Turinys

Lentelių sąrašas

1 UNIX praktika

1.1 Praktikos pagrindai

1.1.1 Prisijungimas

Prisijungus prie UNIX mašinos vartotojas privalo įvesti savo ID (username) ir slaptažodį. Vartotojo vardas yra unikalus duotajai sistemai (arba sistemų grupei), o slaptažodis yra keičiamas simbolių rinkinys, žinomas vien tik vartotojui. UNIX yra svarbios didžiosios ir mažosios raidės.

1.1.2 Terminalo tipas

Visose sistemose (dažniausiai) terminalo tipas yra užduodamas pagal nutylėjimą. Tai dažniausiai yra **vt100** terminalas. Sun mašinos gali naudoti **sun** terminalą. Jei naudojamas X-Terminalas – **xterms** arba **xterm**. Terminalo tipas nurodo UNIX sistemai kaip elgtis su duotąja sesija. Terminalą galime pakeisti, pakeičiant aplinkos kintamąjį, pvz.

```
% TERM=<terminalo_tipas>
```

arba

```
% setenv TERM <terminalo_tipas>
```

1.1.3 Slaptažodis

Kai jums yra sukuriamas vartotojas ir suteikiamas slaptažodis, reikėtų jį pasikeisti. Tai yra svarbu sistemos ir jūsų informacijos saugumui. Slaptažodis yra pakeičiamas naudojant passwd komandą. Reikės įvesti seną ir du kartus naują slaptažodį. Kai kuriais atvejais sistemų administratoriai naudoja specialias programas, kurios patikrina ar įvestasis slaptažodis yra pakankamai saugus. Štai kelios slaptažodžių taisyklės:

Nenaudokite: pilnų bet kokios kalbos žodžių, vardų, informacijos, kurią galima surasti jūsų piniginėje, asmeninės informacijos (paso numerio ir pan.), valdymo klavišų (kai kurios sistemos jų nepalaiko). Nerašykite niekur savo slaptažodžio ir niekad niekam jo nesakykite.

Slaptažodis turėtų būti sudarytas iš raidžių ir skaičių kratinio, įvairaus registro raidžių (didžiosios/mažosios), iš ne mažiau kaip 6 simbolių. Slaptažodį jūs turėtumėte gerai atsiminti. Jis turėtų būti dažnai keičiamas. Kai įvedate slaptažodį sekite ar niekas nežiūri per petį.

1.1.4 Darbo pabaiga

^D – žymi duomenų srauto pabaigą; gali baigti vartotojo darbą shell'e;

^C – įsiterpti į darbą;

^Z – sustabdyti darbą;

`% logout` – išeiti iš sistemos;

`% exit` – išeiti iš shell'o.

1.1.5 Identifikacija

Vartotoją sistemoje identifikuoja vartotojo (userid) ir jo grupės (groupid) numeriai. Juos suteikia sistemos administratorius. Vartotojas gali būti priskirtas kelioms grupėms, tačiau viena iš jų yra pirminė. Naudojant komandą id galima sužinoti savo identifikacinius numerius:

```
% id
uid=1101 (jonas) gid=10 (staff)
```

Kai kurios sistemos pateikia ir papildomas grupes

```
% id
```

```
uid=1101 (jonas) gid=10 (staff) groups=10 (staff), 11 (developers), 12 (sysadmin)
```

Komanda groups pateikia grupių informaciją

```
% groups
staff developers sysadmin
```

1.1.6 UNIX komandinės eilutės struktūra

UNIX komandos turi sekančią struktūrą:

```
% komanda [opcijos] [argumentai]
```

, kur argumentas nurodo objektą su kuriuo tiesiogiai manipuliuoja komanda, pvz. failas ar failai. Opcijos modifikuoja komandą. Komandos reaguoja į raidžių registrą.

Opcijos dažniausiai prasideda brūkšniu (-) ir, daugumoje komandų opcijos gali būti sujungtos, pvz.

```
% ls –alR
```

yra tas pats kaip ir

```
% ls –a –l –R
```

ir ši komanda suformuos ilgą failų bei katalogų sąrašą bei rekursyviai pateiks katalogų turinį.

Kai kurios opcijos reikalauja parametrų, tarkime

```
% lpr –Plaser1 –# 2 failas
```

komanda atspausdina printeryje laser1 dvi failo *failas* kopijas.

1.1.7 Kontroliniai klavišai

Kontroliniai klavišai yra naudojami atlikti specialias funkcijas. Jie surenkami naudojant Ctrl klavišą. Kontroliniai klavišai yra žymimi ^Klavišas, pvz. ^S yra stop signalas, kuris sustabdo informacijos iš terminalo priėmimą. Kad terminalą vėl aktyvuoti yra surenkama ^Q kombinacija. ^U ištrina įvestą eilutės komandą (line-kill komanda).

1.1.8 Terminalo valdymas

stty komanda pateikia ir keičia terminalo valdymo opcijas. Paprastiems vartotojams svarbiausia stty komanda yra trynimo klavišo nustatymas. Su šia komanda taip pat galima nustatyti line-kill klavišą, duomenų perdavimo greitį, TAB klavišo interpretacija, jautrumą į simbolių registrą ir pan. stty sintaksė yra paprasta:

```
% stty [opcijos]
```

, kur opcijos gali būti

(nėra opcijų) – išvesti į stdout pagrindinius terminalo parametrus,

all (-a) – išvesti į stdout visus terminalo parametrus,

kill – nustatyti line-kill klavišą,

erase – nustatyti trynimo klavišą,

intr – nustatyti interupt klavišą ir kt.

Tarkime, norit pakeisti trynimo klavišą iš ^? (DEL) į ^H, reikia surinkti:

```
% stty erase ^H
```

Norint, kad terminalo opcijos butų nustatomos pastoviai, reikia įterpti stty komandos eilutes į .profile (.cshrc ar .login) failus.

1.1.9 Pagalba UNIX sistemoje

UNIX pagalba yra vadinama man puslapiai (manual). Ji suteikia informaciją apie komandų sintaksę ir sistemą aplamai. Sintaksė:

```
% man [opcijos] komanda
```

Visa pagalbinė medžiaga yra suskirstyta į skyrius, kuriems yra priskiriami numeriai, pvz.

```
% man –k password
passwd (5) – password file
passwd (1) – change password information
```

Pagal nutylėjimą sistema pateikia skyriaus su mažiausiu numeriu puslapį, tačiau naudojant komandos opcijas galima pasirinkti kurį norite. Norėdami sužinoti kaip naudotis `man` komanda surinkite

```
% man man
```

Skyrių numeracija įvairioms sistemoms skiriasi, tačiau žemiau lentelėje yra pateikti tradicinių skyrių turinys ir numeris UNIX BSD ir SV sistemoms.

8.1 lentelė. Pagrindiniai man skyriai BSD ir System V sistemose.

Skyriaus turinys	**BSD**	**System V**
Pagalbinės sistemos programos (utilites)	1	1
Sisteminės komandos	2	2
Bibliotekų funkcijos	3	3
Specialūs failai, įrenginių tvarkyklės ir aparatūra	4	7
Konfigūracinių ir sisteminių failų formatai	5	4
Viskas, kas netilpo į kitus skyrius	7	5
Administracinės programos	8	1M

1.1.10 Pagrindinės komandos

1.1.10.1 Katalogų naršymas ir valdymas

UNIX failų sistema yra medžio struktūros su šaknimi (root; /) viršuje. Kiekvienas vartotojas turi savo namus (home; ~) į kuriuos patenka tik prisijungus prie sistemos. Dažniausiai vartotojai savo failus ir katalogus kuria bei saugo namų kataloguose.

8.2 lentelė. Katalogų naršymo ir valdymo komandos

Komanda / sintaksė	**Aprašymas**
cd [katalogas]	Pakeisti darbinį katalogą
ls [opcijos] [katalogas ar failas]	Išvesti katalogo turinį arba failo aprašymą į stdout
mkdir [options] katalogas	Sukurti katalogą
pwd	Išvesti darbinį katalogą į stdout
rmdir [options] katalogas	Ištrinti katalogą

1.1.10.2 Failų valdymo komandos

8.3 lentelė. Failų valdymo komandos

Komanda / sintaksė	**Aprašymas**
chgrp [options] group file	Pakeisti failo grupę
chmod [options] file	Pakeisti failo/katalogo modą (leidimus)
chown [options] owner file	Pakeisti failo savininką
cp [options] file1 file2	Kopijuoti file1 į file2
mv [options] file1 file2	Perkelti/pervadinti file1 į file2
rm [options] file	Trinti failą arba katalogą

1.1.10.3 Išvedimo komandos

8.4 lentelė. Išvedimo komandos

Komanda / sintaksė	**Aprašymas**

cat [options] file	Sujungti (išvesti į stdout) failą
echo [text string]	Išvesti teksto eilutę į stdout
head [-number] file	Išvesti pirmas 10 (arba nurodytą skaičių) eilučių į stdout
more (pg, less) [options] file	Puslapiuoti failą į stdout
tail [-number] file	Išvesti paskutines 10 (arba nurodytą skaičių) eilučių į stdout

1.1.10.4 Sistemos resursai

8.5 lentelė. Sistemos resursų informacijos ir valdymo komandos.

