KERNEL PROBES
STATIC AND DYNAMIC
KPROBES

Important Notice : Courseware - Legal

This courseware is both the product of the author and of freely available opensource and/or public domain materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of the **courseware PDFs are considered proprietary** and thus cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - **source code** and binaries (where applicable) - that form part of this courseware, and that are made available to the participant, are released under the terms of the permissive **MIT license**.
Under the terms of the MIT License, you can certainly use the source code provided here; you must just attribute the original source (author of this courseware and/or other copyright/trademark holders).

VERY IMPORTANT :: Before using this source(s) in your project(s), you *MUST* check with your organization's legal staff that it is appropriate to do so.

The courseware PDFs are *not* under the MIT License, they are to be kept confidential, non-distributable without consent, for your private internal use only.

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant VM are subject to change at any point in time without prior notice to individual participants.

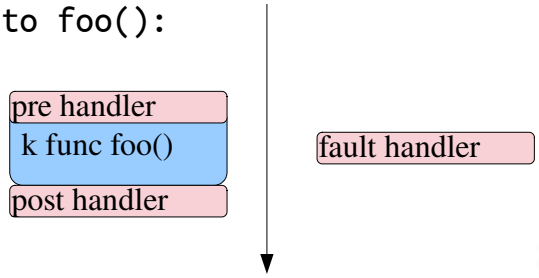
Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind, and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Dynamic Probes – Kprobes

Now refer to the original document for Kprobes (under the kernel src tree):
Documentation/kprobes.txt

Register a kprobe to foo():

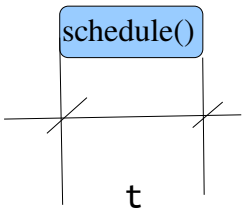
- pre-handler
- post-handler



'fault injection'

sample code: kernel : samples/kprobes

Kprobes : suited to performing instrumentation on production systems !



DEPRECATED:

- 4.15 onward: Jumper probes / jprobe – gives you access to all parameters
- 5.14 onward: fault handler deprecated.

Kretprobe : gives access to return value !

Kprobes facility is in mainline since 2.6.9.

(Minor) Note:
In later kernels (saw this on 2.6.27), the Kprobes configuration option CONFIG_KPROBES (within the kernel configuration menu system), is not under the 'Kernel Hacking' menu section but under the 'General setup' menu.

From the *make menuconfig* : General setup | [*] Kprobes / Help :

--snip--
CONFIG_KPROBES:

Kprobes allows you to trap at almost any kernel address and
execute a callback function. register_kprobe() establishes
a probepoint and specifies the callback. Kprobes is useful
for kernel debugging, non-intrusive instrumentation and testing.
If in doubt, say "N".

Symbol: KPROBES [=y]
Prompt: Kprobes
Defined at arch/Kconfig:19
Depends on: KALLSYMS && MODULES && HAVE_KPROBES

Location:
-> General setup

--snip--

Note:
Certain kernel functions cannot be probed (kprobes/jprobes/kretprobes): these include functionality of kprobes itself. To mark a function as non-probe-able, the kernel developers use a compiler attribute called “_kprobe_blacklist”.

(On recent kernels, it's marked using a macro `NOKPROBE_SYMBOL(funcname)`).

Also, **debugfs** exposes some useful kprobes stuff:

/sys/kernel/debug/kprobes/
blacklist : list of nokprobe symbols
enabled : boolean
list : list of currently active kernel probes.

Practical

Static kprobes:

Refer Kprobes code-level examples from:

- my LKD (Linux Kernel Debugging) book's GitHub repo, here:
  - https://github.com/PacktPublishing/Linux-Kernel-Debugging/tree/main/ch4/kprobes
- kernel source: samples/ :
  - https://elixir.bootlin.com/linux/v6.1.15/source/samples/kprobes

---

Dynamic kprobes or kprobe-based event tracing

From the **perf-tools[-unstable]** package (Brendan Gregg).

Install if not already done so...

```
$ kprobe-perf
USAGE: kprobe [-FhHsv] [-d secs] [-p PID] [-L TID] kprobe_definition [filter]
                -F              # force. trace despite warnings.
                -d seconds      # trace duration, and use buffers
                -p PID          # PID to match on events
                -L TID          # thread id to match on events
                -v              # view format file (don't trace)
                -H              # include column headers
                -s              # show kernel stack traces
                -h              # this usage message

Note that these examples may need modification to match your kernel
version's function names and platform's register usage.
   eg,
       kprobe p:do_sys_open
                                # trace open() entry
       kprobe r:do_sys_open
                                # trace open() return
       kprobe 'r:do_sys_open $retval'
                                # trace open() return value
       kprobe 'r:myopen do_sys_open $retval'
                                # use a custom probe name
       kprobe 'p:myopen do_sys_open mode=%cx:u16'
                                # trace open() file mode
       kprobe 'p:myopen do_sys_open filename=+0(%si):string'
                                # trace open() with filename
       kprobe -s 'p:myprobe tcp_retransmit_skb'
                                # show kernel stacks
       kprobe 'p:do_sys_open file=+0(%si):string' 'file ~ "*stat"'
                                # opened files ending in "stat"
```

See the man page and example file for more info.

