운영체제 및 실습
- Kprobe -

한예진, 김수창, 이승준

Dankook University

Subject

1. Linux Kernel?

2. Linux Kernel Module

3. Kernel Debugging

4. Kprobe? KretProbe? Jprobe?

5. Code Example

Linux Kernel

Que

What is an OS kernel ? How does it differ from an operating system? [closed]

Ask Question

Asked 14 years, 1 month ago Modified 5 years, 11 months ago Viewed 208k times

156

Closed. This question is off-topic. It is not currently accepting answers.

Want to improve this question? Update the question so it's on-topic for Stack Overflow.

Closed 11 years ago.

Improve this question

I am not able to understand the difference between a kernel and an operating system. I do not see any difference between them. Is the kernel an operating system?

The Overflow Blog

How to convince your CEO it's worth paying down tech debt

Optimizing both hardware and software for GenAI

sponsored post

Upcoming Events

2024 Community Moderator Election ends March 13

Linux Kernel

Linux Kernel

79

A *kernel* is the part of the operating system that mediates access to system resources. It's responsible for enabling multiple applications to effectively share the hardware by controlling access to CPU, memory, disk I/O, and networking.

An *operating system* is the kernel plus applications that enable users to get something done (i.e compiler, text editor, window manager, etc).

Share Improve this answer Follow

answered Jan 6, 2010 at 15:30

Erich Douglass

51.8k ● 11 ● 76 ● 60

Linux Kernel

Linux Kernel

Linux Kernel만이 Operating System의 구성품일까?

NO!!!

Linux Kernel

Linux Kernel

What is there in an Operating System other than the kernel

Asked 12 years, 6 months ago Modified 1 year, 8 months ago Viewed 10k times

50

As I understand it, the kernel does all the interaction with the hardware, and manages the memory, the I/O devices, etc. So the kernel is doing everything, yet it is just a part of the operating system. So what else is there in an OS ? Just the applications that come bundled with it ? For example, what does Ubuntu have other than a kernel ? The Gnome Desktop, and a few other applications ?

operating-systems kernel

Share Improve this question Follow

asked Aug 29, 2011 at 17:18

AnkurVj

1,101 ● 3 ● 10 ● 21

Add a comment

Linux Kernel

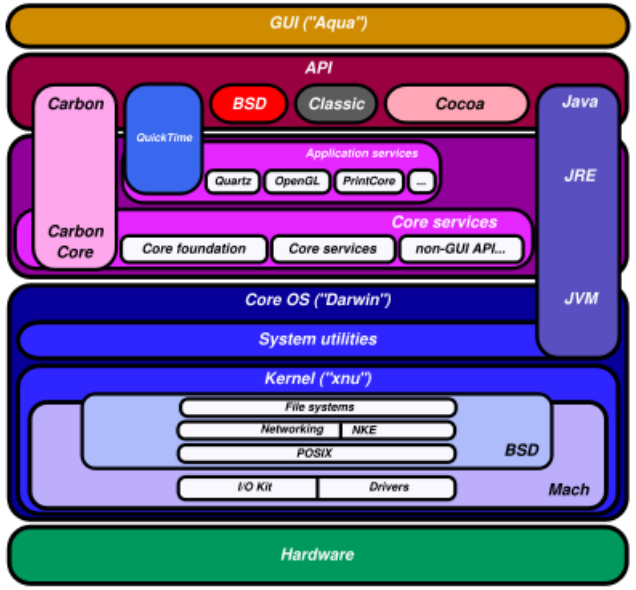
Linux Kernel

The shell, of course.

The original metaphor that got us the word "kernel" for this in the first place is too often forgotten. The metaphor is that an operating system is a seed or a nut. The "kernel" of the seed is the core of the operating system, providing operating system services to applications programs, which is surrounded by the "shell" of the seed that is what users see from the outside.

Linux Kernel

To give an example, I've shown the structure of OS X, as you can see on top of the Kernel (light blue) lies several layers of "core" functionality, system tools, services and APIs (Application Programming Interface) before you even get to the GUI which is what you typically work with in the actual applications themselves.



Most operating systems have a similar structure, but there is wide scope for differences with regard to just how much is actually part of the kernel. See the Wikipedia article on Microkernels for a comparison.