Komanda / sintaksė	Aprašymas
chsh username shell	Pakeisti vartotojo startinį shell'ą
date [options]	Išvesti dabartinę datą ir laiką
df [options] [resource]	Ataskaita apie panaudotus ir laisvus diskų blokus
du [options] [directory or file]	Ataskaita apie naudojamą vietą diske
hostname / uname	Išvesti mašinos (host) vardą, sistemos versiją ir pan.
kill [options] [-signal] [pid] [job]	Procesui (pid) arba darbui (job) siunčia signalą (signal)
passwd [options]	Įvesti ar pakeisti slaptažodį
ps [options]	Išvesti aktyvių procesų informaciją
script file	Viską, kas pasirodo ekrane surašo į failą file
whereis [options] command	Pateikia komandos binarinį, išeities ir man failus
which command	Pateikia pilną komandos command failo kelią
who / w	išveda informacija apie aktyvius vartotojus

1.1.10.5 Spausdinimas

8.6 lentelė. Spausdinimo komandos.

Komanda / sintaksė	Aprašymas
lpq / lpstat [options]	Išvesti informaciją apie spausdinamus darbus
lpr / lp [options] file	Spausdinti į nurodytą printerį
lprm / cancel [options]	Nutraukti spausdinimą arba išmesti darbą iš eilės
pr [options] [file]	Suformuoti failą spausdinimui

1.2 I/O valdymas

1.2.1 Failų deskriptoriai

8.7 lentelė. Failų deskriptoriai UNIX sistemose.

Kodas	Pavadinimas	Sutrumpinimas	Pagal nutylėjimą
0	Standartinis įvedimas	stdin	Klaviatūra
1	Standartinis išvedimas	stdout	Terminalas
2	Standartinė klaida	stderr	Terminalas

1.2.2 Failų (išvedimo / įvedimo) nukreipimas

1.2.2.1 /bin/sh

Standartiniai I/O įrenginiai gali būti pakeisti naudojant nukreipimo techniką.

8.8 lentelė. /bin/sh I/O nukreipimas.

Sintaksė	Aprašymas
cmd > file	Nukreipti cmd rezultatą į failą (perrašyti failą)
cmd >> file	Nukreipti cmd rezultatą į failą (prijungti į failą)
cmd < file	Paimti cmd argumentą iš failo
cmd << text	Skaityti įvedimą iki kol bus įvestas text
cmd >&n	Nukreipti komandos rezultatą į n deskriptorių
cmd m>&n	Nukreipti komandos rezultatą, kuris eitų į m, į n deskriptorių
cmd >&-	Uždaryti standartinį išvedimą
cmd <&n	Komandai cmd argumentus imti iš n deskriptoriaus

cmd m<&n	Komandai cmd argumentus, kuri normaliai imtų iš m, imti iš n deskriptoriaus
cmd <&-	Uždaryti standartinį įvedimą

Pavyzdžiai:

```
$ make mano.c 2>klaidos
$ make mano.c >pranesimai 2>&1
$ (make mano.c >gerai) 2>blogai
$ who | tee vartotojai
$ mail jonas <report

$ sed 's/^/> /g' <<GALAS
> tai
> yra mano
> tekstas
GALAS
> tai
> yra mano
> tekstas

$ echo "Klaida: kreipkis i admin" 1>&2
$ (find / -print > filelist) 2>no_access
```

1.2.2.2 /bin/csh

Keletas csh savybių:

`>& file`	permesti stdio ir stderr į failą file
`>>& file`	prijungti stdio ir stderr į failą file
`|& command`	sujungti į kanalą sdtio ir stderr komandai command

Norint permesti stdio ir stderr į skirtingus failus reikia daryti taip:

```
% (find / >found) >& no_access
```

1.2.3 Manipuliacijos komandine eilute:

`$ find / &`	Vykdyti komandą foniniame režime
`$ cd; ls`	Vykdyti komandas vieną po kitos
`$ (date;who;pwd)>logfile`	Komandų grupė
`$ sort f1|pr –3|lp`	Komandų sujungimas – kanalas (pipe)
`$ mail jan `ls ~``	Komandos argumentas – kitos komandos išvedimas
`$ find / -name gcc && lp`	Jei viena komanda pavyko, vykdyti ir kitą
`$ find / -name gcc || echo Ner`	Jei viena komanda nepavyko, vykdyti kitą

1.3 Teksto apdorojimas

1.3.1 Reguliarieji išsireiškimai

Daugelis UNIX komandų ir pagalbinių programų leidžia naudoti reguliariuosius išsireiškimus. Tačiau yra ir skirtumų, tarkim ? reguliariajame reiškinyje reiškia vieną reiškinį, o failų paieškoje – vieną simbolį. Reguliariuosius išsireiškimus sudaro paprastieji ir specialūs (meta-) simboliai.

Meta-simboliai

`.`	Vienas bet koks simbolis
`*`	Bet koks skaičius prieš tai buvusių simbolių arba nieko
`^`	Eilutės pradžia
`$`	Eilutės pabaiga
`[ ]`	Bent vienas iš apskliaustų simbolių. – reiškia seką; pradžioje esantis ^ reiškia priešingą reikšmę; pirmasis ^ arba – reiškia simbolį sekoje.
`\{n,m\}`	Vienodų, prieš tai einančių simbolių skaičius. \{n\} - tiksliai n simbolių; \{n,\} – bent n simbolių; \{n,m\} – tarp n ir m simbolių.

\	Po to sekantis specialusis simbolis tampa paprastuoju.
\(\)	Apskliaustą pavyzdį užsaugo buferyje. Į buferius galima kreiptis naudojant \1 - \9.
\< \>	Žodis
+	Vienas ar daugiau prieš tai einančių simbolių
?	Vienas ar jokio prieš tai ėjusio išsireiškimo
|	Prieš arba po to einantis išsireiškimas.
( )	Išsireiškimų grupavimas

Paieškos pavyzdžiai:

```
$ grep ‘lab’
$ grep ‘^lab’
$ grep ‘lab$’
$ grep ‘^lab$’
$ grep ‘[Ll]ab’
$ grep ‘l[aeo]b’
$ grep ‘l[^aeo]b’
$ grep ‘l.b’
$ grep ‘^...$’
$ grep ‘^\.’
$ grep ‘^\.[a-z][a-z]’
$ grep ‘^[^.]’
$ grep ‘labas*’
$ grep ‘”labas”‘
$ grep ‘”*labas”*’
$ grep ‘[A-Z][A-Z]*’
$ grep ‘[A-Z].*’
$ grep ‘[A-Z]*’
$ grep ‘[a-zA-Z]’
$ grep ‘[^0-9a-zA-Z]’
```

Pakeitimo/modifikavimo pavyzdžiai:

```
$ sed s/.*/( & )/
$ sed s/.*/mv & &.old/
$ sed /^$/d
$ sed /^[ \t]*$/d
$ sed s/\s\s*/\s/g
$ sed s/[0-9]/Item &:/
$ sed s/\<.*for.*\>/\U&/g
$ sed s/.*/\L&/
$ sed s/\<./\u&/g
$ sed s/yes/no/g
$ sed s/die or do/do or die/
$ sed s/\([Dd]ie\) or \(Dd\)o/\2 or \1/
```

1.4 grep komanda

```
$ grep [options] regexp file
```

Ieško failuose file reguliariųjų išsireiškimų regexp. Grąžina 0 jei rasta nors viena eilutė, 1 – jei nerasta, 2 – jei kilo klaida. Keletas naudingų opcijų:

c – spausdinti tik atitikusių eilučių skaičių;

h – spausdinti tik atitikusias eilutes, be failų pavadinimų;

i – ignoruoti didžiąsias ir mažąsias raides;

l – spausdinti tik failų vardus, bet ne atitikusias eilutes;

n – spausdinti eilutes ir jų numerius;

v – spausdinti tik neatitikusias eilutes.

Pavyzdžiai:

```
$ grep –c /bin/csh /etc/passwd
$ grep –l ‘^#include’ ~/src/*
```

1.4.1 sed komanda

```
$ sed [opcijos] ‘komanda’ failas
```

Tai yra teksto paieškos ir modifikavimo komanda. Ši programa kiekvienai failo eilutei pritaiko pateiktą komanda. sed komandos struktūra:

[adresas] [,adresas] [!] komanda [argumentai]

Viena svarbiausių yra eilutės modifikavimo funkcija:

[adresas] [,adresas] s /ka pakeisti/kuo pakeisti/[opcijos]

Pavyzdžiai:

```
$ sed s/sveix/sveikas/g
$ sed /BSD/d
$ sed /BSD/!d
$ sed /^BEGIN/,/^END/p
$ sed /^BEGIN/,/^END/!s/sveix/sveikas/g
$ sed /function/{ s/”/(/3 s/”/)/4 }
$ sed /Title/s/”//g
$ sed { s/:://p s/”//gp}
$ sed /ifdef/!s/if/\tif/
```

1.4.2 awk/nawk/gawk komanda

Tai teksto skanavimo ir apdorojimo programa. Sintaksė:

```
$ awk pattern{action} [file]
```

Įvedimas yra suskirstytas į įrašus – eilutės ir laukus – simbolių grupės atskirtos tarpu arba TAB’u. Laukų skirtukus galima keisti su kintamuoju NF. Kintamasis $n žymi n-tąjį lauką, o $0 žymi visą įrašą. Eilutės BEGIN ir END žymi viso įvedimo pradžią ir pabaigą. Spausdinimas atliekamas su print ir formatuotu printf komandomis (kaip ir C kalboje).

