ماژول‌نویسی برای هسته لینوکس

سعید تقوی s.taghavi@ece.ut.ac.ir

http://www.irantux.org

ماژول هسته چیست؟

اولیـــن سوالی کـــه ممکن است به ذهن خواننده برسد این است که ماژول هسته دقیقا چیست؟ در پاسخ باید گفت که ماژول‌ها تـکه کدهایی هـستند که در حین اجرای هسته لینوکس می توانند وارد آن شده و یا از آن خارج شوند. این تکه کدها عملکرد هسته را بدون نیاز به راه اندازی دوباره کامپیوتر توسعه می‌دهند.

به عنوان مثال یـــک نـــوع از مـــاژول‌ها device driver ها هستند کـه به هسته امکان استفاده از قابلیت سخت افزار ها را می‌دهند.

اگر ماژول‌ها وجود نداشتند، بـرای هر قابلیتی که می‌خواستیم بـــه هسته اضافه کنیم یـا از آن کم کنیم، می‌بایستی یک بار هسته را کامپایل می‌کردیم و برای استفاده از ان قابلیت یا حذف آن یک بار سیستم را از نو راه اندازی می‌کردیم.

ماژول ها چگونه به هسته وارد می شوند؟

شما می‌توانید بـــا اجرای دستور lsmod ماژول‌هایی کـــه هـــم اکنـون در هسته وارد شده‌انـد را ببینـید و از اطـلاعات آنها باخبر شوید. این دستور اطلاعات خود را از فایل proc/modules/ دریافت می‌کند.

هنگامی که هسته، به امکان و عملکردی نیاز دارد که هم اکنون در آن نیست، یکـــی از deamon های آن بـــه نـام kmod دستور modprobe را اجرا می‌کند تا ماژول مربوطه که آن عملکرد را دارد وارد هسته شود . هنـگامی که modprobe اجرا می شود به آن یک رشته کاراکتر به دو صورت زیر داده می‌شود:

۱) نام ماژول مانند softdog یا ppp

۲) یک مشخصه کلی مانند char-major-10-30

اگر حالت اول بـــه modprobe داده شود، ایـــن دستور بـه دنبال فایلی به نام softdog.ko یا ppp.ko با روشی که در ادامه می‌آید می‌گردد. ولی اگـــر حـــالـت دوم بـــه modprobe داده شود، ایـــن دستـــور ابـــتدا بـــه دنـــبال رشتـه کاراکتر در فایل /etc/modprobe.conf می گردد و اگر توانست alias یا مستعاری مانند:

alias char-major-10-30 softdog

پیدا کند، متوجه می‌شود کـــه ایـــن نـــام کلی که در اینجا char-major-10-30است به ماژول softdog اشاره می‌کـــند که فایل ماژول آن softdof.ko می‌باشد.

در مرحله بعد modprobe فایل /lib/modules/version/modules.dep را باز کرده و به دنبال ماژول‌هایی می‌گـــردد کـه باید قبل از ماژول مورد نظر به هسته وارد شوند. ایـــن فـــایل به وسیله دستور depmod -a ایجاد می‌شود و حـــاوی وابستگی بین ماژول هاست.

به عنوان مثال اگر به دنبال مـــاژول msdos.ko در ایـــن فـــایل بگردید خواهید دید که به ماژول دیگری به نام fat.ko وابسته است یعنی برای اینکه msdos.ko وارد هسته شود حتما باید قبل از ان fat.ko وارد شده باشد. ایـــن مسـاله برای fat.ko نیز تـــکـــرار شده تـــا بـــه مرحله‌ای برسیم که دیگر وابستگی موجود نباشد. در نهایت modprobe دستور insmod را به کار می‌برد تا ابتدا وابستگی‌ها را به هسته وارد کرده و در نهایت ماژول مورد نظر ما به هسته وارد می‌شود.

پس modprobe وظـــیـفه پـیـدا کـــردن مـــاژول، تعیین وابستگیهای آن و وارد کردن آن بـــه هستـــه بـــه وسیـــله صـدا کردن insmod را دارد در حالی که insmod فقط وظیفه وارد کردن آن ماژول به هسته را دارد.

به عنوان مثال اگر بخواهیم به صورت دستی msdos.ko را وارد هسته کنیم به صورت زیر عمل می کنیم :

```
#insmod /lib/modules/2.6.11/kernel/fs/fat/fat.ko
#insmod /lib/modules/2.6.11/kernel/fs/fat/msdos.ko
```

معادل دو دستور بالا با modprobe به صورت زیر است:

```
#modprobe msdos
```

مطلب قابل ذکر این است که insmod مسیر کامل تا فایل ماژول را می‌خواهد در حالی که modprobe فقط نـــام ماژول را می‌گیرد.

قبل از شروع

قبل از اینکه وارد کد و کدزنی شویم چند نکته مهم را بررسی می‌کنیم:

۱) modversioning: یـــک ماژول کـه برای یک هسته خاص کامپایل شده است بر روی هسته دیگر load نخواهد شد مگر اینکه شـــما CONFIG_MODVERSIONS را در هسته فـــعـــال کنـــیـد. در قـســـمت‌های بـعـــد بیشتر به این مقوله خواهیم پرداخت.

۲) ماژول‌ها نمی‌توانند چـــیزی به غیر از خطاها و هشدارها را بر روی صفحه نمایش نـشان دهند. آنها بـــرای نـشان دادن اطلاعات خود، آنها را در log فایلها می‌نویسند.

۳) مورد سوم که کاملا مورد قبول بنده نمی‌باشد این است که نویسنده می گوید:

«اغلب توزیع کنندگان لینوکس کد منبع هسته را که مورد Patch نیز قرار گرفته به طرز غیر استانداردی توزیع می‌کنند که ممکن است باعث ایجاد مشکلاتی شود. یکی از شایع ترین این مشکلات فایل های ناقص Header برای هسته لینوکس هستند. شما برای ماژول نویسی نیاز دارید که فایل های Header زیـــادی را در کـــدهای خـود ضمیمه کـنـید و فایـــلهای ناقص اغلب فایل هایی هستند که برای ماژول نویسی به کار می روند.» نویـــسنده پیشنهاد می‌کند کـه برای جلوگیری از این مشکل هسته را برای خود کامپایل کنید.

یک مثال – ساده ترین ماژول

برای شروع از مثال سنتی Hello World! شروع می‌کنیم. فایلی به نام hello-۱c. باز کرده و کد C زیر را در آن بنویسید:

```
#include  <linux/module.h>  /*needed by all modules  */
#include  <linux/kernel.h>  /*needed for Macros like KERN_INFO */
int  init_module(void)  /* this
function is called as initialization  for all modules */

{
        printk(KERN_INFO “Hello World1.\n”);

        /* if this function returns non
zero means init_module failed and
this module can’t be loaded. */
        return  ;۰
}

void  cleanup_module(void) /* it is
called when module is terminated and unloaded */
{
        printk( KERN_INFO “Goodbye World۱.\n”);
}
```

هـــر ماژول هسته‌ای حداقل بایستی ۲ تابع داشته باشد. اولـی تابع شروع که init_module() نامیده می‌شـــود و هنگام load شدن ماژول در هسته صدا زده می‌شود و دیـــگـــری تـــابـــع پـــایان که

cleanup_module() نامیده می‌شود و هنگام unload شدن ماژول از هسته صدا زده می‌شود. در قسمت‌های بعد به این موضوع می‌پردازیم که بعد از هسته ۲.۳.۱۳ شما می‌توانید هر نام دیگری برای این دو تابع قرار دهید. با این حال خیلی از افراد هنوز از این استاندارد قدیمی استفاده می‌کنند. دو فایل Header در این کد ضمیمه شده‌اند. یکی linux/module.h می‌باشد که برای هر ماژولی مورد نیاز است و تعریف خیلی از توابع را در خود دارد و دیگری linux/kernel.h می‌باشد که حاوی تعدادی ماکرو می‌باشد مانند KERN_INFO.

مختصری درباره printk()

بر خلاف آن چیزی که ممکن است درباره printk() تصور کنید این تابع چیزی در صفحه نمایش چاپ نمی کند و برای کار با کاربر نیست. این تابع برای مکانیزم log هسته به کار میرود . هر printk() با یک اولویت می‌آید که در این مثال ماکروی KERN_INFO برای این منظور به کار رفته است. تعداد ۸ اولویت وجود دارند که به صورت ماکرو در فایل linux/kernel.h تعریف شده‌اند. اگر شما این اولویت را تعیین نکنید به طور پیش فرض DEFAULT_MESSAGE_LOGLEVEL به آن تخصیص می‌یابد .

اگر این اولویت کمتر از اولویت int console_loglevel (که در linux/kernel.h تعریف شده) باشد، message در دستور printk() بر روی صفحه ظاهر می‌شود. اگر syslogd و یا klogd در سیستم در حال اجرا باشند این message در فایل var/log/messages/ نوشته می‌شود.

کامپایل ماژول های هسته

ماژول های هسته کمی متفاوت نسبت به برنامه های معمولی کامپایل می‌شوند. برای اینکه بتوانید یک ماژول هسته را به درستی کامپایل کنید، نیاز به تنظیمات بسیار زیادی دارید. با پیچیده‌تر شدن ماژول‌ها این تنظیمات پیچیده‌تر می‌شوند. خوشبختانه مکانیزمی به نام kbuild وجود دارد که تمام این تنظیمات را انجام می‌دهد. برای اگاهی بیشتر از این مکانیزم به فایلهای مستند هسته که در آدرس linux/Documentation/kbuild/modules.txt کد منبع هسته موجود است مراجعه کنید. برای

اینکه بتوانیم از مکـانیزم kbuild استفاده کنیم، بایستی Makefile ای با استاندارد آن بنویسیم. برای این کار یک فایل به نام Makefile باز کرده و دستورات زیر را در آن بنویسید:

```
obj-m += hello-1.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
```

حال با اجرای دستور make ماژول خود را کامپایل کنید. در هسته ۲.۶ بـــه بـــعد از پسوند ko بـــرای نـــامیدن مـــاژول‌های هسته استفاده شده است که به راحتی قابل تمییز از o که پسوند فایل های object است می‌باشد.

برای بدست آوردن اطلاعاتی از ماژول خود دستور زیر را اجرا کنید:

```
#modinfo hello-1.ko
```

برای وارد کردن ماژول خود در هسته از دستور زیر استفاده کنید:

```
#insmod ./hello-1.ko
```

اگر بعد از اجرای این دستور فایل var/log/messages/ را باز کرده و به انتهای آن بروید، خواهید دیـــد کـه ماژول hello-1 در هسته load شده است. با دستور lsmod نیز ماژول load شده را خواهید دید. بـــرای unload یـــا خارج کردن ماژول خود از هسته از دستور rmmod به صورت زیر استفاده کنید :

```
#rmmod hello-1
```

دوباره اگر فایل var/log/messages/ را بـــاز کنید و به انتهای آن بـــروید خواهید دید که ماژول hello-1 از هسته خارج شده است.

مثال hello world – قسمت دوم

همان طور کـــه در قسمت قبـل گفتیم بعد از هسته ۲.۳.۱۳ می توانید برای دوتابع init_module() و cleanup_module() اســـامی دیگری اختیار کنید. این امکان توسـط دو مـــاکـــروی module_init() و module_exit() که در فایل <linux/init.h> تعریف شده اند میسر می گردد.

دو تـابع شروع و پایان ماژول بایستی قبـــل از این دو ماکرو تعریف شده باشند در غیر این صورت خطای کامپایلر را دریافت خواهید کرد .

برای روشن شدن این موضوع به مثال hello-2.c توجه کنید.

```
/*  hello-2.c  */


#include  <linux/module.h> /*
needed by all modules */


#include  <linux/kernel.h>/*needed
for macros like KERN_INFO,KERN_ALERT,etc */


#include  <linux/init.h>/*needed
for module_init() & module_exit ()*/


static  int  __init
hello_2_init(void)
{
        printk(KERN_INFO “Hello, World 2\n”);
        return 0;
```

```
}

static  void  __exit

hello_2_exit(void)

{

        printk(KERN_INFO “Goodbye, World 2\n”);

}

module_init(hello_2_init);/*sets

hello_2_init() as initialization function */


module_exit(hello_2_exit);/*sets

hello_2_exit() as termination function */
```

برای اینکه این کد مانند مثال hello-1.c مورد کامپایل قرار گیرد کافی است در Makefile خط زیر را اضافه کنید:

```
obj-m += hello-2.o
```

پس از اعمال این تغییر Makefile به صورت زیر خواهد بود:

```
obj-m  +=  hello-1.o

obj-m  +=  hello-2.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
```

ماکروهای __init و __exit

ماکـروی __init در مورد درایـــورهایی کـــه بـــه صـورت built-in در هسته استفاده می‌شوند باعث می‌شود که پس از به اتمام رسیدن تابع init، این تابع از حافظه خارج شده و حافظه‌ای کـــه برای آن گرفته شده است آزاد می‌گـردد. ایـن ماکرو در مورد درایورهایی که قرار است به صورت ماژول به هسته وارد شوند تاثیری نخواهد داشت.

همانند __init کـه برای توابع init در نظر گرفته شده ماکروی __exit برای حذف فضای تابع exit استفاده می‌گـردد و برای درایور هایی که به صورت ماژول وارد هسته می شوند تاثیری نخواهد داشت.

