Modularnost u Linux OS-u

dr Predrag Teodorovic

Fakultet Tehničkih Nauka, Novi Sad

November 1, 2019

Uvod

- Ovo poglavlje obuhvata osnovne koncepte o modulima i kernel programiranju
- Videćemo kako se prave i pokreću kompletni moduli i osvrnuti se na osnovni kod koji svi moduli dele
- Biće prikazani primeri modula koji demonstriraju osnovne principe programiranja
- Pravljenje, učitavanje i modifikacija ovih primera je dobra osnova za razumevanje načina rada drajvera i njihove povezanosti sa kernelom

Zašto modularnost?

- Jedna od dobrih karakteristika Linux-a je mogućnost da se proširi set funkcija koje nudi kernel i to za vreme njegovog rada
- To znači da možemo dodati određene funkcionalnosti kernela (ili oduzeti ako treba) dok je sistem podignut i radi
- Svaki deo koda koji može biti dodat kernelu u vreme rada naziva se **modul**
- Svaki modul je sastavljen od objektnog koda koji može biti dinamički povezan sa kernelom u vreme rada pomoću *insmod* programa ili isključen iz kernela pomoću *rmmod* programa.

Hello World module (a koji drugi???)

```
#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);
```

Šta je šta?

- Ovaj jednostavan modul definiše dve funkcije: jednu koja se poziva kada se modul učitava u kernel (*hello_init*) i drugu koja se poziva kada se modul isključuje iz kernela (*hello_exit*)
- *module_init* i *module_exit* su C makroi koji omogućavaju programeru da definiše funkcije koje će se pozivati prilikom učitavanja modula u kernel i isključivanja modula iz kernela
- Još jedan specijalni makro, MODULE_LICENCE, se koristi da eksplicitno naglasi kernelu da ovaj modul ima slobodnu licencu - bez ovakve deklaracije, kernel bi dao upozorenje kada bi se modul učitao

printk i prioriteti poruka

- Funkcija *printk* je definisana u Linux-ovom kernelu i dostupna je modulima
- Ona se ponaša slično kao standardna funkcija iz C biblioteke printf
- Kernelu treba svoja funkcija za ispis jer sam radi, bez pomoći C biblioteka
- String KERN_ALERT se odnosi na prioritet poruke koja će se ispisivati pomoću funkcije *printk*
- Odredili smo visok prioritet jer se poruka sa podrazumevanim (default) prioritetom možda ne bi pojavila nigde gde je korisno, što zavisi od kernel verzije i konfiguracije
- Modul se može testirati pomoću *insmod* i *rmmode* alata, kao što je prikazano. Samo super korisnik može učitati i odstraniti modul