Ieškomą pavyzdį gali sudaryti keli reguliarūs išsireiškimai ir kombinuojami naudojant skirtukus: || - arba, && - ir, ! – ne. Kabutėmis atskirti išsireiškimai žymi paieškos pradžią ir pabaigą, pvz. /pirmas/,/paskutinis/. Norint parinkti eilutes nuo 15 iki 20 reikia rašyti: NR=15, NR=20.

Atitikimas reguliariam išsireiškimui yra išreiškiamas ~ - atitinka išsireiškimą, !~ - neatitinka išsireiškimo, pvz.:

```
$1 ~ /[Ll]abas/
```

programa yra true, jei pirmajame lauke bet kurioje vietoje yra žodis “labas”. Jei pirmasis laukas turi turi tiesiogiai atitikti žodį ‘Labas’:

```
$1 ~ /^[Ll]abas$/
```

Kai kurios built-in funkcijos:

```
index(s,t) – gr•žina t pozicij• s eilut•je;
length(s) – gr•žina s ilg•;
substr(s,m,n) – gr•žina eilut•s s (m,n) dal•.
```

Valdymo sakiniai (C stiliumi):

```
for(i=1;i<=$1;i++){ veiksmai }
while(i<=$1){ veiksmai }
if(i<10){ veiksmai }
```

Sisteminiai kintamieji:

FILENAME – failo vardas;

FS – lauko skirtukas;

NF – laukų skaičius įraše;

NR – einamojo įrašo numeris;

OFS – išvedimo lauko skirtukas;

ORS – išvedimo įrašų skirtukas;

$0 – visas įrašas;

$n – n-tasis laukas.

Pavyzdžiai:

```
$ awk {print $1}
$ awk /labas/ {print $0}
$ awk NF=2 {print $1+$2 }
$ awk BEGIN { FS="\n"; RS=""}
$ awk $1~/labas/ {print $3, $2}
$ awk –F:{printf("Eilutes Nr.%s suma yra %d\n",NR,$1+$2)}
$ awk /labas/ {++x}; END {print x}
$ awk {total+=$2}; END {print "viso",total}
$ awk length<20
$ awk NF=7 && /^Name:/
$ awk { for(i=NF;i>=1;i--) print $i }
```

1.5 Kitos naudingos komandos

1.5.1 Darbas su failais

8.9 lentelė. Papildomos darbo su failais komandos.

Komanda / sintaksė	Aprašymas
cmp [options] file1 file2	Lygina failus ir išveda skirtumus (text ir bin failai)
diff [options] file1 file2	Lygina failus ir išveda skirtumus (text failai)
cut [options] [files]	Iškerpa failo laukus
paste [options] file1 file2	Sujungia failų laukus
touch [options] file	Sukuria failą/pakeičia jo modifikavimo datą
wc [options] file	Skaičiuoja failo simbolius/žodžius/eilutes
ln [options] source target	Sukuria nuorodą į failą – suteikia failui kitą vardą
sort [options] file	Rūšiuoja failo turinį
tee [options] [failas]	Kopijuoja išvedimą į failą
uniq [options] file [file.new]	Filtruoja vienodas failo eilutes
strings [options] file	Ieško binariniame faile ASCII eilučių
file [options] file	Grąžina failo tipą
tr [options] string1 [string2]	Transliuoja simbolius iš stdin į stdout
find catalog [options] [cmd]	Rekursyviai ieško failų

1.5.2 Failų archyvavimas ir suspaudimas

8.10 lentelė. Failų archyvavimo ir suspaudimo komandos.

Komanda / sintaksė	Aprašymas
compress [options] file1 file2	Suspaudžia failus ir jie tampa .Z failais
uncompress [options] file1 file2	Iškompresuoja .Z failus
gzip/unzip [options] file	Failų archyvacija .gz
tar [opcijos] file	Failų archyvacija .tar

1.6 Komandų interpretatoriai – shells

UNIX shell – tai komandų interpretatoriai. Pirmasis shell'as buvo **sh** – Bourne shell. Vėliau buvo sukurtas C shell – **csh**, kurį dabar galime rasti daugelyje, bet ne visose UNIX operacinėse sistemose. Pagal nutylėjimą sh įvedimo simbolis yra $ (# - root'o shell'ui), csh - %. Taip pat yra kitokių shell'ų – Korn shell'as (ksh), bash iš GNU, T-C shell – tcsh, išplėstas C shell'as – cshe.

Mes mokysimės sh – Bourne shell'ą ir csh - C shellą.

1.6.1 Aplinkos kintamieji

1.6.1.1 /bin/sh

Pagrindiniai aplinkos kintamieji: DISPLAY, EDITOR, GROUP, HOME, HOST, IFS, LOGNAME, PATH (atskirti dvitaškiais), PS1, PS2, SHELL, TERM, USER.

Aplinkos kintamieji priskiriami taip:

```
$ NAME=value; export NAME
```

Startinis sh failas yra `/etc/profile` visiems ir `$HOME/.profile`.

1.6.1.2 /bin/csh

Pagrindiniai aplinkos kintamieji: argv, cwd, history, home, ignoreeof, noclumber, noglob, path, prompt, savehist, shell, status, term, user.

Aplinkos kintamieji priskiriami štai kaip:

```
% set NAME=(value1 value2)
```

csh visiems vartotojams startuoja kokį nors `/etc/csh.login, /etc/cshrc` arba `/etc/login` failą. Kiekvienam vartotojui yra startuojami `~/.cshrc`, poto `~/.login` failai.

1.7 Shell programavimas

1.7.1 Shell programos (skriptai)

1.7.1.1 /bin/sh

Shell skriptas – tai failas, kuriame yra saugomos shell komandos. Pirmoji eilutė shell skripte prasideda #! po kurių eina programa – interpretatorius, pvz:

```
#! /bin/sh
```

Norint vykdyti komandą reikia pakeisti jos režimą

```
$ chmod +x shell_script
```

1.7.1.2 /bin/csh

Programos interpretatoriaus eilutė yra ši:

```
#! /bin/csh
```

1.7.2 Parametrų reikšmės

1.7.2.1 /bin/sh

Parametrų reikšmės priskiriamos paprastai:

```
param=value
```

Jei reikšmei naudosime `` kabutes, pradžioje eilutė bus interpretuojama, pvz.

```
day = `date +%a`
echo $day
```

Kai parametrui yra priskiriama reikšmė, jis gali būti naudojamas `$param` arba `${param}`.

Jei su parametrais yra atliekama aritmetinė operacija, reikia naudoti `let` komandą:

```
x=5;
y=`let $x + 6`
# arba
y=$(($x+6))
```

1.7.2.2 /bin/csh

Parametrai sukuriami ir pasiekiami taip pat kaip ir sh shell'e. Aritmetinėms operacijoms nereikia papildomų funkcijų. jei norite įsitikinti, kad interpretatorius su kintamuoju elgsis kaip su skaičiais, atlikite sekančią operaciją:

```
x=5; y=6
z=$x+$y+0
```

1.7.3 Kintamieji

1.7.3.1 /bin/sh

8.11 lentelė. /bin/sh sistemoje apibrėžti kintamieji

Kintamasis	Reikšmė
$#	Komandinės eilutės argumentų skaičius
$-	Shell opcijos
$?	Paskutinės įvykdytos komandos grąžinta reikšmė
$$	Einamojo proceso PID
$!	Paskutiniojo, fone įvykdyto, proceso numeris
$n	n-tasis argumentas
$0	Shell skripto pavadinimas
$*	Visi argumentai
$@	Visi argumentai kabutėse

1.7.3.2 /bin/csh

8.12 lentelė. /bin/csh sistemoje apibrėžti kintamieji

Kintamasis	Reikšmė
${var}	var kintamojo reikšmė
${var[i]}	i-tasis kintamojo žodis
${#var}	Žodžių var eilutėje skaičius
${#argv}	Argumentų skaičius
$0	Programos pavadinimas
${#argv[n]}	n-tasis argumentas
${n}	n-tasis argumentas
${#argv[*]}	Visi komandos argumentai
${*}	Visi komandos argumentai
${argv[$#argv]}	Paskutinis argumentas
${?var}	Grąžina 1, jei var yra ne NULL ir 0 kitaip
$$	Einamojo shell'o numeris

1.7.4 Parametro reikšmių naudojimas/keitimas

8.13 lentelė. kintamųjų reikšmių keitimas.

