UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Blaž Lampreht

Primerjava hitrosti komunikacije med UDP vtičniki ter UDP skladom

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Borut Robič

Ljubljana, 2010

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01696/2010
Datum: 01.09.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **BLAŽ LAMPREHT**

Naslov: **PRIMERJAVA HITROSTI KOMUNIKACIJE MED UDP VTIČNIKI TER UDP SKLADOM**

COMMUNICATION SPEED BETWEEN UDP SOCKETS AND UDP STACK

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Primerjajte hitrosti dveh načinov komunikacije z UDP paketi. Pri prvem načinu uporabite vtičnike, pri drugem pa komunikacijo med uporabniškim in sistemskim prostorom operacijskega sistema s pomočjo preslikanega (mapiranega) pomnilnika. Pri slednjem načinu po potrebi prilagodite ustrezni gonilnik in druge metode, udeležene pri komunikaciji.

Mentor:

prof. dr. Borut Robič

Fakulteta za računalništvo in informatiko
UNIVERZA V LJUBLJANI
2

Dekan:

prof. dr. Franc Solina

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a BLAŽ LAMPREHT,

z vpisno številko 63020093,

sem avtor/-ica diplomskega dela z naslovom:

PRIMERJAVA HITROSTI KOMUNIKACIJE MED UDP VTIČNIKI TER UDP SKLADOM

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  prof. dr. Boruta Robiča

  in somentorstvom (naziv, ime in priimek)

  _____

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 24. 12. 2010 Podpis avtorja/-ice: _____

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Zahvala

Zahvaljujem se mentorju prof. dr. Borutu Robiču za strokovno pomoč in usmerjanje pri izdelave diplomske naloge.

Posebej se zahvaljujem svojima staršema, Dragu in Mojci Lampreht, za moralno in finančno podporo tekom študija.

Zahvala gre tudi vsem ostalim, ki so na kakeršen koli način pripomogli k nastanku diplomskega dela.

Kazalo

Povzetek

V diplomski nalogi bomo primerjali čase pošiljanja UDP paketov z uporabo dveh načinov komunikacije. Prvi način bo uporabljal običajne vtičnike, drugi način pa bo namesto njih uporabljal pomnilniško preslikane datoteke. V primeru uporabe pomnilniško preslikanih datotek bo med drugim potrebno tudi pravilno obravnavati podatke, pridobljene iz uporabiškega prostora. V sistemskem prostoru bo potrebno sestaviti UDP okvir, vanj vstaviti pridobljene podatke ter okvir poslati prejemniku. Pri načinu komunikacije, ki bo uporabljala vtičnike, pa bo potrebno odpreti vtičnik ter začeti komunikacijo s prejemnikom. Primerjali bomo predvsem čase pošiljanja različno velikih paketov ter različnega števila paketov.

Ključne besede: internetni paket, UDP okvir, vtičniki, Linux, jedro operacijskega sistema, mrežni gonilnik, mrežni vmesnik, pomnilniška preslikava, uporabniški prostor, sistemski prostor.

Abstract

In my thesis, we will compare the UDP packet sending times, using two different communication protocols. The first protocol will use standard sockets and the second will use memory mapped files instead of sockets. In the latter case, there will among other things be the necessity to correctly handle the data received from the user space. The kernel space will have to construct a UDP frame and include the data it received from the user space in it. Then it will have to send the frame to its receiver. Using the protocol with sockets, we will have to open a socket and start communication. We will compare times spent for sending packets with different size and different number of packets between both communication protocols.

Keywords: internet packet, UDP frame, sockets, Linux, operating system kernel, network driver, network interface, memory mapping, user space, kernel space.

Poglavje 1

Uvod

1.1 Kratka zgodovina interneta

Komunikacija z uporabo pošiljanja paketov obstaja že od leta 1969, ko je ARPA (Advanced Research Projects Agency) razvil prvo omrežje in ga poimenoval ARPANET. Na začetku je ARPANET povezoval štiri računalnike jugozahodnih univerz v ZDA, in sicer UCLA, raziskovalni inštitut v Stanfordu, UCSB in univerzo v zvezni državi Utah. Do januarja 1971 se je ARPANET povečal na 13 vozlišč, do aprila 1972 pa na 23. V 70. letih prejšnjega stoletja se je razvil tudi TCP/IP protokol (*angl.* Transmission Control Protocol/Internet Protocol). Leta 1980 ga je sprejelo tudi ameriško ministrstvo za obrambo, do leta 1983 pa se je začel uporabljati po vsem svetu. ARPANET je na začetku dosegal hitrosti do 50 KBitov/s.

1.2 Kratka zgodovina Linuxa

Linux je operacijski sistem, ki ga je za hobi razvil takrat še študent helsinške univerze, Linus Torvalds. Linus je z delom začel leta 1991, ko se je namenil izboljšati majhen UNIX sistem, imenovan Minix. Do leta 1994 je bila na voljo verzija 1.0 Linuxovega jedra operacijskega sistema. Izvorna koda Linuxovega jedra je prosto dostopna za vsakogar. Obstaja mnogo družb in organizacij ter tudi posameznikov, ki so na podlagi Linuxovega jedra razvili svoje inačice operacijskih sistemov. Zaradi svoje prilagodljivosti, robustnosti in tudi proste dostopnosti je Linux postal glavna alternativa Unixovim in Microsoftovim operacijskim sistemom.

1.3 Motivacija

Zaradi vedno večje količine podatkov se velikokrat pojavi potreba po vedno hitrejših prenosih. V ta namen se razvijajo vedno hitrejši mediji za pretakanje podatkov, kot so na primer optični kabli. Vsekakor so pomembni tudi protokoli, ki skrbijo za prenos samih podatkov, torej kako se podatek sestavi v paket ter se ga pripravi na pošiljanje preko komunikacijskega medija, da se potencialna pasovna širina prenosnega medija kar najbolje izkoristi. V pričujočem delu se bomo osredotočil na takšne protokole. Dva načina komunikacije bomo podrobneje razložili in primerjali. Prvi način uporablja takoimenovane vtičnike, ki so precej pogost in ustaljen način komunikacije, drugi način pa je eksperimentalni način, ki se, kot mi je znano, še ni uporabljal. Drugi način uporablja komunikacijo med sistemskim ter uporabniškim prostorom z uporabo preslikanega pomnilnika. Ideja za uporabo le-tega se je pojavila zato, da bi prišlo do hitrejše komunikacije podatkov iz uporabniškega do sistemskega prostora. V tem primeru izgradnja vtičnikov ne bi bila potrebna, kar bi prihranilo nekaj časa. Ta prihranjen čas bi se seveda pokazal šele pri komunikaciji večje količine podatkov, kar je v današnjem času vsekakor zelo dobrodošlo. Kot bomo videli v nadaljevanju, se izkaže, da je komunikacija med uporabniškim ter sistemskim prostorom pri pošiljanju manjših paketov precej primerljiva oziroma celo boljša od komunikacije z uporabo vtičnikov, pri pošiljanju večjih paketov pa je v našem primeru bolje izbrati vtičnike.

1.4 Zgradba dela

Tekom dela bomo v poglavju 2.1 predstavili načine medprocesne komunikacije in oba načina komunikacije, ki sta podrobneje opisana v poglavjih 3.2 in 3.3. Predpostavljeno je, da bralec pozna osnove komunikacije med vozlišči v omrežju, vseeno pa je v razdelku 3.1 predstavljen internetni paket. V razdelkih 3.3.2 in 3.3.3 si lahko bralec prebere o Linuxovih modulih in gonilnikih, v 3.3.4 pa o pomnilniški preslikavi datotek. Nenazadnje je v poglavju 4.1 predstavljeno praktično delo tekom izdelave diplomske naloge, v poglavju 4.2 pa so predstavljeni rezultati testiranj.

Delo in primerjave so potekali na operacijskem sistemu Scientific Linux 5, ki je posebna distribucija Linuxa Red Hat. Operacijski sistem je vseboval jedro verzije 2.34.7. Programi, ki so služili testiranju, so bili napisani v programskem jeziku C.

Poglavje 2

Predstavitev načinov komunikacije

2.1 Medprocesne komunikacije

Proces je instanca izvajajočega programa. Vsak tekoči proces vsebuje programsko kodo, podatkovne vrednosti shranjene v programskih spremenljivkah, ter ostale vrednosti, kot so vrednosti registrov, programskega sklada in podobno. Procesi, ki tečejo na računalniku, imajo možnost komunikacije med seboj. V nadaljevanju bomo videli, da procesoma, ki med seboj komunicirata, ni potrebno, da tečeta na istem sistemu. Obstaja več načinov medprocesne komunikacije (*angl.* Inter Proces Communication - IPC):

- **Komunikacija z uporabo signalov** (*angl.* Signals): Je najstarejši način komunikacije med procesi v operacijskih sistemih Unix. Uporablja se jih za asinhrono signaliziranje enemu ali več procesom. Signal lahko povzročijo prekinitve, na primer zaradi pritisnjene tipke na tipkovnici, lahko pa je signal posledica napake, ker je proces posegel v nedovoljen prostor v pomnilniku. Signali se uporabljajo tudi za sporočanje med procesi. Ko proces A zaključi z delom in so rezultati njegovega dela pomembni za proces B, lahko proces A pošlje signal procesu B, da je končal z delom.

- **Pipe** (*angl.* Pipes): Ponavadi se jih uporablja, da se izhod enega programa pripelje na vhod drugega programa, brez potrebe po dodatnemu hranjenju teh podatkov. Uporabnik lahko pipe enostavno uporablja v konzoli Linux operacijskega sistema:

  ```
  user@linux_box$ ls | grep xy
  ```

  V tem primeru je uporabnik podal ukaz za izpis datotek v trenutnem direktoriju ter z uporabo pipe te podatke posredoval programu 'grep', ki med prejetimi podatki izpiše le tiste, ki vsebujejo niz 'xy'.

- **Vtičniki** (*angl.* Sockets): Internetni vtičnik je končna točka (*angl.* endpoint) dvosmernega toka podatkov za komuniciranje med procesi preko omrežja. Vtičniki so glavni način za komuniciranje z oddaljenimi računalniki. Programi, kot so *telnet*, *ftp* strežniki in odjemalci, *http* strežniki in brskalniki, uporabljajo vtičnike. Da lahko vtičnike uspešno implementiramo, moramo poznati lokalni naslov (IP - Internetni Protokol) ter številko priključka (*angl.* port). Več o vtičnikih v razdelku 2.2.

