Jernej Haložan

NAVIDEZNI ZNAKOVNI GONILNIK ZA LINUXOVO JEDRO

Diplomsko delo

Maribor, september 2010

Diplomsko delo univerzitetnega študijskega programa

NAVIDEZNI ZNAKOVNI GONILNIK ZA LINUX JEDRO

Študent:	Jernej Haložan
Študijski program:	UN ŠP - Računalništvo in informacijske tehnologije
Smer:	Strateška spletna igra
Mentor:	red. prof. dr. Damjan Zazula, univ. dipl. inž. el.
Somentor:	asist. Smiljan Šinjur, univ. dipl. inž. rač. in inf.

Maribor, september 2010

FERI
Fakulteta za elektrotehniko, računalništvo in informatiko
Smetanova ulica 17
2000 Maribor

Številka: BRIT-32
Datum in kraj: 08. 09. 2010, Maribor

Na osnovi 330. člena Statuta Univerze v Mariboru (Ur. l. RS, št. 1/2010)

SKLEP O DIPLOMSKEM DELU

1. **Jerneju Haložanu,** študentu univerzitetnega študijskega programa Računalništvo in informacijske tehnologije, se dovoljuje izdelati diplomsko delo pri predmetu Operacijski sistemi.

2. **MENTOR:** **red. prof. dr. Damjan Zazula**
   SOMENTOR: **asist. Smiljan Šinjur**

3. **Naslov diplomskega dela:**
   ZNAKOVNI GONILNIKI ZA LINUXOVO JEDRO

4. **Naslov diplomskega dela v angleškem jeziku:**
   CHARACTER DRIVERS FOR LINUX KERNEL

5. Diplomsko delo je potrebno izdelati skladno z "Navodili za izdelavo diplomskega dela" in ga oddati v treh izvodih (en vezan izvod in dva nevezana izvoda) ter en izvod elektronske verzije do 08. 09. 2011 v referatu za študentske zadeve.

Pravni pouk: Zoper ta sklep je možna pritožba na senat članice v roku 3 delovnih dni.

Dekan:

UNIVERZA V MARIBORU FAKULTETA ZA ELEKTROTEHNIKO RAČUNALNIŠTVO IN INFORMATIKO 8

Obvestiti:
- kandidata,
- mentorja,
- somentorja,
- odložiti v arhiv.

ZAHVALA

Zahvaljujem se mentorju profesorju Damjanu Zazuli za pomoč in vodenje pri opravljanju diplomskega dela. Prav tako se zahvaljujem somentorju Smiljanu Šinjurju.

Posebna zahvala velja staršem, ki so mi omogočili študij, in punci Aleksandri za pomoč ter vzpodbudo pri študiju.

NAVIDEZNI ZNAKOVNI GONILNIK ZA LINUXOVO JEDRO

Ključne besede: **gonilniki, linux, operacijski sistemi, navidezni gonilnik, testiranje gonilnika, sistemsko programiranje**

UDK: 004.45:004.89(043.2)

Povzetek

Tema diplomske naloge so znakovni gonilniki za linuxovo jedro. Delo obravnava delovanje operacijskih sistemov in gonilnikov. Predstavimo pa tudi mehanizme medsebojne komunikacije strojne opreme, operacijskega sistema in uporabniških programov. Zasnovali in izdelali smo navidezni znakovni gonilnik za operacijski sistem linux. Za preizkušanje gonilnika smo napisali testno aplikacijo. Naš gonilnik je navidezni, to pomeni, da vse operacije izvaja v pomnilniku. Zato za preizkušanje gonilnika ni potrebna posebna, dodatna strojna oprema.

CHARACTER DRIVERS FOR LINUX KERNEL

Key words: drivers, linux, operating systems, virtual drivers, driver testing, system programming

UDC: 004.45:004.89(043.2)

Abstract

This diploma thesis deals with character drivers for Linux kernels. The operation of operating systems and device drivers is investigated. We also present communication mechanism between hardware, operating system and user programs. Our practial goal was to implement a virtual character driver for Linux. For the driver testing porpuses we developed a test aplication. No specially added hardware is necessary to run the driver, because it is virtual and runs exclusively within computer memory.

Kazalo vsebine

Kazalo slik

Kazalo preglednic

Kazalo primerov

Seznam kratic

MTS	Michigan Terminal System
DRS	Device Support Routines
GNU	GNU's Not Unix
API	Application programming interface
WDM	Windows Driver Model
WDF	Windows Driver Foundation
RISC	Reduced Instruction Set Computing
CISC	Complex Instruction Set Computer
GCC	GNU Compiler Collection
FIFO	First In First Out
MS-DOS	Microsoft Disk Operating System
IBM	International Business Machines
PC	Personal Computer
OS	Operating System
FMS	Fortran Monitor System
MIT	Massachusetts Institute of Technology

1 UVOD

V tej diplomski nalogi se bomo seznanili z gonilniki ter njihovim namenom in prikazali njihovo delovanje na praktičnem primeru. Gonilniki spadajo med najpomembnejše dele operacijskega sistema [1]. Imajo vlogo posrednika med strojno opremo in višjenivojskimi aplikacijami. So del praktično vseh modernih operacijskih sistemov [2, 4]. Programerjem omogočajo višjenivojsko programiranje, saj so programi lahko abstraktnejši in neodvisni od strojne opreme [3]. Pisanje novih gonilnikov je v računalništvu vedno prisotno zaradi novih naprav in zaradi izboljšanja stabilnosti starejših različic.

V svetu strojne opreme obstaja veliko število različnih naprav in vsaka naprava potrebuje specifičen gonilnik, pisan samo za njo. Kljub temu ima večina gonilnikov v osnovi enak namen – komunikacijo strojne opreme z operacijskim sistemom [5]. Poznamo več vrst gonilnikov, ena izmed klasifikacij pa razdeli gonilnike v tri skupine, in sicer v gonilnike za:

- znakovne naprave (*char devices*),
- podatkovne naprave (*block devices*) in
- mrežne naprave (*network devices*).

V tem delu se bomo podrobno ukvarjali z gonilniki iz prve skupine, bežno pa bomo tudi omenili posebnosti ostalih skupin gonilnikov.

Zgodovina gonilnikov sega v 60. leta 20. stoletja in se povezuje z operacijskimi sistemi, kot so MTS (*Michigan Terminal System*), Unics in RC 4000 Multiprogramming System. MTS je bil razvit na Univerzi v Michiganu. Bil je eden prvih operacijskih sistemov, ki je ponujal večopravilnost in večprogramirnost. Uporabljalo ga je osem svetovnih univerz vse to leta 1999. Ena izmed funkcionalnosti sistema so bile tudi t. i. rutine za podporo napravam (*Device Support Routines*, DRS), predhodniki sodobnih gonilnikov.

Razlog zakaj so sploh začeli obravnavati gonilnike kot ločene programe operacijskega sistema tiči v standardiziranem modelu dostopa do naprav. V zgodnjih dneh računalništva so programerji vedno znova za vsak program posebej napisali »gonilnik«, ki je dostopal do poljubne naprave (npr. tračne enote). Takšno delo je zahtevalo veliko ponavljanja istih principov programiranja (*»reinventing the wheel«*). Zato so nalogo standardiziranega dostopa do naprav postopoma prevzeli operacijski sistemi in s tem gonilniki.

Gonilnike ponavadi pišejo programerji iz podjetij, kjer se razvija in proizvaja strojna oprema. Ti imajo dostop do dokumentacije strojne opreme in tako lahko podrobno razumejo delovanju naprav, za katere pišejo gonilnike. Nekatera podjetja pa za svoje naprave izdajo javne specifikacije o napravi in v takšnem primeru gonilnike lahko pišejo tudi neodvisni programerji, to so programerji, ki nimajo neposrednega stika z podjetjem, proizvajalcem strojne opreme. Ponavadi so gonilniki zadnjega tipa brezplačni in odprtokodni.

Namen diplomske naloge je predstaviti delovanje gonilnikov, opisati delovanje operacijskega sistema, predstaviti komunikacijo jedra operacijskega sistema med uporabniškimi programi in strojno opremo ter prikazati zasnovo in izdelavo znakovnega gonilnika za operacijski linux.

Najprej smo pregledali stanje na področju gonilnikov. Predpogoj za razumevanje delovanja gonilnikov je vsaj osnovno znanje o delovanju operacijskega sistema. Zato smo v tretjem poglavju splošno opisali operacijske sisteme in podrobno predstavili linuxovo jedro. Zatem smo preučili komunikacijo programov in stojnih naprav z operacijskih sistemom. V petem poglavju smo se lotili gonilnikov. Pogledali smo tri najpogostejše tipe gonilnikov in opisali njihove vloge v delovanju operacijskega sistema. Sledi poglavje, v katerem smo predstavili izdelavo gonilnika na primeru operacijskega sistema linux. Ker vsak gonilnik potrebuje uporabniški program, s pomočjo katerega dostopamo do naprave, smo izdelali tesno aplikacijo, ko jo z vsemi njenimi funkcionalnostmi opisujemo v sedmem poglavju. Na koncu sledijo še rezultati sklepi o uporabi in preizkušanju napisanega gonilnika.

2 PREGLED STANJA NA PODROČJU GONILNIKOV

Pregled stanja gonilnikov povzemamo pri trenutno treh najpopulularnejših operacijskih sistemih: Microsoft Windows, Mac OS X in GNU/Linux.

2.1 MS-DOS / Windows

Windows je skupina operacijskih sistemov iz podjetja Microsoft. Trenutno veljajo za najpopularnejše operacijske sisteme za namizne računalnike. Microsoft je začel svojo pot z operacijskim sistemom MS-DOS [6]. Tega operacijskega sistema niso v celoti napisali sami, ampak so kupili pravice za uporabo, ga dodelali in prodali naprej podjetju IBM, ki ga je vgradilo v svoje računalnike IBM PC.

MS-DOS se izvaja samo v realnem načinu (ali v nezaščitenem načinu), kar pomeni, da je lahko vsak program dostopa do celotnega pomnilnika. Klici storitev aplikacijskega vmesnika (API) v MS-DOS se izvedejo s pomočjo programskih prekinitev. Niso pa na voljo sistemske knjižnice, ki bi olajšale delo programerju. Do verzije 3.0 je bila možnost pisanja gonilnikov samo v zbirnem jeziku, novejše verzije pa omogočajo pisanje gonilnikov v kombinaciji programskega jezika C in zbirnega jezika.

