UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Daniel Doža

RAZVIJANJE VIRTUELNOG FAJL SISTEMA KORIŠĆENJEM MODULA FUSE

master rad

Beograd, 2022.

Mentor:

prof dr Miroslav Marić, redovni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

prof dr Filip Marić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Vladimir Kuzmanović, asistent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane:

Naslov master rada: Razvijanje virtuelnog fajl sistema korišćenjem modula FUSE

Rezime: Fajl sistemi su jedan od ključnih delova operativnog sistema. Bez njih, rukovanje podacima bi bilo izrazito teško. Modul FUSE je omogućio aplikacijama u korisničkom adresnom prostoru da implementiraju virtuelne fajl sisteme i time je stvorio mnogobrojne nove načine za njihovu upotrebu. Cilj ovog rada je da predstavi način na koji ova tehnologija omogućava udaljenim serverima da sinhronizuju podatke i kakvu primenu to može da ima u kompleksnim računarskim sistemima u oblaku.

Ključne reči: fajl sistem, Go, FUSE, računarstvo u oblaku

Sadržaj

1 Uvod **1**

2 Fajl sistem **2**
2.1 Fajlovi . . . 3
2.2 Direktorijumi . . . 4
2.2.1 Jedan nivo dubine . . . 5
2.2.2 Hijerarhijska organizacija . . . 6
2.2.3 Operacije nad direktorijumom . . . 6
2.3 Implementacija fajlova . . . 7
2.3.1 Neprekidna alokacija . . . 7
2.3.2 Povezane liste . . . 9
2.3.3 Indeksirana alokacija . . . 10
2.3.4 Atributi fajla . . . 10

3 Virtuelni fajl sistem i modul FUSE **13**
3.1 Virtuelni fajl sistem . . . 14
3.2 Modul FUSE . . . 15

4 Implementacija virtuelnog fajl sistema **18**
4.1 Primer upotrebe virtuelnih fajl sistema u oblaku . . . 18
4.2 Program SFTPFS . . . 21
4.2.1 Detalji implementacije . . . 21
4.2.1.1 Izbor biblioteke i njen način rada . . . 22
4.2.1.2 Reprezentacija strukture fajlova i direktorijuma . . . 24
4.2.1.3 Implementacija interfejsa fajl sistema . . . 26
4.2.1.4 Komunikacija sa udaljenim serverom . . . 30

5 Zaključak **32**

Bibliografija **33**

Glava 1

Uvod

Fajl sistem je deo operativnog sistema koji pruža organizovano rukovanje fajlovima. Njegov zadatak je da definiše kako su fajlovi struktuirani, kako se identifikuju, koje su operacije moguće nad njima, kako su implementirani itd. Kôd fajl sistema usko je vezan za kernel operativnog sistema što otežava razvoj fajl sistema za specijalizovanu upotrebu.

Virtuelni fajl sistemi nastali su sa ciljem da olakšaju podršku velikog broja različitih fajl sistema u okviru jednog operativnog sistema. Korisničkim procesima predstavljaju sloj apstrakcije iznad konkretnih implementacija fajl sistema. Kako bi konkretna implementacija bila deo virtuelnog fajl sistema, treba da obezbedi ponašanja koja on nalaže.

Modul FUSE predstavlja sponu između virtuelnog fajl sistema i njegove konkretne implementacije u korisničkom adresnom prostoru. Prebacivanjem implementacije u domen korisničkih aplikacija, eliminiše se potreba za menjanjem koda kernela što omogućava jednostavan način za implementaciju specijalizovanih fajl sistema. Za potrebe ovog rada napisan je program koji služi da prikaže način na koji se implementira virtuelni fajl sistem.

Cilj ovog rada je da ukaže na upotrebnu vrednost modula FUSE i da prikaže kako se fajl sistemi mogu koristiti u kontekstu računarstva u oblaku. Rad počinje poglavljem 1 koje sadrži kratak uvod. Poglavlje 2 opisuje rad tradicionalnih fajl sistema. Zatim se u poglavlju 3 opisuje rad virtuelnih fajl sistema i modula FUSE. U poglavlju 4, diskutuje se o primeru jedne konkretne implementacije virtuelnog fajl sistema u programskom jeziku Go. U poslednjem poglavlju 5 se nalazi zaključak.

Glava 2

Fajl sistem

Fajl sistem čini skup metoda i struktura podataka koje se koriste od strane operativnog sistema kako bi se vodila evidencija o podacima na disku. Posledica postojanja fajl sistema je skup funkcija koje korisnik može da poziva i koristi u svom programu. Na apstraktnom nivou, fajl sistem se može posmatrati kao interfejs pomoću kojeg operativni sistem omogućava svojim korisnicima rukovanje fajlovima.

Kako bi se bolje razumela potreba za fajl sistemom, problem skladištenja podataka se može posmatrati kroz potrebe procesa. Svi procesi imaju potrebu da skadište i preuzimaju podatke. Proces može da čuva podatke u glavnoj memoriji ali je ograničen veličinom virtuelnog adresnog prostora. Drugo ograničenje čuvanja podataka u glavnoj memoriji je što se podaci gube nakon prekida rada programa. Naime, nekada je potrebno da procesi pri ponovnom pokretanju nastave odakle su stali kao što je slučaj na primer sa bazama podataka. Takođe, često se javlja potreba za istovremenim pristupom istom skupu podataka od strane više različitih procesa. Ovo je takođe neizvodljivo u slučaju čuvanja podataka u glavnoj memoriji.

Opisani problemi dovode do tri ključna zahteva koja se očekuju od sistema za trajno skladištenje podataka.

1. Čuvanje velikih količina podataka.

2. Otpornost podataka na prekide rada procesa.

3. Istovremeni pristup podacima od strane više procesa.

U skladu sa navedenim zahtevima, jasno je da glavna memorija nije dobar izbor za trajno skladištenje podataka pa zato fajl sistemi koriste druge fizičke medijume. U prošlosti su bili popularni čvrsti diskovi, dok su danas sve više zastupljeni SSD

(engl. solid state drive). Za sve vrste fizičkih medijuma koje koristi fajl sistem, u daljem tekstu će se koristiti termin *disk*.

U nastavku poglavlja biće razmotreni pojmovi koji su potrebni za razumevanje rada fajl sistema.

2.1 Fajlovi

Fajl sistem predstavlja podatke zapisane na disku u vidu apstrakcije koja se naziva *fajl*. Ovakva apstrakcija je namenjena da korisniku pruži jedinstven interfejs za rad sa podacima, pritom sakrivajući od njega informacije o disku na kome se ti podaci nalaze i način na koji se skladište.

Osnovna stvar koja karakteriše fajl sistem jeste skup operacija za rad sa fajlovima koje podržava. Prvo će biti navedene osnovne operacije nad fajlovima podržane od strane svih fajl sistema.

1. Kreiranje - Kreira se nov, prazan fajl i postavljaju mu se neki od atributa.

2. Brisanje - Kada fajl nije više potreban, briše se kako bi se oslobodila memorija na disku.

3. Otvaranje - Pre nego što krene da ga koristi, proces mora da *otvori* fajl. Ova operacija služi kako bi se relevantne informacije o fajlu učitale u glavnu memoriju i bile brzo dostupne za naredne pozive.

4. Zatvaranje - Kada se rad sa fajlom završi, učitane informacije o njemu se brišu iz glavne memorije.

5. Čitanje - Čitanje bajtova iz fajla. Obično se čita sa određene pozicije u fajlu i čita se predefinisan broj bajtova. Korisnik je takođe dužan da obezbedi bafer za skladištenje pročitanih bajtova.

6. Pisanje - Pisanje bajtova u fajl. Kao i prilikom čitanja, uglavnom se piše na trenutnu poziciju u fajlu. Ako je trenutna pozicija na kraju fajla, veličina fajla se povećava, a u koliko je pozicija na sredini, bajtovi koji slede se prepisuju i bivaju izgubljeni.

Pored nabrojanih operacija, vredno je napomenuti još neke od operacija koje se mogu pronaći na većini fajl sistema.

1. Nadovezivanje - Specijalan slučaj pisanja gde se bajtovi dodaju isključivo na kraj fajla.

2. Repozicioniranje - Pomeranje pokazivača na određeni deo fajla. Nakon ove operacije, pozivi za čitanje i pisanje biće izvršeni sa ove pozicije.

3. Dohvatanje atributa - Nekim programima dovoljno je da učitaju samo atribute fajla kako bi izvršili neku operaciju. Primer ovoga je program `ls` koji u detaljnom režimu rada ne čita sadržaj fajla, već samo ispisuje atribute koji se odnose, između ostalog, na prava pristupa, vreme kreiranja, itd.

4. Postavljanje atributa - Može se koristiti, na primer, za promenu prava pristupa ili vlasništva nad fajlom.

5. Preimenovanje - Promena imena fajla.

Još jedna bitna odluka koja razlikuje fajl sisteme je način na koji se fajlovi struktuiraju. U nastavku će biti opisana tri različita načina za organizaciju sadržaja fajla.

Na modernim sistemima dominantno je struktuiranje fajla kao niza bajtova. [2]. Iz ugla fajl sistema, ovakav niz bajtova nema nikakvo specijalno značenje. Njegovo tumačenje je uloga korisničkog programa. Iz ovog razloga, ovaj način struktuiranja pruža najveću fleksibilnost.