Example- trace all open's system-wide:
Find that trying to do so via *do_sys_open()* doesn't help that much; instead use the *vfs_open* or *do_sys_openat2*:

```
# kprobe-perf p:vfs_open
...
     <...>-867525  [005] .... 27054.800574: vfs_open: (vfs_open+0x0/0x40)
     <...>-867525  [005] .... 27054.800604: vfs_open: (vfs_open+0x0/0x40)
    systemd-oomd-664     [001] .... 27055.000487: vfs_open: (vfs_open+0x0/0x40)
    systemd-oomd-664     [001] .... 27055.407545: vfs_open: (vfs_open+0x0/0x40)
    systemd-oomd-664     [001] .... 27055.407687: vfs_open: (vfs_open+0x0/0x40)
    systemd-oomd-664     [001] .... 27055.407756: vfs_open: (vfs_open+0x0/0x40)
    systemd-oomd-664     [001] .... 27055.407797: vfs_open: (vfs_open+0x0/0x40)
    systemd-oomd-664     [001] .... 27055.407825: vfs_open: (vfs_open+0x0/0x40)
    systemd-oomd-664     [001] .... 27055.407849: vfs_open: (vfs_open+0x0/0x40)
    systemd-oomd-664     [001] .... 27055.407875: vfs_open: (vfs_open+0x0/0x40)
 systemd-journal-279     [003] .... 27055.500509: vfs_open: (vfs_open+0x0/0x40)
 systemd-journal-279     [003] .... 27055.500573: vfs_open: (vfs_open+0x0/0x40)
 systemd-journal-279     [003] .... 27055.500587: vfs_open: (vfs_open+0x0/0x40)
...
 snap-seccomp-867538  [001] .... 27055.521421: vfs_open: (vfs_open+0x0/0x40)
    snap-seccomp-867538  [001] .... 27055.522426: vfs_open: (vfs_open+0x0/0x40)
            snap-867526  [004] .... 27055.523538: vfs_open: (vfs_open+0x0/0x40)
            snap-867526  [004] .... 27055.523661: vfs_open: (vfs_open+0x0/0x40)
            snap-867526  [004] .... 27055.524165: vfs_open: (vfs_open+0x0/0x40)
...
 systemd-1226    [000] .... 27055.617141: vfs_open: (vfs_open+0x0/0x40)
         systemd-1226    [000] .... 27055.617152: vfs_open: (vfs_open+0x0/0x40)
         systemd-1226    [000] .... 27055.617167: vfs_open: (vfs_open+0x0/0x40)
...
#
```

TIP- on recent systems (as of Mar '23), tracing file opens works with the *do_sys_openat2()* !

Tracing the open() and printing the file being opened!

```
tracing # kprobe-perf -H -d1 'p:do_sys_openat2 file=+0(%si):string'
Tracing kprobe do_sys_openat2 for 1 seconds (buffered)...
# tracer: nop
#
# entries-in-buffer/entries-written: 31/31   #P:6
#
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              ||| /     delay
#           TASK-PID     CPU#  ||||   TIMESTAMP  FUNCTION
#              | |         |   ||||      |         |
          sleep-1055844 [002] .... 33037.692063: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/etc/ld.so.cache"
          sleep-1055844 [002] .... 33037.692075: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/lib/x86_64-linux-gnu/libc.so.6"
          sleep-1055844 [002] .... 33037.692232: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/usr/lib/locale/locale-archive"
      irqbalance-716     [003] .... 33037.734003: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/proc/interrupts"
      irqbalance-716     [003] .... 33037.734077: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/proc/stat"
      irqbalance-716     [003] .... 33037.734103: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/proc/irq/20/smp_affinity"
      irqbalance-716     [003] .... 33037.734112: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/proc/irq/0/smp_affinity"
      irqbalance-716     [003] .... 33037.734118: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/proc/irq/1/smp_affinity"
      irqbalance-716     [003] .... 33037.734124: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/proc/irq/8/smp_affinity"
      irqbalance-716     [003] .... 33037.734129: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/proc/irq/12/smp_affinity"
      irqbalance-716     [003] .... 33037.734135: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/proc/irq/14/smp_affinity"
      irqbalance-716     [003] .... 33037.734141: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/proc/irq/15/smp_affinity"
    systemd-oomd-664     [003] .... 33037.793507: do_sys_openat2: (do_sys_openat2+0x0/0x150) file="/sys/fs/cgroup/user.slice/user-1000.s
lice/user@1000.service/memory.pressure"
```

[...]

