Prilagođena ljuska za Linux operacijski sustav u ugradbenom sustavu

Brođanac, Marko

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij elektrotehnike

PRILAGOĐENA LJUSKA ZA LINUX OPERACIJSKI SUSTAV U UGRADBENOM SUSTAVU

Diplomski rad

Marko Brođanac

Osijek, 2023.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 12.09.2023.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Marko Brođanac
Studij, smjer:	Diplomski sveučilišni studij Elektrotehnika, smjer Komunikacije i informatika'
Mat. br. Pristupnika, godina upisa:	D-1360, 07.10.2021.
OIB studenta:	41701410330
Mentor:	prof. dr. sc. Marijan Herceg
Sumentor:	,
Sumentor iz tvrtke:	Zvonimir Kaprocki
Predsjednik Povjerenstva:	izv. prof. dr. sc. Ratko Grbić
Član Povjerenstva 1:	prof. dr. sc. Marijan Herceg
Član Povjerenstva 2:	prof. dr. sc. Mario Vranješ
Naslov diplomskog rada:	Prilagođena ljuska za Linux operacijski sustav u ugradbenom sustavu
Znanstvena grana diplomskog rada:	**Telekomunikacije i informatika (zn. polje elektrotehnika)**
Zadatak diplomskog rada:	Ljuska (engl. shell) predstavlja sučelje između korisnika i jezgre Linux operacijskog sustava. Ljuska je aplikacija koja interpretira naredbe dobivene s tipkovnice i izvršava različite programe u ovisnosti o dobivenim naredbama. Primjer Linux ljuske je BASH ljuska. U okviru ovog diplomskog rada potrebno je proučiti osnovne funkcionalnosti postojećih ljuski, npr. BASH ljuske i razviti vlastitu ljusku na Linux ugradbenoj platformi koja će omogućiti rad s datotekama (kopiranje, brisanje, kreiranje novih datoteka, promjenu prava nad datotekama, upravljanje korisnicima, postavljanje mrežnih parametara ,itd.). Nadalje, Linux jezgru je potrebno prilagoditi i pokrenuti na ugradbenoj platformi Raspberry PI 3. Ljuska treba biti napravljena korištenjem C/C++
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	12.09.2023.

Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.*
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 24.09.2023.

Ime i prezime studenta:	Marko Brođanac
Studij:	Diplomski sveučilišni studij Elektrotehnika, smjer Komunikacije i informatika'
Mat. br. studenta, godina upisa:	D-1360, 07.10.2021.
Turnitin podudaranje [%]:	3

Ovom izjavom izjavljujem da je rad pod nazivom**: Prilagođena ljuska za Linux operacijski sustav u ugradbenom sustavu**

izrađen pod vodstvom mentora prof. dr. sc. Marijan Herceg

i sumentora ,

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

Ljuska (engl. *shell*) pruža sučelje koje korisniku omogućuje interakciju s jezgrom (engl. *kernel*) operacijskog sustava. Svaki put kada se korisnik prijavi u sustav ili otvori terminal, operacijski sustav pokreće ljusku. Ona interpretira naredbe koje korisnik unese u terminal te ih prevodi u zapis koji jezgra može izvršiti. Naredbe se koriste za, primjerice, rad s datotekama (kopiranje, brisanje, kreiranje novih datoteka, promjena prava nad datotekama), navigaciju kroz direktorije, upravljanje korisnicima, procesima itd. Osim naredbi, korisnik putem ljuske može izvršavati različite programe i skripte. Putem skripti korisnik ima mogućnost automatizacije izvođenja složenih zadataka. Ljuska definira vlastita sintaksna pravila za pisanje skripti pa se može smatrati oblikom programskog jezika jer uključuje petlje, uvjete, osnovne matematičke operacije te definiranje funkcija i varijabli. Radno okruženje ljuske moguće je prilagoditi i konfigurirati prema vlastitim potrebama što olakšava njezino korištenje. Najčešće korištena ljuska u *Linux* operacijskim sustavima je BASH (engl. *Bourne Again Shell*) [1] ljuska.

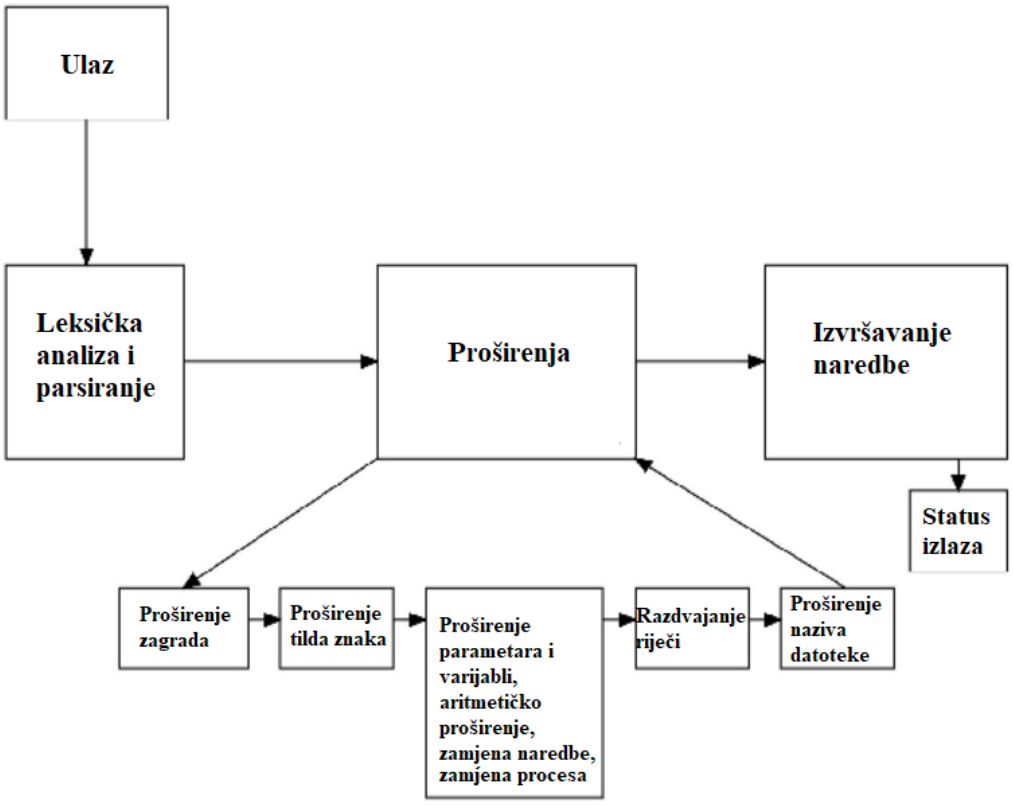
U okviru ovog rada razvijena je ljuska za *Linux* operacijski sustav po uzoru na BASH ljusku u programskom jeziku C. Za početak opisane su značajke i način rada ljuske na primjeru BASH ljuske. Zatim je opisan proces izrade ljuske predstavljene u ovom radu i mogućnosti koje ona pruža. Osim toga, opisana je izgradnja minimalne distribucije *Linux* operacijskog sustava za ugradbenu platformu *Raspberry Pi 3 Model B* na kojoj je pokrenuta razvijena ljuska. Zadnje poglavlje donosi evaluaciju razvijene ljuske.

2. ZNAČAJKE I NAČIN RADA LJUSKE

U ovom poglavlju bit će opisan način obrade unosa, parsiranje, različita proširenja riječi te izvršenje naredbi u BASH ljusci. Osim toga, ovo poglavlje donosi i pregled ostalih ljuski za *Linux* operacijski sustav.

Kada se ulazni podaci čitaju iz terminala ili skripte, oni prolaze kroz više faza transformacije, korak po korak (slika 2.1.). U osnovi, ljuska izvodi sljedeće zadatke [1]:

1. Čita unos iz datoteke ili s terminala.
2. Rastavlja unos u riječi odnosno tokene prema određenim pravilima.
3. Parsira odnosno raščlanjuje tokene na jednostavne ili složene naredbe te argumente.
4. Izvodi različita proširenja ljuske.
5. Izvodi sva potrebna preusmjeravanja te uklanja operatore preusmjeravanja s popisa argumenata.
6. Izvršava naredbu.
7. Čeka da se naredba izvrši te prikuplja njezin status na izlazu.



Slika 2.1. Dijagram toka izvođenja naredbe u BASH ljusci [2]

2.1. Čitanje unosa

Pri čitanju unosa s terminala BASH je u interaktivnom, dok je kod čitanja naredbi iz datoteke odnosno skripte u neinteraktivnom načinu rada te ne postoji interakcija s korisnikom preko terminala. U interaktivnom načinu BASH koristi *readline* [3] biblioteku koja osim samog čitanja unosa omogućava uređivanje naredbenog retka. Također, ova biblioteka nudi niz funkcija koje korisnicima omogućavaju spremanje unesenih naredbi, ponovno pozivanje prethodnih naredbi, pomicanje kursora po retku, umetanje i uklanjanje teksta, dovršavanje djelomično upisanih riječi. BASH i *readline* biblioteka su zajedno razvijeni, no u njoj ne postoji specifično napisan kod za BASH. U neinteraktivnom načinu rada, umjesto *readline* biblioteke, BASH koristi vlastite mehanizme za čitanje unosa. Oba načina rada daju jednak izlaz, a to je niz znakova koji završava znakom za novi red '\n' [2]**.** BASH uvijek čita skriptu ili unos s terminala red po red. Ako redak završava znakom obrnute kose crte, ' \ ', BASH čita idući redak prije obrade unosa i dodaje ga trenutnom retku [4].

2.2. Parsiranje

Parser je implementiran koristeći YACC (engl. *Yet Another Compiler-Compiler*) [5] alat. YACC kao ulaz prima pravila BASH jezika prema kojima kreira parser u C programskom jeziku. C. Ramey, jedan od autora BASH ljuske, u radu [2] navodi kako pravila BASH jezika nisu naročito prikladna za generiranje parsera YACC alatom što zahtijeva kompleksnu leksičku analizu. Također, on navodi kako bi rekurzivno-silazni parser pojednostavio implementaciju.

Proces parsiranja započinje leksičkom analizom. Nakon što je unos u naredbenoj liniji pročitan, leksički analizator provjerava sadrži li on navodnike. Prvi pronađeni znak navodnika postavlja sve znakove koji slijede u stanje navodnika (engl. *quoted state*) sve do idućeg znaka navodnika iste vrste. Ako stanje navodnika počinje dvostrukim navodnikom, ("..."), svi znakovi osim ‘"‘, ‘$‘, ‘`‘, i ‘\‘ gube svoje posebno značenje. Također, ako stanje navodnika počinje jednostrukim navodnikom, ('...'), tada svi znakovi osim ‘ ' ‘, gube svoje posebno značenje. U istoj naredbenoj liniji moguće je navesti više odvojenih naredbi koristeći znak ‘;’ pri čemu se naredbe izvršavaju jedna za drugom. Nakon toga, leksički analizator rastavlja niz znakova na tokene ovisno o metaznakovima, zatim identificira tokene s obzirom na kontekst te ih naposlijetku prosljeđuje parseru koji ih sastavlja u naredbe. Tokeni su odvojeni metaznakovima, a pod metaznakove se ubrajaju razmak, tabulator, novi red ili jedan od sljedećih znakova: ‘|’, ‘&’, ‘;’, ‘(’, ‘)’, ‘<’ ili ‘>’. S obzirom na kontekst, token se može kategorizirati kao riječ, rezervirana riječ, varijabla itd. Više

riječi odvojenih razmacima čine jednostavnu naredbu zajedno s njezinim argumentima, npr. „*ls -l -a*“, dok složenu naredbu čine jednostavne naredbe koje su na određen način povezane, npr. koristeći cjevovode. Nakon što je unos rastavljen na riječi, parser obavlja različita proširenja ljuske. Nakon što se proširenja izvedu, parser briše navodnike.

2.3. Proširenja ljuske

Proširenje ljuske je postupak kojim ljuska modificira unos prije izvršavanja naredbi. Ovim postupkom različite konstrukcije i posebni simboli pretvaraju se u stvarne vrijednosti ili putanje datoteka što doprinosi učinkovitijem radu s datotekama i podacima. Proširenja ljuske izvode se sljedećim redoslijedom [1] :

1. Proširenje zagrada (engl. *brace expansion*)
2. Proširenje tilda znaka (engl. *tilde expansion*), proširenje parametra i varijable (engl. *parameter and variable expansion*), aritmetičko proširenje (engl. *arithmetic expansion*, zamjena naredbe (engl. *command substitution*). Ova se proširenja izvode u isto vrijeme, s lijeva na desno.
3. Razdvajanje riječi (engl. *word splitting*)
4. Proširenje naziva datoteke (engl. *filename expansion*)

2.3.1. Proširenje zagrada

Proširenje zagrada je mehanizam kojim se mogu generirati proizvoljni nizovi znakova (engl. *strings*). Uzorak koji se proširuje ima oblik prefiksa, poslije čega slijedi serija nizova znakova odvojenih zarezima unutar vitičastih zagrada, što je praćeno mogućim dodatkom na kraju [1]. Na slici 2.2 prikazan je primjer upotrebe ovog tipa proširenja ljuske. Prva linija predstavlja upisanu naredbu dok druga linija predstavlja njezino proširenje. Obje linije daju jednak rezultat.

```
mkdir /tmp/{jedan,dva,tri}

mkdir /tmp/jedan /tmp/dva /tmp/tri
```

Slika 2.2. Proširenje zagradama [6]

2.3.2. Proširenje tilda znaka

Ako riječ počinje s tilda znakom „~“, svi znakovi do prve kose crte se smatraju tilda prefiksom koji se tretira kao korisničko ime. Ako tilda prefiks nije upisan, tada se tilda zamjenjuje s vrijednošću *HOME* varijable što se može vidjeti na slici 2.3.

```
cd ~/novaLjuska

cd $HOME/novaLjuska
```

Slika 2.3. Proširenje tilda znaka[6]

2.3.3. Proširenje parametra i varijable

Kada se koristi simbol dolara „$“, to znači da će ljuska izvršiti proširenje varijable ili parametra. Ovaj simbol također može označavati aritmetičko proširenje ili zamjenu naredbe. U najjednostavnijem slučaju, ovo proširenje zamjenjuje naziv varijable ili parametra s njegovom vrijednošću (slika 2.4.). Ako nakon parametra slijedi znak koji nije dio njegova naziva tada je potrebno naziv parametra navesti unutar vitičastih zagrada, ${*parametar*}[1] .

```
shell=BASH
echo "Ovo je $shell"

echo "Ovo je BASH"
```

Slika 2.4. Proširenje parametara i varijabli **[6]**

Ovo proširenje je prilično opširno i pruža razne mogućnosti poput dohvaćanja duljine vrijednosti, uklanjanja dijela vrijednosti koji se podudara s određenim uzorkom s početka ili kraja, zamjene dijela vrijednosti koji se podudara s određenim uzorkom. Osim toga, različita proširenja parametara odvijaju se ovisno o tome je li varijabla postavljena ili nije. Na primjer, ${*parametar:-riječ*} će zamijeniti naziv parametra njegovom vrijednošću ukoliko je parametar postavljen, u suprotnom će naziv parametra zamijeniti riječ [2].