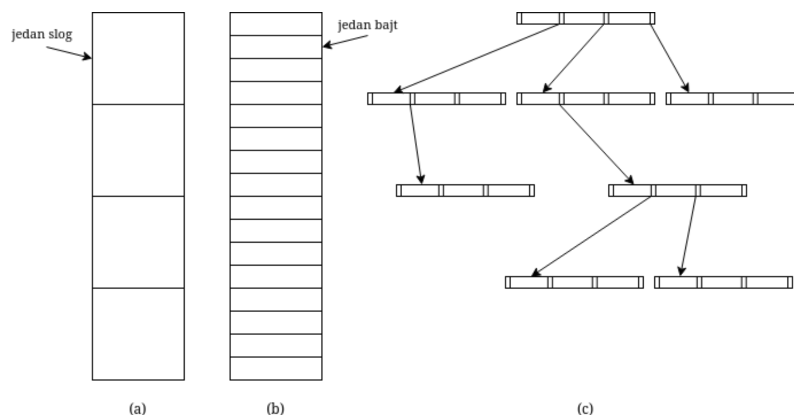
U narednim primerima, gradivna jedinica fajla nije bajt, već grupa bajtova. Ovako grupisani bajtovi nazivaju se slog.

Drugi način struktuiranja fajlova je korišćenjem niza slogova fiksne dužine. Pri ovakvoj strukturi se upis i čitanje vrši slog po slog. Ideja slogova datira iz doba bušenih kartica sa 80 kolona, pa su mejnfrejm (engl. mainframe) računari tog vremena koristili slogove dužine 80 karaktera. [3]. Ova tehnologija je zastarela pa se zato više ne koristi.

Treća ideja je konstruisanje stabla. Fajlove čine slogovi koji predstavljaju čvorove u ovakvom stablu. Čvorovima su dodeljeni ključevi i stablo je sortirano po njima. Osnova ideja je da se ubrza traženje sloga po nekom ključu.

2.2 Direktorijumi

Fajl sistemi organizuju fajlove smeštajući ih u direktorijume. Direktorijumi se mogu konceptualno predstaviti kao tabele koje mapiraju imena fajlova na strukture

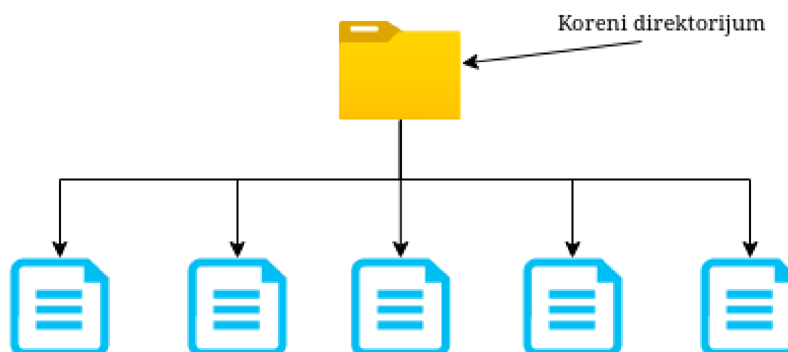


Slika 2.1: Načini struktuiranja fajlova. (a) Niz slogova. (b) Niz bajtova (c) Stablo

koje ih predstavljaju. Na operativnom sistemu Windows koristi se i naziv fascikla (folder). U nastavku će ukratko biti opisane dve vrste organizovanja direktorijuma i operacije koje se vrše nad njima.

2.2.1 Jedan nivo dubine

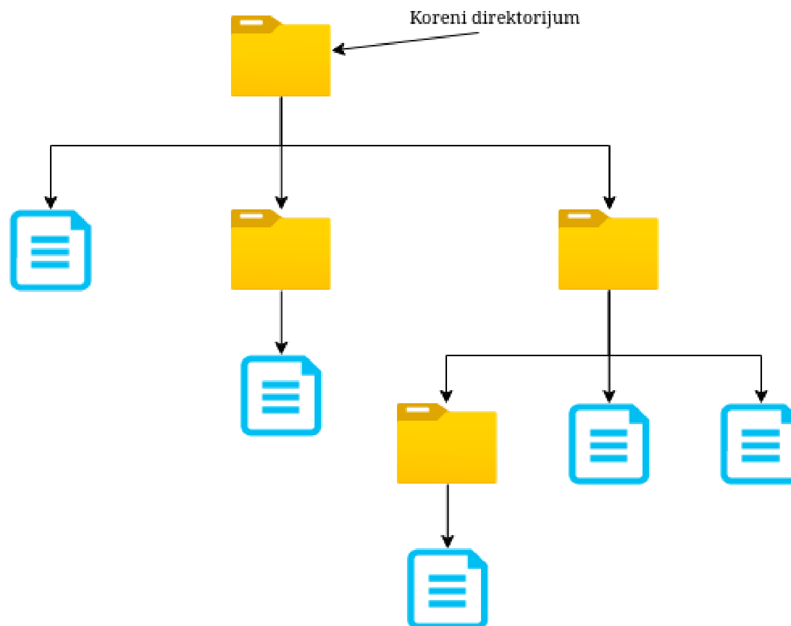
Jedan od načina za organizovanje fajlova je da se svi fajlovi čuvaju u jednom korenom direktorijumu. Ovakva struktura omogućava jednostavnu implementaciju, međutim ima i svoje mane. Naime, svi fajlovi moraju da imaju jedinstvena imena kako bi se razlikovali jedan od drugog. Ovo je održivo dok je broj fajlova mali, međutim kako njihov broj raste, ovo postaje sve teže. Takođe, sa većim brojem fajlova povećava se vreme koje je potrebno za pretragu direktorijuma. Ovaj pristup organizaciji fajlova bio je prihvatljiv u ranom dobu računarstva, delom zato što su računari imali jednog korisnika.



Slika 2.2: Organizacija fajlova u jedan nivo.

2.2.2 Hijerarhijska organizacija

U hijerarhijskom uređenju, direktorijumi pored fajlova mogu da sadrže druge direktorijume. Ideja iza ovakve organizacije je da se direktorijumi organizuju u strukturu stabla. U korenu stabla se nalazi koreni (engl. root) direktorijum. Fajlovima se pristupa preko jedinstvene putanje koja predstavlja put od korenog direktorijuma niz stablo do tog fajla. Korisnici mogu kreirati poddirektorijume po potrebi i njihov broj nije ograničen[1]. Takođe, ovaj pristup eliminiše problem nazivanja fajlova, jer fajlovi moraju biti jedinstveno imenovani samo u direktorijumu koji im je direktan roditelj u stablu. Hijerarhijska struktura daje mnogo veću fleksibilnost u organizovanju fajlova i koristi se na većini modernih fajl sistema.



Slika 2.3: Hijerarhijska organizacija fajlova i direktorijuma.

2.2.3 Operacije nad direktorijumom

Kao što je slučaj sa fajlovima, skup podržanih operacija nad direktorijumom zavisi od sistema do sistema, tako da će biti navedene samo neke od osnovnih operacija.

1. Kreiranje - Kreira se prazan direktorijum.

[1]Dok god ima prostora na disku.

2. Brisanje - U zavisnosti od operativnog sistema, brisanje može biti drugačije implementirano. Operativni sistem Linux, na primer, ne dozvoljava brisanje direktorijuma ukoliko nije prazan.

3. Otvaranje - Slično kao otvaranje fajla. Pre nego što je moguće pročitati sadržaj direktorijuma, on mora biti otvoren.

4. Zatvaranje - Nakon što se izlista, direktorijum bi trebalo zatvoriti kako bi se oslobodila memorija.

5. Listanje - Izlistava se sadržaj direktorijuma.

6. Preimenovanje - Promena imena direktorijuma.

2.3 Implementacija fajlova

U ovoj sekciji biće reči o odlukama koje je potrebno doneti prilikom implementacije fajlova. Jedna takva odluka odnosi se na to koje će informacije o sebi fajl pružati putem njegovih atributa. Drugi problem koji se rešava prilikom implementacije fajlova je održavanje struktura podataka koje pružaju informaciju o tome gde se sadržaj fajla nalazi u memoriji. Memorija u kojoj se čuvaju fajlovi definiše *blok* kao najmanju jedinicu za alokaciju. Navedena su glavna pitanja na koja implementator fajl sistema ima zadatak da odgovori.

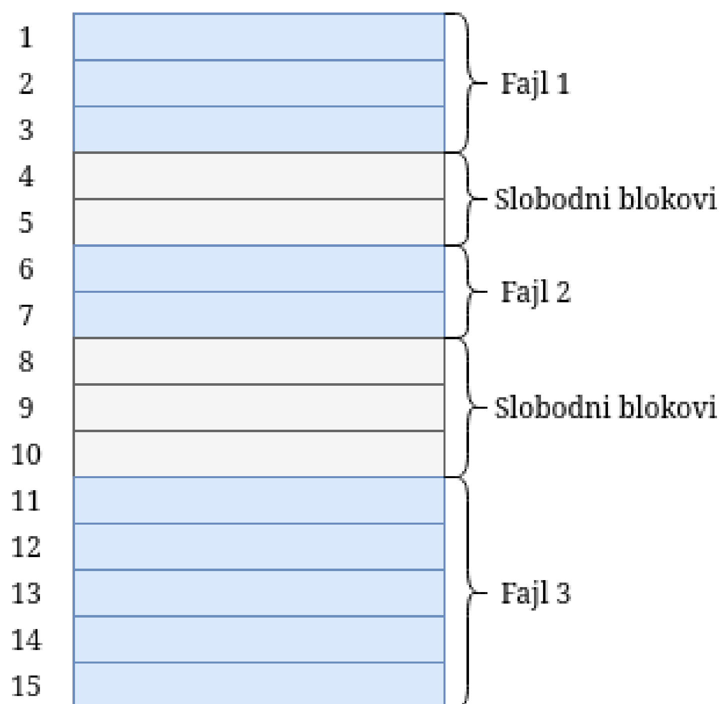
1. Koji blok memorije pripada kom fajlu?