Cmd:

```
sudo kprobe-perf -H -d1 'p:do_sys_openat2 file=+0(%si):string'

-d seconds      # trace duration, and use buffers
-H              # include column headers

-s              # show kernel stack traces
```

Next example:
Trace *softirqs* as they occur!

Q. How will I know which functions are available to trace?
A. Can check like this:

```
# grep 'softirq' /sys/kernel/tracing/available_filter_functions
do_softirq_own_stack
__traceiter_softirq_entry
__traceiter_softirq_exit
__traceiter_softirq_raise
ksoftirqd_should_run
run_ksoftirqd
do_softirq
__raise_softirq_irqoff
raise_softirq
...
```

Lets' trace the *do_softirq()* !
I run this command (below) and in another terminal window, perform a single *ping* !

```
# kprobe-perf -H -s 'p:do_softirq'
Tracing kprobe do_softirq. Ctrl-C to end.
# tracer: nop
#
# entries-in-buffer/entries-written: 0/0   #P:6
#
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              ||| /     delay
#           TASK-PID     CPU#  ||||   TIMESTAMP  FUNCTION
#              | |         |   ||||      |         |
           <...>-1090075 [000] ...1 34127.015421: do_softirq:
(do_softirq+0x0/0x80)
            ping-1090075 [000] ...1 34127.015437: <stack trace>
 => do_softirq
 => ip_finish_output2
 => __ip_finish_output
 => ip_finish_output
 => ip_output                          << read the kernel stack bottom-up >>
 => ip_send_skb
 => udp_send_skb
 => udp_sendmsg
 => inet_sendmsg
 => sock_sendmsg
 => ____sys_sendmsg
 => ___sys_sendmsg
 => __sys_sendmmsg
 => __x64_sys_sendmmsg
 => do_syscall_64
 => entry_SYSCALL_64_after_hwframe
...
```

Similar tracing can be performed upon any visible functions within any kernel module! (just check for the function visibility via */sys/kernel/tracing/available_filter_functions*).

Getting the return value (kretprobe)

Use the 'r:<func-to-get-retval-f>' syntax with kprobe-perf

Eg. ret value of the kmalloc() !

```
$ sudo kprobe-perf -H -d.01 'r:__kmalloc ret=$retval'
Tracing kprobe __kmalloc for .01 seconds (buffered)...
# tracer: nop
#
# entries-in-buffer/entries-written: 238/238   #P:12
#
#                                _-----=> irqs-off/BH-disabled
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              ||| / _-=> migrate-disable
#                              |||| /     delay
#           TASK-PID     CPU#  |||||  TIMESTAMP  FUNCTION
#              | |         |   |||||     |         |
     kprobe-perf-30585   [011] ..... 89022.266432: __kmalloc: (security_prepare_creds+0x80/0xa0 <- __kmalloc) ret=0xffff99fb76840200
     kprobe-perf-30585   [011] ..... 89022.266436: __kmalloc: (security_prepare_creds+0x80/0xa0 <- __kmalloc) ret=0xffff99fb76840200
     kprobe-perf-30585   [011] ..... 89022.266438: __kmalloc: (security_prepare_creds+0x80/0xa0 <- __kmalloc) ret=0xffff99fb76840200
     kprobe-perf-30585   [011] ..... 89022.266440: __kmalloc: (security_prepare_creds+0x80/0xa0 <- __kmalloc) ret=0xffff99fb76840200
     kprobe-perf-30585   [011] ..... 89022.266448: __kmalloc: (security_prepare_creds+0x80/0xa0 <- __kmalloc) ret=0xffff99fb76840200
     kprobe-perf-30585   [011] ..... 89022.266450: __kmalloc: (security_task_alloc+0xa6/0x100 <- __kmalloc) ret=0xffff9a013934eec0
     kworker/3:2-29584   [003] ..... 89022.266456: __kmalloc: (acpi_ex_allocate_name_string+0x6e/0x140 <- __kmalloc) ret=0xffff99fb652ce290
     kworker/3:2-29584   [003] ..... 89022.266472: __kmalloc: (acpi_ns_get_normalized_pathname+0x7e/0x190 <- __kmalloc) ret=0xffff99fde2b7e50
     kworker/3:2-29584   [003] ..... 89022.266479: __kmalloc: (acpi_ns_get_normalized_pathname+0x7e/0x190 <- __kmalloc) ret=0xffff99fde2b7ea0
     kworker/3:2-29584   [003] ..... 89022.266480: __kmalloc: (acpi_ns_get_normalized_pathname+0x7e/0x190 <- __kmalloc) ret=0xffff99fde2b7ea0
     kworker/3:2-29584   [003] ..... 89022.266491: __kmalloc: (acpi_ex_allocate_name_string+0x6e/0x140 <- __kmalloc) ret=0xffff99fb652ce290
     kworker/3:2-29584   [003] ..... 89022.266513: __kmalloc: (acpi_ex_allocate_name_string+0x6e/0x140 <- __kmalloc) ret=0xffff99fb652ce290
     kworker/3:2-29584   [003] ..... 89022.266548: __kmalloc: (acpi_ex_allocate_name_string+0x6e/0x140 <- __kmalloc) ret=0xffff99fb652ce290
           sleep-30587   [008] ..... 89022.266598: __kmalloc: (security_prepare_creds+0x80/0xa0 <- __kmalloc) ret=0xffff9a00f234b7d0
           sleep-30587   [008] ..... 89022.266622: __kmalloc: (load_elf_phdrs+0x4e/0xc0 <- __kmalloc) ret=0xffff99fb584e1000
```