2.3.4. Aritmetičko proširenje

Aritmetičko proširenje koristi se za izvođenje jednostavnih aritmetičkih izraza (slika 2.5.) te započinje s dolar „$“ znakom nakon čega slijedi izraz unutar dvostruke zagrade, $((*izraz*)) [1].

```
echo "Rezultat od 23*4 je $((23*4))"

echo "Rezultat od 23*4 je 92"
```

Slika 2.5. Aritmetičko proširenje **[6]**

2.3.5. Zamjena naredbe

Korištenje ovog proširenja omogućava da izlaz naredbe zamijeni naziv naredbe (slika 2.6.). Sintaksa je $(*naredba*) ili `*naredba*`.

```
echo "Danasnji datum je: $(date)"

echo "Danasnji datum je Wed 12 Jul 16:00:36 CEST 2023"
```

Slika 2.6. Zamjena naredbe **[6]**

2.3.6. Razdvajanje riječi

Izlaz koji nastaje nakon proširenja parametra, aritmetičkog proširenja i zamjene naredbe razdvaja se u riječi. Pri tome je bitno napomenuti da se razdvajanje riječi izvodi na navedenim proširenjima koja se ne pojavljuju unutar dvostrukih navodnika. U ovom procesu ljuska prolazi kroz sve znakove unosa te svaki put kada naiđe na znak za razdvajanje koji se nalazi u IFS (engl. *Input Field Separator*) varijabli, riječ se razdvaja. IFS varijabla standardno sadrži razmak, tabulator i znak za novi red, a moguće ju je postaviti na proizvoljni znak. Ukoliko postoji više znakova za razdvajanje u nizu, tada se oni zamjenjuju sa jednom instancom znaka [6]. Na slici 2.7. prikazan je primjer skripte u kojoj ljuska ovim procesom stvara više argumenata varijable *programs* pri čemu uklanja višestruke IFS znakove.

```
programs="/usr/bin/bash /usr/bin/zshell     /usr/bin/new"
for program in $programs
do
        echo "${program}"
done
```

(a)

```
/usr/bin/bash
/usr/bin/zshell
/usr/bin/new
```

(b)

Slika 2.7. Primjena procesa razdvajanja riječi. (a) Primjer skripte. (b) Ispis skripte. [6]

2.3.7. Proširenje naziva datoteke

Nakon što se odvio proces razdvajanja riječi, ljuska prolazi kroz svaku riječ te provjerava pojavljuju li se sljedeći znakovi: „*“, „?“ i „[“. Ukoliko se jedan od ovih znakova pojavljuje, riječ se smatra uzorkom te se zamjenjuje s listom naziva datoteka iz trenutnog direktorija koji odgovaraju uzorku [1]. Znak „?“ odgovara bilo kojem pojedinačnom znaku dok znak „*“ odgovara bilo kojem nizu znakova, uključujući i prazan niz znakova. Primjerice, ako određeni direktorij sadrži sljedeće datoteke: „*inv1jig.c inv2jig.c inv3jig.c invinitjig.c invpar.c*“, tada će naredba „*ls inv*jig.c*“ ispisati „*inv1jig.c inv2jig.c inv3jig.c invinitjig.c*“ što znači da se ispisuju svi

nazivi koji se podudaraju sa znakovima prije i nakon „*“ znaka koji se zamjenjuje nizom znakova neodređene duljine. U drugom slučaju, naredba „*ls inv?jig.c*“ će ispisati „*inv1jig.c inv2jig.c inv3jig.c*“. Tada ljuska pronalazi jedan znak koji bi mogao zamijeniti znak „?“. Naposljetku, naredbom „*ls inv[13]jig.c*“ ispisat će se „*inv1jig.c inv3jig.c*“ odnosno ispisuju se nazivi datoteka koji sadrže jedan od znakova unutar uglatih zagrada. BASH omogućava kombiniranje ova tri proširenja što, primjerice, omogućava izvršavanje naredbe „*ls *[0-9]*.[co]*“. Ova naredba ispisat će sve izvorne (.c) ili objektne (.o) datoteke koje sadržavaju broj u nazivu [7].

2.4. Izvršavanje naredbi

Ulaz u fazu izvršavanja naredbe predstavlja struktura naredbe koju je izgradio parser i skup proširenih riječi. Najčešće se pojavljuju jednostavne naredbe. Ako naziv naredbe ne predstavlja naziv funkcije ili ugrađene naredbe, BASH prolazi kroz datotečni sustav kako bi pronašao izvršnu datoteku s tim nazivom. Popis direktorija u kojima ljuska pretražuje izvršnu datoteku sadržan je u *PATH* varijabli. Nazivi naredbi koji sadrže kose crte se ne pretražuju, već se izravno izvršavaju. Kada se izvršna datoteka naredbe pronađe, sprema se njezin naziv i putanja u *hash* tablicu koja se pregledava prije idućih pretraživanja direktorija. Kako bi se izvršna datoteka mogla pokrenuti, stvara se novi, podređeni proces. Podređeni proces zapravo je kopija originalnog, nadređenog procesa s manjim izmjenama poput datoteka koje se otvaraju i zatvaraju preusmjeravanjem. Brojne naredbe su dio BASH ljuske, a nazivaju se ugrađenim (engl. *built-in*) naredbama i izvršavaju se bez stvaranja novog procesa. Najčešći razlog implementacije ugrađenih naredbi je održavanje ili izmjena unutrašnjeg stanja ljuske. „*cd*“ je primjer ugrađene naredbe koja služi za pozicioniranje u određeni direktorij [2]. Ako bi *cd* bila izvršna datoteka, ona bi izvršavanjem (u podređenom procesu) promijenila samo svoj trenutni direktorij, a direktorij same ljuske odnosno nadređenog procesa ostao bi nepromijenjen. Dakle, proces ljuske treba izvršiti *cd* naredbu tako da se ažurira njegov trenutni direktorij. Nakon toga, kada ljuska pokrene podređene procese, oni će naslijediti taj direktorij.

2.5. Preusmjeravanje i ulančavanje

Preusmjeravanje je jedna od značajki u Linux operacijskim sustavima koja prilikom izvršavanja naredbi omogućava promjenu izlazne ili ulazne putanje. BASH podržava sljedeće operatore preusmjeravanja [8]:

- „<“ operator za preusmjeravanje ulaza. Naredbi je, primjerice, moguće kao ulaz predati datoteku (slika 2.8.).

```
brodo@debian:~$ wc < r.txt
1 1 5
```

Slika 2.8. Primjer korištenja „<“ operatora

- „<<“ operator koji je također poznat pod nazivom ovdje-dokument (engl. *here-document*). Ovaj operator upućuje ljusku da čita unos iz određenog izvora sve dok ne naiđe na redak koji sadrži specifičnu riječ. Također, on omogućava višeredni unos. (slika 2.9.).

```
brodo@debian:~$ wc << EOF
> jedan
> dva
> EOF
      2       2      10
```

Slika 2.9. Primjer korištenja „<<“ operatora

- „<<-“ operator koji iz svakog retka uklanja sve početne tabultore kod višerednog unosa (slika 2.10.)

```
brodo@debian:~$ cat redirection.sh
#!/bin/bash

more <<- EOF
        prva linija
        druga linija
EOF
brodo@debian:~$ bash redirection.sh
prva linija
druga linija
```

Slika 2.10. Primjer korištenja „<<-“ operatora

- „>“ operator preusmjeravanja izlaza. Izlaz se zapisuje u datoteku pri čemu se postojeći sadržaj datoteke briše (slika 2.11.).

```
brodo@debian:~$ echo "Hello" > r.txt
brodo@debian:~$ cat r.txt
Hello
```

Slika 2.11. Primjer korištenja „>“ operatora

- „>>“ operator preusmjeravanja izlaza. Izlaz se dodaje na kraj datoteke (slika 2.12.)

```
brodo@debian:~$ echo "Prva linija" >> r.txt
brodo@debian:~$ echo "Druga linija" >> r.txt
brodo@debian:~$ cat r.txt

Prva linija
Druga linija
```

Slika 2.12. Primjer korištenja „>>“ operatora

- „ | “ operator ulančavanja. Ulančavanjem se više naredbi povezuje tako da izlaz jedne naredbe postaje ulaz u iduću naredbu (slika 2.13).

```
brodo@debian:~$ cat r.txt | grep "Prva"
Prva linija
```

Slika 2.13. Primjer korištenja „ | “ operatora

- „&>“ operator preusmjeravanja izlaza i izlaza za greške. Oba izlaza se zapisuju u datoteku pri čemu se postojeći sadržaj datoteke briše (slika 2.14.).

```
brodo@debian:~$ mls -la &> r.txt
brodo@debian:~$ cat r.txt
bash: mls: command not found
```

Slika 2.14. Primjer korištenja „ &> “ operatora

- „&>>“ operator preusmjeravanja izlaza i izlaza za greške. Oba izlaza se dodaju na kraj datoteke (slika 2.15).

```
brodo@debian:~$ mls -la &>> r.txt
brodo@debian:~$ datew &>> r.txt
brodo@debian:~$ cat r.txt

bash: mls: command not found
bash: datew: command not found
```

Slika 2.15. Primjer korištenja „ &>>“ operatora

- „<>“ operator koji otvara datoteku za čitanje i pisanje (slika 2.16.).

```
brodo@debian:~$ cat <> hello.txt
Hello World
```

Slika 2.16. Primjer korištenja „ <>“ operatora

Na slici 2.17. prikazan je primjer korištenja više operatora preusmjeravanja unutar jedne naredbene linije. Naredba „*grep*“ kao ulaz prima sadržaj datoteke „*r.txt*“ te ispisuje retke koji sadrže riječ „*Hello*“. Taj ispis se prosljeđuje na ulaz naredbe „*wc*“ koja svoj ispis zapisuje u datoteku „*output.txt*“.

```
brodo@debian:~$ grep "Hello" < r.txt | wc &> output.txt
brodo@debian:~$ cat output.txt
      1       2      12
```

Slika 2.17. Korištenje više operatora preusmjeravanja

2.6. Kontrola posla

Ljuska uobičajeno izvršava naredbe u prednjem planu (engl. *foreground*) gdje čeka da naredba završi nakon čega prikuplja njezin izlazni status. Tijekom izvršavanja naredbe u prednjem planu nije moguće upisivati nove naredbe. No, ako se naredba pokrene u pozadini tada ljuska može odmah čitati novi unos i pokrenuti drugu naredbu. Ispis naredbe koja se izvršava u pozadini, kao i naredbe koja se izvršava u prednjem planu, moguće je vidjeti na standardnom izlazu. Kontrola posla (engl. *job control*) odnosi se na sposobnost ljuske da upravlja procesima (naredbama koje se izvršavaju) njihovim pomicanjem između prednjeg i pozadinskog plana, te zaustavljanjem i nastavkom njihovog izvršenja. Da bi se postigla opisana funkcionalnost, BASH uvodi koncept posla, koji predstavlja naredbu odnosno naredbe koje izvršava više procesa. Obično se posao sastoji od samo jedne naredbe ili procesa, no u slučaju korištenja cjevovoda (operatora ulančavanja) postoji više procesa koji su povezani s poslom. Grupa procesa koristi se za grupiranje procesa zajedno u jedan posao. Terminal ima pridružen ID grupe procesa koja se izvršava (u prednjem planu) te je svaki posao jedinstveno identificiran ID-om grupe procesa. Proces koji ima isti ID kao i ID grupe procesa se naziva voditeljem grupe procesa. Implementacija kontrole posla BASH-a sastoji se od nekoliko jednostavnih struktura podataka. Jedna takva struktura predstavlja podređeni proces (engl. *child process*), koji sadrži ID procesa, stanje i status koji je vratio nakon završetka. Povezani popis struktura podređenih procesa čini cjevovod. Slično tome, posao se sastoji od popisa procesa (naredbi), zajedno s indikatorima stanja posla (npr. izvršava se, zaustavljeno, završeno) i ID-om grupe procesa posla. Stanje i izlazni status posla određuju se na temelju izlaznih statusa njegovih podređenih procesa [2].

Ako se, primjerice, pokrene naredba *„ping -i 5 google.com“* kojom se provjerava dostupnost destinacije svakih 5 sekundi, izvršavanje je moguće zaustaviti kombinacijom tipki CTRL + Z. Tada ljuska šalje *SIGTSTP* signal terminalu koji mu naznačuje da zaustavi trenutno izvršavanje te

ispisuje „*[1]+ Stopped ping -i 5 google.com*“. Nakon toga je terminal ponovno dostupan za upisivanje naredbe. Kako bi ova zaustavljena naredba nastavila s izvršavanjem potrebno je izvršiti naredbu kojom se proces stavlja u prednji plan: „*fg“* ili „*fg %id“* čime se specificira ID posla koji će nastaviti s izvršavanjem. Ako se izvršavanje ovog posla želi prebaciti u pozadinu tada se proces mora zaustaviti pomoću tipki CTRL + Z nakon čega je potrebno upisati naredbu „*bg*“ ili „*bg %id*“. Kako bi se omogućilo zaustavljanje procesa koji se izvršava u pozadini prvo ga je potrebno prebaciti u prednji plan. Proces je moguće postaviti da se odmah izvršava u pozadini unošenjem znaka „&“ na kraj naredbene linije. Izvršavanje naredbe odnosno posla je moguće završiti kombinacijom tipki CTRL+C ili pomoću BASH ugrađene naredbe „*kill %id*“ [9]. Naredbi „*kill*“ moguće je predati argument koji specificira vrstu signala koji se želi poslati određenoj naredbi odnosno procesu. Najkorištenije vrste „*kill“* signala koji prekidaju izvršavanje su: *SIGINT*, *SIGTERM* i *SIGKILL*. *SIGINT* (engl. *signal interrupt*) signal se šalje kombinacijom tipki CTRL+C te se može smatrati zahtjevom za prekid izvršavanja naredbe koji šalje korisnik. Ovim signalom se rukuje (engl. *handle*) ovisno o procesu i situaciji. Ukoliko se ne specificira vrsta signala prilikom korištenja naredbe „*kill*“, šalje se *SIGTERM* signal. Ovaj signal može biti blokiran, obrađen ili ignoriran za razliku od *SIGKILL* signala [10]. *SIGKILL* uzorkuje trenutačni prekid izvršavanja te se njime ne može rukovati i nije ga moguće ignorirati niti blokirati. Korištenje *SIGKILL* može uzorkovati stvaranje takozvanih „*zombie“* procesa jer „ubijeni“ proces nema priliku poslati informaciju svom nadređenom procesu da je primio *„kill“* signal. *Zombie* procesi se ne izvršavaju, nisu memorijski alocirani no sačuvana je njihova ID vrijednost u tablici procesa [11]. Osim ugrađenih „*fg“*, „*bg“* i „*kill“* BASH naredbi za kontrolu posla, implementirane su još i „*wait“, „disown“*„*suspend“,* i „*jobs“* naredba koja ispisuje listu svih aktivnih poslova.

2.7. Pregled ostalih ljuski za Linux operacijski sustav

Objavljena 1977. godine, Bourne ljuska, poznata i kao SH ljuska, najznačajnija je za razvoj današnjih ljuski [12]. Ona je pružala razne mogućnosti: korištenje varijabli, cjevovoda, uvjeta, petlji, te je omogućavala pisanje skripti, no bez mogućnosti definiranja funkcija. Godinu dana poslije, 1978., razvijena je C ljuska, CSH. Stvaranje skriptnog jezika sličnog programskom jeziku C bio je jedan od ključnih ciljeva dizajna ove ljuske. U njoj su implementirane mogućnosti ispisa korištenih naredbi te ponovnog odabiranja prethodnih naredbi. Još neke od značajki C ljuske su upravljanje poslovima, dovršavanje naredbi i mogućnost definiranja aliasa odnosno skraćenica za naredbe [13]. Kako bi se ispravile greške i poboljšale značajke C ljuske, 5 godina poslije razvijena je Tenex C ljuska, TCSH [14]. Iste godine je predstavljena Korn ljuska, KSH [15] koja je gotovo

u potpunosti kompatibilna s Bourneovom ljuskom što znači da je korisnici Bourneove ljuske mogu odmah koristiti. Uz to, Korn ljuska preuzima najbolje značajke C ljuske i dodaje nekoliko vlastitih. Ona omogućuje uređivanje naredbenog retka što u prethodnim ljuskama nije bilo moguće te radi efikasnije od bilo koje prethodne ljuske. Značajke programiranja u Korn ljusci značajno su proširene što uključuje definiciju funkcija, naprednu I/O kontrolu, ugrađene regularne izraze, cjelobrojnu aritmetiku i još mnogo toga. Nakon Korn ljuske razvijena je Bourne-Again ljuska, BASH, čije su značajke opisane u prethodnim potpoglavljima. BASH je također nadogradnja Bourne ljuske pa se većina skripti Bourne ljuske može izvršiti u BASH-u. Ona je zadana (engl. *default*) ljuska u većini *Linux* distribucija te se desetljećima nadograđuje. Z ljuska, ZSH [16], je također temeljena na Bourne ljusci. Ona je zadana ljuska u *macOS* sustavima i uključuje značajke C i Tenex C ljuske te uvodi nove značajke poput provjere pravopisa, mogućnosti praćenja prijava i odjava korisnika. Osim toga, Z ljuska uvodi podršku za znanstvenu notaciju, aritmetiku s pomičnim zarezom, globalne aliase za nazive datoteka te druge značajke. Z ljuska je poput BASH ljuske, ali s više značajki i konfiguracijskih opcija [17]. U radu [18] prikazana je usporedba navedenih ljusaka prema značajkama (tablica 2.1). Značajke su ocijenjene prema njihovim performansama. „+++“ označava najbolju dok „-“ označava najlošiju ocjenu. „- -“ označava da značajka nije prisutna u ljusci. Najbolje ocijenjena je Z ljuska nakon koje slijede Tenex C i BASH ljuske. Tenex C ljuska je po nekim kriterijima bolja od BASH-a, no treba imati na umu da je BASH jako dobro dokumentirana zbog čega je danas jedna od najkorištenijih ljuski. Navedene ljuske su namijenjene za općenitu korisničku upotrebu.