Kompajliranje i pokretanje

```
% make
  make[1]: Entering directory `/usr/src/linux-2.6.10'
  CC [M] /home/ldd3/src/misc-modules/hello.o
  Building modules, stage 2.
  MODPOST
  CC /home/ldd3/src/misc-modules/hello.mod.o
  LD [M] /home/ldd3/src/misc-modules/hello.ko
  make[1]: Leaving directory `/usr/src/linux-2.6.10'
% su
root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

Šta je neophodno da bi ovo radilo?

- Da bi ova sekvenca komandi mogla da se izvrši moramo imati propisno konfigurisano i podešeno kernel stablo na mestu gde izvorna biblioteka koja se kompajlira (hello.c) može da ga pronađe (/usr/src/linux-2.6.10)
- Samo pisanje modula nije toliko teško, najteži deo je zapravo predstavlja razumevanje periferije (hardverske jedinice) za koju se modul razvija i poznavanje načina da se maksimiziraju njene performanse

Koja je razlika kernel modula i aplikacije?

- Dok većina malih i srednjih aplikacija izvode jedan zadatak od početka do kraja, svaki kernel modul se samo registruje u cilju da služi budućim zahtevima, a njegova funkcija inicijalizacije se završava odmah
- Drugim rečima, zadatak funkcije inicijalizacije modula je da pripremi modul za kasnije pozivanje njegovih funkcija
- To je kao da modul može da kaže: “ Ja sam ovde i ovo je spisak funkcionalnosti koje mogu da izvršim.”
- Izlazna funkcija modula, tj. ona koja je definisana makroom *module_exit* se poziva tačno pre nego što se modul isključi iz kernela
- Ona treba da kaže kernelu: “Ja nisam više tamo, nemoj tražiti od mene da radim bilo šta više.”

Koja je razlika kernel modula i aplikacije?

- Ovaj način pristupa programiranju je sličan *event-driven* programiranju i može se reći da je svaki modul *event- driven* dok postoje aplikacije koje to nisu
- Druga važna razlika između *event- driven* aplikacija i kernel koda je u izlaznoj funkciji
- Aplikacije koje se isključuju mogu biti lenje u oslobađanju resursa ili jednostavno izbegavaju potpuno uklanjanje, dok izlazne funkcije modula moraju vrlo pažljivo da vrate sve resurse koje je *module_init* funkcija zauzela ili će delovi ostati neupotrebljivi sve dok se sistem ne resetuje
- Mogućnost odstranjivanja modula je jedna od karakteristika modularizacije koja se jako ceni, jer pomaže da se smanji vreme razvoja usled činjenice da možete testirati verzije novog drajvera bez prolaska kroz dugotrajan proces isključivanja i resetovanja sistema svaki put

Koja je razlika kernel modula i aplikacije?

- Aplikacija može pozivati funkcije koje nisu u njoj definisane: može biti povezana preko spoljnih referenci i tako koristiti odgovarajuće biblioteke funkcija
- Printf je jedna od takvih funkcija i definisana je u *libc* biblioteci
- Modul je, s druge strane, povezan samo sa kernelom i jedine funkcije koje može pozvati su one koje su definisane u kernelu; ne postoje biblioteke sa kojima se može povezati
- *Printk* funkcija korišćena u hello.c je verzija funkcije *printf* definisane u kernelu koja se daje na raspolaganje modulima
- Ponaša se slično kao orginalna funkcija, sa par manjih razlika, a najveća razlika je što ne podržava floating-point aritmetiku

Koja je razlika kernel modula i aplikacije?

- Još jedna važna razlika između kernel programiranja i programiranja aplikacija je način kako svaka od ovih okolina reaguje na greške
- Kod nekih grešaka koje nisu značajne prilikom razvoja aplikacije debager nam može uvek koristiti da nađemo grešku u sors kodu, dok greške u kernelu najčešće automatski terminiraju trenutni proces ako ne i čitav sistem
- Možda i najvažnija razlika je ostavljena za kraj: Modul se izvršava u kernel prostoru dok se aplikacije izvršavaju u korisničkom prostoru. Ovaj koncept je u osnovi teorije operativnih sistema.

Kompajliranje i učitavanje modula

- "Hello world" primer je ukratko demonstrirao pravljenje i učitavanje modula
- Način pravljenja modula se značajno razlikuje od onog koji se koristi u korisničkom prostoru za aplikacije
- Postoje određene pripreme koje treba izvršiti pre pravljenja kernel modula
- Prvo je da se osiguramo da imamo dovoljno dobru verziju kompajlera, alata za pravljenje modula, i ostalih potrebnih alata
- Datoteka *Documentations/Changes* u kernel dokumentaciji uvek sadrži listu potrebnih verzija alata

Kompajliranje i učitavanje modula

- Verzija kompajlera koja je previše nova može praviti jednako problema kao i ona koja je previše stara
- Dalje je potrebno napraviti i konfigurisati kernel stablo. Ne mogu se praviti moduli koji se učitavaju za 2.6 kernel ukoliko nemate konfigurisano kernel stablo
- Kada se sve to podesi na redu je pravljenje mejk-fajla (make file) za module. Za primer »hello world« dovoljna je jedna linija:

```
obj -m := hello.o
```

Kompajliranje i učitavanje modula

- Sve ostalo obavlja kernel sistem
- Gornja linija označava da će se napraviti jedan modul od objektnog fajla *hello.o*, koji se dobija kompajliranjem izvornog fajla *hello.c*
- Rezultujući modul koji je napravljen iz objektog fajla se naziva *hello.ko*
- Ako imamo modul koji se naziva *module.ko* koji je generisan iz dva izvorna fajla (recimo *file1.c* i *file2.c*), pisali bi:

```
obj -m := module.o
module-objs := file1.o file2.o
```

Makefile

- Da bi *makefile* kao ovaj radio, on mora biti pozvan u kontekstu većeg kernel sistema
- Ako se naše *kernel source* stablo nalazi recimo u ~/kernel-2.6 direktorijumu, *make* komanda koja je potrebna da bi se napravio modul bila bi:

```
make -C /root/linux-xlnx-zynqmp-dt-fixes-for-4.10 M='pwd
```

Makefile

- Ova komanda počinje menjajući direktorijum u onaj koji je obezbeđen –C opcijom (to je izvorni kernelov direktorijum)
- Tamo se nalazi kernelov *makefile* najvišeg nivoa
- Opcija M= se koristi da bi se *makefile* vratio nazad u *modul source* direktorijum pre nego što se napravi modul
- Modul koji se kreira se odnosi na listu objektnih fajlova koji se nalaze u **obj –m** promenljivoj (koju smo podesili u *module.o* u našem primeru)

Komplikacija?

- Kucanje prethodne *make* komande može postati zamarajuće za one koji prave module izvan kernel stabla
- Iz tog razloga se kreiranje modula radi korišćenjem *makefile*-a koji izgleda ovako:

```
# If KERNELRELEASE is defined, we've been invoked from
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else
KERNELDIR ?= /root/linux-xlnx-zynqmp-dt-fixes-for-4.10
PWD := $(shell pwd)
default:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

Kako ovo radi?

- Ovde možemo videti na delu produženu GNU make sintaksu
- Kada se *makefile* pozove iz komandne linije primetiće se da KERNELRELEASE promenljiva nije setovana
- Ona nalazi *kernel source* direktorijum koristeći činjenicu da simbolički link build u instaliranom direktorijumu modula pokazuje putanju do kernel stabla
- Kada je *kernel source* stablo nađeno *makefile* poziva automatski liniju koja vrši drugu *make* komandu da bi pozvao kernel sistem za pravljenje modula koji je već opisan
- Prilikom drugog čitanja, *makefile* podešava *obj-m*, a kernel *makefile* praktično kreira modul

Učitavanje i uklanjanje modula

- Sledeći korak pošto se napravi modul je učitavanje modula u kernel korišćenjem insmod programa
- Ovaj program učitava kod modula i podatke u kernel koji zauzvrat izvršava funkciju u kojoj povezuje bilo koji nerešeni simbol u modulu sa tabelom simbola kernela
- Za razliku od linkera kernel ne modifikuje kernelov disk fajl nego ga kopira u memoriju
- Insmod prihvata dodatne argumente iz komandne linije i može dodeliti vrednosti parametrima iz našeg modula pre nego što ga poveže sa kernelom
- Ako je modul pravilno urađen može se konfigurisati u vreme učitavanja - konfiguracija za vreme učitavanja daje korisniku više fleksibilnosti nego konfiguracija za vreme kompajliranja, koja se i dalje ponekad koristi (videćemo na narednim slajdovima kako se ovo koristi)

Šta se dešava u pozadini?

- Interesantno je videti kako kernel podržava insmod
- Kernel se oslanja na sistemski poziv koji je definisan u izvornoj datoteci kernel/module.c
- Funkcija sys_init_module priprema kernel memoriju za modul, zatim kopira sadržaj modula u taj deo memorije, rešava kernel reference u tom modulu preko kernelove tabele simbola i na kraju poziva modulovu funkciju inicijalizacije da sve pokrene

Alternativa za insmod

- Alat *modprobe* takođe učitava module u kernel
- Razlika je u tome da *modprobe* pogleda module koji treba da se učitaju i pogleda da li referenciraju bilo koji simbol koji trenutno nije definisan u kernelu
- Ako su takve reference nađene *modprobe* potraži da li su ti simboli definisani u drugim modulima
- Kada pronađe takve module i njih učita u kernel
- Ako se u ovakvoj situaciji koristi *insmod* komanda neće uspeti i prijaviće se greška »nerazrešeni simbol« u sistemskom logfajlu

Uklanjanje i pregled učitanih modula

- Moduli mogu biti odstranjeni iz kernela pomoću *rmmod* alata
- Odstranjivanje modula nije moguće jedino ako je modul još u upotrebi ili ako je kernel konfigurisan tako da ne dozvoljava odstranjivanje modula
- Moguće je konfigurisati kernel da dozvoljava odstranjivanje modula čak i kada su u upotrebi ali će tada najverovatnije biti potreban reset sistema
- *Lsmod* program pravi listu modula koji su trenutno učitani u kernel
- Takođe imamo informaciju ukoliko neki od modula koriste jedni druge
- Informacija o trenutno učitanim modulima može se naći i u *sysfs* virtualnom fajlsistemu na putanji /sys/module