Linux Kernel Module

Linux Kernel Module

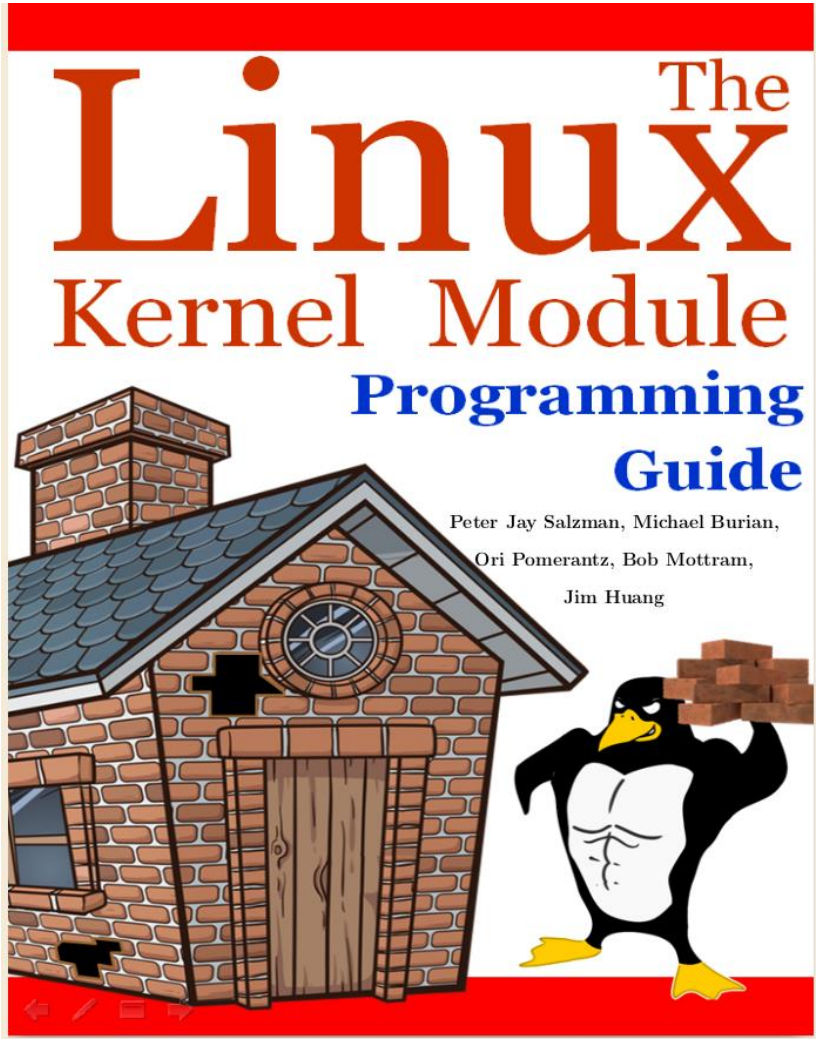
1.1. What Is A Kernel Module?

So, you want to write a kernel module. You know C, you've written a few normal programs to run as processes, and now you want to get to where the real action is, to where a single wild pointer can wipe out your file system and a core dump means a reboot.

What exactly is a kernel module? Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality.

Linux Kernel Module

Linux Kernel



Linux Kernel Module

Linux Kernel

2 Headers

Before building anything, it is necessary to install the header files for the kernel.

On Ubuntu/Debian GNU/Linux:

```
1 sudo apt-get update
2 apt-cache search linux-headers-`uname -r`
```

The following command provides information on the available kernel header files. Then for example:

```
1 sudo apt-get install kmod linux-headers-5.4.0-80-generic
```

On Arch Linux:

```
1 sudo pacman -S linux-headers
```

On Fedora:

```
1 sudo dnf install kernel-devel kernel-headers
```

Linux Kernel Module

Linux Kernel

vi hello.c

```c
/*
 * hello.c - Demonstrating the module_init() and module_exit() macros.
 * This is preferred over using init_module() and cleanup_module().
 */
#include <linux/init.h> /* Needed for the macros */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

static int __init hello_init(void)
{
    pr_info("Hello, world\n");
    return 0;
}

static void __exit hello_exit(void)
{
    pr_info("Goodbye, world\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
```

vi Makefile

```
obj-m += hello.o

PWD := $(CURDIR)

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Linux Kernel Module

Linux Kernel

