دانشگاه صنعتی اصفهان

دانشکده برق و کامپیوتر

آزمایشگاه سیستم عامل

دستور کار جلسه سوم

مهر ۱۴۰۱

فهرست مطالب

LKM چیست؟

یکي از قابلیت‌های لینوکس امکان توسعه کرنل هنگام بالا بودن سیستم عامل است. یعنی می‌توان حین اجرای سیستم عامل قابلیت‌هایی را به آن اضافه یا از آن کم کرد. به قطعه کدهایی که حین اجرا به کرنل افزوده می‌شوند ماژول گفته می‌شود. کرنل لینوکس از انواع مختلف ماژول (مثلا درایورها) پشتیبانی می‌کند. هر ماژولی از Object Code تشکیل شده‌است که می‌تواند به صورت پویا به کرنل لینک شود، بدون نیاز به کامپایل کردن دوباره کل کرنل؛ پس از اضافه شدن یک ماژول به کرنل، اپلیکیشن‌های فضای کاربر می‌توانند از آن ماژول استفاده کنند.

دیوایس‌ها و درایورها

تقریباً هر عملیات سیستمی نهایتاً با یك دستگاه فیزیکی کار خواهد داشت. تمامی عملیات‌های کنترل دستگاه (به جز دستگاه‌هایی مثل پردازنده و حافظه اصلی) توسط قطعه کدهایی انجام می‌شود که مخصوص به دستگاه هدف (دستگاهي که عملیات رو آن انجام می‌شود) است. به این قطعه کدها درایور گفته می‌شود. کرنل باید براي تمامي دستگاه‌هاي متصل به سیستم درایور مخصوص به خودشان را داشته باشد (مثلا براي ماوس و کیبورد باید درایور داشته باشد).

انواع دیوایس

در لینوکس به طور کلی سه مدل دستگاه تعریف می‌شود. هر ماژول هم معمولاً تحت یکی از این سه مدل توسعه می‌یابد که نتیجه آن سه دسته ماژول کاراکتری (Char module)، بلوکی (Block module) و شبکه‌ای(Network module) است. البته می‌توانیم این دسته‌بندی را رعایت نکنیم و ماژولی بنویسیم که بتواند قابلیت‌هایی از هر سه دسته را داشته باشد اما این ماژول مقیاس پذیر و توسعه پذیر نخواهد بود.

آماده سازی سیستم عامل برای کامپایل کرنل ماژول‌ها

سیستم باید برای ساخت کد هسته آماده باشد و برای این کار باید هدرهای Linux را در دستگاه خود نصب کنید. به عنوان مثال، برای سیستم عامل Debian 64 bit می‌توانید از موارد زیر استفاده کنید:

```
sudo apt-get update
sudo apt-get install linux-headers-4.19.0-4-amd64
```

در حقیقت نیاز دارید تا هدرهای مربوط به ورژن کرنل خود را نصب کنید. برای یافتن ورژن کرنل می‌توانید از دستور زیر استفاده کنید:

```
uname -r
```

یک ماژول ساده

چرخه اجرای یک برنامه رایانه‌ای معمولی بسیار منطقی است. یک لودر حافظه را برای برنامه اختصاص می‌دهد. سپس برنامه و تمام کتابخانه‌های متشرک مورد نیاز را بارگیری می‌کند. اجرای دستورالعمل در برخی از قسمت‌های ورودی(به طور معمول main در برنامه‌های C/C++) شروع می‌شود، دستورات اجرا می‌شوند، استثنائات پرتاب می‌شوند، حافظه پویا اختصاص داده می‌شود و از هم جدا می‌شود و در نهایت برنامه به اتمام می‌رسد. در هنگام خروج از برنامه، سیستم عامل هرگونه نشت حافظه را مشخص می‌کند و حافظه از دست رفته را آزاد می‌کند. اما ماژول هسته یک برنامه کاربردی نیست. برای شروع هیچ ()main وجود ندارد. برخی از تفاوت‌های اساسی در ماژول‌های هسته است:

- به صورت خط به خط و پشت سرهم اجرا نمی‌شود: ماژول هسته خود را ثبت می‌کند تا با استفاده از توابع خود، به درخواست‌ها رسیدگی کند. نوع درخواست‌هایی که می‌تواند انجام دهد در کد ماژول تعریف شده است. این کاملا شبیه به مدل برنامه نویسی رویداد محور است که معمولا در برنامه‌های گرافیکی(GUI) استفاده می‌شود.

