Turinys

Lentelių sąrašas

Iliustracijos

Rekomenduojamos knygos

Fiamingo F.G., DeBula L., Condron L., 1998: Introduction to Unix. The Ohio State University, University Technology Services.

Gilly D., et al, 1992: UNIX in a Nutshell: A Desktop Quick Reference for System V & Solaris 2.0. O'Reilly & Associates, Inc, ISBN 0-56592-001-5.

O'Reilly & Associates , 1990: UNIX in a Nutshell for BSD 4.3: A Desktop Quick Reference For Berkeley. O'Reilly & Associates, Inc, ISBN 0-937175-20-X.

Becker G., Morris M.E.S., Slattery K., 1995: SOLARIS Implementation. A Guide for System Administrators. SunSoft Press, A Prentice Hall Title.

Abrahams P.W., Larson B.R., 1992: UNIX for the Impatient. Addison-Wesley Publishing Company, ISBN 0-201-55703-7.

Brian W. Kernighan & Rob Pike, 1984: The UNIX Programming Environment. Prentice Hall.

Nemeth E., Snyder G., Seebass S., Hein T.R., 2001: UNIX System Administration Handbook. Third Edition. Prentice Hall PTR, ISBN 0-13-020601-6.

Frisch A., 1993: Essential System Administration. Help for UNIX System Administrators. O'Reilly & Associates, Inc, ISBN 0-937175-80-3.

Chris Drake, Kimberley Brown, 1995. PANIC! UNIX® System Crash Dump Analysis. Sun Soft Press, A Prentice Hall Title, ISBN 0-13-149386-8.

Robin Burk, David B. Horvath, 1998. UNIX® Internet Edition. Unleashed, Sams Corporation, ISBN 0-672-31205-0.

Андрей Робачевский, 1998. Операционная система UNIX. BHV – Санкт-Петербург, ISBN 5-7791-0057-8.

1 Įvadas į UNIX

1.1 Trumpa UNIX istorija

1965 – Bell Laboratories, MIT ir General Electric susitaria kurti naują operacinę sistemą, Multics, turinčią šias savybes: daugia-vartotojiškumą, daugia-procesoriškumą, hierarchinę failų sistemą ir kitas savybes.

1969 – AT&T netenkina kūrimo progresas ir ji palieka projektą. Keli Bell Labs programuotojai – Ken Thompson, Dennis Ritchie, Rudd Canaday ir Doug McIlroy suprojektuoja ir sukuria pirmąją UNIX failų sistemos versiją PDP-7 mašinai. Brian Kernighan ją pavadina UNIX.

1970, Sausio 1 – UNIX pradžia (time zero).

1971 – Sistema dirba PDP-11 mašinoje, kur 16K užėmė sistema, 8K skirti vartotojo programoms; disko dydis – 512K, o maksimalus failo dydis galėjo siekti 64K.

Ši programa daugiausia buvo naudojama Bell Labs patentų departamente tekstų tvarkymui. Šia kryptimi toliau ir buvo vystoma UNIX sistema. UNIX tapo populiari tarp programuotojų dėl šių savybių:

- programavimo aplinka;
- paprastas vartotojų interfeisas;
- paprastos pagalbinės programos, kurias apjungus atliekamos sudėtingos funkcijos;
- hierarchinė failų sistema;
- paprastas aparatinės įrangos interfeisas sutapatinamas su failais;
- daugia-vartotojiška, daugia-procesoriška sistema;
- nepriklausoma architektūra.

1973 – UNIX sistema perrašoma C kalba. Tai tuo metu nauja kalba, kurią sukūrė Dennis Ritchie. Ji evoliucionavo iš Tomson sukurtos B kalbos, kuri, savo ruožtu, buvo FORTRAN kompiliatoriaus tobulinimo paseka. C kalba žymiai palengvino sistemos adaptavimą naujoms mašinoms.

1974 – Ritchie ir Thompson parašo straipsnį “Communications of the ACM” žurnale, kuriame aprašo naują OS – UNIX. Tai sukelia akademinės visuomenės entuziazmą, kuri mato UNIX esant gera mokymosi priemone. AT&T buvo supančiotas sutartimis ir negalėjo pardavinėti naujo produkto, todėl universitetams, mokymosi tikslams, suteikė nemokamas licenzijas.

1.2 Sistemos redakcijos

1970-ųjų pabaigoje AT&T sukūrė UNIX Support Group (USG, vėliau UNIX System Laboratories, USL), kuri turėjo paruošti UNIX komercijai. Bell Labs ir USG toliau vystė sistemą, tačiau jų nuomonės išsiskyrė. USL išleido labai populiarias System III ir System V UNIX versijas.

Kiekvienai sistemos versijai buvo rašomi programuotojo vadovai, jų vis naujos redakcijos (edition) ir nuo to prie UNIX prikibo redakcijos pavadinimas. Viso yra išleista 10 redakcijų, pirmoji 1970, o paskutinioji – 1989 metais. Pirmosios 7 sukurtos Bell Labs CRG (Computer Research Group) ir buvo skirtos PDP, o vėliau VAX kompiuteriams.

System 1 (1971) – parašyta asemblerio kalba PDP-11 mašinai. Turėjo B kompiliatorių ir įvairių pagalbinių komandų, tokių kaip `cat, chdir, chmod, cp` ir pan.

System 2 (1973) – atsirado komanda `cc(1)`, kuri paleido C kompilatorių.

System 3 (1973) – C kalba perrašytas branduolys.

System 6 (1975) – pirmoji redakcija, kuri peržengė Bell Labs sienas. Visiškai perrašyta su C. Atsiranda naujos UNIX versijos, didėja populiarumas. Tomson ją suinstaliavo Berkeley universitete, kurios pagrindu atsirado BSD.

System 7 (1979) – atsirado Bourne Shell'as, Kernighan ir Ritchie C kompiliatorius. Branduolys perrašomas palengvinant OS pernešimą ant kitų sistemų. Šios sistemos licenziją nusipirko Microsoft ir jos pagrindu vėliau sukūrė savo sistemą XENIX.

1.3 Tipai

Grynos BSD ar System V sistemos niekada neturėjo komercinės vertės. Dauguma UNIX sistemų yra vystomos arba BSD, arba System V pagrindu. Todėl atsirado daugybė UNIX variantų ir versijų, nors jos visos kilo iš vieno iš dviejų arba abiejų (hibridinės sistemos) protėvių.

1.3.1 System V

1976 metais UNIX tapo laisvai prieinama universitetams. Taip ji pagrindine mokymo klasių, tyrinėjimo projektų operacine sistema.

1982 metais AT&T, nenorėdama prarasti iniciatyvos UNIX vystyme, surinko visas esančias UNIX versijas ir išleido System III UNIX'ą. Tai buvo sistema skirta platinimui. Ir dabar yra plačiai naudojamos System III pagrindu sukurtos sistemos.

1983 Bell Labs išleidžia System V. Neužilgo ją patobulina ir išleidžia System V Release 2 (SVR2). Buvo pristatyta nemažai naujovių, kaip kad tarp-procesinė sąveika (IPC).

1987 metais pasirodęs SVR3 jau turi daug naujų savybių, tokių kaip streams I/O posistemę, failų sistemos switch'ą, kuris leidžia vienu metu palaikyti įvairių tipų failų sistemas, programinį tinklo interfeisą (TLI – Trasport Layer Interface).

Galų gale, 1989 metais išeina SVR4, kuri paveldi ne tik savo pirmtakų savybes, bet ir įgauna naujų pasiskolintų iš SunOS ir BSD sistemų. Štai kelios naujovės:

- Korn ir C komandiniai interpretatoriai (BSD);
- simbolinės nuorodos;
- terminalinį I/O paremtą streams technologija (SV);
- NFS ir RPC (SunOS);
- FFS (BSD);
- socket's programinį interfeisą (BSD).

1.3.2 BSD

1977 Computer Systems Research Group (CSRG) iš University of California, Berkeley, nusiperka kodo licenziją iš AT&T ir taip prasideda Berkeley UNIX šaka. Dar 1977 metais CSRG išleidžia 1BSD (BSD – Berkeley Software Distribution) skirtą PDP-11 mašinai. Ši grupė 1993 išleido paskutinę – 4.4 versiją.

AT&T kodo licenzijos visada buvo labai brangios (išskyrus akademines įstaigas). Universitetams licenzijos pradžioje buvo nemokamos arba labai pigios, tačiau kai pradėta UNIX komercija – kainos greitai kilo. Dėl pastarųjų priežasčių Berkeley iškėlė sau tikslą – perrašyti visą AT&T. Tai labai ilgas ir kruopštus darbas, todėl dar jo nebaigus CSRG neteko finansavimo ir buvo išformuota.

Prieš išformavimą Berkeley išleido savo paskutinę laisvo kodo versiją – 4.4BSD-Lite. Daugelis dabartinių BSD UNIX sistemų (BSD/OS, FreeBSD, NetBSD ir OpenBSD) yra sukurti būtent 4.4BSD-Lite pagrindu.

1.3.3 OSF/1

1988 metais, kai Sun ir AT&T sudarė sutartį dėl tolimesnio SV vystymo, kitos IT kompanijos, tokios kaip DEC, IBM, HP, sukūrė organizaciją Open Software Foundation (OSF) ,kuri turėtų sukurti naują, nepriklausomą nuo AT&T operacinę sistemą. Pirmasis rezultatas buvo OSF/1. Tai nebuvo kažkokia naujovė, greičiau politinis žingsnis siekiant sumažinti SV įtaką UNIX pasauliui.

1.3.4 Linux

Pastaruoju metu sumaištį UNIX padangėje sukėlė Linux – UNIX tipo branduolys, kurį parašė Torvald Linus, studentas iš Suomijos. Linux istorija prasidėjo 1991 metais. Linux tobulinimo darbą entuziastingai dirbo ir dirba Atvirojo kodo bendruomenė. Pirmoji pilnavertė Linux OS atsirado sujungus branduolį (Linux) ir GNU projekto programas. Pradinė GNU paskirtis buvo atviro kodo C kompiliatoriaus sukūrimas (dabar žinomas kaip GNU C, gcc), tačiau vėliau projektas išsiplėtė į daugelio dabartinių Linux apvalkalo programų projektą.

1.3.5 Kaip atskirti SVR* nuo BSD?

1.1 lentelė. Kai kurie System V ir BSD UNIX šeimų skirtumai

Indikatorius	Tipiška SVR*	Tipiška BSD
Branduolio pavadinimas	/unix	/vmunix
Startavimo failai	katalogai /etc/rc*.d	failai /etc/rc.*
Montuojamos failų sistemos	/etc/mnttab	/etc/mtab
Tipiškas komandų interpretatorius	sh, ksh	csh
“Gimtoji” failų sistema	S5	UFS
Spausdinimo sistema	lp, lpstat, cancel	lpr, lpq, lprm
Terminalų valdymas	terminfo	termcap
Aktyvių procesų informacija	ps -ef	ps -aux

1.4 UNIX versijos

1.4.1 AIX

IBM UNIX versija naudojanti SVR2 bazę, turi daug SVR4, BSD ir OSF/1 bruožų ir ypač neblogą administravimo sistemą (SMIT).

1.4.2 HP-UX

HP UNIX versija. Ji yra BSD ir SVR4 hibridas. Turi nemažai savotiškų dalykėlių ,tokių kaip simetrinės daugia-procesorinės sistemos, ypač didelės failų sistemos (iki 128Gb), aplikacijų procesų erdvė padidinta iki 3,75Gb.

1.4.3 IRIX

Silicon Graphics UNIX sistema, skirta jo paties MIPS aparatūrai. Tai pilnai 64 bitų sistema, kuri pradžioje buvo grynai BSD, dabar jau greičiau SVR4 atstovas.

1.4.4 Digital UNIX

Tai firmos Digital Equipment OSF/1 UNIX versija. Prieš tai ji vadinosi DEC OSF/1 ir buvo BSD tipo OS, nors turėjo nemažai SVR4 bruožų. Tai pilnai 64 bitų, pradžioje Alpha platformai skirta sistema. Palaiko daug tinklinių interfeisų.

1.4.5 SCO UNIX

1988 metais Santa Cruz Operations (SCO), Microsoft ir Interactive Systems pabaigė bendrą System V Release 3.2 adaptavimą Intel 386 platformai. Tada SCO nusipirko iš AT&T teisę ją platinti ir pavadino sistemą SCO UNIX System V/386. 1995 m. SCO išleido SCO OpenServer

Release 5, kodiniu pavadinimu Everest, kuris palaiko > 9000 aparatinių platformų ir > 2000 įvairių įrenginių.

1.4.6 Sun Solaris

Ji turi daug atmainų, tačiau viena iš labiausiai vykusių yra Sun Solaris 2. Pirmoji SunOS versija – Solaris 1 – buvo paremta BSD branduoliu, tačiau dėl glaudaus Sun bendradarbiavimo su AT&T buvo nuspręsta pereiti prie SVR4 branduolio. Mūsų nagrinėjama sistema ir yra Sun Solaris 2.7, 8, 9 su SVR4 branduoliu.

1.4.7 FreeBSD

Nagrinėsime FreeBSD/i386 4.5-RELEASE sistemą, kurios pagrindu yra minėta 4.4BSD-Lite. Ji yra nemokama ir ją galima rasti adresu: http://www.freebsd.org.

1.5 UNIX populiarumo priežastys

Pagrindinės UNIX populiarumo priežastys yra šios:

- UNIX yra parašyta naudojant aukšto lygio C kalbą. Tai leidžia ją greitai adaptuoti skirtingoms platformoms. Pagal Ritche, perrašius UNIX iš asemblerio į C UNIX sistemos apimtis padidėjo 20 – 40 %, o darbas sulėtėjo apie 20%. Tačiau sistemos atvirumas žymiai padidėjo.
- UNIX – daugia-procesinė, daugia-vartotojiška sistema. Su vienu galingu serveriu galima pateikti servisus šimtams vartotojų.
- Standartai. Nors UNIX versijų yra begalės, tačiau visos jos yra iš principo vienodos architektūros. Bet kuris administratorius, perkandęs vieną UNIX versiją galės administruoti ir kitą.
- Paprastas, bet labai galingas modulinis vartotojo interfeisas. Yra labai daug pagalbinių programų, moduliukų, kurie atlieka t.t. funkciją, tačiau juos sujungus galima atlikti tikrai sudėtingus ir painius uždavinius.
- Vieninga hierarchinė failų sistema. Per šią failų sistemą prieinama ne tik prie failų, bet ir prie terminalų, spausdintuvų, magnetinių juostų, tinklo ir net atminties.

2 UNIX struktūra

UNIX sistemas sudaro 3 abstraktūs sluoksniai:

- geležis (aparatūrinė įranga),
- branduolys (kernel) ir
- vartotojo programos.

2.1 Branduolys (kernel)

Branduolys paslepia aparatūrinę įrangą nuo abstraktaus, aukšto lygio programų sluoksnio. Jis atlieka daug darbų, kurių rezultatais naudojasi vartotojo lygmens programos. Pavyzdžiui branduolys realizuoja ir tvarko šiuos žemiau pateiktus UNIX architektūrinius sprendimus:

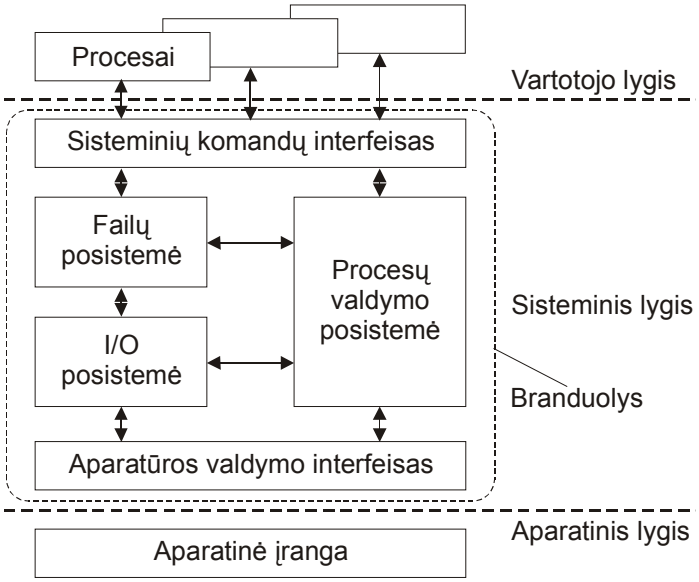
- procesus,
- signalus ir semaforus,
- virtualią atmintį,
- failų sistemą ir
- sąsają tarp procesų (sockets).

Branduolys tiesiogiai dirba su geležies tvarkyklėmis (device driver), kurie iš dalies ir sudaro branduolio dalį, kurio likusi beveik visa dalis yra nuo geležies nepriklausoma. Branduolys su tvarkyklėmis bendrauja taip pat, kaip ir programos su branduoliu, pavyzdžiui, kai procesas paprašo branduolio “perskaityk pirmus 64 `/etc/passwd` failo bitus”, tai branduolys nusiunčia tvarkyklei instrukciją, pvz. “Fetch block 3348 from device 3”. Tvarkyklė toliau dirbs savo darbą – suskaldys šią komandą į bitų blokus ir pasius juos į aparato valdymo registrus.

Branduolys yra parašytas C kalba su nedideliu kiekiu asemblerio kodo. Pradžioje branduolys buvo mažiukas, gal truputi virš pusės Mb, tačiau dabar, palaikantys sudėtingus tinklo protokolus, daugia-procesiškumą, jie siekia net iki 15Mb.

Solaris naudoja praktiškai pilnai modulinį branduolį. Tai suteikia galimybę pakrauti aparatūrinės įrangos tvarkyklę tada, kai jos reikia. Nereikia prieš tai sutvarkyti begalės konfigūracinių failų, perkompiliuoti branduolio ir pan.; tai įmanoma iš dalies dėl aiškios ir griežtos Sun SPARC mašinų architektūros. Kai Solaris suranda naują aparatūrinę įrangą, jis pats, jei randa, pakrauna ir tvarkyklę.

FreeBSD, kaip ir visos kitos BSD sistemos, turi tiksliai žinoti kokia aparatūrinė įranga yra sistemoje dar prieš branduolio kompiliaciją. Kai kuriais atvejais reikia nurodyti ne tik kokia aparatūrinė įranga yra, bet taip pat ir kur ji yra. Kai kuriais atvejais tenka tiesiog išsitraukti aparatūrinės įrangos modulį ir pažiūrėjus nustatyti – koks tiksliai jis yra.



2.1 pav. Bendra branduolio struktūra.

Linux branduolį galima tvarkyti taip pat kaip ir BSD sistemas – nurodyti tvarkykles ir sukompiliuoti, tačiau iš dalies galima ir naudotis modulinės architektūros privalumais, kaip Solaris. Tačiau dažnai pastarasis veiksmas naudos neduoda, nes PC architektūra yra sudėtinga ir nenuspėjama.

2.1 lentelėje pateikiami standartiniai katalogai, kuriuose galima rasti branduolio failus.

2.1 lentelė. Standartiniai branduolio failų katalogai.

Sistema	**Išeities kodų katalogas**	**Branduolio katalogas**
Solaris	-	`/kernel/unix`
Linux	`/usr/src/linux`	`/vmlinuz arba /boot/vmlinuz`
FreeBSD	`/usr/src/sys`	`/kernel`

Kai sistema instaliuojam pirmą kartą (dažniausiai iš spec. instaliacinio disko), jos branduolys turi dauguma ApI tvarkyklių, kad veiktų ant standartinių mašinų. Tačiau konkrečiai mašinai tereikia tam tikrų, tik jai skirtų tvarkyklių, todėl yra gerai perkompiliuoti branduolį. Nors

dabartiniai branduoliai yra geresni ir jie išmėto nereikalingas tvarkykles iš atminties, tačiau opcijos kraunantis vis tiek lieka. Branduolio perkompiliavimas nėra sudėtingas, tačiau jo teisingas konfigūravimas – yra.

Branduolį sudaro šie trys komponentai:

- procesų ir atminties valdymo posistemė,
- failų posistemė ir
- I/O posistemė.

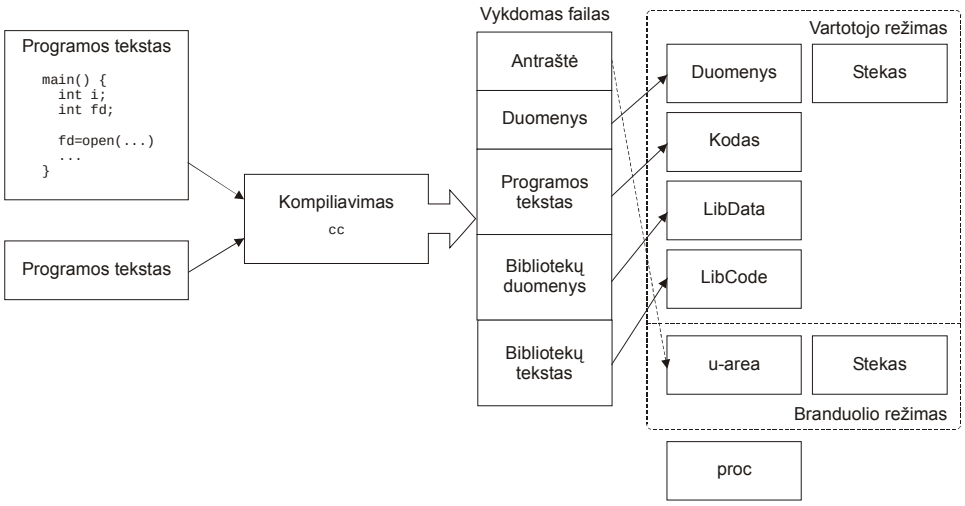
Bendrą branduolio vidinė struktūrą iliustruoja 2.1 paveikslas.

3 Procesų valdymo posistemė

UNIX OS širdis – procesų valdymo posistemė. Visas OS funkcionalumas galų gale priklauso nuo vienų ar kitų procesų vykdymo. Viskas – run-levels, servisai, spausdinimas ir pan. yra t.t. procesų vykdymo pasekmė. Procesai UNIX sistemoje yra tiesiogiai susiję su dviem resursais: procesoriumi (-iais) ir operatyvine atmintimi. Galioja taisyklė, kad šių resursų niekada nėra per daug, todėl OS pastoviai vyksta arši konkurencinė kova dėl jų. Nagrinėsime, kokiu būdu OS branduolys valdo atmintį, kai kiekvieno proceso adresinė erdvė siekia kelis gigabaitus, kaip OS vienu metu gali vykdyti daug procesų, kai vienas CPU tegali vykdyti tik vieną užduotį. Procesas, tai rėmelis į kurį yra įstatoma programa. Nors viena iš branduolio užduočių yra proceso izoliavimas, yra pakankamai svarbus ir duomenų tarp procesų apsikeitimo organizavimas. Tam UNIX suteikia nemažai priemonių – nuo paprastų signalų iki sudėtingų tarp-procesinės sąveikos mechanizmų – IPC SV ir soketų BSD sistemose.

3.1 Procesų valdymo pagrindai

Procesas, tai vykdomos programos pavidalas, tam tikros branduolio duomenų struktūros su nuorodomis į atmintyje esančius vykdomą kodą, duomenis, steką ir bibliotekas (3.1 pav.).



3.1 pav. Bendroji proceso struktūra.

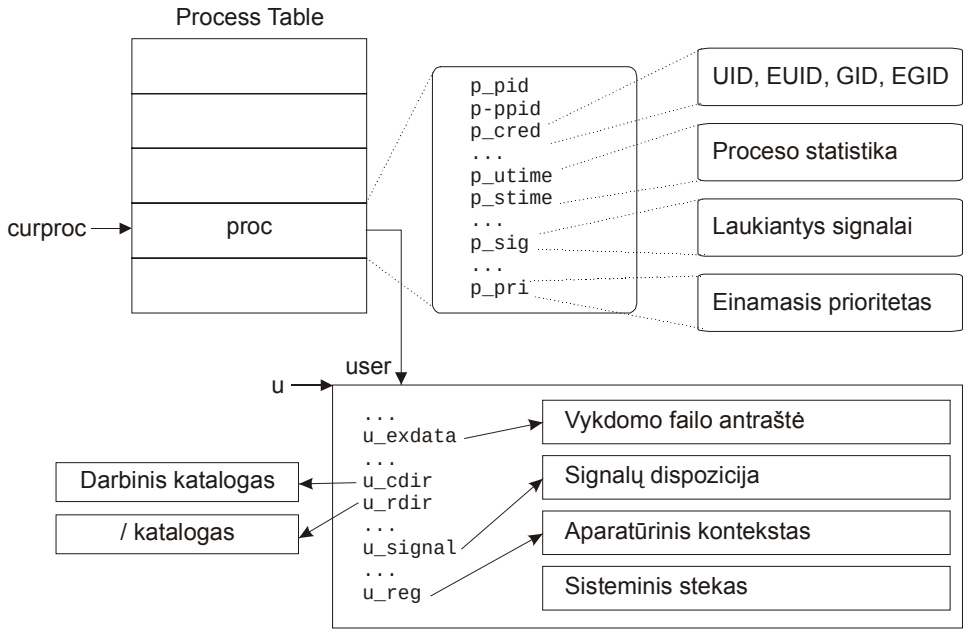
Pradžioje UNIX vienu metu galėjo vykdyti tik du procesus, t.y. po vieną kiekvienam prie PDP-7 prijungtiems terminalams. Vėliau šis procesų skaičius gerokai padidėjo, atsirado sisteminė komanda (system call) `fork(2)`. System 1 jau turėjo `exec(2)` komandą, bet vienu metu galėjo pakrauti tik vieną procesą. Po to, kai į PDP-11 buvo įdiegtas MMU (Memory Management Unit) tapo įmanoma operatyvinėje atmintyje laikyti kelis procesus, taip sumažinant I/O operacijas. Tačiau iki pat 1972, kol UNIX nebuvo perrašytas C kalba, visos I/O operacijos buvo sinchroniškos, t.y. kol vienas procesas nebaigdavo I/O operacijos, visi kiti

privalėjo laukti. Nuo 1973 metų procesų valdymo pagrindai praktiškai nepasikeitė. Proceso vykdymas CPU vyksta dviejuose režimuose:

1. vartotojo režimas (user mode) yra vykdomos programos instrukcijos. Tuo metu procesoriui nėra prieinami sisteminiai resursai. Kai jam prireikia kokio nors sisteminio resurso, tarkim skaitymo iš failo, jis atlieka sisteminę komandą ir pereina į
2. branduolio režimą (kernel mode). Joje yra vykdomos sisteminės komandos, apdorojamas sisteminis iškvietimas. Tokiu būdu vartotojo programa yra izoliuojama nuo sisteminių duomenų struktūrų (negali pakenkti) ir t.t. komandos tegali būti vykdomos branduolio režime, tarkim registro reikšmių keitimas.

Dėl šios proceso dvilypybės, proceso atvaizdas atmintyje yra iš dviejų dalių – užduoties ir branduolio duomenų. CPU dirbdamas branduolio režimu turi priėjimą prie visų duomenų aplamai.

3.2 Proceso duomenų struktūros



3.2 pav. Pagrindinės proceso duomenų struktūros.

Kiekvienas procesas turi dvi pagrindines duomenų struktūras – proc ir user, kurie yra aprašyti failuose <sys/proc.h> ir <sys/user.h>. Šių failų turinys priklauso nuo platformos. Mes peržiūrėsime konkrečiai SCO UNIX (SV i386) duomenų struktūrą proc:

```
char p_stat; // proceso b•sena
char p_pri; // proceso einamasis prioritetas
unsigned int p_flag; // papildoma proceso b•senos informacija
unsigned short p_uid; // UID
unsigned short p_suid; // EUID
int p_sid; // sesijos ID
short p_pgrp; // proces• grup•s identifikatorius
short p_pid; // PID
short p_ppid; // t•vinio proceso PID
sigset_t p_sig; // laukiantys signalai
unsigned int p_size; // proceso adresin•s erdv•s dydis
time_t p_utime; // vartotojo režime vykdytas laikas
time_t p_stime; // branduolio režime vykdytas laikas
caddr_t p_ldt; // proceso LDT nuoroda
struct pregion *p_region; // proceso atminties sri•i• s•rašas
short p_xstat; // t•vui gr•žinamos reikšm•s s•rašas
unsigned int p_utbl[]; // nuoroda • atmintyje esan•i• u-area
```

Visų procesų proc struktūros visada yra operatyviojoje atmintyje, o visos kito, įskaitant ir u-area, g.b. nusiųstos į antrinę atmintį (swap). Naudojant proc branduolys gali prieiti prie visos proceso informacijos.

proc struktūros sudaro procesų lentelę, į kurią rodo kintamasis curproc – CPU vykdomas procesas. Kai tik procesorius yra perduodamas kitam procesui – curproc pakeičiamas.

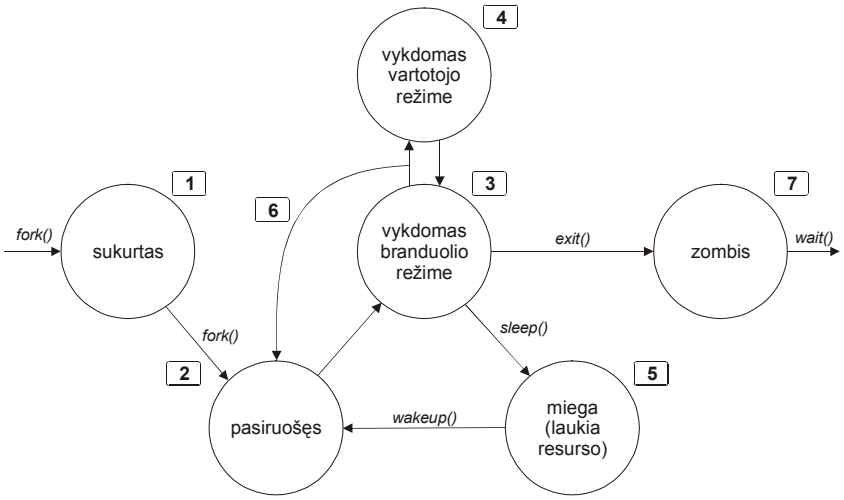
user struktūroje (kitaip u-area, u-block) yra papildoma informacija apie procesą. Ji procesoriui reikalinga tik betarpiškai vykdant procesą. Į ją rodo kintamasis u. u-area saugo sekančią informaciją:

- atidarytų failų deskriptoriai;
- proceso vykdymo statistika;
- kai proceso vykdymas sustabdomas – registrų reikšmės.
- sisteminis stekas. Kai procesas yra vykdomas branduolio režime, CPU naudoja šį steką.

Pagrindines procesų duomenų struktūros yra parodytos 3.2 paveiksle.

3.3 Procesų būsenos

Procesų gyvavimas gali būti suskaidytas į keletą būsenų (3.3 pav.). Perėjimas iš vienos būsenos į kitą įvyksta tada, kai kas nors pasikeičia sistemoje.



3.3 pav. Proceso būsenos.

1. Procesas sukurtas komanda fork(2) ir yra tarpinėje būsenoje. Dar negali būti vykdomas.
2. Procesas yra pasiruošęs vykdymui ir laukia, kol planuotojas jam suteiks procesorių.
3. Procesas yra vykdomas branduolio režimu. Tuo metu procesorius atlieka sistemines operacijas proceso vardu.
4. Procesas vykdo programos instrukcijas – vartotojo režimas.
5. Procesas miega, laukdamas kokio nors resurso, tarkim I/O operacijos pabaigos.
6. Procesui grįžtant iš branduolio į vartotojo režimą planuotojas sugalvojo pakeisti kontekstą, t.y. suteikti procesorių kitam procesui.
7. Procesas iškvietė exit(2) sisteminę komandą ir perėjo į zombio būseną. Lieka įrašai atmintyje, kuriuos sunaikina tėvinis procesas.

Tėvinis procesas naudodamas sisteminę komandą fork(2) sukuria naują procesą. Tada jis stoja į eilę prie procesoriaus. Sisteminė komanda arba pertraukimas nutraukia proceso vykdymą vartotojo režime. Tada procesorius apdoroja OS instrukcijas ir po to:

- procesorius gali būti vėl perduodamas tam pačiam procesui,
- jei procesui trūksta kokio nors resurso, procesas užmigdomas arba,
- jei eilėje prie CPU yra procesas su aukštesniu prioritetu, einamasis procesas statomas į eilę prie procesoriaus.

Konteksto pakeitimas (context switch), tai procesoriaus perdavimas kitam procesui, kai pakeičiama visa proceso aplinka. T.y. išsaugoma einamojo proceso būsena ir vykdymo resursai perduodami kitam procesui. Konteksto pakeitimas įmanomas tik procesui esant vartotojo režime; jei kontekstas butų keičiamas branduolio režime, gali būti pagriautas sistemos vientisumas. Todėl yra baigiamos sisteminės instrukcijos ir, jei reikia, pereinant į vartotojo režimą yra pakeičiamas kontekstas.

4.xBSD, o vėliau ir SVR4 sistemose, realizuotos kelios naujos procesų būsenos. Procesui pasiųstas SIGSTOP, SIGTTIN arba SIGTTOU signalas jį sustabdo. Jei procesas būvo vykdomas arba buvo eilėje prie procesoriaus, jis nedelsiant nusiunčiamas į būseną “sustabdytas” (stopped); jei procesas buvo miegantis, t.y. laukė kokio nors resurso, jis nusiunčiamas į būseną ‘sustabdytas miegantis’. Kad vėl paleisti procesą jam reikia pasiųsti signalą SIGCONT. Procesai pastatomi į tas eiles, kuriose buvo sustabdyti.

3.4 Atminties valdymo principai

Viena pagrindinių OS funkcijų – operatyvinės atminties (RAM) valdymas, nes operatyvinė atmintis yra brangus resursas. Duomenys iš procesoriaus į atmintį keliauja tik kelis CPU taktus, t.y. labai greitai, todėl ten laikyti duomenis yra labai efektyvu. Kadangi pagrindinė atmintis yra ribota, netelpantys duomenys yra saugomi antrinėje atmintyje (swap), kurios vaidmenį dažniausiai atlieka diskas. Diskai yra gerokai lėtesni ir reikalauja papildomos OS priežiūros.

UNIX atlieka efektyvų operatyvinės atminties paskirstymą tarp procesų, pirminės ir antrinės atminties valdymą. Dalį šito darbo atlieka atminties valdymo moduli – MMU (Memory Management Unit).

Primityvios OS į atmintį pakrauna vieną programą ir perduoda jai valdymą. Tokios programos, gavusios visus įmanomus resursus ir naudodamos fizinius atminties adresus veikia labai greitai ir efektyviai. Tokią schemą dažnai naudoja specialios paskirties OS, tačiau bendrosios, tokioms kaip UNIX, toks principas yra visai netinkamas.

Reikalavimai operatyvinės atminties valdymo moduliui:

- uždavinių, didesnių nei atminties kiekis, vykdymas;
- dalinai pakrautų uždavinių vykdymas;
- kelių uždavinių pakrovimas ir efektyvus vykdymas;
- uždavinio pakrovimas į tam tikrą vykdymo sritį;
- uždavinio suskaidymas ir saugojimas įvairiose priminės ir antrinės atminties srityse;
- užtikrinti bendrą atminties sričių naudojimą keliems uždaviniams.

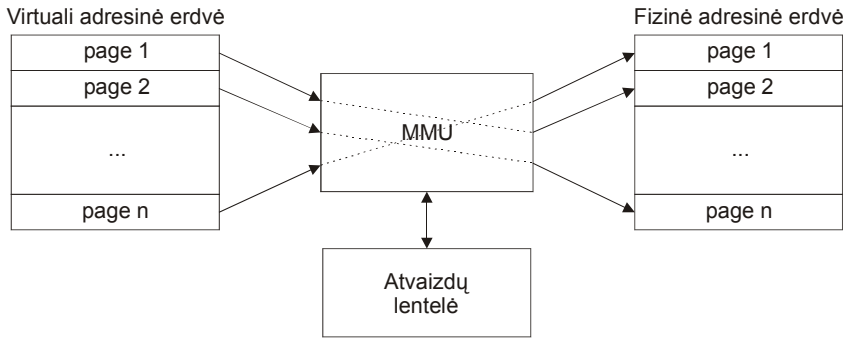
Visa tai realizuojama UNIX OS su virtualios atminties pagalba. Tačiau tai kainuoja: virtualios atminties valdymo mechanizmas užima atmintį, reikalauja procesoriaus ir ilgai trunkančių I/O operacijų. Vidutinio apkrovimo sistemose apie 7% procesoriaus laiko užima virtualios atminties valdymo mechanizmo realizacija. Todėl nuo efektyvaus jo darbo priklauso ir visos OS efektyvumas.

Fizinis adresas yra vienas iš nuoseklaus baitų sąrašo (fizinės atminties) elementų. Jei procesas tiesiogiai naudotų fizinį adresą, tai:

- negalima būtų apsaugoti t.t. atminties sričių;
- kompiliacijos metu reikėtų tiksliai žinoti kur ir kas bus padėta;
- adresinė erdvė apsiribotų atminties kiekiu. Tarkim 8Mb RAM turinčioje mašinoje butų galima pakrauti iki 8 procesų po 8Mb kiekvienas, kai šiuolaikinėse sistemose paleidžia vidutiniškai po 80 procesų, kurių atminties erdvė siekia po kelis Gb.

Šiuolaikinėse sistemose kiekvienas procesas dirba savo virtualios atminties erdvėje ir jam susidaro įspūdis, kad visa atmintis priklauso tik jam vienam ir jos yra tikrai pakankamai. Procesai tampa izoliuoti vienas nuo kito.

Virtualių adresų transliaciją į fizinius atlieka MMU aparatūriniame lygyje. Šiuolaikiniai procesoriai palaiko nuoseklius kintamo dydžio atminties plotus – segmentus ir fiksuoto dydžio plotus – puslapius. Kiekvienam segmentui arba puslapiui yra sukuriama sava virtuali adresinė erdvė (3.4 pav).



3.4 pav. Virtuali ir fizinė atminties erdvė.

Virtualios atminties gali būti gerokai daugiau nei fizinės. Jei fizinėje atmintyje nebelieka vietos, dalis duomenų numetama į antrinę atmintį – swap'ą. Jei kreipiantis tam tikru adresu pasirodo, kad to fizinio puslapyje atmintyje nėra, generuojamas pertraukimas page fault, nustatoma kurioje antrinės atminties vietoje yra reikiamas puslapis, jis pakraunamas į pirminę atmintį, pakeičiami atvaizdų lentelės parametrai ir paprašoma procesoriaus pakartoti operaciją. Pats virtualių adresų transliacijos į fizinius procesas priklauso nuo aparatūrinės platformos. Panagrinėsime SCO UNIX i386 atvejus.

3.4.1 Segmentai

Segmentai yra loginiai kintamo dydžio atminties gabalai, sudaryti iš nuoseklios adresinės erdvės, esančios duotajame diapazone. Virtualūs adresai yra tiesiogiai transliuojami į nuoseklią tokio pat dydžio fizinių adresų sritį.

Virtualų adresą sudaro dvi dalys: segmento selektorius (segment selector) ir poslinkis (offset). Segmento selektorius rodo į segmento deskriptorių (segment descriptor), kuris turi tokius parametrus kaip fizinės atminties adresas, dydis ir priėjimo teisės.

i386 palaiko netiesioginį adresavimą per segmento deskriptorius, kurie yra saugomi specialiose lentelėse, t.y. tam tikrose atminties vietose, į kurias rodo specialūs procesoriaus registrai. OS branduolys yra atsakingas už lentelių užpildymą ir registro reikšmių nustatymą, t.y. operacinė sistema atsako už atvaizdą, o aparatūra – už transliaciją.

3.5 pav. Segmento selektorius.

Segmento deskriptoriai kiekvieno proceso atžvilgiu yra dviejose lentelėse – Local Descriptor Table (LDT) ir Global Descriptor Table (GDT). LDT transliuoja virtualius proceso programos,

vartotojo režimo adresus, o GDT – branduolio režimo adresus. LDT yra sukuriama kiekvienam procesui sava, o GDT yra bendra visiems.

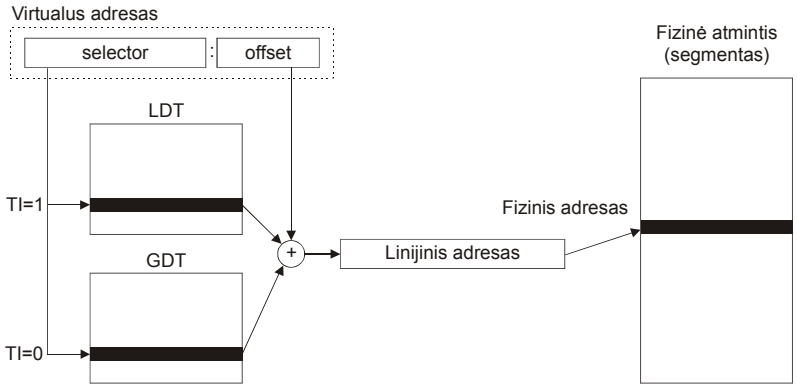
3.5 paveiksle yra pateiktas segmento selektorius. TI =0 – GDT selektorius; TI=1 – LDT selektorius; RPL – segmento privilegijos. Jei procesas vartotojo režime bandys pasiekti GDT lentelę, jis gaus SIGSEGV signalą.

Kiekvienas LDT ir GDT lentelės įrašas yra segmento deskriptorius. Yra įvairių segmentų deskriptorių, naudojamų programos kodo, duomenų ir steko segmentams.

Segmento deskriptorių sudaro šie laukai:

- bazinis adresas – 32 bitų fizinių adresų erdvės pradžia;
- segmento dydis – jei gautas fizinis adresas viršija šią ribą – generuojama klaida. Kai kurios OS tiesiog padidina segmentą ir paprašo įvykdyti komandą iš naujo.
- privilegijos (DPL – Descriptor Privilege Level). Norint t.t. selektoriui su t.t. RPL pasiekti segmentą, reikia kad RPL≥DPL.
- būvimo pagrindinėje atmintyje požymis. jei jis yra lygus 0, CPU generuoja specialią situaciją ir pakrauna segmentą iš swap'o.
- segmento tipas. Mechanizmas neleidžia duomenų segmento traktuoti kaip kodo segmento.
- naudojimo teisės – segmentas gali būti prieinamas tik skaitymui arba skaitymui ir rašymui.

Visas atminties segmentų valdymo mechanizmas parodytas 3.6 paveiksle.



3.6 pav. Segmentų valdymo mechanizmas.

3.4.2 Puslapių mechanizmas

Segmentų mechanizmas nėra tobulas, nes reikalauja, kad visas segmentas butų pakrautas į atmintį, o segmentai gali būti dideli ir, vykdant kelis procesus, kyla didelė konkurencija dėl atminties, vykdoma daug lėtų I/O operacijų.

Puslapinis mechanizmas yra žymiai lankstesnis. Visa virtuali adresinė erdvė (4 Gb i386 procesoriams) yra suskirstoma į vienodo dydžio blokus, puslapius. Daugelis Intel procesorių dirba su 4K dydžio puslapiais. Kaip ir segmentų atveju puslapis gali būti operatyvinėje atmintyje, swap'e arba faile. Puslapis yra pakankamai nedidelė atminties sritis, todėl jo pagalba yra efektyviau dalinama atmintis tarp procesų. Dalis segmento gali būti atmintyje, o dalis – swap'e. Branduolys pagrindinėje atmintyje palieka tik tuos puslapius, kurie yra betarpiškai reikalingi duotuoju laiko momentu. Virtuali adresinė erdvė yra nuosekli, o fiziniai puslapiai – nebūtinai.

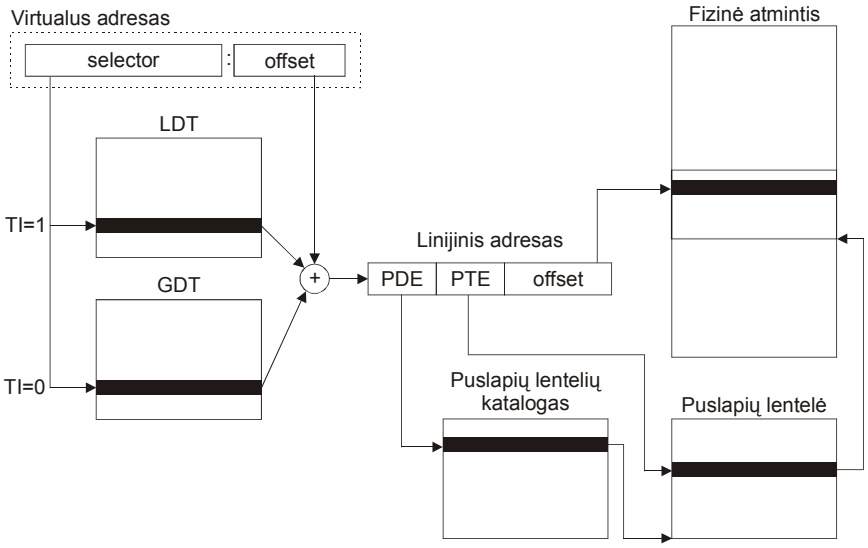
Naudojant puslapių mechanizmą linijinis adresas, gautas sudėjus bazinį adresą ir poslinkį dar papildomai apdorojamas puslapių mechanizmo modulio (3.7 pav.).

Pirmasis adreso laukas PDE (Page Directory Descriptor, 22 – 31 bitai) rodo į lentelių katalogo įrašą. Šis katalogas užima vieną puslapį, t.y. turi iki 1024 nuorodų į puslapių lenteles. Kitas laukas – PTE (Page Table Entry, 12 – 21 bitai) rodo į puslapių lentelės įrašą, t.y. konkretaus puslapio adresą. Likę 11 bitų yra naudojami konkrečiam poslinkiui puslapyje identifikuoti.

Taigi, procesas, naudodamas vieną lentelių katalogą gali operuoti 1024x1024X4096=4Gb fizine atmintimi.

Kiekvienas puslapių lentelės elementas turi šiuos laukus:

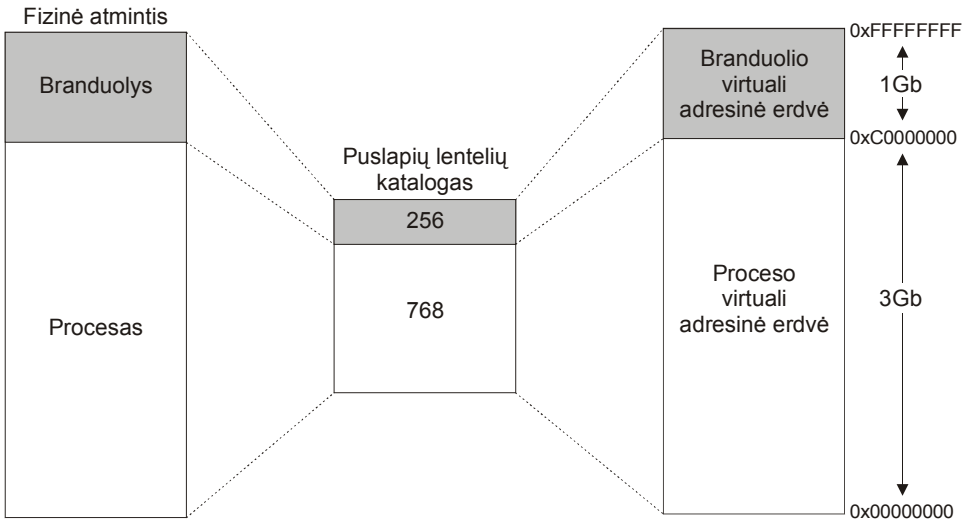
- P – bitas, identifikuojantis puslapio buvimą atmintyje;
- R/W – tik skaitymui (0) arba ir rašymui (1);
- U/S – naudojamas tik branduolio režime (0) arba ir vartotojo režime (1);
- Adresas – fizinis (bazinis) adresas.



3.7 pav. Puslapių mechanizmas.

3.5 Proceso adresinė erdvė

Branduolio adresinė erdvė dažniausiai sutampa su duotuoju momentu vykdomo proceso adresine erdve, ir, tokiu atveju sakoma, kad branduolys yra tame pačiame kontekste kaip ir procesas. Kiekvieną kartą, kai procesas gauna procesorių, yra sakoma, kad atstatomas proceso kontekstas, t.y. bendro naudojimo ir segmentinių registrų reikšmės bei nuorodos į



3.8 pav. Proceso adresinė erdvė.

puslapių lenteles, atvaizduojančias vartotojo režimo virtualią atmintį. Proceso adresinė erdvė pateikta 3.8 paveiksle.

Specialus registras (Intel – CR3) rodo į puslapių lentelių katalogą atmintyje. SCO UNIX yra naudojamas vienintelis katalogas, todėl CR3 registro reikšmės nekinta viso sistemos gyvavimo ciklo metu. Branduolio kodas ir duomenys yra proceso dalis, tai puslapių lentelės dalis, atvaizduojanti vyresnįjį 1Gb virtualios atminties nėra keičiama keičiant proceso kontekstą. Branduolio atvaizdui yra naudojami vyresnieji 256 katalogo elementai.

Keičiant kontekstą yra keičiami kiti 768 katalogo elementai, t.y. proceso adresinė erdvė. Viso jie atvaizduoja 3Gb virtualios atminties. Tokiu būdu pakeitus kontekstą, naujojo proceso adresinė erdvė tampa matoma, o senojo – ne.