```
~/kernel make
make -C /lib/modules/6.5.0-21-generic/build M=/home/tuuna/kernel  modules
make[1]: 디렉터리 '/usr/src/linux-headers-6.5.0-21-generic' 들어감
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc-12 (Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0
  You are using:           gcc-12 (Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0
  CC [M]  /home/tuuna/kernel/hello.o
  MODPOST /home/tuuna/kernel/Module.symvers
  CC [M]  /home/tuuna/kernel/hello.mod.o
  LD [M]  /home/tuuna/kernel/hello.ko
  BTF [M] /home/tuuna/kernel/hello.ko
Skipping BTF generation for /home/tuuna/kernel/hello.ko due to unavailability of vmlinux
make[1]: 디렉터리 '/usr/src/linux-headers-6.5.0-21-generic' 나감
~/kernel modinfo hello.ko
filename:       /home/tuuna/kernel/hello.ko
license:        GPL
srcversion:     80158CC88F1748420B63317
depends:
retpoline:      Y
name:           hello
vermagic:       6.5.0-21-generic SMP preempt mod_unload modversions
~/kernel
```

```
~/kernel lsmod | grep hello
✘ ~/kernel sudo insmod hello.ko
~/kernel lsmod | grep hello
hello                  12288  0
~/kernel
~/kernel sudo rmmod hello
~/kernel
```

```
~/kernel sudo dmesg | grep world
[  910.078713] Hello, world 2
[ 3918.339608] Goodbye, world 2
[ 3976.637995] Hello, world
[ 3999.381582] Goodbye, world
```

Linux Kernel Debugging

Linux Kernel Debugging

So, How To Debug Linux Kernel?

Tools:
1. ftrace
2. Kprobe
3. perf
4. printk … XD

Printk

Use Printk

```
1 int inet_accept(struct socket *sock, struct socket *newsock, int flags,
2         bool kern)
3 {
4     struct sock *sk1 = sock->sk, *sk2;
5     int err = -EINVAL;
6
7     /* DEBUG SECTION */
8     printk(KERN_INFO "Listener Socket State: %d\n", sock->state);
9 }
~
```

Kprobe

Kprobe

```
 1 Concepts: Kprobes and Return Probes
 2 ======================================
 3
 4 Kprobes enables you to dynamically break into any kernel routine and
 5 collect debugging and performance information non-disruptively. You
 6 can trap at almost any kernel code address [1]_, specifying a handler
 7 routine to be invoked when the breakpoint is hit.
 8
 9 .. [1] some parts of the kernel code can not be trapped, see
10        :ref:`kprobes_blacklist`)
11
12 There are currently two types of probes: kprobes, and kretprobes
13 (also called return probes).  A kprobe can be inserted on virtually
14 any instruction in the kernel.  A return probe fires when a specified
15 function returns.
~
```

Kprobe

Linux Kernel

```
In the typical case, Kprobes-based instrumentation is packaged as
a kernel module.  The module's init function installs ("registers")
one or more probes, and the exit function unregisters them.  A
registration function such as register_kprobe() specifies where
the probe is to be inserted and what handler is to be called when
the probe is hit.

There are also ``register_/unregister_*probes()`` functions for batch
registration/unregistration of a group of ``*probes``. These functions
can speed up unregistration process when you have to unregister
a lot of probes at once.
```

Linux Kernel

How Does a Kprobe Work?

Kprobe

Linux Kernel

```
1 When a kprobe is registered, Kprobes makes a copy of the probed
2 instruction and replaces the first byte(s) of the probed instruction
3 with a breakpoint instruction (e.g., int3 on i386 and x86_64).
4
5 When a CPU hits the breakpoint instruction, a trap occurs, the CPU's
6 registers are saved, and control passes to Kprobes via the
7 notifier_call_chain mechanism.  Kprobes executes the "pre_handler"
8 associated with the kprobe, passing the handler the addresses of the
9 kprobe struct and the saved registers.
```

Kprobe

Linux Kernel

```
1 Next, Kprobes single-steps its copy of the probed instruction.
2 (It would be simpler to single-step the actual instruction in place,
3 but then Kprobes would have to temporarily remove the breakpoint
4 instruction.  This would open a small time window when another CPU
5 could sail right past the probepoint.)
6
7 After the instruction is single-stepped, Kprobes executes the
8 "post_handler," if any, that is associated with the kprobe.
9 Execution then continues with the instruction following the probepoint.
```

Linux Kernel

Changing Execution Path

Kprobe

Linux Kernel

