Kprobes and SystemTap

October, 2017, Tokyo

Joey Lee
SUSE Labs Taipei

Agenda

- Kprobes
- Kprobe-based Event Tracing
- SystemTap
- Q&A

Kprobes

Concepts

- Kprobes enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively.
- You can trap at almost any kernel code address(*), specifying a handler routine to be invoked when the breakpoint is hit. [1]

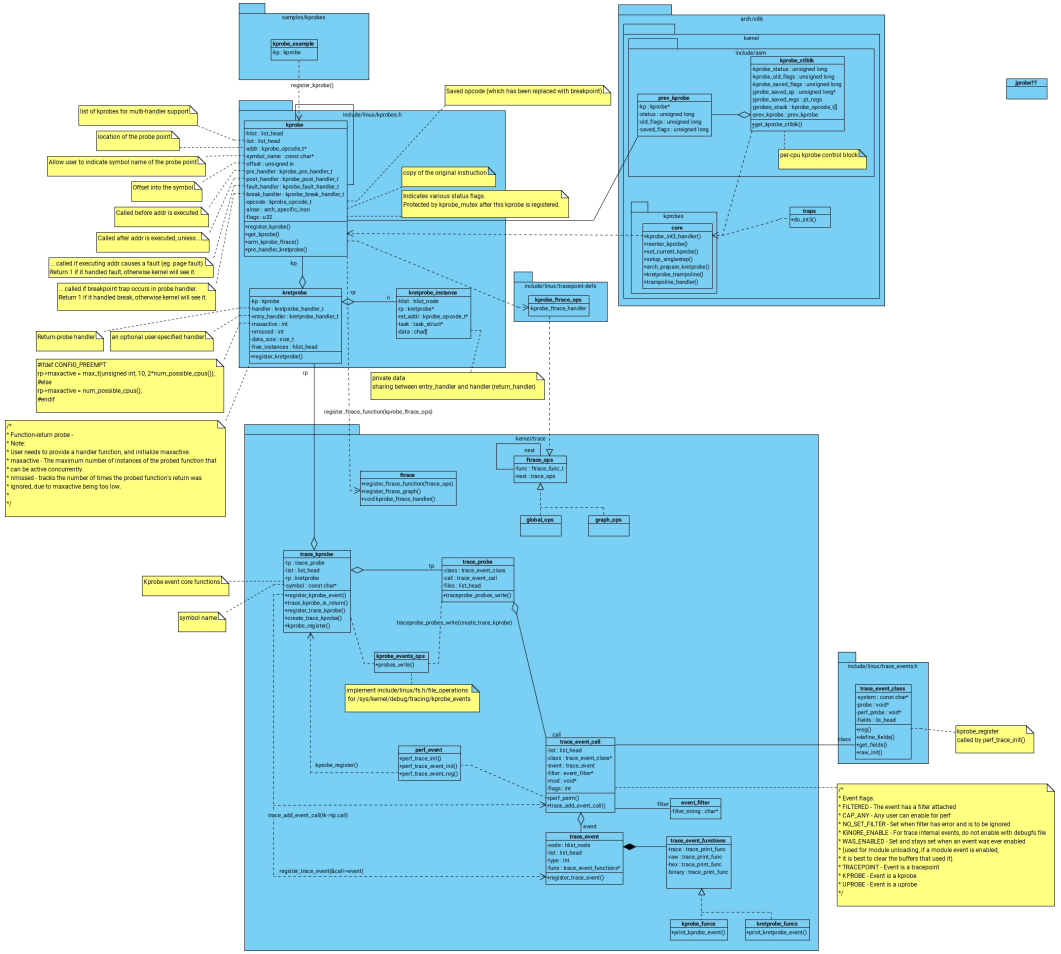
arch/x86/kernel/traps.c

```
/* May run on IST stack. */
dotraplinkage void notrace do_int3(struct pt_regs *regs, long error_code)
{
#ifdef CONFIG_DYNAMIC_FTRACE
        /*
         * ftrace must be first, everything else may cause a recursive crash.
         * See note by declaration of modifying_ftrace_code in ftrace.c
         */
        if (unlikely(atomic_read(&modifying_ftrace_code)) &&
            ftrace_int3_handler(regs))
                return;
#endif
        if (poke_int3_handler(regs))
                return;

        ist_enter(regs);
        RCU_LOCKDEP_WARN(!rcu_is_watching(), "entry code didn't wake RCU");
#ifdef CONFIG_KGDB_LOW_LEVEL_TRAP
        if (kgdb_ll_trap(DIE_INT3, "int3", regs, error_code, X86_TRAP_BP,
                                SIGTRAP) == NOTIFY_STOP)
                goto exit;
#endif /* CONFIG_KGDB_LOW_LEVEL_TRAP */

#ifdef CONFIG_KPROBES
        if (kprobe_int3_handler(regs))
                goto exit;
#endif
```

Conceptual model for Kprobe



Configuring Kprobes

- Kprobes
  - CONFIG_KPROBES=y
- Load and unload Kprobes-base instrumentation modules
  - CONFIG_MODULES=y
  - CONFIG_MODULE_UNLOAD=y
- For kallsyms_lookup_name()
  - CONFIG_KALLSYMS=y
  - CONFIG_KALLSYMS_ALL=y
- Compile the kernel with debug info for “objdump -d -l vmlinux”
  - CONFIG_DEBUG_INFO