Operacija	Reikšmė
$param	Pakeičiama parametro reikšme
${param}	Pakeičiama parametro reikšme
$param=	Reikšmė tampa NULL
${param-def}	Jei param=NULL, naudojame def
${param=def}	Jei param=NULL, jam priskiriame def ir naudojame param
${param+val}	Jei param!=NULL, naudoti val. param lieka tas pats
${param?msg}	Jei param=NULL, išvedame pranešimą msg

1.7.5 Here Document struktūra

```
#! /bin/sh
labas=sveikas gyvas
cat <<EOF
Mano
mielas drauge
```

```
$labas
EOF
cat <<EOF
Mano
mielas drauge
$labas
EOF
```

1.7.6 Interaktyvus įvedimas

1.7.6.1 /bin/sh

```
#! /bin/sh
echo “Iveskite fraze: \c”
read fraze
echo fraze=$fraze
```

1.7.6.2 /bin/csh

```
#! /bin/csh -f
echo –n “Iveskite fraze: ”
set fraze=$<
echo fraze=$fraze
```

1.7.7 Funkcijos

```
ll(){ ls –la “$@”; }
```

1.7.8 Valdymo sakiniai

1.7.8.1 Sąlygos sakiniai

1.7.8.1.1 /bin/sh

```
if [ $# -ge 2 ] then
       echo $2
elif [ $# -eq 1 ]; then
       echo $1
else
       echo No input
fi

case $1 in
aa|ab) echo A;;
b?) echo B;;
c*) echo C;;
*) echo D;;
esac
```

1.7.8.1.2 /bin/csh

```
if($#argv >= 2); then
       echo $2
else if($#argv == 1); then
       echo $1
else
       echo No Input
endif

switch($1)
       case aa:
       case ab:
              echo A
              breaksw
       case b?:
              echo B
              breaksw
       case c*:
              echo C
              breaksw
       default:
```

```
            echo D
endsw
```

1.7.8.2 Ciklo sakiniai

1.7.8.2.1 /bin/sh

```
for file in *.old do
       newf = `basename $file .old`
       cp $file ${newf}.new
done

while [ $# -gt 0 ]; do
       echo $1
       shift
done

until [ $# -le 0 ]; do
       echo $1
       shift
done
```

1.7.8.2.2 /bin/csh

```
foreach file(*.old)
       set newf = `basename $file.old`
       cp $file $newf.new
end

while ($#argv!=0)
       echo $argv[1]
       shift
end
```

1.7.9 Loginiai ir veiksmų operatoriai

1.7.9.1 /bin/sh – test komanda

Test tikrina sąlygą. Ją pakeičia [salyga] skliaustai, kuriuose yra užrašoma salyga.

Test opcijos dirbant su failais (-option filename):

-r	true, jei failas yra ir yra skaitomas
-w	true, jei failas yra ir yra rašomas
-x	true, jei failas yra ir yra vykdomas
-f	true, jei failas yra ir yra paprastas failas (ne katalogas)
-d	true, jei failas yra ir yra katalogas
-h ar –L	true, jei failas yra ir yra simbolinė nuoroda
-c	true, jei failas yra ir yra simbolinis įrenginys
-b	true, jei failas yra ir yra blokinis įrenginys
-p	true, jei failas yra ir yra vardinis kanalas (fifo pipe)
-u	true, jei failas yra ir turi nustatytą setuid
-g	true, jei failas yra ir turi nustatytą setgid
-k	true, jei failas yra ir yra nustatytas lipnusis bitas
-s	true, jei failas yra ir jo dydis yra didesnis nei 0

Testai eilutėms:

-z string	true, jei string ilgis yra 0
-n string	true, jei string ilgis yra ne 0
string1=string2	true, jei string1 yra lygi string2
string1!=string2	true, jei string1 nėra lygi string2
string	true, jei string nėra NULL

Testai sveikiems skaičiams:

n1 –eg n2	true, jei n1 yra lygus n2
n1 –ne n2	true, jei n1 nėra lygus n2
n1 –gt n2	true, jei n1 didesnis nei n2
n1 –ge n2	true, jei n1 didesnis ar lygus n2
n1 –lt n2	true, jei n1 mažesnis už n2
n1 –le n2	true, jei n1 mažesnis arba lygus n2

Loginiai operatoriai:

!	unarinis ne
-a	and
-o	or
()	grupavimas

1.7.9.2 /bin/csh

(..)	išsireiškimų grupavimas
~	inversija
!	loginis neigimas
*, /, %	daugyba, dalyba, modulis
+, -	sudėtis, atimtis
<<, >>	binarinis postūmis į kairę, binarinis postūmis į dešinę
<=	mažiau arba lygu
>=	daugiau arba lygu
<, >	mažiau ir daugiau
==	lygu
!=	nelygu
=~	atitikti eilutę
!~	neatitikti eilutės
&, ^, |	binariniai AND, XOR ir OR
&&	loginis AND
||	loginis OR
{komanda}	1, jei komanda grąžina 0, 1 – jei komanda grąžina ne 0

C shell'as taip pat kaip ir Bourne shell'as turi operatorius loginėms operacijoms susijusioms su failais:

-r	true, jei failas yra ir yra skaitomas
-w	true, jei failas yra ir yra rašomas
-x	true, jei failas yra ir yra vykdomas
-f	true, jei failas yra ir yra paprastas failas (ne katalogas)
-d	true, jei failas yra ir yra katalogas
-e	true, jei failas yra
-o	true, jei failas yra ir jo savininkas yra einamasis vartotojas
-z	true, jei failas yra ir jo dydis yra lygus 0

1.8 C programavimas

1.8.1 Kompiliavimas ir surišimas

Paprastai UNIX sistemos turi du C kompiliatorius – `cc(1)` ir `gcc(1)`. Kompiliatoriaus parametrai yra nurodomi `makefile` faile arba betarpiškai kompiliavimo komandinėje eilutėje. Kompiliatorius sukuria daug objektinių failų su standartiniu plėtiniu `.o`.

Vėliau objektiniai failai yra surišami su surišėju (linker), ld(1), kuris sujungia bibliotekas ir vykdomuosius objektinius failus sukurdamas vieną paleidimui skirtą failą. ld(1) dirba dviejuose režimuose:

1. statinis režimas – sukuriamas vienas failas, kuriame yra susiejami visi objektiniai failai ir statinės bibliotekos (`*.a`) į vieną vykdomąjį failą, kuriame yra visas programos vykdymui reikalingas kodas;
2. dinaminis režimas – ryšių redaktorius pagal galimybę susieja vykdomąjį failą su bendrai naudojamomis, dinaminėmis bibliotekomis (*.so). Tokiu atveju dinaminės bibliotekos yra prijungiamos programos darbo metu.

Abiem atvejais pagal nutylėjimą yra sukuriamas `a.out` vykdomasis failas. Paprastiems uždaviniams užtenka surinkti komandą

```
$ make prog
```

arba jos ekvivalentą

```
$ cc –o prog prog.c
```

, kurios sukurs vykdomąjį objektinį failą `prog`. Pažymėtina, kad komanda `cc` yra kompiliatoriaus ir ryšių redaktoriaus kombinacija, kurią ir rekomenduojama naudoti.

Pavyzdžiui, pradžioje sukompiliuosime du išeities teksto failus, o po to juos ir dar vieną biblioteką (`libnsl.a` arba `libnsl.so`) sujungsime į vieną vykdomąjį failą:

```
$ cc –c file1.c file2.c
$ cc –o prog file1.o file2.o –lnsl
```

1.8.2 Klaidų apdorojimas

Sisteminė komanda sėkmingo užduoties įvykdymo atveju grąžina 0 arba, jei tai funkcija, reikšmę; priešingu atveju –1. Ji taip pat nustato kintamąjį `errno` į t.t. reikšmę, kuri identifikuoja klaidos priežastį. Faile `<errno.h>` yra nustatytos galimos `errno` kintamojo reikšmės ir trumpas klaidos aprašymas. Bibliotekų funkcijos paprastai nenustato `errno` reikšmių ir grąžina įvairias reikšmes. Standartizuotas yra tik branduolio sisteminių komandų klaidų identifikavimo mechanizmas. Minėtas kintamasis yra nustatomas sekančiai:

```
external int errno;
```

Kadangi po sėkmingai įvykdytos komandos errno nėra nustatoma lygi nuliui, tai `errno` reikšmės tikrinimas turi prasmę tik klaidos atveju. ANSI C standartas numato dvi funkcijas, kurios leidžia sužinoti daugiau apie klaidą, t.y. `strerror(3C)` ir `perror(3C)`:

// Funkcija grąžina klaidos aprašyma remdamasi pateiktu klaidos kodu

```
#include <strings.h>
char *strerror(int errnum);
```

// Funkcija į stdout išveda klaidos aprašymą pagal errno reikšmę

// argumentu yra papildomos informacijos apie klaidą eilutė

```
#include <errno.h>
#include <stdio.h>
void perror(const char *s);
```