```
 1 Since kprobes can probe into a running kernel code, it can change the
 2 register set, including instruction pointer. This operation requires
 3 maximum care, such as keeping the stack frame, recovering the execution
 4 path etc. Since it operates on a running kernel and needs deep knowledge
 5 of computer architecture and concurrent computing, you can easily shoot
 6 your foot.
 7
 8 If you change the instruction pointer (and set up other related
 9 registers) in pre_handler, you must return !0 so that kprobes stops
10 single stepping and just returns to the given address.
11 This also means post_handler should not be called anymore.
~
```

Linux Kernel

Return Probes

Linux Kernel

How Does a Return Probe Work?

Kprobe

Linux Kernel

```
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.
```

Kprobe

Linux Kernel

Okay then… How to use kprobe or return probe?

Kprobe

Linux Kernel

```
Architectures Supported
=======================

Kprobes and return probes are implemented on the following
architectures:

- i386 (Supports jump optimization)
- x86_64 (AMD-64, EM64T) (Supports jump optimization)
- ppc64
- ia64 (Does not support probes on instruction slot1.)
- sparc64 (Return probes not yet implemented.)
- arm
- ppc
- mips
- s390
- parisc
```

```
Configuring Kprobes
===================

When configuring the kernel using make menuconfig/xconfig/oldconfig,
ensure that CONFIG_KPROBES is set to "y". Under "General setup", look
for "Kprobes".

So that you can load and unload Kprobes-based instrumentation modules,
make sure "Loadable module support" (CONFIG_MODULES) and "Module
unloading" (CONFIG_MODULE_UNLOAD) are set to "y".

Also make sure that CONFIG_KALLSYMS and perhaps even CONFIG_KALLSYMS_ALL
are set to "y", since kallsyms_lookup_name() is used by the in-kernel
kprobe address resolution code.

If you need to insert a probe in the middle of a function, you may find
it useful to "Compile the kernel with debug info" (CONFIG_DEBUG_INFO),
so you can use "objdump -d -l vmlinux" to see the source-to-object
code mapping.
```

Kprobe

API Reference

```
register_kprobe
---------------
    #include <linux/kprobes.h>
    int register_kprobe(struct kprobe *kp);

With the introduction of the "symbol_name" field to struct kprobe,
the probepoint address resolution will now be taken care of by the kernel.
The following will now work::

kp.symbol_name = "symbol_name";

register_kprobe() returns 0 on success, or a negative errno otherwise.
```

```
User's pre-handler (kp->pre_handler)::

    #include <linux/kprobes.h>
    #include <linux/ptrace.h>
    int pre_handler(struct kprobe *p, struct pt_regs *regs);

User's post-handler (kp->post_handler)::

    #include <linux/kprobes.h>
    #include <linux/ptrace.h>
    void post_handler(struct kprobe *p, struct pt_regs *regs,
              unsigned long flags);

p and regs are as described for the pre_handler.  flags always seems
to be zero.

User's fault-handler (kp->fault_handler)::

    #include <linux/kprobes.h>
    #include <linux/ptrace.h>
    int fault_handler(struct kprobe *p, struct pt_regs *regs, int trapnr);
```

Kprobe

API Reference

```
register_kretprobe
------------------
    #include <linux/kprobes.h>
    int register_kretprobe(struct kretprobe *rp);

register_kretprobe() returns 0 on success, or a negative errno
otherwise.

User's return-probe handler (rp->handler)::

    #include <linux/kprobes.h>
    #include <linux/ptrace.h>
    int kretprobe_handler(struct kretprobe_instance *ri,
                  struct pt_regs *regs);

regs is as described for kprobe.pre_handler.  ri points to the
kretprobe_instance object, of which the following fields may be
of interest:

- ret_addr: the return address
- rp: points to the corresponding kretprobe object
- task: points to the corresponding task struct
- data: points to per return-instance private data; see "Kretprobe
   entry-handler" for details.
```

Kprobe

API Reference

We will Hook `inet_csk_accept` function

What is the `inet_csk_accept`?

Accept → __sys_accept4 → __sys_accept4_file → do_accept → inet_accept → **inet_csk_accept**

accept함수 호출 시 3 Way Handshake가 끝난 소켓을 Userspace에게 알려주는 중요한 함수

Kprobe Example

