

Lecture 14.1

Loadable Kernel Modules
your first code inside the kernel

Md Mahbub E Noor

Assistant Professor, Department of CSE
University of Barishal

```
● ● ● insmod hello.ko; dmesg (real)
hello: loading out-of-tree module
      taints kernel.
hello: loaded into the kernel,
      HZ = 1000
# QEMU guest, Linux 6.8.0-146
```

What you will learn today



Write, build, load and remove a kernel module; read its messages with dmesg.



Explain a .ko file as a relocatable ELF object, and how the kernel links it into itself.



Use module parameters, exported symbols and a /proc file to talk to a module.



Follow the rules of kernel C; read an oops and the taint flags.

Today's plan (60 min)

0-5

Recap of 13

5-10

Why modules

10-20

A first module

20-32

How insmod works

32-42

Parameters, exports, /proc

42-50

Kernel C, oops

50-60

Lab, checkpoint

Lab 14.1: sysprog_proc.c, a /proc file.

Quick recap of Lecture 13

Q1

exc printed trap 14, err 0x4 for read *NULL. Decode 0x4.

USER, read, page not present. The kernel turned it into SIGSEGV, SEGV_MAPERR.

Q2

Why must an interrupt handler never call mutex_lock?

It may sleep, and in interrupt context there is no process to put to sleep.

Q3

What is the IDT?

A table of 256 handler addresses; the CPU uses IDT[vector] for every exception and interrupt.

PART 1

Why modules?

Add code to a running kernel, and take it out again, without a reboot.

A module is a piece of the kernel you can plug in while it runs

Small kernel: Ubuntu's kernel image holds only what every PC needs; drivers for the rest are modules on disk.

Only what you have: When the kernel finds a device, udev loads the matching module automatically (Lecture 14.2).

Fast development: Change, rebuild, rmmod, insmod: seconds, not a reboot.

Same power as the kernel: A module runs in kernel mode with full rights. No protection at all: one bad pointer stops the machine.

Our rule: build on the lab PC, **load only in the QEMU guest** (Lab 7's VM). A crash there costs one reboot of the VM.

Analogy: a phone and its apps? Closer: a car and a new engine part fitted **while driving**. It works at full power, and if it is faulty the whole car stops.

SAME AS C

Like a shared library

Lecture 5.2: code loaded at run time, symbols resolved then. A module is the same idea for the kernel.

DIFFERENT FROM C

No process around it

A .so lives in one process's memory. A module lives in the kernel, shared by everything.

Book: Love, LKD Ch. 17 (modules); LDD3 Ch. 2

PART 2

A first module

Twenty lines of C, a two-line Makefile, and four commands.

hello.c and its Makefile

```
hello.c

#include <linux/init.h>
#include <linux/module.h>

static int __init hello_init(void)
{
    pr_info("hello: loaded into the kernel, HZ = %d\n", HZ);
    return 0;    /* < 0: insmod fails with that errno */
}

static void __exit hello_exit(void)
{
    pr_info("hello: removed\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```

```
Makefile (Kbuild)

obj-m += hello.o
KDIR ?= /lib/modules/$(shell uname -r)/build
all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
```

No main(): the kernel calls `hello_init` when loading, `hello_exit` when removing

No printf: `pr_info` writes to the kernel log, read with `dmesg`

__init: the code is freed after init: it runs once

Book: LDD3 Ch. 2; Love, LKD Ch. 17

Build it, load it, look at it, remove it

● ● ● make (real, shortened)

```
make -C /usr/src/linux-headers-6.8.0-146-generic M=... modules
CC [M]  hello.o
MODPOST Module.symvers
CC [M]  hello.mod.o
LD [M]  hello.ko
warning: the compiler differs ... (only the name differs)
```

● ● ● lsmod; cat /proc/modules (real)

Module	Size	Used by	Tainted: G
hello	12288	0	
hello 12288 0 - Live 0xffffffffc030e000 (OE)			

● ● ● QEMU guest, as root (real)