Kernelova tabela simbola

- Ova tabela sadrži adrese globalnih kernelovih promenljivih i funkcija koje su potrebne za implementaciju modularizovanih drajvera
- Kada se modul učita svaki simbol koji je dat na raspolaganje od strane modula postaje deo kernelove tabele simbola
- Najčešće modul implementira svoju funkcionalnost bez potrebe da eksportuje simbole uopšte, a potreba za eksportovanjem simbola se javlja kada postoji potreba da ih drugi moduli koriste
- Novi moduli mogu koristiti simbole koje eksportuje naš modul i tako možemo nadovezati te module na naš modul
- Nadovezivanje modula (tzv. *module stacking*) je implementirano u glavnim kernel sorsovima, takođe
- Naprimer, svaki modul za USB uređaj se nadovezuje na *usbcore* i *input* module

Eksportovanje simbola

- Eksportovanje simbola se vrši pomoću sledećih makroa:

```
EXPORT_SYMBOL(name);
EXPORT_SYMBOL_GPL(name);
```

- Oba makroa daju na raspolaganje date simbole izvan modula, ali _GPL verzija daje na raspolaganje simbole samo GPL (general public licence) licenciranim modulima
- Simboli moraju biti eksportovani u globalnom delu fajla modula izvan svake funkcije jer makroi proširuju deklaraciju promenljive sa posebnom upotrebom za koju se očekuje da joj se može pristupiti globalno
- Ova promenljiva se smešta u specijalnom delu modula (ELF sekcija) koju koristi kernel u trenutku učitavanja da bi našao promenljive koje eksportuje modul

Instrukcije za pravljenje modula

- Svaki kod modula mora sadržati sledeće linije:

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

- Module.h sadrži veliki broj definicija simbola i funkcija koji su najčešće neophodni modulima koji se učitavaju
- Init.h je potreban za inicijalizacionu i funkciju čišćenja
- Većina modula takođe uključuje i moduleparam.h da bi se modulu omogućio pristup parametrima u vreme učitavanja (više reči o tome uskoro)

Inicijalizacija modula

- Funkcija za inicijalizaciju modula registruje određene blokove koje nudi modul
- Pod blokovima podrazumevamo nove funkcionalnosti bez obzira da li je to čitav drajver ili nova softverska apstrakcija kojoj može pristupiti aplikacija
- Definicija funkcije za inicijalizaciju uvek izgleda ovako:

```
static int __init initialization_function(void)
{
    /* Initialization code here */
}
module_init(initialization_function);
```

init prefiks

- Funkcije za inicijalizaciju treba da se deklarišu kao *static* jer ne treba da se vide izvan određenog fajla
- Prefiks __init daje kernelu do znanja da se data funkcija koristi samo u vreme inicijalizacije
- Pošto se modul učita modul loader uklanja funkciju za inicijalizaciju i memoriju koja bi bila korišćenja za njeno skladištenje daje na raspolaganje u druge svrhe
- Postoji i sličan tag za podatke (__*initdata*) koji se koriste prilikom inicijalizacije
- Upotreba __*init* i __*initdata* je opciona, ali se isplati
- Naravno treba ih koristiti samo ako smo sigurni da te funkcije ili podatke nećemo koristiti posle inicijalizacije

module_init makro

- Korišćenje *module_init* makroa je obavezno
- Ovaj makro dodaje poseban deo objektnom kodu modula koji daje informaciju o tome gde se nalazi funkcija za inicijalizaciju modula (prethodno smo rekli da se ova funkcija poziva kao poslednja operacija prilikom pozivanja *insmod* programa)
- Bez ove definicije naša funkcija za inicijalizaciju neće nikada biti pozvana

Uklanjanje modula

- Svaki netrivijalni modul zahteva i funkciju za uklanjanje iz kernela koja raskida interfejs i vraća sve resurse sistemu pre nego što se modul odstrani
- Ova funkcija se definiše kao:

```
static void __exit cleanup_function(void)
{
    /* Cleanup code here */
}
module_exit(cleanup_function);
```