- پاکسازی خودکار ندارید: هر منبعی که به ماژول اختصاص داده شود باید هنگام خروج ماژول از کرنل به صورت دستی آزاد شود در غیر این صورت ممکن است تا راه اندازی مجدد سیستم در دسترس نباشد.
- تابع printf را ندارید: کد هسته نمی‌تواند به کتابخانه‌های کد نوشته شده برای فضای کاربر لینوکس دسترسی داشته باشد. ماژول هسته در فضای هسته زندگی می‌کند و اجرا می‌شود که فضای آدرس حافظه خاص خود را دارد. رابط بین فضای هسته و فضای کاربر به وضوح تعریف و کنترل می‌شود. با این حال ما یک تابع printk داریم که می‌تواند اطلاعات را چاپ کند و از درون فضای کاربر این اطلاعات مشاهده شود.
- می‌تواند قطع شود: یک از جنبه‌های مشکل ماژول‌های هسته‌ای این است که می‌توانند توسط چندین برنامه/فرآیند مختلف به طور همزمان استفاده شوند. ما باید با دقت ماژول‌های خود را بسازیم تا هنگام وقوع وقفه، آن‌ها رفتاری پایدار و معتبر داشته باشند.
- در سطح بالاتری از دسترسی اجرا می‌شوند: به طور معمول، چرخه‌های CPU بیشتری نسبت به برنامه‌های فضای کاربر به ماژول‌های هسته اختصاص می‌یابد. به نظر می‌رسد این یک مزیت است اما باید بسیار مراقب باشید که ماژول شما بر عملکرد کلی سیستم شما تاثیر منفی نداشته باشد.

مثال: کد زیر یک ماژول ساده است که در ادامه به معرفی اجزای آن می‌پردازیم.

یک فایل به نام hello.c بسازید و کد زیر را در آن بنویسید.

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE(“GPL”);
MODULE_AUTHOR(“OSLAB”);
static int hello_init(void){
        printk(KERN_ALERT “Hello world!\n”);
        return 0;
}
static void hello_exit(void){
        printk(KERN_ALERT “Goodbyte cruel world\n”);
}
module_init(hello_init);
```

```
module_exit(hello_exit);
```

دو خط اول مربوط به کتابخانه‌های کرنل هستند. همانگونه که بیان شد در فضای کرنل به کتابخانه‌های فضای کاربری دسترسی نداریم و باید از کتابخانه‌های موجود در فضای کرنل استفاده کنیم. دو خط بعد معرفی نویسنده و لایسنس ماژول است. همچنین با استفاده از پارامتر MODULE_DESCRIPTION می‌توان توضیحات اضافی به ماژول اضافه کرد. دو تابع hello_init و hello_exit برای نقطه شروع و پایان ماژول هستند که حتما باید static باشند تا محدودیت دسترسی به این دو تابع تنها در همین فایل باشد. hello_init یک بار در هنگام بارگذاری ماژول اجرا می‌شود و با استفاده از تابع module_init به سیستم عامل معرفی می‌شود. hello_exit هنگام خارج شدن ماژول از کرنل یک بار اجرا می‌شود و با استفاده از module_exit به سیستم عامل معرفی می‌گردد.

کامپایل کردن ماژول

کامپایل کردن این ماژولها با برنامه‌های عادی کمی متفاوت است و باید از هدرهایی که در مرحله قبل نصب کردیم استفاده کنیم. به این منظور از Makefile استفاده خواهیم کرد. در دایرکتوری پروژه یک فایل به نام Makefile بسازید و خطوط زیر را در آن بنویسید.

```
Obj-m+=hello.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
```

خط اول این Makefile به عنوان تعریف هدف است و ماژول ساخته شده را تعریف می‌کند(hello.o) و obj-m یک هدف lkm را تعریف می‌کند. بقیه Makefile شبیه به یک Makefile معمولی است. $(shell uname -r) دستوری مفید برای مشخص کردن نسخه فعلی کرنل است. گزینه C- مسیر جاری را به مسیر مشخص شده منتقل می‌کند. تخصیص متغیر M=$(PWD) به دستور make که در آن فایل‌های پروژه واقعی وجود دارد، می‌گوید هدف ماژول‌ها، هدف پیش فرض ماژول‌های هسته خارجی است. پس از ذخیره سازی فایل Makefile می توانید با دستور زیر، ماژول را کامپایل کنید.