- **Skupni (deljeni) pomnilnik** (*angl.* Shared memory): Zanimivost skupnega pomnilnika je, da lahko delimo pomnilnik med več procesi. Procesi enakopravno dostopajo do istega rezerviranega segmenta v pomnilniku. Deljen pomnilnik je torej pomnilnik, do katerega dostopa več procesov, in na ta način komunicirajo med seboj. Slabost deljenega pomnilnika je v tem, da se lahko uporablja le med uporabniškimi procesi. Za komunikacijo med uporabniškim ter sistemskim procesom je torej potrebno poseči po drugačnem mehanizmu.

- **Pomnilniško preslikane datoteke** (*angl.* Memory mapped files): Pomnilniška preslikava je edini način izmenjave podatkov med uporabniškim in sistemskim prostorom, ki ne vključuje eksplicitnega kopiranja podatkov in je najhitrejši način za obravnavanje velikih količin podatkov. Pomnilniško preslikana datoteka je segment navideznega pomnilnika, ki je običajno povezan z neko datoteko na trdem disku, lahko pa je tudi naprava, skupni pomnilnik ali katerikoli drug resurs, do katerega lahko operacijski sistem dostopa preko deskriptorja datoteke. Ko je segment navideznega pomnilnika rezerviran, lahko aplikacija do njega dostopa, kot bi bil njen primarni pomnilnik. Eden izmed razlogov za uporabo pomnilniško preslikanih datotek je deljenje pomnilnika med procesi. V modernih operacijsih sistemih (Windows in izvedenke Unix operacijskih sistemov) se procesom ne dovoli istočasne uporabe istega pomnilniškega prostora. Obstaja seveda mnogo načinov za reševanje te pomanjkljivosti. Pomnilniško preslikane datoteke so ene izmed najbolj pogostih. Dva ali več procesov lahko istočasno preslika eno fizično datoteko v pomnilnik ter nato dostopa do tega pomnilnika. Več o uporabi pomnilniško preslikanih datotek v razdelku 3.3.4.

Obstajajo še ostali načini medprocesne komunikacije, o katerih si lahko bralec več prebere v [2] in [3].

2.2 Vtičniki

Vtičnik je torej posrednik za medprocesne komunikacije preko interneta. Kakršnekoli vhodno-izhodne operacije, ki se dogajajo v Unix sistemih, kakršen je tudi Linux, potekajo z uporabo datotek ali, bolje rečeno, z uporabo deskriptorjev datotek (*angl.* file descriptor). Deskriptor datoteke je celo število, ki se nanaša na odprto datoteko in se uporabljajo za vsa vhodno-izhodna opravila, pa naj bo to mrežna povezava, pipe (*angl.* pipes) ali pa lokalna datoteka na trdem disku.

2.2.1 Tipi vtičnikov

Poznamo več vrst internetnih vtičnikov, vendar se bomo tu omejili le na dva tipa. Procesi, ki želijo komunicirati preko omrežja, lahko komunicirajo na dva načina. Prvi način je z uporabo povezano orientiranega modela (*angl.* connection oriented model), katerega uporabljajo procesi, ki morajo poslati neprekinjen tok podatkov na neko oddaljeno lokacijo. Primer take povezave

je seja s strežnikom. Drugi način pa je z uporabo nepovezano orientiranega modela (*angl.* connectionless oriented model), kjer proces pošlje sporočilo na nek naslov, takoj za tem pa lahko pošlje sporočilo na nek drug naslov. Oba modela lahko dobro primerjamo z vsakdanjim telefoniranjem, ki je pravzaprav povezan model komunikacije. Primer za nepovezan model pa bi bilo pošiljanje pošte na več naslovov. Za povezano orientiran model se uporablja takoimenovan Transmission Control Protocol ali TCP, za nepovezan model pa User Datagram Protocol ali UDP. Datagram je v bistvu le drug izraz za paket, ki vsebuje neko sporočilo. Če primerjamo oba protokola, lahko ugotovimo, da je TCP protokol zanesljiv glede prispelih paketov, medtem ko je UDP protokol nezanesljiv. Kadar uporabljamo TCP protokol, vemo, da če se zgodi, da nek paket ni prišel na svoj cilj, ga bo njegov pošiljatelj poslal ponovno. Edini način, da paket pri uporabi TCP protokola ne pride na cilj, je, če se celotna povezava onemogoči. Pri UDP protokolu pa pošiljatelj ne more vedeti, ali je njegov paket dosegel cilj ali ne. Možno je, da se je na poti izgubil. Pri uporabi TCP protokola obstaja vrstni red paketov, medtem ko ga pri UDP protokolu ni. Če pošljemo več paketov z uporabo UDP protokola, ne moremo vedeti, v kakšnem vrstnem redu bodo prispeli na cilj, razen seveda če vanje ne shranimo zaporednih številk ter jih na cilju preberemo. Torej, kadar komuniciramo s pomembnimi podatki, je bolje poseči po TCP protokolu, saj je zanesljivejši, toda kadar je naša glavna skrb hitrost, posežemo po UDP protokolu, saj je precej hitrejši. Glede na to, da je naš namen primerjava hitrosti med dvema načinoma komunikacije, smo torej izbrali komunikacijo z uporabo UDP protokola.

2.3 Komunikacija med uporabniškim in sistemskim prostorom

V operacijskem sistemu Linux, kot tudi v ostalih operacijskih sistemih, so ukazi ločeni na sistemske ter na uporabniške ukaze. Pod ukaze sistemskega prostora spadajo vsi ukazi, ki neposredno uporabljajo strojno opremo. Uporabnik neposredno ne more podajati ukazov mrežnemu vmesniku, CD-ROM enoti ali ostali strojni opremi, lahko pa pošlje ukaz v jedro operacijskega sistema ter tako poda prošnjo, ki se izvede znotraj sistemskega prostora. Za take uporabnikove ukaze skrbijo gonilniki, ki tečejo znotraj sistemskega prostora ter zato lahko pošiljajo ukaze na strojno opremo. Torej, gonilniki so vmesniki med uporabniškim prostorom ter strojno opremo. Za komunikacijo med uporabniškim prostorom ter sistemskim prostorom bomo izbrali komunikacijo z uporabo preslikanega pomnilnika oziroma, bolje rečeno, pomnilniško preslikane datoteke.

Z uporabo tega načina bosta sistemski in uporabniški prostor pisala in brala v/iz iste datoteke ter na ta način komunicirala. Oba morata seveda vedeti, kdaj lahko v datoteko pišeta in kdaj so v njej pripravljeni podatki za branje. Kot vidimo, je stvar nekoliko bolj zapletena kot pri uporabi vtičnikov. Bralec si lahko podrobneje o pomnilniški preslikavi datotek ter sporočanju pripravljenosti podatkov v datoteki prebere v razdelkih 3.3.4 in 3.3.5.

Poglavje 3

Opis uporabljenih tehnologij

3.1 Internetni paket

3.1.1 Splošno

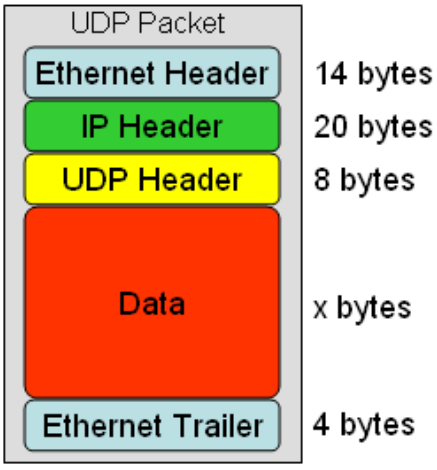
Kadarkoli imamo opravka s komunikacijo preko interneta, pravzaprav komuniciramo z uporabo paketov. Kadar na primer uporabljamo internetni brskalnik in si želimo ogledati neko stran na internetu, se mora stran prenesti na naš računalnik, da jo brskalnik lahko prikaže. Podatki, ki opisujejo internetno stran, prispejo na naš računalnik v zaporedju paketov. Paketi so enakomerno razdeljeni podatki, ki jih gonilnik preda mrežnemu vmesniku in se nato kot zaporedje ničel in enic pošljejo preko komunikacijskega medija. Na prejemnikovi strani se nato paketi zopet sestavijo v prvotne, brskalniku razumljive podatke.

3.1.2 Enkapsulacija paketa

Preden lahko paket pošljemo, ga je najprej potrebno pravilno sestaviti. Kar se tiče uporabnika, je dovolj, da navede podatke, ki jih želi poslati. Uporabniku ni potrebno skrbeti za velikost podatkov ali fragmentacijo - razbitje podatkov na manjše pakete. Enkapsulacija paketov je za uporabnika aplikacije, ki pošilja podatke, transparentna, nekoliko manj transparentna za razvijalca aplikacije ter popolnoma netransparentna za razvijalca mrežnega gonilnika. Razvijalec aplikacije mora poznati različne protokole, ki so lahko uporabljeni pri vzpostavljanju komunikacije, torej UDP ali TCP, poznati mora internetni naslov prejemnika in pošiljatelja ter priključek, ki se uporabi pri kreiranju vtičnika. Ni pa mu potrebno skrbeti za fragmentacijo in enkapsulacijo podatkov. Razvijalec mrežnih gonilnikov mora poznati celoten postopek fragmentacije in

enkapsulacije paketov. Enkapsulacija paketov je postopek pravilne priprave paketa za pošiljanje. Podatkom je potrebno dodati informacije, ki so potrebne za pravilno pošiljanje. Informacijam, ki se podatkom dodajo, se reče glava (*angl.* header). V postopku pripravljanja paketa se paketu dodajo tri glave ter rep:

- UDP glava (*angl.* UDP header): Podatkom, ki pridejo od uporabnika, se najprej doda UDP glava, ki je velika 8 bajtov. UDP glavo sestavljata prejemnikov in pošiljateljev priključek, kjer je vsak velik po 4 bajte.
- IP glava (*angl.* IP header): IP glava je velika 20 bajtov in med drugim vsebuje tip protokola, pošiljateljev in prejemnikov IP, verzijo IP protokola (npr. IPv4, IPv6).
- Ethernet glava (*angl.* Ethernet header): Vsebuje prejemnikov in pošiljateljev MAC naslov mrežnega vmesnika (enoličen identifikator mrežnega vmesnika (*angl.* Media Access Control address)), kar zasede 12 bajtov ter 2 bajta za določitev enkapsuliranega protokola (npr. IPv4, IPv6, ARP). Skupaj torej 14 bajtov.
- Ethernet rep (*angl.* Ethernet footer/trailer): Rep je velik 4 bajte in vsebuje CRC (ciklično preverjanje redundance (*angl.* Cyclic Redundancy Check)) in je metoda za odkrivanje napak pri prenosu paketov.



