Vežba 6 : Drajveri za sekvencijalne (karakter) uređaje

1. Pomoćne strukture za rad sa fajlovima

Na prošlim vežbama su demonstrirana dva načina (statički i dinamički) da se rezervišu upravljački brojevi za neki modul, te da se napravi *node* fajl u direktorijumu */dev* kako bi mogli pristupiti modulu. Na ovim vežbama će se demonstrirati pravilno rukovanje podacima unutar funkcija za rad sa fajlovima (*open, release, read, write*). Kako bismo krenuli da modifikujemo te funkcije, moramo razumeti strukture koje im se prosleđuju preko parametara.

1.1. inode struktura

Linux operativni sistemi se zasnivaju na zlatnom pravilu: “Sve je fajl”. Dakle, svemu se pristupa preko imena u fajl sistemu, bilo da je tekstualna datoteka, direktorijum, blok uređaj (npr. hard-disk, SSD, CD), karakter uređaj (tastatura, miš, serijski port) ili pak nešto peto. Za čuvanje informacija o fajlovima se koristi struktura *inode*. Svaki fajl u fajl sistemu ima sebi pridruženu *inode* strukturu koja između ostalog sadrži:

- mod - dozvole čitanja, upisa i izvršavanja
- tip fajla - da li je obična datoteka, direktorijum, sekvencijalni uređaj, blok uređaj, itd
- vlasništvo - ko je vlasnik datoteke, kojoj grupi pripada, itd
- veličina - koliko bajtova u memoriji zauzima fajl
- vremenske informacije - datum kreiranja, poslednje modifikacije, itd
- veličina memorijskog bloka - minimalana veličina bloka u memoriji koji se može adresirati (u bajtovima)
- blok pokazivače - adrese memorijskih blokova na kojima se može naći sadržaj fajla

Međutim, nama su za *node* fajlove najbitna polja:

- **dev_t** *i_rdev*;
- **struct cdev** **i_cdev*;

Kao što je pomenuto na prethodnim vežbama, *dev_t* polje sadrži glavni (major) i sporedni (minor) broj uređaja kojem se pristupa. Kada u modulu (drajveru) registrujemo *cdev* strukturu funkcijom *cdev_add(struct cdev my_cdev, dev_t my_dev, int count),* kernel će pronaći *node* fajl koji ima iste upravljačke brojeve na polju *dev_t*, te će pokazivač *i_cdev* postaviti na

prosleđenu strukturu. Na taj način, je napravljena veza između modula i *node* fajla u /dev direktorijumu.

Kada se izvršavaju operacije *open* i *release*(close) nad drajverom, kernel kao parametar šalje *inode* strukturu. Na ovaj način drajver može pristupiti svim informacijama o fajlu koji ga je pozvao.

1.2. file struktura

Druga bitna struktura za opis fajlova je *struct file*. Ona nema korelacije sa FILE strukturom o kojoj je bilo priče u trećoj vežbi (tokovi, stream). FILE struktura je C-ovska te se nikada ne pojavljuje u kernel prostoru, kao što se *file* struktura nikada ne pojavljuje u korisničkom prostoru. *file* struktura u kernelu predstavlja instancu otvorenog fajla. Ova struktura postoji za svaki otvoreni fajl u Linux-u te nije specifična za drajvere. Kernel kreira strukturu pri otvaranju fajla, dok je uništava pri zatvaranju fajla. Prosleđuje se kao parametar svim funkcijama za rad sa fajlovima (open, release, write, read...).
Najbitnija polja file strukture su sledeća:

- **mode_t** *f_mode*
  - Mod u kojem je otvoren fajl, može biti otvoren za čitanje (*FMODE_READ* bit setovan), za upis (*FMODE_WRITE* bit setovan) ili oboje.
- **loff_t** *f_ops*
  - Trenutna pozicija (kursora) za čitanje i upis. Ovo je 64-bitna vrednost. Drajver može da čita ovu vrednost ako je potrebna, ali ne treba da je modifikuje.
- **unsigned int** *f_flags*
  - Signalne zastavice (flag) kao *O_READONLY* ili *O_NONBLOCK*. Signaliziraju drajveru na koji način da rukuje fajlom.
- **struct file_operations** **f_ops*
  - Pokazivač na strukturu file_operations koja sadrži pokazivače na funkcije za rad sa fajlovima (objašnjeno na prethodnim vežbama). Dozvoljeno je promeniti funkcije u toku rada drajvera čime se obezbeđuje velika fleksibilnost, ali nije preporučivo.
- **struct inode** **f_inode*
  - Pokazivač na prethodno opisanu *inode* strukturu. Budući da *inode* struktura poseduje polje koje sadrži upravljačke brojeve (**dev_t** i_rdev), preko ovog *f_inode* pokazivača se može pristupiti sporednom (minor) upravljačkom broju kada drajver kontroliše više uređaja.
- **void** **private_data*
  - Polje koje može da čuva bilo koji tip informacije koji je potreban dizajneru drajvera.

