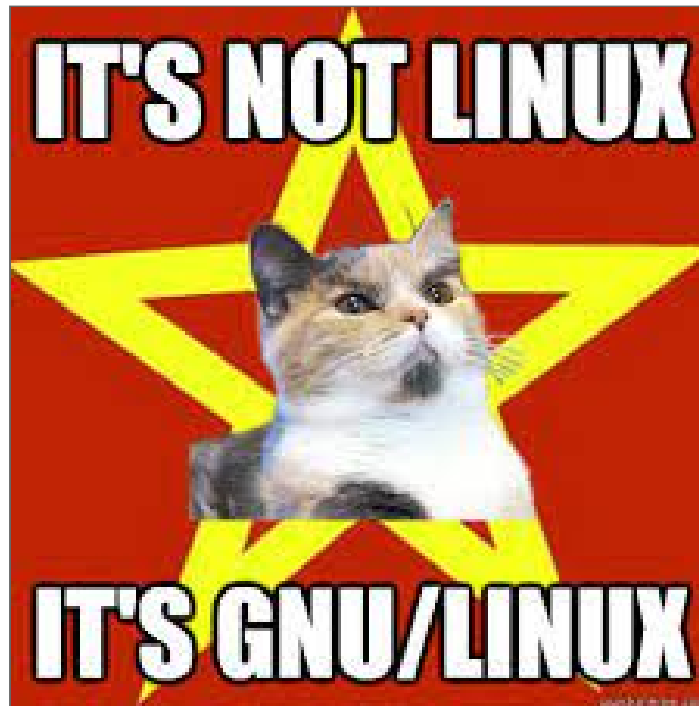


KERNEL LINUX - part 2



Introduction

Modules

Why modules?

- Modules make it easy to develop drivers without rebooting.
- Keeps the kernel image small and versatile.
- Reduce boot time: don't spend time initializing drivers, devices and kernel features you don't need now.

What is a module?

```
$ file hello.ko
hello.ko: ELF 64-bit LSB relocatable, x86-64,
          version 1 (SYSV), not stripped
$ modinfo ./hello.ko
filename: /home/halfr/hello/./hello.ko
author:   Lionel Auroux
description: Hello module
license:   GPL
depends:
vermagic: 4.0.3-1-ARCH SMP preempt mod_unload
          modversions
```

Build system

- The Linux build system is complex, yet is very easy to use, and almost always produce the desired result.
- Compiling a module is trivial:

Makefile

```
obj-m := hello.o
```

- Build

```
$ make -C ~/linux M=`pwd` modules
```

- Build system
- The Linux build system is complex, yet is very easy to use, and almost always produce the desired result.
- Compiling a module is trivial:

Makefile

```
obj-m := hello.o  
hello-objs := hello-file1.o hello-file2.o
```

- Build

```
$ make -C ~/linux M=`pwd` modules
```

- Build system
- Using a variable from Kconfig:
- *Kconfig*

```
config HELLO  
    tristate "Build with the hello support?"
```

- *Makefile*

```
obj-$(CONFIG_HELLO) := hello.o  
hello-objs := hello-file1.o hello-file2.o
```


Generated files

- *hello.o*: The compiled module.
- *hello.mod.{c,o}*: Additional sections and metadata to be linked in the module
- *hello.ko*: The linked module.
- *Module.symvers*: Symbol versions
- *modules.order*: Lists modules that appears in the *Makefile*, used in case of duplicate module match by *modprobe*.
- *.tmp_versions* and *.*.cmd*: Miscellaneous files.

Cryptographic signing of modules

- Since Linux 3.7
- Requires `CONFIG_MODULE_SIG=y`.
- Cryptographically signs modules during installation (*make modules_install*)
- Uses RSA and SHA-{1,224,256,384,512}.
- The private key is only needed during the build, after which it can be deleted or stored securely.
- The public key gets built into the kernel so that it can be used to check the signatures as the modules are loaded.
- See public keys in `/proc/keys`.