```
$ insmod hello.ko
hello: loading out-of-tree module taints kernel.
hello: module verification failed: signature ... missing
hello: loaded into the kernel, HZ = 1000
$ rmmod hello; dmesg | tail -1
[ 8.467078] hello: removed
```

● ● ● ls /sys/module/hello; cat ... (real)

coresize	holders	initsize	initstate	notes
refcnt	sections	srcversion	taint	uevent
initstate: live				
refcnt: 0				
taint: OE				
coresize: 12288				

printk levels, and what init's return value means

macro	level	use for
pr_emerg	0 KERN_EMERG	the system is unusable
pr_err	3 KERN_ERR	an error: something failed
pr_warn	4 KERN_WARNING	suspicious, but going on
pr_info	6 KERN_INFO	normal news (our hello)
pr_debug	7 KERN_DEBUG	only if dynamic debug is on

```
● ● ● include/linux/printk.h (real)
527 #define pr_info(fmt, ...) \
528     printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
```

```
● ● ● init returns -EINVAL (hello_param, real)
if (times < 1 || times > 5) {
    pr_err("... times must be 1..5, got %d\n", times);
    return -EINVAL;
}
$ insmod hello_param.ko times=9; echo $?
insmod: can't insert 'hello_param.ko': Invalid argument
22
```

SAME AS C

errno, from the other side

Lecture 9: system calls return `-errno` and `libc` sets `errno`. Now WE return `-EINVAL`, and `insmod`'s `finit_module` sees `errno 22 = EINVAL`.

If `init` fails, the module is not loaded at all and `exit` is never called: `init` must undo everything it did before the failure (in reverse order: Homework 14.2).

Book: LDD3 Ch. 2 and 4 (printk)

PART 3

How insmod works

A .ko is a .o file; the kernel is its linker and loader.

hello.ko is a relocatable ELF file, like hello.o (Lectures 5 and 6)

```
readelf -h -S hello.ko (real, shortened)
```

```
Type:      REL (Relocatable file)
Machine:   Advanced Micro Devices X86-64
[ 4] .init.text      PROGBITS   size 0x36   AX
[ 5] .rela.init.text RELA       4 entries
[ 6] .exit.text      PROGBITS   size 0x2a   AX
[12] .modinfo         PROGBITS   license, vermagic ...
[17] __versions       PROGBITS   CRCs of used symbols
[25] .gnu.linkonce.this_module struct module
```

```
modinfo hello.ko (real, shortened)
```

```
license:    GPL
depends:
vermagic:   6.8.0-146-generic SMP preempt mod_unload modversions
```

```
nm hello.ko (real, shortened)
```

```
0000000000000000 D __this_module
0000000000000010 t hello_init
0000000000000010 T init_module
0000000000000010 T cleanup_module
                U __printk
                U __fentry__
                U __x86_return_thunk
```

SAME AS C

U means "undefined"

Exactly like hello.o's U puts in Lecture 5.1: a hole that someone must fill. Here that someone is the kernel.

Book: CS:APP §7.3–7.7 (relocatable objects, symbols, relocation)

The hole for `_printk`, and how the kernel fills it

```
objdump -dr -j .init.text hello.ko (real)
0000000000000010 <init_module>:
 10: e8 00 00 00 00 call 15          R_X86_64_PLT32 __fentry__-0x4
 15: 55          push %rbp
 16: be e8 03 00 00 mov $0x3e8,%esi  # HZ = 1000
 1b: 48 c7 c7 00 00 00 00 mov $0x0,%rdi
                        R_X86_64_32S .rodata.str1.8
 25: e8 00 00 00 00 call 2a          R_X86_64_PLT32 _printk-0x4
 2a: 31 c0       xor %eax,%eax    # return 0
```

HZ is already 1000 in the code (0x3e8): a macro, decided at compile time from the headers. That is one reason a module must be built for **exactly** the kernel it will run in.

1 the kernel's own symbol table: `_printk` is at **ffffffff929b84b0** (/proc/kallsyms)

2 the call's 4 bytes sit at P = **ffffffffffc0314026** (hello_init's real address + 0x16)

3 write $S + A - P = \text{ffffffff929b84b0} - 4 - \text{ffffffffffc0314026} = \text{0xd26a4486}$

4 the bytes become **e8 86 44 6a d2**: the call reaches `_printk`

Lecture 5.1 again: $R_X86_64_PLT32 = S + A - P$, a 32-bit offset from the next instruction. The linker does it for programs; the kernel does it for modules.

modpost writes a second C file: hello.mod.c