2. Prebacivanje podataka između korisničkog i kernel prostora

Dizajneri kernel modula moraju biti oprezni kada rade sa bilo kojim podacima koji su prosledjeni iz korisničkog prostora. U ovu svrhu postoji makro "__user" koji u funkcijama označava parametre koji dolaze direktno iz korisničkog prostora. Kod funkcije za upis (*my_write* u nastavku), podaci koji se šalju iz korisničkog prostora su prosleđeni preko pokazivača na prvi karakter (*buffer*) i dužine niza (*length*).

```
ssize_t my_write(   struct file *pfile,
                    const char __user *buffer,
                    size_t length,
                    loff_t *offset)
```

Loša praksa je dereferencirati pokazivač poslat od strane procesa iz korisničkog prostora bez dodatnih provera. Pokazivač može imati nevalidnu adresu ili može pokazivati na adresu koja pripada adresnom prostoru kernela. Ukoliko bi inženjer pokušao da dereferencira takav pokazivač, mogao bi dovesti do greške koja bi rezultovala u padu sistema, ili da uvede opasnu rupu u sigurnosti sistema. U ovu svrhu je napravljena funkcija koja proverava da li su vrednosti pokazivača koje šalje korisnik validne adrese unutar korisničkog prostora, te ako jesu, kopira vrednosti na koje one pokazuju u kernel prostor. Ova funkcija se zove *copy_from_user* i prima tri parametra:

```
unsigned long copy_from_user (void *  to,
                              const void __user *  from,
                              unsigned long  n);
```

Prvi parametar "to" je pokazivač na prvi član niza u kernel prostoru gde će podaci biti smešteni. Drugi parametar "from" je niz podataka iz korisničkog prostora koji će biti kopirani u kernel prostor. Treći parametar "n" je dužina niza koji se kopira. Pre nego što se kopiraju podaci potrebno je alocirati niz u kernel prostoru.

Ekvivalentno prethodnom slučaju, kod funkcije za čitanje, podaci iz kernela se moraju smestiti u niz karaktera određen pokazivačem na prvi (buffer) i dužinom (length).

```
ssize_t my_read(   struct file *pfile,
                   char __user *buffer,
                   size_t length,
                   loff_t *offset)
```

Procesi iz korisničkog prostora nemaju pristup memorijskim adresama kernela, te se niz iz kernela mora kopirati u korisnički adresni prostor. U ovu svrhu se koristi funkcija *copy_to_user* koja ima slične parametre kao prethodno pomenuta funkcija *copy_from_user*. Jedina razlika je što u ovom slučaju parametar "to" ima oznaku __user, zato što je sada on pokazivač u korisničkom prostoru.

```
unsigned long copy_to_user (  void __user * to,
                              const void *  from,
                              unsigned long  n);
```

Obe funkcije vraćaju broj bajtova koji je nije uspešno kopiran. Prema tome, pri uspešnom izvršavanju funkcije se vraća vrednost 0. U suprotnom je potrebno obavestiti korisnika da adrese nisu validne (EFAULT)[1]. Korišćenjem ovih funkcija se izbegava mogućnost da korisnička greška ili zlоćudna operacija kompromituje stabilnost sistema. Kako bi drajver imao pristup deklaraciji ovih funkcija, potrebno je uključiti <linux/uaccess.h>.