Slika 3.1: Zgradba UDP paketa

Vsak UDP paket je sestavljen tako, kot kaže Slika 3.1. Vse tri glave so potrebne, da paket prispe do pravilnega prejemnika ter do želenega procesa. Pod *prejemnik* mislimo računalnik, na katerem teče proces, s katerim želimo komunicirati. Prejemnik je definiran z Ethernet glavo, ki vsebuje MAC naslov prejemnikovega mrežnega vmesnika, ter IP naslovom, ki je shranjen v IP glavi. Da na prejemnikovi strani paket prispe do pravilnega procesa, pa potrebujemo tudi številko priključka, preko katerega proces komunicira. Ta številka je shranjena v UDP glavi. Med potovanjem paketa po prenosnem mediju pa lahko pride do napak paketa, kar pomeni, da lahko do prejemnika pride s poškodovanimi podatki. Za preverjanje pravilnosti paketa je uporabljen Ethernet rep, v katerem je shranjeno 32-bitno (4 bajti) število, ki je izračunano na podlagi podatkov v paketu. Ko prejemnik prejme paket, sam izračuna to 32-bitno število na enak način, kot je to naredil pošiljatelj. Nato izračunano število primerja s shranjenim, in če sta enaki, pomeni, da so podatki nepoškodovani. V nasprotnem primeru je paket poškodovan in se lahko zavrže.

3.1.3 Velikost paketa in fragmentacija

Kot omenjeno v razdelku 3.1.2, mora razvijalec mrežnih gonilnikov poleg enkapsulacije paketa poznati tudi njegovo fragmentacijo. Velikost UDP paketa, ki ga gonilnik lahko pošlje na mrežni vmesnik, je omejena na 1522 bajtov. To se zdi za čas 10 gigabitnih povezav precej malo, in tudi je. Toda ne smemo pozabiti, da so *ethernet* povezave prisotne že dolgo časa. Povezave so bile, zgodovinsko gledano, precej počasnejše ter precej manj zanesljive kot danes. Poleg tega je *ethernet* obratno združljiv (*angl.* backward compatible). Med seboj, na primer, lahko povežemo mrežni vmesnik, ki omogoča 10-gigabitne povezave z vmesnikom, ki je zmožen le 10 megabitnih. Vse glave ter rep, ki so opisani v razdelku 3.1.2, zasedejo 46 bajtov. Za podatke, ki jih želimo poslati, ostane 1476 bajtov. Gonilnik mora poskrbeti, da podatke, prejete iz uporabniškega procesa, pravilno fragmentira v manjše pakete, jim doda glave ter rep in jih nato postavi v vrsto za pošiljanje na mrežni vmesnik. Na tem mestu bi bilo vredno omeniti, da paketom, ki jih gonilnik dobi s fragmentacijo, ne moremo več reči paketi, saj so v bistvu le del paketa. Za te manjše pakete oziroma fragmente se je ustalil izraz okvir (*angl.* frame). Čeprav gonilnik poskrbi za pravilno fragmentacijo od uporabniškega procesa prejetih podatkov, pa ti podatki tudi ne smejo biti neskončno veliki. Največja dovoljena velikost paketa podatkov, prejetega iz uporabniškega prostora, je 65507 bajtov. Skupaj z UDP in IP glavo je celoten paket podatkov velik 65535 bajtov.

Če omejitve velikosti podatkovnih paketov ne bi bilo, bi na prejemnikovi strani lahko prišlo do težav. Kot rečeno, oddajnikov gonilnik pakete fragmentira tako, da okvirji skupaj z glavami ne presegajo 1522 bajtov. Na prejemnikovi strani pa mora gonilnik prejete okvirje pravilno združiti. Preden začne prejemnik posamezne okvirje združevati, počaka, da prispejo vsi. Zaradi omejenih velikosti notranjih spremenljivk (16 bitov) lahko prejemnik sprejme le skupno velikost 65535 bajtov vseh poslanih okvirjev. Če bi poskusili obiti omejitev velikosti, bi lahko prišlo do sesutja prejemnikovega sistema, vnovičnega zagona sistema in podobnih težav.

3.2 Implementacija vtičnikov

3.2.1 Vtičniki

Kot omenjeno, se bomo osredotočili na UDP protokol, zato bomo na tem mestu pogledali, kako implementirati vtičnike za uporabo pri komunikaciji z UDP protokolom. Da lahko v aplikaciji uspešno odpremo vtičnik, moramo vedeti, kakšen protokol bomo uporabljali, kakšen IP naslov uporablja pošiljatelj ali prejemnik paketov ter priključek (*angl.* port), ki se bo uporabljal za pošiljanje in prejemanje paketov. Ko zberemo vse navedene informacije, lahko kreiramo vtičnik. Deskriptor vtičnika dobimo tako, da opravimo sistemski klic *socket*. Le-ta vrne deskriptor, s pomočjo katerega komuniciramo z uporabo ukazov *send*, *recv* ter ostalih. Uporabljali bi lahko tudi običajne *read* in *write* ukaze, ki jih sicer uporabljamo za branje in pisanje v datoteke, toda ukazi *send* in *recv* ponujajo boljši nadzor nad prenosom podatkov.

3.2.2 IOCTL

V poglavju 2.1, kjer smo govorili o načinih medprocesne komunikacije, smo omenili tudi signale. Signali nam pridejo prav, kadar želimo nek proces asinhrono obvestiti o nekem dogodku. Način uporabe signalov za obveščanje o specifičnem dogodku je sistemski klic *ioctl*. Ime pride iz *input/output control* oziroma kontrola vhoda in izhoda. Kot že ime pove, se sistemski klic uporabja za nastavitev signalov za obveščanje glede vhodnih in izhodnih operacij. V našem primeru želimo biti obveščeni takoj, ko računalnik prejme nek paket. Zato smo skupaj s sistemskim klicem *ioctl* uporabili signal SIOCSPGRP, ki se proži vedno, ko pride do spremembe na vtičniku, torej tudi ob prispetju novega paketa. Ko ugotovimo, kateri signal moramo uporabiti za določen dogodek, moramo tudi definirati funkcijo, ki se izvrši ob prejetju signala. To

naredimo s sistemskim klicem *signal*. V sistemski klic navedemo naslov naše funkcije, ki obravnava prejetje signala. V funkciji nato obravnavamo prejeti paket, izračunamo čas potovanja paketa ter pošljemo povratni paket.

3.3 Implementacija komunikacije med uporabniškim in sistemskim prostorom

3.3.1 Gonilniki v Linux operacijskem sistemu

Gonilnik je program na najnižjem nivoju, ki teče na računalniku in neposredno komunicira s strojno opremo. Gonilniki naprav imajo posebno vlogo v Linux opracijskih sistemih. Delujejo kot črne škatle, tako da uporabniku ni potrebno vedeti, kako naprava, za katero so gonilniki napisani, deluje. Gonilniki so napisani tako, da jih lahko uporabnik uporablja neodvisno od tipa strojne opreme. Če vzamemo za primer mrežne vmesnike, vemo, da obstaja malo morje različnih proizvajalcev. Bilo bi precej nerodno, če bi moral uporabnik za vsak mrežni vmesnik uporabljati svoje ukaze. Zato so gonilniki napisani na standardiziran način, kar uporabniku omogoča enak način uporabe vsakega mrežnega vmesnika. Naloga gonilnika pa je, da uporabnikove ukaze pravilno interpretira za uporabo na dotični strojni opremi. Izgradnja gonilnikov je lahko ločena od izgradnje jedra operacijskega sistema in se jih, ko so zgrajeni, enostavno vključi vanj kot tako imenovane Linux module.

3.3.2 Linux moduli

Module je možno vključiti v operacijski sistem, medtem ko sistem teče (*angl.* at runtime). To pomeni, da medtem ko je sistem vzpostavljen, lahko dodajamo (ali odstranjujemo) funkcionalnosti operacijskega sistema. Vsak del kode, ki jo lahko med vzpostavljenostjo operacijskega sistema dodajamo ali odstranjujemo, se imenuje modul. V Linuxu obstajajo tri skupine modulov, izmed katerih vsak opisuje svojo skupino naprav:

- **Znakovne naprave** (*angl.* character devices): Znakovne naprave se pravzaprav lahko obravnava kot datoteke. Do njih se dostopa s tokom podatkov (bajt po bajt), za kar poskrbi znakovni gonilnik. Tak gonilnik ponavadi implementira vsaj *open*, *close*, *read* in *write* sistemske klice. Primeri za znakovne naprave so tekstovna konzola in serijski priključki (npr. USB). Datoteke, kot so /dev/tty0 in /dev/lp0, predstavljajo uporabnikov dostop do naprav.

- **Blokovne naprave** (*angl.* block devices): Sem spadajo naprave, na katerih je možno vzpostaviti datotečni sistem (npr. trdi diski). Do takih naprav se dostopa z uporabo datotek, kot so /dev/hda1. Od znakovnih naprav se razlikujejo po tem, da se do njih dostopa z bloki podatkov (pogosto z bloki po 512 ali 1024 bajtov) namesto s tokom podatkov.

- **Mrežni vmesniki** (*angl.* network interfaces): Do mrežnih vmesnikov se ne dostopa z uporabo datotek naprav, kot smo to videli pri znakovnih ali blokovnih napravah. Mrežnemu vmesniku se vseeno dodeli ime (npr. eth0), toda ime ni preslikano v datotečni sistem, kot pri znakovnih napravah. Namesto *read* in *write* klicev mora zato jedro operacijskega sistema klicati pravilne funkcije, ki so povezane s prenašanjem paketov. Temu bi lahko kdo nasprotoval in rekel, da aplikacije uporabljajo standardne *read* in *write* ukaze nad vtičniki, toda ti klici delujejo nad programskimi objekti, ki so ločeni od mrežnih vmesnikov, torej *read* in *write* ukazov ne moremo neposredno uporabljati nad vmesniki.