Vartotojo režimo virtualios atminties formatas priklauso nuo vykdomojo failo formato, tačiau nepriklausomai nuo to, ji negali viršyti 3Gb. Užduoto proceso atminties atvaizdo vartotojo režime keisti negalima; tokiu būdu yra apsisaugoma kito proceso atminties erdvė. Tačiau dirbdamas privilegijuotame branduolio režime procesas gali šį atvaizdą keisti.

3.6 Vykdomų failų formatai

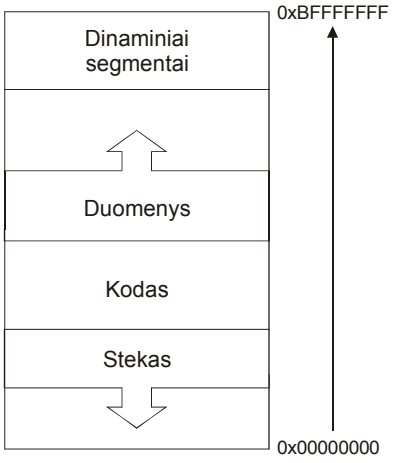
Dauguma šiuolaikinių UNIX OS naudoja COFF (Common Object File Format) ir ELF (Executable and Linking Format). Vykdomų failų formatai atsako į eilę klausimų, tarkim:

- kokias programos dalis būtina pakrauti į atmintį?
- kaip yra sukuriama neapibrėžtų duomenų sritis?
- kokios proceso dalys turi būti nuskaitytos į swap sritį, o kokios gali likti faile?
- kur atmintyje yra programos instrukcijos ir duomenys?
- kokios bibliotekos yra būtinos programos vykdymui?
- kaip yra susiję vykdomas failas diske, programos atvaizdas atmintyje ir diskinė swap sritis?

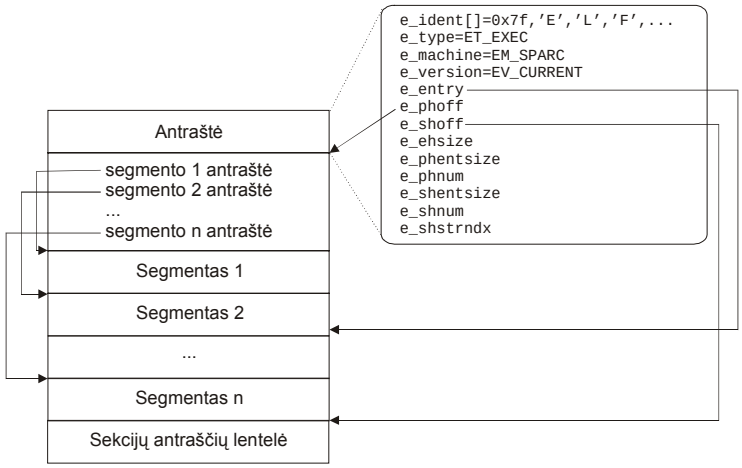
3.6.1 ELF formatas

ELF failo atvaizdas proceso virtualioje adresinėje erdvėje pateikiamas 3.9 paveiksle. ELF failo standartas gali būti kelių tipų:

- pernešamas failas (relocatable file), kuriame yra duomenys ir instrukcijos, kurie vėliau gali būti sujungti su kitais objektiniais failais. Tokio sujungimo rezultatas gali būti bendras objektinis arba vykdomasis failas;
- bendrame objektiniame faile (shared object file) taip pat yra saugomos instrukcijos ir duomenys, tačiau jis gali būti panaudotas dvejopai. Pirma, jis gali būti sujungtas su kitu bendruoju objektiniu arba pernešamu failu ir tokiu būdu gaunamas naujas objektinis failas. Antra, operacinei sistemai paleidžiant programą jis gali būti dinamiškai susiejamas su vykdomuoju failu, ko pasėkoje bus sukurtas vienas vykdomos programos atvaizdas – procesas. Šiuo atveju, kalbama apie bendras bibliotekas.
- vykdomas failas saugo pilną informaciją, reikalingą procesui sukurti, t.y. instrukcijas, duomenis, bendrų objektinių bibliotekų aprašymą ir būtiną valdymo informaciją.



3.9 pav. ELF failo atvaizdas proceso adresinėje erdvėje.



3.10 pav. ELF vykdomojo failo formatas.

Paveiksle 3.10 yra pateikiama vykdomojo failo struktūra, kurios pagrindu OS gali sukurti programos atvaizdą – procesą ir jį paleisti. Failo antraštė yra fiksuotoje failo vietoje, o kiti komponentai gali būti rasti pagal antraštėje esančią informaciją. Antraštę sudaro šie laukai:

- e_ident[] – baitų masyvas, suteikiantys bendrąją informaciją apie failą, t.y. failo formatas (ELF), versijos numeris, sistemos architektūra (pvz. 16 arba 32) ir pan.;
- e_type – kadangi ELF palaiko keletą failo tipų, šis laukas rodo failo tipą;
- e_machine – aparatūrinės platformos architektūra, kuriai yra sukurtas šis failas (pvz. EM_SPARC, EM_386);
- e_entry – virtualus adresas, kuris bus perduotas OS vykdymui po programos pakrovimo;
- e_phoff – programos segmentų antraščių adresas faile (poslinkis nuo failo pradžios);
- e_shoff - programos sekcijų antraščių adresas faile (poslinkis nuo failo pradžios);
- e_ehsize – failo antraštės dydis;
- e_phentsize – kiekvienos segmento antraštės dydis;
- e_phnum – segmentų skaičius;
- e_shentsize – kiekvienos sekcijos antraštės dydis;
- e_shnum – sekcijų skaičius.

Segmentų antraščių lentelėje yra saugoma informacija, kuri nurodo OS branduoliui, kaip reikia sukurti procesą. Daugelis segmentų yra tiesiog kopijuojami (atvaizduojami) į atmintį ir tampa tam tikrais proceso segmentais, tarkim duomenų arba kodo segmentais. Kiekviena segmento antraštė aprašo vieną iš segmentų ir pateikia tokią informaciją:

- segmento tipas ir OS veiksmai su šiuo segmentu;
- segmento vieta faile;
- segmento pradžios adresas proceso virtualioje atmintyje;
- segmento dydis faile;
- segmento dydis atmintyje;
- segmento leidimai – skaitymui, rašymui, vykdymui.

Dalis segmentų yra LOAD tipo, nurodantys branduoliui proceso kūrimo metu sudaryti specialias duomenų struktūras, t.y. atitinkamo tipo nuoseklias atminties sritis proceso virtualioje erdvėje ir sudėti tam tikrus atributus. Tokiu atveju visas segmentas bus atvaizduotas pradedant segmento antraštėje nurodytu adresu į virtualią proceso erdvę. Jei tokio tipo segmentai, tarkim kodo ar duomenų, yra mažesni nei adresų sritis, tai laisva erdvė užpildoma nuliais.

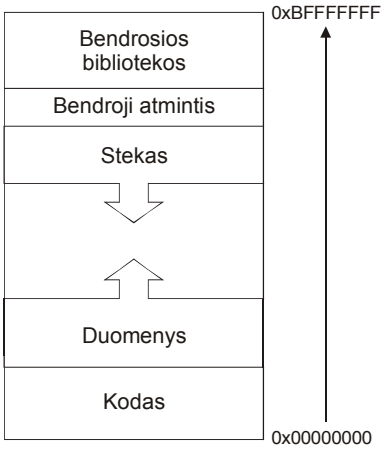
INTERP tipo segmente yra patalpintas programinis interpretatorius, kuris yra naudojamas dinaminiam programų sujungimui. Tokio sujungimo esmė yra ta, kad išorinė biblioteka yra pajungiama ne kompiliacijos, o vykdymo metu. Pradžioje į atmintį yra pakraunama ne programa, o dinaminis ryšių redaktorius, kuris, kartu su OS branduoliu sukuria pilną vykdomos programos pavidalą. Dinaminis redaktorius pasirūpina, kad būtų pakrauti ir sujungti reikalingi, įvairiuose segmentuose nurodyti bendrieji objektiniai failai. Po to valdymas yra perduodamas programai.

Sekcijos nurodo tam tikros paskirties atminties sritis, reikalingas programos vykdymo metu. Tarkim, jei kodo segmentui reikės hash lentelės, ji bus nurodoma specialia sekcija.

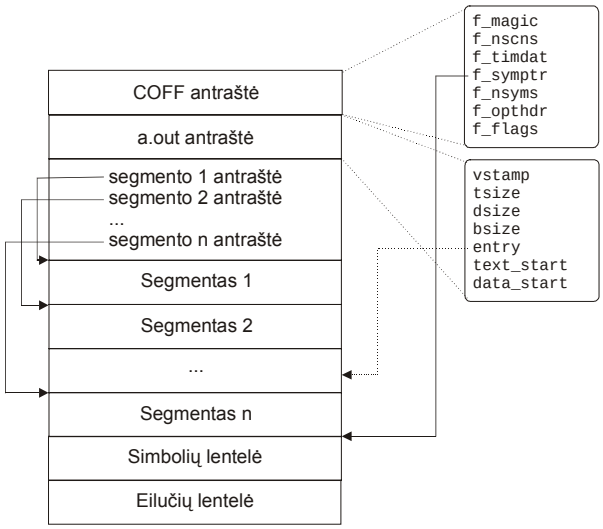
3.6.2 COFF formatas

COFF failo atvaizdas proceso virtualioje adresinėje erdvėje pateikiamas 3.11 paveiksle, o failo formatas – 3.12 paveiksle.

Paveiksle COFF formatą sudaro dvi pagrindinės antraštės – COFF antraštė ir standartinė UNIX a.out antraštė. Toliau seka segmentų antraštės ir patys segmentai. Faile yra saugomi tik inicijuoti duomenys, nes neinicijuotiems duomenims saugoti pakanka pradinio atminties adreso ir dydžio; visa ši erdvė užpildoma nuliais. Simbolinę informaciją sudaro simbolių ir eilučių lentelės. Pirmojoje saugomi simboliai, jų adresai ir tipai. Tarkim, simbolis locptr yra nuoroda ir jo virtualus adresas yra 0x7feh0. Toliau programa naudoja duotąjį adresą, siekdama išsiaiškinti simbolio reikšmę. Simbolių lentelė turi fiksuotą ilgį. Jei simbolio ilgis viršija 8 – jo vardas yra saugomas simbolių lentelėje. Paprastai šios lentelės visada yra vykdomuose failuose, nebent yra išmetamos specialiai (su komanda strip(1)).



3.11 pav. COFF failo atvaizdas proceso adresinėje erdvėje.



3.12 pav. COFF vykdomojo failo struktūra

Simboliai ir eilutės yra bet kuris programos kintamasis, funkcija arba žymė. Su šia informacija galima nustatyti bet kurio simbolio virtualų adresą. Šią informaciją naudoja įvairios programavimo (debug) primonės, pvz. ps(1) programa naudoja šią informaciją.

Kaip ir ELF formato atveju, pateiksime antraščių struktūrą, pradžioje COFF antraštė:

- f_magic – failo aparatinė platforma;
- f_nscns – segmentų skaičius faile;
- f_timdat – failo sukūrimo data ir laikas;
- f_symptr – simbolių lentelės vieta faile;
- f_nsyms – simbolių lentelės dydis;
- f_opthdr – a.out antraštės dydis;
- f_flags – bendra failo informacija.

Programos paleidimui ir jos atvaizdo procese konstravimui yra būtina a.out antraštė:

- vstamp – antraštės versijos numeris;
- tsize – kodo (text) segmento ilgis faile;
- dsize – inicijuotų duomenų (data) ilgis faile;
- bsize – neinicijuotų duomenų (bss) ilgis faile;
- entry – programos startavimo taškas;
- text_start – kodo segmento adresas proceso virtualios atminties erdvėje;
- data_start – duomenų segmento adresas proceso virtualioje erdvėje.

3.7 Proceso atminties valdymas

Virtualios atminties realizacijai ir valdymui didelę įtaką turi aparatūrinė įranga. OS turi rūpintis šiais uždaviniais:

- puslapių lentelių ir jų katalogo pakrovimas į operatyvinę atmintį ir registro, rodančio į lentelių katalogą (Intel - CR3) inicijavimas. Sistemose, kurios naudoja daugiau nei

vieną puslapių lentelių katalogą, šio registro reikšmė yra saugoma u-area struktūroje. Pakeičiat kontekstą ši reikšmė yra atstatoma.

- reikšmių į puslapių lentelę įrašymas, keitimas.
- page fault pertraukimo realizavimas, t.y. apsikeitimo puslapiais tarp pirminės ir antrinės atminties realizavimas.
- laikinojo saugojimo buferio (cache) realizavimas.
- klaidų apdorojimas.

Efektyvus šių uždavinių vykdymas didžiąja dalimi priklauso nuo atminties atvaizdų duomenų struktūrų. Jų formatas priklauso nuo platformos architektūros ir UNIX versijos. Nagrinėsime SCO UNIX architektūrą.

3.7.1 Atminties sritys

Proceso adresinė erdvė yra suskaidyta į kelias dalis, vadinamas sritis (region). Sritis, tai nuosekli adresinė erdvė, kuri OS yra traktuojama atskiru objektu, naudojamu bendrai arba apsaugotu nuo pašalinio priėjimo. Sritis gali būti naudojama įvairių duomenų saugojimui, pvz. kodui, duomenims, bibliotekų kodui, bibliotekų duomenims ir pan. Kiekviena aktyvi atminties sritis yra laikoma specialia OS branduolio duomenų struktūra ir naudojama atminties valdymui.

Kiekviena sritis yra laikoma atskiru duomenų segmentu, kuris, kartu su puslapiniu mechanizmu padeda efektyviau valdyti atmintį.

Sritys gali būti kelių procesų naudojamos bendrai; tokiu atveju branduoliui nėra būtina kiekvienam procesui kurti atskirų sričių, tiesiog sukuriamas atitinkamas atvaizdas (srities virtualūs adresai gali ir nesutapti). Tokiu pavyzdžiu gali būti bendra atmintis, bendros bibliotekos ir failų atvaizdai atmintyje. Informacija apie kiekvieną aktyvią sritį yra saugoma `region` struktūroje.

Kadangi kiekvieną sritį gali naudoti keli procesai, tai kiekvienam procesui yra sukuriamas nuorodų į `region` struktūros – `pregion`. Nuoroda į pirmąjį `pregion` sąrašo elementą yra saugoma `proc` struktūroje. Struktūrų organizacija yra parodyta 3.13 paveiksle.

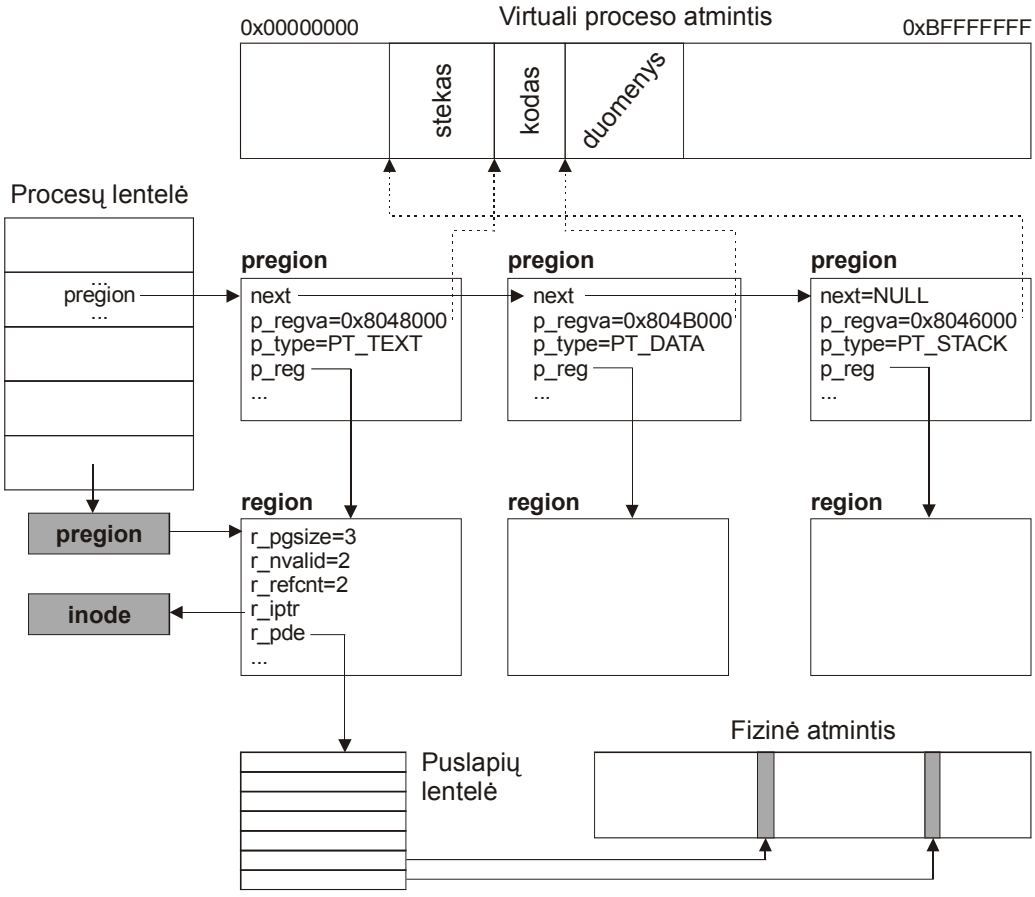
Struktūroje `pregion` yra saugomi ne tik nuoroda į `region` struktūrą ir nuoroda į kitą `pregion` elementą, bet ir įvairius privilegijų duomenis, blokavimo režimus ir pan. Laukas p_type nurodo srities tipą, kurių galimos reikšmės yra pateiktos 3.1 lentelėje.

3.1 pav. `pregion` struktūroje nurodomas `region` panaudojimo tipas.

Reikšmė	Aprašymas
`PT_UNUSED`	Sritis nenaudojama
`PT_TEXT`	Kodo segmentas
`PT_DATA`	Duomenų segmentas
`PT_STACK`	Vartotojo režimo stekas
`PT_SHMEM`	Bendrai naudojama atmintis
`PT_LIBTXT`	Bibliotekų kodo segmentas
`PT_LIBDAT`	Bibliotekų duomenų segmentas
`PT_SHFIL`	Failo atvaizdo atmintyje segmentas

`p_regva` nurodo srities pradžios adresą proceso virtualios atminties erdvėje.

`region` struktūros laukų reikšmės: `r_pgsz` – srities puslapių skaičius iš kurių `r_nvalid` yra pagrindinėje atmintyje; `r_refcnt` saugo skaičių procesų, kurie naudoja duotą atminties sritį; `r_pde` rodo į srities puslapių lentelę (jei vienos puslapių lentelės neužtenka, reikia >4Mb, tai yra saugomas puslapių lentelių katalogo elementų sąrašas); `r_iptr` rodo į duomenų failo `i-node`, tarkim kodo segmento `r_iptr` rodys į vykdomo failo `i-node`.



3.13 pav. Proceso sričių duomenų struktūros

3.7.2 crash(1M) – informacija apie proceso atminties sritis

Su komanda `crash(1M)` galima gauti informaciją apie UNIX OS būseną ir tuo pačiu, apie procesų atminties struktūras. Pateiksime vieną pavyzdį:

```
# crash
dumpfile = /dev/mem, namelist = /unix, outfile = stdout
> pregion 101
SLOT   PREG   REG#   REGVA         TYPE   FLAGS
101    0      12     0x700000      text   rdonly
       1      22     0x701000      data
       2      23     0x7fffffc     stack
       3      145    0x80001000    lbtxt  rdonly
       4      187    0x80031000    lbdat  pr
```

Kaip matome, duotasis procesas (PID=101) naudoja 5 sritis – kodo, duomenų, steko ir bibliotekos: kodo bei duomenų. `REG#` rodo į konkretų sričių lentelės įrašą, tai yra sritį. Tikrasis `pregion` struktūros laukas saugo nuorodą į `region` struktūra. Naudojant šią informaciją galime sužinoti `region` struktūros duomenis:

```
> region 12 22 23
SLOT   PGSZ   VALID  SMEM   REF    NSW    FORW   BACK   INOX   TYPE   FLAGS
12     1      1      1      11     0      15     5      154    stxt   done
22     3      1      0      1      0      238    23     154    priv   done
23     2      1      1      1      0      135    24            priv   stack
```

`PGSZ` – srities dydis puslapiais, `VALID` – puslapių, esančių pagrindinėje atmintyje skaičius. `INOX` laiko nuorodas į inode lenteles, t.y. į failų meta-duomenis iš kurių buvo pakrauti šie segmentai. Galime rasti daugiau informacijos apie šį failą:

```
> inode 154
```

```
INODE TABLE SIZE = 472
SLOT   MAJ/MIN      FS     INUMB  RCNT   LINK   UID    GID    SIZE   MODE
154    1,42         2      1562   3      1      123    56     8972   f---755
```

`MAJ/MIN` – failų sistema, kurioje yra failas ir jo diskinio inode numeris – `INUMB`. Su komanda `ncheck(1)` galima sužinoti vykdomojo failo vardą:

```
$ ncheck –i 1562
/de/root:
1562          /home/valdo/bin/test
```

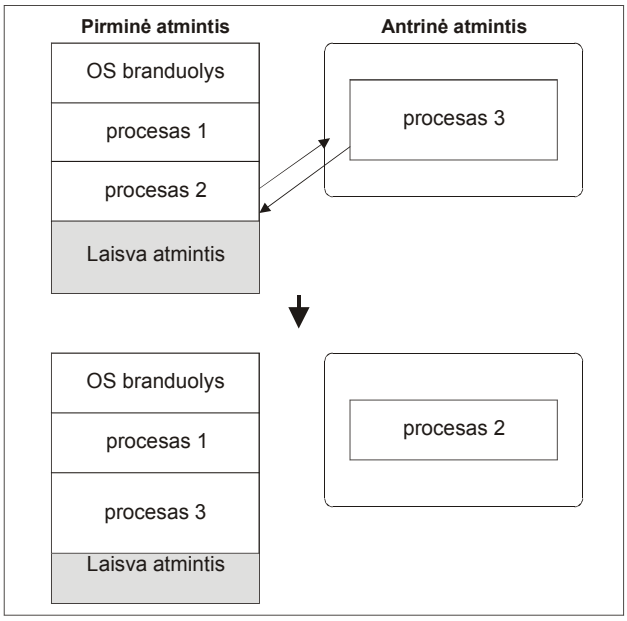
3.7.3 Puslapių pakeitimas

Naudojant tik atminties segmentus procesai į operatyvinę atmintį turi būti pakraunami pilnai, t.y. jie gali būti numetami pilnai į swap'ą arba pakraunami atgal. Tokiu atveju tik nedidelį kiekį procesų galime vienu metu pakrauti į atmintį (4.14 pav.).

Pirmasis puslapių pakeitimo mechanizmas, kuris leidžia pakrauti į operatyvinę atmintį procesus iš dalies, buvo įdiegtas 1978 metais VAX-11/780 mašinoje, kuri jau turėjo 32 bitų architektūrą, 4Gb proceso adresinę erdvę ir aparatūrinį virtualių adresų palaikymo mechanizmą. Pirmoji tokią puslapių pakeitimo sistemą turėjo 3.xBSD OS.

Konkretus puslapių pakeitimo mechanizmas priklauso nuo trijų pagrindinių principų realizacijos:

1. fetch policy – esant kokioms sąlygoms sistema pakrauna puslapį į atmintį;
2. placement policy – kokiose atminties vietose yra pakraunamas puslapis;
3. replacement policy – kokiu būdu sistema parenka puslapį, kurį reikėtų numesti į antrinę atmintį.



3.14 pav. Segmentinis atminties pakeitimas

Paprastai visos laisvos atminties sritys vienodai tinka puslapio patalpinimui, todėl patalpinimo principas (placement policy) neturi esminės įtakos mechanizmui. Tokiu būdu kitų dviejų principų – išrinkimo ir pakeitimo – daugiausia ir priklauso mechanizmo efektyvumas. Paprastuose puslapiniuose mechanizmuose yra naudojamas elementarus išrinkimo principas – jei puslapis yra betarpiškai reikalingas – jis pakraunamas. Tačiau šiuolaikinėse UNIX sistemose sistema pakrauna kelis puslapius iškarto. Tai yra tie puslapiai, kurių panaudojimas labiausiai tikėtinas duotajame kontekste.

Šis mechanizmas leidžia procesui naudoti žymiai didesnę nei fizinė virtualią adresinę erdvę. Nuo proceso yra paslepiama tikroji fizinių puslapių būvimo vieta:

- Virtualus adresas gali būti susietas su fizinės atminties puslapiu. Tokiu atveju bus tiesiogiai dirbama, nereikės papildomų OS operacijų.
- Puslapis gali būti numestas į antrinę atmintį (swap'ą). Tai įvyksta tuo atveju, kai reikia atlaisvinti pagrindinę atmintį kitam procesui. Jei virtualusis adresas kreipsis į puslapį, kuris tuo metu yra swap'e, sistema sugeneruos page fault pertraukimą, kuris, savo ruožtu, iššauks šio puslapio pakrovimą į pagrindinę atmintį, puslapių atvaizdų lentelės pakeitimus ir paprašys programos pakartoti operaciją. Jei kada nors vėliau reikės permesti šį puslapį permesti į antrinę atmintį, tai bus padaryta tik tuo atveju, jei po paskutinio pakrovimo puslapis buvo modifikuotas.
- Adresuojamas puslapis yra diske – faile. Tipiniu tokiu atveju yra kodo segmentas, kai programa skaitoma iš failo arba naudojama failų atvaizdų atmintyje sritis. Jei virtualus adresas kreipsis į šį puslapį, bus sugeneruotas page fault pertraukimas ir OS pakraus reikiamą puslapį iš failo.
- Adresuojamo puslapio nėra nei faile, nei atmintyje. Tipinis atvejis – neinicijuoti duomenys. Jei programa kreipsis tokiu adresu, OS sukurs naują puslapį atmintyje.

Puslapinis pakeitimo mechanizmas turi daug privalumų palyginus su segmentiniu:

- Programos dydis yra ribojamas tik virtualios proceso atminties dydžiu – 4Gb 32 bitų architektūros mašinose.
- Programos paleidimas vyksta labai greitai, nes nereikia pakrauti visos programos.
- Vienu metu gali būti pakrauta daug programų ir jos gali būti vykdomos vienu metu.
- Puslapių permetimas iš pagrindinės atminties į antrinę ir atgal reikalauja žymiai mažiau resursų, nes permetama ne visa programa, o tik tam tikras puslapis.

3.8 Procesų vykdymo planavimas

Procesorius, kaip ir operatyvinė atmintis yra labai brangus resursas, todėl jo efektyvus dalinimas procesams taip pat yra svarbus uždavinys. UNIX yra paskirstyto laiko operacinė sistema, tai reiškia, kad kiekvienam procesui skaičiavimo resursai yra suteikiami tam tikram laiko intervalui, po kurio CPU perduodamas kitam procesui. Maksimalus laiko intervalas, kuriam gali būti suteiktas procesorius, vadinamas laiko kvantu (time quantum arba time slice). Tokiu būdu sukuriama iliuzija, kad keli procesas dirba vienu metu, nors faktiškai vienu metu dirba vienintelis procesas.

Skirtingos programos kelia skirtingus uždavinius operacinei sistemai iš procesų planavimo požiūrio:

- Interaktyvios programos. Šiai klasei priskiriami teksto redaktoriai, komandų interpretatoriai ir kitos programos, tiesiogiai bendraujančios su sistemos vartotoju. Tokios programos daugiausia laiko praleidžia laukdamos vartotojo įvedimo, tarkim, klavišo nuspaudimo arba veiksmo pele. Tačiau, gavus įvedimą, jos turi greitai jį apdirbti, kitaip dirbant su aplikacija nebus komforto. Leidžiamas sistemos reakcijos laikas yra 100 – 200 milisekundžių.
- Foninės aplikacijos. Šiai klasei priklauso programos, kurių darbui nereikia vartotojo įsiterpimo, pvz. kompiliatoriai, skaičiavimo programos ir pan. Šių programų didžiausias reikalavimas greitas skaičiavimas, kad darbas neužtruktų pernelyg ilgai.
- Realaus laiko programos. Šios programos reikalauja papildomų UNIX savybių, kurios garantuotų tam tikros operacijos atlikimo reikiamu laiku. Tarkim video programos reikalauja, kad kadras būtų perpiešiamas tam tikru metu.

Procesų planavimas remiasi t.t. taisyklėmis, remiantis kuriomis procesorius yra perduodamas vienam ar kitam procesui. Anksčiau minėtos programos kelia daug reikalavimų, tačiau šiuolaikinės UNIX sistemos neatsižvelgia į jas visas, o stengiasi rasti aukso vidurį.

3.8.1 Taimerio pertraukimų apdorojimas

Kiekvienas kompiuteris turi aparatūrinį taimerį arba sisteminį laikrodį, kuris kas nustatytą laiko intervalą generuoja aparatūrinius pertraukimus. Laiko intervalas tarp pertraukimų yra vadinamas procesoriaus tiku arba paprastai tiku (CPU tick, clock tick). Laiko intervalas gali būti įvairus, tačiau UNIX sistemose jis dažniausiai yra lygus 10 milisekundžių. Dauguma UNIX OS šią reikšmę saugo `HZ` konstantoje, kuri yra nustatyta faile `<param.h>`. Tarkim 10 sekundžių tikui `HZ=100`.

Taimerio pertraukimų apdorojimas gali būti įvairus, tačiau mes peržiūrėsime bendruosius apdorojimo principus. Taimerio pertraukimo prioritetas yra pats aukščiausias, todėl jo apdorojimas turi būti labai efektyvus ir užimti nedidelę laiko dalį. Bendruoju atveju apdorojimas susideda iš šių uždavinių:

- Einamojo proceso CPU naudojimo statistikos atnaujinimas.
- Funkcijų, susijusių su procesų planavimu įvykdymas, tame tarpe prioritetų perskaičiavimas ir laiko kvanto, einamajam procesui tikrinimas.
- Jei procesas viršija nustatytą laiko kvantą, jam nusiunčiamas SIGXCPU signalas.
- Sisteminio laiko atnaujinimas (datos ir laiko) ir kitų, su laiku susijusių uždavinių įvykdymas.
- Atidėtų iškvietimų (callout) apdorojimas.
- Aliarmų (alarm) apdorojimas.
- Jei reikia, sisteminių procesų, tokių kaip puslapių dispečerio arba swapper'io, prikėlimas.

Dalis šių uždavinių gali būti kiekvieno tiko metu, o kas tam tikrą tiką, vadinamo pagrindinio tiko metu (major tick), kuris įvyksta kas n tikų. n svyruoja priklausomai nuo platformos ir UNIX versijos, pvz. 4.3BSD – kas 4 tikai, o SVR4 – kartą į sekundę.

3.8.2 Atidėti iškvietimai (callout)

Atidėtas iškvietimas yra funkcija, kuri bus iškviesta po tam tikro laiko. Tarkim SVR4 kiekviena branduolio posistemė gali užregistruoti callout'ą naudojant sekančią funkciją:

```
int co_ID = timeout(void (*fn) (), caddr_t arg, long delta);
```

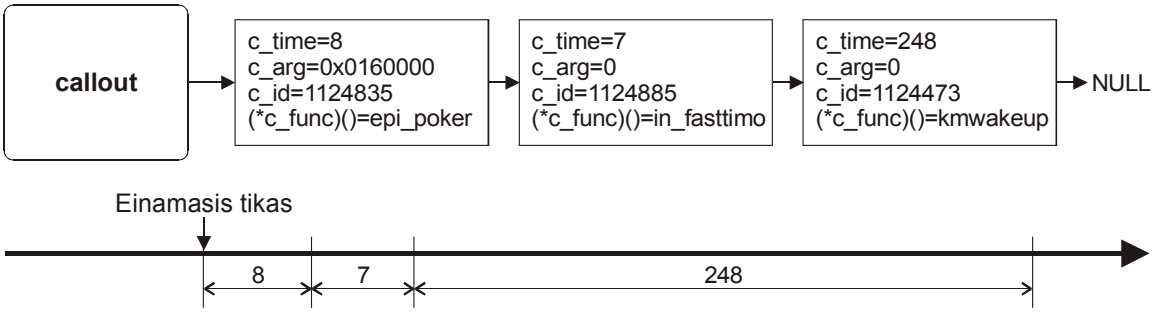
, kur `fn()` – funkcijos adresas, kurią reikia iškviesti po `delta` tikų ir perduoti `arg` argumentą.

Branduolys funkciją vykdys sisteminiame kontekste, todėl ji neturėtų naudoti einamojo proceso adresinę erdvę (vartotojo režime), nes ji gali su einamuoju procesu nieko bendro neturėti. Taip pat ji negali pereiti į "miego" būseną.

Atidėti iškvietimai gali būti naudojami atlikti eilei funkcijų, tokių kaip:

- planuotojo arba atminties valdymo posistemės funkcijos;
- aparatūros tvarkyklių darbui, kai tam tikro įvykio tikimybė yra santykinai didelė, tarkim TCP protokolo modulis, realizuojantis pakartotinį paketų perdavimą pagal taim-out'ą;
- aparatūros, nepalaikančios pertraukimų, apklausimas.

Atidėtų iškvietimų realizacija užsiima speciali programa, kuri yra pakraunama po taimerio pertraukimo apdorojimo. Ji naudoja specialią duomenų struktūrą, surūšiuotą taip, kad greičiausiai įvykdytinas iškvietimas butų pirmasis. Po kiekvieno tiko tereikia sumažinti pirmojo elemento `c_time` lauką ir, kai jis taps lygus 0, iškviesti jame nurodytą funkciją. Visa callout struktūra yra pateikta 3.15 paveiksle.



3.15 pav. Atidėtų iškvietimų duomenų struktūros.

3.8.3 Aliarmai

Procesas gali paprašyti branduolio nusiųsti signalą – aliarmą – kai praeis tam tikras laiko intervalas. Su visais šiais aliarmais yra susijęs intervalo taimeris, kuris yra mažinamas su kiekvienu tiku. Kai jis pasiekia 0 – procesui yra nusiunčiamas specialus signalas. Egzistuoja trys aliarmų tipai:

1. Realaus laiko (real-time alarms, ITIMER_REAL interval timer) aliarmas yra naudojamas realaus laiko skaičiavimo uždaviniuose. Kai taimerio reikšmė tampa lygi nuliui procesui yra nusiunčiamas SIGALRM signalas.
2. Profiliavimo (profiling alarms, ITIMER_PROF interval timer) aliarmai naudojami sisteminiuose uždaviniuose. Jų taimerių reikšmės yra mažinamos tik tada, kai procesas yra vykdomas branduolio arba vartotojo režimuose. Kai jis tampa lygus nuliui, procesui pasiunčiamas SIGPROF signalas.
3. Virtualaus laiko (virtual time alarm, ITIMER_VIRT interval timer) taimeris mažinamas, tada, kai procesas yra vykdomas vartotojo režime. Kai taimeris tampa lygus nuliui, procesui yra nusiunčiamas SIGVALRM signalas.

BSD UNIX sistemose visų tipų aliarmų nustatymui naudojama sisteminė komanda settimer(2), kurios reikšmė yra nustatoma milisekundėmis. Sistemos branduolys šią reikšmę paverčia į tikus. System V sistemose realaus laiko aliarmų nustatymui naudojama alarm(2) sisteminė komanda; intervalas nustatomas sekundėmis. UNIX SVR4 su komanda hrtsys(2) leidžia nustatyti didelio tikslumo taimerius mikrosekundėmis. BSD sistemose alarm funkcija yra realizuota kaip bibliotekos funkcija.

Nereikėtų manyti, kad realaus laiko aliarmai užtikrina didelį tikslumą. Ilguose laiko intervaluose išryškėja nemaža paklaida. Tai susiję su tuo, kad kai intervalo taimerio reikšmė pasiekia 0, signalas greitai, sekančio tiko metu yra nusiunčiamas reikiamam procesui. Tačiau šis procesas, norėdamas jį apdoroti, turi būti pasirinktas planuotojo. Jei jo prioritetas nėra aukštas – laukimas eilėje prie CPU gali užtrukti. Tikslumas užtikrinamas tik ilgesniems laiko terminalams ir aukšto prioriteto procesams.

3.8.4 Proceso kontekstas

Kiekvienas procesas turi kontekstą, kurį sudaro visa informacija reikalinga aprašyti duotąjį procesą. Ji yra išsaugoma, kai proceso vykdymas yra sustabdomas; kai jo vykdymas pratęsiamas, kontekstas atstatomas. Proceso kontekstą sudaro:

- Proceso adresinė erdvė vartotojo režime. Jį sudaro kodas, duomenys ir stekas; taip pat gali būti ir kitos struktūros, tokios kaip dalinama atmintis, bibliotekos ir pan.
- Proceso valdymo struktūros. Branduolys naudoja dvi struktūras – proc ir user.
- Proceso aplinka. Ją sudaro visi aplinkos kintamieji `NAME=VALUE` tipo.

- Aparatūrinis kontekstas. Į jį įeina bendrieji ir sisteminiai procesoriaus registrai. Sisteminiai registrai yra šie:
    - instrukcijų registras, rodantis į sekančią instrukciją;
    - steko rodyklės registras;
    - slankaus kablelio registrai;
    - atminties valdymo registrai, kurie yra atsakingi už virtualių adresų transliaciją į fizinius.

Procesoriaus perdavimas kitam procesui yra vadinamas konteksto pakeitimu. konteksto pakeitimas yra pakankamai brangi operacija, neskaitant reikšmių užsaugojimo procesoriui reikia atlikti begalę kitų operacijų, tarkim kai kurioms UNIX sistemoms tenka išvalyti instrukcijų, duomenų ir adresų transliacijos buferius (cache). Dėl to, naujai paleistas procesas pradeda darbą be buferio, kas taip pat atsiliepia jo darbo greičiui.

Konteksto perjungimas įmanomas šiose situacijose:

- procesas užmiega laukdamas kokio nors resurso;
- procesas baigia darbą;
- perskaičiavus prioritetus pastebėta, kad eilėje laukia procesas su didesniu prioritetu;

3.8.5 Procesų planavimo principai

UNIX procesų planavimo algoritmai įgalina vienu metu vykdyti fonines ir interaktyvias programas. Toks principas pilnai tinka darbo stotims, prie kurių yra pasijungę keli vartotojai, kurie naudoja teksto redaktorius, kompiliuoja programas ir pan.

UNIX sistemose procesų planavimas remiasi procesų prioritetais. Planuotojas visada išrenka procesą su aukščiausiu prioritetu. Prioritetas nėra fiksuotas dydis, o dinamiškai keičiamas priklausomai nuo procesoriaus naudojimo, laukimo eilėje laiko ir proceso būsenos. Jei procesas su aukščiausiu prioritetu atsistoja į eilę prie procesoriaus, tai branduolys sustabdo einamojo proceso vykdymą (su mažesniu prioritetu), net jei jis nepilnai išnaudojo savo laiko kvanto.

Kiekvienas procesas turi du prioriteto atributus: einamasis prioritetas, pagal kurį atliekamas planavimas, ir suteiktas santykinis prioritetas – nice number (arba paprastai nice), kuris yra užduodamas sukuriant procesą ir įtakoja einamąjį prioritetą.

Einamasis prioritetas kinta nuo 0 (žemiausio) iki 127 (aukščiausio). Vartotojo režime dirbantys procesai turi žemiausius prioritetus, 0 – 65, o branduolio režime – aukštesnius, 66 – 95. Procesai, kurių prioritetai yra diapazone 96 – 127, yra vadinami fiksuoto prioriteto procesais, ir naudojami realaus laiko programose.

Procesui, kuris duotuoju laiku laukia kokio nors resurso yra suteikiamas ‘miego prioritetas’, t.y. išrenkamas prioritetas iš sisteminių prioritetų sekos priklausomai nuo įvykio sukėlusio tokią proceso būseną. 3.2 lentelėje yra pateikiami kai kurie proceso migo prioritetai; pastebėsim, jog BSD ir SCO UNIX prioritetų reikšmės yra priešingos.

3.2 lentelė. Sisteminiai miego prioritetai.

Įvykis	**4.3BSD prioritetas**	**SCO prioritetas**
Segmento/puslapio pakrovimo į pagrindinę atmintį laukimas	0	95
Indekso deskriptoriaus laukimas	10	88
I/O operacijos laukimas	20	81
Buferio laukimas	30	80
Terminalinio įvedimo laukimas		75
Terminalinio išvedimo laukimas		74
Vykdymo pabaigos laukimas		73
Žemo prioriteto įvykio laukimas	40	66

Kai procesas atsibunda, jis yra nusiunčiamas į eilę prie procesoriaus su sisteminiu miego prioritetu; tokiu atveju jo prioritetas yra sisteminis, t.y. aukštas, todėl yra didelė tikimybė, kad atsibudęs procesas bus greitai pasirinktas planuotojo vykdymui.

Grąžinant procesą į vartotojo režimą, yra atstatomas jo buvęs prioritetas, t.y. einamasis prioritetas dažniausiai yra pažeminamas, tai gali sukelti konteksto perjungimą.

Einamasis proceso prioritetas vartotojo režime p_priuser priklauso nuo dviejų reikšmių: nice number – `p_nice` ir procesoriaus naudojimo laipsnio `p_cpu`:

```
p_priuser = a*p_nice – b*p_cpu
```

Kai procesas naudoja procesorių jo prioriteto sudedamoji `p_cpu` yra pastoviai didinama; tokiu būdu einamasis prioritetas linijiškai mažėja. Kiekvieną sekundę procesorius perskaičiuoja laukiančių procesų, kurių prioritetas mažesnis už 65) prioritetus, t.y. juo padidina sumažindamas `p_cpu` sudedamąją.

SVR3 naudoja sekančią formulę

```
p_cpu = p_cpu/2
```

Ši paprasta schema neatsižvelgia į procesų skaičių eilėje, t.y. sistemos apkrovimą. Labai apkrautoje sistemoje procesų p_cpu reikšmė tampa labai maža, todėl mažo nice skaičiaus procesai praktiškai negauna procesoriaus.

4.3BSD UNIX sistemose naudojama kita formulė

```
p_cpu = p_cpu*(2*load)/(2*load+1)
```

Parametras `load` yra lygus paskutinę sekundę prie procesoriaus buvusių procesų skaičiui ir charakterizuoja vidutinį sistemos apkrovimą duotuoju momentu. Tai iš dalies leidžia atsižvelgti į sistemos apkrovimą, nes, stipriai apkrautoje sistemoje `p_cpu` sudedamoji mažėja lėčiau.

Ši planavimo sistema leidžia efektyviai dalinti procesorių procesams. Žemo prioriteto procesai kuo ilgiau laukia, tuo labiau didėja tikimybė, kad planuotojas juos galiausiai pasirinks. Interaktyvios aplikacijos daugiausia laukia įvedimo operacijos, turėdamos aukštą miego prioritetą. Kai gauna reikiamą resursą – greitai būna įvykdyti. Foninės aplikacijos pastoviai laukia procesoriaus – turi didelę `p_cpu` reikšmę ir, tuo būdu, mažą prioritetą.

Paprastai UNIX sistemos naudoja ne vieną procesų prie procesoriaus eilę. SCO UNIX turi 127 eile, po vieną kiekvienam prioritetui. BSD UNIX naudoja 32 eiles, kurių kiekviena aptarnauja procesus iš tam tikro prioritetų diapazono, tarkim 0 – 3, 4 – 7 ir t.t.

Kiekvienu laiko kvantu sistema paleidžia specialų apskrito lygiavimo (round robin) mechanizmą, kuris iš vienodo prioriteto procesų išrenka vieną paleidimui.

3.9 Proceso sukūrimas

UNIX sistemose yra tiksliai pravesta riba tarp programos ir proceso. Kiekvienas procesas duotuoju laiko momentu vykdo vienos arba kito programos instrukcija (keli procesai gali vykdyti vieną ir tą pačią programą). Tokios situacijos pavyzdys – komandų interpretatorius – daug vartotojų naudojasi vienos ir tos pačios programos instrukcijomis.

Kiekvienas procesas bet kuriuo metu gali paleisti naują programą, t.y. procesą, kuris vykdytų tos programos instrukcijas.

UNIX OS yra skirtingos sisteminės komandos, kurios sukuria procesą ir paleidžia programą. Sisteminė komanda `fork(2)` sukuria naują procesą, t.y. beveik tikslią tėvinio proceso kopiją. Įvykdžius šią sisteminę komandą abu procesai, tėvas ir jo vaikas, vykdys vienos ir tos pačios programos instrukcijas ir turės tuos pačius duomenų ir steko segmentus. Tačiau tarp jų atsiras keletas skirtumų, tokių kaip:

- naujai sukurtam procesui suteikiamas naujas PID;
- tokiu būdu skiriasi ir PPID;

- naujas procesas gauna naujas proc ir u-area duomenų struktūras ir failų deskriptorius, kurie rodo į tuos pačius failus;
- vaikui yra išvalomi visi laukiantys signalai;
- vaikui yra išvalomi vartotojo ir branduolio režime praleisto laiko skaitliukai;
- tėvinio proceso atminties blokavimas nėra paveldimas.

Lentelėje 3.3 yra detaliau pateikiamas paveldimumas sukuriant procesą ir startuojant programą.

3.3 lentelė. Proceso atributų paveldimumas naudojant fork(2) ir exec(2) komandas.

Atributas	**Paveldėjimas po `fork(2)`**	**Paveldėjimas po `exec(2)`**
Kodo segmentas (text)	Taip, bendras	Ne
Duomenų segmentas (data)	Taip, copy-on-write	Ne
Aplinka	Taip	Gali būti
Argumentai	Taip	Gali būti
UID	Taip	Taip
GID	Taip	Taip
EUID	Taip	Taip (ne, jei setuid(2))
EGID	Taip	Taip (ne, jei setgid(2))
PID	Ne	Taip
PGRP	Taip	Taip
PPID	Ne	Taip
Nice number	Taip	Taip
Priėjimo teisės sukurtam failui	Taip	Taip
Failo apimties apribojimai	Taip	Taip
Signalai, apdorojami pagal nutylėjimą	Taip	Taip
Ignoruojami signalai	Taip	Taip
Perimami signalai	Taip	Ne
Failų deskriptoriai	Taip	Taip, jei ne FD_CLOEXEC
Nuorodos į failus	Taip, bendri	Taip, jei ne FD_CLOEXEC

Trumpai tariant `fork(2)` atlieka sekančius veiksmus:

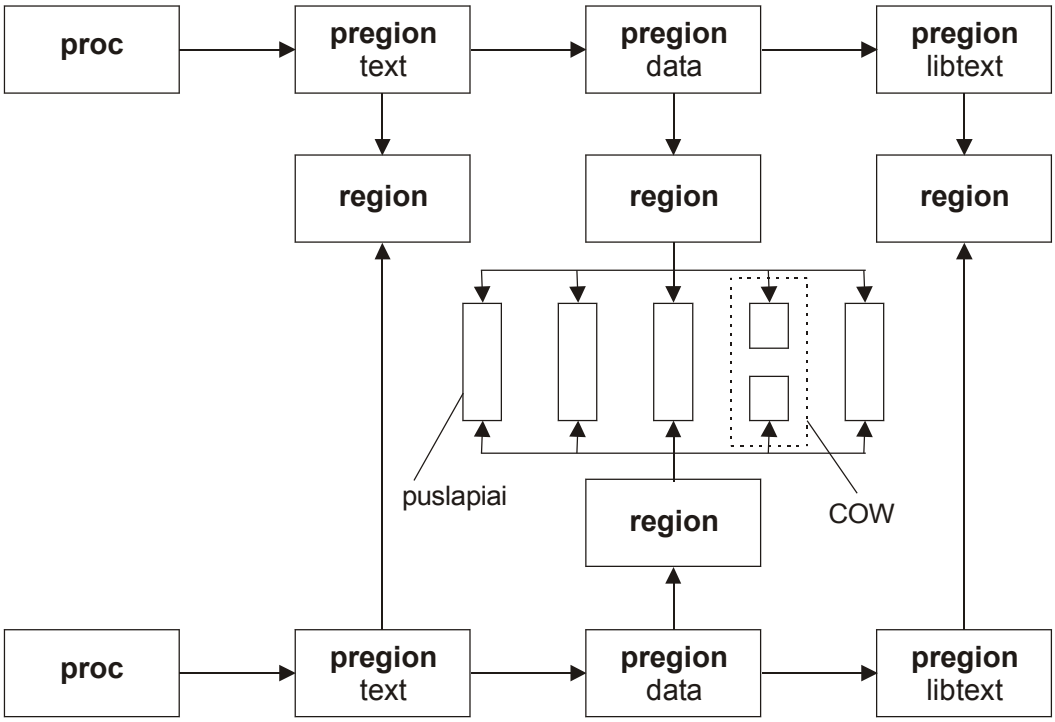
1. Antrinėje atmintyje rezervuoja vietą proceso duomenų ir steko segmentams.
2. Į `proc` lentelę įdeda naują įrašą (sukuria naują struktūrą) ir priskiria jam unikalų PID.
3. Patalpina naujas virtualių adresų atvaizdų lenteles.
4. Sukuria naują `u-area` struktūrą ir nukopijuoja į ją duomenis iš tėvo `u-area` struktūros.
5. Sukuria atitinkamas proceso atminties sritis, dalis kurių sutampa su tėvo sritimis.
6. Inicijuoja aparatūrinį tėvo kontekstą, t.y. nukopijuoja jį iš tėvo.
7. Vaiko procesui grąžina nulį (0).
8. Tėvui grąžina vaiko PID.
9. Patalpina naują procesą į eilę prie procesoriaus.