__exit prefiks

- Prefiks __*exit* označava kod koji je isključivo za odstranjivanje modula, označavajući kompajleru da ga smesti u posebnu ELF sekciju
- Ako je naš modul napravljen direkno u kernelu ili ako je kernel konfigurisan tako da ne dozvoljava odstranjivanje modula funkcije označene sa __*exit* su prosto zanemarene
- Iz tog razloga funkcije koje su označene kao __*exit* mogu biti pozvane samo u vreme odstranjivanja modula ili gašenja sistema
- Bilo kakva druga upotreba bila bi greška
- Još jednom, *module_exit* deklaracija je potrebna da bi omogućila kernelu da pronađe našu funkciju za uklanjanje iz kernela
- Ako u našem modulu ne definišemo funkciju za uklanjanje, kernel neće dozvoliti modulu ni da se učita

Tipične greške prilikom inicijalizacije

- Jedna stvar koju moramo uvek imati na umu kada registrujemo nove funkcionalnosti u kernelu je da registracija uvek može biti neuspešna
- Čak i najjednostavnije akcije često zahtevaju rezervisanje memorije, a zahtevana memorija nije uvek dostupna
- Usled toga, kod modula mora uvek proveravati povratne vrednosti, i imati potvrdu da su se tražene operacije zapravo izvršile
- Ako se desi bilo koja greška prilikom registracije nove funkcionalnosti prvo što se uradi je odlučivanje da li modul može da nastavi inicijalizaciju sam
- Često modul može da nastavi da radi i posle neuspešne registracije, sa degradiranim funkcionalnostima ako je potrebno
- Kad god je moguće modul treba da nastavi dalje i obezbedi što je više mogućnosti

Tipične greške prilikom inicijalizacije

- Ako se ispostavi da modul ne može da se učita posle greške određene vrste, moramo vratiti sve aktivnosti vezane za registraciju koje su urađene pre greške
- Linux ne drži registre za module ili nove funkcionalnosti koje su registrovane pa modul mora sam sve resurse vratiti ako inicijalizacija u nekoj tački ne uspe
- Ako ne uspemo da vratimo ono što smo promenili prilikom pokušaja registracije, kernel će biti ostavljen u nestabilnom stanju - on sadrži interne pokazivače na kod koji više ne postoji
- U ovakvim situacijama jedino rešenje je obično resetovanje sistema
- Zato treba uraditi pravu stvar kada se desi greška pri inicijalizaciji

goto naredba?

- Probleme oporavka od grešaka je nekada najbolje rešiti naredbom *goto*
- Pažljiva upotreba naredbe *goto* u situacijama sa greškama može eliminisati veliki deo komlikovane visoko osetljive strukturne logike
- Kod na narednom slajdu ponaša se korektno ako se u inicijalizaciji javi greška u bilo kom trenutku
- Ovaj kod pokušava da registruje tri nove funkcionalnosti
- Naredba goto se koristi u slučaju greške tako što odstranjuje samo one funkcionalnosti koje su bile uspešno registrovane pre nego što se desila greška

Primer vraćanja registrovanih resursa sistemu

```
int __init my_init_function(void)
{
    int err;
    /* registration takes a pointer and a name */
    err = register_this(ptr1, "skull");
    if (err)
        goto fail_this;
    err = register_that(ptr2, "skull");
    //nastavak na sledećem slajdu
```

Primer vraćanja registrovanih resursa sistemu

```
//nastavak od prethodnog slajda
    if (err)
        goto fail_that;
    err = register_those(ptr3, "skull");
    if (err)
        goto fail_those;
    return 0; /* success */
    fail_those: unregister_that(ptr2, "skull");
    fail_that: unregister_this(ptr1, "skull");
    fail_this: return err; /* propagate the error */
}
```

Tipične greške prilikom inicijalizacije

- Drugi način da se reši ovaj problem bez korišćenja *goto* naredbe je da vodimo računa o tome šta se uspešno registrovalo i pozivamo funkciju za uklanjanje iz našeg modula u slučaju greške
- Funkcija za uklanjanje vraća samo one korake koji su uspješno odrađeni pre nego što se desila greška
- Međutim ova alternativa zahteva više koda i CPU vremena
- Povratna vrednost od *my_init_function*, *error*, je kod greške
- U Linux-ovom kernelu kodovi greške su negativni brojevi koji pripadaju setu definisanom u <linux/errno.h>

Vraćanje resursa u funkciji za uklanjanje

- Očigledno, modulova funkcija za čišćenje mora vratititi sve registracije koje je izvršila funkcija za inicijalizaciju i uobičajno je (ali ne i obavezno) da odstranjuje funkcionalnosti u obrnutom redosledu od kojeg su registrovane

```
void __exit my_cleanup_function(void)
{
    unregister_those(ptr3, "skull");
    unregister_that(ptr2, "skull");
    unregister_this(ptr1, "skull");
    return;
}
```