2. Koji način alokacije blokova se korsti?

2.3.1 Neprekidna alokacija

Najjednostavniji pristup za alokaciju blokova fajla je *neprekidna alokacija*. Ideja je da blokovi koji predstavljaju jedan fajl čine neprekidan niz u memoriji. Na primer, ako je blok dužine 500 bajtova, a fajl koji treba da se alocira je veličine 1200 bajtova, taj fajl bi bio predstavljen sa 3 uzastopna bloka (pošto veličina fajla nije deljiva veličinom bloka, poslednjih 300 bajtova iz trećeg bloka ostaje neiskorišćeno). Ovakav pristup ilustrovan je na slici 2.4.

Alokacija blokova na ovakav način ima svoje prednosti. Prva prednost odnosi se na jednostavnost ovakve implementacije. Struktura koja predstavlja fajl treba samo



Slika 2.4: Neprekdina alokacija.

da zapamti dva broja: adresu prvog bloka u fajlu i broj blokova koji čine taj fajl. Druga prednost je što je čitanje fajla veoma brzo zbog sekvencionalnosti blokova.

Međutim, ovakav pristup krije ozbiljnu manu. Vremenom, brisanje fajlova u fajl sistemu dovodi do fragmentacije. Fragmentacija se manifestuje kao „rupe" neiskorišćenih blokova između fajlova. Na slici 2.4 ovakve „rupe" su označene sivom bojom. Ovaj problem kreće da se manifestuje tek kada za nove fajlove nema više mesta na kraju diska. Lako se dolazi do situacije gde na disku više nema mesta za upis novih fajlova, a da je stvarna situacija da ima mnogo neiskorišćenih blokova.

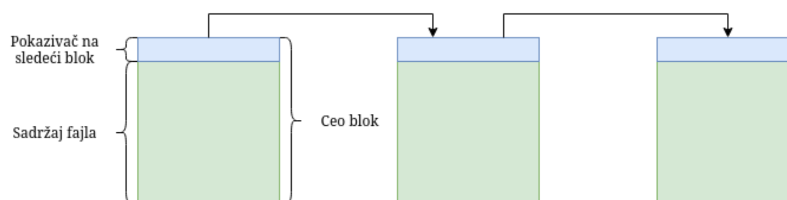
Jedno rešenje bi bilo da se uradi defragmentacija koja bi preuredila fajlove na disku tako da čine neprekidan niz od početka diska. Ovom operacijom bi blokovi koji su bili neiskorišćeni mogli biti opet u upotrebi, ali je problem što je ona veoma vremenski zahtevna. Može se voditi evidencija o praznom prostoru između fajlova kako bi se prostor iskoristio za nove fajlove, međutim ovo je neizvodljivo jer bi bilo potrebno da se veličina fajla zna unapred kako bi se utvrdilo da može da se upiše u taj prostor.

Iako se, zbog navedenih mana, neprekidna alokacija ne koristi u modernim fajl sistemima, značajno je napomenuti da za nju ipak postoji primena i danas. Naime,

optički medijumi kao što su CD-ROM i DVD mogu da iskoriste prednosti neprekidne alokacije jer se na njih fajlovi upisuju samo jednom i njihova veličina je poznata pre upisa.

2.3.2 Povezane liste

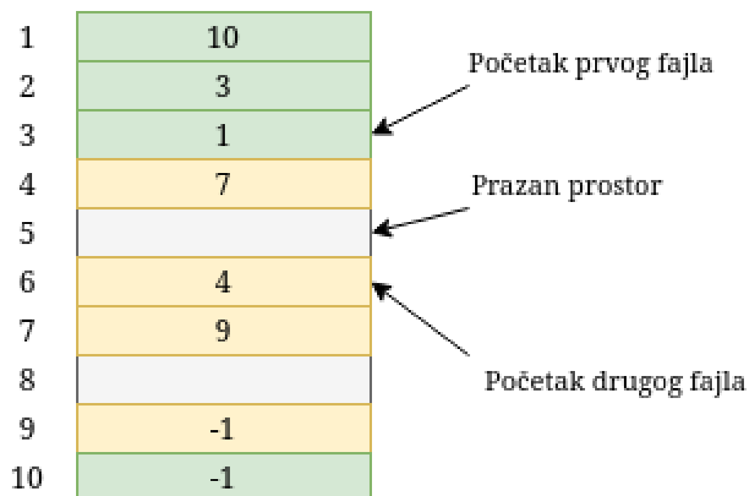
Rešavanje problema fragmentacije se može postići tako što se fajlovi predstavljaju kao povezana lista blokova na disku. Osnovna ideja je da svaki blok pokazuje na sledeći blok, gde bi pri tome prvih par bajtova svakog bloka služili za čuvanje pokazivača na sledeći blok. Ovim pristupom fajl struktura treba da zapamti samo adresu prvog bloka fajla, a ostali se dohvataju prateći pokazivače.



Slika 2.5: Primer povezane liste.

Osnovni problem kod ovog pristupa je slučajan pristup blokovima. Da bi se pročitao neki konkretan blok iz liste, potrebno je pročitati svaki blok pre njega što predstavlja ogroman broj čitanja ukoliko je veličina fajla dovoljno velika i zahtevani blok je daleko od početka fajla. Drugi problem je što korišćenje prvih par bajtova narušava činjenicu da su blokovi veličine koja je stepen dvojke. Ovaj problem nije nepremostiv, ali dovodi do neefikasnosti, uzimajući u obzir da dosta programa čita i piše u blokovima čija je veličina stepen dvojke [3].

Ovi nedostaci se mogu eliminsati održavanjem tabele koja se naziva *FAT* (File Allocation Table). Ova tabela se može zamisliti tako da svaki red odgovara fizičkom bloku na disku, a unos u tabeli je broj sledećeg bloka fajla ili -1 što bi označavalo kraj fajla. Dohvatanje slučajnog bloka i dalje zahteva prolazak kroz sve prethodne blokove, međutim sada ovakva tabela može da se učita u glavnu memoriju, pa je samim tim pretraga za konkretnim blokom efikasna. Ovakav pristup je ilustrovan slikom 2.6.



Slika 2.6: Primer tabele FAT.

2.3.3 Indeksirana alokacija

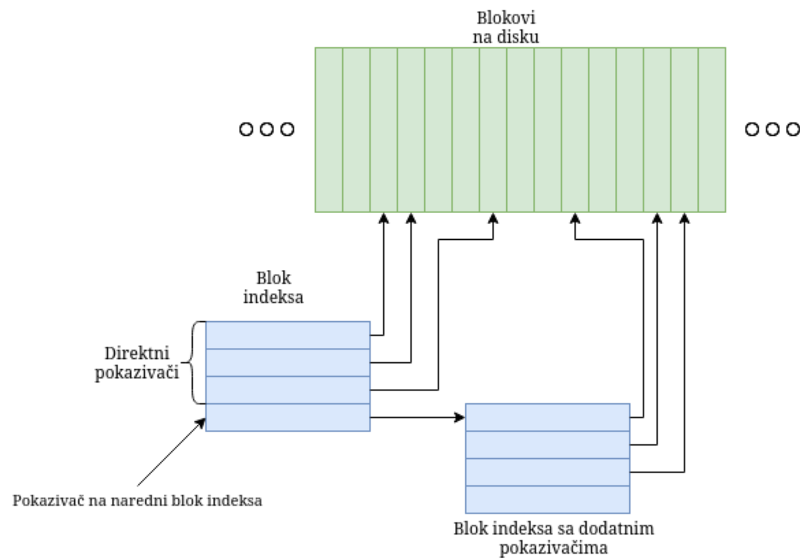
Indeksirana alokacija optimizuje direktan pristup blokovima. Za svaki fajl, održava se struktura podataka koja se sastoji od njegovih atributa i blok indeksa. Blok indeks se sastoji od pokazivača na fizičke blokove i njegovim čitanjem ostvaruje se direktan pristup traženom bloku.

Postavlja se pitanje kako odrediti broj pokazivača u bloku indeksa. Ukoliko je potrebno čuvati velike fajlove, jasno je da treba da ih ima mnogo. Ako je ovo slučaj, prilikom čuvanja malih fajlova, potrebno je svega nekoliko pokazivača, a ostatak memorije se gubi. Sa druge strane kada bi broj pokazivača bio mali, onda fajl sistem ne bi mogao da čuva velike fajlove.

U praksi se ovo najčešće rešava hibridnim pristupom. Blok indeksi pridruženi fajlovima sadrže neki manji broj direktnih pokazivača na fizičke blokove fajla, a nakon toga pokazivači pokazuju na druge blok indekse koji pokazuju na disk ili naredni blok indeks. Ovakvo indeksiranje na više nivoa radi dobro kada su mali fajlovi u pitanju (koriste se samo direktni pokazivači), a veliki fajlovi koriste više nivoa indeksiranja i veličina im nije ograničena. Slika 2.7 pokazuje ovaj način.

2.3.4 Atributi fajla

Atributi fajla su njegovi metapodaci koji pružaju dodatne informacije pored njegovog sadržaja. Oni se mogu koristiti u različite svrhe. Neki atributi služe da bliže opišu sadržaj fajla (na primer veličina fajla) dok se drugi atributi mogu koristiti



Slika 2.7: Indeksiranje na više nivoa. Hibridni pristup.

za formiranje odluka koje donosi fajl sistem (na primer određivanje prava pristupa). Tačan skup atributa se razlikuje od sistema do sistema. Sledi podskup čestih atributa.

- Ime fajla;
- Prava pristupa;
- Identifikator vlasnika;
- Vreme nastanka;
- Vreme poslednje modifikacije;
- Vreme poslednjeg pristupa;
- Veličina u bajtovima;
- Lokacija na disku.

Atributi mogu biti različitih formata. Za veličinu se mogu koristiti brojevi. Identifikator vlasnika fajla može biti tekst. Postoje takođe i flegovi (zastavice, indikatori) koji su u glavnom predstavljeni jednim bitom, gde nula označava da fleg nije postavljen, a jedinica da jeste. Jedan primer primene flegova bi bio sakrivanje određenih fajlova od korisnika.