Tablica 2.1. Usporedba ljuski prema značajkama [18]

Kriterij	SH	KSH	BASH	ZSH	CSH	TCSH
Mogućnost konfiguracije ljuske	-	+	++	+++	+	++
Izvršavanje naredbi	+	+	+	++	+	++
Dovršavanje (naziva) naredbi	- -	+	++	+++	+	++
Uređivanje naredbenog retka	-	+	++	++	-	++
Proširenje naziva datoteke	+	+	++	++	+	++
Povijest	- -	+	++	++	+	++
Preusmjeravanja i cjevovodi	+	+	+	++	+	+
Provjera pravopisa	- -	- -	- -	+	- -	+
Postavke znaka za unos naredbe	+	+	+	++	+	++
Upravljanje poslovima	- -	+	+	+	+	+
Kontroliranje izvršavanja	+	+	+	+	+	+
Rukovanje signalima	+	+	+	+	-	-

Ljuska predstavljena u radu [19] temeljena je na funkcijskom programiranju, FP. U funkcijskom programiranju program predstavlja funkciju. Značajke FP jezika su funkcionalni oblici odnosno konstrukcije koje služe za kombiniranje programa u nove programe. Ova ljuska uvodi funkcionalne oblike te interpretira standardne Unix naredbe kao FP primitivne funkcije i standardne Unix datoteke kao FP objekte. Ovu ljusku moguće je koristiti za učenje FP jezika te upotrebu FP jezika za izvršavanje naredbi, a prednost u odnosu na regularne ljuske (navedene u tablici 2.1.) pronalazi u primjenama gdje je učinkovitost računanja od primarne važnosti. Još jedna ljuska namijenjena za specifičnu primjenu predstavljena je u radu [20]. Riječ je o R ljusci, čija je svrha podrška u istraživanju i razvoju softvera u području robotike. R ljuska, osim regularnih Unix naredbi i ostalih izvršnih datoteka, može izvršavati vlastiti skup ugrađenih naredbi koje pružaju učinkovit alat za stvaranje podataka za testiranje i evaluaciju programa u robotici.

3. IZRADA LJUSKE ZA LINUX OPERACIJSKI SUSTAV

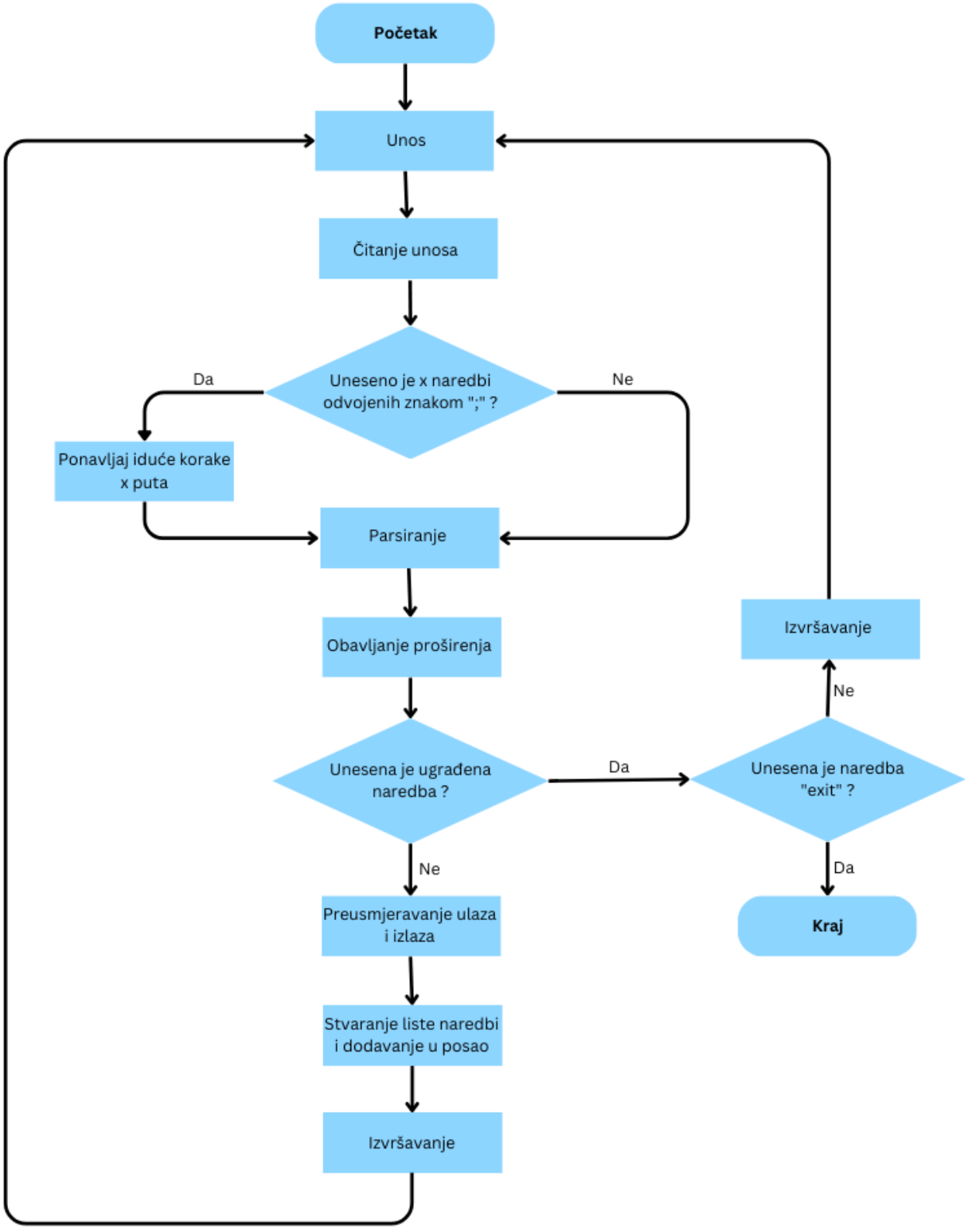
U ovom poglavlju opisan je praktični dio rada u kojem je razvijena jednostavna ljuska čija će se izrada i značajke opisati u navedenim potpoglavljima. Ljuska ima mogućnost izvršavanja jednostavnih i složenih naredbi, ulančavanja naredbi operatorom „ | “ te preusmjeravanja ulaza i izlaza koristeći operatore „< “, „>“, „>> “, „&>“. Implementirano je proširenje naziva datoteke i proširenje varijabli. Ljuska sadrži funkcionalnosti kontrole posla kao što su prebacivanje izvršavanja naredbi između prednjeg i pozadinskog plana, zaustavljanje i prekidanje izvršavanja naredbi te nastavljanje izvršavanja zaustavljenih naredbi. Više redaka unosa moguće je unijeti u jednom retku odvajajući ih znakom „ ; “.

3.1. Opis radnog okruženja

Ljuska je razvijena na *Debian* operacijskom sustavu. *Debian*, poznat i pod nazivom *Debian GNU/Linux*, je *Linux* distribucija što znači da sadrži *Linux* kernel. Korišteno je *Eclipse* razvojno okruženje te programski jezik C. Kod pronalaženja i otklanjanja pogrešaka prilikom razvoja ljuske korišten je GDB (engl. *The GNU Debugger*) alat.

3.2. Način rada ljuske

Na slici 3.1. prikazan je dijagram toka izvođenja ljuske. Ljuska poslije pokretanja čeka na korisnikov unos. Nakon toga čita unos te ako je uneseno više naredbi koje su odvojene znakom „ ; “ tada se one odvijaju jedna za drugom. Pri tome, ovisno o tome koliko je naredbi uneseno, toliko puta se ponavljaju svi idući koraci. Pročitani unos šalje se u dio ljuske zadužen za parsiranje. Nad izlazom iz parsera, ukoliko je potrebno, odvijaju se implementirana proširenja ljuske. Ako je upisana naredba dio ugrađenih naredbi ljuske tada se ona odmah izvršava te se opet čeka na novi unos, a inače se radi o naredbi čija se izvršna datoteka pronalazi u direktorijima navedenim u *PATH* varijabli. Unos može sadržavati jednu naredbu ili više njih koje su povezane cjevovodima te se u oba slučaja stvara lista naredbi koja se dodaje u posao. Izvršavanje naredbe odvija se u zasebnom procesu, no prije toga, ako je potrebno, odvijaju se preusmjeravanja izlaza i ulaza naredbe. Nakon što se naredba izvrši ljuska opet čeka novi unos. Ljuska se izvršava u beskonačnoj petlji sve dok se ne unese naredba *exit* za izlaz.



Slika 3.1. Dijagram toka izvođenja ljuske

3.3. Pokretanje ljuske i čitanje unosa

Prije pokretanja ljuske potrebno ju je prevesti (engl. *compile*) kako bi se stvorila izvršna datoteka. Pokretanjem izvršne datoteke ispisuje se znak za unos naredbe (engl. *prompt*) (slika 3.2.). Kompletan programski kod ljuske dan je u prilogu P.3.1.

```
brodo@debian:~/eclipse-workspace/projekt2$ gcc pipred.c commands.c jobs.c -lreadline -o shell
brodo@debian:~/eclipse-workspace/projekt2$ ./shell
SHELL$
```

Slika 3.2. Prevođenje i pokretanje ljuske

Čitanje unosa predstavlja prvi korak kod izvršavanja naredbi. Ljuska čita unos s *readline()* funkcijom koja je dio <*readline/readline.h*> biblioteke [3]. Funkcija *readline()* kao parametar prima niz znakova koji predstavljaju znak za unos naredbe, u ovom slučaju je to „*SHELL$*“. Funkcija čita liniju unosa te vraća pokazivač na pročitani niz znakova. Također treba napomenuti kako funkcija sa kraja niza briše znak za novi red , „*\n*“. Ova funkcija za čitanje linije izabrana je zato što korisniku nudi mogućnost uređivanja naredbene linije, odnosno korisnik se može pomicati po naredbenoj liniji i brisati ili dodavati druge znakove.

3.4. Parsiranje

Nad pročitanom linijom odvija se proces parsiranja, no prije toga provjerava se sadrži li linija znak „ ;“. Ako unos sadrži navedeni znak, izvršava se funkcija koja prolazi kroz cijelu liniju znak po znak i sprema nizove znakova koji se nalaze prije znaka „ ;“ čime se stvara polje pokazivača na nizove znakova gdje svaki niz znakova predstavlja jednu liniju koja prolazi kroz sve faze obrade unosa i izvršavanja naredbe. Proces parsiranja započinje leksičkom analizom kojom se unos dijeli na tokene odnosno nizove znakova koji su odvojeni određenim znakovima. Skup tih posebnih znakova, metaznakova, čine: „< “, „>“, „ | “, „&“. Na slici 3.3. prikazan je primjer unosa i izlaz leksičke analize gdje svaki redak predstavlja jedan token.

```
brodo@debian:~/eclipse-workspace/projekt2$ ./shell
SHELL$ cat < file1 | head -6 |tail -3 >file2
cat
<
file1
|
head -6
|
tail -3
>
file2
SHELL$
```

Slika 3.3. Unos i izlaz leksičke analize

Kako bi se dobio prikazani izlaz razvijene su tri glavne funkcije. Prva funkcija *char*** *makeList(char** line,int numofAll)* prima adresu pokazivača na pročitanu liniju te cjelobrojnu vrijednost *numofAll* koja predstavlja ukupan broj tokena (slika 3.4.). Funkcija *makeList()* vraća pokazivač na polje tokena. Token zapravo predstavlja polje pokazivača na nizove znakova. Token može biti metaznak (npr. „<“), naziv naredbe zajedno sa svojim argumentima (npr. „*head -6*“) ili naziv datoteke (npr . „*file1*“) koja se koristi kod preusmjeravanja. Unutar *makeList()* funkcije izvršava se petlja *numofAll* puta pri čemu se u svakoj iteraciji poziva funkcija *char** *addCmd(char** line)* koja vraća token.

```
char*** makeList(char** line,int numofAll){
    char*** polje=(char***)malloc((numofAll+1)*sizeof(char**));
    int i=0;
    for(i=0;i<numofAll;i++){
        polje[i]=addCmd(line);
    }
    polje[numofAll]=NULL;
    return polje;
}
```

Slika 3.4. Funkcija koja vraća pokazivač na polje tokena

Kompletan programski kod funkcije *addCmd()* dan je u prilogu P.3.2. Ona prolazi kroz pročitanu liniju te kada naiđe na metaznak, sprema ga. U slučaju da funkcija *addCmd()* naiđe na naredbu ili datoteku, poziva se funkcija *char* addArg(char** p)* koja vraća pokazivač na niz znakova koji može predstavljati naziv naredbe (npr. „*head*“) ,argument naredbe (npr. „*-6*“) ili naziv datoteke (prilog P.3.3.). Na ovaj način pročitana linija rastavljena je na tokene. Preko pokazivača koji vraća funkcija *makeList()* može se pristupiti svakom tokenu, odnosno svakom metaznaku, naredbi ili datoteci. Nakon što je unos rastavljen na tokene izvršavaju se proširenja opisana u poglavlju 3.9. Zatim, ako se tokeni podudaraju s nekom od ugrađenih naredbi, ta se naredba odmah izvršava i ljuska opet čeka korisnikov unos.

