دانشگاه صنعتی اصفهان

دانشکده برق و کامپیوتر

آزمایشگاه سیستم عامل

دستورکار جلسه هفتم

پاییز ۱۴۰۲

LKM چیست؟

یکی از قابلیت های خوب لینوکس امکان توسعه کرنل هنگام بالا بودن سیستم عامل است. یعنی میتوان حین اجرای سیستم عامل قابلیت هایی را به آن اضافه یا از آن کم کرد (می توانید در موردUptime بالای سرورهای لینوکسی تحقیق کنید). به قطعه کدهایی که حین اجرا به کرنل افزوده میشوند ماژول گفته می شود. کرنل لینوکس از انواع مختلف ماژول (مثلا درایورها) پشتیبانی میکند. هر ماژولی ازObject Code تشکیل شده است که می تواند به صورت پویا به کرنل لینک شود، بدون نیاز به کامپایل دوباره کل کرنل. پس از اضافه شدن یک ماژول به کرنل، اپلیکیشن های فضای کاربر می توانند از آن ماژول استفاده کنند.

دیوایس‌ها و درایورها

تقریباً هر عملیات سیستمی نهایتاً با یک دستگاه فیزیکی کار خواهد داشت. تمامی عملیات های کنترل دستگاه (به جز دستگاه هایی مثل پردازنده و حافظهٔ اصلی) توسط قطعه کدهایی انجام میشود که مخصوص به دستگاه هدف (دستگاهی که عملیات رو آن انجام می شود) است. به این قطعه کدها درایور گفته میشود. کرنل باید برای تمامی دستگاه های متصل به سیستم درایور مخصوص به خودشان را داشته باشد) مثلا برای ماوس و کیبورد و درایو باید درایور داشته باشد.

انواع دیوایس

در لینوکس به طور کلی سه مدل دستگاه تعریف میشود. هر ماژول هم معمولاً فقط تحت یکی از این سه مدل توسعه می یابد که نتیجهٔ آن سه دسته ماژول کاراکتری (Char Module)، بلوکی (Block Module) و شبکه ای (Network Module) است. البته میتوانیم این دسته بندی را رعایت نکنیم و ماژولی بنویسیم که بتواند قابلیت هایی از هر سه دسته داشته باشد اما این ماژول مقیاس پذیر و توسعه پذیر نخواهد بود.

کاراکتر دیوایس

یک دستگاه کاراکتری دستگاهی‌ست که بتوان با آن مثل یک فایل رفتار کرد؛ یعنی مثل یک جریانی از بایت ها (Stream of Bytes). یک درایور کاراکتری چنین رفتار فایل مانندی را برای این دستگاه کنترل و پیاده سازی می کند. این درایورهای کاراکتری معمولاً فراخوانی های سیستمی بازکردن(open)، بستن (close)، خواندن (read) و نوشتن (write) را پیاده سازی می کنند. به عنوان مثال کنسول متنی (/dev/console) و پورت های سریال (/dev/tty0 و مشابه‌های آن) دستگاه های کاراکتری هستند.

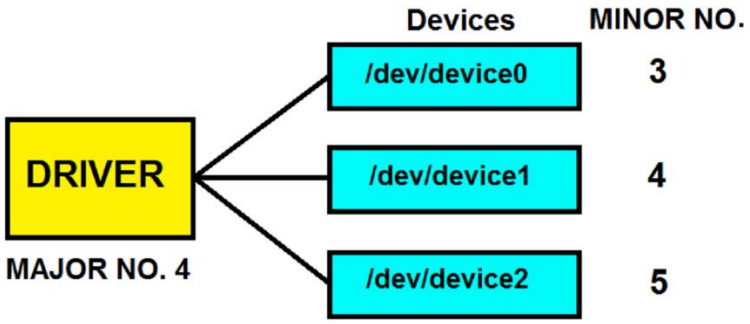
دسترسی به دستگاه های کاراکتری به کمک گره های فایل سیستم (Filesystem Nodes) انجام می شود. تنها تفاوت قابل توجه بین دستگاه های کاراکتری و فایل های معمولی این است که در فایل های معمولی می توان به عقب و جلو حرکت کرد اما معمولاً دستگاه های کاراکتری کانال های داده ای هستند که فقط به صورت سری (Sequentially) قابل دسترسی هستند.

