Vežba 5 : Drajveri za sekvencijalne (karakter) uređaje

1. Rad u Kernel prostoru

Sistemska memorija u Linux operativnim sistemima je podeljena u dva prostora:

- Kernel prostor - deo memorije koji je rezervisan za izvršavanje kernela, njegovih ekstenzija i drajvera za uređaje. Rad u kernel prostoru je privilegovan, te daje neograničen pristup svim hardverskim resursima koji su prisutni u sistemu.
- Korisnički prostor - deo memorije rezervisan za rad korsničkih procesa (aplikacija). Svaki proces u korisničkom prostoru ima virtuelizovanu memoriju, te nema pravo pristupa memorijskim lokacijama drugih procesa i kernela.

Kernel prostoru se može pristupiti iz korisničkog prostora samo pomoću sistemskih poziva. Proces pravi sistemski poziv kao zahtev da kernel izvrši neku privilegovanu funkciju - najčešće komunikacija sa hardverom.

Dobra karakteristika Linux operativnog sistema je mogućnost da se proširi set funkcija koje nudi kernel i to za vreme njegovog rada. Svaki deo koda koji može biti dodat kernelu u vreme rada naziva se modul. Budući da moduli postaju deo kernela, pisanje koda mora biti prilagođeno kernel prostoru. Kod u kernel prostoru ima unikatne odlike u odnosu na aplikaciju koja se izbršava u korisničkom prostoru. Najvažnije razlike pisanja koda u kernel i korisničkom prostoru su:

1. Kernel nema pristup standardnim C bibliotekama. Postoji više razloga za ovo, ali su primarni brzina izvršavanja i prostor u memoriji. Kompletna C biblioteka ili bilo koji njen koristan podskup je prevelik i nedovoljno efikasan za izvršavanje u kernelu. Ipak, to ne znači da je korištenje biblioteka zabranjeno, te postoje kernelske implementacije većine poznatih C funkija. Na primer, za korištenje funkcija za manipulaciju stringovima je dovoljno uključiti <linux/string.h>. Bazni fajlovi za ove biblioteke se mogu naći na putanji /include u izvornom kernelovom stablu.
2. Kernel Linux-a je napisan u C programskom jeziku, ali ne u standardnom ANSI C-u. Budući da se linux kernel kompajlira pomoću GCC-a, gde god se to moglo primeniti, kernel programeri su koristili različite jezičke ekstenzije koje GCC podržava. Kernel programeri koriste ISO C99 i GNU C ekstenzije standardnog C jezika. Najvažnije razlike su podrška “inline” funkcija i anotacija uslovnih naredbi. Oznaka “inline” pri deklarisanju funkcije govori kompajleru da poziv te funkcije

zameni sa njenim kodom, čime se izbegava menjanje konteksta i dodatno gubljenje vremena pri pozivu funkcije. Inline funkciju je opravdano koristiti samo ako se ta funkcija sastoji od nekoliko linija koda. Anotacije grananja koriste makroe “unlikely” i “likely” kako bi naznačili kernelu koja je verovatnoća da će se skok desiti. Samim tim se generiše optimizovaniji asemblerski kod za datu arhitekturu procesora.