Funkcijų panaudojimo pavyzdys:

```
#include <errno.h>
#include <stdio.h>
#include <strings.h>
main(int argc, char *argv[]){
       fprintf(stderr, "ENOMEM: %s\n",strerror(ENOMEM));
       errno=ENOEXEC;
       perror(argv[0]);

}
```

Standartinės UNIX funkcijos taip pat naudojasi šiomis funkcijomis, pvz.

```
$ rm not_exist
not_exist: No such file or directory
```

1.8.3 main funkcija

POSIX.1 standartas nustato tokią įėjimo į programą, programos pradžios, funkciją

```
int main(int argc, char *argv[], char *envp[]);
```

, kur `argc` yra argumentų skaičius, `argv` – argumentų masyvas, `envp` – aplinkos kintamųjų masyvas. ANSI standartas nusako paprastesnę funkcijos antraštę:

```
int main(int argc, char *argv[]);
```

, o aplinkos kintamuosius siūlo paimti per globalų `environ` kintamąjį

```
extern char **environ;
```

Žemiau pateikiame C programos, atspausdinančios visus komandinės eilutės argumentus ir aplinkos kintamuosius pavyzdį:

```
#include <stddef.h>
extern char **environ;
int main(int argc, char *argv[]){
       int i;
       for(i=0; i<argc;i++) printf("argv[%d]=%s\n",i,argv[i]);
       i=0;
       while(environ[i]!=NULL) printf("environ[%d]=%s\n",i,environ[i++]);
       return 0;
}
```

Aplinkos kintamieji yra keičiami naudojant sekančias funkcijas:

```
#include <stdlib.h>
char *getenv(const char *name);
int putenv(const char *string);
```

Funkcija main sekmės atveju turi grąžinti 0, o nesėkmės !0. Pavyzdžiui komanda `grep` grąžina: 0 – rasta sutapimų, 1 – nerasta sutapimų ir 2 – sintaksinė klaida arba klaida skaitant failą.

Išeinama iš programos keliais būdais – iškvietus funkciją `return x;` funkcijos main kūne arba iškvietus funkciją `exit(x);` bet kurioje programos vietoje.

Jei norite programos pabaigoje įvykdyti tam tikras funkcijas, pvz. uždaryti atidarytus failus – iškvieskite funkciją `atexit(3C)`, kurios pagalba galima užregistruoti funkcijas, kurios bus iškviestos nutraukus programos darbą. Jos iškviečiamos LIFO principu (Last Input First Output), pvz.:

```
#include <stdlib.h>
#include <stdio.h>

void exit_1(void){ printf(“In exit_1!\n”); }
void exit_2(void){ printf(“In exit_2!\n”); }
void exit_3(void){ printf(“In exit_3!\n”); }

int main(){
       printf(“Registering onexit_1...\n”); atexit(exit_1());
       printf(“Registering onexit_2...\n”); atexit(exit_2());
       printf(“Registering onexit_3...\n”); atexit(exit_3());
       exit(0);
}
```

1.8.4 Procesų programavimas

1.8.4.1 Proceso sukūrimas

Naujas procesas yra sukuriamas naudojant sisteminę komandą `fork(2)`:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Grąžinama `fork()` funkcijos reikšmė turi didelę prasmę, nes taip yra atskiriamos programos dalys:

```
main(){
       int pid;
       pid=fork();
       if(pid == -1){
              perror(“fork”);
              exit(1);
       }
       if(!pid)
              printf(“Vaikas\n”);
       else
              printf(“Tevas\n”);
}
```

Vaiko kodo vietoje reikia iškviesti sisteminę komandą exec(2), pvz.:

```
int execve(const char *path, char *const argv[], char *const envp[]);
int execvp(const char *file, char *const argv[]);
```

OS tėviniam procesui suteikia eilę funkcijų, kurios leidžia kontroliuoti vaikų darbus:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Komandinio interpretatoriaus pavyzdys:

```
pid=fork();
if(!pid){
       execvp(cmd,arg);
       pexit(cmd);
}else{
       wait(&status);
}
```

1.8.4.2 Apribojimai

UNIX daugia-vartotojiška aplinka gali kontroliuoti vartotojo procesų darbą ir gali uždėti jiems apribojimus. Sekančios sisteminės komandos pateikia ir uždeda apribojimus procesams:

```
#include <sys/time.h>
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, const struct rlimit *rlp);
```

, kur `resource` nurodo resurso tipą (žr. lentelę), o struktūra `rlimit` susideda iš dviejų laukų

```
struct rlimit {
rlim_t rlim_cur;
rlim_t rlim_max;
};
```

, kur yra nustatomi kintantis (soft) ir absoliutus (hard) apribojimas. Kintantį apribojimą gali keisti patys vartotojai, t.y. jo procesai iki pat absoliutaus apribojimo, kurį jie gali tik mažinti. Pastarąjį padidinti gali tik super-vartotojas. Paprastai jis būna nustatomas sistemos darbo pradžioje ir nekinta, bet yra paveldimas vaikų. Tarkim, kintantį proceso atidarytų failų kiekį galime nustatyti lygų 64, o absoliutusis apribojimas liks lygus 1024.

Maksimalus resurso naudojimo kiekis, jei nėra uždėtas kitoks, gali būti begalinis. Tada `rlim_max` reikšmė nustatoma lygi `RLIM_INFINITY`. Tokiu atveju, resurso panaudojimo lygis bus lygus absoliučiam resurso panaudojimo kiekiui, tarkim disko panaudojimas lygus jo talpai.

8.14 lentelė. Procesų apribojimai UNIX sistemose.

Resursas	**Tipas**	**Efektas**
`RLIMIT_CORE`	Maksimalus proceso core failo dydis. Jei 0, tai core nebus generuojamas.	Generuojant core failą, rašymas į jį bus sustabdomas kai pasieks t.t. nustatytą ribą.
`RLIMIT_CPU`	Maksimalus procesoriaus panaudojimo laikas sekundėmis.	Kai procesas viršys nustatytą laiką, jam bus nusiųstas signalas `SIGXCPU`.
`RLIMIT_DATA`	Maksimalus duomenų segmento dydis baitais, t.y. maksimalus brake adreso postūmis.	Kai pasiekiama riba procesas baigia darbą su klaida `ENOMEM`.
`RLIMIT_FSIZE`	Maksimalus sukuriamo failo dydis. Jei 0, tai procesas negali sukurti failo.	Kai proceso kuriamas failas viršija nustatytą dydį, jam nusiunčiamas signalas `SIGXFSZ`, jei jis jį perima ar ignoruoja – procesas sustabdomas su klaida `EFBIG`.
`RLIMIT_NOFILE`	Maksimalus proceso naudojamų failų deskriptorių kiekis.	Kai pasiekiama riba ir norima gauti dar vieną failų deskriptorių yra generuojama klaida `EMFILE`.
`RLIMIT_STACK`	Maksimalus proceso steko dydis.	Jei procesas bando didinti steką virš nustatytos ribos, jam nusiunčiamas signalas `SIGSEGV`. Jei jį procesas ignoruoja arba perima ir nesukuria alternatyvaus steko su funkcija `sigaltstack(2)`, tai minėto signalo dispozicija nustatoma į reikšmę pagal nutylėjimą.
`RLIMIT_VMEM`	Maksimalus proceso naudojamos virtualios atminties kiekis (tik SV).	Kai pasiekiama riba, tai sekantys `brk(2)` arba `mmap(2)` funkcijų iškvietimai baigsis su klaida `ENOMEM`.
`RLIMIT_NPROC`	Maksimalus procesų, su vienu realiu UID, skaičius (tik BSD).	Kai viršijama riba, sekantis `fork` iškvietimas baigiasi klaida `EAGAIN`.
`RLIMIT_RSS`	Maksimalus proceso rezidentinės atminties kiekis (Resident Set Size), t.y. procesui suteiktos fizinės atminties kiekis (tik BSD).	Jei sistemoje pritrūks fizinės atminties, tai sistema atlaisvins atmintį savo RSS viršijusių procesų sąskaita.
`RLIMIT_MEMLOCK`	Maksimalus fizinių puslapių skaičius, kurį procesas gali užblokuoti su komanda `mlock` (tik BSD).	Kai viršys ribą, komanda `mlock` baigsis sus klaida `EAGAIN`.

