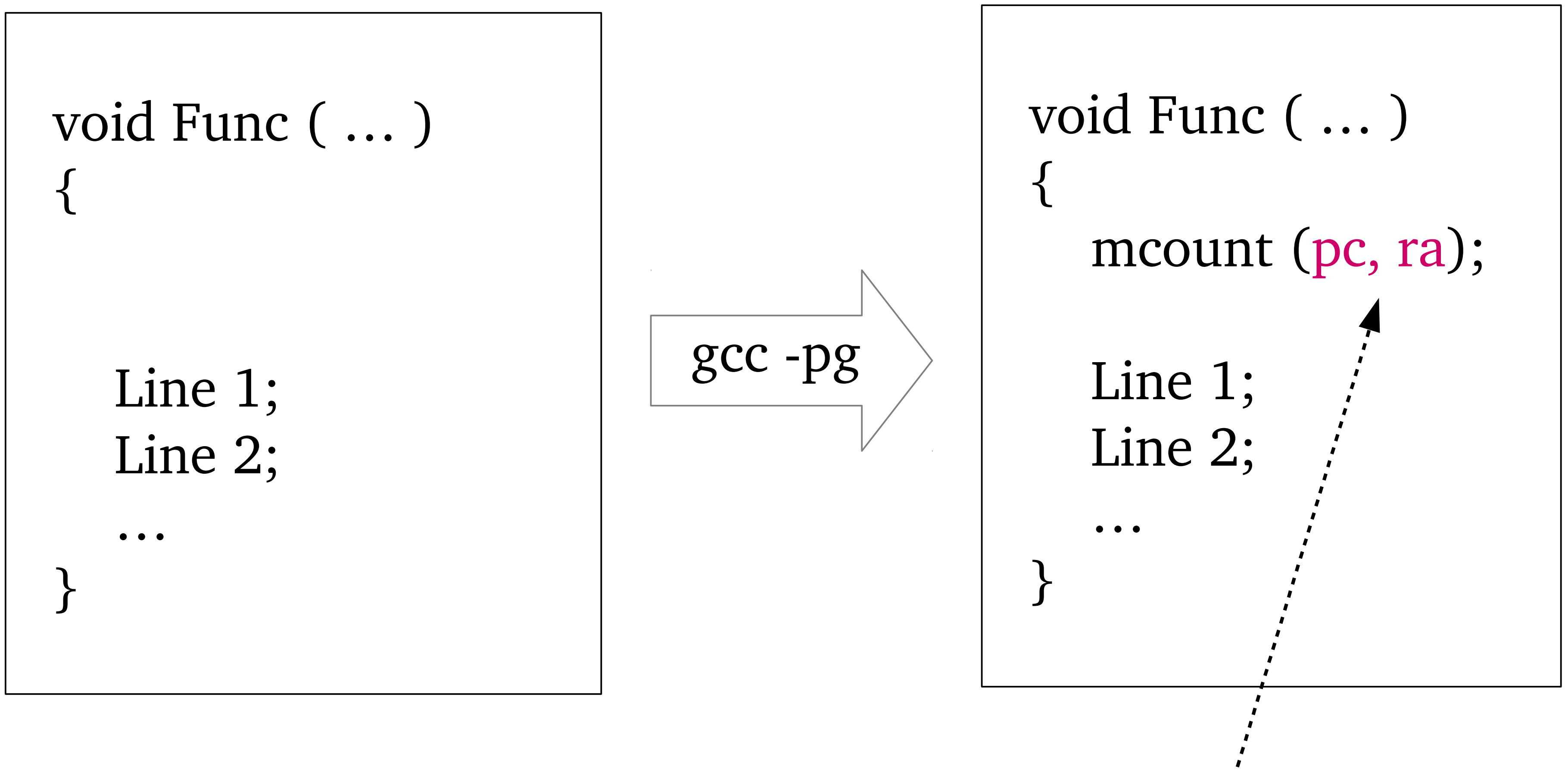
Function Tracer


```
void Func ( … )
{


    Line 1;
    Line 2;
    …
}
```

```
void Func ( … )
{
    mcount (pc, ra);

    Line 1;
    Line 2;
    …
}
```

Re-use gprof mechanism, then re-implement mcount()

Function Tracer

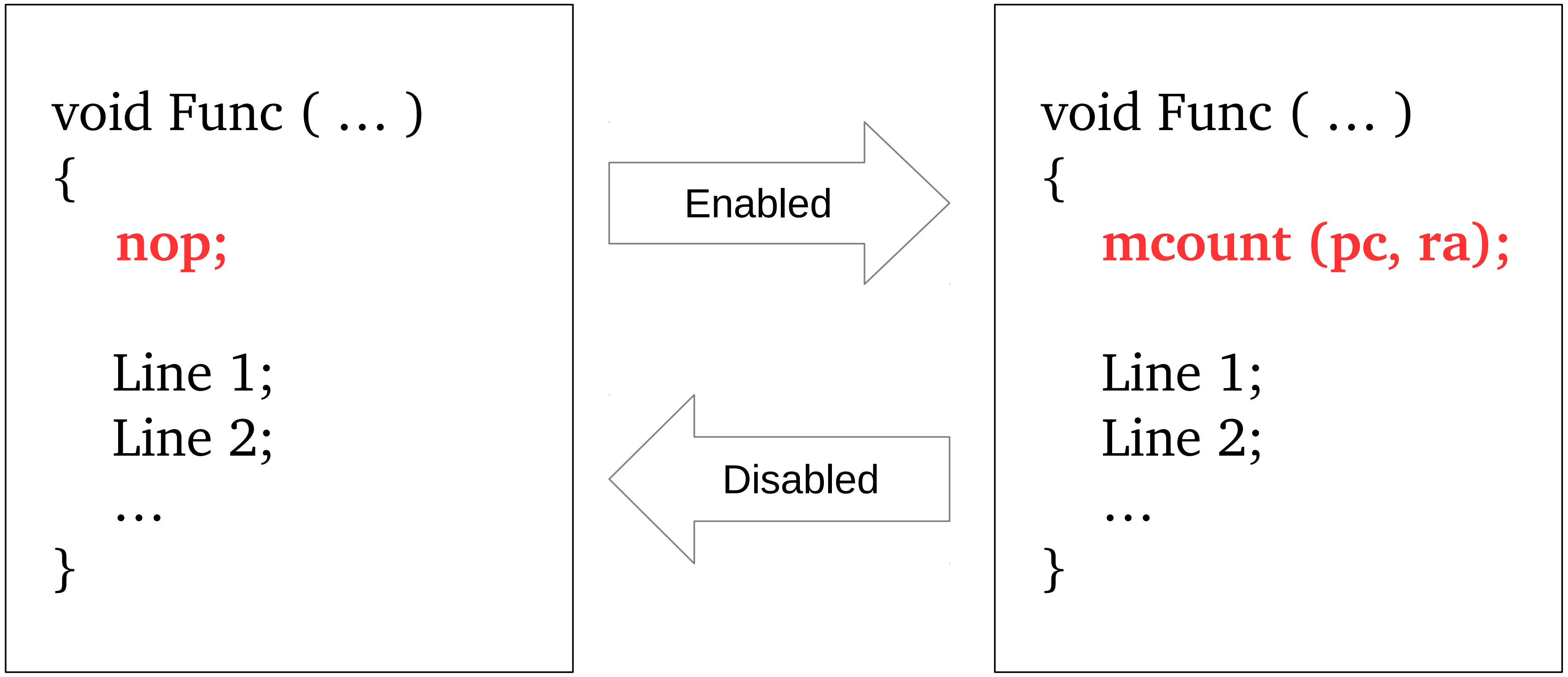


```
void Func ( … )
{


    Line 1;
    Line 2;
    …
}
```

```
void Func ( … )
{
    mcount (pc, ra);

    Line 1;
    Line 2;
    …
}
```

Data recorded: function and its caller

Dynamic Function Tracer



```
void Func ( … )
{
    nop;

    Line 1;
    Line 2;
    …
}
```

```
void Func ( … )
{
    mcount (pc, ra);

    Line 1;
    Line 2;
    …
}
```

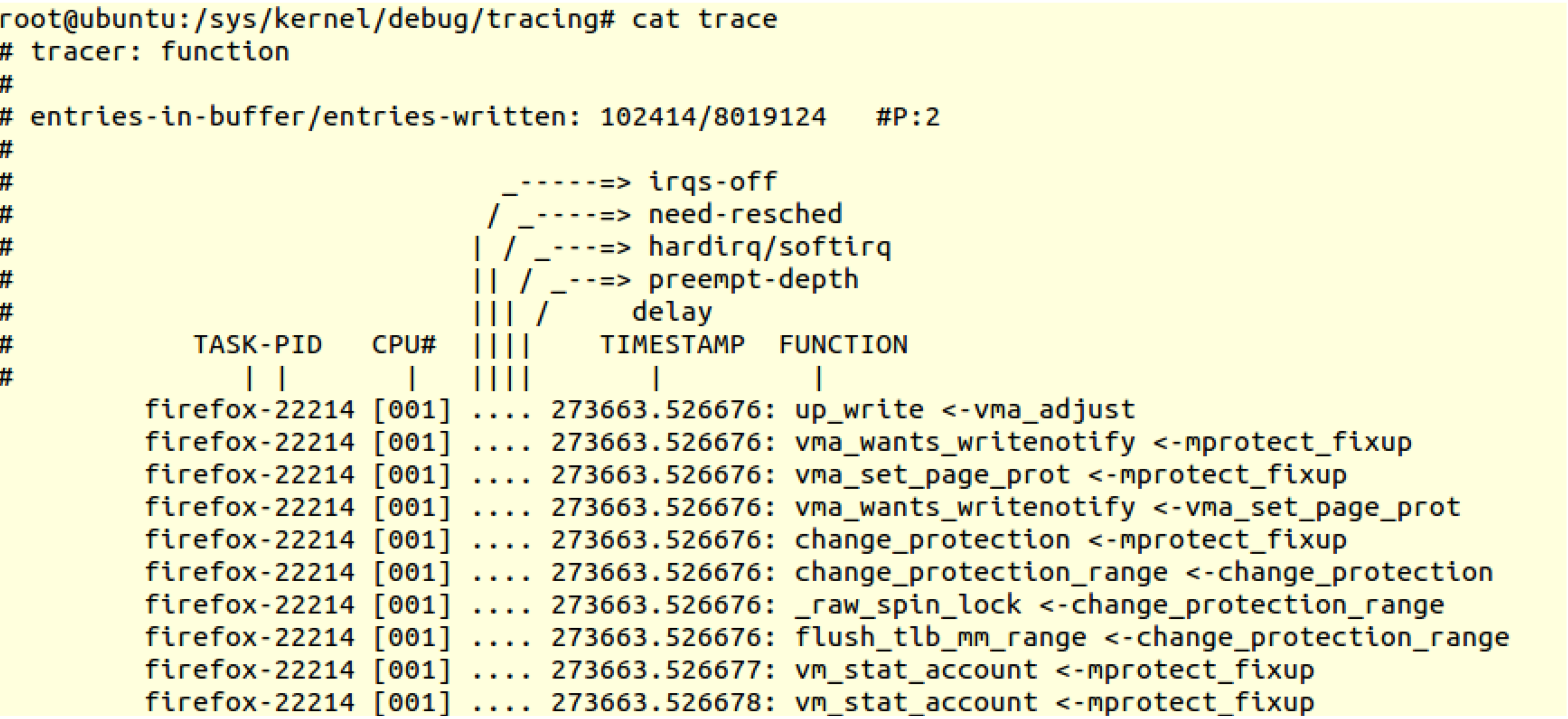
Tracing Data Output

- trace_printk()

  “Writing into the ring buffer with trace_printk() only takes around a tenth of a microsecond or so” ~ [1]