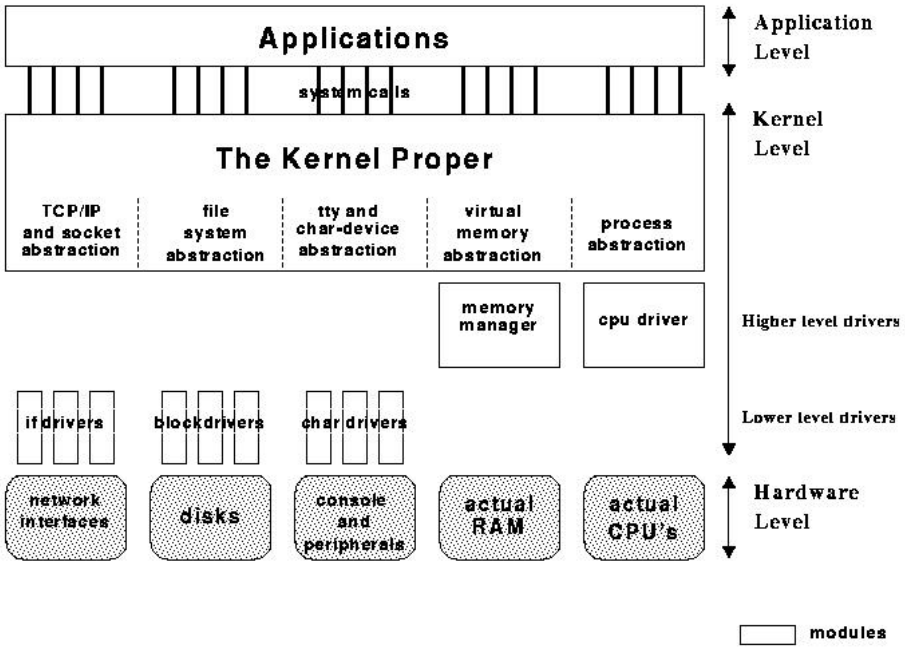
Slika 3.2 prikazuje jedro operacijskega sistema Linux ter uporabniški prostor, ki komunicira s sistemskim prostorom preko sistemskih klicev. Na spodnjemu delu slike lahko vidimo vse tri module, ki opisujejo tri skupine naprav, o katerih smo govorili. V Linuxu obstajata dva programa, katerih naloga je dodajanje ali odstranjevanje modulov. Z uporabo programa *insmod* modul povežemo v operacijski sistem, s pomočjo programa *rmmod* pa modul odstranimo. Da lahko programa uporabljamo, moramo imeti administratorske (root) pravice. Zopet si poglejmo primer na mrežnih gonilnikih. Če želimo obstoječemu mrežnemu gonilniku dodati preprosto funkcionalnost, da vsakič, ko mrežni vmesnik prejme nek paket, gonilnik izpiše opozorilo, je v gonilniku najprej potrebno pravilno spremeniti funkcijo, ki skrbi za take dogodke. Ko modul uspešno zgradimo je potrebno prejšnji, nespremenjeni modul odstraniti z uporabo programa *rmmod*:

```
root@linux_box# rmmod ime_modula
```

Nato novi modul naložimo z uporabo programa *insmod*:

```
root@linux_box# insmod ime_modula
```

Mrežni gonilnik sedaj ob vsakem prejetju paketa izpiše opozorilo.



Slika 3.2: Jedro operacijskega sistema Linux

3.3.3 Linuxov mrežni gonilnik

Ko smo predstavljali različne vrste naprav v Linux operacijskem sistemu, smo omenili, da se mrežni vmesniki od preostalih dveh naprav nekoliko razlikujejo. Najpomembnejša razlika pa je, da se preostala dva tipa naprav odzivata le na ukaze in dogodke iz jedra operacijskega sistema, medtem ko mrežni gonilniki dobivajo pakete asinhrono od zunaj. Mrežni gonilniki morajo biti sposobni nastavljanja naslovov in modifikacij parametrov, povezanih s prenašanjem paketov, z vzdrževanjem prometa in odzivov na napake. API za mrežne gonilnike ponuja prav to in si ga bomo v nadaljevanju podrobneje pogledali. Mrežni podsistem Linuxovega jedra je narejen tako, da je popolnoma neodvisen od uporabljenega protokola. To velja za mrežne protokole (Internetni Protokol, IPX in ostali) ter za strojne protokole (Ethernet, obroč z žetoni (*angl.* token ring)).

Registracija naprav

Ko gonilnikov modul naložimo v jedro operacijskega sistema, modul zahteva sredstva (npr. mrežni vmesnik), ki jih nameravamo uporabiti. Gonilnik to naredi tako, da za vsako na novo zaznano napravo v seznam mrežnih naprav vstavi svojo podatkovno strukturo. Vsak vmesnik je nato opisan v strukturi, ki se imenuje *net_device*. Kot podobne strukture mora biti tudi *net_device* alocirana dinamično. Funkcija, ki se jo za to lahko uporabi, se imenuje *alloc_netdev* in je definirana z naslednjim prototipom:

```
struct net_device *alloc_netdev(int sizeof_priv,
                                const char *name,
                    void (*setup)(struct net_device *));
```

Argument *sizeof_priv* je gonilnikov prostor za 'osebne podatke'. Pri mrežnih napravah je ta prostor rezerviran skupaj s strukturo *net_device*. Argument *name* je ime naprave, kakor jo vidijo aplikacije uporabniškega prostora (eth0, eth1 in podobno). Zadnji argument funkcije, *setup*, je kazalec na inicializacijsko funkcijo, ki do konca nastavi strukturo *net_device*. Mrežni podsistem predstavlja veliko inačic funkcije *alloc_netdev*, ki so prilagojeni za posamezne tipe vmesnikov. Eden najbolj pogostih in tudi tisti, ki je bil uporabljen v našem mrežnem gonilniku, se imenuje *alloc_etherdev*. Prototip funkcije je naslednji:

```
struct net_device *alloc_etherdev(int sizeof_priv);
```

Tako kot funkcija *alloc_netdev*, tudi *alloc_etherdev* vrača kazalec na strukturo tipa *net_device*. Ko je naprava inicializirana ter polja strukture *net_device* nastavljena, je potrebno še registrirati napravo. To se opravi s klicem funkcije *register_netdev*, katere prototip izgleda tako:

```
register_netdev(struct net_device);
```

Registracija naprave je lahko opravljena šele, ko je inicializacija naprave končana, saj lahko začne gonilnik izvajati operacije na napravi takoj po registraciji.

Struktura net_device

Struktura *net_device* ima v mrežnem gonilniku pomembno vlogo in si zasluži nekoliko podrobnejšo razlago nekaterih njenih polj, katerih poznavanje bomo v nadaljevanju potrebovali:

- netdev→name: Znakovna tabela, kjer je definirano ime naprave. Ime je navadno oblike eth%d, kjer je %d zaporedno število inicializirane naprave.

- netdev→tx_queue_len: Polje vsebuje največje dovoljeno število okvirjev, ki se lahko naberejo v vrsti za pošiljanje.
- netdev→dev_addr: Vsebuje MAC naslov naprave. Naslov je dolg 6 oktetov in se uporablja za generiranje ethernet glave pri enkapsulaciji paketa.
- netdev→trans_start: Vsebuje čas začetka prenosa okvirja.
- netdev→watchdog_timeo: Vsebuje časovno periodo, ki mora preteči od začetka pošiljanja okvirja, da gonilnik predvidi napako.
- netdev→netdev_ops: Vsebuje naslov strukture *net_device_ops*, ki vsebuje kazalce na takoimenovane povratne funkcije (*angl.* callback functions). Te funkcije se izvajajo ob določenih dogodkih, na primer ob inicializaciji naprave, ob prejemu novega okvirja in podobno.

Struktura sk_buff

Ime strukture pride iz okrajšave besedne zveze *socket buffer*. Čeprav ni nujno, da se struktura kakorkoli nanaša na vtičnike, se je ta besedna zveza ustalila. Struktura *sk_buff* je opis okvirja, ki ga gonilnik pošlje na komunikacijski medij. Da lahko gonilnik to naredi, je potrebno polja strukture *sk_buff* pravilno nastaviti. Najprej je potrebno rezervirati prostor za strukturo. Ponavadi za strukturo izberemo ime *skb*. To naredimo s klicem:

```
skb = dev_alloc_skb(paysize);
```

paysize je celo število, ki pove število bajtov podatkov vseh podatkov, ki bodo shranjeni v strukturi *skb*. Med te podatke spadajo vse glave (IP, UDP in ethernet) in uporabnikovi podatki. Ko je prostor rezerviran, moramo strukturo napolniti s podatki. Kot kaže Slika 3.1, je potrebno na vrh okvirja vstaviti ethernet glavo, pod njo IP glavo, tik nad podatki pa UDP glavo. Vsako glavo je potrebno najprej pravilno nastaviti, nato pa jo vključiti v *skb* strukturo. Nastavljanje posameznih struktur:

Ethernet glava:

```
eth->h_proto=htons(ETH_P_IP);
eth->h_source[0]=source[0];
eth->h_source[1]=source[1];
eth->h_source[...]=source[...];
eth->h_source[5]=source[5];
eth->h_dest[0]=destination[0];
```

```
eth->h_dest[1]=destination[1];
eth->h_dest[...]=destination[...];
eth->h_dest[5]=destination[5];
```

Najprej nastavimo tip enkapsuliranega protokola, nato sledita pošiljateljev ter prejemnikov MAC naslov.

IP glava:

```
iph->version=4;
iph->ihl=5;
iph->tos=0;
iph->frag_off=0;
iph->ttl=20;
iph->protocol=IPPROTO_UDP;
iph->saddr=sourceIP;
iph->daddr=destinationIP;
```

IP glavi moramo nastaviti verzijo IP protokola, ki je lahko 4 ali 6. Polje *ihl* predstavlja velikost IP glave, in sicer v vrsticah po 4 bajte oziroma po 32 bitov. Normalno je IP glava velika 20 bajtov, obstajajo pa tudi druge različice IP glave. V našem primeru bomo uporabili standardno IP glavo, zato je vrednost polja *ihl* enaka 5. Polje *tos - type of service* označuje prioriteto paketa. Število 0 označuje normalno prioriteto. Polje *ttl - time to live* označuje število 'skokov' paketa od enega vozlišča do drugega. Ob vsakem 'skoku' se število zmanjša. S tem se rešuje neskončne zanke skokov paketov. Če število pride do 0, se paket zavrže. Sledi polje za opis tipa protokola, ki je v našem primeru UDP, ter pošiljateljev ter prejemnikov IP naslov.