این دو ماکرو در فایل <linux/init.h> تعریف شده‌اند و باعث گرفتن و آزاد کردن حافظه از فضای هسته می‌شوند.

اگر در هنگام بوت هسته پیغامی مانند:

Freeing unused kernel memory : 236k freed

را دریافت کردید دلیل این موضوع استفاده از این دو ماکرو است. عـــلت استـــفـــاده از static را در قســـمت بعد بـه تفصیل بررسی خواهیم کرد.

مثال Hello world – قسمت سوم

به مانند دو ماکروی __init و __exit که برای توابع به کار می‌رود، مـــاکـــرویی به نام __initdata داریم که برای متغیرها به کار می رود و همان تاثیرات ذکر شده در بالا برای متغیر ها را دارد. به مثال ماژول hello-3.c توجه کنید :

```
/*  hello-3.c  */
#include  <linux/module.h>
#include  <linux/kernel.h>
#include  <linux/init.h>
static  int  hello_3_data
__initdata  =  3;
```

```
static int __init
hello_3_init(void)
{
        printk(KERN_INFO “Hello, World %d\n”, hello_3_data);
        return 0;
}
static void __exit
hello_3_exit(void)
{
        printk(KERN_INFO “Goodbye, World 3\n”);
}
module_init(hello_3_init);
module_exit(hello_3_exit);
```

به مانند قبل با اضافه کردن obj-m += hello-3.o در Makefile این فایل مورد build قرار می گیرد.

مثال Hello world – قسمت چهارم

در این مثال به بررسی چند ماکـــروی دیگر می‌پردازیـــم که بعضی امکانات مفید مانند license و documentation را به یک ماژول اضافه می کنند.

ایـــن ماکروها عموما از هستـه۲.۴ و به بعد اضافه شده‌انـد . license را می توانید با ماکروی()MODULE_LICENSE تعیین کنید. مـــاکـــروهـــای ()MODULE_DESCRIPTION و ()MODULE_AUTHOR بـــه ترتیب بـــرای توضیح کاری که ماژول انجام می‌دهد و نویسنده ماژول به کار می روند.

برای اینکه مشخص کنید که ماژول چه دسته‌ای از دستگاه‌ها را پشتیبانی می‌کند از ماکروی MODULE_SUPPORTED_DEVICE() استفاده کنید. این ماکروها همگی در <linux/module.h> تعریف شده‌اند.

این اطلاعات که در ماژول‌ها ذخیره می‌شوند، توسط ابزارهایی مانند objdump که برای نشان دادن اطلاعاتی از فایل های object به کار می‌روند قابل مشاهده هستند. به مثال hello-4.c که این موارد را نشان می‌دهد توجه کنید:

```
/*  hello-4.c  */
#include  <linux/module.h>
#include  <linux/kernel.h>
#include  <linux/init.h>
#define  DRIVER_AUTHOR  “Peter Jay
Salzman  <p@dirac.org>”
#define  DRIVER_DESC  “A sample
driver”
static  int  __init
hello_4_init(void)
{
        printk(KERN_INFO “Hello, World 4\n”);
        return 0;
}
static  void  __exit
hello_4_exit(void)
{
        printk(KERN_INFO “Goodbye, World 4\n”);
```

```
}

module_init(hello_4_init);

module_exit(hello_4_exit);

MODULE_LICENSE(“GPL”);/* macro for setting license information/*

MODULE_AUTHOR(DRIVER_AUTHOR); /*idenfying module author/*

MODULE_DESCRIPTION(DRIVER_DESC); /*describe module with this macro/*

MODULE_SUPPORTED_DEVICE(“testdevice”); /* identifying devices module can support and work – here is test and is not important/*
```

فرستادن ورودی های خط دستور به یک ماژول

ماژولها می‌توانند ورودی‌هایی از طـریق خط دستور دریافت کنند البته این کار با استفاده از argc و argv کـه در برنامه‌های معمولی استفاده می شوند نیست.

بـــرای این کار متغیرهایی را که می‌خواهید از طریق خط فرمان مقداردهی کنید را بـــه صـــورت global تعـــریف کـــرده و بـا استفاده از مـــاکـــروی ()MODULE_PARM این مکانیزم را تنظیم کنید. ایـــن ماکـــرو دو ورودی می‌گیـــرد. اولـی نام متغیر و دومی نوع متغیر . نوع های قابل قبول عبارتند از:

(s" string) - ("l" long) - ("i" integer) - ("h" short int) - ("b" byte ")

String ها باید به صورت *char تعریف شوند. insmod در زمـــان اجرا فضای لازم برای ان را می گیرد . به عنوان مثال شما می توانید متغیر myvariable را که یک int است در کد خود تعریف کرده و از طریق خط فرمان ان را مقدار دهی کنید .

```
int myvariable = 10 ; // 10 is default value
MODULE_PARM( myvariable , "i" );//setting the type (amout of memory) for it
```

سپس به صورت زیر آن را مقدار دهی کنید:

```
#insmod ./mymodule myvariable=250
```

ساختار داده ارایه نیز در این روش مورد پشتیبانی است . به عنوان مثال در کد زیر:

```
int myarray[4];
MODULE_PARM( myarray , "3-9i" );/*setting an array of integer with max & min
values 9 , 3 respectively */
```

یک آرایه ۴ تایی از integer به ()MODULE_PARM داده شده و مقادیر مینیمم و ماکزیمم ان نیز به وسیله دو عدد ۳ و ۹ تعیین شده است.

مثال hello-5.c که مثال جامع تری است تمام این موارد را نشان می‌دهد. (به comment های کد توجه شود).

```
/* hello-5.c */
#include <linux/module.h> /* needed by all modules */
#include <linux/moduleparam.h> /* needed by macros like module_param,()
module_param_array(),etc . it also has definitions of many functions and structure
like kernel_param */
#include <linux/kernel.h> /* needed for macros like KERN_INFO */
#include <linux/init.h>/* needed for init and exit macros */
#include <linux/stat.h>/* needed by macros like S_IRUSR , S_IWUSR , S_IRGRP , S_IWGRP */
MODULE_LICENSE("GPL");//setting the module license
MODULE_AUTHOR("Peter Jay Salzman");//setting the module author
/* defining some variable for input from shell prompt */
static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "blah;"
/* S_IRUSR , S_IWUSR , S_IRGRP , S_IWGRP macros are used for setting
permissions to access to variables and fuctions as common Linux permission
management system . you know in Unix systems we have 3 group of people
 (owner – group – others ) and 3 kind of permissions ( read – write – execute (rwx))
for each group. in modules we use bitwise OR of these macros for setting a user to
set or get a variable or function . */
module_param( myshort , short , S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
MODULE_PARM_DESC(myshort, "A short integer");
module_param( myint , int , S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long , S_IRUSR);
```

```
MODULE_PARM_DESC(mylong, “A long integer”);
module_param(mystring, charp , 0000);
MODULE_PARM_DESC(mystring, “A character string”);
static int __init hello_5_init(void)
{
printk( KERN_INFO “Hello, World 5\n”=============);
printk( KERN_INFO “myshort is a short integer: %hd\n” , myshort );
printk( KERN_INFO “myint is an integer: %d\n” , myint );
printk( KERN_INFO “mylong is a long integer: %ld\n” , mylong );
printk( KERN_INFO “mystring is a string: %s\n”, mystring);
return 0;
}
static void __exit hello_5_exit(void)
{
printk( KERN_INFO “Goodbye, world 5\n”);
}
module_init(hello_5_init);
module_exit(hello_5_exit);
```

دستورات زیر را اجرا کنید تا با نحوه کار این ماژول بیشتر اشنا شوید (فایل var/log/messeges/ دیده شود.)

```
#insmod ./hello-5.ko mystring=”bebop” mybyte=255 myintarray=-1
#rmmod hello-5
#insmod ./hello-5.ko mystring=”supercalifragilisticexpialidocious\ ”
<mybyte=256 myintarray=-1,-1
#rmmod hello-5
#insmod ./hello-5.ko mylong=hello
```

نوشتن یک ماژول در چندین فایل

بعضی اوقات نیاز دارید که یک ماژول هسته را در چندین فایل پیاده سازی کنید.

مثال مورد بررسی شامل دو فایل start.c و stop.c است.

```
/* start.c */
#include <linux/kernel.h>
#include <linux/module.h>
int init_module(void)
{
printk(“Hello, world – this is the kernel speaking\n”);
return 0;
}
```

```
/* stop.c */
#include <linux/kernel.h>
#include <linux/module.h>
void cleanup_module(void)
{
printk(“<1>short is the life of a kernel module\n;)”
}
```

نکته مهم: نحوه build با استفاده از kbuild است . برای این موضوع به تغییرات حاصل در Makefile توجه کنید.

```
obj-m += hello-1.o

obj-m += hello-2.o

obj-m += hello-3.o

obj-m += hello-4.o

obj-m += hello-5.o

obj-m += startstop.o

startstop-objs := start.o stop.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
```

ماژول ها چگونه آغاز و پایان می‌یابند؟

برنامه‌های معمولی , معمولا با تابعی به نام ()main اغاز شده , لیستی از دستورات را انجام داده و به پایان می‌رسند. ماژول‌های هسته دراین مورد متفاوت عمل می‌کنند. یک ماژول همیشه با تابع ()init_module و یا تابعی که به وسیله ماکروی ()module_init به عنـــوان ورودی ثبت شده اغاز می‌گردد. در حقیقت‌این تابع قابـــلیت خود را به هسته اعلام می‌دارد و به هسته‌این امـــکان را داده که در موقع نیاز از توابع ماژول استفاده کند. پس ازاینکه‌این تابع به پایان می‌رسد توابع ماژول دیگر اجرا نخواهند شد تا زمانی که هسته بخواهد ازاین توابع استفاده کند.

تمامی‌ماژول ها با صدا کردن cleanup_module() و یا تابعی که به وسیله ماکروی module_exit() به عنوان ورودی ثبت شده به پایان می‌رسند.این تابع تمامی‌اعمالی را که تابع ورودی انجام داده خنثی می‌کند.

توابعی که در اختیار ماژول ها هستند

توسعه دهندگان معمولا از توابعی استفاده می‌کنند که خود انها را تعریف نکرده اند. یک مثال ساده ازاین توابع , تابع ()printf می‌باشد. شما ازاین توابع که در کتابخانه استاندارد زبان C تعریف شده اند استفاده

می‌کنید. کداین توابع تا زمان لینک یا پیوند در کد شما وارد نمی‌شوند و در زمـــان لینک ادرس موردنظر در کد شما به آدرس این کد اشاره داده خواهد شد.

ماژول‌های هسته دراین مورد نیز متفاوت هستند. شما اگر در مثال‌های قسمت قبل دقت کرده باشید ما از تابعی به نام ()printk استفاده کردیم امـــا از کتابخانه استانداردی بـــرای IO استفاده نکردیم. ایـــن مـــوضـــوع بـــه ایـــن خـــاطر است که ماژول ها فایل‌های objectای هستند که سمبول‌هایشان در هنـــگـــام insmod مشخص می‌گردند. کداین سمبول ها در خود هسته وجود دارد. اگـــر می‌خواهیـــد سمبـــولهایی را کـــه هسته تعـــریف کـــرده اســـت را مشاهده کنید به فایل /proc/kallsyms رجوع کنید.

یکـــی از نـــکات بسیار مهمی‌که باید مـــورد توجه قرار گیـــرد تفـــاوت بین توابع کتابخانه‌ای (library functions) و توابع سیستمی(system calls) است. تـــوابـــع کتـــابخانه‌ای توابعی سطح بالا هستند که در فضای کاربر اجرا می‌شوند و در حقیقت واسط بین کاربر و توابع سیستمی‌که اصل کار هر برنامه را می‌کنند می‌باشند. توابع سیستمی‌در فضای هسته اجرا می‌شوند. تابع کتابخانه‌ای ()printf یک تابع نمایش بسیار عمومی‌به نظر می‌اید اما در حقیقت‌این تابع رشته ورودی را شکل دهی کرده و ان را تحویل تابع سیستمی()write (که کار حقیقی را انجام می‌دهد) می‌دهد.

برای‌اینکه از جزییات عملکرد ()printf با خبر شوید کد زیر را در یک فایل به نام hello.c بنویسید.

```
#include <stdio.h>
int main() { printf("hello"); return 0 ;}
```

و با دستور زیر ان را به فایل اجرایی hello تبدیل کنید:

```
$gcc -Wall -o hello hello.c
```

حال دراین مرحله hello را به صورت زیر اجرا کنید:

```
$strace ./hello
```

چیزی که مشاهده خواهید کرد مجموعه توابع سیستمی‌است که برنامه hello صدا زده است. در چند خط اخر خروجی دستور قبل , خطی به صورت زیر خواهید دید :

write(1 , ”hello” , 5hello)

این خط در حقیقت خطی است که اصل عملکرد ()printf اجرا شده است. برای اگاهی بیشتر از تابع سیستمی()write دستور man 2 write را اجرا کنید.

شما حتی می‌توانید ماژول‌هایی بنویسید که توابع سیستمی‌هسته را تغییر دهد. cracker ها معمولا ازاین خاصیت برای نوشتن backdoor یا trojan ها استفاده می‌کنند.