البته دستگاه های کاراکتری‌ای هم وجود دارد که مثل نواحی داده‌ای (Data Area) رفتار میکنند و میتوان در آنها به عقب و جلو حرکت کرد.

اعداد ماژور و مینور

به هر درایور در سیستم یک عدد یکتا تخصیص داده می شود که به آن Major Number می‌گویند. بدین ترتیب موقع load هر درایور در سیستم باید یک عدد ماژور آزاد به آن تخصیص داده شود. توجه داشته باشید که دیوایس های سخت افزاری مختلفی می توانند از طریق یک نوع درایور کنترل شوند. مثلا تصور کنید دو هارد اکسترنال مشابه به سیستم شما وصل باشد. این دو هارد اکسترنال از یک درایور یکسان استفاده می کنند اما سیستم باید راهی جهت تفکیک این دو دیوایس داشته باشد. به همین دلیل عدد دیگری با نام Minor Number برای دیوایس های مختلفی که از یک درایور واحد استفاده می کنند در نظر گرفته می شود.

کرنل از عدد ماژور استفاده میکند تا درایور مرتبط را پیدا کند و درایور از عدد مینور جهت کار با دستگاه مشخصی استفاده میکند.



یک فایل خاص و روش ایجاد آن

یکی از مفاهیم مهم در سیستم عامل های مبتنی بر یونیکس مفهوم فایل بودنِ تقریباً همه چیز است. یعنی منابع ورودی/خروجی مختلفی (مثل اسناد، دایرکتوری‌ها، درایوها، مودم، کیبورد، پرینتر و حتی برخی ipcها و ارتباطات شبکه ای) وجود دارد که همگی جریان های بایتی ساده ای هستند. مزیت چنین رویکردی این است که ابزارها و APIهای یکسانی را میتوان برای دسترسی و ارتباط با چندین منبع مختلف استفاده کرد.

البته میتوان این مفهوم را دقیقتر هم بیان کرد و گفت هر چیزی یک File Descriptor است. چرا که هنگام باز کردن یک فایل معمولی یا ایجاد پایپ های ناشناس یا ساخت سوکت شبکه، File Descriptorهایی ساخته می شود که راه ارتباطی و رابط بین کد با آن منبع خواهد بود.

دستگاه های کاراکتری از طریق اسم شان در فایل سیستم قابل دسترسی هستند و میتوان با آنها مثل یک فایل رفتار کرد. این اسامی را "فایل های خاص"، "فایل های دستگاهی" یا حتی "گره‌هایی در درخت فایل سیستم" گوییم. اما این فایل خاص

کجاست؟ معمولاً فایل درایور مرتبط با هر دیوایس در دایرکتوری "/dev" قرار دارد. ls /dev -l را اجرا کنید تا فایل‌های مربوط به ماژول های کنونی سیستم تان را مشاهده کنید. همانطور که می بینید اولین کاراکتر از رشته permission هر فایل، مشخص کننده نوع دیوایس یا فایل ماژول است (c به معنی دیوایس کاراکتری و b به معنی دیوایس بلوکی). همچنین غیر از نام فایل، دو ستون عددی وجود دارد که یکی بیانگر عدد ماژور و دیگری بیانگر عدد مینور دیوایس است. همانطور که گفتیم دیوایس های مختلفی ممکن است از یک ماژول استفاده کنند که بدین ترتیب همه دارای یک عدد ماژور ولی عددهای متفاوت مینور هستند.

