[[si:lab:lab6]]

Device Drivers

Table of Contents

Device Drivers
Implementarea unui device driver
Timere
Obținerea datei curente în kernel
Puncte de interes în kernel

Un device driver permite interfatarea unui periferic prin accesarea resurselor acestuia la nivelul kernelului. De cele mai multe ori, interfatarea presupune efectuarea de operatii de scriere si citire din registrele asociate dispozitivului respectiv. Din punctul de vedere al accesului utilizatorului, exista doua tipuri de device drivere:

Dispozitive tip bloc

Un dispozitiv bloc (block device) contine de obicei un sistem de fisiere, de exemplu o unitate de disc sau o memorie flash. Accesul la acest tip de dispozitiv se face doar prin scrierea si citirea de date la nivel de bloc, unde un bloc este de obicei 1kB de date.

Dispozitive tip caracter

Dispozitivele de tip caracter pot fi accesate in acelasi mod ca un fisier. Driver-ul de tip char implementeaza operatiile de deschidere, inchidere, citire si scriere. Consola si portul serial sunt doar doua exemple de asemenea dispozitive.

Implementarea unui device driver

In laboratorul anterior ati implementat un modul simplu de kernel. Acesta este primul pas in crearea unui device driver. Dupa cum probabil ati observat, un simplu modul de kernel nu permite primirea de comenzi din user space. Acest lucru poate fi implementat doar prin adaugarea unor noi functii modulului nostru si crearea unui char driver. Orice char driver trebuie sa implementeze patru functii:

- **open** - se executa la deschiderea dispozitivul de catre un proces
- **write** - implementeaza comportamentul dispozitivului la o comanda de scriere (ex. *echo blabla>/dev/my_device*)
- **read** - implementeaza comportamentul dispozitivului la o comanda de citire (ex. *cat /dev/my_device*
- **release** - se executa cand un proces incearca inchiderea dispozitivului

Inregistrarea unui nou character driver se face la initializare prin apelarea functiei ***register_chrdev()***. Aceasta primeste drept parametri numele device driver-ului si o structura tip *file_operations* care contine pointeri catre functiile de open, read, write si release. In urma inregistrarii cu succes, functia intoarce ca parametru un identificator intreg ce reprezinta numarul major (*Major number*) al dispozitivului. Acesta este unic pentru fiecare device in parte.

```
Major = register_chrdev(0, "my_device", &fops);
```

Numarul major joaca un rol important deoarece cu ajutorul lui se poate lega I/O-ul unui dispozitiv la un fisier anume. Acest lucru se poate realiza din consola prin comanda **mknod**:

```
mknod /dev/hello c Major 0
```

Primul parametru al acestei comenzi specifica numele fisierului prin care va putea fi accesat dispozitivul nou creat. Al doilea parametru specifica tipul device driverului asociat, iar ultimii doi parametri sunt numerele major si minor.

Operatia inversa, de inlaturare a unui device driver se poate face in cod prin functia ***unregister_chrdev***:

```
 unregister_chrdev(Major, "my_device");
```

