CS225 Projektni zadatak

Jesenji semetar, 2021/22

Predmet: **CS225: Operativni sistemi**

Profesor: **Nemanja Zdravković**

Asistent: **Bojana Tomašević Dražić**

Ime i prezime: **Nikola Tasić**

Broj indeksa: **3698**

Datum izrade: **20.01.2022.**

Kernel programiranje

1.1 Šta je kernel modul/driver?

Šta je tačno kernel modul? Moduli su delovi koda koji mogu da se dinamički loadu-uju i unload-uju po potrebi. Oni proširiju funkcionalnost Linux kernela bez potrebe za restartovanjem(reboot) sistema. Na primer jedan tip kernel modula je driverski program. On omogućava kernelu da komunicira sa hardverom koji je povezan na sistem. Bez ove mogućnosti morali bi da imamo monolitni kernel i da nove funkcionalnosti dodajemo direktno u sam kernel. Ovo ima implikacije da bismo morali da rekompajliramo kernel i reboot-ujemo sistem svaki put kada dodajemo novu funkcionalnost.

1.2 Kako se moduli dodaju u kernel

Svi trenutno aktivni moduli mogu biti izlistani korišćenjem `lsmod` komande koja informacije dobavlja koristeći fajl `/proc/modules`.

Kako se moduli dodaju u kernel? Kada funkcionalnost nije pristuna u kernelu, daemon za kernel module `kmod(1)` poziva `modprobe` da učita modul. `modprobe` prima tekstualni argument u jednom od dva oblika:

- Ime modula - `nvidia`, `vboxdrv`, `uvcvideo`
- Generički identifikator `char-major-10-30`

Ako se `modprobe`-u prosledi generički parameter on prvo potraži alias za taj parametar u fajli `/etc/modprobe.conf(2)`:

```
alias char-major-10-30 bluetooth
```

i tako dobije informaciju da je `char-major-10-30` predstavlja modul `bluetooth.ko`.

Dalje `modprobe` pretražuje `/lub/modules/version/modules.dep` da proveri ostale module koji bi možda trebali biti učitani pre zahtevanog modula. Ovaj fajl je kreiran od strane `depmod -a` i sadrži zavisnosti modula. Na primer `msdos.ko` zahteva `fat.ko` modul koji je već učitan u kernel. Zahtevani

modul ima zavisnost ka drugom modulu ako drugi modul definiše simbole (funkcije ili promenljive) koje taj modul koristi.

Na kraju, `modprobe` koristi `insmod` da učita zavisnosti modula u kernel, a tek onda zahtevani modul. `insmod` je dizajniran tako da ne bude previše inteligentan što se tiče lokacije modula, dok je `modprobe` zadužen za poznavanje lokacija i i razrešavanje zavisnosti u određenom redosledu. Tako da, na primer, kada bi hteli da učitamo modul `msdos` morali bi da pokrenemo komande:

```
insmod /lib/modules/$(uname -r)/kernel/fs/fat/fat.ko
insmod /lib/modules/$(uname -r)/kernel/fs/fat/msdos.ko

# ili samo
modprobe msdos
```

Kao što smo videli `insmod` zahteva punu putanju do fajla i tačan redosled dodavanja modula, dok `modprobe` samo prihvata ime, bez bilo koje ekstenzije, i sam pronalazi sve što treba da zna parsirajući informacije iz `/lib/modules/version/modules.dep`

Linux distribucije pružaju `modprobe`, `insmod` i `depmod` kao paket uglavnom nazvan `module-init-tools` (naravno to se razlikuje od distribucije do distribucije). Veoma starijim verzijama taj paket se mogao na naći pod nazivom `modutils`.

2.1 Hello world 1: najprostiji modul

Kao što sam više puta imao prilike da primetim sva uputstva koja imaju bilo kakve veze sa programiranjem često počinju čuvenim programom ”Hello world” tako da ni ovaj tekst neće biti ni malo druagičiji. Proćićemo kroz osnove kreiranja kernel modula.