With stack

```
$ sudo kprobe-perf -Hs -d.01 'r:__kmalloc ret=$retval'
Tracing kprobe __kmalloc for .01 seconds (buffered)...

...
            sudo-844084  [003] d... 34696.475668: __kmalloc:
(tty_buffer_alloc+0x47/0x90 <- __kmalloc) ret=0xffff8909c2d5e000
            sudo-844084  [003] d... 34696.475668: <stack trace>
 => [unknown/kretprobe'd]
 => __tty_buffer_request_room
 => tty_insert_flip_string_fixed_flag
 => tty_insert_flip_string_and_push_buffer
 => pty_write
 => n_tty_write
 => file_tty_write.constprop.0
 => tty_write
 => new_sync_write
 => vfs_write
 => ksys_write
 => __x64_sys_write
 => do_syscall_64
 => entry_SYSCALL_64_after_hwframe

...
```

Another screenshot of the same:

```
tracing # kprobe-perf -Hs -d.01 'r:__kmalloc ret=$retval'
Tracing kprobe __kmalloc for .01 seconds (buffered)...
# tracer: nop
#
# entries-in-buffer/entries-written: 48/48   #P:6
#
#                               _-----=> irqs-off
#                              / _----=> need-resched
#                             | / _---=> hardirq/softirq
#                             || / _--=> preempt-depth
#                             ||| /     delay
#           TASK-PID     CPU#  ||||   TIMESTAMP  FUNCTION
#              | |         |   ||||      |         |
     kprobe-perf-1107984 [004] d... 34696.388376: __kmalloc: (security_prepare_creds+0x7a/0xa0 <- __kmalloc) ret=0xffff890985cf0a98
     kprobe-perf-1107984 [004] d... 34696.388379: <stack trace>
 => [unknown/kretprobe'd]
 => prepare_creds
 => do_faccessat
 => __x64_sys_access
 => do_syscall_64
 => entry_SYSCALL_64_after_hwframe
     kprobe-perf-1107984 [004] d... 34696.388383: __kmalloc: (security_prepare_creds+0x7a/0xa0 <- __kmalloc) ret=0xffff890985cf0a98
     kprobe-perf-1107984 [004] d... 34696.388383: <stack trace>
 => [unknown/kretprobe'd]
 => prepare_creds
 => do_faccessat
 => __x64_sys_access
 => do_syscall_64
 => entry_SYSCALL_64_after_hwframe
```

Tracing the 'exec' of processes system-wide

Useful... perhaps for an audit..

The syscall to trace is of course the execve(); it becomes the do_execve().