Pateiksime programą, kuri išveda duotojo proceso apribojimus:

```
#include <sys/types.h>
#include <sys/resource.h>

void disp_limit(int resource, char *rname){
       struct rlimit rlm;
       getrlimit(resource, &rlm);
// Spausdiname resurso pavadinim•
       printf("%-13s ", rname);
// Spausdiname kintam• apribojim•
       if(rlm.rlim_cur == RLIM_INFINITY) print("infinite\t");
       else printf(%10ld\t",rlm.rlim_cur);
// Spausdiname absoliut• apribojim•
       if(rlm.rlim_max == RLIM_INFINITY) print("infinite\n");
       else printf(%10ld\n",rlm.rlim_max);
}

main(){
       desp_limit(RLIMIT_CORE, "RLIMIT_CORE");
       desp_limit(RLIMIT_CPU, "RLIMIT_CPU");
       desp_limit(RLIMIT_DATA, "RLIMIT_DATA");
       desp_limit(RLIMIT_FSIZE, "RLIMIT_FSIZE");
       desp_limit(RLIMIT_NOFILE, "RLIMIT_NOFILE");
       desp_limit(RLIMIT_STACK, "RLIMIT_STACK");
// BSD resurs• apribojimai
#ifdef RLIMIT_NPROC
       desp_limit(RLIMIT_NPROC, "RLIMIT_NPROC");
#endif
#ifdef RLIMIT_RSS
       desp_limit(RLIMIT_RSS, "RLIMIT_RSS");
#endif
#ifdef RLIMIT_MEMLOCK
       desp_limit(RLIMIT_MEMLOCK, "RLIMIT_MEMLOCK");
#endif
// SV resurs• apribojimai
#ifdef RLIMIT_VMEM
       desp_limit(RLIMIT_VMEM, "RLIMIT_VMEM");
#endif
}
```

1.8.5 Signalų dispozicija

Pateiksime pavyzdį programos, kuri perima `SIGINT` signalą:

```
#include <signal.h>
#include <stdio.h>

static void sig_handler(int signo){
/* Atstatome signalo dispozicija */
       signal(SIGINT, SIG_DFL);
       printf("Gautas SIGINT signalas.\n");
}

int main(){
/* Nustatome signalo dispozicija */
       signal(SIGINT, sig_handler);
       signal(SIGUSR1, SIG_DFL);
       signal(SIGUSR2, SIG_IGN);
/* Begalinis ciklas */
       while(1) pause();
}
```

1.8.6 IPC programavimas

1.8.6.1 FIFO

FIFO pavyzdys:

```
// ======================= server.c

#include <sys/types.h>
```

```
#include <sys/stat.h>
#define FIFO        “fifo.1”
#define MAXBUFF     80

main(){
       int readfd, n;
       char buff[MAXBUFF];
       if(mknod(FIFO, S_FIFO | 0666, 0)<0){
              printf(“Negaliu sukurti FIFO\n”);
              exit(1);
       }
       if((readfd=open(FIFO, O_RDONLY))<0){
              printf(“Negaliu atidaryti FIFO\n”);
              exit(2);
       }
       while((n=read(readfd, buff, MAXBUFF))>0)
              if(write(1,buff, n)!=n){
                     printf(“Isvedimo klaida\n”);
                     exit(3);
              }
       close(readfd);
       exit(0);
}

// ======================== client.c

#include <sys/types.h>
#include <sys/stat.h>
#define FIFO        “fifo.1”

main(){
       int writefd, n;
       if((writefd=open(FIFO, O_WRONLY))<0){
              printf(“Negaliu atidaryti FIFO\n”);
              exit(1);
       }
       if(write(writefd,”Labas pasauli!\n”,16)!=16){
              printf(“Ivedimo klaida\n”);
              exit(2);
       }
       close(writefd);

       if(unlink(FIFO)<0){
              printf(“Negaliu sunaikinti FIFO\n”);
              exit(3);
       }
       exit(0);
}
```

1.8.6.2 Pranešimų eilės

```
// ======================== mesg.h

#define MAXBUFF     80
#define PERM        0666
typedef struct our_msgbuf {
       long mtype;
       char buff[MAXBUFF];
} Message;

// ======================== server.h

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include “mesg.h”
main(){
       Message message;
       key_t key;
       int msgid, length;

       key=ftok(“server”,’A’);
       message.mtype=1L;
```

```
        msgid=msgget(key,PERM | IPC_CREAT);
        length=msgrcv(msgid, &message, sizeof(message), message.mtype, 0);
        if(length >0) write(1, message.buff, length);
        exit(0);
}

// ======================= client.h

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include “mesg.h”
main(){
        Message message;
        key_t key;
        int msgid, length;

        key=ftok(“server”,’A’);
        message.mtype=1L;
        msgid=msgget(key,0);
        length=sprintf(message.buff, “Labas, pasauli!\n”);
        msgsnd(msgid, (void*) &message, length,0);
        msgctl(msgid, IPC_RMID,0);
        exit(0);
}
```

1.8.6.3 Bendrai naudojama atmintis ir semaforas

```
// ======================== shmem.h
#define MAXBUFF     80
#define PERM        0666

// Bendrai naudojama duomenų struktūra
typedef struct mem_msg {
        int segment;
        char buff[MAXBUFF];
} Message;

// Kliento darbo pradžios laukimas
static struct sembuf proc wait [1] = {
        1, -1, 0 };

// Pranešimas serveriui apie kliento darbo pradžią
static struct sembuf proc_start[1] = {
        1, 1, 0 };

// Bendros atminties blokavimas
static struct sembuf mem_lock[2] = {
        0, 0, 0,
        0, 1, 0 };

// Bendros atminties atlaisvinimas
static struct sembuf mem_unlock[1] = {
        0, -1, 0 };

// ======================== server.h
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include “shmem.h”

main(){
        Message *msgptr;
        key_t key;
        int shmid, semid;

// Vienas raktas bendrai atminčiai ir semaforui
```

```
       key=ftok(“server”,’A’);

// Sukuriame bendros atminties sritį
       shmid=shmget(key, sizeof(Message), PERM | IPC_CREAT);

// Prijungiame bendrą atmintį
       msgptr=(Message*)shmat(shmid,0,0);

// Sukuriame semaforų grupę, kurios
// 0 – darbo su bendra atmintimi sinchronizavimui, o
// 1 – procesų darbo sinchronizavimui
       semid=semget(key, 2, PERM | IPC_CREAT);

// Laukiame, kol klientas pradės darbą
       semop(semid, &proc_wait[0], 1);

// Laukiame, kol klientas baigs rašyti į atmintį, po to ją užblokuojame
       semop(semid, &mem_lock[0], 2);

// Spausdiname buferio turinį
       printf(“%s”, msgptr->buff);

// Atlaisviname atmintį
       semop(semid, &mem_unlock[0],1);

// Atsijungiame nuo atminties
       shmdt(msgptr);

    exit(0);
}

// ======================== klientas.h
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include “shmem.h”

main(){
       Message *msgptr;
       key_t key;
       int shmid, semid;

// Vienas raktas bendrai atminčiai ir semaforui
       key=ftok(“server”,’A’);

// Prieiname prie bendros atminties
       shmid=shmget(key, sizeof(Message), 0);

// Prijungiame bendrą atmintį
       msgptr=(Message*)shmat(shmid,0,0);

// Prieiname prie semaforo
       semid=semget(key, 2, PERM);

// Blokuojame atmintį
       semop(semid, &mem_lock[0], 2);

// Pranešame serveriui apie darbo pradžią
       semop(semid, &proc_start[0], 1);

// Rašome į bendrą atmintį
       sprintf(msgptr->buff, “Labas, pasauli!\n”);

// Atlaisviname bendrą atmintį
```

```
        semop(semid, &mem_unlock[0], 1);

// Laukiame, kol serveris neatlaisvins atminties
        semop(semid, &mem_lock[0], 2);

// Atsijungiame nuo atminties
        shmdt(msgptr);

// Sunaikiname IPC
        shmctl(shmid, IPC_RMID, 0);
        semctl(semid, 0, IPC_RMID);

    exit(0);
}
```

1.8.7 Sisteminis žurnalas (syslog)

UNIX sistemose dažniausiai veikia `syslogd` sisteminis žurnalo daemon'as, kuris surenka pranešimus iš kitų servisų, vartotojų ir pan. ir surašo į sisteminį žurnalą. Failas, kuriame yra surašomi pranešimai yra nurodytas `/etc/syslog.conf` faile. Funkcija, kuria siunčiamas pranešimas turi tokį pavidalą:

```
#include <syslog.h>
void syslog(int priority, char *logstring);
```

, kur `logstring` yra pranešimas, o `priority` yra viena iš šių reikšmių:

`LOG_EMERG`	Sistemoje panika, išsiuntinėjama visiems vartotojams
`LOG_ALERT`	Nenormali situacija, tarkim nugriuvo DB
`LOG_CRIT`	Kritinė situacija, tarkim klaida diske
`LOG_ERR`	Klaida
`LOG_WARNING`	Perspėjimas
`LOG_NOTICE`	Dėmesį reikalaujanti atkreipti informacija
`LOG_INFO`	Informuojantis pranešimas
`LOG_DEBUG`	programos klaidų tvarkymo informacija

Funkcija `openlog` leidžia nustatyti žurnalo pildymo parametrus:

```
void openlog(char *ident, int logopt, int facility);
```

, kur `ident` bus rašoma prieš kiekvieną pranešimą; `logopt` nustato papildomas opcijas:

`LOG_PID`	Rašomas proceso `PID`
`LOG_CONS`	Jei negali rašyti į žurnalą, išveda į konsolę

`facility` nurodo pranešimų šaltinį:

`LOG_KERN`	Branduolys
`LOG_USER`	Vartotojo programa (pagal nutylėjimą)
`LOG_MAIL`	Pašto servisas
`LOG_DAEMON`	Sisteminis servisas
`LOG_NEWS`	USENET sistema
`LOG_CRON`	`cron` servisas

Po darbo su žurnalu jį reikia tvarkingai uždaryti:

```
void closelog(void);
```

1.8.8 Daemon programavimas

Daemon'ai yra UNIX sistemos servisai. Kai kurie daemon'ai dirba pastoviai, pvz. `init` procesas; kiti yra paleidžiami t.t. momentais, pvz. `cron`. Daemon'ai neturi terminalinio valdymo. Suformuluosime keletą taisyklių, kurios turi užtikrinti UNIX servisų normalų darbą:

- servisas neturi reaguoti į vartotojo užduočių valdymo signalus, t.y. daemon'as po kurio tai laiko turi nusiimti asociaciją nuo valdančiojo terminalo, tačiau pradžioje jam gali reikėti išvesti kai kurią informaciją į ekraną;
- reikia uždaryti visus atidarytus failus, kurių daugelis yra susiję su terminaliniais įrenginiais, o servisas privalo dirbti ir tada, kai vartotojas baigė darbą – išėjo iš sistemos;
- pranešimus apie daemon'o darbą reikia siųsti į specialų sisteminį žurnalą (log'ą) naudojant funkciją `syslog(2)`;
- būtina pakeisti einamąjį katalogą į šakninį. nes kitaip, jei tarkim daemon'o einamasis katalogas bus primontuotoje failų sistemoje, tai kol servisas dirbs, jos nebus galima atjungti.

Daemon'o skeleto pavyzdys:

```
#include <stdio.h>
#include <syslog.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/resource.h>

main(int argc, char **argv){
	int fd;
	struct rlimit flim;
// Jei t•vas init – galima nesir•pinti d•l terminalini• signal•,
// jei ne – b•tina juos ignoruoti
if(getppid()!=1){
		signal(SIGTTOU, SIG_IGN);
	signal(SIGTTIN, SIG_IGN);
		signal(SIGTSTP, SIG_IGN);
// Sukuriame proces• vaik•, o t•v• nužudome
	if(fork()!=0) exit(0);
// Padarome vaik• grup•s lyderiu
	setsid();
}
// Uždarome visus atidarytus failo deskriptorius
	getrlimit(RLIMIT_NOFILE, &flim);
	for(fd=0;fd<flim.rlim_max;fd++) close(fd);
// Pakei•iame einam•j• katalog•
	chdir("/");
// Pranešame apie save • sistemin• žurnal•
	openlog("Demon example", LOG_PID | LOG_CONS, LOG_DAEMON);
	syslog(LOG_INFO, "Daemon example started job...");
// Jus• kodas


// Pabaigus darb• uždarome žurnal•
	closelog();
}
```

1.8.9 BSD UNIX socket programavimas

Tipinis socket serverio darbo scenarijus:

```
sockfd=socket(...);
bind(sockfd, ...);
listen(sockfd, ...);
for( ; ; ){
	newsockfd=accept(sockfd, ...);
	if(!fork()){
		close(sockfd);
```

```
                ...
                exit(0);
        }else
                close(newsockfd);
}
```

AF_UNIX domeno ir datagramų programavimo pavyzdys:

```
// ============================== server.c
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define MAXBUF      256

char buf[MAXBUF];

main(){

        struct sockaddr_un serv_addr, clnt_addr;
        int sockfd;
        int saddrlen, caddrlen, max_caddrlen, n;

        if((sockfd=socket(AF_UNIX, SOCK_DGRAM, 0))<0){
                printf("Negaliu sukurti socket objekto!\n");
                exit(1);
        }

        unlink("./echo.serv");
        bzero(&serv_addr, sizeof(serv_addr));
        serv_addr.sun_family=AF_UNIX;
        strcpy(serv_addr.sun_path,"./echo.serv");
        saddrlen=sizeof(serv_addr.sun_family)+strlen(serv_addr.sun_path);

        if(bind(sockfd, (struct sockaddr *)&serv_addr, saddrlen)<0){
                printf("Negaliu priristi socket objekto!\n");
                exit(2);
        }

        max_caddrlen=sizeof(clnt_addr);

        for(;;){
                caddrlen=max_caddrlen;
                n=recvfrom(sockfd,    buf,    MAXBUF,    0,    (struct    sockaddr
*)&clnt_addr, &caddrlen);
                if(n<0){
                        printf("Klaida priimant duomenis!\n");
                        exit(3);
                }
                if(sendto(sockfd,  buf,  n,  0,  (struct  sockaddr  *)&clnt_addr,
caddrlen)!=n){
                        printf("Klaida siunciant duomenis!\n");
                        exit(4);
                }
        }
}
// ============================== server.c
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define MAXBUF      256

char *msg="Labas pasauli!\n";
char buf[MAXBUF];

main(){

        struct sockaddr_un serv_addr, clnt_addr;
        int sockfd;
        int saddrlen, caddrlen, msglen, n;

        bzero(&serv_addr, sizeof(serv_addr));
        serv_addr.sun_family=AF_UNIX;
```

```
	strcpy(serv_addr.sun_path,"./echo.serv");
	saddrlen=sizeof(serv_addr.sun_family)+strlen(serv_addr.sun_path);

	if((sockfd=socket(AF_UNIX, SOCK_DGRAM, 0))<0){
		printf("Negaliu sukurti socket objekto!\n");
		exit(1);
	}

	bzero(&clnt_addr, sizeof(clnt_addr));
	clnt_addr.sun_family=AF_UNIX;
	strcpy(clnt_addr.sun_path,"/tmp/clnt.XXXX");
	mkstemp(clnt_addr.sun_path);
	caddrlen=sizeof(clnt_addr.sun_family)+strlen(clnt_addr.sun_path);

	if(bind(sockfd, (struct sockaddr *)&clnt_addr, caddrlen)<0){
		printf("Negaliu priristi socket objekto!\n");
		exit(2);
	}

	msglen=strlen(msg);

	if(sendto(sockfd,  msg,  msglen,  0,  (struct  sockaddr  *)&serv_addr,
saddrlen)!=msglen){
		printf("Klaida siunciant duomenis!\n");
		exit(3);
	}

	if((n=recvfrom(sockfd, buf, MAXBUF, 0, 0, 0))<0){
		printf("Klaida priimant duomenis!\n");
		exit(3);
	}

	printf("Echo: %s\n",buf);

	close(sockfd);
	unlink(clnt_addr.sun_path);
	exit(0);

}
```

1.8.10 Tinklo programavimas

1.8.10.1 Soketų programavimas

```
// ============================
// inet_server.c
// ============================

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <fcntl.h>
#include <netdb.h>

#define PORTNUM 1500

int main(argc,argv)
int argc; char *argv[];
{
	int s, ns, pid, nport;
	struct sockaddr_in serv_addr, clnt_addr;
	char buf[80], hname[80];

	nport=PORTNUM;
	nport=htons((u_short)nport);

	if((s=socket(AF_INET, SOCK_STREAM, 0))==-1){
		perror("Klaida sukuriant soketa");
		exit(1);
	}
```

```
	bzero(&serv_addr,sizeof(serv_addr));
	serv_addr.sin_family=AF_INET;
	serv_addr.sin_addr.s_addr = INADDR_ANY;
	serv_addr.sin_port = nport;

	if(bind(s,(struct sockaddr *)&serv_addr,sizeof(serv_addr))==-1){
		perror("Klaida iskvieciant bind()");
		exit(2);
	}

	fprintf(stderr,"Serveris                                        pasiruoses:
%s\n",inet_ntoa(serv_addr.sin_addr));

	if(listen(s,5)==-1){
		perror("Klaida iskvieciant listen()");
		exit(3);
	}

	while(1){

		int addrlen;
		bzero(&clnt_addr,sizeof(clnt_addr));
		addrlen=sizeof(clnt_addr);

		if((ns=accept(s,(struct sockaddr *)&clnt_addr,&addrlen))==-1){
			perror("Klaida priimant klienta");
			exit(4);
		}

		fprintf(stderr,"Klientas                                          =
%s\n",inet_ntoa(clnt_addr.sin_addr));

		if((pid=fork())==-1){
			perror("Klaida iskvieciant fork()");
			exit(5);
		}

		if(!pid){
			int nbytes;
			int fout;

			close(s);

			while((nbytes = recv(ns,buf,sizeof(buf),0))!=0){
				send(ns,buf,sizeof(buf),0);
			}

			close(ns);
			exit(0);
		}

		close(ns);
	}
	return 0;
}

// ============================
// inet_client.c
// ============================

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <fcntl.h>
#include <netdb.h>

#define PORTNUM 1500

int main(argc,argv)
int argc; char *argv[];
{
```

```
        int s, pid, i, j;
        struct sockaddr_in serv_addr;
        struct hostent *hp;
        char buf[80]="Labas, pasauli!";

        if(!(hp=gethostbyname(argv[1]))){
                perror("Klaida iskvieciant gethostbyname()");
                exit(1);
        }

        bzero(&serv_addr, sizeof(serv_addr));
        bcopy(hp->h_addr,&serv_addr.sin_addr, hp->h_length);
        serv_addr.sin_family = hp->h_addrtype;
        serv_addr.sin_port = htons(PORTNUM);

        if((s=socket(AF_INET, SOCK_STREAM, 0))==-1){
                perror("Klaida sukuriant socketa");
                exit(2);
        }

        fprintf(stderr,"Kliento                    adresas                    =
%s\n",inet_ntoa(serv_addr.sin_addr));

        if(connect(s,(struct sockaddr *)&serv_addr, sizeof(serv_addr))==-1){
                perror("Klaida iskvieciant connect()");
                exit(3);
        }

        send(s,buf,sizeof(buf),0);

        if(recv(s,buf,sizeof(buf),0)<0){
                perror("Klaida iskvieciant recv()");
                exit(4);
        }

        printf("Gauta is serverio: %s\n",buf);

        close(s);
        printf("Klientas baige darba!\n\n");

        return 0;
}
```

