Fakultet elektrotehnike i računarstva Sveučilišta u Zagrebu
Zavod za elektroničke sustave i obradbu informacija
Laboratorij za sustave i signale

Programska podrška mjernih i procesnih sustava

vježba br. 9: **I/O SYSTEM**

UVOD

Pisanje device drivera je u uskoj vezi sa samom jezgrom operacijskog sustava. Zato je potrebno dobro poznavanje jezgre za uspješno implementiranje device drivera.

Primjeri kôda u ovom textu će raditi na svim 2.6.x verzijama kernela a već buildani moduli (`*.ko`) su napravljeni za 2.6.11 verziju (koja dolazi sa Knoppix 3.8.1 distribucijom) i na drugim verzijama vjerojatno neće raditi.

Ovdje ćemo se zadržati samo na najosnovnijim konceptima kako bi se dobio osjećaj što je device driver i koja je njegova struktura.

Ukratko, driver je dio jezgre koji je zadužen za komuniciranje sa određenim uređajem koji je spojen na računalo. Većina uređaja je korisniku predstavljena preko datoteke, tj. čitanjem odnosno pisanjem u datoteku pristupa uređaju.

Postoje tri osnovne vrste uređaja:

- *Character devices*

Primjer su serijski i paralelni port. Nad njima se najčešće obavljaju funkcije poput `read()`, `write()`, `open()`, `close()`. Obilježja su da se komunikacija sa njima svodi na obično slanje/primanje byte-ova. Može se usporediti sa stream-om (tokom).

- *Block devices*

Tipičan predstavnik je hard disk. Za njih je specifično da mogu na sebi sadržavati cijeli datotečni sustav. To omogućava izvršenje naredbe `mount` nad njima. Također moguće se je micati po block device-u te zapisivati podatke gdje hoćemo, na točno određena mjesta (adrese).

- *Network interface*

Svi prijenosi preko mreže se obavljaju preko mrežnog sučelja. Tu se vodi računa o primanju i slanju paketa i sl. (TCP/IP stack). Primjer toga je mrežna kartica. Ovo je specifičan device jer se njemu ne pristupa preko datoteke.

Kada korisnik u svom programu počne čitati i pisati po datoteci, koja zapravo predstavlja uređaj, tada jezgra OS-a taj zahtijev proslijedi driver-u koji zna kako koristiti dotični uređaj.

U ovoj vježbi mi ćemo se primarno baviti character uređajima jer su oni jednostavniji za koristiti.

ZADACI ZA VJEŽBU

1. HELLO WORLD DEVICE DRIVER

Kao prvi primjer drivera, proučit ćemo sljedeći kôd:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

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);
```

U trenutku učitavanja driver-a starta se funkcija koja je definirana preko `module_init(ime_funkcije)` koja obično služi za inicijalizaciju drivera. Kod brisanja, izvrši se funkcija definirana sa `module_exit(ime_funkcije)`.

Zanimljivo je primijetiti da driver nema `main()` funkciju. Aplikacija obično izvrši zadatak i onda umire. Za razliku od nje driver se sastoji od skupa funkcija za rukovanje uređajima i te funkcije se zovu onda kada se zatrebaju. Ovaj gornji primjer driver-a ne implementira niti jednu upotrebljivu funkciju nego je samo dan primjer onog što svaki driver mora imati.

`printk()` funkcija je ekvivalentna `printf()` funkciji samo je `printk()` definirana unutar kernela. Naime kernel se izvodu bez pomoći C biblioteke.

Postoji bitna razlika između programa koji izvršavaju funkciju driver-a i programa (aplikacije) koji je napisao korisnik. Dva su prostora u kojima se može izvršavati proces. Može biti *user space* ili *kernel space*. Aplikacija se izvršava u *user space*-u i ona koristi biblioteke koje su joj tamo dostupne (npr. standardna C biblioteka `libc`). U *kernel space*-u se izvršavaju funkcije driver-a i one se povezuje sa bibliotekom od kernela. Zato je također posao drivera da podatke prebacuje između *kernel space* i *user space* gdje je startana aplikacija koja je tražila podatke sa uređaja.

Gornji primjer drivera je zamišljen da bude implementiran kao modul. Rezultat *build-a* modula je datoteka `hello.ko`. Kako se do njega dođe iz *sourcea* nije pokazano jer zahtjeva neke datoteke koje zbog štednje prostora ne postoje na Knoppix CD distribuciji.

Također da bi vidjeli sadržaj koji ispisuje `printk()` moramo spremati poruke kernela u datoteku (*log file*). To radi daemon `klogd`. Zato ga treba prvo pokrenuti:

```
# /sbin/klogd –f /home/knoppix/kernel.log
```

Rezultat sada možemo pratiti na drugom virtualnom terminalu pomoću naredbe:

```
# tail –f /home/knoppix/kernel.log
```

Modul se uključuje u jezgru naredbom `insmod`.

```
# insmod hello.ko
```

Da vidimo da je modul doista učitan možemo izlistati sve uključene module sa `lsmod` naredbom.

```
# lsmod
```

Kad ga hoćemo maknuti potrebno je otipkati:

```
# rmmod hello
```

2. DEVICE DRIVER

U ovom dijelu vježbe će se koristiti memorija kao uređaj. Koristit ćemo driver koji će puniti i prazniti memorijski prostor. Taj memorijski prostor će biti globalni i perzistentan. Globalan znači da će svi koji koriste taj driver koristiti isti memorijski prostor a perzistentan znači da se memorija ne oslobađa pozivom `close()`. Takvim driver-om je moguće zauzeti svu raspoloživu memoriju računala. Međutim to se ne preporuča jer računalo postaje vrlo slabo interaktivno. Dakle pisanjem i čitanjem specifične datoteke mi punimo i praznimo memoriju. Primjer tog drivera je uzet iz knjige Rubini: "Linux Device Drivers", O'Reilly.

Kod je dan u datotekama `scull.h` i `main.c`.

Modul je već izgrađen (`scull.ko`) i učitava se i briše sa `insmod` i `rmmod` naredbama. Source datoteke su ostale nepromijenjene kako bi se dobio uvid u tipičnu strukturu jednog drivera.

Bitno je uočiti funkcije: `scull_open, scull_read, scull_write` koje se pozivaju kad se pokrenu funkcije `open, read` i `write` nad datotekom `/dev/scull`. Koje točno funkcije se pozivaju pri pozivu generičkih funkcija zapisano je u strukturi `struct file_operations`. Mi u tu strukturu navedemo puna imena funkcija kao u gornjem slučaju `scull_open` i sl. U kodu `main.c` pronađite te funkcije i tu strukturu (pri dnu).

Bitno je znati sponu koja povezuje funkcije drivera sa datotekom koja predstavlja uređaj. Naime svaka datoteka u `/dev` direktoriju ima, umjesto veličine, dva broja. Zovu se *major* i *minor* brojevi. Primjer (major=61, minor=1):

```
crw-rw-rw-     root root      61,  1 Sep 11 2001  scull
```

Svaki driver kad se registrira definira za koji *major* broj je određen. *Minor* broj određuje drugi uređaj koji zahtijeva isti driver (npr. drugi serijski port). Koji driver je zadužen za koji *major* broj piše u datoteci */proc/devices* kada se modul učita. `scull` modul dinamički traži *major* broj tako da nakon što učitamo modul moramo pogledati koji je to broj:

```
# insmod scull.ko
# cat /proc/devices
```

Datoteka koja se asocira sa driverom se može ručno napraviti pomoću naredbe `mknod`.

```
# mknod /dev/scull c major_broj 1
```

`major_broj` je *major* a `1` *minor* broj, `c` označava da se radi o character device-u.

Tako da kad mi otvorimo datoteku to se zapravo prenese driver-u sa tim *major* brojem. Bitno je da dvije datoteke koje predstavljaju različite uređaje nemaju iste *major* brojeve. Zato je poželjno te brojeve dinamički određivati. Driver u kodu registrira *major* broj sa funkcijom `register_chrdev()`, odnosno odjavljuje sa `unregister_chrdev()`. Pronađite ih u kodu.

Svaka funkcija read ima između ostalog i zadatak prebacivati sadžaj iz kernel space-a u user space. To se u verziji kernela 2.6.x obavlja funkcijom `copy_to_user()`. Funkcija write() radi suprotnu stvar tj. prebacuje iz user space-a u kernel space. To obavlja funkcija `copy_from_user()`. Pronađite gornje funkcije u kodu.

Kad je modul scull uspješno učitan tada se može isprobati njegova funkcionalnost.
Mi možemo pisati u datoteku `/dev/scull` koristeći i osnovne naredbe ljuske. Tako ako hoćemo u `/dev/scull` upisati 10MB (10240 kbyte-a) nula to možemo obaviti sljedećom naredbom.