Atribut koji je važno posebno pomenuti jeste ime fajla. Može se reći da je ime fajla atribut koji svaki sistem mora da sadrži. Spajanjem imena fajla i putanje njegovog roditelja u hijerarhiji, dobija se putanja koja služi kao njegov jedinstveni identifikator u fajl sistemu. Kada korisnik želi da uradi bilo kakvu operaciju nad fajlom, on koristi upravo ovu putanju kako bi ga locirao.

Neki sistemi koriste ime fajla da prenesu informaciju o njegovom tipu. Na primer, operativni sistem Windows posmatra imena fajlova iz dva dela, odvojena tačkom. Deo imena pre tačke je pravo ime fajla, dok se deo nakon tačke naziva *ekstenzija* (nastavak). Ekstenzija fajla govori koji je tip fajla u pitanju.

Sa druge strane, operativni sistem Linux ne razlikuje tipove fajlova na osnovu njihovog imena. Fajlovi se razlikuju na osnovu magičnog broja koji se nalazi u zaglavlju fajla. Magični brojevi neće biti detaljnije razmatrani u sklopu ovog rada. Korišćenje ekstenzija u imenu fajla na ovakvim sistemima je samo konvencija i služi kako bi korisnik na osnovu imena znao koji je tip fajla u pitanju, dok operativni sistem ne koristi ovu informaciju.

Glava 3

Virtuelni fajl sistem i modul FUSE

U poglavlju 2 se vidi da konkretna implementacija fajl sistema treba da podrži razne operacije i da donese odluku o tome na koji način će implementirati svaku od tih operacija. Postoje razne implementacije fajl sistema i neke od njih se mogu videti u tabeli 3.1.

Moderni operativni sistemi moraju da podrže više različitih fajl sistema istovremeno kako bi pružili udobnost korisnicima. Očigledan slučaj je čitanje zapisa sa CD-ROM diska ili čitanje USB fleš diska koji je drugačije formatiran. Metode koje se koriste za istovremenu podršku različitih fajl sistema, variraju u zavisnosti od operativnog sistema. Na primer, operativni sistem Windows, svakoj particiji diska dodeljuje slovo i na svakoj particiji se zasebno nalazi fajl sistem. Ovako je korisniku eksplicitno prikazano da su fajl sistemi razdvojeni. Za razliku od ovog pristupa, operativni sistemi nalik na UNIX koriste drugačiji pristup. Kod njih se fajl sistem može postaviti (engl. mount) na slučajnu putanju i korisniku svi postavljeni fajl sistemi

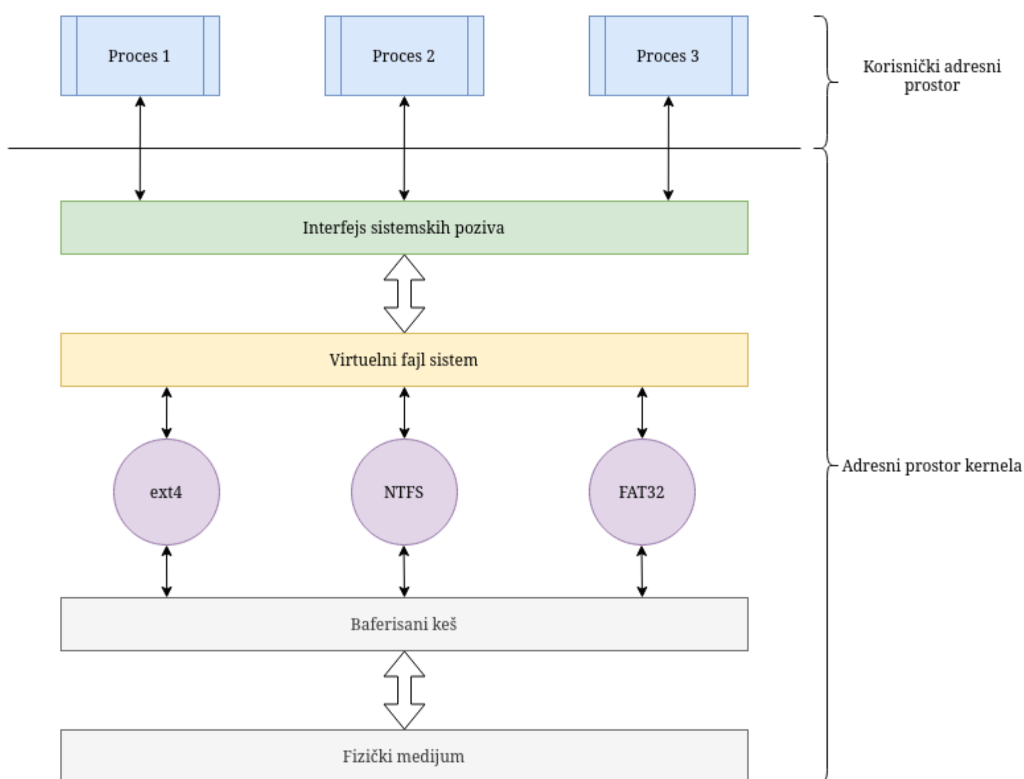
Tabela 3.1: Različiti fajl sistemi i operativni sistemi iz kojih su potekli.

Naziv fajl sistema	**Godina nastanka**	**Operativni sistem**
FAT16	1984	PC DOS 3.0, MS-DOS 3.0
ext	1992	Linux
NTFS	1993	Windows NT 3.1
ext2	1993	Linux, Hurd
HFS Plus	1998	Mac OS 8.1
ext3	1999	Linux
ZFS	2004	Solaris
ext4	2006	Linux
exFAT	2006	Windows CE 6.0

deluju kao jedna homogena struktura.

3.1 Virtuelni fajl sistem

Virtuelni fajl sistemi (VFS, Virtual File System, Virtual Filesystem Switch) nastali su iz težnje ka olakšanoj podršci velikom broju fajl sistema u okviru jednog operativnog sistema. Oni se danas koriste na svim operativnim sistemima nalik na UNIX. Cilj virtuelnih fajl sistema je da apstrahuju operacije koje izvršavaju konkretne implementacije različitih fajl sistema i da ih korisničkim procesima predstave u vidu jedinstvenog interfejsa. Na slici 3.1 se može videti kako procesi putem sistemskih poziva komuniciraju sa virtuelnim fajl sistemom koji zahteve prosleđuje konkretnim implementacijama fajl sistema. Kako su operacije na disku vremenski zahtevne, operacije koje vrše fajl sistemi se prvenstveno upisuju u keš pa se tek nakon toga fizički zapisuju na disk.



Slika 3.1: Arhitektura virtuelnih fajl sistema.

Sa slike 3.1 se može zaključiti da se virtuelni fajl sistem sastoji zapravo iz dva interfejsa. Prvi je interfejs koji je predstavljen korisničkim procesima putem poznatih

metoda `read`, `write` itd. Drugi interfejs je nižeg nivoa i okrenut je ka implementacijama fajl sistema. On propisuje skup operacija i način na koje one moraju da budu implementirane od strane fajl sistema kako bi mogao da se koristi kroz virtuelni fajl sistem. Ove operacije su niskog nivoa i rade na nivou čitanja i pisanja blokova na disku.

Do sada je diskutovano samo o fajl sistemima koji rade sa particijama na lokalnom disku, ali virtuelni fajl sistemi nisu ograničeni samo na ovu upotrebu. Zapravo, originalna zamisao je bila da se pruži podrška za NFS (Network File System) protokol. NFS je mrežni protokol koji omogućava distribuirano deljenje fajlova.

3.2 Modul FUSE

Modul FUSE postoji na operativnim sistemima nalik na UNIX i služi za razvoj virtuelnih fajl sistema u korisničkom adresnom prostoru. Cilj je da se omogući korisnicima da implementiraju specijalizovane fajl sisteme bez potrebe da modifikuju kôd samog kernela. Ova činjenica poboljšava udobnost razvoja s obzirom na to da je razvoj i modifikacija kernelskog koda netrivijalan zadatak.

Način na koji se ostvaruje mogućnost da se fajl sistemi implementiraju u korisničkom adresnom prostoru je upravo preko korišćenja tehnologije virtuelnih fajl sistema. Ideja je da postoji implementacija interfejsa koji nalaže virtuelni fajl sistem, ali da umesto što piše na disk, zahteve prosleđuje korisničkom programu koji na njih odgovara i predstavlja konkretnu implementaciju fajl sistema.

U nastavku se navode tri glavne celine koje čine modul FUSE i koje obezbeđuju opisano ponašanje.

1. Kernelski modul `fuse.ko`.

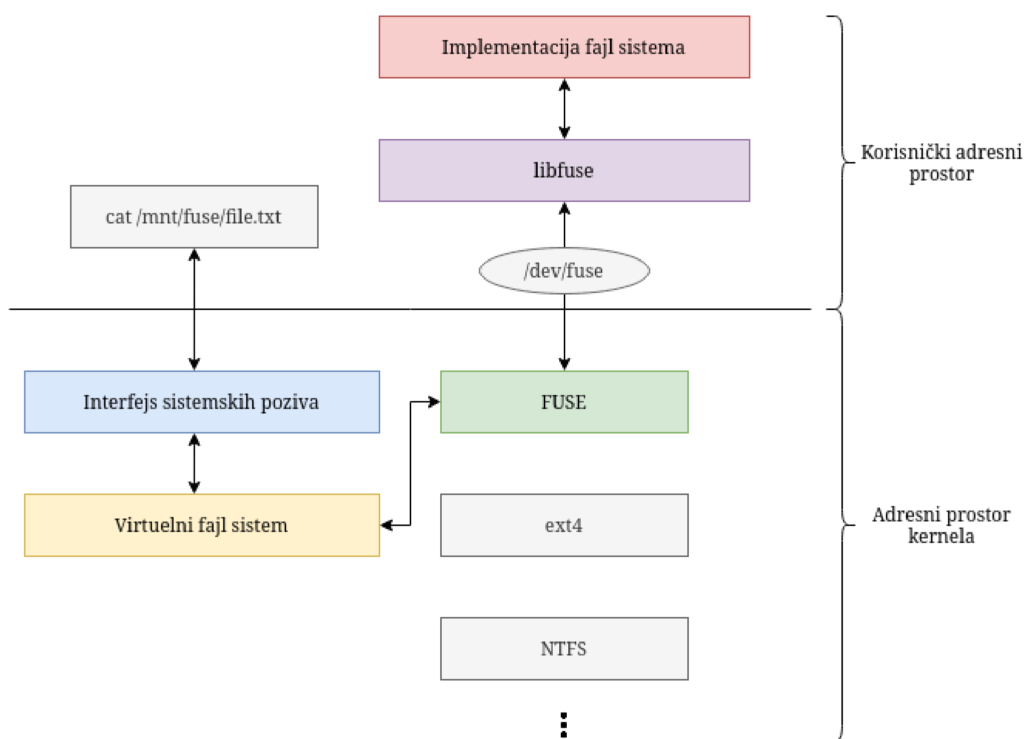
2. Korisnička biblioteka `libfuse`.