Kai tik naujas procesas gauna procesorių, jis iškviečia komandą `exec(2)` ir, sunaikinęs sukurtąsias atminties struktūras sukuria naujas. Toks mechanizmas nėra efektyvus, nes be reikalo yra eikvojama atmintis procesoriaus laikas. UNIX System V pirmą kartą pasiūlytas naujas požiūris, vadinamas “kopijavimas rašant” (copy-on-write, COW). Šio mechanizmo esmė yra ta, kad naujai sukurtas procesas rodo į tuos pačius atminties puslapius kaip ir tėvas, tačiau jam jie yra prieinami tiktai skaitymui. Jei procesas pabando juos modifikuoti, yra sukeliamas pertraukimas, padaroma reikiamo puslapio kopija, modifikuojama puslapių lentelė

ir paprašoma atlikti operaciją iš naujo. Tokiu atveju, kopijuojami yra tik tie puslapiai, kuriuos procesui prireikia modifikuoti.

Kitą požiūrį naudoja BSD sistemos. Jose yra realizuotas nauja sisteminė komanda `vfork(2)`. Iškvietęs šią komandą tėvas perduoda savo adresinę erdvę vaikui ir užmiega, kol vaikas neatliks `exec(2)` arba `exit(2)` komandos. Šiuo atveju proceso sukūrimo procesas pasiekia maksimalų greitį, nes nereikia nieko kopijuoti. Tačiau šis metodas potencialiai yra pavojingas, nes vaikinis procesas gali ne tik naudoti, bet ir modifikuoti tėvo duomenis.

Vaiko pregion struktūros rodo į tas pačias region sritis, kaip ir tėvo, todėl jos yra vadinamos bendrojo naudojimo. Jei bendras srities panaudojimas nėra įmanomas, tai branduolys sukuria vaikinam procesui iliuzija, kad jis dirba su savo sritimi – region struktūra kita, tačiau ji rodo į tuos pačius puslapius. Naujo puslapio sukūrimą gali sukelti tiks modifikavimo operacijos, tokios kaip COW (3.16 paveikslas).



3.16 pav. Proceso ir jo vaikinio proceso susijusios duomenų struktūros.

3.10 Naujos programos paleidimas

Naujos programos paleidimas įvykdomas su sistemine komanda `exec(2)`. Taip sukuriamas ne naujas procesas, o nauja jo adresinė erdvė. Jei procesas buvo sukurtas su `vfork(2)` komanda, sena adresinė erdvė yra atiduodama tėviniam procesui, o kitu atveju – tiesiog sunaikinama. Baigus vykdyti `exec(2)` komandą procesas pradeda vykdyti naują programą.

UNIX operacinė sistema palaiko kelis vykdomų failų formatus – senąjį a.out, COFF ir ELF. Bet kuriuo atveju vykdomasis failas turi antraštę, kuri padeda branduoliui teisingai pakrauti ir sukonfigūruoti proceso adresinę erdvę.

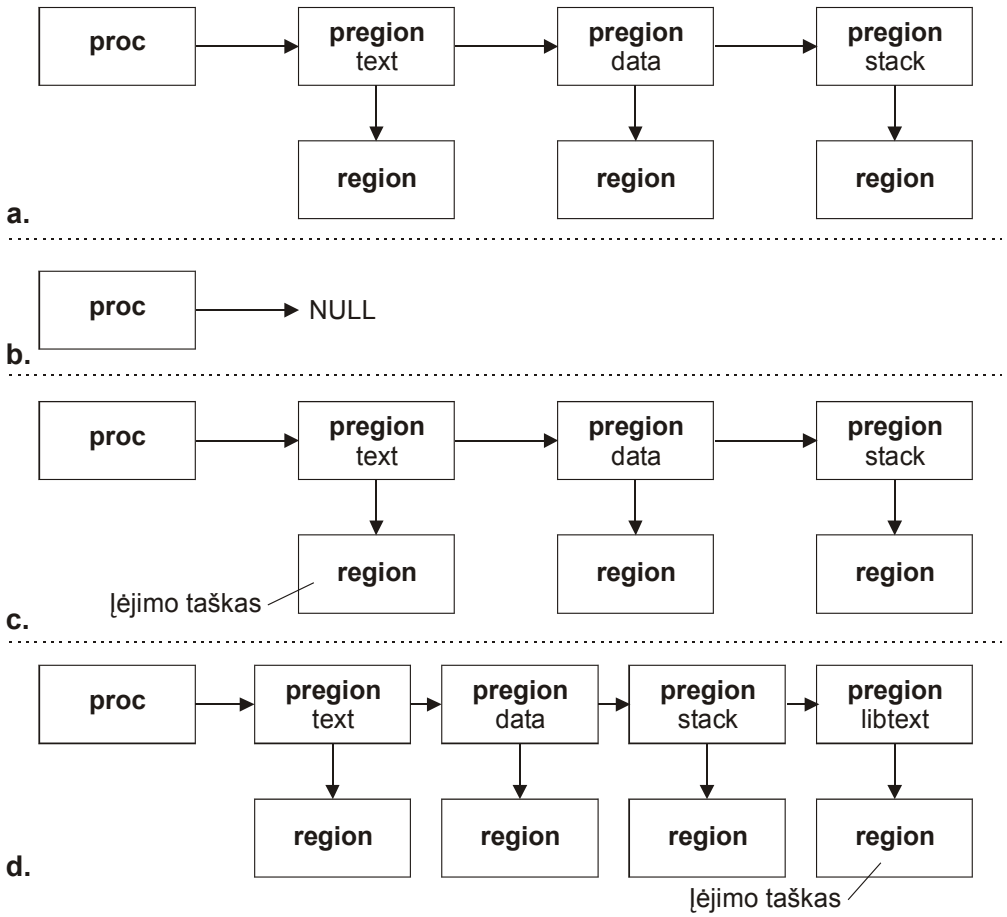
Pateiksime veiksmų seką, kurią atlieka `exec(2)` komandą pakraudama naują programą:

- Transliuoja failo vardą. Gaunamas failo deskriptorius, kuriuo pagalba yra kreipiamasi į failą. Tuo pačiu yra patikrinamos teisės į šį failą.
- Skaitoma failo antraštė ir tikrinama, ar jis yra vykdomasis. `exec(2)` komanda atpažįsta komandinio interpretatoriaus skriptus (shell scripts). Jis nuskaito pirmąją eilutę, kuri paprastai turi specialią struktūrą, tarkim `#!/bin/perl`. Jei failas nėra

binarinis ir jo pirmojoje eilutėje nenurodytas interpretatorius, tai paleidžiamas interpretatorius pagal nutylėjimą: `/bin/sh`, `/usr/bin/sh` arba `/usr/bin/ksh`. Interpretatoriui yra perduodamas skript failas kaip įvedimas.

- Jei vykdomasis failas turi SUID ir SGID, tai pakeičiami proceso efektyvūs UID ir GID.
- Užsaugo `exec(2)` argumentus ir proceso aplinką branduolio adresinėje erdvėje, nes proceso duomenų struktūros bus sunaikintos.
- Rezervuoja antrinėje atmintyje vietą duomenų ir steko segmentams.
- Atlaisvina senas proceso adresines sritis esančias pirminėje ir antrinėje atmintyse. Jei procesas buvo sukurtas su vfork(2) – senos atminties sritys gražinamos tėvui.
- Patalpina ir apibrėžia atminties atvaizdų lenteles naujiems kodo, duomenų ir steko segmentams.Jei kodo segmentas jau yra aktyvus, t.y. tą pačią programą jau vykdo koks nors procesas, tai einamasis pasijungia ją naudoti bendrai. Kitu atveju pakraunamas reikalingas programos gabalas iš failo. Neapibrėžtų duomenų sritis užpildoma nuliais. Programos failo kopijavimas į pirminę atmintį vyksta po puslapį ir tik tada, kai procesas kreipiasi į trūkstamą puslapį.
- Kopijuoja užsaugotus argumentus ir aplinkos kintamuosius į naujo proceso steką.
- Nustato visų signalų apdorojimą į reikšmes pagal nutylėjimą.
- Inicijuoja aparatūrinį kontekstą. Po šio veiksmo instrukcijų registras (IR) jau rodo į pirmąją programos instrukciją – įėjimo tašką.

Tuo atveju, kai programa naudoja dinamines bibliotekas, tai atitinkamas vykdomojo failo skyrius turi ne biblioteką, o dinaminio ryšio redaktoriaus vardą. Tada į dinaminės bibliotekos kodo sritį yra pakraunamas ryšio redaktorius, kuris pasirūpina, kad būtų surastos ir pakrautos reikiamos bibliotekos. Tai turi būti atlikta prieš pradedant programos vykdymą, t.y. įėjimo taškas nustatomas į dinaminės bibliotekos sritį. Baigęs darbą redaktorius perduoda procesorių pagrindinei programai analizuodamas failo antraštę.



3.17 pav. Programos procese paleidimo stadijos.

Naujos programos pakrovimo stadijas iliustruoja 3.17 paveikslas.

3.11 Proceso vykdymo pabaiga

Iškvietęs funkciją `exit(2)` procesas baigia savo gyvavimą. Funkciją gali iškviesti tiek pats procesas, tiek branduolys. `exit(2)` funkcija atlieka tokius veiksmus:

- Atjungia visus signalus.
- Uždaro atidarytus failus.
- Užsaugo panaudotų skaičiavimo resursų statistiką `proc` struktūroje.
- Pakeičia proceso būseną į zombis.
- Padaro procesą init(1M) visų proceso vaikų tėvu.
- Atlaisvina proceso adresinę erdvę, u-area, atvaizdų lenteles ir antrinės atminties sritį susijusią su procesu.
- Nusiunčia tėviniam procesui informaciją apie vaiko mirtį – signalą `SIGCHLD`.
- Prikelia tėvą, jei jis laukia vaiko darbo pabaigos.
- Iškviečia konteksto perjungimo funkciją.

Tapęs zombiu procesas teturi įrašą `proc` lentelėje, kuris saugo proceso CPU naudojimo statistiką ir grąžinamą reikšmę. Šios informacijos gali prireikti tėviniam procesui, todėl tik tėvas, naudodamas sisteminę komandą `wait(2)`, kuri grąžina statistiką ir kodą sunaikina šią

paskutinę struktūrą. Jei tėvas anksčiau baigia savo gyvenimą nei vaikas, tai vaiko tėvu tampa `init(1M)` procesas. Jis pasirūpina našlaičių sunaikinimu.

3.12 Signalai

Signalai, tam tikra prasme yra paprasčiausia tarp-procesinio bendravimo priemonė, kuri praneša procesams ar jų grupei apie kokį nors įvykį. Jų pagalba OS branduolys arba kitas procesas nusiunčia pranešimą t.t. procesui. Tokiu būdu yra realizuojamas paprasčiausias programinis pertraukimas – normalus programos darbas pertraukiamas ir reaguojama į signalą, pvz. nuspaudus `Ctrl + C` einamajam procesui yra nusiunčiamas `SIGINT` signalas, kuris nutraukia proceso darbą. Signalus t.t. procesui nusiunčia speciali komanda `kill(1)`, pvz.

```
$ kill sig_no PID
```

Gavęs signalą procesas gali pasielgti trejopai:

1. Signalo ignoravimas.
2. Paprašyti įvykdyti veiksmą pagal nutylėjimą. Tačiau tai dažniausiai yra darbo nutraukimas.
3. Perimti signalą. Tarkim gavęs signalą `SIGTERM` arba `SIGINT` procesas gali sunaikinti laikinas struktūras ir failus, atlaisvinti atmintį ir pasiruošti tvarkingai numirti. Kai kurių signalų, pvz. – `SIGKILL` ar `SIGSTOP` perimti negalima.

Pagal nutylėjimą komanda kill procesui nusiunčia 15 signalą – `SIGTERM`, kuris reiškia proceso darbo pabaigą. Šį signalą programa gali perimti, susitvarkyti savo darbo aplinką ir numirti. Tačiau galime nusiųsti ir 9 signalą – `SIGKILL` – kurie kietai nužudys procesą.

Signalai gali būti naudojami ne tik procesų sustabdymui ir nužudymui, bet ir specialių uždavinių atlikimui, pvz. DNS serveriui named pasiųstas signalas `SIGHUP` priverčia jį perskaityti konfigūracinius failus iš naujo:

```
# kill –HUP `cat /etc/named.pid`
```

Lentelėje 3.4 yra pateikti svarbesnieji UNIX OS signalai. Kaip matyti dauguma veiksmų pagal nutylėjimą yra proceso nužudymas. Kai kuriais atvejais yra generuojamas `core` – t.y. darbiniame kataloge generuojamas proceso atminties atvaizdas, kuris po to gali būti analizuojamas specialiomis debug programomis. core nebus sugeneruotas štai kuriais atvejais:

- jei vykdomasis proceso failas turi nustatytą `setuid`, o realusis proceso savininkas nėra vykdomojo failo savininku;
- jei vykdomasis proceso failas turi nustatytą `setgid`, o realusis proceso savininkas nėra vykdomojo failo grupės nariu;
- procesas neturi teisės rašyti į darbinį katalogą;
- jei `core` failo dydis butų didesnis nei leidžiamas apribojimas `RLIMIT_CORE`.

3.4 lentelė. Svarbesnieji UNIX signalai.

Signalas	**Veiksmas pagal nutylėjimą**	**Paaiškinimas**
SIGABRT	Terminate+core	Signalas generuojamas iškvietus sisteminę komandą abort(2).
SIGALRM	Terminate	Signalas nusiunčiamas, kai suveikia taimeris, su anksčiau iškviestais alarm(2) arba settimer(2).
SIGBUS	Terminate+core	Aparatūrinė klaida. Tarkim, jei procesas kreipiasi į virtualų adresą faile esantį už failo ribų.
SIGCHLD	Ignore	Informuojamas tėvinis procesas apie vaiko darbo pabaigą.
SIGEGV	Terminate+core	Generuojamas, jei procesas bandė pasiekti jam neleistiną adresą, t.y. į kurį procesas neturi pakankamai privilegijų.

SIGFPE	Terminate+core	Generuojamas kylant ypatingai situacijai, pvz. dalyba iš 0 arba operacijos su slankiuoju kableliu perpildymas.
SIGHUP	Terminate	Signalas yra nusiunčiamas proceso lyderiui susietam su terminalu, kai pastarasis yra išjungiamas (hangup). Kartais signalas naudojamas kitais tikslais servisams (demonams). Toks pasirinkimas yra protingas, nes demonai nebuna susiję su terminalais ir tokio signalo negauna.
SIGILL	Terminate+core	Generuojamas, jei procesas pabando įvykdyti neleistiną instrukciją.
SIGINT	Terminate	Nusiunčiamas nuspaudus Ctrl+C klavišus.
SIGKILL	Terminate	Baigiamas proceso darbas. Šio signalo negalima nei perimti, nei ignoruoti.
SIGPIPE	Terminate	Signalas generuojamas tada, kai procesas pabando nusiųsti duomenis per kanalą ar soket'ą kitam procesui, kuris jau yra baigęs darbą.
SIGPOLL	Terminate	Nusiunčiamas kai kyla ypatinga situacija apklausiant aparatūrinius įrenginius.
SIGPWR	Ignore	Signalas generuojamas kylant maitinimo netekimo grėsmei (UPS).
SIGQUIT	Terminate+core	Siunčiamas procesų grupei nuspaudus klavišus Ctrl+/
SIGSTOP	Stop	Procesas sustabdomas nuspaudus klavišą Ctrl+Z
SIGSYS	Terminate+core	Signalą generuoja branduolys, kai procesas pabando įvykdyti neleistiną instrukciją.
SIGTERM	Terminate	Šis signalas reiškia perspėjimą, kad procesas tuoj bus nužudytas, t.y. paliekama laiko tam pasiruošti. Signalas nusiunčiamas komanda kill pagal nutylėjimą.
SIGTTIN	Stop	Signalą branduolys sugeneruoja tada, kai procesas pabando skaityti iš valdymo terminalo.
SIGTOU	Stop	Signalą branduolys sugeneruoja tada, kai procesas pabando rašyti į valdymo terminalą.
SIGUSR1	Terminate	Skirtas vartotojo reikmėms, kad pranešti kokią nors informaciją procesui.
SIGUSR2	Terminate	Skirtas vartotojo reikmėms, kad pranešti kokią nors informaciją procesui.

Daugiau informacijos apie Jūsų sistemos signalus galite rasti /usr/include/signal.h arba

```
$ man signal.
```

Reikia skirti dvi signalų mechanizmo darbo stadijas – signalo generavimas arba išsiuntimas ir jo pristatymas bei apdorojimas.

3.12.1 Signalų generavimas ir išsiuntimas

Signalus gali sugeneruoti sekantys įvykiai:

- Ypatingosios situacijos. Proceso darbas sukuria ypatingą situaciją, tarkim, dalyba iš 0.
- Terminaliniai pertraukimai. Nuspaudus Ctrl+C yra nusiunčiamas signalas einamąjam procesui, kuris yra susietas su terminalu.
- Kiti procesai. Kai vienas procesas nusiunčia kitam procesui signalą, pranešdamas, kad pastarasis turi atlikti vieną ar kitą veiksmą.
- Uždavinių valdymas. Naudojami uždavinių valdymui, tarkim, tėvui yra nusiunčiamas signalas, kai jo vaikas baigia darbą.
- Kvotos. Kai procesas viršija jam skirtą skaičiavimo ar failo resursų kvotą, jam pasiunčiamas signalas.

- Informacija. Procesui yra nusiunčiama informacija apie vieną ar kitą įvykį, tarkim, aparatūrinės įrangos pasiruošimą darbui ir pan.
- Aliarmai. Jei procesas nustatė taimerį, tai nustatytu metu jam bus atsiųstas signalas.

Vienas procesas kitam signalą gali išsiųsti naudodamas specialią funkciją:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

3.12.2 Signalų priėmimas ir apdorojimas

Signalo priėmimas vykdomas tada, kai branduolys proceso vardu iškviečia sisteminę komandą `issig()`, kuri patikrina, ar eilėje nėra procesų. Smarkiai apkrautose sistemose signalo apdorojimas gali užtrūkti, t.y. išsiuntus signalą reakcija į jį galima tik tada, kai procesas gauna procesorių. Funkcija `issig()` yra iškviečiama trimis atvejais:

1. prieš grįžtant iš branduolio į vartotojo režimą;
2. prieš pereinant procesui į miego būseną;
3. tuojau po signalo perėjimo iš miego į būseną vykdytinas.

Jei funkcija `issig()` suranda laukiantį signalą, ji, atsižvelgdama į proceso signalų dispozicijas, saugomas `u-area` struktūroje (kintamasis `u_signal`), iškviečia funkciją apdorojimui pagal nutylėjimą arba iškviečia specialią funkciją `sendsig()`. Ši funkcija iškviečia proceso užregistruotą signalo apdorojimo funkciją, t.y. `sendsig()` įveda procesą į vartotojo režimą ir perduoda valdymą signalo apdorojimo funkcijai, o po to atstato proceso kontekstą.

Paprasčiausias signalų valdymo interfeisas yra suteikiamas naudojant funkciją `signal`. Su jos pagalba galima pakeisti signalų dispoziciją, t.y. pakeisti OS veiksmą pagal nutylėjimą. Tėvo sukurtas, naujas procesas paveldi signalų dispozicijas, tačiau iškvietus `exec` komandą signalų dispozicijos bus nustatomos į reikšmes pagal nutylėjimą. Tai visai normalu, nes naujoji programa galbūt neturi tokios signalo apdorojimo funkcijos. Funkcijos apibrėžimas:

```
#include <signal.h>
void (*signal (int sig, void (*disp) (int))) (int);
```

Argumentas sig nurodo signalą, kurio dispoziciją reikia pakeisti. Argumetas disp užduoda naują signalo sig dispoziciją, kuri gali būti nauja signalo apdorojimo funkcija arba viena iš reikšmių:

- `SIG_DFL` – nurodo branduoliui, kad gavus signalą reikia kviesti sisteminę funkciją, t.y. atlikti veiksmą pagal nutylėjimą.
- `SIG_IGN` – nurodo, kad signalą reikėtų ignoruoti. Kai kurių signalų ignoruoti nėra galima.

POSIX.1 standartas nustatė naują signalo valdymo funkcijų aibę, kuri pirmiausia buvo realizuota 4.2BSD UNIX sistemoje. Šis standartas remiasi į signalų rinkinio sampratą – `sigset_t`. Daugelyje sistemų `sigset_t` turi maksimalią 32 signalų eilę. Kiekvienas procesas turi signalų aibę ir jiems nustatytas dispozicijas:

```
#include <signal.h>

// Išvalomas signal• rinkinys
int sigemptyset(sigset_t *set);

// Signal• rinkinys užpildomas visais OS žinomais signalais
int sigfillset(sigset_t *set);

// •terpiamas ar išmetamas vienas signalas
int sigaddset(sigset_t *set, int signo);
int sigdellset(sigset_t *set, int signo);

// Patikrinama, ar duotasis signalas yra rinkinio narys
int sigismember(sigset_t *set, int signo);
```

Funkciją `signal` pakeitė funkcija `sigaction`:

```
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

Viskas ko reikia nustatyti naują signalų dispoziciją yra saugoma `sigaction` struktūroje:

```
struct sigaction {
       void (*sa_handler)();
       void (*sa_sigaction)(int, siginfo_t *, void *);
       sigset_t sa_mask;
       int sa_flags;
};
```

, kur signalą apdorojanti funkcija `sa_handler` turi tokį pavidalą

```
void handler(int signo, int code, struct sigcontext *scp);
```

Struktūra `siginfo_t` suteikia daugiau informacijos apie procesą, pasiuntusį signalą. Norint rasti informacijos kreipkitės į savo sistemos `<siginfo.h>` failą.

3.13 Ryšys tarp procesų - IPC

Kaip buvo minėta – procesai yra vykdomi savo adresinėje erdvėje ir yra izoliuoti vienas nuo kito. Taip yra eliminuojama tikimybė, kad vienas procesas turės įtakos kito proceso darbui. Tačiau dažnai iškyla tarp-procesinio bendravimo mechanizmo poreikis, kuri privalo

- suteikti procesams įrankius, leidžiančius bendrauti ir
- eliminuoti nepageidautiną proceso įtaką kitam.

Ryšys tarp procesų atlieka sekančius uždavinius:

- Duomenų perdavimas. Vienas procesas gali perduoti kitam įvairios apimties duomenis – nuo kelių baitų iki kelių mega-baitų.
- Bendras duomenų naudojimas. Vietoj duomenų kopijavimo, procesai bendrai naudoja vieną duomenų kopiją, kuriuos pakeitus vienam procesui, tai tuojau pat bus pastebėta ir kito proceso. Bendram duomenų naudojimui prireikia specialaus protokolo, kuris sprendžia konfliktus ir išlaiko duomenų vientisumą.
- Informavimas. Procesas gali pranešti procesui ar jų grupei apie tam tikrą įvykį. Tai gali būti reikalinga dėl veiksmų sinchronizacijos.

Matyti, kad tokių uždavinių sprendimo negalima patikėti patiems procesams, o bendrosios paskirties sistemose – tai netgi pavojinga ir todėl neįmanoma. Todėl tarp-procesinius mechanizmus turi suteikti pati operacinė sistema ir tokie mechanizmai vadinasi IPC (Inter-Process Communication).

UNIX OS turi tokias IPC realizacijas:

- signalai,
- kanalai
- FIFO (vardiniai kanalai),
- pranešimai (pranešimų eilės) (SV IPC),
- semaforai (SV IPC),
- bendrai naudojama atmintis (SV IPC) ir
- soketai (BSD).

Soketus jau palaiko dauguma UNIX sistemų, tačiau istoriškai jie priklauso BSD sistemoms.

Signalai, kurie pradžioje buvo naudojami tik kaip informavimo apie klaidas mechanizmai, gali būti naudojami kaip IPC. Tačiau jų naudojimas tokiems uždaviniams nėra efektyvus, nes signalų mechanizmas naudoja daug resursų. Signalų nusiuntimas reikalauja sisteminės

komandos iškvietimo, o jo priėmimas – proceso darbo nutraukimo ir intensyvių operacijų su proceso steku, norint iškviesti apdorojimo komandą ir grąžinimo į normalią būseną. Signalai taip pat perduoda mažai informacijos ir jų skaičius yra labai ribotas.

3.13.1 Kanalai

Kanalai komandų interpretatoriuje yra sukuriami sekančiai:

```
$ cat myfile | wc
```

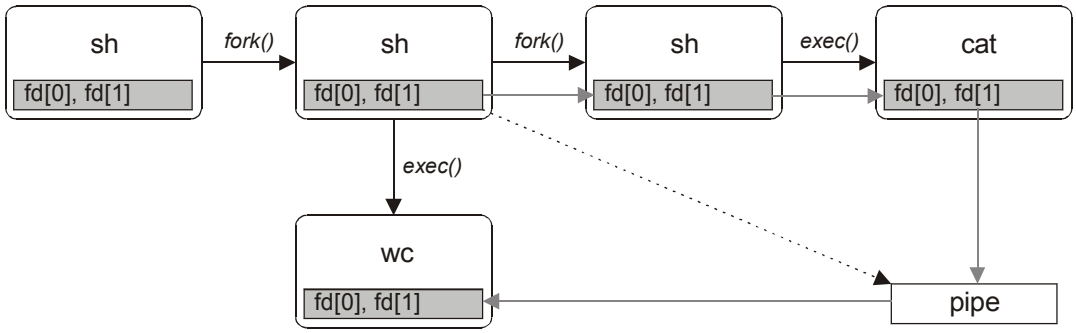
, kurios pasekoje `cat` komandos standartinis išvedimas – rezultatas yra pervedamas į standartinį komandos `wc` įvedimą. Šiam uždaviniui yra sukuriamas vienos krypties duomenų perdavimo kanalas.

Kanalo sukūrimui yra naudojama sisteminė komanda `pipe(2)`:

```
int pipe(int *filedes);
```

, kuri grąžina du failų deskriptorius – `filedes[0]` rašymui ir `filedes[1]` skaitymui. Tokių būdu, jei vienas procesas įrašo duomenis į `filedes[0]`, tai kitas naudodamas `filedes[1]` gali juos perskaityti. Lieka vienas klausimas – kaip procesui gauti tą failo deskriptorių?

Sukuriant procesą vaikas paveldi tam tikrą tėvo informaciją, tame tarpe ir failo deskriptorius, t.y. failo deskriptorius, kuriuos grąžina `pipe(2)` gali naudoti pats funkciją iškvietęs procesas ir visi jo vaikai. Iš to seka, kad kanalus gali naudoti tik giminingi procesai ir jų negali naudoti procesai iš šalies.



3.18 pav. Kanalo tarp cat ir wc komandų sukūrimas.

Gali atrodyti, kad pavyzdyje pateikti procesai yra nepriklausomi, tačiau tai yra ne taip – 3.18 pav.

3.13.2 FIFO

FIFO IPC realizuoja FIFO (First In First Out) struktūrą, kuri yra labai panaši į kanalą, nes taip pat yra vienos krypties duomenų perdavimo mechanizmas. FIFO skiriasi nuo kanalų tuo, kad turi vardą, todėl kartais yra vadinami vardiniais kanalais. Tai yra System V mechanizmas, kurio BSD sistemose nėra, nors FIFO pirmą kartą realizuota System III sistemose, tačiau iki šiol yra prastai dokumentuota ir todėl retai naudojama.

FIFO yra atskiras failų sistemos tipas – `ls -l` pirmasis simbolis yra `p`. Kuriant vardinį kanalą yra naudojama sekanti sisteminė komanda:

```
int mknod(char *pathname, int mode, int dev);
```

, kur `pathname` – failų sistemos failo vardas (FIFO vardas), `mode` – failo moda (privilegijos ir pan.), `dev` – ignoruojamas.

Vardinį kanalą galime sukurti ir iš komandinio interpretatoriaus įvedimo lauko:

```
$ mknod name p
```

FIFO ir paprasti kanalai dirba pagal sekančias taisykles:

- skaitant mažesnį nei yra kanale baitų skaičių, yra grąžinamas prašomas baitų skaičius, o likutis paliekamas sekančiam skaitymui;
- skaitant didesnį nei yra kanale baitų skaičių, yra grąžinamas leistinas duomenų kiekis, o su likusia dalimi turi susitvarkyti pats klientas;
- jei kanalas tuščias ir nei vienas procesas jo neatidarė rašymui, tai skaitant iš kanalo yra grąžinama 0 baitų. Jei nors vienas procesas atidaro kanalą rašymui, tai skaitantieji procesai taps užblokuoti iki kol pasirodys duomenys (nebent kanalas turi nustatytą `O_NDELAY` neleistino blokavimo požymį);
- jei keli procesai vienu metu rašo į kanalą, jų duomenys nesusimaišo, nes kitų procesų rašymas yra blokuojamas (atomarinė operacija);
- jei rašoma daugiau duomenų, nei jų telpa į kanalą, tai rašymas yra blokuojamas, kol atsiras daugiau vietos. Jei procesas pabando rašyti į kanalą, kurio niekas neatidarė skaitymui, tai generuojamas `SIGPIPE` signalas, kurio apdorojimas pagal nutylėjimą yra proceso darbo nutraukimas.

3.13.3 IPC identifikatoriai ir vardai

Vardai yra svarbūs IPC komponentai, nes be galimybės identifikuoti perdavimo kanalus pašalinis procesas negalės jais naudotis. Kiti System V IPC – pranešimų eilės, semaforai ir bendra atmintis naudoja sudėtingesnį identifikavimo mechanizmą. Jų vardai yra vadinami raktais (key) ir yra generuojami naudojant funkciją `ftok(3C)` iš dviejų komponentų – failo vardo ir projekto identifikatoriaus:

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *filename, char proj);
```

, kur `filename` gali būti bet koks failas, žinomas abiem procesams ir egzistuojantis IPC sukūrimo metu. Taip pat svarbu, kad IPC naudojimo metu minėtas failas nebūtų sunaikintas ir sukurtas iš naujo, nes rakto generavimo procese yra naudojamas failo `inode`.

Kiekvienas sukurtas IPC operacinėje sistemoje turi nuosavą identifikatorių ir su juo susijusias duomenų struktūras. Šias struktūras bendraujantys procesai naudoja taip pat, kaip ir atidaryto failo deskriptorių. Visų minėtų tipų IPC turi unikalius bendravimo kanalų identifikatorius. 3.5 lentelėje yra pateikti IPC objektų identifikatoriai.

3.5 lentelė. IPC objektų identifikacija

IPC objektas	Vardų sritis	Deskriptorius
Kanalas	-	Failų deskriptorius
FIFO	Failo vardas	Failų deskriptorius
Pranešimai	Raktas	Identifikatorius
Semaforas	Raktas	Identifikatorius
Bendrai naudojama atmintis	Raktas	Identifikatorius

Darbas System V sistemose su IPC objektais turi panašų interfeisą, pvz. norint gauti pranešimų objektą iškviečiama funkcija `msgget(2)`, semaforų objektą – `semget(2)`, bendrai naudojamos atminties objektą - `shmget(2)`. Visos šios sisteminės komandos sėkmės atveju grąžina objektą, o nesėkmės -1. Visos get funkcijos grąžina ne objektą, o jo deskriptorių, kurį naudojant galima kreiptis į IPC. Taip pat jos, kaip argumentus, naudoja raktą ir `ipcflags` požymius. Kiti argumentai priklauso nuo objekto tipo. Kintamasis `ipcflags` nurodo priėjimo prie objekto teises `PERM` ir ar sukuriamas naujas objektas, ar sukuriamas priėjimas prie jau esančio – `IPC_CREAT` ir `IPC_EXCL` požymiai.

Priėjimo teisės – PERM požymis – įgaunančios reikšmės yra nurodytos 3.6 lentelėje.

3.6 lentelė. PERM požymio reikšmės, nurodančios priėjimo prie IPC objekto teises.

Reikšmė	Failų teisių analogas	Leidimas
0400	r - - - - - - - -	Skaitymas savininkui – naudotojui

0200	- w - - - - - - -	Rašymas savininkui – naudotojui
0040	- - - r - - - - -	Skaitymas savininkui – grupei
0020	- - - - w - - - -	Rašymas savininkui – grupei
0004	- - - - - - r - -	Skaitymas visiems
0002	- - - - - - - w -	Rašymas visiems

`ipcflags` kombinacijos gali būti įvairios, tuo gaudamos skirtingus rezultatus; jos pateiktos 3.7 lentelėje.

3.7 lentelė. `ipcflags` požymių kombinacijos ir rezultatai.

Argumento ipcflags reikšmė	**Funkcijos `get` rezultatas**	
	Objektas yra	**Objekto nėra**
`0`	Grąžinamas deskriptorius	Klaida; objekto nėra (`ENOENT`)
`PERM | IPC_CREAT`	Grąžinamas deskriptorius	Sukuria objektą su atitinkamomis `PERM` teisėmis
`PERM | IPC_CREAT | IPC_EXCL`	Klaida; objektas jau egzistuoja (`EEXIST`)	Sukuria objektą su atitinkamomis `PERM` teisėmis

System V sistemose darbas su IPC objektais yra panašus į darbą su UNIX failais. Skirtumas tas, kad failo deskriptorius galioja proceso ribose, o IPC deskriptorius yra vienodas visai sistemai.

UNIX OS kiekvienam IPC objektui sukuria atitinkamą objektą, priklausomai nuo IPC tipo, tačiau viena jų dalių yra vienoda – tai `ipc_perm` struktūra, kuri nurodo teises į objektą. Pagrindiniai struktūros laukai yra šie: uid – objekto savininko id, gid – objekto savininko grupė, cuid – objektą sukūrusio vartotojo ID, cgid – objektą sukūrusio vartotojo GID, mode – 9 priėjimo prie objekto teisių bitai, key – objekto raktas.

Operacinė sistema nesunaikina IPC objekto, net jei nei vienas procesas jo nenaudoja. Objekto sunaikinimas yra procesų reikalas, kuriems yra suteikiamos funkcijos – `msgctl(2)`, `semctl(2)` ir `shmctl(2)`. Šių funkcijų pagalba galima keisti kai kuriuos IPC objekto struktūros laukus ir sunaikinti patį objektą. Dažniausiai yra viena programa, vadinama serveris, kuri sukuria IPC objektą, o darbo pabaigoje jį sunaikina.

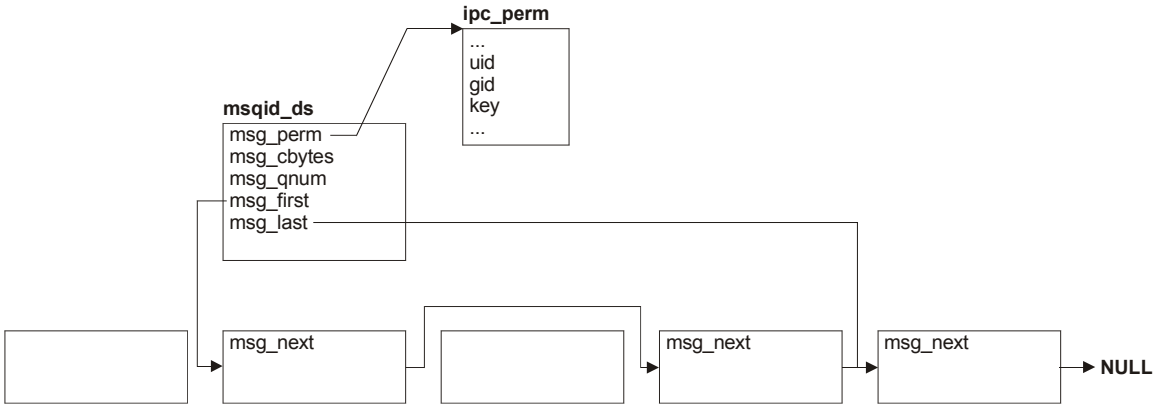
3.13.4 Pranešimai

Kaip jau buvo minėta, pranešimų eilės yra System V OS dalis, kuri jas valdo ir talpina savo adresinėje erdvėje. Kiekviena pranešimų eilė turi savo unikalų identifikatorių. Procesas, patalpinęs pranešimą į eilę, gali nelaukti kol kitas procesas nuskaitys tą pranešimą. Jis palieka eilėje pranešimą, kurį vėliau kitas procesas perskaito.

Šis mechanizmas leidžia keistis struktūriniais duomenimis, turinčiais sekančius atributus:

- pranešimo tipas (naudojamas pranešimų multipleksavimui);
- pranešimo duomenų ilgis baitais (g. b. 0);
- patys duomenys (jei ilgis nėra lygus 0, tai jie gali būti struktūriniai).

Pranešimai branduolio adresinėje erdvėje yra saugomi vienos krypties sąraše. Kiekvienai eilei yra sukuriama eilės antraštė (msqid_ds), kurioje yra saugomos privilegijos (msg_perm), būsena (msg_cbytes – baitų skaičius, msg_qnum – pranešimų skaičius eilėje), nuorodos į pirmąjį (msg_first) ir paskutinįjį (msg_last) eilės pranešimą (3.19 pav.).



3.19 pav. Pranešimų eilės struktūra.

Norint sukurti naują pranešimų eilę arba gauti priėjimą prie jau esančios iškviečiama sisteminė komanda:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflag);
```

Ši funkcija grąžina identifikatorių, kurį naudodama funkcija gali:

patalpinti pranešimą į eilę su funkcija `msgsnd(2)`;

gauti tam tikro tipo pranešimą su funkcija `msgrcv(2)`;

kontroliuoti pranešimus su funkcija `msgctl(2)`.

Sisteminės komandos, realizuojančios minėtas operacijas:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz,
long msgtyp, int msgflg);
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

, kur `msgid` yra IPC objeto identifikatorius, `msgp` – yra nuoroda į buferį, kuriame yra pranešimo tipas ir duomenys, kurio dydis yra lygus `msgsz` baitams. Buferyje yra sekantys laukai:

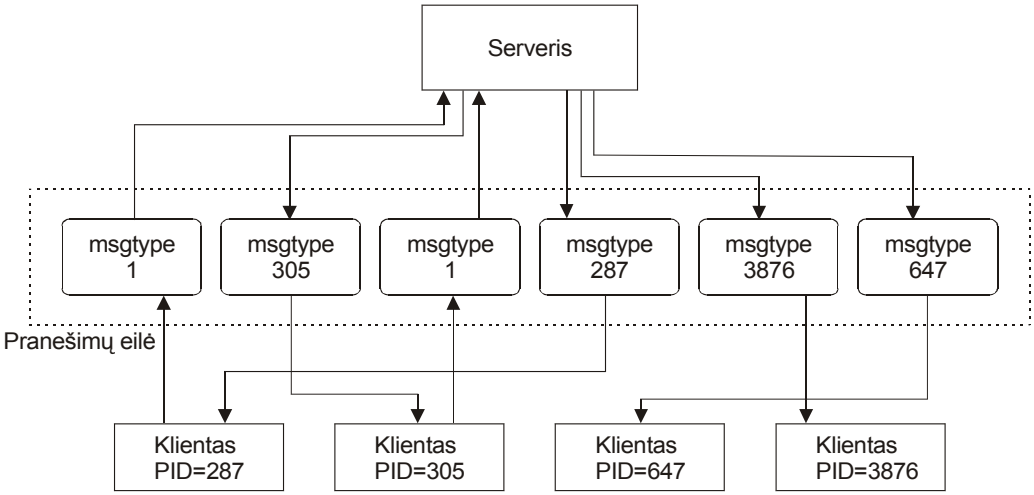
```
long msgtype;
char msgtext[];
```

Funkcijoje `msgrcv` parametras `msgtyp` nurodo reikalaujamo pranešimo tipą. Jei msgtyp reikšmė yra lygi 0 – grąžinamas pirmasis eilėje esantis pranešimas; jei mažesnis už 0 – grąžinamas mažiausios tipo reikšmės pranešimas.

Pranešimai turi labai gerą savybę – viena pranešimų eile galima multipleksiškai perduoti daugelio tipų pranešimus. Tam naudojamas `msgtype` atributas, kurio pagalba bet kuris procesas gali filtruoti pranešimų eilę, ieškodamas t.t. tipo pranešimo.

Peržiūrėsim tipišką pranešimų eilės panaudojimo pavyzdį, kai procesas – serveris naudodamas vieną pranešimų eilę keičiasi duomenimis su keliais klientais. Visi pranešimai, pasiųsti serveriui turi tipo reikšmę 1, o klientams – jų PID. Klientas perduodamas duomenis į jų struktūrą įsiūna savo PID, kurį serveris vėliau naudoja perduodamas informaciją klientui (3.20 pav.).

Naudojant tipo identifikatorių galima keisti pranešimų eiliškumą.



3.20 pav. Multipleksinis pranešimų perdavimas naudojant vieną eilę.

3.13.5 Semaforai

Procesų darbo sinchronizacijai, tiksliau priėjimo prie bendrai naudojamų resursų valdymui yra naudojami semaforai. Tai yra viena iš UNIX IPC formų, tačiau ji nėra skirta didelio duomenų kiekio perdavimui, kaip FIFO ar pranešimai, tačiau jie atlieka savo darbą pagal pavadinimą – leidžia arba neleidžia naudotis vienu arba kitu resursu.

Trumpas semaforo pavyzdys: tarkim yra koks nors bendrai naudojamas resursas, tarkim failas. Kai vienas iš procesų atlieka rašymą į tą failą, jį reikia blokuoti, kad kiti procesai nesugadintų šios operacijos vientisumo. Šiam uždaviniui, su šiuo resursu yra susiejamas skaitliukas, kuris įgyja reikšmes 1 – resursas prieinamas ir 0 – neprieinamas. Prieš pradėdamas darbą su resursu, procesas privalo patikrinti skaitliuko (semaforo) reikšmę, jei ji yra lygi 1, procesas užblokuoja resursą, t.y. nustato skaitliuko reikšmę lygią 0, atlieka reikiamas operacijas ir, po to jį atlaisvina.

Sklandaus semaforo darbo užtikrinimas yra įmanomas tenkinant dvi sąlygas:

semaforo reikšmė turi būti prieinama visiems suinteresuotiems procesams, t.y. ji turi būti saugoma branduolio adresinėje erdvėje;

operacijos su semaforu privalo būti atomarinės, t.y. nepertraukiamos. Kitokiu atveju, kai vienas procesas tikrins semaforo reikšmę, kitas ją pakeis. Atomarinę (kritinę) operaciją gali užtikrinti tik branduolio režime dirbanti programa.

Todėl semaforai yra sisteminis resursas; priėjimas prie jo yra suteikiamas per sistemines komandas.

System V semaforai turi tokias savybes:

- semaforu yra ne vienas, o iš kelių skaitliukų sudaryta grupė, kuri yra charakterizuojama viena požymių aibe, t.y. vienu deskriptoriumi, priėjimo teisėmis ir pan.;
- kiekvienas skaitliukas gali įgyti ne tik 0 ar 1, o bet kokią reikšmę `unsigned short` intervale.

Kiekvienai semaforų grupei, toliau paprastai – semaforui, branduolys palaiko duomenų struktūrą `semid_ds`:

```
struct semid_ds {
  struct ipc_perm sem_perm;      // pri•jimo teisi• aprašymas
  struct sem *sem_base;    // nuoroda • pirm•j• semafor• masyvo element•
  ushort sem_nsems; // semafor• skai•ius grup•je
  time_t sem_otime; // paskutin•s operacijos laikas
```

```
  time_t sem_ctime; // paskutinio modifikavimo laikas
};
```

Konkretus semaforas yra aprašomas sekančioje struktūroje:

```
struct sem {
  ushort semval;    //semaforo reikšm•
  pid_t sempid;     //proceso, kuris atliko paskutin• operacij• PID
  ushort semncnt;   //proces• skai•ius, laukian•i• semval>0
  ushort semzcnt;   // proces• skai•ius, laukian•i• semval=0
};
```

Semaforo deskriptorius yra gaunamas panaudojant sisteminę komandą

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget (key_t key, int nsems, int semflag);
```

Operacijos su semaforu yra atliekamos naudojant `semop` komandą:

```
int semop (int semid, struct sembuf *semop, size_t nops);
```

, kur antruoju argumentu yra perduodamos operacijos, o trečiuoju – jų skaičius. Svarbu yra tai, kad ši funkcija užtikrina operacijos atomariškumą kitų procesų atžvilgiu. Kiekvienas operacijų rinkinio elementas `semop` yra aprašomas sekančia struktūra:

```
struct sembuf {
  short sem_num;    // semaforo grup•je numeris
  short sem_op;           // operacija
  short sem_flg;          // operacijos atributai
};
```

UNIX sistemoje yra galimos trys operacijos su semaforais:

1. semop>0, tai semval=semop;
2. `semop=0`, tai procesas laukia, kol bus `semval=0`;
3. `semop<0`, tai procesas laukia, kol `semval•semop`, ir tada `semval+=semop`.

Pirmoji operacija betarpiškai pakeičia semaforo reikšmę, antroji – ją patikrina, o trečioji – patikrina ir pakeičia.

OS neapriboja procesų veiksmų susijusių su semaforais, t.y. jie patys turi nuspręsti kokia semaforo reikšmė yra leidžianti, kokia draudžianti ir pan.

Pavyzdžiui panagrinėsime binarinio (reikšmės 1 arba 0) semaforo panaudojimo atvejį. Pirmuoju atveju 0 – leidžia, o 1 – blokuoja. Apibrėšime dvi operacijas – blokuojančią ir nuimančią blokavimą:

```
static struct sembuf sop_lock[2] {
       0, 0, 0,           // laukiame, kol semaforo reikšm• taps lygi 0
       0, 1, 0            // padarome j• lygi• 1
};

static struct sembuf sop_unlock[1] {
       0, -1, 0
};
```

Norėdamas užblokuoti resursą procesas iškviečia komandą

```
semop (semid, &sop_lock[0], 2);
```

, kuri atlieka dvi operacijas:

1. laukia, kol resursas taps laisvu, jei semaforas yra lygus 1, procesas bus sustabdytas iki kol resursas atsilaisvins ir tada
2. užblokuoja resursą, nustatydamas semaforo reikšmę lygią 1.

Norėdamas atlaisvinti resursą, procesas iškviečia komandą

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

, kuri sumažina semaforo reikšmę vienetu. Jei kuris nors procesas buvo sustabdytas belaukdamas šio resurso, jis prikeliamas ir gali jį užsiblokuoti.

Paimkime kitą pavyzdį su priešingomis semaforo reikšmėmis, t.y. 1 – resursas laisvas, 0 – užblokuotas. Šiuo atveju reikalingos truputi kitokios operacijos:

```
static struct sembuf sop_lock[1] {
       0, -1, 0
};

static struct sembuf sop_unlock[1] {
       0, 1, 0
};
```

Ir atitinkamai yra iškviečiamos komandos resurso blokavimui ir atlaisvinimui:

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

Antruoju atveju operacijos yra paprastesnės, tačiau šis atvejis turi potencialų pavojų: kai semaforas yra sukuriamas, jo reikšmės yra priskiriamos 0, t.y. iš karto resursas tampa užblokuotu. Norint išvengti dviprasmiškų situacijų, semaforą sukūręs procesas turi iškviesti resurso atblokavimo komandą. Tarkim, gali būti panaudotas sekantis sprendimas:

```
if((semid = semget(key, nsems, perms | IPC_CREAT | IPC_EXCL))<0){
       if(errno == EEXIST) semid = semget(key, nsems, perms);
       else return –1;
}else semop(semid, &sop_unlock[0], 1);
```

3.13.6 Bendrai naudojama atmintis

Intensyviai naudojant aukščiau išvardintus IPC mechanizmus gali kristi sistemos efektyvumas; ypač jei keičiamasi nemažais duomenų kiekiais. Tai susiję su tuo, kad duomenys yra kopijuojami iš proceso adresinės erdvės į branduolio erdvę, o po to vėl į proceso erdvę. Bendra atmintis suteikia tokį servisą, kad nereikia kopijuoti duomenų, o naudojama viena duomenų kopija.

Bendrai naudojamos atminties scenarijus galėtų būti toks:

1. serveris, naudodamas semaforą, gauna priėjimą prie resurso;
2. serveris užblokuoja resursą ir vykdo duomenų rašymą;
3. pabaigęs darbą serveris atlaisvina resursą naudodamas semaforą;
4. klientas, gavęs priėjimą prie resurso, jį užblokuoja ir skaito;
5. baigęs operacijas, klientas atlaisvina resursą.