Nadalje, konstruirana je struktura koja predstavlja jednostavnu naredbu (slika 3.5.). Ona se točno odnosi na *Linux* naredbe, ugrađene naredbe nisu predstavljene ovom strukturom. Članovi strukture su: naziv naredbe zajedno s argumentima, naziv datoteke iz koje se preusmjerava ulaz, naziv datoteke u koju se preusmjerava izlaz, vrijednost koja označava način na koji se preusmjerava izlaz, pokazivač na iduću naredbu, ID procesa te status koji označava trenutno stanje naredbe (izvršavanje, zaustavljeno, završeno, prekinuto). U idućem koraku parsiranja stvara se povezana lista naredbi pomoću funkcije *void appendCmd(job** job)* (prilog P.3.4.). U listu se dodaju naredbe koje su povezane cjevovodima.

```
typedef struct simpleCmd{
   char **name;
   char *in;
   int modeOut;
   char *out;
   struct simpleCmd* next;
   pid_t pid;
   int status;
}cmd;
```

Slika 3.5. Struktura koja predstavlja naredbu

```
typedef struct job
{
  int id;
  struct job *next;
  int mode;
  struct simpleCmd* root;
  char*** commands;
  pid_t pgid;
  int numSimpleCmds;
  int notified;
  char *line;

} job;
```

Slika 3.6. Struktura posla

Osim toga, funkcija *appendCmd()* prima strukturu posla koja sadrži pokazivač na prvi element liste naredbi te se na taj način lista naredbi dodaje u posao. Svaki posao dodaje se u listu poslova pomoću funkcije *job* appendJob(job** head,int back)*(prilog P.3.5.). Struktura posla prikazana je na slici 3.6. Nju čine ID posla, pokazivač na idući posao, oznaka radi li se o izvršavanju u prednjem planu ili pozadini, pokazivač na listu naredbi odnosno na naredbe povezane cjevovodima, pokazivač na polje tokena, ID grupe procesa, broj naredbi u retku, oznaka je li korisnik obaviješten o stanju posla i pokazivač na unesenu liniju. Nakon što je napravljena lista naredbi potrebno je dodijeliti vrijednosti elementima struktura. Ove vrijednosti se dodjeljuju tako da se prolazi kroz polje tokena nastalih leksičkom analizom. Prema određenim uvjetima, za svaku strukturu naredbe postavlja se naziv naredbe, nazivi datoteka vezani uz preusmjeravanje i način preusmjeravanja izlaza (slika 3.7.). Ovime je cjelokupan proces parsiranja završen.

```
if(cmds[j+1]!=NULL && (!strcmp(cmds[j+1][0],"<")) && cmds[j+2]!=NULL){
     cmd->in=cmds[j+2][0];
     if(cmds[j+3]!=NULL &&(!strcmp(cmds[j+3][0],">")) &&cmds[j+4]!=NULL){
          cmd->out=cmds[j+4][0];
          cmd->modeOut=1;
          j=j+5;
          cmd=cmd->next;
          continue;
     }
```

Slika 3.7. Primjer jednog od uvjeta prema kojima se postavljaju vrijednosti strukture naredbe

3.5. Preusmjeravanje i ulančavanje

Nadovezujući se na prethodno poglavlje i sliku 3.7. dopuštene su sljedeće sintakse korisnikova unosa:

- *naredba1 < datoteka1*

- *naredba1 > datoteka1*
- *naredba1 < datoteka1 > datoteka2*
- *naredba1 | naredba2 |...| naredbaN*
- *naredba1 < datoteka1 | naredba2 |...| naredbaN*
- *naredba1 | naredba2 |...| naredbaN > datoteka1*
- *naredba1 < datoteka1 | naredba2 |...| naredbaN > datoteka2*

Pri tome se za preusmjeravanje ulaza koristi „<“ operator dok je za preusmjeravanje izlaza moguće koristiti jedan od sljedećih operatora: „>“, „>>“, „&>“. Osim toga, implementiran je operator ulančavanja, „ | “. Navedeni operatori imaju istu funkciju kao u BASH ljusci što je opisano u poglavlju 2.5. Primjer funkcije za preusmjeravanje izlaza pomoću „>>“ operatora dan je na slici 3.8.

```
void redirectAppend(char *fileName){
    int out=open(fileName, O_WRONLY | O_APPEND);
    if(out==-1){
            fprintf(stderr,"%s does not exists\n",fileName);
            exit(EXIT_FAILURE);
        }
    dup2(out,1);
    close(out);
}
```

Slika 3.8. Funkcija koja obavlja preusmjeravanje izlaza s „>>“ operatorom

Pomoću sistemskog poziva *int open (const char* Path, int flags)* pokušava se otvoriti datoteka za pisanje gdje se sadržaj dodaje na kraj datoteke (slika 3.8.). Ukoliko datoteka postoji, vraća se pozitivan cijeli broj koji predstavlja deskriptor datoteke. Deskriptor datoteke je identifikator otvorene datoteke ili ulazno/izlaznih (I/O) resursa koje pruža operacijski sustav. Preko njega je moguće pristupiti datoteci u svrhu zapisivanja ili čitanja sadržaja. Tri standardna deskriptora datoteka koji se otvaraju za svaki proces su: standardni ulaz čija je vrijednost deskriptora 0, standardni izlaz (1) te standardni izlaz za pogreške (2). Deskriptori za ostale datoteke dodjeljuju se od broja 3 i nadalje. Nakon što je datoteka otvorena, stvarno preusmjeravanje odvija se koristeći *int dup2(int oldfd, int newfd)* sistemski poziv. On omogućuje da se dva različita deskriptora odnose na istu datoteku ili resurs. Deskriptor *oldfd* kopira se na mjesto *newfd*. Ako se *newfd* odnosi na već otvorenu datoteku (u ovom slučaju je to standardni izlaz), ona se prvo zatvara te se onda uspostavlja veza s *oldfd*. Stoga, ispis naredbe koji je trebao završiti na standardnom izlazu preusmjerava se u datoteku identificiranu deskriptorom *oldfd*. Na kraju izvršava se sistemski poziv *int close(int fd)* kojim se zatvara datoteka specificirana deskriptorom *fd*. Prva naredba iz kreirane

liste naredbi opisane u prošlom potpoglavlju može čitati ulaz sa standardnog ulaza ili datoteke, dok posljednja naredba svoj izlaz može poslati na standardni izlaz ili u datoteku.

Kompletan programski kod funkcije *exeCmds()* u kojoj su implementirane funkcionalnosti preusmjeravanja i ulančavanja nalazi se u prilogu P.3.6. Kao što je već opisano, cjevovod predstavlja operator ulančavanja „ | “, a njime se izlaz jedne naredbe prosljeđuje na ulaz iduće naredbe. On se implementira pomoću *int pipe(int fd[2])* sistemskog poziva (slika 3.9.). Funkciji *pipe()* predaje se cjelobrojno polje veličine dva elementa. Ona stvara dva deskriptora datoteke i zapisuje ih u predano polje. Pri tome je *fd[0]* deskriptor koji predstavlja kraj veze za čitanje, dok *fd[1]* predstavlja kraj veze za pisanje. Broj poziva funkcije *pipe()* jednak je broju naredbi u listi.

```
for(i=0; i<numSimpleCmds; i++) {
    if(pipe(fd+i*2)<0) {
      perror("pipe");
      exit(EXIT_FAILURE);
    }
  }
```

Slika 3.9. Isječak funkcije *exeCmds()* kojim se stvaraju cjevovodi

Izlaz naredbe preusmjerava se na deskriptor *fd[1]* pomoću *dup2()* sistemskog poziva. Taj izlaz je moguće pročitati na deskriptoru *fd[0]*. Iduća naredba, također pomoću funkcije *dup2()*, preusmjerava svoj ulaz na deskriptor *fd[0]* od prošle naredbe jer on predstavlja izlaz koji je prošla naredba proizvela (slika 3.10.).

```
if(dup2(fd[j+1],1)<0) {
     perror("dup2");
     exit(EXIT_FAILURE);
}
```

(a)

```
if(dup2(fd[j-2],0)<0) {
     perror("dup2");
     exit(EXIT_FAILURE);
}
```

(b)

Slika 3.10. Isječak funkcije *exeCmds()* kojim je implementirano ulančavanje:
(a) Preusmjeravanje izlaza na kraj veze za pisanje **(b)** Preusmjeravanje ulaza na kraj veze za čitanje prošle naredbe

Primjeri korištenja operatora preusmjeravanja i ulančavanja prikazani su na slici 3.11.

```
debian@debian:~/starizImageizSRV/brshell$ ./shell
SHELL$ cat< file1
1
2
SHELL$ cat < file1| head -2 | tail -1>> file2
SHELL$ cat<file2
1
2
2
```

Slika 3.11. Korištenje operatora preusmjeravanja i ulančavanja

3.6. Kontrola posla i izvršavanje naredbi

U ovom potpoglavlju opisana je kontrola posla zajedno s izvršavanjem naredbi budući da je jedno usko povezano s drugim. Odmah nakon što je pokrenuta, ljuska mora osigurati da ju vanjska, nadređena ljuska postavi u prednji plan kako bi se omogućila kontrola posla (slika 3.12.).

```
mainpgrp=getpid();
setpgid(mainpgrp,mainpgrp);
tcsetpgrp(0,mainpgrp);
signal (SIGINT, SIG_IGN);
signal (SIGQUIT, SIG_IGN);
signal (SIGTSTP, SIG_IGN);
signal (SIGTTIN, SIG_IGN);
signal (SIGTTOU, SIG_IGN);
```

Slika 3.12. Postavljanje ljuske u prednji plan

Prvo se uzima ID procesa ljuske funkcijom *getpid()* te se on postavlja u vlastitu grupu procesa pomoću funkcije *setpgid()*. Nakon toga izvršavanjem funkcije *tcsetpgrp()* grupa procesa ljuske ima kontrolu nad terminalom te je stavljena u prednji plan. Kako ljuska nebi slučajno zaustavila samu sebe, signali se „ignoriraju“ pomoću funkcije *signal()* kojoj se osim signala, kao drugi parametar predaje *SIG_IGN* vrijednost. „Ignorirani“ signali u podređenim procesima postavljaju se na zadane vrijednosti tako da se funkciji *signal()* preda *SIG_DFL* vrijednost. Stoga kada korisnik, primjerice, pošalje *SIGINT* signal putem tipki CTRL+C prekinut će izvršavanje podređenog procesa, a ne same ljuske.

Nakon parsiranja kreirana lista naredbi povezana je s poslom te poziva se funkcija *void exeCmds(int numSimpleCmds,job* job)* (prilog P.3.6.) u kojoj se odvija proces ulančavanja, preusmjeravanja i izvršavanja naredbi. Dio ove funkcije koji se bavi preusmjeravanjem i ulančavanjem objašnjen je u prethodnom poglavlju. Parametar *int numSimpleCmds* označava broj naredbi u listi naredbi, a parametar *job* job* označava pokazivač na posao koji će se izvršiti. Izvršavanje naredbe odvija se s *int execvp(const char *file, char *const argv[])* sistemskim pozivom (slika 3.13.). Ovaj sistemski poziv može pokrenuti bilo koju programsku datoteku, što uključuje izvršne datoteke i skripte ljuske.

```
if(execvp(cmd->name[0],cmd->name)<0){
        perror(cmd->name[0]);
        exit(EXIT_FAILURE);
}
```

Slika 3.13. Isječak funkcije *exeCmds()* kojim se izvršavaju naredbe

Kao parametre *execvp()* prima naziv naredbe i polje koje sadrži argumente naredbe. Naredbu, odnosno izvršnu datoteku ova funkcija pronalazi u direktorijima *PATH* varijable. Pokretanjem *execvp()* funkcije zamjenjuje se trenutna slika procesa s novom slikom procesa. Zbog toga, potrebno je izvršavanje naredbe provesti u zasebnom procesu. Za svaku naredbu stvara se poseban, podređen proces pomoću sistemskog poziva *pid_t fork(void)*. Kreirani proces je kopija nadređenog procesa i izvršavaju se u isto vrijeme. Sistemski poziv *fork()* vraća 0 u podređen (kreirani) proces, a ID kreiranog procesa vraća u nadređen proces. Ako je kreiran proces za prvu naredbu u listi tada se njegov ID postavlja kao ID grupe procesa odnosno posla. Pomoću funkcije *setpgid()* se ID svakog sljedeće kreiranog procesa dodaje u grupu procesa. Grupa procesa omogućava da se, primjerice, određeni signal može odjednom poslati svim procesima unutar grupe.

```
for(cmd=job->root;cmd!=NULL;cmd=cmd->next){
      if((pid = fork())==-1){
          perror("fork");
          exit(EXIT_FAILURE);
}
```

Slika 3.14. Isječak funkcije *exeCmds()* kojim se stvaraju podređeni procesi

Ako se posao izvršava u prednjem planu, nadređen proces može nastaviti s izvršavanjem tek kada posao promijeni svoje stanje. Zbog toga nadređen proces mora čekati da se stanja podređenih procesa posla promijene. Promjena stanja uključuje završetak, zaustavljanje ili nastavak izvršavanja procesa. Sistemski poziv *pid_t waitpid(pid_t pid, int *status, int options)* zaustavlja izvršavanje procesa u kojem je pokrenut, sve dok podređeni proces specificiran parametrom *pid* ne promijeni svoje stanje. Tada *waitpid()* vraća identifikator podređenog procesa i informaciju o promjeni stanja sprema u **status* varijablu. Identifikator procesa, *pid*, koji se predaje funkciji može imati različita značenja. Primjerice, u ovom slučaju koristi se negativna vrijednost *pid* parametra grupe procesa (slika 3.15.). U tom slučaju će funkcija *waitpid()* vratiti vrijednost kada bilo koji od podređenih procesa promijeni stanje. No, potrebno je čekati da svaki od podređenih procesa to učini. Zbog toga se ova funkcija izvršava u petlji sve dok se ne dobije povratna vrijednost za svaki proces. Po zadanome, *waitpid()* vraća vrijednost samo u slučaju završetka izvršavanja procesa. No, ako se za parametar *options* preda zastavica *WUNTRACED* ili *WCONTINUED* tada *waitpid()* vraća vrijednost i u slučaju zaustavljanja ili nastavka izvršavanja procesa. Vrijednost spremljena u *status* varijablu može se interpretirati pomoću različitih makronaredbi. *WIFEXITED()* vraća ne-nultu vrijednost ukoliko se proces normalno izvršio. Pomoću *WSTOPSIG()* moguće je otkriti koji je proces zaustavljen ili u slučaju *WIFSIGNALED()* naglo prekinut. Kompletan programski kod funkcije *waitForJob()* dan je u prilogu P.3.7.

```
do{
    waitPid=waitpid(-job->pgid,&status,WUNTRACED);
    waitCount++;
    if(WIFEXITED(status)){
        setCmdStatus(job,waitPid,COMPLETED);
    }
    else if(WSTOPSIG(status)){
            setCmdStatus(job,waitPid,STOPPED);
    }
    else if(WIFSIGNALED(status)){
            setCmdStatus(job,waitPid,TERMINATED);
    }
}while(waitCount<cmdsCount);
```

Slika 3.15. Isječak funkcije *waitForJob()* u kojem se čeka da se promijeni stanje posla koji se izvršava u prvom planu

Ako je naredba pokrenuta u pozadini, nije potrebno čekati procese da promjene stanje već je korisniku potrebno odmah omogućiti upotrebu terminala. Tada se koristi *WNOHANG* zastavica koja ne zaustavlja daljnje izvršavanje nadređenog procesa. Funkcija *void updateStatus()* (slika 3.16.) poziva se na početku beskonačne petlje prije čitanja unosa i ona kontrolira stanja procesa koji završe s izvršavanjem u pozadini te procesa koji su prekinuti ugrađenom naredbom *kill()*. Na ovaj način sprječava se nastanak *zombie* procesa. Budući da se posao u pozadini ne može zaustaviti nema potrebe za korištenjem *WSTOPSIG()* makronaredbe. Funkcija *updateStatus()* također poziva *void deleteCompletedJobs()* funkciju koja briše sve prekinute i završene poslove pri čemu oslobađa zauzete resurse te ispisuje stanje posla (prilog P.3.8.). Obavijest o završenom (*COMPLETED*) poslu ispisuje se samo u slučaju kad posao završi s izvršavanjem u pozadini.

```
void updateStatus(){
    int status,waitPid;
    while((waitPid=waitpid(WAIT_ANY,&status,WNOHANG))>0){
        job* j=getJobByPID(waitPid);
        if(WIFEXITED(status)){
            setCmdStatus(j,waitPid,COMPLETED);
        }
        else if(WIFSIGNALED(status)){
            setCmdStatus(j,waitPid,TERMINATED);
        }
    }
    deleteCompletedJobs();
}
```

Slika 3.16. Funkcija *updateStatus()*

3.7. Ugrađene naredbe

Popis ugrađenih naredbi prikazan je u tablici 3.1.