3. Pomoćni alat za postavljanje fajl sistema `fusermount`.

Kernelski modul `fuse.ko` implementira interfejs virtuelnog fajl sistema. Njegova uloga je centralna jer pruža način za komunikaciju između kernela i korisničkog procesa. Ovaj modul definiše specijalan protokol za komunikaciju koji se ostvaruje čitanjem i pisanjem specijalnog fajla koji je vidljiv korisničkim procesima.

Zarad boljeg razumevanja, ovaj mehanizam biće objašnjen na konkretnom primeru operacije čitanja fajla. Kao primer biće uzet program `cat` koji ima zadatak

da pročita fajl i da ga ispiše na standardni izlaz. Program `cat` vrši operaciju čitanja i to radi tako što poziva `read` sistemski poziv. Ovakav zahtev stiže do virtuelnog fajl sistema koji odlučuje koju konkretnu implementaciju fajl sistema treba da pozove na osnovu putanje na kojoj se fajl nalazi. Virtuelni fajl sistem poziva modul `fuse.ko`. Nakon što je primio zahtev, modul na osnovu protokola formira odgovarajući zahtev i upisuje ga u fajl koji se koristi za komunikaciju. Korisnički proces koji predstavlja implementaciju fajl sistema, čita komunikacioni fajl, tumači zahtev koji mu je poslat, kreira odgovor i upisuje odgovor u isti fajl. Kernelski modul čita ovaj odgovor, prosleđuje ga virtuelnom fajl sistemu koji na kraju odgovara na inicijalni sistemski poziv.



Slika 3.2: Dijagram toka zahteva kroz modul FUSE.

Biblioteka `libfuse` je korisnička biblioteka napisana u programskom jeziku C koja pruža razvojni okvir za korisničke aplikacije koje implementiraju operacije fajl sistema. Svojim korisnicima pruža interfejs višeg nivoa tako što apstrahuje komunikacioni protokol sa modulom `fuse.ko`. Ovo je jednostavan klijent—server protokol gde modul `fuse.ko` ima ulogu klijenta, a korisnička aplikacija ima ulogu servera [1]. U okviru protokola definisane su dve osnovne poruke: zahtev i odgovor.

Svaki zahtev koji korisnička aplikacija pročita počinje zaglavljem `fuse_in_header`

datim u isečku 3.1. Polje `len` daje informaciju o dužini čitave poruke, uključujući i zaglavlje. Polje `opcode` predstavlja tip operacije koju treba da izvrši fajl sistem. Polje `unique` je jedinstveni identifikator tog konkretnog zahteva. Polje `nodeid` označava identifikator fajla nad kojim treba izvršiti operaciju. Polja `uid`, `gid` i `pid` su identifikatori vezani za proces u čije ime treba da se obavi operacija. Telo zahteva dolazi nakon zaglavlja. Dužina i struktura tela zahteva zavise od operacije na koju se poruka odnosi, odnosno zavise od toga koji `opcode` je prosleđen.

Listing 3.1: Izgled zaglavlja zahteva.

```
struct fuse_in_header {
   uint32_t len;
   uint32_t opcode;
   uint64_t unique;
   uint64_t nodeid;
   uint32_t uid;
   uint32_t gid;
   uint32_t pid;
   uint32_t padding;
};
```

Nakon što izvrši zahtevanu operaciju, korisnička aplikacija treba da odgovori porukom koja će počinjati zaglavljem datim u isečku 3.2. Polje `len` označava dužinu poruke. Ukoliko je došlo do neke greške, aplikacija može iskoristiti polje `error` tako što će ga postaviti na neku dogovorenu vrednost. Polje `unique` treba da bude postavljeno na odgovarajuću jedinstvenu vrednost zahteva na koji se ovaj odgovor odnosi. Telo odgovora je varijabilne dužine i zavisi od toga na koji se zahtev odgovara. Ukoliko je polje `error` postavljeno onda se ne šalje telo poruke.

Listing 3.2: Izgled zaglavlja odgovora.

```
struct fuse_out_header {
    uint32_t len;
    int32_t  error;
    uint64_t unique;
};
```

Treba napomenuti da biblioteka `libfuse` nije neophodna za implementaciju fajl sistema. Opisani komunikacioni protokol može biti implementiran i u drugim programskim jezicima.

Glava 4

Implementacija virtuelnog fajl sistema

Moderne veb aplikacije su često veoma kompleksne i zahtevaju komplikovanu infrastrukturu za njihovo pokretanje. Intenzitet saobraćaja i količina izračunavanja nekada prevazilaze mogućnosti inicijalno postavljene infrastrukture i zahtevaju dodatne resurse. Takođe, obično je slučaj da intenzitet sobraćaja nije uvek isti, već se dešava u naletima. Pošto su resursi skupi, pitanje je kako napraviti optimalnu infrastrukturu tako da ne troši puno novca dok nije pod opterećenjem, a takođe da ne prestane sa radom kada se to opterećenje neizbežno pojavi.

Razvoj računarstva u oblaku omogućio je korisnicima da na lak način dinamički kreiraju i uklanjaju instance servera. Kako bi se optimizovao pomenut problem, u trenucima kada je količina saobraćaja velika, kreiraju se dodatne instance servera sa programima koji opslužuju zahteve i saobraćaj se balansira između njih. Pored ovoga, postoje još razne zanimljive primene. Kako bi se bolje razumelo kakvu ulogu mogu da igraju virtuelni fajl sistemi u ovakvom kontekstu, biće naveden primer jedne konkretne aplikacije.

4.1 Primer upotrebe virtuelnih fajl sistema u oblaku

Neka je korisnik istraživač iz oblasti bioinformatike koji želi da vrši obradu nekih podataka. Neretko su podaci iz te oblasti nauke veoma veliki. Na primer, podaci koji se čitaju iz sekvencera, za ceo ljudski genom, često su veći od 100 gigabajta. Takođe,

korisnik želi da izvršava nekakve algoritme nad ovim podacima sa ciljem da dobije neke korisne informacije iz njih. Bioinformatički algoritmi su vrlo često računski intenzivni.

Uzimajući ove činjenice u obzir, postaje jasno da lični računari uglavnom nisu dorasli ovom zadatku. Kako zbog mesta na disku za čuvanje ovih podataka, tako i zbog ograničene procesorske moći. Ideja veb aplikacije je da korisniku omogući upravo ove dve stvari. Jedan deo aplikacije bi bio centralni i služio bi za skladištenje korisničkih podataka, dok bi drugi deo bio zadužen za njihovu obradu.

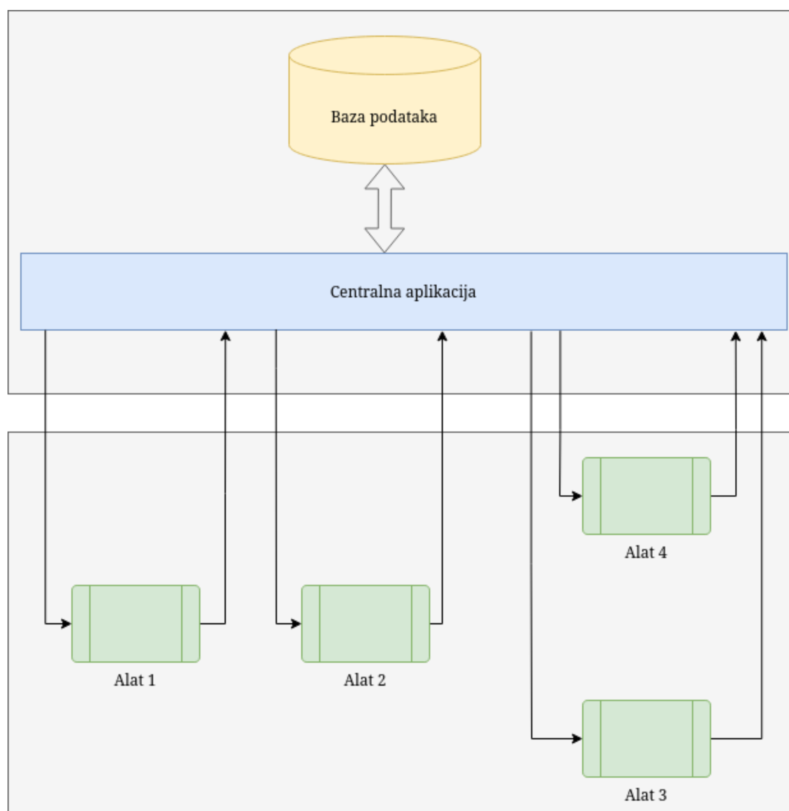
Bioinformatička istraživanja često zahtevaju kompoziciju više specijalizovanih alata gde svaki od njih predstavlja neki algoritam koji prima neke podatke na ulazu i rezultat mu je nekakav transformisani izlaz. Ovako transformisan izlaz se često postavlja kao ulaz nekog drugog alata koji vrši dalju obradu. Povezivanje alata na ovaj način može se zamisliti kao usmereni aciklični graf, gde su čvorovi grafa upravo ti alati, a grane predstavljaju tok podataka. U daljem tekstu će se ovaj graf nazivati radni tok.