```
make
```

اگر فرایند make بدون مشکل انجام شود باید یک فایل به نام hello.ko به دایرکتوری اضافه شود. برای اضافه کردن این ماژول به کرنل از دستور زیر استفاده کنید:

```
sudo insmod hello.ko
```

و برای خارج کردن ماژول از کرنل از دستور زیر استفاده کنید:

```
sudo rmmod hello.ko
```

همچنین با دستور lsmod می‌توانید لیست ماژول‌هایی که در حال حاضر در کرنل موجود هستند را ببینید. برای دیدن متن‌های چاپ شده توسط ماژول می‌تونید از دستور journalctl استفاده کنید. با استفاده از پارامتر f- می‌توان لاگ‌های خروجی را به صورت زنده مشاهده کرد.

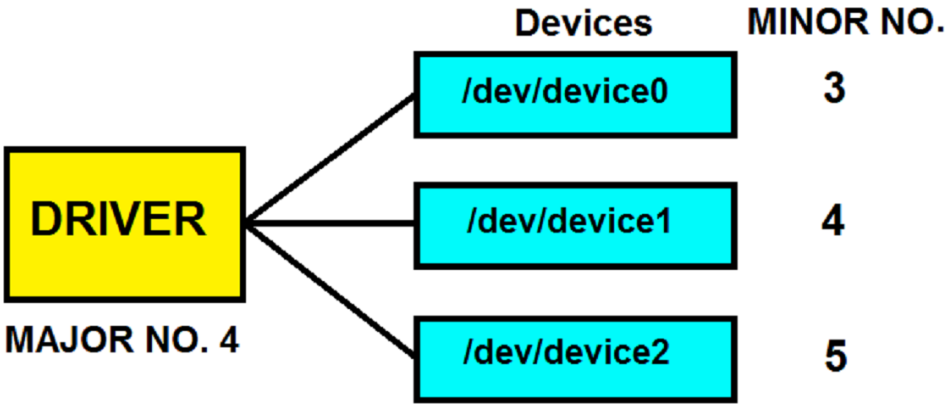
کاراکتر دیوایس

دستگاه کاراکتری، دستگاهی‌ست که می‌توان با آن مانند یک فایل رفتار کرد؛ یعنی مانند جریانی از بایت‌ها(Stream of Bytes). یک درایور کاراکتری چنین رفتار فایل مانندی را براي این دستگاه کنترل و پیاده‌سازی می‌کند. این درایورهای کاراکتری معمولاً فراخوانی‌های سیستمی بازکردن(open)، بستن(close)، خواندن(read) و نوشتن(write) را پیاده‌سازی می‌کنند. به عنوان مثال کنسول متنی (/dev/console) و پورت‌های سریال (/dev/ttyS0 و مشابه‌هاي آن) دستگاه‌های کاراکتری هستند. دسترسی به دستگاه‌های کاراکتری به کمک گره‌های فایل‌سیستم(Filesystem Nodes) انجام می‌شود. تنها تفاوت قابل توجه بین دستگاه‌های کاراکتری و فایل‌های معمولی این است که در فایل‌های معمولی می‌توان به عقب و جلو حرکت کرد اما معمولًا دستگاه‌های کاراکتری کانال‌های داده‌ای هستند که فقط به صورت سری(Sequentially) قابل دسترسی هستند. البته دستگاه‌های 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 يك عدد یکتا تخصیص داده می‌شود که به آن Major Number مي‌گویند. بدین ترتیب موقع load هر درایور در سیستم باید يك عدد ماژور آزاد به آن تخصیص داده شود. توجه داشته باشید که دیوایس‌های سخت‌افزاری مختلفی می‌توانند از طریق يك 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 می‌کنند اما سیستم باید راهی برای تفکیک این دو دیوایس داشته باشد. به همین دلیل عدد دیگري با نام Minor Number برای دیوایس‌های مختلفی که از يك درايور واحد استفاده می‌کنند در نظر گرفته می‌شود. کرنل از عدد ماژور استفاده میکند تا درایور مرتبط را پیدا کند و درایور از عدد مینور جهت کار با دستگاه مشخصی استفاده می‌کند.



دستور mknod

يكي از مفاهيم مهم در سیستم‌عامل‌های مبتنی بر یونیکس مفهوم فایل بودن تقریباً همه چیز است. یعنی منابع ورودی/خروجی مختلفی(مانند اسناد، دایرکتوری‌ها، درایوها، مودم، کیبورد، پرینتر و حتی برخی IPCها و ارتباطات شبکه‌ای) وجود دارد که همگی جریان‌های بایتی ساده‌ای هستند. مزیت چنین رویکردی این است که ابزارها و API های یکسانی را می‌توان براي دسترسی و ارتباط با چندین منبع مختلف استفاده کرد. البته می‌توان این مفهوم را دقیق‌تر هم بیان کرد و گفت هر چيزي يک File Descriptor است. چرا که هنگام باز کردن يك فايل معمولی یا ایجاد پایپ‌های ناشناس یا ساخت سوکت شبکه، File Descriptorهایی ساخته می‌شود که راه ارتباطی و رابط بین کد با آن منبع خواهد بود. دستگاه‌های کاراکتری از طریق اسم‌شان در فایل‌سیستم قابل دسترسی هستند و می‌توان با آنها مانند یک فایل رفتار کرد. این اسامي را "فايل‌هاي خاص"، "فايل‌هاي دستگاهي" یا حتی "گره‌هایی در درخت فایل‌سیستم" گوییم. اما

