Kprobe-based Event Tracing

Author: Masami Hiramatsu

Overview

These events are similar to tracepoint based events. Instead of Tracepoint, this is based on kprobes (kprobe and kretprobe). So it can probe wherever kprobes can probe (this means, all functions except those with __kprobes/nokprobe_inline annotation and those marked NOKPROBE_SYMBOL). Unlike the Tracepoint based event, this can be added and removed dynamically, on the fly.

To enable this feature, build your kernel with CONFIG_KPROBE_EVENTS=y.

Similar to the events tracer, this doesn't need to be activated via current_tracer. Instead of that, add probe points via /sys/kernel/debug/tracing/kprobe_events, and enable it via /sys/kernel/debug/tracing/events/kprobes/<EVENT>/enable.

You can also use /sys/kernel/debug/tracing/dynamic_events instead of kprobe_events. That interface will provide unified access to other dynamic events too.

Synopsis of kprobe_events

```
 p[:[GRP/]EVENT] [MOD:]SYM[+offs]|MEMADDR [FETCHARGS]  : Set a probe
 r[MAXACTIVE][:[GRP/]EVENT] [MOD:]SYM[+0] [FETCHARGS]  : Set a return probe
 p:[GRP/]EVENT] [MOD:]SYM[+0]%return [FETCHARGS]       : Set a return probe
 -:[GRP/]EVENT                                         : Clear a probe

GRP            : Group name. If omitted, use "kprobes" for it.
EVENT          : Event name. If omitted, the event name is generated
                 based on SYM+offs or MEMADDR.
MOD            : Module name which has given SYM.
SYM[+offs]     : Symbol+offset where the probe is inserted.
SYM%return     : Return address of the symbol
MEMADDR        : Address where the probe is inserted.
MAXACTIVE      : Maximum number of instances of the specified function that
                 can be probed simultaneously, or 0 for the default value
                 as defined in Documentation/trace/kprobes.rst section 1.3.1.

FETCHARGS      : Arguments. Each probe can have up to 128 args.
 %REG          : Fetch register REG
 @ADDR         : Fetch memory at ADDR (ADDR should be in kernel)
 @SYM[+|-offs] : Fetch memory at SYM +|- offs (SYM should be a data symbol)
 $stackN       : Fetch Nth entry of stack (N >= 0)
 $stack        : Fetch stack address.
 $argN         : Fetch the Nth function argument. (N >= 1) (\*1)
 $retval       : Fetch return value.(\*2)
 $comm         : Fetch current task comm.
 +|-[u]OFFS(FETCHARG) : Fetch memory at FETCHARG +|- OFFS address.(\*3)(\*4)
 \IMM          : Store an immediate value to the argument.
 NAME=FETCHARG : Set NAME as the argument name of FETCHARG.
 FETCHARG:TYPE : Set TYPE as the type of FETCHARG. Currently, basic types
                 (u8/u16/u32/u64/s8/s16/s32/s64), hexadecimal types
                 (x8/x16/x32/x64), "string", "ustring" and bitfield
                 are supported.

 (\*1) only for the probe on function entry (offs == 0).
 (\*2) only for return probe.
 (\*3) this is useful for fetching a field of data structures.
 (\*4) "u" means user-space dereference. See :ref:`user_mem_access`.
```

Types

Several types are supported for fetch-args. Kprobe tracer will access memory by given type. Prefix 's' and 'u' means those types are signed and unsigned respectively. 'x' prefix implies it is unsigned. Traced arguments are shown in decimal ('s' and 'u') or hexadecimal ('x'). Without type casting, 'x32' or 'x64' is used depends on the architecture (e.g. x86-32 uses x32, and x86-64 uses x64). These value types can be an array. To record array data, you can add '[N]' (where N is a fixed number, less than 64) to the base type. E.g. 'x16[4]' means an array of x16 (2bytes hex) with 4 elements. Note that the array can be applied to memory type fetchargs, you can not apply it to registers/stack-entries etc. (for example, '$stack1:x8[8]' is wrong, but '+8($stack):x8[8]' is OK.) String type is a special type, which fetches a "null-terminated" string from kernel space. This means it will fail and store NULL if the string container has been paged out. "ustring" type is an alternative of string for user-space. See :ref:`user_mem_access` for more info.. The string array type is a bit different from other types. For other base types, <base-type>[1] is equal to <base-type> (e.g. +0(%di):x32[1] is same as +0(%di):x32.) But string[1] is not equal to string. The string type itself represents "char array", but string array type represents "char * array". So, for example, +0(%di):string[1] is equal to +0(+0(%di)):string. Bitfield is another special type, which takes 3 parameters, bit-width, bit- offset, and container-size (usually 32). The syntax is:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\trace\[linux-master][Documentation][trace]kprobetrace.rst`, **line 71**); *backlink*
>
> Unknown interpreted text role "ref".

```
b<bit-width>@<bit-offset>/<container-size>
```

Symbol type('symbol') is an alias of u32 or u64 type (depends on BITS_PER_LONG) which shows given pointer in "symbol+offset" style. For $comm, the default type is "string"; any other type is invalid.

User Memory Access

Kprobe events supports user-space memory access. For that purpose, you can use either user-space dereference syntax or 'ustring' type.

The user-space dereference syntax allows you to access a field of a data structure in user-space. This is done by adding the "u" prefix to the dereference syntax. For example, +u4(%si) means it will read memory from the address in the register %si offset by 4, and the memory is expected to be in user-space. You can use this for strings too, e.g. +u0(%si):string will read a string from the address in the register %si that is expected to be in user- space. 'ustring' is a shortcut way of performing the same task. That is, +0(%si):ustring is equivalent to +u0(%si):string.

