Gabriele **Proietti Mattia**

Advanced Operating Systems and Virtualization

[Lab 09] Kprobes and `ftrace`

Department of Computer, Control and Management Engineering "A. Ruberti", Sapienza University of Rome

gpm.name · proiettimattia@diag.uniroma1.it

A.Y. 2020/2021 · v1

Introduction

Examples are from the folders

https://github.com/gabrielepmattia/aosv-code-examples/tree/main/11-kprobes

https://github.com/gabrielepmattia/aosv-code-examples/tree/main/12-ftrace

Kprobes

Kernel Exported Symbols

Symbols from the Kernel or from modules can be exported. An exported symbol can be referenced by other modules. If a module references a symbol which is not exported, then loading the module will fail. The macro

```
EXPORT_SYMBOL(symbol)
```

is defined in include/linux/module.h.

This must be configured:

`CONFIG_KALLSYMS = y`

`CONFIG_KALLSYMS_ALL = y` (include all symbols)

Exported symbols are placed in the `__ksymtab` section.

Kprobes

Kprobes are meant as a support for dynamic tracing in the Kernel. Suppose that you want to track the execution of a function (=symbol). You can register a **pre** and **post** function handlers by registering a kprobe with:

```
int register_kprobe(struct kprobe *p)
```

declared include/linux/kprobes.h. The struct kprobe specifies where the probe is to be inserted and what pre_ and post_ handlers are to be called when the probe is hit.

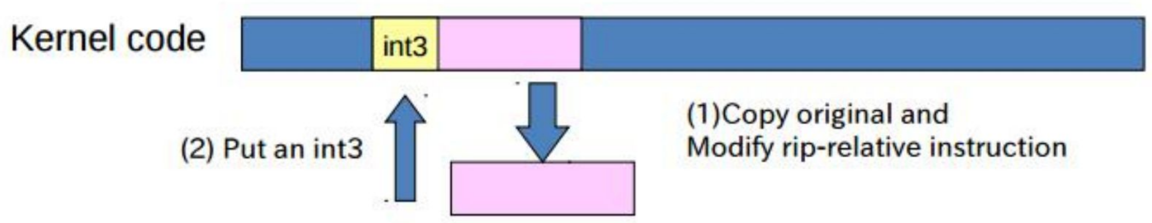
```
unregister_kprobe(struct kprobe *p)
typedef int (*kprobe_pre_handler_t) (struct kprobe *, struct pt_regs *)
```

Kprobes must be enabled (usually they are) with:

`CONFIG_KPROBES=y` and `CONFIG_KALLSYMS=y` or `CONFIG_KALLSYMS_ALL=y`

How it works

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).