(From my LKD book:

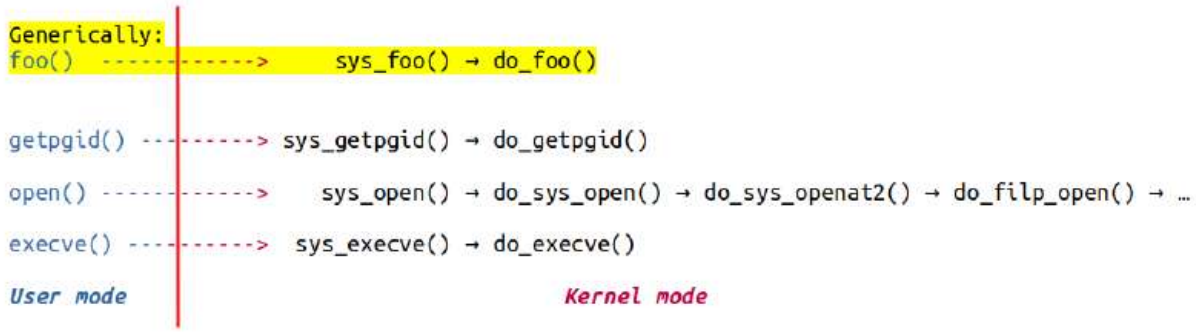


Figure 4.13 – How user mode system calls map within the kernel

).

BUT:
It just doesn't work with perf or kprobe-perf:

```
~ # execsnoop-perf -h
USAGE: execsnoop [-hrt] [-a argc] [-d secs] [name]
                 -d seconds      # trace duration, and use buffers
                 -a argc         # max args to show (default 8)
                 -r              # include re-execs
                 -t              # include time (seconds)
                 -h              # this usage message
                 name            # process name to match (REs allowed)
  eg,
       execsnoop                 # watch exec()s live (unbuffered)
       execsnoop -d 1            # trace 1 sec (buffered)
       execsnoop grep            # trace process names containing grep
       execsnoop 'udevd$'        # process names ending in "udevd"

See the man page and example file for more info.
~ #
~ # execsnoop-perf -d 1
Tracing exec()s for 1 seconds (buffered)...
ERROR: adding a kprobe for execve. Exiting.
~ #
```

So: let's use the modern **eBPF** framework! Via the excellent ***bpfcc-tools*** package.

"Check out the following link to a picture that opens your eyes to just how many powerful BCC/BPF tools are available to help trace different Linux subsystems and hardware: https://www.brendangregg.com/BPF/bcc_tracing_tools_early2019.png".

```
# dpkg -l|grep bpf
ii  bpfcc-tools                                  0.18.0+ds-2
all          tools for BPF Compiler Collection (BCC)
...
```

Once installed (on Ubuntu 22.04 LTS) here, a plethora of useful tooling's available:

```
~ # (cd /usr/sbin ; ls *-bpfcc)
argdist-bpfcc*       dcsnoop-bpfcc*        javagc-bpfcc*          perlflow-bpfcc*     shmsnoop-bpfcc*     tcprtt-bpfcc*
bashreadline-bpfcc*  dcstat-bpfcc*         javaobjnew-bpfcc*      perlstat-bpfcc*     slabratetop-bpfcc*  tcpstates-bpfcc*
bindsnoop-bpfcc*     deadlock-bpfcc*       javastat-bpfcc*        phpcalls-bpfcc*     sofdsnoop-bpfcc*    tcpsubnet-bpfcc*
biolatency-bpfcc*    dirtop-bpfcc*         javathreads-bpfcc*     phpflow-bpfcc*      softirqs-bpfcc*     tcpsynbl-bpfcc*
biolatpcts-bpfcc*    drsnoop-bpfcc*        killsnoop-bpfcc*       phpstat-bpfcc*      solisten-bpfcc*     tcptop-bpfcc*
biosnoop-bpfcc*      execsnoop-bpfcc*      klockstat-bpfcc*       pidpersec-bpfcc*    sslsniff-bpfcc*     tcptracer-bpfcc*
biotop-bpfcc*        exitsnoop-bpfcc*      llcstat-bpfcc*         profile-bpfcc*      stackcount-bpfcc*   threadsnoop-bpfcc*
bitesize-bpfcc*      ext4dist-bpfcc*       mdflush-bpfcc*         pythoncalls-bpfcc*  statsnoop-bpfcc*    tplist-bpfcc*
bpflist-bpfcc*       ext4slower-bpfcc*     memleak-bpfcc*         pythonflow-bpfcc*   swapin-bpfcc*       trace-bpfcc*
btrfsdist-bpfcc*     filelife-bpfcc*       mountsnoop-bpfcc*      pythongc-bpfcc*     syncsnoop-bpfcc*    ttysnoop-bpfcc*
btrfsslower-bpfcc*   fileslower-bpfcc*     mysqld_qslower-bpfcc*  pythonstat-bpfcc*   syscount-bpfcc*     vfscount-bpfcc*
cachestat-bpfcc*     filetop-bpfcc*        netqtop-bpfcc*         readahead-bpfcc*    tclcalls-bpfcc*     vfsstat-bpfcc*
cachetop-bpfcc*      funccount-bpfcc*      nfsdist-bpfcc*         reset-trace-bpfcc*  tclflow-bpfcc*      wakeuptime-bpfcc*
capable-bpfcc*       funcinterval-bpfcc*   nfsslower-bpfcc*      rubycalls-bpfcc*    tclobjnew-bpfcc*    xfsdist-bpfcc*
cobjnew-bpfcc*       funclatency-bpfcc*    nodegc-bpfcc*          rubyflow-bpfcc*     tclstat-bpfcc*      xfsslower-bpfcc*
compactsnoop-bpfcc*  funcslower-bpfcc*     nodestat-bpfcc*        rubygc-bpfcc*       tcpaccept-bpfcc*    zfsdist-bpfcc*
cpudist-bpfcc*       gethostlatency-bpfcc* offcputime-bpfcc*      rubyobjnew-bpfcc*   tcpconnect-bpfcc*   zfsslower-bpfcc*
cpuunclaimed-bpfcc*  hardirqs-bpfcc*       offwaketime-bpfcc*     rubystat-bpfcc*     tcpconnlat-bpfcc*
criticalstat-bpfcc*  inject-bpfcc*         oomkill-bpfcc*         runqlat-bpfcc*      tcpdrop-bpfcc*
dbslower-bpfcc*      javacalls-bpfcc*      opensnoop-bpfcc*       runqlen-bpfcc*      tcplife-bpfcc*
dbstat-bpfcc*        javaflow-bpfcc*       perlcalls-bpfcc*       runqslower-bpfcc*   tcpretrans-bpfcc*
~ #
```