Slede koraci kojima se može pojednostavljeno predstaviti ponašanje korisnika na platformi.

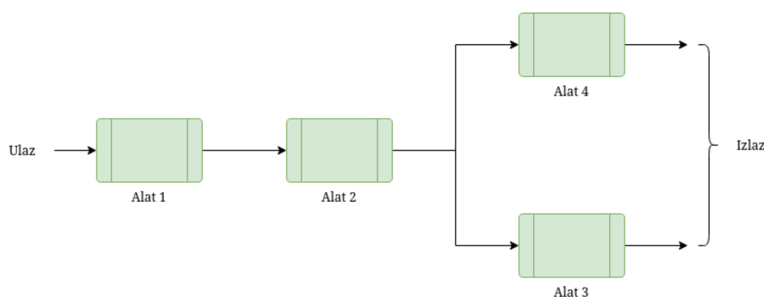
1. Dopremanje podataka na platformu.
2. Pokretanje radnog toka.
3. Analiza rezultata.

Za uspešno izvršavanje koraka 3, svi rezultati iz radnog toka treba da se nalaze na platformi nakon završetka rada. Važno je napomenuti da su korisniku od interesa ne samo finalni rezultati već i svi međurezultati (rezultati rada unutrašnjih čvorova radnog toka).

Navedeno je da su bioinformatički algoritmi računski intenzivne operacije, tako da je jedno od mogućih rešenja da se izvršavanje radnog toka realizuje tako što će se za svaki čvor u grafu dinamički kreirati instanca servera koja će izvršavati konkretan alat. Pre početka izvršavanja, ulazni podaci za alat moraju se sa centralnog servera dopremiti kako bi bili dostupni za obradu. Nakon izvršavanja, instanca se može ukloniti ili iskoristiti za neki drugi drugi alat, ali ključno je da pre nego što se instanca ukloni, svi rezultati izračunavanja budu otpremljeni na centralni server. Primer toka podataka u ovakvoj arhitekturi može se videti na slici 4.1, dok je izvršavanje radnog toka iz ugla korisnika prikazano slikom 4.2.



Slika 4.1: Tok podataka prilikom izvršavanja radnog toka.



Slika 4.2: Izvršavanje radnog toka iz ugla korisnika.

Kakvu ulogu ovde igraju virtuelni fajl sistemi? Velika količina bioinformatičkih alata su programi napisani od strane nekih drugih programera. Cilj je da se što više ovakvih gotovih alata iskoristi, ako je moguće, bez ikakvih prilagođavanja. Kako su ti alati pisani van konteksta opisane veb aplikacije, oni nemaju nikakvu ideju o ostatku sistema, niti o tome da se nalaze u čvoru nekog radnog toka. Svoje ulazne podatke čitaju sa lokalnog fajl sistema, i takođe na njemu čuvaju rezultat svog rada. Virtuelni fajl sistemi u ovom slučaju mogu ovim alatima da pruže interfejs rada na

lokalnom fajl sistemu, dok u pozadini vrše otpremanje i dopremanje podataka sa centralnog servera.

4.2 Program SFTPFS

Za potrebe rada implementiran je virtuelni fajl sistem. Ideja je da program demonstrira način na koji se implementira jedan program koji sinhronizuje lokalni fajl sistem sa fajl sistemom koji se nalazi na udaljenom serveru. Ovaj program može služiti krajnjim korisnicima da sinhronizuju fajlove, ali se u opštem slučaju može koristi da sinhronizuje dva servera kao što je opisano u primeru 4.1.

Zbog nedostatka neophodne infrastrukture opisane u primeru 4.1, za prenos podataka korišćen je protokol SFTP (SSH File Transfer Protocol). Treba napomenuti da bi se u slučaju specijalizovanih aplikacija koristio neki prilagođeni način za otpremanje i preuzimanje podataka.

Program je napisan u programskom jeziku Go. Izabran je pre svega zbog svojih odličnih performansi i zbog reputacije koje je stekao kao jezik za implementaciju sistemskih alata i programa u oblaku. Ogromna prednost programskog jezika Go u odnosu na druge jezike, je to što je je kompajliran i statički povezan. U kontekstu primera 4.1, a i generalno u sistemima u oblaku, nije potrebno posebno podešavati server na kome se program izvršava, već treba samo dopremiti *jedan* binarni fajl. Na primer, aplikacije pisane u programskom jeziku Java zahtevaju da se na serveru nalazi Java virtuelna mašina koja bi izvršila kôd. Takođe, bolji je od programskog jezika C jer se sve zavisnosti nalaze u okviru izvršnog fajla i ne zahteva se postojanje deljenih biblioteka u okruženju gde se aplikacija izvršava.

Aplikacija je otvorenog koda i nalazi se na servisu Github na adresi `https://github.com/doza-daniel/sftpfs`.

4.2.1 Detalji implementacije

Pošto program SFTPFS nije napisan u programskom jeziku C, nema pristup `libfuse` biblioteci pa je bilo potrebno naći zamenu. Zadatak ove biblioteke je da implementira komunikacioni protokol sa kernelskim modulom na niskom nivou, a korisniku da predstavi jednostavan interfejs u duhu programskog jezika. Komunikacija sa kernelom se odvija upisom i čitanjem iz specijalnog fajla na operativnom

sistemu. [1]

4.2.1.1 Izbor biblioteke i njen način rada

U programskom jeziku Go postoji više različitih implementacija protokola koji se koristi u okviru modula FUSE. U okviru programa SFTPFS korišćena je biblioteka `https://pkg.go.dev/github.com/jacobsa/fuse`. U nastavku će biti detaljnije opisano kako je ova biblioteka iskorišćna za implementaciju virtuelnog fajl sistema.

Navode se tri glavna modula od kojih se biblioteka sastoji.

1. `fuse`;
2. `fuseutil`;
3. `fuseops`.

Glavna metoda biblioteke se nalazi u okviru modula `fuse` i služi za postavljanje fajl sistema na zadatoj putanji. Isečak 4.1 prikazuje postavljanje fajl sistema na putanju `/mnt`. Ključna funkcija u ovom isečku je `fuse.Mount`. Ona prima nisku koja označava putanju na kojoj će fajl sistem biti postavljen, interfejs `fuse.Server` i konfiguracionu strukturu `fuse.MountConfig`. Treba primetiti da varijable `server` i `config` nisu inicijalizovane zbog konciznosti. Pored operacije postavljanja fajl sistema, dodat je i kôd koji reaguje na signale operativnog sistema koji zahtevaju prekid rada programa i izvršava neophodne operacije za korektan prekid rada programa. U slučaju da ovaj kôd nije dodat, prekid rada programa bi ostavio sistem u nekonzistentnom stanju i putanja `mnt` ne bi mogla da se koristi pre nego što korisnik ručno pozove `umount` komandu.

[1] Na operativnom sistemu Linux se ovaj fajl nalazi na putanji /dev/fuse.

Listing 4.1: Primer postavljanja fajl sistema.

```
var server fuse.Server
var config fuse.MountConfig

mountedFS, err := fuse.Mount("/mnt", server, &config)
if err != nil {
	log.Fatalf("mount␣error:␣%v", err)
}

sigs := make(chan os.Signal, 1)

signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)

go func() {
	<-sigs
	mountpoint := mountedFS.Dir()
	if err := fuse.Unmount(mountpoint); err != nil {
		log.Fatalf("failed␣to␣unmount:␣%v", err)
	}
}()

if err := mountedFS.Join(context.Background()); err != nil {
	log.Fatalf("join␣error:␣%v", err)
}
```

Potpis interfejsa `fuse.Server` dat u isečku 4.2. Njegova uloga je da bude spona između operacija koje zahteva kernel i njihove implementacije u programu. Metoda `ServeOps` prima kao argument strukturu `Connection` koja služi kao apstrakcija za komunikacioni protokol. Dve glavne metode vezane za ovu strukturu su `ReadOp` i `Reply`. Implementacija serverskog interfejsa može se opisati narednim koracima.

1. Pročitati sledeću pristiglu operaciju koristeći metodu `ReadOp`.
2. Utvrditi tip operacije i pozvati odgovarajuću implementaciju konkretne operacije.
3. Proslediti odgovor kernelu koristeći metodu `Reply`.

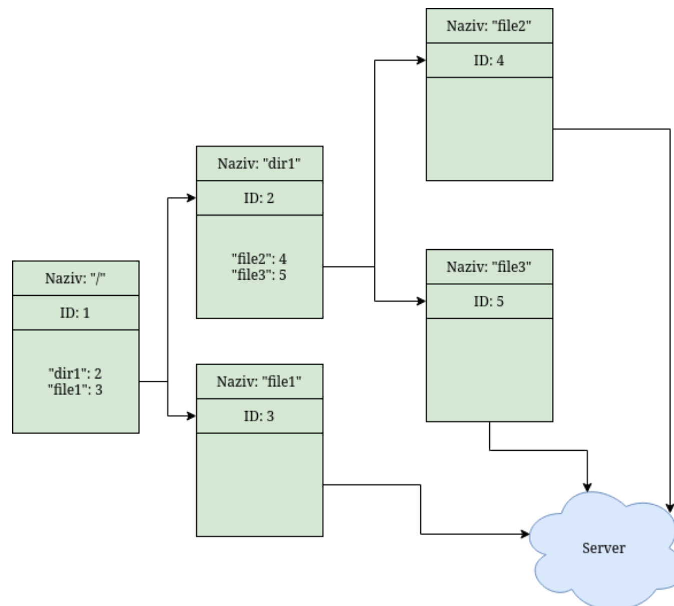
Listing 4.2: Server interfejs.

```
type Server interface {
        ServeOps(*Connection)
}
```

Interfejs `fuse.Server` vrlo precizno definiše očekivano ponašanje, tako da većina njegovih implementacija izgleda isto. Da se ne bi svaki put iznova implementirao, modul `fuseutil` pruža metodu `NewFileSystemServer` koja vraća podrazumevanu implementaciju. Ova metoda očekuje da joj se prosledi implementacija interfejsa `FileSystem`. Lista metoda koju definiše ovaj interfejs može se videti u tabeli 4.1. Logika programa se u velikoj meri svodi na implementaciju ovog interfejsa tako da će njegove metode i implementacija biti detaljnije opisane u sekciji 4.2.1.3.