MS-DOS so nasledili operacijski sistemi serije Windows 9x. V to serijo spadajo Windows 95, Windows 98 in Windows Me. Ta serija operacijskih sistemov je 32-bitna in lahko deluje v zaščitenem in nezaščitenem načinu. Zaradi kompatibilnosti s starejšimi programi, se stari gonilniki lahko izvajajo, vendar le s pomočjo upravnika navideznih strojev (*Virtual Machine Manager*, VMM), v zaščitenem načinu pa tečejo t. i. gonilniki navideznih naprav (VxD). Windows 98 pa je predstavil nov model WDM (*Windows Driver Model*) [7], ki omogoča drugačen pogled na gonilnike, tako da uvede več vrst, med katerimi pomenijo gonilniki naprav le eno podskupino. WDM temelji na modelu gonilnikov za Windows NT, ki je bil razvit za operacijski sistem Windows NT 3.1.

Nasledniki serije 9x pri Microsoftu so operacijski sistemi serije Windows NT. V to družino spada veliko sistemov, najpomembnejši pa so Windows XP, Windows Vista in Windows 7. V tej seriji se za gonilnike še naprej uporablja model WDM. So pa z izdajo operacijskega sistema Windows Vista predstavili model WDF (*Windows Driver*

Foundation) [8], ki predstavlja samo nov sloj nad WDM, s katerim so izboljšali stabilnost in robustnost gonilnikov.

2.2 Mac OS

Mac OS je serija operacijskih sistemov za računalnike Macintosh [9]. Mac OS ni bil prvi operacijski sistem iz Appla, ampak je bil in še vedno je njihov najuspešnejši operacijski sistem. Prva verzija je izšla leta 1984. Takrat se je imenovala preprosto System 1 [10]. Tako poimenovanje so ohranili vse do verzije System 7, leta 1997 pa so izdali operacijski sistem z imenom Mac OS 8, ki je naslednik prejšnjih sistemov z drugim imenom.

Operacijski sistemi Mac OS so znani po izpopolnjenem grafičnem uporabniškem vmesniku. Vendar so idejo grafičnega uporabniškega vmesnika prevzeli od Xeroxovega operacijskega sistema Alto OS [10]. Slednji je sodil med napredne 16-bitne sisteme, ki so omogočali upravljanje uporabniškega vmesnika z miško in mreženje računalnikov, imeli pa so vgrajene gonilnike za disk, miško in zaslon.

NextSTEP [10] je operacijski sistem, ki je prav tako pomembno vplival na Mac OS. Nastal je v podjetju Next, ki ga je ustanovil Steve Jobs (soustanovitelj Appla), potem ko so ga leta 1985 odpustili iz njegovega lastnega podjetja. Izdan je bil leta 1989, temeljil pa je na unixu in je podpiral dinamično nalaganje gonilnikov.

Zadnja verzija operacijskega sistema Mac OS je X (deseta verzija), temelji na sistemu NextSTEP. Izdana je bila leta 2000, trenutna verzija pa je 10.6. V Mac OS X so vpeljali sistem *I/O kit* iz operacijskega sistema Apple Rhapsody. Gre za objektno orientirano zbirko programerskih ogrodij (*frameworks*), knjižnic in programov, napisano v C++. Gonilniki, pisani z I/O kit, se lahko izvajajo v jedrnem ali uporabniškem načinu. Apple vzpodbuja programerje, naj čim več gonilnikov pišejo v uporabniškem načinu, saj je celotni sistem tako varnejši pred nepredvidenimi napakami gonilnika.

2.3 Unix / linux

Začetki unixa [11] segajo v leto 1969, ko so očetje modernega računalništva (Ken Thompson, Dennis Richie, Brian Kernighan in drugi) pri Bell Labs (razvojni oddelek

podjetja AT&T) razvili operacijski sistem Multics. Sistem je bil inovativen, vendar je imel mnogo težav. Delo na njem so opustili in iz tega se je razvil eksperimentalni sistem Unics. Kasneje so eksperimentalni sistem nadgradili v stabilen operacijski sistem in ga preimenovali v Unix (Version 1). Danes velja za enega najpomembnejših operacijskih v zgodovini računalništva. Zgled je bil sistemom BSD, minixu, linuxu in mnogim drugim.

Naš gonilnik bo tekel na operacijskem sistemu linux. V linuxu obstajata dva načina izvajanja gonilnikov: jedrni in modularni. V jedrnem načinu je gonilnik preveden kot del jedra, naloži se ob zagonu sistema in ga kasneje ni mogoče odstraniti iz pomnilnika. Modularni način pa je nekoliko fleksibilnejši, saj omogoča nalaganje gonilnika v času izvajanja operacijskega sistema. To pomeni, da lahko v samo jedro vključimo le najnujnejše gonilnike, ostale pa nalagamo in odstranjujemo po potrebi. Tako prihranimo pomnilniški prostor.

3 OPERACIJSKI SISTEMI

3.1 Namen

Namen operacijskega sistema je poenostaviti delo z računalnikom. Naloge operacijskega sistema lahko grobo razdelimo v dve veliki skupini. V prvo skupino spadajo mehanizmi upravljanja s strojno opremo, v drugo skupino pa lahko uvrstimo mehanizme, ki upravljajo z uporabniškimi programi. Operacijski sistem tako razdeli računalniški sistem na logični ali uporabniški nivo ter fizični ali strojni nivo. Sam pa tudi služi kot vmesnik med obema nivojema računalniškega sistema. Obenem pa operacijski sistem predstavlja model virtualnega stroja (*virtual machine*), ki poenostavi zapleteno zgradbo računalniškega sistema. Funkcionalnosti modernega operacijskega sistema so:

- skrbi za olajšan dostop uporabniškim programom do strojne opreme,
- upravlja s pomnilnikom sistema,
- nadzoruje mrežno komuniciranje,
- upravljati s posli in procesi in
- zagotavljati sistemsko varnost.

Če ne bi uporabljali operacijskih sistemov, bi moral vsak program posebej imeti napisane rutine za dostop in upravljanje s strojno opremo, kar pa bi pomenilo veliko zelo kompleksnih in različnih nestandarziranih računalniških sistemov.

3.2 Zgodovina

Razvoj operacijskih sistemov je tesno povezan z razvojem računalniških arhitektur, ki so jih poganjale, zato obravnavamo zgodovino operacijskih sistemov skupaj z zgodovino računalnikov.

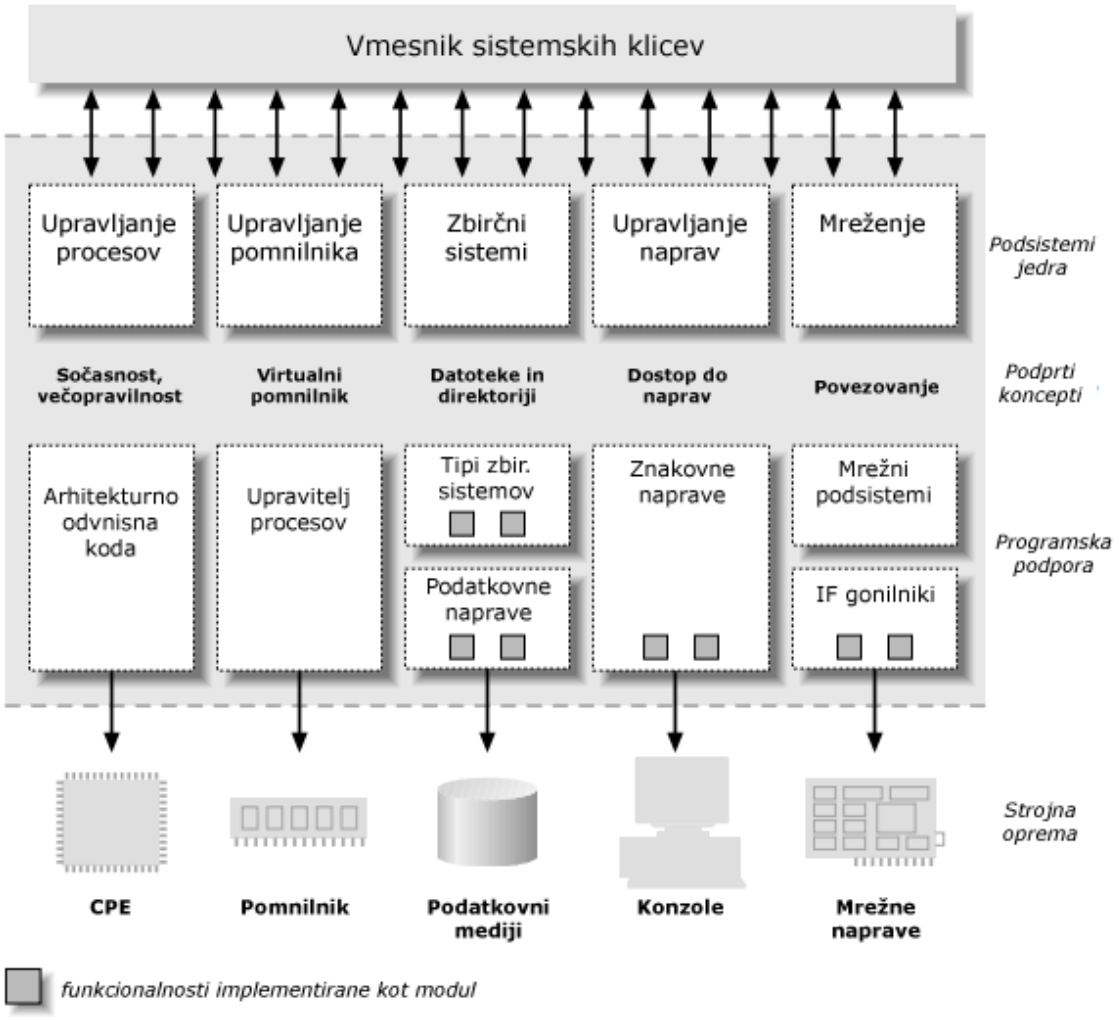
Prvi računalniki so se pojavili okrog začetka druge svetovne vojne. Temeljili so na elektronkah in so bili upravljani povsem ročno. Niso poznali nobenega od programskih jezikov, niti zbirnega jezika. Kasneje so jih poimenovali računalniki prve generacije.

Razvoj prvih operacijskih sistemov sega v sredino 50. leta 20. stoletja, ko so se pojavili prvi računalniki s tranzistorji. Operacijski sistemi te dobe so tekli na računalnikih druge generacije. Razvoj programa je potekal tako, da so programerji pisali programe v programskem jeziku FORTRAN ali v zbirnem jeziku, ga vnesli na luknjane kartice in te posredovali naprej operaterjem računalnika. Čez nekaj časa so od operaterja računalnika dobili vrnjen odgovor sistema. Primer takih operacijskih sistemov sta bila FMS (*Fortran Monitor System*) in IBSYS za računalnike IBM 7094.