Minimizacija dupliranja koda

- Da bi se minimizovalo dupliciranje koda funkcija za uklanjanje se poziva u inicijalizaciji kad god se desi greška
- Ova funkcija tada mora da proveri status svakog dela modula pre nego što ga odstrani
- U najjednostavnijem obliku kod izgleda ovako:

Minimizacija dupliranja koda

```
struct something *item1;
struct somethingelse *item2;
int stuff_ok;
void my_cleanup(void)
{
    if (item1)
        release_thing(item1);
    if (item2)
        release_thing2(item2);
    if (stuff_ok)
        unregister_stuff( );
    return;
}
```

Minimizacija dupliranja koda

```
int __init my_init(void)
{
    int err = -ENOMEM;
    item1 = allocate_thing(arguments);
    item2 = allocate_thing2(arguments2);
    if (!item2 || !item2)
        goto fail;
    err = register_stuff(item1, item2);
    if (!err)
        stuff_ok = 1;
    else
        goto fail;
    return 0; /* success */
    fail: my_cleanup( );
    return err;
}
```

Komentar gornjeg koda

- Kao što je pokazano u kodu eksterni flegovi prilikom beleženja uspešnosti inicijalizacionog koraka nam mogu zatrebati ili ne što zavisi od semantike funkcije koja se poziva za registraciju
- Bez obzira da li su flegovi potrebni ili ne ovaj način inicijalizacije je skaliran dobro i često je bolji od tehnike koja je prikazana ranije
- Treba primetiti da se funkcija za uklanjanje ne može označiti kao __*exit* kada se poziva u neizlaznom kodu

Parametrizovani moduli

- Više parametara koje drajver treba da zna se mogu promeniti od sistema do sistema
- Ako naš drajver kontroliše stari hardver on mora eksplicitno da kaže gdje se nalaze I/O portovi drajvera ili I/O adrese memorije
- Kernel podržava ove potrebe omogućavajući drajveru da naznači parametre za koje postoji mogućnost da su promenjeni kada se modul drajvera učita
- Ove vrednosti parametara mogu biti dodeljene u vreme učitavanja pomoću *insmod* ili *modprobe*
- Ove komande prihvataju specifikacije više tipova vrednosti u komandnoj liniji

Parametrizovani moduli

- Kao način demonstracije ove mogućnosti uzećemo primer “hello world” modula
- Dodaćemo mu dva parametra: celobrojnu vrednost koja se naziva *howmany* i string koji se naziva *whom*
- Naš modul koji do sada nije imao nikakvu funkcionalnost sad prilikom učitavanja pozdravlja *whom* ne samo jednom već *howmany* puta
- Takav modul treba onda učitati sa komandne linije na sledeći način:

```
insmod hellop.ko howmany=10 whom="Mom"
```

Parametrizovani moduli

- Na taj način kada se modul učita ispisaće poruku »Hello, Mom« deset puta
- Međutim pre nego što *insmod* promeni parametre modula, modul ih mora staviti na raspolaganje
- Parametri se deklarišu sa *module_param* makroom, koji je definisan u moduleparam.h
- *Module_param* uzima tri parametra: ime promenljive, njen tip i masku za dozvolu koja se koristi da bi se postigao sysfs pristup
- Makro treba da bude napisan izvan bilo koje funkcije i najčešće se nalazi blizu zaglavlja izvornog fajla
- Za prethodni primer to bi izgledalo ovako:

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

Argumenti module_param makroa

- Numerički tipovi koji su podržani za parametre modula su:
  - bool,
  - invbool (true -> false i obrnuto),
  - charp (pokazivač na char),
  - int,
  - long,
  - short,
  - uint,
  - ulong,
  - ushort.