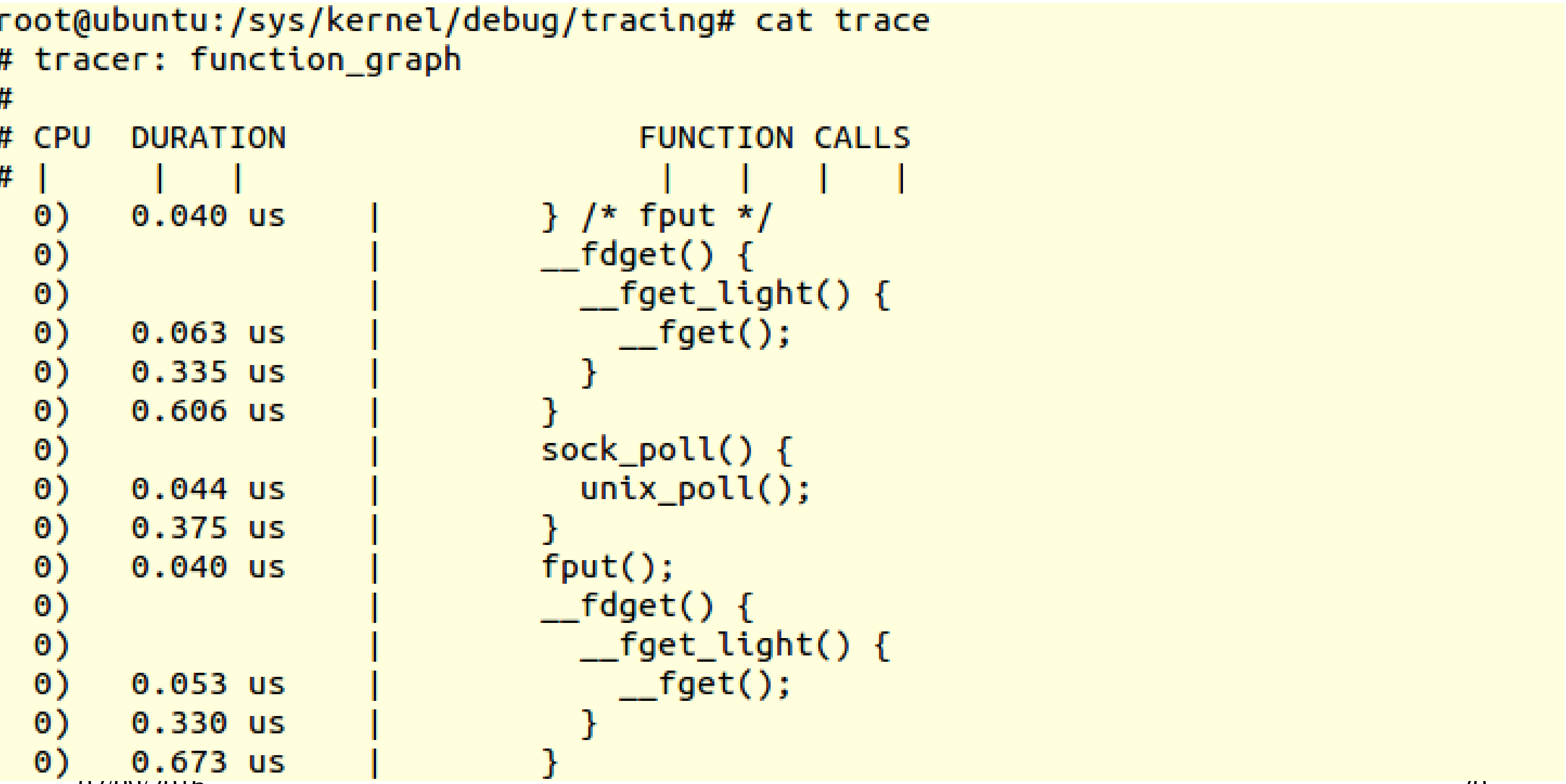
- /sys/kernel/debug/tracing/
  - tracefs (debugfs in the beginning)

Example: Function Tracer



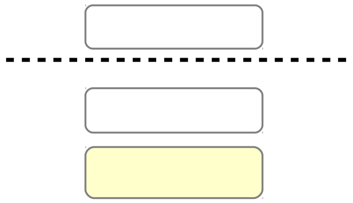
```
root@ubuntu:/sys/kernel/debug/tracing# cat trace
# tracer: function
#
# entries-in-buffer/entries-written: 102414/8019124   #P:2
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
         firefox-22214 [001] .... 273663.526676: up_write <-vma_adjust
         firefox-22214 [001] .... 273663.526676: vma_wants_writenotify <-mprotect_fixup
         firefox-22214 [001] .... 273663.526676: vma_set_page_prot <-mprotect_fixup
         firefox-22214 [001] .... 273663.526676: vma_wants_writenotify <-vma_set_page_prot
         firefox-22214 [001] .... 273663.526676: change_protection <-mprotect_fixup
         firefox-22214 [001] .... 273663.526676: change_protection_range <-change_protection
         firefox-22214 [001] .... 273663.526676: _raw_spin_lock <-change_protection_range
         firefox-22214 [001] .... 273663.526676: flush_tlb_mm_range <-change_protection_range
         firefox-22214 [001] .... 273663.526677: vm_stat_account <-mprotect_fixup
         firefox-22214 [001] .... 273663.526678: vm_stat_account <-mprotect_fixup
```

Example: Function Graph Tracer



```
root@ubuntu:/sys/kernel/debug/tracing# cat trace
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
  0)   0.040 us    |          } /* fput */
  0)               |          __fdget() {
  0)               |            __fget_light() {
  0)   0.063 us    |              __fget();
  0)   0.335 us    |            }
  0)   0.606 us    |          }
  0)               |          sock_poll() {
  0)   0.044 us    |            unix_poll();
  0)   0.375 us    |          }
  0)   0.040 us    |          fput();
  0)               |          __fdget() {
  0)               |            __fget_light() {
  0)   0.053 us    |              __fget();
  0)   0.330 us    |            }
  0)   0.673 us    |          }
```

Tracepoint



Tracepoint

- Linux-2.6.32
- Define and insert hook in static point like printk()

Tracepoint – Declare Event

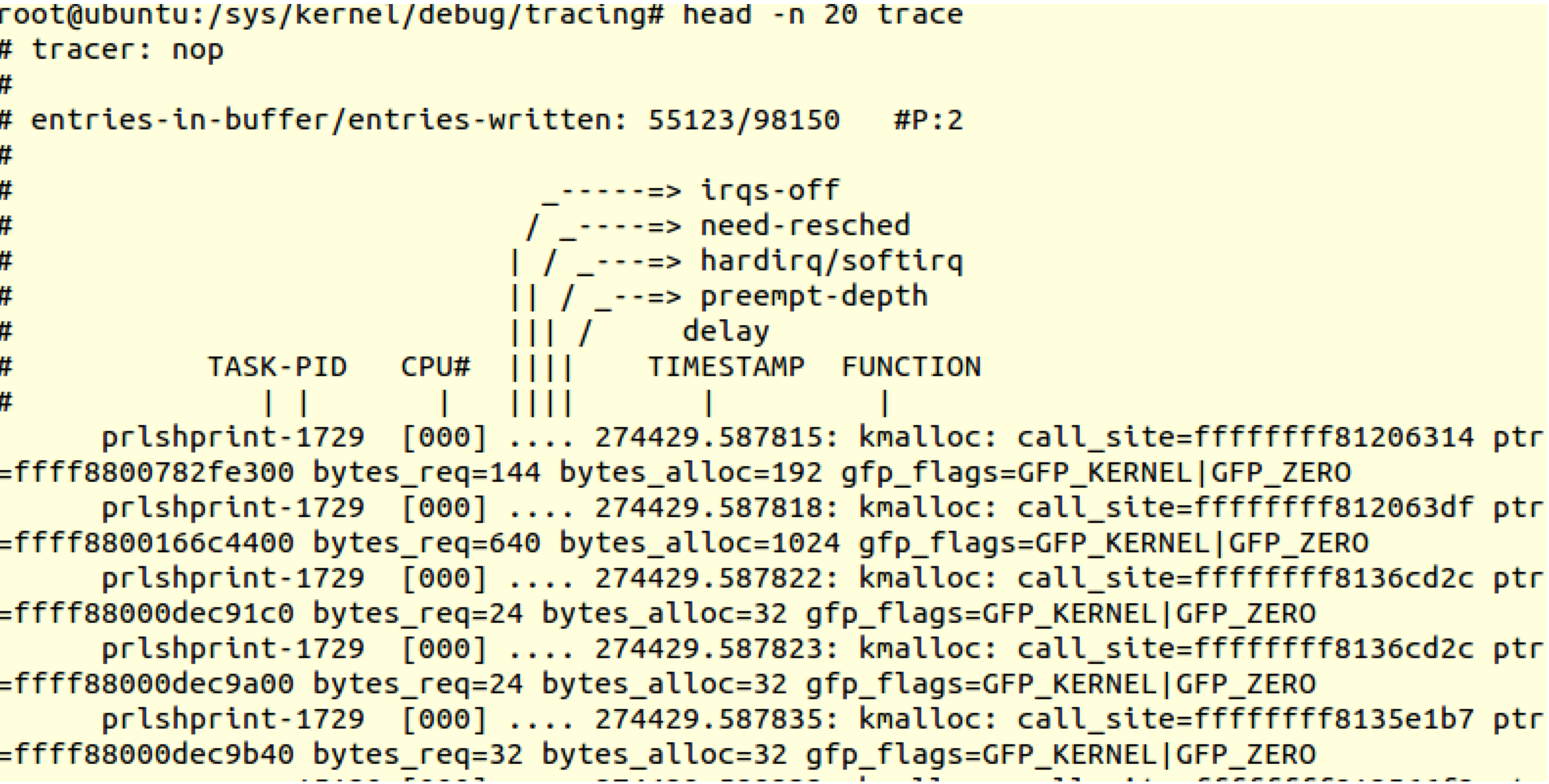
```
#include <linux/tracepoint.h>

TRACE_EVENT(mm_page_allocation,

    TP_PROTO(unsigned long pfn, unsigned long free),

    TP_ARGS(pfn, free),

    TP_STRUCT__entry(
            __field(unsigned long, pfn)
            __field(unsigned long, free)
    ),

    TP_fast_assign(
            __entry->pfn = pfn;
            __entry->free = free;
    ),

    TP_printk("pfn=%lx zone_free=%ld", __entry->pfn, __entry->free)
    );
```

Tracepoint – Probe Event

```
. . .

 trace_mm_page_allocation(page_to_pfn(page),
                zone_page_state(zone, NR_FREE_PAGES));

 . . .
```

Data recorded: custom defined data

Example: Tracepoint

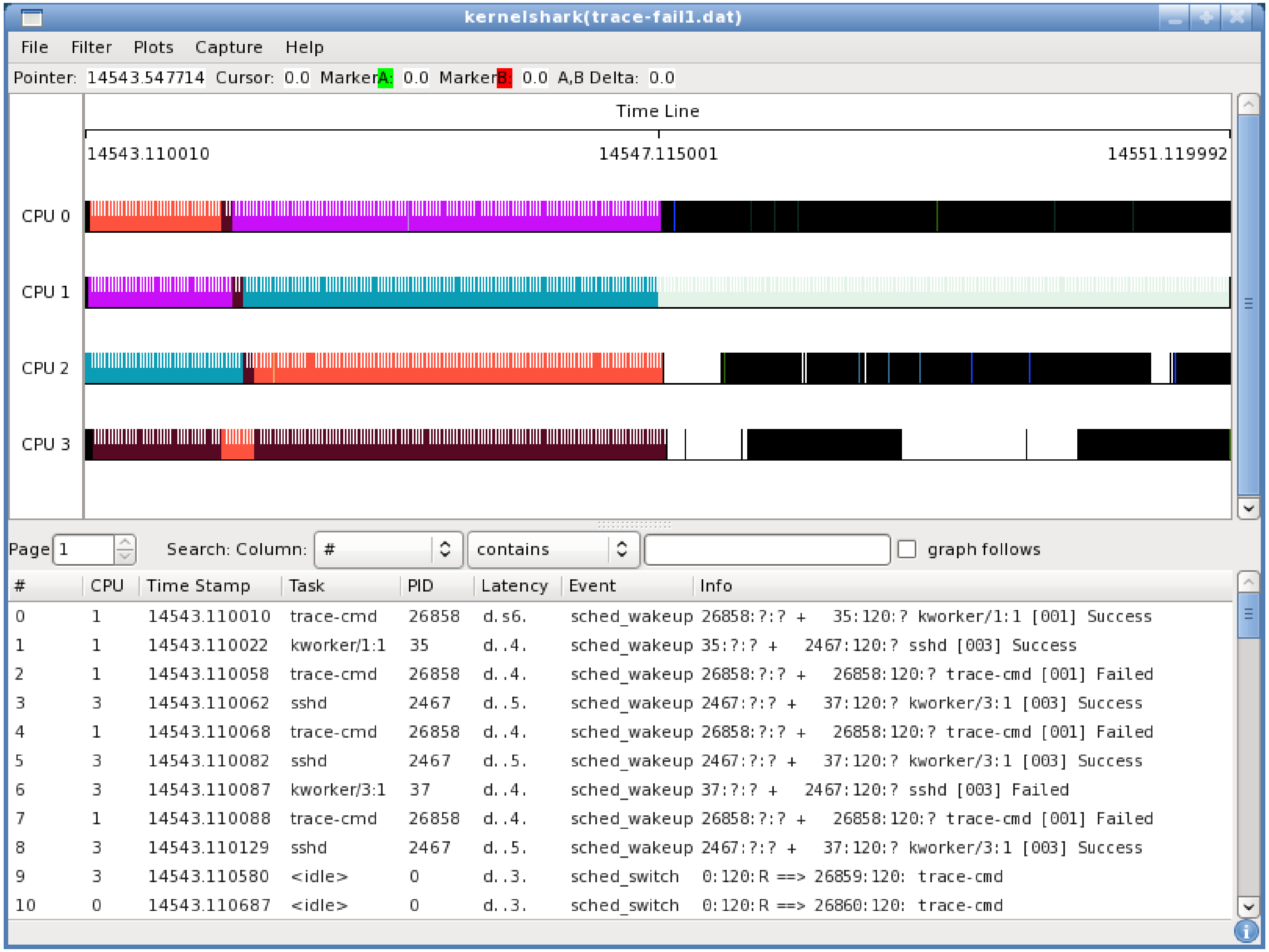


```
root@ubuntu:/sys/kernel/debug/tracing# head -n 20 trace
# tracer: nop
#
# entries-in-buffer/entries-written: 55123/98150   #P:2
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
      prlshprint-1729  [000] .... 274429.587815: kmalloc: call_site=ffffffff81206314 ptr
=ffff8800782fe300 bytes_req=144 bytes_alloc=192 gfp_flags=GFP_KERNEL|GFP_ZERO
      prlshprint-1729  [000] .... 274429.587818: kmalloc: call_site=ffffffff812063df ptr
=ffff8800166c4400 bytes_req=640 bytes_alloc=1024 gfp_flags=GFP_KERNEL|GFP_ZERO
      prlshprint-1729  [000] .... 274429.587822: kmalloc: call_site=ffffffff8136cd2c ptr
=ffff88000dec91c0 bytes_req=24 bytes_alloc=32 gfp_flags=GFP_KERNEL|GFP_ZERO
      prlshprint-1729  [000] .... 274429.587823: kmalloc: call_site=ffffffff8136cd2c ptr
=ffff88000dec9a00 bytes_req=24 bytes_alloc=32 gfp_flags=GFP_KERNEL|GFP_ZERO
      prlshprint-1729  [000] .... 274429.587835: kmalloc: call_site=ffffffff8135e1b7 ptr
=ffff88000dec9b40 bytes_req=32 bytes_alloc=32 gfp_flags=GFP_KERNEL|GFP_ZERO
```

trace-cmd

```
# trace-cmd record -e 'sched_wakeup*' -e sched_switch your-application

# kernelshark
```