فضای کاربر در مقابل فضای هسته

هسته به تمام منابع سیستم دسترسی مستقیم دارد.این منابع می‌توانند کارت ویدیو , دیسک سخت یا حافظه باشند. در حالی که برنامه‌های معمولی، بـــر سر تصاحب منابع سیستم رقابت دارند. هم اکنون که من درحال نوشتن‌این متن هستم، updatedb در حال به روز رسانی پایگاه داده locate می‌باشد، logger ها در حـــال ثبـــت وقـایع هستند. بنابراین برنامه‌های syslogd، updatedb،openoffice.org بـــه طـــور همـــزمان از هارد دیسک استفاده می‌کنند. واضح است که هسته بایستی ترتیب استفاده را مشخص کند و به برنامه‌ها و کاربران اجازه دسترسی بـــه منـــابع هـر زمان که دوست دارند ندهد.

بـــرای رسیدن به‌این هدف یک CPU در حالـــت‌های مختلفی می‌تواند بـــه اجرای دستورات بپردازد. هـــر حـــالتی سطح مختلـفـــی از آزادی بـــرای کســـب منـابـع سیستم در اختیار کاربران قرار می‌دهد. معـماری Intel 80386 چهار حالت یا اصطلاحا mode دارد. لینوکس تنها از دو حالت استفاده می‌کند:

۱) بالاترین سطح آزادی یا حالت سرپرست و مدیر: که در ایـــن حالت همه چیز امکان پذیر است و به همه منابع سیستم می‌توان دسترسی مستقیم داشت.

۲) پایین ترین سطح آزادی یا حالت کاربر: که دراین حالت استفـــاده از منابع سیستم تنـــها بـــا اجازه هسته امکان پذیر است.

بحث قبلی در مورد توابع کتابخانه‌ای و توابع سیستمی‌را به خاطر آورید. تـــوابع کتابخانه‌ای اغلب در حالت کاربر استفاده می‌شوند.این توابع نیز یک یا چند تابع سیستمی‌را صدا می‌زنند. ایـن توابع سیستمی‌بااین که از طرف توابع کتابخانه‌ای صـــدا زده می‌شوند در فضای هسته اجرا می‌شـــوند بدلیل اینـــکه ایـن توابع

بخشی از هسته هستند. هنـــگـــامی کـــه تابع سیستمی‌به طور کامل اجرا شد حالت اجرای دستورات به حالت کاربر بر می‌گردد.

فضای متغیرها (Name Space)

هنگامی کـــه شما یک برنامه به زبان C می‌نویسید از اسامی‌راحتی به عنوان نام متغیرهایتان استفاده می‌کنید و با این کار خود سعی در هرچه بیشتر خوانا تر کردن کد خود می‌کنید. اگـــر شـــما روتین‌هایی بنویسید که بخشی از یک مساله بـــزرگتر باشند، هر متغیر عمومی‌یا global که استفاده می‌کنید جزیی از مجموعه متغیرهای عمومی‌دیگران خـــواهد بود. بنابراین مواردی پیش خواهد امد که متغیرهای عمومی‌در دو کد مجزا یکسان باشند و کار دچار مشکل شود.

هنگامی کـــه یک برنامه دارای متغیرهای عمومی‌بسیاری می‌باشد کـــه بـــه اندازه کافی معنادار نیستند به‌این مساله آلودگی فضای متغیرها یا name space pollution گویند.

در پروژه‌های بزرگ بایستی سعی شود که روش‌هایی برای‌ایجاد نام‌های متغیرها ایجاد شوند که نام متغیرها هـــم یکتا باشند و هم دارای معنی مناسب باشند. یکی از همین پروژه‌های بسیار بزرگ هسته لینوکس است. هنـــگام نوشتن کد هسته، حتـــی کوچکترین ماژول نیز با کل هسته پیوند (link) خواهد شد. بـــنابراین‌این موضوع از اهمیت بسیار بالایی برخوردار است. بهترین راه برای حل‌این مساله تعریف کردن متغیرهای عمومی بـــه صورت static و یا استفاده از پیشوند یـــا پسوندهای مناسب برای نام گذاری است. معمولا تمام پیشوندهایی که در هسته تعریف می‌شوند با حروف کوچک اغاز می‌شوند.

اگـــر شما نمی‌خواهید که متغیرهایتان را به صورت static تعریف کنید گزینه دیگری که پیش روی شماست تعریف یک جدول سمبول ها (symbol table) و ثبت آن در هسته است. بعدا به‌این مقوله بیشتر خواهیم پرداخت.

فایل proc/kallsyms/ تمامی سمبول‌هایی که در هسته تعریف شده‌اند و شما می‌توانید از آنـــها برای مـــــاژول‌های خود استفاده کنید را نگهداری می‌کند.

فضای کد (Code Space)

مدیریت حافظه یکی از پیچیده ترین و با اهمیت ترین موضوعات در هسته است (موضوعی که بیشتر کتاب OReilly's Understanding the Linux Kernel دراین باره می‌باشد). ما دراین راهنما قصد نداریم که در زمینه مدیریت حافظه حرفه‌ای شویم. اما نیاز داریم که حقایق بسیار مهمی‌را بدانیم تا بتوانیم ماژول‌های واقعی برای هسته لینوکس بنویسیم.

اگر شما چیزی در مورد segfault ها نمی‌دانید ممکن است متعجب شوید که اشاره گر ها (pointers) که در برنامه نویسی به خصوص با زبان C به کار می‌روند واقعا به آدرسی از حافظه اشاره نمی‌کنند.

هنگامی‌که یک پروسه ایجاد می‌شود، هسته قسمتی از حافظه فیزیکی را گرفته و در اختیار پروسه قرار می‌دهد که برای اجرای کد، نگهداری متغیرها، stack،heap و تمام چیزهایی که یک پروسه نیاز دارد از ان استفاده کند.این فضا برای تمام پروسه‌ها از آدرس ۰ شروع شده و به میزان خطوط آدرس دهی (Address Bus) حافظه (۲^۳۲ بایت) قابل گسترش است. از آنجایی که پروسه‌ها بر روی یکدیگر قرار نمی‌گیرند، بنابراین هر چند پروسه که به یک آدرس دسترسی دارند (مثلا xbfffff978۰) در حقیقت به آدرس‌های متفاوتی از حافظه فیزیکی دسترسی دارند! این مطلب دقیقا همان است که اشاره گرها به آدرسی از حافظه فیزیکی اشاره نمی‌کنند.

در حقیقت در تمام پروسه‌هایی که دارای اشاره گری با مقدار xbfffff978۰ هستند، این اشاره گر به نوعی از offset که فقط در آن پروسه تعریف شده است اشاره می‌کند. هیچ وقت دو پروسه نمی‌توانند به فضای یکدیگر وارد شوند (البته راه‌هایی وجود دارد که بعدا ذکر خواهیم کرد).

خود هسته نیز فضایی از حافظه را برای خود در اختیار دارد. یک ماژول کدی است که به صورت دینامیک می‌تواند وارد این فضا از حافظه شده یا از ان خارج شود. توجه به‌این نکته بسیار حیاتی است که هر ماژول سهمی‌از فضای هسته را استفاده می‌کند و فضای جدیدی برای آن گرفته نمی‌شود. بنابراین اگر ماژول شما دچار segmentation fault شود، کل هسته دچار segmentation fault خواهد شد.

نکته دیگراین است که‌این بحث برای تمام سیستم‌هایی که به صورت microkernel هستند درست است. دو نمونه ازاین سیستم عامل ها GNU Hurd و QNX Nutrino هستند.

راه انداز ها (Device Drivers)

یکی از انواع ماژول ها راه اندازهای قطعات سخت افزاری هستند که روتین‌های لازم برای استفاده از قطعات سخت افزاری مانند کارت TV یا پورت سریال و.... را در اختیار قرار می‌دهند.

در Unix هر قطعه سخت افزاری با یک فایل که در dev/ قرار می‌گیرد مشخص می‌گردد. به‌این فایل، فایل دستگاه (device file) گویند که برای برقراری ارتباط برنامه‌های مختلف با قطعه سخت افزاری مورد نظر می‌باشد.

راه اندازی که به‌این فایل مربوط می‌شود، در مقابل ارتباط برنامه‌های کاربران در مورد این قطعه سخت افزاری پاسخ می‌دهد. بنا براین راه انداز کارت صدایی مانند es1370.o فایل دستگاه dev/sound/ را به کارت صدای Ensoniq IS1370 متصل می‌کند. یک برنامه مانند mp3blaster می‌تواند از dev/sound/ بدون‌اینکه حتی بداند چه کارت صدایی نصب شده است استفاده کند.

اعداد اصلی (major) و فرعی (minor)

بیایید چند فایل device را مورد بررسی قرار دهیم. در مثال زیر ۳ پارتیشن اول هارددیسک master نشان داده شده اند:

```
$ls -l /dev/had[1-3]
brw-rw---- 1 root disk 3, 1 Jul 5 2000 /dev/hda1
brw-rw---- 1 root disk 3, 2 Jul 5 2000 /dev/hda2
brw-rw---- 1 root disk 3, 3 Jul 5 2000 /dev/hda3
```

به ستونی که با کاما از هم جدا شده‌اند توجه کنید. اولین عدد دراین ستون عدد اصلی دستگاه یا device major number نامیده می‌شود. دومین عدد دراین ستون عدد فرعی دستگاه یا device minor number می‌باشد. عدد اصلی به شما می‌گوید که چه راه اندازی برای‌این دستگاه مورد استفاده قرار گرفته است. به هر راه اندازی یک عدد یکتا نسبت داده شده است. تمام دستگاه‌های با عدد اصلی

یکسان توسط یک راه انداز کنترل می‌شوند. در مثال بالا عدد اصلی هر سه دستگاه ۳ می‌باشد که نشان می‌دهد که هر سه توسط یک راه انداز کنترل می‌شوند.

عدد فرعی توسط خود راه انداز برای تمایز بین دستگاه‌هایی که تحت کنترل دارد استفاده می‌شود. در مثال بالا بااینکه هر سه دستگاه توسط یک راه انداز کنترل می‌شوند ولی عددهای فرعی متفاوتی (و البته یکتا) دارند چون که از دید راه انداز ان ها سه دستگاه متفاوت هستند.

دستگاه ها به دو دسته تقسیم می‌شوند:

۱) دستگاه‌های کاراکتری (character devices)

۲) دستگاه‌های بلوکی (block devices)

تفاوت بین این دو دسته در این است که دستگاه‌های بلوکی برای انجام تقاضاهای مختلف از بافر (buffer) استفاده می‌کنند. در دستگاه‌های ذخیره سازی اطلاعات خواندن یا نوشتن اطلاعات به صورت مجموعه‌ای (بلوک یا سکتور) از اهمیت بالایی برخوردار است. تفاوت دیگر بین‌این دو نوع دستگاه‌این است که دستگاه‌های بلوکی فقط به صورت بلوکی از داده ها (که می‌تواند اندازه متغیری داشته باشد) ورودی دریافت می‌کنند و خروجی بر می‌گردانند در حالی که دستگاه‌های کاراکتری به هر تعداد بایت می‌توانند ورودی دریافت کنند و خروجی برگردانند.

بیشتر دستگاه ها از نوع کارکتری هستند به دلیل‌اینکه در اکثر موارد نیازی به‌این نوع بافر وجود ندارد و انها اغلب با یک بلوک ثابتی از داده ها کار نمی‌کنند. برای فهمیدن‌اینکه کدام دستگاه به صورت بلوکی و کدام به صورت کاراکتری عمل می‌کنند اولین حرف از خروجی دستور ls –l نوع دستگاه را مشخص می‌کند. اگر c بود کاراکتری و اگر b بود بلوکی می‌باشد. در مثال سه پارتیشن هارد دیسک همگی از نوع بلوکی هستند.

مثالی از دستگاه‌های کاراکتری پورت سریال می‌باشد :

```
$ls -l /dev/ttyS[0-3]
crw-rw---- 1 root dial 4, 64 Feb18 23:34 /dev/ttyS0
crw-rw---- 1 root dial 4, 65 Nov17 10:26 /dev/ttyS1
```

crw-rw---- 1 root dial 4, 66 Jul 5 2000 /dev/ttyS2

crw-rw---- 1 root dial 4, 67 Jul 5 2000 /dev/ttyS3

اگر می‌خواهید بدانید که چه اعداد اصلی در حال حاضر ثبت شده هستند به فایل زیر در کد منبع هسته لینوکس مراجعه کنید :

linux/Documentation/devices.txt

هنگامی‌که سیستم را نصب می‌کنید تمام ان فایل‌های دستگاه ها با دستور mknod ایجاد می‌شوند.

برای‌ایجاد یک فایل دستگاه جدید از نوع کاراکتری به نام coffee با اعداد اصلی و فرعی ۱۲ و ۲ به صورت زیر می‌توان عمل کرد :

#mknod /dev/coffee c 12 2

معمولا فایل‌های دستگاه ها در dev/ قرار می‌گیرد. لینوس توروالدز فایل‌های دستگاه‌هایش را برای اولین بار در dev/ قرار داد و این کار تقریبا مرسوم شده است. با این حال اگر برای تست ماژول هسته‌ای که نوشته‌اید می‌خواهید فایل دستگاهی ایجاد کنید، بهتر است که آن را در دایرکتوری جاری قرار دهید.