CAUTION! All may not work on all platforms... test and see.

Let's try the exec one – execsnoop-bpfcc:

```
# execsnoop-bpfcc -h
usage: execsnoop-bpfcc [-h] [-T] [-t] [-x] [--cgroupmap CGROUPMAP] [--mntnsmap
MNTNSMAP] [-u USER] [-q] [-n NAME] [-l LINE]
                       [-U] [--max-args MAX_ARGS]

Trace exec() syscalls

options:
  -h, --help            show this help message and exit
  -T, --time            include time column on output (HH:MM:SS)
  -t, --timestamp       include timestamp on output
  -x, --fails           include failed exec()s
  --cgroupmap CGROUPMAP
                        trace cgroups in this BPF map only
  --mntnsmap MNTNSMAP   trace mount namespaces in this BPF map only
  -u USER, --uid USER   trace this UID only
  -q, --quote           Add quotemarks (") around arguments.
  -n NAME, --name NAME  only print commands matching this name (regex), any arg
  -l LINE, --line LINE  only print commands where arg contains this line (regex)
  -U, --print-uid       print UID column
  --max-args MAX_ARGS   maximum number of arguments parsed and displayed,
defaults to 20

examples:
    ./execsnoop           # trace all exec() syscalls
    ./execsnoop -x        # include failed exec()s
    ./execsnoop -T        # include time (HH:MM:SS)
    ./execsnoop -U        # include UID
    ./execsnoop -u 1000   # only trace UID 1000
    ./execsnoop -u user   # get user UID and trace only them
    ./execsnoop -t        # include timestamps
    ./execsnoop -q        # add "quotemarks" around arguments
    ./execsnoop -n main   # only print command lines containing "main"
    ./execsnoop -l tpkg   # only print command where arguments contains "tpkg"
    ./execsnoop --cgroupmap mappath  # only trace cgroups in this BPF map
    ./execsnoop --mntnsmap mappath   # only trace mount namespaces in the map
~ #
```

Example run:

```
# execsnoop-bpfcc -x 2>/dev/null
PCOMM            PID    PPID   RET ARGS
snap             1141378 1226     0 /usr/bin/snap run snapd-desktop-integration
snap             1141378 1226     0 /snap/snapd/current/usr/bin/snap run snapd-
desktop-integration
snap-seccomp     1141390 1141378   0 /snap/snapd/18357/usr/lib/snapd/snap-
seccomp version-info
snap-confine     1141378 1226     0 /snap/snapd/18357/usr/lib/snapd/snap-confine
--base core20 snap.snapd-desktop-integration.snapd-desktop-integration
/usr/lib/snapd/snap-exec snapd-desktop-integration
```

```
snap-exec        1141378 1226     0 /usr/lib/snapd/snap-exec snapd-desktop-
integration
snapcraft-runne  1141378 1226     0
/snap/snapd-desktop-integration/49/snap/command-chain/snapcraft-runner
/snap/snapd-desktop-integration/49/snap/command-chain/desktop-launch
/snap/snapd-desktop-integration/49/usr/bin/snapd-desktop-integration
desktop-launch   1141378 1226     0
/snap/snapd-desktop-integration/49/snap/command-chain/desktop-launch
/snap/snapd-desktop-integration/49/usr/bin/snapd-desktop-integration
date             1141411 1141378   0 /usr/bin/date +%s.%N
getent           1141413 1141412   0 /usr/bin/getent passwd 1000
cut              1141414 1141412   0 /usr/bin/cut -d : -f 6
chmod            1141415 1141378   0 /usr/bin/chmod 700
/home/osboxes/snap/snapd-desktop-integration/49/.config
md5sum           1141417 1141416   0 /usr/bin/md5sum
cat              1141418 1141378   0 /usr/bin/cat /home/osboxes/snap/snapd-
desktop-integration/49/.config/user-dirs.dirs.md5sum
...
...
```