Kernelshark



Pointer: 14543.547714 Cursor: 0.0 MarkerA: 0.0 MarkerB: 0.0 A,B Delta: 0.0

14543.110010 14547.115001 14551.119992

Page 1 Search: Column: # contains

#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	1	14543.110010	trace-cmd	26858	d.s6.	sched_wakeup	26858:?:? + 35:120:? kworker/1:1 [001] Success
1	1	14543.110022	kworker/1:1	35	d..4.	sched_wakeup	35:?:? + 2467:120:? sshd [003] Success
2	1	14543.110058	trace-cmd	26858	d..4.	sched_wakeup	26858:?:? + 26858:120:? trace-cmd [001] Failed
3	3	14543.110062	sshd	2467	d..5.	sched_wakeup	2467:?:? + 37:120:? kworker/3:1 [003] Success
4	1	14543.110068	trace-cmd	26858	d..4.	sched_wakeup	26858:?:? + 26858:120:? trace-cmd [001] Failed
5	3	14543.110082	sshd	2467	d..5.	sched_wakeup	2467:?:? + 37:120:? kworker/3:1 [003] Success
6	3	14543.110087	kworker/3:1	37	d..4.	sched_wakeup	37:?:? + 2467:120:? sshd [003] Failed
7	1	14543.110088	trace-cmd	26858	d..4.	sched_wakeup	26858:?:? + 26858:120:? trace-cmd [001] Failed
8	3	14543.110129	sshd	2467	d..5.	sched_wakeup	2467:?:? + 37:120:? kworker/3:1 [003] Success
9	3	14543.110580	<idle>	0	d..3.	sched_switch	0:120:R ==> 26859:120: trace-cmd
10	0	14543.110687	<idle>	0	d..3.	sched_switch	0:120:R ==> 26860:120: trace-cmd

https://static.lwn.net/images/2011/ks-fail1-open.png

hist triggers

- Introduced in Linux-4.7
- Create custom, efficient, in-kernel histograms

```
# echo 'hist:key=common_pid.execname:values=ret:sort=ret if ret >= 0' \
    > /sys/kernel/tracing/events/syscalls/sys_exit_read/trigger
```

Example hist triggers Logs

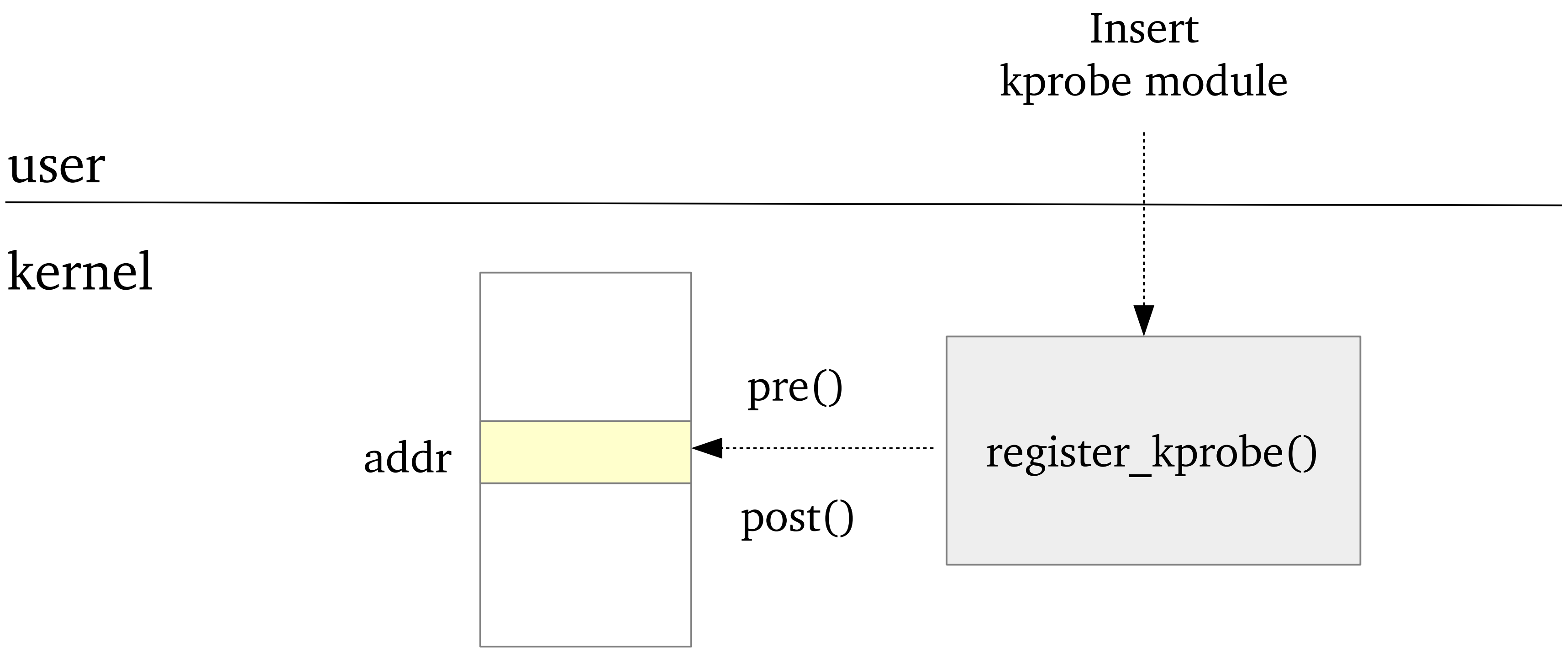
```
# cat /sys/kernel/tracing/events/syscalls/sys_exit_read/hist
[...]
{ common_pid: bash          [   16608] } hitcount:      4  ret:      11722
{ common_pid: bash          [   16616] } hitcount:      4  ret:      12386
{ common_pid: bash          [   16617] } hitcount:      4  ret:      12469
{ common_pid: irqbalance    [    1189] } hitcount:     36  ret:      21702
{ common_pid: snmpd         [    1617] } hitcount:     75  ret:      22078
{ common_pid: sshd          [   32745] } hitcount:    329  ret:     165710
[...]
```

http://www.brendangregg.com/blog/2016-06-08/linux-hist-triggers.html

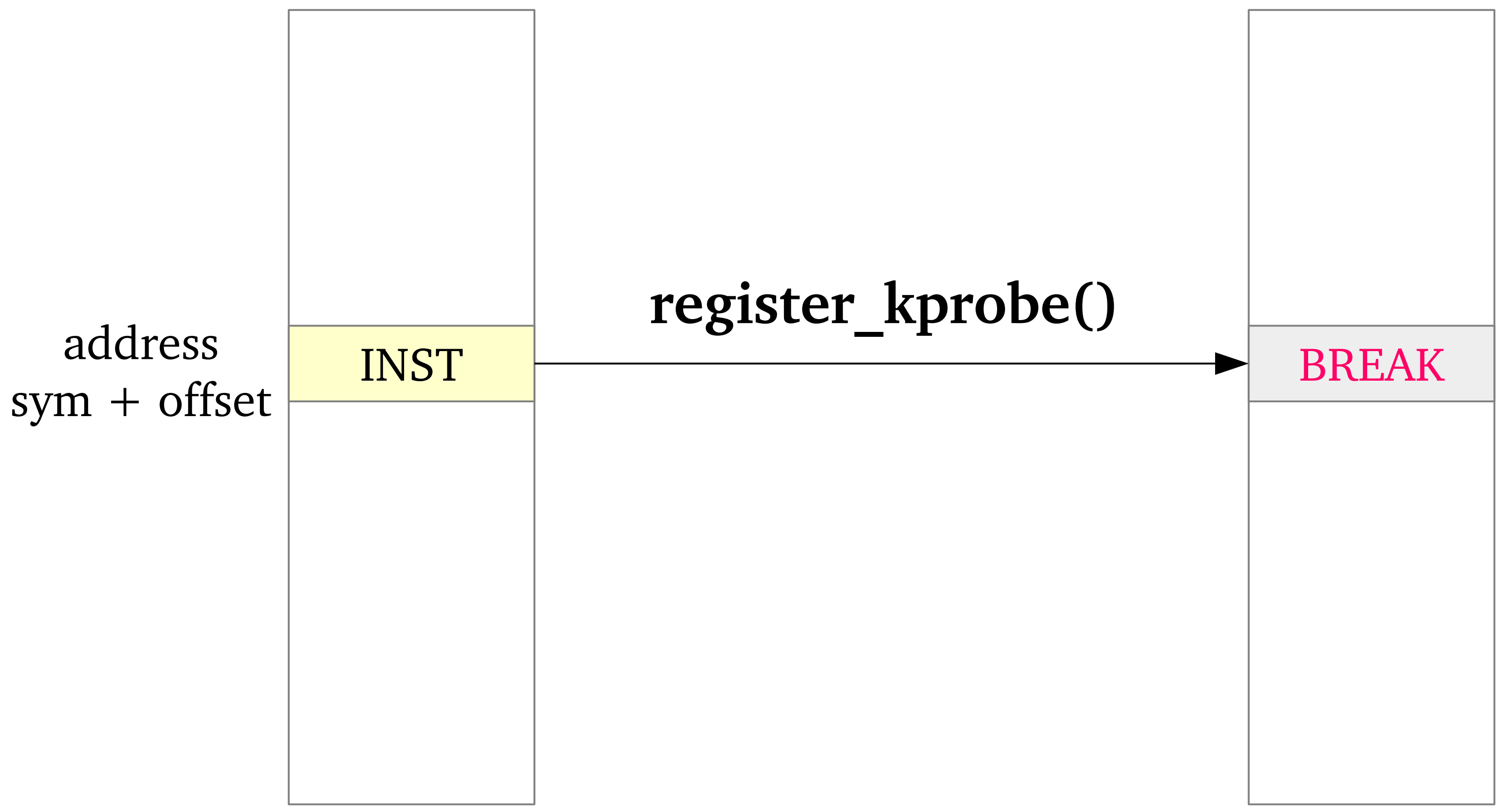
Kprobe Family

Kprobe

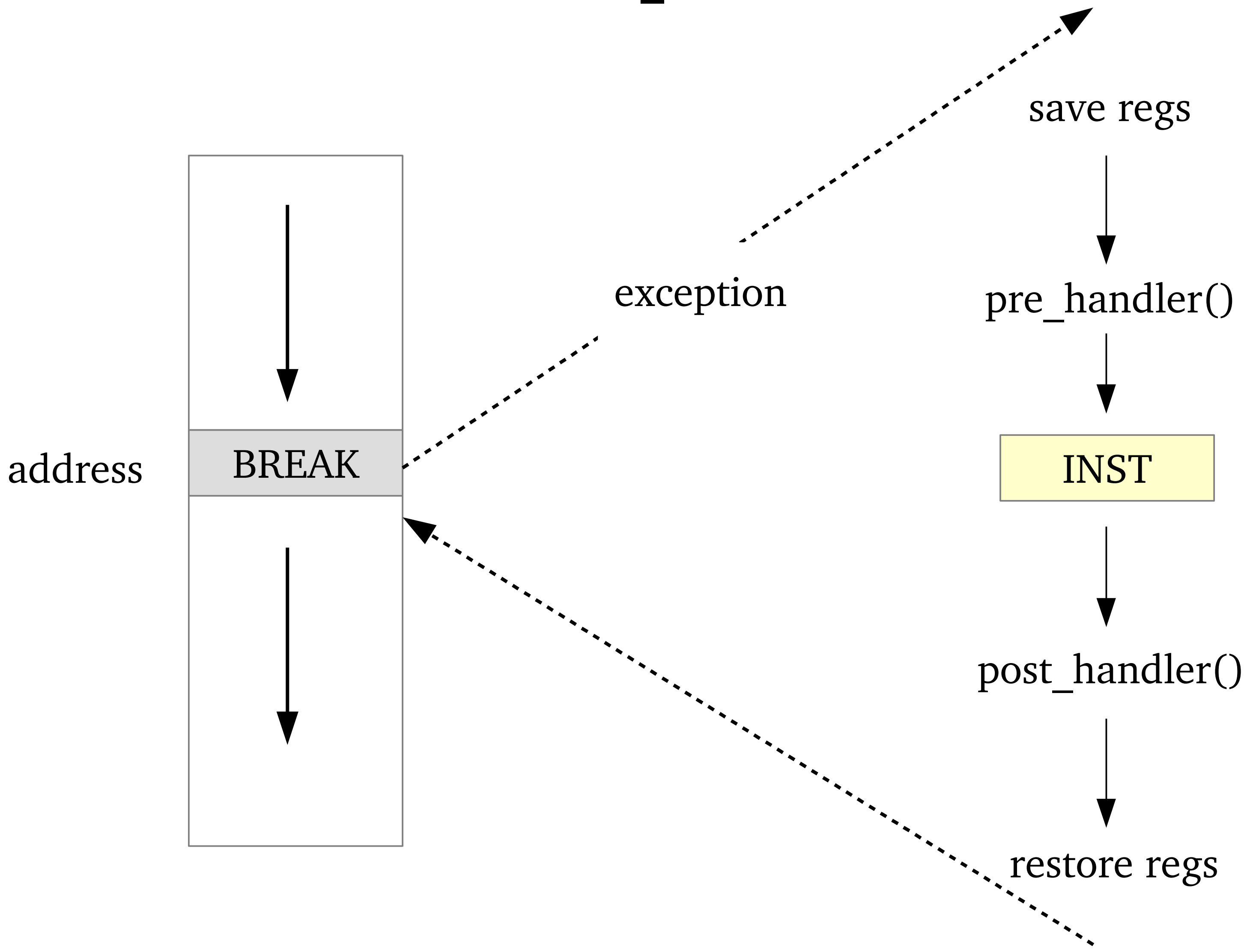
- Linux-2.6.9
- Write probe hooks in kernel module



