آزمایشگاه سیستم عامل

دانشکده برق و کامپیوتر -
دانشگاه صنعتی اصفهان

پاییز 1398

موضوع: کرنل ماژول

کرنل ماژول قابل بارگذاری..... 2

آماده سازی سیستم عامل برای کامپایل کرنل ماژول ها..... 3

یک ماژول ساده..... 3

کامپایل کردن ماژول..... 6

ماژول کاراکتری..... 7

عدد های Major و Minor در ماژول..... 8

ساختمان داده File Operations در ماژول..... 8

کد درایور کاراکتری..... 8

ارتباط کاربر و ماژول..... 12

کرنل ماژول قابل بارگذاری

LKM مکانیزمی برای اضافه کردن کد یا حذف کد از هسته لینوکس در زمان اجرا است. آنها برای درایورها ایده آل هستند و هسته را قادر می سازد بدون نیاز به دانستن نحوه کار سخت افزار ، با سخت افزار ارتباط برقرار کند.راه حل جایگزین LKM، اضافه کردن کد برای هر درایور به کرنل Linux است.

بدون داشتن این قابلیت ماژول کرنل لینوکس بسیار بزرگ خواهد بود، زیرا باید از هر درایوری که در سیستم ها موجود است پشتیبانی کند. همچنین لازم است هربار که سخت افزار جدیدی به سیستم اضافه می‌شود یا درایور یک دستگاه بروز رسانی می‌شود، کرنل مجددا کامپایل شود. نکته منفی کرنل ماژول ها این است که فایل های مربوط به هر درایور برای هر دستگاه نگه داری شوند.

LKM ها در زمان اجرا به سیستم اضافه می‌شوند ولی آن ها همچنان قسمتی از کرنل محسوب می‌شوند و ادر user space اجرا نمی‌شوند.

ماژول های هسته در فضای هسته اجرا می شوند و برنامه های کاربردی در فضای کاربر اجرا می شوند. فضای هسته و فضای کاربر هر دو فضای آدرس های حافظه منحصر به فرد خود را دارند و با یکدیگر همپوشانی ندارند. سپس خدمات هسته با استفاده از system call به صورت کنترل شده در اختیار فضای کاربر قرار می گیرند.

آزمایشگاه سیستم عامل

دانشکده برق و کامپیوتر –
دانشگاه صنعتی اصفهان

پاییز 1398

آماده سازی سیستم عامل برای کامپایل کرنل ماژول ها

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 دبیان 64 بیتی می توانید از موارد زیر استفاده کنید:

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

و برای توزیع های آرچ و مانجارو از دستور زیر:

```
Sudo pacman -Sy linux-headers419
```

در حقیقت نیاز دارید که هدر های مربوط به ورژن کرنل خود را نصب کنید که ورژن کرنل در زمان نوشتن این پیش‌گزارش ۴.۱۹.۰ بوده است.

برای یافتن ورژن کرنل می‌توانید از دستور زیر استفاده کنید:

```
Uname -r
```

یک ماژول ساده

چرخه اجرای یک برنامه رایانه ای معمولی بسیار منطقی است. یک لودر حافظه را برای برنامه اختصاص می‌دهد ، سپس برنامه و تمام کتابخانه های مشترک مورد نیاز را بارگیری می کند. اجرای دستورالعمل در برخی از قسمت های ورودی (به طور معمول main() در برنامه های C / C ++) شروع می شود ،

دستورات اجرا می شوند ، استثنائات پرتاب می شوند ، حافظه پویا اختصاص داده می شود و از هم جدا می شود و در نهایت برنامه به اتمام می رسد. در هنگام خروج از برنامه ، سیستم عامل هرگونه نشت حافظه را مشخص می کند و حافظه از دست رفته را آزاد می کند.