UDP glava:

```
uh->source=htons(S_PORT);
uh->dest=htons(D_PORT);
```

UDP glavi moramo pravilno nastaviti pošiljateljev ter prejemnikov priključek. Ko glave pravilno nastavimo, je potrebno glave in podatke vstaviti v *skb* strukturo:

```
eth=(struct ethhdr*)skb_push(skb,sizeof(struct ethhdr));
iph=(struct iphdr*)skb_put(skb,sizeof(struct iphdr));
uh=(struct udphdr*)skb_put(skb,sizeof(struct udphdr));
payload=(u_char*)skb_put(skb,paysize);
```

Na ta način rezerviramo strukture, nato pa jih vključimo v strukturo *skb*:

```
skb_reset_mac_header(skb);
skb_set_network_header(skb, sizeof(struct ethhdr));
skb_set_transport_header(skb, sizeof(struct iphdr));
```

Ko izvršimo vse opisane ukaze, je naš UDP sklad sestavljen in pripravljen za pošiljanje. Če bi želeli do podatkov našega okvirja dostopati (npr. ob vsakemu prejetju okvirja izpisati njegove podatke), bi morali podrobneje preučiti polja strukture *skb*. Glede na to, da smo predstavili glave ter podatke strukture *skb*, opišimo torej, v katera polja strukture se shranijo:

- skb→mac_header: V to polje strukture *skb* se shrani ethernet glava UDP okvirja.
- skb→network_header: Sem se shrani IP glava okvirja.
- skb→transport_header: Tu je shranjena UDP glava.
- skb→data: Podatki okvirja, shranjeni pod vsemi glavami.

Navedena polja niso vsa, ki zasedajo strukturo *skb*. Če želi bralec izvedeti več, tako o *skb* kot o *net_device* strukturi, si lahko prebere [1]. Še več podrobnosti pa lahko bralec izve s pregledom izvorne kode Linux jedra, ki si ga lahko pretoči iz [6]. Struktura *skb* je definirana v ⟨linux/skbuff.h⟩, struktura *net_device* pa v ⟨linux/netdevice.h⟩.

Pošiljanje paketov

Najbolj pomembni nalogi mrežnega vmesnika sta pošiljanje in prejemanje paketov. Kadarkoli želi jedro operacijskega sistema poslati nek okvir, mora poklicati gonilnikovo *hard_start_transmit* funkcijo, da doda okvir v vrsto odhajajočih okvirjev. Vsak okvir, ki ga jedro želi poslati, je vsebovan v strukturi *sk_buff*. Kadar funkcija okvir uspešno vstavi v vrsto, vrne 0, kadar pa vstavljanje okvirja v vrsto ni uspelo, pa vrne od nič različno število. Funkcija vrne vrednost šele potem, ko programska oprema konča s podajanjem ukazov strojni opremi glede prenosa okvirjev. Mrežni vmesnik pošilja okvirje preko medija asinhrono in ima omejeno količino pomnilnika za odhajajoče okvirje. Ko je pomnilnik zapolnjen, mora gonilnik nehati sprejemati prošnje za pošiljanje novih okvirjev. Funkcija *hard_start_transmit* mora skrbeti za to, da pravočasno ugotovi, če je pomnilnik mrežnega vmesnika poln ter v skladu s tem tudi ukrepati. Enako mora, ko se pomnilnik mrežnega vmesnika sprosti, ponovno začeti s sprejemanjem prošenj za pošiljanje okvirjev.

Večina gonilnikov, ki se ukvarja z nekimi napravami, mora biti pripravljenih, da se naprava občasno ne bo odzvala. Take težave so pri uporabi osebnih računalnikov precej pogoste, zato morajo biti gonilniki pripravljeni nanje. Večina gonilnikov se s temi težavami spopada z implementacijo časovnikov - če se operacija ni izvršila do konca vnaprej definiranega časa, je predpostavljena napaka. Mrežni gonilniki morajo zato v strukturi *net_device* nastaviti polje *watchdog_timeo*. To polje vsebuje čas, ki mora od začetka pošiljanja okvirja preteči, preden gonilnik zazna napako.

Prejemanje paketov

Obstajata dva načina sprejemanja paketov. En način uporablja prekinitve, drugi način pa izpraševanje (*angl.* polling). Način, ki smo ga uporabili mi, uporablja prekinitve. Ko naš računalnik prejme okvir od pošiljatelja, se zgodi prekinitev. Ob prekinitvi se pokliče gonilnikova funkcija *e1000_receive_skb*, ki obravnava take prekinitve. Naloga te funkcije je, da prejete okvirje vstavi v vrsto prejetih okvirjev. Ko so prejeti vsi, je potrebno vse okvirje iz vrste defragmentirati - sestaviti nazaj v prvoten paket. Ko je paket sestavljen, se lahko na podlagi informacij UDP glave posreduje izven sistemskega prostora v uporabniški prostor.

Nastavitve, opisi funkcij ter klici in strukture, ki smo jih opisali, pa seveda niso vse, kar se v mrežnih gonilnikih pojavi. Zavedati se moramo, da povprečen mrežni gonilnik vsebuje okoli 5000 vrstic kode, zato se zaradi prevelike obsežnosti nismo spuščali v vse podrobnosti. Če bralec želi, si lahko več o mrežnih in ostalih gonilnikih prebere v [1].

3.3.4 Pomnilniško preslikane datoteke

Pomnilniško preslikana datoteka je datoteka, katere vsebina je preslikana v navidezni pomnilnik. Uporablja se za lažje izmenjavanje podatkov med procesi. Bilo bi sila neprijetno, če bi bilo potrebno za izmenjavanje podatkov med procesi vedno znova odpirati in zapirati datoteko. V ta namen datoteko preslikamo v navidezni pomnilnik in zagotovimo dostop do njega vsem sodelujočim procesom. V nadaljevanju si bomo podrobneje pogledali pomnilniško preslikane datoteke ter njihovo implementacijo v uporabniškem in sistemskem prostoru.

Uporabniški prostor

Preden lahko naredimo klic za preslikavo pomnilnika, moramo dobiti deskriptor datoteke z uporabo naslednje funkcije:

```
int open(const char *path, int flags)
```

Argument *path* je pot do datoteke, *flags* pa so pravice, s katerimi datoteko odpiramo. V našem primeru odpremo datoteko s pravicami za branje ter pisanje. Ko imamo deskriptor datoteke, lahko uporabimo funkcijo za pomnilniško preslikavo:

```
void *mmap(void *addr, size_t len,
           int prot, int flags,
           int fildes, off_t off);
```

addr je naslov, v katerega se datoteka preslika. Najboljši način za definicijo tega parametra je, da ga označimo z NULL in pustimo operacijskemu sistemu, da sam določi naslov. Če naslov določimo sami in se operacijski sistem s tem ne 'strinja', klic vrne napako. *len* je velikost v bajtih, ki jo želimo preslikati in jo potem uporabljati. Velikost mora biti mnogokratnik velikosti strani navideznega pomnilnika operacijskega sistema. V nasprotnem primeru funkcija *mmap* vrne napako. Velikost strani navideznega pomnilnika na večini današnjih sistemov je 4 kilobajte. Argument *prot* definira pravice dostopa do preslikane datoteke. Pravica mora biti enaka kot pravice, ki smo jih uporabili za odpiranje datoteke. *flags* so dodatne zastavice, ki jih lahko nastavimo. Glede na to, da želimo uporabljati preslikano datoteko za deljenje informacij med procesi, tu nastavimo zastavice na MAP_SHARED. Če bi želeli, da je preslikana datoteka zasebna, bi zastavice nastavili na MAP_PRIVATE. *filedes* je deskriptor datoteke, ki smo ga dobili s klicem *open*. *offset* je zamik v datoteki, od kje naprej jo želimo preslikati. Če pride pri izvajanju klica *mmap* do napake, funkcija vrne -1, v nasprotnem primeru pa vrne kazalec na preslikani naslov.

Sistemski prostor

V sistemskem prostoru je potrebnega malo več dela. Potrebno je pripraviti metode za preslikavo datotek. Le-te se kličejo, ko iz uporabniškega prostora podamo določeno zahtevo glede preslikave (npr. klic funkcije mmap). Da je sistemski prostor obveščen, kateri klic se je v uporabniškem prostoru pravkar zgodil, je uporabljena struktura tipa *vm_operations_struct*, ki vsebuje imena povratnih funkcij. Struktura izgleda tako:

```
struct vm_operations_struct mmap_vm_ops = {
        .open   = mmap_open,
        .close  = mmap_close,
        .fault  = mmap_fault
};
```

Poglejmo si, kaj vsaka izmed implementiranih funkcij naredi:

mmap_open in *mmap_close*
Funkciji pravzaprav služita le sledenju dogodkov, kolikokrat smo izvršili preslikavo. Funkcija *mmap_open* to število poveča, *mmap_close* pa zmanjša.

mmap_fault
Funkcija naredi dejansko preslikavo datoteke v pomnilnik, katerega si potem delita sistemski in uporabniški prostor. Najpomembnejša funkcija, ki se jo kliče iz *mmap_fault*, je *virt_to_page*, ki rezervira prostor v navideznem pomnilniku. Funkcija vrne kazalec na strukturo *struct page*, ki vsebuje naslov do rezervirane strani navideznega pomnilnika. Funkcija *mmap_fault* se izvrši le, ko želi uporabniški prostor prvič dostopati do preslikanega pomnilnika.

Za operacije, ki jih izvajamo nad preslikanim pomnilnikom, skrbijo funkcije, ki so navedene v strukturi tipa *file_operations*:

```
static const struct file_operations my_fops = {
        .open    = my_open,
        .release = my_close,
        .mmap    = my_mmap,
        .read    = my_read,
        .write   = my_write,
        .fasync  = my_fasync
};
```

Struktura vsebuje funkcije, ki izvajajo operacije nad različnimi vhodno/izhodnimi strukturami, naj bo to datoteka, preslikan pomnilnik ali podobne strukture.

my_open
V tej funkciji rezerviramo pomnilnik, ki se bo kasneje delil med sistemskim in uporabniškim procesom.

my_close
Funkcija sprosti pomnilnik, ki smo ga rezervirali v funkciji *my_open*.

my_mmap
S pomočjo te funkcije pokažemo na našo implementacijo strukture tipa *vm_operations_struct*, torej v našem primeru na strukturo *mmap_vm_ops*.

my_read
Funkcija se izvrši vsakokrat, ko želi proces uporabniškega prostora nekaj prebrati iz pomnilniško preslikane datoteke. Torej, ko prispe nov okvir in ga želi sistemski proces posredovati uporabniškemu procesu, le-ta prebere pomnilniško preslikano datoteko. Ko uporabniški proces izvrši funkcijo za branje, se v sistemskem prostoru izvrši funkcija *my_read*.

my_write
Podobno kot *my_read*, se *my_write* izvrši vedno, ko želi uporabniški prostor pisati v pomnilniško preslikano datoteko. V tej funkciji postanejo podatki, ki jih je želel uporabniški prostor posredovati sistemskemu, na voljo sistemskemu prostoru. V našem primeru je ta funkcija tudi tista, v kateri se pridobljeni podatki enkapsulirajo v okvir ter posredujejo funkciji za pošiljanje okvirja.

my_fasync
Funkcija *my_fasync* sicer nima nič opraviti s pomnilniško preslikanimi datotekami, ampak se uporablja za nastavljanje signalizacije uporabniškemu prostoru.