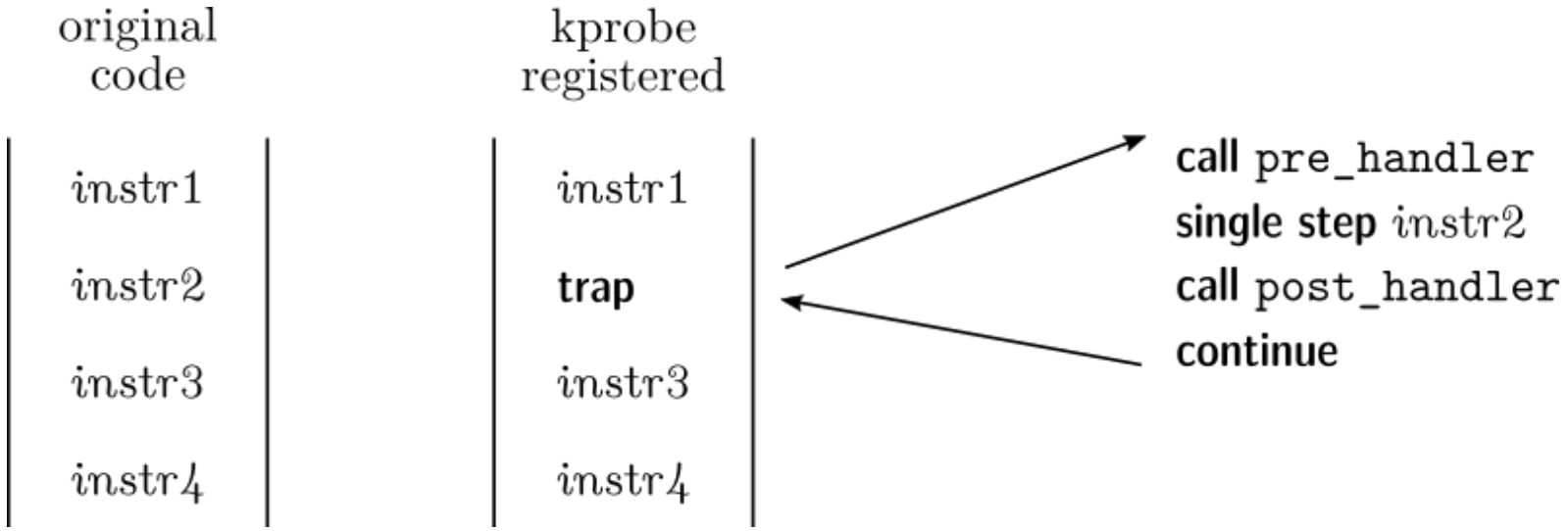
kprobes, jprobes, and kretprobes

- A kprobe can be inserted on virtually any instruction in the kernel.
- A jprobe is inserted at the entry to a kernel function, and provides convenient access to the function's arguments.
- A return probe (kretprobes) fires when a specified function returns. [1]

How Does a Kprobe Work

- When a kprobe is registered, Kprobes makes a copy of the probed instruction and replaces the first byte(s) of the probed instruction with a breakpoint instruction (e.g., int3 on i386 and x86_64).
- When a CPU hits the breakpoint instruction, a trap occurs, the CPU's registers are saved, and control passes to Kprobes via the notifier_call_chain mechanism. Kprobes executes the "pre_handler" associated with the kprobe, passing the handler the addresses of the kprobe struct and the saved registers.
- Next, Kprobes single-steps its copy of the probed instruction. (It would be simpler to single-step the actual instruction in place, but then Kprobes would have to temporarily remove the breakpoint instruction. This would open a small time window when another CPU could sail right past the probepoint.)
- After the instruction is single-stepped, Kprobes executes the "post_handler," if any, that is associated with the kprobe. Execution then continues with the instruction following the probepoint. [1]

How Does a Kprobe Work (cont.)



* trap is interrupt 3 (aka int3).

[2] How Linux kprobes works

int register_kprobe(struct kprobe *kp)

- Kprobe:
  - 1) Sets a breakpoint at the address kp→addr.
  - 2) When the breakpoint is hit, Kprobes calls kp->pre_handler.
  - 3) probed instruction is single-stepped
  - 4) Kprobe calls kp→post_handler
- If a fault occurs during execution of kp->pre_handler or kp→post_handler or during single-stepping of the probed instruction:
  - Kprobes calls kp→fault_handler
- Any or all handlers can be NULL. [1]

samples/kprobes/kprobe_example.c

- CONFIG_SAMPLE_KPROBES = m
- # insmod kprobe_example.ko

kprobe_example.ko (dmesg)

```
[329331.351215] systemd-journald[12121]: Sent WATCHDOG=1 notification.
[329381.125312] Planted kprobe at ffffffff8107e110
[329382.810325] pre_handler: p->addr = 0xffffffff8107e110, ip = ffffffff8107e111, flags = 0x246
[329382.813400] post_handler: p->addr = 0xffffffff8107e110, flags = 0x246
[329384.690772] pre_handler: p->addr = 0xffffffff8107e110, ip = ffffffff8107e111, flags = 0x246
[329384.693405] post_handler: p->addr = 0xffffffff8107e110, flags = 0x246
[329384.695544] systemd-journald[12121]: Sent WATCHDOG=1 notification.
[329388.402773] pre_handler: p->addr = 0xffffffff8107e110, ip = ffffffff8107e111, flags = 0x246
[329388.406049] post_handler: p->addr = 0xffffffff8107e110, flags = 0x246
[329388.409093] pre_handler: p->addr = 0xffffffff8107e110, ip = ffffffff8107e111, flags = 0x246
```

kprobe_example.c

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>

/* For each probe you need to allocate a kprobe structure */
static struct kprobe kp = {
        .symbol_name    = "_do_fork",
};

/* kprobe pre_handler: called just before the probed instruction is executed */
static int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
#ifdef CONFIG_X86
        printk(KERN_INFO "pre_handler: p->addr = 0x%p, ip = %lx,"
                        " flags = 0x%lx\n",
                p->addr, regs->ip, regs->flags);
#endif
#ifdef CONFIG_PPC
        printk(KERN_INFO "pre_handler: p->addr = 0x%p, nip = 0x%lx,"
                        " msr = 0x%lx\n",
                p->addr, regs->nip, regs->msr);
#endif
#ifdef CONFIG_MIPS
        printk(KERN_INFO "pre_handler: p->addr = 0x%p, epc = 0x%lx,"
                        " status = 0x%lx\n",
                p->addr, regs->cp0_epc, regs->cp0_status);
#endif
```

kprobe_example.c (cont.)

```c
/*
 * fault_handler: this is called if an exception is generated for any
 * instruction within the pre- or post-handler, or when Kprobes
 * single-steps the probed instruction.
 */
static int handler_fault(struct kprobe *p, struct pt_regs *regs, int trapnr)
{
        printk(KERN_INFO "fault_handler: p->addr = 0x%p, trap #%dn",
                p->addr, trapnr);
        /* Return 0 because we don't handle the fault. */
        return 0;
}

static int __init kprobe_init(void)
{
        int ret;
        kp.pre_handler = handler_pre;
        kp.post_handler = handler_post;
        kp.fault_handler = handler_fault;

        ret = register_kprobe(&kp);
        if (ret < 0) {
                printk(KERN_INFO "register_kprobe failed, returned %d\n", ret);
                return ret;
        }
        printk(KERN_INFO "Planted kprobe at %p\n", kp.addr);
        return 0;
}

static void __exit kprobe_exit(void)
{
        unregister_kprobe(&kp);
```

pt_regs (i386)

arch/x86/include/asm/ptrace.h

```
#ifdef __i386__

struct pt_regs {
        unsigned long bx;
        unsigned long cx;
        unsigned long dx;
        unsigned long si;
        unsigned long di;
        unsigned long bp;
        unsigned long ax;
        unsigned long ds;
        unsigned long es;
        unsigned long fs;
        unsigned long gs;
        unsigned long orig_ax;
        unsigned long ip;
        unsigned long cs;
        unsigned long flags;
        unsigned long sp;
        unsigned long ss;
};
```

pt_regs (x86_64)

arch/x86/include/asm/ptrace.h

```
#else /* __i386__ */

struct pt_regs {
/*
 * C ABI says these regs are callee-preserved. They aren't saved on kernel entry
 * unless syscall needs a complete, fully filled "struct pt_regs".
 */
        unsigned long r15;
        unsigned long r14;
        unsigned long r13;
        unsigned long r12;
        unsigned long bp;
        unsigned long bx;
/* These regs are callee-clobbered. Always saved on kernel entry. */
        unsigned long r11;
        unsigned long r10;
        unsigned long r9;
        unsigned long r8;
        unsigned long ax;
        unsigned long cx;
        unsigned long dx;
        unsigned long si;
        unsigned long di;
/*
 * On syscall entry, this is syscall#. On CPU exception, this is error code.
 * On hw interrupt, it's IRQ number:
 */
        unsigned long orig_ax;
/* Return frame for iretq */
        unsigned long ip;
        unsigned long cs;
        unsigned long flags;
        unsigned long sp;
        unsigned long ss;
/* top of stack page */
};
```

kernel/fork.c

