ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

HỆ ĐIỀU HÀNH

ĐỒ ÁN

TÌM HIỂU VÀ LẬP TRÌNH LINUX KERNEL MODULE

Thành viên nhóm:

Nguyễn Trung Kiên – 18127123

Nguyễn Quang Pháp – 18127174

Nguyễn Hà Thành – 18127214

Thành phố Hồ Chí Minh - 2020

NỘI DUNG TÌM HIỂU

- Khái niệm Linux Kernel:
  - Process management
  - Memory management
  - Device management
  - File system management
  - Networking management
  - System call interface

- Các khái niệm khác:
  - User space và kernel space
  - User mode và kernel mode
  - System call và ngắt
  - Process context và interrupt context

- Loadable Kernel Module

- Character Device Driver:
  - Device file
  - Device number

CHI TIẾT TỪNG PHẦN

1. **Khái niệm Linux kernel**: Linux Kernel được chia làm 6 thành phần:

- *Process management*: có nhiệm vụ quản lý các tiến trình, bao gồm các công việc:

  + Tạo/hủy các tiến trình.

  + Lập lịch cho các tiến trình (CPU sẽ thực thi chương trình khi nào, thực thi trong bao lâu, tiếp theo là chương trình nào).

  + Hỗ trợ các tiến trình giao tiếp với nhau.

  + Đồng bộ hoạt động của các tiến trình xảy ra tranh chấp tài nguyên.

- *Memory management*: có nhiệm vụ quản lý bộ nhớ, bao gồm các công việc:

  + Cấp phát bộ nhớ trước khi đưa chương trình vào, thu hồi bộ nhớ khi tiến trình kết thúc.

  + Đảm bảo chương trình nào cũng có cơ hội được đưa vào bộ nhớ.

  + Bảo vệ vùng nhớ của mỗi tiến trình.

- *Device management*: có nhiệm vụ quản lý thiết bị, bao gồm các công việc:

  + Điều khiển hoạt động của các thiết bị.

  + Giám sát trạng thái của các thiết bị.

  + Trao đổi dữ liệu với các thiết bị.

  + Lập lịch sử dụng các thiết bị, đặc biệt là thiết bị lưu trữ (ví dụ ổ cứng).

- *File system management*: có nhiệm vụ quản lý dữ liệu trên thiết bị lưu trữ (như ổ cứng, thẻ nhớ) gồm các công việc: thêm, tìm kiếm, sửa, xóa dữ liệu

- *Networking management*: có nhiệm vụ quản lý các gói tin (packet) theo mô hình TCP/IP.

- *System call interface*: có nhiệm vụ cung cấp các dịch vụ sử dụng phần cứng cho các tiến trình. Mỗi dịch vụ được gọi là một **system call**.

2. Các khái niệm khác:

- Bộ nhớ RAM chứa các lệnh/dữ liệu dạng nhị phân của Linux kernel và các tiến trình. RAM chia làm 2 miền:

  + Kernel space là vùng không gian chứa các lệnh và dữ liệu của kernel.

  + User space là vùng không gian chứa các lệnh và dữ liệu của các tiến trình.

- CPU có 2 chế độ thực thi:

  + Khi CPU thực thi các lệnh của kernel, thì nó hoạt động ở chế độ **kernel mode**, CPU sẽ thực hiện bất cứ lệnh nào trong tập lệnh của nó và CPU có thể truy cập bất cứ địa chỉ nào trong không gian địa chỉ.

  + Khi CPU thực thi các lệnh của tiến trình, thì nó hoạt động ở chế độ **user mode**, CPU chỉ thực hiện mooth phần tập lệnh của nó và CPU cũng chỉ được phép truy cập một phần không gian địa chỉ.

- System call và ngắt:

  + Khi một tiến trình cần sử dụng một dịch vụ nào đó của kernel, tiến trình sẽ gọi một system call. System call cho người dùng cách tiếp cận những tiện ích/dịch vụ của hệ điều hành, yêu cầu hệ điều hành thực hiện tác vụ cho mình. Các system call được cung cấp bởi kernel, khi được tiến trình gọi các system call, CPU chuyển sang chế độ kernel mode để thực thi các lệnh của kernel cho xong hoàn toàn thì CPU mới chuyển sang chế độ user mode để thực thi tiếp các lệnh của tiến trình.