3. Kada aplikacija u korisničkom prostoru pokuša da pristupi nedozvoljenoj memorijskoj lokaciji, kernel može poslati *SIGSEGV* signal i uništiti proces. Nasuprot tome, kada kernel pokuša da pristupi nedozvoljenoj memorijskoj lokaciji, rezultati su manje kontrolisani, jer nema tko da nadzire kernel. U tom slučaju pri ilegalnom pristupu memoriji kao što je dereferenciranje NULL pokazivača, desiće se kernelski eror koji često rezultuje u padu kompletnog sistema. Dakle, potrebna je velika opreznost pri radu sa memorijom u kernelskom prostoru.
4. Kada procesi u korisničkom prostoru koriste operacije sa pokretnom tačkom (float, double), kernel je zadužen da izvrši promenu sa celobrojnog režima rada na režim sa pokretnom tačkom. Za razliku od korisničkog prostora, kernel nema luksuz automatske podrške operacija sa pokretnom tačkom, te je potrebno samostalno sačuvati i vratiti vrednosti registara za rad sa pokretnom tačkom (pored mnogih drugih zadataka). Baratanje registrima može često dovesti do nepovratnih grešaka te je najjednostavnije rešenje da se izbegnu operacije sa pokretnom tačkom ukoliko je to moguće.
5. Aplikacije u korisničkom prostoru mogu statički alocirati veliki broj promenljivih na steku, uključujući velike nizove sa hiljadama elemenata. Ovo je dozvoljeno jer je stek u korisničkom prostoru dovoljno velik, a i moguće ga je dinamički proširiti. Stek u kernel prostoru ima mnogo manje lokacija i fiksne je veličine, koja većinom zavisi od arhitekture procesora. U praksi je čest slučaj da je stek u kernelu 8KB na 32-bitnim a 16KB na 64-bitnim arhitekturama. Naravno, svaki proces ima svoj stek.
6. Kernel omogućava istovremeno izvršavanje više procesa, gde planer odlučuje koji od njih će se u svakom trenutku izvršavati na procesoru. Takođe kernel ima podršku za rad sa više procesorskih jezgara koji u svakom trenutku mogu istovremeno pristupiti nekom deljenom resursu. Ovi slučajevi mogu dovesti do čitanja pogrešne vrednosti iz memorije, a samim tim i izvršavanja programa na pograšan način. U kernelu je neophodno zaštititi deljene resurse kernel objektima kao što su semafori, muteksi i spin-lokovi. Ovoj temi će biti posvećena posebna pažnja u nekoj od narednih vežbi.

2. Moduli

Svaki modul može biti dinamički povezan sa kernelom u vreme rada pomoću **insmod** programa ili isključen iz kernela pomoću **rmmod** programa. U nastavku teksta je dat jednostavan modul u kome su definisane dve funkcije: hello_init i hello_exit. Pri pozivu insmod komande, kernel u fajlu traži makro *module_init,* kojem se kao parametar prosledjuje ime funkcije koju treba da izvrši. Dakle, *module_init* definiše funkciju koja se poziva pri izvršavanju insmod komande. Ekvivalentno tome, makro *module_exit* definiše funkciju koja se poziva pri pokretanju *rmmod* komande.

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

MODULE_LICENSE("Dual BSD/GPL");

static int __init hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

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

U datom “hello” modulu, ove dve funkcije samo pozivaju kernelovu ugrađenu funkciju za ispis poruka u terminalu - *printk*. Printk funkcioniše na sličan način kao i *printf* funkcija, razlika je da je moguće naglasiti nivo prioriteta ispisa poruke korišćenjem jednog od predefinisanih makroa:

0	KERN_EMERG	Hitan problem, sistem će uskoro da padne
1	KERN_ALERT	Veći problem, zahteva momentalnu pažnju i rešavanje
2	KERN_CRIT	Problem koji potenijalno može biti kritičan za rad sistema
3	KERN_ERR	Greška (error)
4	KERN_WARNING	Upozorenje (warning)
5	KERN_NOTICE	Normalna poruka, obaveštenje
6	KERN_INFO	Dodatna informaija
7	KERN_DEBUG	Poruka za debagovanje

Ukoliko se ne naglasi nivo prioriteta, u većini linux distribucija će biti postavljen na KERN_WARNING. Kao što se može primetiti iz tabele, što je niži broj pridružen makrou, to je poruka bitnija. U sistemu je postavljen određeni nivo prioriteta, te se samo poruke sa manjim brojem on njega ispisuju. Trenutno podešavanje se može pročitati u fajlu /proc/sys/kernel/printk (prvi broj u ovom fajlu definiše trenutno podešavanje).

Napomena: U novijim distribucijama linux-a se poruke iz kernela ne ispisuju u terminalu sem ako je fizički (npr. serijska komunikacija), te je prethodne poruke moguće videti pokretanjem *dmesg* komande. Ukoliko se žele izbrisati poruke iz dmesg bafera, moguće je pokrenuti komandu sa parametrom -c (*dmesg -c*).