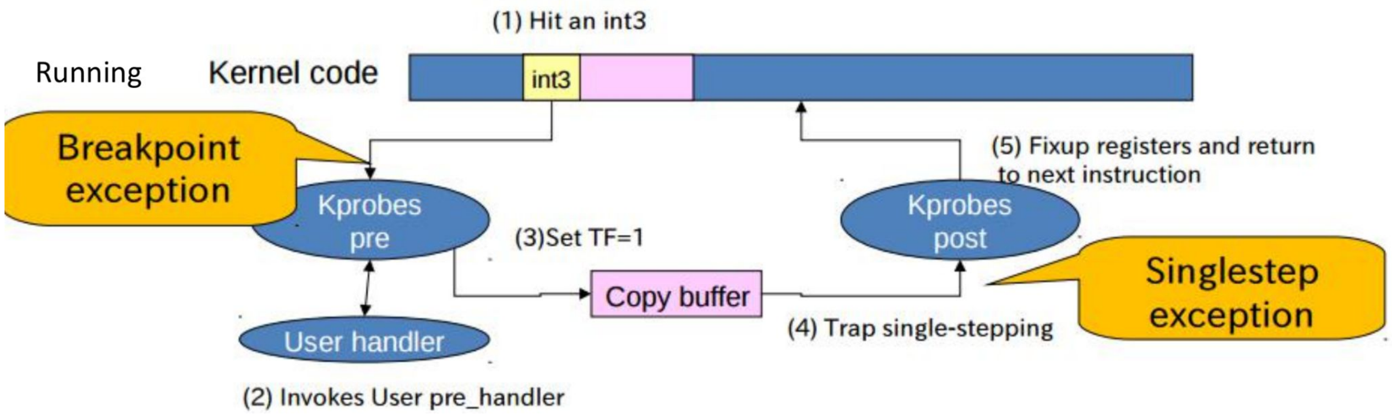


When a CPU hits the breakpoint instruction, a **trap** occurs, the CPU's registers are saved, and control passes to Kprobes via the `notifier_call_chain` mechanism. Kprobes executes the "`pre_handler`" associated with the kprobe, passing the handler the addresses of the kprobe struct and the saved registers.

https://www.kernel.org/doc/html/latest/trace/kprobes.html

How it works

Next, Kprobes **single-steps** its copy of the probed instruction. 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.



When the debugger sets the trap flag, this causes the processor to automatically execute an **int 1 interrupt after every instruction**. This allows the debugger to single-step by instructions, without having to insert an int 3 instruction. You do not have to invoke this interrupt explicitly.

https://www.kernel.org/doc/html/latest/trace/kprobes.html

Return Kprobes

A return probe, differently from a standard Kprobe only replaces the return address of a symbol in order to execute a custom function.

How it works

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.

When the probed function executes its return instruction, **control passes to the trampoline and that probe is hit**. Kprobes' trampoline handler **calls the user-specified return handler** associated with the kretprobe, then sets the saved instruction pointer to the saved return address, and that's where execution resumes upon return from the trap.

https://www.kernel.org/doc/html/latest/trace/kprobes.html

Impact on performances

On a typical CPU in use in 2005, a kprobe hit takes **0.5 to 1.0 microseconds** to process. Specifically, a benchmark that hits the same probepoint repeatedly, firing a simple handler each time, reports 1-2 million hits per second, depending on the architecture. A return-probe hit typically takes 50-75% longer than a kprobe hit. When you have a return probe set on a function, adding a kprobe at the entry to that function adds essentially no overhead.

Here are sample overhead figures (in usec) for different architectures:

```
k = kprobe; r = return probe; kr = kprobe + return probe on same function

i386: Intel Pentium M, 1495 MHz, 2957.31 bogomips
k = 0.57 usec; r = 0.92; kr = 0.99

x86_64: AMD Opteron 246, 1994 MHz, 3971.48 bogomips
k = 0.49 usec; r = 0.80; kr = 0.82

ppc64: POWER5 (gr), 1656 MHz (SMT disabled, 1 virtual CPU per physical CPU)
k = 0.77 usec; r = 1.26; kr = 1.45
```

https://www.kernel.org/doc/html/latest/trace/kprobes.html

Limitations

Kprobes can be installed anywhere in the kernel

- Multiple probes at the same address
- Multiple handlers (or multiple instances of the same handler) may run concurrently on different CPUs.

Registered kprobes are visible under the `/sys/kernel/debug/kprobes/` directory. When registered, probes are saved in a hash table hashed by the address of the probe. The Hash table is protected by `kprobe_lock` (a spinlock)

Kprobes cannot probe itself, it uses a blacklist to prevent recursive traps

Probe handlers are run with preemption disabled. Depending on the architecture and optimization state, handlers may also run with interrupts disabled (not on x86/x86-64). In any case, should not yield the CPU (e.g., by attempting to acquire a semaphore).

https://www.kernel.org/doc/html/latest/trace/kprobes.html

Example

Non-Exported Symbols

```
// Get a kernel probe to access flush_tlb_all
memset(&kp, 0, sizeof(kp));
kp.symbol_name = "flush_tlb_all";
if (!register_kprobe(&kp)) {
    flush_tlb_all_lookup = (void *)kp.addr;
    unregister_kprobe(&kp);
}
```

ftrace

Hot Patching

In some cases you cannot allow to reboot the machine for updating. For this reason you need a way for patching the kernel at runtime.