```c
#define pr_fmt(fmt) "%s: " fmt, __func__

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

static char symbol[KSYM_NAME_LEN] = "inet_csk_accept";
module_param_string(symbol, symbol, KSYM_NAME_LEN, 0644);

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

/* kprobe pre_handler: called just before the probed instruction is executed */
static int __kprobes handler_pre(struct kprobe *p, struct pt_regs *regs)
{
    printk(KERN_INFO "I am Handler Pre\n");
#ifdef CONFIG_X86
    pr_info("<%s> p->addr = 0x%p, ip = %lx, ax = 0x%lx, cx = 0x%lx, sp = 0x%lx\n",
        p->symbol_name, p->addr, regs->ip, regs->ax, regs->cs, regs->sp);
#endif
    return 0;
}
```

```c
/* kprobe post_handler: called after the probed instruction is executed */
static void __kprobes handler_post(struct kprobe *p, struct pt_regs *regs,
                unsigned long flags)
{
    printk(KERN_INFO "I am Handler Post\n");
#ifdef CONFIG_X86
    pr_info("<%s> p->addr = 0x%p, ip = %lx, ax = 0x%lx, cx = 0x%lx, sp = 0x%lx\n",
        p->symbol_name, p->addr, regs->ip, regs->ax, regs->cs, regs->sp);
#endif
}

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

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

static void __exit kprobe_exit(void)
{
    unregister_kprobe(&kp);
    pr_info("kprobe at %p unregistered\n", kp.addr);
}

module_init(kprobe_init)
module_exit(kprobe_exit)
MODULE_LICENSE("GPL");
```

Kprobe

Kprobe example

```
~/kernel  make
make -C /lib/modules/6.5.0-21-generic/build M=/home/tuuna/kernel  modules
make[1]: 디렉터리 '/usr/src/linux-headers-6.5.0-21-generic' 들어감
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc-12 (Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0
  You are using:           gcc-12 (Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0
  CC [M]  /home/tuuna/kernel/hello.o
  MODPOST /home/tuuna/kernel/Module.symvers
  CC [M]  /home/tuuna/kernel/hello.mod.o
  LD [M]  /home/tuuna/kernel/hello.ko
  BTF [M] /home/tuuna/kernel/hello.ko
Skipping BTF generation for /home/tuuna/kernel/hello.ko due to unavailability of vmlinux
make[1]: 디렉터리 '/usr/src/linux-headers-6.5.0-21-generic' 나감
~/kernel  sudo insmod hello.ko
~/kernel  lsmod | grep hello
hello                  12288  0
~/kernel
```

Kprobe Example

Kprobe example

```
[12945.114033] kprobe_init: Planted kprobe at 000000000f0c9994
[12979.719234] I am Handler Pre
[12979.719238] handler_pre: <inet_csk_accept> p->addr = 0x000000000f0c9994, ip = ffffffff9b6e0eb1, ax = 0xffffffff9b
e0eb0, cx = 0x10, sp = 0xffff99d04869bd38
[12979.719246] I am Handler Post
[12979.719247] handler_post: <inet_csk_accept> p->addr = 0x000000000f0c9994, ip = ffffffff9b6e0eb5, ax = 0xffffffff9
6e0eb0, cx = 0x10, sp = 0xffff99d04869bd38
~/kernel
```

kprobe_handler_pre → **SINGLE STEP** → kprobe_handler_post

단지 명령어를 Single Step 한번 했기 때문에 IP만 opcode 크기만큼 증가하고 register는 opcode에 따라 달라짐

Kretprobe Example


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

static char func_name[KSYM_NAME_LEN] = "inet_csk_accept";
module_param_string(func, func_name, KSYM_NAME_LEN, 0644);
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;
}
NOKPROBE_SYMBOL(entry_handler);
```

```c
/*
 * 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)
{
    unsigned long retval = regs_return_value(regs);
    struct my_data *data = (struct my_data *)ri->data;
    s64 delta;
    ktime_t now;

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

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,
};