هرگاه دیوایسی به سیستم اضافه می شود باید حتما فایل درایور متناظرش در شاخه /dev قرارگیرد و در واقع از طریق نام همین فایل است که در کد اپلیکیشن می توانیم مشخص کنیم با کدام دیوایس کار داریم و عملیات open، close، read و write را روی چه دیوایسی انجام می دهیم. البته این فایل برای پروژهٔ ما باید توسط خودمان ساخته شود. ساخت این فایل به کمک دستور mknod انجام می‌شود. می توانید به کمک man page مرتبط به این دستور اطلاعات خوبی در مورد نحوهٔ کار با آن به دست آورید. در عین حال روش هایی جهت ساخت این فایل با استفاده از کدنویسی هم وجود دارد.

تفاوت‌های LKMها و برنامه‌های عادی

ماژول‌ها در فضای کرنل اجرا میشوند نه فضای کاربر. به همین علت نمی توان از برخی توابع معروف برای نوشتن کد ماژول‌ها استفاده کرد. مثلا به جای printf از printk استفاده می شود که در لاگ کرنل رشتهٔ مورد نظر را چاپ می کند (به جای خروجی استاندارد). در کد نمونه، با این تابع آشنا خواهید شد.

برنامه های معمولی از بالا (از اولین خط) شروع به اجرا شده و در پایین (در آخرین خط) خاتمه می یابند. ماژول‌ها چنین روش اجرایی ندارند. مثلاً درایورها به رویدادهایی مرتبط با دیوایس متناظرشان پاسخگو هستند (رویدادمحور هستند). اغلب رویدادهایی که در این آزمایشگاه مورد استفاده قرار میگیرند رویدادهای مربوط به load و unload ماژول، باز و بسته کردن فایل ماژول، خواندن از و نوشتن به فایل ماژول است.

یکی از جنبه‌های فنّی ماژولهای هستهای این است که میتوانند توسط چندین برنامه/فرآیند مختلف به طور همزمان استفاده شوند. باید با دقت برنامه‌نویسی ماژول‌ها را انجام دهیم تا هنگام وقوع وقفه، رفتاری پایدار و معتبر داشته باشند.

ماژول‌ها در سطح بالاتری از دسترسی اجرا میشوند. به طور معمول، چرخه‌های پردازنده بیشتری نسبت به برنامه‌های فضای کاربر به ماژول‌های هسته اختصاص می‌یابد. این یک مزیت است اما باید مراقب بود تا ماژول نوشته شده بر عملکرد کلی سیستم تاثیر منفی نداشته باشد.

هدرهای لینوکس

برای ساختن ماژول به هدرفایل‌های مخصوصی نیاز داریم. نام این هدرفایل‌ها Linux Headers است. این هدرفایل ها در فرایند کامپایل استفاده میشوند. به کمک این هدرها بررسی میشود که آیا توابع مرتبط با کرنل (مثلاً هنگام نوشتن کد ماژول) به

درستی استفاده شده‌اند یا خیر. همین هدرفایل ها هستند که ارتباط بین اجزای کرنل را میسر می سازند و همچنین این هدرفایل‌ها بین فضای کاربر و فضای کرنل به عنوان یک اینترفیس رفتار می کنند. شما باید برای نسخهٔ لینوکس خودتان این هدرفایل ها را دریافت و نصب کنید. مثلا برای اوبونتو می توان از دستور زیر استفاده کرد.

```
sudo apt install linux-headers-$(uname -r)
```

کامپایل کردن ماژول

کامپایل کردن ماژول‌ها با برنامه‌های عادی کمی متفاوت است و باید از هدرهایی که در مرحله قبل نصب کردیم استفاده کنیم. بدین منظور از Makefile زیر استفاده میکنیم.

```
obj-m+=myapp.o
all:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
```

نام ماژول در خط اول تعیین میشود (myapp.o). سپس ابزار make، از Makefile سطح بالای کرنل جهت تولید ماژول استفاده میکند. این کار از طریق تغییر مسیر اصلی ابزار به مسیر build سیستم‌عامل میسر میشود (آپشن C-). سپس مسیر کنونی به عنوان مسیر خروجی ابزار تعیین میشود (عبارت M). دقت کنید که modules و clean هردو تارگت‌هایی در Makefile سطح بالای کرنل هستند و از قبل در آنجا تعریف شده اند (فقط از آنها استفاده میکنیم).