Kiekvienai bendrai naudojamos atminties sričiai branduolys saugo atitinkamą struktūrą

```
struct shmid_ds {
  struct ipc_perm shm_perm;      // Naudojimo teis•s
  int shm_segsz;           // Segmento dydis
  ushort shm_nattch;       // Proces• – naudotoj• skai•ius
  time_t shm_atime;        // Paskutinio prisijungimo laikas
  time_t shm_dtime;        // Paskutinio atsijungimo laikas
  time_t shm_ctime;        // Paskutinio modifikavimo laikas
};
```

Bendros atminties sukūrimui arba gavimui priėjimo prie jau esančios naudojama sekanti sisteminė komanda:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflag);
```

Sekmės atveju funkcija grąžina IPC deskriptorių, o nesekmės –1. Ši funkcija tik sukuria bendrai naudojamą atminties resursą, tačiau neleidžia dirbti su ja. Norint ją panaudoti yra iškviečiama prijungimo (attach) funkcija -

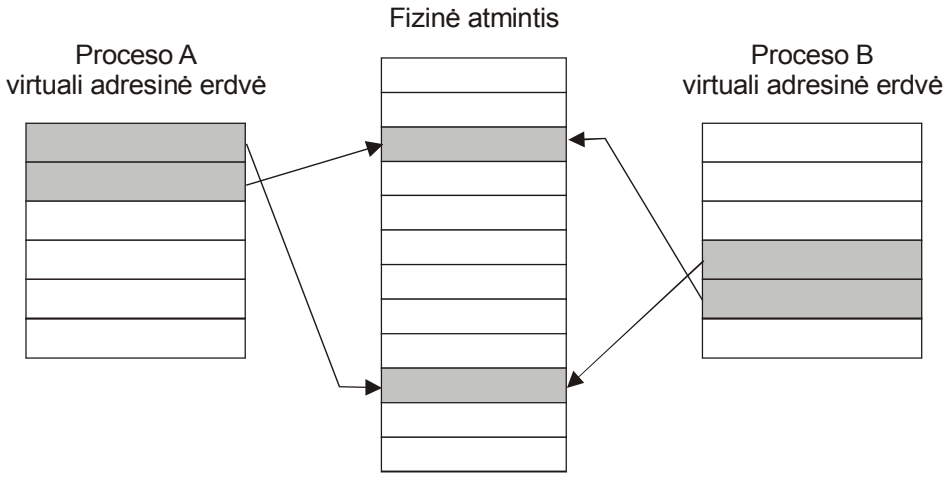
```
char *shmat(int shmid, char *shmaddr, int shmflag);
```

, kuri grąžina atminties srities pradžios adresą, lygų prieš tai nurodyto `size` dydžiui. Šioje erdvėje bendraujantys procesai gali talpinti jiems reikalingas duomenų struktūras. Adreso gavimo taisyklės yra sekančios:

- jei argumentas `shmaddr` yra lygus nuliui, tai sistema parenka adresą savarankiškai;
- jei argumentas `shmaddr` nėra lygus nuliui, tai grąžinamo adreso reikšmė priklauso nuo `SHM_RND` požymio buvimo `shmflag` argumente:
- jei `SHM_RND` nėra, tai sistema prideda bendrą atmintį prie nurodyto adreso `shmaddr`;
- jei `SHM_RND` yra, tai sistema grąžina adresą, gautą pridedant bendros atminties pradžios adresą ir suapvalintą iki `SHMLBA` `shmaddr` argumentą.

Pagal nutylėjimą, bendrai naudojama atmintis prijungiama skaitymui ir rašymui, tačiau nurodžius `shmflag SHM_RDONLY` požymį atmintis prijungiama tik skaitymui.

Tokiu būdu, keli procesai gali nurodyti t.t. atminties plotelį įvairiose savo virtualios erdvės vietose (3.21 pav.).



3.21 pav. Bendrai naudojami fizinės atminties puslapiai

Baigus darbą procesas atjungia bendrai naudojamą atminties sritį su komanda

```
int shmdt (char *shmaddr);
```

Dirbant su bendra atmintimi yra svarbu tvarkingai sinchronizuoti priėjimą prie resurso. Tam dažniausiai naudojami semaforai, kurių skaičius priklauso nuo konkretaus bendros atminties panaudojimo atvejo.

3.14 Soketai

UNIX BSD kūrėjai projektuodami IPC mechanizmą rėmėsi keletu reikalavimų:

- pirma, ryšys tarp procesų turi būti unifikuotas, nepriklausomai nuo to ar bendrauja vieno kompiuterio procesai, ar procesai, dirbantys skirtinguose tinkle sujungtuose kompiuteriuose. Optimaliausias šio reikalavimo sprendimas yra modulinė IPC architektūra, kuri remiasi bendra UNIX tinklo palaikymo posisteme. Ją naudojant butų galima naudoti įvairius: objektų adresavimo schemas, duomenų perdavimo protokolus ir pan.
- antra, bendravimo charakteristikos turi būti unifikuotos, t.y. turi turėti galimybę paprašyti tam tikro ryšio tipo, tarkim, virtualaus kanalo (virtual circuit) arba datagramų (datagram).

Įvesime porą sąvokų:

- komunikacinis domenas (communication domain) – apibrėžia nustatytų bendravimo charakteristikų rinkinį, kitaip tariant, bendravimo aplinka;
- soketas (socket) – objekto (proceso) komunikacinis mazgas, atliekantis duomenų priėmimo ir išsiuntimo funkciją. Soketai yra sukuriami t.t. komunikacinio domeno rėmuose (kaip failai – failų sistemos). Soketai yra adresuojami t.t. sveiku skaičiumi – deskriptoriumi, tačiau jie, ne kaip failai, yra virtualūs objektai, kurie egzistuoja tol, kol į juos rodo nors vienas iš procesų.

Soketai yra skiriami į tipus pagal keletą charakteristikų:

- duomenų perdavimo eiliškumas,
- duomenų dubliavimo eliminavimas,
- patikimas duomenų perdavimas,
- pranešimo ribų išsaugojimas,
- ekstra, skubių duomenų perdavimas ir
- išankstinis sujungimo sukūrimas.

Tarkim, aukščiau nagrinėti kanalai tenkina pirmąsias tris charakteristikas. Skubus duomenų perdavimas yra realizuojamas tada, kai reikia greitai, ne normaliu būdu perduoti duomenis, į kuriuos programa turi staigiai reaguoti. Išankstinis virtualaus kanalo sukūrimas yra naudingas, kai pradžioje yra nustatomi identifikaciniai parametrai, o po to jie automatiškai yra prijungiami prie kiekvieno iš paketų.

BSD UNIX sistemose yra realizuoti sekantys soketų tipai:

1. datagramų soketas (datagram socket) – teoriškai nepatikimas be išankstinio sujungimo duomenų paketų perdavimas;
2. srauto soketas (stream socket) – patikimas, be pranešimo ribų išsaugojimo baitų srauto perdavimas; palaiko ekstra duomenų perdavimą;
3. paketų soketas (packet socket) – patikimas, nuoseklus, be dubliavimo, su išankstiniu ryšio sukūrimu, duomenų perdavimas; išsaugomos pranešimų ribos;
4. žemo lygio soketas (raw socket) – suteikia betarpišką priėjimą prie protokolo.

Tam, kad procesai galėtų bendrauti soketų pagalba yra reikalinga vardų sritis, t.y. soketus privalome kažkaip identifikuoti. Soketo vardas turi prasmę tik tame domene, kuriame jis yra sukurtas. Jei System V sistemose yra naudojama rakto samprata, tai čia – adreso.

3.14.1 Soketų programinis interfeisas (API)

3.8 lentelė. Soketų tipai

Konstanta	Tipas
SOCK_DGRAM	Datagramos soketas
SOCK_STREAM	Srauto soketas
SOCK_SEQPACKET	Paketų soketas
SOCK_RAW	Žemo lygio soketas

Soketo bendravimo charakteris priklauso nuo tipo (3.8 lentelė), o bazines bendravimo savybes nurodo komunikacinis domenas. Sukuriant soketą procesas privalo nurodyti jo tipą ir komunikacinį domeną, kurio rėmuose buss naudojamas duotasis soketas. taip pat galime nurodyti ir konkretų bendravimo protokolą, tačiau jei jis nėra nurodytas, tai sistema parenka labiausiai tinkantį duotajame domene. Jei sistemoje nurodytame soketo domene nėra nei vieno protokolo (sistema nepalaiko domeno), tai soketo sukūrimo sisteminė komanda baigiasi su klaida.

Soketai yra sukuriami naudojant sekančią komandą:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

, kur jei protokolas nežinomas, reikia palikti 0. Sėkmės atveju, ši komanda grąžina skaičiuką, kuris bus naudojamas duotojo soketo adresavimui.

Iš tikrųjų, tai domenas nurodo protokolų šeimą (protocol family), galimą šiame domene. Tarkim yra galimos sekančios domeno reikšmės:

`AF_UNIX` lokalus tarp-procesinis ryšys vienos UNIX OS rėmuose. Naudojami vidiniai protokolai;
`AF_INET` nutolusių sistemų bendravimo domenas. Internet protokolai (TCP/IP);
`AF_NS` nutolusių sistemų bendravimo domenas. Xerox NS protokolas.

Domenas ir protokolų šeima nurodo procesų bendravimo objektų adresinę erdvę, kurią sudaro leistini adresai ir jų formatai, todėl domenų konstantų pavadinimai prasideda prefiksu AF (address family). Kartais yra naudojami ir PF (protocol family) prefiksai.

Konkretūs domenai palaiko tam tikrus bendravimo protokolus – 3.9 lentelėje yra pateikti du domenai – `AF_UNIX`, skirtas vienos OS procesų bendravimui ir `AF_INET`, skirtas procesų, esančių skirtingose mašinose, sujungtose į tinklą.

3.9 lentelė. Protokolų palaikymas domenuose.

Protokolas	**Domenas**	
	AF_UNIX	**AF_INET**
`SOCK_DGRAM`	Palaikomas	Palaikomas
`SOCK_STREAM`	Palaikomas	Palaikomas
`SOCK_SEQPACKET`	Nepalaikomas	Nepalaikomas
`SOCK_RAW`	Nepalaikomas	Palaikomas

Taip pat yra leistinos ne visos soketo tipo ir komunikacinio protokolo kombinacijos. Tarkim, `AF_INET` domene galimi protokolų variantai yra pateikti 3.10 lentelė.

3.10 lentelė. Interneto domeno soketų protokolų variantai.

Soketas	**Protokolas**
`SOCK_DGRAM`	`IPPROTO_UDP (UDP)`
`SOCK_STREAM`	`IPPROTO_TCP (TCP)`
`SOCK_RAW`	`IPPROTO_ICMP (ICMP)`
`SOCK_RAW`	`IPPROTO_RAW (IP)`

Šie protokolai yra TCP/IP šeimos protokolai.

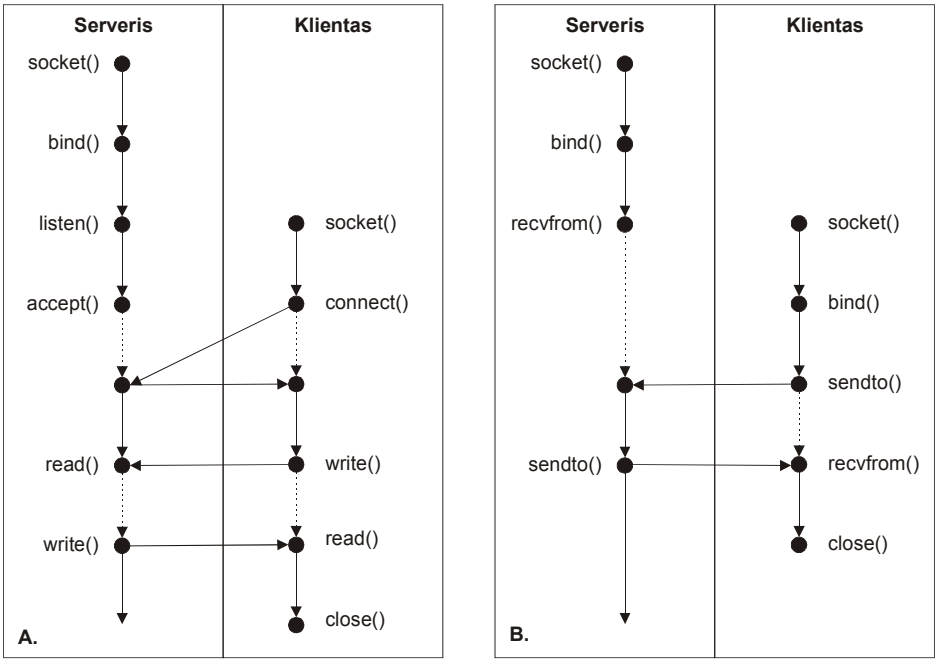
Soketo sukūrimas nėra paties komunikacinio mazgo sukūrimas. Siekiant vienareikšmiškai identifikuoti soketą, būtina jį pozicionuoti tam tikro domeno adresinėje erdvėje. Bendruoju atveju, kiekvienas komunikacinis kanalas yra charakterizuojamas dviem mazgais – duomenų šaltiniu ir gavėju, ir gali būti identifikuojamas penkiais parametrais:

- komunikacinis protokolas,
- vietinis adresas,
- vietinis procesas,
- nutolęs adresas ir
- nutolęs procesas.

Adresas nurodo operacinę sistemą (arba tinklo mazgą, host'ą), o procesas – konkrečią šioje operacinėje sistemoje dirbančią aplikaciją. Tačiau konkrečias reikšmes apibrėžia nustatytas komunikacinis domenas.

Kadangi soketo sukūrimo metu yra nurodomas tik komunikacinis protokolas, tai prieš pradedant duomenų perdavimą yra būtina nurodyti kitus keturis parametrus. Šių parametrų perdavimas ir nustatymas privalo būti unifikuotas abiejose bendraujančiose pusėse.

3.22.A paveiksle yra pateiktas kanalo inicijavimo ir duomenų perdavimo seka su iš anksto sukuriamu sujungimu, o 3.22.B paveiksle – be išankstinio sujungimo (datagramų perdavimas).



3.22 pav. Bendravimas tarp procesų sukuriant virtualų kanalą (A) ir be virtualaus kanalo (datagramos) (B).

Kaip matyti, prieš pradedant duomenų perdavimą atliekamas surišimas (binding), kurio metu yra perduodama papildoma informacija. Surišimas gali būti atliekamas naudojant sisteminę funkciją `bind(2)`:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *localaddr, int addrlen);
```

, kur `sockfd` yra soketo deskriptorius, `localaddr` yra soketo vietinis adresas, o `addrlen` – adreso struktūros ilgis. Ši funkcija suriša sukurtą soketą su vietiniu adresu, t.y. sukuria vieną komunikacinio kanalo mazgą.

Vidinio UNIX domeno adresas yra nurodytas `<sys/un.h>` faile:

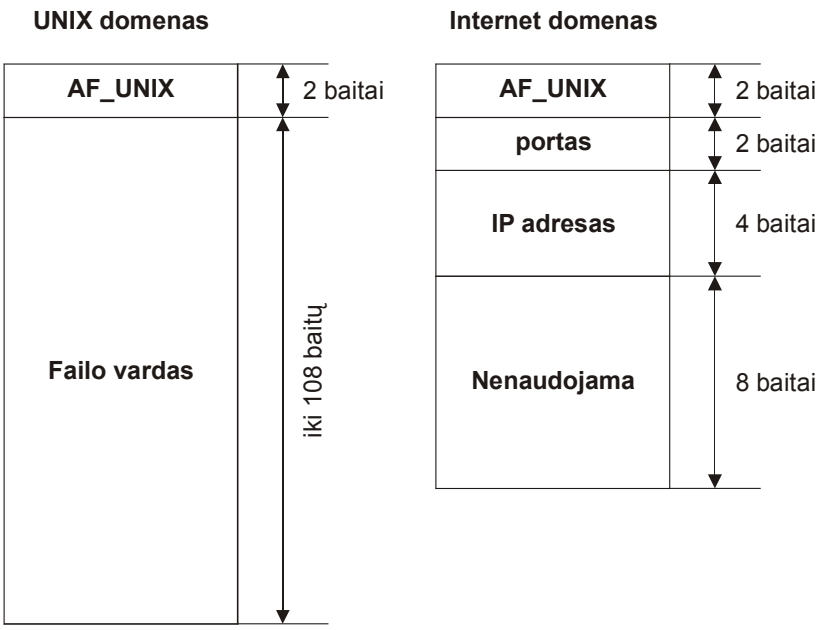
```
struct sockaddr_un {
       short sun_family; // ==AF_UNIX
       char sun_path[108];
};
```

Vienos operacinės sistemos rėmuose proceso adresams sudaryti yra naudojami failų vardai.

TCP/IP protokolų šeimos domene yra nurodomas ne tik procesas, bet ir tinklo operacinė sistema – host'as. Adreso struktūra yra nurodyta `<netinet/in.h>` faile ir turi sekančią išraišką:

```
struct sockaddr_in {
       short sin_family; // ==AF_INET
       u_short sin_port;
       struct in_addr sin_addr;
       char sin_zero[8];
};
```

Šioje struktūroje host'o adresas yra 4 baitų (32 bitų) ilgio `sin_addr` parametras, o procesas nurodomas 2 baitų (16 bitų) `sin_port` parametre. Soketų adresai yra pateikti 3.23 paveiksle.



3.23 pav. Soketų adresų formatai.

`bind(2)` funkcija yra iškviečiama sekančiais atvejais:

- serveris užregistruoja savo adresą. Šis adresas turi būti iš anksto žinomas klientams;
- klientas užregistruoja savo adresą tuo atveju, kai yra bendraujama be išankstinio ryšio sukūrimo (datagramos), šį adresą naudos serveris, siųsdamas duomenis klientui;
- klientas, net ir naudodamas virtualų kanalą užregistruoja savo adresą.

Virtualų kanalą sukuria klientas, iškviesdamas sisteminę komandą `connect(2)`:

```
int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

Ši sisteminė komanda gali būti naudojama ir prieš datagramų naudojimą; tokiu būdu nėra sukuriamas virtualus kanalas, tačiau yra išsaugomi sujungimo parametrai, kurie bus kartojami kiekvieno duomenų paketo siuntimo metu.

Sekanti komanda yra naudojama serverio pusėje sistemos informavimui apie tai, kad serveris yra pasiruošęs klientų priėmimui:

```
int listen(int sockfd, int backlog);
```

, kur backlog nurodo maksimalų prisijungimo su serveriu laukiančių klientų skaičių. Jei eilinis klientas viršija šį parametrą, jų sisteminė komanda `connetc(2)` baigia darbą su klaida: `AF_UNIX` domene – `ECONNREFUSED`, `AF_INET` – `ETIMEDOUT`. Faktinį klientų užklausų apdorojimą atlieka sekanti komanda:

```
int accept(int sockfd, struct sockaddr *clntaddr, int *addrlen);
```

Ji paima pirmutinį eilėje laukiantį klientą ir grąžiną kliento adresą. Tipinis serverio darbo scenarijus:

```
sockfd=socket(...);
bind(sockfd, ...);
listen(sockfd, ...);
for( ; ; ){
      newsockfd=accept(sockfd, ...);
      if(!fork()){
            close(sockfd);
            ...
            exit(0);
```

```
        }else
                close(newsockfd);
}
```

Tuo atveju, kai yra sukuriamas virtualus kanalas galima naudoti standartines `read` ir `write` funkcijas. Kitu, pvz. datagramų atveju reikia naudotis specialiomis funkcijomis: `send(2), sendto(2), recv(2), recvfrom(2)`.

3.15 IPC sistemų palyginimas

3.11 lentelėje yra pateikiamas IPC mechanizmų palyginamas.

Charakteristika	**IPC sistema**					
	Kanalai	**FIFO**	**Pranešimai**	**Semaforai**	**Bendra atmintis**	**Soketai**
Adresų sritis	-	Failo vardas	Raktas	Raktas	Raktas	Adresas
Objektas	Sisteminis kanalas	Vardinis kanalas	Pranešim• eil•	Semaforas	Bendra atmintis	Komunikacinis kanalas
Objekto sukūrimas	pipe()	mknod()	msgget()	semget()	shmget	socket()
Surišimas	pipe()	open()	msgget()	semget()	shmget	bind() connet()
Duomenų perdavimas	read() write()	read() write()	msgrcv() msgsnd()	semop()	Betarpinis atminties naudojimas	read() write() recv() send() recvfrom sendto()
Objekto sunaikinimas	close()	close() unlink	msgctl()	semctl()	shmdt()	close() unlink()

Greičiausias yra bendros atminties naudojimo mechanizmas, tačiau reikia sinchronizuoti procesų darbą.

Pranešimų naudojimas yra efektyvus, jei keičiamasi ne didesniais kaip 1 Kb dydžio duomenimis. Kitu atveju yra apkraunama sistema, padidėja sisteminių komandų skaičius.

IPC naudojimo intensyvumą sistemoje atskleidžia `sar -m` komanda. Jos išvedimas pateikia naudojamų IPC objektų skaičius per sekundę.

4 Failų posistemė

Dauguma duomenų UNIX sistemose yra saugoma failuose, kurie yra išdėstyti hierarchinėje duomenų struktūroje. Failai dažniausiai yra saugomi vietiniame diske, tačiau gali būti ir kituose kompiuteriuose, pvz. NFS (Network File System) leidžia tvarkyti failus nutolusiuose kompiuteriuose. Failai tai pat gali būti saugomi ir kitokiuose kaupikliuose, pvz. disketėse, CD-ROM įrenginiuose ir pan., tačiau mes nagrinėsime paprasčiausius kietuosius diskus.

Pirmoji pasirodė System V failų sistema s5fs, vėliau 4.2BSD sistemose buvo sukurta FFS (Fast File System). Pastaroji yra greitesnė, didesnio funkcionalumo ir patikimumo. Nors šiuolaikinės sistemos naudoja sudėtingesnes failų sistemas, tačiau pagrindinės idėjos išliko. Todėl mes peržiūrėsime šias pagrindines failų sistemas: System V – s5fs ir BSD – FFS.

Pradžioje, prieš pasirodant FFS UNIX sistemos palaikė tik vieną failų sistemą, t.y. jau kompiliavimo metu turėjo būti nurodyta, su kokia failų sistema dirbs ši operacine sistema. Šis nepatogumas buvo išspręstas panaudojus vadinamąją nepriklausomą arba virtualią failų sistemą, kuri dirba su įvairiomis failų sistemomis įvairiuose duomenų kaupikliuose. Pradžioje peržiūrėsime Sun Microsystems sukurtą virtualią failų sistemą, kuri yra SVR4 sistemų standartas.

4.1 Failų sistemos hierarchija

Nors UNIX failų sistema nėra labai tvarkinga, tačiau reikia bendrą katalogų ir failų sistemų paskirtį reikia žinoti:

`/` (root) failų sistemoje yra `/` katalogas, branduolys (`/unix`, `/vmunix`, `/kernel` ar `/stand`), įrenginių failai (`/dev`), sistemos konfigūracijos failai (`/etc`), minimali aibė komandų (`/sbin`, `/bin`) ir, kartais, laikinų failų katalogas (`/tmp`).

`/usr` – failų sistemoje yra saugomos pagrindinės sistemos programos, man failai ir bibliotekos (`/usr/lib`, iš kurio yra nuorodos į `/lib`).

`/var` – šioje failų sistemoje yra saugomi įvairūs spool, log ir kiti, greitai ir dažnai kintantys failai.

`/home` – vartotojų namai, kurie paprastai yra saugomi atskiroje failų sistemoje, pvz. `/home`.

4.2 Failų tipai

4.2.1 Paprasti failai

Tai yra paprasti baitų rinkiniai kurių struktūra OS sistemai yra visai nesvarbi.

4.2.2 Katalogai

Kataloguose yra saugomos nuorodos į failus. Sakoma, kad failai yra kataloge. Katalogas yra sukuriamas komanda `mkdir`, tuščias pašalinamas `rmdir`, o netuščias – `rm -R` (rekursyviai).

“.” – naudojamas nurodyti einamąjį katalogą, “..” – katalogo tėvinis katalogas.

Failų vardai kataloge tėra nuorodos į t.t. baitų rinkinį diske – tai yra vadinamasis kietoji jungtis (hard link). Kietąją jungtį galima sukurti su komanda ln, pvz. ln oldfile newfile sukuria dar vieną nuorodą į failą oldfile. Failų atributai, tokie kaip nuosavybė, teisės ir pan. galioja visoms kietoms jungtims.

4.2.3 Simbolinių ir blokinių įrenginių failai

Tai failai, kurių pagalba yra bendraujama su aparatūrinės įrangos tvarkyklėmis. Šie failai nėra tvarkyklės, jie tik persiunčia komandas tvarkyklėms.

Simboliniai aparatūrinės įrangos failai perduoda duomenų srautą nuosekliai, t.y. srauto buferį realizuoja tvarkyklės. Blokinių failų tvarkyklės dirba su dideliais bitų blokais ir, tokiu atveju, srauto buferius realizuoja branduolys.

Aparatūrinės įrangos failai yra sukuriami su komanda `mknod`. Dauguma sistemų suteikia `/dev/MAKEDEV` programą, kuri sukuria šiuos failus.

4.2.4 Vietiniai kanalai (domain sockets)

Tai yra tarp procesiniai ryšio kanalai. Procesų sukurtus kanalų failus gali skaityti vartotojai, tačiau juos modifikuoti gali tik procesai, kuriuos betarpiškai sieja konkretus kanalas. Šiuos kanalus sukuria socket() sisteminė funkcija, o sunaikina `rm` (arba `unlink()`), kai jis tampa nebereikalingas.

4.2.5 FIFO kanalai (Named Pipes)

Kaip ir vietiniai, FIFO kanalai sukuria ryšį tarp procesų, tačiau jie realizuoja FIFO duomenų struktūrą.

4.2.6 Simbolinė (minkšta) nuoroda

Simbolinė nuoroda, skirtingai nuo kietosios, yra surišama ne tiesiogiai, per adresą, o per vardą. Kai branduolys vykdo operaciją simbolinės nuorodos kataloge, jis iš tikrųjų dirba

kataloge, į kurį rodo nuoroda. Simbolinę nuorodą galime sukurti naudojant tiek absoliutų, tiek santykinį kelią. Pavyzdžiui, jei norime kataloge `/usr/include/ufs` turėti katalogo `/usr/include/bsd/sys/ufs` turinį, rašome tokią komandą:

```
# ln –s ../../ufs /usr/include/bsd/sys/ufs
```

4.3 Failų atributai

Failo atributai nurodo failo savininką (-us), tipą ir pan. Kiekvienas failas turi 16 bitų (vieną žodį) skirtą failo atributams saugoti. 9 leidimo bitai nurodo, kas gali skaityti, modifikuoti ir vykdyti failą. Su sekančiais 3-mis bitais, kurie valdo failo vykdymą, jie sudaro failo būsena (mode). Likę 4 bitai nurodo failo tipą, kuris yra nustatomas failą sukuriant ir vėliau nebekeičiamas. Kitus 12 bitų gali keisti failo savininkas arba super-vartotojas (root'as) naudodami komandą chmod (change mode).

Failai taip pat turi setuid (4000xoc) ir setgid (2000xoc) bitus, kurie nurodo failo savininkus – vartotoją ir grupę. Naujai sukurti failai įgauna katalogo, o ne jį sukūrusio vartotojo setgid reikšmę. Taip yra palengvinama apsikeitimo failais funkcija.

Vienas iš senų laikų atavizmų UNIX sistemose yra vadinamasis "kibusis" (sticky) bitas (1000xoc). Jo paskirtis PDP mašinose, kuriose, beje, UNIX pradėjo savo dienas, buvo požymis, kad failas turi likti atmintyje ar swap įrenginyje. Šiuo metu, kai atmintis yra labai pigi, kibusis bitas yra ignoruojamas, tačiau jis vis dar lieka svarbus operuojant failais – ne failo savininkas ar super-vartotojas negali failo ištrinti, nors tokias teises ir turi.

9 leidimų bitai yra organizuoti sekančiai: 400, 200 ir 100 yra failo savininko, 40, 20 ir 10 – grupės, 4, 2 ir 1 – visų kitų vartotojų, "pasaulio" leidimo bitai. Didžiausios reikšmės (400, 40 ir 4) nurodo leidimą skaityti failą, viduriniosios (200, 20 ir 2) – rašymo ir mažiausios (100, 10 ir 1) – vykdymo. Failo trynimą ir pervadinimą valdo katalogo, kuriame betarpiškai yra failas, leidimo bitai.

Vykdymo bitai leidžią failą vykdyti ir nurodo koks yra jo tipas. Jis gali būti binarinis – betarpiškai vykdomas procesoriuje ir skriptas – pradžioje interpretuojamas shell'u (ar kokia kita programa). Interpretatorių nurodo pirmoji skripto eilutė:

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

, jei šios eilutės nėra – bandoma vykdyti einamajame shell'e.

Katalogo vykdymo bitas leidžia katalogą naudoti kelyje. Vykdymo ir skaitymo bitų kombinacija leidžia peržiūrėti katalogo turinį, o vykdymo ir rašymo – kurti, trinti ir pervadinti failus kataloge.

Failo atributai yra peržiūrimi su komanda ls –l.