Fantastic.

<< OLDER >>

Specialized Probes

Also, there are two special types of kprobes – **jprobes** and **kretprobes**.

<<
NOTE! NOTE! NOTE!

From kernel ver 4.15 (rel 28 Jan 2018), the Jprobes infrastructure has been REMOVED from the kernel!

The commit says:
Disable the jprobes APIs and comment out the jprobes API function code. This is in preparation of removing all jprobes related code (including kprobe's break_handler).

Nowadays ftrace and other tracing features are mature enough to replace jprobes use-cases. Users can safely use ftrace and perf probe etc. for their use cases.

LWN : kprobes: Abolish jprobe APIs, 06 Oct 2017.
>>

Jprobes (jumper probes) are used when the point of interest is the *entry* to a function. You need to have the probe handler conform to the signature of the target function. This gives one immediate access to the function arguments (without any complicated and arch-dependant stack manipulation).

A **return probe** (or kretprobe) is a mechanism that works well when you need to trap a function's *return* point. With normal kprobes, you'd need a kprobe installed at every single point of return of the target function (which could be plenty); with a return probe, one is sufficient.
--snip--
...

SystemTap

Source

SystemTap provides free software (GPL) infrastructure to simplify the gathering of information about the running Linux system. This assists diagnosis of a performance or functional problem. SystemTap eliminates the need for the developer to go through the tedious and disruptive instrument, recompile, install, and reboot sequence that may be otherwise required to collect data.

SystemTap provides a simple command line interface and scripting language for writing

instrumentation for a live running kernel *plus* user-space applications. We are publishing samples, as well as enlarging the internal "tapset" script library to aid reuse and abstraction.

Among other tracing/probing tools, SystemTap is the tool of choice for complex tasks that may require live analysis, programmable on-line response, and whole-system symbolic access. SystemTap can also handle simple tracing jobs.

...

Documentation including excellent 'stap' script examples

LWN – A SystemTap Update

"... SystemTap allows users to write and reuse simple scripts to deeply examine the activities of a running Linux system. These scripts can be designed to extract data, filter it, and summarize it quickly (and safely), enabling the diagnosis of complex performance (or even functional) problems.

The essential idea behind a SystemTap script is to name *events*, and to give them *handlers*. When SystemTap runs the script, SystemTap monitors for the event; once the event occurs, the Linux kernel then runs the handler as a quick sub-routine, then resumes.

There are several kind of events; entering/exiting a function, timer expiration, session termination, etc. A handler is a series of script language statements that specify the work to be done whenever the event occurs. This work normally includes extracting data from the event context, storing them into internal variables, and printing results.
..."

A few examples of "BEST" stap scripts:

...

- general/helloworld.stp - SystemTap "Hello World" Program
  keywords: _BEST SIMPLE

A basic "Hello World" program implemented in SystemTap script. It prints out "hello world" message and then immediately exits.

```
# stap helloworld.stp
```

- general/para-callgraph.stp - Callgraph Tracing with Arguments
  keywords: _BEST TRACE CALLGRAPH

Print a timed per-thread microsecond-timed callgraph, complete with function parameters and return values. The first parameter names the function probe points to trace. The optional second parameter names the probe points for trigger functions, which acts to enable tracing for only those functions that occur while the current thread is nested within the trigger.

sample usage in general/para-callgraph.txt

- general/varwatch.stp - Watch a Variable Changing Value in a Thread
  keywords: _BEST MONITORING

This script places a set of probes (specified by $1), each of which monitors the state of some context $variable expression (specified by $2). Whenever the value changes, with respect to the active thread, the event is traced.

sample usage in general/varwatch.txt

[...]