Exemplul de mai jos arata un device driver tip caracter care implementeaza operatia de citire si contorizeaza numarul de accese.

```
#if defined(CONFIG_MODVERSIONS) && ! defined(MODVERSIONS)
#include <linux/modversions.h>
#define MODVERSIONS
#endif
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>  /* for put_user */

MODULE_DESCRIPTION("Chardev Module");
MODULE_AUTHOR("A Guy");
MODULE_LICENSE("GPL");

/*  Fuction Prototypes */
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev" /* Dev name as it appears in /proc/devices   */
#define BUF_LEN 80            /* Max length of the message from the device */

static int Major;            /* Major number assigned to our device driver */
static int Device_Open = 0;  /* Is device open?  Used to prevent multiple  */
```

```
                            /*   access to the device                       */
static char msg[BUF_LEN];    /* The msg the device will give when asked    */
static char *msg_Ptr;

static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};

int init_module(void)
{
    Major = register_chrdev(0, DEVICE_NAME, &fops);
    if (Major < 0) {
    printk ("Registering the character device failed with %d\n", Major);
    return Major;
    }

    printk("<1>I was assigned major number %d.  To talk to\n", Major);
    printk("<1>the driver, create a dev file with\n");
    printk("<1>'mknod /dev/hello c %d 0'.\n", Major);
    printk("<1>Try various minor numbers.  Try to cat and echo to\n");
    printk("<1>the device file.\n");
    printk("<1>Remove the device file and module when done.\n");

    return 0;
}

void cleanup_module(void)
{
    printk("Cleanup time\n");
    /* Unregister the device */
    unregister_chrdev(Major, DEVICE_NAME);
}


/* Called when a process tries to open the device file, like
 * "cat /dev/mycharfile"
 */
static int device_open(struct inode *inode, struct file *file)
{
    static int counter = 0;
    if (Device_Open) return -EBUSY;
    Device_Open++;
    sprintf(msg,"I already told you %d times Hello world!\n", counter++);
    msg_Ptr = msg;
    return SUCCESS;
}

/* Called when a process closes the device file.
*/
static int device_release(struct inode *inode, struct file *file)
{
    Device_Open --;      /* We're now ready for our next caller */
    /* Decrement the usage count, or else once you opened the file, you'll
    never get get rid of the module. */
    return 0;
}

/* Called when a process, which already opened the dev file, attempts to
read from it.*/
static ssize_t device_read(struct file *filp,
                           char *buffer,    /* The buffer to fill with data */
                           size_t length,   /* The length of the buffer     */
                           loff_t *offset)  /* Our offset in the file       */
{
    /* Number of bytes actually written to the buffer */
    int bytes_read = 0;

    /* If we're at the end of the message, return 0 signifying end of file */
    if (*msg_Ptr == 0) return 0;

    /* Actually put the data into the buffer */
    while (length && *msg_Ptr)  {

        /* The buffer is in the user data segment, not the kernel segment;
        * assignment won't work.  We have to use put_user which copies data from
        * the kernel data segment to the user data segment. */
        put_user(*(msg_Ptr++), buffer++);

        length--;
        bytes_read++;
        }

    /* Most read functions return the number of bytes put into the buffer */
    return bytes_read;
}
```

```
/*  Called when a process writes to dev file: echo "hi" > /dev/hello */
static ssize_t device_write(struct file *filp,
                            const char *buff,
                            size_t len,
                            loff_t *off)
{
    printk ("<1>Sorry, this operation isn't supported.\n");
    return -EINVAL;
}
```

Exercitiul 1

- Implementati un device driver tip caracter care sa aprinda un LED la scrierea valorii 1 si stingerea LED-ului la scrierea valorii 0.

```
echo 1 > /dev/led
```

Timere

Sunt situatii in care este utila introducerea unui timer in structura dispozitivului vostru. Acesta implementeaza o actiune repetata la un interval prestabilit (de exemplu un semnal de refresh pentru un afisaj). Un device driver poate implementa mai multe timere simultan, ele fiind declarate la initializare si apoi introduse intr-o lista inlantuita.

```
struct timer_list my_timer;
init_timer(&my_timer);      //initilizeaza timerul
add_timer(&my_timer);       //adauga timerul in lista
del_timer(&my_timer);       //inlatura timerul din lista
```

Pentru fiecare timer in parte, trebuie declarate perioada de declansare, un pointer catre o functie handler si un pointer catre eventualul parametru al functiei handler.

Timerul trebuie inlaturat din lista de timere active la descarcarea modulului din kernel.