```
hello.mod.c (generated; real, shortened)

MODULE_INFO(vermagic, VERMAGIC_STRING);
MODULE_INFO(name, KBUILD_MODNAME);

__visible struct module __this_module
__section(".gnu.linkonce.this_module") = {
    .name = KBUILD_MODNAME,
    .init = init_module,
    .exit = cleanup_module,
};

static const char ____versions[]
__used __section("__versions") =
    "\x10\x00\x00\x00\x7e\x3a\x2c\x12" "_printk\0"
    "\x18\x00\x00\x00\x13\xc9\x9a\x0d" "module_layout\0\0\0"
MODULE_INFO(depends, "");
```

struct module: the module's identity card: name, init, exit. The loader finds it in its own section.

vermagic: the kernel version and build options it was compiled for

__versions: a CRC for each kernel symbol used: 0x122c3a7e is a checksum of _printk's prototype

depends: other modules it needs (empty for hello; "counter" for user.ko, slide 20)

Inside the kernel: finit_module → load_module

```
kernel/module/main.c (real lines)

3172 SYSCALL_DEFINE3(finit_module, int, fd, ...)
3154 static int idempotent_init_module(...)
3126 static int init_module_from_file(...)
2831 static int load_module(struct load_info *info, ...)
2851     err = module_sig_check(info, flags);
2860     err = elf_validity_cache_copy(info, flags);
2864     err = early_mod_check(info, flags); /* vermagic */
2869     mod = layout_and_allocate(info, flags);
2918     err = simplify_symbols(mod, info);
2922     err = apply_relocations(mod, info);
2984     return do_init_module(mod);
2504 static noinline int do_init_module(struct module *mod)
2533     ret = do_one_initcall(mod->init);
2546     mod->state = MODULE_STATE_LIVE;
```

1 **check:** signature, ELF headers, vermagic, CRCs

2 **place:** allocate memory, copy the sections

3 **link:** look up every U symbol; patch every relocation

4 **run:** call init; if it returns 0 the module is Live; free .init.text

Remember these names: the oops on slide 24 prints exactly this chain: `do_one_initcall` ← `do_init_module` ← `load_module` ← ... ← `__x64_sys_finit_module`.

Book: Love, LKD Ch. 17 (loading modules)

Watch insmod hello.ko, step by step

insmod hello.ko, step by step

READCHECKPLACESYMBOLS**RELOCATE**INIT

hello.ko (ELF, type REL)

.init.text	hello_init
.exit.text	hello_exit
.rodata	"hello: loaded ..."
.modinfo	vermagic=6.8.0-146-generic ...
__versions	_printk 0x122c3a7e ...
.rela.init.text	call _printk at +0x26

\$ insmod hello.ko

```
fd = open("hello.ko")
finit_module(fd, "", 0)
```

kernel: load_module()

1. finit_module(fd): syscall 313
2. check ELF, vermagic, signature
3. allocate kernel memory, copy sections
4. look up undefined symbols
- 5. apply relocations**
6. call init: do_one_initcall

kernel memory (real addresses)

```
ffffffff929b84b0 T _printk (the kernel)

fffffffc0314010 hello_init (.init.text)
+0x25: e8 86 44 6a d2
call _printk

fffffffc030e000 hello_exit (.text)
fffffffc0310040 __this_module (.data)

hello: module verification failed:
signature ... missing - tainting kernel
```

Watch for

1. **finit_module**: one syscall.
2. **vermagic** and the signature.
3. Sections copied to **0xffffffffc0...**
4. **_printk** found, then the call **patched**.
5. **init** runs: Live.

The relocation R_X86_64_PLT32 says: at +0x26, write $S + A - P = \text{ffffffff929b84b0} - 4 - \text{fffffffc0314026}$. The four zero bytes become 86 44 6a d2: the call now reaches `_printk`. Exactly Lecture 5.1's rule.

Click the video to play or pause. About 1.2 minutes, no sound.

Built for another kernel? Three experiments (real)

● ● ● QEMU guest, Linux 6.8.0-146: patched copies of hello.ko

