Introduction to Kernel Module

cfmc30

March 3, 2025

Outline

1. What is a Kernel Module

2. Kernel Module Implementation

3. Character Device Drivers

4. Memory

5. Q/A

Presentation agenda

1. What is a Kernel Module
2. Kernel Module Implementation
3. Character Device Drivers
4. Memory
5. Q/A

What is a Kernel Module

- A code segment that can be dynamically loaded and unloaded within the kernel as needed.
- Role of kernel module
  - Device Driver
    Enables interaction with hardware components connected to the system.
  - File System
    Extends or modifies file system support.
  - ...
- Without Kernel Modules:
  - All functionality must be integrated into the kernel image.
    - Large kernel image
    - Rebuild kernel when need new functionality

Kernel Module in Linux

- Checking Available Kernel Modules
  - List all available kernel modules in the system:

    ```
    $ find /lib/modules/$(uname -r) -type f -name '*.ko'
    ```

  - Check currently loaded kernel modules:

    ```
    $ lsmod
    ```

  - Alternatively, check module info in:

    ```
    $ cat /proc/modules
    ```

Kernel Module in Linux

- Loading and Unloading Kernel Modules
  - Load a module manually:

    ```
    # insmod <module.ko>
    ```

  - Remove a loaded module:

    ```
    # rmmod <module>
    ```

  - Load/unload modules with automatic dependency handling:

    ```
    # modprobe <module>
    ```

Presentation agenda

1. What is a Kernel Module
2. Kernel Module Implementation
3. Character Device Drivers
4. Memory
5. Q/A

Programming Language

- C
- Rust

Development and Test Environment

- Kernel development is conducted using the QEMU emulator.
- The professor provides a script for setting up the development environment:
    - Provided files:

      `dist-6.6.17.tbz`

      `hellomod-6.6.17.tbz`
- Demo

Makefile

```
obj-m     += hellomod.o
ccflags-y += -DEXPORT_SYMTAB
all: hello
        make -C ../dist/modulebuild M=$(PWD) modules
hello: hello.c
        $(CROSS_COMPILE)gcc -o $@ $< -Wall -static
clean:
        rm -f hello
        make -C ../dist/modulebuild M=$(PWD) clean
install: all
        mkdir -p ../rootfs/modules
        cp hello hellomod.ko ../rootfs/modules
```

Module Initialization and Cleanup

- Register `init` and `cleanup` functions:

```
module_init(hellomod_init);
module_exit(hellomod_cleanup);
```

- `__init` and `__exit` macros:
    - `__init` is discarded after execution when built as a built-in driver.
    - `__exit` is ignored when the module is built into the kernel.

Licensing

- Specify the module license using:

  `MODULE_LICENSE("GPL");`
- Common license examples:
  - `"GPL"`
  - `"GPL v2"`
  - `"GPL and additional rights"`
  - `"Dual BSD/GPL"`
  - `"Dual MIT/GPL"`
  - `"Dual MPL/GPL"`
  - `"Proprietary"`

Functions Available to Modules

- Standard C library functions like `printf()` are not available.
- Modules can only use functions exported by the kernel:
  - `EXPORT_SYMBOL()` / `EXPORT_SYMBOL_GPL()`
    - Proprietary modules cannot use symbols exported with `EXPORT_SYMBOL_GPL()`.
  - Available kernel symbols can be found in:

    ```
    cat /proc/kallsyms
    ```

Presentation agenda

1. What is a Kernel Module
2. Kernel Module Implementation
3. Character Device Drivers
4. Memory
5. Q/A

The file_operations Structure

- Defined in `include/linux/fs.h`
- Used to specify available operation handlers for a device.

Registering a Device

```
// create char dev
if(alloc_chrdev_region(&devnum, 0, 1, "updev") < 0)
        return -1;
// type of device (e.g. audio, network)
if((clazz = class_create("upclass")) == NULL)
        goto release_region;
clazz->devnode = hellomod_devnode;
if(device_create(clazz, NULL, devnum, NULL, "hello_dev")
   == NULL)
        goto release_class;
cdev_init(&c_dev, &hellomod_dev_fops);
if(cdev_add(&c_dev, devnum, 1) == -1)
        goto release_device;
```

ioctl

- Provides an out-of-band communication channel for device control.
- Macros for defining ioctl operations:

macro	Usage
_IO	an ioctl with no parameters
_IOW	an ioctl with write parameters (copy_from_user)
_IOR	an ioctl with read parameters (copy_to_user)
_IOWR	an ioctl with both write and read parameters.

ioctl

- We use macro to adding new ioctl's to the kernel.

example_ioctl.h

```
#define IOC_MAGIC '\x66'
#define IOCTL_VALSET _IOW(IOC_MAGIC, 0, struct ioctl_arg)
#define IOCTL_VALGET _IOR(IOC_MAGIC, 1, struct ioctl_arg)
#define IOCTL_VALGET_NUM _IOR(IOC_MAGIC, 2, int)
#define IOCTL_VALSET_NUM _IOW(IOC_MAGIC, 3, int)
```

ioctl

- Handling `ioctl` calls in kernel space:

```
static long dev_ioctl(struct file *file,
    unsigned int cmd, unsigned long arg) {
    struct ioctl_arg temp_arg;
    switch (cmd) {
        case IOCTL_VALSET:
          // copy struct ioctl_arg from user
            break;
        case IOCTL_VALSET_NUM:
            break;
        default:
            return -EINVAL;
    }
    return 0;
}
```

Using `ioctl` in User Space

- Once implemented in the kernel, you can invoke it from user space:

```
#include "example_ioctl.h"

ioctl(fd, IOCTL_VALSET, &arg);
ioctl(fd, IOCTL_VALSET_NUM, &num_val);
```

Presentation agenda

1. What is a Kernel Module
2. Kernel Module Implementation
3. Character Device Drivers
4. Memory
5. Q/A

User-Space and Kernel-Space

- Memory in kernel space and user space is not shared.
  - You cannot directly dereference a user-space pointer in kernel space.
- To transfer data between user space and kernel space, use:
  - `copy_from_user()` – Copies data from user space to kernel space.
  - `copy_to_user()` – Copies data from kernel space to user space.

Presentation agenda

1. What is a Kernel Module
2. Kernel Module Implementation
3. Character Device Drivers
4. Memory
5. Q/A

Q/A