3. Pomoćne funkcije za rad sa stringovima

Linux kernel nema pristup standardnim C bibliotekama, ali mnoge funkcije su implementirane u linux kernelskim bibliotekama. Iako je dozvoljeno da dizajner drajvera implementira svoje pomoćne funkcije, preporučuje se upotreba ugrađenih zbog brzine i sigurnosti. Budući da se na ovom kursu posvećuje pažnja drajverima za sekvencijalne (karakter) uređaje, funkcije za rad sa stringovima će biti od ključnog značaja. Većina ovih funkcija se nalazi deklarisana u heder fajlovima <linux/kernel.h> i <linux/string.h>. U nastavku će biti navedene preporučene funkcije iz ovih biblioteka, koje su studentima poznate iz rada sa aplikacijama u korisničkom prostoru.

Sve funkcije koje su dostupne u **<linux/kernel.h>** kao i njihovi opisi se mogu pogledati na adresi: https://www.kernel.org/doc/htmldocs/kernel-api/libc.html#id-1.4.3

Za koverziju stringa u celobrojni tip (u zavisnosti od širine i predznaka):

int **kstrtoull**(const char *s, unsigned int base, unsigned long long *res);
int **kstrtoll**(const char *s, unsigned int base, long long *res);
int **kstrtoul**(const char *s, unsigned int base, unsigned long *res)
int **kstrtol**(const char *s, unsigned int base, long *res)
int **kstrtouint**(const char *s, unsigned int base, unsigned int *res);

[1] Za pravilno korištenje error notacije u linux kernelu pročitati dodatak na kraju ovog pdf-a

int **kstrtoint**(const char *s, unsigned int base, int *res);

Za formatirani ispis u string:
int **scnprintf**(char *buf, size_t size, const char *fmt, ...);

Za parsiranje stringa:
int **sscanf**(const char *, const char *, ...);

Ukolko je dodatna manipulacija stringovma potrebna, funkcije koje su dostupne u **<linux/string.h>** kao i njihovi opisi se mogu pogledati na adresi: https://www.kernel.org/doc/htmldocs/kernel-api/ch02s02.html

4. Primer drajvera (Storage)

U nastavku teksta će prethodne funkcije biti demonstrirane na jednostavnom primeru drajvera koji čuva niz od deset celobrojnih elemenata - Storage drajver. Za čitanje vrednosti svih deset elemenata je moguće korišćenje storage_read funkcije: cat /dev/storage. Za upis vrednosti u niz je potrebno pozvati storage_write funkciju i proslediti joj vrednost i poziciju (indeks) odvojene zarezom: echo “5,3” > /dev/storage. Prethodni poziv bi upisao celobrojnu vrednost 5 u član niza sa indeksom 3. U drajveru je definisan globalni statički niz storage (int storage[10]). U *storage_init* funkciji se vrednosti ovog niza postave na nule koristeći for petlju.

4.1. storage_write funkcija

Ova funkcija se poziva kada proces iz korisničkog prostora pokuša da upiše neki podatak u drajver. Podatak (string) je predstavljen pomoću pokazivača na prvi karakter (**buffer*) i broja karaktera koji je potrebno upisati (*length*). Kao i kod svih funkcija za rad sa fajlovima, prosleđen je pokazivač na file strukturu (*pfile). Poslednji parametar ***offset* je pokazivač na tip loff_t (long offset type) preko kojega korisnik nagoveštava na koje mesto u fajlu želi da upiše tu informaciju. Ukoliko drajver ne implementira funkcije za rad sa kursorom, ovaj parametar nije potrebno koristiti i ažurirati pri svakom upisu. Povratna vrednost ove funkcije je broj uspešno upisanih bajtova (karaktera). Ukoliko se desi greška potrebno je vratiti negativnu vrednost.

```
ssize_t storage_write(struct file *pfile,
                      const char __user *buffer,
                      size_t length,
                      loff_t *offset)
{
```

```
	char buff[20];
	int position, value, ret;



	ret = copy_from_user(buff, buffer, length);
	if(ret)
		return -EFAULT;

	buff[length-1] = '\0';

	ret = sscanf(buff,"%d,%d",&value,&position);

	if(ret==2)//two parameters parsed in sscanf
	{
		if(position >=0 && position <=9)
		{
			storage[position] = value;
			printk(KERN_INFO "Succesfully wrote value %d in
position %d\n", value, position);
		}
		else
		{
			printk(KERN_WARNING "Position should be between 0 and
9\n");
		}
	}
	else
	{
		printk(KERN_WARNING "Wrong command format\nexpected:
n,m\n\tn-position\n\tm-value\n");
	}
	return length;
}
```

Funkcija kreće kopiranjem informacija iz niza buffer (korisnički prostor) u niz buff (kernel prostor) korišćenjem funkcije *copy_from_user*. Ukoliko je kopiranje uspešno izvršeno, funkcija će vratiti povratnu vrednost 0. U suprotnom slučaju vraća broj članova niza koji nije uspešno kopiran, te vraćamo error EFAULT. Kako bismo mogli koristiti niz karaktera *buff* kao string u pomoćnim funkcijama, na poslednje mesto postavljamo NULL terminator.