Tretja generacija računalnikov nastopi s pojavom integriranih vezij v sredini 60. let in traja vse do leta 1980. V tej generaciji se pojavi operacijski sistem IBM System/360. Njegova naloga je bila zagotoviti enako delovanje programov na različnih procesorjih. Tak sistem je bil napisan v zbirnem jeziku in je bil izjemno kompleksen, hkrati pa tudi še poln napak. V tem času so na univerzi MIT in pri Bell Lab razvili operacijski sistem Multics, ki je predhodnik popularnega sistema unix.

V četrti generaciji računalnikov, ki traja od začetka 80. let prejšnjega stoletja pa vse do danes, sta prevladovala predvsem MS-DOS in unix. MS-DOS se je kasneje razvil v Microsoftova okna, unix pa je bil vzor sistem tipa minix in linux. Slednji je trenutno zelo popularen na področju strežnikov in super računalnikov, medtem ko na področju namiznih računalnikov z izjemnim tržnim deležem (okrog 90 odstotkov) [12] prevladujejo Microsoftovi operacijski sistemi.

3.3 Zgradba jedra



Slika 1: Zgradba linuxovega jedra

Slika 1 prikazuje arhitekturo linuxovega jedra. Prikazani so glavni podsistemi jedra, ki komunicirajo preko sistemskih klicev z uporabniškimi programi ali drugimi deli jedra. Pod njimi je označen koncept, ki ga podpirajo, in programska koda, ki stoji za omenjenimi funkcionalnostmi. Koda, ki je zadolžena za upravljanje, in dostopi do podatkovnih medijev, konzol in mrežnih naprav je vgrajena v gonilnike naprav.

Pogledali si bomo koncepte jedra operacijskega sistema linux in njegovo arhitekturo. Jedro linux in večine sodobnih operacijskih sistemov lahko razdelimo na sledeče dele:

Upravitelj procesov

Naloga upravitelja procesov je ustvarjanje in končanje procesov ter njihovo medsebojno povezovanje. Skrbi za stabilno delovanje sistema. Nadzoruje razvrščanje

in izvajanje procesov tako, da jim preprečuje, da bi drug drugemu prepisali podatke ali izvršilno kodo. Prav tako določa kako si procesi delijo centralno procesno enoto (CPE).

Upravljanje s pomnilnikom

Računalniški pomnilnik je zelo pomemben del operacijskega sistema, saj predstavlja glavno shrambo podatkov in programov, zato je pravilno upravljanje s pomnilnikom zelo pomembno za zmogljivost in stabilnost sistema. Operacijski sistem v tam namen razbija procese po straneh (*paging*) in jih segmentira (*segmentation*), da doseže čim bolj optimalno zasedenost pomnilnika. Različni deli sistema ali jedra dostopajo do operacij pomnilnika s sistemskimi klici.

Upravljanje z zbirčni sistemi

Trajno shranjevanje podatkov je stalnica od zgodnjih začetkov računalništva. Zbirčni sistem predstavlja način shranjevanja in organizacije podatkov na podatkovnem mediju. Moderni operacijski sistemi ponujajo v uporabo več različnih zbirčnih sistemov hkrati. Unix in linux bazirata na filozofiji zbirčnih sistemov, saj je skoraj vsaka naprava na sistemu predstavljena kot datoteka v zbirčnem sistemu. Takšen način predstavitve naprav ponuja dobro abstrakcijo, ki velja za celoten sistem.

Upravljanje z napravami

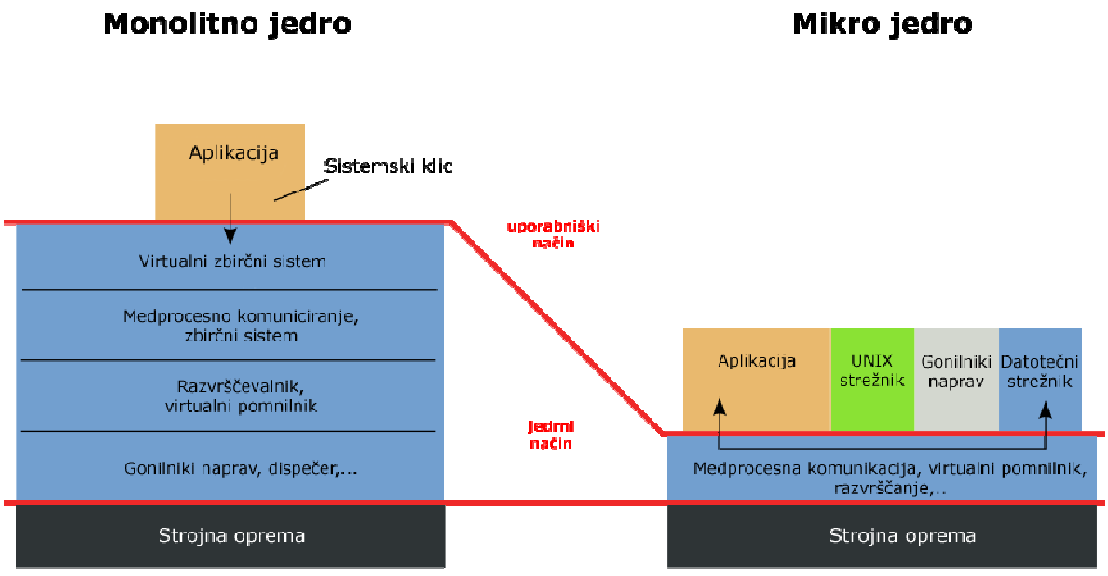
Večina operacij sistema je povezana z napravami, ki so priključene na sistem. Vsaka naprava je nekoliko drugačna in vsako napravo je treba obravnavati ločeno. Tukaj vstopijo gonilniki naprav, ki poskrbijo za raznolikost vseh naprav in višjenivojskim programom ponujajo skupen vmesnik za upravljanje z napravami. Komunikacija med deli operacijskega sistema poteka s pomočjo sistemskih klicev oziroma programskih prekinitev. V tej diplomski nalogi se najbolj osredotočamo ravno na ta del operacijskega sistema.

Mreženje

Komunikacijo med računalniki mora obvladovati operacijski sistem, saj mrežne operacije niso vezane na uporabniške procese. Prejemanje vhodnih paketov je asinhroni dogodek in operacijski sistem mora identificira, zbrati, urediti in posredovati prejete podatke naprej ustreznemu procesu. Prav tako mora sistem skrbeti za pošiljanje odhodnih podatkov ter za usmerjanje in dekodiranje ciljnih naslovov.

Za operacijske sisteme obstaja več različnih jedrnih arhitektur. V osnovi jih lahko razdelimo v naslednje skupine:

- monolitna jedra (*monolithic kernel*) so jedra, v katerih je zapakirano vse razen uporabniških programov. So kompleksna in učinkovita ter prevladujejo na trgu osebnih in strežniških računalniških sistemov. V to skupino jeder spadajo unixu podobni operacijski sistemi, kot sta linux in operacijski sistemi BSD, MS-DOS, serija Microsoft Windows 9x, OpenVMS in drugi;
- mikro jedra (*mikro kernel*) so jedra, ki so minimalna po zgradbi in vsebujejo samo osnovne komponente upravljanja z naslovnim prostorom, upravljanja niti in upravljanja medprocesne komunikacije. Vse ostale komponente operacijskega sistema se nahajajo med uporabniškimi programi. Klasičen primer mikro jedra pomenita minix in L4.
- hibridna jedra (*hybrid kernel*) so jedra, ki kombinirajo koncepte monolitnih in mikro jeder. To pomeni, da se nekatere storitve operacijskega sistema izvajajo v jedrnem načinu, nekatere pa v uporabniškem načinu. Primer take arhitekture, je serija jeder za Microsoft Windows NT.



Slika 2: Razlika med monolitnim in mikro jedrom

4 KOMUNIKACIJA Z OPERACIJSKIM SISTEMOM

4.1 Prekinitve in izjeme

Za uspešno delovanje računalniškega sistema morajo strojne naprave učinkovito komunicirati z operacijskim sistemom. Ker procesor in strojne naprave delujejo ob različnih hitrostih, je treba sporazumevanje sinhronizirati. Prvi način komunikacije je takšen, da procesor v neskončni zanki preverja stanje naprav. Ko so podatki na voljo, jih procesor prebere in obdela. Ta tehnika deluje, vendar ima slabosti. Prva slabost je ta, da procesor neprestano izvaja neskončno zanko za poizvedovanje stanja naprav. To pomeni, da troši veliko procesorske časa, ki bi ga sicer lahko porabil za bolj uporabne naloge. Druga slabost pa je, da lahko pride do izgube podatkov, če količina presega pasovno širino prenosne poti.

Zato je bil razvit koncept prekinitev (*interrupts*). Prekinitev pomeni, da se je zgodil dogodek, ki preusmeri izvajanje procesorja v prekinitveno rutino. Da se tak dogodek lahko zgodi, se sproži električni signal iz naprave ali iz notranjosti procesorja.

Ko naprava želi pozornost procesorja, npr. pripravljena je za delo, pripravljena je pisati podatke v pomnilnik itd., se sproži proces strojne prekinitve (*hardware interrupt*). Naprava pošlje električni signal skozi integrirano vezje za nadzor prekinitev (*interrupt controller chip*). Ta služi kot vmesnik med periferno napravo in procesorjem in skrbi za obveščanje procesorja o strojnih prekinitvah. Ko procesor prejme tako obvestilo, prekine izvajanje trenutnega programa in začne izvajati funkcijo, ki je bila vnaprej določena za obravnavanje prekinitev. Prekinitve takega tipa se imenujejo asinhrone prekinitve, ker se lahko zgodijo kdajkoli neodvisno od stanja procesorja. Strojne prekinitve se delijo v dve skupini: prekinitve, ki jih lahko prezremo (*maskable interrupts*), če tako želimo in prekinitve, ki jih ni možno prezreti (*nonmaskable interrupts*) in jih je treba vsakič obravnavati.

Druga pomembna vrsta prekinitev so programske prekinitve (*software interupts*). Prekinitve tega tipa so sinhrone prekinitve, saj se izvedejo s strani izvajanega programa oz. procesorja samega. Najpogosteje se uporabljajo za izvedbo sistemskih klicev v operacijski sistem.