Insmod komanda kao parametar prima fajl sa ekstenzijom .ko (npr insmod hello.ko). Dakle, hello.c fajl je potrebno kompajlirati pre nego što se uključi u kernel. Kompajliranje se mora izvršiti pomoću istog makefile-a pomoću kojega se kompajlirao kernel. Iz ovoga razloga makefile za kompajliranje hello.c modula će promeniti direktorijum na putanju /lib/modules/$(shell uname -r)/build gde se nalazi kernelski makefile kao i sve biblioteke potrebne za kompajliranje modula ($(shell uname -r) je ništa više nego trenutna verzija kernela). Više detalja o ovom makefile-u

se može naći na predavanjima. Makefile je dat u nastavku teksta, te funkcioniše podjednako dobro za sve linux distribucije:

```
ifneq ($(KERNELRELEASE),)
      obj-m := Hello.o
else
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
            $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
            $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
            rm -f *~
endif
```

3. Upravljački brojevi

Sekvencijalnim (karakter) uređajima se pristupa preko *node* fajlova (čvorova) u fajl sistemu, na putanji */dev*. Ovi fajlovi su specijalne datoteke koje prave spregu između korisničkog i kernel prostora. Ukoliko se ispiše sadržaj */dev* direktorijuma koristeći ls -l komandu, dobijamo rezultat sličan sledećem:

```
brw-rw----   1 root   disk     8,   0      Nov 13 08:19 sda
brw-rw----   1 root   disk     8,   1      Nov 13 08:19 sda1
brw-rw----   1 root   disk     8,   2      Nov 13 08:19 sda2
crw-rw----   1 root   disk    21,   0      Nov 13 08:19 sg0
crw-------   1 root   root    10, 231      Nov 13 08:19 snapshot
crw-rw-rw-   1 root   tty      5,   0      Nov 13 11:30 tty
crw--w----   1 root   tty      4,  10      Nov 13 08:19 tty10
crw--w----   1 root   tty      4,  11      Nov 13 08:19 tty11
crw--w----   1 root   tty      4,  12      Nov 13 08:19 tty12
crw--w----   1 root   tty      4,  13      Nov 13 08:19 tty13
```

Prvo slovo označava tip drajvera, gde “c” predstavlja char (sekvencijalne) drajvere, dok “b” predstavlja block (blok) drajvere. U ovom kursu će se pažnja posvetiti samo sekvencijalnim (char) uređajima i drajverima za njih. Za razliku od običnih fajlova, mogu se primetiti dva broja odvojena zarezom. Oni se zovu upravljački brojevi uređaja, gde se prvi zove glavni (major) a drugi sporedni (minor) broj. Konvencija je da svaki drajver ima svoj unikatni glavni (major) broj. Ukoliko više uređaja koristi isti drajver, svaki od uređaja će imati drugačiji sporedni (minor) broj,

kako bi se moglo pristupiti svakom od njih. U prethodnom ispisu možemo videti da se serijskim terminalima (*tty, teletetypewriter*) pristupa preko glavnog broja 4 dok sporedni brojevi 10, 11, 12 i 13 govore o kojem se terminalu radi.

Unutar kernela, dev_t tip (denisan u <linux/types.h>) se koristi za predstavljanje brojeva uređaja, kako glavnih, tako i sporednih. Tip dev_t je 32- bitni: 12 bita se izdvaja za glavne brojeve i 20 za sporedne. Kod drajvera, ipak, ne treba praviti pretpostavke o načinu zapisivanja glavnih I sporednih brojeva u tipu dev_t, zato što se on menja od kernela do kernela. Bolje rešenje je da se koriste makroi definisani u <linux/kdev_t.h>.

MAJOR(dev_t dev);
MINOR(dev_t dev);

Ukoliko imamo na raspolaganju glavni i sporedni broj nekog uređaja, koje treba pretvoriti u dev_t tip, koristi se makro:

MKDEV(int major, int minor);

3.1. Statičko alociranje upravljačkih brojeva

Upravljački brojevi se koriste kako bi kernel povezao *node* fajl u */dev* direktorijumu sa određenim modulom (drajverom) koji je uključen u kernel pomoću *insmod* komande. U drajveru je potrebno zatražilii od kernela određene upravljačke brojeve. Ovo se radi funkcijom *register_chrdev_region* koja je definisana u <linux/fs.h>.

int register_chrdev_region(**dev_t** first,
unsigned int count,
char *name);

Prvi argument "*first*" definiše glavni i sporedni upravljački broj uređaja za koji se razvija drajver. Argument "*count"* predstavlja ukupan broj uređaja koji će biti pokriveni drajverom. Ukoliko u argumentu "first" imamo glavni broj "n" i sporedni broj "m", na ovaj način možemo automatski da dobijemo na korišćenje upravljačke brojeve: (n,m), (n,m+1), … , (n, m+count-1). Poslednji argument je ime drajvera, te je konvencija da se postavi na ime "node" fajla u /dev direktorijumu. Kao i većina kernelovih funkcija, vrednost koju vraća funkcija register_chrdev_region će biti 0 ako je uspešno izvršena alkoacija.