```
% ls –l /bin/sh
-rwxr-xr-x   1      root   bin    85924  Sep 27 1997   /bin/sh
```

Pirmajame informacijos lauke yra pateikiamas failo tipas ir būsena. Failo tipų kodai yra pateikiami 4.1 lentelėje.

4.1 lentelė. Failo tipo kodai.

Failo tipas	**Kodas**
Paprastas failas	-
Katalogas	d
Simbolinio įrenginio failas	c
Blokinio įrenginio failas	b
UNIX vietinis kanalas	s
FIFO kanalas	p
Simbolinė (minkšta) nuoroda	l

Toliau devyni leidimo simboliai: r – skaitymas (read), w – rašymas (write) ir x – vykdymas (execute). Pirmi trys – savininko, sekantys trys – grupės, likusieji – pasaulio.

Sekantis laukas rodo kiek "kietų" nuorodų į failą yra padaryta, šiuo atveju – viena. Visi katalogai turi 2 nuorodas: vieną – tėvo kataloge, kitą – "." pačiame savyje.

Sekančiuose laukuose yra pateikiami failo savininkai: vartotojas ir grupė, failo dydis baitais, paskutinio modifikavimo data ir failo vardas.

Įrenginių failų informacijos laukai šiek tiek skiriasi – vietoj dydžio pateikiami pirmasis ir antrasis įrenginio numeriai.

FreeBSD ir kitos 4.4BSD sistemos leidžia failamas turėti papildomus bitus. Juos peržiūrėti galima su komanda ls –lo, o pakeisti su chflags komanda. Daugiau informacijos rasite chflags man puslapiuose.

4.2 lentelėje yra pateikiama leidimo bitų keitimo su komanda chmod sistema.

4.2 lentelė. Leidimo bitų kodai

xOC	xbin	Leidimas
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

chmod komanda gali naudoti aštuntainės sistemos skaičius arba simbolinius kodus (1 ir 3 lentelės kolonėlės).

Su komanda umask galima nustatyti kokie leidimai yra suteikiami pagal nutylėjimą. Tai galima nurodyti startavimo failuose (`.profile` arba/ir `.cshrc`).

4.4 Disko geometrijos apžvalga

Šiame skyrelyje pateiksime trumpą disko geometrijos ir terminologijos apžvalgą. 4.1 paveiksle yra pateikti disko dalys. Nors diskų principai pernelyg nepasikeitė, tačiau programos kuo toliau, tuo mažiau turi žinoti apie jų fizines charakteristikas.

Tipiškas kietasis diskas yra sudarytas iš kelių magnetinių plokštelių, kurios sukasi ant vienos ašies. Duomenys yra skaitomi ir rašomi naudojant mažas galvutes, kurios pirmuoju atveju perskaito magnetinių dalelių kryptį, o antruoju – ją pakeičia. Kietasis diskas yra tikslus ir sandarus įrenginys, todėl yra daug patikimesnis nei keičiamos duomenų kaupimo priemonės.

Pirmuosiuose diskuose buvo naudojama viena magnetinė plokštelė, todėl stengiantis padidinti disko talpą buvo didinamas disko diametras. Kai kurie diskai pasiekdavo net 1.5 metro diametrus, o jų talpa buvo tik 280K.

Šiuolaikiniai diskai turi kelias plokšteles. Abi plokštelės pusės yra įmagnetintos, tačiau tik viena jų yra naudojama duomenims; kitoje pusėje yra galvutės pozicionavimo informacija. Vienos dabartinės plokštelės talpa gali pasiekti 10G, tačiau jai galioja Moor'o dėsnis[1].

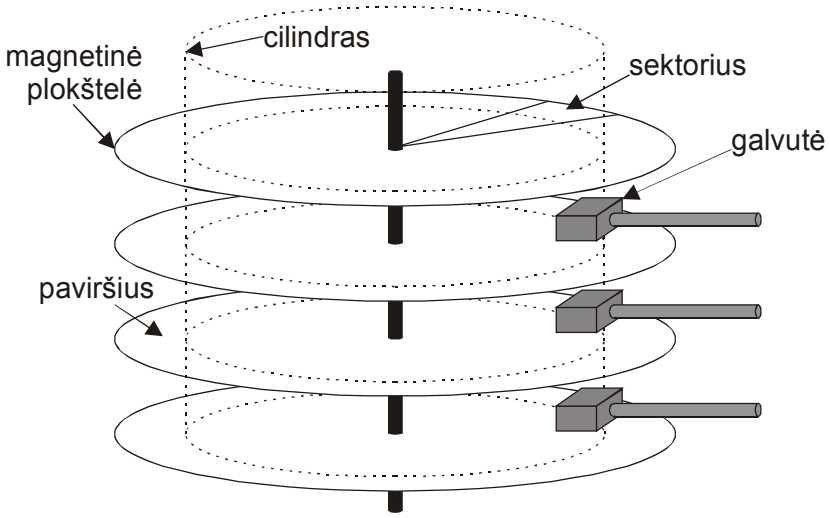
Magnetinės plokštelės sukasi pastoviu greičiu. Skaitymo galvutės yra labai arti magnetinių plokštelių, tačiau jų neliečia. Jei galvutė paliestų magnetinę plokštelę, tai butų vadinama galvutės susidūrimas (head crash) ir gali baigtis liūdnai.

Disko sukimosi greičiai sparčiai kyla. Seni diskai sukdavosi po 3600 ir 5400 RPM (rotations per minute). Šiuolaikinis standartas yra 7200 RPM, tačiau diskai su 10000 ir 15000 RPM tampa vis populiaresni. Šie diskai kaista, todėl jiems reikia geros ventiliacijos sistemos.

Kiekvienam paviršiui yra reikalinga po vieną galvutę. Pirmųjų diskų galvutės turėdavo sukarti didelius atstumus, tačiau šiuolaikiniuose diskuose, kuriuose yra naudojamos mažo diametro

[1] Moor'o dėsnis teigia, kad technologijos – CPU dažnis, disko talpa ir pan. – pakyla dvigubai kas 18 mėnesių.

plokštelės ir didelius jų kiekis – tai nebėra problema. Diskų diametras vis mažėja: nuo 14 colių prieš 20 metų, iki 3.5 ir mažesnių šiandien.



4.1 pav. Disko dalys ir jų terminai.

Galvutės judesys į t. t. reikiamą poziciją vadinamas kreiptis (seeking). Minimali galvutės pozicija, kurioje ji gali būti užfiksuota yra vadinama taku (track). Takai yra sudalinti į sektorius , kurie paprastai yra 512 bitų dydžio.

Takų, esančių skirtingose plokštelėse ir nutolusių vienodu atstumu nuo ašies aibė yra vadinama cilindru. Jei galvutės juda sinchroniškai, kaip yra daugumoje diskų, tai informacijos, esančios viename cilindre skaitymui nereikia atlikti galvutės kreipties. Nors galvutės juda labai greitai, tačiau jos vis dar atsilieka nuo disko apsisukimo greičio, todėl informacijos esančios viename cilindre skaitymas vyksta greičiau.

Dauguma failų sistemų bando išnaudoti disko fizines savybes ir atlieka tam tikras algoritmų optimizacijas, tačiau praktiškai tai vis dar lieka fikcija.

4.5 Bazinė System V failų sistema

Kiekvienas kietasis diskas yra sudarytas iš vienos ar kelių loginių dalių, vadinamųjų skyrių (partitions). Disko loginė struktūra – skyrių išsidėstymas ir jų dydžiai privalo būti žinomi prieš kompiliaciją. UNIX sistemose disko skyriai yra traktuojami kaip atskiri diskai, jiems taikomos tokios pat taisyklės, kaip ir atskiriems diskams.

Pavyzdžiui, diską gali sudaryti 4 skyriai, kurių kiekvienas turės savo failo sistemą. Kitu atveju vienas diskas turi vieną skyrių, t.y. failų sistemą.

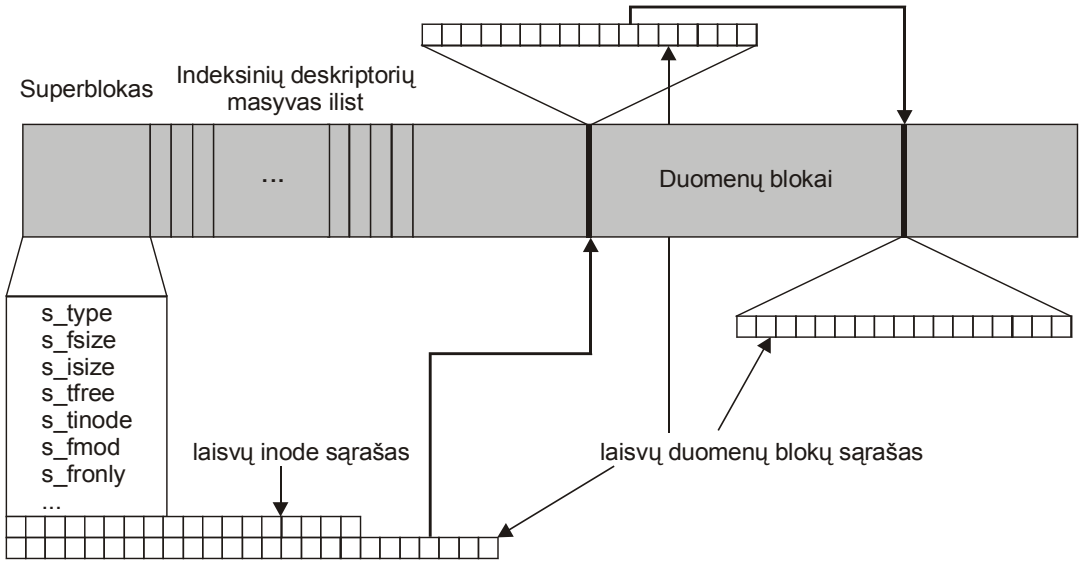
Failų sistema s5fs yra sudaryta iš trijų pagrindinių dalių (4.2 pav.):

superblokas (superblock). Jame saugoma bendra informacija apie failų sistemą, jos architektūrą, bendrą blokų ir indeksinių deskriptorių skaičių.

indeksų deskriptorių sąrašas (ilist). Visų failų sistemos failų metaduomenys. Juose saugoma statinė informacija apie failą ir nuorodą į jį patį. Branduolys kreipiasi į inode pagal masyvo ilist indeksą. Vienas iš inode yra šakninis, per kurį yra prieinama katalogų struktūra.Masyvo ilist dydis yra fiksuotas, nurodomas sukuriant failų sistemą. Tokiu būdu failų sistemoje s5fs yra fiksuotas failų sistemų dydis, nepriklausomai nuo jų dydžio.

failų blokai. Paprastų failų ir katalogų duomenys yra saugomi šiuose blokuose. Duomenų apdorojimas prieinamas per inode, kuriame yra saugomos nuorodos į failų blokus. Blokas yra

512 baitų dydžio, tačiau kai kurios sistemos palaiko didesnius, pvz. S51K SCO UNIX bloko dydis yra 1K.



4.2 pav. s5fs failų sistemos struktūra

4.5.1 Superblokas

Superbloke yra saugoma informacija apie visą failų sistemą, kuri gali būti reikalinga, tarkim, montuojant failų sistemą arba įterpiant naujus failus. Kiekvienoje failų sistemoje yra tik vienas superblokas, kuris yra patalpinamas skyriaus pradžioje. Superblokas yra nuskaitomas į atmintį primontuojant failų sistemą ir ten reziduoja iki jos numontavimo. Superbloke yra saugoma tokia informacija:

- failų sistemos tipas – s_type;
- failų sistemos dydis, įskaitant superbloką, ilist ir failų duomenų blokus – s_fsize;
- indeksų deskriptorių masyvo dydis – s_isize;
- laisvų failų blokų skaičius – s_tfree;
- laisvų inode skaičius – s_tinode;
- požymiai (modifikacijos – s_fmod, montavimo režimo – s_fronly);
- loginio bloko dydis (512, 1024, 2048);
- laisvų inode numerių sąrašas;
- laisvų blokų adresų sąrašas.

Visų laisvų inode ir blokų sąrašo saugojimas yra nepraktiškas, nes jis užima labai daug vietos, todėl yra saugoma tiktai dalis. Jei jie sunaudojami, OS branduolys peržiūri ilistą, suranda laisvus inode ir suformuoja naują laisvų inode sąrašą, kurį surašo į superbloką.

Tokios schemos failų blokams pritaikyti nėra galima, nes perskaičius duomenų bloką nėra įmanoma nustatyti ar jis tuščias. Superbloke yra saugomas tik vienas laisvų blokų blokas. Tai yra nuorodos į kitus blokus, kuriuose taip pat saugomi adresai ir t.t. Tokiu būdu, kai diskas yra pustuštis, turime pakankamai vietos saugoti laisvų blokų adresams. Kai jis pilnai užpildomas – visi laisvų blokų adresai telpa į vieną bloką, t.y. yra saugomi superbloke.

4.5.2 Indeksų deskriptoriai

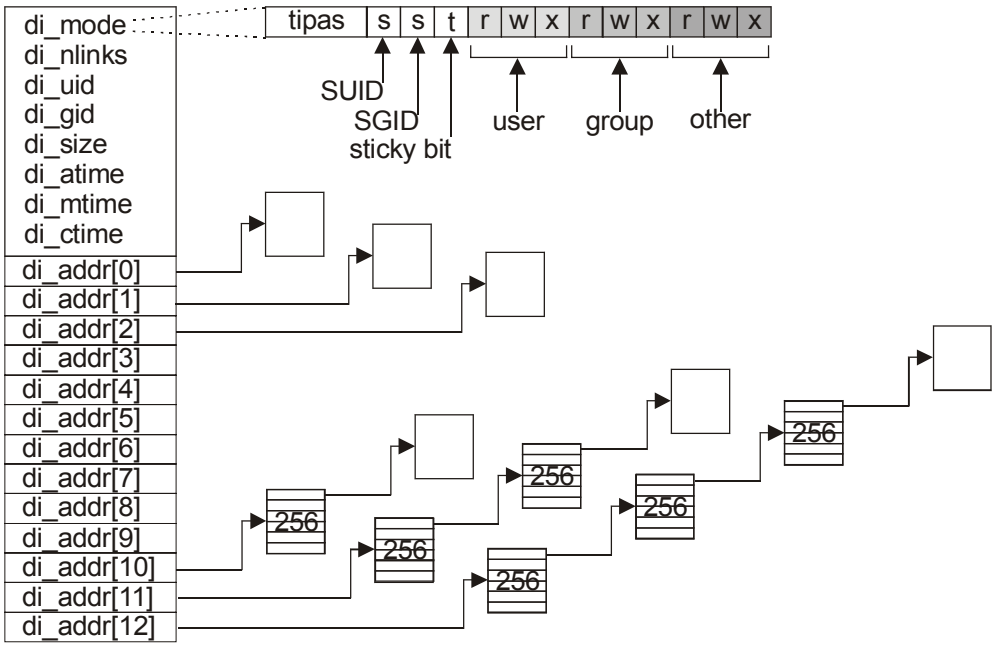
Kiekvienas inode saugo informaciją apie failą, kuri yra būtina failui apdoroti, t.y. failo metaduomenis. Kiekvienam failui yra priskiriamas vienas inode, nors jis gali turėti keletą vardų, kurie rodo į vieną inode.

Indekso deskriptoriuje nėra:

- failo vardo, kuris yra saugomas katalogo duomenų blokuose;
- failo turinio, kuris yra saugomas failo duomenų blokuose.

Atidarydamas failą branduolys patalpina inode kopiją į atmintį – į lentelę in-core inode. Disko indekso deskriptoriaus struktūra yra pateikta 4.3 paveiksle. Pagrindiniai atributai yra šie:

- di_mode failo tipas, papildoma vykdymo informacija ir priėjimo teisės;
- di_nlinks nuorodų į inode skaičius, t.y. failo vardų skaičius;
- di_uid, di_gid savininkas-vartotojas ir grupė-savininkas;
- di_size failo dydis baitais, specialių įrenginių failuose šiuose laukuose yra saugomi mažesnysis ir vyresnysis įrenginio numeriai;
- di_atime paskutinio failo skaitymo laikas;
- di_mtime paskutinio failo modifikavimo laikas;
- di_ctime paskutinės inode modifikacijos laikas, išskyrus laukus di_atime it di_mtime;
- di_addr[13] duomenų blokų adresų masyvas.



4.3 pav. inode struktūra.

Lauke di_mode yra saugoma keletas failo atributų: failo tipas (IFREG – paprastas failas, IFDIR – katalogas, IFZBLK ir IFCHR – blokinis ar simbolinis įrenginys), priėjimo teisės visoms trims vartotojų klasėms ir papildomi vykdymo atributai, tokie kaip SUID, SGID ir sticky bit.

Pastebėsime, kad inode struktūroje nėra saugoma informacija apie failo sukūrimo laiką, tačiau saugomi trys kiti laikai – paskutinio skaitymo, paskutinio modifikavimo ir paskutinės inode modifikacijos laikas.

Indekso deskriptoriuje saugomi visi duomenų blokų adresai, nes failo duomenys gali būti saugomi ne nuosekliuose blokų eilėse, o bet kur. Toks požiūris sukuria papildomus nepatogumus, nes paskutinis blokas dažniausiai būna pustuštis ir su laiku diskas labai defragmentuojasi, t.y. to paties failo duomenų blokai būna išmėtyti įvairiose disko vietose ir jų

skaitymui reikia daugiau laiko. Norint tai sutvarkyti reikia užsaugoti failų sistemą į tarpinį diską ir, po to , atstatyti atgal į vietą. Masyvas yra fiksuoto, 13 elementų dydžio. Pirmieji 10 elementų adresuoja tiesiogiai į failo duomenų blokus, vienuoliktasis elementas adresuoja į bloką, kuriame yra kiti failo blokų adresai. Dvyliktasis nurodo į dvigubos adresacijos bloką, t.y. jis rodo į bloką, kurio kiekvienas elementas rodo į blokus, kurių elementai rodo į tiesioginius failų duomenų blokus. Tryliktasis elementas rodo į trigubos adresacijos bloką.

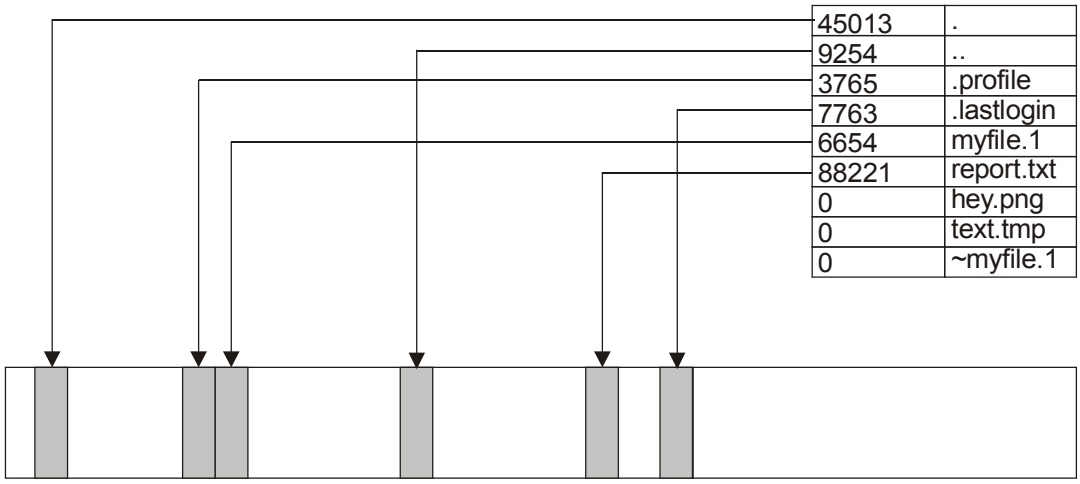
Tokia konstrukcija leidžia efektyviai saugoti mažus ir didelius failus. Tarkim ,jei blokas yra 1024 baitų dydžio, o failas yra pakankamai mažas – iki 10K, tai jam saugoti pakanka tiesioginės adresacijos (pirmųjų dešimties adresų masyvo elementų). Failams, kurių dydis neviršyja 266K (10K + 256 * 1024), pakanka pirmųjų 11 elementų. Pateiksime skaičiavimų lentelę:

- 10 elementų: 1024 * 10 = 10240 = 10K
- 11 elementų: 10240 + 256 * 1024 = 263168 = 266K
- 12 elementų: 263168 + 256 * 256 * 1024 = 67372032 = 65793K = 64M
- 13 elementų: 67372032 + 256 * 256 * 256 * 1024 = 17247241216 = 16843009K = 16448M

UNIX OS failai gali turėti skyles, tarkim procesas sukuria failą, po to su funkcija lseek(2) nustumkime tolyn failo deskriptoriaus rodyklę ir pradėkime rašyti. Tarp failo pradžios ir pirmųjų duomenų atsiranda skylė, t.y. neprirašyta erdvė. Skylei nereikia saugoti duomenų ir tų blokų nuorodos yra lygios nuliui, skaitant jos grąžina 0 – nulius. Jei bandoma rašyti į skylę, sukuriamas blokas ir nuoroda į jį. Skylės failuose gali sukelti nepageidaujamų padarinių, nes kopijuojant failą gali labai išsipūsti failo dydis, t.y. jam priskiriami blokai, kurie yra užpildomi nuliais.

4.5.3 Failų vardai

Kaip jau pastebėjote, nei metaduomenys, nei duomenų blokai nesaugo failo vardo. Failo pavadinimas yra saugomas specialiuose failuose – kataloguose. Tokia konstrukcija leidžia faktiniams duomenims – failui, turėti kelis vardus. Tokiu būdu keli failo pavadinimai rodys į tuos pačius metaduomenis ir duomenis ir yra vadinami kietomis nuorodomis (ryšiais). s5fs katalogas yra lentelė, kurios kiekvienas elementas yra 16 baitų dydžio, kur 2 baitai rodo į failo deskriptorių (inode), o likę 14 yra skirti failo pavadinimui. Ši konstrukcija uždeda apribojimą inode skaičiui – 65535. Taip pat apribojamas ir failo pavadinimo ilgis – 14 simbolių. Katalogo struktūra pateikiama 4.4 paveiksle. Pirmieji du elementai adresuoja patį katalogą - “.” ir tėvinį katalogą – “..”.



4.4 pav. s5fs katalogo struktūra.

Trinant failą, tarkim su `rm` komanda atitinkamo elemento inode nuoroda priskiriama nuliui. Branduolys paprastai nesunaikina laisvų masyvo elementų, todėl trinant katalogo masyvo dydis nesikeičia. Tai tampa problema kataloguose, kuriuose yra buvę daug failų, o vėliau jie buvo ištrinti.

Šešioliktainį neinterpretuotą failo dump'ą išveda komanda hd(1M), kuriame atsispindi ir ištrinti failai. Jų komanda `ls` nebeišveda.

4.5.4 Trūkumai ir apribojimai

s5fs yra patraukli savo paprastumu, tačiau yra nepatikima ir lėta.

Patikimumo požiūriu silpna vieta yra superblokas. Jame yra pagrindinė informacija apie visą failų sistemą. Sugedus superblokui visa failų sistema tampa nebetinkama naudojimui. Šioje failų sistemoje superblokas yra saugomas viena kopija, todėl klaidų tikimybė yra pakankamai didelė.

Darbo greičio problemos kyla tiesiogiai dirbant su failais: metaduomenys yra saugomi disko pradžioje, o duomenų blokai yra išmėtyti visame diske. Dirbant su failu reikia pastoviai kreiptis tai į metaduomenis, tai į duomenis. Tokiu būdu yra šokinėjama per visą diską. Šis efektas yra vadinamas disko fragmentacija.

Taip pat nėra optimaliai naudojama disko erdvė. Greičio atžvilgiu yra patogiau saugoti failus didesniuose blokuose – pagreitinamas skaitymas. Tačiau tokiu būdu yra eikvojama disko erdvė. Paprastai yra prarandama po pusę bloko kiekvienam failui. SVR2 failų sistemos blokas yra 512, o SVR3 – 1024 baitų dydžio.

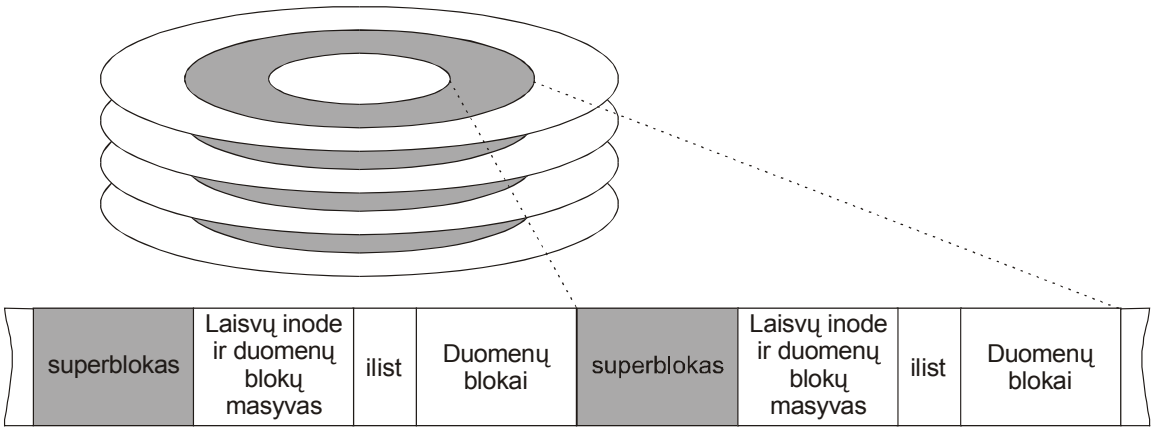
inode masyvas yra fiksuotas ir tai apriboja failų skaičių, kurie gali egzistuoti duotoje failų sistemoje. inode ir duomenų blokų riba gali būti nustatyta neoptimaliai: paprastai yra pritrūkstama failų skaičiaus arba laisvų duomenų blokų. Šios ribos dinamiškai keisti negalima.

Taip pat yra pernelyg kieti apribojimai failo pavadinimui (14 simbolių) ir inode skaičiui (65535).

4.6 BSD UNIX failų sistema

4.3BSD OS buvo padaryti dideli pakeitimai failų sistemose, kurie padidino failų sistemos darbo greitį ir patikimumą. Nauja failų sistema buvo pavadinta Berkeley Fast File System (FFS).

FFS išlaikė pilną s5fs funkcionalumą ir naudoja tas pačias branduolio lygmens duomenų struktūras. Pagrindiniai pakeitimai buvo atlikti failų išdėstymui diske, diskinėse duomenų struktūrose ir blokų užpildymo duomenimis algoritmuose.



4.5 pav. Cilindrų grupės duomenų struktūra FFS failų sistemoje.

Kaip ir s5fs, taip ir FFS yra naudojamas superblokas, kuriame yra saugomi bendrieji duomenys apie failų sistemą. Tačiau jame nėra informacijos apie laisvus inode ir blokus, todėl

superblokas FFS yra nekintamas visu failų sistemos egzistavimo metu. Turima omenyje, kad superblokas yra labai svarbus ir jis yra dubliuojamas.

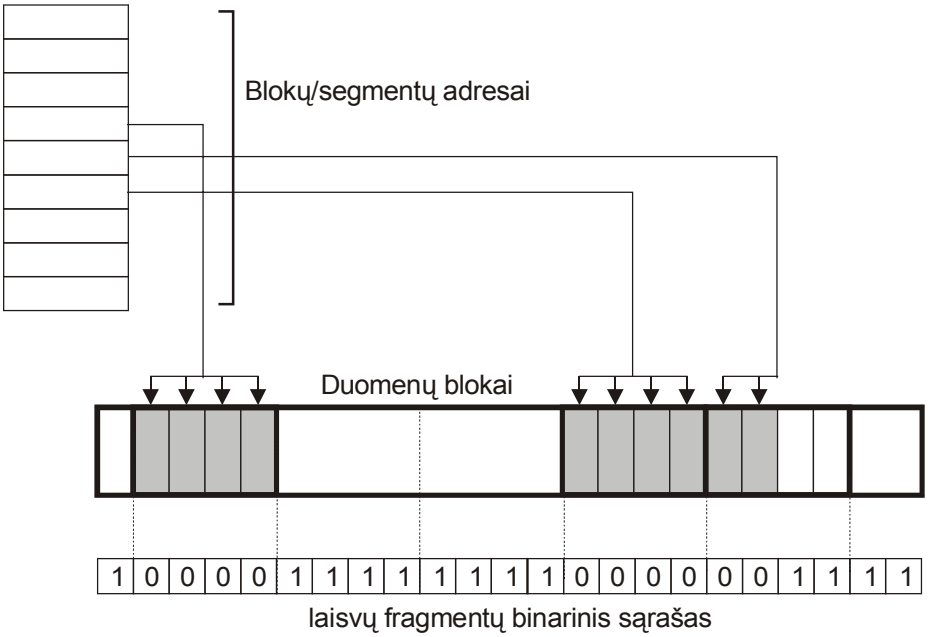
Loginis disko skyrius yra skaidomas į vieną arba kelias cilindrų grupes (cylinder groups). Cilindrų grupe yra keli nuoseklūs disko cilindrai. Kiekvienai grupei yra sukuriama valdymo informacija, t. y. superbloko kopija, inode masyvas, duomenys apie laisvus duomenų blokus ir pan. (žr. 4.5 paveikslą).

Sukuriant failų sistemą kiekvienai cilindrų grupei yra sukuriamas tam tikras inode kiekis. Paprastai kiekvienam 2K disko vietos yra sukuriamas vienas inode. Kadangi inode kiekis yra nurodomas failų sistemos kūrimo metu, tai inode kiekis tampa fiksuotas.

FFS sistemose stengiamasi nesurašyti visų inode į disko pradžią, o sudaryti inode klasterius, kurie būtų nuosekliai paskirstyti po visą diską. Taip sutrumpėja duomenų skaitymo laikas, nes failo metaduomenys ir duomenų blokai yra arti vienas kitas, nereikia galvutei pernelyg šokinėti. Tuo pačiu padidėja failų sistemos patikimumas, nes sumažėja tikimybė, kad bus prarasti visi failų sistemos inode. Tarkim išėjus iš rikiuotės vienam sektoriui yra prarandami tik tame sektoriuje buvę inode ir duomenys.

Kaip minėjome, failų sistemos greitis didžiąja dalimi priklauso nuo blokų dydžio: kuo didesnis blokas, tuo daugiau duomenų galima perskaityti neatliekant paieškos ir disko galvutės perkėlimo. FFS palaiko iki 64K dydžio blokus. Tačiau tokiu atveju kyla kita problema: UNIX OS didelė dalis failų yra nedideli ir saugant juos dideliuose blokuose yra iššvaistoma apie 60% naudingos disko erdvės.

Šie trūkumai buvo įveikti panaudojant bloko fragmentaciją, t. y. kiekvienas blokas gali būti perskeltas į 2, 4 arba 8 fragmentus, nors blokas lieka I/O operacijos duomenų perdavimo vienetas, o fragmentas tampa minimaliu adresuojamu duomenų vienetu. Taip buvo surastas kompromisas tarp duomenų saugojimo optimalumo ir jų perdavimo greičio. Fragmento dydis yra nustatomas failų sistemos sukūrimo metu ir jo maksimali reikšmė yra 0.5 bloko dydžio, o minimali apsprendžiama disko fizinėmis charakteristikomis, t. y. minimaliu disko adresavimo vienetu, sektoriumi.



4.6 pav. FFS failų sistemos duomenų blokai ir fragmentai.

Informacija apie laisvą erdvę cilindrų grupėje yra saugoma laisvų blokų pavidalu, o binariniame blokų fragmentų sąraše. Šis sąrašas yra susietas su konkrečia cilindrų grupe ir aprašo laisvą erdvę fragmentais (žr. 4.6 paveikslą).

Žymūs patobulinimai palietė laisvų blokų užpildymo ir inode išdėstymo algoritmus, kurie savo ruožtu lemia failų išdėstymą diske. s5fs yra naudojamas labai paprastas algoritmas, kuris tiesiog paima laisvą bloką arba inode iš sąrašo pabaigos. Tai galų gale sukelia fragmentacijos problemą. FFS naudoja sekančius failų sistemos efektyvumą padidinančius principus:

- failo duomenys pagal galimybę yra patalpinami vienoje cilindrų grupėje kartu su metaduomenimis. Tai lemia darbo greitį, nes dauguma operacijų reikalauja darbo su metaduomenimis ir duomenimis;
- visi katalogo failai, pagal galimybę yra patalpinami vienoje cilindrų grupėje. Kai kurios komandos dirba tiek su visai katalogo failais;
- kiekvienas naujas katalogas pagal galimybę yra patalpinamas į kitą, nei tėvinis katalogas cilindrų grupę. Taip užtikrinamas nuoseklus duomenų išdėstymas diske;
- failo duomenų blokai yra patalpinami remiantis fizinėmis disko charakteristikomis. Idėja remiasi tuo, kad egzistuoja tam tikras laiko tarpas tarp vieno bloko skaitymo pabaigos ir kito bloko skaitymo pradžios. Per tą laiką diskas spėja pasisukti tam tikru kampu ir sekantis blokas optimaliausiai turėtų būti patalpintas toje vietoje, t. y. su tam tikru poslinkiu nuo prieš tai buvusio. Tokiu atveju vieno failo skaitymui nereikia daryti tuščių disko apsisukimų.

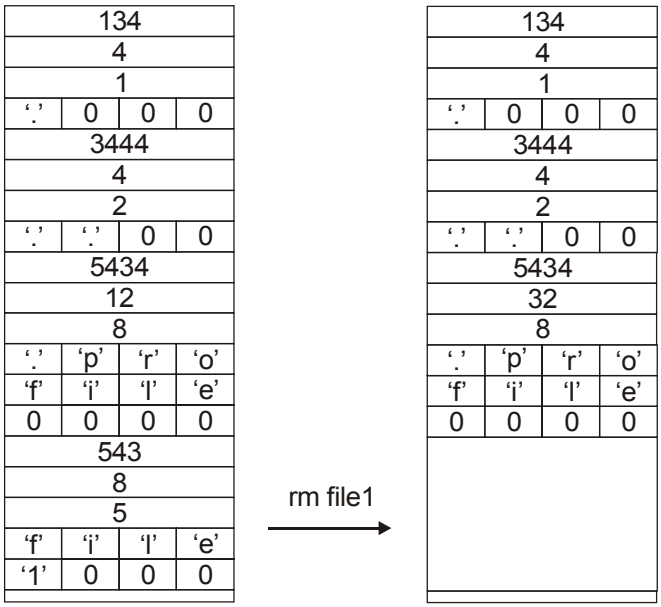
Šis mechanizmas iš vienos pusės sutrumpina duomenų skaitymo laiką, o iš kitos pusės padidina duomenų patikimumą nuosekliai išdėstydamas duomenis visame diske. Nuo šių koncepcijų protingo balanso priklauso bendras failų sistemos funkcionalumas.

Tačiau, kai diske labai sumažėja vietos, krenta sistemos darbo greitis, nes algoritmas neranda pakankamai vietos optimaliam duomenų išdėstymui. Praktika rodo, kad FFS optimizacija yra efektyvi, kai diske yra ne mažiau kaip 10% laisvos vietos.

4.6.1 Katalogai

Katalogo struktūra buvo pakeista siekiant išplėsti failo vardą iki 255 simbolių. Vietoj fiksuoto dydžio įrašų, FFS katalogas susideda iš sekančių įrašų lentelės:

- `d_ino` inode numeris ilist sąraše
- `d_reclen` įrašo ilgis
- `d_namlen` failo vardo ilgis
- `d_name[]` failo vardas

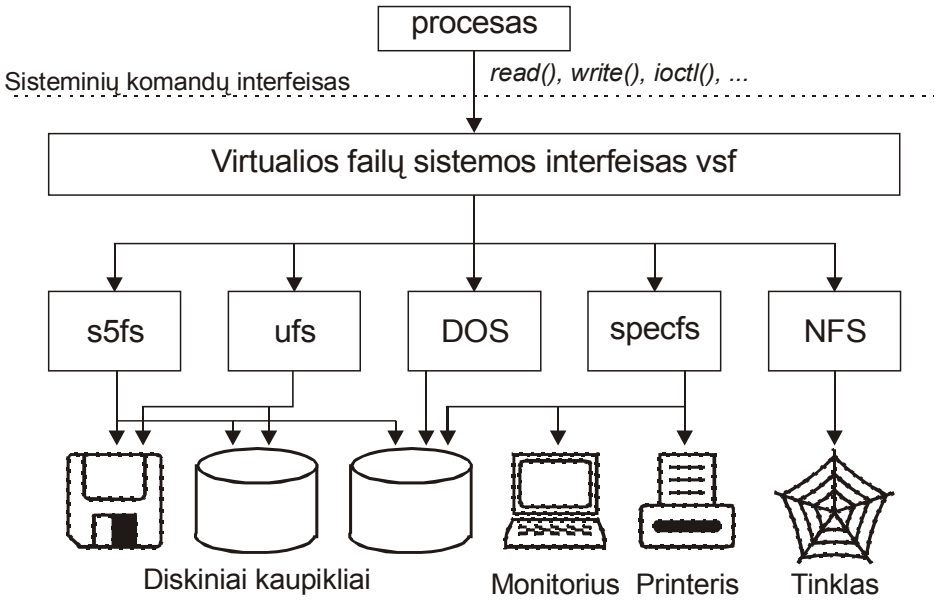


4.7 pav. Katalogo strukūra ir failo trynimo operacija.

Failo vardas yra kintamo dydžio ir yra užpildomas nuliais iki 4 baitų. Ištrinant failą, jam priklausanti informacija yra priskiriama prieš tai einančiam failui ir pakeičiamas jo įrašo dydis d_reclen reikšmė. Jei trinamas pirmasis įrašas, jo d_ino reikšmė prilyginama nuliui. Katalogo struktūra ir trynimas yra pateiktas 4.7 paveiksle.

4.7 Virtualios failų sistemos architektūra

Kaip jau žinote, įvairių tipų failų sistemos gana žymiai skiriasi savo vidine sandara ir darbo organizacija, tačiau nežiūrint to, šiuolaikinės UNIX sistemos leidžia vienu metu dirbti su keliomis jų. Tarp vienu metu naudojamų failų sistemų galima išskirti lokalias failų sistemas, nutolusias failų sistemas ir labai skirtingas nuo UNIX sistemų, pvz. DOS. Tai tampa įmanomu suskirstant failų sistemas į priklausomą nuo realizacijos ir nepriklausomą. Pastaroji yra bendra visoms failų sistemoms ir yra kitoms branduolio posistemėms kaip viena abstrakti failų sistema, dar vadinama virtuali failų sistema (4.8 pav.). Realios failų sistemos yra prijungiamos tokiu pat principu, kaip ir aparatūriniai įrenginiai – naudojant tvarkykles.



4.8 pav. Virtualios failų sistemos architektūra.

4.7.1 Virtualūs indeksų deskriptoriai

Paprastai failas diske turi tam tikrą su juo susietą duomenų struktūrą, vadinamą metaduomenys arba inode, kur yra saugomi pagrindinės failo charakteristikos, kurias naudojant yra įmanomas priėjimas prie failo duomenų. Kaip to išimtis yra failų sistema DOS, kuriose naudojamos visai kitokios duomenų struktūros, tačiau virtualiose failų sistemose jiems yra sukuriamos standartinės duomenų struktūros, o priėjimas valdomas per tvarkykles. Darbo su failu interfeisu virtualioje failų sistemoje yra vnode (virtual inode).

Šis interfeisas buvo sukurtas Sun Microsystems kompanijoje 1984 metais siekiant unifikuoti darbą su failais, esančiais ufs (FFS) ir NFS. Dabar ši virtuali failų sistema yra SVR4 UNIX OS standartu., nors dauguma kitų UNIX OS realizuoja panašias sistemas.

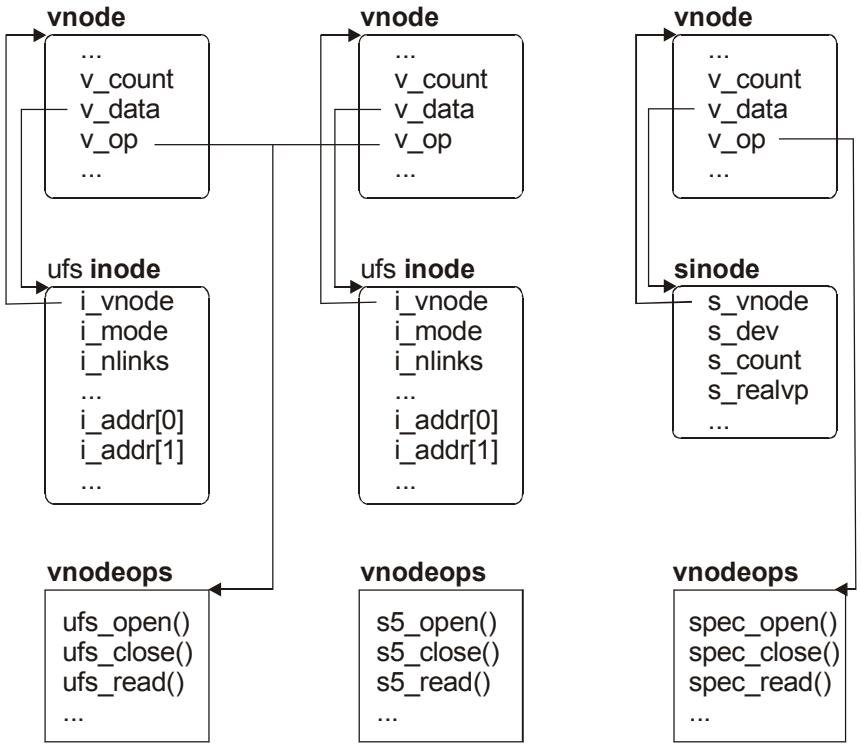
Visų aktyvių failų (failų, su kuriais yra susijęs bent vienas procesas) metaduomenys yra saugomi pagrindinėje atmintyje ir yra vadinami in-core inode. Tai yra vnode analogai. vnode struktūra yra vienoda nepriklausomai nuo realios failų sistemos. Joje yra saugoma informacija, būtina virtualios failų sistemos funkcionalumui užtikrinti ir nekintamos failo charakteristikos. Štai pagrindiniai vnode laukai:

u_short vflag	vnode požymiai
u_short v_count	nuorodų į vnode skaičius
struct filock *v_filock	failo blokavimai
struct vfs *v_vfsmountedhere	nuoroda į prijungtą failų sistemą, jei pastarasis failas yra failų sistemos montavimo taškas
struct vfs *v_vfsp	nuoroda į failų sistemą, kurioje yra duotasis failas
enum vtype v_type	vnode tipas
caddr_t v_data	nuoroda į duomenis realioje failų sistemoje
struct vnodeopts *v_op	vnode operacijos

Kiekvienas vnode turi v_count lauką, kurio reikšmė padidinama, jei failas atidaromas ir sumažinama, jei uždaromas. Jei jo reikšmė pasiekia 0 yra iškviečiama vn_inactive() operacija, kuri praneša realiai failų sistemai, kad į duotą inode daugiau nieks nesikreipia. Po to failų sistema gali atlaisvinti vnode arba patalpinti jį į buferį tolimesniam naudojimui.

Lauke v_vfsp yra saugoma nuoroda į failų sistemą (struktūrą vfs), kurioje yra duotasis failas, adresuojamas šiuo vnode. Jei duotasis failas yra kitos failų sistemos prijungimo taškas, tai laukas v_vfsmountedhere rodo į prijungtą failų sistemą ir ji perdengia failo duomenis.

Laukas v_data rodo į realų failo inode.



4.9 pav. Virtualios failų sistemos failo metaduomenys.

Operacijos su duotuoju vnode yra saugomos lauke v_op. Remiantis objektinio programavimo terminais šis rinkinys yra virtualūs vnode klasės metodai. Jos suteikia priėjimo prie realios failų sistemos funkcijų interfeisą, tokiu būdu yra sukuriamas unifikuotas operacijų interfeisas (rinkinys), o operacijas atlieka specifinės funkcijos. Štai kai kurios operacijos:

- int (*vn_open)() atidaromas failas. Taip pat galimas failo klonavimas;
- int (*vn_close)() failas uždaromas,
- int (*vn_read)() skaitomi failo duomenys,
- int (*vn_write)() duomenys rašomi į failą,
- int (*vn_getaddr)() gražinami failo atributai,
- int (*vn_setaddr)() nustatomi failo atributai,
- int (*vn_access)() patikrinamos teisės į vnode nurodomą failą,
- int (*vn_lookup)() transliuojamas failo vardas į vnode,
- int (*vn_create)() sukuriamas naujas failas ir atitinkamas vnode,
- int (*vn_remove)() pašalinamas failo vardas iš atitinkamo vnode rodomo katalogo,
- int (*vn_link)() sukurti naują failo vardą (kietą nuorodą),
- int (*vn_mkdir)() sukurti naują katalogą ir atitinkamą vnode,

- int (*vn_rmdir)() pašalinamas katalogas,
- int (*vn_readdir)() perskaitomas katalogo turinys,
- int (*vn_symlink)() sukuriama minkšta nuoroda konkretų failą,
- int (*vn_readlink)() failo, simbolinės nuorodos, skaitymas,
- int (*vn_inactive)() leidimas sunaikinti vnode.

Ryšys tarp nepriklausomų deskriptorių – vnode ir priklausomų nuo realizacijos failo metaduomenų parodytas 4.9 paveiksle.

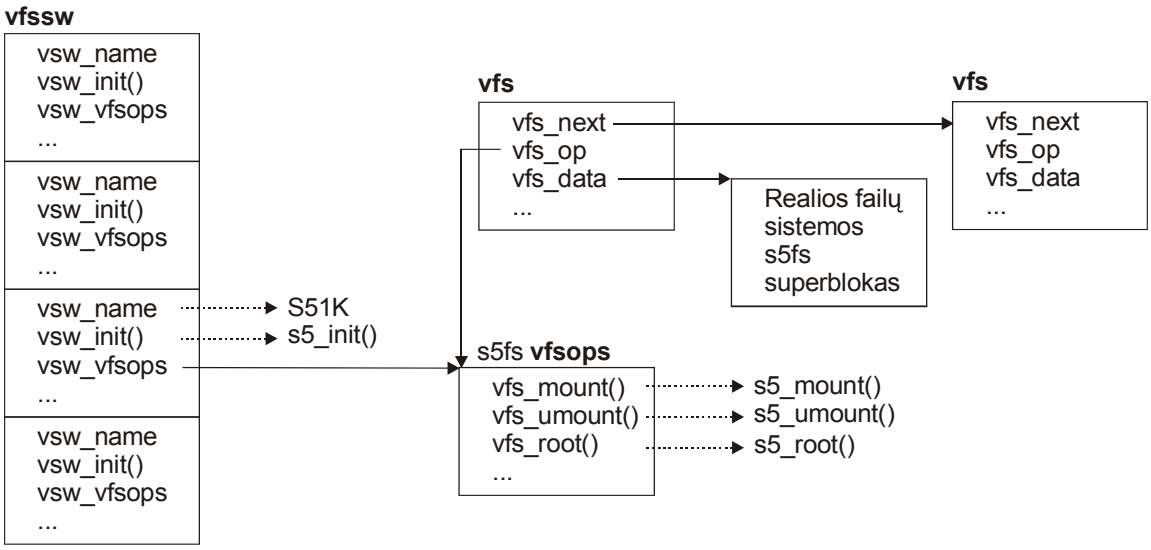
4.7.2 Failų sistemos prijungimas (montavimas)

Prieš pradedant darbą su failų sistema, ją reikia prijungti, vadinama primontuoti prie vieno hierarchinio medžio.

Kiekvienai primontuotai failų sistemai nepriklausomame lygyje yra saugoma vfs struktūra. Jos yra susietos į vienos krypties sąrašą, kuriame yra saugoma tiek bendra virtualios failų sistemos informacija, tiek duomenys apie realias failų sistemas. Pirmasis sąrašo elementas yra šakninė failų sistema. Failų sistemų sąrašas yra vadinamas montavimo lentele. Pateiksime vfs struktūros elementus:

`struct vfs`	`*vfs_next`	sekanti failų sistema montavimo lentelėje
`struct vfsops`	`*vfs_op`	failų sistemos operacijos
`struct vnode`	`*vfs_vnodecovered`	vnode, kurį perdengia duotoji failų sistema
`int`	`vfs_flags`	požymiai: tik skaitymui ir pan.
`int`	`vfs_bsize`	failų sistemos bloko dydis
`caddr_t`	`vfs_data`	nuoroda į failų sistemos specifinius duomenis

Laukas vfs_data rodo į realios failų sistemos duomenis, tokius kaip s5fs superblokas, reziduojantis atmintyje.



4.10 pav. Virtualios failų sistemos duomenų

vfs_op saugo failų sistemos operacijų aibę, t. y. virtualius metodus, kuriuos perdengia realios konkrečios failų sistemos funkcijos. Kadangi šios funkcijos tiesiogiai priklauso nuo konkrečios failų sistemos architektūros, tai montavimo metu jos tampa nuorodomis į reikalingas funkcijas. Štai keletas galimų failų sistemų funkcijų:

`int (*vfs_mount)()`	Primontuoja failų sistemą. Paprastai ši funkcija pakrauna superbloką į atmintį ir sukuria įrašą montavimo lentelėje.

`int`	`(*vfs_unmount)()`	Numontuoja failų sistemą; tuo pačiu atlieka failų sistemos duomenų aktualizaciją
`int`	`(*vfs_root)()`	Grąžina šakninį failų sistemos katalogą.
`int`	`(*vfs_statfs)()`	Grąžina bendrą informaciją apie failų sistemą, pvz. bloko dydį, blokų skaičių, laisvų blokų kiekį, inode skaičių ir pan.
`int`	`(*vfs_sync)()`	Aktualizuoja visus kešuojamus failų sistemos duomenis.
`int`	`(*vfs_fid)()`	Grąžina failo identifikatorių fid (file identifier), kuris vienareikšmiškai identifikuoja failą duotoje failų sistemoje, pvz. inode.
`int`	`(*vfs_vget)()`	Grąžina nuorodą į duotos failų sistemos failo, pateikto fid pavidalu, vnode.

Realios failų sistemos inicializacijai ir montavimui UNIX OS yra naudojami failų sistemų komutatoriai (File System Switch), kurie yra visų OS palaikomų failų sistemų procedūrinis interfeisas, kuriuos palaiko branduolys. UNIX System V yra naudojama globali lentelė, kurios kiekvienas elementas atitinka vieną realų failų sistemos tipą, pvz. s5fs, ufs ir nfs. Šios lentelės elementas vfssw turi sekančius laukus:

`char`	`*vsw_name`	Failų sistemos tipo pavadinimas
`int`	`(*vsw_init)()`	Inicializacijos procedūros adresas
`struct vfsops`	`*vsw_vfsops`	Nuoroda į failų sistemos operacijų vektorių
`long`	`vws_flag`	Požymiai

Virtualios failų sistemos struktūrų sąsajas iliustruoja 4.10 paveikslas.

Failų sistemos prijungimas yra atliekamas iškvietus sisteminę komandą mount(2). Argumentais yra perduodami: failų sistemos tipas, katalogo, prie kurio yra jungiama failų sistema, pavadinimas (montavimo taškas), įvairūs požymiai ir papildoma informacija, kuri priklauso nuo montuojamos failų sistemos tipo. Procedūra pradžioje ieško montavimo taško vnode (naudojamos vardų transliavimo operacijos lookup() arba namei()) ir tikrinama, ar duotasis vnode yra katalogas ir ar nėra naudojamas kitų failų sistemų prijungimui.

Tada yra ieškomas duoto failų sistemos tipo komutatorius – vfssw[]. Jei toks elementas randamas, iškviečiama inicializacijos operacija adresuojama vsw_init(). Jos metu yra sukuriamos specifinės šiam failų sistemos tipui skirtos duomenų struktūros, o po to ir vfs struktūra, kuri yra patalpinama į prijungtų failų sistemų sąrašą (4.10 pav.). Laukas vfs_vnodecovered rodo į montavimo taško vnode. Šakninėje failų sistemoje šis laukas yra lygus 0, o failų sistema yra pirmoji sąraše. Laukas vfs_op rodo į šio failų sistemos tipo operacijų vektorių. Nuoroda į naujai sukurtą vfs elementą yra išsaugoma montavimo taško virtualaus deskriptoriaus vnode lauke v_vfsmountedhere.

Po to yra iškviečiama operacija vfs_mount(), atitinkanti duotą failų sistemos tipą. Realūs veiksmai priklauso nuo failų sistemos, tarkim ufs atveju bus nuskaitomas superblokas, o nfs – kviečiamas failų serveris. Nežiūrint to, yra atliekamos tam tikros tipinės operacijos:

tikrinama ar yra atitinkamos teisės, leidžiančios atlikti failų sistemos montavimą;

pakraunami ir inicijuojami specifiniai duomenys, į kuriuos nuoroda yra saugoma vfs elemente vfs_data;

patalpinamas šakninio katalogo vnode ir užsaugoma nuoroda į jį. Priėjimas prie šakninio katalogo yra vykdomas iškvietus komandą vfs_root().

Vieną kartą prijungus failų sistemą, į ją galima kreiptis naudojantis montavimo taško vardu, t.y. katalogo, prie kurio ji yra prijungta pavadinimu. Tarkim, norint atjungti t.t. failų sistemą, sisteminei komandai unmount(2) yra pateikiamas montavimo taško pavadinimas. Tokiu būdu yra sukuriamas vieningas priėjimo prie įvairių failų sistemų interfeisas.

4.7.3 Vardų transliavimas

Daugelis vartotojo programų kaip argumentą naudoja failų vardus, kaip tuo tarpu branduolys, dirbdamas su failais naudoja indeksinius deskriptorius. Tokiu būdu reikia atlikti failų vardų transliaciją.

Formaliai failo vardas yra "/" simboliu atskirta žodžių seka. Kiekvienas vardo komponentas, išskyrus paskutinįjį yra katalogo pavadinimas. Kiekvienas pilnas vardas gali būti absoliutus arba santykinis. Jei failo pavadinimas prasideda "/" simboliu, kuris žymi visos failų sistemos šaknį (pradžią), tai šis pavadinimas reiškia absoliutų failo vardą, kuris vienareikšmiškai identifikuoja failą virtualioje failų sistemoje. Kitu atveju failo vardas yra santykinis ir adresuoja failą nuo einamojo katalogo. Tarkim, include/sys/user.h yra santykinis, o /usr/include/sys/user.h – absoliutus failo user.h pavadinimas. Tokiu būdu, vardo transliacijai esminę įtaką turi du katalogai: šakninis ir einamasis. Kiekvienas procesas saugo nuorodas į juos u-area struktūroje:

`struct vnode *u_cdir`	Nuoroda į einamojo katalogo vnode
`struct vnode *u_rdir`	Nuoroda į šakninio katalogo vnode

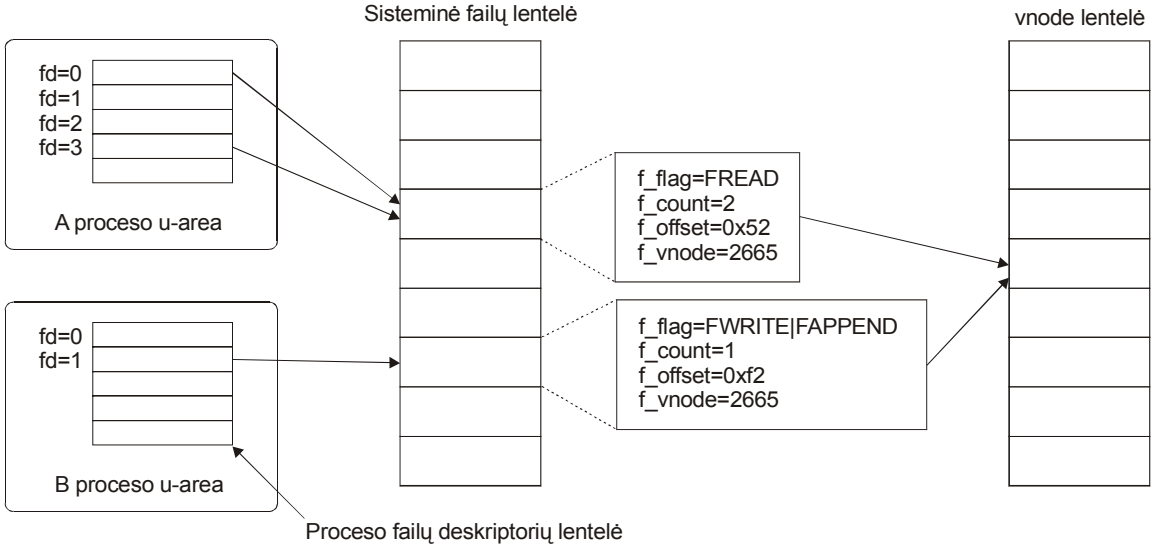
Priklausomai nuo pateikto failo vardo transliacija pradedama šakniniu arba einamuoju katalogu. Šis procesas vykdomas kiekvienam vardo komponentui atskirai: procedūrai vn_lookup() yra pateikiamas einamasis vnode ir vardo komponentas, o ji grąžina kitą vnode ir t. t. Jei kelias kerta katalogą, prie kurio yra primontuota failų sistema, tai procedūra naudodamasi nuoroda vn_mountedhere pasiekia vfs struktūrą, iškviečia jos vfs_root() funkciją ir, gavusi šakninį vnode, tęsia darbą nuo jo. Jei failų sistemos montavimo taškas yra sutinkamas einant medžiu aukštyn, tai tiesiog yra sutinkamas šakninis vnode (v_flag yra požymis VROOT) ir operacija seka nuorodą vfs_vnodecovered einamojoje failų sistemoje.

Jei ieškomasis katalogas yra simbolinė nuoroda, tai yra iškviečiama komanda vn_readlink(), kuri grąžina tikrojo failo pavadinimą.

Procesas yra tęsiamas visiems failo pavadinimo komponentams arba kol yra surandama klaida, tarkim proceso savininkas neturi teisės peržiūrėti sekančio katalogo (vykdymo teisė).

4.8 Failų sistemos naudojimas

Kaip jau žinote, procesas naudoja savo lokalią failų deskriptorių numeraciją, t. y. sveikus skaičius, kurie neturi nieko bendro su kito proceso ar tikrais failų sistemos deskriptoriais. Procesas, iškviečia sisteminę komandą open(2) arba creat(2), kuri atlieka vardo transliaciją ir grąžina failo deskriptorių, kuris adresuoja į realaus failo inode (arba vnode). Paveiksle 4.11 yra pateiktos pagrindinės branduolio duomenų struktūros, skirtos failų naudojimui.



4.11 pav. Vidinės branduolio struktūros skirtos priėjimui prie failo.

Proceso failo deskriptorius yra ne kas kitas, o procesų failo deskriptorių lentelės (file descriptor table) indeksas. Kiekvienas procesas turi savo lentelę, kuri yra saugoma jo u-area struktūroje. Kiekvienas šios lentelės įrašas rodo į atidarytą failą, t.y. adresuoja sisteminės failų lentelės (system file table) įrašą. Jame yra saugomi tokie duomenys, kaip atidaryto failo režimas (skaitymui, rašymui, papildymui ir pan.), einamoji failo žymeklio padėtis, nuoroda į failo vnode ir pan. Sisteminė failų lentelė yra viena visiems procesams. Keli šios lentelės įrašai gali rodyti į vieną ir tą patį failą, kuris yra išreikštas vienu vnode.

4.8.1 Failų deskriptoriai

Failų deskriptorius yra teigiamas sveikas skaičius, kuris yra grąžinamas iškvietus sistemines komandas open(2), creat(2) arba pipe(2). Toliau procesas gali jį naudoti darbui su failu tokiose komandose kaip read(2), write(2) ir pan.

Branduolys palaiko proceso darbą su failais naudodamasis įvairiomis duomenų struktūromis, kurių dalis yra proceso u-area. Deskriptorių lentelėje yra du laukai, kurie betarpiškai yra naudojami darbui su failais:

`u_ofile`	Nuoroda į sisteminę failų lentelę
`u_pfile`	Deskriptoriaus požymiai

Šiuo metu yra naudojamas tik vienas failo deskriptoriaus požymis – FD_CLOEXEC; jei šis požymis yra nustatytas tai iškvietus sisteminę komandą exec(2) šis failas yra uždaromas, t.y. nėra paveldimas.

Senesnės UNIX versijos palaikydavo tik statines failų deskriptorių lenteles, kurios pilnai yra patalpinamos į u-area struktūrą. Deskriptoriaus numeris yra lentelės indeksas. Tokiu būdu lentelės dydis, kuris dažniausiai būna 64, apriboja proceso atidarytų failų skaičių. Paskutinėse UNIX versijose yra palaikomos dinaminės failo deskriptorių lentelės. Taip pat gali būti uždėtas RLIMIT_NOFILE apribojimas. Kai kuriose sistemose, pvz. Solaris 2.5 deskriptoriai yra saugomi ne lentelės, o struktūros sąrašo pavidalu.

4.8.2 Sisteminė failų lentelė

Failų deskriptoriaus laukai – u_ofile ir u_pofile saugo tik pirminę informaciją, kuri yra reikalinga darbui su failu. Papildoma informacija yra laikoma sisteminėje failų lentelėje ir failo indeksiniame deskriptoriuje. Siekdamas suteikti procesui galimybę dirbo su failu interfeisą branduolys sukuria visą duomenų struktūrų grandinę (kaip parodyta 4.11 paveiksle).

Kiekviename sisteminės failų lentelės elemente yra saugoma darbui sus failu būtina informacija. Jei keli procesai atidaro vieną ir tą patį failą, kiekvienas jų gauna savo failų lentelės elementą, nors visi jie dirba su vienu failu. Pateiksime pagrindinius sisteminės failų lentelės pagrindinius elementus:

`f_flag`	Atidaryto failo požymiai, daugiausia teisės į t.t. operacijas su failais: FREAD (skaitymui), FWRITE (rašymui), FAPPEND (papildymui), FNONBLOCK ar FNDELAY (sisteminė komanda nelaukia operacijos pabaigos), FSYNC (duomenys sinchronizuojami su disko meta- ir duomenų struktūromis), FDSYNC (duomenys sinchronizuojami su disko duomenų struktūromis), FRSYNC (kartu su paskutiniais dviem požymiais sinchronizuoja failo duomenis skaitymo metu)
`f_count`	Failo deskriptorių, naudojančių šį sisteminės failų lentelės įrašą skaičius. Vieną įrašą gali naudoti keli deskriptoriai, pvz. po fork() ir exec() sisteminių komandų.
`f_vnode`	Nuoroda į virtualų failo indeksinį deskriptorių.
`f_offset`	Einamasis poslinkis faile. Pradedant nuo šios pozicijos bus atliekama sekanti skaitymo arba rašymo operacija.

4.8.3 Failo blokavimas

Tradiciškai UNIX operacinės sistemos leidžia keliems procesams vienu metu atlikti operacijas su vienu failu. Nors operacijos read(2) ir write(2) yra atomarinės, UNIX sistemose pagal nutylėjimą atliekama sisteminių komandų sinchronizacija. Kitais žodžiais tariant, vienam procesui atliekant skaitymą, o kitam jį modifikuojant, gali kilti failo duomenų vientisumo pažeidimas. Tai yra nepageidaujamas reiškinys.

UNIX OS leidžia t. t. failo baitų diapazoną arba įrašą užblokuoti. Tai atliekama naudojant sisteminę failo valdymo komandą fcntl(2) arba bibliotekos funkcija lockf(2), kuri yra tiesiogiai skirta failo blokavimo valdymui. Tokiu būdu, procesas, prieš atlikdamas operaciją su failu, nustato atitinkamą blokavimą (skaitymo ar rašymo). Jei blokavimas įvyko sėkmingai, tai reiškia, kad sekanti operacija su failu nesukels konflikto.

Pagal nutylėjimą blokavimas yra rekomendacinio lygmens (advisory lock). Tai reiškia, kad kartu dirbantys failai gali naudotis sukurtomis blokuotėmis, bet branduolys nedraudžia vykdyti skaitymo ar rašymo operacijų blokuotoje srityje. Šis blokavimo lygmuo įpareigoja procesą patį pasitikrinti ar resursas, t.y. failas nėra užblokuotas.

Failo blokavimą aprašo struktūra flock:

`short`	`l_type`	Blokavimo tipas: F_RDLCK (read lock), F_WRLCK (write lock), F_UNLCK (unlocked).
`short`	`l_whence`	Poslinkio matavimo pradžios atskaitos taškas: SEEK_CUR (nuo einamosios padėties), SEEK_END (nuo failo pabaigos), SEEK_SET (nuo failo pradžios).
`off_t`	`l_start`	Blokuojamo įrašo poslinkis nuo l_whence.
`off_t`	`l_len`	Blokuojamo intervalo ilgis. Jei lygi 0, tai blokuojama iki failo galo.
`pid_t`	`l_pid`	Proceso, sukūrusio šį blokavimą identifikatorius, grąžinamas naudojant komandą GETLK.

Blokavimas galimas skaitymui ir rašymui. Bet kurio failo baito blokavimo taisyklės yra pateiktos 4.3 lentelėje.

4.3 lentelė. Failo baito blokavimo taisyklės.

Egzistuojantis blokavimas	**Bandymas blokuoti**	
	F_RDLCK	**F_WRLCK**
F_UNLCK	1	1
F_RDLCK	1	0
F_WRLCK	0	0

Programos, naudojančios blokavimą fragmentas:

```
...
struct flock lock;
...
lock.l_type=F_WRLCK;
lock.whence=SEEK_SET;
lock.l_start=0;
lock.len=0;
fcntl(fd, SETLKW, &lock);
...
write(fd, record, sizeof(record));
...
lock.l_type=F_UNLCK;
fcntl(fd, SETLKW, &lock);
```

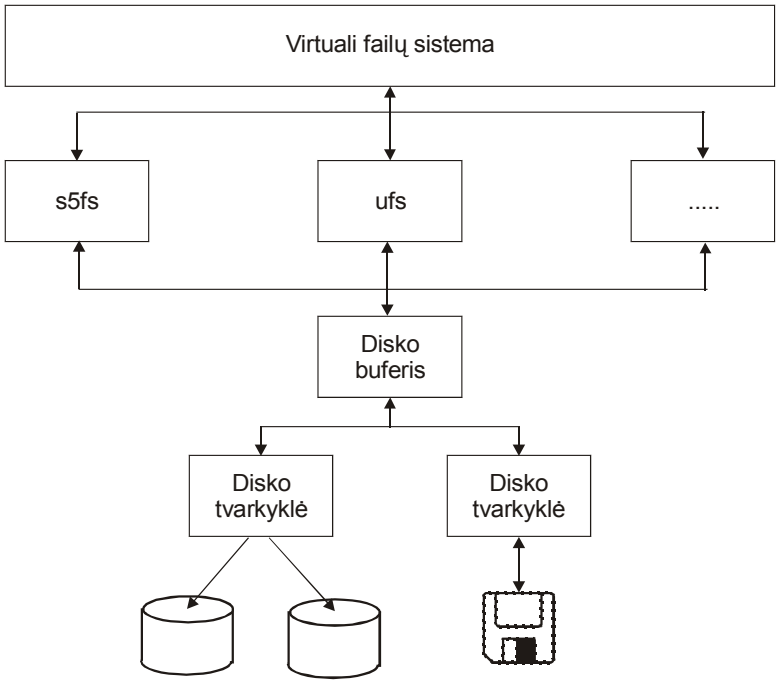
UNIX sistemose yra ir privalomasis blokavimas (mandatory lock), kurio blokavimą kontroliuoja sistemos branduolys. Privalomųjų blokavimo mechanizmų realizacija priklauso nuo realizacijos. Tarkim, SCO UNIX (SVR3) nuėmus failo grupės x bitą ir nustačius SGID visi blokavimai tampa privalomi. UNIX SVR4 yra specialūs blokavimo požymiai, kurie rodo, kad yra užblokuota privalomuoju blokavimu.

Privalomasis blokavimas turi potencialų pavojų, nes, jei procesas užblokuoja kokį nors gyvybiškai svarbų failą ir dėl kokių nors priežasčių miršta, tai nei vienas procesas nebegalės prieiti prie failo.

4.9 Disko buferis

Failų sistemos darbas yra tiesiogiai susijęs su duomenų apsikeitimu su išoriniais įrenginiais. Paprastų failų ir katalogų atveju, tai yra įrenginys, kuriame yra reali failų sistema, t. y. diskinis įrenginys; specialiems įrenginių failams – tai gali būti spausdintuvas, terminalas arba tinklo adapteris. Nesileisdami į I/O detales peržiūrėkime darbo su paprasčiausiu periferiniu įrenginiu – disku – ypatybes.[2]

Diskinės I/O operacijos yra labai lėtos lyginant su operatyvinės ar virš-operatyvinės atminties operacijomis. Duomenų skaitymas iš disko ir iš operatyvinės atminties gali skirtis kelis tūkstančius kartų. Siekiant pagreitinti lėtas diskines operacijas UNIX OS kešuoja diskinės atminties blokus operatyvinėje atmintyje.



4.12 pav. Disko buferis.

Šiam darbui yra skiriama speciali operatyvinės atminties sritis, kurioje yra saugomi dažniausiai naudojami diskiniai failų blokai. Ši atminties sritis ir su ja susijęs procedūrinis interfeisas yra vadinami buferiniu kešu (catche) arba disko buferiu. Įvairių branduolio posistemių sąveika su disko buferiu yra pateikta 4.12 paveiksle.

4.9.1 Vidinės diskinio buferio struktūros

Diskinį buferį sudaro duomenų buferiai, kurių kiekvieno dydis yra pakankamas vieno disko bloko saugojimui. Su kiekvienu duomenų bloku yra susiejama buferio antraštė, struktūra buf, kurios pagalba branduolys valdo konkretų duomenų buferį. Antraštė yra naudojama ir skaitymo/rašymo operacijoms su įrenginio tvarkykle atlikti. Kai kyla būtinybė atlikti skaitymo

[2] Dažnai failai yra ir specialiose failų sistemose, tokiose kaip NFS (Network File System), kuriose nevyksta realių disko įvedimo/išvedimo operacijų. tačiau nežiūrint to, buferio panaudojimas gerokai pagreitina darbą.

arba rašymo operaciją branduolys patalpina operacijos parametrus į duomenų buferio antraštę ir perduoda jo funkcijas įrenginio tvarkyklei. Po operaciją buferyje lieka jos rezultatai.

Pagrindiniai buf struktūros laukai:

`b_flags`	Požymiai. Apibrėžia buferio būseną kiekvienu laiko momentu: B_BUSY (buferis užimtas), B_DONE (operacija baigta), B_READ, B_WRITE ir B_PHYS (duomenų perdavimo kryptis).
`av_forw, av_back`	Dvikrypčio buferio darbinio sąrašo nuorodos. Sąrašas sudaromas laukiant, kol įrenginio tvarkyklė apdoros buferius.
`b_bcount`	Baitų, kuriuos reikia perduoti, skaičius.
`b_un.b_addr`	Buferio virtualus adresas.
`b_blkno`	Duomenų pradžios bloko numeris įrenginyje.
`b_dev`	Vyresnysis ir jaunesnysis įrenginio numeriai.

b_flags lauke yra saugomi įvairūs buferio požymiai. Tarkim, naudojant B_BUSY yra sinchronizuojamas darbas su buferiu. Požymis B_DELWRI parodo, kad buferis yra modifikuotas arba “purvinas”, t. y. prieš sekantį jo panaudojimą buferio turinį reikia išsaugoti disko atmintyje. Požymius B_READ, B_WRITE, B_ASYNC, B_DONE ir B_ERROR naudoja įrenginio tvarkyklė.

Buferis naudoja atidėto įrašymo mechanizmą (write-behind), t. y. modifikavus buferį duomenys nėra tuojau pat įrašomi į diską. Tokie duomenys yra pažymimi kaip “purvini” ir jų sinchronizacija su disko duomenimis yra atliekama praėjus t. t. laiko intervalui, o ne betarpiškai. Paprastai trims duomenų modifikavimo tenka viena įrašymo į diską operacija. Buferis gerokai sumažina darbo su disku operacijų skaičių[3]. Tačiau jis gali sukelti kitų problemų, tokių kaip failų sistemos vientisumo praradimas ir pan.

4.9.2 Įvedimo/išvedimo operacijos

Paveiksle 4.13 yra pateiktos tipinės disko skaitymo ir rašymo operacijos naudojant buferį.

Kai procesui reikia perskaityti iš disko ar įrašyti informaciją į diską jis naudoja sistemines komandas read(2) ir write(2), kurios šią operaciją perduoda failų sistemai. Failų sistema pertransliuoja šią komandą į failo blokų skaitymo operacijas ir perduoda jas disko buferio mechanizmui. Pradžioje šis mechanizmas peržiūri ar operatyvinėje atmintyje nėra reikiamo bloko buferio. Jei toks blokas surastas, tai jis, skaitymo atveju, tiesiog kopijuojamas į proceso adresinę erdvę, o rašymo atveju – įvykdoma atvirkštinė operacija. Jei kešo mechanizmas reikiamo failo bloko neranda, jis paima laisvą (nenaudojamą) buferį, jį suriša su konkrečiu disko bloku, užpildo antraštę reikiamos operacijos požymiais ir pasiunčia užklausą įrenginio (disko) tvarkyklei. Paprastai yra naudojama skaitymo pirmyn taktika (read-ahead), kai yra nuskaitomas ne tik reikalaujamas disko blokas, bet ir sekantys failo blokai, kurių panaudojimo tikimybė yra labai didelė. Tokiu būdu sekanti read(2) operacija greičiausiai nebesukels papildomos disko operacijos, o užteks tik buferio skaitymo. Kai yra vykdoma bloko modifikacija, tai taip pat liečia tik buferį, kurį branduolys pažymi kaip “purviną”. Norint tokį buferį panaudoti kitam disko blokui saugoti, reikia įrašyti buferio turinį į su juo susietą disko bloką.

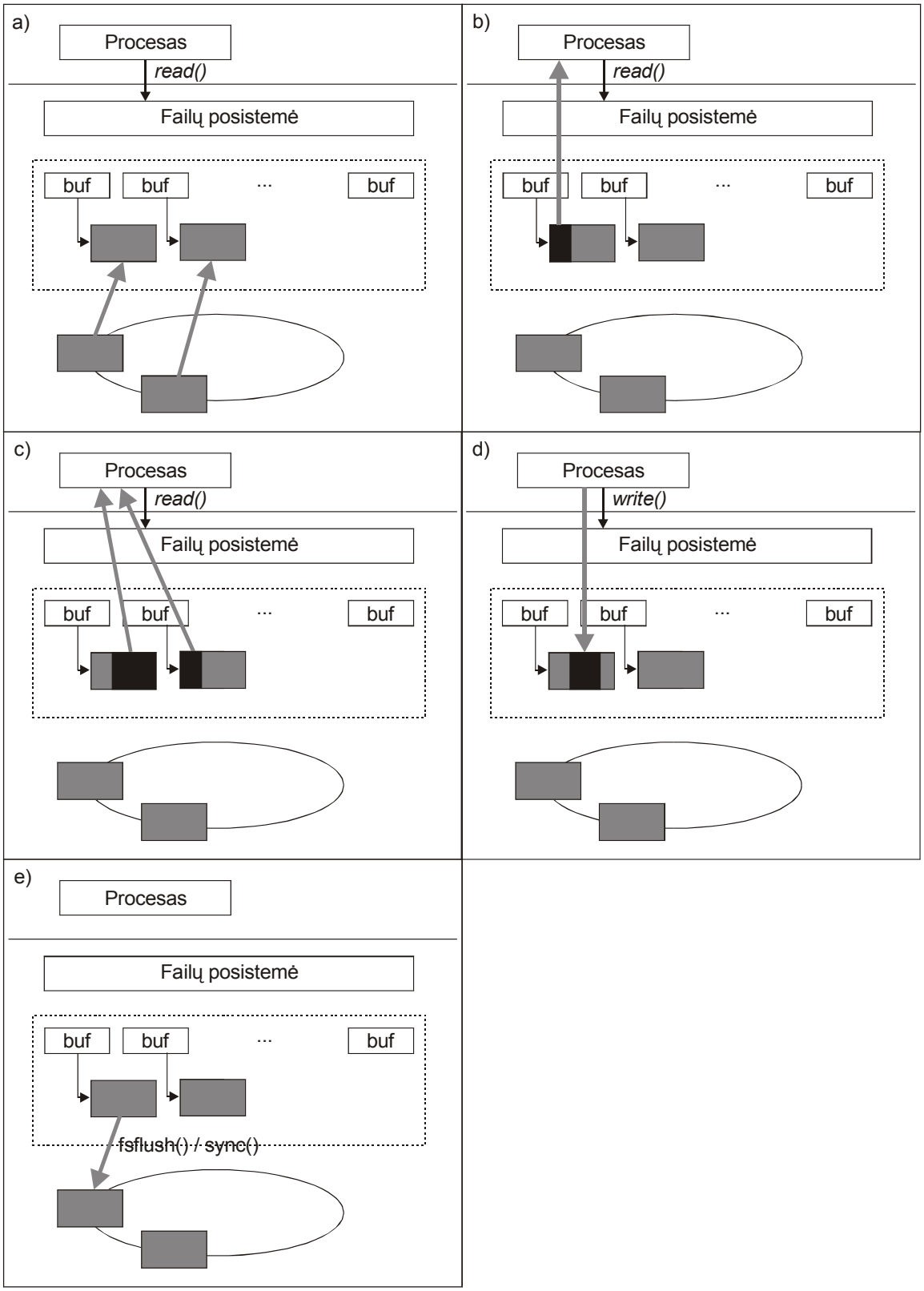
Prieš atliekant operacijas su buferiu, jis yra užblokuojamas, kad kiti procesai nesukeltų problemų. Kreipdamasis į jau užblokuotą buferį procesas yra nusiunčiamas miegoti iki kol buferis taps laisvu.

Neužblokuoti buferiai yra patalpinami į specialų sąrašą. Buferiai šiame sąraše yra išdėstomi pagal tai, kaip retai jie yra naudojami (Least Recently Used, LRU). Tokiu būdu, kai branduoliui prireikia buferio, jis išsirenka tą, kuris pastaruoju metu yra rečiausiai naudojamas. Tada, kai

[3] Tipinėse UNIX sistemose buferis padeda išvengti apie 95% skaitymo iš disko ir apie 85% rašymo į diską operacijų.

buferis tampa nebereikalingas, jis patenka į sąrašo galą, todėl jo sekantis panaudojimas yra mažiausiai įmanomas. Tokiu būdu, jei procesui vėl reikės to bloko, jis vėl dirbs tik su buferiu.

Pagrindine diskinio buferio problema yra informacijos diske senėjimas. Kaip matyti, dauguma informacijos modifikavimo yra atliekama su buferiu, o diskas lieka nuošalyje. Jei sistema dirba normaliai, jokių problemų nekyla; tačiau jai “nulūžus” diske bus likusi sena informacija. Siekiant to išvengti UNIX sistemose yra naudojamos kelios technikos:



4.13 pav. Diskinio buferio darbo schema.

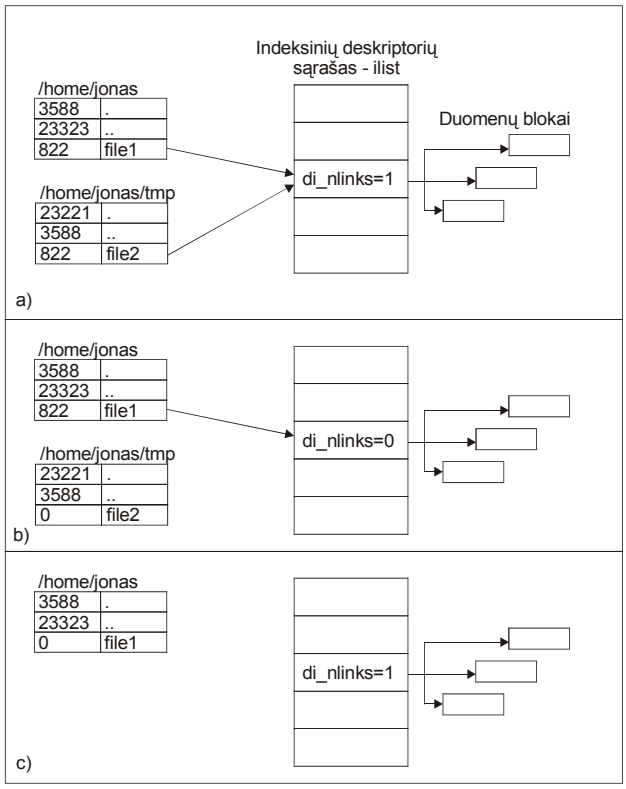
- galima iškviesti sync(2) sisteminė komanda, kuri susinchronizuoja purvinų buferių duomenis su disko blokais. Svarbu paminėti, kad ši funkcija nelaukia, kol bus baigtos

I/O operacijos, todėl nėra garantijos, kad įvykdžius šią komandą buferis nebebus purvinas[4];

- procesas gali atidaryti failą sinchroniniame režime O_SYNC. Tokiu būdu, visos failo modifikacijos bus betarpiškai fiksuojamos diske;
- tam tikrais intervalais UNIX sistemose yra prikeliamas specialus sisteminis procesas – buferinio kešo dispečeris (įvairiose UNIX sistemose jis yra vadinamas įvairiai – fsflush arba bdflush), kuris užsaugo "purvinų" buferių informaciją į diską ir pažymi buferį švariu.

4.10 Failų sistemos vientisumas

Nemaža dalis failų sistemos pastoviai reziduoja operatyvinėje atmintyje, tarkim, superblokas, aktyvių failų metaduomenys ir kai kurie failo duomenų blokai buferyje.



4.14 pav. Failų sistemos vientisumo praradimas.

Jei staigiai bus nutraukiamas operacinės sistemos darbas, nepakeisti buferio duomenys nesukels katastrofinių padarinių, nes failų turinys neturi didelės įtakos failų sistemos vientisumui.

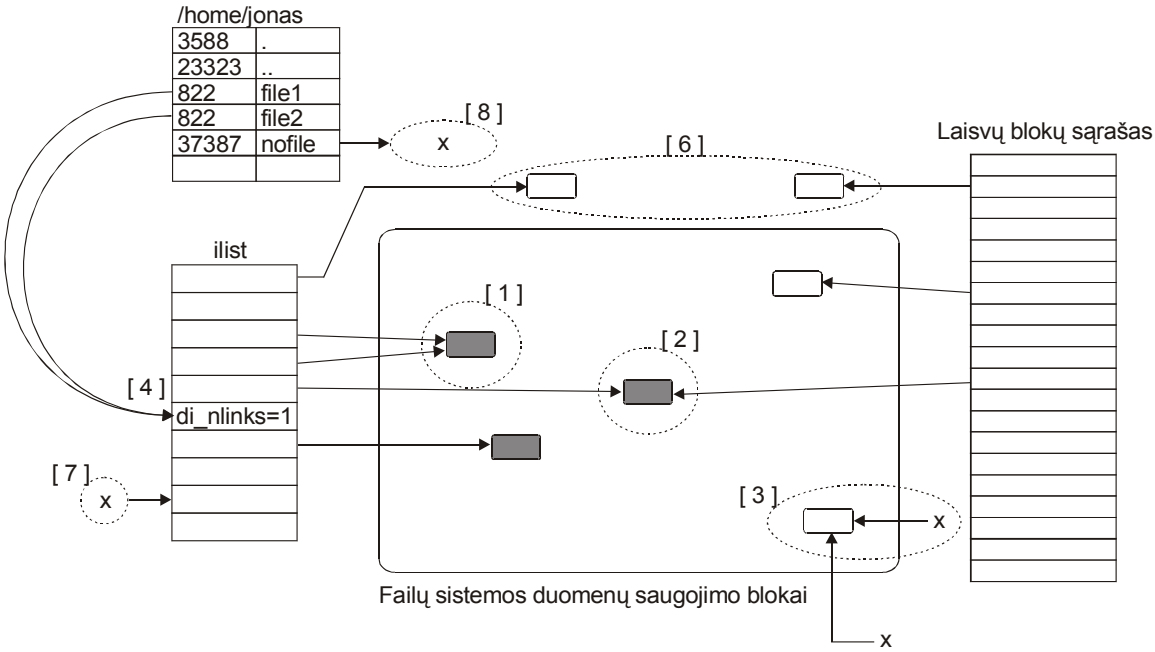
Žymiai prastesnius padarinius gali sukelti failų metaduomenų ir superbloko informacijos sugadinimas. Peržvelkime antro failo vardo sukūrimo operaciją, kurią sudaro dvi funkcijos:

1. naujo įrašo kataloge sukūrimas, kuris rodo į t. t. inode ir
2. inode vardų skaičiaus skaitliuko padidinimas vienetu.

[4] Kai kada administratoriai naudoja kelių sync(2) komandų iškvietimą vieną po kitos. Tokiu būdu yra didesnė tikimybė, kad diske bus išsaugota visa aktuali informacija.

Tarkim, avarinis sistemos sustabdymas įvyksta tarp pirmosios ir antrosios operacijos. Tada, po sistemos startavimo egzistuos du failo vardai (du įrašai kataloge), kurie adresuoja į inode, kuriame esantis laukas di_nlinks yra lygus 1 (4.14 pav., a). Jei bus ištrintas vienas iš vardų, tai sukels failo sunaikinimą, t. y. atlaisvinami duomenų blokai ir inode. Likęs failo vardas rodys į nebeaktyvų failo deskriptorių, kuris ilgainiui taps visai kitu failu (4.14 pav., b).

Didelę reikšmę turi ir darbo su failo metaduomenimis operacijų eiliškumas. Tarkime, sukeiskime aukščiau pateiktas operacijas vietomis ir nutraukime sistemos darbą po pirmosios, inode vardų skaitliuko padidinimo. Tokiu būdu, inode vardų skaitliukas bus didesnis nei realus failo vardų skaičius ir ištrinus visus prieinamus failo vardus pats failas realiai nebus ištrintas, t. y. nebus atlaisvintas inode ir duomenų blokai (4.14 pav., c).



4.15 pav. Galimos failų sistemos klaidos.

Paprastai operacijų eiliškumas yra parenkamas stengiantis minimizuoti avarinių situacijų padarinius ir siekiant optimizuoti bendrą funkciją.

Disko ir jo duomenų atvaizdo atmintyje sinchronizacijos praradimas kylant avarinei situacijai gali sukelti sekančias klaidas (4.15 pav.):

1. į vieną duomenų bloką adresuoja keli inode (blokas priklauso keliems failams);
2. duomenų blokas pažymėtas, kad yra laisvas, tačiau į jį rodo inode;
3. duomenų blokas pažymėtas, kad užimtas, kai tuo tarpu jis yra laisvas;
4. neteisingas nuorodų į inode skaičius (trūksta failo vardų arba yra jų perteklius);
5. nesutampa failo dydis ir inode adresuojamų duomenų blokų suma;
6. neteisingi duomenų blokų adresai (adresas išeina iš failų sistemos ribos);
7. prarasti failai (tvarkingi inode, bet nėra įrašų kataloguose);
8. neleistini inode numeriai katalogų įrašuose.

Šias klaidas gali padėti ištaisyti fsck(1M) programa, kuri pabando ištaisyti failų sistemą. Šią programą galima automatiškai paleisti kiekvieną kartą startuojant sistemą arba administratoriui surinkus sekančią komandą:

```
# fsck [options] filesystem
```

Failų sistemos tikrinimas ir taisymas turėtų vykti tik numontavus tikrinamą failų sistemą. Kitu būdu programa gali ne ištaisyti, bet sugadinti failų sistemą. Išimtį sudaro šakninė failų sistema, kurios nėra galima numontuoti, todėl fsck programą reikia paleisti naudojant –b opciją, kuri perkrauna sistemą iškarto po failų sistemos taisymo.

5 I/O posistemė

UNIX I/O architektūra nuo vartotojo proceso yra paslėpta po keliais interfeisais, vienas kurių – failų sistema. Nors procesui yra pateikiamas patogus interfeisas per vieningą failų sistemos hierarchiją, tačiau realų darbą su fiziniais įrenginiais, skirtingai nei paprastų failų atveju, atlieka įvedimo/išvedimo posistemė.

5.1 Aparatūrinių įrenginių tvarkyklės

Tvarkyklės suteikia interfeisą tarp UNIX branduolio ir aparatūros. Jie paslepia kompiuterio techninės įrangos savitumus nuo kitų operacinės sistemos dalių ir tuo žymiai supaprastina įrenginių palaikymą ir sistemos pernešimą ant kitokios konfigūracijos kompiuterio.

UNIX sistemose egzistuoja labai daug tvarkyklių, kurių dalis suteikia priėjimą prie aparatūrinių įrenginių, tokių kaip spausdintuvai, kieti diskai ir terminalai, o kiti suteikia su aparatūra nesusijusius servisus. Pastarųjų pavyzdžiu gali būti /dev/kmem tvarkyklės, skirtos darbui su virtualiąja atmintimi ir /dev/null “nulinis” įrenginys.

Sistemos startavimo metu branduolys paleidžia nustatytų tvarkyklių iniciacijos procedūras. Daugumoje UNIX sistemų šios tvarkyklės išveda terminale pranešimus apie tai, kad tvarkyklė yra surasta ir iniciacija atlikta tvarkingai, o taip pat ir įrenginio bei tvarkyklės parametrus.

5.1.1 Tvarkyklių tipai

Tvarkyklės yra skirstomos pagal jų galimybes bei kokiu būdu su jais yra bendraujama. Paprastai yra pateikiami trys tvarkyklių tipai:

simbolinės (baitinės) tvarkyklės. Šios tvarkyklės skaitymo ir rašymo operacija atlieka po vieną baitą, t. y. nuosekliai, po vieną baitą yra skaitomas įvedimo ar rašomas išvedimo srautas. Tokiais įrenginiais yra modemai, terminalai, spausdintuvai, manipuliatoriai (pelė) ir pan. Priėjimas prie tokių tvarkyklių yra atliekamas nenaudojant buferio. Jei vis dėlto buferis yra reikalingas, tai yra naudojamos specialios duomenų struktūros, vadinamos clist.

blokinės tvarkyklės. Šios tvarkyklės rašymo ir skaitymo operacijas atlieka po vieną bloką, t. y. fiksuotomis baitų porcijomis. Tarkim disko minimalus adresuojamas vienetas yra sektorius (keli šimtai baitų), todėl ir I/O operacijos yra atliekamos atitinkamais blokais. Paprastai blokiniai įrenginiai naudoja duomenų perdavimo buferius, kurie ir yra tikrasis tarpininkas tarp failų sistemos ir I/O posistemės bei fizinio įrenginio. Nors tvarkyklės leidžia atlikti operacijas su mažesniais duomenų gabalais nei vienas blokas, bet tai vis tiek sukelia vieno bloko nuskaitymą, jo dalies modifikavimą ir pakeisto bloko įrašymą.

žemo lygio tvarkyklės (raw drivers). Šis blokinių tvarkyklių tipas leidžia atlikti duomenų apsikeitimą su įrenginiais apeinant buferius. Tai leidžia kreiptis į įrenginį naudojant įvairaus dydžio elementais, t.y. nesutampančiais su bloku. Apsikeitimas duomenimis vyksta nepriklausomai nuo failų sistemos ir buferio ir tai leidžia branduoliui vykdyti greitą duomenų apsikeitimą su įrenginiu, nedarant papildomų duomenų kopijų.

Paveiksle 5.1 yra pateikiama supaprastinta įrenginių tvarkyklių sąveika su kitomis UNIX posistemėmis schema.

Ne visos tvarkyklės yra naudojamos darbui su fiziniais įrenginiais, tokiais kaip tinklo adapteris ar monitorius. Dalis tvarkyklių nėra susiję su fiziniais įrenginiais, o tiesiog naudojasi I/O posistemės mechanizmais ir teikia vartotojo procesams įvairius servisus. Jos yra vadinamos programinėmis arba pseudo-įrenginių tvarkyklės. Pateiksime keletą pseudo-įrenginių ir jų tvarkyklių:

`/dev/kmem` Suteikia priėjimą prie branduolio virtualios atminties. Žinant branduolio

vidinių struktūrų virtualius adresus, procesas gali skaityti juose saugomus duomenis. Naudojantis šia tvarkyklė, pavyzdžiui, galima parašyti savo ps(1) programos versiją.

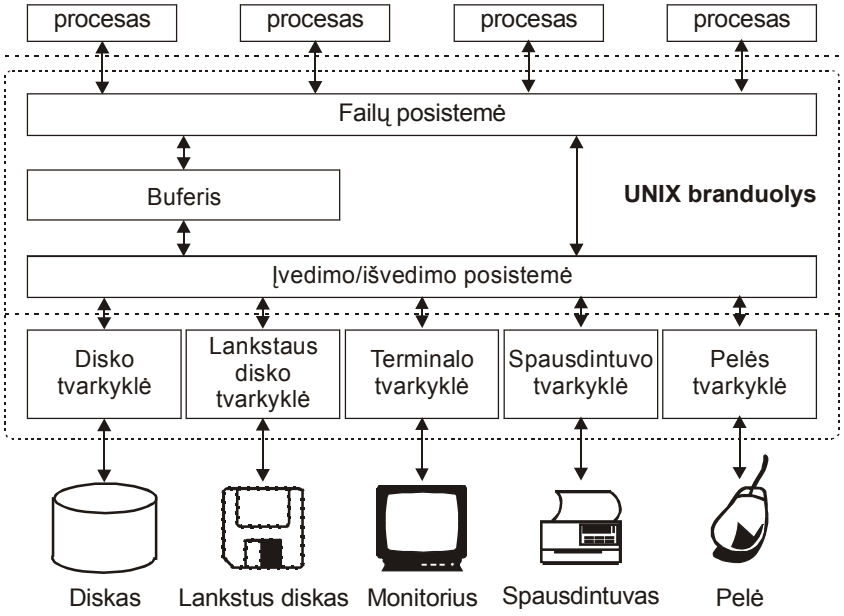
`/dev/ksyms` Suteikia priėjimą prie vykdomo failo branduolio srities skyriaus, kuriame yra saugomos simbolių lentelės. Kartu su /dev/kmem ši tvarkyklė suteikia patogų branduolio vidinių struktūrų analizės mechanizmą.

`/dev/mem` Suteikia priėjimą prie fizinės atminties.

`/dev/null` "Nulinis" įrenginys. Rašant į šį įrenginį duomenys paprasčiausiai yra ištrinami, o skaitant yra grąžinama 0 baitų. Kartais jis yra naudojamas siekiant paslėpti išvedimą, tarkime apie klaidas:

```
$ find / >/dev/null
```

`/dev/zero` Užpildo pateiktą buferį nuliais. Dažnai naudojamas inicijuojant atminties sritį.



5.1 pav. UNIX tvarkyklės.

5.1.2 Tvarkyklių vardų mnemonika UNIX failų sistemoje

Išvedant katalogo turinį su `$ ls -l` komanda, stulpelyje, kuriame paprastiems failams yra pateikiamas failo dydis, tvarkyklėms yra išvedami du skaičiai. tai yra vadinamasis vyresnysis (major) ir jaunesnysis (minor) skaičiai. Vyresnysis skaitmuo nurodo konkrečią tvarkyklę, pvz. pseudo-terminalą, o jaunesnysis yra perduodamas tai tvarkyklei ir rodo į konkretų įrenginį, tarkim pseudo-terminalą numeris du (5.2 pav.).

Aparatūrinių įrenginių failų pavadinimai labai priklauso nuo konkrečios UNIX sistemos. Tačiau net gerai nežinant sistemos, naudojantis bendra pavadinimų logika galima atskirti į kokį įrenginį rodo konkretus failas. Taip pat galima atskirti kokio tipo yra duotoji tvarkyklė, tarkim Solaris sistemose priėjimą prie disko skirsnio yra pateikiamas sekančiu pavidalu:

```
/dev/dsk/c0t4d0s2
```

Šis failas yra blokinis disko skirsnio interfeisas, o jam atitinkantis simbolinis interfeisas yra pateikiamas taip

```
/dev/rdsk/c0t4d0s2
```

Priėjimo prie disko failai yra patalpinami į: blokiniai - /dev/dsk, o simboliniai - /dev/rdsk. Tokia duomenų saugojimo struktūra yra charakteringa visoms System V šeimos sistemoms. Failo vardą galima pateikti sekančiu būdu:

```
c#tLdMsN
```

, kur K – kontrolerio numeris, L – įrenginio numeris (SCSI diskui tai yra jo ID), M – skyriaus numeris, N – loginis SCSI įrenginio numeris (LUN). Tokiu būdu įrenginio failas c0t4d0s2 suteikia priėjimą prie pirmojo SCSI kontrolerio disko, kurio ID=4, LUN=2 pirmojo skirsnio.

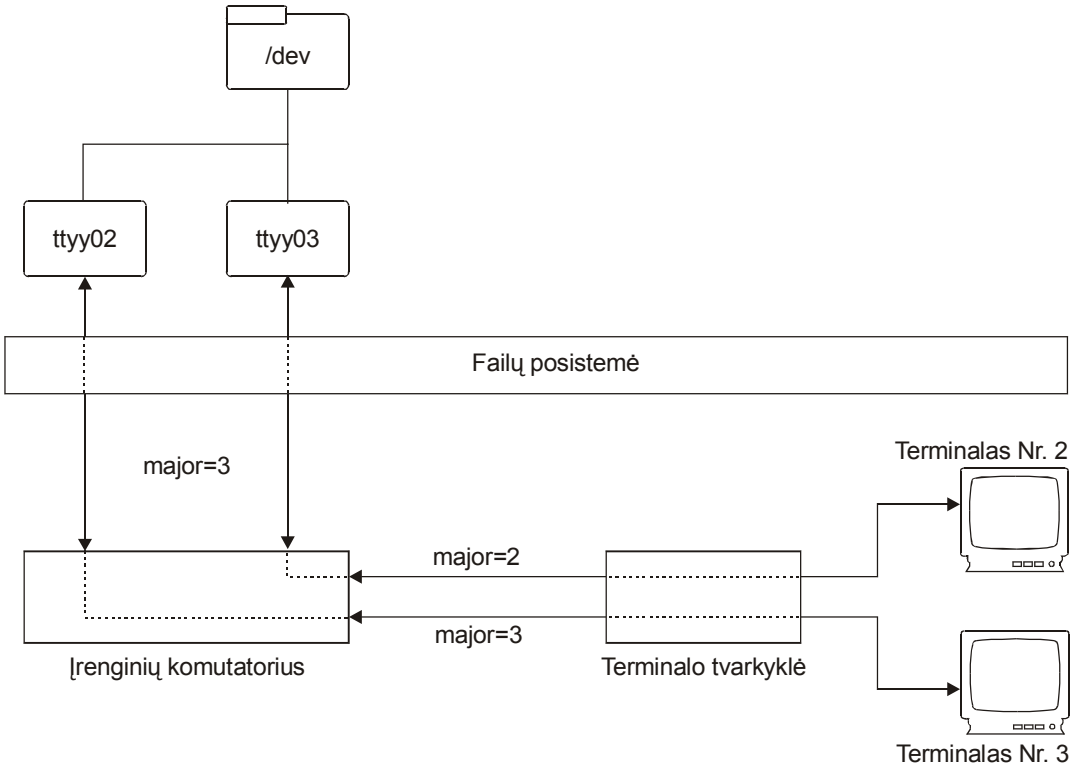
Kartais yra naudojamos simbolinės nuorodos, kurios supaprastina įrenginių failų pavadinimus, tarkim

```
/dev/root ->/dev/dsk/c0t4d0s2
```

5.1 lentelėje yra pateikiami specialiųjų įrenginių failų pavadinimų pavyzdžiai.

5.1 lentelė. Specialiųjų įrenginių failų pavadinimų pavyzdžiai.

Bendras vardas	Pavyzdys	Aprašymas
`/dev/rmt`*n*	`/dev/rmt0`	Magnetinės juostos įrenginys
`/dev/nrmt`*n*	`/dev/nrmt0`	Magnetinės juostos įrenginys, kuris darbo pabaigoje neatsuka juostos į pradžią.
`/dev/rst`*n*	`/dev/rst0`	SCSI magnetinės juostos įrenginys
`/dev/cd`*n*	`/dev/cd0`	CD-ROM įrenginys
`/dev/cdrom`	-	
`/dev/ttyp`*n*	`/dev/ttyp0`	Pseudo-terminalas (slave)
`/dev/ptyp`*n*	`/dev/ptyp0`	Pseudo-terminalas (master)
`/dev/console`	-	Sisteminė konsolė
`/dev/tty`	-	Einamojo proceso valdančiojo terminalo terminalinės linijos sinonimas



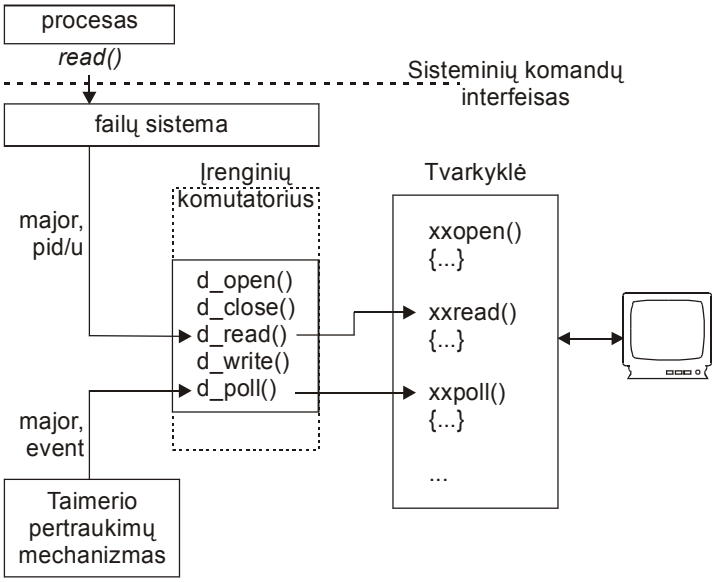
5.2 pav. UNIX tvarkyklių vyresnysis ir jaunesnysis skaičiai.

5.1.3 Tvarkyklių architektūra

Įrenginio tvarkyklė yra adresuojama naudojant vyresnįjį įrenginio numerį (major number).Jaunesnysis numeris (minor number) yra interpretuojamas pačios tvarkyklės. Tokiu pavyzdžiu gali būti disko interfeisas: kiekvienas disko skirsnis yra prieinamas naudojant vieną

ir tą pačią tvarkyklę, t. y. ta patį vyresnįjį skaičių, o konkrečius skirsnius žymi jaunesnysis numeris.

Priėjimą prie tvarkyklių branduolys organizuoja per specialias duomenų struktūras – įrenginių komutatorius, kurių kiekvienas saugo nuorodas į atitinkamas tvarkylių funkcijas – įėjimo taškus (entry points). Vyresnysis baitas tėra struktūros numeris komutatorių eilėje. Per šiuos įrenginių komutatorius yra suteikiamas unifikuotas interfeisas.



5.3 pav. Darbas su baitiniu (simboliniu) įrenginiu.

Šis interfeisas skiriasi baitiniams – `cdevsw` - ir blokiniams – `bdevsw` – įrenginiams. Branduolys sukuria du atskirus masyvus kiekvienam komutatoriaus tipui ir visos tvarkyklės turi savo įrašus šiuose masyvuose. Jei įrenginys turi blokinę ir baitinę tvarkyklę, tai jis turi po įrašą abiejuose masyvuose.

Tipinis šių masyvų aprašymas gali būti toks:

```
struct bdevsw[] {
       int (*d_open)();
       int (*d_close)();
       int (*d_strategy)();
       int (*d_size)();
       int (*d_xhalt)();
       ...
} bdevsw[];
struct cdevsw[] {
       int (*d_open)();
       int (*d_close)();
       int (*d_read)();
       int (*d_write)();
       int (*d_ioctl)();
       int (*d_xpoll)();
       int (*d_xhalt)();
       struct streamtab *d_str;
       ...
} cdevsw[];
```

Reikiamos tvarkyklės funkciją `open()` branduolys iškviečia sekančiu būdu:

```
(*bdevsw[getmajor(dev)].d_open)(dev,...);
```

, kur vienu iš kintamųjų `dev` (tipas `dev_t`) yra perduodami vyresnysis ir jaunesnysis skaičiai. Naudojant makro-programą `getmajor()` iš `dev` yra išgaunamas vyresnysis numeris.

Taigi, komutatorius suteikia vieningą įrenginių interfeisą. Funkcijų realizaciją atlieka konkrečių įrenginių tvarkyklės. Jei kai kurių funkcijų tvarkyklė nepalaiko, tai ji, nežiūrint to, privalo

realizuoti tuos įėjimo taškus panaudodama specialias “uždarymo” procedūras. Tvarkyklių funkcijų pavadinimai yra suteikiami naudojant tam tikrus susitarimus. Kiekviena tvarkyklė turi unikalų dviejų simbolių prefiksą, tarkim `/dev/kmem` naudoja prefiksą mm ir jos funkcijos turi pavadinimus `mmopen(), mmclose(), mmread(), mmwrite()` ir pan.

Lentelėje 5.2 yra pateikimos kai kurios standartinės tvarkyklių funkcijos, kuriose prefiksus žymi *xx* simboliai.

5.2 lentelė. Tipiniai tvarkyklių įėjimo taškai.

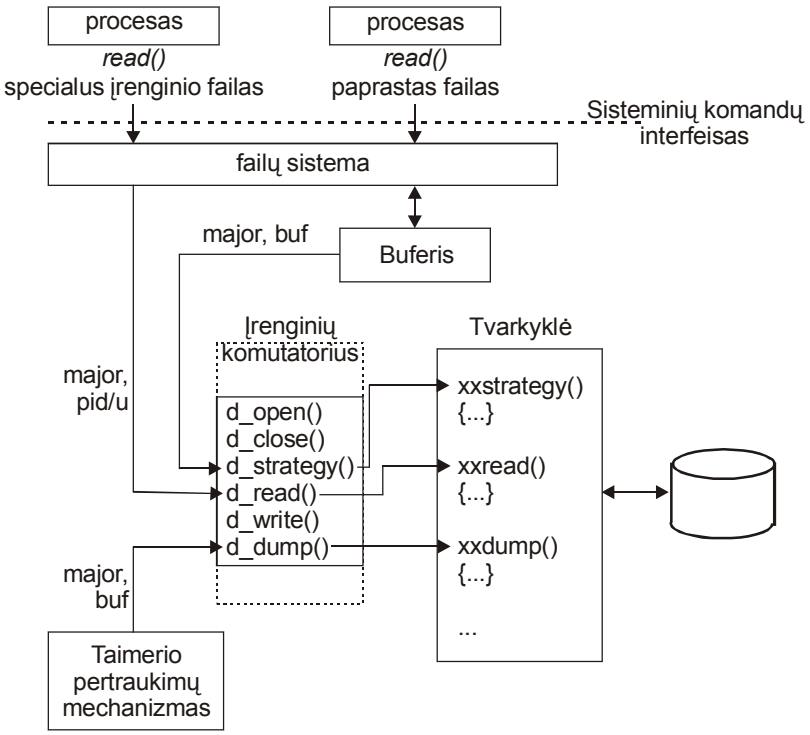
Funkcija	**Baitinis**	**Blokinis**	**Žemo lygio**	**Aprašymas**
xx`open()`	1	1	1	Inicijuoja fizinį įrenginį ir vidines tvarkyklės struktūras. Pavyzdžiui, kiekvieną kartą atidarant yra sukuriamos buferių duomenų struktūros, kurios leidžia įrenginį naudoti keliems procesams
xx`close()`	1	1	1	Iškviečiamas tada, kai nei vienas procesas nebesikreipia į įrenginį; gali įvykdyti įrenginio išjungimą ir pan.
xx`read()`	1	0	1	Skaitomi įrenginio duomenys.
xx`write()`	1	0	1	Rašomi duomenys į įrenginį.
xx`ioctl()`	1	0	1	Bendras įrenginio valdymo interfeisas.
xx`intr()`	1	1	1	Iškviečiama, kai kyla pertraukimas, susijęs su duotuoju įrenginiu. Gali įvykti duomenų nuskaitymas į buferį, iš kurio vėliau procesas vykdys skaitymą.
xx`poll()`	1	0	1	Apklausia, dažniausiai pertraukimų nepalaikančius, įrenginius
xx`halt()`	1	1	1	Sustabdoma tvarkyklė.
xx`strategy()`	0	1	1	Suteikia bendrą blokinių įrenginių I/O operacijų interfeisą. Skaitymui ir rašymui gali būti suteikta sava strategija, kuria bus stengiamasi padidinti darbo greitį. Jei įrenginys užimtas, užklausa patalpinama į eilę, o faktinė operacija bus atliekama vėliau.
xx`print()`	0	1	1	Tvarkyklės pranešimo išvedimas į ekraną, kuris dažniausiai vykdomas sistemos startavimo metu.

Branduolys iškviečia tam tikras tvarkyklių funkcijas tada, kai procesas savo ruožtu iškviečia t. t. sisteminę funkciją. Pavyzdžiui, jei procesas iškviečia simbolinio įrenginio read(2) funkciją, branduolys suranda reikiamą tvarkyklę ir sužadina jos xxread(2) procedūrą. Jei tas įrenginys buvo blokinis, tai branduolys iškvies atitinkamos tvarkyklės xxstrategy() funkciją.

Egzistuoja penki pagrindiniai branduolio funkcijų iškvietimo atvejai:

1. auto-konfigūracija, kuri paprastai vyksta UNIX sistemos startavimo metu, kai branduolys nustatinėja kokie įrenginiai yra prieinami.
2. įvedimas / išvedimas, kurį gali inicijuoti tiek vartotojo programos, tiek kitos branduolio posistemės, tokios kaip atminties valdymo mechanizmas.
3. pertraukimų apdorojimas. Kai kyla įrenginio inicijuotas pertraukimas (jei jis gali jį inicijuoti), tai branduolys iškviečia įrenginio pertraukimo apdorojimo funkciją.
4. specialių komandų apdorojimas. Branduolys iškviečia atitinkamą tvarkyklės funkciją, kai gaunama specialioji sisteminė ioctl(2) komanda.

5. reinicializacija/sustabdymas, kurio reikalauja kai kurie įrenginiai. Taip pat šios funkcijos yra iškviečiamos kai UNIX sistema yra stabdoma (išjungiama).



5.4 pav. Darbas su blokiniu įrenginiu.

5.3 ir 5.4 paveiksluose yra pateikiamos darbo su blokiniais ir baitiniais įrenginiais schemos.

Svarbu pažymėti, kad dauguma tvarkyklių funkcijų atsakingų už duomenų tarp proceso atminties ir įrenginio apsikeitimą, taip pat atlieka informacijos kopijavimą iš branduolio adresinės erdvės į proceso adresinę erdvę. Kai branduolys iškviečia tvarkyklės funkciją, visi veiksmai yra vykdomi sisteminiame proceso kontekste, tačiau pats iškvietimas gali būti įvairus:

- funkcija gali būti iškviesta paprašius procesui; tarkim, jei procesas įvykdo sisteminę komandą read(2), branduolys iškviečia atitinkamą tvarkyklės funkciją, kuri atlieka tiesioginį darbą su failu. Tada yra sakoma, kad funkcija dirba užduoties kontekste.
- funkcija gali būti iškviesta paprašius kuriai nors kitai branduolio posistemei. Tarkim, blokinio įrenginio tvarkyklės funkciją *xx*`strategy()` gali iškviesti puslapinio mechanizmo demonas, stengdamasis numesti einamuoju momentu nereikalingus puslapius (kaip taisyklė, į kietąjį diską). Puslapinio mechanizmo daemonas yra sisteminis procesas, kuris yra vykdomas tik sisteminiame branduolio kontekste, todėl *xx*`strategy()` funkcija šiuo atveju dirbs sisteminiame kontekste.
- jei tvarkyklės funkcija yra iškviečiama apdorojant pertraukimą, tai ji dirba pertraukimo kontekste, t.y. specialiame sisteminiame kontekste.

Funkcijos kontekstas ir jos iškvietimo priežastys leidžia perskelti tvarkyklę į dvi abstrakčias dalis: viršutinę (top half) ir apatinę (bottom half).

Viršutinė tvarkyklės dalis dirba sinchroniniu režimu, t. y. funkcijas iškviečia vartotojo procesas ir ji dirba jo kontekste. Tokiu būdu, funkcijai yra prieinama proceso adresinė erdvė ir u-area. Esant reikalui šios funkcijos gali pervesti procesą į miego režimą (sleep() funkcija) . Šiam tipui priklauso įvedimo ir išvedimo bei įrenginio valdymo funkcijos.

Apatinės dalies funkcijos dirba asinchroniniu režimu. Tarkim, pertraukimo apdorojimo funkcijos iškvietimo negalima prognozuoti ir branduolys negali kontroliuoti, kada vieną ar kitą pertraukimo apdorojimo funkciją gali iškviesti. Tokių funkcijų darbas vyksta sisteminiame branduolio kontekste ir neturi jokios sąsajos su einamojo proceso kontekstu. Tokiu būdu šios funkcijos neturi priėjimo prie einamojo proceso duomenų struktūrų ir negali juo manipuliuoti (tarkim, užmigdyti).

Visos aukščiau pateiktos funkcijos, išskyrus xxhalt(), xxpoll() ir xxintr(), priklauso viršutiniam tvarkyklės lygmeniui. Funkcija xxhalt() iškviečiama branduolio tada, kai sistema yra stabdoma, todėl ji dirba sisteminiame kontekste, kuris nėra susijęs su jokiu procesu.

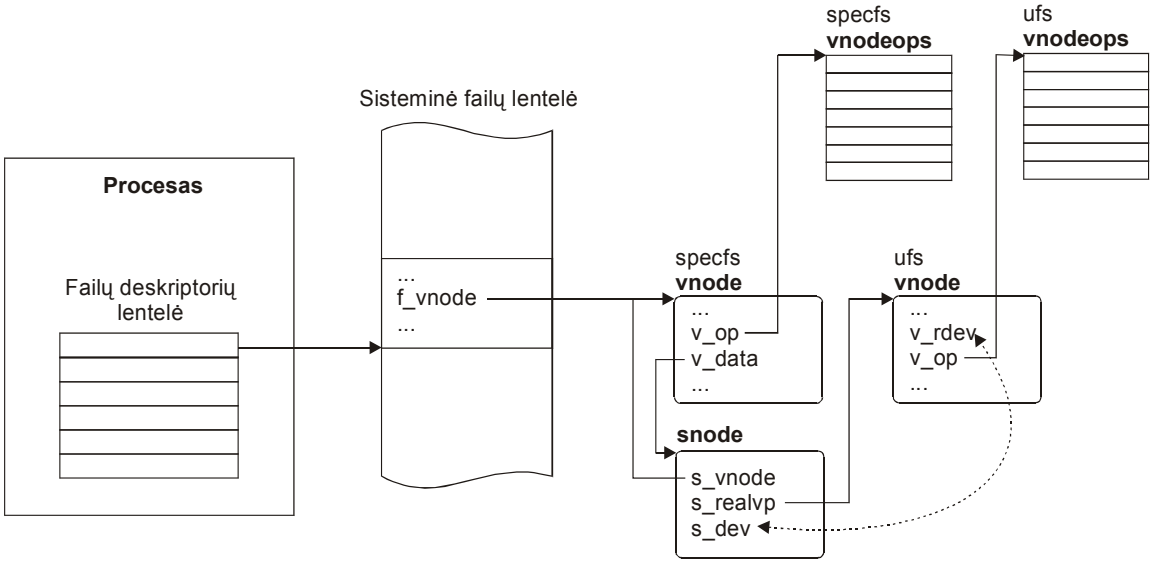
Funkcija xxpoll() yra iškviečiama kas n-tąjį taimerio tiką ir ji apklausia įrenginį, kuris yra pažymėtas, kad negali arba nenori naudoti aparatūrinių pertraukimų. Taip pat xxpoll() gali būti iškviečiama siekiant imituoti pertraukimus, t.y. iš karto sužadinti xxintr() funkciją. Kaip matyti xxpoll() ir xxintr() funkcijos nėra susijusios su jokiu vartotojo procesu. Daugumoje UNIX sistemų šios funkcijos yra iškviečiamos ne per įrenginių komutatorių, o naudojant specialias branduolio lenteles.

SVR4 sistemose yra apibrėžtos dvi papildomos funkcijos – init() ir start(). Sistemos startavimo metu, prieš pakraunant branduolį yra paleidžiamos tvarkyklių xxinit() funkcijos, o tuojau po branduolio pakrovimos – xxstart().

5.1.4 Tvarkyklių failų sistemos interfeisas

Nagrinėdami virtualios failų sistemos interfeisą minėjome virtualius indeksinius deskriptorius (vnode), kurių kiekvienas turi savitą abstrakčių operacijų rinkinį, per kurias yra realizuojamas priėjimas prie realių įrenginių funkcijų.

Tokia failų valdymo architektūra daugumoje šiuolaikinių UNIX operacinių sistemų reikalauja atitinkamų struktūrų, kurios leistų failų sistemai prieiti prie įrenginių tvarkyklių. Kaip jau minėjome UNIX sistemose periferinių įrenginių interfeisas yra realizuojamas per specialius įrenginių failus, kurie yra saugomi specialiame kataloge šakninėje failų sistemoje. Ta failų sistema taip pat yra realizuota kažkokiame įrenginyje, tarkim diske, kurio tipas yra ufs. Tokiu būdu įrenginio failo paslaugoms bus suteiktos visos ufs failų sistemos funkcijos.

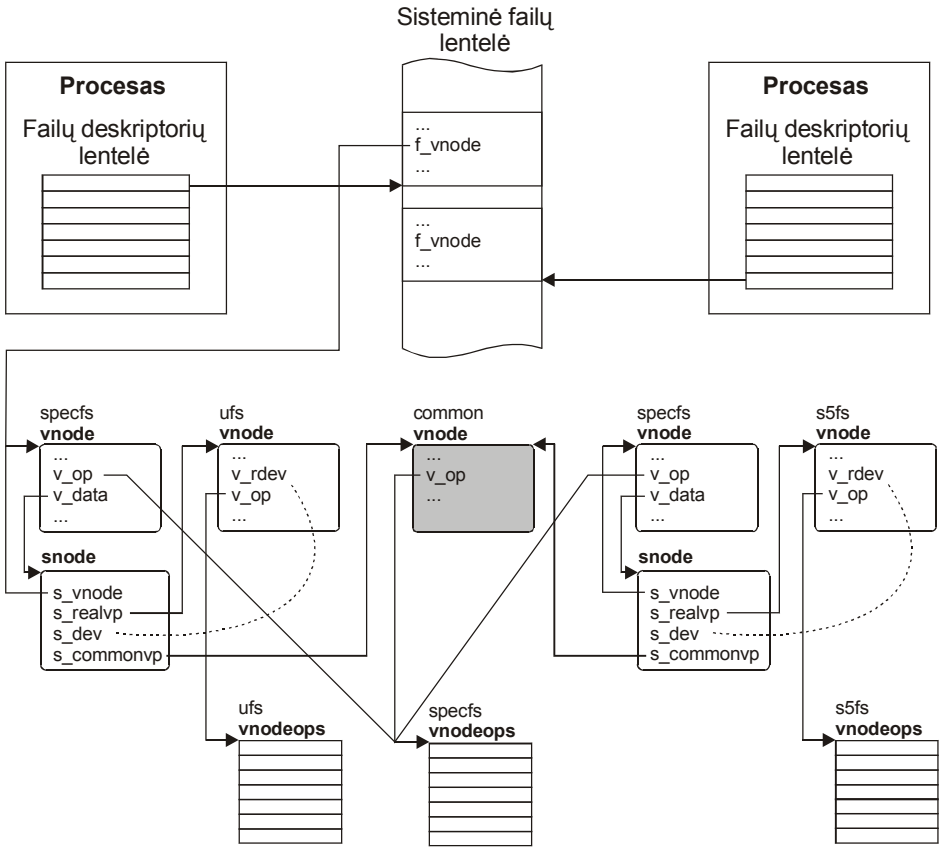


5.5 pav. Proceso ryšys su įrenginio failu.

Tačiau įrenginių failai nėra paprasti ufs failai ir standartinio failų sistemos funkcionalumo čia jau nepakanka. Faktiškai visos įmanomos įrenginių failų operacijos yra realizuojamos tvarkyklėse, todėl žymiai logiškiau įrenginio vnode operacijas atvaizduoti ne į failų sistemos operacijų vektorių, o tiesiogiai į įrenginių komutatorių.

System V šakos operacinės sistemos naudoja specialų failų sistemos tipą, vadinamą devfs arba specfs[5]. Šio tipo failų vnode rodo į reikiamo įrenginio komutatoriaus operacijų vektorių. Po failo atidarymo yra sukuriamas specialus inode, kuris peradresuoja operacijas įrenginių tvarkyklėms. Tačiau atidarant įrenginio failą dalį operacijų atlieka failą talpinanti failų sistema, pavyzdžiui ufs failų sistema atlieka failo vardo transliaciją, ko pati specfs negali atlikti, nes praktiškai ji yra virtuali failų sistema.

Pateiksime pavyzdį: tarkim procesas specialiam įrenginio failui /dev/kmem iškviečia sisteminę open(2) funkciją. Iškviečiama failų sistemos ufs funkcija ufs_lookup() funkcija, kuri transliuoja failo vardą, jį suranda ir sukuria failui vnode (jei jo dar nėra). Tačiau ufs_lookup() nustato, kad duotasis failas yra IFCHR, simbolinio įrenginio tipo, todėl vietoj ufs_open(), kuri būtų visiškai beprasmiška įrenginio failui, bus iškviesta speciali specfs funkcija. Pastaroji pažiūri, ar duotajam įrenginiui nėra sukurtas specialusis indeksinis deskriptorius (special inode, snode) ir jei tokio nėra, jį sukuria. Pagal standartinę procedūra taip pat bus sukuriamas virtualusis indeksinis deskriptorius, vnode, kuris rodo į specops – specialiosios failų sistemos operacijų vektorių, kurios jau tiesiogiai dirba su tvarkyklės operacijomis. Tarkim, specfs funkcijos spec_open(), spec_read() arba spec_write() iškvies atitinkamus tvarkyklių įėjimo taškus: xxopen(), xxread() ir xxwrite(). Po to, ufs_open() funkcijai bus perduotas vnode adresas, pastaroji jį perduos open(2) sisteminei funkcijai, o ši patalpins tą adresą į proceso u-area failo deskriptorių lentelę ir grąžins procesui indeksą. Tokiu būdu, visas kitas operacijas perima specfs ir iškviečia atitinkamos tvarkyklės operacijos. Proceso ryšys su įrenginio failu yra pateiktas 5.5 paveiksle.



5.6 pav. Priėjimas prie įrenginio naudojant įvairias duomenų struktūras.

[5] SVR4 naudojama specfs failų sistemos terminologija. SCO UNIX, kuri formaliai yra SVR3.2 ir turi daug SVR4 bruožų šį failų sistemos tipą vadina devfs.

Visgi, aukščiau pateikta schema yra nepilna ir turi keletą žymių trūkumų. Problema yra tame, kad įrenginių failai gali būti išdėstomi skirtingose nei tvarkyklės failų sistemose. Tokiu atveju, branduolys negali nustatyti kiek vartotojo procesų naudoja įrenginį, o šio skaičiaus reikia, kai visi procesai baigia darbą su įrenginiu ir reikia iškviečiama xxclose() komanda.

Šios problemos sprendimui specfs failų sistema sukuria specialų indeksinį deskriptorių – bendrasis snode (common snode), kuris kontroliuoja priėjimą prie įrenginio tvarkyklės ir yra vienintelis tvarkyklės interfeisas. Kiekvienam įrenginiui, t.y. kiekvienai įrenginio tvarkyklei yra sukuriamas vienintelis csnode, tai atliekamas pirmą kartą kreipiantis į įrenginį. Kiekvienas specialusis įrenginio failas turi po savo snode specfs failų sistemoje ir po vieną vnode virtualiojoje ir inode fizinėje failų sistemoje.

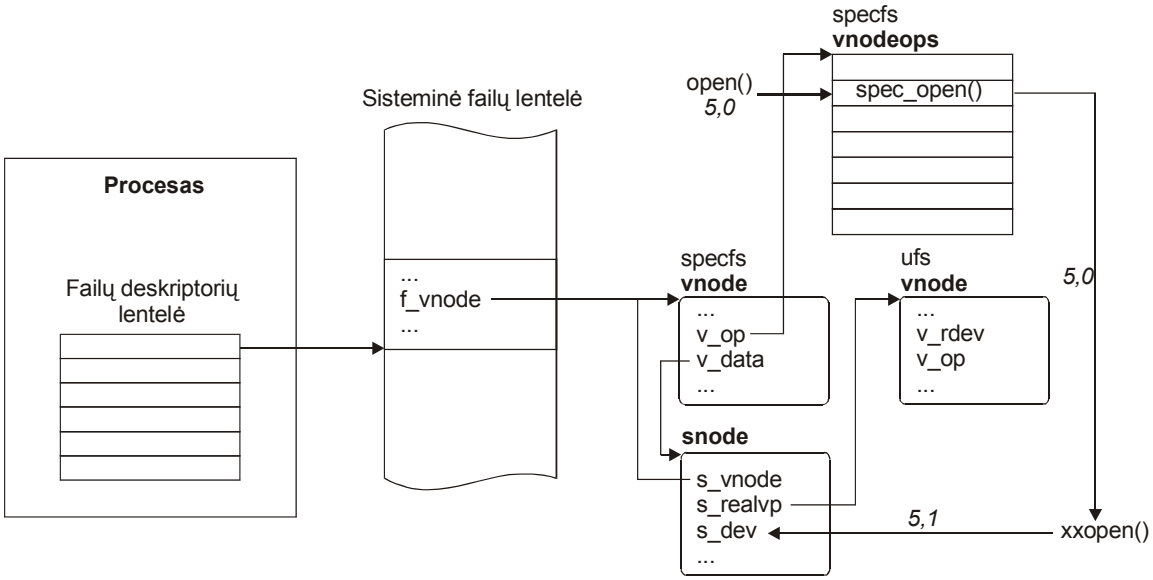
Siekiant susieti visas šias struktūras, snode turi du laukus: s_commonvp, rodantis į common snode, ir s_realvp, rodantis į failų sistemos, kurioje yra specialusis įrenginio failas, vnode.

Vienų ar kitų indeksinių deskriptorių panaudojimas priklauso nuo konkrečių operacijų, kurios yra atliekamos su įrenginiu ar jo failu. Dauguma operacijų su įrenginiu nepriklauso nuo paties įrenginio failo, ir tada yra naudojami specfs vnode. Kai prireikia atlikti operaciją su pačiu įrenginio failu – tarkim, pervadinti failą ar tikrinamos priėjimo prie failo teises – dirbama su realios failų sistemos, pvz. ufs vnode.

Pateiktos architektūros schema yra pateikta 5.6 paveiksle.

5.1.5 Klonai

Kaip jau minėta, vyresnysis įrenginio numeris (skaičius) identifikuoja įrenginio tvarkyklę, o jaunesnysis gali būti naudojamas priėjimui prie įrenginio, tarkim prie tam tikro disko skyriaus.



5.7 pav. Klonų sukūrimas naudojant rezervuotus jaunesniuosius įrenginio skaičius.

Tačiau mažasis įrenginio skaičius gali būti panaudojamas ir kitiems tikslams. Pateiksime du, su jaunesniaisiais įrenginio numeriais susijusius atvejus:

1. jaunesnysis numeris naudojamas loginės tvarkyklės (klono) identifikacijai. Pavyzdžiui, kai procesas kreipiasi į tinklo adapterio tvarkyklę, jam reikalingas tik vyresnysis skaičius,o jaunesnysis nėra naudojamas. Tai atliekant yra sukuriamas `snode` ir yra iškviečiama `spec_open()` funkcija. Pastaroji, savo ruožtu iškviečia atitinkamos tvarkyklės `xxopen()` funkciją, perduodama jai `snode.s_dev` argumentą, kuriame yra saugomi įrenginio numeriai. Ši funkcija išrenka naują laisvą identifikatorių, priskiria jo reikšmę jaunesniajam įrenginio numeriui `snode.s_dev` kintamajame ir sukuria naujas duomenų struktūras. Visos kitos operacijos su įrenginiu naudoja šį `snode.s_dev` kintamąjį ir jo pagalba prieina

prie atitinkamų duomenų struktūrų, vadinamos loginės tvarkyklės arba klono. Ši schema yra pateikta 5.7 paveiksle.

2. naudojama specialus pseudo-įrenginys – klonų tvarkyklė (clone driver). Tokiu būdu, visi įrenginių failai, kurie naudojasi klonų įrenginiu turi tą patį vyresnįjį numerį, klonų tvarkyklės indeksą. Jaunesnysis skaičius identifikuoja tikrąją realaus įrenginio tvarkyklę, kuriai yra sukuriamas klonas. Šios schemos pagrindu dažnai yra realizuojami tinkliniai interfeisai, pseudo-terminalai ir pan. Tokių failų sąrašas atrodytų panašiai į žemiau pateiktąjį:

Kai procesas atidaro, tarkim, tcp failą, yra iškviečiama klonų tvarkyklės clopen() funkcija, kuriai yra perduodami atitinkami numeriai. Ši funkcija pagal jaunesnįjį skaičių atranda reikiamą įėjimo tašką cdevsw[] komutatoriuje ir iškviečia, šio atveju, tcpopen() funkciją, kuriai parduoda įrenginio numerius ir požymį CLONEOPEN. Tai gavusi, tcpopen() funkcija sugeneruoja unikalų identifikatorių, sukuria reikiamas duomenų struktūras (loginį įrenginį, kloną) ir modifikuoja snode.s_dev. Tokiu būdu kiekvienam procesui yra sukuriamas unikalus tcp sujungimas.

5.1.6 Tvarkyklių įterpimas į branduolį

Įrenginių tvarkyklės yra operacinės sistemos branduolio kodo dalis, kuri suteikia jam interfeisą su fiziniai ir pseudo- įrenginiais. Egzistuoja du tvarkyklių įterpimo į branduolį metodai:

1. branduolio perkompiliavimas ir
2. dinaminis tvarkyklės įterpimas sistemos darbo metu.

Tradiciškai tvarkyklių įterpimas reikalauja branduolio perkompiliavimo ir sistemos perkrovimo. Iš principo ši procedūra niekuo nesiskiria nuo paprasto programos kompiliacijos: visi branduolio komponentai yra objektiniai moduliai ir surišėjas (linker) juos suriša į vieną programą. Tokiu būdu tvarkyklės yra įterpiamos statiškai, t.y. nepriklausomai ar faktiškai toks įrenginys yra ar ne, tvarkyklės kodas ir duomenys bus branduolyje iki sekančio jo perkompiliavimo.

Tačiau šiuolaikinių UNIX operacinių sistemų vystymosi tendencija krypsta link dinaminio tvarkyklių įterpimo mechanizmo. Tai daugiausia liečia failų sistemų, tinklinių įrenginių ir jų protokolų tvarkykles (tiksliau, STREAMS posistemės tvarkykles). Tokią galimybę, kuri leidžia plėsti branduolio funkcionalumą nepertraukiant sistemos darbą suteikia pakraunamo tvarkyklės. Vietoj to, kad tvarkyklės modulius įterpinėti į statines lenteles ir interfeisus, branduolys palaiko eilę funkcijų, leidžiančių pakrauti naujas tvarkykles ir išmesti, kai jų nebereikia. Tokiu atveju visos duomenų struktūros taip pat tampa dinaminėmis.

Dinaminis tvarkyklės į branduolį įterpimas reikalauja sekančių operacijų:

- tvarkyklės simbolių dinaminis įterpimas ir surišimas. Ši procedūra analogiška dinaminių bibliotekų pakrovimo atvejui ir yra atliekama specialios pakrovimo programos;
- tvarkyklės ir įrenginio inicializacija;
- tvarkyklės įėjimo taškų (funkcijų) įterpimas į atitinkamų įrenginių komutatorių;
- pertraukimo apdorojimo mechanizmo įterpimas.

Aišku, kad dinamiškai įterpiamų tvarkyklių programa yra žymiai sudėtingesnė ir, šalia standartinių funkcijų, privalo palaikyti papildomas pakrovimo ir išmetimo funkcijas bei papildomas duomenų struktūras. Papildomų funkcijų ir duomenų struktūrų pavyzdys, kurios realizuotos sistemose Solaris 2.5 ir vėlesnėse:

`_init()`	Inicializacijos ir pakrovimo funkcija, kuri yra iškviečiama tvarkyklės pakrovimo metu.
`_fini()`	Funkcija skirta tvarkyklės išmetimui iš branduolio.
`_info()`	Funkcija, kuri suteikia informacija apie tvarkyklę.
`struct modlinkage`	Struktūra, kurią naudoja aukščiau išvardintos funkcijos.
`struct modldrv`	Struktūra, kurioje yra saugomi įėjimo į tvarkyklę taškai

(funkcijos), kurios vėliau yra eksportuojamos į branduolį.

Nežiūrint to, Solaris 2.5 ir vėlesni suteikia papildomas funkcijas darbui su dinaminėmis tvarkyklėmis: `mod_install(9F), mod_remove(9F), mod_info(9F)`.

5.2 Blokiniai įrenginiai

Blokinių įrenginių tvarkyklės yra skirtos periferinių įrenginių aptarnavimui, kurie apsikeičia fiksuoto dydžio duomenų fragmentais – blokais, kurių dydis žymiai viršija vieną baitą. Dažniausiai blokinių tvarkyklių paslaugomis naudojasi failų ir atminties valdymo posistemės. Tarkim, swap mechanizmas dirba su antrinės atminties įrenginiu, kuriame duomenys saugomi blokais, kurių dydis paprastai sutampa su puslapiais, t.y. 4K arba 8K.

Priėjimas prie blokinių įrenginių per tvarkykles yra organizuojamas dviem būdais, priklausomai nuo panaudojimo konteksto:

1. per failų sistemą ir tuo pačiu naudojant buferinį mechanizmą, ir
2. tiesiogiai skaitant iš įrenginio failo.

Pirmuoju atveju dažniausiai yra skaitomas/rašomas paprastas failų sistemos, kuri yra duotajame įrenginyje failas, o antruoju, skaitymas atliekamas naudojant įrenginio failą betarpiškai. Jei yra naudojamas buferis, tai operacijas su įrenginiu atlieka tvarkyklės funkcija xxstrategy(), kuriai perduodamas vienintelis argumentas – nuoroda į struktūrą buf. Antruoju atveju, kai operacijos yra atliekamos tiesiogiai su įrenginio failu naudojamos xxread() ir xxwrite() tvarkyklės komandos, o duomenys diske yra adresuojami fiziniu įrenginio poslinkiu nuo jo pradžios. Tiesioginis kreipimasis į įrenginį, kuriame yra saugoma failų sistema yra įmanomas tik tada, kai pastaroji yra numontuota, t.y. bus užtikrinama, kad nekils nesusipratimų.

5.3 Baitiniai (simboliniai) įrenginiai

Baitiniai įrenginiai sudaro didelę aparatūrinių įrenginių šeimą, į kurią įeina, pavyzdžiui, terminalai, manipuliatoriai (pelė), klaviatūra ir vietiniai (betarpiškai prijungti prie kompiuterio) spausdintuvai. Šie įrenginiai, lyginant su blokiniais perduoda nedidelius duomenų kiekius.

Duomenų apsikeitimas su šiais įrenginiais vyksta tiesiogiai, apeinant buferinius mechanizmus. Tokiu būdu, duomenys yra tiesiogiai kopijuojami iš/į proceso, dirbančio su įrenginiu, adresinės erdvės į/iš įrenginį.

Jei procesas iškviečia sisteminę komandą read(2) arba write(2) iš specialaus įrenginio failo, tai ši užklausa yra nusiunčiama failų sistemai specfs. Gavusi šią užklausą failų sistema iškviečia atitinkamą spec_read() arba spec_write() komandą. Abi šios funkcijos elgiasi panašiai: patikrina ar duotasis įrenginys yra baitinis ir įrenginių komutatoriuje suranda reikiamą tvarkyklę ir iškviečia jos xxread() arba xxwrite() funkcijas. Joms yra perduodami įrenginio numeriai, papildomi parametrai, kurie priklauso nuo konkretaus įrenginio ir tiesiogiai ar netiesiogiai proceso atminties adresas su kuriuo reikia atlikti duomenų apsikeitimą[6].

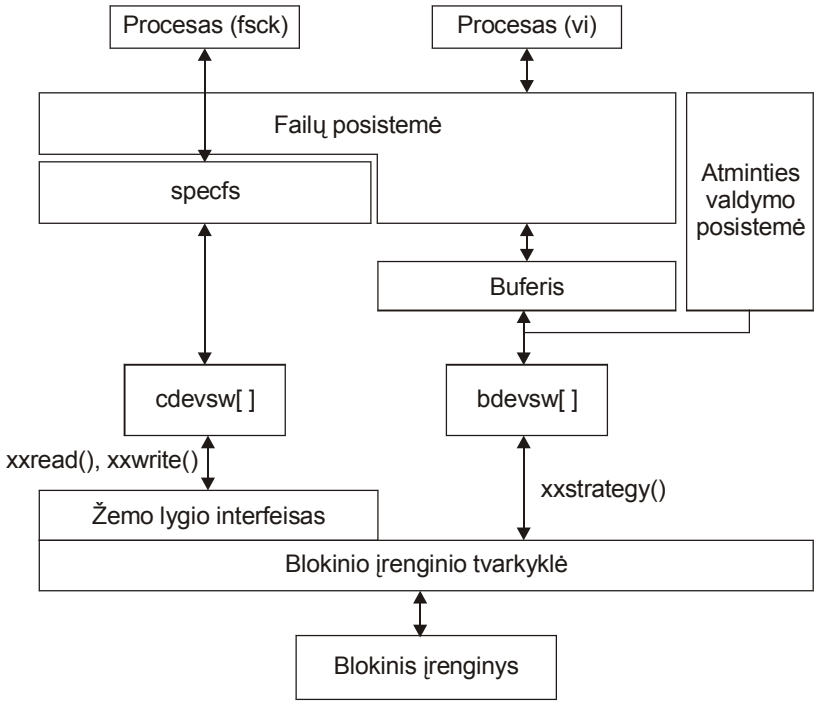
5.3.1 Žemo lygio interfeisas

Kartu su simbolinių įrenginių tvarkyklėmis yra registruojamos ir blokinių įrenginių žemo lygio tvarkykles. Tokios tvarkyklės nuo joms atitinkamų įrenginių blokinių tvarkyklių skiriasi skaitymo ir rašymo operacijomis – duomenų apsikeitimas su įrenginiu vyksta tiesiogiai su proceso adresine erdve, aplenkiant buferinį mechanizmą. Tokiu būdu nėra dubliuojami duomenys ir nėra atliekamos papildomos skaitymo/rašymo operacijos, bet tuo pačiu ir nėra naudojamasi sisteminėmis duomenų kešavimo paslaugomis.

---

[6] Truputi kitokia schema yra naudojama tvarkyklių STREAMS posistemėje, kuri taip pat naudoja baitinį įrenginių adapterių.

Tokius žemo lygio interfeisus naudoja daugelis sisteminių pagalbinių programų, tokių kaip fsck(1M), tar(1) ar cpio(1). Taip pat juos gali naudoti ir įvairios vartotojo programos, tokios kaip duomenų bazių valdymo sistemos, kurios pačios organizuoja duomenų kešavimą.



5.8 pav. Įvairūs priėjimo prie blokinio įrenginio tipai.

5.8 paveiksle yra parodytas darbas su blokiniu įrenginiu per failų sistemą ir naudojant žemo lygio tvarkykles.

5.3.2 Baitinių įrenginių buferis

Duomenų įrenginiui perdavimas po vieną baitą yra neefektyvus. Dirbant tokiu būdu kiekvienas baitas yra pradžioje kopijuojamas į tvarkyklės adresinę erdvę, o vėliau, po t. t. laiko jis yra perduodamas įrenginiui. Jei įrenginys tuo metu yra užimtas, tai procesas yra nusiunčiamas “miegoti” ir įvyksta konteksto perjungimas.

Egzistuoja keletas metodų, kurie leidžia to išvengti, tačiau visi jie remiasi duomenų buferiu, kurio darbą organizuoja įrenginio tvarkyklė. Pateiksime du metodus, kurie skiriasi priklausomai nuo paties įrenginio:

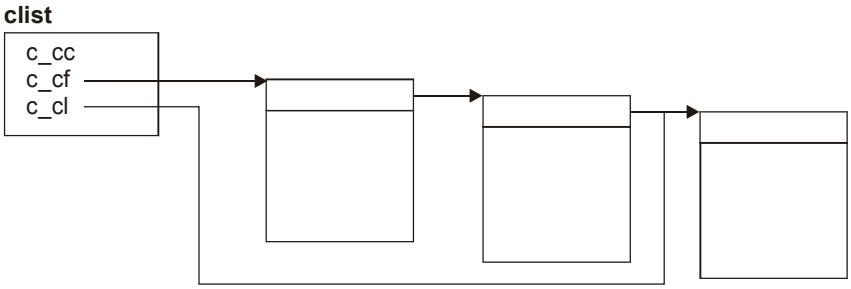
1. kai iš arba į įrenginį yra skaitoma arba rašoma generuojamas aparatūrinis pertraukimas, kurį apdoroja xxintr() funkcija. Ji surašo duomenis į buferį, kuriuos vėliau nuskaito funkcijos xxread() arba xxwrite().
2. jei įrenginys nepalaiko pertraukimų, juos galima emuliuoti su xxpoll() funkcija, kuri kas t. t. intervalą (dažniausiai, kas vieną taimerio tiką) apklausia įrenginius. Ji paprasčiausiai iškviečia xxintr() funkciją.

Simbolinių įrenginių duomenų buferizacija yra atliekama naudojant specialias duomenų struktūras, vadinamus clist (character list). Kiekviena clist struktūra turi sekančius argumentus:

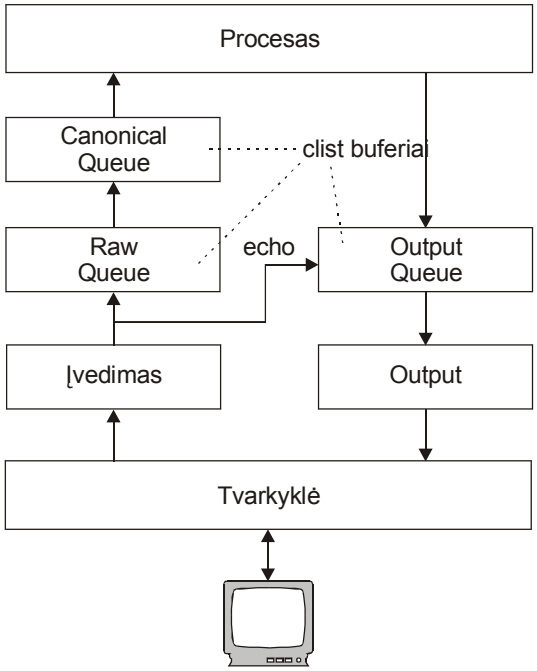
```
int                 c_cc;
struct cblock       *c_cf;
struct cblock       *c_cl;
```

Lauke c_cc yra saugomas cblock buferio dydis – simbolių skaičius, o laukai c_cf ir c_cl yra nuorodos į pirmąjį ir paskutinįjį buferio blokus atitinkamai, kurie yra išdėstyti į sąrašą.

Kiekviena cblock struktūra saugo t. t. kiekį simbolių. Kai buferis užsipildo, yra automatiškai išskiriamas naujas buferio blokas ir patalpinamas į sąrašo galą (5.9 pav.).



5.9 pav. `clist` buferio mechanizmas.



5.10 pav. `clist` buferių panaudojimas terminalo tvarkyklėse.

5.10 paveiksle yra pateikta terminalo įvedimo/išvedimo operacijų buferių mechanizmas.

5.4 Terminalų architektūra

Raidinis-skaitinis terminalas – nuoseklaus duomenų apsikeitimo įrenginys, su kuriuo operacinė sistema duomenimis keičiasi naudodama nuoseklų interfeisą, vadinamąją terminalinę liniją. Su kiekviena terminaline linija yra susietas vienas specialusis baitinio įrenginio failas `/dev/ttyxx`.

Terminalinės tvarkyklės atlieka tas pačias funkcijas, kaip ir kitos tvarkyklės, t.y. vykdo duomenų apsikeitimo operacijas tarp terminalo ir OS. Tačiau terminalai turi vieną išskirtinę savybę – jie suteikia vartotojui interfeisą su operacine sistema. Terminalo tvarkyklės realizuoja specialius komandų įvedimo ir išvedimo interpretacijos modulį, kuris vadinasi disciplinos linija (line discipline).

Egzistuoja du terminalinio įvedimo režimai:

1. kanoninis režimas, t.y. įvedimas iš terminalo yra vykdomas užbaigtomis eilutėmis ir
2. nekanoninis režimas, t.y. įvedimas nėra interpretuojamas.

Kanoniniame režime nestruktūriniai iš klaviatūros įvesti duomenys yra interpretuojami į eilutes, į kanoninę formą. Po to eilutė yra pasiunčiama procesui. Tarkim, kai yra įvedama komanda, gali būti klaidų, kurios yra trinamos specialiuoju klavišu (arba jų kombinacija). Terminalo tvarkyklė priima visą įvestą srautą, o tuo pačiu ir trynimo klavišą. Tvarkyklė sulaukia `Enter` simbolio, interpretuoja įvestą simbolių seką ir perkopijuoja ją to laukiančiam procesui. Tokiu režimu dirba komandiniai interpretatoriai (shell).

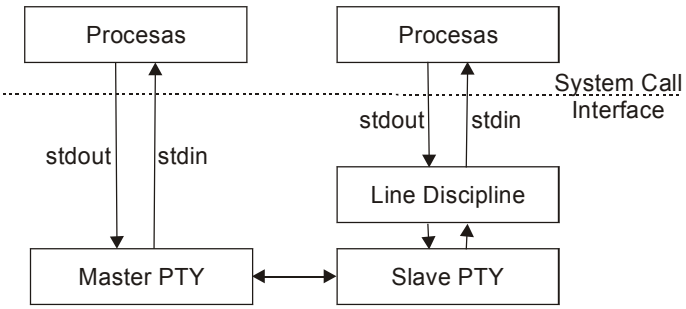
Į disciplinos linijos funkcijas taip pat įeina:

- eilutės suformavimas,
- trynimo pirmyn simbolių apdorojimas,
- trynimo atgal simbolių apdorojimas,
- gautų simbolių atvaizdavimas terminale,
- įvestų duomenų performatavimas, tarkim, `Tab` simbolio pakeitimas `Space` simbolių seka,
- specialiųjų simbolių pašalinimas.

Taip pat egzistuoja papildoma gaunamų ar siunčiamų simbolių apdorojimo galimybė naudojant atvaizdų lentelę. Tokią galimybę suteikia pagalbinė programa `mapchan`.

5.4.1 Pseudo-terminalai

Pseudo-terminalai yra specialūs įrenginiai, emuliuojantys standartinę terminalinę liniją. Jie primena paprastus IPC – kanalus. Tačiau skirtingai nei kanalai, pseudo-terminalai suteikia papildomą funkcionalumą, specifinį visoms terminalinėms linijoms. Schematinė pseudo-terminalo architektūra yra pateikta 5.11 paveiksle.



5.11 pav. Pseudo-terminalo procesų sąsaja.

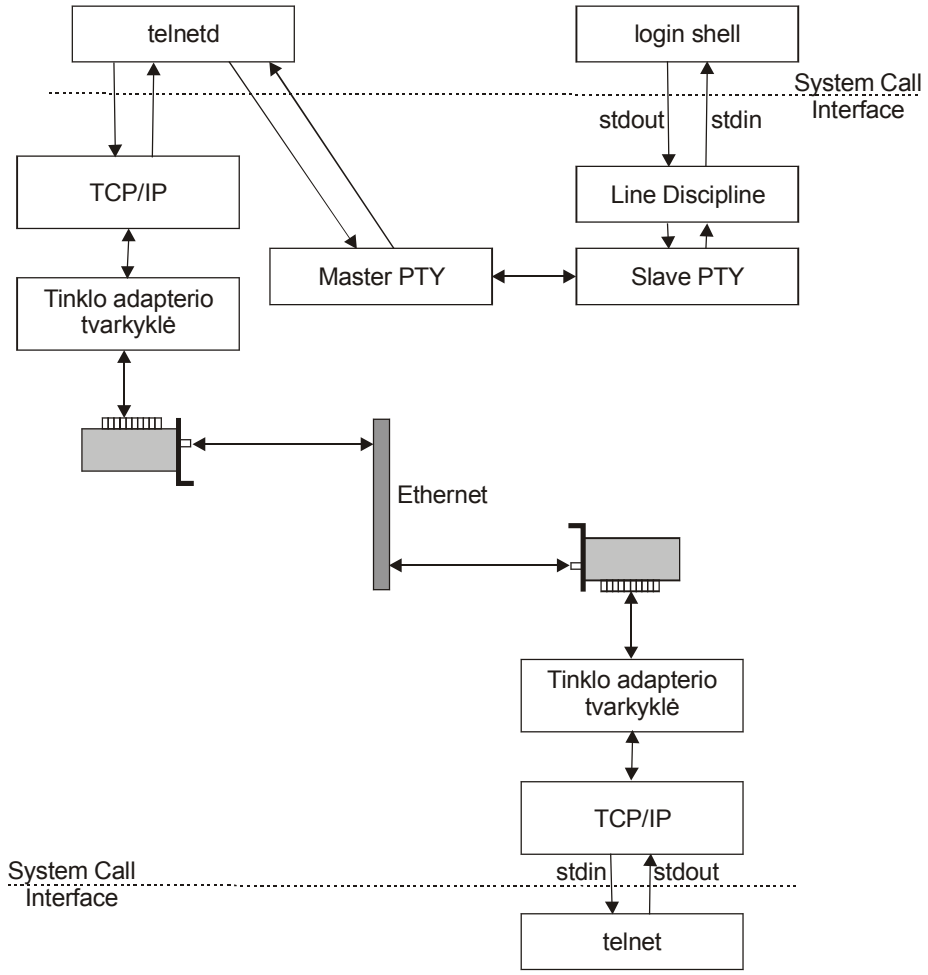
Pseudo-terminalus naudoja darbą su nutolusia operacine sistema realizuojančios programos, tokios kaip rlogin(1), telnet(1), ssh(2) ir xterm (X Window System). Kai vartotojas yra registruojamas sistemoje, pseudo-terminalas emuliuoja įprastą terminalinę liniją, todėl vartotojas nemato skirtumo tarp darbo su lakaliu kompiuteriu (terminalu) ir nutolusia mašina (pseudo-terminalu).

Pseudo-terminalą sudaro dvi skirtingos tvarkyklės. Viena jų atrodo kaip paprasta terminalinė tvarkyklė ir yra vadinama priklausomas įrenginys (slave). Kitas yra vadinamas pagrindiniu įrenginiu (master).

Priklausomasis įrenginys turi visas terminalo charakteristikas, procesas su juo gali susieti savo standartinius įvedimo, išvedimos ar klaidos duomenų srautus. Tačiau, jei tikrojo terminalo atveju duomenys yra betarpiškai atvaizduojami į fizinį įrenginį, terminalą, tai šiuo atveju, duomenys yra perduodami pagrindiniam pseudo-terminalo įrenginiui, taip kaip dirba kanalas. Tačiau skirtingai nei kanalas, priklausomam įrenginiui papildomą funkcionalumą suteikia disciplinos linija, kurio reikia programoms, tokioms kaip komandų interpretatoriai (shell) ir pan.

Tarkim, vartotojas paleidžia programą telnet(1) ir jos pagalba kreipiasi į nutolusią mašiną (serverį). Ši programa nusiunčia užklausą serveryje esančiam procesui telnetd(1), kuris, atlikęs reikiamus patikrinimus, paleidžia programą login(1). Tuo pačiu, standartiniai įvedimo, išvedimo ir klaidos duomenų srautai susiejami nes su terminaliniu failu, kaip yra daroma tikrame terminale su getty(1M) programos pagalba, o su priklausomu pseudo-įrenginiu. Pagrindinis pseudo-terminalo įrenginys yra susiejamas su telnetd(1) procesu. Programa login(1) paprašo vartotojo įvesti vardą ir slaptažodį, lygiai taip pat, kaip ir įėjus per getty(1M). login(1) net neįtaria, kad dirba su terminalo emuliatoriumi, o ne su tikrąja terminaline linija. Visa klientui skirta informacija patenka į telnetd(1) procesą ir yra perduodama per tinklo adapterį kliento telnet(1) procesui. Vėliau darbas yra organizuojamas lygiai taip pat kaip ir įprasto terminalo atveju: jei vartotojas ir slaptažodis buvo teisingi, login(1) programa paleidžia reikiamą komandų interpretatorių (login shell). Nors darbas bus organizuotas kaip ir su tikrais terminalais, bet kai kurie parametrai, tokie kaip perdavimo greitis ir pan. bus ignoruojami.

Paveiksle 5.12 paveiksle yra pateikta vartotojo darbo schema naudojant pseudo-terminalą.



5.12 pav. Darbas iš nutolusios mašinos naudojant `telnet(1)` programą ir pseudo-terminalus.

5.5 STREAMS posistemė

I/O posistemę STREAMS pirmasis 1984 metais aprašė D. M. Ritchie straipsnyje “A Stream Input-Output System” (At&T Bell Laboratories Technical Journal Vol. 63, No. 8, Oct. 1984). Praėjus dviems metams ji buvo realizuota komercinėje UNIX SVR3 versijoje.

Štai priežastys, kurios paskatino naujos I/O sistemos sukūrimą:

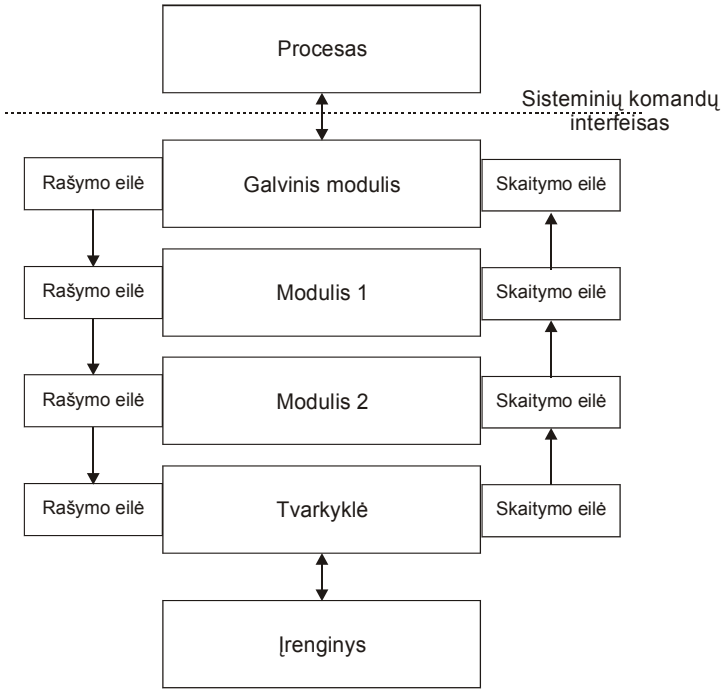
1. tradicinė I/O sistema dirba per tvarkykles pakankamai aukštame lygyje, tuo perduodama didžiąją dalį darbo tvarkyklei, kurios tik nedidelė kodo dalis priklauso nuo aparatūros. Kita tvarkyklių dalis gali būti visiškai vienoda dideliam įrenginių spektrui. Tokiu būdu, kai operacinė sistema palaiko vis daugiau įvairių įrenginių, be reikalo yra dubliuojamas kodas;
2. didėjant simbolinių įrenginių greičiams atsirado poreikis simbolinių duomenų buferizacijai. Buvo bandoma tai atlikti panaudojant aukščiau minėtą `clist` ar panašius mechanizmus, tačiau jie nėra pakankamai efektyvūs, nes yra realizuojami tvarkyklėje (neefektyviai valdoma atmintis);
3. atsirado poreikis palaikyti hierarchinius tinklo protokolus. Duomenų perdavimas tinkle vyksta paketais arba pranešimais, kur kiekvienas protokolo lygis atitinkamai apdoroja duomenis ir perduoda juos kitam lygiui. Kiekvienas iš lygių turi standartinius darbo su aukštesniu ir žemesniu lygiais interfeisus.Taip pat jie gali būti įvairūs, tarkim IP protokolas (3 ISO lygis) gali palaikyti kelis aukštesnio lygio protokolus, tokius kaip TCP ir UDP ir kelis žemesnius – Ethernet, Token Ring ir pan.

Siekiant patenkinti šiuos reikalavimus buvo sukurta STREAMS I/O sistema, kurioje duomenų apdorojimas yra realizuotas nepriklausomuose moduliuose, o pati tvarkyklė pasiekia reikiamą funkcionalumą surišdama reikiamus modulius.

5.5.1 STREAMS architektūra

STREAMS posistemė sukuria srautus – duomenų perdavimo kanalus tarp proceso ir įrenginio tvarkyklės[7]. Paveiksle 5.13 yra parodyta bendra STREAMS komunikacinio kanalo schema. Visas duomenų srautas yra realizuojamas branduolio atminties erdvėje, todėl ir visos jį aptarnaujančios funkcijos dirba sisteminiame kontekste. Tipinis srautas yra sudarytas iš galvinio modulio (head module), įrenginio tvarkyklės ir vieno ar daugiau tarpinių modulių. Galvinis modulis su procesu bendrauja per sisteminių komandų interfeisą. Tvarkyklė kitame srauto gale betarpiškai bendrauja su aparatūriniu įrenginiu arba pseudo-įrenginiu, kurio vaidmenį gali atlikti kitas srautas. Tarpiniai moduliai atlieka perduodamų duomenų apdorojimą.

[7] STREAMS įrenginių tvarkyklės yra kitokios, nei aukščiau nagrinėtos simbolinių įrenginių tvarkyklės.

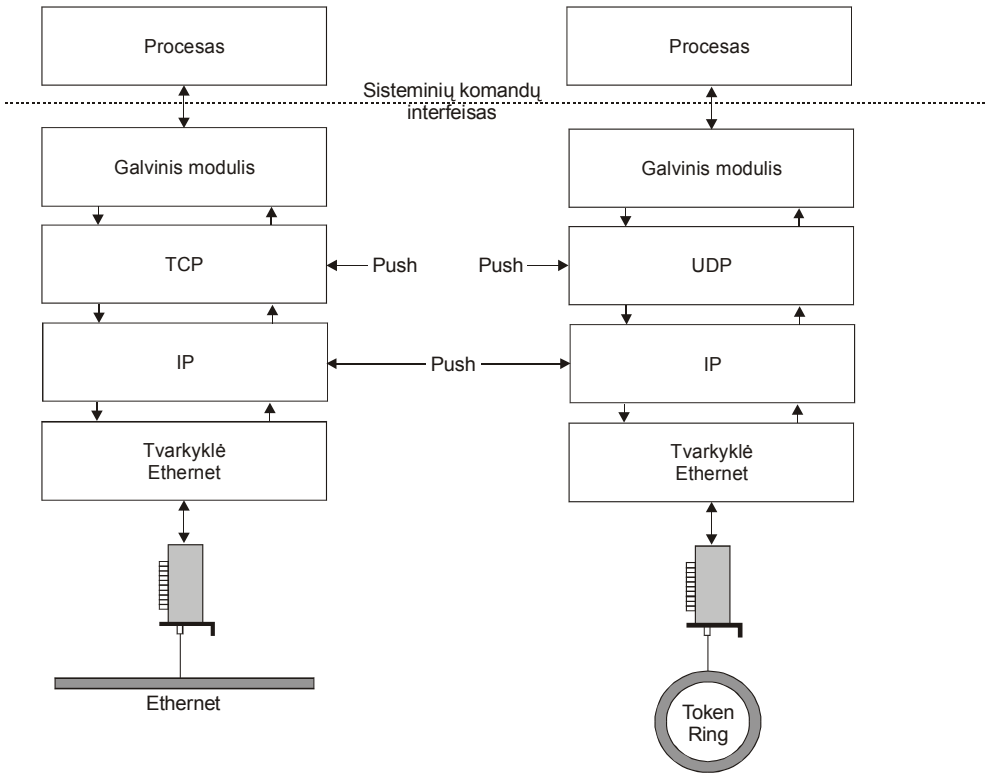


5.13 pav. Bazinė srauto architektūra.

Procesas su galviniu moduliu bendrauja naudodamas standartines sistemines komandas: `open(2)`, `close(2)`, `read(2)`, `write(2)` ir `ioctl(2)`. Papildomai gali būti naudojamos `poll(2)`, `putmsg(2)` ir `getmsg(2)` funkcijos. Duomenys sraute yra perduodami pranešimų pavidalu, kuriuose yra saugomi patys duomenys, pranešimo tipas ir valdančioji informacija. Kiekvienas modulis, tame tarpe ir galvinis bei tvarkyklė, turi dvi duomenų perdavimo eiles – skaitymui (read queue) ir rašymui (write queue). Modulis apdoroja pranešime esančią informaciją ir perduoda ją kitam moduliui: rašymo atveju perdavimas vyksta žemyn srautu (downstream), o skaitymo – aukštyn (upstream). Pranešimas gali būti perduodamas ir į porinę eilę (duoblestream), t.y. tam tikras modulis praneimą iš rašymo eilės gali permesti į skaitymo eilę, jį apdoroti ir perduoti aukštyn. Tokiu būdu yra realizuojamas lankstus mechanizmas, kuris užtikrina nepriklausomą nuo konkretaus srauto modulių darbą ir leidžia viename sraute naudoti įvairių autorių modulius.

STREAMS posistemė palaiko dinaminio modulių įterpimo į srautą (push) ir išmetimo iš srauto (pop) mechanizmus. Modulio įterpimas galimas tik betarpiškai žemiau galvinio modulio, su kuriuo, kaip ir su kitais moduliais žemyn srautu, vėliau yra suderinami eilių mechanizmai. Tokiu būdu naujai įterptas modulis pakeičia srauto funkcionalumą. Kai prireikia, modulis gali būti pašalinamas (pop) iš srauto.

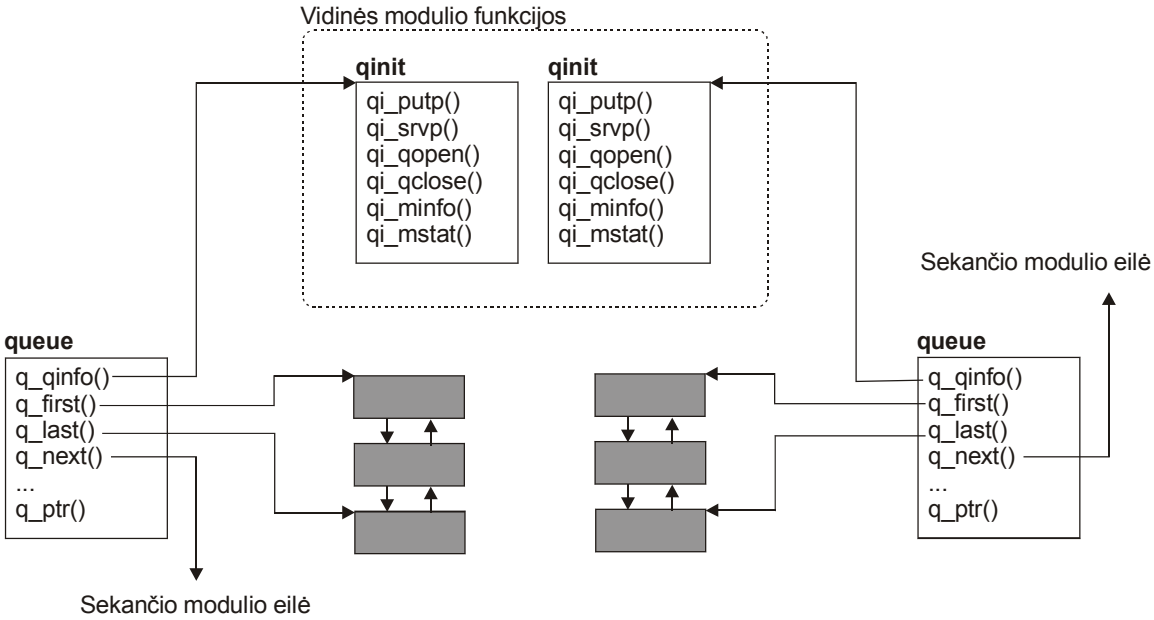
Paveiksle 5.14 yra parodyti du skirtingi srautai, suburti iš kelių standartinių protokolų, kuriuos realizuojantys moduliai – IP, TCP ir UDP gali būti sukurti vienų, o tvarkyklės kitų autorių. Pradžioje STREAMS sukuria elementarų srautą, sudarytą tik iš galvinio modulio ir tvarkyklės, o vėliau įterpia reikiamus modulius.



5.14 pav. Vienų ir tų pačių modulių panaudojimas skirtingų srautų sukūrimams.

5.5.2 Moduliai

Moduliai yra pagrindiniai STREAMS komponentai. Kiekvienas modulis yra sudarytas iš dviejų eilių – skaitymo ir rašymo, o taip pat duomenų apdorojimo it pranešimų perdavimo funkcijų. Modulio architektūra yra pavaizduota 5.15 paveiksle.



5.15 pav. STREAMS modulis.

Eiles realizuoja queue struktūros, kurių pagrindiniai laukai yra sekantys:

q_qinfo	Rodyklė į qinit struktūrą, kurioje yra aprašytos pranešimus apdorijančios funkcijos.
q_first, q_last	Nuorodos į eilėje esančius atitinkamus pranešimus.
q_next	Nuoroda į sekančio modulio eilę aukštyn arba žemyn srautu.
q_ptr	Nuoroda į eilės vidinius duomenis.

Duomenų perdavimą viršun ar žemyn srautu atlieka modulio funkcijos, saugomos qinit struktūroje. Modulis privalo realizuoti keturias eilės apdorojimo procedūras: xxput(), xxservice(), xxopen(), xxclose(), kur xx – tvarkyklės prefiksas. Šios funkcijos yra adresuojamos rodyklėmis (*qi_putp)(), (*qi_srvp)(), (*qi_qopen)() ir (*qi_qclose)(). Funkcija xxopen() yra iškviečiama tada, kai procesas sukuria srautą arba kai modulis yra įterpiamas į jau egzistuojantį srautą. xxput() apdoroja eilėje esančius pranešimus. Jei xxput() negali perduoti pranešimo sekančio modulio eilei (tarkim, kai ji yra perpildyta), jis yra patalpinamas į savo eilę. Sistemos branduolys periodiškai kiekvienma aktyvios eilės moduliui iškviečia xxservice() procedūrą, kuri pagal galimybę perduoda atidėtus pranešimus. Jei modulis neturi realizuotos xxservice() procedūros, xxput() privalo pranešimą perduoti iš karto.

5.5.3 Pranešimai

STREAMS posistemėje visi duomenys yra perduodami pranešimų pavidalu. Taip pat pranešimai yra naudojami valdymo informacijos perdavimui tarp modulių (taip pat galvinio modulio – proceso ir modulio – tvarkyklės). Tokiu būdu pranešimas yra vienintelė informacijos perdavimo priemonė STREAMS posistemėje.

Pranešimai yra aprašomi dvejomis duomenų struktūromis: pranešimo msgb (message block) ir duomenų datab (data block) blokų antraštėmis. Pranešimo antraštę sudaro sekantys laukai:

b_next, b_prev	Rodyklės, naudojamos pranešimų sąrašo formavimui eilėse.
b_cont	Rodyklė, naudojama vieno pranešimo atskiroms dalims surišti.
b_datap	Rodyklė į duomenų bloko antraštė datab.
b_rptr, b_wptr	Rodyklės į duomenų pradžią ir pabaigą duomenų buferyje.

datab struktūra yra naudojama buferio aprašymui ir turi sekančius laukus:

db_base	Buferio pradžios adresas
db_lim	Buferio pabaigos adresas + 1 (buferio dydis = db_lim - -db_base)
db_type	Pranešimo tipas
db_ref	Duomenų bloką adresuojančių pranešimo antraščių skaičius

Laukas b_cont pranešimo antraštėje leidžia sujungti kelis duomenų blokus į vieną pranešimą. Ši savybė yra ypač efektyviai naudojama realizuojant duomenų perdavimo tinklu protokolus, nes šie protokolai yra hierarchiniai. Kiekvienas sraute esantis modulis (realizuojantis tam tikrą protokolo lygį) prideda savo informaciją. Kadangi aukščiau esantys moduliai nieko nežino apie žemesniųjų modulių architektūrą ir nežino kokio dydžio buferį reikia realizuoti. Todėl kiekvienas modulis prideda valdančiąją informaciją atskirų dalių pavidalu panaudojant b_cont. Taip yra realizuojama duomenų inkapsuliacija.

db_ref laukas struktūroje leidžia keliems pranešimams naudoti tą patį duomenų bloką. Tokiu būdu yra atliekamas virtualus pranešimo kopijavimas, kur kiekviena kopija yra apdorojama atskirai. Paprastai toks buferis yra naudojamas tik skaitymui, nors pati STREAMS sistema jo neblokuoja.

Kaip tokios architektūros privalumų pavyzdys gali būti pateiktas TCP protokolas, kuris, būdamas patikimu, reikalauja gavėjo patvirtinimo apie sėkmingą paketo perdavimą. Tokiu būdu TCP modulis privalo saugoti visų nepatvirtintų pranešimų kopijas. Vietoj to, kad darytų fizines jų kopijas, jis atlieka virtualų kopijavimą, t.y. IP moduliui (protokolui) perduoda naujai sukurtą pranešimo antraštę, rodančią į tą patį duomenų bloką. Tvarkyklei išsiuntus vieną

pranešimų, IP protokolas savo kopiją sunaikiną (`db_ref` laukas sumažinamas vienetu), o TCP protokolas jį saugo iki patvirtinimo.

5.5.3.1 Pranešimų tipai

Kiekvienas pranešimas priklauso t.t. tipui, tačiau visi yra skirstomi į dvi kategorijas – paprasti ir prioritetiniai. Nuo to priklauso, kokia tvarka šie pranešimai bus apdorojami modulių eilėse. Paprastųjų pranešimų tipai yra pateikti 5.3 lentelėje. Paprastiesiems pranešimams galioja įprasti srauto valdymo mechanizmai, tačiau jie negalioja prioritetiniams pranešimams. Jų perdavimas nepriklauso nuo to, ar sekančio modulio pranešimų eilė yra perpildyta, ar ne. Prioritetiniai pranešimai yra pateikti 5.4 lentelėje.

5.3 lentelė. Žemo prioriteto STREAMS pranešimų tipai.

Tipas	**Aprašymas**
`M_DATA`	Paprasti duomenys, dažniausiai perduodami su sisteminėmis komandomis read(2) ir write(2).
`M_PROTO`	Valdančiosios informacijos pranešimas. Dažnai kartu yra perduodama ir keli `M_DATA` blokai.
`M_BREAK`	Siunčiamas tvarkyklei generuojant komandą `break`.
`M_PASSFP`	Naudojamas STREAMS kanaluose (pipe), kai reikia perduoti nuorodą į failą nuo vieno kanalo gali į kitą.
`M_SIG`	Moduliai ir tvarkyklės generuoja ir juo aukštyn procesui perduoda signalą.
`M_DELAY`	Siunčiamas įrenginio tvarkyklei ir nurodo laikinai pristabdyti duomenų perdavimą. Dažniausiai naudojama tada, kai norima išvengti įrenginių buferio persipildymo.
`M_CTL`	Naudojami vieno srauto rėmuose keičiantis informaciją tarp modulių. Juos galvinis modulis automatiškai sunaikina.
`M_IOCTL`	Šiuo pranešimu galvinis modulis atsako į proceso ioctl(2) komandas, siekiant sukurti multipleksinį srautą.
`M_SETOPTS`	Naudojamas konfigūruojant galvinį modulį.
`M_RSE`	Rezervuotas. Moduliai ir tvarkyklės jį turi perduoti nekeitę.

5.4 lentelė. Aukšto prioriteto STREAMS pranešimų tipai.

Tipas	**Aprašymas**
`M_COPYIN`	Perduodama aukštyn galviniam moduliui su nuoroda atlikti duomenų priėmimą iš proceso pagal komandą ioctl(2).
`M_COPYOUT`	Perduodama aukštyn galviniam moduliui su nuoroda perduoti duomenis procesui pagal komandą ioctl(2).
`M_ERROR`	Perduodama galviniam moduliui ir pranešama apie klaidą srauto moduliuose.
`M_FLUSH`	Gavęs šį pranešimą modulis privalo išvalyti eilę (skaitymo, rašymo arba abi).
`M_HANGUP`	Perduodama galviniam moduliui ir pranešama, kad tvarkyklė nebegali atlikti duomenų perdavimo, dažniausiai dėl ryšio linijos problemų.
`M_IOCACK`	Buvusio `M_IOCTL` pranešimo patvirtinimas.
`M_IOCNAK`	Jei ioctl(2) komanda nepavyko, galviniam moduliui grąžinamas šis pranešimas, o pastarasis generuoja klaidą ir perduoda ją procesui.
`M_PCPROTO`	Aukšto prioriteto `M_PROTO` pranešimas.
`M_PCSIG`	Aukšto prioriteto `M_SIG` pranešimas.
`M_PCRSE`	Rezervuota STREAMS posistemės vidiniams reikalams.
`M_READ`	Perduodama žemyn srautu, kai procesas iškviečia komandą read(2), o galvinis modulis neturi duomenų.
`M_STOP`	Liepia nedelsiant sustabdyti perdavimą.
`M_START`	Liepia tęsti perdavimą, po sustabdymo su pranešimu `M_STOP`.

5.5.4 Duomenų perdavimas

Procesas iškviesdamas sistemines komandas write(2) arba putmsg(2), kurios betarpiškai bendrauja su srauto galviniu moduliu. Pastarasis suformuoja pranešimą, nukopijuoja į jį iš proceso adresinės erdvės duomenis ir perduoda pranešimą žemyn srautu. Srauto gale pranešimą gauna tvarkyklė, kuri savo ruožtu atlieka reikalingas operacijas su įrenginiu. Kai įrenginio tvarkyklė iš įrenginio gauna duomenis, ji taip pat suformuoja pranešimą ir siunčia jį aukštyn srautu. Procesas naudodamas sistemines komandas read(2) ir getmsg(2) juos pasiima. Jei galviniame modulyje duomenų nėra, procesas yra užmigdomas.

Duomenų perdavimas sraute vyksta asinchroniškai ir negali būti blokuojamas procesas. Jis gali būti blokuojamas tik vykstant perdavimui tarp proceso ir galvinio modulio. Jei procedūra xxput() negali perduoti duomenų žemyn/aukštyn srautu, modulis patalpina pranešimą į savo vidinę eilę, kurios apdorojimu jau rūpinasi xxservice(). xxservice() yra periodiškai iškviečiama taip pat sisteminiame kontekste.

Pranešimo perdavimas kitam moduliui vykdomas iškviečiant funkciją