این فایل خاص کجاست؟ معمولًا فایل درایور مرتبط با هر دیوایس در دایرکتوري "/dev" قرار دارد. `ls /dev -l` را اجرا کنید تا فایل‌هاي مربوط به ماژول‌های کنونی سیستم‌تان را مشاهده کنید. همانطور که می‌بینید اولین کاراکتر از رشته permission هر فایل، مشخص‌کننده نوع دیوایس یا فایل ماژول است(c به معنی دیوایس کاراکتر و b به معنی دیوایس بلوکی است). 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 دیوایس‌های مختلفی ممکن است از یك ماژول استفاده کنند که بدین ترتیب همه دارای یك عدد ماژور ولی عددهای متفاوت مینور هستند. هرگاه دیوایسی به سیستم اضافه می‌شود باید حتما فایل درایور متناظرش در شاخه dev/ قرارگیرد و در واقع از طریق نام همین فایل است که در کد اپلیکیشن می‌توانیم مشخص کنیم با کدام دیوایس کار داریم و عملیات open، close، read و write را روي چه دیوایسی انجام می‌دهیم. ساخت این فایل به کمك دستور "mknod" انجام مي‌شود. می‌توانید به کمك man page مرتبط به این دستور اطلاعات خوبی در مورد نحوه کار با آن به دست آورید. در عین حال روش‌هایی جهت ساخت این فایل با استفاده از کدنویسی هم وجود دارد.

نوشتن یك درایور کاراکتری

در ادامه یك دیوایس ساده که یك پیغام را به کاربر نشان می‌دهد، می‌نویسیم. مراحل اولیه ساخت درایور کاراکتری همانند یک کرنل ماژول است. فایلی خالی با نام char_device.c ایجاد کنید. همچنین یک فایل Makefile همانند Makefileای که برای کرنل ماژول ایجاد کردیم، ایجاد کنید.

Unload و Load ، File Operations

همانطور که گفته شد کرنل ماژول و درایور ذاتا واکنشی هستند و به رخدادها پاسخ می‌دهند. توابعی که برای پاسخ به رویدادها به کرنل معرفی می‌کنیم قالب مشخصی دارند. جهت معرفی توابع مرتبط با load و unload ماژول از ماکروهای module_init و module_exit استفاده می‌کنیم و برای معرفی توابع باز و بسته کردن دستگاه (مثلا open کردن فایل خاصي که مي‌سازیم) و خواندن/نوشتن از/به دستگاه از ساختمان داده file_operations استفاده می‌کنیم.

کد نمونه دستگاه ساده:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

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

static int major; // device major number. driver reacts to this major number.

static int __init iut_init(void) {
   major = register_chrdev(0, DEVICE_NAME, &fops); // 0: auto major ||| name is displayed in /proc/devices ||| 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");
}

static int iut_open(struct inode *inodep, struct file *filep) {
  printk(KERN_INFO " iut  device opened.\n");
  return 0;
}

static int dev_release(struct inode *inodep, struct file *filep) {
```

```
  printk(KERN_INFO "iut  device closed.\n");
  return 0;
}

static ssize_t dev_read(struct file *filep, char *buffer, size_t len, loff_t *offset) {
        int errors = 0;
        char *message = “IUT device example”;
        errors = copy_to_user(buffer, message, strlen(message));
        return errors == 0 ? strlen(message) : -EFAULT;
}

module_init(iut_init);
module_exit(iut_exit);
```