یک ماژول هسته یک برنامه کاربردی نیست - برای شروع هیچ ()main ای وجود ندارد! برخی از تفاوتهای اساسی در ماژول های هسته است:

به صورت خط به خط و پشت سر هم اجرا نمی‌شود - ماژول هسته خود را ثبت می کند تا با استفاده از توابع خود ، به درخواست ها رسیدگی می‌کند. نوع درخواست هایی که می تواند انجام دهد در کد ماژول تعریف شده است. این کاملاً شبیه به مدل برنامه نویسی رویداد محور است که معمولاً در برنامه های گرافیکی (GUI) استفاده می شود.

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 نمی تواند به کتابخانه های کد نوشته شده برای فضای کاربر لینوکس دسترسی داشته باشد. ماژول هسته در فضای هسته زندگی می کند و اجرا می شود که فضای آدرس حافظه خاص خود را دارد. رابط بین فضای هسته و فضای کاربر به وضوح تعریف و کنترل می شود. با این حال ما یک تابع printk داریم که می تواند اطلاعات را چاپ کند و از درون فضای کاربر این اطلاعات مشاهده شود.

می تواند قطع شود - یکی از جنبه های مشکل ماژول های هسته ای این است که می توانند توسط چندین برنامه / فرآیند مختلف به طور همزمان استفاده شوند. ما باید با دقت ماژول های خود را بسازیم تا هنگام وقوع وقفه ، آنها رفتاری پایدار و معتبر داشته باشند.

در سطح بالاتری از دسترسی اجرا می‌شوند - به طور معمول ، چرخه های CPU بیشتری نسبت به برنامه های فضای کاربر به ماژول های هسته اختصاص می یابد. به نظر می رسد این یک مزیت است ، اما باید بسیار مراقب باشید که ماژول شما بر عملکرد کلی سیستم شما تاثیر منفی نداشته باشد.

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 "Goodbye cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

دو خط اول مربوط به کتابخانه های کرنل هستند همانگونه که بیان شد در فضای کرنل به کتابخانه های فضای کاربر دسترسی نداریم و باید از کتابخانه های موجود در فضای کرنل استفاده کنیم.

دو خط بعد مربوط به نویسنده این کد و لایسنس این ماژول هستند و در صورت نیاز، می‌توانید یک توضیح نیز با دستور MODULE_DESCRIPTION به این ماژول اضافه کنید.
سپس نیاز داریم دو تابع برای نقطه شروع و پایان این ماژول بنویسیم که نوع این توابع حتما باید static باشد که منجر می شود محدوده دسترسی به این توابع تنها در همین فایل باشد.
تابع init نقطه شروع ماژول است و هنگام لود شدن کرنل ماژول در هسته لینوکس یک بار اجرا می شود. با تابع module_init این تابع را به سیستم‌عامل معرفی می‌کنیم.

تابع exit هنگام خارج شدن این ماژول از کرنل یک بار اجرا می‌شود و با تابع module_exit این تابع را به سیستم‌عامل معرفی خواهیم کرد.

کامپایل کردن ماژول

کامپایل کردن این ماژول ها با برنامه های عادی کمی متفاوت است و باید از هدر هایی 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 گفته می شود و ماژول ساخته شده را تعریف می کند (hello.o) و obj-m یک هدف lkm را تعریف می کند.

بقیه Makefile شبیه به Makefile معمولی است $(shell uname -r) یک دستور مفید برای رفتن به نسخه فعلی هدر های هسته است. گزینه -C قبل از انجام هرگونه کار کامپایل ، دایرکتوری را به دایرکتوری هسته تغییر می دهد.
تخصیص متغیر M=$(PWD) به دستور make که در آن فایلهای پروژه واقعی وجود دارد، می گوید هدف ماژول ها هدف پیش فرض ماژول های هسته خارجی است.
پس از اضافه کردن این فایل به دایرکتوری پروژه با دستور زیر می‌توانید این ماژول را کامپایل کنید.

```
Make
```

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

```
sudo insmod hello.ko -f
```

و برای خارج کردن ماژول از کرنل :

```
sudo rmmod hello.ko
```

همچنین با دستور lsmod می‌توانید لیست ماژول هایی که در حال حاضر در کرنل موجود هستند را ببینید.