Od korisnika očekujemo format “vrednost,pozicija” te string možemo parsirati pomoću funkcije *sscanf* gde naznačimo da očekujemo dva *int* broja odvojena zarezom (%d,%d). Te vrednosti smeštamo u lokalne promenljive *value* i *position*. Povratna vrednost funkcije *sscanf* je broj uspešno parsiranih argumenata, te stoga proveravamo da li je povratna vrednost 2. Ukoliko je pozicija (indeks) u opsegu storage niza (0-9), vrednost value se upisuje na mesto *storage[position]*.

Za kraj je potrebno vratiti broj uspešno upisanih karaktera, što radimo pozivom funkcije “return length”.

4.2. storage_read funkcija

U slučaju *storage_read* funkcije, parametar buffer je pokazivač na adresu u korisničkom prostoru gde je potrebno smestiti podatke, dok je *length* broj bajtova koji se želi pročitati. Ukoliko se pozove komanda “*cat /dev/storage*” želimo da se ispišu svi članovi niza odvojeni *space* karakterom. Budući da cat comanda služi za isčitavanje čitavog sadržaja fajla, ona će tražiti maksimalan dozvoljeni broj karaktera na trenutnom sistemu (ARG_MAX, npr. 131072). Kada god drajver pomoću *return* komande vrati vrednost veću od nule, cat komanda će ponovo pozvati *read* funkciju jer ona očekuje još podataka. Kada drajver pomoću return komande vrati vrednost 0, to označava da se došlo do kraja fajla (EOF, end of file), te se *cat* komanda terminira.

```
ssize_t storage_read(struct file *pfile,
                     char __user *buffer,
                     size_t length,
                     loff_t *offset)
{
      int ret;
      char buff[20];
      long int len;
      if (endRead){
            endRead = 0;
            pos = 0;
            printk(KERN_INFO "Succesfully read from file\n");
            return 0;
      }
      len = scnprintf(buff, strlen(buff), "%d ", storage[pos]);
      ret = copy_to_user(buffer, buff, len);
      if(ret)
            return -EFAULT;
      pos ++;
      if (pos == 10) {
```

```
            endRead = 1;}
      return len;
}
```

Ova osobina segmentisanog isčitavanja podataka je iskorištena u *storage_read* funkciji kako bi se pri svakom pozivu cat komande poslao tačno jedan elemenat niza *storage*. Za implementaciju funkcije su nam potrebne dve celobrojne globalne promenljive: *pos* i *endRead*. Promenljiva *pos* pokazuje na element koji se treba vratiti u trenutnom pozivu cat komande. Promenljiva *endRead* signalizira da se došlo do krajnjeg elementa, te da se u sledećem pozivu cat komande vrati vrednost nula.

Pri prvom pozivu funkcije read, promenljive *endRead* i *pos* su jednake nula. Uzima se element sa indeksom *pos*=0, te se konvertuje u string *buff* funkcijom *scnprintf*. Niz *buff* (kernel prostor) se kopira u niz *buffer* (korisnički prostor) funkcijom *copy_to_user*. Iterator *pos* se inkrementira, i *read* funkcija vrati vrednost *len*. Budući da je *len* veće od nula, *cat* komanda ponovo poziva *read* funkciju. U drugom pozivu je jedina promena da je iterator *pos*=1, te se u ovoj iteraciji šalje element sa indeksom 1. Ovaj ciklus se ponavlja 10 puta - za svaki element niza. U poslednjoj iteraciji (pos=10), promenljiva *endRead* će se postaviti na 1. Kada *cat* ponovno pozove *read* funkciju, izraz if(*endRead*) će biti zadovoljen, te će se vratiti vrednost nula (return 0). Komanda *cat* ovo permutuje kao da se došlo do kraja fajla, te prestaje da poziva *read* funkciju.