پس از کامپایل کردن ماژول، یک فایل با پسوند ko ساخته می شود (ko = Kernel Object). این همان فایلی است که به کرنل متصل خواهد شد و قابلیت هایی که نیاز داریم (و کد آنها را نوشته ایم) را به کرنل اضافه خواهد کرد. برای load کردن ماژول (اتصال به کرنل) از دستور insmod استفاده می شود. اگر این دستور بدون خطا اجرا شود می توان ماژول اضافه شده را به کمک دستور lsmod مشاهده کرد. برای حذف ماژول از لیست ماژول های فعال (unload کردن) از دستور rmmod استفاده می شود.

File Operations، Load و Unload

همانطور که گفته شد ماژول‌ها ذات ری‌اکتیو دارند و به رخدادها پاسخ می دهند. توابعی که برای پاسخ به رویدادها به کرنل معرفی می کنیم قالب مشخصی دارند. جهت معرفی توابع مرتبط با load و unload ماژول از ماکروهای module_init و module_exit استفاده می کنیم و جهت معرفی توابع باز و بسته کردن دستگاه (مثلاً open کردن فایل خاصی که می‌سازیم) و خواندن از و نوشتن به دستگاه از ساختمان دادهٔ file_operations استفاده می کنیم.

کد نمونه:

```c
#include <linux/init.h> // For module init and exit
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h> // For fops
#include <linux/uaccess.h>
//#include <string.h> // Can't use it!

#define DEVICE_NAME "iut_device"
MODULE_LICENSE("GPL");

// FILE OPERATIONS
static int iut_open(struct inode*, struct file*);
static int iut_release(struct inode*, struct file*);
static ssize_t iut_read(struct file*, char*, size_t, loff_t*);

static struct file_operations fops = {
   .open = iut_open,
   .read = iut_read,
   .release = iut_release,
};

// Why "static"? --> To bound it to the current file.
static int major; // Device major number. Driver reacts to this major number.

// Event --> LOAD
static int __init iut_init(void) {
    major = register_chrdev(0, DEVICE_NAME, &fops); // 0: dynamically assign a major number ||| name is
displayed in /proc/devices ||| fops.
    if (major < 0) {
        printk(KERN_ALERT "iut_device load failed.\n");
        return major;
    }
    printk(KERN_INFO "iut_device module loaded: %d\n", major);
    return 0;
}

// Event --> UNLOAD
static void __exit iut_exit(void) {
    unregister_chrdev(major, DEVICE_NAME);
    printk(KERN_INFO "iut_device module unloaded.\n");
}

// Event --> OPEN
static int iut_open(struct inode *inodep, struct file *filep) {
   printk(KERN_INFO "iut_device opened.\n");
   return 0;
}

// Event --> CLOSE
static int iut_release(struct inode *inodep, struct file *filep) {
   printk(KERN_INFO "iut_device closed.\n");
   return 0;
}

// Event --> READ
static ssize_t iut_read(struct file *filep, char *buffer, size_t len, loff_t *offset) {
    char *message = "IUT OS         2!";
    int errors = 0;
    errors = copy_to_user(buffer, message, strlen(message));
    return errors == 0 ? strlen(message) : -EFAULT;
}

// Registering load and unload functions.
module_init(iut_init);
module_exit(iut_exit);
```

بررسی کد نمونه:

در کد یک ماژول کاراکتری، توابع پیش فرضی وجود دارند که شما باید به صورت اختصاصی با توجه به هدف ماژول آن ها را پیاده سازی کنید. تابع init هنگام load یک ماژول در سیستم فراخوانی می شود لذا در این تابع، عملیات مربوط به رجیستر کردن ماژول در سیستم انجام می شود. در مقابل وقتی ماژولی unload می‌شود، تابع exit فراخوانی می شود؛ پس در این تابع، منطقی است که ماژول را unregister کرده و عدد ماژور آن را آزاد کنیم. توجه کنید که دو تابع نامبرده به کمک ماکروهای module_init و module_exit در انتهای کد، به کرنل معرفی شده اند.