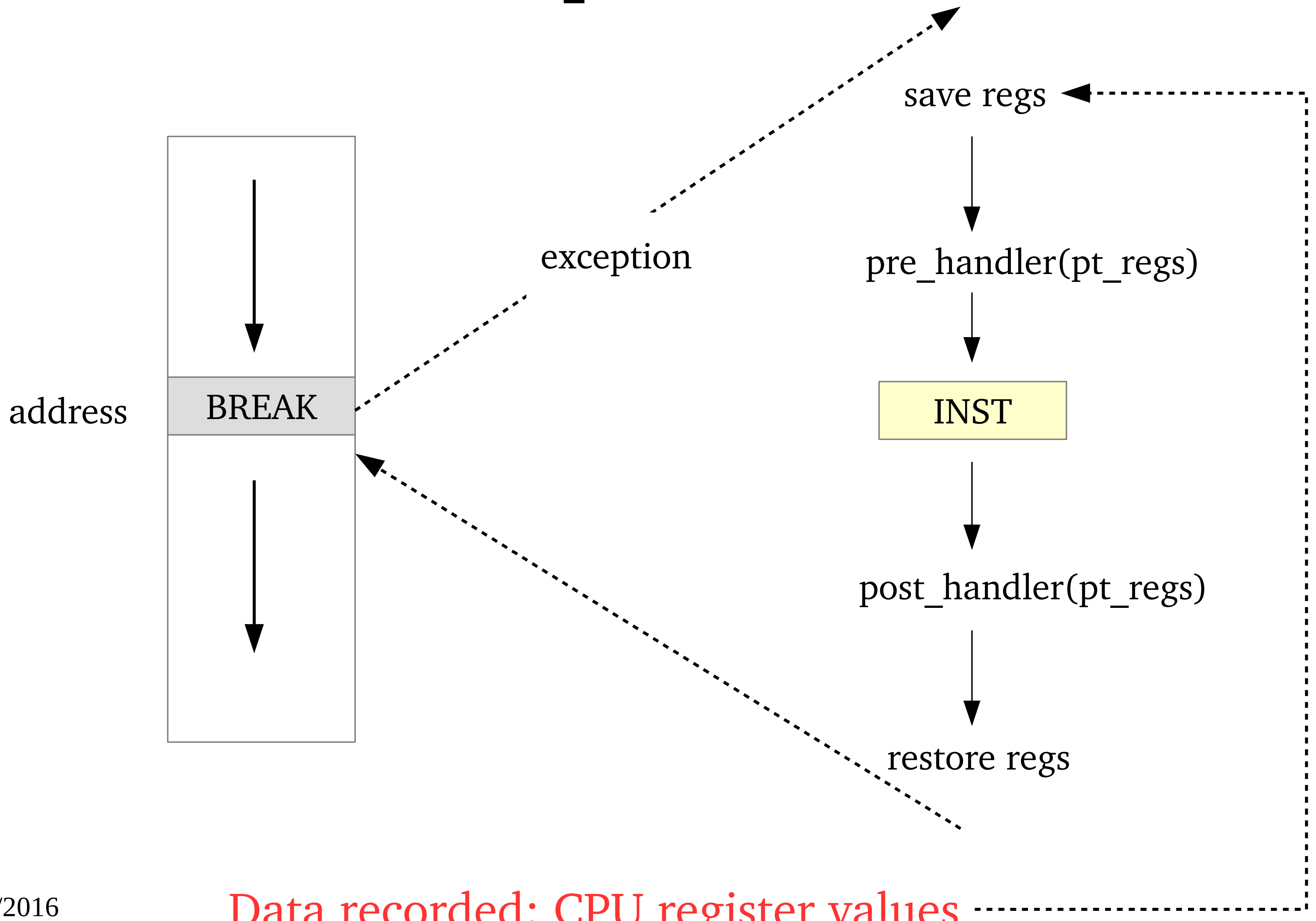
Kprobe



Kprobe

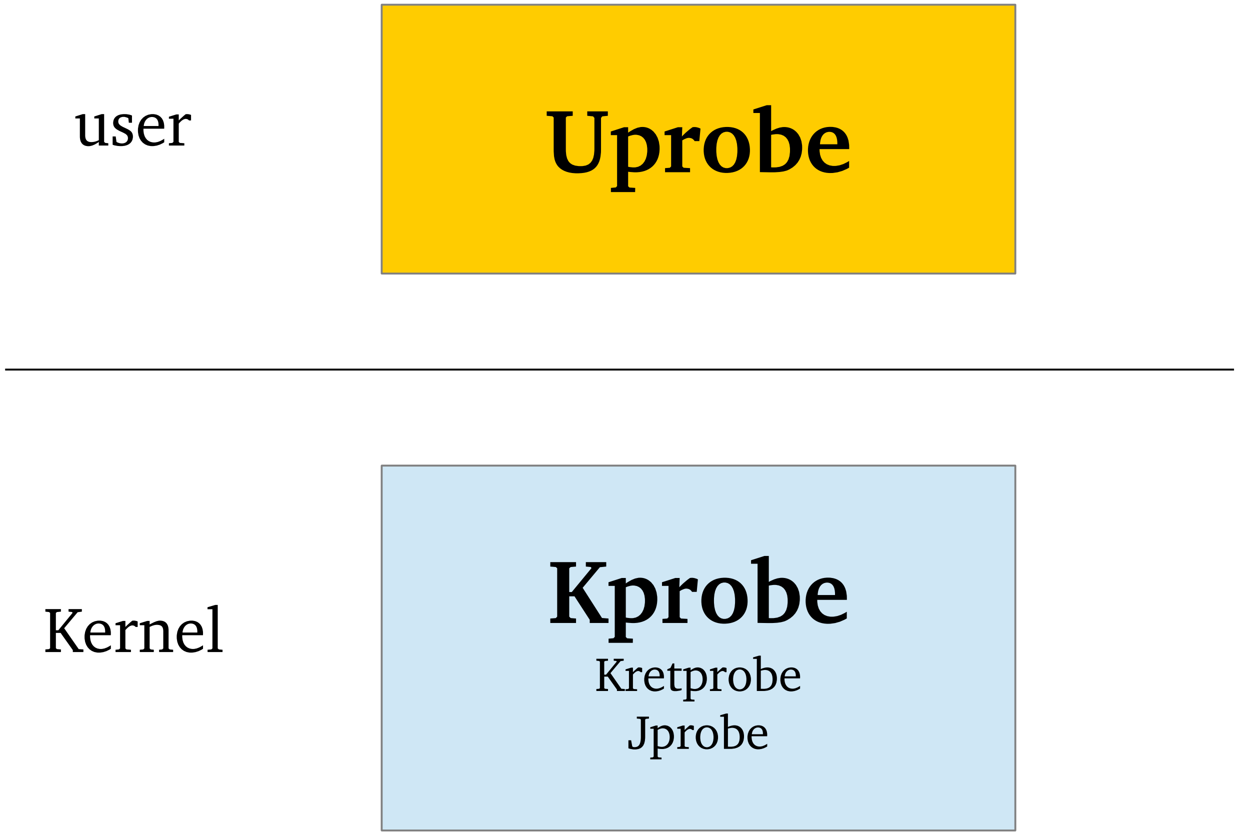

save regs
exception
pre_handler()
address
BREAK
INST
post_handler()
restore regs

Kprobe



Data recorded: CPU register values

Kprobe Variants



Uprobe

- Linux-3.5
- userspace breakpoints in kernel

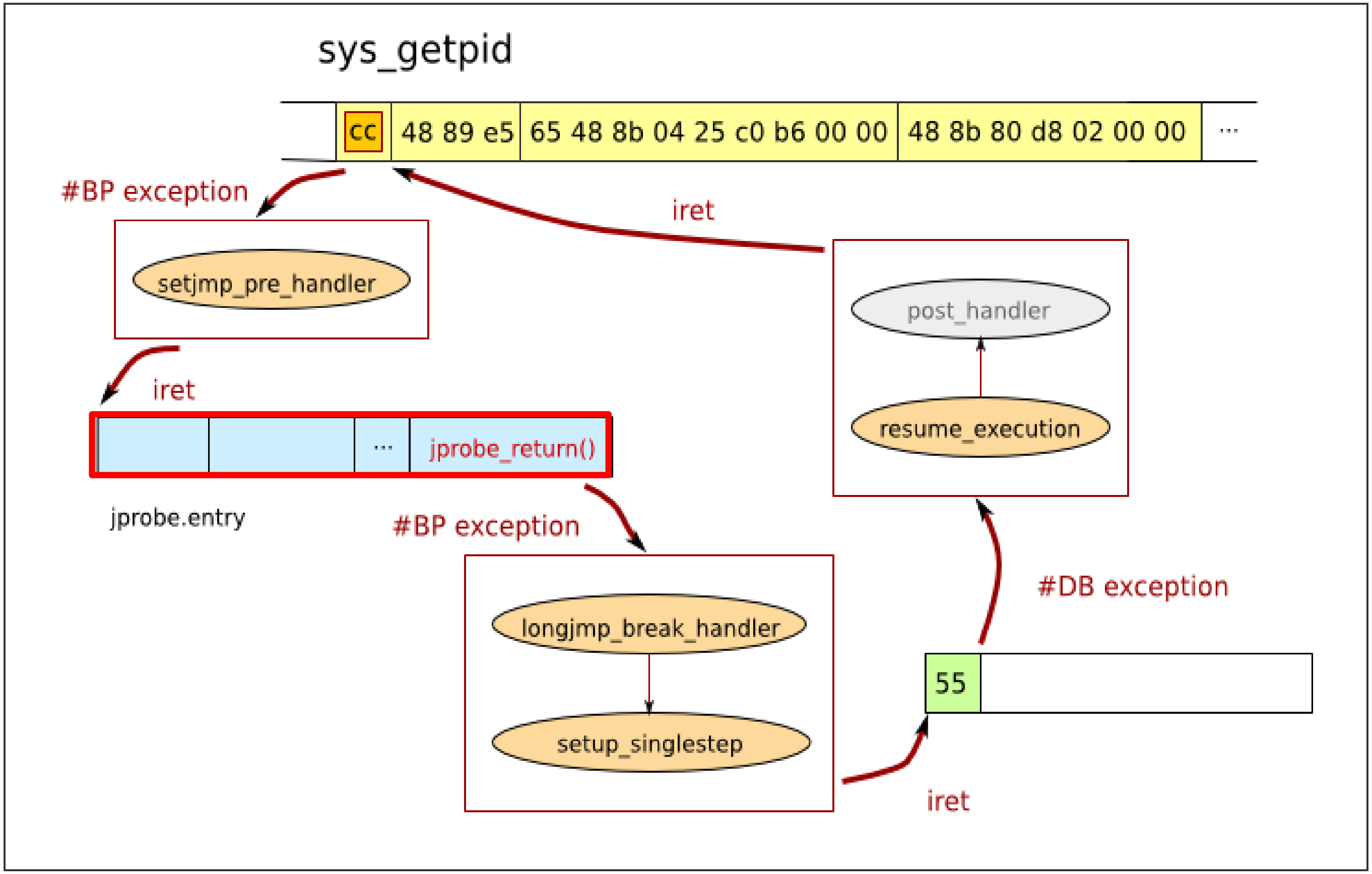
```
echo 'p:myapp /bin/bash:0x4245c0' > /sys/kernel/tracing/uprobe_events
```

jprobe

```
static long jsys_execve(const char __user *filenamei,
                        const char __user *const __user *argv,
                        const char __user *const __user *envp,
                        struct pt_regs *regs)
{
        printk(KERN_INFO "jprobe: regs = 0x%p\n",
               regs);

        /* Always end with a call to jprobe_return(). */
        jprobe_return();
        return 0;
}

static struct jprobe my_jprobe = {
        .entry                  = jsys_execve,
        .kp = {
                .symbol_name    = "SyS_execve",
        },
};
```