  + Ngắt là một sự kiện làm gián đoạn hoạt động bình thường của CPU, buộc CPU phải chuyển sang thực thi một đoạn mã lệnh đặc biệt để xử lý sự kiện đó. Khi một thiết bị muốn trao đổi dữ liệu với CPU, nó sẽ gửi một tín hiệu ngắt tới CPU, khi đó CPU sẽ ngừng thực thi các lệnh của tiến trình lại, chuyển sang chế độ kernel mode rồi thực thi một chương trình đặc biệt của kernel để xử lý tín hiệu ngắt đó cho xong rồi CPU mới trở lại chế độ user mode và tiếp tục thực hiện các lệnh tiếp theo của tiến trình.

- Process context và interrupt context:

+ Một trong những nhiệm vụ chính của process là thực thi lệnh chương trình. Những lệnh này được đọc từ một file thực thi và thực thi cùng với không gian địa chỉ của chương trình. Một chương trình thông thường như thế xảy ra ở user space. Khi một chương trình thực hiện gọi tới system call, nó nhảy vào kernel space. Ở thời điểm này, kernel thay mặt cho process thực thi chương trình và ngữ cảnh tại đây gọi là process context.

+ Khi thực thi một trình phục vụ ngắt, đơn giản là kernel đang ở trong interrupt context. Interrupt context không liên kết gì với một process, vô hiệu hóa bộ lập lịch, nó tồn tại một cách độc lập với process context với mục đích giúp trình phục vụ ngắt phản hồi thật nhanh. Interrupt context vô cùng khắt khe về mặt thời gian vì trình phục vụ ngắt đã ngắt một chương trình khác, vậy nên trình phục vụ phải nhanh và đơn giản.

3. **Loadable Kernel Module**: còn được gọi là linux kernel module là một file với tên mở rộng là (.ko). Nó có thể được lắp vào hoặc tháo ra khỏi kernel khi cần thiết. Loadable kernel module giúp bổ sung code vào linux kernel tring quá trình vận hành. Các module này thường được sử dụng để hỗ trợ cho driver thiết bị, driver filesystem hoặc lệnh hệ thống.

4. **Character Device Driver**: là các device được truy cập như một luồng nhị phân và có nhiệm vụ thực hiện những thao tác đọc ghi này:
   - *Device file*: được hiểu là nơi để user application giao tiếp với device driver. Giả sử, khi user application muốn đọc dữ liệu từ thiết bị thì gọi hàm system call để tương tác device file, rồi nạp entry point read của device driver để đọc dữ liệu.
   - *Device number*: là một bộ số gồm hai số major number và minor number:
     - Major number: giúp Kernel nhận biết device driver nào tương ứng với device file nào.
     - Minor number: giúp device driver nhận biết nó cần điều khiển thiết bị nào, nếu như device driver đang điều khiển nhiều thiết bị.

GIẢI THÍCH SOURCE CODE

1. **Ý tưởng**: Thiết bị này sẽ có các entry point open (mở), read (đọc dữ liệu lên) và release (đóng). Trong hàm read, cứ mỗi lần được gọi, ta sẽ dùng hàm *get_random_bytes* của thư viện <linux/random.h> để sinh được một số ngẫu nhiên ở kernel space; thực chất hàm *get_random_bytes* này trả về một số lượng bytes ngẫu nhiên theo yêu cầu, và lưu trong một buffer. Vậy trong hàm read ta chỉ cần return số bytes này là đã được một số ngẫu nhiên.

2. **Tóm tắt bước làm:**
   - B1**:** Tạo driver với cặp số <major, minor> cùng với các entry point cần thiết (open, release, read,…)
   - B2: Tạo device file để quản lí driver đã tạo trên
   - B3: Viết chương trình giao tiếp giữa người dùng và kernel

3. **Chi tiết source code**:

- Trước tiên cần tham chiếu tới file của Linux kernel là **<linux/module.h>** chứa 2 macro quan trọng là: **module_init** và **module_exit**

```
module_init(random_driver_init);
module_exit(random_driver_exit);
```

- module_init xác định hàm sẽ được thực thi ngay khi ***lắp*** module vào kernel.
- module_exit xác định hàm sẽ được thực thi ngay khi ***tháo*** module ra kernel.