Konsistentnost podatkov

Če bi več procesov želelo dostopati in spreminjati podatke preslikane datoteke istočasno, bi prišlo do nekonsistentnosti podatkov. V tem primeru bi bilo potrebno uvesti zaklepanje datotek, semaforje ali pa kakšen podoben način preprečevanja nekonsistentnosti. Ker v našem primeru v danem trenutku teče le ena instanca programa, ki dostopa do preslikane datoteke, in ker sistemski proces in uporabniški proces dostopata do datoteke zaporedno in ne istočasno, do podobnih zapletov ne more priti, zato uporaba navedenih mehanizmov ni bila potrebna.

Podobno kot pri implementaciji vtičnikov, potrebujemo tudi pri implementaciji pomnilniško preslikanih datotek način signaliziranja dogodkov, kot sta branje iz datoteke in pisanje vanjo. V sistemskem prostoru imamo za to na

voljo strukturo *file_operations*. Na ta način je proces, ki teče v sistemskem prostoru, obveščen takoj, ko uporabniški proces izvrši branje ali pisanje v datoteko. Za podobno obveščanje v uporabniškem prostoru moramo poseči po drugačnem mehanizmu, saj taka struktura tam ne obstaja. Prav tako ne moremo poseči po že prej omenjenem *ioctl* klicu, saj je ta namenjen izmenjavanju signalov le med uporabniškimi procesi. Obstaja pa podoben sistemski klic, ki je namenjen signaliziranju iz sistemskega prostora v uporabniški prostor. Klic se imenuje *fasync* in pravzaprav deluje na podoben način kot *ioctl*, le da moramo signal poslati sami.

3.3.5 FASYNC

Da lahko uporabljamo signalizacijo iz sistemskega v uporabniški prostor, je potrebno pravilno nastaviti tako uporabniški kot sistemski proces. V uporabniškem prostoru je potrebno izvršiti klic *signal*, s katerim določimo naslov funkcije, ki se mora v uporabniškem prostoru klicati vedno, ko prejme signal iz sistemskega prostora. Potrebno je še določiti, na katere signale se želimo odzvati. To naredimo s klicem *fcntl*. V sistemskem prostoru je potrebno napisati funkcijo, ki se izvede, ko uporabniški proces definira uporabo signalov. To funkcijo smo omenili, ko smo opisovali strukturo *file_operations*. Tam smo jo poimenovali *my_fasync*. Ko je inicializacija končana, je potrebno le še izbrati prostor v programski kodi, od koder želimo signal pošiljati. Ko najdemo pravo mesto, je potrebno le poklicati funkcijo *kill_fasync*. Le-ta pošlje signal, ki ga uporabniški proces 'ulovi'. Signalizacija je bila koristna v primeru prejemanja okvirjev. Ko je sistemski proces - gonilnik prejel okvir ter njegove podatke zapisal v pomnilniško preslikano datoteko, je poslal signal. Uporabniški proces je ob prejetju signala vedel, da so zanj na voljo podatki, ter jih prebral.

Poglavje 4

Potek dela

4.1 Potek implementacije komunikacij

Uporabljeni test

Za izvajanje testov z uporabljenima komunikacijama sta bila uporabljena dva računalnika s popolnoma enako konfiguracijo. Računalnika sta vsebovala centralno procesno enoto Intel Core2Quad Q9400, ki deluje pri frekvenci 2.66 GHz. Oba sistema sta vsebovala 4 GB glavnega pomnilnika. Računalnika sta vsebovala Intelove mrežne vmesnike, ki zmorejo prenašati podatke s hitrostjo 1Gb/s. Za testiranje hitrosti posamezne komunikacije je bil uporabljen test 'roundtrip'. To pomeni, da je eno vozlišče začelo s pošiljanjem paketa, ga poslalo drugemu vozlišču, le-to pa je odgovorilo s pošiljanja paketa nazaj proti prvemu vozlišču. Za lažje razumevanje označimo vozlišče, ki začne s pošiljanjem paketov kot vozlišče A ter drugo vozlišče kot vozlišče B. Pot enega pošiljanja je torej izgledala tako:

A–>B–>A

Ko so bili vsi paketi poslani, se je test zaključil. Čas poslanih paketov je meril računalnik, ki je začel s pošiljanjem paketov. Vozlišče A je pred vsakim poslanim paketom izvršilo ukaz za pridobitev trenutnega sistemskega časa. Čas se je shranil, in ko je vozlišče A dobilo povratni paket od vozlišča B, je vozlišče A zopet izvršilo ukaz za pridobitev sistemskega časa. Končni čas se je nato odštel od začetnega in tako je bil izračunan čas potovanja paketa od vozlišča A do B in nazaj. Opravljena so bila testiranja za pakete od velikosti 32 bajtov do 64 kilobajtov, kjer je vsak naslednji paket enkrat večji od prejšnjega. Na opisani način so bili izmerjeni vsi paketi vsakega testa. Iz posameznih časov je bil izračunan povprečni čas celotne poti paketa, nadalje pa je bil izmerjen

tudi celoten čas pošiljanja vseh paketov posameznega testiranja. Celoten čas je bil izmerjen tako, da je vozlišče A pred začetkom pošiljanja prvega paketa izvršilo ukaz za pridobitev sistemskega časa ter ta ukaz ponovilo, ko je bil v vozlišče A prejet poslednji paket vozlišča B. Razlika med začetnim in končnim časom je čas pošiljanja vseh paketov.

Primerjava hitrosti med obema načinoma komunikacije se mi je zdela zanimiva predvsem iz eksperimentalnega razloga. Želel sem videti, če obstaja boljši način komunikacije od že obstoječih načinov. Na pohitritev bi najbolj vplival način komunikacije med uporabniškim ter sistemskim prostorom, ki bi zamenjalo prejšnji vmesnik med uporabniški in sistemskim prostorom - vtičnik. Kreacija vtičnikov ne bi bila več potrebna, potrebno pa bi bilo rezervirati pomnilnik ter vanj preslikati datoteko, ki bi jo uporabljal tako sistemski kot uporabniški prostor. Ideja je bila pridobiti na hitrosti pri izmenjavi podatkov med uporabniškim in sistemskim prostorom.

Uporaba vtičnikov

Najprej je bila implementirana komunikacija z uporabo vtičnikov. Implementacija je sledila ukazom in uporabam funkcij, kot smo jih opisali v razdelku 3.2. Testni program, ki je bil napisan, je razmeroma preprost. Program ob zagonu potrebuje dodaten argument, in sicer internetni naslov drugega testirnega računalnika. Ko uporabnik program požene, se program ustavi in čaka, da uporabnik poda ukaz za začetek testiranja. Po uporabnikovem ukazu za začetek, se začne s pošiljanjem paketov ter merjenjem časa. Po poslanem zadnjem paketu program izračuna povprečne čase in celotni čas pošiljanja, vse zapiše v datoteko, nato pa se konča.

Uporaba pomnilniške preslikave

Pri uporabi pomnilniško preslikanih datotek so testi potekali na enak način. Na tem mestu pa je potrebno omeniti, da pošiljanje paketov ni bilo povsem enako kot pri pošiljanju paketov z uporabo vtičnikov. Moja implementacija komunikacije med uporabniškim in sistemskim prostorom ne uporablja fragmentacije, kot smo jo opisali v razdelku 3.1.3, kar pomeni, da lahko pošiljamo le pakete, manjše od 1476 bajtov. Če želimo pošiljati večje pakete, je potrebno podatke uporabniškega prostora najprej razdeliti na manjše pakete (velike do 1476 bajtov) in nato vsakega posebej poslati v sistemski prostor. Vsekakor pa to povzroči dodaten čas pošiljanja podatkov, zato pri paketih, večjih od 1476, bajtov pričakujemo precejšnje povečanje časa pošiljanja.

4.2 Rezultati

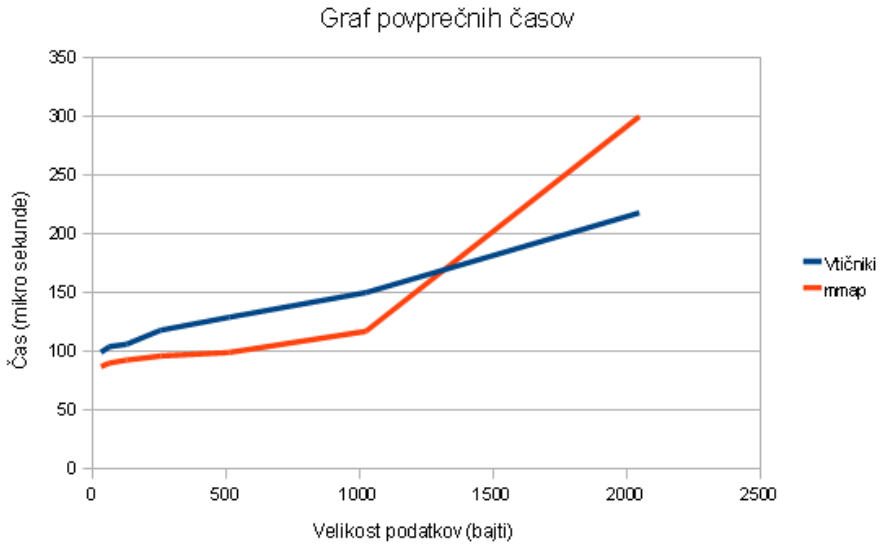
V tem poglavju si bomo pogledali rezultate primerjav pošiljanja paketov z uporabo obeh predstavljenih načinov komunikacije. Rezultati so predstavljeni s pomočjo grafov, ki prikazujejo povprečne čase potovanja paketov v odvisnosti od velikosti paketa, ter z grafi, ki prikazujejo celotne čase pošiljanja paketov ene velikosti v odvisnosti od števila poslanih paketov. Najprej si poglejmo tabelo izmerjenih povprečnih časov ter grafa, ki čase prikazujeta.