در ماژول این بخش برای چاپ کردن hello world در تابع init که هنگام اضافه کردن ماژول به کرنل اجرا می‌شود و تابع exit که نقطه خروج ماژول است از تابع printk استفاده کردیم که در حقیقت در لاگ های کرنل چاپ می‌شوند.
برای دیدن این لاگ ها می‌توانید از دستور journalctl و برای نمایش زنده این لاگ ها از دستور journalctl -f استفاده کنید که پارامتر f- در حقیقت به معنای follow است.

ماژول کاراکتری

یک ماژول کاراکتری به طور معمول داده ها را به یک برنامه کاربر منتقل می کند. این ماژول ها مانند لوله ها یا درگاه های سریال رفتار می کنند و فوراً داده های بایت را در یک جریان کاراکتر به کاراکتر می خوانند یا می نویسند. این چارچوب برای بسیاری از درایورهای معمولی ، مانند مواردی که برای رابط در ارتباطات سریال ، فیلمبرداری و دستگاههای صوتی مورد نیاز هستند ، فراهم می شود. جایگزین اصلی برای یک دستگاه کاراکتری دستگاه بلوک است. دستگاههای بلوک به روشی مشابه فایل های معمولی رفتار می کنند و این امکان را می دهند که مجموعه ای از داده های ذخیره شده، با عملکرد هایی مانند خواندن ، نوشتن و جستجو دستکاری شوند. هر دو نوع دستگاه از طریق فایلهای دستگاهی که به درخت سیستم فایل متصل هستند قابل دسترسی است. به عنوان مثال ، کد برنامه ای که در این دستور کار ارائه شده است برای تبدیل شدن به دستگاه /dev/oslabchar ساخته شده است.
این مقاله یک درایور کاراکتر ساده را توصیف می کند که می تواند برای انتقال اطلاعات بین یک برنامه فضای کاربر لینوکس و یک ماژول هسته قابل بارگذاری (LKM) ، که در فضای هسته لینوکس در حال

اجراست ، استفاده شود. در این مثال ، یک برنامه فضای کاربر C رشته ای را به LKM ارسال می کند. سپس LKM با پیامی که به همراه تعدادی نامه ارسال شده به پیام ارسال شده پاسخ می دهد.

عدد های Major و Minor در ماژول

این عدد ها فایل های مربوط به درایور ها را مشخص می‌کنند اگر لیست محتویات دایرکتوری dev/ را با استفاده از دستور ls -l ببینیم مشاهده می‌کنیم که دستگاه های کاراکتری با یک حرف c مشخص شده اند و دو عدد Major و Minor را خواهیم دید.
major number توسط هسته برای شناسایی درست درایور دستگاه هنگام دستیابی به دستگاه استفاده می شود. نقش "شماره جزئی" وابسته به دستگاه است ، و در داخل درایور کنترل می شود.
و باید توجه کنید که زوج این شماره ها باید غیر تکراری باشد.

ساختمان داده File Operations در ماژول

ساختار داده file_operations که در linux/fs.h/ تعریف شده است نشانگرهایی را برای عملکردها (نشانگرهای عملکرد) در درایور نگه می دارد که به شما امکان می دهد رفتار برخی عملکردهای فایل را تعریف کنید. درایور موجود در این دستور کار عملیاتی را برای بعضی از فراخوانی های سیستم مثل

خواندن ، نوشتن ، باز کردن و آزادسازی فایل، ارائه می دهد. اگر یکی از اعضای این ساختار داده را پیاده سازی نکنید ، به سادگی به NULL اشاره خواهد کرد و آن را غیرقابل دسترسی می کند.