```
/*
 *  Ok, this is the main fork-routine.
 *
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 */
long _do_fork(unsigned long clone_flags,
              unsigned long stack_start,
              unsigned long stack_size,
              int __user *parent_tidptr,
              int __user *child_tidptr,
              unsigned long tls)
{
        struct task_struct *p;
        int trace = 0;
        long nr;

        /*
         * Determine whether and which event to report to ptracer.  When
         * called from kernel_thread or CLONE_UNTRACED is explicitly
         * requested, no event is reported; otherwise, report if the event
         * for the type of forking is enabled.
         */
        if (!(clone_flags & CLONE_UNTRACED)) {
```

System V ABI – X86_64:

RDI: clone_flags
RSI: stack_start
RDX: stack_size
RCX: parent_tidptr
...

v4.4

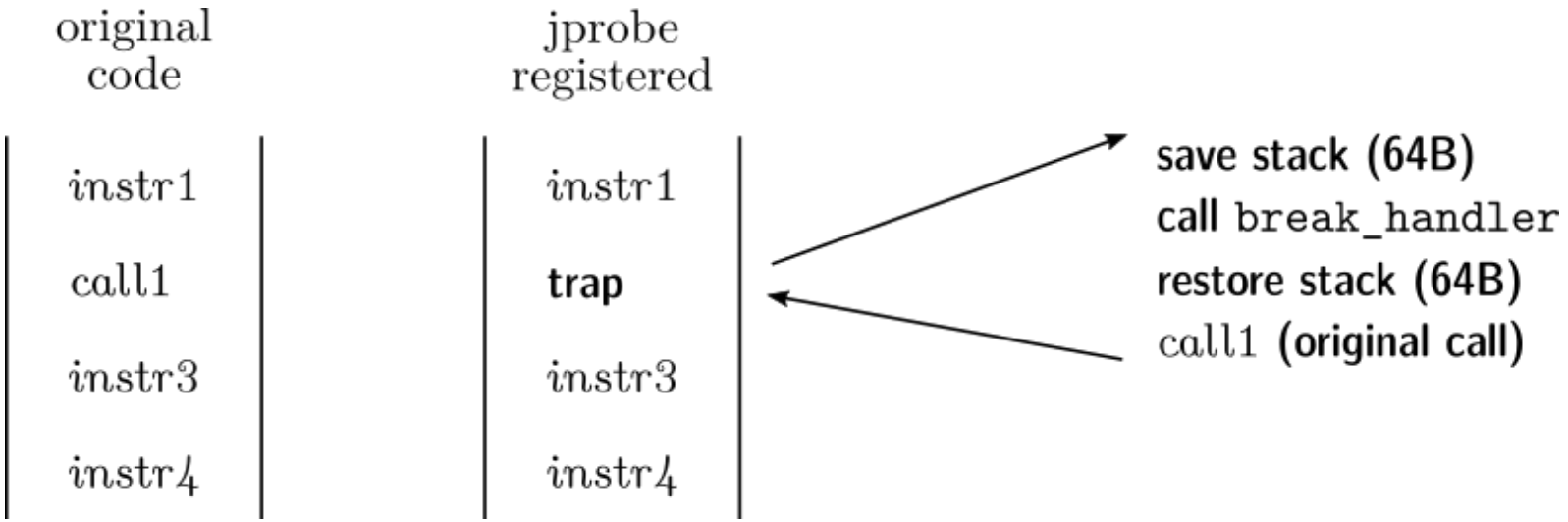
How Does a Jprobe Work?

- A jprobe is implemented using a kprobe that is placed on a function's entry point. It employs a simple mirroring principle to allow seamless access to the probed function's arguments. The jprobe handler routine should have the same signature (arg list and return type) as the function being probed, and must always end by calling the Kprobes function jprobe_return().
- When the probe is hit, Kprobes makes a copy of the saved registers and a generous portion of the stack (see below). Kprobes then points the saved instruction pointer at the jprobe's handler routine, and returns from the trap. As a result, control passes to the handler, which is presented with the same register and stack contents as the probed function. When it is done, the handler calls jprobe_return(), which traps again to restore the original stack contents and processor state and switch to the probed function. [1]

How Does a Jprobe Work? (cont.)

- By convention, the callee owns its arguments, so gcc may produce code that unexpectedly modifies that portion of the stack. This is why Kprobes saves a copy of the stack and restores it after the jprobe handler has run. Up to MAX_STACK_SIZE bytes are copied – e.g., 64 bytes on i386.

How Does a Jprobe Work? (cont.)



[2] How Linux kprobes works

samples/kprobes/jprobe_example.c

```c
/* Proxy routine having the same arguments as actual _do_fork() routine */
static long j_do_fork(unsigned long clone_flags, unsigned long stack_start,
              unsigned long stack_size, int __user *parent_tidptr,
              int __user *child_tidptr)
{
        pr_info("jprobe: clone_flags = 0x%lx, stack_start = 0x%lx "
                "stack_size = 0x%lx\n", clone_flags, stack_start, stack_size);

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

static struct jprobe my_jprobe = {
        .entry                  = j_do_fork,
        .kp = {
                .symbol_name    = "_do_fork",
        },
};

static int __init jprobe_init(void)
{
        int ret;

        ret = register_jprobe(&my_jprobe);
        if (ret < 0) {
                printk(KERN_INFO "register_jprobe failed, returned %d\n", ret);
                return -1:
```

The same arguments and return type with _do_fork

jprobe_example.ko (dmesg)

```
[340911.351067] systemd-journald[12121]: Sent WATCHDOG=1 notification.
[340955.281629] Planted jprobe at ffffffff8107e110, handler addr ffffffffa03ee000
[340956.008739] jprobe: clone_flags = 0x1200011, stack_start = 0x0 stack_size = 0x0
[340958.007804] jprobe: clone_flags = 0x1200011, stack_start = 0x0 stack_size = 0x0
[340959.239834] jprobe: clone_flags = 0x1200011, stack_start = 0x0 stack_size = 0x0
[340960.407847] jprobe: clone_flags = 0x1200011, stack_start = 0x0 stack_size = 0x0
[340963.024339] jprobe: clone_flags = 0x1200011, stack_start = 0x0 stack_size = 0x0
[340964.128255] jprobe: clone_flags = 0x1200011, stack_start = 0x0 stack_size = 0x0
[340965.176114] jprobe: clone_flags = 0x1200011, stack_start = 0x0 stack_size = 0x0
[340968.600854] jprobe: clone_flags = 0x1200011, stack_start = 0x0 stack_size = 0x0
[340968.620048] jprobe at ffffffff8107e110 unregistered
linux-g35h:/home/linux/tmp/samples/kprobes #
```

How Does a Return Probe Work?

- When you call register_kretprobe(), Kprobes establishes a kprobe at the entry to the function. When the probed function is called and this probe is hit, Kprobes saves a copy of the return address, and replaces the return address with the address of a "trampoline." The trampoline is an arbitrary piece of code -- typically just a nop instruction. At boot time, Kprobes registers a kprobe at the trampoline. [1]

register_kretprobe()

kernel/kprobes.c