Intelova terminologija govori o izjemah (*exceptions*) in tudi njih razdelimo v več skupin:

- procesorsko zaznane izjeme (*processor-detected exceptions*): nastanejo, ko procesor pri izvajanju operacije zazna napako, ke je ponavadi posledica programerske napake,
- napake (*faults*): jih je mogoče popraviti in ko se sproži mehanizem za popravljanje napake, se lahko prvotni program znova izvede; primer take izjeme je napka strani pri delovanju virtualnega pomnilnika (naslovljene strani ni v pomnilniku). Pomnilniška stran (*page*) je shranjena na trdem disku in je ni mogoče direktno nasloviti. Sproži se prekinitev in z njo opravilo operacijskega sistema, ki ustrezno stran skopira v pomnilnik, in program se lahko nadaljuje,
- pasti (*traps*): zgodijo se takoj po izvedbi inštrukcije za past (*trapping instruction*) in v ustreznem registru je zapisan naslov prekinitvene rutine, ki se izvede po tej izjemi; ponavadi se uporablja za razhroščevanje, saj mehanizem omogoča obveščanje razhroščevalnika o dosegu prekinitvene točke; po tej izjemi je mogoče normalno nadaljevati prvotni program
- preklic (*abort*): zgodila se je resna napaka in program se mora končati,
- programirane izjeme (*programmed exceptions*): zgodijo se na zahtevo programerja, z izvedbo inštrukcije *INT*, kar pomeni programsko prekinitev, lahko pa tudi z *INTO* in *BOUND*, če pogoj, ki ga preverjata, ni izpolnjen; uporabljajo se za implementacijo sistemskih klicev in za pomoč pri razhroščevanju.

Opisane vrste prekinitev najdemo v skoraj vsaki pomembnejši procesorski arhitekturi. Obstaja še nekaj drugih vrst prekinitev, ki pa na tem mestu niso pomembne in so seveda arhitekturno odvisne.

Ko se zgodi prekinitev, jo je treba obdelati. V tem trenutku se pokliče določena skupna funkcija, ki zna obravnavati prekinitve. Imenuje je upravnik prekinitev (*interupt handler*) in ima pomembno vlogo v vsakem operacijskem sistemu. Upravnik prekinitev je opravilo operacijskega sistema, ki se izvaja med delovanjem uporabniškega programa, v katerem se je prekinitev zgodila.

Naloge upravnika prekinitev so naslednje:

- ugotoviti mora kritičnost prekinitve – npr. jedro se nahaja sredi izvajanja pomembnega procesa in se zgodi prekinitev. V nekaterih primerih lahko manj

pomembne prekinitve počakajo, da jedro dokonča trenutno delo, v nekaterih primerih pa je prekinitev bolj pomembna od trenutnega procesa in sledi preklop med procesi;

- omogočati mora gnezdenje prekinitev, saj tako ohranja strojne naprave čim bolj zasedene. To pomeni, da se lahko zgodi prekinitev znotraj prekinitve, znotraj te prekinitve se lahko zgodi še ena prekinitev in tako naprej;
- zagotavljati mora kritične odseke znotraj prekinitev, kar pomeni, da nekaterih pomembnih prekinitev ni mogoče gnezditi in morajo ostale prekinitve počakati, da se trenutna prekinitev konča, in
- po koncu obdelave prekinitve se mora vrniti izvajanje uporabniškega programa znotraj katerega se je zgodila prekinitev.

4.2 Sistemski klici

Sistemski klici prestavljajo mehanizem za dostop do storitev jedra operacijskega sistema. So odvisni od izvedbe operacijskega sistema in arhitekture procesorja.

Program lahko pokliče sistemski klic samo, če ima za to dovoljenje. Procesorji imajo možnost delovanja v več različnih načinih. Tako imamo najmanj vsaj uporabniški in jedrni način delovanja procesorja. Ponavadi je načinov delovanja še več. Intelova arhitektura IA-32 (tudi x86) ponuja 4 načine delovanja: od načina, označenega z 0, ki je najbolj privilegiran, pa do 3, kateri ima najnižjo prioriteto. Razlika med različnimi načini delovanja je v tem, da imajo programi, ki se izvajajo v uporabniškem načinu (stanje 3 na arhitekturi IA-32) omejen naslovni prostor. Kar pomeni, da ne morejo dostopati do poljubnega dela pomnilnika in hote ali nehote vplivati na delovanje operacijskega sistema.

Sistemskih klicev je več vrst, v osnovi pa jih lahko razdelimo v naslednje kategorije. Sistemski klici za:

- nadzor procesov,
- upravljanje z datotekami,
- upravljanje z napravami,
- pridobivanje informacij in

- komunikacijo z napravami.

Nekaj primerov sistemskih klicev pri operacijskem sistemu linux: *open*, *read*, *write*, *close*, *fork* ,*exit*, *wait* in drugi. 32-bitna različica operacijskega sistema linux ima približno 300 različnih sistemskih klicev, ki so definirani v */usr/include/asm/unistd.h*.

Sistemski klici se ponavadi kličejo iz programskih knjižnic, kot je *libc* za programski jezik C, v starejših operacijskih sistemih kot je npr. MS-DOS, pa take knjižnice niso obstajale in je moral programer sam poskrbeti za izvedbo sistemskega klica. To se naredi tako, da se pokliče primerna programska prekinitev, v ustrezne registre pa se vpišejo številka sistemskega klica in vhodni parametri. Klicanje sistemske klica s pomočjo programskih prekinitev je značilno za procesorje tipa RISC (*Reduced Instruction Set Computing*).

Ker preklop iz uporabniškega v jedrni način s pomočjo prekinitev pobere nekaj časa, so pri vodilnih podjetjih za razvoj procesorjev – Intel in AMD – razvili hitrejši mehanizem. Gre za novejše procesorje tipa CISC (*Complex Instruction Set Computing*), ki imajo na voljo ukaze za izvedbo sistemskih klicev *SYSCALL/SYSENTER* in *SYSRET/SYSEXIT* in omogočajo hitro preklapljanje stanja procesorja.

5 GONILNIKI NAPRAV

5.1 Namen

Gonilniki naprav predstavljajo pomemben del vsakega operacijskega sistema. Razvoj na tem področju je »nikoli končana zgodba«. Vedno znova se sproti dodajajo podpore za nove naprave in arhitekture, stari gonilniki pa se nadgrajujejo zaradi odpravljanja napak ali spremenjene kode jedra operacijskega sistema.

Namen gonilnika naprave je preko enotnega vmesnika omogočiti enostavno upravljanje naprave iz uporabniškega programa. Vsaka naprava potrebuje svoj gonilnik, zato so gonilniki odvisni od naprav in operacijskih sistemov. Vsak operacijski sistem ima svoj način dela z gonilniki, vendar vseeno najdemo skupne značilnosti in dobre prakse pri programiranju gonilnikov za naprave.

5.2 Tipi gonilnikov

V svetu linuxa lahko gonilnike razdelimo v tri skupine, vedno pa se najdejo posebne naprave, ki jih ni mogoče preprosto vključiti v obstoječi model. Prav tako se zaradi različnih pristopov modeli delitve gonilnikov lahko razlikujejo med operacijskimi sistemi.

Znakovne naprave (character devices)

Znakovna naprava je naprava, do katere lahko dostopamo s tokom znakom (*stream of bytes*). Gonilnik znakovne naprave pa je zadolžen, da poskrbi za izvedbo operacij z nizom znakov. Osnovni gonilnik znakovne naprave premore vsaj štiri osnovne sistemske klice: *open*, *close*, *read*, *write*. Primera takšnih naprav v linuxu sta tekstovna konzola */dev/console* in serijska vrata */dev/ttyS0*. Ponavadi znakovna naprava deluje v načinu FIFO (*First In First Out*). Tako dobimo abstrakcijo prenosa podatkov, ki se kot tok vode pretaka v napravo in iz nje. Kot bomo kasneje videli, bo naprava, za katero bomo napisali gonilnik, omogočala tudi določanje premika v napravi s sistemskim klicem *llseek*. Znakovne naprave so v linuxu, tako kot vse ostale naprave, zapisane v direktoriju */dev*. Znakovno napravo določa prvi znak imena naprave, če je to črka *c*.

Podatkovne naprave (block devices)

Podatkovne ali bločne naprave se uporabljajo za trajno shranjevanje večjih količin podatkov. Primeri takih naprav so trdi diski, spominski ključi USB, optični pogoni, disketni pogoni in podobno. Naprave te vrste omogočajo delo z večjimi količinami podatkov, zato jedro praviloma omogoča vmesno shranjevanje podatkov (*buffering*). Do naprave se dostopa enako kot do znakovne naprave, saj so pri linuxu podatkovne naprave tudi vpisane v direktorij */dev* – primeri: */dev/hda*, */dev/sda*, */dev/fd*, */dev/cdrom*. Od zunaj se zdijo podobne znakovnim napravam, le da je prvi znak v menu za podatkovno napravo *b* namesto *c*. Dejansko obstaja velika razlika v delovanju gonilnikov. Omeniti je tudi treba, da podatkovne naprave lahko gostijo zbirčni sistem.

Mrežne naprave (network devices)

Vse mrežne povezave so speljane preko mrežnih vmesnikov (*network interfaces*) Ponavadi za mrežnim vmesnikom stoji naprava (mrežna kartica), ki je sposobna komunicirati z oddaljenimi računalniki in si izmenjevati podatke. Lahko pa je mrežni vmesnik izveden tudi programsko. Naprava *loopback* v linuxu je takšen primer. Pri linuxu mrežne naprave niso vpisane v direktoriju */dev*, kakor smo razložili za znakovne in podatkovne naprave. Mrežna naprava je v večini primerov avtomatično ustvarjena s strani gonilnika ob inicializaciji. Večina mrežnih povezav je tokovno orientirana (*stream oriented*), medtem ko mrežne naprave ne delujejo na tak način, ampak na način prenašanja blokov podatkov. Mrežni gonilnik se mrežnih tokov ne zaveda, ampak dala samo s podatkovnimi paketi.

Glavna značilnost dobro načrtovanih gonilnikov je njihova neodvisnost od načina uporabe naprave. To pomeni, da se v načrtovanju gonilnika ne ukvarjamo s tem, kdo lahko dostopa do gonilnika in naprave, katere funkcionalnosti lahko uporablja in kakšne podatke lahko pridobiva iz naprave. Zagotavljanje pravilnega načina uporabe naprave je naloga višjenivojskih programov, ki gonilnik uporabljajo, in ne gonilnika samega. Res je, da je treba včasih pri nekaterih napravah narediti izjemo, vendar se ne glede na izjeme skušamo držati pravila o gonilnikih, katerih uporaba ni odvisna od naprave.

Večina gonilnikov naprav se izvaja v jedrnem načinu (*kernel mode*), medtem ko se uporabniške aplikacije, ki uporabljajo gonilnike, izvajajo v uporabniškem načinu (*user mode*). Razlika med obema stanjema je v tem, da ima jedrni način neposredni dostop do naprav in ostalih zaščitenih delov jedra, medtem ko uporabniški način tega nima. Jedrni način pa nima dostopa do knjižnic in programov iz uporabniškega načina. To pomeni, da

sta oba načina dobro ločena, kar pripomore k večji varnosti. Nekateri gonilniki se lahko izvajajo tudi v uporabniškem načinu (*user mode drivers*), kar jih naredi bolj stabilne, vendar zato izkazujejo nižjo zmogljivost.