یک ساختارداده بسیار مهم از نوع file_operations در هر ماژول وجود دارد. در این ساختارداده، توابعی که برای ماژول موردنظر در سطح کاربر قابل استفاده است معرفی می‌شود. در واقع API همه ماژول های کاراکتری ثابت است اما پیاده سازی این API در دست برنامه نویس ماژول است. همان طور که بیان شد عملیات روی ماژول کاراکتری کاملا شبیه عملیات روی فایل است که این توابع شامل open، close، read و write است. از طریق ساختمان‌داده file_operations توابع پیاده سازی شده توسط برنامه نویس ماژول را برای هرکدام از توابع نامبرده معرفی می‌کنیم. مثلا در کد نمونه می بینید که توابع open، read و release پیاده‌سازی و نام آن ها در file_operations مشخص شده است (release هنگام بستن فایل یا همان close کردن آن اجرا میشود). دقت کنید که متناسب با انتظاری که از ماژول داریم توابع را پیاده سازی می کنیم.

شما یک بار ماژول را در سیستم load می‌کنید و اپلیکیشن های مختلف چندین بار (حتی به صورت همزمان) از ماژول load شده استفاده می‌کنند یعنی توابع file_operations را برای آن فراخوانی می‌کنند. پس به ازای هربار open کردن ماژول در یک اپلیکیشن یک File Descriptor برای استفاده از آن ساخته می شود و اپلیکیشن پس از آن با استفاده از آن File Descriptor می‌تواند از ماژول بخواند یا در آن بنویسد. مثلا تصور کنید قراراست بافر یک دیوایس سخت افزاری، از اطلاعاتی که یک اپلیکیشن برای آن ارسال می کند پر شود(write در ماژول) یا اپلیکیشن اطلاعاتی را از بافر سخت افزار بخواند (read از ماژول).

چون اطلاعات بین فضای کاربر و کرنل جا به جا می شود باید از توابع مخصوص مثل copy_to_user و copy_from_user استفاده شود. همچنین نحوه مدیریت داده در ماژول به عهده برنامه نویس ماژول است.

درمورد توابع مختلفی که در کد می بینید از طریق اینترنت و manual لینوکس می توانید اطلاعات خوبی کسب کنید.

استفاده از ماژول ساخته شده

باید برای استفاده از ماژول، فایل دیوایس آن را در /dev ایجاد شود. این کار از طریق mknod قابل انجام است (عدد ماژور ماژول load شده را می توان از طریق جستجوی نام ماژول در فایل /proc/devices به دست آورد). با اینکه فایل ساخته شده به کمک mknod یک فایل خاص است اما ارتباط با آن کار سختی نیست. به کمک هر زبانی می توانیم ماژول نوشته

شده را تست کنیم. البته فراموش نکنید که هنگام اجرای برنامهٔ تست باید از sudo استفاده کنیم تا برنامه بتواند فایل ماژول را باز کند.

دقت کنید خروجی توابع printk در کرنل لاگ قرار می گیرد که از طریق فایل‌های /var/log یا با استفاده از دستور dmesg قابل مشاهده هستند.

نمونه کد اپلیکیشن سطح کاربر:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
int main()
{
      char readBuffer[128]={0};
      int fd=open("/dev/iut_device", O_RDONLY);
      read(fd,readBuffer, 128);
      fprintf(stdout,"%s\n",readBuffer);
      close(fd);
      return 0;
}
```

IOCTL

در کرنل لینوکس یک System Call دیگر به نام ioctl وجود دارد که برای ارتباط با دستگاه کاراکتری کاربرد دارد. این فراخوانی سیستمی در صورتی که قصد ارسالی دستوری متفرقه برای دستگاه را داشته باشیم به کار می‌آید. تابع متناظر این system call در یک درایور به شکل زیر در file_operations تعریف می‌شود.