```
int register_kretprobe(struct kretprobe *rp)
```

kernel/kprobes.c

```
#ifdef CONFIG_KRETPROBES
/*
 * This kprobe pre_handler is registered with every kretprobe. When probe
 * hits it will set up the return probe.
 */
static int pre_handler_kretprobe(struct kprobe *p, struct pt_regs *regs)
```

arch/x86/kernel/kprobes/core.c

```
void arch_prepare_kretprobe(struct kretprobe_instance *ri, struct pt_regs *regs)
{
        unsigned long *sara = stack_addr(regs);

        ri->ret_addr = (kprobe_opcode_t *) *sara;

        /* Replace the return addr with trampoline addr */
        *sara = (unsigned long) &kretprobe_trampoline;
}
NOKPROBE_SYMBOL(arch_prepare_kretprobe);
```

arch/x86/kernel/kprobes/core.c

```
/*
 * When a retprobed function returns, this code saves registers and
 * calls trampoline_handler() runs, which calls the kretprobe's handler.
 */
asm(
        ".global kretprobe_trampoline\n"
        ".type kretprobe_trampoline, @function\n"
        "kretprobe_trampoline:\n"
[...snip]
```

maxactive

- While the probed function is executing, its return address is stored in an object of type kretprobe_instance. Before calling register_kretprobe(), the user sets the maxactive field of the kretprobe struct to specify how many instances of the specified function can be probed simultaneously. register_kretprobe() pre-allocates the indicated number of kretprobe_instance objects. [1]
- Example:
  - non-recursive function + spinlock: maxactive = 1
  - non-recursive function + semaphore or premption: maxactive = NR_CPUS
  - Default maxactive = NR_CPU
  - CONFIG_PREEMPT default maxactive = max (10, 2*NR_CPUS)

maxactive (cont.)

- It's not a disaster if you set maxactive too low; you'll just miss some probes. In the kretprobe struct, the nmissed field is set to zero when the return probe is registered, and is incremented every time the probed function is entered but there is no kretprobe_instance object available for establishing the return probe. [1]

Kretprobe entry-handler

- Kretprobes also provides an optional user-specified handler which runs on function entry. This handler is specified by setting the entry_handler field of the kretprobe struct.
- If the entry_handler returns 0 (success) then a corresponding return handler is guaranteed to be called upon function return. If the entry_handler returns a non-zero error then Kprobes leaves the return address as is, and the kretprobe has no further effect for that particular function instance.

pre_handler_kretprobe()

kernel/kprobes.c

```
ri->rp = rp;
ri->task = current;

if (rp->entry_handler && rp->entry_handler(ri, regs)) {
        raw_spin_lock_irqsave(&rp->lock, flags);
        hlist_add_head(&ri->hlist, &rp->free_instances);
        raw_spin_unlock_irqrestore(&rp->lock, flags);
        return 0;
}

arch_prepare_kretprobe(ri, regs);
```

private data

- a user may also specify per return-instance private data to be part of each kretprobe_instance object.
- This is especially useful when sharing private data between corresponding user entry and return handlers.
- The size of each private data object can be specified at kretprobe registration time by setting the data_size field of the kretprobe struct.

samples/kprobes/kretprobe_example.c

```
static char func_name[NAME_MAX] = "_do_fork";
module_param_string(func, func_name, NAME_MAX, S_IRUGO);
MODULE_PARM_DESC(func, "Function to kretprobe; this module will report the"
                       " function's execution time");

/* per-instance private data */
struct my_data {
        ktime_t entry_stamp;
};

/* Here we use the entry_hanlder to timestamp function entry */
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;
```

samples/kprobes/kretprobe_example.c (cont.)

```
static struct kretprobe my_kretprobe = {
        .handler                = ret_handler,
        .entry_handler          = entry_handler,
        .data_size              = sizeof(struct my_data),
        /* Probe up to 20 instances concurrently. */
        .maxactive              = 20,
};

static int __init kretprobe_init(void)
{
        int ret;

        my_kretprobe.kp.symbol_name = func_name;
        ret = register_kretprobe(&my_kretprobe);
        if (ret < 0) {
                printk(KERN_INFO "register_kretprobe failed, returned %d\n",
                                ret);
                return -1;
        }
        printk(KERN_INFO "Planted return probe at %s: %p\n",
                        my_kretprobe.kp.symbol_name, my_kretprobe.kp.addr);
        return 0;
}

static void __exit kretprobe_exit(void)
{
        unregister_kretprobe(&my_kretprobe);
        printk(KERN_INFO "kretprobe at %p unregistered\n",
```

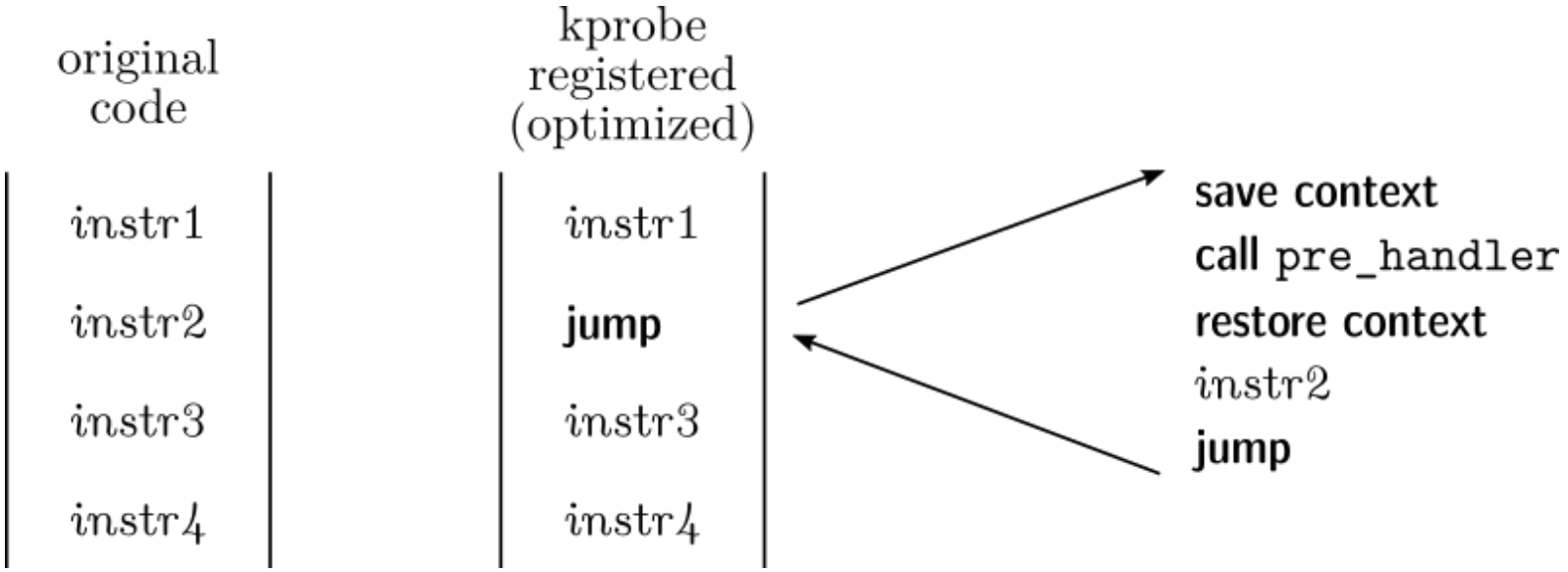
kretprobe_example.ko (dmesg)