4.2.1.2 Reprezentacija strukture fajlova i direktorijuma

Biblioteka nalaže rad sa strukturom podataka `inode` koja predstavlja fajlove i direktorijume u virtuelnom fajl sistemu. Pojednostavljeno[2], sadržaj fajl sistema je stablo čiji su čvorovi upravo strukture `inode`. Ilustracija je data na slici 4.3.



Slika 4.3: Inode sturktura stabla.

Svaki čvor ima svoj jedinstveni identifikator po kome mu se pristupa. Glavni zadatak programa je da održava strukturu opisanu na slici 4.3 i da odgovori na

[2]Nisu razmatrani simbolički linkovi. U tom slučaju struktura bi bila graf.

Tabela 4.1: Lista metoda interfejsa FileSystem.

Potpis metode
`StatFS(context.Context, *fuseops.StatFSOp) error`
`LookUpInode(context.Context, *fuseops.LookUpInodeOp) error`
`GetInodeAttributes(context.Context, *fuseops.GetInodeAttributesOp) error`
`SetInodeAttributes(context.Context, *fuseops.SetInodeAttributesOp) error`
`ForgetInode(context.Context, *fuseops.ForgetInodeOp) error`
`BatchForget(context.Context, *fuseops.BatchForgetOp) error`
`MkDir(context.Context, *fuseops.MkDirOp) error`
`MkNode(context.Context, *fuseops.MkNodeOp) error`
`CreateFile(context.Context, *fuseops.CreateFileOp) error`
`CreateLink(context.Context, *fuseops.CreateLinkOp) error`
`CreateSymlink(context.Context, *fuseops.CreateSymlinkOp) error`
`Rename(context.Context, *fuseops.RenameOp) error`
`RmDir(context.Context, *fuseops.RmDirOp) error`
`Unlink(context.Context, *fuseops.UnlinkOp) error`
`OpenDir(context.Context, *fuseops.OpenDirOp) error`
`ReadDir(context.Context, *fuseops.ReadDirOp) error`
`ReleaseDirHandle(context.Context, *fuseops.ReleaseDirHandleOp) error`
`OpenFile(context.Context, *fuseops.OpenFileOp) error`
`ReadFile(context.Context, *fuseops.ReadFileOp) error`
`WriteFile(context.Context, *fuseops.WriteFileOp) error`
`SyncFile(context.Context, *fuseops.SyncFileOp) error`
`FlushFile(context.Context, *fuseops.FlushFileOp) error`
`ReleaseFileHandle(context.Context, *fuseops.ReleaseFileHandleOp) error`
`ReadSymlink(context.Context, *fuseops.ReadSymlinkOp) error`
`RemoveXattr(context.Context, *fuseops.RemoveXattrOp) error`
`GetXattr(context.Context, *fuseops.GetXattrOp) error`
`ListXattr(context.Context, *fuseops.ListXattrOp) error`
`SetXattr(context.Context, *fuseops.SetXattrOp) error`
`Fallocate(context.Context, *fuseops.FallocateOp) error`
`Destroy()`

zahteve koji su vezani za nju. Ti zahtevi mogu biti, na primer, kreiranje, listanje (ako je u pitanju direktorijum), postavljanje atributa itd.

Program SFTPFS koristi interfejse date u isečku 4.3. Najopštiji je interfejs `Inode` a njega proširuju `FileInode` i `DirInode`. Treba napomenuti da se u trenutnoj implementaciji, interfejs `FileInode` ne razlikuje od interfejsa `Inode` ali je dodat kako bi u kodu mogle da se implementiraju provere koje fajlove razlikuju od direktorijuma. Takođe ako u budućnosti bude potrebe za novim metodama vezano isključivo za fajlove, taj interfejs će biti pravo mesto za njihovo dodavanje.

Listing 4.3: Interfejsi Inode, FileInode i DirInode.

```
type Inode interface {
        InodeID() fuseops.InodeID
        Name() string
        GetAttributes() *fuseops.InodeAttributes
}

type FileInode interface {
        Inode
}

type DirInode interface {
        Inode
        LookUpChild(ctx context.Context, name string) Inode
        GetEntries(ctx context.Context) ([]Inode, error)
        AddEntry(name string, in Inode)
        RemoveEntry(name string)
}
```

4.2.1.3 Implementacija interfejsa fajl sistema

Centralnu ulogu programa SFTPFS ima upravo implementacija `FileSystem` interfejsa. U okviru programa nalazi se paket `filesystem` koji sadrži istoimenu strukturu koja predstavlja ovu implementaciju. U okviru ove strukture, između ostalog, nalaze se dve mape. Mapa `inodes` pruža pristup svim fajlovima i direktorijumima na osnovu njihovog identifikatora, dok mapa `handles` predstavlja fajlove i foldere koji su otvoreni sistemskim pozivom `open`. Održavanje stanja ove dve mape je

ključno za implementaciju fajl sistema i njihova upotreba će biti detaljnije opisana u nastavku kada se budu opisivale konkretne metode koje ih koriste.

Listing 4.4: Struktura filesystem.

```
type filesystem struct {
        inodes       map[fuseops.InodeID]inode.Inode
        nextHandleID func() fuseops.HandleID

        handles     map[fuseops.HandleID]handle.Handle
        nextInodeID func() fuseops.InodeID

        sftpClient *sftp.Client
}
```

Treba obratiti pažnju na potpis metoda interfejsa `FileSystem`. Primećuje se da sve metode (osim metode `Destroy`) imaju vrlo sličan potpis. Prvi argument metode je interfejs `context.Context`. Ovaj interfejs je jedan od najkorišćenijih interfejsa iz standardne biblioteke programskog jezika Go. Najčešće se koristi da signalizira metodama koje se dugo izvršavaju da treba da prekinu sa radom. Pored ovoga, sadrži i mapu netipiziranih vrednosti koje pružaju dodatne informacije metodi pored njenih argumenata. Drugi argument je pokazivač na strukturu koja predstavlja operaciju iz paketa `fuseops`. Ova struktura sadrži polja sa neophodnim informacijama za metodu koja se izvršava. Dodatno, u nekim slučajevima se od metode zahteva da postavi neko polje ove strukture kao rezultat izvršene operacije. U nastavku su opisane neke od metoda `FileSystem` interfejsa u cilju oslikavanja rada fajl sistema.

Metoda `StatFS` treba da pruži statistiku o samom fajl sistemu. Konkretno, metoda treba da postavi polja strukture `StatFSOp` na odgovarajuće vrednosti. Polja ove strukture prikazana su u isečku 4.5. Kako ove vrednosti u slučaju programa SFTPFS ne oslikavaju vrednosti fizičkog diska, jedino što je važno je da ove vrednosti pružaju informaciju da je fajl sistem dovoljno velikog kapaciteta.

Listing 4.5: Struktura StatFSOp

```
type StatFSOp struct {
        BlockSize       uint32
        Blocks          uint64
        BlocksFree      uint64
        BlocksAvailable uint64
        IoSize          uint32
        Inodes          uint64
        InodesFree      uint64
}
```

Metoda `LookUpInode` služi za pronalaženje fajla ili foldera nad kojim treba da se izvrši neka operacija. Argumenti su joj polja `Parent` i `Name` strukture `LookUpInodeOp`. Prvi argument predstavlja identifikator direktorijuma u kome treba da se nađe `inode` sa imenom `Name`. Ako takav `inode` postoji, onda se kao odgovor popunjava polje `Entry`, inače se vraća greška.

Kako bi se bolje prikazala upotreba ove operacije, biće naveden jedan primer. Svaka metoda interfejsa `FileSystem` koja treba da izvrši neku operaciju nad konkretnim fajlom ili direktorijumom, zahteva da joj se prosledi identifikator strukture `inode` tog fajla ili direktorijuma. Pošto sistemski pozivi rade sa putanjama, zadatak virtuelnog fajl sistema je da pronađe taj identifikator. Na primer ako se pozove sistemski poziv `open` nad putanjom `/dir1/file2` sa slike 4.3, metoda `LookUpInode` u ovom slučaju biće pozvana dva puta. Prvi put da nađe identifikator direktorijuma `dir1` u korenom direktorijumu[3], a nakon toga da nađe identifikator fajla `file2` u okviru direktorijuma `dir1`. Tek nakon što je identifikator željenog fajl pronađen, poziva se metoda `OpenFile` koja taj identifikator koristi za izvršavanje operacije.