data: probed function arguments

jprobe

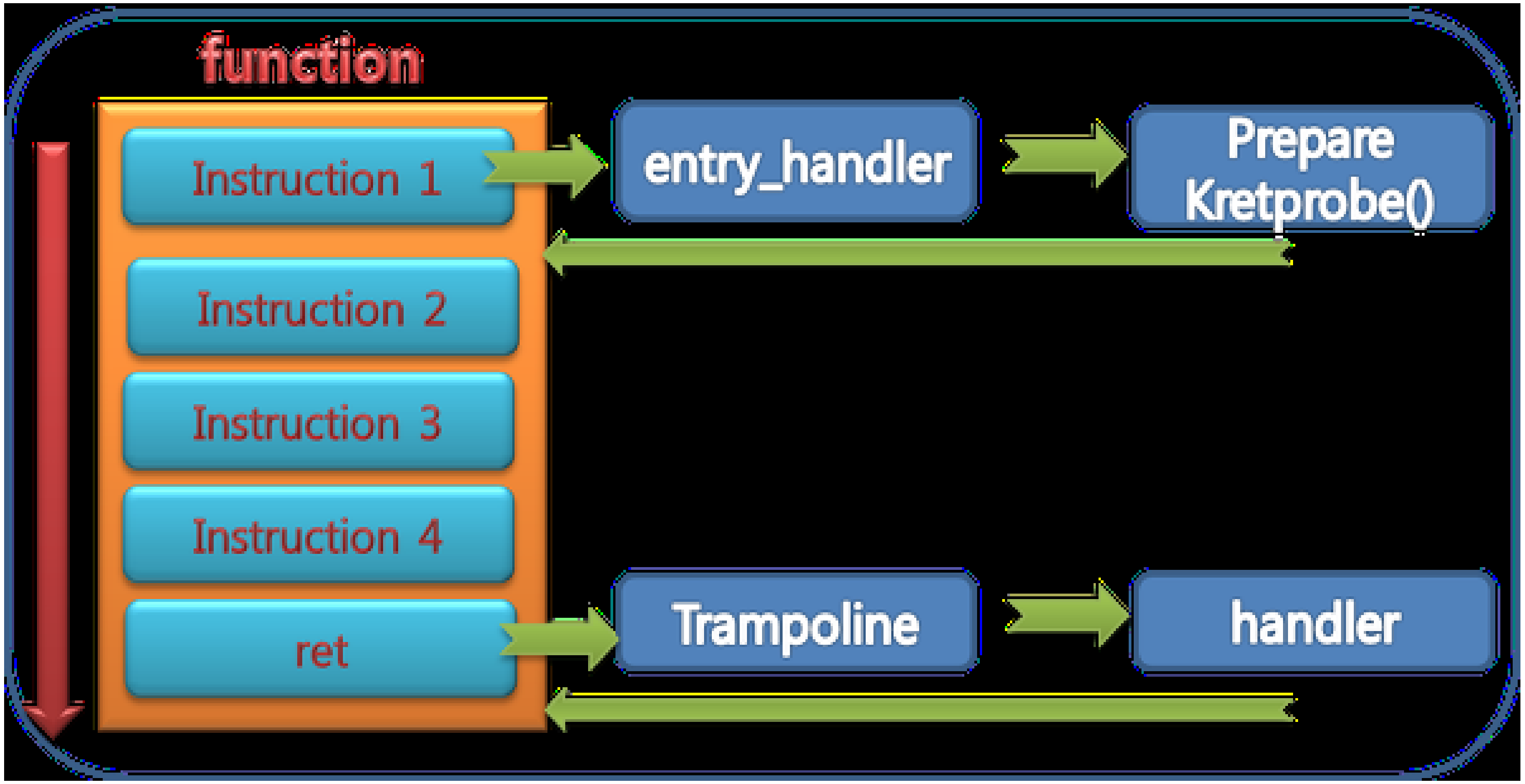


http://pds19.egloos.com/pds/201008/02/35/c0098335_4c55a764e1689.png

kretprobe

```
static int entry_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
        struct my_data *data;

        if (!current->mm)
                return 1;       /* Skip kernel threads */

        data = (struct my_data *)ri->data;
        data->entry_stamp = ktime_get();
        return 0;
}

/*
 * Return-probe handler: Log the return value and duration. Duration may turn
 * out to be zero consistently, depending upon the granularity of time
 * accounting on the platform.
 */
static int ret_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
        int retval = regs_return_value(regs);
        struct my_data *data = (struct my_data *)ri->data;
        s64 delta;
        ktime_t now;

        now = ktime_get();
        delta = ktime_to_ns(ktime_sub(now, data->entry_stamp));
        printk(KERN_INFO "%s returned %d and took %lld ns to execute\n",
                        func_name, retval, (long long)delta);
        return 0;
}
```

kretprobe



http://cfile26.uf.tistory.com/image/1311D5455136D6AF3B7251

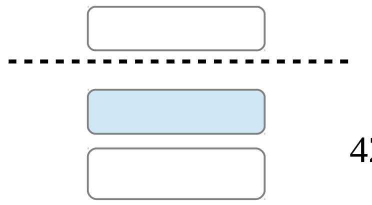
Kprobe Overhead [7]

cycles per iteration

	AMD Athlon 1.7GH	Pentium III 860MHz
kprobe	0.99 us	0.95 us
jprobe	0.82 us	1.61 us

Kprobe-based Event Tracing

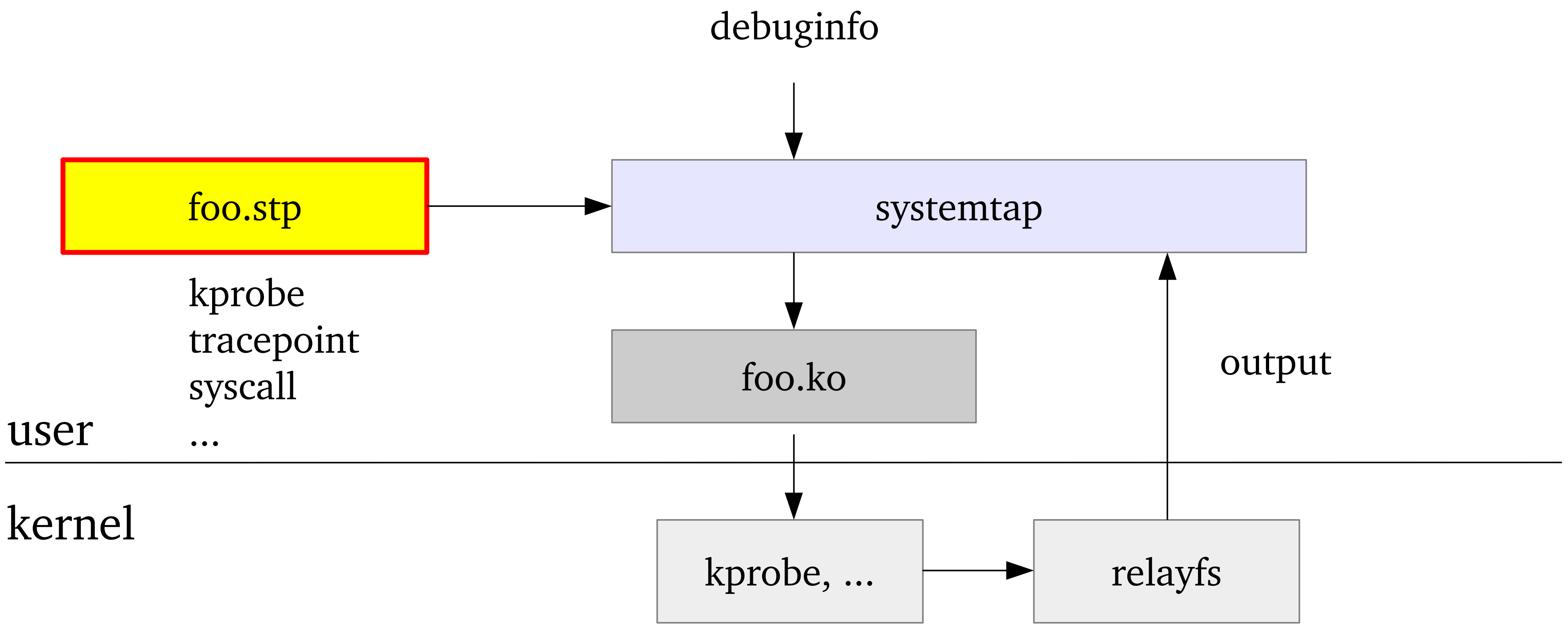
```
# echo 'r:myretprobe do_sys_open $retval' >> /sys/kernel/tracing/kprobe_events

# echo 1 > /sys/kernel/tracing/events/kprobes/myretprobe/enable

# cat /sys/kernel/tracing/trace
# tracer: nop
#
#         TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#            | |       |   ||||       |         |
          sh-746   [000] d...   40.96: myretprobe: (SyS_open+0x2c/0x30 <- do_sys_open) arg1=0x3
          sh-746   [000] d...   42.19: myretprobe: (SyS_open+0x2c/0x30 <- do_sys_open) arg1=0x3
                                                  .....
```



Utilities for Kprobe

- tracefs files
  - perf probe
- systemtap
  - debuted in 2005 in Red Hat Enterprise Linux 4
  - Probe by DSL script based on kprobe