Un exemplu de modul care implementeaza o functie timer este dat mai jos.

```
#include <linux/module.h>
#include <linux/autoconfig.h>
#include <linux/init.h>

MODULE_DESCRIPTION("Blinkenled Module");
MODULE_AUTHOR("SI");
MODULE_LICENSE("GPL");

struct timer_list my_timer;

char ledstatus = 0;

#define BLINK_DELAY   HZ/5

static void my_timer_func(unsigned long ptr)
{
 /* user code */

 my_timer.expires = jiffies + BLINK_DELAY;
 add_timer(&my_timer);
}

static int __init leds_init(void)
{
 int i;

 printk(KERN_INFO "blinkenled: loading\n");

 /*
  * Set up the LED blink timer the first time
  */
 init_timer(&my_timer);
 my_timer.function = my_timer_func;
 my_timer.data = (unsigned long)&ledstatus;
 my_timer.expires = jiffies + BLINK_DELAY;
 add_timer(&my_timer);

 return 0;
}

static void __exit leds_cleanup(void)
{
 printk(KERN_INFO "blinkenled: unloading...\n");
 del_timer(&my_timer);

}

module_init(leds_init);
module_exit(leds_cleanup);
```

Intrebare: Ce este "jiffies" si ce valoare are?

Exercitiul 2:

- Pornind de la exemplul anterior, scrieti un device driver care sa porneasca/opreasca aprinderea intermitenta a unui led de pe placa.

Functionalitatea pe care trebuie s-o implementeze este urmatoarea:

```
echo 1 > /dev/my_device   -> porneste secventa
echo 0 > /dev/my_device   -> opreste secventa si stinge ledul
```

Obținerea datei curente în kernel

Scrierea unui device driver pentru ceas pune următoarea problemă: cum aflu data curentă atunci când mă aflu în cod kernel? Există următoarea funcție:

```
void do_gettimeofday(struct timeval *tv);
```

Funcția de față returnează însă o structură timeval, asemănătoare cu structura cu același nume din userspace:

```
struct timeval {
        __kernel_time_t         tv_sec;         /* seconds */
        __kernel_suseconds_t    tv_usec;        /* microseconds */
};
```

tv_sec este numărul de secunde trecute de la 1 Ianuarie 1970 (The Epoch)! Transformarea în ore, minute și secunde ale zilei curente se poate face de mână (ugh!), sau se poate folosi o funcție din linux/rtc.h, rtc_time_to_tm:

```
void rtc_time_to_tm(unsigned long time, struct rtc_time *tm)
```

rtc_time se dovedește a fi ceea ce căutam:

```
struct rtc_time {
        int tm_sec;
        int tm_min;
        int tm_hour;
        int tm_mday;
        int tm_mon;
        int tm_year;
        int tm_wday;
        int tm_yday;
        int tm_isdst;
};
```

Punând cap la cap, următoarea secvență de cod obține timpul în ore, minute și secunde (conform system clock):

```
struct rtc_time now;
struct timeval tv;

do_gettimeofday(&tv);
rtc_time_to_tm(tv.tv_sec, &now);
printk(KERN_ALERT "Hello at %02d:%02d:%02d!\n",now.tm_hour,now.tm_min,now.tm_sec);
```

O altă variantă ar obține timpul chiar din hardware clock, interfațând direct cu driverul de rtc (realtime clock).

```
int rtc_read_time(struct rtc_device *rtc, struct rtc_time *tm);
```

Cea de-a treia variantă constă în citirea directă a regiștrilor RTC-ului, de fapt a registrului care conține timpul curent. Maparea perifericelor în memorie este dată în datasheet-ul AP7000-ului la pagina 75, acolo veți vedea adresa la care se află regiștrii RTC. Apoi aveți date generale despre funcționarea RTC-ului la pagina 120, iar la pagina 122 veți vedea tabelul cu regiștrii. VAL este registrul care conține timpul curent, îl veți putea citi cu:

```
rawtime = __raw_readl(adresa_de_bază_a_perifericului + offset_registru);
```

Exercițiul 3

- Folosind scheletul pentru acest exercițiu, obțineți data curentă în cele 3 modalități (prima este dată ca exemplu).
- Adaugati o noua functionalitate driverului de la exercitiul 2:

```
cat /dev/my_device        -> afiseaza data si ora curenta in consola
```

Puncte de interes în kernel

- arch/avr32/mach-at32ap/at32ap700x.c - search după 'rtc' și găsiți definirea perifericului și înregistrarea lui la inițializare
- drivers/rtc/rtc-lib.c
- drivers/rtc/rtc-at32ap700x.c - driver-ul de rtc pentru ap7000