Za oslobađanje upravljačkih brojeva se koristi funkcija unregister_chrdev_region:

void unregister_chrdev_region(**dev_t** first,

unsigned int count);

Ovaj način alokacije je dobar kada u sistemu ne postoji mogo uređaja i ako unapred znamo koji su upravljački brojevi slobodni u sistemu. U ovom slučaju, u drajveru možemo pomoću makroa MKDEV i prethodno opisane funkcije *register_chrdev_region* rezervisati opseg upravljačkih brojeva, a zatim u terminalu pozvati komandu za pravljenje "node" fajla:

mknod /dev/hello c 240 0

Nakon ove komande će se u */dev* direktorijumu pojaviti "node" fajl za sekvencijalni uređaj "hello" sa upravljačkim brojevima 240 i 0.

U nastavku teksta se mogu videti *module_init* i *module_exit* funkcije za modul sa rezervisanim brojevima 240 i 0.

```
dev_t dev_id;
static int __init hello_init(void)
{
	int ret = 0;
	dev_id = MKDEV(240,0);
	ret = register_chrdev_region(dev_id,1,"Hello");
	if(ret)
	{
		printk(KERN_ERR "failed to register char device\n");
		return ret;
	}

	printk(KERN_INFO "Hello, world\n");
	return 0;
}

static void __exit hello_exit(void)
{
	unregister_chrdev_region(dev_id,1);
	printk(KERN_INFO "Goodbye, cruel world\n");
}
```

NAPOMENA: Ukoliko se statički alociraju upravljački brojevi, neophodno je u terminalu pokrenuti komandu: **mknod** /dev/hello c 240 0
kako bi se napravio node fajl u direktorijumu /dev.

3.2. Dinamičko alociranje upravljačkih brojeva

Dinamička dodela upravljačkih brojeva u većini slučajeva mnogo bolja opcija, zato što ne zahteva od korisnika da sam traži slobodne upravljačke brojeve u /dev direktorijumu i garantuje da će se naći brojevi nezavisno od sistema na kojem se nalazi. U ovom slučaju je potrebno koristiti funkciju **alloc_chrdev_region** umesto funkcije register_chrdev_region.

```
int alloc_chrdev_region (   dev_t * dev,
                            unsigned baseminor,
                            unsigned count,
                            const char * name);
```

Ova funkcija se poziva na sličan način kao i register_chrdev_region, sem što joj se promenljiva “dev” koja sadrži upravljačke brojeve, prosledi po referenci. Ukoliko je povratna vrednost 0, alokacija je uspešno izvršena i rezervisani upravljački brojevi se nalaze u provenljivoj “dev”.

Pri dinamičkoj alokaciji se za pravljenje “node” fajla više ne koristi MKNOD komanda, zato što se unapred ne znaju upravljački brojevi. Node fajl je moguće kreirati iz module_init funkcije pozivom sledeće dve funkcije:

```
struct class * class_create (struct module*  owner,
                             const char *  name);

struct device * device_create (   struct class *  class,
                                  struct device* parent,
                                  dev_t          dev_t,
                                  void *         drvdata,
                                  const char*    name);
```

Funkcija class_create je neophodna jer se njena povratna vrednost koristi kao prvi parametar u funkciji device_create. Ostali argumenti su isti kao i u prethodno pomenutim funkcijama. Parametri *parent* i *drvdata* se mogu postaviti na NULL;
Za povratak resursa se koriste sledeće dve funkcije:

```
void device_destroy (struct class *class, dev_t devt);
void class_destroy (struct class *class);
```

U nastavku teksta su date *module_init* i *module_exit* funkcije za modul koji rezerviše prve slobodne upravljačke brojeve i automatski pravi *node* fajl u /*dev* direktorijumu.

```c
dev_t my_dev_id;
static struct class *my_class;
static struct device *my_device;

static int __init hello_init(void)
{
   int ret = 0;
   ret = alloc_chrdev_region(&my_dev_id, 0, 1, "hello");
   if (ret){
      printk(KERN_ERR "Failed to register char device\n");
      return ret;
   }
   printk(KERN_INFO "Char device region allocated\n");

   my_class = class_create(THIS_MODULE, "hello_class");
   if (my_class == NULL){
      printk(KERN_ERR "Failed to create class\n");
      goto fail_0;
   }
   printk(KERN_INFO "Class created\n");

   my_device = device_create(my_class, NULL, my_dev_id, NULL, "hello");
   if (my_device == NULL){
      printk(KERN_ERR "Failed to create device\n");
      goto fail_1;
   }
   printk(KERN_INFO "Device created\n");
   printk(KERN_INFO "Hello world\n");
   return 0;

   fail_1:
      class_destroy(my_class);
   fail_0:
      unregister_chrdev_region(my_dev_id, 1);
   return -1;
}

static void __exit hello_exit(void)
{
   device_destroy(my_class, my_dev_id);
   class_destroy(my_class);
   unregister_chrdev_region(my_dev_id,1);
   printk(KERN_INFO "Goodbye, cruel world\n");
}
```