Sledi najprostiji kernel modul:

```c
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void) {
    printk(KERN_INFO "Hello world 1\n");

    // ne-nula povratna vrednost označava
    // da modul nije moguće bilo učitati
    return 0;
}

void cleanup_module(void) {
    printk(KERN_INFO "Goodbye world 1\n");
}
```

Kernel moduli imaju najmanje dve funkcije: ’start’ funkciju koja se zove `init_module()` koja se poziva kada se modul doda u kernel, i ’end’ koja se zove `cleanup_module()` koja se poziva tik pre nego što se modul unload-uje iz kernela (`rmmod`). To je naravno stariji način struktuirannja modula. Moderni linux kerneli provajduju C makro-e koji omogućavaju da funkcije nazovemo kako god želimo a označimo kao ’start’ i ’end’. Međutim mnogi i dalje svoje metode nazivaju baš onako kako se ranije to radilo.

Uglavnom, `init_module()` ili registruje neki `handler` za nešto u vezi sa kernelom, ili zamenjuje neku funkcionalnost kernela sa sopstvenim kodom. U drugom primeru česta praksa je da se prvo izvši neki kod a zatim pozove originalna funkcija. Time dobijamo neku vrstu proširenja originalne funkcionalnosti kernela. S druge strane `cleanup_module()` ima inverzan zadatak u odnosu na `init_module()` tako da se modul može unload-ovati bezbedno.

Na kraju svaki kernel modul mora da `#include`-uje `linux/module.h`. U primeru smo morali da uvezemo i `linux/kernel.h` koji nije neophodan ali nam pruža `printk()` funkciju i `KERN_*` makroe.

2.1.1. printk

Bez obzira na to što potpis metode `printk()` i njena semantika podsećaju na `printf()` funkciju, njena uloga nije da prenosi informacije do korisnika. Ona ima ulogu kao loging mehanizam za kernel i koristi se da loguje informacije i pruža upozorenja. S obzirom na to da se koristi kao logging mehanizam svaki poziv može da ima prioritet. Postoji 8 nivoa prioriteta i makro za svakog od njih. Ako ne specificiramo prioritet, koristi se podrazumevani prioritet - DEFAULT_MESSAGE_LOGLEVEL. Takođe svaki od levela ima svoje značenje koje se može pročitati u headeru `linux/kernel.h`.

Ako je prioritet manji od `int console_loglevel`, poruka se takođe štampa u aktivnom terminalu. Ako su `syslogd` i `klogd` aktivni, poruka se takođe apenduje na odgovarajući log fajl bez obzira da li je štampana u konzoli ili ne. Koristimo visoki prioritet kada želimo da poruke prikažemo i u konzoli a ne samo u logovima. Kada pišemo realne kernel module uvidećemo značaj ovih prioriteta i kako oni zavise od situacije.

2.2 Kompajliranje kernel modula

Kornel modul ise kompajliraju malo drugačije od običnih userspace programa. Sva podešavanja koja se tiču kompilacije se tradicionalno nalaze u Makefile-ovima. Olakšicu nam ovde pruža `kbuild` koji ceo proces kompajliranja integriše u stadardan proces kompajliranja kernela.

Navodimo primer Makefile-a koji kompajlira naš modul `hello-1.c`:

```
obj-m+=hello-1.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

S tehničke tačke gledišta samo prva linija je važna.

Ako sačuvamo ovaj fajl pod imenom `Makefile` možemo kompajlirati kernel koristeći GNU-Make komandom `make`. Standardni izlaz bi bio sličan ovome:

```
make  -C  /lib/modules/5.16.1-zen1-1-zen/build  M=/home/nik/.local/src/c/hello-kernel modules
make[1]: Entering directory '/usr/lib/modules/5.16.1-zen1-1-zen/build'
  CC [M]  /home/nik/.local/src/c/hello-kernel/hello.o
  LD [M]  /home/nik/.local/src/c/hello-kernel/hello-module.o
  MODPOST /home/nik/.local/src/c/hello-kernel/Module.symvers
  CC [M]  /home/nik/.local/src/c/hello-kernel/hello-module.mod.o
  LD [M]  /home/nik/.local/src/c/hello-kernel/hello-module.ko
  BTF [M] /home/nik/.local/src/c/hello-kernel/hello-module.ko
make[1]: Leaving directory '/usr/lib/modules/5.16.1-zen1-1-zen/build'
```

Standardna je konvencija da kernel moduli odnosno njihovi objektni fajlovi imaju ekstenziju `.ko` da bi se razlikovali od standardnih C objektnih fajlova sa ekstenzijom `.o`. Razlog za ovu distinkciju je to što kernel moduli imaju dodatnu sekciju `.modinfo` koja ima dodatne informacije o njemu samom.

Sada kada smo kompajlirali modul možemo da koristimo komandu `modinfo hello-1.ko` da vidimo informacije sačuvane u navdenoj `.modinfo` sekciji:

```
filename:       /home/nik/.local/src/c/hello-kernel/hello-module.ko
author:         Nikola Tasić
description:    Test kernel module
license:        GPL
srcversion:     1DFF2B524CC39837C22BB87
depends:
```

```
retpoline:      Y
name:           hello_module
vermagic:       5.16.1-zen1-1-zen SMP preempt mod_unload
```

U primeru iznad vidimo dosta različitih informacija a način kako se one mogu dodati u modul ćemo pomenuti kasnije.

Sada kada smo kompajlirali modul možemo ga load-ovati u kernle koristeći komandu `insmod` koju smo pomenuli ranije. Izlaz logfajla možemo pogledati komandom `dmesg` i ako je sve prošlo kako treba tamo bi trebalo da se nađe tekst iz poziva `printk()`. Takođe možemo unload-ovati modul koristeći `rmmod`.

2.3 Hello world 2

Kao što smo već pomenuli moguće je koristiti bilo koje ime za init i cleanup funkcije. To možemo postići koristeći `inti_module()` i `cleanup_module()` makroe. Ti makroi su definisani u `linux/init.h` headeru. Naravno te funkcije moraju da budu definisane pre poziva makroa.

Ovde možemo da vidimo izmenjen prethodni primer:

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

static int __init hello_init(void) {
    printk(KERN_INFO "Hello world 1\n");

    // ne-nula povratna vrednost označava
    // da modul nije moguće bilo učitati
    return 0;
}

static void __exit hello_exit(void) {
    printk(KERN_INFO "Goodbye world 1\n");
}

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

2.4 Hello world 3 - __init i __exit makroi

Primetili ste da postoji promena u potpisu init i cleanup funkcija. `__init` makro dovodi do toga da se funkcija koja inicijalizuije modul briše iz memorije kada se izvrši. Ovo važi za drajvere ali ne i za module. Takođe postoji i makor `__initdata` koji je korisi za promenljive.

`__exit` makro definiše da se funkcija ne kompajlira u slučaju kada se modul ugradi u kernel i takođe kao ni `__init` nema nikakvog efekta kod modula. Ako razmislimo kada se cleanup funkcija poziva vidimo da ima smisla da se ona uopšte ne kompajlira ako se modul ugrađuje u kernel jer da - neće se nikada pozvati.

2.5 Hello world 4 - Dokumentacija

Ako dodajemo bilo koji ”proprietary” (u nedostatku odgovarajućeg prevoda) `insmod` će nam vratiti poruku sličnu ovoj:

```
# insmod xxxxxx.o
Warning: loading xxxxxx.ko will taint the kernel: no license
  See http://.tux.org/lkml/#export/tainted for information about tainted modules
Module xxxxxx loaded, with warnings
```

U kernelu postoji mehanizam koji nam omogućava da identifikujemo da li je modul izdad pod GPL ili srodnom licencom tako da ljudi koji dodaju module mogu da budu upozoreni o kodu koji nije open-source. Ovo se postiže koristeci `MODULE_LICENCE()` makro.

Slično, postoje ostali makroi koji se koriste za dokumentaciju. Kao što je na primer `MODULE_DESCRIPTION()` koji ocekuje da mu prosledimo kratak opis funkcionalnosti modula, `MODULE_AUTHOR()` opisuje autora modula dok `MODULE_SUPPORTED_DEVICE()` označava koji su podržani uređaju od strane modula.

Ovi makroi su definisani u `linux/module.h` i ne koriste se od strane kernela. Oni su prosta dokumentacija koja se može prikazati i alatima kao što su `objdump`.

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#define DRIVER_AUTHOR "Nikola Tasić"
#define DRIVER_DESC   "Hello driver"

static int __init hello_init(void) {
    printk(KERN_INFO "Hello world 1\n");

    // ne-nula povratna vrednost označava
    // da modul nije moguće bilo učitati
    return 0;
}

static void __exit hello_exit(void) {
    printk(KERN_INFO "Goodbye world 1\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");           // licenca
MODULE_AUTHOR(DRIVER_AUTHOR);    // autor
MODULE_DESCRIPTION(DRIVER_DESC); // opis
```

3.1 Moduli vs programi

3.1.1 Kako moduli počinju i kako se završavaju

Program obično počinje funkcijom `main()`, izvršava gomilu instrukcija i završava se s izvršenjem tih instrukcija. Kernel moduli fukncionišu malo drugačije. Modul uvek počinje funkcijom `init_module()` ili funkcijom koja je definisana `module_init()` makroom. Ovo je takozvani entry-point za svaki modul - on govori kernelu kakvu fukncionalnost modul ima i kako se ona inicijalizuje. Takođe konfguriše kernel kako da poziva ostatak njegovih funkcija. Kada se ta funkcija izvrši ona jednostavno return-uje i modul ne radi više ništa dok kernel ne pozove neku od njegovih funkcionalnosti.

Svi moduli se završavaju pozivom `cleanup_module()` funkcije ili pozivom funkcije koja je specificirana od strane `module_exit()` makroa. Ovo je exit funkcija za module; ona treba da odradi inverzno od svega što je init funkcija odradila. Ona unregister-uje svu funkcionalnost koju je je init funkcija registrovala.

3.1.2 Funkcije dostupne modulima

Programeri mogu da koriste funkcije iz ostalih biblioteka kao što je na primer standardna C bibliote - libc. Ove fukncije ne ulaze u sastav programa sve dok linking faze, koja osigurava da je kod, naime fukcije kao što su na primer `printf()` dostupne i ispravlja `call` instrukciju da pokazuje na odgovarajući kod.

Kernel moduli su malo drugačiji. I našem primeru korisitmo funkciju `printk()` bez uvoza `stdio.h`. Ovo je zbog toga što su moduli objektni fajlovi - njihovi simboli se učitavaju tek posle poziva `insmod`. Definicije ovih simbola dolaze iz samog kernela; jedine eksterne funkcije koje su dostupne su one koje kernel pruža. Ako želimo da vidimo koje simbole kernel pruža možemo pogledati `/proc/kallsyms`.

Jedna stvar koju trebamo imati na umu ke razlika između bibliotečkih funkcija i sistemskih poziva. Bibliotečke funkcije su višeg nivoa, izvršavaju se kompletno u userspace-u i pružaju ergonomičniji interfejs ka funkcijama koje zapravo rade onaj važan deo posla - sistemske pozive. Sistemski poziv se izvršava u kernel modu i provajdovani su od strane samog kernela. Na primer funkcija `printf()` deluje kao generička funkcija koja služi za štampanje ali zapravo ona formatira podatke u string i upisuje sadržaj koristeći sistemski poziv nižeg nivoa `write()` koji kasnije upisuje te podatke na standardni izlaz.