در کد یک ماژول کاراکتری، توابع پیش‌فرضی وجود دارند که باید به صورت اختصاصی با توجه به هدف ماژول آن‌ها را پیاده‌سازی کرد. تابع init هنگام load یك ماژول در سیستم فراخوانی می‌شود لذا در این تابع، عملیات مربوط به رجیستر کردن ماژول در سیستم انجام می‌شود. در مقابل وقتی ماژولی unload می‌شود، تابع exit فراخوانی می‌شود؛ پس در این تابع، مناسب 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 فایل به کرنل معرفی شده‌اند. 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 ماژول‌های کاراکتری ثابت است اما پیاده‌سازی این API در دست برنامه‌نویس ماژول است. همان‌طور 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 متغیر file_operations توابع پیاده‌سازی شده توسط برنامه‌نویس ماژول را برای هرکدام از توابع نامبرده معرفی می‌کنیم. مثلا در کد نمونه می‌بینید که تابع open، read و write پیاده‌سازی و نام آن‌ها در file_operations مشخص شده‌است. دقت کنید که متناسب با انتظاری که از ماژول داریم توابع open، read و write و... را پیاده‌سازی می‌کنیم. شما یك بار ماژول را در سیستم load می‌کنید و اپلیکیشن‌های مختلف و متعدد چندین‌بار(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 می‌شود و اپلیکیشن پس از آن با استفاده از آن File Descriptor مي‌تواند از ماژول بخواند یا به آن بنویسد.

تصور کنید قراراست بافر يك ديوايس سخت‌افزاري از اطلاعاتی که یک اپلیکیشن برای آن ارسال می‌کند پر شود(write در ماژول) یا اپلیکیشن اطلاعاتی را از بافر سخت‌افزار بخواند(read از ماژول) در اینجا چون اطلاعات (داده) بين فضاي كاربر و كرنل جابه‌جا مي‌شود باید از توابع مخصوص مثل copy_to_user و copy_from_user استفاده شود. همچنین نحوه مدیریت داده در ماژول به عهده برنامه‌نویس ماژول است. در کد نمونه، داده پس از دریافت شدن از اپلیکیشن توسط ماژول، به سخت‌افزار ارسال نشده چون این کد مربوط به ماژولی است که به منظور ارتباط با سخت‌افزار نوشته نشده‌است. درمورد توابع مختلفی که در کد می‌بینید از طریق اینترنت و manual لینوکس می‌توانید اطلاعات خوبی کسب کنید.

استفاده از ماژول ساخته شده

پس از کامپایل کردن ماژول، یک فایل با پسوند "ko" ساخته می‌شود که می‌توانید آن را با استفاده از دستور insmod در کرنل بارگذاری کنید. جهت استفاده از ماژول، فایل دیوایس آن را در dev/ ایجاد کنید(عدد ماژور ماژول load شده را مي‌توان از طريق جستجوي نام ماژول در فایل proc/devices/ به دست آورید). با اینکه فایل ساخته شده به کمك mknod يك فايل خاص است اما ارتباط با آن کار سختی نیست. به کمک هر زبانی می‌توانیم ماژول نوشته شده را تست کنیم. البته فراموش نکنید که هنگام اجرای برنامه تست باید از sudo استفاده کنیم تا برنامه بتواند فایل درایور را باز کند.

```
Import os
path = "/dev/iut_device"
fd = os.open(path, os.O_RDONLY)
data = os.read(fd, 128)
print(f'Number of bytes red: {len(data)}')
print(data.decode())
os.close(fd)
```

نکته: خروجی توابع printk در کرنل لاگ قرار می‌گیرد که از طریق مشاهده فایل‌هاي var/log/ یا با استفاده از دستور dmesg قابل مشاهد هستند.