Source

History
SystemTap debuted in 2005 in Red Hat Enterprise Linux 4 Update 2 as a technology preview.[2]

After four years in development, SystemTap 1.0 was released in 2009.[3]

As of 2011 SystemTap runs fully supported in all Linux distributions including RHEL / CentOS 5[4] since update 2, SLES 10,[5] Fedora, Debian and Ubuntu.

Tracepoints in the CPython VM and JVM were added in SystemTap 1.2.[6]
...

MUST-READ
Julia Evans' :: Linux tracing systems & how they fit together

Uprobes

Dynamic tracing in Linux user and kernel space
Brendan Gregg's uprobe utility

Brendan Gregg's uprobe[-perf]

- wrapper over the kernel's uprobe-events subsystem
- a wrapper bash script: was earlier just called 'uprobe', is now called 'uprobe-perf'
- will create, trace, then destroy a given uprobe definition
- see *Documentation/trace/uprobetracer.txt* for details, syntax
- *From the man page uprobe-perf(1)*
  ...

```
 WARNING:  This  uses  dynamic tracing of user-level functions, using some relatively new
kernel code. I have seen this cause target processes to fail, either entering endless spin
```

```
loops or crashing on illegal instructions.  I believe newer kernels (post 4.0) are
relatively safer, but use caution. Test in a lab environment, and know what  you  are
doing,  before  use.  Also  consider  other  (more  developed)  user-level  tracers
(perf_events, LTTng, etc.).
...

REQUIREMENTS:  FTRACE  and  UPROBE  CONFIG,  which  you may already have on recent kernel
versions, file(1), ldconfig(8), objdump(1), and some version of awk.  Also, currently only
executes on Linux 4.0+ (see WARNING) unless -F is used.

...
  -s     Print user-level stack traces after each event. These are currently printed in
hex,  and  need  post processing to see user-level symbols (eg, addr2line; I should
automate that).

       -v     Show  the  uprobe format file only (do not trace), identifying possible
variables for use in a custom filter.

       -p PID Only trace user-level functions when this process ID is on-CPU.
...

EXAMPLES
       These examples may need modification to match your target software function names
and  platform's  register usage.  If  using  platform  specific  registers becomes too
painful in practice, consider a debuginfo-based        tracer, which can trace variables
names instead (eg, perf_events).

       trace readline() calls in all running "bash" executables:
              # uprobe p:bash:readline
<< p = set a uprobe ; r = set a return uprobe >>

       trace readline() with explicit executable path:
              # uprobe p:/bin/bash:readline

       trace the return of readline() with return value as a string: << whoa! >>
              # uprobe 'r:bash:readline +0($retval):string'

       trace sleep() calls in all running libc shared libraries:
              # uprobe p:libc:sleep

       trace sleep() with register %di (x86):
              # uprobe 'p:libc:sleep %di'

       trace this address (use caution: must be instruction aligned):
              # uprobe p:libc:0xbf130

       trace gettimeofday() for PID 1182 only:
              # uprobe -p 1182 p:libc:gettimeofday

       trace the return of fopen() only when it returns NULL:
              # uprobe 'r:libc:fopen file=$retval' 'file == 0'

FIELDS
       The output format depends on the kernel version, and headings can be printed using -
H.  The  format  is  the same  as  the  ftrace  function  trace  format,  described
in  the  kernel  source  under  Documenta-
       tion/trace/ftrace.txt.

       Typical fields are:

       TASK-PID
              The process name (which could include dashes), a dash, and the process ID.

       CPU#   The CPU ID, in brackets.
```

```
       ||||   Kernel state flags. For example, on Linux 3.16 these are for irqs-off, need-
resched, hardirq/softirq, and preempt-depth.

       TIMESTAMP
              Time of event, in seconds.

       FUNCTION
              User-level function name.

OVERHEAD
       This  can  generate  a lot of trace data quickly, depending on the frequency of the
traced events. Such data will cause performance overheads.  This also works without
buffering by default, printing function events as
they  happen (uses trace_pipe), context switching and consuming CPU to do so. If needed,
you can try the "-d secs" option, which buffers events instead, reducing overhead. If you
think  the  buffer  option  is  losing events, try increasing the buffer size
(buffer_size_kb).

       If  you  find  a  use  for  uprobe(8)  where  the  overhead is prohibitive, consider
the same enabling using perf_events where overhead should be reduced.

SOURCE
       This is from the perf-tools collection:

              https://github.com/brendangregg/perf-tools

 Also look under the examples directory for a text file containing example usage, output,
and commentary  for this tool.

STABILITY
       Unstable - in development.

AUTHOR
       Brendan Gregg

SEE ALSO
       kprobe(8)
```