Userspace Scripts: systemtap



perf + Tracing

# perf

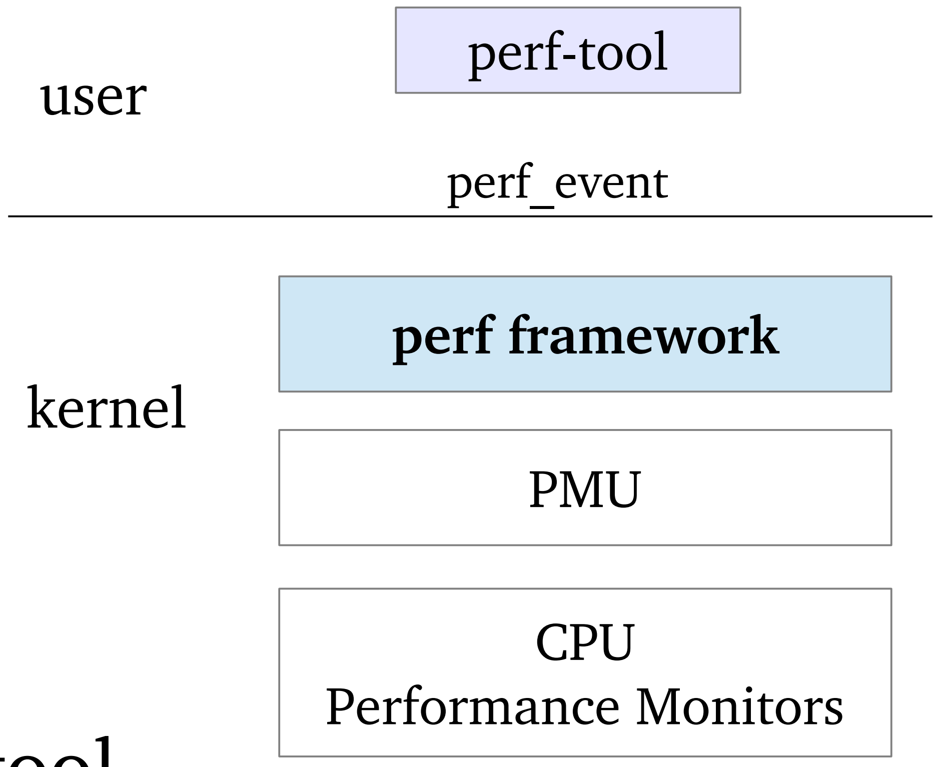
- Linux-2.6.31
- Statistics data

  # perf stat my-app args

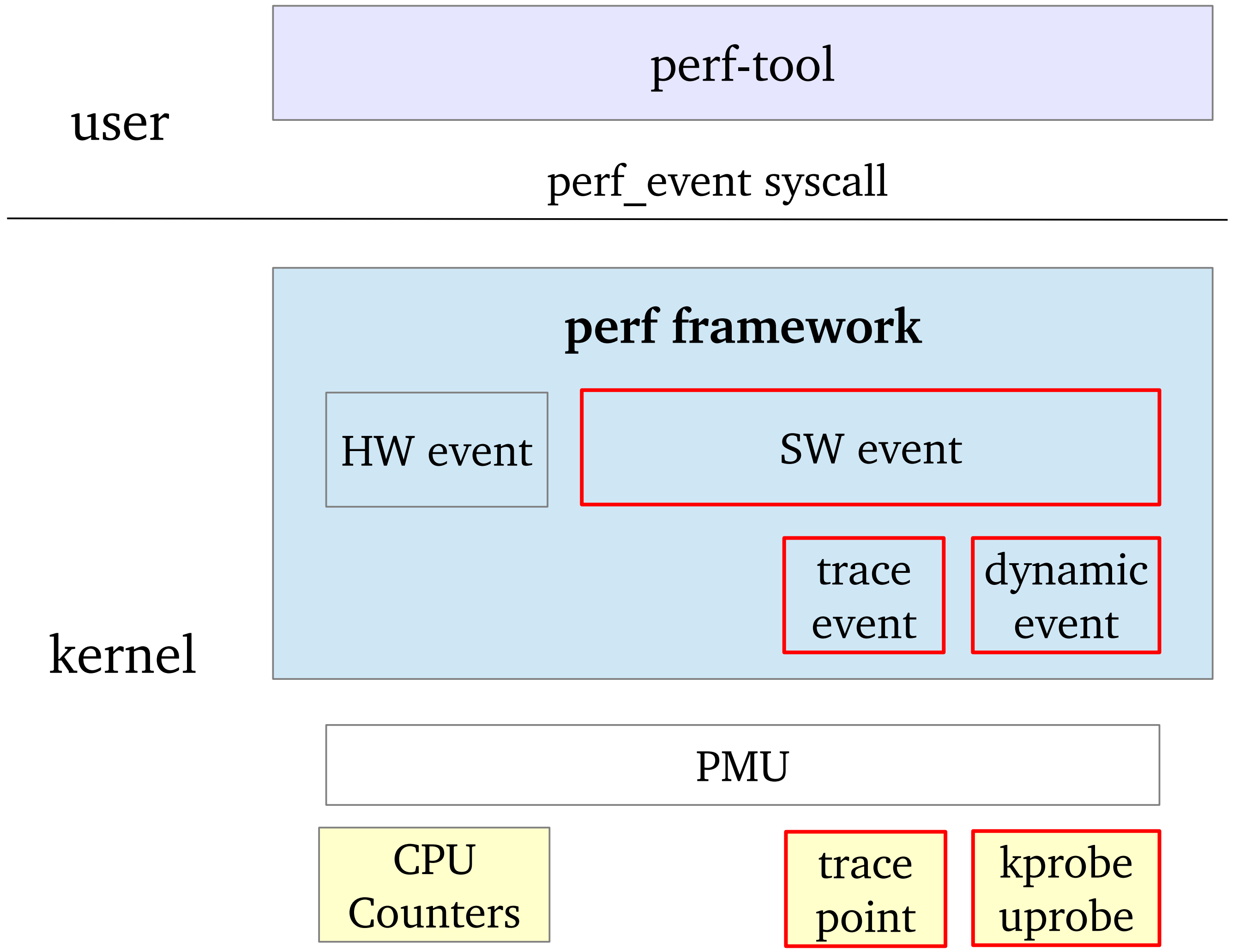
- Sampling record

  # perf record my-app args

- Other sub cmds of perf tool



perf Events

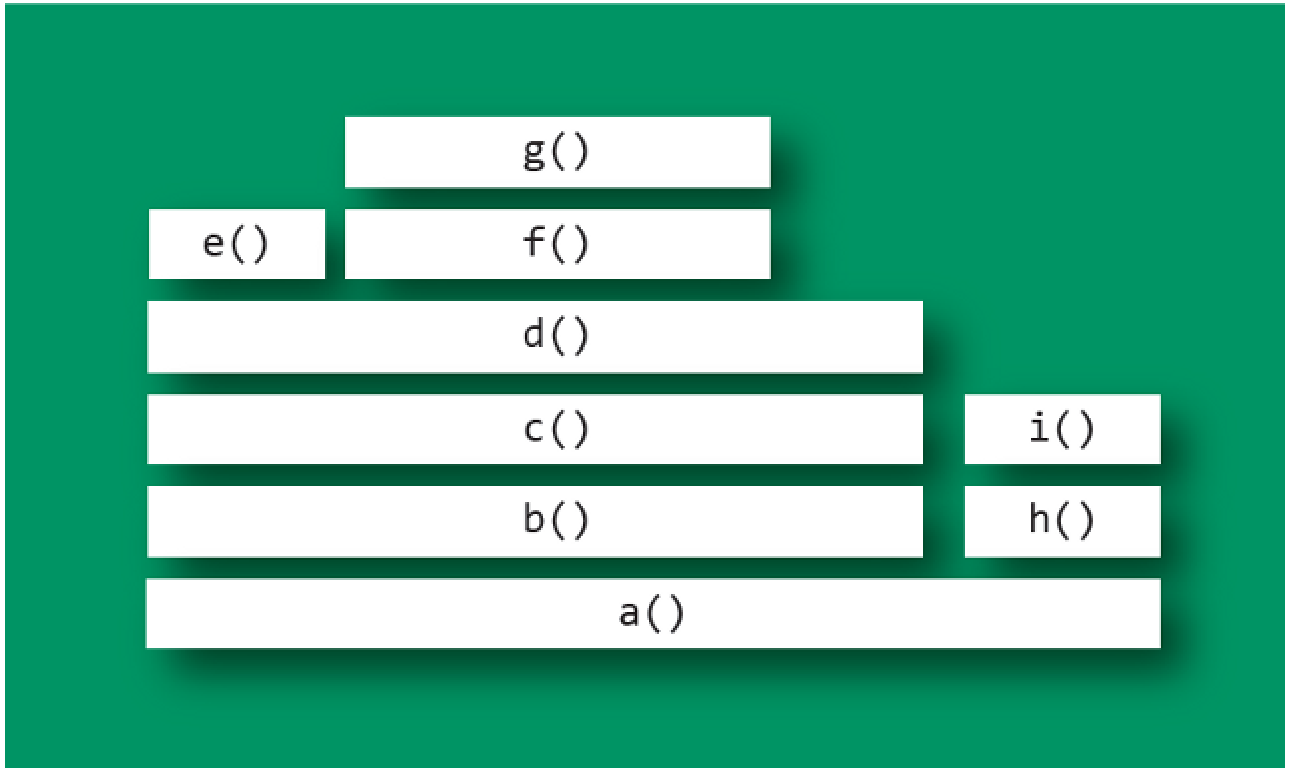

perf-tool
user
perf_event syscall
perf framework
HW event
SW event
trace event
dynamic event
kernel
PMU
CPU Counters
trace point
kprobe uprobe

perf Events

```
# perf record -e 'syscalls:sys_enter_*' -a -g -- sleep 60
```

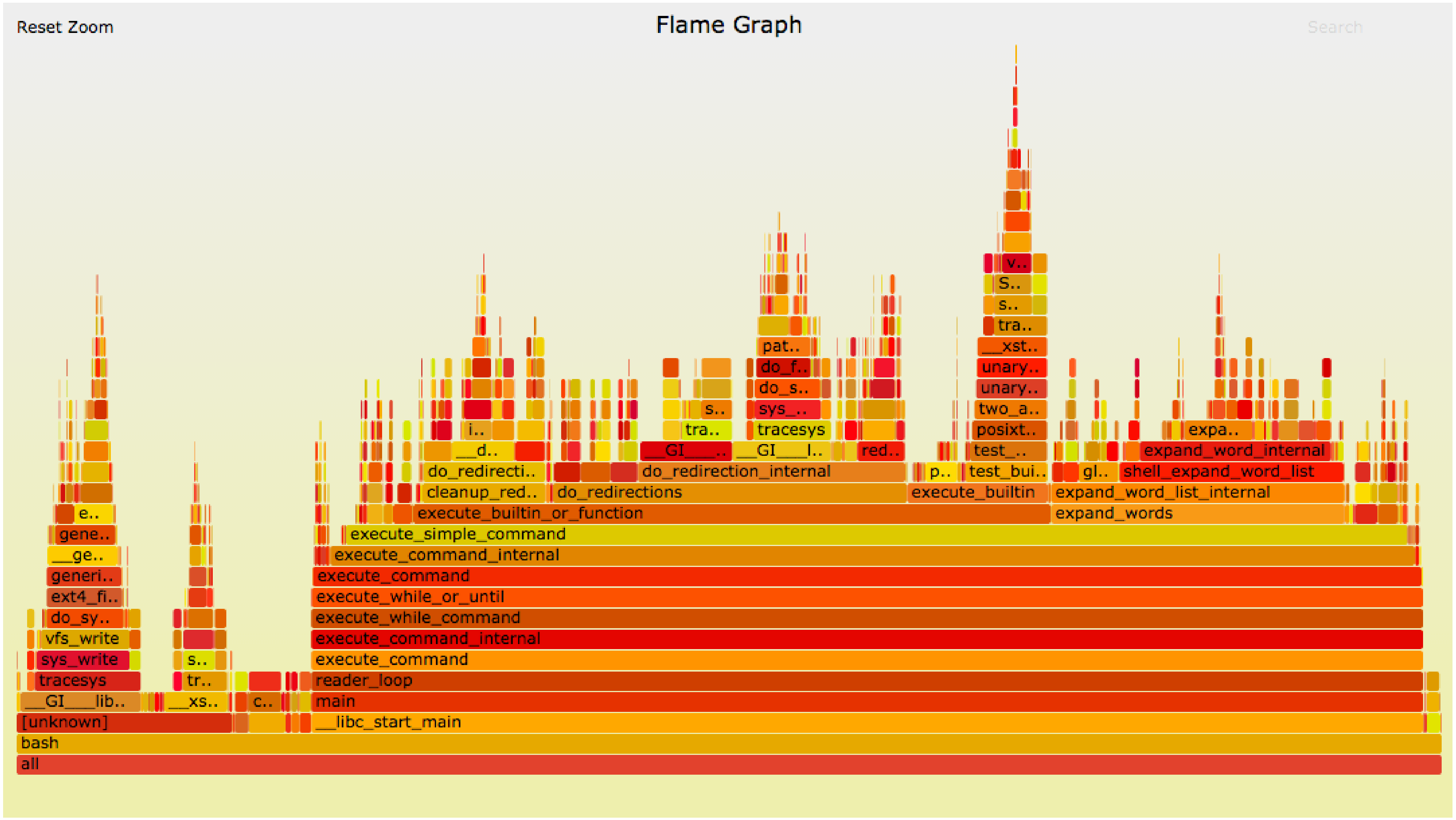
Flame Graph

FIGURE 6: **EXAMPLE FOR INTERPRETATION**



http://deliveryimages.acm.org/10.1145/2930000/2927301/gregg6.png

Flame Graph

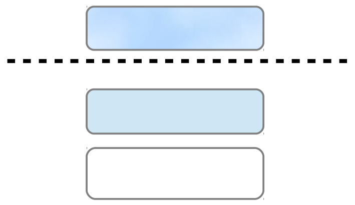


http://www.brendangregg.com/FlameGraphs/cpu-bash-flamegraph.png

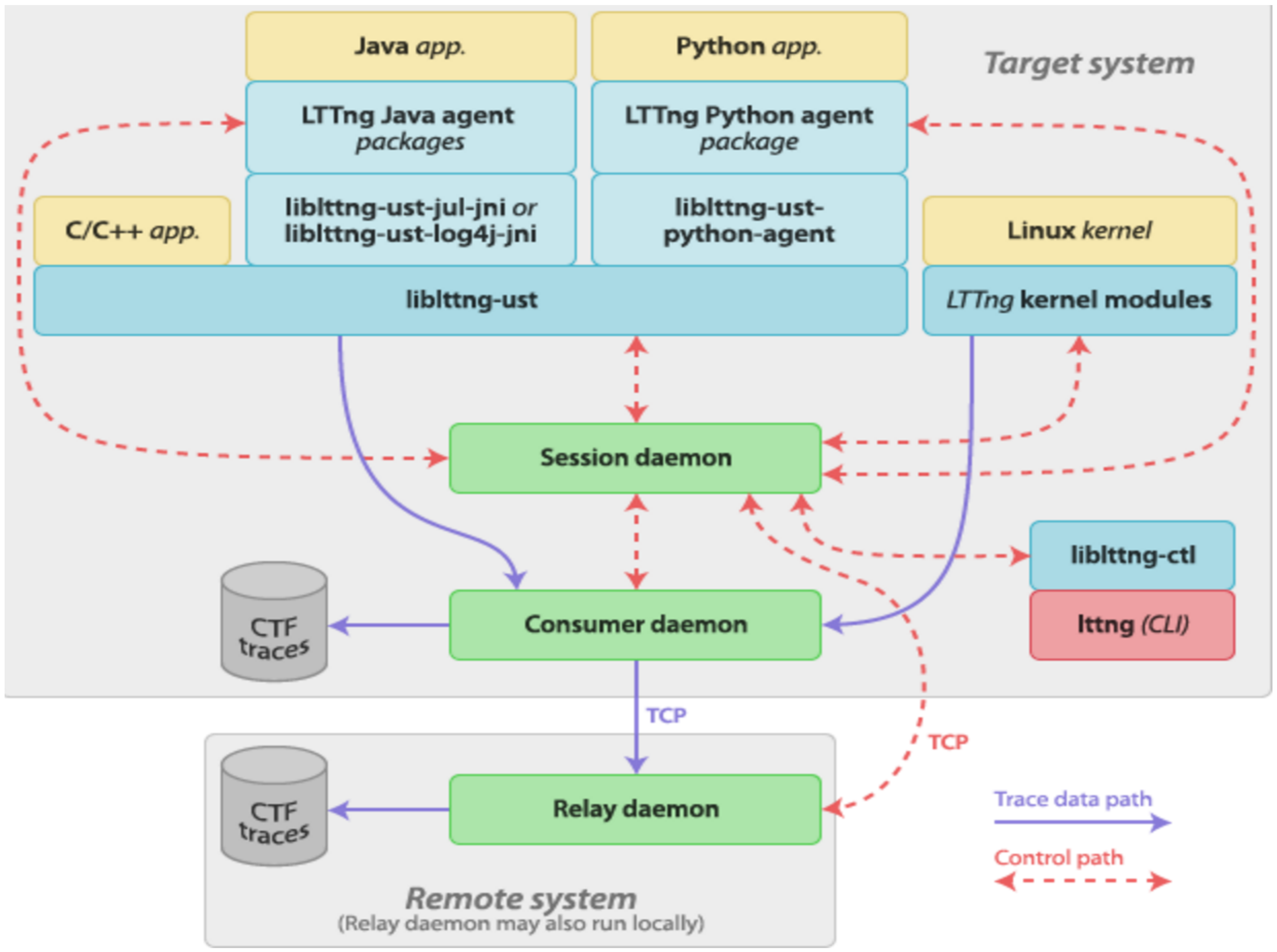
Flame Graph Tools for perf Data

```
# perf record -F 99 -a -g -- sleep 60

# perf script > out.perf


# /path/to/flamegraph/stackcollapse-perf.pl out.perf > out.folded

# /path/to/flamegraph/flamegraph.pl out.kern_folded > kernel.svg
```