کد درایور کاراکتری

در این قسمت بجز توابع exit و init نیاز داریم که توابع دیگری را نیز برای ارتباط با فضای کاربر اضافه کنیم که در نهایت باید یکی از توابع file_operations به آن ها اشاره کند.
تابع dev_open که هنگامی که یک برنامه از فضای کاربر این ماژول را باز کند اجرا خواهد شد.

تابع dev_write زمانی که داده ای از سمت فضای کاربر به ماژول ارسال می‌شود، اجرا خواهد شد.
تابع dev_read هنگامی که داده ای از ماژول به فضای کاربر ارسال می‌شود، اجرا خواهد شد.
تابع dev_release هنگامی که ماژول از سمت فضای کاربر بسته شود اجرا می‌شود.
نمونه کد یک ماژول کاراکتری:
در این کد نیاز داریم که کلاس و دیوایس بسازیم به همین دلیل توابع init و exit متفاوتی نسبت به ماژول قبل دارد.
در تابع init ابتدا باید از سیستم عامل یک major number درخواست کنیم پس از آن نیز کلاس و دیوایس را در کرنل رجیستر کنیم و در تابع exit نیز باید این منابع را آزاد کنیم.

```
#include <linux/init.h>
      // Macros used to mark up functions e.g. __init __exit

#include <linux/module.h>
      // Core header for loading LKMs into the kernel
```

```
#include <linux/device.h>
      // Header to support the kernel Driver Model

#include <linux/kernel.h>
      // Contains types, macros, functions for the kernel
#include <linux/fs.h>
      // Header for the Linux file system support

#include <linux/uaccess.h>
      // Required for the copy to user function

#define  DEVICE_NAME "oslabchar"
// The device will appear at /dev/oslabchar using this value

#define  CLASS_NAME  "os"
      // The device class -- this is a character device driver


MODULE_LICENSE("GPL");
MODULE_AUTHOR("os lab");
MODULE_DESCRIPTION("A simple Linux char driver for the BBB");

static int    majorNumber;
      ///< Stores the device number -- determined automatically

static char   message[256] = {0};
      ///< Memory for the string that is passed from userspace

static short  size_of_message;
      ///< Used to remember the size of the string stored

static int    numberOpens = 0;
      ///< Counts the number of times the device is opened

static struct class*  labcharClass  = NULL;
      ///< The device-driver class struct pointer

static struct device* labcharDevice = NULL;
      ///< The device-driver device struct pointer
```

```
// The prototype functions for the character driver
      // must come before the struct definition
static int  dev_open(struct inode *, struct file *);
static int  dev_release(struct inode *, struct file *);

static ssize_t dev_read(struct file *, char *, size_t, loff_t *);

static ssize_t dev_write(struct file *, const char *, size_t, loff_t *);

static struct file_operations fops =
{
   .open = dev_open,
   .read = dev_read,
   .write = dev_write,
   .release = dev_release,
};


static int __init labchar_init(void){
   printk(KERN_INFO "LABChar: Initializing the LABChar LKM\n");

   // Try to dynamically allocate a major number for the device
            //more difficult but worth it
   majorNumber = register_chrdev(0, DEVICE_NAME, &fops);
   if (majorNumber<0){
      printk(KERN_ALERT "LABChar failed to register a major number\n");
      return majorNumber;
   }
   printk(KERN_INFO "LABChar: registered correctly with major number %d\n",
majorNumber);

   // Register the device class
   ebbcharClass = class_create(THIS_MODULE, CLASS_NAME);
   if (IS_ERR(labcharClass)){
       // Check for error and clean up if there is
      unregister_chrdev(majorNumber, DEVICE_NAME);
      printk(KERN_ALERT "Failed to register device class\n");
      return PTR_ERR(labcharClass);
      // Correct way to return an error on a pointer
```

```
 }
   printk(KERN_INFO "LABChar: device class registered correctly\n");

   // Register the device driver
   ebbcharDevice = device_create(labcharClass, NULL,
                       MKDEV(majorNumber, 0), NULL, DEVICE_NAME);
   if (IS_ERR(labcharDevice)){
      // Clean up if there is an error
      class_destroy(labcharClass);
      // Repeated code but the alternative is goto statements
      unregister_chrdev(majorNumber, DEVICE_NAME);
      printk(KERN_ALERT "Failed to create the device\n");
      return PTR_ERR(labcharDevice);
   }
   printk(KERN_INFO "LABChar: device class created correctly\n");
      // Made it! device was initialized
   return 0;
}

static void __exit labchar_exit(void){
   device_destroy(labcharClass, MKDEV(majorNumber, 0));
      // remove the device

   class_unregister(labcharClass);
      // unregister the device class

   class_destroy(labcharClass);
      // remove the device class

   unregister_chrdev(majorNumber, DEVICE_NAME);
      // unregister the major number

   printk(KERN_INFO "LABChar: Goodbye from the LKM!\n");
}

static int dev_open(struct inode *inodep, struct file *filep){
   numberOpens++;
   printk(KERN_INFO "EBBChar: Device has been opened %d time(s)\n",
numberOpens);
```

```
   return 0;
}

static ssize_t dev_read(struct file *filep, char *buffer, size_t len, loff_t
*offset){
   int error_count = 0;
   // copy_to_user has the format ( * to, *from, size) and returns 0 on
success
   error_count = copy_to_user(buffer, message, size_of_message);

   if (error_count==0){            // if true then have success
      printk(KERN_INFO "EBBChar: Sent %d characters to the user\n",
size_of_message);
      return (size_of_message=0);
      // clear the position to the start and return 0
   }
   else {
      printk(KERN_INFO "EBBChar: Failed to send %d characters to the user\n",
error_count);
      return -EFAULT;
      // Failed -- return a bad address message (i.e. -14)
   }
}

static ssize_t dev_write(struct file *filep, const char *buffer, size_t len,
loff_t *offset){
   sprintf(message, "%s(%zu letters)", buffer, len);
       // appending received string with its length

   size_of_message = strlen(message);
      // store the length of the stored message

   printk(KERN_INFO "EBBChar: Received %zu characters from the user\n", len);
   return len;
}

static int dev_release(struct inode *inodep, struct file *filep){
   printk(KERN_INFO "EBBChar: Device successfully closed\n");
   return 0;
```

```
}
module_init(labchar_init);
module_exit(labchar_exit);
```