```
# 1. vermagic says PREEMPT instead of preempt
insmod: can't insert 'hello_badflags.ko': invalid module format
hello: version magic '6.8.0-146-generic SMP PREEMPT ...'
      should be '6.8.0-146-generic SMP preempt ...'
# 2. the CRC of _printk changed to 0
insmod: can't insert 'hello_badcrc.ko': Invalid argument
hello: disagrees about version of symbol _printk
# 3. vermagic says 6.8.0-999-generic
hello: loaded into the kernel, HZ = 1000      # !
```

● ● ● kernel/module/version.c (real)

```
80 int same_magic(const char *amagic,
      const char *bmagic, bool has_crcs)
82 {
83     if (has_crcs) {
84         amagic += strcspn(amagic, " ");
85         bmagic += strcspn(bmagic, " ");
87         return strcmp(amagic, bmagic) == 0;
```

Why did 3 load? With CRCs, the kernel skips the version number (up to the first space) and trusts the per-symbol CRCs instead.

The rule: build every module against the headers of the kernel that will load it: `/lib/modules/$(uname -r)/build`. The checks catch most mistakes, not all of them.

Taint: the kernel remembers what you did to it

kernel/panic.c and include/linux/panic.h (real)

```
466 const struct taint_flag taint_flags[...] = {
467     [ TAINT_PROPRIETARY_MODULE ] = { 'P', 'G', true },
474     [ TAINT_DIE ]                = { 'D', ' ', false },
479     [ TAINT_OOT_MODULE ]         = { 'O', ' ', true },
480     [ TAINT_UNSIGNED_MODULE ]    = { 'E', ' ', true },
panic.h 64 #define TAINT_DIE                7
        69 #define TAINT_OOT_MODULE      12
        70 #define TAINT_UNSIGNED_MODULE  13
```

C link: a bit set, like file permissions: tainted = $1 \ll 12 \mid 1 \ll 13 = 4096 + 8192 = 12288$. Ismod shows "Tainted: G" (GPL only, no P).

moment (QEMU guest)	/proc/sys/kernel/tainted	bits
after boot	0	none
after insmod hello.ko	12288	4096 (O) + 8192 (E)
after the oops (slide 24)	12416	+ 128 (D: died)

Why taint? When a kernel crashes, its developers first look at the taint letters. "OE" means: someone loaded code we have never seen. Taint never goes away until reboot, even after rmmod.

Book: [Documentation/admin-guide/tainted-kernels.rst](#) (kernel source)

PART 4

Talking to a module

Parameters in, symbols across, files out.

Module parameters: like argv for a module

● ● ● hello_param.c (core lines)

```
static char *who = "world";
static int times = 1;
module_param(who, charp, 0644); /* root may change it */
module_param(times, int, 0444); /* read-only */
MODULE_PARM_DESC(who, "whom to greet");
```

● ● ● change them later (real)

```
$ echo Dhaka > ../parameters/who
$ cat ../parameters/who
Dhaka
$ echo 2 > ../parameters/times
can't create ../times: Permission denied
```

● ● ● QEMU guest (real)

```
$ insmod hello_param.ko who=Barishal times=3
hello_param: hello, Barishal (1 of 3)
hello_param: hello, Barishal (2 of 3)
hello_param: hello, Barishal (3 of 3)
$ ls -l /sys/module/hello_param/parameters
-r--r--r-- 1 0 0 4096 times
-rw-r--r-- 1 0 0 4096 who
```

SAME AS C

Like argc/argv

main(int argc, char *argv[]) gets text; module_param converts the text to the type you name (charp, int, bool, ...) and stores it in your variable. The permission bits work like chmod: 0444 is read-only even for root.

EXPORT_SYMBOL: modules that use each other

counter.c / user.c (core lines)

```
/* counter.c */
int sysprog_next(void)
{ return atomic_inc_return(&value); }
EXPORT_SYMBOL_GPL(sysprog_next);
/* user.c: #include "counter.h" */
pr_info("... %d, %d, %d\n", sysprog_next(),
        sysprog_next(), sysprog_next());
```

Module.symvers after the build (real)

```
0x4fd609bc  sysprog_next  .../export/counter
EXPORT_SYMBOL_GPL
```

3, 2, 1, not 1, 2, 3? C does not say in which order a call's arguments are evaluated: gcc did them right to left here. Homework 14.1 task 4.

QEMU guest (real)