```
[383141.351054] systemd-journald[19832]: Sent WATCHDOG=1 notification.
[383161.204938] Planted return probe at _do_fork: ffffffff8107e110
[383162.081700] _do_fork returned 19922 and took 263367 ns to execute
[383163.992987] _do_fork returned 19923 and took 110317 ns to execute
[383169.986164] _do_fork returned 19925 and took 152583 ns to execute
[383173.130452] _do_fork returned 19926 and took 246446 ns to execute
```

How Does Jump Optimization Work?

- If your kernel is built with CONFIG_OPTPROBES=y (currently this flag is automatically set 'y' on x86/x86-64, non-preemptive kernel) and the "debug.kprobes_optimization" kernel parameter is set to 1 (see sysctl(8)), Kprobes tries to reduce probe-hit overhead by using a jump instruction instead of a breakpoint instruction at each probepoint. [1]

How Does Jump Optimization Work? (cont.)



Blacklist

- Kprobes can probe most of the kernel except itself. This means that there are some functions where kprobes cannot probe.
- Probing (trapping) such functions can cause a recursive trap (e.g. double fault) or the nested probe handler may never be called.
- If you want to add a function into the blacklist, you just need to
  - (1) include linux/kprobes.h and
  - (2) use NOKPROBE_SYMBOL() macro to specify a blacklisted function.
- Kprobes checks the given probe address against the blacklist and rejects registering it, if the given address is in the blacklist. [1]

Kprobes Features and Limitations

- Kprobes allows multiple probes at the same address.
  - Currently, however, there cannot be multiple jprobes on the same function at the same time.
- a probepoint for which there is a jprobe or a post_handler cannot be optimized. So if you install a jprobe, or a kprobe with a post_handler, at an optimized probepoint, the probepoint will be unoptimized automatically.
- In general, you can install a probe anywhere in the kernel. In particular, you can probe interrupt handlers.

Kprobes Features and Limitations (Cont.)

- The register_*probe functions will return -EINVAL if you attempt to install a probe in the code that implements Kprobes:
  - kernel/kprobes.c and arch/*/kernel/kprobes.c
  - do_page_fault and notifier_call_chain

…. [1]

Kprobe-based Event Tracing

kprobe-based event tracer

- One of the more significant limitations is the lack of dynamic tracing; ftrace can easily trace function calls or use static tracepoints placed in the kernel source, but it cannot add its own tracepoints on the fly. [3]
- Kprobes are, of course, dynamic tracepoints; by use of on-the-fly code patching, the kernel can hook into its own code at any point. [3]
- With SystemTap, though, these probes are inserted by way of a special kernel module generated on the fly - a bit of a tricky interface. Masami's patch aims to turn the insertion of dynamic probes into something close to a command-line operation. [3]

Command format

- Set a probe:

  p[:[GRP/]EVENT] [MOD:]SYM[+offs]|MEMADDR [FETCHARGS]

- Set a return probe:

  r[MAXACTIVE][:[GRP/]EVENT] [MOD:]SYM[+0] [FETCHARGS]

- Clear a probe:

  -:[GRP/]EVENT

Command format (cont.)

- p[:[GRP/]EVENT] [MOD:]SYM[+offs]|MEMADDR [FETCHARGS]
  - 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.
  - MEMADDR: Address where the probe is inserted.
  - FETCHARGS: Arguments. Each probe can have up to 128 args.
    - %REG: Fetch register REG
    - @ADDR: Fetch memory at ADDR (ADDR should be in kernel)
    - $stack: Fetch stack address.
    - $retval: Fetch return value.(*)
    - +|-offs(FETCHARG) : Fetch memory at FETCHARG +|- offs address.(**)

(*) only for return probe.

(**) this is useful for fetching a field of data structures.

myprobe

- echo 'p:myprobe do_sys_open dfd=%ax filename=%dx flags=%cx mode=+4($stack)' > /sys/kernel/debug/tracing/kprobe_events
- This sets a kprobe on the top of do_sys_open() function with recording 1st to 4th arguments as "myprobe" event. [4]
- 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/). [4]

fs/open.c::do_sys_open()

```
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
        struct open_flags op;
        int fd = build_open_flags(flags, mode, &op);
        struct filename *tmp;

        if (fd)
                return fd;

        tmp = getname(filename);
        if (IS_ERR(tmp))
                return PTR_ERR(tmp);

        fd = get_unused_fd_flags(flags);
        if (fd >= 0) {
                struct file *f = do_filp_open(dfd, tmp, &op);
                if (IS_ERR(f)) {
                        put_unused_fd(fd);
                        fd = PTR_ERR(f);
                } else {
                        fsnotify_open(f);
                        fd_install(fd, f);
                }
        }
        putname(tmp);
        return fd;
}
```

myprobe (cont.)

```
linux-ylrs:/sys/kernel/debug/tracing # echo 'p:myprobe do_sys_open dfd=%ax filename=%dx flags=%cx mode=+4($stack)' > /sys/kernel/debug/tracing/kprobe_events
linux-ylrs:/sys/kernel/debug/tracing # cat /sys/kernel/debug/tracing/kprobe_events
p:kprobes/myprobe do_sys_open dfd=%ax filename=%dx flags=%cx mode=+4($stack)
linux-ylrs:/sys/kernel/debug/tracing # cat /sys/kernel/debug/tracing/events/kprobes/myprobe/enable
0
linux-ylrs:/sys/kernel/debug/tracing # echo 1 > /sys/kernel/debug/tracing/events/kprobes/myprobe/enable
linux-ylrs:/sys/kernel/debug/tracing # cat /sys/kernel/debug/tracing/events/kprobes/myprobe/enable
1
linux-ylrs:/sys/kernel/debug/tracing # cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
# entries-in-buffer/entries-written: 210/210   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
             cat-6017  [003] d... 667254.356660: Unknown type 1120
             cat-6017  [003] d... 667254.356713: Unknown type 1120
             cat-6359  [001] d... 673320.374401: myprobe: (do_sys_open+0x0/0x1f0) dfd=0x2 filename=0x88000 flags=0x1 mode=0x82df00a0ffffffff
             cat-6359  [001] d... 673320.374404: myprobe: (do_sys_open+0x0/0x1f0) dfd=0x2 filename=0x88000 flags=0x1 mode=0x82df00a0ffffffff
             cat-6359  [001] d... 673320.374443: myprobe: (do_sys_open+0x0/0x1f0) dfd=0x2 filename=0x8000 flags=0x0 mode=0x1000ffffffff
          <...>-6360  [000] d... 673322.043746: myprobe: (do_sys_open+0x0/0x1f0) dfd=0x2 filename=0x88000 flags=0x1 mode=0xffffffffffffffff
          <...>-6360  [000] d... 673322.043763: myprobe: (do_sys_open+0x0/0x1f0) dfd=0x2 filename=0x88000 flags=0x8148 mode=0xe681d920ffffffff
          <...>-6360  [000] d... 673322.043955: myprobe: (do_sys_open+0x0/0x1f0) dfd=0x2 filename=0x88000 flags=0x628 mode=0xe681dd30ffffffff
          <...>-6360  [000] d... 673322.043967: myprobe: (do_sys_open+0x0/0x1f0) dfd=0x2 filename=0x88000 flags=0x1b6 mode=0xb4a030ffffffff
```

myretprobe

- 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. [4]
- You can see the format of these events via

  /sys/kernel/debug/tracing/events/kprobes/<EVENT>/format.
- 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). [4]

myretprobe (cont.)