6 IZVEDBA ZNAKOVNEGA GONILNIKA

6.1 Zahteve in specifikacije

Zahteve znakovnega gonilnika so:

- delovanje v okolju operacijskega sistema linux,
- izvajanje v jedrnem načinu,
- navidezni način delovanja,
- operacije s pomnilnikom in ne neposredno z napravo,
- upravljanje s podatkovnimi strukturami,
- izvajanje gonilnika kot linuxovega modula,
- izvajanje gonilnika kot dela jedra,
- vpis znakov v gonilnik z metodo *write*,
- branje znakov iz gonilnika z metodo *read*,
- izvajanje IOCTL-klicev:
    - branje znakov in ob tem spreminjanje v velike črke,
    - branje znakov in ob tem spreminjanje vsakega drugega v veliko črko,
    - možnost obračanja shranjenega niza,
    - možnost brisanja vseh znakov iz naprave, v našem primeru iz pomnilnika.

Ker se gonilniki v nekaterih operacijskih sistemih lahko izvajajo tudi v uporabniškem načinu, je treba točno definirati, v katerem načinu se bo naš gonilnik izvajal in na katerem operacijskem sistemu ter računalniški arhitekturi ga bomo lahko uporabljali. V našem primeru je zahtevan operacijski sistem, ki temelji na linuxu, to pa pomeni tudi, da se bi gonilnik načeloma lahko izvajal na katerem koli sistemu, ki podpira POSIX.standard. Gonilnik je napisan za eno najbolj popularnih arhitektur v računalništvu – Intelovo arhitekturo IA-32 (x86).

6.2 Razvojno okolje

Gonilnik smo razvijali v distribuciji Ubuntu 10.4, ki velja za eno najpopularnejših distribucij linuxa. Za pisanje kode smo uporabili namizno okolje GNOME in njihov privzeti urejevalnik besedila *Gedit*. Za prevajanje in povezovanje smo uporabili orodje *make* in prevajalnik *gcc* projekta GNU (*GNU's Not Unix*).

Make je program, ki avtomatično zgradi izvedljiv program iz množice izvornih datotek, knjižnic in modulov. To stori tako, da bere in interpretira posebne datoteke, imenovane *Makefiles*. Tako z enim ukazom iz ukazne vrstice lahko zgradimo celotni gonilnik, saj *make* kliče izbrani program za prevajanje izvorne kode in povezovanje ustreznih izvedljivih modulov. *Make* je zalo razširjen pri sistemih unix. Originalni program *make* je bil napisan za unix konec 70. let 20. stoletja. Sčasoma je doživel več izboljšav in več različnih izvedb. Med drugimi obstaja tudi verzija za Microsoftova okna.

GCC (*GNU Compiler Collection*) je zbirka prevajalnikov, ko so del projekta GNU. Gre za prosto in odprto kodo, kar je tudi glavna filozofija projekta GNU. Sprva je v zbirko GCC sodil prevajalnik za programski jezik C, kasneje pa so projekt razširili s podporo za jezike C++, objective-C, objective-C++, java, FORTRAN, pascal, ada in mnoge druge. GCC teče praktično na vseh pomembnih sodobnih računalniških arhitekturah. V mnogih sodobnih sistemih unix je privzeti prevajalnik, obstaja pa tudi izvedba za Microsoftova okna, vključena v program MinGW.

Ko zgradimo gonilnik, imamo na voljo dve možnosti. Prva je ta, da lahko gonilnik vključimo (in izključimo) v jedro operacijskega sistema dinamično, druga pa je, da ga prevedemo kot del jedra in kasnejše odstranjevanje gonilnika ni več mogoče.

Izvedba gonilnika kot modula nam omogoča prilagodljivejšo zasnovo jedra operacijskega sistema, saj v času zaganjanja sistema (*boot time*) ni treba nalagati vseh gonilnikov, ampak samo najbolj nujne. Če kasneje vključimo novo napravo v sistem, se gonilnik dinamično naloži v jedro operacijskega sistema. Postopek prevajanja gonilnika se nekoliko razlikuje od postopka prevajanja za navadne uporabniške aplikacije. V slednjem primeru prevajalnik neposredno tvori izvedljivo kodo, ki jo lahko takoj poženemo, v primeru gonilnika pa se tvori ali objektna koda ali dinamična knjižnica (DLL), ki jo naložimo v jedro na zgoraj omenjena dva nična.

Da prevedemo gonilnik kot modul, je treba napisati datoteko *Makefile*, ki se mora nahajati v istem direktoriju kot izvorna koda gonilnika. Naša datoteka *Makefile* ima naslednjo vsebino:

```
obj-m += cdriver.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
```

V ukazni vrstici poženemo program *make* in posledično se aktivira linuxov sistem prevajanja jedra *Kbuild*, ki izkoristi razširjeno sintakso programa *make*. Nato se za ustrezne datoteke v direktoriju pokliče program *gcc*, ki izvorno kodo prevede v objektno kodo in jo nato poveže v gonilniški modul. V linuxu ima gonilniški modul končnico *ko*.

Modul v jedro naložimo s sistemskim programom *insmod* (*install module*) tako, da kot prvi parameter podamo ime prevedenega gonilniškega modula. V našem primeru je ukaz naslednji:

```
insmod ./cdriver.ko
```

Seznam vseh naloženih gonilniških modulov dobimo tako, da poženemo program *lsmod*. Če pa želimo modul odstraniti iz jedra, poženemo sistemski program *rmmod* (*remove module*), kot parameter pa podamo ime modula. V našem primeru je ukaz naslednji:

```
rmmmod cdriver
```

Če želimo gonilnik vključiti neposredno v jedro, je potrebno prevajanje celotnega linuxovega jedra. V tem primeru se gonilnik naloži ob zagonu operacijskega sistema. Prikazali bomo primer za distribucijo Ubuntu 10.4.

Najprej je treba pridobiti izvorno kodo jedra. To lahko storimo na več načinov. Mi smo jo naložili iz uradnega Ubuntu repozitorija kode z naslednjim ukazom:

```
apt-get install linux-source
```

Tako pridobimo najnovejšo stabilno različico izvorne kode jedra, ki je v našem primeru verzije *2.6.32*. Koda je shranjena v direktoriju /usr/src in stisnjena z algoritmom *bzip2*. Razširimo arhiv z ukazom:

```
tar xvfj /usr/src/linux-source-2.6.32
```

V trenutnem direktoriju se nam ustvari nov poddirektorij z vso razpoložljivo izvorno kodo. Sedaj konfiguriramo jedro s programom *make menuconfig* (ali z *make xconfig*). Izberemo funkcionalnosti, ki jih želimo imeti v novem jedru in rezultat se shrani v posebno datoteko .config. Ta datoteka na začetku prevajanja tolmači, kako se prevedejo izbire uporabnika v končno sliko jedra. Mi smo nato ročno v datoteko *Makefile*, ki je v direktoriju jedra drivers/char/Makefile. dodali ukaz:

```
obj-y: += cdriver.o
```

Ukaz pomeni, da se bo objektna koda našega gonilnika prevedla kot del jedra. Če bi hoteli gonilnik na tem mestu prevesti kot modul, bi izbrali ukaz *obj-m*.

Sedaj prevedemo jedro z ukazom:

```
MAKEFLAGS="HOSTCC=/usr/bin/gcc CCACHE_PREFIX=distcc" make-kpkg --rootcmd fakeroot --initrd --append-to-version=-suspend2 kernel-image kernel-headers kernel-source
```

make-kpgk je program iz distribucije Debian, na kateri bazira tudi Ubuntu. Prevede jedro in ustvari *.deb* paketke, ki se uporabljajo v Ubuntovih in Debianovih programskih repozitorijih. Pred njim nastavimo spremenljivko *MAKEFLAGS* z ukazom *distcc*, ki nekoliko pospeši prevajanje jedra. Če želimo hitro prevajanje jedra, je treba pred tem optimizirati izbrane nastavitve. Najpreprostejša nastavitev je, da izklopimo možnost razhročevanja jedra. Posledica tega je manjša velikost slike jedra in izvedljivih modulov.

Ko imamo sliko jedra, jo je treba namestiti za uporabo v zagonskem nalagalniku. V našem primeru uporabljamo zagonski nalagalnik *GNU/GRUB*. Ker je slika jedra shranjena v programskih paketih enostavno instaliramo ustvarjene paketke, za povezavo z *GRUB* pa v ozadju poskrbijo mehanizmi Ubuntuja. To storimo z ukazom:

```
dpkg -i linux-image-2.6.32-jernej.deb
dpkg -i linux-headers-2.6.32-jernej.deb
```

6.3 Registracija gonilnika

Pred inicializacijo gonilnika je treba poskrbeti za dostop do naprave. Do znakovnih naprav v operacijskih sistemih linux dostopamo s pomočjo zapisov v zbirčnem sistemu. Njihova naloga je na logičnem nivoju predstaviti uporabniku strojno napravo. V linuxu je prvi znak imena znakovne naprave označen s črko *c*, ime pa je vpisano v direktoriju */dev*.

Zapis v direktoriju */dev* enolično določa gonilniški modul, s katerim se moramo povezati pri delu z izhodno napravo. To dosežemo z glavno (*major number*) in podštevilko gonilnika (*minor number*). Glavna številka nam pove številko modula, ki ga naj operacijski sistem uporabi, podštevilka pa enolično določi, s katero napravo naj gonilniški modul izvede zahtevano operacijo, če je zadolžen za več naprav hkrati.

crw-rw-rw-	1	root	root	1,	3	Apr 11 2002	null
crw-------	1	root	root	10,	1	Apr 11 2002	psaux
crw-------	1	root	root	4,	1	Oct 28 03:04	tty1
crw-rw-rw-	1	root	tty	4,	64	Apr 11 2002	ttys0
brw-rw-rw	1	root	root	5	1	Apr 12 2002	sda0

Preglednica 1: Podroben prikaz vsebine direktorija

V tabeli 1 je prikazan izpis iz direktorija */dev: ls –l /dev*. V prvem stolpcu vidimo pravice dostopa do datoteke, prvi znak pa označuje tip datoteke. Peti stolpec prikazuje glavno številko gonilnika, šesti stolpec pa podštevilko naprave. Posebni vpis */dev/sda0* vodi do gonilnika bločne naprave, katerega glavna številka je 5, naprava pa je prva pri tem gonilniku (označena s podštevilko 1).