Nizovi parametara

- Nizovi parametara su takođe podržani
- Da bi deklarisali niz koristimo:

```
module_param_array(name,type,num,perm);
```

- *Name* je ime niza tj. parametra, *type* je tip elementa niza, *num* je celobrojna vrednost a *perm* je uobičajna vrednost dozvole
- Ako se ovakav parametar postavlja u vreme učitavanja, *num* je postavljeno na broj vrednosti koje učitavamo
- Deo zadužen za učitavanje modula odbija da prihvati više vrednosti nego što može da stane u niz

Postavljanje dozvole pristupa paremetrima kroz SYSFS

- Poslednje polje u *module_param* je vrednost dozvole
- Trebali bi da koristimo definicije koje se nalaze u <linux/stat.h>
- Ova vrednost kontroliše ko može pristupiti reprezentaciji parametara modula u okviru *sysfs*
- Ako je perm podešeno na 0, nema ulaska *sysfs* uopšte, u suprotnom parametar se pojavljuje pod /sys/module sa datim dozvolama
- Koristimo S_IRUGO da bi dozvolili čitanje parametra svima ali ne i njegovo menjanje dok S_IRUGO|S_IWUSR dozvoljava menjanje parametara
- Treba primetiti da ukoliko je parametar promenjen od strane *sysfs* vrednost parametra se vidi po promenama u našem modulu ali modul nije obavešten na bilo koji drugi način
- U ovom slučaju, najbolje je onemogućiti menjanje parametara osim ako nismo spremni da detektujemo promenu i reagujemo u skladu sa njom

Drajveri u korisničkom prostoru

- Prednosti drajvera koji se pišu u korisničkom prostoru:
  - Čitave C biblioteke se mogu dodati
  - Drajver može obavljati mnoge egzotične zadatke bez pomoći eksternih programa
  - Programer može koristiti konvencionalni debager za drajver kod bez potrebe da prolazi kroz poteškoće prilikom debagovanja kernela koji radi
  - Ukoliko drajver koji je pisan u korisničkom prostoru ima rupu u sebi, možemo ga prosto odstraniti. Problemi sa drajverom će vrlo retko dovesti do pada čitavog sistema, osim ako se hardver koji je kontolisan njime jako loše ponaša;
  - (nastavak na sledećem slajdu)

Drajveri u korisničkom prostoru

- (nastavak sa prethodnog slajda)
  - Korisnička memorija je zamenljiva (omogućava virtualizaciju), za razliku od kernelove. Periferija koja se ne koristi često sa ogromnim drajverom neće okupirati RAM koji bi drugi programi mogli da koriste, osim kad je stvarno u upotrebi;
  - Dobro napravljeni drajveri mogu kao i drajveri u kernelu da dozvole konkurentni pristup periferiji;
  - Ako moramo da pišemo *closed-sorce* drajver, opcije u korisničkom prostoru nam olakšavaju da izbegnemo zahtevne situacije licenciranja i probleme sa menjanjem kernel interfejsa

Drajveri u korisničkom prostoru

- Obično onaj ko piše drajvere u korisničkom prostoru implementira server proces, koji od kernela preuzima zadatak da bude jedini zadužen za kontrolu hardvera
- Korisničke aplikacije se tako mogu konektovati na server da izvrše stvarnu komunikaciju sa periferijom
- Usled toga, pametni drajver procesi dozvoljavaju konkurentan pristup periferiji
- Upravo na ovaj način radi X server

Drajveri u korisničkom prostoru

- Međutim pristup pisanja drajvera u korisničkom prostoru ima veliki broj mana. Najvažnije su:
  - Prekidi nisu dostupni u korisničkom prostoru
  - Direktan pristup memoriji je moguć samo pomoću mmapping-a /dev/mem, i samo privilegovani korisnik može ovo da radi
  - Pristup I/O portovima je moguć samo pozivanjem *ioperm* i *iopl*. Štaviše, ne podržavaju sve platforme ove sistemske pozive a pristup /dev/port-u može biti previše spor da bi bio efikasan. I sistemski pozivi i fajlovi periferija su rezervisani samo za privilegovanog korisnika
  - (nastavak na sledećem slajdu)

Drajveri u korisničkom prostoru

- (Nastavak sa prethodnog slajda)
  - Vreme odziva je veće (sporiji odziv), jer je potrebna zamena konteksta (*context switch*) da bi se prenosile informacije ili akcije između klijenta i hardvera. Još gori slučaj je ako drajver nema fiksno mesto u memoriji na disku - vreme odziva je neprihvatljivo dugačko. Korišćenje *mlock* sistemskog poziva može pomoći ali obično moramo »zaključati« veliki deo memorije, jer se program pisan u korisničkom prostoru oslanja izuzetno mnogo na brojne biblioteke. *Mlock*, takođe, može koristiti samo privilegovani korisnik
  - Drajveri za najvažnije uređaje ne mogu biti napisani u korisničkom prostoru. Neki od njih su mrežni interfejs i blok uređaji.