Možemo lako videti koje sistemske pozive printf poziva. Možemo kompajlirati prost program koji poziva `printf()` i taj program pzovati komandom `strace(4)`. `strace` je program koji nam daje detalje o svim sistemskim pozivima koje program pravi zajedno sa informacjiom o tome koji su mu bili argumenti i koja mu je povratna vrednost. Ako analiziramo izlaz iz poziva `strace`-a videćemo da pri kraju on poziva sistemski poziv `write` na način sličan ovome: `write(1, "neki tekst\n", 11)`. Kao i za druge sistemski pozive možemo pogledati informacije u Linux priručniku(`man(1)`) pozivom `man 2 write` - sekcija 2 je vezana za sistemske pozive, 3 za funkcije biblioteke.

Programer može takođe da piše programe koji zamenjuju sistemske pozive. Ovu tehniku koriste neki od malicioznih porgrama ali naravno se mogu koristiti i za šaljive stvari kao što je pisanje podrugljivih poruka kada neko pokuša da obriše fajl sa sistema.

3.1.3 User space vs kernel space

Jedna od najvažnijih uloga kernela je baratanje resursima, bilo da su ti resursi video kartic, hard disk ili memorija. Programi se takmiče ko će dobiti resurs. Mnogi prigrami istovremeeno koriste hard disk na primer i uloga kernela je da sinhronizuje te aktivnosti tako da se one uređeno odvijaju. Procesor naravno da bi opslužio korisnika radi u različitim modovima. Svaki od modova daje određene privilegije i slobodu šta može da se uradi sa sistemom. Intel x86 arhitektura definiše 4 moda odnosno prstena. Unix bazirani sistemi koriste 2 od ovih 4 moda - najviši prsten (ring 0 - supervisor mode) i najniži (user mode).

Funkcije biblioteka se pozivaju u user mode-u. Ta funkcija može da pozove jedan, ili više sistemskih poziva koji se pozivaju na konto funkcije i izvršavaju u supervizor odnosno kernel režimu. Kada se izvršavanje sistemskog poziva završi egzekucija se vraća u user mod.

3.1.4 Name space

Kada pišemo mali C program, koristimo promenljive koje si pogodne i imaju smisla čitaocu. Ako, s druge strane, pišemo funkcije i promenljive koje su deo velikog sistema sve globalne promenljive koje pišemo su deo velike grupe svih globalnih promenljivih celog sistema/programa. U tom kontekstu može doći do poklapanja imena simbola(funkcija ili promenljivih) i tada nastaje problem koji se zove namespace pollution(zagadjenje imenskog prostora). U velikim programima postoji napor da se rezervišu imena i razvije konvencija davanja imena simbolima.

Kada se piše kernel kod čak i najmanji modul će biti linkovan zajedno sa ostatkom čitavog kernela tako namespace pollution može da predstavlja ogroman problem. Najbolji način da se ovo izbegne

jeste da deklarišemo naše promenljive ključnom rečju static i da imamo smislen prefiks. Po konvenciji svi prefiksi za kernel kod su pisani malim slovima.

Fajl `/proc/kallsysm` sadrži sve simbole koji su poznati kernelu i samim tim dostupni svim modulima.

3.1.5 Code space

Upravljanje memorijom je veoma komplikovana tema. Navikli smo da pokazivači odnosno pointeri pokazuju na delove memorije odnosno memorijske lokacije. Zapravo nije tako. Pri pokretanju procesa , kernel odvaja deo realne fizičke memorije i daje je procesu. Svaki proces ima memoriju koja počinje sa 0x00000000 i nastavlja se dokle god. Bilo koja dva procesa mogu da targetiraju identičnu adresu u memoriji (npr. 0xabcdef55) ali ta adresa će rezultirati u različitim fizičkim adresama za svaki od procesa. Ovaj koncept se zove virtuelna memorija.

Kernel takođe ima svoju memoriju. S obzirom na to da je modul kod koji se dinamčki insertovan u kernel, on deli code space sa kernelom umesto da ima svoj. Tako da kada modul doživi SEGFAULT - i kernel takođe. Ako modul krene da piše po memoriji zbog proste greške - on piše po memoriji kernela. Ovo je mnogo mnogo gore nego što zvuči tako treba biti oprezan.