Tablica 3.1. Popis ugrađenih naredbi

Sintaksa naredbe	Opis
cd <direktorij>	Pozicioniranje u direktorij naziva <direktorij>
exit	Izlaz iz ljuske
history	Ispis prethodno korištenih naredbi
fg %<id>	Izvršavanje naredbe u prednjem planu čiji je ID jednak <id>
bg %<id>	Izvršavanje naredbe u pozadini čiji je ID jednak <id>
kill %<id>	Prekidanje naredbe čiji je ID jednak <id>
jobs	Ispis aktivnih poslova (zaustavljeni poslovi i oni koji se trenutno izvršavaju)

Programski kod kojim su implementirane navedene ugrađene naredbe dan je u prilogu P.3.9. Naredba *fg %<id>* prvo poziva funkciju koja vraća pokazivač na posao čiji je ID jednak *<id>*. Nakon toga ispisuje se obavijest *„JOB %id IN FOREGROUND“* i poziva se funkcija *void put_in_fg(job* job)*. Ova funkcija se također poziva na kraju funkcije *exeCmds()* opisane u prethodnom poglavlju. Kao što je prikazano na slici 3.17. prvo se funkcijom *tcsetpgrp()* grupa procesa odnosno posao postavlja u prednji plan čime dobiva kontrolu nad terminalom i korisnik ne može unositi nove naredbe dok izvršavanje nije završeno ili prekinuto. Zatim, ako je posao zaustavljen, šalje se *SIGCONT* signal grupi procesa (poslu) čime se svaki zaustavljeni proces nastavlja izvršavati. Ako se posao tek pokreće funkcijom *exeCmds()* ili se njegovo izvršavanje prebacuje iz pozadine u prednji plan tada nema potrebe slati *SIGCONT* signal. Naposlijetku, čeka se promjena stanja posla (funkcijom *waitForJob()* opisanom u prethodnom potpoglavlju) nakon čega se u prednji plan stavlja glavni proces ljuske zbog čega korisnik opet može unositi naredbe.

```
void put_in_fg(job* job,int cont){
      tcsetpgrp(0,job->pgid);
      if(cont==1){
         job->notified=0;
         setStatusRunning(job);
         if(kill(-job->pgid,SIGCONT)<0){
               perror("kill sigcont\n");
         }
      }
      waitForJob(job);
      tcsetpgrp(0,mainpgrp);
}
```

Slika 3.17. Funkcija *put_in_fg()*

Funkcija *bg %<id>* implementirana je na gotovo jednak način kao i *fg %<id>*. Razlika je u tome što se izvršavanjem ove funkcije ispisuje „*JOB %id IN BACKGROUND*“ poslije čega se poziva funkcija *void put_in_bg(job* job)* (slika 3.18.) koja samo šalje *SIGCONT* zaustavljenom poslu. U ovom slučaju glavni proces ljuske ostaje u prednjem planu što omogućava korisniku da unosi nove naredbe dok se posao izvršava.

```
void put_in_bg(job* job,int cont){
      if(cont==1){
          job->notified=0;
          setStatusRunning(job);
          if(kill(-job->pgid,SIGCONT)<0){
               perror("kill sigcont\n");
          }
      }
}
```

Slika 3.18. Funkcija *put_in_bg()*

Naredba *kill %<id>* jednostavno šalje SIGKILL signal procesima posla čiji je ID jednak *<id>*. Funkcija *updateStatus()* će nakon korištenja *kill* naredbe ispisati obavijest o prekinutom poslu te će ga nakon toga obrisati.

Naredba *history* implementirana je pomoću *<readline/history.h>* biblioteke [3]. Funkcija *void using_history()* omogućava upotrebu ostalih funkcija iz biblioteke. Pročitani unos prosljeđuje se funkciji *void add_history (const char *string)* koja ga dodaje na kraj liste. Osim toga, omogućen je odabir prethodno upisanih naredbi pomoću tipki sa strelicama gore i dolje. Implementirana naredba *history* ispisuje listu korištenih naredbi (slika 3.19.) pomoću funkcije *HIST_ENTRY ** history_list (void)* koja je također dio navedene bibiloteke.

```
brodo@debian:~/eclipse-workspace/projekt2$ ./shell
SHELL$ pwd
/home/brodo/eclipse-workspace/projekt2
SHELL$ chmod 777 file2
SHELL$ date
Thu 27 Jul 15:13:33 CEST 2023
SHELL$ history
1: pwd
2: chmod 777 file2
3: date
4: history
```

Slika 3.19. Korištenje *history* naredbe

Na slici 3.20. prikazano je korištenje funkcija kontrole posla. Naredba *ping* pokreće se u pozadini nakon čega se prebacuje u prednji plan. Zatim zaustavlja se slanjem *SIGTSTP* signala putem tipki CTRL+Z. Nakon toga prebacuje se nazad u pozadinu, nakon čega se prekida slanjem *SIGKILL*

signala pomoću ugrađene funkcije *kill*. Na kraju se pokreće nova naredba u pozadini. Ispis naredbe *jobs* prikazuje samo naredbu koja se trenutno izvršava, jer je prekinuta naredba izbrisana.

```
brodo@debian:~/eclipse-workspace/projekt2$ ./shell
SHELL$ ping -i 5 google.com&
SHELL$ PING google.com (142.250.180.238) 56(84) bytes of data.
64 bytes from bud02s34-in-f14.1e100.net (142.250.180.238): icmp_seq=1 ttl=117 time=19.0 ms
64 bytes from bud02s34-in-f14.1e100.net (142.250.180.238): icmp_seq=2 ttl=117 time=24.4 ms

SHELL$ fg %1
JOB [1] IN FOREGROUND
64 bytes from bud02s34-in-f14.1e100.net (142.250.180.238): icmp_seq=3 ttl=117 time=18.2 ms
64 bytes from bud02s34-in-f14.1e100.net (142.250.180.238): icmp_seq=4 ttl=117 time=17.4 ms
^Z [1]STOPPED ping -i 5 google.com&
SHELL$ bg %1
JOB [1] IN BACKGROUND
SHELL$ 64 bytes from bud02s34-in-f14.1e100.net (142.250.180.238): icmp_seq=5 ttl=117 time=18.
5 ms
64 bytes from bud02s34-in-f14.1e100.net (142.250.180.238): icmp_seq=6 ttl=117 time=19.4 ms

SHELL$ kill %1
SHELL$
 [1]TERMINATED ping -i 5 google.com&
SHELL$ ping -i 5 google.com | grep PING &
SHELL$ PING google.com (142.250.180.238) 56(84) bytes of data.

SHELL$ jobs
[1]RUNNING ping -i 5 google.com | grep PING &
SHELL$
```

Slika 3.20. Kontrola posla

Naredba *cd* implementirana je pomoću *int chdir(const char *path)* funkcije koja je dio <*unistd.h*> biblioteke.

3.8. Varijable

Još jedna mogućnost ljuske je inicijalizacija varijabli prema jednostavnoj sintaksi: <*ime_varijable*> = <*vrijednost*>. Također je moguće mijenjati vrijednosti varijabli okruženja (engl. *environmental variables*) koje ljuska nasljeđuje od ljuske iz koje je pokrenuta, u ovom slučaju BASH ljuske. Moguće je dodati novu varijablu okruženja tako da se naziv varijable navede velikim slovima. Svaka varijabla definirana je strukturom prikazanoj na slici 3.21.

```
typedef struct variable{
    int env;
    char *name;
    char *value;
    struct variable* next;
}var;
```

Slika 3.21. Struktura koja predstavlja varijablu

Osim imena i vrijednosti, struktura se sastoji od pokazivača na iduću varijablu i od cjelobrojne vrijednosti koja označava radi li se o varijabli okruženja ili o korisnički inicijaliziranoj varijabli. Pomoću varijable *extern char **environ* dohvaćaju se varijable okruženja nakon čega se

funkcijom *void addVariable(char *name,char *value,int env)* dodaju u listu struktura (prilog P.3.10.). Funkcijom *int isVar(char ***cmds)* provjerava se odnosi li se unos na incijalizaciju ili ažuriranje vrijednosti varijable (slika 3.22.). Ukoliko se naziv varijable pronađe u listi tada se njezina vrijednost ažurira. Ako naziv ne postoji u listi tada se varijabla dodaje u listu. Ažuriranje vrijednosti i incijalizacija varijabli okruženja obavlja se *int setenv(const char *name, const char *value, int overwrite)* funkcijom koja je dio <*stdlib.h*> biblioteke. Kompletan programski kod funkcije *isVar()* dan je u prilogu P.3.11.

```
if(var=findVar(name)){
     val=malloc(strlen(e+1)+1);
     strcpy(val,e+1);
     var->value=val;
     if(var->env==1){
          setenv(var->name,var->value,1);
     }
 }else{
     upper=checkUpperLetters(name);
     if(upper){
          addVariable(name,val,1);
          setenv(name,val,1);
     }else{
          addVariable(name,val,0);
     }
 }
```

Slika 3.22. Isječak funkcije *isVar()* gdje se provjerava radi li se o inicijalizaciji varijable ili o ažuriranju njezine vrijednosti

3.9. Proširenja ljuske

Od proširenja ljuske implementirano je proširenje varijabli i proširenje naziva datoteke. Proširenja se izvode nakon što je unos rastavljen na tokene. Prvo se izvodi proširenje varijabli. Sintaksa ovog proširenja je *{varijabla}*, gdje se naziv varijable zamjenjuje njezinom vrijednošću. Osim s korisnički inicijaliziranim varijablama, proširenje funkcionira i sa varijablama okruženja. Treba napomenuti kako je nad jednim tokenom moguće izvršiti jedno proširenje varijable što se može vidjeti na slici 3.23. prilikom korištenja naredbe „*echo {a}{b}*“ gdje izraz „*{a}{b}*“ predstavlja jedan token budući da varijable nisu odvojene razmakom jedna od druge. Kompletan kod implementacije proširenja varijabli dan je u prilogu P.3.12.

Proširenje naziva datoteke izvodi se odmah nakon proširenja varijabli. Pomoću posebnih znakova moguće je ispisati listu naziva datoteka koje odgovaraju navedenom uzorku. Implementirano je proširenje s „?“ i „*“ znakovima koji su opisani u 2.3.7 poglavlju. Na slici 3.24. prikazano je korištenje navedenih znakova koji proširuju uzorak u listu naziva datoteka. Moguće je kombinirati

više znakova „*“ i „?“ u istom tokenu. Također, moguće je koristiti proširenje naziva datoteke zajedno s proširenjem varijable. Programski kod implementacije proširenja naziva datoteke prikazan je u prilogu P.3.13. Proširenje naziva datoteke implementirano je pomoću *<glob.h>* biblioteke [21]. Funkcija *int glob(const char *restrict pattern, int flags, int (*errfunc)(const char *epath, int eerrno), glob_t *restrict pglob)* prema određenim pravilima traži sve nazive datoteka unutar trenutnog direktorija koji odgovaraju uzorku. Lista pronađenih naziva sprema se u *glob_t* strukturi nakon čega je potrebno zamijeniti uneseni uzorak sa listom. Naposlijetku se dinamički zauzeta memorija oslobađa funkcijom *void globfree(glob_t *pglob)*.

```
brodo@debian:~/eclipse-workspace/projekt2$ ./shell
SHELL$ a=2 b=1
SHELL$ echo {a} {b}
2 1
SHELL$ cat file{b}
test
SHELL$ c=ile
SHELL$ cat f{c}1
test
SHELL$ echo {PATH}
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
SHELL$ echo {a}{b}
2{b}
SHELL$
```

Slika 3.23. Primjeri proširenja varijable

```
brodo@debian:~/eclipse-workspace/projekt2$ ./shell
SHELL$ ls
 5kUkodu      hard10k               out         shellCat
 commands.c   hcommands.h           pipred.c    shellmeasure
 Debug        hjobs.h               sh          shl
 file1        hs                    she         sh-noRl
 file1.c      jobs.c                she.c       skripta.sh
 file2       '--library=readline'   shell       test
 file2.txt    lsFile.txt            shell1000   test.c
 gg           ncurses-6.2+20201114  shell.c
SHELL$ ls *.c
commands.c  file1.c  jobs.c  pipred.c  she.c  shell.c  test.c
SHELL$ ls file?
file1  file2
SHELL$ ls s??*
she  she.c  shell  shell1000  shell.c  shellCat  shellmeasure  shl  sh-noRl  skripta.sh
SHELL$ var=fil
SHELL$ ls {var}?*
file1  file1.c  file2  file2.txt
SHELL$
```

Slika 3.24. Primjeri proširenja naziva datoteke

3.10. Implementacija ljuske na ugradbenu platformu Raspberry Pi 3

Raspberry Pi je računalo namijenjeno za opću upotrebu te se može povezati s različitom opremom poput senzora i aktuatora zbog čega se koristi u projektima koji uključuju ugradbene računalne sustave i Internet objekata. Glavni dio *Raspberry Pi* računala čini SoC (engl. *System On a Chip*). Uz SoC, na pločici su prisutni i RAM memorija, HDMI izlaz i Ethernet priključak. Pomoću GPIO

pinova (engl. *General Purpose Input Output*) korisnici mogu povezati opremu s *Raspberry Pi* računalom [22].

U svrhu implementacije razvijene ljuske na ugradbenu platformu koristi se *Raspberry Pi 3 Model B* računalo koje karakterizira četverojezgreni 64-bitni *ARM Cortex-A53* procesor, *Broadcom BCM2837* SoC i 1 GB LPDDR2 radna memorija. Za izgradnju prilagođenog *Linux* operacijskog sustava za *Raspberry Pi* računalo koristio se *Buildroot* alat. On omogućuje automatsko stvaranje, *bootloadera*, *toolchain* alata, *kernela* i *root* datotečnog sustava iz izvornog koda. Nakon što je izgradnja *Linux*-a završila, kopirale su se datoteke za učitavanje operacijskog sustava (*firmware*, *bootloader* i *kernel*) na mikro SD karticu koja je zatim stavljena u *Raspberry Pi*. Razvojno računalo povezano je s *Raspberry* računalom koristeći pretvornik USB na serijsku komunikaciju (engl. *USB-serial adapter*). Kako bi se omogućila serijska veza instaliran je *picocom* alat. Komunikacija se provjerila izvršavanjem naredbe: *sudo picocom −b 115200 /dev/ttyUSB0*. Budući da je veza uspostavljena, ispisala se poruka „*Terminal ready*“. Zatim je bilo potrebno spojiti *Raspberry Pi* na napajanje nakon čega su se u terminalu, gdje je otvorena serijska veza, ispisale poruke *U-boot bootloadera* te je prekinuto automatsko podizanje *Linux*-a. Uspostava serijske komunikacije i ispis poruka *bootloadera* prikazani su u prilogu P.3.14. Kako bi se sustav mogao pravilno pokrenuti definirana je varijabla okruženja *bootcmd* koja učitava *zImage* i DTB (engl. *Device Tree Blob*) datoteku na odgovarajuće adrese: *setenv bootcmd „fatload mmc 0:1 0x01000000 zImage; fatload mmc 0:1 0x2000000 bcm2710-rpi-3-b.dtb; bootz 0x01000000 – 0x2000000“*. *Root* datotečni sustav je na *Raspberry* računalu učitan preko NFS servera i TFTP protokola kako bi se omogućilo jednostavno kopiranje datoteka na *Raspberry* sa razvojnog računala. Stoga je instaliran i pokrenut TFTP server i NFS server na razvojnom računalu. Kako bi se omogućio prijenos datoteka TFTP protokolom, *Raspberry* i razvojno računalo povezani su mrežnim kabelom koristeći USB na Ethernet pretvornik.

Na razvojnom računalu pojavila se nova mrežna veza kojoj je bilo potrebno postaviti statičku IP adresu na vrijednost 192.170.0.1. Na *Raspberry* računalu odnosno u *U-boot* naredbenom retku postavile su se varijable okruženja koje odgovaraju IP adresi servera (razvojno računalo) i klijenta (*Raspberry*): *setenv serverip 192.170.0.1; setenv ipaddr 192.170.0.100*. Naposlijetku, prije pokretanja jezgre, definirano je koji će *root* datotečni sustav biti pokrenut preko NFS-a te se stoga postavila *bootargs* varijabla okruženja: *bootargs=root=/dev/nfs rw ip=192.170.0.100 8250.nr_uarts=1 console=ttyS0,115200 nfsroot=192.170.0.1:/home/debian/novo/nfsroot*. Nakon toga jezgra je pokrenuta naredbom *run bootcmd*. Kada je jezgra uspješno učitana ispisuje se redak

za prijavu u sustav. U prilogu P.3.15. prikazana je prijava u sustav te primjer korištenja osnovnih naredbi datotečnog sustava.