```
static struct file_operations fops = {
    .open = iut_open,
    .read = iut_read,
    .write = iut_write,
    .release = iut_release,
    .unlocked_ioctl = iut_ioctl,
};
```

با استفاده از ioctl می‌توان دستورات مختلفی تعریف کرد و متناظر با آن دستورات در درایور فعالیت‌هایی را تعریف کرد.

برای تعریف تابع ioctl باید تابعی با ساختار زیر تعریف کرد.

```
static long iut_ioctl(struct file *file, unsigned int req, unsigned long pointer)
```

- پارامتر اول اطلاعاتی درباره فایلی که روی آن کار میکنیم، در اختیار قرار می‌دهد.
- پارمتر دوم دستور را مشخص می‌کند. این پارامتر از نوع unsigned int است ولی هر عددی را نمیتوان برای این پارامتر استفاده کرد. ماکرو _IO(chr, number) یک عدد مناسب برای این پارامتر تولید می‌کند.
- پارامتر سوم اشاره گری به فضای حافظه کاربر است. این پارامتر برای ارسال داده از فضای کاربر به فضای کرنل استفاده می‌شود. برای استفاده از این پارامتر می‌توان یک نوع داده تعریف کرد و اشاره گر به آن داده را در دستور IOCTL ارسال کرد.

یک IOCTL ساده

به مثال زیر توجه کنید. در این مثال سه فایل داریم. فایل اول util.h است که در آن دستورات و یک ساختمان داده تعریف شده است. در فایل driver.c کد درایور تعریف شده است. به تابع iut_init و iut_ioctl در این فایل دقت کنید.

فایل utils.h:

```
enum {
    Command1 = _IO('A', 1),
};
struct iut_data {
    int arg1;
}
```

فایل driver.c:

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/miscdevice.h>
#include <linux/module.h>
#include <linux/netdevice.h>
#include <linux/list.h>
#include <linux/version.h>
#include <linux/wait.h>
#include <asm/uaccess.h>
#include "utils.h"
#define DEVICE_NAME "iut_device"
static int major;
static long iut_ioctl(struct file *file, unsigned int req, unsigned long pointer) {
  struct iut_data *data;
  data = kzalloc(sizeof(struct iut_data), GFP_KERNEL);
  if (!data)
    return -ENOMEM;
  if (copy_from_user(data, ( void * ) pointer, sizeof(struct iut_data)))
    return -EFAULT;
  switch (req) {
    case Command1:
      printk(KERN_INFO "Command1 %d\n", data->arg1);
      return 1;
  }
  return -EINVAL;
}
static int iut_open(struct inode *inode, struct file *file) {
  printk("device opened\n");
  return 0;
}
static int iut_release(struct inode *inode, struct file *file) {
  printk("device closed\n");
  return 0;
}
static const struct file_operations fops = {
  .open = iut_open,
  .release = iut_release,
  .unlocked_ioctl = iut_ioctl,
};
static int __init iut_init(void) {
  major = register_chrdev(0, DEVICE_NAME, &fops);
  if (major < 0){
    printk(KERN_ALERT "Device001 load failed!\n");
    return major;
  }
  printk(KERN_INFO "iut device module has been loaded: %d\n", major);
  return 0;
}
static void __exit iut_exit(void) {
  unregister_chrdev(major, DEVICE_NAME);
  printk(KERN_INFO "iut device module has been unloaded.\n");
}
module_init(iut_init);
module_exit(iut_exit);
MODULE_LICENSE("GPL");
```

فایل user.c:

```c
#include <sys/ioctl.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "utils.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int fd, ret;
  struct iut_data *iut_data1 = malloc(sizeof(struct iut_data));
  memset(iut_data1, 0, sizeof(struct iut_data));
  iut_data1->arg1 = 100;
  fd = open("/dev/iut_device", O_RDWR);
  if (fd < 0) {
    perror("open");
    exit(0);
  }
  ret = ioctl(fd, Command1 , iut_data1);
  if (ret < 0) {
    perror("ioctl");
    exit(0);
  }
  close(fd);
  return 0;
}
```