Rezultat poziva je ispis u terminalu deset elemenata odvojenih space karakterom, npr:

$cat /dev/storage
0 0 0 5 0 0 0 0 0

5. Drajver za rad sa više uređaja

Kako bi se demonstrirao rad drajvera koji mora da kontroliše više uređaja, prethodni primer je modifikovan na takav način da se svakom članu *storage* niza prisupa preko jednog *node* fajla u /dev direktorijumu. Dakle, svi članovi niza imaju isti glavni (*major*) broj, ali svaki od njih ima zaseban sporedni (*minor*) broj. Na osnovu minor broja će drajver znati sa kojim elementom niza storage da komunicira. Sadržaj /dev direktorijuma bi trebalo da ima sledeće članove:

```
crw-rw-rw-  1 root   root    511,   0 Nov 20 00:31 storage0
crw-rw-rw-  1 root   root    511,   1 Nov 20 00:31 storage1
crw-rw-rw-  1 root   root    511,   2 Nov 20 00:31 storage2
crw-rw-rw-  1 root   root    511,   3 Nov 20 00:31 storage3
crw-rw-rw-  1 root   root    511,   4 Nov 20 00:31 storage4
crw-rw-rw-  1 root   root    511,   5 Nov 20 00:31 storage5
```

```
crw-rw-rw-  1 root   root   511,   6 Nov 20 00:31 storage6
crw-rw-rw-  1 root   root   511,   7 Nov 20 00:31 storage7
crw-rw-rw-  1 root   root   511,   8 Nov 20 00:31 storage8
crw-rw-rw-  1 root   root   511,   9 Nov 20 00:31 storage9
```

Ukoliko upišemo decimalnu vrednost u jedan od uređaja, npr. ***echo "7" > /dev/storage3*** , želimo da se u elemenat sa indeksom 3 upiše vrednost 7. Ukoliko pokušamo da pročitamo vrednost nekog elementa, npr. ***cat /dev/storage4***, želimo da se u terminalu ispiše vrednost elementa sa indeksom 4.

Prvi korak u implementiranju ovog zadatka je da se u *module_init* funkciji uvedu sledeće izmene:

- Rezervisati 10 sporednih (*minor*) upravljačkih brojeva umesto jednog:

```
int num_of_minors = 10;
ret = alloc_chrdev_region(&my_dev_id, 0, num_of_minors,
"storage");
```

- Napraviti 10 node fajlova prolazeći kroz for petlju i pozivajući funkciju *device_create*. Imena uređaja (*storage0*, *storage1* ...) se mogu generisati koristeći *scnprintf* funkciju, a dev_t strukture pomoću makroa (*MKDEV*, *MAJOR*, *MINOR*). Nema potrebe za višestrukim pozivanjem *class_create* funkcije jer će sve uređaje kontrolisati isti drajver, te sistematski sigurno pripadaju istoj klasi uređaja. Dakle, može se iskoristiti ista struktura my_class za svih 10 uređaja.

```
for (i = 0; i < num_of_minors; i++)
{
    printk(KERN_INFO "created nod %d\n", i);
    scnprintf(buff, 10, "storage%d", i);
    my_device = device_create(my_class, NULL,
MKDEV(MAJOR(my_dev_id), i), NULL, buff);
}
```

- Registrovati uređaje pozvajući funkciju *cdev_add* sa parametrom *count=10*

```
  ret = cdev_add(my_cdev, my_dev_id, num_of_minors);
```

Drugi korak je modifikovanje *module_exit* funkcije na sličan način - uništavanje 10 uređaja (*node* fajlova) kroz for petlju, i oslobađanje opsega od 10 upravljačih brojeva.

```
for (i = 0; i < num_of_minors; i++)
    device_destroy(my_class, MKDEV(MAJOR(my_dev_id), i));
class_destroy(my_class);

unregister_chrdev_region(my_dev_id,num_of_minors);
```