```
linux-ylrs:/sys/kernel/debug/tracing # echo > /sys/kernel/debug/tracing/kprobe_events
linux-ylrs:/sys/kernel/debug/tracing # echo 'r:myretprobe do_sys_open $retval' >> /sys/kernel/debug/tracing/kprobe_events
linux-ylrs:/sys/kernel/debug/tracing # cat /sys/kernel/debug/tracing/kprobe_events
r:kprobes/myretprobe do_sys_open arg1=$retval
linux-ylrs:/sys/kernel/debug/tracing # cat /sys/kernel/debug/tracing/events/kprobes/myretprobe/enable
0
linux-ylrs:/sys/kernel/debug/tracing # echo ^C
linux-ylrs:/sys/kernel/debug/tracing # echo 1 > /sys/kernel/debug/tracing/events/kprobes/myretprobe/enable
linux-ylrs:/sys/kernel/debug/tracing # cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
# entries-in-buffer/entries-written: 247/247   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
            cat-6017  [003] d... 667254.356660: Unknown type 1120
            cat-6017  [003] d... 667254.356713: Unknown type 1120
    irqbalance-888   [001] d... 670024.458432: myretprobe: (SyS_open+0x1e/0x20 <- do_sys_open) arg1=0x3
    irqbalance-888   [001] d... 670024.458807: myretprobe: (SyS_open+0x1e/0x20 <- do_sys_open) arg1=0x3
         <...>-6164  [001] d... 670031.248419: myretprobe: (SyS_open+0x1e/0x20 <- do_sys_open) arg1=0x3
         <...>-6164  [001] d... 670031.248455: myretprobe: (SyS_open+0x1e/0x20 <- do_sys_open) arg1=0x3
         <...>-6164  [001] d... 670031.248956: myretprobe: (SyS_open+0x1e/0x20 <- do_sys_open) arg1=0xfffffffffffffffe
         <...>-6164  [001] d... 670031.248975: myretprobe: (SyS_open+0x1e/0x20 <- do_sys_open) arg1=0x3
```

myretprobe (format)

```
linux-ylrs:/sys/kernel/debug/tracing # cat /sys/kernel/debug/tracing/events/kprobes/myretprobe/format
name: myretprobe
ID: 1124
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_func;       offset:8;       size:8; signed:0;
        field:unsigned long __probe_ret_ip;     offset:16;      size:8; signed:0;
        field:u64 arg1; offset:24;      size:8; signed:0;

print fmt: "(%lx <- %lx) arg1=0x%Lx", REC->__probe_func, REC->__probe_ret_ip, REC->arg1
```

Clear a probe by ‘-:myprobe’

```
linux-g35h:/sys/kernel/debug/tracing # echo 'p:myprobe do_sys_open+260 dfd=%ax filename=%dx flags=%cx mode=+4($stack)' > /sys/kernel/debug/tracing/kprobe_events
linux-g35h:/sys/kernel/debug/tracing # echo 1 > /sys/kernel/debug/tracing/events/kprobes/myprobe/enable
linux-g35h:/sys/kernel/debug/tracing # echo '-:myprobe' > /sys/kernel/debug/tracing/kprobe_events
-bash: /sys/kernel/debug/tracing/kprobe_events: 裝置或系統資源忙碌中
linux-g35h:/sys/kernel/debug/tracing # echo 0 > /sys/kernel/debug/tracing/events/kprobes/myprobe/enable
linux-g35h:/sys/kernel/debug/tracing # echo '-:myprobe' > /sys/kernel/debug/tracing/kprobe_events
-bash: echo: 寫入錯誤： 沒有此一檔案或目錄
linux-g35h:/sys/kernel/debug/tracing # ls /sys/kernel/debug/tracing/events/kprobes
ls: 無法存取 '/sys/kernel/debug/tracing/events/kprobes': 沒有此一檔案或目錄
linux-g35h:/sys/kernel/debug/tracing #
```

SystemTap

Systemtap

- Systemtap is a tool that allows developers and administrators to write and reuse simple scripts to deeply examine the activities of a live Linux system.
- The essential idea behind a systemtap script is to name events, and to give them handlers.
- There are several kind of events, such as entering or exiting a function, a timer expiring, or the entire systemtap session starting or stopping.
- A handler is a series of script language statements that specify the work to be done whenever the event occurs. [6]

Systemtap process

- Process
  - translating the script to C
  - running the system C compiler to create a kernel module from that.
  - When the module is loaded, it activates all the probed events by hooking into the kernel.
  - as events occur on any processor, the compiled handlers run.
  - Eventually, the session stops, the hooks are disconnected, and the module removed.
- This entire process is driven from a single command-line program, stap. [6]

hello-world.stp

zypper -v in systemtap

```
linux-g35h:/home/linux/tmp/systemtap-test # cat hello-world.stp
probe begin
{
        print ("hello world\n")
        exit ()
}
linux-g35h:/home/linux/tmp/systemtap-test # stap hello-world.stp
hello world
```

strace-open.stp

```
linux-g35h:/home/linux/tmp/systemtap-test # cat strace-open.stp
probe syscall.open
{
        printf ("%s(%d) open (%s)\n", execname(), pid(), argstr)
}
probe timer.ms(4000) # after 4 seconds
{
        exit ()
}
```

Missing kernel debuginfo

```
linux-g35h:/home/linux/tmp/systemtap-test # stap strace-open.stp
semantic error: while resolving probe point: identifier 'kernel' at /usr/share/systemtap/tapset/linux/syscalls2.stp:197:24
        source: probe __syscall.open = kernel.function("sys_open").call
                                       ^

semantic error: missing x86_64 kernel/module debuginfo [man warning::debuginfo] under '/lib/modules/4.4.74-18.20-default/build'

semantic error: while resolving probe point: identifier '__syscall' at :177:47
        source: probe syscall.open = __syscall.compat_open ?, __syscall.open
                                                              ^

semantic error: no match

Pass 2: analysis failed.  [man error::pass2]
Number of similar error messages suppressed: 1.
Rerun with -v to see them.
linux-g35h:/home/linux/tmp/systemtap-test #
```

Need kernel-default-debuginfo

- since systemtap examines the kernel's debugging information to relate object code to source code. It works like a debugger: if you can name or place it, you can probe it. [6]
- zypper -v in kernel-default-debuginfo
  - The version should be aligned with kernel-default

strace-open.stp output