به عنوان اخرین نکته، هنگامی که ما می‌گوییم سخت افزار منظورمان کمی متفاوت نسبت به یک قطعه سخت افزاری مثلا PCI Card است که شما می‌توانید در دست خود بگیرید.

به مثال زیر توجه کنید :

$ls -l /dev/fd0 /dev/fd0u1680

brwxrwxrwx 1 root floppy 2, 0 Jul 5 2000 /dev/fd0

brw-rw---- 1 root floppy 2, 44 Jul 5 2000 /dev/fd0u1680

با توجه به چیزی که تاکنون گفتیم شما به راحتی می‌توانید بگویید که هر دو دستگاه بالا بلوکی هستند و توسط یک راه انداز کنترل می‌شوند. شما ممکن است متوجه شده باشید که هر دو درایو فلاپی شما را

نشان می‌دهند. در صورتی که شما فقط یک دستگاه فلاپی در سیستم تان دارید. پس چرا ۲ فایل دستگاه برای آن ایجاد شده است ؟ جواب ایـن سوال در حقیقت پاسخ مساله‌ای است که در بالا اشاره شد.

اولین فایل فلاپی شما با MB۱.۴۴ حافظه را مشخص می‌کند. دومین فایل همان فـــلاپی است با ایـــن تفاوت که توانایی خواندن و نوشتن فـــلاپی‌های بـــا حـــافـــظه MB۱.۶۸ (که به آنها super formatted می‌گویند) را دارد. بنابراین مشاهده می‌کنید که دو فایل دستگاه یک دستگاه را مشخص می‌کند. بنابراین در بحث مان بیشتر به معنی دستگاه و سخت افزار توجه داشته باشید. در اینجااین قسمت به پایان می‌رسد.

راه اندازهای دستگاه‌های کاراکتری

ساختار داده file_operations

ساختار داده file_operations در <linux/fs.h> تعریف شـــده است و اشـــاره گرهایی به توابع مختلف که توسط راه انداز تعریف شده و عملیات مختلفی بر روی دستگاه انجام می‌دهند را نگه داری می‌کند. هر فیلدی از این ساختار داده متناظر آدرس تابعی تعریف شده از راه انداز است که می‌تواند یک درخواست را براورده سازد. به عنوان مثال هر راه انداز دستگاه کاراکتری نیاز به تابعی دارد که بتواند از دستگاه بخواند.

ساختار داده file_operations ادرس توابع مـاژول که این عملیات را انجام می‌دهند در خود نگه می‌دارد. در ذیل تعریف این ساختار داده را بر اساس هسته ۲.۶.۱۳ مشاهده می‌کنید:

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t* );
    ssize_t (*aio_read) (struct kiocb *, char __user *, size_t,  loff_t);
    ssize_t (*write) (struct file *, const char __user *, size_t,  loff_t* );
```

ssize_t (*aio_write) (struct kiocb *, const char __user *,size_t, loff_t);

int (*readdir) (struct file *, void *, filldir_t);

unsigned int (*poll) (struct file *, struct poll_table_struct*);

int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);

long (*compat_ioctl) (struct file *, unsigned int, unsigned long);

int (*mmap) (struct file *, struct vm_area_struct*);

int (*open) (struct inode *, struct file*);

int (*flush) (struct file*);

int (*release) (struct inode *, struct file*);

int (*fsync) (struct file *, struct dentry *, int datasync);

int (*aio_fsync) (struct kiocb *, int datasync);

int (*fasync) (int, struct file *, int);

int (*lock) (struct file *, int, struct file_lock*);

ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t*);

ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);

ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void*);

ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);

unsigned long (*get_unmapped_area)(struct file *,unsigned long, unsigned long, unsigned long, unsigned long);

int (*check_flags)(int);

```
    int (*dir_notify)(struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock* );
};
```

بعضی از عملگرها توسط راه انداز پیاده سازی نمی‌شوند. به عنوان مثال یک راه انداز کارت گرافیک نیازی ندارد که تابع مربوط به خواندن از ساختارهای دایرکتوری (directory structures) را پیاده سازی کند. فیلد متناظر در ساختار داده file_operations که استفاده نمی‌شود به NULL مقدار دهی می‌شود.

امکاناتی در gcc (GNU C Compiler) وجود دارد که مقدار دادن به این ساختار داده را بسیار راحت می‌کند. شما این موارد را در راه اندازهای مدرن امروزی خواهید دید که ممکن است باعث تعجب شما شود. در زیر نحوه مقدار دهی را با استفاده از این امکانات می‌بینید:

```
struct file_operations fops = {
        read: device_read,
        write: device_write,
        open: device_open,
        release: device_release<
};
```

Syntax رایج تر برای مقدار دادن به این ساختار داده روش C99 است که برای سازگاری بیشتر توصیه می‌گردد که از این روش استفاده گردد. در زیر این روش را مشاهده می‌کنید:

```
struct file_operations fops = {
        . read = device_read,
        . write = device_write,
```

```
    . open = device_open,

    . release = device_release

};
```

معنی هر دو ساختار تقریبا مشخص است. توجه داشته باشید که بقیه فیلدهای ساختار داده file_operations که توسط دو روش بالا مقداردهی به صورت مشخص نگردد توسط gcc به NULL مقدار دهی می‌شود.

ساختار داده file

هر دستگاه در هسته توسط یک ساختار داده file نشان داده می‌شود که در <linux/fs.h> تعریف شده است. توجه داشته باشید که این ساختار یک ساختار سطح هسته است و هرگز در برنامه‌های معمولی که در فضای کاربر اجرا می‌شوند ظاهر نخواهد شد. این ساختار با ساختار FILE که در کتابخانه استاندارد زبان C تعریف شده است متفاوت می‌باشد.

نام این ساختار ممکن است شما را دچار اشتباه کند. در حقیقت این ساختار یک تجرید (abstraction) می‌باشد و تصور اینکه این ساختار یک فایل در دیسک که با ساختار inode مشخص می‌گردد را نشان می‌دهد نیز کاملا اشتباه است.

یک اشاره گر به ساختار داده file معمولا filp نامیده می‌شود. در زیر تعریف ساختار file در هسته ۲.۶.۱۳ را می‌بینید:

```
struct file {
    struct list_head
        f_list;
    struct dentry
      * f_dentry;
    struct vfsmount
```

```
        * f_vfsmnt;
    struct file_operations
        *    f_op;
    atomic_t
            f_count;
    unsigned int
        f_flags;
    mode_t
    f_mode;
    loff_t                      f_pos;
    struct fown_struct   f_owner;
    unsigned int          f_uid, f_gid;
    struct file_ra_state  f_ra;
    size_t               f_maxcount;
    unsigned long     f_version;
    void                *f_security;

    /* needed for tty driver,and maybe others */
    void              *private_data;
#ifdef CONFIG_EPOLL
    /* Used by fs/eventpoll.c to link all the hooks to this file */
    struct list_head   f_ep_links;
    spinlock_t          f_ep_lock;
```

```
#endif /* #ifdef CONFIG_EPOLL*/
    struct address_space   *f_mapping;
};
```

بیشتر فیلدهایی که در این ساختار به کار رفته اند (مانند dentry) معمولا توسط راه اندازها استفاده نمی‌شوند.

ثبت (register) یک دستگاه در هسته

همان طور که در قسمت قبل بحث کردیم، دستگاه‌های کاراکتری از طریق فایل‌های دستگاه که معمولا در dev/ قرار می‌گیرند قابل دسترسی هستند. در قسمت قبل به این موضوع نیز پرداختیم که عدد اصلی هر فایل دستگاه نشان می‌دهد که دستگاه توسط چه راه اندازی کنترل شده و عدد فرعی توسط خود راه انداز برای تمایز بین دستگاه‌های تحت کنترل خود به کار می‌رود.

اضافه کردن یک راه انداز به سیستم به معنی ثبت آن در هسته است. مترادف این جمله آن است که در هنگام شروع به کار ماژول (initialization)، به آن یک عدد اصلی نسبت داده شود. شما می‌توانید این کار را با استفاده از تابع register_chrdev که در <linux/fs.h> تعریف شده است انجام دهید:

```
int register_chrdev(unsigned int major, const char *name,
struct file_operations fops);
```

unsigned int major عدد اصلی مورد درخواست و const char *name نام دستگاه شما است که در proc/devices/ ظاهر خواهد شد و struct file_operations *fops نیز اشاره‌گر به ساختار داده file_operations مورد استفاده راه انداز هستند. اگر مقدار بازگشتی این تابع منفی باشد ثبت دستگاه در هسته موفقیت آمیز نبوده است.

به این نکته توجه داشته باشید که ما عدد فرعی‌ای به این تابع پاس نکردیم چون که این موضوع برای هسته اهمیتی ندارد و ماژول خود از ان استفاده می‌کند.

سوال بسیار مهمی‌کـــه ممکن است مطـــرح شـــود این است که چگونه عدد اصلی ای به تابع register_chrdev بدهیم و مطمئن باشیم که قبلا توسط راه انداز دیگری اختیار نشده است؟

آسان‌ترین راه برای ایـــن کار رجـــوع بـــه فایل linux/Documentation/devices.txt از کد منبع هسته لینوکس و انتخاب یک عدد رزرو نشده است. مطمئنا این راه بدترین راه است چـــون هیـــچ اطمینانی در قابـــل استفاده بودن آن عـــدد اصـــلی نمی‌دهد. بهترین راه آنست کـــه شما از خود هسته بخواهید که به صورت دینامیک یک عدد اصلی در اختیار شما بگذارد. اگر ورودی اول تابع register_chrdev که همان unsigned int major است را صفر قرار دهید، هسته یک عدد اصلی را به عنوان مقدار بازگشتی تابع register_chrdev به شما باز می‌گرداند.

خروج (unregister) یک دستگاه از هسته

ما نمی‌توانیم اجازه دهیم که هر هنگام یک ماژول هسته rmmod شـــود. اگـــر فایل دستگاه توسط یک پروسه بـــاز شـده باشد و مـــا مـــاژول را از هسته خارج کنیم، آن پروسه بـه آن قسمت از حافظه که آدرس آن تـــابـــع از راه انـــداز است، دسترسی داشته و می‌تواند آن تابع را صدا نماید. اگر خیلی خوش شانس باشیم و کـــد جدیدی در آن آدرس load نشده باشد، فقط یک پیغام خطا دریافت خواهیم کرد. ولـــی اگـر خیلی خوش شانس نباشیم و ماژول جدیدی در آن ادرس load شده باشد، به این معنی است که در میـــان تابع جدیدی از هسته خـــواهیم پرید. نتیجه این کار به هیچ وجه قابل پیش بینی نیست ولی در اکثر موارد اتفاق‌های نسبتا ناگواری در سیستم به بار می‌آید.

معمولا برای اینکه شما اجازه انجام کاری را ندهید، یک پیغام خطا (یک عدد منفی) توسط آن تابع انجام دهنده کـــار بر خواهید گردانـــد. در مـــورد تابع cleanup_module ایـــن کار ممکن نیست، چـون ایـــن تـــابـــع void بـر می‌گرداند (چیزی بر نمی‌گرداند). راه متداول این کار وجود یک شمارنده در ماژول است کـه تعداد پروسس‌هایی که از ماژول استفاده می‌کنند را نگه می‌دارد. شما می‌توانید مقدار این شمارنده را در سومین فیلد proc/modules/ مشاهده کنیـــد. اگـر این مقدار غیر صفر باشد دستور rmmod با شکست مواجه خواهد شد.

توجه داشته باشیـــد کـــه شما نبایستی این شمـارنده را در cleanup_module چک نمایید بدلیل اینکه این ارزیابی توسط تابع سیستمیsys_delete_module که در linux/module.c تعریف شده است

برای شما انجام می‌شود. شما نمی‌توانید از این تابع به طور مستقیم استفاده نمایید اما توابعی در <linux/modules.h> تعریف شده است که اجازه افزایش، کاهش و مشاهده این شمارنده را می‌دهد. به عنوان مثال:

try_module_get(THIS_MODULE)

که یک واحد شمارنده را زیاد می‌کند و

try_module_get(THIS_MODULE)

که یک واحد شمارنده را کاهش می‌دهد.

نکته بسیار مهم این است که مقدار این شمارنده بایستی همیشه صحیح نگه داشته شود. اگر این مقدار خراب شود به هیچ عنوان نمی‌توان ماژول را unload کرد. در این مرحله هیچ کاری به جز reboot نمی‌توان کرد.

در قسمتهای پیشین با مفاهیم ابتدایی دستگاههای کاراکتری آشنا شدیم و نحوه ثبت این گونه دستگاه‌ها در هسته را متوجه شدیم . در این قسمت مطالبی را که در دو قسمت قبل فرا گرفتیم در ساده ترین مثال بررسی می‌نماییم.

کد این مثال را که حاوی توضیحات تقریبا کاملی است، می‌توانید از اینجا بدست آورید. پس از کامپایل ماژول از دستور زیر برای وارد کردن این ماژول در هسته استفاده کنید:

#insmod ./chardev.ko

اگر اکنون به انتهای فایل /var/log/messages بروید مشاهده می‌کنید که هسته به صورت دینامیک به ماژول شما یک عدد اصلی اختصاص داده است . با استفاده از راهنمایی‌ای که در این فایل شده، دستور mknod را برای ایجاد /dev/chardev به صورت ذکر شده اجرا کنید. اکنون راه انداز شما قابل استفاده است. می‌توانید آن را باز کنید، از آن بخوانید و در آن بنویسید.

برای خواندن از این دستگاه کاراکتری به صورت زیر عمل کنید:

#cat /dev/chardev

و یا برای نوشتن عبارتی مثلا "Hello" در آن به صورت زیر عمل کنید:

```
#echo "Hello" > /dev/chardev
```

همان گونه که ذکر شد توضیحات کامل این مثال در کد مثال به صورت comment آمده است. در زیر بـــه دو نـکتـــه از ایـــن مثال اشاره کرده و در نهایت بحث این مثال را با یک سوال به پایان می‌بریم.

۱) در تابع device_read همان طـــور کـــه مشـاهده می‌کنید از تابع put_user استفاده شده است. بـه طور کلی هر پروسس دارای یک بافر در فضای کاربر است و هـر ماژول نیز دارای یک بافر در فضای هسته است. عملی که این تابع و توابع مشابه آن انجام می‌دهند، اطلاعات را بـــه صـــورت کامـــلا محافظت شده بین این بافرها جابجا می‌کنند.

۲) همان طور که در تابع device_write می‌بینید، ایـن تابع عملی انجام نمی‌دهد. شما می‌توانید این تابع را به هر صورتی که می‌خواهید تغییر دهید. به عنوان مثال می‌توانید بـــا دانستـــن ساختار سخت افزاری مودم خود این تابع را پیاده سازی کرده و بدین ترتیب اطلاعات خود را در مودم خـود بنویسید. البته برای خواندن از مودمتان باید تابع device_read را با توجه به ساختار مودمتان تغییر دهید.

سوال: اکنون در کد مثال خط زیر را پیدا کنید:

```
static char msg[BUF_LEN];
```

و به صورت زیر تغییر دهید:

```
static char *msg;
```

یعنی در حقیقت برای اشاره گر بافر ماژول فضایی در نظر نگیرید. حـــال دوباره مـاژول را در هسته وارد کرده و عملیات cat را انجام دهید. مشـــاهده می‌کنید کـــه بـه پیغام segmentation fault برخورد خواهید کرد. در صـورتی که در چند قسمت قبل گفتیم که اگر ماژول دچار seg fault شود هسته دچـــار seg fault خواهد شد.

در قسمت های قبل با اصول و مبانی ماژول نویسی برای هسته لینوکس آشنا شدیم و ابتدایی ترین مفاهیم نوعی از دستگاهها موسوم به دستگاه های کاراکتری را بررسی نمودیم. در این قسمت و دو قسمت آینده

مطالبمان را با بررسی فایل سیستم proc/ و کاربرد آن در ماژول نویسی برای هسته لینوکس ادامه خواهیم داد.

در لینوکس مکانیزم ویژه‌ای برای هسته و ماژولهای هسته برای ارسال و دریافت اطلاعات از پروسس‌ها وجود دارد که در قالب فایل سیستم مجازی proc/ پیاده سازی شده است. این فایل سیستم برای سهولت دسترسی به اطلاعاتی در زمینه پروسس‌ها طراحی شده است. به عنوان مثال proc/modules/ لیستی از ماژول‌های وارد شده در هسته و proc/meminfo/ اماری از میزان مصرف حافظه را نشان می‌دهند. برای آشنایی بیشتر با این فایل سیستم این مقاله را مطالعه بفرمایید.

روشی که برای استفاده از فایل سیستم proc/ به کار می‌رود بسیار شبیه روشی است که در مورد راه اندازها به کار می‌رود، یک نمونه یا instance از struct ای که تمامی این اطلاعات را به همراه اشاره گرهایی به توابع مورد نظر ایجاد می‌گردد. سپس در تابع شروع ماژول که همان init_module است این ساختار داده در هسته ثبت شده و در هنگام اتمام ماژول که cleanup_module صدا زده می‌شود این ساختار داده از هسته خارج می‌گردد.

بحثمان را با یک مثال شروع می‌کنیم. کد این مثال و مثال بعدی را که حاوی comment کاملی هستند را می‌توانید از اینجا بدست آورید. با مثال اول (فایل procfs1.c) شروع می‌کنیم. این مثال از ۳ قسمت تشکیل شده است: در تابع init_module فایل proc/helloworld/ ایجاد می‌شود، هنگامی‌که از این فایل خوانده می‌شود تابع procfs_read صدا زده می‌شود که یک مقدار (و یک بافر) بر می‌گرداند. در نهایت در تابع cleanup_module این فایل حذف می‌گردد.

فایل proc/helloworld/ هنگامی‌که ماژول در هسته وارد می‌شود توسط تابع create_proc_entry ایجاد می‌گردد. مقدار بازگشتی این تابع یک 'struct proc_dir_entry *' است که برای پیکربندی فایل proc/helloworld/ (به عنوان مثال تعیین صاحب فایل) به کار می‌رود. مقدار بازگشتی NULL نشان می‌دهد که اجرای این تابع ناموفق بوده است.

هر هنگام که از فایل proc/helloworld/ خوانده می‌شود تابع procfs_read صدا زده می‌شود. دو پارامتر ورودی این تابع بسیار مهم هستند. buffer (اولین پارامتر) و offset (سومین پارامتر). محتوای بافر به برنامه‌ای که تقاضای خواندن داده است باز می‌گردد (به عنوان مثال دستور cat). پارامتر

offset نیز مکان فعلی در فـــایـــل را نـشـان می‌دهد. اگر مقدار بازگشتی این تابع NULL نباشد، این تابع دوباره صدا زده خواهد شد. بنـــابـراین مراقب این تابع باشید اگر مقدار بازگشتی این تابع هیجگاه صفر نشود صدا زدن این تابع به صورت بی پایان ادامه خواهد داشت.

مثالprocfs1.c را کامپایل کرده و ماژول تولیدی را در هسته وارد نمـــایـیـــد. بـا استفاده از دستور زیر از proc/helloworld/ بخوانید :

#cat /proc/helloworld

خواندن از و نوشتن در یک فایل proc/

مثال قبل که مثال ساده ای از خواندن از یک فایل proc/ بود را دیدیم. نکته‌ای کـــه می‌خواهیم در ایـــن قســـمت بـررسـی کنیم، نوشتن در یک فایل proc/ است. هنگام نوشتن در فایل proc/ مانند حالت خواندن یک تابع مانند procfs_write صدا زده می‌شود. اما تفاوتهایی بیـــن خـــواندن و نوشتن وجـــود دارد کـــه مـهم ترین آن انتقال یافتن اطلاعات از فضای کاربر به فضای هسته در حال نوشتن است که این کار توسط توابعی مانند copy_from_user یا get_user انجام می‌شود.

دلیل وجود توابعی مانند دو تابع بالا این است که حافظه در لینـــوکس (در معماری پردازنده اینتل، ممکن است در پردازنده های دیگر متفاوت باشد) بـــه segment هایی تقسیم شده است. ایـــن بدان معنا است که یک اشاره گر به تنهایی به ادرس یکتایی در حافظه اشاره نمی‌کند , بلکه به موقعیتی در segment اشاره می‌کند و شما نیاز داید که segment حافظه را بدانید تا بتوانید از ان استفاده کنید.

فقط یک segment برای هسته وجود دارد و برای هر پروسس نیز یک segment اختصاص می‌یابد. تنها segment ای که یک پروسس می‌تواند به ان دسترسی داشته باشد segment خود پروسس است. بنابراین هنگامی‌که شما برنامه‌ای توسعه می‌دهید یا اجرا می‌کنید، واقعا لازم نیست که نگران مدیریت segmentهای حافظه باشید. امـــا هنـگـامی که یک ماژول هسته می‌نویسید، معمـــولا می‌خواهید کـــه بـــه فضـــای حـافـظه segment هسته کـــه توسط سیستم راه اندازی می‌شود دسترسی داشته باشید. با این حال هنگامی‌که نیـــاز است محتـوای یک بافر حافظه بین یک پروسس و هسته رد و بدل شود هسته یـــک اشاره گر به بافر حافظه segment پروسس دریافت می‌دارد. ماکروهای put_user و get_user اجازه دسترسی به این حافظه را می‌دهند. این توابع فقط یک کاراکتر تحویل می‌دهـند. شما می‌توانید با

استفاده از توابع copy_to_user و copy_from_user کاراکترهای متعددی را دریافت دارید و یا به هسته تحویل دهید.

چون که بافر (در توابع خواندن و نوشتن) در فضای هسته است، برای نوشتن شما نیاز دارید که اطلاعاتتان را import کنید اما در تابع خواندن اطلاعات هم اکنون در فضای هسته است.

در قسمت قبل با چگونگی خواندن از و نوشتن در فایل سیستم مجازی proc/ آشنا شدیم. در این قسمت مدیریت این فایل سیستم را با استفاده از inodeها و seq_fileها از نظر می‌گذرانیم.

در هسته لینوکس مکانیزم استانداردی برای ثبت فایل سیستم وجود دارد. از آن جاییکه هر فایل سیستم بایستی توابع مخصوص به خود برای به کار بردن inode و عملیات فایل داشته باشد ساختار ویژه‌ای به نام struct inode_operations برای نگهداری اشاره‌گر به تمام آن توابع وجود دارد که در خود اشاره‌گری به ساختار struct file_operations دارد.

در فایل سیستمproc/ هنگامی‌که یک فایل جدید ثبت می‌شود ما اجازه خواهیم داشت که ساختار inode_operations ای که برای دسترسی به این فایل به کار می‌رود را تعیین کنیم.

به طور خلاصه در این مکانیزم یک struct inode_operations وجود دارد که در خود اشاره‌گری به یک struct file_operations دارد که این ساختار نیز اشاره‌گرهایی به توابعی مانند procfs_read و procfs_write دارد که وظیفه خواندن از و نوشتن در proc/ را بر عهده دارند.

نکته قابل توجه دیگر وجود توابعی مانند module_permission است. این تابع هنگامی که یک پروسه عملی در proc/ انجام می‌دهد صدا زده شده و تعیین می‌کند که آیا پروسه مذکور حق انجام عمل مورد نظر را دارد یا نه؟

در حال حاضر این قضاوت بر اساس عمل مذکور و uid کاربر حاضر (که در متغییر current که اشاره‌گری به ساختاری در مورد اطلاعات پروسه حاضر است , وجود دارد.) صورت می‌پذیرد. اما باید توجه داشت که این قضاوت می‌تواند به دلخواه ما که ماژول هسته را می‌نویسیم صورت گیرد.

نکته دیگری که در اینجا بایستی ذکر گردد نقش توابع read و write برای هسته برعکس نقشی است که در مورد یک پروسه وجود دارد. تابع read به عنوان تابع خروجی و تابع write به عنوان تابع ورودی در هسته استفاده می‌شود. دلیل این موضوع آن است که هنگامی‌که یک پروسه از هسته چیزی می‌خواند هسته بایستی آن را به عنوان خروجی بیرون دهد و هنگامیکه یک پروسه در هسته می‌نویسد هسته بایستی آن را به عنوان ورودی دریافت دارد.

برای فهم عمیق‌تر و چگونگی کاربرد ساختارهای inode_operations و module_permission به مثال procfs3.c که توضیحات کاملی در کد برای آن وجود دارد مراجعه کنید. کد این مثال را از اینجا [۱] می‌توانید بدست آورید. ماژول را کامپایل نموده و در هسته وارد نمایید و با دستورات cat و echo عملکرد آن را با بررسی فایل var/log/messages/ مورد بررسی قرار دهید.

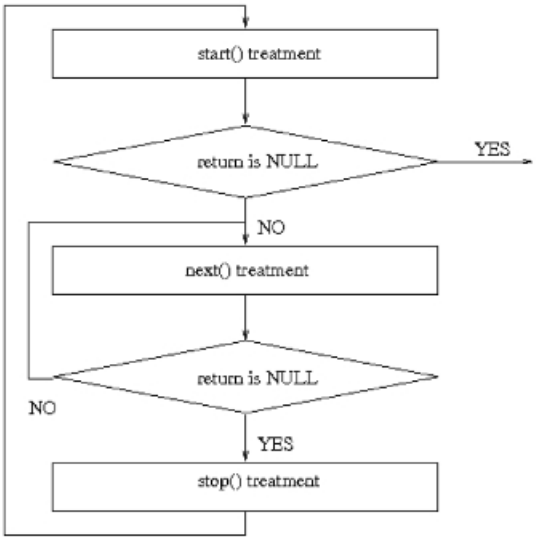
مدیریت فایل سیستم مجازی proc/ با استفاده از seq_file

همان گونه که دیدیم کار با یک فایل proc/ ممکن است کاملا پیچیده باشد. برای رفع این پیچیدگی API ای به نام seq_file در هسته لینوکس وجود دارد که ما را در قالب بندی یک فایل proc/ برای خروجی یاری می‌نماید که بر اساس ترتیب (sequence) می‌باشد. این ترتیب از سه تابع ()start و ()next و ()stop تشکیل شده است. seq_file هنگامی که یک کاربر از proc/ می‌خواند راه اندازی می‌شود.

یک ترتیب با صدا کردن تابع ()start آغاز می‌شود. اگر مقدار بازگشتی این تابع NULL نباشد تابع ()next صدا زده می‌شود. نقش این تابع به عنوان یک iterator می‌باشد و هدف از صداکردن پشت سرهم این تابع جابجا شدن به اطلاعات بعدی فایل proc/ می‌باشد.

هر هنگام که این تابع صدا زده می‌شود تابع دیگری به نام ()show نیز صدا زده می‌شود که اطلاعات دریافتی را در بافری که توسط پروسه فراهم شده می‌نویسد. تابع ()next تا زمانیکه NULL را به عنوان خروجی برگرداند صدا زده می‌شود. در این هنگام ترتیب با صدا زده شدن تابع ()stop خاتمه می‌پذیرد.

توجه نمایید که هنگامی‌که یک ترتیب خاتمه می‌پذیرد ترتیب جدیدی شروع می‌شود. این بدان معناست که در انتهای تابع stop() تابع start() دوباره صدا زده می‌شود. این چرخه، هنگامی‌که start() مقدار بازگشتی NULL باز می‌گرداند خاتمه می‌پذیرد. نحوه عملکرد seq_file در شکل ۱ نشان داده شده است.



شکل ۱ – نحوه عملکرد seq_file

seq_fileتوابع ساده‌ای مانند seq_read و seq_lseek و... را برای file_operations فراهم می‌نماید. اما نمی‌تواند در /proc بنویسد. برای انجام این کار بایستی مانند مثال قبلی عمل کرد.

به عنوان مثالی از seq_file به مثال procfs4.c مراجعه کنید. برای کسب اطلاعات بیشتر در این زمینه به این صفحات وب در سایت‌های LWN و KernelNewbies مراجعه کنید:

http://lwn.net/Articles/22355

http://www.kernelnewbies.org/documents/seq_file_howto.txt

همچنین می‌توانید پیاده سازی seq_file در هسته لینوکس را در مسیر linux source/fs/seq_file.c مورد مطالعه قرار دهید. در قسمت آینده مرور مختصری بر proc/ به عنوان ورودی و sysfs خواهیم داشت.

در قسمت قبل بررسی مدل جدید راه اندازها در هسته لینوکس که به Unified Device Model موسوم است را با معرفی چند ساختار داده اصلی مانند kobject , ktype , kset و subsystem شروع نمودیم. در این قسمت به نحوه استفاده از این ساختار های داده در ماژول نویسی هسته لینوکس خواهیم پرداخت. به دلیل حجم بالای مطالب نکات اصلی این توابع مورد بحث وبررسی قرار خواهند گرفت و جزییات بیشتر به خواننده واگذار می شود.

مدیریت kobject ها

با توجه به آنچه در قسمت قبل در مورد kobject ها فرا گرفتیم به بررسی توابعی که مدیریت kobject ها را تسهیل می کنند می پردازیم. ساختار داده kobject معمولا مستقیما به کار نمی رود. بلکه در درون ساختار داده دیگری (به عنوان مثال cdev که در قسمت قبل آن را بررسی نمودیم) جاسازی می شود.

اولین قدم در استفاده از kobject ها تعریف و مقداردهی اولیه آنهاست. این کار با تابع kobject_init انجام می گیرد که در <linux/kobject.h> تعریف شده است.