Inserting a module

- Userspace tools: *insmod* and *modprobe*.

modprobe will try to insert other modules so

- *that all symbol are resolvable.*

- They use the *init_module(2)* syscall, which performs (roughly):

- *signature checks (if enabled)*
- *mernel memory allocation*
- *module .text section copy*
- *license and version checks*
- *symbol resolving using the kernel symbol*
- *table*
- *sysfs and internal registrations*

Unloading a module

- Userspace tools: *rmmod* and *modprobe -r*
- Uses *delete_module(2)*.
- Won't work if the kernel thinks the module is still in use:

- The module is a dependency of another*
- *module.*
 - *A file descriptor is owned by this module.*
 - *...*

Versions

- A module has to be recompiled for each version of the kernel that you want to link it to.
- If `CONFIG_MODVERSIONS=y`, a simple ABI consistency check is performed over the prototypes of exported symbols.

When building a module, a CRC is computed for each exported symbol, and stored in `Module.symvers` and in the

- *generated `module.mod.c`.*

- Versions checks can be bypassed, but it is not wise to do so.

modalias

- What peripherals does a module handle? *modalias*!

```
cat /sys/devices/pci0000:00/0000:00:1d.0/usb4/[...]  
usb:v05E3p0608d8537dc09dsc00dp01ic09isc00ip00in00
```

- *usb* is the device class, the rest is class-specific (*vendor id*, *device id*, etc.)
- A module defines a *MODULE_ALIAS*("usb:...") that pattern-matches the *modalias* class-specific string.
- When a new device is detected, the module loader is called with the *modalias* string and loads the matching module.
- See devices supported by *modalias*: *less /lib/modules/\$(uname -r)/modules.alias*
- Example:

```
alias pci:v00001002d000099A4sv*sd*bc*sc*i* radeon
```


Parameters

- Parameters are typed *key=value* settings.
- They are defined as such:

```
static char *asso = "Triton.";
static int votes = 1;
module_param(triton, charp, S_IRUGO)
module_param(votes, int, S_IRUGO)
```

- Available types: *bool*, *invbool*, *charp* (memory is auto-allocated), *int*, *long*, *short*, *uint*, *ulong*, *ushort*
- Arrays are also supported:

```
module_param_array(name, type, num, perm);
```

- The parameters will appear in */sys/module/MODULE/parameters/PARAM*.

- The final field controls the permission value of this file.

The “misc” device class

Description

- *misc* is a type of char devices.
- Used for *small* drivers:
 - *Various mice*
 - *Watchdog*
 - *Clocks*
 - *Control device for other modules*
 - *...*
- They have a name.
- Appear at */dev/NAME*.
- Can be integrated into *modalias*: *MODULE_ALIAS_MISCDEV(minor)*:
 - *char-major-10-minor*

Device definition

```
#include <linux/miscdevice.h>

static struct miscdevice mymisc = {
    .minor = MISC_DYNAMIC_MINOR,
    .name  = "mymisc",
    .fops  = &mymisc_fops,
};
```

- Minor (and major) numbers are described in the next lectures.

Registering a misc device

```
int misc_register(struct miscdevice *misc);

static int __init mymisc_init(void)
{
    int r;

    r = misc_register(&mymisc);
    if (r < 0) {
        pr_warn("misc_register() failed: %d\n", r);
        return r;
    }

    return 0;
}
```


File operations

```
static struct file_operations mymisc_fops = {  
    .owner          = THIS_MODULE,  
    .read           = mymisc_read,  
    .write          = mymisc_write,  
    .open           = mymisc_open,  
    .release        = mymisc_release,  
};
```

open() and *release()*

```
int mymisc_open(struct inode *i, struct file *f);
```

- Called when user space opens */dev/mymisc*.
- *struct inode* represents a element of the file system.
- *struct file* is created every time a file is opened.
- Allows initializing the *private_data* member of *struct file*.

```
int mymisc_release(struct inode *i, struct file *f);
```

- Called when user space closes the file.

read()

```
ssize_t mymisc_read(struct file *file,  
                    char __user *buf,  
                    size_t count,  
                    loff_t *off);
```

- Called when user space uses the *read()* syscall on the device.
- *buf*: buffer of the user (cf. *__user*).
- *count*: size of the buffer.
- *ppos*: must be updated to keep the current location.
- Returns the number of bytes read.

write()

```
ssize_t mymisc_write(struct file *file,  
                     char __user *buf,  
                     size_t count,  
                     loff_t *off);
```

- Called when user space uses the *write()* syscall on the device.
- The opposite of *read*, must read at most *count* bytes from *buf* and write it to the device, update *off* and return the number of bytes written.