Preden lahko uporabimo gonilnik, moramo ustvariti posebno datoteko v direktoriju */dev*. To storimo s klicem operacijskemu sistemu *mkdev*. Za znakovne gonilnike uporabimo naslednjo sintakso:

```
mknod /dev/cdriver c 100 0
```

Prvi parameter je ime naprave (*/dev/cdriver*), drugi predstavlja tip naprave (*c – character*), tretji glavno številka gonilnika (*100*) in četrti podštevilko gonilnika (*0*). Zgornji ukaz ustvari poseben vpis v direktorij */dev*, s katerim povežemo ime in številko gonilnika, hkrati pa nastavimo dostopne pravice za nadrejenega uporabnika (*root user*). To pomeni, da nepriviligirani uporabnik sistema ne morejo uporabljati gonilnika za svoje delo. Zato poženemo program *chmod* za spreminjanje dostopnih pravic in z naslednjim ukazom omogočimo vsem uporabnikom sistema dostop do našega gonilnika:

```
chmod a+rw /dev/cdriver
```

Inicializacija gonilniškega modula je prva funkcija, ki se pokliče ob nalaganju gonilnika v pomnilnik. Naloga inicializacije je obvestiti operacijski sistem, da je na voljo nov gonilnik, ki je pripravljen za delo, ponuja pa določene funkcionalnosti.

Prva naloga, ki jo opravi inicializacija je, da določi modulu na nivoju jedra glavno številko gonilnika, ki smo jo vpisali v */dev/cdriver*. To storimo z klicem funkcije

```
int register_chrdev_region (dev_t  from, unsigned  count, const char *  name);
```

Kot prvi parameter podamo spremenljivko, v katero se shrani glavna številka gonilnika, kot drugo spremenljivko podamo število naprav, ki jih bomo uporabljali z modulom (podštevilka), tretji parameter pa nastavi interno ime gonilnika.

6.4 Upravljanje s pomnilnikom

Ena od zahtev gonilnika je, da zna sprejemati in zapisovati poljubno število znakov v napravo, zato je treba uvesti upravljanje z dinamičnim pomnilnikom. Vključimo zaglavno datoteko (*header file*) linked_list.h, v kateri je definirana struktura LinkedList za delo z dvosmerno povezanim seznamom. Prav tako so vgrajene pomožne funkcije, ki olajšajo delo z dinamičnim pomnilnikom:

- void LL_init(): inicializira spremenljivke, potrebne za delo s seznamom;
- int LL_insert(char *data, int count): vstavi podano število znakov v seznam na ustrezno mesto;
- int LL_read(char* data, int count, int offset, int mode): prebere želeno število znakov iz seznama, ob tem pa upošteva zamik (*offset*); kot zadnji parameter lahko določi način branja podatkov:
    - READ_NORMAL: vrne črke v mali pisavi,
    - READ_UPPER: vrne črke v veliki pisavi in
    - READ_MIXED: vrne izmenjujoče se črke v veliki in majhni pisavi
- void LL_delete_all(): zbriše vse znake iz seznama.

Ena izmed možnosti gonilnika je tudi obračanje vrstnega reda seznama z ustreznim klicem IOCTL (o tem več kasneje), zato vse zgoraj naštete funkcije za delo z dinamičnim

pomnilnikom upoštevajo tudi to postavko. Ko imamo pripravljene funkcije za delo s pomnilnikom, v inicializaciji gonilniškega modula pokličemo LL_init.

6.5 Definicija gonilnikovih funkcionalnostih

Jedrni način delovanja je t. i. nezaščiteni način. Treba se je pravilno odzvati na asinhrone dogodke in ustrezno zaščititi kritični odsek kode. V ta namen v inicializaciji gonilniškega modula inicializiramo binarni semafor, in sicer s klicem funkcije *init_MUTEX* v katero kot parameter vnesemo naslov semaforja, ki ga bomo uporabljali znotraj gonilnika. Za zapiranje semaforja bomo skozi celoten gonilnik uporabljali funkcijo *down_interruptible*. Prednost slednje pred funkcijo *down* je v tem, da je možno čakanje prekiniti tako, da se ne ustavi izvajanje uporabniške aplikacije, ki je sprožila klic. Za odpiranje semaforja bomo uporabili funkcijo *up*.

Do sedaj smo z glavnimi in s pomožnimi gonilnikovimi številkami registrirali logično oz. konceptualno povezavo med uporabniškim in jedrnim prostorom. Sedaj je treba povezati gonilnikove funkcionalnosti med obema prostoroma. Še prej pa je treba spoznati tri podatkovne strukture, ki so nujne za osnovno delo z gonilniki. Vse tri so definirane v datoteki <linux/fs.h>.

Vsaka odprta datoteka na uporabniškem nivoju je v jedru predstavljena s strukturo file. Avtomatično jo ustvari jedro linuxa ob odpiranju datoteke. Ta struktura nima nič skupnega s strukturo *FILE*, ki se uporablja na uporabniškem nivoju za delo z datotekami. Struktura ima več polj, za nas pa so najpomembnejša naslednja:

- *mode_t f_mode*: hrani informacijo ali je datoteka odprta za branje ali za pisanje ali pa za oboje hkrati. Pri realizaciji funkcije za branje in pisanje gonilnika te informacije ni treba eksplicitno preverjati, saj to stori jedro sistema implicitno namesto nas. Kazalec na to spremenljivko jedro linuxa ob klicu funkcije gonilnika poda avtomatično (ne velja za funkciji *open* in *close*);
- *loff_t f_pos*: hrani trenutni položaj branja ali pisanja v napravo. Te vrednosti ne smemo spreminjati v nobeni gonilniški funkciji razen v funkciji, ki je posebej namenjena za spreminjanje trenutnega položaja (obravnavamo jo pri naslednji

strukturi). Kazalec na to spremenljivko dobi gonilnik pri funkcijah branja in pisanja gonilnika;

- *struct file_operations *f_op*: kazalec na strukturo operacij gonilnika - jedro linuxa priredi kazalec za inicializirane podatke, ko odpira gonilnik;. Ko pride do sistemskega klica tukaj definirane funkcije jedro preusmeri izvajanje k vgrajeni funkcionalnosti. To spremenljivko lahko tudi ročno spreminjamo, tako da lahko dosežemo upravljanje več naprav z enim modulom, saj glede na gonilnikovo podštevilko zamenjamo kazalec na strukturo z operacijami, ki so definirane za drugo napravo, in
- *void *private_data*: ta kazalec lahko uporabimo za shranjevanje poljubnih podatkov, ki jih uporabljamo z odprto napravo. Po odpiranju naprave je nastavljena na *NULL*, po koncu dela z napravo pa je seveda treba pomnilniški prostor sprostiti

Struktura *file_operations* poskrbi za tehnično povezavo sistemskih klicev s funkcijami gonilnika. V tej strukturi so navedeni kazalci na funkcije, ki jih podpira gonilnik. Ob sistemskem klicu iz uporabniškega programa se s povratnim klicem (*callback*) pokliče ustrezna gonilnikova funkcija. V gonilnik smo implementirali *file_operation* strukturo, kot jo prikazuje primer 1. Glave najpomembnejših funkcij so naštete spodaj:

- *int (*open) (struct inode *, struct file *)*: pokliče se ob odpiranju povezave z napravo. V tej funkciji inicializiramo pomnilniški prostor potreben za delovanje naprave, in pripravimo napravo na delo;
- *int (*release) (struct inode *, struct file *)*: pokliče se ob zapiranju povezave z napravo; v tej funkciji sprostimo vse vire, ki smo jih uporabljali za napravo,
- *ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)*: pokliče se ob branju z naprave. Prvi parameter je kazalec na strukturo *file*, drugi parameter kaže na pomnilniško lokacijo uporabniške aplikacije, kamor lahko shranimo podatke, tretji parameter pove velikost podatkov, ki zanimajo uporabnika, zadnji parameter pa pove trenutno pozicijo v napravi;
- *ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)*: pokliče se ob pisanju na napravo. Pomen parametrov je enak kot pri funkciji *read*, s tem da je tukaj drugi parameter pomnilniška lokacija, s katere bomo prebrali podatke v gonilnik;

- *int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long)*: pokliče se ob sistemskem klicu IOCTL in se uporablja za upravljanje naprave in
- *loff_t (*llseek) (struct file *, loff_t, int)*: pokliče se ob zamenjavi trenutnega položaja v datoteki.

```
struct file_operations my_fops = {
    .owner   =  THIS_MODULE,
    .read    =  cdriver_read,
    .write   =  cdriver_write,
    .open    =  cdriver_open,
    .release =  cdriver_release,
    .llseek  =  cdriver_llseek,
    .ioctl   =  cdriver_ioctl,
};
```

Primer 1: Vsebina *file_operations* strukture našega gonilnika

Zadnja pomembna struktura za delo z gonilniki je struktura *inode*. Uporablja se za interno predstavitev datotek v jedru linuxa. Medtem ko se za vsako odprto datoteko ustvari ločena struktura *file*, je *inode* za vsako datoteko samo ena. Vsebuje veliko informacij o posamezni datoteki, za razvijalce gonilnikov pa sta pomembni samo naslednji dve polji:

- *dev_t i_rdev*: hrani glavno številko in podštevilko gonilnika in
- *struct cdev *i_cdev*: kazalec na interno strukturo, s katero se v jedru predstavi znakovna naprava. Polje *ops* v strukturi *cdev* kaže na strukturo *file_operations*, v kateri so kazalci na vgrajene funkcije gonilnika.

```
struct cdev      *my_cdev;
   my_cdev  =  cdev_alloc();
my_cdev->owner  =  THIS_MODULE;
 my_cdev->ops  =  &my_fops;
     result  =  cdev_add(my_cdev,dev,1);
```

Primer 2: inicializacija strukture *cdev*

Zgornji primer 2 prikazuje inicializacijo strukture *cdev* in dodajanje strukture v jedro operacijska sistema. Od tega trenutka naprej je možno klicati metode gonilnika, zato je pomembno, da so pred tem klicem postorjena vsa ostala opravila v zvezi z inicializacijo gonilniškega modula.

6.6 Realizacija osnovnih gonilnikovih funkcij

Med osnovne gonilnikove funkcije spadajo inicializacija odpiranje naprave (*open*), pisanje na napravo (*write*), branje z naprave (*read*), določanje položaja v napravi (*llseek*) in zapiranje naprave (*close*).

Funkciji za odpiranje naprave (*open*) in za zapiranje naprave (*close*) sta v našem primeru prazni, ker v našem gonilniku ni treba posebej pripraviti naprave, saj je gonilnik navidezni in svoje operacije izvaja v pomnilniku. Kljub temu sta obe funkciji obvezni, ob inicializaciji jedro linuxa avtomatično ustvari strukturo *file*, ki je pomembna gonilniška struktura.