3.1.6 Drajveri

Jedna klasa modula su drajverski programi koji omogućavaju hardverske funkcionalnosti kao što su grafička kartica ili web kamera. Na Unix-u, svaka hardverska komponenta prestavlja jedan fajl u `/dev` folderu koji pruža onsnovni interfejs za interakciju sa uređajem. Na primer driver za zvučnu karticu povezuje `/dev/sound` fajl sa odgovarajućom zvučnom karticom a user space program kao na primer pulseaudio koristi `/dev/sound` bez potrebe da zna sa kojim drajverom komunicira.

3.1.6.1 Major i minor brojevi uređaja

Možemo da bacimo pogled na neke od uređaja koji predstavljaju particije diska:

```
nik@mariner ▶ ~ ▶ ls /dev/nvme0n1??
brw-rw---- 1 root disk 259, 1 Jan 19 10:00 /dev/nvme0n1p1
brw-rw---- 1 root disk 259, 2 Jan 19 10:00 /dev/nvme0n1p3
brw-rw---- 1 root disk 259, 3 Jan 19 10:00 /dev/nvme0n1p4
```

Možemo primetiti kolonu brojeva odvojenih zarezom. Prvi broj je major broj uređaja. Drugi broj je minor broj. Major broj govori o tome koji drajver se koristi da pristupi uređaju. Svaki drajver ima svoj jedinstveni major broj - svi fajlovi uređaja(device files) sa istim major brojem su kontrolisani od strane istog drajvera. Svi uređaji na ispisu su kontrolisani od strane drajvera sa major brojem 259.

Minor broj se koristi da razlikuje različite hardverske komponente. Osvrćući se na prethodni primer vidimo da se brojevi razlikuju - to znači da ih drajver vidi kao različite hardverske komponente.

Uređaji su podeljeni u dve kategorije: character uređaji i block uređaji. Razlika je u tome sto block uređaji imaju bafer za zahteve, tako da mogu da biraju redosled u po kojem će da odgovaraju na zahteve. Ovo je važno za skladišta, gde je brže čitati ili pisati po sektorima koji su bliže jedan do drugog od onih koji su udaljeni. Jos jedna razlika je u tome sto block uređaji mogu da prime input i daju output samo u blokovima(koji variraju od uređaja do uređaja) dok character uređaji mogu da pišu ili čitaju koliko god bajtova žele. Većina uređaja su character uređaji jer im ne treba buferovanje i ne operišu nad blokovima fiksne veličine. Možemo odrediti tip fajla gledajući priv karakter output-a komande `ls -l`. Ako je karakter `b` onda je uređaj block a ako je `c` onda je uređaj characrter. U navedenom primeru vidimo block uređaje.

Kada je sistem bio intaliran ti fajlovi su kreirani komandom `mknod`. Da bi kreirali novi character uređaj pod nazivom na primer `hello` sa major i minor borjevima 12 i 2 pozivamo `mknod /dev/hello c 12 2`. Ne moramo da stavljamo character fajlove u `/dev` ali se tamo nalaze po konvenciji. Medjutim

kreiranje za char fajla za potrebe testiranja je sasvim okej stavljanje tog fajla u na primer radni direktorijum. Ali naravno treba voditi računa o tome kde se nalazi taj fajl kada zapravo kompajliramo modul.

Kada se pristupa fajlu, kernel koristi major broj fajla da odredi koji driver je zadužen za hendlovanje. Ovo nam govori da kernelu nije bitan minor broj fajla. Drajver je jedini koji je zapravo zainteresovan za minor broj jer ga koristi da razlikuje različite fizičke uređaje.

Kada kažemo različite uređaje mislimo na apstrakciju uređaja koje nam kernel pruža ne na konkretan fizički uređaj.

U prilogu se nalazi implementacija prostog character device-a koji može da se koristi kao osnova za kreiranje upravljačkog(drajver) programa.

4 Reference

- The Linux Kernel Module Programming Guide, Peter Jay Salzman, Michael Burian, Ori Pomerantz, https://tldp.org/LDP/lkmpg/2.6/lkmpg.pdf
- linux, Linus Torvalds, https://github.com/torvalds/linux