Argumenti pomenute metode `OpenFile` se nalaze u okviru `OpenFileOp` strukture. Polje `Inode` označava identifikator fajla koji treba otvoriti. Još jedan argument koji ova operacija dobija jeste polje `OpenFlags` koje je tipa `fusekernel.OpenFlags`. Ovaj tip je samo imenovana verzija primitivnog tipa `uint32` čiji bitovi predstavljaju standardne flegove za otvaranje fajlova. Polje `Handle` biće objašnjeno kratkim primerom. Naime, kada neka aplikacija pozove sistemski poziv `open`, kao odgovor od kernela dobije fajl deskriptor. Naredni sistemski pozivi za rad sa otvorenim fajlom primaju pomenuti fajl deskriptor i na osnovu njega vrše odgovarajuću operaciju. Po-

[3] Koreni direktorijum ima predefinisan identifikator 1.

lje `Handle` služi kao jedinstveni identifikator otvorenog fajla. Zadatak implementacije fajl sistema je da u sklopu odgovora na `OpenFile` metodu, generiše ovakav identifikator i postavi polje `Handle` na generisanu vrednost. Naredni pozivi nad otvorenim fajlom, na primer `ReadFile`, imaće postavljen ovaj identifikator. Na ovaj način, fajl sistem može da vodi evidenciju o otvorenim fajlovima i operacijama nad njima. Upravo ovome služi pomenuta mapa `handles` u strukturi prikazanoj u isečku 4.4.

Listing 4.6: Struktura OpenFileOp.

```
type OpenFileOp struct {
        Inode      InodeID
        OpenFlags  fusekernel.OpenFlags
        Handle     HandleID

        // ...
}
```

Odgovor na sistemski poziv `close` je metoda `ReleaseFileHandle`. Argument ovoj metodi je polje `Handle` strukture `ReleaseFileHandleOp`. Svrha ove metode je da signalizira da je fajl zatvoren i da fajl sistem može osloboditi resurse vezane za identifikator `Handle`.

Još jedna metoda koja se poziva po zatvaranju fajla je `FlushFile`. Neke implementacije fajl sistema ne upisuju odmah sve promene direktno na krajnju destinaciju (disk, otpremanje na udaljeni server itd.) već zarad performansi koriste keš i u nekom trenutku asinhrono beleže promene. Ova operacija služi da signalizira takvim sistemima da bi trebalo promene koje se nalaze u kešu trajno zabeležiti. Implementacije fajl sistema mogu odlučiti da ignorišu ovu operaciju, tako u kontekstu programa SFTPFS ova operacija nije implementirana jer se pisanje fajla vrši direktnim pozivom ka serveru, bez keširanja.

Metoda `GetInodeAttributes` treba da odgovori sa atributima tražene strukture `inode`. Identifikator strukture se prosleđuje u vidu polja `Inode` u okviru strukture `GetInodeAttributesOp`. Odgovor se upisuje u polje `Attributes`. Ova metoda nije vezana za konkretan sistemski poziv. Naime, navigacija po fajl sistemu zahteva čest pristup strukturama `inode`. Kako bi se izbeglo često čitanje diska, virtuelni fajl sistem održava keš njihovih atributa. Ako je keš ustajao, metoda `GetInodeAttributes` se poziva kako bi se keš osvežio. Vreme za koje unos u kešu postaje ustajao, može biti kontrolisano od strane implementacije fajl sistema tako što će postaviti parametar `AttributesExpiration`.

Listing 4.7: Struktura GetInodeAttributesOp.

```
type GetInodeAttributesOp struct {
        Inode                InodeID
        Attributes           InodeAttributes
        AttributesExpiration time.Time
        // ...
}
```

Poslednja metoda koja će biti opisana je `ReadFile`. Ona služi da pruži odgovor na sistemski poziv `read`. Svoje argumente ova metoda čita iz strukture `ReadFileOp`. Polje `Inode` je identifikator za strukturu `inode` koja predstavlja fajl koji iz kog treba čitati, a `Handle` predstavlja opisani identifikator za otvoren fajl. Polja `Offset` i `Size` određuju od koje pozicije u fajlu treba čitati i koliko bajtova treba pročitati. Rezultat čitanja se postavlja u polje `Dst` i broj pročitanih bajtova se postavlja u polje `BytesRead`.

Listing 4.8: Struktura ReadFileOp.

```
type ReadFileOp struct {
        Inode     InodeID
        Handle    HandleID
        Offset    int64
        Size      int64
        Dst       []byte
        BytesRead int

        // ...
}
```

Na slici 4.4 je prikazano izvršavanje programa `cat` nad fajlom u virtuelnom fajl sistemu implementiranom u okviru programa SFTPFS. U gornjem terminalu je izvršen poziv programa `cat`, a u donjem terminalu je pokrenut program SFTPFS koji štampa sve operacije virtuelnog fajl sistema koje izvršava. Mogu se primetiti sve opisane operacije i red kojim se izvršavaju.

4.2.1.4 Komunikacija sa udaljenim serverom

Za komunikaciju sa udaljenim serverom, korišćen je klijent iz biblioteke `https://pkg.go.dev/github.com/pkg/sftp`. Prednost ove biblioteke je što poseduje im-

```
[daniel@vgl ~] $ cat /tmp/mnt/cat_test.txt
cat test!
[daniel@vgl ~] $ 

[daniel@vgl sftpfs] [master*]$ ./sftpfs -m /tmp/mnt -u mi13119 -p
Enter Password:
2022/09/22 19:45:02 GetInodeAttributes[InodeID: 1]
2022/09/22 19:45:02 LookUpInode[Parent: 1, Name: cat_test.txt
2022/09/22 19:45:02 GetInodeAttributes[InodeID: 22]
2022/09/22 19:45:02 OpenFile[Inode: 22, Handle: 1]
2022/09/22 19:45:02 ReadFile[InodeID: 22, HandleID: 1]
2022/09/22 19:45:02 FlushFile[InodeID: 22, HandleID: 1]
2022/09/22 19:45:02 ReleaseFileHandle[Handle: 1]
```

Slika 4.4: Prikaz log linija izvršenih operacija u fajl sistemu tokom izvršavnja programa *cat*.

plementaciju svih funkcija koje su potrebne fajl sistemu.

Na samom početku programa SFTPFS, u funkciji `main`, inicijalizuje se klijent `sftp.Client`. Ovaj klijent zahteva uspostavljanje SSH konekcije sa serverom, tako da je neophodna autentikacija korisnika. Korisnik može proslediti svoje korisničko ime putem argumenata komandne linije ili putem promenljivih iz okruženja. Adresa servera se takođe može zadati argumentima komandne linije, a podrazumevana vrednost je `alas.math.rs:22`.

Nakon uspešne inicijalizacije, klijent se postavlja u strukturu opisanu u isečku 4.4 putem funkcije `filesystem.New`. Tokom izvršavanja, ova funkcija komunicira sa udaljenim serverom, čita hijerarhiju fajlova i direktorijuma i kreira odgovarajuću strukturu `Inode` objekata lokalno. Treba naglasiti da se sadržaj fajlova ne čita u ovom trenutku, već samo njihovi atributi.

Biće predstavljen primer čitanja fajla. U metodi `OpenFile` otvara se udaljeni fajl i dobija se njegova apstrakcija `sftp.File`. Kreira se u objekat tipa `FileHandle` i u njega se smešta navedena fajl apstrakcija. Ovako kreiran `FileHandle` objekat se registruje u mapu sa ključem koji odgovara generisanom `Handle` identifikatoru. Kada na red dođe izvršavanje metode `ReadFile`, objekat `FileHandle` se dohvata na osnovu identifikatora, i koristi se kako bi se pozvala metoda `ReadAt` nad fajl apstrakcijom koja čita sadržaj fajla sa udaljenog servera. Pročitani sadržaj se smešta u polje `Dst` strukture `ReadFileOp` i ovim putem se vraća kernelu. Metoda `ReleaseFileHandle` oslobađa resurse koje je zauzela fajl apstrakcija pozivanjem metode `Close` i briše `FileHandle` iz mape.

Glava 5

Zaključak

U ovom radu je prikazan način na koji virtuelni fajl sistemi u korisničkom adresnom prostoru mogu da se iskoriste u računarstvu u oblaku. Ključna tehnologija koja ovo omogućava je modul FUSE.

Glavni deo rada prikazuje detaljan opis implementacije virtuelnih fajl sistema u programskom jeziku Go. Ovaj opis je dat kako bi pokazao da se na efikasan način može dobiti rešenje koje donosi veliku vrednost kompleksnim veb aplikacijama. Analiziran je i jedan konkretan primer takve aplikacije koji naglašava korisnost primene virtuelnih fajl sistema u oblaku. Takođe, doprinos ovog rada je i program SFTPFS koji je otvorenog koda koji se može koristiti ili proširivati za specijalizovane potrebe.

Trebalo bi naglasiti da, iako virtuelni fajl sistemi predstavljaju elegantno rešenje za sinhronizaciju fajlova na udaljenim serverima, njihova implementacija nije lak zadatak. Tokom izrade rada, primećeno je da može doći do raznih nepredviđenih problema. Jedan od problema na koji se naišlo jeste nestabilnost mreže. Zarad uklanjanja ovog problema, razmatrana je implementacija lokalnog keša čiji bi se sadržaj sinhronizovao sa serverom u trenutku kada se mreža stabillizuje. Međutim, ovo otvara nova pitanja, na primer, kako garantovati konzistentnost podataka. Još jedan izazov koji je uočen tokom implementacije je to što je interfejs za komunikaciju grešaka veoma ograničen, što stvara poteškoće prilikom njihovog tumačenja. Uočeni problemi otvaraju mogućnost za dalji razvoj u cilju poboljšanja implementacija virtuelnih fajl sistema.

U svakom slučaju, virtuelni fajl sistemi mogu pružiti efikasan i jednostavan način za rešavanje jednog od glavnih izazova računarstva u oblaku, a to je sihnronizacija podataka.

Bibliografija

[1] Julia Computing Inc. FUSE manual page, Linux, Feb 2018. on-line at: `https://man7.org/linux/man-pages/man4/fuse.4.html`.

[2] Miroslav Marić. *Operativni Sistemi*. Univerzitet u Beogradu - Matematički fakultet, Studentski trg 16, Beograd, 2015.

[3] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th ed. Perason Education, Inc., Upper Saddle River, New Jersey, USA, 2008.