```
$ insmod user.ko
insmod: can't insert 'user.ko': unknown symbol in module...
user: Unknown symbol sysprog_next (err -2)
$ insmod counter.ko; insmod user.ko
user: sysprog_next() = 3, 2, 1
$ lsmod
user      12288  0
counter   12288  1 user
$ rmmod counter
rmmod: can't unload module 'counter': Resource temporarily unavailable
$ depmod; modprobe user; lsmod      # loads counter first
user      12288  0
counter   12288  1 user
```

A file made by your module: /proc/sysprog (Lab 14.1)

sysprog_proc.c (core lines)

```
static int sysprog_show(struct seq_file *m, void *v)
{
    unsigned long up = jiffies - loaded_at;
    seq_printf(m, "owner:      %s\n", owner);
    seq_printf(m, "reads:       %d\n", atomic_inc_return(&reads));
    seq_printf(m, "loaded for: %lu jiffies = %u ms (HZ = %d)\n", ..., ...);
    seq_printf(m, "reader:      %s (pid %d)\n", current->comm, current->pid);
    return 0;
}
/* init: */
if (!proc_create_single("sysprog", 0444, NULL, sysprog_show))
    return -ENOMEM;
/* exit: */
remove_proc_entry("sysprog", NULL);
```

QEMU guest (real)

```
$ insmod sysprog_proc.ko owner=Mahbub
$ cat /proc/sysprog
owner:      Mahbub
reads:      1
loaded for: 15 jiffies = 15 ms
reader:     cat (pid 136)
$ cat /proc/sysprog
reads:      2 ... reader: cat (pid 137)
$ rmmod sysprog_proc; cat /proc/sysprog
cat: can't open '/proc/sysprog': No such file...
```

SAME AS C

printf into a file

seq_printf works like fprintf. The text is made when someone reads: the file has size 0 (Lecture 12.1's /proc).

PART 5

Kernel C is different

The same language, without the safety net.

What changes when your C code runs in the kernel

user C (first semester)	kernel C (today)
printf, malloc, free, atoi, strcpy	pr_info, kmalloc / kfree, kstrtoint, strscpy: no libc at all
8 MiB stack, grows on demand	16 KiB stack per thread: no big local arrays, no deep recursion
float and double everywhere	no floating point (the FPU registers belong to user programs)
a bad pointer: SIGSEGV, only you die	a bad pointer: an oops, the kernel is damaged (next slide)
memory freed when the process exits	nothing freed for you: every kmalloc needs its kfree in exit
one thread unless you ask	always concurrent: other CPUs, interrupts → atomic_t, mutex, spinlock
errors in errno	return -EINVAL, -ENOMEM ...; NULL or ERR_PTR for pointers

Same C, though: structs, pointers, function pointers, macros, the preprocessor. Everything from your C course still applies; only the library and the safety net are gone.

Book: Love, LKD Ch. 2 (a beast of a different nature)

A NULL pointer in a module: the oops (real, QEMU guest)

insmod oops.ko; dmesg (real, shortened)

```
oops: about to read t->b
BUG: kernel NULL pointer dereference, address: 0000000000000004
#PF: supervisor read access in kernel mode
#PF: error_code(0x0000) - not-present page
Oops: 0000 [#1] PREEMPT SMP NOPTI
CPU: 0 PID: 211 Comm: insmod Tainted: G OE 6.8.0-146-generic
RIP: 0010:oops_init+0x15/0xff0 [oops]
Call Trace:
 do_one_initcall+0x5e/0x340
 do_init_module+0x97/0x290
 load_module+0xb5f/0xca0
 init_module_from_file+0x96/0x100
 idempotent_init_module+0x11c/0x310
 __x64_sys_finit_module+0x64/0xd0
 do_syscall_64+0x87/0x180
Modules linked in: oops(OE+) [last unloaded: sp_drv(OE)]
```

oops.c (the bug)

```
struct thing { int a, b; };
struct thing *t = find_thing();
/* returned NULL, never checked */
pr_info("... %d\n", t->b);
```

after the oops (real)