Linux introduced the kpatch subsystem that tries to overcome the costs and problems of rebooting systems to apply patches. It is based on two "simple" steps:

1. **build** the patch module (`kpatch-build foo.patch`)
2. **apply** the patch (`kpatch load kpatch-foo.ko`)

Building the Patch

However, building the patch is much harder that apply it. You need for example to **compile** kernel with/without patch, **compare** binaries, **detect** which functions have changed and **extract** object code of changed functions into patch module.

Hot Patching

Applying the Patch

For applying the patch you need to:

- **load** new functions into memory
- **link** new functions into kernel, allows access to unexported kernel symbols
- activeness safety check
    - prevent old & new functions from running at same time
    - `stop_machine()` + stack backtrace checks
- **patch** the functions, with `ftrace`

What is ftrace

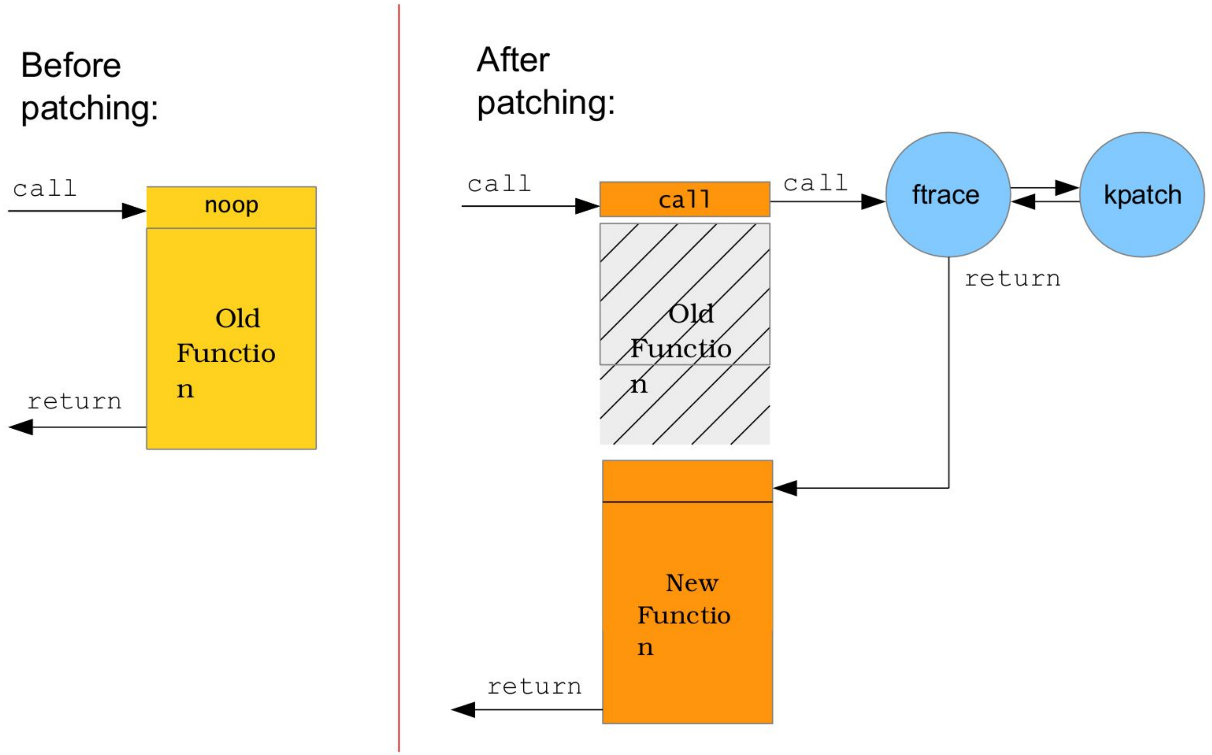
Ftrace is the first generic tracing system to get mainlined.

- Mainlined in 2.6.27
- Derived from RT-preempt latency tracer

It provides a **generic framework** for tracing

- infrastructure for defining tracepoints
- ability to register different kinds of tracers
- specialized data structure (ring buffer) for trace data storage

Patching with ftrace



Advanced Operating Systems and Virtualization

[Lab 09] Kprobes and ftrace

LECTURER

Gabriele **Proietti Mattia**

gpm.name · proiettimattia@diag.uniroma1.it

DIAG