Potrebno je još modifikovati *storage_read* i *storage_write* funkcije. U obe funkcije se kao parametar prosleđuje pokazivač na **file** strukturu **pfile*. Kao što je prethodno pomenuto, jedno od polja file strukture je pokazivač na ***inode*** strukturu *f_inode*, dok inode struktura kao jedno od polja ima **dev_t** *i_rdev*. Dakle, ukoliko preko file strukture pristupimo polju *i_rdev*, imamo vrednosti upravljačkih brojeva node fajla koji je pozvao funkciju, i znamo na osnovu minor broja sa kojim uređajem (elementom niza) treba da komuniciramo. Ekstrahovanje sporednog (minor) upravljačkog broja na prethodno opisan način se zapisuje:

```
  int minor = MINOR(pfile->f_inode->i_rdev);
```

Kada smo razrešili ovu komplikaciju, *storage_read* i *storage_write* funkcije postaju jednostavnije nego u prethodnom primeru jer se u svakom trenutku radi sa tačno jednim članom niza. Pomoću promenljive *minor* indeksiramo niz *storage* i vršimo upis ili čitanje na isti način kao u prethodnom primeru. U nastavku su date definicije obe funkcije:

```
ssize_t storage_read(struct file *pfile, char __user *buffer,
size_t length, loff_t *offset)
{
  int ret;
  char buff[20];
  long int len;
  int minor = MINOR(pfile->f_inode->i_rdev);
  if (endRead){
    endRead = 0;
    printk(KERN_INFO "Succesfully read from file\n");
    return 0;
```

```
  }
  len = scnprintf(buff, strlen(buff), "%d\n", storage[minor]);
  ret = copy_to_user(buffer, buff, len);
  if(ret)
    return -EFAULT;
  endRead = 1;
  return len;
}

ssize_t storage_write(struct file *pfile, const char __user
*buffer, size_t length, loff_t *offset)
{
  char buff[20];
  int value;
  int ret;
  int minor = MINOR(pfile->f_inode->i_rdev);
  ret = copy_from_user(buff, buffer, length);
  if(ret)
    return -EFAULT;
  buff[length-1] = '\0';

  ret = sscanf(buff,"%d",&value);
  storage[minor] = value;
  printk(KERN_INFO "Succesfully wrote value %d at position %d\n",
value, minor);
  return length;
}
```

Dodatak A : Pravilno korišćenje error enumeracije u Linux-u

Kako bi se lako i na sistematičan način pronašle greške u Linux-u, definisani su makroi za sve tipične greške koje su razlog neuspešnog poziva (izvršavanja) funkcija. U kernelu je potrebno uključiti <linux/errno.h> ,a u korisničkom prostoru <errno.h>. Na ovaj način umesto signaliziranja da se samo desila greška, možemo naznačiti koja greška je dovela do neuspešnog izvršavanja funkcije. Ukoliko pozivamo neku funkciju, te ona vrati negativnu vrednost, dobra praksa je da tu vrednost sačuvamo u lokalnoj promenljivoj (npr. *int ret*), te da error poruku propagiramo nivo iznad (*return ret*). Time će se vrednost erora propagirati iz funkcije čije ga je izvršavanje prouzrokovalo, sve do originalne funkcije (na najvišem nivou).

Međutim, neke funkcije ne vraćaju broj erora, te je u tom slučaju potrebno pronaći dokumentaciju te funkcije za savete. Na primer, prethodno pomenute funkcije *copy_to_user* i *copy_from_user* vraćaju broj podataka koji je neuspešno kopiran - što je pozitivan broj. U njihovoj dokumentaciji je naznačeno da je u slučaju neuspešnog kopiranja potrebno vratiti error EFAULT. Ovaj makro je definisan u errno.h fajlu i signalizira da je prosleđena adresa nevalidna. Za slučaj ovih funkcija bi pravilno opsluživanje greške izgledalo kao u nastavku:

```
ret = copy_from_user(buff, buffer, length);
if(ret)
   return - EFAULT;
```

Bitno je ne zaboraviti negativan predznak pre definisanog eror makroa!

U korisničkom prostoru u <string.h> postoji funkcja:

char ***strerror**(int errnum);

Ova funkcija kao parametar prima broj erora, a kao povratnu vrednost vraća string koji opisuje značenje tog eror broja. U kombinaciji sa standardnim funkcijama za ispis teksta u terminal (printf), može predstavljati moćan alat za pronalaženje uzroka greške.

Svi definisani eror makroi se mogu pogledati na sledećem linku:
http://man7.org/linux/man-pages/man3/errno.3.html