```
linux-g35h:/home/linux/tmp/systemtap-test # stap strace-open.stp
systemd-udevd(8298) open ("/proc/self/oom_score_adj", O_WRONLY|O_NOCTTY|O_CLOEXEC)
systemd-udevd(8298) open ("/sys/module/stap_53f60e5ae1efc4427140a45a48968ab3__8297/uevent", O_RDONLY|O_CLOEXEC)
systemd-udevd(8298) open ("/run/udev/data/+module:stap_53f60e5ae1efc4427140a45a48968ab3__8297", O_RDONLY|O_CLOEXEC)
systemd-udevd(8298) open ("/sys/module/uevent", O_RDONLY|O_CLOEXEC)
systemd-udevd(8298) open ("/run/udev/data/+module:module", O_RDONLY|O_CLOEXEC)
systemd-udevd(533) open ("/sys/fs/cgroup/systemd/system.slice/systemd-udevd.service/cgroup.procs", O_RDONLY|O_CLOEXEC)
irqbalance(787) open ("/proc/interrupts", O_RDONLY)
irqbalance(787) open ("/proc/stat", O_RDONLY)
linux-g35h:/home/linux/tmp/systemtap-test #
```

Tracing

- The simplest kind of probe is simply to trace an event.
- It just asks systemtap to print something at each event. To express this in the script language, you need to say where to probe and what to print there.
- The library of scripts that comes with systemtap, each called a ``tapset''

events

- See the stapprobes man page for details on these and many other probe point families. [6]
  - begin: The startup of the systemtap session.
  - end : The end of the systemtap session.
  - kernel.function("sys_open"): The entry to the function named sys_open in the kernel.
  - syscall.close.return: The return from the close system call.
  - module("ext3").statement(0xdeadbeef): The addressed instruction in the ext3 filesystem driver.
  - timer.ms(200): A timer that fires every 200 milliseconds.
  - timer.profile: A timer that fires periodically on every CPU.
  - perf.hw.cache_misses: A particular number of CPU cache misses have occurred.
  - procfs("status").read: A process trying to read a synthetic file.
  - process("a.out").statement("*@main.c:200"): Line 200 of the a.out program.

What to print

- containing the function name. In order to make that list easy to read, systemtap should indent the lines so that functions called by other traced functions are nested deeper. To tell each single process apart from any others that may be running concurrently, systemtap should also print the process ID in the line. [6]

What to print (cont.)

- See the function::* man pages for those functions and more defined in the tapset library, but here's a sampling:
  - tid(): The id of the current thread.
  - pid(): The process (task group) id of the current thread.
  - uid(): The id of the current user.
  - execname(): The name of the current process.
  - cpu(): The current cpu number.
  - ppfunc(): If known, the the function name in which this probe was placed.
  - print_backtrace(): If possible, print a kernel backtrace.

  ...

socket-trace.stp

```
linux-g35h:/home/linux/tmp/systemtap-test # cat socket-trace.stp
probe kernel.function("*@net/socket.c").call {
        printf ("%s -> %s\n", thread_indent(1), ppfunc())
}
probe kernel.function("*@net/socket.c").return {
        printf ("%s <- %s\n", thread_indent(-1), ppfunc())
}
linux-g35h:/home/linux/tmp/systemtap-test # stap socket-trace.stp
WARNING: function sock_init is in blacklisted section: keyword at socket-trace.stp:1:1
 source: probe kernel.function("*@net/socket.c").call {
         ^
     0 ntpd(1176): -> sock_poll
     3 ntpd(1176): <- sock_poll
     0 ntpd(1176): -> sock_poll
     1 ntpd(1176): <- sock_poll
     0 ntpd(1176): -> sock_poll
```

socket-trace.stp

```
linux-g35h:/home/linux/tmp/systemtap-test # cat socket-trace.stp
probe kernel.function("*@net/socket.c").call {
        printf ("%s -> %s\n", thread_indent(1), ppfunc())
}
probe kernel.function("*@net/socket.c").return {
        printf ("%s <- %s\n", thread_indent(-1), ppfunc())
}
linux-g35h:/home/linux/tmp/systemtap-test # stap socket-trace.stp
WARNING: function sock_init is in blacklisted section: keyword at socket-trace.stp:1:1
 source: probe kernel.function("*@net/socket.c").call {
         ^
     0 ntpd(1176): -> sock_poll
     3 ntpd(1176): <- sock_poll
     0 ntpd(1176): -> sock_poll
     1 ntpd(1176): <- sock_poll
     0 ntpd(1176): -> sock_poll
```

```
static int __init sock_init(void)
{
        int err;
        /*
```

Analysis

- With systemtap, it is possible to analyze that data, to filter, aggregate, transform, and summarize it. Different probes can work together to share data.
- Probe handlers can use a rich set of control constructs to describe algorithms, with a syntax taken roughly from awk.
- With these tools, systemtap scripts can focus on a specific question and provide a compact response: no grep needed.

Basic constructs

- if/else statement:
  - if (EXPR) STATEMENT [else STATEMENT]
- while loop
  - while (EXPR) STATEMENT
- for loop
  - for (A; B; C) STATEMENT
- Scripts may use break/continue as in C. Probe handlers can return early using next as in awk.
- String concatenation is done with the dot ("a" . "b"). Some examples:
  - "hello" . " " . "world"

Variables

- By default, variables are local to the probe they are used in. That is, they are initialized, used, and disposed of at each probe handler invocation.
- To share variables between probes, declare them global anywhere in the script.
- Because of possible concurrency (multiple probe handlers running on different CPUs), each global variable used by a probe is automatically read- or write-locked while the handler is running.

Timer-jiffies.stp

```
linux-g35h:/home/linux/tmp/systemtap-test # cat /boot/config-4.4.74-18.20-default  | grep CONFIG_HZ=
CONFIG_HZ=250
linux-g35h:/home/linux/tmp/systemtap-test # cat timer-jiffies.stp
global count_jiffies, count_ms
probe timer.jiffies(100) { count_jiffies ++ }
probe timer.ms(100) { count_ms ++}
probe timer.ms(12345)
{
        hz=(1000*count_jiffies) / count_ms
        printf ("jiffies:ms ratio %d:%d => CONFIG_HZ=%d\n",
                count_jiffies, count_ms, hz)
        exit ()
}
linux-g35h:/home/linux/tmp/systemtap-test # stap timer-jiffies.stp
jiffies:ms ratio 30:123 => CONFIG_HZ=243
linux-g35h:/home/linux/tmp/systemtap-test #
```

Target variables

- A class of special ``target variables'' allow access to the probe point context.
  - & operator: take their address
  - $ and $$ suffix: pretty-print structures
  - $$vars and related variables: pretty-print multiple variables in scope
  - @cast operator: cast pointers to their types
  - @defined operator: test their existence / resolvability

stap -L