Velikost paketa [B]	Vtičniki [μs]	mmap [μs]
32	99.1	87
64	104	90
128	106	92.5
256	118	96
512	129	99
1024	150	117
2048	218	300
4096	237	602
8192	302	1247
16384	492	2405
32768	985	5085
65536	1482	10004

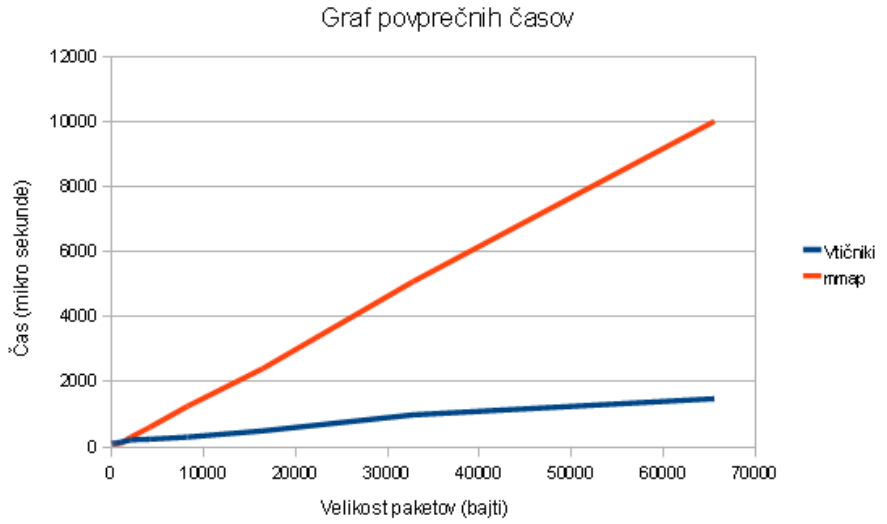
Tabela 4.1: Tabela povprečnih časov pošiljanja paketov

Grafa 4.1 in 4.2 prikazujeta povprečne čase potovanja paketa na poti A-B-A. Kot lahko vidimo v legendi, modra črta označuje naraščanje povprečnega časa pri uporabi vtičnikov, rdeča črta pa pri uporabi pomnilniško preslikanih datotek. Graf 4.1 zaradi večje preglednosti prikazuje čas za pakete velikosti od 32 bajtov do 2048 bajtov. Na njem lahko vidimo, da je način uporabe komunikacije med sistemskim in uporabniškim prostorom boljši, toda le, dokler velikost paketa ne preseže 1024 bajtov. Glede na povedano je to razumljivo, saj je naslednji paket velik 2048 bajtov, kar presega največjo dovoljeno velikost okvirja (1476 bajtov), ki ga lahko gonilnik preda mrežnemu vmesniku. Ker moj način implementacije komunikacije med uporabniškim in sistemskim prostorom ne uporablja fragmentacije paketov, ampak prevelike pakete že v uporabniškem prostoru razdeli na ustrezno število manjših, pride pri temu načinu komunikacije do dodatnih zakasnitev. Do zakasnitev pride zaradi večkratnega posredovanja podatkov iz uporabniškega v sistemski prostor namesto enega

samega prenosa podatkov. Na Grafu 4.2 opazimo, da je uporaba vtičnikov neprimerno boljša, kadar presežemo velikost paketa nad 1024 bajtov.

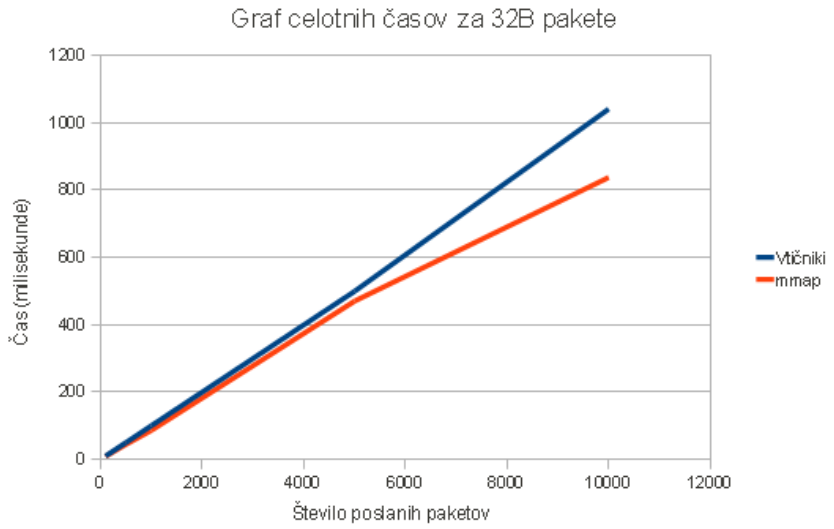


Slika 4.1: Graf časa v odvisnosti od velikosti paketa (od 32B-2048B)

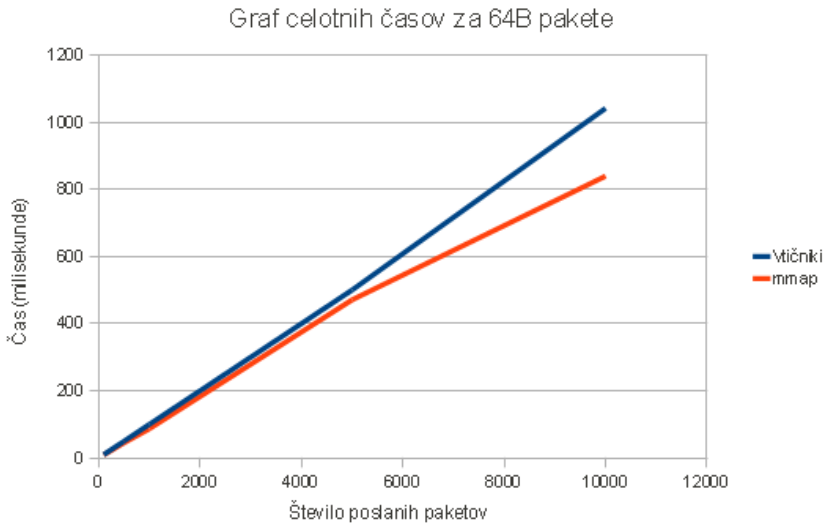


Slika 4.2: Graf časa v odvisnosti od velikosti paketa (od 32B-64 kB)

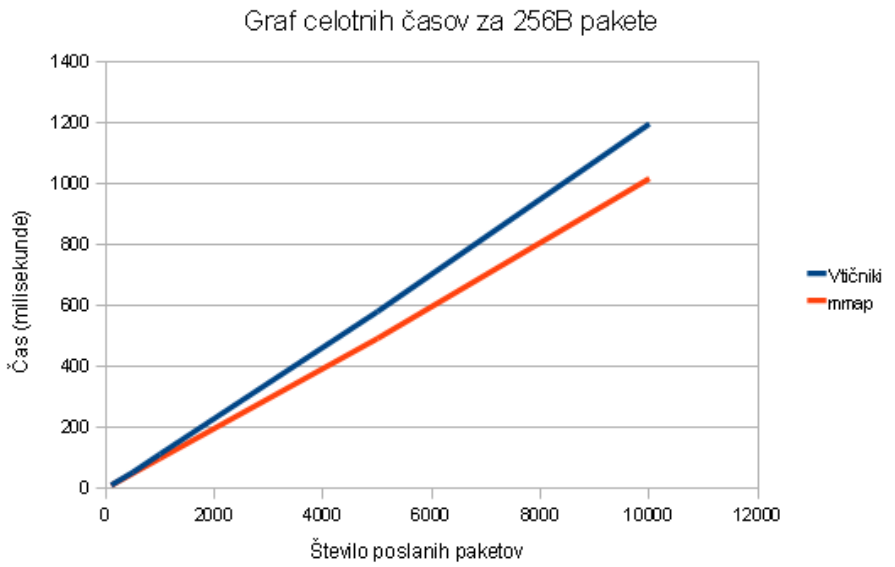
Grafi, ki sledijo, prikazujejo celotni čas pošiljanja vseh paketov v odvisnosti od števila paketov. Serije paketov, za katere so bili izmerjeni časi pošiljanja, so bile velike 100, 500, 1000, 5000 in 10000 paketov.



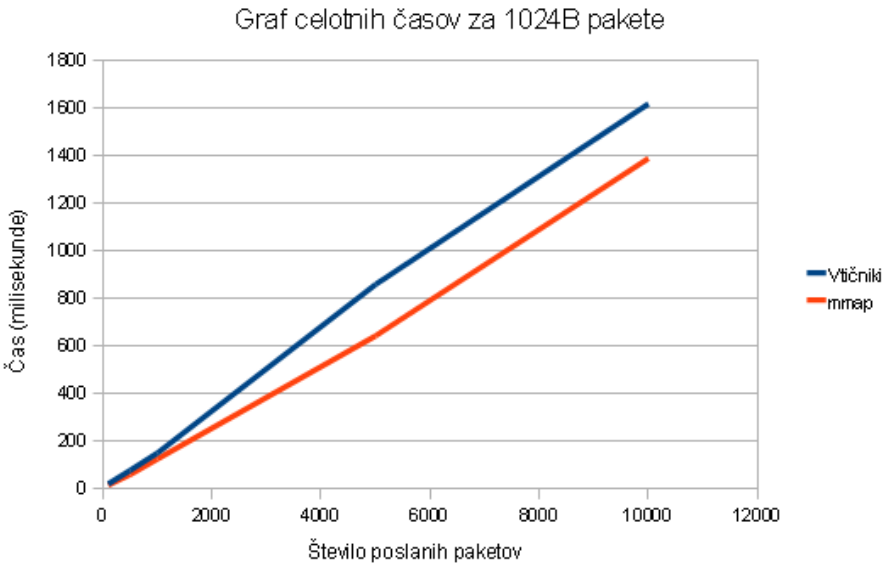
Slika 4.3: Graf časa v odvisnosti od števila paketov za 32B velike pakete



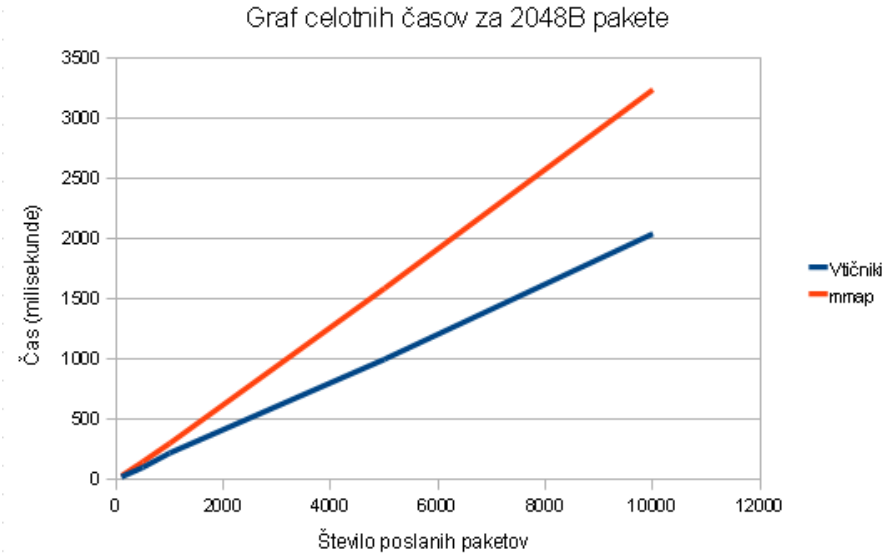
Slika 4.4: Graf časa v odvisnosti od števila paketov za 64B velike pakete