LTTng

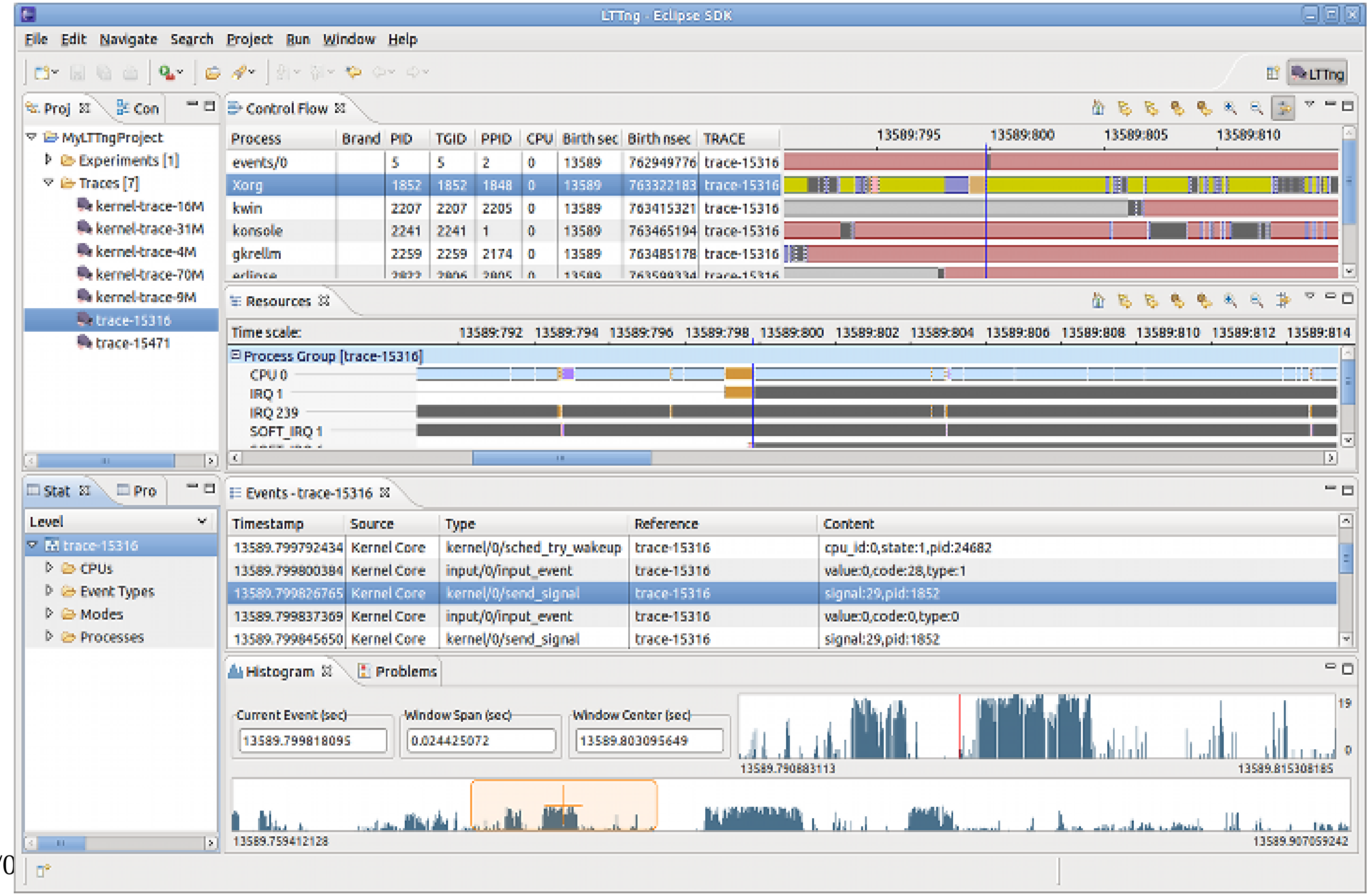


LTTng



http://lttng.org/images/docs27/plumbing-27.png

Eclipse LTTng Support



Process	Brand	PID	TGID	PPID	CPU	Birth sec	Birth nsec	TRACE
events/0		5	5	2	0	13589	762949776	trace-15316
Xorg		1852	1852	1848	0	13589	763322183	trace-15316
kwin		2207	2207	2205	0	13589	763415321	trace-15316
konsole		2241	2241	1	0	13589	763465194	trace-15316
gkrellm		2259	2259	2174	0	13589	763485178	trace-15316
eclipse		2822	2806	2805	0	13589	763509334	trace-15316

Timestamp	Source	Type	Reference	Content
13589.799792434	Kernel Core	kernel/0/sched_try_wakeup	trace-15316	cpu_id:0,state:1,pid:24682
13589.799800384	Kernel Core	input/0/input_event	trace-15316	value:0,code:28,type:1
13589.799826765	Kernel Core	kernel/0/send_signal	trace-15316	signal:29,pid:1852
13589.799837369	Kernel Core	input/0/input_event	trace-15316	value:0,code:0,type:0
13589.799845650	Kernel Core	kernel/0/send_signal	trace-15316	signal:29,pid:1852

Current Event (sec)	Window Span (sec)	Window Center (sec)
13589.799818095	0.024425072	13589.803095649

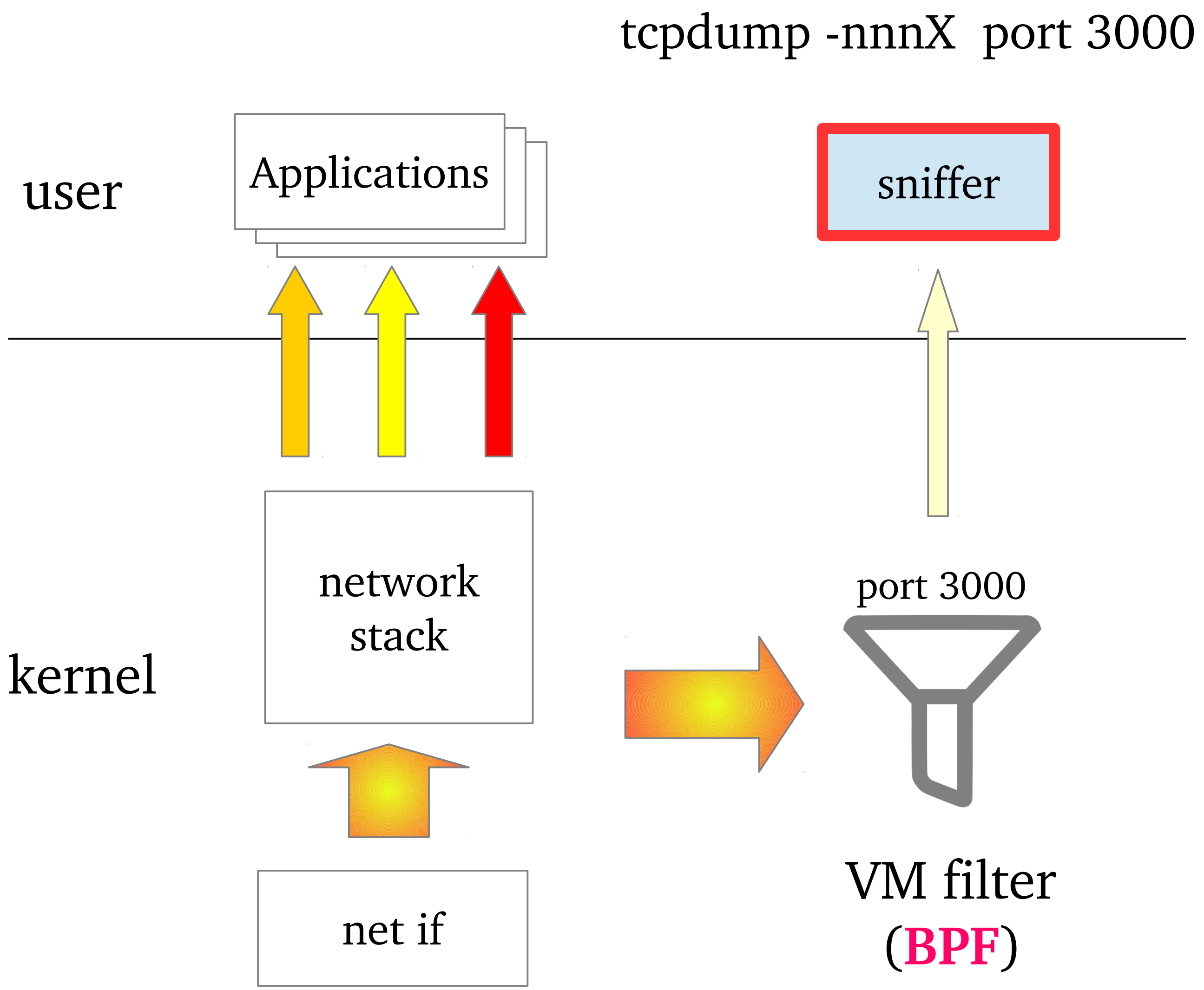
https://wiki.eclipse.org/images/e/ec/LTTngPerspective.png

Disadvantage of Previous Kernel Tracing

- Components are isolated
- Complex filters and scripts can be expensive
- Need more comprehensive tools. Some solutions
  - systemtap
  - LTTng
  - Dtrace
  - ktap

Tracing + eBPF

BPF – In-kernel Packet Filter



http://www.iconsdb.com/icons/download/gray/empty-filter-512.png

eBPF

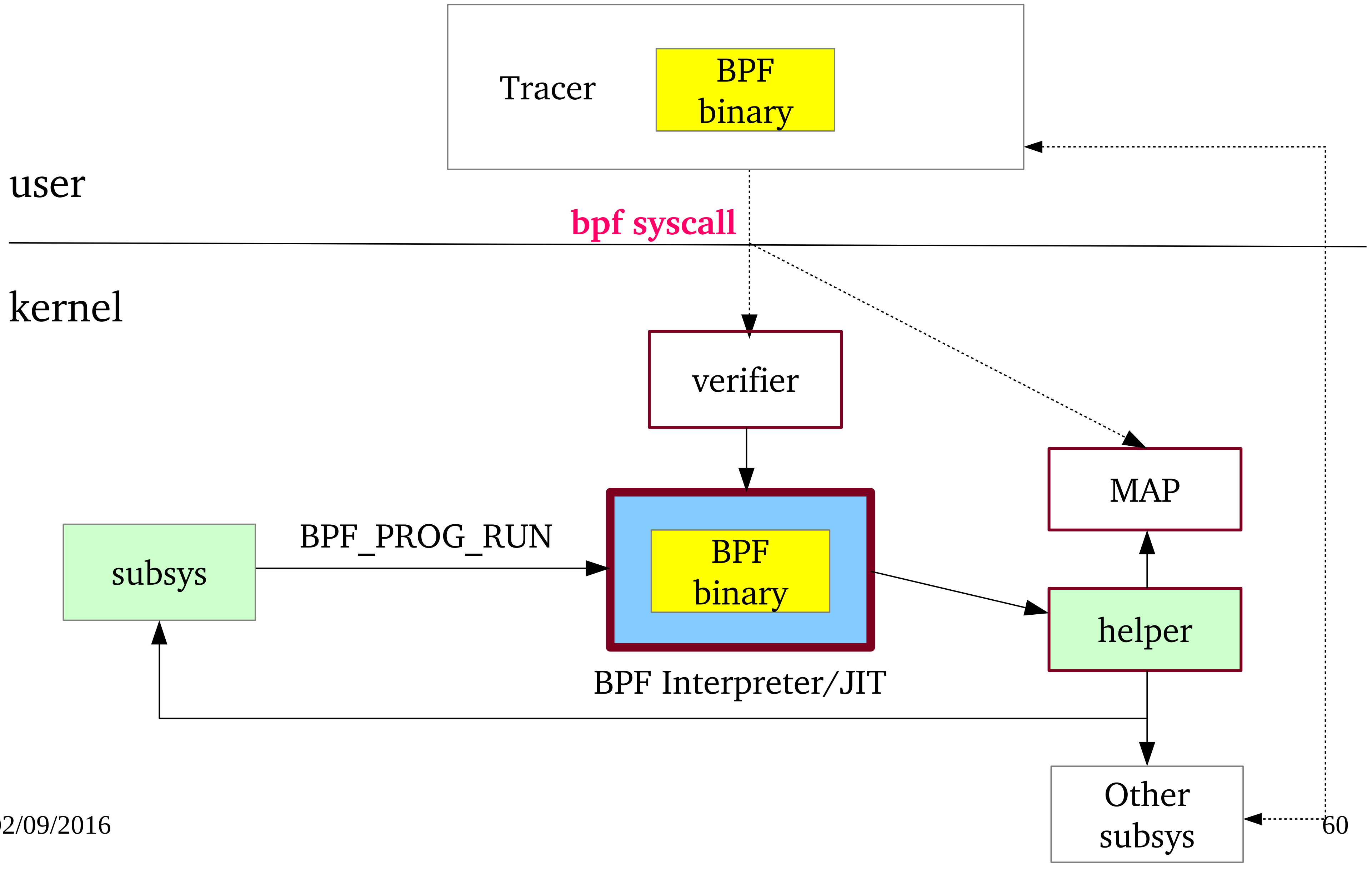
- (Linux-3.15) Re-designed by Alexei Starovoitov
  - Write programs in restricted C
    - compile to BPF with LLVM
  - Just-in-time map to modern 64-bit CPU with minimal performance overhead