1.8.10.2 TLI programavimas

```
//  tli_client.c

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <fcntl.h>
#include <netdb.h>
#include <tiuser.h>

#define PORTNUM 1500

int main(argc,argv)
int argc; char *argv[];
{
        int tn, flags;
        struct sockaddr_in serv_addr;
        struct hostent *hp;
        char buf[80]="Labas pasauli!";
        struct t_call *call;

        if(!(hp=gethostbyname(argv[1]))){
                perror("Klaida iskvieciant gethostbyname()");
                exit(1);
        }

        if((tn=t_open("/dev/tcp",O_RDWR,NULL))==-1){
```

```
                t_error("Klaida iskvieciant t_open()");
                exit(1);
        }

        if(t_bind(ntn,(struct t_bind *)0,(struct t_bind *)0)<0){
                t_error("Klaida iskvieciant t_bind()");
                exit(1);
        }

        fprintf(stderr,"Klientas                                        pasiruoses:
%s\n",inet_ntoa(serv_addr.sin_addr));

        bzero(&serv_addr,sizeof(serv_addr));
        bcopy(hp->h_addr,&serv_addr.sin_addr,hp->h_length);
        serv_addr.sin_family=hp->h_addrtype;
        serv_addr.sin_port = htons(PORTNUM);

        if((call=(struct t_call*)t_alloc(tn,T_CALL,T_ADDR))==NULL){
                t_error("Klaida iskvieciant t_alloc()");
                exit(1);
        }

        call->addr.maxlen=sizeof(serv_addr);
        call->addr.len=sizeof(serv_addr);
        call->addr.buf=(char*)&serv_addr;
        call->opt.len=0;
        call->udata.len=0;

        if(t_connect(tn,call,(struct t_call*)0)<0){
                perror("Klaida iskvieciant t_connect()");
                exit(1);
        }

        t_snd(tn,buf,sizeof(buf),0);
        if(t_rcv(tn,buf,sizeof(buf),&flags)<0){
                perror("Klaida iskvieciant t_rcv()");
                exit(1);
        }

        printf("Gauta is serverio: %s\n",buf);
        t_close(tn);

        return 0;
}

// tli_server.c

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <fcntl.h>
#include <netdb.h>
#include <tiuser.h>

#define PORTNUM 1500

int main(argc,argv)
int argc; char *argv[];
{
        int tn, pid, ntn, flags, nport;
        struct sockaddr_in serv_addr, *clnt_addr;
        struct hostent *hp;
        char buf[80], hname[80];
        struct t_bind req;
        struct t_call *call;

        if((tn=t_open("/dev/tcp",O_RDWR,NULL))==-1){
                t_error("Klaida iskvieciant t_open()");
                exit(1);
        }
```

```
        nport=PORTNUM;
        nport=htons((u_short)nport);
        bzero(&serv_addr,sizeof(serv_addr));
        serv_addr.sin_family=AF_INET;
        serv_addr.sin_addr.s_addr = INADDR_ANY;
        serv_addr.sin_port = nport;
        req.addr.len=sizeof(serv_addr);
        req.addr.buf=(char*)&serv_addr;
        req.qlen=5;

        if(t_bind(tn,&req,(struct t_bind *)0)<0){
               t_error("Klaida iskvieciant t_bind()");
               exit(1);
        }

        fprintf(stderr,"Serveris                                        pasiruoses:
%s\n",inet_ntoa(serv_addr.sin_addr));

        if((call=(struct t_call*)t_alloc(tn,T_CALL,T_ADDR))==NULL){
               t_error("Klaida iskvieciant t_alloc()");
               exit(1);
        }

        call->addr.maxlen=sizeof(serv_addr);
        call->addr.len=sizeof(serv_addr);
        call->opt.len=0;
        call->udata.len=0;

        while(1){

               if(t_listen(s,call)<0){
                      perror("Klaida iskvieciant t_listen()");
                      exit(1);
               }

               clnt_addr=(struct sockaddr_in *)call->addr.buf;
               fprintf(stderr,"Klientas                                                =
%s\n",inet_ntoa(clnt_addr.sin_addr));

               if((ntn=t_open("/dev/tcp",O_RDWR,(struct t_info*)0))<0){
                      t_error("Klaida iskvieciant t_open()");
                      exit(1);
               }

               if(t_bind(ntn,(struct t_bind *)0,(struct t_bind *)0)<0){
                      t_error("Klaida iskvieciant t_bind()");
                      exit(1);
               }

               if(t_accept(tn,ntn,call)<0){
                      perror("Klaida iskvieciant t_accept()");
                      exit(1);
               }


               if((pid=fork())==-1){
                      perror("Klaida iskvieciant fork()");
                      exit(1);
               }

               if(!pid){
                      int nbytes;
                      t_close(tn);

                      while((nbytes = t_rcv(ntn,buf,sizeof(buf),&flags))!=0){
                             t_snd(ntn,buf,sizeof(buf),0);
                      }

                      t_close(ntn);
                      exit(0);
               }

               t_close(tn);
```

```
        }
        return 0;
}
```

1.8.10.3 RPC programavimas

RPC funkcijos aprašymas yra atliekamas specialiu formatu, pvz. kaip šis rpc_log.x failas:

```
program LOG_PROG {
        version LOG_VER {
                int RLOG(string)=1;
        }=1;
}=0x3213456;
```

Naudojant programą rpcgen(1)

```
$ rpcgen rpc_log.x
```

yra sugeneruojami trys failai rpc_log.h – bendrai naudojama antraštė, rpc_log_svc.c – RPC serverio stub funkcijos realizacija ir rpc_log_clnt.c – RPC kliento stub funkcijos realizacija. Žemiau yra pateikiami rpc_log.h, rpc_log_server.c (serveris) ir rpc_log_client.c (klientas) programos tekstai:

```
// rpc_log.h

/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _RPC_LOG_H_RPCGEN
#define       _RPC_LOG_H_RPCGEN

#include <rpc/rpc.h>

#define       LOG_PROG ((unsigned long)(0x3213456))
#define       LOG_VER ((unsigned long)(1))
#define       RLOG ((unsigned long)(1))
extern  int * rlog_1();
extern int log_prog_1_freeresult();

#endif /* !_RPC_LOG_H_RPCGEN */

// rpc_log_server.c

#include <rpc/rpc.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "rpc_log.h"

int *rlog_1(char **arg){

static int result;
int fd;
int len;
result=1;

if((fd=open("./server.log",O_CREAT|O_RDWR|O_APPEND))<0) return &result;
len=strlen(*arg);
if(write(fd,*arg,strlen(*arg))!=len)
        result = 1;
else
        result = 0;
close(fd);
return &result;
}

// rpc_log_client.c
```

```
#include <rpc/rpc.h>
#include <stdio.h>
#include "rpc_log.h"

main(int argc,char *argv[]){

CLIENT *cl;
char *server, *mystring, *clnttime;
time_t bintime;
int *result;

if(argc!=2){
       fprintf(stderr, "Usage: %s server_host\n",argv[0]);
       exit(1);
}

server=argv[1];

if((cl=clnt_create(server,LOG_PROG,LOG_VER,"udp"))==NULL){
       clnt_pcreateerror(server);
       exit(1);
}

mystring=(char*) malloc(100);
bintime=time((time_t*)NULL);
clnttime=ctime(&bintime);

sprintf(mystring,"Klientas startavo %s",clnttime);

if((result=rlog_1(&mystring,cl))==NULL){
       fprintf(stderr,"Klaida siunciat log'a (1)\n");
       clnt_perror(cl,server);
       exit(1);
}

if(*result!=0)
       fprintf(stderr,"Klaida siunciat log'a (2)\n");
clnt_destroy(cl);
exit(0);

}
```

Šie failai yra kompiliuojami sekančiu būdu:

```
$ gcc –o rpc_log_server rpc_log_server.c rpc_log_svc.c -lnsl

$ gcc –o rpc_log_client rpc_log_client.c rpc_log_clnt.c –lnsl
```