```
void kobject_init(struct kobject *kobj);
```

این تابع kobject ورودی را گرفته و فیلد های آن را مقداردهی می کند. قبل از صدا کردن این تابع بایستی فضای حافظه kobject صفر شود. این کار را می توان با تابع memset انجام داد.

```
memset(kobj, 0, sizeof (*kobj));
```

بعد از صفر کردن kobject می توان parent و kset آن را مقدار دهی کرد. به عنوان مثال:

```
kobj = kmalloc(sizeof (*kobj), GFP_KERNEL);
if (!kobj)
return -ENOMEM;
memset(kobj, 0, sizeof (*kobj));
kobj->kset = kset;
kobj->parent = parent_kobj;
kobject_init(kobj);
```

بـــعداز مـــقدار دهی اولیه بایستی نامی برای kobject در نظر گرفته شود.این کار بااستفاده ازتابع kobject_set_name()انجام می گیرد:

int kobject_set_name(struct kobject * kobj, const char * fmt, ...);

syntax این تابع مانند printk است کـــه مـی‌توان آن را بـــا fmt بـه صـــورت دلخواه نام گذاری کرد. با استفاده از این تابع k_name در ساختار داده kobject مقداردهی می گردد.

بعد از ایجاد یک kobject و مقدار دهی آن شما نیاز دارید که فیـــلدهـــای kset و ktype آن را تنظـــیم نمایید. اگر kset نوع kobject را مشخص نکرده باشد تنظیم ktype اجباری می شود در غیر این صورت اختیاری است.

Reference Counts

یکی از امکانات اولیه ای که توسط kobject هـــا فراهم شده است مکانیزم یکتایی برای شمارش ارجاعات است. بعد از مقداردهی اولیه مقدار شمارنده ارجاعات kobject به مـــقدار ۱ تنظیم می شود. تا زمـــانی که این شمارنده صفر نشده است object به حیات خود در حافظه ادامه خواهد داد.

هر کدی که یک ارجاع به object دارد ابتدا یک واحد این شمارنده بالا برده می شود و هنگامی که این کد به پایان رسید یک واحد این شمارنده کاهش می یابد. هنگامی که مقدار این شمارنده به صـــفر رســـید , objectاز بین رفته و حافظه تخصیص یافته به آن ازاد می شود.

افزایش این شمارنده توسط تابع kobject_get() انجام می گیرد. این تابع اشاره گری به kobject و در صورت خطا NULL بر می گرداند.

struct kobject * kobject_get(struct kobject *kobj);

متقابلا کاهش این شمارنده توسط تابع kobject_put() صورت می پذیرد.

void kobject_put(struct kobject *kobj);

اگر شمارنده به صفر برسد تابع release ای که در ktype به آن اشاره شده است صدا زده خواهد شد.

kref

شمارنده kobject توســـط ساختار داده kref که در <linux/kref.h> تعریف شده است و در lib/kref.cپیاده سازی شده است.

```
struct kref {
atomic_t refcount;
};
```

تنها فیلد این ساختار داده متغیر refcount است که به صورت atomic تعریف شده و مـــقدار شمارنده را در خود نگه می دارد. قبل از استفاده از kref بایستی آن را با استفاده از kref_init() مقدار دهی کنید.

```
void kref_init(struct kref *kref)
{
atomic_set(&kref->refcount, 1);
}
```

برای گرفتن یک ارجاع (بالا بردن شمارنده) از تابع kref_get() استفاده می شود.

```
void kref_get(struct kref *kref)
{
WARN_ON(!atomic_read(&kref->refcount));
atomic_inc(&kref->refcount);
}
```

برای رها کردن یک ارجاع (پایین اوردن شمارنده) از تابع kref_put() استفاده می شود. هنگامی که مقدار شمارنده به صفر رسید تابعی که توسط اشاره گر به تابع release فراهم شده است صدا زده می شود.

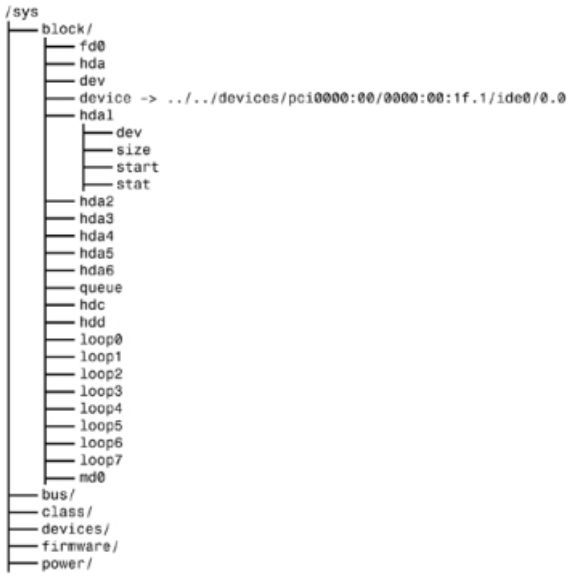
```
void kref_put(struct kref *kref, void (*release) (struct kref *kref))
{
WARN_ON(release == NULL);
WARN_ON(release == (void (*)(struct kref *))kfree);
if (atomic_dec_and_test(&kref->refcount))
release(kref);
}
```

فایل سیستمsysfs

فایل سیستم sysfs یک فایل سیستم مجازی در حافظه است که نمایش درختی از kobject ها را برای ما فراهم می کند. این فایل سیستم توپولوژی دستگاه ها را در یک فایل سیستم به ما نمایش می دهد. با استفاده از attribute ها kobject ها می توانند با استفاده از فایل ها این اجازه را بدهند که بتوان بعضی از متغیرهای هسته را خواند و یا در انها نوشت.

اگر چه هدف ابتدایی این مدل جدید ایجاد یک فرایند کنترل انرژی هوشمند در یک سیستم کامپیوتری بود توسعه دهندگان هسته لینوکس به این نتیجه رسیدند که برای امکانات رفع خطای ساده تر آن را به صورت یک فایل در معرض دید بگذارند و sysfs ایجاد شد و به مرور در حال جایگزینی فایل‌های دستگاه‌ها در /proc می‌شود. امروزه تمام سیستم هایی که از کرنل ۲.۶ استفاده می کنند فایل سیستم sysfs را به صورت mount شده در سیستم خود دارند.

نمایشی از این فایل سیستم که در /sys , mount شده است را ملاحظه می فرمایید.



شکل ۱ -نمایی از فایل سیستم /sys

ریشه این فایل سیستم به طور استاندارد حاوی ۷ دایرکتوری است(در هسته های مختلف ممکن است متفاوت باشد) : block , bus , class , devices , firmware , module , power

دایرکتوری block برای هر دستگاه (block device) یک دایرکتوری جداگانه دارد. دایرکتوری bus نمایشی از bus سیستم را در خود نگه می دارد. دایرکتوری class نمایشی از دستگاه ها را که توسط توابع سطح بالایی سازمان یافته اند را در بر دارد. دایرکتوری devices نمایشی از توپولوژی دستگاه های موجود در سیستم را در خود نگه می دارد. این دایرکتوری مستقیما به ساختار درختی که از ساختارهای داده ذکر شده در هسته تشکیل شده انگاشته می شود. دایرکتوری firmware نیز درختی از اجزای سطح پایین سیستم مانند ACPi , EDD , EFi و ... را نگه می دارد. دایرکتوری های module و power نیز به ترتیب ساختارهایی از ماژول های کرنل و مدیریت انرژی در کرنل را نگه می دارند.

مهمترین دایرکتوری در این فایل سیستم devices است که مدلی از دستگاه های موجود در سیستم را در خود نگه می دارد. تعداد کثیری از فایل های موجود در دایرکتوری های دیگر در حقیقت اشاره گرهایی به فایل های این دایرکتوری هستند. برای اشنایی بیشتر با این فایل سیستم بهتر است ترمینال سیستم گنو/لینوکس خود را باز کرده و چرخی در این فایل سیستم بزنید تا با اجزای آن بیشتر اشنا شوید.

توابع کار با فایل سیستم sysfs

در این قسمت به دلیل تعداد زیاد توابع , توابع اصلی به صورت تیتروار مورد بررسی قرار می گیرند. برای جزییات بیشتر به کتاب ها یا مقالات اشنایی با هسته لینوکس (مانند منابع انتهای مقاله) مراجعه کنید.

kobject هایی که مقدار اولیه داده شده اند به طور اتوماتیک به این فایل سیستم اضافه نمی شوند. برای این کار از تابع kobject_add استفاده می شود. مکان این فایل در این فایل سیستم از موقعیت kobject در ساختار درختی خود تعیین می شود. با استفاده از تابع ()kobject_register می توان دو عمل kobject_init و kobject_add را به یکباره انجام داد. برای حذف نمایشkobject درsysfs ازتابعkobject_dell استفاده می شود. تابع kobject_unregister ترکیبی از دو تابع kobject_dell و kobject_put می باشد.

kobject ها به دایرکتوری ها در sysfs نگاشته می شوند. برای نگاشتن فایل ها به sysfs از فیلد attribute موجود در kobject ها و ktype ها استفاده می شود. ساختار داده attribute در <linux/sysfs.h> تعریف شده و به صورت زیر می باشد:

```
struct attribute{
char *name; /* attribute's name */
struct module *owner; /* owning module, if any */
mode_t mode; /* permissions */
};
```

فیلدهای owner , name و mode به ترتیب بیانگر نام , صاحب و سصح دسترسی فایل موجود در sysfs می باشد.

همان طور که در قسمت قبل در بخش ktype گفتیم رفتار پیش فرض دسته ای از kobject ها در ساختار ktype در فیلد default_attrs قرار می گیرد. ولی با استفاده از فیلد sysfs_ops در همین ساختار داده می توان رفتار های kobject را مشخص کرد.