Areas Use eBPF

more than a filter today

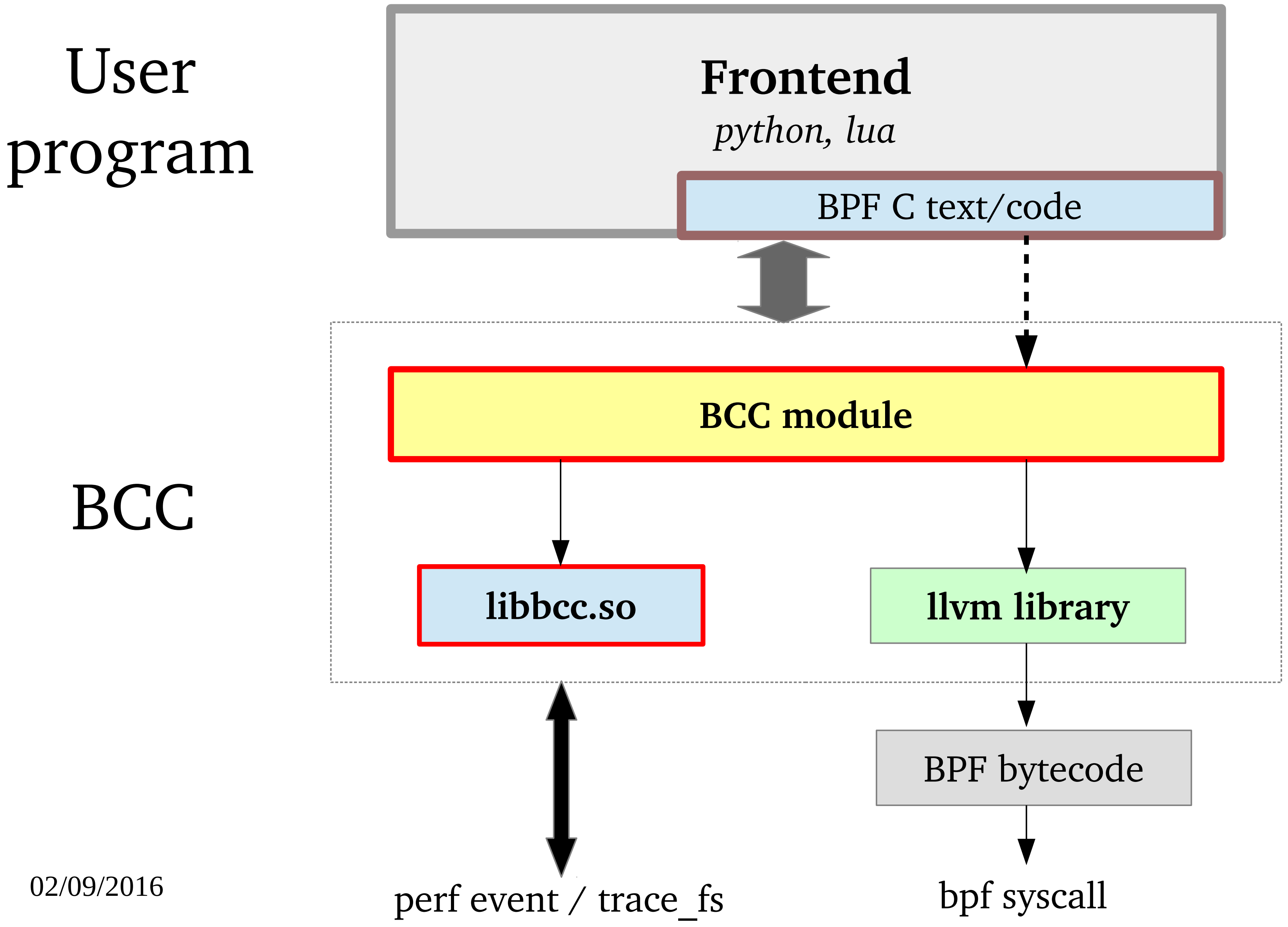
- Seccomp filters of syscalls (chrome sandboxing)
- Packet classifier for traffic contol
- Actions for traffic control
- Xtables packet filtering
- Tracing
  - (Linux-4.1) attach to kprobe
  - (Linux-4.7) attach to tracepoint

eBPF Architecture



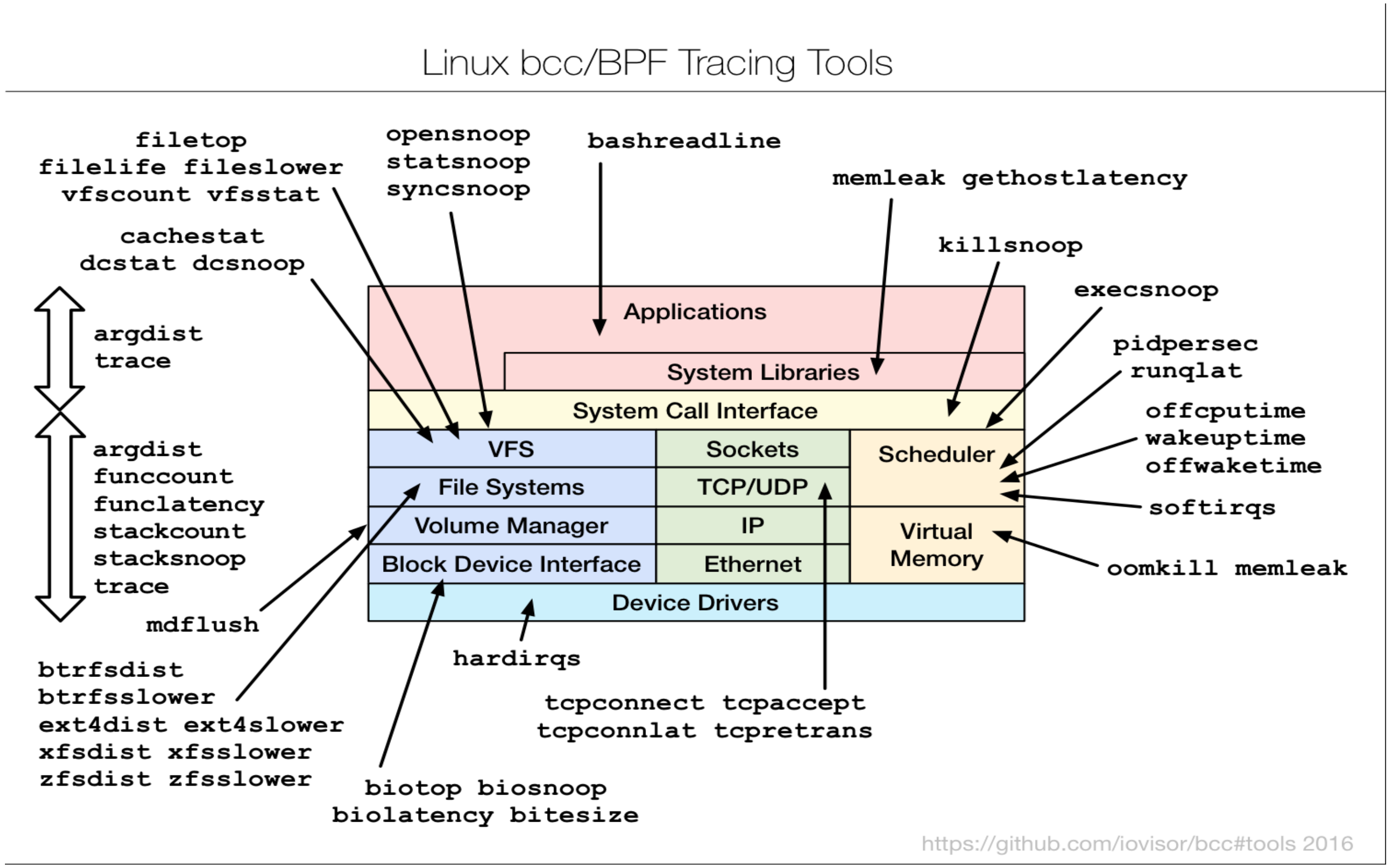
Write Customized Tracing Script Is Possible Now!

eBPF Utilitiy – IO Visor BCC



Current Tracing Scripts in BCC

Tools for BPF-based Linux IO analysis, networking, monitoring, and more



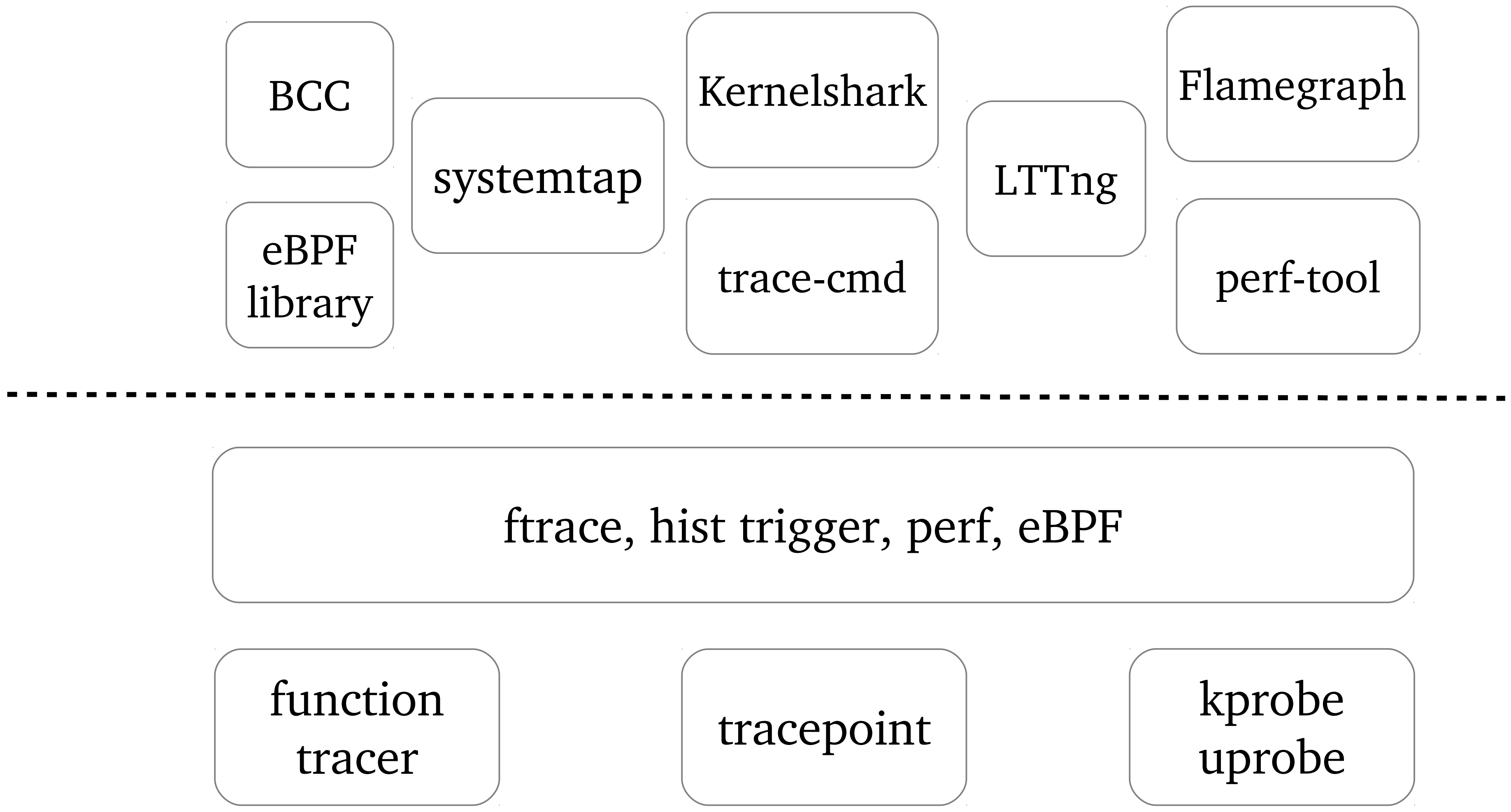
https://raw.githubusercontent.com/iovisor/bcc/master/images/bcc_tracing_tools_2016.png

perf + eBPF [8]

- Linux-4.8-rc (?) by Wang Nan in Huawei
- On-goning staff and future plans
  - Load BPF
  - Tracing rare outliner
  - Integrate LLVM and other frontend

Summary

Linux Kernel Tracing



Q & A

Reference

[1] Steven Rostedt (Dec. 2009), “Debugging the kernel using Ftrace - part 1”, LWN

[2] Steven Rostedt (Feb. 2011), “Using KernelShark to analyze the real-time scheduler”, LWN

[3] 章亦春 , “ 动态追踪技术漫谈”

[4] Brendan Gregg, (Feb. 2016), "Linux 4.x Performance Using BPF Superpowers", presented at Performance@ scale 2016

[5] Gary Lin (Mar. 2016), “eBPF: Trace from Kernel to Userspace ”, presented at OpenSUSE Technology Sharing Day 2016

[6] Kernel documentation, “Using the Linux Kernel Tracepoints”

[7] William Cohen (Feb. 2005), “cost of kprobe and jprobe operations”, systemtap mailing list

[8] Wang Nan (Aug. 2016), “Performance Monitoring and AnalysisUsing perf+BPF” , LinuxCon North America 2016

Rights to Copy

copyright © 2016 Viller Hsiao

THE END