```
linux-g35h:/home/linux/tmp/systemtap-test # stap -L 'kernel.function ("vfs_write")'
kernel.function("vfs_write@../fs/read_write.c:523") $file:struct file* $buf:char const* $count:size_t $pos:loff_t*
linux-g35h:/home/linux/tmp/systemtap-test # stap -L 'kernel.function ("vfs_read")'
kernel.function("vfs_read@../fs/read_write.c:440") $file:struct file* $buf:char* $count:size_t $pos:loff_t*
linux-g35h:/home/linux/tmp/systemtap-test #
```

inode-watch.stp

```
linux-g35h:/home/linux/tmp/systemtap-test # cat inode-watch.stp
probe kernel.function ("vfs_write"),
      kernel.function ("vfs_read")
{
        if (@defined($file->f_path->dentry)) {
                dev_nr = $file->f_path->dentry->d_inode->i_sb->s_dev
                inode_nr = $file->f_path->dentry->d_inode->i_ino
        } else {
                dev_nr = $file->f_dentry->d_inode->i_sb->s_dev
                inode_nr = $file->f_dentry->d_inode->i_ino
        }

//      if (dev_nr == ($1 << 20 | $2) # major/minor device
        if (inode_nr == $3)
                printf ("%s(%d) %s 0x%x/%u\n",
                        execname(), pid(), ppfunc(), dev_nr, inode_nr)
}
linux-g35h:/home/linux/tmp/systemtap-test # stat -c "%D %i" socket-trace.stp
2f 317
linux-g35h:/home/linux/tmp/systemtap-test # stap inode-watch.stp 0x00 47 317
cat(32418) vfs_read 0x22/317
cat(32418) vfs_read 0x22/317
^Clinux-g35h:/home/linux/tmp/systemtap-test #
```

Functions

- Like global variables, systemtap functions may be defined anywhere in the script.
- They may take any number of string or numeric arguments (by value), and may return a single string or number.
- Local and global script variables are available, but target variables are not.
- A function is defined with the keyword function followed by a name. Then comes a comma-separated formal argument list (just a list of variable names).

functions.stp

```
linux-g35h:/home/linux/tmp/systemtap-test # cat functions.stp
# openSUSE /etc/login.defs UID_MIN 1000
function system_uid_p (u) { return u < 1000}

probe begin
{
        printf ("uid: %d, system_uid_p(): %d\n", uid(), system_uid_p(uid()))
}

# kernel device number assymbly marco
function makedev (major,minor) { return major << 20 | minor}

probe kernel.function ("vfs_write"),
      kernel.function ("vfs_read")
{
        if (@defined($file->f_path->dentry)) {
                dev_nr = $file->f_path->dentry->d_inode->i_sb->s_dev
                inode_nr = $file->f_path->dentry->d_inode->i_ino
        } else {
                dev_nr = $file->f_dentry->d_inode->i_sb->s_dev
                inode_nr = $file->f_dentry->d_inode->i_ino
        }

        if (inode_nr == $3)
                printf ("%s(%d) %s 0x%x/%u, input dev_nr: 0x%x\n",
                        execname(), pid(), ppfunc(), dev_nr, inode_nr, makedev($1,$2))
}
linux-g35h:/home/linux/tmp/systemtap-test # stap functions.stp 0x00 47 317
uid: 0, system_uid_p(): 1
cat(1990) vfs_read 0x22/317, input dev_nr: 0x2f
cat(1990) vfs_read 0x22/317, input dev_nr: 0x2f
```

Arrays

- Systemtap offers associative arrays for this purpose. These arrays are implemented as hash tables with a maximum size that is fixed at startup.
  - *global b[400]* declare array, reserving space for up to 400 tuples
  - *times [tid()] = get_cycles()* set a timestamp reference point
  - *if ([4,"hello"] in foo) { }* membership test
  - *delete times[tid()]* deletion of a single element
  - *delete times* deletion of all elements

Arrays (cont.)

- One final and important operation is iteration over arrays. This uses the keyword foreach.
- the iteration may be sorted by any single key or the value by adding an extra + or - code.
  - *foreach (x = [a,b] in foo) { fuss_with(x) }* simple loop in arbitrary sequence
  - *foreach ([a,b] in foo+ limit 5) { }* loop in increasing sequence of value, stop after 5
  - *foreach ([a-,b] in foo) { }* loop in decreasing sequence of first key

System-Wide Event Enabling with SystemTap

- In SystemTap, tracepoints are accessible using the kernel.trace() function call. The following is an example that reports every 5 seconds what processes were allocating the pages. [7]

mm_page_alloc.stp

```
global page_allocs

probe kernel.trace("mm_page_alloc") {
        page_allocs[execname()]++
}

function print_count() {
        printf ("%-25s %-s\n", "#Pages Allocated", "Process Name")
        foreach (proc in page_allocs-)
                printf("%-25d %s\n", page_allocs[proc], proc)
        printf ("\n")
        delete page_allocs
}

probe timer.s(5) {
        print_count()
}
```

mm_page_alloc.stp (output)

```
linux-g35h:/home/linux/tmp/systemtap-test # stap mm_page_alloc.stp
#Pages Allocated          Process Name
2                         kworker/0:2
2                         irqbalance

#Pages Allocated          Process Name
147                       btrfs-transacti
3                         kworker/0:2

#Pages Allocated          Process Name
2                         kworker/0:2
2                         irqbalance

#Pages Allocated          Process Name
3                         kworker/0:2
1                         systemd
```

triggering an error path

```
probe module("mpt3sas").function("mpt3sas_config_get_sas_iounit_pg0").return
{
        $return = -1
}
```

- The above small SystemTap script hooks into the mpt3sas module and changes the return value of the "mpt3sas_config_get_sas_iounit_pg0" function to be -1 triggering an error path. [Johannes Thumshirn]

Kernel API wrapper by embedded C

```
%{
#include <linux/mm.h>
%}

function dump_page (page:long) %{
        struct page *p = (struct page *)((long)STAP_ARG_page);
        dump_page (p, "isolation failed");
%}

probe kernel.function("isolate_lru_page").return {
        if ($return != 0)
                dump_page ($page)
}
```

Q&A

Reference

- [1] Documentation/kprobes.txt, Jim Keniston, Linux Kernel
- [2] How Linux kprobes works, vjp, http://vjordan.info
- [3] Dynamic probes with ftrace, Jonathan Corbet, LWN.net, July 28 2009
- [4] Documentation/trace/kprobetrace.txt, Masami Hiramatsu, Linux Kernel
- [5] Dynamic Event Tracing in Linux Kernel, Masami Hiramatsu, April 5 2010
- [6] Systemtap tutorial, Frank Ch. Eigler, https://sourceware.org/systemtap/tutorial
- [7] Documentation/trace/tracepoint-analysis.txt, Mel Gorman, Linux Kernel

Reference (cont.)

- [8] Minimizing instrumentation impacts, Jake Edge, LWN.net, December 9, 2009
- [9] http://wiki.osdev.org/System_V_ABI

Feedback to
jlee@suse.com

Thank you.


SUSE®
We adapt. You succeed.

Corporate Headquarters
Maxfeldstrasse 5
90409 Nuremberg
Germany

+49 911 740 53 0 (Worldwide)
www.suse.com

Join us on:
www.opensuse.org