```
struct sysfs_ops{
/* method invoked on read of a sysfs file */
ssize_t (*show) (struct kobject *kobj,
struct attribute *attr,
char *buffer);
/* method invoked on write of a sysfs file */
ssize_t (*store) (struct kobject *kobj,
struct attribute *attr,
const char *buffer,
size_t size);
};
```

به طور خلاصه متد ()show برای خواندن به کار می رود. این تابع مقدار attr را در بافری به نام buffer کپی می کند. متد ()store نیز برای نوشتن به کار می رود. این تابع به اندازه size از buffer می خواند و در attr می نویسد. روش ذکر شده برای دسته ای از kobject ها کارایی دارد. برای تنظیم attribute برای یک kobject از توابع زیر استفاده می شود:

```
int sysfs_create_file(struct kobject *kobj, const struct attribute *attr);
int sysfs_create_link(struct kobject *kobj, struct kobject *target, char *name);
void sysfs_remove_file(struct kobject *kobj, const struct attribute *attr);
void sysfs_remove_link(struct kobject *kobj, char *name);
```

نتیجه گیری: در این دو قسمت اخیر با مدل جدیدی از دستگاه ها موسوم به sysfs and kobjects اشنا شدیم. ساختار های داده ای مانند kref , ktype , attribute , subsystem , kset و ... را مورد

بررسی قرار دادیم و با روش های متفاوتی نحوه استفاده و مدیریت kobject ها و نحوه نمایش انها در sysfs را بررسی کردیم. مطالبی که در این دو قسمت بررسی شدند از جدیدترین موضوعاتی هستند که در هسته لینوکس مورد پیاده سازی قرار گرفته اندو به نظر می رسد که اشنایی هر توسعه دهنده هسته با این مبانی کاملا ضروری می نماید. مطالب این مدل جدید را در همین جا به پایان می بریم و یادگیری جزییات بیشتر را به خواننده واگذار می کنیم.

در این قسمت نحوه پیاده تابع ioctl برای یک راه‌انداز را بررسی خواهیم کرد و با نوشتن یک برنامه نحوه‌استفاده‌از پیاده سازیمان را فرا خواهیم گرفت.

IOCTL

فایل های دستگاه (که معمولا در dev/ قرار می گیرند) برای نمایش و استفاده‌از دستگاه‌های فیزیکی ایجاد شده‌اند. بیشتر دستگاه‌های فیزیکی همان طور که برای خروجی به کار می روند به عنوان ورودی نیز استفاده می شوند. بنابراین بایستی در هسته مکانیزمی وجود داشته باشد که بتواند خروجی را دریافت کند تا از پروسه ها به دستگاه ها بفرستد. این رویه با باز کردن فایل دستگاه برای خروجی و نوشتن در ان انجام می شود , دقیقا مانند نوشتن در یک فایل معمولی. در مثال پیش روی که می توانید از این آدرس دریافت دارید این تابع با نام device_write در فایل chardev.c پیاده سازی شده‌است.

پیاده سازی توابعی که تاکنون دیده‌ایم در یک راه‌انداز معمولا کفایت نمی کند. فرض کنید که شما یک پورت سریال دارید که به یک مودم متصل است (حتی اگر شما یه مودم داخلی داشته باشید از دید پردازنده‌این یک پورت سریال است که به یک مودم متصل شده‌است). رویه متداول این است که شما از فایل دستگاه برای نوشتن و خواندن از مودم استفاده می کنید. سوالی که ممکن است مطرح شود این است که چگونه می توان با خود پورت سریال صحبت کرد؟ به عنوان مثال سرعت دریافت یا انتقال داده را تعیین کرد.

جواب این سوال در یونیکس استفاده‌از تابع ویژه‌ای به نام ioctl (کوتاه شده Input Output Control) است. هر دستگاه می تواند دستور ioctl مخصوص به خود داشته باشد , که هم می تواند بخواند (اطلاعات را از یک پروسه به هسته بفرستد) و هم بنویسد (اطلاعات را به پروسه برگرداند).

صورت کلی تابع ioctl به صورت زیر است :

```
#include <sys/ioctl.h>

int ioctl(int fd, int request, … );
```

آرگومان اول یک file descriptor است که همان فایل دستگاه می باشد که بایستی قبلا باز شده باشد و fd ان به عنوان آرگومان اول در ioctl استفاده می شود. آرگومـان دوم (request) اصصلاحـا عدد ioctl نـــامیده شده کـــه عدد اصلی دستگاه , نوع ioctl , دستور و نوع پارامتر (آرگومان سوم) را در خود کد می کند. برای تولید این عدد بسته به پارامتر از ماکـــروهایی چون IO_ , IOR_ , IOW_ , IOWR_ و ... استفاده می شود.

این ماکروها بایستی در یک فایـــل هدر قرار گرفته (در این مثال chardev.h) ودر هر دو فایل ماژول هسته (در این مثال chardev.c) و کد برنامـه (در این مثال ioctl.c) ضمیـمه شود.

اگر شما می‌خواهید از ioctl در مـــاژول خود استـــفاده کنید , بهترین راه‌انست که مستقیما ioctl را در ماژول خود پیاده سازی کنید (در مثال به طور کاملا واضحی توضیح داده شده‌است) واگر از ioctl دیگران استفاده کنید ممکن است دچار مشکل شوید .

برای کسب اطلاعات بیشتربه فایل Linux Source Code/Documentation/ioctl-number.txt مراجعه کنید.

درباره مثال

ابتدا مثال را از این آدرس http://www.irantux.org/images/down/chardev.tar.bz2 دریافـت کـــرده و از حالت فشـرده خـــارج سازید. سپس از دسـتور make بـرای ایجاد ماژول استفاده کنید . فایلی به نام chardev.ko ساخته خواهد شد . با استفاده‌از دستور زیر ماژول را در هسته وارد نمایید.

```
#insmod ./chardev.ko
```

با وارد کردن ماژول در هسته دستور العمل لازم در انـــــتهای فایل var/log/messages/ که فایل log هسته‌است نوشته خواهد شد. بر طبق دستورعمل کرده و فایل دستگاه char_dev را بسازید. اکنون ماژول

شما در هسته وارد شده و شما می توانید با استفاده‌از فایل char_dev در ان بنویسید یا از ان بخوانید. سپس با استفاده‌از دستور زیر فایل اجرایی ioctl را بسازید.

```
$gcc ioctl.c -o ioctl
```

و با اجرای ان می توانید دستورات ioctl پیاده سازی شده در chardev را امتحان کنید و خروجی ماژول خود را مشاهده نمایید.

در این قسمت به بررسی دو موضوع نسبتا جدا از هم در ماژول نویسی هسته لینوکس خواهیم پرداخت. در بخش اول با عنوان توابع سیستمی با نحوه نوشتن و تغییر توابع سیستمی هسته لینوکس اشنا خواهیم شد و در بخش دوم با عنوان متوقف کردن پروسه ها نحوه مدیریت پروسه های در حال اجرا و پروسه های در حال انتظار را فرا خواهیم گرفت.

توابع سیستمی (System Calls)

انچه تاکنون در ماژول نویسی هسته لینوکس انجام دادیم استفاده از مکانیزم های دقیق هسته مانند ثبت فایل در proc/ و راه انداز فایل ها و ... بود. همه چیز مرتب است اگر شما بخواهید کاری انجام دهید که توسعه دهندگان هسته ان را پیش بینی کرده و مکانیزمی در هسته برای کار شما در هسته ایجاد کرده باشند مانند نوشتن راه انداز دستگاه. اما اگر شما بخواهید کار غیر معمولی انجام دهید که توسعه دهندگان هسته راه کاری برای ان ایجاد نکرده اند چطور ؟ در اغلب این موارد مسئولیت همه چیز با خودتان است.

اینجا جایی است که برنامه سازی هسته خطرناک می شود. با نوشتن ماژول هسته مثال شما یکی از این کارهای خطرناک را انجام خواهید داد. کد این مثال را از اینجا [۱]می توانید بدست اورید. در این مثال شما تابع سیستمی open را تعویض خواهید نمود. این بدان معنی است که هیچ کس در سیستم دیگر نمی تواند فایلی را باز کند (هیچ کس نمی تواند برنامه ای اجرا کند و حتی کامپیوتر را خاموش نماید). تنها راه راه اندازی سخت افزاری سیستم می باشد. خوشبختانه هیچ فایلی از بین نخواهد رفت. برای حصول اطمینان از خراب نشدن فایل ها قبل از هر insmod و هر rmmod یک بار دستور sync را اجرا کنید.

فایل های proc/ و فایل های دستگاه ها در dev/ را فراموش کنید. پروسه اصلی در مکانیزم ارتباط با هسته (که توسط تمام پروسه ها استفاده می شود) استفاده از توابع سیستمی می باشد. هنگامی که یک

پروسه سرویسی را از هسته درخواست می نماید (مانند باز کردن یک فایل , ایجاد یک پروسه جدید یا درخواست حافظه بیشتر) از این مکانیزم استفاده می شود. اگر شما می خواهید رفتار هسته را به دلخواه خود تغییر دهید اینجا جایی است که می توانید تغییرات خود را اعمال نمایید. برای دانستن توابع سیستمی که در برنامه ها به کار می روند می توانید از دستور strace به صورت زیر استفاده نمایید.