3.3. file_operations struktura

Budući da se sa drajverima za sekvencijalne uređaje komunicira kao sa tekstualnim fajlovima u fajl sistemu, potrebno je definisati funkcionalnosti koje se trebaju izvršiti kada se pozove neka od funkcija za rad sa fajlovima. U ovu svrhu služi struktura file_operations čija su polja pokazivači na funkcije za rad sa fajlovima. U sledećem kodu su prikazana samo neka od polja strukture:

```
struct file_operations {
       struct module *owner;
       loff_t (*llseek) (struct file *, loff_t, int);
       ssize_t (*read) (struct file *, char *, size_t, loff_t *);
       ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
       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 (*fasync) (int, struct file *, int);
   ...
};
```

U modulu je potrebno deklarisati i definisati željene funkcije, na takav način da imaju isti tip povratne vrednosti i iste argumente. Nakon toga se može napraviti struktura tipa *file_operations*, čiji se pokazivači mogu inicijalizovati na funkcije modula:

```
int hello_open(struct inode*, struct file*);
int hello_close(struct inode*, struct file*);
ssize_t hello_read(struct file*, char*, size_t, loff_t*);
ssize_t hello_write(struct file*, const char* ,size_t, loff_t*);

struct file_operations my_fops =
{
   .owner = THIS_MODULE,
   .read = hello_read,
   .write = hello_write,
   .open = hello_open,
   .release = hello_close,
};

int hello_open(struct inode *pinode, struct file *pfile)
{
            printk(KERN_INFO "Succesfully opened file\n");
            return 0;
```

```
}

int hello_close(struct inode *pinode, struct file *pfile)
{
            printk(KERN_INFO "Succesfully closed file\n");
            return 0;
}

ssize_t hello_read(struct file *pfile, char __user *buffer, size_t
length, loff_t *offset)
{
            printk(KERN_INFO "Succesfully read from file\n");
            return 0;
}

ssize_t hello_write(struct file *pfile, const char __user *buffer,
size_t length, loff_t *offset)
{
            printk(KERN_INFO "Succesfully wrote into file\n");
            return length;
}
```

3.4. cdev struktura

Poslednji korak u definisanju drajvera je registrovanje c_dev strukture. Deklaracija cdev strukture se nalazi u <linux/cdev.h>. Potrebno je prvo alocirati memoriju za cdev strukturu, a zatim postaviti njena polja *ops* i *owner*. Polje *ops* se postavlja na prethodno definisanu strukturu file_operations.

struct cdev *my_cdev = **cdev_alloc**();
my_cdev->ops = &my_fops;
my_cdev->owner = THIS_MODULE;

Alternativno, isti posao može da se uradi i na sledeći način:

struct cdev my_cdev;
cdev_init(my_cdev, &my_fops);

Kada su polja cdev strukture postavljena, poslednji korak je da kernel obavi registrovanje upravo kreirane *cdev* strukture pomoću funkcije *cdev_add*-

int cdev_add(**struct cdev** *dev,
dev_t num,

unsigned int count);

Na ovaj način su upravljački brojevi (*dev_t*) vezani za *cdev* strukturu, čije polje *ops* pokazuje na *file_operations* strukturu. Ukoliko korisnička aplikacija pokuša da izvrši neku operaciju sa *node* fajlom u */dev* direktorijumu, kernel će tu operaciju preusmeriti u modul sa istim upravljačkim brojevima. U zavisnosti od operacije pozvaće se određena funkcija na koju pokazuje adekvatno polje unutar file_operation strukture. Na primer, ukoliko se pokuša otvoriti node fajl /dev/hello, kernel će pozvati funkciju na koju pokazuje polje “open” u strukturi file_operations.