Kako bi se razvijena ljuska pokrenula na *Raspberry* računalu, koristio se križni prevoditelj (engl. *cross compiler*) *arm-linux-gnueabihf-gcc*. Njime se izvorni kod ljuske na razvojnom računalu prevodi u izvršni kod koji se može izvoditi na ciljanoj *ARMhf* platformi. Prevođenje se izvelo sljedećom naredbom: *arm-linux-gnueabihf-gcc jobs.c pipred.c commands.c -o shell-arm -static*. Korištena je *-static* opcija čime generirana izvršna datoteka sadrži sve biblioteke koje su potrebne da se program izvrši. Na taj način program, odnosno ljuska, potpuno je samostalna te ne ovisi o postojećim bibliotekama na izgrađenom *Linux* operacijskom sustavu. Izvršna datoteka se zatim kopirala u *root* datotečni sustav nakon čega je pokrenuta (slika 3.25.).

```
buildroot:~# ./shell-arm
SHELL$ cat file1 | grep test | tail -1
test
SHELL$ exit
buildroot:~#
```

Slika 3.25. Korištenje razvijene ljuske na Raspberry Pi platformi

4. EVALUACIJA IZRAĐENE LJUSKE I USPOREDBA S BASH LJUSKOM

U ovom poglavlju prvo je opisan način provedenih mjerenja performansi ljuske predstavljene ovim radom i BASH ljuske. Zatim su dani rezultati izvedenih mjerenja performansi na osobnom računalu i *Raspberry Pi* računalu.

4.1. Opis provedenih mjerenja performansi

U svrhu analize performansi posebno je provedeno mjerenje vremena prilikom čitanja i parsiranja naredbi te zasebno kod izvršavanja naredbi. Pod pojmom „izvršavanje naredbi“ misli se na funkciju koja obuhvaća preusmjeravanje, ulančavanje i direktno izvršavanje naredbe. Mjerenja su provedena za sljedeće naredbe: *cat file1*, *cat < file1*, *cat file1 > file2*, *cat file1 | cat*, *cat < file1 | cat | cat | cat > file2*. Veličina datoteke *file1* korištene za preusmjeravanje ulaza iznosi 5 bajtova. Mjereno je stvarno vrijeme (engl. *wall time*). Budući da su za istu naredbu pri svakom izvođenju dobivena različita vremena izvršavanja i parsiranja, mjerenje svake naredbe provedeno je na 10000 ponavljanja kako bi se dobile pouzdanije vrijednosti. Zbog toga je izmijenjen izvorni kod predstavljene ljuske i BASH-a tako da se umjesto beskonačne petlje postavila petlja koja se izvršava 10000 puta. Uz to, funkcije koje čitaju korisnikov unos izmijenjene su na način da ljuske više ne čekaju korisnikov unos već se u svakoj iteraciji vraća isti niz znakova odnosno ista naredba.

Mjerenja su provedena unutar izvornih kodova razvijene ljuske i BASH ljuske *int timespec_get(struct timespec *ts, int base)* funkcijom. Ova funkcija dohvaća trenutno vrijeme i sprema ga u strukturu *timespec* na koju pokazuje pokazivač *ts*. Vrijeme se u strukturu sprema u obliku sekunda i nanosekunda. Na slici 4.1 prikazan je način mjerenja vremena izvršavanja funkcije koja izvršava naredbe u predstavljenoj ljusci. Vrijeme se dohvaća neposredno prije te nakon izvršavanja ove funkcije. Vremena u nanosekundama se oduzimaju te se rezultat dijeli sa 10^9 kako bi se dobilo vrijeme u sekundama. Taj rezultat dodaje se razlici vremena u sekundama te dobiva se konačno vrijeme. To je mjereno vrijeme za jednu iteraciju petlje odnosno jedno izvođenje naredbe. Za svaku iteraciju (njih 10000) računaju se vremena čitanja i parsiranja te izvršavanja naredbi. Vremena dobivena u svakoj iteraciji petlje se zbrajaju te se dobiva ukupno vrijeme čitanja i parsiranja naredbi i ukupno vrijeme izvršavanja naredbi.

```
timespec_get(&begin2,TIME_UTC);
exeCmds(numSimpleCmds,new_job);
timespec_get(&end2,TIME_UTC);
double timespent2=(end2.tv_sec-begin2.tv_sec)+
                  (end2.tv_nsec-begin2.tv_nsec)/1000000000.0;
```

Slika 4.1. Mjerenje vremena s *timespec_get()* funkcijom

4.2. Rezultati mjerenja performansi

4.2.1. Mjerenje performansi na osobnom računalu

U tablicama 4.1. - 4.5. prikazani su rezultati obavljenih mjerenja na virtualnom stroju pokrenutom na osobnom računalu. Računalo sadrži *Intel Core i7-6700* procesor brzine 3.40 GHz te 16 GB radne memorije s 8 logičkih procesora, dok je virutalnom stroju dodijeljeno 8 GB radne memorije i 2 logička procesora. Rezultati svih mjerenja prikazani su u sekundama.

Tablica 4.1. Rezultati mjerenja za naredbu *cat file1*

cat file1	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
Ukupno vrijeme	0.067274	0.657455	26.098858	30.171520
St.devijacija	0.000020	0.000076	0.001217	0.001280
Prosječno vrijeme	0.000007	0.000066	0.002610	0.003017
Min. Vrijeme	0.000002	0.000026	0.001203	0.001464
Maks. Vrijeme	0.001180	0.002542	0.016197	0.016917
Medijan	0.000004	0.000047	0.002239	0.002611

Tablica 4.2. Rezultati mjerenja za naredbu *cat < file1*

cat < file1	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
Ukupno vrijeme	0.075375	0.723475	26.749378	32.630756
St.devijacija	0.000037	0.000095	0.001229	0.001487
Prosječno vrijeme	0.000008	0.000072	0.002675	0.003263
Min. Vrijeme	0.000003	0.000027	0.001230	0.001451
Maks. Vrijeme	0.002880	0.002843	0.013738	0.023203
Medijan	0.000005	0.000049	0.002300	0.002811

Tablica 4.3. Rezultati mjerenja za naredbu *cat file1 > file2*

cat file1 > file2	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
Ukupno vrijeme	0.045871	0.438206	24.884458	29.644882
St.devijacija	0.000008	0.000043	0.000421	0.000808
Prosječno vrijeme	0.000005	0.000044	0.002488	0.002964
Min. Vrijeme	0.000003	0.000027	0.001410	0.001715
Maks. Vrijeme	0.000422	0.002567	0.011553	0.018393
Medijan	0.000004	0.000035	0.002429	0.002851

Tablica 4.4. Rezultati mjerenja za naredbu *cat file1 | cat*

cat file | cat	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
Ukupno vrijeme	0.086357	0.729670	29.696754	44.818660
St.devijacija	0.000028	0.000098	0.001352	0.001694
Prosječno vrijeme	0.000009	0.000073	0.002970	0.004482
Min. Vrijeme	0.000004	0.000030	0.001469	0.002061
Maks. Vrijeme	0.001775	0.004493	0.013966	0.018271
Medijan	0.000005	0.000050	0.002441	0.004077

Tablica 4.5. Rezultati mjerenja za naredbu *cat < file1 | cat | cat | cat > file2*

cat < file1 | cat | cat | cat > file2	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
Ukupno vrijeme	0.051319	0.632814	45.414920	61.482278
St.devijacija	0.000010	0.000056	0.000703	0.001033
Prosječno vrijeme	0.000005	0.000063	0.004541	0.006148
Min. Vrijeme	0.000003	0.000036	0.003219	0.004292
Maks. Vrijeme	0.000616	0.000726	0.015323	0.018543
Medijan	0.000004	0.000045	0.004366	0.005901

Prema tablicama 4.1. – 4.5. može se vidjeti kako izvršavanje naredbi zahtijeva znatno više vremena od čitanja i parsiranja. Razlog toga je činjenica da se kod izvršavanja naredbe izvode brojni sistemski pozivi poput *fork()*, *pipe()*, te glavni sistemski poziv koji direktno izvršava naredbu *execvp()* (kojeg koristi ljuska predstavljena u ovom radu, a koja je u tablicama navedena pod nazivom *Shell*). BASH koristi *execve()* poziv koji se razlikuje od *execvp()*, po tome što mu je potrebno predati putanju izvršne datoteke. Što znači da BASH ima posebnu funkciju koja pronalazi putanju datoteke, dok sistemski poziv *execvp()*, kao što je već objašnjeno, ima ugrađenu funkcionalnost pronalaženja putanje preko *PATH* varijable okruženja. U obje ljuske, unutar funkcije gdje se izvršava naredba, odvijaju se i preusmjeravanja izlaza i ulaza gdje se koriste sistemski pozivi *open()* i *close()* što također povećava ukupno vrijeme izvršavanja naredbe.

Svakim izvođenjem programa sustav odlučuje koliko će resursa određenom procesu dati za izvršavanje. Stoga, naredba će se u svakoj od 10000 iteracija izvesti u različitom vremenu. Zato se izvršavanje naredbe ne može promatrati kao deterministički proces što predstavlja problem prilikom mjerenja stvarnog vremena. To se može vidjeti po standardnim devijacijama i vrijednostima minimalnih i maksimalnih vremena u danim tablicama. No, mogu postojati razlozi zbog kojih će se program u određenom vremenskom intervalu dulje izvršavati. Primjerice, ovisno

o opterećenju sustava, vrijeme izvođenja programa može se mijenjati. Programu, odnosno ljusci, dodijeliti će se manje resursa u slučaju opterećenja sustava drugim procesima. To može uzrokovati sporije izvođenje programa što se može vidjeti prema tablici 4.6. gdje su prikazane prosječne vrijednosti mjerenja u sekundama, pri čemu je sustav opterećen pomoću alata *stress* [23]. Pokretanjem naredbe „*stress -m 2 --vm-bytes 2800M --vm-keep*“ iskorištenost oba logička procesora na osobnom računalu odnosno virutalnom stroju postala je 100% pri čemu je zauzeto 93.8% radne memorije. U tablici 4.7. prikazani su omjeri prosječnih vrijednosti mjerenja na opterećenom sustavu i prosječnih vrijednosti mjerenja na sustavu koji nije opterećen. Može se vidjeti kako opterećenje sustava ima značajan utjecaj na brzinu izvršavanja naredbi budući da su vremena izvršavanja porasla od 3.7 do 14.9 puta u slučaju ljuske predstavljene u ovom radu, dok su u slučaju BASH ljuske vremena izvršavanja porasla od 2.8 do 11.4 puta. Kao što je već objašnjeno, kod izvršavanja naredbi izvode se brojni sistemski pozivi koji zauzimaju resurse sustava zbog čega opterećenje sustava ima utjecaj na njihovo izvođenje. S druge strane, prilikom čitanja i parsiranja naredbi izvode se funkcije koje ne koriste sistemske pozive i budući da se za dio naredbi vrijeme potrebno za čitanje i parsiranje povećalo, a za drugi dio smanjilo, ne može se zaključiti kako opterećenje sustava utječe na brzinu čitanja i parsiranja naredbi.

Tablica 4.6. Prosječne vrijednosti mjerenja na opterećenom sustavu

NAREDBA	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
cat file1	0.000006	0.000061	0.009571	0.008410
cat < file1	0.000005	0.000063	0.008377	0.009991
cat file1 > file2	0.000007	0.000056	0.020070	0.010660
cat file | cat	0.000009	0.000076	0.022787	0.028579
cat < file1 | cat | cat | cat > file2	0.000009	0.000083	0.067827	0.069857

Tablica 4.7. Omjeri prosječnih vrijednosti mjerenja na opterećenom sustavu i na sustavu koji nije opterećen

NAREDBA	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
cat file1	0.905402	0.920904	3.667338	2.787499
cat < file1	0.666985	0.864196	3.131574	3.061690
cat file1 > file2	1.627477	1.270008	8.065237	3.595928
cat file | cat	1.037125	1.045984	7.673123	6.376489
cat < file1 | cat | cat | cat > file2	1.726242	1.316799	14.934908	11.362096

Veći broj ponavljanja izvođenja programa doprinijet će i pouzdanijem konačnom rezultatu. Prema tablicama 4.1. – 4.5. može se primijetiti kako je medijalna vrijednost u svakom slučaju mjerenja manja od prosječne vrijednosti što znači da postoje izuzetno visoke vrijednosti ekstrema koje utječu na veći prosjek.

Svako dobiveno prosječno vrijeme, na sustavu koji nije opterećen, za BASH i ljusku predstavljenu u ovom radu može se staviti u omjer kako bi se lakše usporedile brzine izvođenja naredbi. Prema tablici 4.8. može se vidjeti da razvijena ljuska izvršava naredbe od 1.16 do 1.51 puta brže u odnosu na BASH. Iako vrijeme utrošeno na čitanje i parsiranje naredbe čini neznatni dio ukupnog vremena kod izvođenja naredbi, može se primijetiti da je to vrijeme od 8 do 12 puta kraće nego kod BASH ljuske.

Tablica 4.8. Omjeri prosječnih vrijednosti dobivenih mjerenjem u BASH ljusci i predstavljenoj ljusci za osobno računalo

NAREDBA	Čitanje i parsiranje	Izvršavanje	Ukupno
cat file1	9.772795	1.156048	1.178201
cat < file1	9.598342	1.219870	1.243412
cat file1 > file2	9.553007	1.191301	1.206686
cat file | cat	8.449460	1.509211	1.529334
cat < file1 | cat | cat | cat > file2	12.330989	1.353790	1.366181

4.2.2. Mjerenje performansi razvijene ljuske na Raspberry Pi računalu

Mjerenje performansi ljuske obavljeno je i na *Raspberry Pi* računalu čije su specifikacije dane u poglavlju 3.10. Mjerenje je provedeno na jednak način kao na osobnom računalu. Za pokretanje BASH ljuske na *Raspberry Pi* računalu korištena je konfiguracijska datoteka (engl. *configure*) koja dolazi uz izvorni kod BASH-a. *Configure* je skripta koja omogućuje prilagodbu prevođenja i instalacije programa za određenu platformu. Stoga, prije pokretanja *make* naredbe kojom se prevodi izvorni kod i stvara izvršna datoteka BASH ljuske, potrebno je izvršiti naredbu „ *./configure --build x86_64-pc-linux-gnu --host arm-linux-gnueabihf --enable-static-link* “. Argument *--build* specificira platformu na kojoj se vrši izgradnja programa, u ovom slučaju je to osobno računalo odnosno *x86_64* sustav. Pomoću argumenta *--host* specificira se platforma na kojoj se program želi pokrenuti, u ovom slučaju je to *Raspberry Pi* odnosno *ARMhf* sustav. Rezultati provedenih mjerenja na *Rasbperry Pi* računalu navedeni su u tablicama 4.9. – 4.13.