Najpomembnejši osnovni funkciji gonilnika sta seveda funkciji za branje in pisanje na napravo. Zato bomo tudi več pozornosti namenili omenjenima funkcijama.

Funkcija za pisanje na napravo ima naslednjo glavo:

```
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)
```

Opis parametrov funkcije:

*struct file *filp*	kazalec na strukturo *file*, ki shrani vse podatke odprte datoteke (*file descriptor*)
*const char __user *buf*	kazalec na pomnilniško lokacijo, v katero je uporabniški program shranil podatke za pisanje v gonilnik
size_t count	število podatkov v zlogih, ki jih zapišemo na napravo
*loff_t *f_pos*	pozicija v napravi, kamor bomo zapisali vhodne podatke

Preglednica 2: Opis parametrov funkcije *write*

Ker vstopamo v kritični odsek kode, ne želimo, da bi dva procesa hkrati imela možnost pisanja podatkov na napravo, zato moramo na začetku funkcije *write* zapreti binarni semafor.

Zatem ustvarimo začasni prostor v pomnilniku s programom *kmalloc*, v katerega skopiramo podatke iz uporabniške aplikacije. Pri kopiranju podatkov med jedrnim in uporabniškim slojem ne smemo uporabiti neposrednega naslova spremenljivke namesto kazalca nanjo (*dereferencing*) zaradi naslednjih razlogov:

- odvisno od arhitekture in konfiguracije jedra podani naslov mogoče sploh ne obstaja in ga je treba ustrezno preobdelati,
- pomnilnik uporabniških programov je razdeljen na strani in naslovljena lokacija se mogoče nahaja v virtualnem pomnilniku, kar bi pripeljalo do napake strani in končanje uporabniške aplikacije,
- podani naslov bi lahko bil vodil v kodo z napako ali zlonamerno kodo, kar lahko privede do varnostne luknje. Treba ga je ustrezno preveriti, ali je primeren za delo.

Zaradi zgornjih razlogov moramo uporabiti posebno funkcijo:

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

Funkcija *copy_from user* poskrbi za pravilno kopiranje podatkov med uporabniškim in jedrnim slojem. Funkcija prejme kazalec, ki kaže, kam naj shrani podatke, od kod jih naj prebere in koliko zlogov naj skopira.

Ko imamo veljavne podatke v gonilnikovem vmesnem prostoru, pokličemo funkcijo *LL_insert*, ki shrani podatke v dinamičen seznam. Na koncu funkcije *write*, sprostimo uporabljen začasni pomnilniški prostor in odpremo binarni semafor. Funkcija vrne število znakov, ki so bili uspešno skopirani in shranjeni na napravo.

Funkcija za branje z naprave ima naslednjo glavo:

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

Opis parametrov funkcije:

*struct file *filp*	kazalec na strukturo *file*, ki shrani vse podatke odprte datoteke (*file descriptor*)
*const char __user *buf*	kazalec na pomnilniško lokacijo, katero je uporabniški program pripravil za shranjevanje podatkov iz gonilnika
size_t count	število podatkov v zlogih, ki jih preberemo iz naprave
*loff_t *f_pos*	začetna pozicija v napravi, od koder bomo brali podatke

Preglednica 3: Opis parametrov funkcije *read*

Funkcija za branje podatkov iz gonilnika deluje zelo podobno funkciji za pisanje. V začetku zapremo binarni semafor. To je potrebno zato, da ne morejo obenem drugi procesi

pisati novih podatkov v napravo. Na tem mestu bi lahko dodatno vgradili mehanizem istočasnega branja več različnih procesov, vendar smo se odločili za enostavnejšo možnost.

Funkcija preveri, ali ta podana parametra številom znakov, ki jih želimo prebrati, in z veljavnim odmikom od začetka seznama. Če sta parametra veljavna, funkcija dinamično rezervira prostor v pomnilniku v katerega skopiramo podatke iz seznama s klicem funkcije *LL_read*. V nasprotnem primeru funkcija vrne uporabniškemu programu napako. Za kopiranje podatkov uporabljamo funkcijo *copy_to_user*, ki je po funkcionalnosti identična funkciji *copy_from_user*, s tem da kopira v uporabniški program.

Na koncu funkcije *read* sprostimo začasni pomnilniški prostor in odpremo semafor. Funkcija vrne število znakov, ki so bili uspešno prebrani in shranjeni v uporabniški program.

Funkcija za določanje položaja v seznamu ima naslednjo glavo:

```
loff_t cdriver_llseek(struct file *filp, loff_t off, int whence)
```

Opis parametrov funkcije:

*struct file *filp*	kazalec na strukturo *file*, ki shrani vse podatke odprte datoteke (*file descriptor*)
loff_t off	odmik, ki ga nastavimo pri napravi
int whence	položaj od katerega nastavimo odmik; možnosti so: od začetka datoteke, trenutnega položaja, konca datoteke

Preglednica 4: Opis parametrov funkcije *llseek*

Funkcija *llseek* je zelo preprosta, saj glede na položaj odmika ustrezno nastavi polje *f_pos* v strukturi *file*. Vrne pa novi položaj v datoteki.

6.7 IOCTL-klici

IOCTL-klici se ponavadi uporabljajo za nastavitve in posebne operacije z napravami. V našem gonilniku jih bomo uporabili za upravljanje s podatki na napravi.

Funkcija *ioctl* ima naslednjo glavo:

```
int (*ioctl) (struct inode *inode, struct file *filp, unasigned int cmd, unsigned long arg);
```

Opis parametrov funkcije:

*struct inode *inode*	kazalec na strukturo *inode*, ki vsebuje glavno in podštevilko gonilnika, informacije o znakovni napravi in druge podatke o datoteki
*struct file *filp*	kazalec na strukturo *file*, ki shrani vse podatke odprte datoteke (*file descriptor*)
unasigned int cmd	ukaz, ki ga naj izvede klic *ioctl*
unsigned long arg	kazalec na vhodne parametre – vedno ga podamo s podatkovnim tipom *int* in ga kasneje eksplicitno pretvorimo v ustrezen tip

Preglednica 5: Opis parametrov za sistemski klic *ioctl*

Uporabniški program poda pri klicu *ioctl* z argumentom *cmd* številko ukaza, ki se naj izvede v gonilniku. Da lahko ohranjamo konsistentnost ukazov na obeh nivojih, moramo v zaglavni datoteki cdriver.h definirati vsak ukaz kot celoštevilsko konstanto. Datoteko cdriver.h moramo potem vključiti na vseh mestih, kjer kličemo sistemski klic *ioctl* v uporabniškem programu, in na vseh mestih, kjer klic *ioctl* obravnavamo v gonilniku.

Če ne nameravamo vključiti našega gonilnika v uradni linux repozitorij, si lahko ukazne konstante poljubno izmislimo. V nasprotnem primeru je treba konstante določiti po posebnih pravilih. Razvijalci jedra se trudijo, da bi bile konstante za *ioctl* klice unikatne na celotnem sistemu. Varneje je, če pošljemo neveljaven ukaz želeni napravi in dobimo vrnjen odgovor, kot pa da nenamerno izvedemo veljaven ukaz na neželeni napravi. Za generiranje unikatnih konstant uporabimo makre *_IO* (želimo izvesti samo ukaz brez prenašanja podatkov v napravo), *_IOR* (ob izvedbi ukaza želimo od naprave sprejeti še podatke), *_IOW* (ob izvedbi ukaza želimo napravi poslati še podatke) in *_IOWR* (ob izvedbi ukaza želimo napravi poslati še podatke in jih prav tako prejeti).

V našem klicu *ioctl* je definiranih pet ukazov:

- CDRIVER_IOC_READ: prebere znake iz gonilnika in vse spremeni v velike črke,
- CDRIVER_IOC_WRITE: zapiše ustrezne podatke v gonilnik,

- CDRIVER_IOC_READ_MIXED: prebere znake iz gonilnika v mešanem formatu zaporednih velikih in malih črk,
- CDRIVER_IOC_READ_REV: obrne podatkovni seznam, tako da vpisano besedilo prebere od zadaj naprej, in
- CDRIVER_IOC_DELETEALL: počisti vse podatke iz pomnilnika – sprazni seznam.

Pri zadnjih dveh ukazih gre samo za posredovanje naloge napravi in v tem primeru ni nobenega prenosa podatkov med uporabniškim in jedrnim nivojem. Za zgornje tri ukaze pa si za pošiljanje podatkov pomagamo s strukturo *RWStructure*, ki smo jo definirani v cdriver.h. V primeru 3 je prikazana definicija strukture.

```
struct RWStructure
{
int num         /* število znakov, ki jih bomo prebrali */
int offset      /* zamik – uporablja se samo pri branju */
char* text      /* besedilo, ki ga bomo zapisali/brali */
char* error     /* napaka v operaciji */
};
```

Primer 3: Struktura *RWStructure* za prenos podatkov pri IOCTL klicu

Na začetku klica *ioctl* preverimo, ali je podani ukaz sploh ustrezen.

```
if (_IOC_TYPE(cmd) != CDRIVER_IOC_MAGIC) return -ENOTTY;
if (_IOC_NR(cmd) > CDRIVER_IOC_MAXNR) return -ENOTTY;
```

Primer 4: Preverjanje ustreznosti ukaza za IOCTL-klic

To storimo tako, da preverimo ujemanje prvih osmih bitov ukaza z vnaprej določeno konstanto CDRIVER_IOC_MAGIC, ki je skupna vsem ukazom v našem klicu *ioctl*. Z drugim *if*-stavkom pa še preverimo, ali je podan ukaz znotraj območja definiranih ukazov. Opisan postopek prikazuje primer 4.

Sedaj preverimo s funkcijo *access_ok*, če je pomnilniški prostor na uporabniškem nivoju pripravljen za delo, kar nam prikazuje primer 5.

```
if (_IOC_DIR(cmd) & _IOC_READ)
    err = !access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)
    err =  !access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd));
if (err) return -EFAULT;
```

Primer 5: Preverjanje pravilnosti ukaza IOCTL klica

Vhodni ukaz in pomnilniški prostor sta veljavna, zato lahko začnemo z izvajanjem ukaza. Sledi stavek *switch* v katerem preverimo vsebino ukaza in glede na vrednost se pokliče ustrezna funkcija – branje, pisanje, obračanje ali brisanje seznama.