```
# dd if=/dev/zero of=/dev/scull bs=1024k count=10
```

To će uzrokovati da se 10 blokova od po 1MB prebaci iz datoteke `/dev/zero` u `/dev/scull`. Sjetite se da se čitanjem datoteke `/dev/zero` dobijaju same nule.

Naredbom ljuske `free` moguće je dobiti ispis trenutno zauzete i prazne memorije. Tako da možemo puniti memoriju pomoću `dd` naredbe i gledati kako se ona smanjuje sa `free` naredbom.

```
# free
```

Driver `scull` je tako implementiran da svaki put kad se piše u datoteku memorijski prostor se prvo oslobodi. Tako da uzastopnim pisanjem identične `dd` naredbe, neće se memorija smanjivati jer će se svaki puta upisivati isti sadržaj. Zato ako se hoće uzeti više/manje memorije potrebno je povećati/smanjiti argument `count`.

Memorija će se oslobodit tek sa micanjem kompletnog modula iz jezgre:

```
# rmmod scull
```

3. KORIŠTENJE `IOCTL()` FUNKCIJE

Za podešavanje nekih kontrolnih parametara uređaja posebno je pogodna `ioctl()` funkcija. Neke kontrolne operacije se ne mogu dobiti slanjem i čitanjem podataka sa uređaja. Za takve stvari koristi se `ioctl()` funkcija. Npr. ne možemo čitanjem i pisanjem serijskog porta promijeniti baud rate nego to moramo učiniti sa `ioctl()` funkcijom.

Njenu primjenu ćemo isprobati na primjeru ledica na tastaturi. Zadatak se sastoji u tome da treba na tastaturi naizmjenično paliti i gasiti ledicu za *num_lock*, *scroll_lock* i *caps_lock*. Datoteka u `/dev` direktoriju koja predstavlja tastaturu je `/dev/console`.

Otvaranje tog device-a postižemo sa npr.

```
fd=open("/dev/console",O_RDONLY)
```

Ovdje smo otvorili sa zastavicom `O_RDONLY` ali to nije bitno jer mi nećemo ni čitati ni pisati.

Opći oblik `ioctl` naredbe koja se koristi u aplikaciji (user space) je slijedeći:

```
int ioctl(int fd, int cmd, ...);
```

`fd` je file descriptor i on se dobije funkcijom `open`.
`cmd` određuje naredbu koju želimo izvesti.
Povratna vrijednost je `0` za uspješnu operaciju.

Pozivanjem te funkcije se poziva `ioctl` funkcija koju implementira driver. Ona ima malo drugačije argumente. Mi ćemo se u ovom dijelu baviti isključivo upravljanjem iz aplikacije i koristit ćemo već napisani driver za tastaturu (`/dev/console`).

Tako za upravljanje LED-om se koriste dve naredbe.

```
ioctl(fd,KDGETLED,&led);
ioctl(fd,KDSETLED,led);
```

`led` je varijabla tipa `char` gdje su zadnja 3 bita zapravo stanja 3 LED-a i to redom od veće do manje značajnog bita: *caps lock*, *num lock*, *scroll lock*. Ako je argument funkcije `KDGETLED` onda se u varijablu `led` sprema sadržaj stanja LEDica a ako je `KDSETLED` onda se LEDice postavljaju u skladu sa vrijednostima zadnja tri bita varijable `led`.

Zadatak:

- Napisati program gdje se LEDice neprekidno pale jedna za drugom, krećući od lijevo prema desno pa onda od desno prema lijevo. Program nije zamišljen kao modul nego kao samostojeća aplikacija. Kostur programa je dan u datoteci `ledice.c`. Morate biti *root* zbog restriktivnih prava pristupa datoteci `/dev/console`.