Tablica 4.9. Rezultati mjerenja za naredbu *cat file1*

cat file1	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
Ukupno vrijeme	0.213640	140.821885	288.327637	392.161824
St.devijacija	0.000003	0.003900	0.009343	0.006773
Prosječno vrijeme	0.000021	0.014082	0.028833	0.039216
Min. Vrijeme	0.000018	0.006402	0.014640	0.023918
Maks. Vrijeme	0.000105	0.034321	0.168216	0.085773
Medijan	0.000021	0.013472	0.027589	0.038593

Tablica 4.10. Rezultati mjerenja za naredbu *cat < file1*

cat < file1	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
Ukupno vrijeme	0.247973	150.448795	313.037220	399.378521
St.devijacija	0.000003	0.004876	0.009215	0.007748
Prosječno vrijeme	0.000025	0.015045	0.031304	0.039938
Min. Vrijeme	0.000022	0.006490	0.015219	0.023700
Maks. Vrijeme	0.000105	0.087871	0.075673	0.094403
Medijan	0.000024	0.013958	0.030128	0.039339

Tablica 4.11. Rezultati mjerenja za naredbu *cat file1 > file2*

cat file1 > file2	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
Ukupno vrijeme	0.276408	109.126971	544.966244	484.996606
St.devijacija	0.000002	0.002363	0.004792	0.005531
Prosječno vrijeme	0.000028	0.010913	0.054497	0.048500
Min. Vrijeme	0.000024	0.006517	0.029396	0.036748
Maks. Vrijeme	0.000081	0.090224	0.163985	0.197271
Medijan	0.000027	0.010752	0.053888	0.047704

Tablica 4.12. Rezultati mjerenja za naredbu *cat file1 | cat*

cat file | cat	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
Ukupno vrijeme	0.283660	139.409773	364.335832	479.133269
St.devijacija	0.000003	0.003871	0.009056	0.007325
Prosječno vrijeme	0.000028	0.013941	0.036434	0.047913
Min. Vrijeme	0.000025	0.006064	0.017734	0.028941
Maks. Vrijeme	0.000120	0.086174	0.074539	0.104775
Medijan	0.000028	0.013451	0.035569	0.047649

Tablica 4.13. Rezultati mjerenja za naredbu *cat < file1 | cat | cat | cat > file2*

cat < file1 | cat | cat | cat > file2	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
Ukupno vrijeme	0.486984	107.375139	601.924518	499.573206
St.devijacija	0.000003	0.002007	0.005174	0.004914
Prosječno vrijeme	0.000049	0.010738	0.060192	0.049957
Min. Vrijeme	0.000046	0.006591	0.037301	0.036338
Maks. Vrijeme	0.000135	0.088395	0.186131	0.195010
Medijan	0.000048	0.010327	0.059644	0.049503

Prema vrijednostima standardnih devijacija dobivenih mjerenjem vremena izvršavanja naredbi na *Raspberry Pi* računalu može se vidjeti da su odstupanja od prosječne vrijednosti puno manja u odnosu na vrijednosti odstupanja dobivenih mjerenjem na osobnom računalu. Također, uspoređujući medijalne vrijednosti mjerenja na osobnom računalu i *Raspberry Pi* računalu može se vidjeti da su u slučaju mjerenja na *Raspberry Pi* računalu one puno bliže prosječnim vrijednostima što ukazuje na to da su vrijednosti simetrično raspodijeljene. Izmjerene vrijednosti nisu raspršene te se većina vrijednosti nalazi blizu prosjeka zbog čega prosječne vrijednosti mjerenja na *Raspberry Pi* računalu daju pouzdanije rezultate mjerenja od prosječnih vrijednosti dobivenih mjerenjem na osobnom računalu. *Raspberry Pi* sadrži minimalnu distribuciju *Linux* operacijskog sustava što rezultira stabilnijim uvjetima izvođenja programa odnosno postoji mali broj pozadinskih procesa koji se izvode u istovremeno s pokrenutim programom. U slučaju osobnog računala, broj pozadinskih procesa je puno veći što doprinosi većim odstupanjima prilikom mjerenja.

Zbog očigledno slabijih specifikacija *Raspberry Pi* računala u odnosu na osobno računalo očekivano je i dulje vrijeme izvršavanja te čitanja i parsiranja naredbi što se može vidjeti prema vrijednostima navedenim u tablici 4.14. Svaka vrijednost predstavlja omjer prosječnog vremena dobivenog na *Rasbperry Pi* računalu i prosječnog vremena dobivenog na osobnom računalu. U odnosu na mjerenja provedena na osobnom računalu, može se odmah primijetiti kako je BASH ljusci na *Raspberry Pi* računalu potrebno puno više vremena za čitanje i parsiranje naredbi što u ovom slučaju čini značajan udio ukupnog vremena pri izvođenju naredbi. Također, prema tablici 4.15. može se vidjeti kako je čitanje i parsiranje naredbi i do nekoliko stotina puta sporije u BASH ljusci u odnosu na razvijenu ljusku. Uspoređujući odnose vremena pri izvršavanju naredbi (tablica 4.15.), BASH brže izvršava naredbe u slučajevima gdje se koristi operator preusmjeravanja izlaza,

„>“. No, zbog vrlo dugog vremena potrebnog za čitanje i parsiranje naredbe u BASH ljusci, ukupno vrijeme izvođenja naredbi je u tim slučajevima podjednako za BASH i razvijenu ljusku.

Tablica 4.14. Omjeri prosječnih vrijednosti dobivenih mjerenjem na *Raspberry Pi* računalu i osobnom računalu

NAREDBA	Čitanje i parsiranje- Shell	Čitanje i parsiranje- BASH	Izvršavanje- Shell	Izvršavanje- BASH
cat file1	3.175670	214.192431	11.047519	12.997748
cat < file1	3.289857	207.952998	11.702598	12.239328
cat file1 > file2	6.025768	249.031211	21.899864	16.360214
cat file | cat	3.284737	191.058661	12.268541	10.690486
cat < file1 | cat | cat | cat > file2	9.489351	169.678830	13.253894	8.125483

Tablica 4.15. Omjeri prosječnih vrijednosti dobivenih mjerenjem u BASH ljusci i predstavljenoj ljusci za *Raspberry Pi*

NAREDBA	Čitanje i parsiranje	Izvršavanje	Ukupno
cat file1	659.155051	1.360126	1.847166
cat < file1	606.714421	1.275818	1.755038
cat file1 > file2	394.803953	0.889957	1.089650
cat file | cat	491.467859	1.315087	1.696407
cat < file1 | cat | cat | cat > file2	220.490076	0.829960	1.007531

Uz to što je interpretator unosa u naredbenom retku, BASH je i skriptni jezik. U odnosu na razvijenu ljusku BASH pruža korisniku puno više mogućnosti. Primjerice, to uključuje više proširenja ljuske i preusmjeravanja koja se mogu koristiti. Uz to, moguće je pisati funkcije, petlje i uvjete unutar naredbenog retka kao i unutar skripti. Gramatika prema kojoj radi BASH parser omogućava veću fleksibilnost i kompliciranije izraze pri unosu u odnosu na razvijenu ljusku. No, to opet utječe na kompleksnost implementacije parsera što dovodi do duljeg vremena parsiranja. Osim toga, BASH parser je razvijen za potrebe rada na osobnom računalu te se vrijeme parsiranja dodatno povećava na ugradbenoj platformi zbog nedostatka optimizacije. Također, velik broj mogućnosti koje BASH pruža utječu i na kompleksnost implementacije pri izvršavanju naredbi zbog čega se za istu naredbu unutar BASH-a izvodi veći broj funkcija u odnosu na razvijenu ljusku.

5. ZAKLJUČAK

Ljuska, kao sučelje između korisnika i operacijskog sustava, predstavlja neophodan alat za izvršavanje naredbi. BASH ljuska, kao jedna od najkorištenijih ljuski u *Linux* operacijskim sustavima, pruža širok spektar mogućnosti s ciljem olakšavanja komunikacije korisnika s operacijskim sustavom. Na temelju BASH ljuske, razvijena je ljuska za *Linux* operacijski sustav u programskom jeziku C.

Ljuska, razvijena u ovom radu, čita korisnikov unos, parsira ga, obavlja proširenja, preusmjerava ulaze i izlaze naredbi te naposlijetku sistemskim pozivom izvršava naredbu i njezin izlaz prikazuje korisniku. Ona omogućava izvršavanje jednostavnih naredbi s više argumenata kao i složenih naredbi sastavljenih od više jednostavnih naredbi. Više redaka unosa je moguće navesti unutar jedne naredbene linije odvajajući ih znakom „ ; “. Omogućeno je ulančavanje naredbi operatorom „ | “ i preusmjeravanje ulaza i izlaza koristeći operatore „< “, „>“, „>> “, „&>“. Od proširenja ljuske implementirano je proširenje naziva datoteke i proširenje varijabli. Bitna značajka razvijene ljuske je kontrola posla koja korisniku omogućava prebacivanje izvršavanja naredbi u prednji ili pozadinski plan kao i zaustavljanje, prekidanje i nastavljanje izvršavanja naredbi. Ljuska je pokrenuta na posebno izgrađenoj distribuciji *Linux* operacijskog sustava za *Raspberry Pi* ugradbenu platformu.

Prema provedenim mjerenjima performansi na osobnom računalu ukupno vrijeme koje je potrebno za izvođenje naredbi je od 1.2 do 1.5 puta kraće u predstavljenoj ljusci u odnosu na BASH. Na *Raspberry Pi* platformi ukupno vrijeme za ove dvije ljuske podjednako je u slučajevima gdje se koristi operator preusmjeravanja izlaza, „>“, dok u ostalim slučajevima predstavljena ljuska zahtijeva i do 1.8 puta manje ukupnog vremena za izvođenje naredbi od BASH ljuske. Širok raspon mogućnosti BASH ljuske povećava kompleksnost implementacije što uzrokuje smanjenje njezinih performansi odnosno povećava se vrijeme izvođenja naredbi. Kod općenite korisničke upotrebe ljuske ove se razlike u vremenu izvođenja naredbi teško mogu primijetiti.

Ugradbeni sustavi često koriste minimalne izvedbe operacijskih sustava zbog čega je smanjen utjecaj drugih procesa na izvršavanje određenog programa što rezultira stabilnijim uvjetima u kojima se program izvodi. Zbog toga su odstupanja od prosječnih vrijednosti puno manja u slučaju mjerenja na *Raspberry Pi* ugradbenoj platformi, ali je ukupno vrijeme izvođenja naredbi dulje zbog slabijih specifikacija ugradbenog sustava u odnosu na osobno računalo. Također, često je potrebno dodatno prilagoditi i optimizirati program za rad na ugradbenoj platformi, posebice ako

se radi o kompleksnijem programu poput BASH parsera jer se u suprotnom njegovo vrijeme izvršavanja dodatno povećava.

Predstavljenu ljusku moguće je poboljšati na način da se implementiraju dodatne mogućnosti i prošire i poboljšaju njezine postojeće značajke. Primjerice, daljnji rad bi uključivao razvoj ostalih operatora preusmjeravanja i proširenja ljuske. Dakako, ako bi se išlo prema razvoju ljuske koja je slična današnjim modernim ljuskama to bi uključivalo implementaciju čitavog skriptnog jezika što bi zahtijevalo razvoj novog parsera. No, za potrebe izvođenja nekakvog kontinuiranog procesa gdje je brzina izvršavanja od presudne važnosti, daljnji rad bi uključivao optimizaciju postojećeg rješenja.

LITERATURA

[1] C. Ramey i B. Fox, „Bash Reference Manual“. 19. rujan 2022.

[2] A. Brown, Ur., *The architecture of open source applications: elegance, evolution, and a few fearless hacks*. s.l., 2011.

[3] C. Ramey, „The GNU Readline Library“. Pristupljeno: 06. srpanj 2023. [Na internetu]. Dostupno na: https://tiswww.case.edu/php/chet/readline/rltop.html

[4] „The Bash Parser“. https://mywiki.wooledge.org/BashParser (pristupljeno 07. srpanj 2023.).

[5] S. C. Johnson, „Yacc: Yet Another Compiler-Compiler“, *Unix Program. Man.*, sv. 2, stu. 2001.

[6] D. Kerr, „Effective Shell“, *Understanding Shell Expansion*. https://effective-shell.com/part-6-advanced-techniques/understanding-shell-expansion/ (pristupljeno 12. srpanj 2023.).

[7] M. K. Dalheimer i M. Welsh, *Running Linux*, 5th ed. Sebastopol: O'Reilly Media, Inc., 2009.

[8] K. Rajalingham, „Bash Linux Redirection Operators“. https://linuxhint.com/redirection-operators-bash/ (pristupljeno 18. srpanj 2023.).

[9] J. Ellingwood, „How To Use Bash's Job Control to Manage Foreground and Background Processes“. https://www.digitalocean.com/community/tutorials/how-to-use-bash-s-job-control-to-manage-foreground-and-background-processes (pristupljeno 17. srpanj 2023.).

[10] N. J. Dato, „SIGINT And Other Termination Signals in Linux“. https://www.baeldung.com/linux/sigint-and-other-termination-signals (pristupljeno 17. srpanj 2023.).

[11] „IBM Documentation“, *Process termination*. https://www.ibm.com/docs/en/aix/7.2?topic=management-process-termination (pristupljeno 17. srpanj 2023.).

[12] S. Bourne, „The Unix Shell“, *BYTE*, str. 187–204, listopad 1983.

[13] M. Jones, „Evolution of shells in Linux“. https://developer.ibm.com/tutorials/l-linux-shells/#resources (pristupljeno 14. srpanj 2023.).

[14] „Tenex C Shell“. Pristupljeno: 15. srpanj 2023. [Na internetu]. Dostupno na: https://www.tcsh.org/

[15] B. Rosenblatt, *Learning the Korn shell*, 1st ed. u A Nutshell handbook. Sebastopol, CA: O'Reilly & Associates, 1993.

[16] P. Falstad, „Z shell“. Pristupljeno: 15. srpanj 2023. [Na internetu]. Dostupno na: https://zsh.sourceforge.io/

[17] C. Hoffman, „What’s the Difference Between Bash, Zsh, and Other Linux Shells?“ https://www.howtogeek.com/68563/htg-explains-what-are-the-differences-between-linux-shells/ (pristupljeno 15. srpanj 2023.).

[18] A. Taddei, „Shell Choice - A shell comparison“. 28. rujan 1994.

[19] Y. H. Kamath i M. M. Matthe, „Implementation of an FP-Shell“, *IEEE Trans. Softw. Eng.*, sv. SE-13, izd. 5, str. 532–539, svi. 1987, doi: 10.1109/TSE.1987.233198.

[20] M. I. Vuskovic, „R-shell: a UNIX-based development environment for robotics“, u *Proceedings. 1988 IEEE International Conference on Robotics and Automation*, Philadelphia, PA, USA: IEEE Comput. Soc. Press, 1988, str. 457–460. doi: 10.1109/ROBOT.1988.12093.

[21] M. Kerrisk, „glob(3) - Library Functions Manual“. https://man7.org/linux/man-pages/man3/glob.3.html.

[22] „Upoznavanje s Raspberry Pi 3 računalom. Izgradnja Linuxa za Raspberry Pi 3“. Materijali s laboratorijskih vježbi iz kolegija: Linux u ugradbenim sustavima, FERIT Osijek, 2023.

[23] A. Waterland, „stress - tool to impose load on and stress test a computer system“. Ubuntu Manpage Repository. [Na internetu]. Dostupno na: https://manpages.ubuntu.com/manpages/lunar/en/man1/stress.1.html

SAŽETAK

U ovom diplomskom radu napravljena je ljuska za *Linux* operacijski sustav koja ima mogućnost izvršavanja jednostavnih i složenih naredbi, ulančavanja naredbi operatorom „|“ te preusmjeravanja ulaza i izlaza koristeći operatore „<“, „>“, „>> “, „&>“. Također, implementirano je proširenje naziva datoteke, proširenje varijabli i kontrola posla. Analiza performansi razvijene ljuske i BASH ljuske provedena je mjerenjem vremena prilikom čitanja i parsiranja naredbi te uslijed izvršavanja naredbi. Mjerenja su izvedena na osobnom računalu te na posebno izgrađenoj *Linux* distribuciji za *Raspberry Pi* ugradbenu platformu. Razvijena ljuska u prosjeku brže čita i parsira te izvršava naredbe od BASH ljuske.

KLJUČNE RIJEČI:

Linux, ljuska, BASH, *Raspberry Pi*

CUSTOM SHELL FOR LINUX BASED EMBEDDED SYSTEM

ABSTRACT

In this master's thesis, a shell for the *Linux* operating system has been developed. Features of the proposed shell are executing simple and complex commands, piping with the " | " operator and redirecting input and output using the operators "< ", ">", ">> ", "&>". Additionally, file name expansion, variable expansion and job control are implemented. Performance analysis of the developed shell and the BASH shell was performed by measuring the time taken separately during command reading and parsing, as well as during command execution. The measurements were performed on a personal computer as well as on a custom-built *Linux* distribution for the *Raspberry Pi* embedded platform. The developed shell, on average, reads, parses, and executes commands faster than the BASH shell.