```
Killed # insmod, status 137
oops 16384 1 [permanent]
tainted: 12416 # + D
```

address 4 = offsetof(struct thing, b): NULL + 4. error_code 0:
kernel mode, read, not present (Lecture 13's bits).

LAB 14.1

sysprog_proc.c: your /proc file

A parameter in, a file out, built on the PC and loaded in the VM.

Lab 14.1 tasks, step by step

1 Demo: `cd demo/hello; make; ../../guest/run_guest.sh` . then `insmod, dmesg, lsmod, rmmod`

2 **TODO 1** in `sysprog_proc_starter.c`: `module_param(owner, charp, 0444)`

3 **TODO 2**: four `seq_printf` lines: `owner, reads, loaded for, reader`

4 **TODO 3**: `proc_create_single` in `init`; `NULL` → return `-ENOMEM`

5 **TODO 4**: `remove_proc_entry` in `exit`. Test: `insmod owner=..., cat twice, rmmod, cat`

Forget TODO 4 and the first `cat` after `rmmod` runs code that is gone: an oops. In the VM that is a lesson; on the PC it would be a crash.

● ● ● the guest, with the finished lab (real)

```
guest 6.8.0-146-generic: your modules are in /mod:
$ insmod sysprog_proc.ko owner=Mahbub
$ cat /proc/sysprog
owner:      Mahbub
reads:      1
loaded for: 15 jiffies = 15 ms (HZ = 1000)
reader:     cat (pid 136)
$ rmmod sysprog_proc
$ dmesg | tail -2
sysprog: /proc/sysprog created (owner=Mahbub)
sysprog: /proc/sysprog removed after 2 reads
```

Checkpoint: test yourself

- Q1** What kind of ELF file is hello.ko, and what fills its U symbols? → relocatable (REL); the kernel's loader, with its symbol table
- Q2** hello_init returns -ENOMEM. What does insmod see? Is exit called? → errno ENOMEM, "Cannot allocate memory"; no, nothing was loaded
- Q3** Why can't you rmmod counter while user is loaded? → user depends on it: refcount 1 ("Used by user")
- Q4** A module has a 10 KiB local array. Problem? → yes: the kernel stack is only 16 KiB
- Q5** After an oops in init, can you rmmod the module? → no: it is stuck as [permanent]; reboot the VM

Key ideas from today



A module: init and exit functions, built with Kbuild against the running kernel's headers.



A .ko is a relocatable ELF; `load_module` checks, places, links (symbols, relocations) and runs init.



Parameters in, `EXPORT_SYMBOL` across modules, `/proc` (or `sysfs`) files out to user space.



Kernel C has no libc, no FPU, a 16 KiB stack and no protection: one NULL is an oops.

Homework 14.1 (due before Lecture 14.2)

1. Read the oops

From the oops on slide 24: which function, which byte offset, which data address? Why 4? Decode `error_code(0x0000)` with Lecture 13's bits. Why did `insmod` end with status 137?

2. Version checks

Explain the three experiments on slide 16: which check failed in 1 and 2, and why 3 loaded although its `vermagic` was wrong (`version.c` lines 80–88).

3. Writable `/proc`

Make `/proc/sysprog` writable: `echo Rahim > /proc/sysprog` sets owner. Use `proc_create` with a `proc_ops` table and `copy_from_user`. Why not just `strcpy` from `buf`?

4. 3, 2, 1

`user.ko` printed `sysprog_next() = 3, 2, 1`. Explain with the C standard. Change `user.c` so that it prints 1, 2, 3 for sure.

5. Taint

Decode 12288 and 12416 bit by bit with `panic.h`. Which letters does `lsmod` show, and why does taint stay after `rmmod`?

Next lecture and reading

NEXT: LECTURE 14.2

The device driver model

Buses, devices, drivers and classes; how the kernel matches a driver to a device and calls probe; sysfs and uevents. Lab: a platform driver with its own sysfs file.

Kernel side

R. Love, **LKD**, Ch. 17 (modules today; the device model and sysfs for 14.2).

Driver book

Corbet, Rubini & Kroah-Hartman, **LDD3** (free), Ch. 2 today; Ch. 14 (the Linux device model) for 14.2.

Source

Linux 6.8: kernel/module/main.c, version.c, kernel/panic.c, include/linux/module.h (Linux v6.8 source online: elixir.bootlin.com/linux/v6.8).

Lab files

`lab14_files.zip`: guest/run_guest.sh, lab14_1/, demo/ (hello, param, export, oops).