Uklanjanje cdev strukture iz kernela se vrši pomoću funkcije cdev_del:

void cdev_del(**struct cdev** *dev);

Ukoliko u hello modul dodamo file_oerations strukturu iz prethodnog poglavlja, te napravimo i registrujemo cdev strukturu, kompletan kod modula sa dinamičkim alociranjem upravljačkih brojeva izgleda kao u nastavku:

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/cdev.h>

MODULE_LICENSE("Dual BSD/GPL");

dev_t my_dev_id;
static struct class *my_class;
static struct device *my_device;
static struct cdev *my_cdev;

int hello_open(struct inode *pinode, struct file *pfile);
int hello_close(struct inode *pinode, struct file *pfile);
ssize_t hello_read(struct file *pfile, char __user *buffer, size_t
length, loff_t *offset);
ssize_t hello_write(struct file *pfile, const char __user *buffer,
size_t length, loff_t *offset);
```

```
struct file_operations my_fops =
{
      .owner = THIS_MODULE,
      .open = hello_open,
      .read = hello_read,
      .write = hello_write,
      .release = hello_close,
};


int hello_open(struct inode *pinode, struct file *pfile)
{
           printk(KERN_INFO "Succesfully opened file\n");
           return 0;
}

int hello_close(struct inode *pinode, struct file *pfile)
{
           printk(KERN_INFO "Succesfully closed file\n");
           return 0;
}

ssize_t hello_read(struct file *pfile, char __user *buffer, size_t
length, loff_t *offset)
{
           printk(KERN_INFO "Succesfully read from file\n");
           return 0;
}

ssize_t hello_write(struct file *pfile, const char __user *buffer,
size_t length, loff_t *offset)
{
           printk(KERN_INFO "Succesfully wrote into file\n");
           return length;
}

static int __init hello_init(void)
{
   int ret = 0;
   ret = alloc_chrdev_region(&my_dev_id, 0, 1, "hello");
   if (ret){
```

```
      printk(KERN_ERR "failed to register char device\n");
      return ret;
   }
   printk(KERN_INFO "char device region allocated\n");

   my_class = class_create(THIS_MODULE, "hello_class");
   if (my_class == NULL){
      printk(KERN_ERR "failed to create class\n");
      goto fail_0;
   }
   printk(KERN_INFO "class created\n");

   my_device = device_create(my_class, NULL, my_dev_id, NULL,
"hello");
   if (my_device == NULL){
      printk(KERN_ERR "failed to create device\n");
      goto fail_1;
   }
   printk(KERN_INFO "device created\n");

     my_cdev = cdev_alloc();
     my_cdev->ops = &my_fops;
     my_cdev->owner = THIS_MODULE;
     ret = cdev_add(my_cdev, my_dev_id, 1);
     if (ret)
     {
      printk(KERN_ERR "failed to add cdev\n");
           goto fail_2;
     }
   printk(KERN_INFO "cdev added\n");
   printk(KERN_INFO "Hello world\n");
   return 0;

   fail_2:
     device_destroy(my_class, my_dev_id);
   fail_1:
      class_destroy(my_class);
   fail_0:
      unregister_chrdev_region(my_dev_id, 1);
   return -1;
}
```

```
static void __exit hello_exit(void)
{
   cdev_del(my_cdev);
   device_destroy(my_class, my_dev_id);
   class_destroy(my_class);
   unregister_chrdev_region(my_dev_id,1);
   printk(KERN_INFO "Goodbye, cruel world\n");
}


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

4. Testiranje modula

Ukoliko nakon kompajliranja modula, izvršimo komandu “insmod Hello.ko”, te pokrenemo komandu dmesg, dobijamo sledeći rezultat:

[3496.825447] char device region allocated
[3496.825454] class created
[3496.825502] device created
[3496.825504] cdev added
[3496.825505] Hello world

Kada upišemo nešto u *node* fajl komandom “echo ‘asdf’ > /dev/hello”, dobijamo sledeće poruke:

[3840.417950] Succesfully opened file
[3840.422678] Succesfully wrote into file
[3840.423291] Succesfully closed file

Pri pokušaju čitanja komandom “cat /dev/hello” dobijamo sledeće poruke:

[3955.299105] Succesfully opened file
[3955.299112] Succesfully read from file
[3955.299117] Succesfully closed file

Ovim je potvrđeno da modul radi kako je planirano i da se uspešno pozivaju sve funkcije koje smo definisali.