Note that kprobe-event provides the user-memory access syntax but it doesn't use it transparently. This means if you use normal dereference or string type for user memory, it might fail, and may always fail on some archs. The user has to carefully check if the target data is in kernel or user space.

Per-Probe Event Filtering

Per-probe event filtering feature allows you to set different filter on each probe and gives you what arguments will be shown in trace buffer. If an event name is specified right after 'p:' or 'r:' in kprobe_events, it adds an event under tracing/events/kprobes/<EVENT>, at the directory you can see 'id', 'enable', 'format', 'filter' and 'trigger'.

enable:
: You can enable/disable the probe by writing 1 or 0 on it.

format:
: This shows the format of this probe event.

filter:
: You can write filtering rules of this event.

id:
: This shows the id of this probe event.

trigger:
: This allows to install trigger commands which are executed when the event is hit (for details, see Documentation/trace/events.rst, section 6).

Event Profiling

You can check the total number of probe hits and probe miss-hits via /sys/kernel/debug/tracing/kprobe_profile. The first column is event name, the second is the number of probe hits, the third is the number of probe miss-hits.

Kernel Boot Parameter

You can add and enable new kprobe events when booting up the kernel by "kprobe_event=" parameter. The parameter accepts a semicolon-delimited kprobe events, which format is similar to the kprobe_events. The difference is that the probe definition parameters are comma-delimited instead of space. For example, adding myprobe event on do_sys_open like below

```
p:myprobe do_sys_open dfd=%ax filename=%dx flags=%cx mode=+4($stack)
```

should be below for kernel boot parameter (just replace spaces with comma)

```
p:myprobe,do_sys_open,dfd=%ax,filename=%dx,flags=%cx,mode=+4($stack)
```

Usage examples

To add a probe as a new event, write a new definition to kprobe_events as below:

```
echo 'p:myprobe do_sys_open dfd=%ax filename=%dx flags=%cx mode=+4($stack)' > /sys/kernel/debug/tracing/kprobe_eve
```

This sets a kprobe on the top of do_sys_open() function with recording 1st to 4th arguments as "myprobe" event. Note, which register/stack entry is assigned to each function argument depends on arch-specific ABI. If you unsure the ABI, please try to use probe subcommand of perf-tools (you can find it under tools/perf/). As this example shows, users can choose more familiar names for each arguments.

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

This sets a kretprobe on the return point of do_sys_open() function with recording return value as "myretprobe" event. You can see the format of these events via /sys/kernel/debug/tracing/events/kprobes/<EVENT>/format.

```
cat /sys/kernel/debug/tracing/events/kprobes/myprobe/format
name: myprobe
ID: 780
format:
        field:unsigned short common_type;       offset:0;       size:2; signed:0;
        field:unsigned char common_flags;       offset:2;       size:1; signed:0;
        field:unsigned char common_preempt_count;       offset:3; size:1;signed:0;
        field:int common_pid;   offset:4;       size:4; signed:1;

        field:unsigned long __probe_ip; offset:12;      size:4; signed:0;
        field:int __probe_nargs;        offset:16;      size:4; signed:1;
        field:unsigned long dfd;        offset:20;      size:4; signed:0;
        field:unsigned long filename;   offset:24;      size:4; signed:0;
        field:unsigned long flags;      offset:28;      size:4; signed:0;
        field:unsigned long mode;       offset:32;      size:4; signed:0;


print fmt: "(%lx) dfd=%lx filename=%lx flags=%lx mode=%lx", REC->__probe_ip,
REC->dfd, REC->filename, REC->flags, REC->mode
```

You can see that the event has 4 arguments as in the expressions you specified.

```
echo > /sys/kernel/debug/tracing/kprobe_events
```

This clears all probe points.

Or,

```
echo -:myprobe >> kprobe_events
```

This clears probe points selectively.

Right after definition, each event is disabled by default. For tracing these events, you need to enable it.

```
echo 1 > /sys/kernel/debug/tracing/events/kprobes/myprobe/enable
echo 1 > /sys/kernel/debug/tracing/events/kprobes/myretprobe/enable
```

Use the following command to start tracing in an interval.

```
# echo 1 > tracing_on
Open something...
# echo 0 > tracing_on
```

And you can see the traced information via /sys/kernel/debug/tracing/trace.

```
cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
           <...>-1447  [001] 1038282.286875: myprobe: (do_sys_open+0x0/0xd6) dfd=3 filename=7fffd1ec4440 flags=800
           <...>-1447  [001] 1038282.286878: myretprobe: (sys_openat+0xc/0xe <- do_sys_open) $retval=fffffffffffff
           <...>-1447  [001] 1038282.286885: myprobe: (do_sys_open+0x0/0xd6) dfd=ffffff9c filename=40413c flags=80
           <...>-1447  [001] 1038282.286915: myretprobe: (sys_open+0x1b/0x1d <- do_sys_open) $retval=3
           <...>-1447  [001] 1038282.286969: myprobe: (do_sys_open+0x0/0xd6) dfd=ffffff9c filename=4041c6 flags=98
           <...>-1447  [001] 1038282.286976: myretprobe: (sys_open+0x1b/0x1d <- do_sys_open) $retval=3
```

Each line shows when the kernel hits an event, and <- SYMBOL means kernel returns from SYMBOL(e.g. "sys_open+0x1b/0x1d <- do_sys_open" means kernel returns from do_sys_open to sys_open+0x1b).