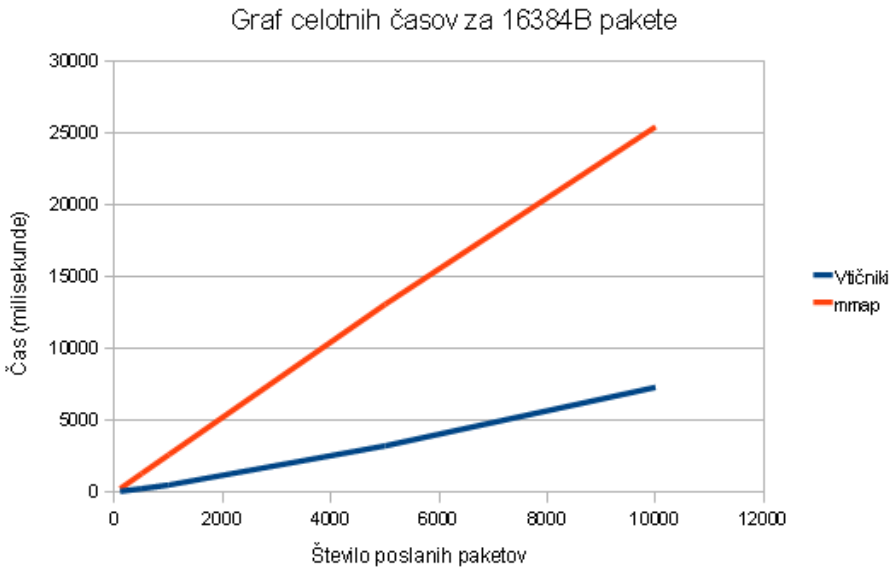
Slika 4.5: Graf časa v odvisnosti od števila paketov za 256B velike pakete



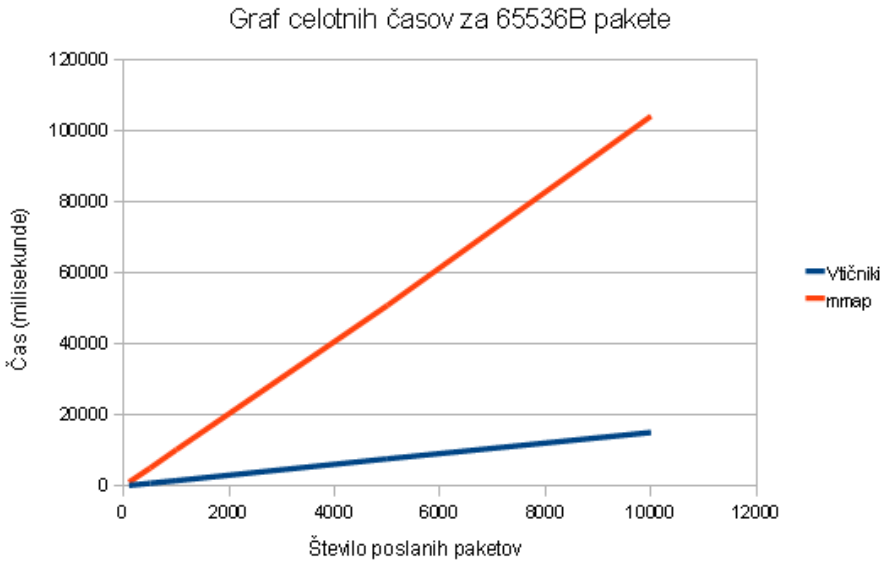
Slika 4.6: Graf časa v odvisnosti od števila paketov za 1024B velike pakete



Slika 4.7: Graf časa v odvisnosti od števila paketov za 2048B velike pakete



Slika 4.8: Graf časa v odvisnosti od števila paketov za 16384B velike pakete



Slika 4.9: Graf časa v odvisnosti od števila paketov za 65536B velike pakete

Kot lahko vidimo, se Grafa 4.3 in 4.4 skoraj ne razlikujeta. To je tudi razlog, da niso prikazani grafi pošiljanja paketov vseh velikosti. Prikazani grafi so bili izbrani na podlagi čim večjih razlik med njimi, da lahko bralec opazi, kako se celotni čas pošiljanja dejansko veča z velikostjo paketov ter številom paketov. Največja razlika je med Grafoma 4.6 in 4.7, kjer opazimo popoln zasuk rezultatov. Na Grafu 4.6 so celotni časi komunikacije med uporabniškim in sistemskim prostorom nižji od celotnih časov vtičnikov. Če pa pogledamo na Graf 4.7, pa so nižji celotni časi komunikacije z uporabo vtičnikov. Da bi si grafa lažje predstavljali, si poglejmo naslednji dve tabeli, ki prikazujeta celotne čase vseh serij pošiljanja za oba načina komunikacije.

Št. paketov	Vtičniki [ms]	mmap [ms]
100	20.2	14.9
500	76.92	59.53
1000	149.86	124.77
5000	858.63	642.59
10000	1617.43	1388.11

Tabela 4.2: Tabela časov pošiljanja serij 1024B velikih paketov

Št. paketov	Vtičniki [ms]	mmap [ms]
100	22.3	30.12
500	100.6	147.82
1000	220.31	300.4
5000	998.65	1587.15
10000	2041.1	3237.68

Tabela 4.3: Tabela časov pošiljanja serij 2048B velikih paketov

Če podrobneje preučimo Tabelo 4.2, vidimo, da je čas pošiljanja serije 10000 paketov pri uporabi pomnilniško preslikanih datotek za 14.17 % manjši kot pri komunikaciji z vtičniki. Če enako primerjavo med komunikacijama naredimo še na Tabeli 4.3, pa vidimo, da je čas pošiljanja 10000 paketov pri uporabi vtičnikov manjši za kar 36.95 %. Do take spremembe je v časih pri uporabi pomnilniško preslikanih datotek prišlo zaradi pomanjkanja fragmentacije. Pri uporabi komunikacije z uporabo vtičnikov uporabljamo standarden nespremenjen gonilnik, ki ob prejemu paketov iz uporabniškega prostora opravi potrebno fragmentacijo, če so ti paketi večji od 1476 bajtov. Uporaba komunikacije med uporabniškim in sistemskim prostorom pa uporablja spremenjen gonilnik. Ta spremenjen gonilnik ne uporablja fragmentacije in lahko zato prejema le pakete do velikosti 1476 bajtov. Ob pošiljanju podatkov, večjih od 1476 bajtov, mora za pravilno prerazporeditev skrbeti uporabniški proces, tu pa se pojavi dodatna zakasnitev, saj je zaradi tega potrebnih veliko več pisanj in branj v/iz pomnilniško preslikane datoteke. Kot vemo, so bralne in pisalne operacije precej potratne s časom. Če pogledamo grafe, pri katerih so paketi večji od 2048 bajtov, lahko opazimo stopnjevano slabšanje komunikacije med uporabniškim in sistemskim prostorom. Čeprav so se časi komunikacije z uporabo pomnilniško preslikanih datotek med testiranji dvignili nad čase komunikacije z uporabo vtičnikov, pa ne moremo zaključiti, da je ta način komunikacije slabši. Kot omenjeno na začetku pričujočega dela, je komunikacija z uporabo pomnilniško preslikanih datotek le eksperimentalni način komunikacije in zato ne more biti izpopolnjen na tak način, kot komunikacija z uporabo vtičnikov. V moji implementaciji komunikacije z uporabo preslikanih datotek gonilnik ni uporabljal fragmentacije podatkov, prejetih iz uporabniškega prostora. To bi bila vsekakor zanimiva izboljšava, ki bi jo lahko naredili na tem načinu komunikacije ter bi na ta način pripomogli k doseganju boljših rezultatov.

Poglavje 5

Zaključek

Tekom dela smo si pogledali načine medprocesnih komunikacij, definicijo internetnega paketa in okvirja, sestavo paketa, Linuxove module in gonilnike ter načine implementacije obeh uporabljenih komunikacij. Primerjali smo čase pošiljanja paketov, tako povprečne čase posameznih paketov kot tudi celotne čase pošiljanja serij paketov. Do določene velikosti paketa se je za boljši način izkazal način, ki uporablja pomnilniško preslikane datoteke. Ko je velikost paketa narasla preko 1476 bajtov, so se povprečni časi paketov in celotni časi pošiljanja serij precej povečali. Pridemo torej do zaključka, da se je uporaba komunikacije z uporabo pomnilniško preslikanih datotek na koncu izkazala za slabšo, vendar pa sem mnenja, da bi se njeni rezultati precej izboljšali, če bi v prilagojenem gonilniku uporabili fragmentacijo paketov.

Po pregledu rezultatov ter pojasnitvi njihovega pomena naj za konec poudarim, da je izboljšanje protokola za pošiljanje podatkov omejeno s pasovno širino povezovalnega medija. Kljub dobremu in hitremu protokolu, ki ga komunikacija uporablja, prenosi ne morejo iti hitreje, kot to dopušča medij. Kot izboljšave protokolov so zato zelo pomembne tudi izboljšave povezovalnih medijev, glede na to, da stremimo k vedno višjim hitrostim prenosov. Za testiranja so bile uporabljene 1-Gbitne linije, trenutno pa so na voljo tudi povezovalni mediji, ki omogočajo višje hitrosti, na primer optične linije.

Literatura

[1] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux Device Drivers*, California: O'Reilly Media, 3rd Edition, 2005.

[2] Brian Hall, *Beej's Guide to Unix IPC*, 2010

[3] Keith Haviland, Dina Gray and Ben Salama, *UNIX system Programming: A programmer's guide to software development*, England: Addison-Wesley, 2nd Edition, 1999, pogl. 6 in pogl. 10.

[4] (2010) A Brief History of the Internet. Dostopno na: http://www.walthowe.com/navnet/history.html

[5] (2010) The Linux Kernel Archives. Dostopno na: http://www.kernel.org

[6] (2010) The Linux Home Page at Linux Online. Dostopno na: http://www.linux.org

[7] (2008) Kernel Space - User Space Interfaces. Dostopno na: http://people.ee.ethz.ch/∼arkeller/linux/kernel_user_space_howto.html

Stvarno kazalo