7 IZVEDBA TESTNE APLIKACIJE

7.1 Zahteve in specifikacije

Namen testne aplikacije je preizkusiti funkcionalnosti navideznega znakovnega gonilnika. Zahteve testne aplikacije so naslednje:

- povezati se z znakovnim gonilnikom, ki se izvaja v jedru operacijskega sistema,
- možnost prekinitve povezave,
- možnost pisanja poljubnega števila znakov z metodo *write*,
- možnost branja podatkov z metodo *read* in upoštevanjem števila želenih znakov ter odmika,
- možnost izvajanja IOCTL-klicev za:
    - branje velikih znakov,
    - branje izmenjujočih se velikih in malih znakov,
    - pisanje znakov v gonilnik,
    - obračanje seznama in
    - brisanje vseh podatkov iz pomnilnika.

7.2 Razvojno okolje

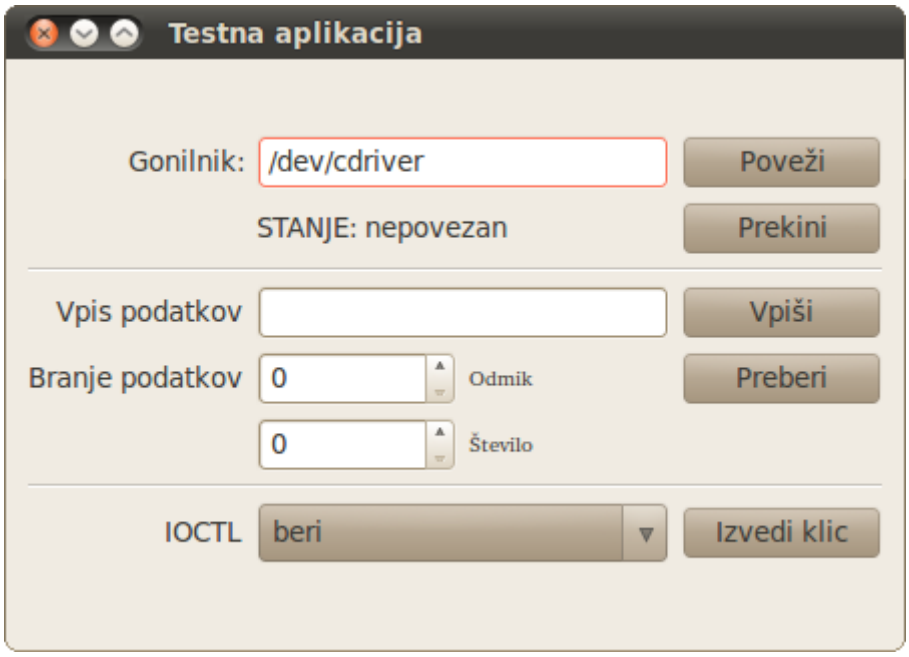
Aplikacija se bo izvajala na popularni distribuciji linuxa Ubuntu 10.4. Testna aplikacija je napisna v programskem jeziku C++, za prikaz uporabniškega vmesnika pa smo uporabili knjižnico Qt. Za pisanje kode smo uporabljali razvojno okolje QtCreator, ki je specializirano za delo s knjižnico Qt.

Qt je arhitekturno neodvisna knjižnica namenjena pisanju predvsem arhitekturno neodvisnih grafičnih aplikacij. To pomeni, da program napišemo za linux in ga lahko kasneje poženemo tudi v operacijskem sistemu okna, brez, ne da bi bilo treba izvorno kodo posebej prilagajati za okna. Ob podpori za arhitekturno neodvisne grafične aplikacije, imamo v Qt-ju na voljo tudi podporo za delo z bazami, datotekami XML ter večnitnim in

mrežnim programiranjem. Razvoj Qt poteka pod okriljem Nokie, nastal pa je v norveškem podjetju Trolltech, ki ga je Nokia prevzela leta 2008.

7.3 Izvedba aplikacije

Grafični uporabniški vmesnik smo oblikovali s QtCreatorjem in ga s funkcionalnostjo signalov in rež (*signals and slots*) v Qt povezali z razredom *Driver*, ki skrbi za klicanje funkcij gonilnika.



Slika 3: Grafični vmesnik testne aplikacije

Qt-jev mehanizem signalov in rež se uporablja za komunikacijo med objekti. Po tem mehanizmu se verjetno Qt najbolj razlikuje od večine ostalih programskih ogrodij. Najprej je treba s specifičnimi makro ukazi za Qt povezati dve metodi med seboj. Ob nekem dogodku v prvem razrezu se pokliče njegova metoda in sproži signal (*emit signal*). Mehanizem signalov in rež dostavi signal prej povezani metodi poljubnega objekta. Prva funkcija, ki se je izvedla ob dogodku pravzaprav ne ve nič o poslanem signalu, saj vso obdelavo opravi mehanizem signalov in rež. Pomembno pa je, da imata obe funkciji enako glavo, saj je tako klic tipsko varen (*type safe*).

Dogodki ob kliku na gumbe so vezani (preslikani) na metode objekta *Driver*. Gre za preprost razred, ki odpre datoteko in kliče sistemske funkcije knjižnice C (open, *read*,

write, *lseek*, *ioctl*), iz teh pa se sprožijo sistemski klici ter tako komunicirajo z gonilnikom. Objekt *Driver* sprejme vrnjene rezultate in napake ter jih posreduje grafičnemu oknu, kjer se tudi prikažejo.

8 SKLEP

V tej diplomski nalogi smo predstavili gonilnike naprav s teoretičnega in praktičnega vidika.

Pregledali smo stanje na področju gonilnikov v trenutno najpopularnejših operacijskih sistemih in ugotovili, da operacijska sistema linux in Microsoftova okna težita k izvajanju gonilnikov na jedrnem nivoju, medtem ko razvijalce za Mac OS X vzpodbujajo k pisanju gonilnikov na uporabniškem nivoju.

Ker je poznavanje operacijskih sistemov predpogoj za delo z gonilniki naprav, smo si pogledali namen, zgodovino, zasnovo in delovanje linuxovega jedra. Poglavje smo povzeli z delitvijo operacijskih sistemov in povedali, da trenutno temeljijo najpopularnejši operacijski sistemi na konceptu monolitnega jedra v katero je zapakirano vse razen uporabniških programov.

Da lahko operacijski sistem normalno deluje, je treba komunicirati med uporabniškimi programi, jedrom operacijskega sistema in strojnimi napravami. Komunikacija med deli sistema je tako ključna za delovanje operacijskega sistema. Zato smo v nadaljevanju predstavili koncepte prekinitev in sistemskih klicev. Spoznali smo, da je treba preklop med jedrnim in uporabniškim načinom izvesti čim bolj učinkovito, zato smo predstavili ukrepe novejših procesorjev na tem področju.

Zasnova in izdelava navideznega znakovnega gonilnika predstavlja bistvo diplomske naloge. Zato smo podrobno opisali delovanje znakovnih gonilnikov pri linuxu in razložili našo izvedbo našega gonilnika. Na koncu smo predstavili še uporabniško testno aplikacijo, ki kliče rutine gonilnika.

9 LITERATURA

[1] J. Gorbet, A. Rubini, G. Kroah-Hartman, *Linux device drivers*, 3. izdaja, O'Reilly Media, 2005

[2] D. Zazula, *Operacijski sistemi*, tretji ponatis, FERI, 2008

[3] D. P. Bovet, M. Caseti, *Understanding the Linux Kernel*, 3. izdaja, O'Reilly Media, 2005

[4] A. Tanenbaum, A. Woodhull, *Operating systems: design and implementation*, 2. izdaja, Prentice Hall, 1987

[5] D. Zazula, M. Lenič, *Principi sistemske programske opreme*, 1. izdaja, FERI, 2006

[6] *History of Microsoft*, http://en.wikipedia.org/wiki/History_of_Microsoft (18.9.2010)

[7] *Windows Driver Model*, http://en.wikipedia.org/wiki/Windows_Driver_Model (18.9.2010)

[8] *Windows Driver Foundation*, http://www.microsoft.com/whdc/driver/wdf/default.mspx (18.9.2010)

[9] *Mac OS*, http://en.wikipedia.org/wiki/Mac_OS (18.9.2010)

[10] *A History of Apple's Operating Systems* http://www.kernelthread.com/publications/appleoshistory/ (18.9.2010)

[11] *Unix*, http://en.wikipedia.org/wiki/Unix (18.9.2010)

[12] *Operating System Market Share*, http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8 (18.9.2010)

FERI
Fakulteta za elektrotehniko, računalništvo in informatiko
Smetanova ulica 17
2000 Maribor

IZJAVA O USTREZNOSTI DIPLOMSKEGA DELA

Podpisani mentor Damjan Zazula izjavljam, da je
(ime in priimek mentorja)
študent Jernej Haložan izdelal diplomsko
(ime in priimek študenta-tke)
delo z naslovom: Znakovni gonilniki za linuxovo jedro
(naslov diplomskega dela)

v skladu z odobreno temo diplomskega dela, Navodili o pripravi diplomskega dela in mojimi navodili.

Datum in kraj: Maribor, 22. 9. 2010

Podpis mentorja: Damjan Zazula

FERI
Fakulteta za elektrotehniko, računalništvo in informatiko
Smetanova ulica 17
2000 Maribor

IZJAVA O ISTOVETNOSTI TISKANE IN ELEKTRONSKE VERZIJE DIPLOMSKEGA DELA IN OBJAVI OSEBNIH PODATKOV DIPLOMANTOV

Ime in priimek diplomanta-tke: JERNEJ HALOŽAN

Vpisna številka: E1003011

Študijski program: R-IT (UN)

Naslov diplomskega dela: ZNAKOVNI GONILNIKI ZA LINUXOVO JEDRO

Mentor: red. prof. dr. DAMJAN ZAZULA

Somentor: asis. SMILJAN ŠIRJUR

Podpisani-a JERNEJ HALOŽAN izjavljam, da sem za potrebe arhiviranja oddal elektronsko verzijo zakljucnega dela v Digitalno knjižnico Univerze v Mariboru.
Diplomsko delo sem izdelal-a sam-a ob pomoci mentorja. V skladu s 1. odstavkom 21. clena Zakona o avtorskih in sorodnih pravicah (Ur. l. RS, št. 16/2007) dovoljujem, da se zgoraj navedeno zakljucno delo objavi na portalu Digitalne knjižnice Univerze v Mariboru.

Tiskana verzija diplomskega dela je istovetna elektronski verziji, ki sem jo oddal za objavo v Digitalno knjižnico Univerze v Mariboru.

Podpisani izjavljam, da dovoljujem objavo osebnih podatkov vezanih na zakljucek študija (ime, priimek, leto in kraj rojstva, datum diplomiranja, naslov diplomskega dela) na spletnih straneh in v publikacijah UM.

Datum in kraj: 22. 9. 2010, MARIBOR

Podpis diplomanta-tke: Jernej Haložan