KEYWORDS

Linux, shell, BASH, *Raspberry Pi*

ŽIVOTOPIS

Marko Brođanac rođen je 22. srpnja 1999. godine u Osijeku. Nakon završene srednje škole, III. gimnazije Osijek, 2018. godine upisuje preddiplomski sveučilišni studij elektrotehnike na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Na drugoj godini odabire smjer komunikacije i informatika. Akademski naziv sveučilišnog prvostupnika elektrotehnike i informacijske tehnologije stječe 2021. godine kad upisuje diplomski studij elektrotehnike, smjer Mrežne tehnologije.

_____

Potpis autora

PRILOZI

Prilog P.3.1. Kompletan programski kod izrađene ljuske (elektronički prilog)

Prilog P.3.2. Programski kod funkcije koja vraća pokazivač na token

```
char** addCmd(char** line){
    int max_len=5;
    char **p=line;
    int f=0;
    int i=0,j=0;
    char **token=(char**)malloc(max_len*sizeof(char*));
    char znak[2];
    char *znk=(char*)malloc(2*sizeof(char));
    if(strchr(specialchars,**p)){
        znak[0]=**p;
        znak[1]='\0';
        if(**p=='>' && (*(*p+1))=='>'){
            znak[0]='+';
            *p=*p+1;
        }
        else if(**p=='&' && (*(*p+1))=='>' ){
            znak[0]='#';
            *p=*p+1;
        }

        strcpy(znk,znak);
        token[0]=znk;
        token[1]=NULL;
        *p=*p+1;
        *line=*p;
        return token;
    }
    while(**p != '\0'){
        while(  strchr(whitespace,**p) && **p!='\0' ){
            *p=*p+1;
        }
        if(strchr(specialchars,**p)){
            *line=*p;
            break;
        }
        token[i]=addArg(p);
        i++;
        if(i>=max_len){
            max_len=max_len+5;
            token=(char**)realloc(token,max_len*sizeof(char*));
            }
        }
    token[i]=NULL;
    return token;
}
```

Prilog P.3.3. Programski kod funkcije koja vraća pokazivač na argument

```
char* addArg(char** p){
    char *s;
    s=*p;
    int max_len=10;
    int i=0;
    char *polje=(char*)malloc(max_len*sizeof(char));
    while(strchr(whitespace,*s)==NULL && strchr(specialchars,*s)==NULL){
        polje[i]=*s;
```

```
            i++;
            if(i>=max_len){
                        max_len=max_len+10;
                        polje=(char*)realloc(polje,max_len*sizeof(char));
                        if(polje==NULL){
                            fprintf(stderr,"%s","Aloc error\n");
                            exit(0);
                        }
            }
            s++;
        }
        *p=s;
        polje[i]='\0';
        return polje;
}
```

Prilog P.3.4. Programski kod funkcije koja dodaje naredbu u listu struktura naredbi

```
void appendCmd(job** job){
    cmd* new_cmd=(cmd*)malloc(sizeof(cmd));
    cmd* last=(*job)->root;
    new_cmd->next=NULL;
    new_cmd->in=NULL;
    new_cmd->out=NULL;
    new_cmd->status=RUNNING;
    if((*job)->root==NULL){
        (*job)->root=new_cmd;
        return;
    }
    while(last->next!=NULL){
        last=last->next;
    }
    last->next=new_cmd;
}
```

Prilog P.3.5. Programski kod funkcije koja dodaje posao u listu struktura poslova

```
job* appendJob(job** head,int back){
    job* new_job=(job*)malloc(sizeof(job));
    job* last=*head;
    new_job->next=NULL;
    new_job->root=NULL;
    new_job->pgid=0;
    new_job->notified=0;
    if(back==1){
        new_job->mode=0; //background
    }else{
        new_job->mode=1; //foreground
    }
    if(*head==NULL){
        (*head)=new_job;
        new_job->id=1;
        return new_job;
    }
    while(last->next!=NULL){
        last=last->next;
    }
    new_job->id=last->id+1;
    last->next=new_job;
    return new_job;
}
```

Prilog P.3.6. Programski kod funkcije koja izvršava naredbe

```
void exeCmds(int numSimpleCmds,job* job) {
  cmd* cmd;
  int i, j = 0;
  int b;
  pid_t pid;
  int fd[2 * numSimpleCmds];
  pid_t id;

  //STVARANJE CJEVOVODA
  for(i=0; i<numSimpleCmds; i++) {
    if(pipe(fd + i * 2)<0) {
      perror("pipe");
      exit(EXIT_FAILURE);
    }
  }
  for(cmd=job->root;cmd!=NULL;cmd=cmd->next){
    if((pid = fork())==-1){
      perror("fork");
      exit(EXIT_FAILURE);
    }
    //PODREĐEN PROCES
    else if(pid==0){
        id=getpid();
        if(job->pgid==0){
            job->pgid=id;
        }
        setpgid(id,job->pgid);
        if(job->mode==1){
            tcsetpgrp(0,job->pgid);
        }
        signal (SIGINT, SIG_DFL);
        signal (SIGQUIT, SIG_DFL);
        signal (SIGTSTP, SIG_DFL);
        signal (SIGTTIN, SIG_DFL);
        signal (SIGTTOU, SIG_DFL);
        if(cmd->next==NULL){ //PREUSMJERAVANJE IZLAZA
            if(cmd->out!=NULL){
                if(cmd->modeOut==1){
                    redirectOut(cmd->out);
                }
                else if(cmd->modeOut==2){
                    redirectAppend(cmd->out);
                }
                else if(cmd->modeOut==3){
                    redirectErrOut(cmd->out);
                }

            }
        }
        else{
            //ULANČAVANJE
            if (dup2(fd[j+1],1)<0) {
                  perror("dup2");
                  exit(EXIT_FAILURE);
            }
        }

      if(j!=0){
        if (dup2(fd[j-2],0)<0) {
          perror("dup2");
          exit(EXIT_FAILURE);
        }
      }else{
          //PREUSMJERAVANJE ULAZA
          if(cmd->in!=NULL){
```

```
                redirectIn(cmd->in);
            }
        }

        for (b=0;b<2*numSimpleCmds;b++) {
          close(fd[b]);
        }
          //IZVRŠAVANJE NAREDBE
        if (execvp(cmd->name[0],cmd->name)<0) {
          perror(cmd->name[0]);
          exit(EXIT_FAILURE);
        }
        exit(0);
      }
      //NADREĐEN PROCES
      else{
          //DODAVANJE PODREĐENOG PROCESA U GRUPU PROCESA
          cmd->pid=pid;
          if(job->pgid==0){
              job->pgid=pid;
          }
          setpgid(pid,job->pgid);
      }
      j=j+2;
    }
    //ZATVARANJE DESKRIPTORA
    for (i=0;i<2*numSimpleCmds;i++) {
      close(fd[i]);
    }
    if(job->mode==1){
        put_in_fg(job,0);
    }else{
        put_in_bg(job,0); //AKO JE NA KRAJU ZNAK &
    }                     //POSTAVI IZVRŠAVANJE U POZADINU

}
```

Prilog P.3.7. Programski kod funkcije koja čeka da se promijeni stanje posla koji se izvršava

```
void waitForJob(job* job){
    int cmdsCount=job->numSimpleCmds;
    int waitCount=0;
    pid_t waitPid;
    int status=0;
    do{
        waitPid=waitpid(-job->pgid,&status,WUNTRACED);
        waitCount++;
        if(WIFEXITED(status)){
            setCmdStatus(job,waitPid,COMPLETED);
        }
        else if(WSTOPSIG(status)){
                setCmdStatus(job,waitPid,STOPPED);
        }
        else if(WIFSIGNALED(status)){
                setCmdStatus(job,waitPid,TERMINATED);
        }

    }while(waitCount<cmdsCount);
    if(jobCompleted(job)){
        deleteJob(job->id);
    }
}
```

Prilog P.3.8. Programski kod funkcije kojom se brišu izvršene i prekinute naredbe

```
void deleteCompletedJobs(){
    job* j,*jnext;
    for(j=first_job;j!=NULL;j=jnext){
        jnext=j->next;
        if(jobCompleted(j)){
                    printJobInfo(j,"COMPLETED");
                    jnext=j->next;
                    deleteJob(j->id);
                }
        else if(jobTerminated(j)){
            printJobInfo(j,"TERMINATED");
            jnext=j->next;
            deleteJob(j->id);
        }
        else if(jobStopped(j) && j->notified==0){
            j->notified=1;
            printJobInfo(j,"STOPPED");
        }
    }
}
```

Prilog P.3.9. Programski kod implementiranih ugrađenih naredbi

```
int builtins(char*** cmds){
    if(!strcmp(cmds[0][0],"exit")){
            freeVars();
            freeHistory();
            return 0;
    }
    else if(isVar(cmds)){
            return -1;
    }
    else if(!strcmp(cmds[0][0],"history")){
            HIST_ENTRY **list;
            int i;
            list=history_list();
            if(list){
                for (i=0;list[i];i++){
                     printf ("%d: %s\n", i+history_base,list[i]->line);
                }
            }
        return -1;
    }
    else if(!strcmp(cmds[0][0],"kill") && (cmds[0][1][0]=='%')){
                int jobId=atoi(cmds[0][1]+1);
                pid_t pgid= getpgId(jobId);
                job *j=getJobById(jobId);
                if(!j){
                    return -1;
                }
                kill(-pgid,SIGKILL);
                return -1;
            }
    else if(!strcmp(cmds[0][0],"cd") ){
                if(chdir(cmds[0][1])<0){
                    perror("directory not found");
                }
                return -1;
            }
    else if(!strcmp(cmds[0][0],"fg") && cmds[0][1] &&( cmds[0][1][0]=='%')){
                int jobId=atoi(cmds[0][1]+1);
```

```
                job *j=getJobById(jobId);
                if(!j){
                    return -1;
                }
                printf("JOB [%d] IN FOREGROUND\n",jobId);
                put_in_fg(j,1);
                return -1;
            }
    else if(!strcmp(cmds[0][0],"bg") && cmds[0][1] && (cmds[0][1][0]=='%')){
                int jobId=atoi(cmds[0][1]+1);
                job *j=getJobById(jobId);
                if(!j){
                    return -1;
                }
                printf("JOB [%d] IN BACKGROUND\n",jobId);
                put_in_bg(j,1);
                return -1;
            }
    else if(!strcmp(cmds[0][0],"jobs")){
                printfjobs();
                return -1;
            }
    return 1;
}
```

Prilog P.3.10. Programski kod funkcije koja dodaje varijable u listu struktura varijabli

```
void addVariable(char *name,char *value,int env){
    var *new_var=(var*)malloc(sizeof(var));
    var *last=first_var;
    char *name_=malloc(strlen(name)+1);
    char *value_=malloc(strlen(value)+1);
    strcpy(name_,name);
    strcpy(value_,value);

    new_var->name=name_;
    new_var->value=value_;
    new_var->next=NULL;
    new_var->env=env;
    if(first_var==NULL){
        first_var=new_var;
        return;
    }
    while(last->next!=NULL){
        last=last->next;
    }
    last->next=new_var;
    return;
}
```

Prilog P.3.11. Programski kod koji obavlja inicijalizaciju varijabli i ažuriranje vrijednosti varijable

```
int isVar(char ***cmds){
    int a,b,f=0,upper;
    for(a=0;cmds[a]!=NULL;a++){
        for(b=0;cmds[a][b]!=NULL;b++){
            char* e=strchr(cmds[a][b],'=');
            if(e){
                f=1;
```

```
                int len=e-cmds[a][b];
                char name[len+1];
                name[len]='\0';
                strncpy(name,cmds[a][b],len);
                char *val=e+1;
                var* var;
                if(var=findVar(name)){
                    val=malloc(strlen(e+1)+1);
                    strcpy(val,e+1);
                    var->value=val;
                    if(var->env==1){
                        setenv(var->name,var->value,1);
                    }
                }else{
                    upper=checkUpperLetters(name);
                    if(upper){
                        addVariable(name,val,1);
                        setenv(name,val,1);
                    }else{
                        addVariable(name,val,0);
                    }
                }

            }
        }
    }
    if(f==0){
        return 0;
    }else{
        return 1;
    }
}
```

Prilog P.3.12. Programski kod funkcije kojom je implementirano proširenje varijabli

```
void checkVars(char ***cmds){
             int a,b;
            for(a=0;cmds[a]!=NULL;a++){
                for(b=0;cmds[a][b]!=NULL;b++){
                    char *p=strchr(cmds[a][b],'{');
                    char *p2=strchr(cmds[a][b],'}');
                    if(p && p2){
                        int nameLen=p2-p-1;
                        int len=p-cmds[a][b];
                        char prijeStr[len+1];
                        strncpy(prijeStr,cmds[a][b],len);
                        prijeStr[len]='\0';
                        char name[nameLen+1];
                        strncpy(name,p+1,nameLen);
                        name[nameLen]='\0';
                        int poslijeLen=strlen(p2+1);
                        char poslijeStr[poslijeLen+1];
                        strncpy(poslijeStr,p2+1,poslijeLen);
                        poslijeStr[poslijeLen]='\0';
                        char* val=getVarValue(name);
                        if(val){
                                cmds[a][b]=(char*)realloc(cmds[a][b],strlen(val)
                                          +strlen(prijeStr)+strlen(poslijeStr)+1);
                                strcpy(cmds[a][b],prijeStr);
                                strcat(cmds[a][b],val);
                                strcat(cmds[a][b],poslijeStr);
}}}}}
```

Prilog P.3.13. Programski kod funkcije kojom je implementirano proširenje naziva datoteke

```
void globbing(char ***cmds){
    int max_len=0;
    int a,b,j;
    int br=0;
    for(a=0;cmds[a]!=NULL;a++){
        for(b=0;cmds[a][b]!=NULL;b++){
                glob_t globBuff;
                int globCount=0;
                if(strchr(cmds[a][b],'?')!=NULL || strchr(cmds[a][b],'*')!=NULL ){
                        glob(cmds[a][b],GLOB_ERR ,NULL,&globBuff);
                        globCount=globBuff.gl_pathc;
                }
                if(globCount>0){
                    br=b;
                    max_len=b+globCount+1;
                    cmds[a]=(char**)realloc(cmds[a],max_len*sizeof(char*));
                    for(j=0;j<globCount;j++){
                        cmds[a][br]=strdup(globBuff.gl_pathv[j]);
                        br++;
                    }
                    globfree(&globBuff);
                    cmds[a][br]=NULL;
                    br=0;
                }
        }
    }
}
```

Prilog P.3.14. Uspostava serijske komunikacije i ispis poruka *bootloadera*

```
debian@debian:/dev$ sudo picocom -b 115200 /dev/ttyUSB0
[sudo] password for debian:
picocom v3.1

port is        : /dev/ttyUSB0
flowcontrol    : none
baudrate is    : 115200
parity is      : none
databits are   : 8
stopbits are   : 1
escape is      : C-a
local echo is  : no
noinit is      : no
noreset is     : no
hangup is      : no
nolock is      : no
send_cmd is    : sz -vv
receive_cmd is : rz -vv -E
imap is        :
omap is        :
emap is        : crcrlf,delbs,
logfile is     : none
initstring     : none
exit_after is  : not set
exit is        : no

Type [C-a] [C-h] to see available commands
Terminal ready


U-Boot 2018.09 (Apr 06 2023 - 14:36:34 +0200)

DRAM:  924 MiB
RPI 3 Model B (0xa02082)
MMC:   mmc@7e202000: 0, sdhci@7e300000: 1
Loading Environment from FAT... OK
In:    serial
Out:   vidconsole
Err:   vidconsole
Net:   No ethernet found.
starting USB...
USB0:   scanning bus 0 for devices... 3 USB Device(s) found
       scanning usb for storage devices... 0 Storage Device(s) found
Hit any key to stop autoboot:  0
U-Boot>
U-Boot>
```

Prilog P.3.15. Prijava u sustav na Raspberry Pi računalu te primjer korištenja naredbi datotečnog sustava

```
Welcome to Buildroot
buildroot login: root
buildroot:~# ls
armMeasure    file2         readtime.txt  shell-his
exectime.txt  hello-arm     shCatFile     shell-hist
file1         pid.c         shell-arm
buildroot:~# echo Hello
Hello
buildroot:~#
```