- Trong trường hợp này, ngay khi lắp module vào kernel, hàm **random_driver_init** sẽ được gọi, mục đích hàm này là để khởi tạo driver, đồng thời xin cấp pháp các thông số **<major, minor>** cho device.

```
/* ham khoi tao driver */
static int __init random_driver_init(void)
{
        /* cap phat device number */
        major = register_chrdev(0, "random_device", &fops);
        if (major < 0)
                printk("Fail to register device number dynamically\n");
        else
                printk("Allocated device number (%d,0)\n", major);

        return 0;
}
```

- *int register_chrdev (int major, char * name, const struct file_operations * fops)* để đăng kí một major number cho device này, trong đó:

+ major: số major của device; hoặc bằng **0** để cấp phát động

+ name: tên của device (do người lập trình đặt).

+ file_operations: là một cấu trúc để liên kết với thiết bị.

```
static struct file_operations fops = {
        .owner = THIS_MODULE,
        .open = random_driver_open,
        .release = random_driver_release,
        .read = random_driver_read,
};
```

- Hiểu một cách đơn giản; các system calls open, release, read đã được kết nối với hàm tương ứng; khi ở user_space gọi system calls này thì hàm kết nối với nó sẽ được thực thi.
- Ví dụ, system calls **open** kết nối với hàm **random_driver_open,** khi user_space gọi system calls open thì random_driver_open ở kernel sẽ được thực thi.
- Để cấp phát động/giải phóng major number, ta cần tham chiếu tới thư viện **<linux/fs.h>**

- Các entry point mà device này có

```
/* cac ham entry points */
static int random_driver_open(struct inode *inode, struct file *flip) {
        printk("Random device is opening\n");
        return 0;
}

static int random_driver_release(struct inode *inode, struct file *flip) {
        printk("Closing device successfully\n");
        return 0;
}

static ssize_t random_driver_read(struct file *flip, char *user_buf, size_t len, loff_t *off) {
        int rand;
        get_random_bytes(&rand, sizeof(rand));
        return rand;
}
```

Hàm **random_driver_read** như đã nói ở phần *ý tưởng*, đơn giản chỉ là return một số bytes ngẫu nhiên và hiển thị cho người dùng ở user_space thấy.

+ int (*open) (struct inode *inode, struct file *filp)

- *Chức năng*: khi một tiến trình trên user space gọi system call openđể mở device file tương ứng với char driver này, thì hàm này sẽ được gọi để thực hiện một số việc như kiểm tra thiết bị đã sẵn sàng chưa, khởi tạo thiết bị nếu cần, lưu lại minor number…
- *inode: địa chỉ của cấu trúc inode. Cấu trúc này dùng để mô tả các file trong hệ thống.
- *filp: địa chỉ của cấu trúc file. Cấu trúc file dùng để mô tả một file đang mở.
- *Giá trị trả về*: Hàm này trả về 0 để thông báo mở device file thành công. Ngược lại, trả về một số khác 0 để thông báo mở device file thất bại.

+ int (*release) (struct inode *inode, struct file *filp)

- *Chức năng* ngược lại với hàm open

+ ssize_t (*read) (struct file *filp, char __user *buff, size_t size, loff_t *off)

- *Chức năng*: đọc dữ liệu từ buffer của char device vào kernel buffer, sau đó, sao chép dữ liệu từ kernel buffer vào trong user buffer của tiến trình.

- filp: địa chỉ của cấu trúc file, cấu trúc này mô tả một device file đang mở.
- *buff: địa chỉ của user buffer.
- size: số lượng byte dữ liệu mà tiến trình cần đọc.
- *off: địa chỉ của cấu trúc loff_t. Cấu trúc này cho biết vị trí trên buffer của char device mà dữ liệu bắt đầu được đọc ra.

- Khi remove module ra khỏi kernel, hàm sau sẽ được tự động thi hành, thông báo driver đã được gỡ thành công.

```
/* ham ket thuc driver */
static void __exit random_driver_exit(void) {
        unregister_chrdev(major, "random_device");
        printk("~ Exit random driver ~\n");
}
```

unregister_chrdev (<số major>, <tên thiết bị>) thu hồi lại số major đã cấp

- Ở user_space sẽ viết một chương trình đơn giản để giúp cho người dùng giao tiếp với device này (chi tiết xem trong source code).