```

Kretprobe Example


```c
static int __init kretprobe_init(void)
{
    int ret;

    my_kretprobe.kp.symbol_name = func_name;
    ret = register_kretprobe(&my_kretprobe);
    if (ret < 0) {
        pr_err("register_kretprobe failed, returned %d\n", ret);
        return ret;
    }
    pr_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);
    pr_info("kretprobe at %p unregistered\n", my_kretprobe.kp.addr);

    /* nmissed > 0 suggests that maxactive was set too low. */
    pr_info("Missed probing %d instances of %s\n",
        my_kretprobe.nmissed, my_kretprobe.kp.symbol_name);
}

module_init(kretprobe_init)
module_exit(kretprobe_exit)
MODULE_LICENSE("GPL");
```

```
~/kernel make
make -C /lib/modules/6.5.0-21-generic/build M=/home/tuuna/kernel  modules
make[1]: 디렉터리 '/usr/src/linux-headers-6.5.0-21-generic' 들어감
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc-12 (Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0
  You are using:           gcc-12 (Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0
  CC [M]  /home/tuuna/kernel/hello.o
  MODPOST /home/tuuna/kernel/Module.symvers
  CC [M]  /home/tuuna/kernel/hello.mod.o
  LD [M]  /home/tuuna/kernel/hello.ko
  BTF [M] /home/tuuna/kernel/hello.ko
Skipping BTF generation for /home/tuuna/kernel/hello.ko due to unavailability of vmlinux
make[1]: 디렉터리 '/usr/src/linux-headers-6.5.0-21-generic' 나감
~/kernel sudo insmod hello.ko
~/kernel lsmod | grep hello
hello                  12288  0
~/kernel
```

```
[13481.712943] Planted return probe at inet_csk_accept: 00000000f0c9994
[13508.431976] inet_csk_accept returned 18446619351857781248 and took 4372 ns to execute
~/kernel
```

Jprobe

So Hard…

Jprobe! **But….**

Jprobes is now a deprecated feature. People who are depending on it should migrate to other tracing features or use older kernels. Please consider to migrate your tool to one of the following options:

XD

Kprobe

실전!

어떻게 Kprobe를 잘 사용했다고 소문이 날까?

Offset!

kprobe는 특정 주소에 대해 single step한다.
즉, 특정 주소에 offset을 더해 디버깅한다면...?

```
struct kprobe {
        struct hlist_node hlist;
        /* list of kprobes for multi-handler support */
        struct list_head list;
        /*count the number of times this probe was temporarily disarmed */
        unsigned long nmissed;
        /* location of the probe point */
        kprobe_opcode_t *addr;
        /* Allow user to indicate symbol name of the probe point */
        const char *symbol_name;
        /* Offset into the symbol */
        unsigned int offset;
        /* Called before addr is executed. */
        kprobe_pre_handler_t pre_handler;
        /* Called after addr is executed, unless... */
        kprobe_post_handler_t post_handler;
        /* Saved opcode (which has been replaced with breakpoint) */
        kprobe_opcode_t opcode;
        /* copy of the original instruction */
        struct arch_specific_insn ainsn;
        /*
         * Indicates various status flags.
         * Protected by kprobe_mutex after this kprobe is registered.
         */
        u32 flags;
};
```

Kprobe

과정

1. vmlinux

정확한 디버깅을 위해 리눅스 커널을 컴파일하고 빌드할 필요가 있음
리눅스 커널을 빌드하면 vmlinux라는 바이너리가 생김

vmlinux를 gdb로 디버깅한다면?

offset을 알 수 있음
inet_accept+offset

그리고 rbp에 접근해서 지역변수 체크!

```
pwndbg> disassemble inet_accept
Dump of assembler code for function inet_accept:
Address range 0xffffffff81de5ec0 to 0xffffffff81de605e:
   0xffffffff81de5ec0 <+0>:     endbr64
   0xffffffff81de5ec4 <+4>:     call   0xffffffff8109a640 <__fentry__>
   0xffffffff81de5ec9 <+9>:     push   rbp
   0xffffffff81de5eca <+10>:    movzx  ecx,cl
   0xffffffff81de5ecd <+13>:    mov    rbp,rsp
   0xffffffff81de5ed0 <+16>:    push   r14
   0xffffffff81de5ed2 <+18>:    push   r13
   0xffffffff81de5ed4 <+20>:    mov    r13,rdi
```

Kprobe

Debugging Kernel …

Next Time ~

Kprobe

Question

Question