ارتباط کاربر و ماژول

قطعه کد زیر نیز برای ارتباط فضای یوزر با این ماژول است:

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>

#define BUFFER_LENGTH 256
      ///< The buffer length (crude but fine)
static char receive[BUFFER_LENGTH];
      ///< The receive buffer from the LKM

int main(){
   int ret, fd;
   char stringToSend[BUFFER_LENGTH];
   printf("Starting device test code example...\n");
   fd = open("/dev/oslabchar", O_RDWR);         // Open the device with
read/write access
   if (fd < 0){
      perror("Failed to open the device...");
      return errno;
   }
   printf("Type in a short string to send to the kernel module:\n");
   scanf("%[^\n]%*c", stringToSend);
      // Read in a string (with spaces)
   printf("Writing message to the device [%s].\n", stringToSend);
```

```
   ret = write(fd, stringToSend, strlen(stringToSend));
       // Send the string to the LKM
   if (ret < 0){
      perror("Failed to write the message to the device.");
      return errno;
   }

   printf("Press ENTER to read back from the device...\n");
   getchar();

   printf("Reading from the device...\n");
   ret = read(fd, receive, BUFFER_LENGTH);
      // Read the response from the LKM
   if (ret < 0){
      perror("Failed to read the message from the device.");
      return errno;
   }
   printf("The received message is: [%s]\n", receive);
   printf("End of the program\n");
   return 0;
}
```

برای کامپایل از همان فایل make قبل استفاده کنید.

ابتدا ماژول را به کرنل اضافه کنید و سپس برنامه فضای کاربر را با دسترسی root اجرا کنید.