```
$strace programname
```

به طور کلی یک پروسه نمی تواند به هسته دسترسی داشته باشد. بدان معنی که نمی تواند به حافظه هسته دسترسی داشته باشد و نمی تواند توابع هسته را صدا نماید. (این سیستم ها protected mode نامیده می شوند) . توابع سیستمی استثنای این قاعده کلی هستند. اتفاقی که می افتد این است که پروسه رجیسترهای پردازنده را با مقادیر مناسب پر کرده و دستور خاصی را صدا کرده که باعث پرش به محل از پیش تعیین شده ای از هسته می شود. (که البته ان محل از حافظه توسط پروسه ها قابل خواندن است ولی قابل نوشتن نیست). در پردازنده های اینتل این رویه با استفاده از وقفه (interrupt) شماره x80۰ انجام می گیرد.

CPU به عنوان سخت افزار کامپیوتر می داند هنگامی که شما به این نقطه پرش می کنید نمی خواهید که در حالت محدود شده ادامه کار دهید و به عنوان کدی از هسته سیستم عامل به شما اجازه هر کاری که بخواهید را می دهد. مکانی در هسته که پروسه می تواند به ان پرش نماید system_call نامیده می شود. رویه در این موقعیت از هسته به این صورت است که شماره تابع سیستمی که بیانگر سرویسی است که پروسه از هسته می خواهد چک شده و در جدول توابع سیستمی (sys_call_table) تابع موردنظر برای صدا زدن جستجو خواهد شد و سپس تابع موردنظر صدا زده می شود و پس از بازگشت تابع و انجام چندین چک از طرف سیستم به پروسه بازگشت خواهد شد (یا اگر زمان پروسه تمام شده باشد به پروسه دیگری ارجاع خواهد شد)

برای مشاهده کد اسمبلی این رویه به آدرس

Linux Source Code/arch/<$architecture$>/kernel/entry.S از کد هسته لینوکس بعد از خط ENTRY(system_call) مراجعه کنید.

بنابراین اگر بخواهیم عملکرد تابع سیستمی ای را تغییر دهیم کافیست تابع خودمان را بنویسیم و اشاره گر به تابع اصلی در sys_call_table را با اشاره گر به تابع خودمان جایگزین نماییم تا به تابع ما اشاره کند. فقط بایستی یادمان باشد که در تابع cleanup_module تمام تغییراتی که در این جدول اعمال کرده ایم را به حالت اولیه بازگردانیم تا سیستم در حالت ناپایدار باقی نماند.

کد مثالی که از http://www.irantux.org/images/down/syscall.tar.bz2 می توانید دریافت کنید کد یک ماژول کرنل می باشد. ما می خواهیم جاسوسی یکی از کاربران را انجام دهیم به صورتی که هرگاه کاربر موردنظر ما فایلی را باز کرد یک پیغام به وسیله ()printk در فایل log هسته چاپ کنیم. برای این کار تابع سیستمی ()open را با تابع خودمان به نام our_sys_open جایگزین می نماییم. این تابع uid (user's id) پروسه جاری را چک کرده و اگر برابر با uid کاربر موردنظر ما بود پیغام موردنظر ما را چاپ می کند و در نهایت فایل موردنظر کاربر را به وسیله تابع اصلی ()open باز می نماید.

در تابع init_module اشاره گر موردنظر در جدول sys_call_table جایگزین شده و مقدار اصلی این اشاره گر در یک متغیر ذخیره می شود. تابع cleanup_module از این متغیر استفاده کرده و همه چی را به حالت عادی باز می گرداند. /P

این روش , روشی خطرناک است چون ممکن است که دو ماژول هسته قصد تعویض یک تابع سیستمی را داشته باشند. فرض کنید که دو ماژول کرنل A و B داشته باشیم. تابع ()open جایگزین شونده در ماژول A_open , A و در ماژول B_open , B است. ماژول A در هسته وارد می شود بنابراین تابع سیستمی ()open با A_open جایگزین می شود. حال ماژول B در هسته وارد می شود که باعث جایگزینی A_open با B_open می شود.در مورد خروج ماژول ها از هسته اگر ماژولB و سپس ماژول A از هسته خارج شوند همه چیز درست خواهد بود اما اگر برعکس خارج شوند چطور ؟ خودتان می توانید حدس بزنید که چه اتفاقی خواهد افتاد . اگر A ابتدا از هسته خارج شود مقدار جاری اشاره گر به تابع سیستمی که به B_open اشاره می کند را با open اصلی تعویض خواهد کرد و اگر در این لحظه B از هسته خارج شود مقدار جاری اشاره گر که به open اصلی اشاره می کند را با مقذار ذخیره کرده خود که همان A_open است جایگزین می نماید و چون A_open در هسته وجود ندارد سیستم دچار crash خواهد شد. و موارد زیاد دیگری که می تواند سیستم را دچار crash کند.

مسائلی از این قبیل اجازه کار با توابع سیستمی را برای کارهای تولیدی که به استفاده عموم می رسد را غیر ممکن می سازد. برای جلوگیری از انجام کارهای خطرناک توسط افراد , دیگر متغیر sys_call_table که به جدول توابع سیستمی اشاره می کند در فضای هسته قرار داده نشده است. بنابراین اگر شما می خواهید که ماژول مثال مورد بحث را بتوانید در هسته وارد کنید بایستی هسته جاری خود را patch کرده و دوباره کامپایل نمایید تا متغیر sys_call_table را در حافظه هسته داشته باشید. این patch را می توانید در دایرکتوری مثال پیدا کنید. (شاید نیاز داشته باشید که برای نسخه هسته خود کمی کد نیز به صورت دستی اعمال کنید)

متوقف کردن پروسه ها (Blocking Processes)

هنگامی که کسی از شما چیزی می خواهد که شما نی توانید انجام دهید چه می کنید ؟ اگر شما یک انسان باشید و کسی که از شما درخواست کرده نیز یک انسان باشد به او می گویید : "الان نه , سرم شلوغه" . اما اگر شما یک ماژول هسته باشید که یک پروسه از شما درخواستی کرده باشد امکان دیگری نیز خواهید داشت . شما می توانید پروسه را به حالت خواب ببرید تا زمانی که بتوانید به ان سرویس دهید .

ماژول هسته کد مثال که از http://www.irantux.org/images/down/blockproc.tar.bz2 می‌توانید دریافت کنید مثالی از این کار است. در این مثال فایل proc/sleep/ توسط فقط یک پروسه در هر لحظه می تواند باز شود. اگر فایل در حال حاضر باز باشد ماژول هسته تابع wait_event_interruptible را صدا زده که باعث تغییر وضعیت کار (task) (task ساختار داده ای در هسته است که اطلاعاتی در مورد پروسه و تابع سیستمی ای که پروسه در ان است نگه می دارد) به حالت TASK_INTERRUPTIBLE می شود بدین معنا که کار تا زمانی که کسی ان پروسه را بیدار نکند ادامه نمی یابد و ان را به صفی به نام WaitQ که پروسه های منتظر دسترسی به فایل proc/sleep/ هستند اضافه می کند و به scheduler سیستم عامل دستور context switch را می دهد که CPU را به دست پروسه دیگری بدهد.

هنگامی که یک پروسه کار خود را با فایل به اتمام رساند ان را می بندد که باعث به صدا درامدن تابع module_close در ماژول هسته می شود. این تابع تمام پروسه های موجود در صف WaitQ را بیدار می کند و پروسه ای که بتواند فایل را تصاحب کند به کار خود ادامه می دهد. این پروسه کار خود را از

تابع module_interruptible_sleep_on اغاز کرده و متغیری عمومی را تنظیم می کند تا به پروسه های دیگر نشان دهد که فایل هم اکنون باز شده است. پروسه های دیگر به دیدن وضعیت این متغیر به حالت خوابیده باز می گردند.

در این مثال ما با استفاده از دستور tail -f فایل مورد نظرمان را در پس زمینه باز نگه می داریم و با استفاده از فایل اجرایی cat_nonblock که با استفاده از دستور زیر ایجاد می شود فایل را دوباره باز می کنیم.

$gcc cat_nonblock.c -o cat_non_block

اگر با استفاده از دستور kill %1 اولین پروسه پس زمینه را بکشیم پروسه دوم که به حالت خوابیده رفته بود به کار خود ادامه داده و نهایتا پایان می یابد.

درباره مثال قسمت دوم : کد مثال را از اینجا [۲] می‌توانید دریافت کنید. ماژول مثال را که در فایل sleep.c پیاده سازی شده است را با استفاده از دستور make کامپایل کنید و کد cat_nonblock.c را همانطور که در بالا اشاره شد با دستور gcc کامپایل کنید.

debian:~/lkmpg13/blockproc# insmod sleep.ko

debian:~/lkmpg13/blockproc# cat_noblock /proc/sleep

Last input:

debian:~/lkmpg13/blockproc# tail –f /proc/sleep&

Last input:

Last input:

Last input:

Last input:

Last input:

Last input:

```
Last input:

tail: /proc/sleep: file truncated

 6540 [1]

debian:~/lkmpg13/blockproc# cat_noblock /proc/sleep

Open would block

debian:~/lkmpg13/blockproc# kill %1

Terminated tail −f /proc/sleep +[1]

debian:~/lkmpg13/blockproc# cat_noblock /proc/sleep

Last input:

debian:~/lkmpg13/blockproc#
```

در این قسمت با بررسی چند موضوع باقیمانده مجموعه مقالات اشنایی با ماجول نویسی هسته لینوکس را به پایان می بریم. در ابتدا با دو مثال دو راه ساده ارتباط هسته با دنیای بیرون را شرح خواهیم داد. سپس با نحوه زمان بندی کارها در هسته اشنا خواهیم شد و در ادامه روتین های رسیدگی کننده به وقفه ها (Interrupt Handlers) را مورد بررسی قرار خواهیم داد. در انتها نیز با ذکر چند نکته و جمع بندی این مجموعه مقالات را به پایان خواهیم برد.

تعویض printk

در این بخش نحوه فرستادن پیغام ها از طرف ماجول هسته به tty ها را بررسی خواهیم کرد. این کار با استفاده از متغیر current که اشاره گری به پروسه در حال اجرا است صورت می پذیرد. با استفاده از ساختار داده tty مربوط به این پروسه و صدا کردن تابعی که می تواند یک رشته کاراکتر را بر روی tty چاپ کند این کار انجام می گیرد. مثال print_screen.c نحوه انجام این کار را نشان می دهد

فلاش LED های کیبورد

در بعضی شرایط شما مایلید که از یک راه ساده با دنیای بیرون ارتباط برقرار کنید. روشن و خاموش کردن LED های کیبورد می تواند یکی از این راه ها باشد. این راه یک راه سریع برای جلب توجه و نشان دادن وضعیت سیستم می تواند باشد. LED های کیبورد در هر سخت افزاری وجود دارند, همیشه قابل دیدن هستند, به نصبی یا چیزی مشابه ان احتیاج ندارند و کارکرد ساده تری نسبت به نوشتن در tty یا یک فایل دارند. در این مثال که کد ان را می توانید از] بدست اورید ماجول هسته کوچکی نوشته شده است که از زمان وارد شدن به هسته تا زمان خارج شدن چراغ های کیبورد شما را خاموش و روشن می کند. (البته این مثال کمی تغییر داده شده تا جذاب تر باشد

زمان بندی کارها

اغلب ما کارهایی داریم که می خواهیم سر زمان مشخصی اجرا شوند. اگر کار ما توسط پروسه ها قابل انجام است ان را در فایل crontab می گذاریم. اگر قرار است این کار توسط هسته انجام شود دو امکان در اختیار داریم: راه اول این است که پروسه موردنظرمان را در crontab بگذاریم و ماجول موردنظر را با استفاده از تابع سیستمی ای که در این پروسه بیدار نماییم. مثلا با باز کردن فایلی ماجولی را بیدار کنیم. همانطور که حدس زده اید این کار کاملا غیر بهینه است. با استفاده از crontab پروسه ای را اجرا می کنیم و فایل اجرایی را به حافظه می اوریم که فقط یک ماجول هسته را بیدار کنیمبه جای این کار می توانیم تابعی ایجاد کنیم که در هر وقفه زمانی (timer interrupt) صدا زده شود. برای انجام این کار ابتدا بایستی کار (task) مورد نظرمان را ایجاد نماییم, ان را در ساختاری به نام workqueue_struct قرار دهیم. در واقع اشاره گری به این تابع در این صف قرار می گیرد. سپس از تابع queue_delayed_work برای قراردادن ان کار در لیست کارهای my_workqueue که لیست کارهایی که در وقفه زمانی بعد اجرا می شوندقرار می دهیمکد مثال sched.c یک نمونه ای از این زمان بندی را نشان می دهد. در این کد کار خاصی در هر وقفه زمانی انجام می شود.

راه انداز وقفه ها (Interrupt Handlers)

مواردی که تاکنون در این مجموعه مورد بررسی قرار دادیم توابعی بودند که به عنوان پاسخی برای پروسه هایی که ان امکانات را درخواست داده بودند نوشته می شدند. به عنوان مثال توابعی برای ارتباط با یک فایل, پاسخی به ioctl() یا صدا کردن یک تابع سیستمی برای پاسخ به یک درخواست. اما باید توجه داشت که وظیفه هسته تنها پاسخ گویی به درخواست های پروسه ها نیست. یکی دیگر از وظایف مهم هسته ارتباط با سخت افزار های متصل به سیستم است. به طور کلی دو نوع روش کلی ارتباط بین CPU و دیگر سخت افزار های کامپیوتر وجود دارد. روش اول بدین صورت است که CPU به طور متناوب به دستگاه های دیگر سرک کشیده و دستورات لازم را می دهد. در روش دوم دستگاه هر وقت نیاز داشت که با CPU ارتباط برقرار کند ان را خبر می کندروش دوم که اصطلاحا interrupt یا وقفه نامیده می شود, نسبتا پیاده سازی سخت تری نسبت به روش اول دارد بدلیل اینکه راحتی را برای دستگاه ها به همراه می اورد نه CPU. دستگاه های سخت افزاری معمولا حافظه کمی دارند و اگر شما نتوانید در زمان معین اطلاعات انها را بخوانید ان اطلاعات از دست می روندر لینوکس وقفه های سخت افزاری Interrupt IRQ (Request) نامیده می شوند. IRQ ها دو دسته هستند: کوتاه وبلند. یک IRQ کوتاه زمان کوتاهی دارد لذا در حین این IRQ دیگر قسمت های سیستم متوقف شده و وقفه دیگری پاسخ داده نمی شود. IRQ بلند زمان بیشتری طول می کشد و در طی ان وقفه های دیگری نیز ممکن است اتفاق بیفتد. (البته این وقفه ها از دستگاه های یکسانی نمی تواند باشد). بنابراین بهتر است که یک راه انداز وقفه به صورت بلند تعریف شودهنگامی که یک وقفه به CPU می رسد, CPU کار خود را متوقف کرده مگر اینکه رسیدگی به وقفه جاری مهم تر از رسیدگی به وقفه رسیده باشد, در این صورت اجرای وقفه رسیده تا زمانی که وقفه مهم تر تمام شود به تاخیر می افتد), پارامترهای معینی در حافظه پشته (stack) ذخیره و تابع رسیدگی کننده به وقفه صدا زده می شوداین بدان معنی است که بعضی کارهای معین حق اجرا شدن در تابع رسیدگی کننده به وقفه را ندارند بدلیل اینکه سیستم در وضعیت نامشخصی می باشد. راه حل این مساله این است که تابع رسیدگی کننده به وقفه بعضی کارهای لازم را بی درنگ انجام داده که معمولا چیزی از سخت افزار می خواند یا چیزی به ان می فرستد و ادامه کار را در زمان دیگری زمان بندی می کند (که به این زمان "bottom half" گفته می شود) و باز می گرددهسته تضمین می کند که ادامه کار را در اولین

زمان ممکن انجام دهدمثال زیر نحوه پیاده سازی این رویه را نشان می دهد. کد این مثال را از] دریافت کنید.

جمع بندی مطالب

در این مجموعه از مقالات با مفاهیم کلی و نحوه پیاده سازی بعضی از ماجول های هسته لینوکس اشنا شدیم. در ابتدا مفاهیم کلی را بررسی کردیم. سپس به دستگاه های کاراکتری و ویژگی های انها پرداختیم. با فایل سیستم proc/ به عنوان یکی از راه های ورودی خروجی هسته اشنا شدیم. به بررسی sysfs به عنوان یک متدلوژی هوشمند انرژی در سیستم پرداختیم و با مرور مباحثی چون توابع سیستمی, متوقف کردن پروسه ها, زمان بندی کارها, پاسخ گویی به وقفه ها و ... اشنا شدیم. این مجموعه مسلما فقط در حد اشنایی بوده و تسلط کافی برای طراحی و پیاده سازی ماجول های هسته را به شما نخواهد داد. لذا در ذیل منابعی معرفی شده اند که با مطالعه انها می توانید این تسلط لازم را کسب نمایید:

- Linux Kernel Source Code
- Linux Kernel Documentation (Linux Source Code/Documentations)
- Understanding the Linux Kernel,2nd edition, O'Reilly.
- Linux Kernel Development, Sams publishing
- Linux Device Drivers, 3rd edition, O'Reilly.
- List of Kernel Resources