```
#include <sys/stream.h>
#include <sys/ddi.h>
int putnext(queue_t *q, mblk_t *mp);
```

, kur `q` – sekančio modulio eilė, o `mp` – pranešimas. Ši funkcija iškviečia modulio procedūrą xxput(); betarpiškas iškvietimas nėra leidžiamas, nes pakistų modulių pernešamumas.

Kai procedūra xxput() negali priimti pranešimo yra iškviečiama procedūra

```
#include <sys/stream.h>
int putq(queue_t *q, mblk_t *mp);
```

, kuri patalpina pranešimą į eilę `q` ir taip pat įdeda šios eilės rodyklę į sąrašą eilių, kurias reikia apdoroti. Tokioms eilėms branduolys automatiškai iškviečia xxservice(). Pastarąją iškviečia branduolio funkcija runqueues(), kuri suveikia dviem atvejais:

- kai koks nors vykdo I/O operaciją ir
- kai bet koks procesas yra pervedamas iš branduolio į vartotojo režimą.

Eilių aptarnavimas planuojamas visai STREAMS sistemai, o kiekvienam procesui/srautui atskirai.

Kiekvienai eilei yra nustatomos dvi vaterlinijos: viršutinė ir apatinė. Kai pranešimų skaičius eilėje pasiekia viršutinę vaterliniją, eilė laikoma perpildyta. Ji tokia lieka, kol pranešimų eilėje skaičius nepasiekia žemosios vaterlinijos.

6 Kompiuterinis tinklas

Kaip viena iš UNIX sistemų sėkmės ir ilgaamžiškumo priežasčių, buvo tai, kad jose vienose pirmųjų buvo realizuotas kompiuterinis tinklas.

Nors šiuolaikinės UNIX sistemos palaiko daug tinklo protokolų, tačiau šio kurso metu mes detaliau apžvelgsime plačiausiai paplitusią TCP/IP protokolų šeimą. Šie protokolai buvo ilgai tobulinti, kol tapo vieno XX amžiaus fenomenų – Interneto širdimi.

6.1 TCP/IP protokolų šeima

Protokolų šeimos pavadinimą sudaro dviejų protokolų pavadinimai – TCP ir IP, tačiau tai nereiškia, kad tik jie du ir sudaro visą šeimą.

Defence Advanced Research Projects Agency (DARPA) 1969 metais pradėjo finansuoti eksperimentinio kompiuterinio tinklo projektą (packet switching network). Šis tinklas, vadinamasis ARPANET, pradžioje jungė keturis mazgus: Stanford Research Institute, University of California at Santa Barbara, University of California at Los Angeles ir University of Utah. Mini-kompiuteriai Honeywell 316 (Interface Message Processor, IMP) atliko komunikatorių rolę.

ARPANET bandymai buvo sėkmingi, todėl daugelis JAV universitetų kreipėsi į Nacional Science Foundation (NSF) prašydami finansuoti tokio tinklo sukūrimą moksliniams tikslams. 1981 metais buvo sukurtas CSNET (Computer Science Network).

1984 metais ARPANET skilo į du tinklus: MILNET – skirtas kariškių ir ARPANET – taikiems tikslams.

1986 metais NSF finansavo tinklo stuburo su 56 Kbit/s pralaidumu įkūrimą, kuris turėjo jungti 6 JAV superkompiuterių centrus. Šis tinklas pragyvavo iki 1995 metų ir buvo pagrindinė Interneto magisralė. Per tą laiką jos pralaidumas išaugo iki 45 Mbit/s.

Greitai plečiantis NSFNET tinklui JAV vyriausybė priėmė sprendimą nutraukti ARPANET eksploataciją, tačiau vystant būtent šį tinklą buvo sukurti komunikaciniai protokolai – duomenų perdavimui būtinas bendras taisyklių ir duomenų struktūrų rinkinys, tarp jų ir TCP/IP protokolai, kurių realizaciją UNIX operacinėse sistemose Kalifornijos universitete DARPA pradėjo finansuoti 1983 metais.

Pradžioje buvo sukurtas protokolas, kuris reikalavo absoliučiai patikimo ryšio kanalo, vadinamasis Network Control Program (NCP). Tačiau nei tada, kai kanalai buvo tikrai siauri, nei dabar, visiško patikimumo negalime garantuoti. Atliekant tyrimus, kurių metu reikėjo sukurti patikimą duomenų perdavimą nepatikimais tinklais, buvo sukurtas Internet Protokol (IP), kuris rupinasi paketų baziniu perdavimu heterogeninėse aplinkose ir Transmission Control Protocol (TCP), kuris rūpinasi ryšio seansu ir perduodamų duomenų kontrole. Šie protokolai tapo JAV Gynybos ministerijos standartais – MIL-STD-1777 ir MIL-STD-1778 atitinkamai.

6.1.1 TCP/IP architektūra

TCP/IP protokolų šeimos architektūra remiasi požiūriu, kad tinklo infrastruktūrą sudaro trys objektai: procesai, kompiuteriai (hosts) ir tinklai. Pagrindiniai mazgai yra procesai, tarp kurių ir vyksta apsikeitimas duomenimis. Procesų vykdymas vyksta tam tikruose kompiuteriuose, sujungtuose į tinklą (hosts). Informacijos perdavimas vyksta tinklais.

Šios sąvokos perša paprastą mintį: norint perduoti paketą reikiamam procesui, pradžioje (1) reikia jį perduoti reikiamam kompiuteriui, o vėliau – (2) reikiamam procesui. Šias dvi fazes galima atlikti nepriklausomai. Remiantis darbų pasiskirstymo idėja buvo išskirti keturi lygiai:

- Aplikacijos/proceso sluoksnis (Application/process layer)
- Transportinis sluoksnis (Host-to-host layer)
- Interneto sluoksnis (Internet layer)
- Tinklo sluoksnis (Network Interface layer)

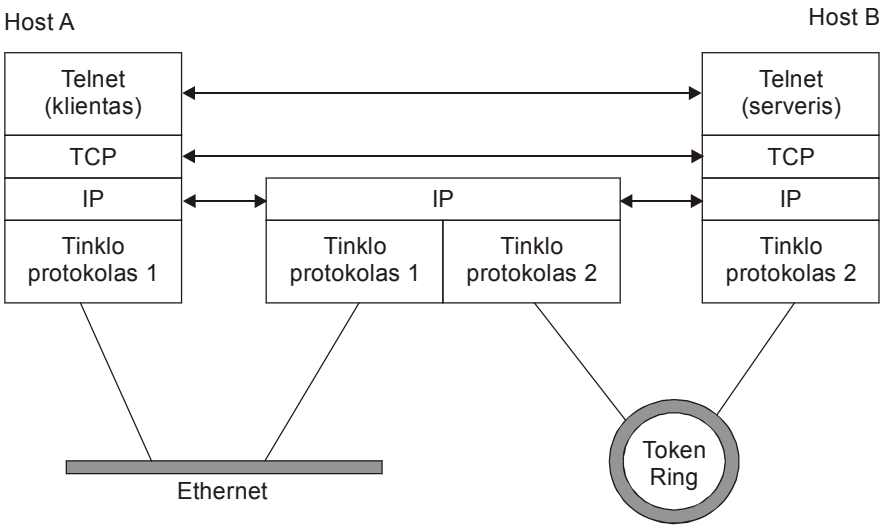
Tinklo interfeiso lygį sudaro protokolai, tiesiogiai perduodantys duomenis fiziniais tinklais. Šiuos protokolus privalo palaikyti visi aktyvūs tinklo mazgai, tokie kaip koncentratoriai (switches). Tokiais protokolais yra Ethernet, IEEE802.x, Slip, PPP ir pan. Formaliai šis sluoksnis nėra nagrinėjamas TCP/IP šeimoje, tačiau Interneto standartai apibrėžia, kaip šiame lygyje turi būti vykdomas duomenų perdavimas.

Interneto lygmens protokolai atlieka paketų perdavimą tarp tinklo kompiuterių (hosts). Vienas pagrindinių protokolu realizuojamų uždavinių – perdavimo maršruto parinkimas arba maršrutizacija. Tinklo elementai, perduodantys paketus iš vieno tinklo į kitą yra vadinami šliuzai (gateways) arba maršrutizatoriai (routers). Jie turi kelis tinklo interfeisus, prijungtus prie skirtingų tinklų. Pagrindinis šio sluoksnio atstovas – protokolas IP.

Transportinio lygmens protokolai atlieka duomenų perdavimą procesams. Taip pat jie atlieka ir kitas naudingas užduotis, tokias kaip perdavimo garantavimas, virtualaus kanalo sukūrimas ir pan. Šio sluoksnio atstovais yra TCP ir UDP protokolai.

<table>
<tr><td>FTP
RFC 959</td><td>SMTP
RFC 821</td><td>Telnet
RFC 854</td><td>NFS
RFC 1094</td><td>SNMP
RFC 1157</td><td>Aplikacijos lygis</td></tr>
<tr><td colspan="3">TCP
RFC 793</td><td colspan="2">UDP
RFC 768</td><td>Transportinis lygis</td></tr>
<tr><td>Address Resoliution
ARP RFC 826
RARP RFC 903</td><td colspan="2">IP
RFC 791</td><td colspan="2">ICMP
RFC 792</td><td>Interneto lygis</td></tr>
<tr><td colspan="5">Ethernet, Token Ring, FDDI, serial, ATM</td><td rowspan="2">Tinklo interfeisų lygis</td></tr>
<tr><td colspan="5">Vyta pora, koksalinis kabelis, optinis kabelis, bevielis ir palydovinis ryšys</td></tr>
</table>

6.1 pav. Hierarchinė TCP/IP protokolų schema.



6.2 pav. Principinė duomenų perdavimo skirtingais tinklais schema.

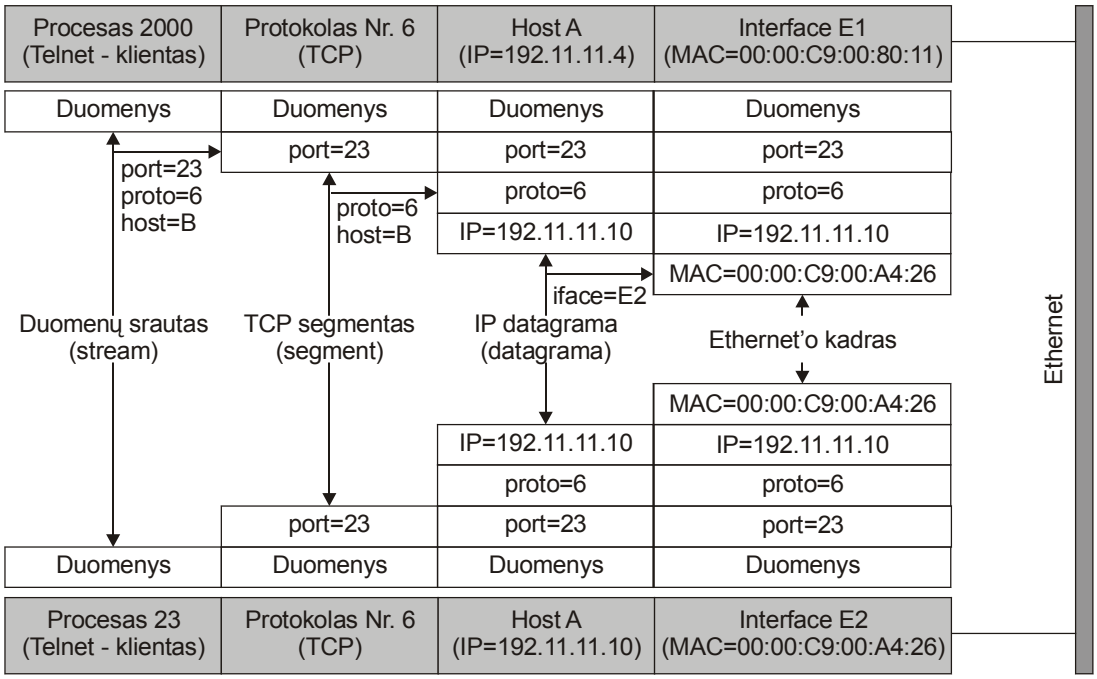
Aplikacijų lygmens protokolai atlieka programų lygmens užduotis pačiuose procesuose – servisuose. Tokių protokolų pavyzdžiais gali būti Telnet, FTP, SMTP ir kiti protokolai.

Paveiksle 6.1 yra pateikta hierarchinė TCP/IP protokolų šeimos schema. 6.2 paveiksle pateikiama principinė duomenų perdavimo skirtingais tinklais schema.

Norint perduoti paketą reikiamam tinklo mazgui yra būtina jį identifikuoti, todėl visi tinklo mazgai privalo būti vienareikšmiškai identifikuojami. Todėl yra naudojama sekanti adresavimo schema (pagal lygius):

- tinklo interfeisų lygis – MAC adresas (lokaliame tinkle),
- Interneto lygis – IP adresas (Internete) ir
- transporto lygis – proceso numeris (port, kompiuteryje).

Norint perduoti duomenis tinklu yra būtina žinoti gavėjo kompiuterio adresą (IP) ir proceso numerį (port). MAC adresas kinta priklausomai nuo fizinio įrenginio ir yra nustatomas automatiškai kiekviename perdavimo etape (hop) tarp maršrutizatorių. Adresavimo informacija yra įterpiama kiekvieno sluoksnio apdorojimo metu – suformuojamas protokolo duomenų vienetas (Protocol Data Unit, PDU), sudarytas iš duomenų, gautų iš aukštesnio lygmens protokolo ir iš antraštės (antraščių), kuriose yra saugoma valdančioji informacija. Antraštes paketo valdymui ir/arba perdavimui naudoja tokio pat lygmenis protokolas kitame komunikacinio kanalo gale. Paveiksle 6.3 yra demonstruojama duomenų inkapsuliacija paketo perdavimo metu.



6.3 pav. TCP/IP protokolai: duomenų apdorojimas ir inkapsuliacija.

6.1.2 OSI modelis

Hierarchinė TCP/IP protokolų struktūra leidžia atlikti nepriklausomą kiekvieno iš sluoksnių realizaciją. Intensyviai vystantis komutuojamo tinklo technologijai, kilo būtinybė standartizuoti apsikeitimo duomenimis procesą.

Tokį bendrą modelį 1983 metais patvirtino tarptautinė standartizacijos organizacija (International Organization for Standartization, ISO) ir pavadino jį Atvirų sistemų bendravimo modelis (Open Systems Interconnection, OSI). Šis modelis yra bendrojo pobūdžio standartas, skirtas įvairiarūšių kompiuterių bendravimui heterogeninėje aplinkoje.

OSI modelį sudaro septyni sluoksniai, kurie yra pateikti ir aiškinami 6.1 lentelėje.

6.1 lentelė. OSI modelio sluoksniai.

Pavadinimas	**Aprašymas**
Application Layer	Vartotojiškas priėjimo prie duomenų interfeisas.
Presentation Layer	Suteikia nepriklausomumą nuo duomenų formato.
Session Layer	Tinklo ir vartotojo programos interfeisas.
Transport Layer	Sukuria skaidrų duomenų perdavimą tarp tinklo mazgų. Kontroliuoja duomenų srautą ir taiso klaidas.
Network Layer	Aukšto ir žemo lygmens protokolų interfeisas. Atsako už sujungimo sukūrimą, palaikymą ir pabaigą.
Data Link Layer	Kontroliuoja duomenų perdavimą fiziniu kanalu. Atlieka duomenų sinchronizaciją, klaidų taisymą ir srauto valdymą.
Physical Layer	Atsako už nestruktūrinio duomenų srauto perdavimą fizinėje aplinkoje. Nusako fizines duomenų perdavimo kanalo charakteristikas.

Šis modelis yra daugiau teorinis ir praktiškai niekada nėra įgyvendintas. Todėl mes daugiau jo nenagrinėsime ir pereisime prie realesnių dalykų, tokių kaip pagrindiniai TCP/IP šeimos protokolai.

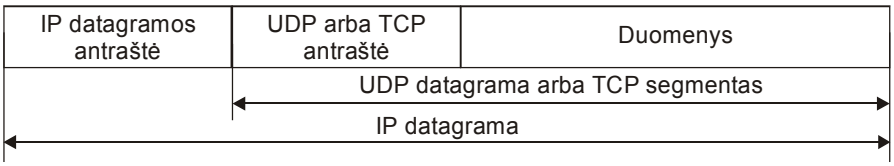
6.2 IP protokolas

Duomenų perdavimo tinklu protokolas IP (Internet Protocol) vienu metu persiunčia duomenų fragmentą, vadinamą datagrama. Protokole nėra realizuotos tokios funkcijos, kaip perdavimo patvirtinimas, kontrolė, nuoseklumo išsaugojimas ir pan., todėl šis protokolas yra vadinamas nepatikimu. Patikimumą užtikrinančios funkcijos yra realizuojamos aukštesniais protokolais.

IP realizuoja tris funkcijas:

- adresavimas,
- fragmentacija ir
- maršrutizacija.

IP protokolas kiekvieną datagramą (6.4 pav.) apdoroja kaip atskirą ir nepriklausomą nuo kitų perduodamų datagramų objektą. Datagramos nenaudoja jokių virtualių kanalų ar kitų loginių perdavimo kelių.



6.4 pav. IP datagrama.

IP modulis perduoda datagramą gavėjui adresuojamam IP adresu, kuris yra nurodomas datagramos antraštėje. Perdavimo kelio parinkimas yra vadinamas maršrutizacija.

Kartais IP modulis atlieka perduodamų duomenų fragmentaciją, nes įvairūs tinklai naudoja nevienodo dydžio kadrus, pavyzdžiui FDDI kadras yra iki 4470 oktetų, o Ethernet – iki 1500 oktetų.

6.2.1 IP datgramų adresavimas

Kiekvienas IP adresas yra sudarytas iš dviejų dalių – tinklo adreso ir kompiuterio adreso tinkle. Egzistuoja penki galimų adresų formatai, kuriuos atskirti galima pagal pirmuosius tris adreso bitus. Nuo klasės A iki D adresų formatai pateikti 6.2 lentelėje.

6.2 lentelė. IP adresų klasės.

Klasė	1-as baitas	Formatas (bitais)	Komentaras
A	1 – 126	0[7 net][24 host]	Rezervuota ir labai ankstyvi tinklai
B	128 – 191	10[14 net][16 host]	Labai dideli potinkliai
C	192 – 223	110[21 net][8 host]	Įprasti adresai
D	224 – 239	1110[28 combined]	Kol kas nenaudojami adresai
E	240 – 254	11110[27 combined]	Eksperimentiniai adresai

Tinkle (Internete) esantys kompiuteriai (tinklai) turi sudaryti vientisą adresinę erdvę. Visų tinklo kompiuterių (host'ų) adresai privalo būti unikalūs. Tai praktikoje yra pasiekiama naudojant hierarchiją, paslėptą baziniame adreso formate.

A klasės adresai 7 bitus naudoja tinklo adresavimui ir tokiu būdu tinklų kiekis yra apribojamas iki 126[8]. Tokie adresai buvo naudojami ARPANET tinkle. Tokio tipo adresus naudoja tik labai ankstyvi tinklai ir jie jau senokai nėra išduodami.

Unikalių B klasės tinklų yra žymiai daugiau – 16 382, nes šiuose adresuose tam skirta 14 bitų. Pastaruoju metu ir šie adresai nėra išduodami.

[8] Šiaip 7 bitai įgalintų adresuoti 128 mazgus, bet 0 ir 127 yra rezervuoti.

Šiuo metu yra išduodami C klasės adresai. Šios klasės tinklų gali būti iki 2 097 150. Kiekviename tokio tipo tinkle gali būti tik iki 256 kompiuterių (host'ų).

Greitas Interneto augimas 80-ajame dešimtmetyje sukėlė adresų krizę. To priežastimi tapo lokalių tinklų populiarumas, pavyzdžiui jei organizacija turi keturis kompiuterius ir rezervuoja vieną C klasės tinklo adresą – 250 adresų prapuola. Siekiant efektyviau paskirstyti IP adresus buvo pasiūlyta papildoma adresų hierarchinė sistema. Dabar kompiuterio adresas gali būti perskeltas į dvi dalis – potinklo (subnetwork) ir kompiuterio adresus. Potinklis IP protokolo atžvilgiu yra atskiras tinklas ir tarp dviejų potinklių reikia statyti maršrutizatorių.

Potinklio ir kompiuterio adresus viename IP adrese atskiria vadinamoji potinklio raktas (netmask), kurį, vienetinėje išraiškoje, sudaro vienetukai – potinklio adresas ir nuliukai – kompiuterio potinklyje adresas. IP protokolo modulis su raktu ir konkrečiu adresu atlieka loginę operaciją AND ir nustato, ar datagrama yra skirta duotajam kompiuteriui, ar ji skirta visam tinklui, ar ją reikia perduoti kitam šliuzui (gateway), tolimesniam perdavimui.

Išplėstas adresavimas vadinamas CIDR (Classless Inter-Domain Routing) ir kartu su NAT (Network Address Translation) technologija yra plačiai naudojamas. CIDR adreso formatas yra pavyzdžiui toks:

```
IP (CIDR):    128.138.243.100/26
Mask bits:    11111111.11111111.11111111.11000000
Mask bytes:   255.255.255.192
Adress bits:  10000000.10001010.11110011.01100100
Network:      128.138.243.64
Broadcast:    128.138.243.127
First Host:   128.138.243.65
Last Host:    128.138.243.126
Total Hosts:  62
PTR:          100.243.138.128.in-addr.arpa
Address (hex):      808AF364
```

6.3 lentelėje yra pateikiamos tinklo konfigūracijos įvairaus ilgio potinklo raktams.

6.3 lentelė. Tinklo konfigūracijos įvairaus ilgio potinklo raktams.

Ilgis	Kompiuterio adreso bitų skaičius	Kompiuterių skaičius tinkle[9]	Potinklio raktas (dec)	Potinklio raktas (hex)
/20	12	4094	255.255.240.0	0xFFFFF000
/21	11	2046	255.255.248.0	0xFFFFF800
/22	10	1022	255.255.252.0	0xFFFFFC00
/23	9	510	255.255.254.0	0xFFFFFE00
/24	8	254	255.255.255.0	0xFFFFFF00
/25	7	126	255.255.255.128	0xFFFFFF80
/26	6	62	255.255.255.192	0xFFFFFFC0
/27	5	30	255.255.255.224	0xFFFFFFE0
/28	4	14	255.255.255.240	0xFFFFFFF0
/29	3	6	255.255.255.248	0xFFFFFFF8
/30	2	2	252	0xFFFFFFFC

Dabar daugelis kompanijų ir organizacijų naudoja vidinius arba privačius IP adresus. Jų principas yra toks pat kaip ir potinklių adresacijos, tačiau šiuos IP adresus kompanijos pasirenka pačios, o šliuzo mazge stato adresus transliuojančias programas – NAT (Network Address Translation).

Jei kompiuteris arba šliuzas nežino potinklio rakto, jis formuoja ICMP (Internet Control Message Protokol) pranešimą ADDRESS MASK REQUEST ir pasiunčia jį, laukdamas ICMP ADDRESS MASK REPLY pranešimo iš grėtimo šliuzo.

[9] Kompiuterių skaičius tinkle yra 2×[bitų skaičius]-2, kur –2 yra du rezervuoti adresai, kur visi 0 ir visi 1.

6.4 lentelėje yra pateikti specialieji IP adresai, kurių negalima priskirti kompiuteriams.

6.4 lentelė. Specialieji IP adresai.

IP adresas[10]	Paaiškinimas
0.0.0.0	Duotasis kompiuteris duotajame tinkle
0.0.0.5	Tam tikras kompiuteris duotajame tinkle
255.255.255.255	Grupinis visų duotojo potinklio kompiuterių adresas
128.138.243.255	128.138.243.0 potinklio visų kompiuterių adresas
127.0.0.1	Vidinio loginio kompiuterio adresas (localhost)

6.3 Transportiniai protokolai

Transporto lygmens protokolai dirba išimtinai kompiuteryje ir yra datagramų siuntėjai ar priėmėjais. Šio lygmens protokolai nėra skaitomi ir apdorojami šliuzuose (gateway).

Dažniausiai viename kompiuteryje dirba ne vienas procesas, kuris yra pasiruošęs priimti datagramas. Todėl yra būtina identifikuoti procesą kompiuteryje, kuriam yra skirta datagama.

Kaip jau buvo minėta kiekvienas tinklo protokolas turi savi adresavimo sistemą. Tarkim, tinklo interfeiso (fizinis) lygmuo naudoja 48 bitų MAC (Media Access Control) adresas, paprastai įrašytas į tinklo kontrolerio atmintį.

Interneto lygmuo (tinklinis lygmuo OSI modelyje) naudoja IP adresą. IP protokolas naudoja specialų antraštės lauką `Protocol`, kuris identifikuoja aukštesnio lygmens protokolą.

Transportinio lygmens protokolai naudoja portų numerius (port number), kuriuos kiekvienas procesas naudoja pasirinktinai. Šis numeris yra nurodomas transportinio lygmens protokolo perduodamo paketo antraštėje.

Portų numerį sudaro 16 bitų ir standartiniuose protokoluose jie yra standartizuoti. Pilnas standartinių numerių sąrašas yra pateiktas RFC 1700 “Assigned Numbers”. Kai kurie jų yra pateikti 6.5 lentelėje.

6.5 lentelė. Kai kurie standartiniai portų numeriai.

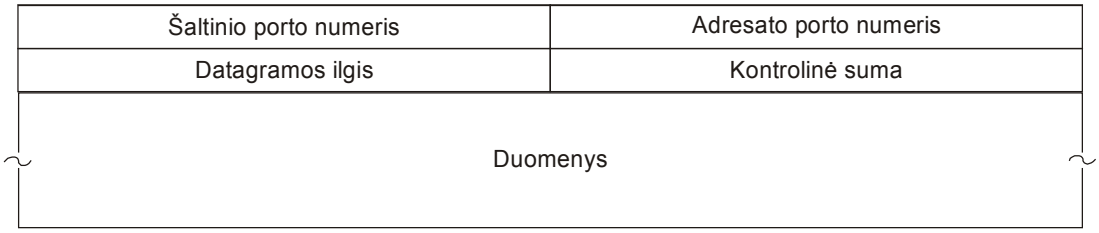
Porto numeris	Pavadinimas	Aprašymas
7	echo	Echo procesas
20	ftp-data	Duomenų perdavimas FTP protokolu
21	ftp	Valdančios FTP komandos
23	telnet	Telnet programa
25	smtp	Elektroninis paštas (Simple Mail Trasfer Protokol)
53	domain	Tinklo vardų serveris (Domain Name Server)
67	bootp	Sistemos pakrovimo serveris (Bootstrap Protocol)
68	bootpc	Sistemos pakrovimo klientas (Bootstrap Protocol)
69	tftp	Duomenų perdavimas (Trivial File Trasfer Protocol)
70	gopher	Gopher informacinė sistema
80	www-http	WWW (Hyper Text Trasfer Protocol)
110	pop3	Elektroninis paštas (POP 3 versija)
119	nntp	Telekonferencija (Network News Transfer Protocol)
123	ntp	Laikrodžio sinchronizacija (Network Time Protocol)
161	snmp	Valdymas/statistika (Simple Network Management)
179	bgp	Maršrutizavimo informacija (Border Gateway Protocol)

6.3.1 User Datagram Protocol (UDP)

Šis protokolas perduoda duomenis tinklu be išankstinio ryšio kanalo sukūrimo. UDP naudoja IP protokolą, ir kaip ir pastarasis, neatlieka duomenų perdavimo patikimumo kontrolės. Dėl

[10] Pavyzdyje pateikiamas 128.138.243/24 tinklas.

šios priežasties, jei programoms yra reikalingas patikimumas, jos privalo apsikeisti specialiais, gavimą patvirtinančiais pranešimais. Kadangi šis protokolas suteikia minimalų funkcionalumą, jo sukūrimas nebuvo labai ilgas, palyginant su TCP. UDP antraštę sudaro 8 oktetai (6.5 pav.). UDP protokolo reikšmė IP protokolo `Protocol` antraštėje yra lygi 17.



6.5 pav. UDP datagrama.

Štai keletas UDP protokolo pavyzdžių:

- apsikeitimas duomenimis su DNS serveriu, porto numeris 53;
- sisteminio laiko sinchronizacijos protokolas NTP, porto numeris 123;
- sistemos pakrovimo protokolas BOOTP, porto numeriai 67 ir 68;
- nutolusio duomenų kopijavimo protokolas TFTP, porto numeris 69;
- nutolusių procedūrų RPC iškvietimas, porto numeris 111.

Aplikacijos, kurioms yra reikalingas patikimesnis apsikeitimo duomenimis būdas naudoja kitą transportinio lygmens protokolą – TCP.

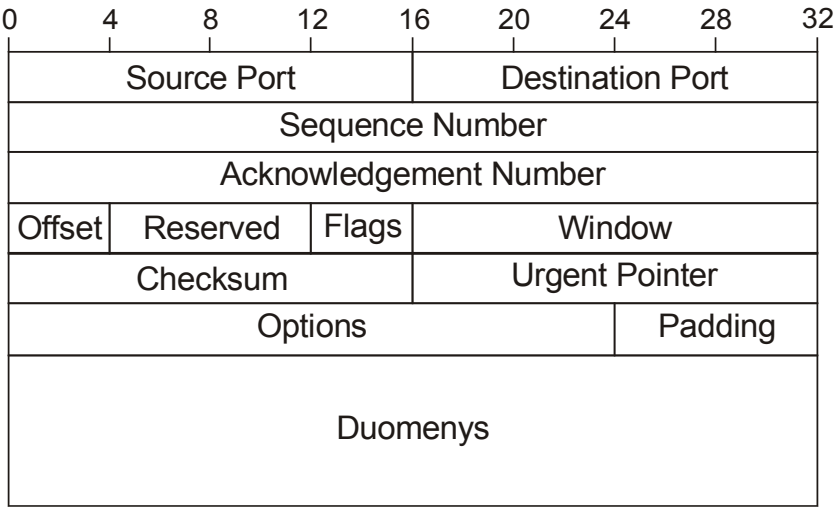
6.3.2 Transmission Control Protocol (TCP)

Šis protokolas užtikrina patikimą duomenų perdavimą. Pradžioje yra sukuriamas ryšys su kita kanalo puse (virtualus kanalas), o tik po to yra atliekamas duomenų perdavimas. Šio protokolo pagrindu yra sukurtos tokios aplikacijos kaip Telnet, FTP ir HTTP. Kai kurios TCP protokolo savybės:

- prieš perduodant duomenis yra būtina sukurti virtualų ryšio kanalą, o atlikus duomenų perdavimą kanalas privalo būti uždarytas;
- informacijos perdavimas yra patikimas, t.y. kontroliuojama, kad būtų perduoti duomenys, neįvyktų dubliavimas ar eilės tvarkos pakeitimas;
- yra galimas duomenų srauto valdymas, leidžiantis išvengti perpildymo ir kamščio;
- yra galimas skubių duomenų perdavimas be eilės.

Visą aukščiau išvardintą funkcionalumą atlieka TCP protokolas, todėl aukščiau esanti aplikacija gali būti paprastesnė.

TCP-kanalas yra multipleksinis duomenų srautas tarp dviejų ryšio mazgų – šaltinio ir gavėjo. Duomenys yra perduodami kintamo ilgio paketais, vadinamais segmentais. Kiekvienas TCP-segmentas (6.6 pav.) prasideda antrašte, o baigiasi aplikacijos duomenimis.



6.6 pav. TCP segmentas.

Kiekvienas srautu perduodamas segmentas turi eilės numerį (`Sequence Number`). Šis numeris yra naudojamas ir segmento gavimo patvirtinimui (`Acknowledgement Number`), kuriuo yra perduodama siuntėjui gautų nuoseklių paketų kiekis.

Šie du numeriai užima po 32 bitus, todėl jų maksimali reikšmė gali būti 2^{32}-1, po kurio seka 0. Sukuriant ryšio kanalą yra susitariama dėl pradinių eilės numerių į vieną ir į kitą pusę – Initial Sequence Number, ISN. Pirmojo perduodamo segmento eilės numeris bus lygus ISN+1.

Duomenų srautas valdomas vadinamo slenkančio lango (`Sliding Window`) pagalba. Kiekvienoje segmento antraštėje yra laukas `Window`, kuriame yra nurodomas duomenų kiekis, kurį adresatas pasiruošęs priimti pradedant segmentu, kurio eilės numeris yra lygus `Acknowledgement Number`.

Duomenys prasideda po `Offset` lauke nurodytų 32-jų bitų. Lauke `Flags` gali būti nurodomas viena iš šių reikšmių:

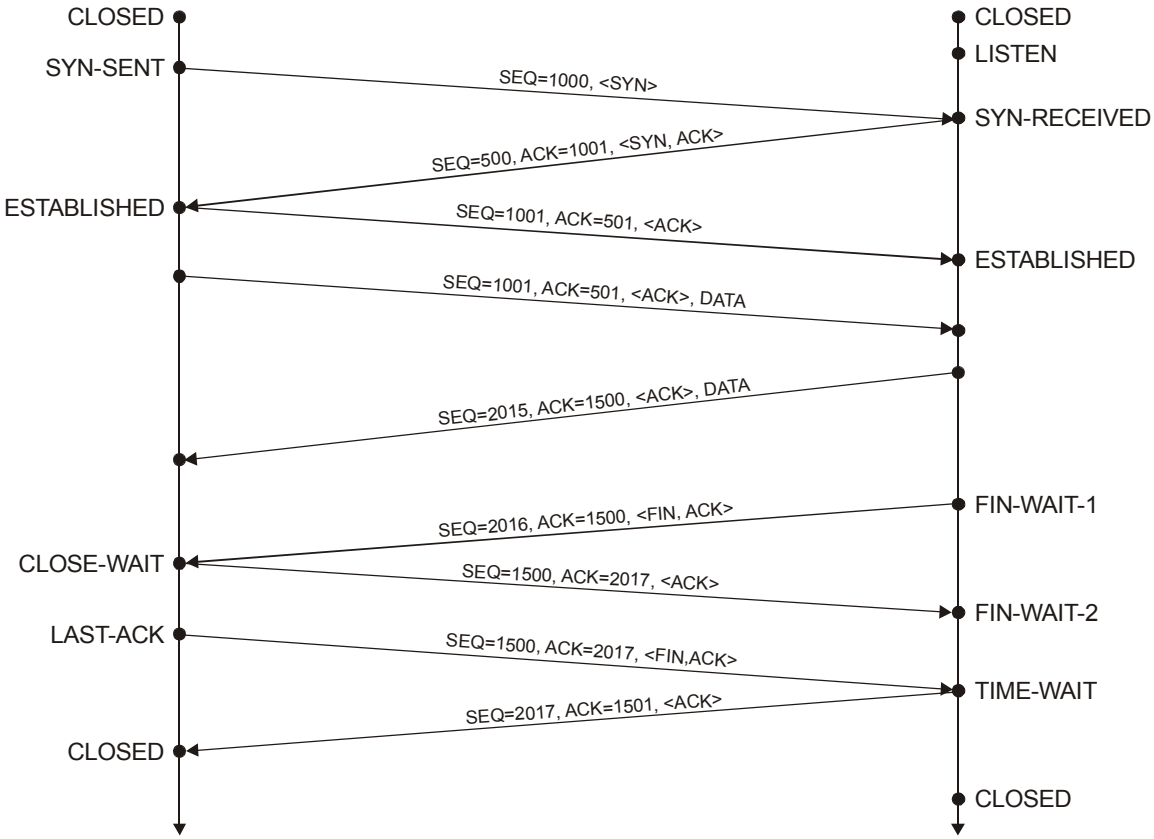
`URG`	Segmente yra perduodami ekstriniai duomenys, o jų vieta nurodoma antraštės lauke Urgent pointer.
`ACK`	Nurodo, kad antraštės lauke Acknowledgement Number yra perduodamas duomenų gavimo patvirtinimas.
`PSH`	Nurodo, kad buferyje esantys duomenys privalo būti perduoti nedelsiant, t. y. nelaukiant, kol jų kiekis bus lygus maksimaliam segmento dydžiui.
`RST`	Nurodo, kad yra būtina nutraukti TCP-kanalą.
`SYN`	Nurodo, kad šis segmentas yra vienas iš kanalo valdymo pranešimų.
`FIN`	Nurodo, kad segmento siuntėjas nori baigti duomenų perdavimą ir uždaryti virtualų kanalą.

Laukas `Checksum` yra naudojamas apsaugai nuo perdavimo klaidų. Kontrolinė suma yra apskaičiuojama panaudojant 12 virtualios antraštės oktetų, kuriame yra šaltinio ir gavėjo IP adresai.

6.3.2.1 TCP seanso būsenos

Pradinė seanso fazė yra vadinamasis “trigubas rankos paspaudimas” – three-way handshake. Šis procesas yra asimetriškas – viena iš pusių yra klientas, o kita – serveris. Klientas nusiunčia serveriui segmentą `SYN`. Šis segmentas neturi jokios papildomos informacijos, išskyrus kliento identifikaciją (porto numerį) ir požymį `SYN` lauke `Flags`. Jei serveris yra pasiruošęs priimti klientą, jis sukuria virtualų loginį kanalą (t.y. sukuria reikiamas duomenų struktūras) ir nusiunčia klientui segmentą su nustatytu pradiniu segmento eilės numeriu ir požymiais `SYN` ir `ACK`. Klientas atsako serveriui nusiųsdamas segmentą su `ACK`

požymiu, pranešdamas, kad sėkmingai gavo serverio patvirtinimą ir savo ruožtu sukūrė loginio kanalo struktūras savo pusėje. Visas šis procesas yra pateiktas 6.7 paveiksle.



6.7 pav. Komunikacinių mazgų būsenos TCP-seanso metu.

Po to abi pusės pradeda segmentų su duomenimis perdavimą, kur kiekviename segmente yra nusiunčiamas ankstesnio segmento gavimo patvirtinimas ir nauja “lango” reikšmė. Pradedant nuo paskutinio patvirtinto segmento šaltinis gali, nelaukdamas kiekvieno segmento gavimo patvirtinimo, perduoti duomenis, kurių kiekis ne didesnis nei “lango” reikšmė. Jei šaltinis po tam tikro laiko negauna patvirtinimo apie duomenų gavimą, jis pakartoja duomenis, nuo paskutinio patvirtinto okteto. Duomenų perdavimo patikimumu rūpinasi protokolas, o ne aplikacija naudojanti protokolą, todėl protokolas privalo saugoti visas nusiųstų, bet nepatvirtintų segmentų kopijas. Jei viena iš pusių gauna sugadintos eilės duomenis ir kai kurių segmentų trūksta, ji saugo duomenis laukdamas, kol bus atsiųsti trūkstami segmentai. Tokių duomenų gavimas nėra patvirtinamas.

TCP seanso pabaiga taip pat, kaip ir jo sukūrimas vyksta keliais etapais. Bet kuri iš pusių gali nutraukti kanalą nusiųsdama segmentą su nustatytu požymiu `FIN`. Šio segmento patvirtinimas visiškai nutraukia ryšį ir kanalas yra sunaikinamas.

Visos komunikacinio mazgo – kliento arba serverio būsenos yra pateiktos 6.6 lentelėje.

6.6 lentelė. TCP seanso būsenos.

Būsena	**Aprašymas**
`LISTEN`	Komunikacinis mazgas yra pasiruošęs ryšiui su bet kuriuo iš nutolusių mazgų.
`SYN-SENT`	Patvirtinimo dėl ryšio sukūrimo laukimas.
`SYN-RECEIVED`	Patvirtinimo dėl ryšio sukūrimo atsakymo gavimo laukimas.
`ESTABLISHED`	Kanalo būsena, kai yra galimas dupleksinis apsikeitimas duomenimis.
`CLOSE-WAIT`	Kanalo uždarymo pranešimo laukimas.
`LAST-ACK`	Patvirtinimo dėl ryšio kanalo nutraukimo laukimas. Ši būsena kyla po kanalo

	nutraukimo susitarimo, kai kanalas jau yra tapęs simpleksinis.
FIN-WAIT-1	Mazgas yra nusiuntęs ryšio nutraukimo pranešimą (iniciatorius) ir laukia patvirtinimo (kanalas jau simpleksinis).
FIN-WAIT-2	Kanalo pabaigos patvirtinimo laukimas.
CLOSING	Kanalo pabaigos patvirtinimo laukimas.
TIME-WAIT	Laukiama galutinių duomenų gavimo patvirtinimas. Laukiamas t.t. laiko intervalas, vadinamas 2×MSL[11] (Maximum Segment Lifetime).
CLOSED	Fiktyvi būsena, kurios metu kanalas praktiškai neegzistuoja.

6.3.2.2 Duomenų perdavimas TCP kanalu

Nors faktiškai duomenų perdavimas vyksta segmentų pavidalu, tačiau logiškai šis srautas yra nuoseklus oktetų srautas, kur kiekvienas oktetas yra adresuojamas atskiru eilės numeriu. Kiekvieno segmento antraštėje yra perduodamas pirmojo perduodamų duomenų okteto eilės numeris. Paprastai TCP modulis nusprendžia savarankiškai kada suformuoti segmentą ir jį siųsti ir kada perduoti duomenis laukiančiam procesui. Jei duomenis reikia perduoti nedelsiant, aukštesnysis protokolas (aplikacija) nustato požymį PSH, kuris nurodo TCP moduliui tuojau pat siųsti duomenis laukiančius eilėje.

Gavęs duomenis TCP modulis naudoja kontrolinę sumą duomenų vientisumui nustatyti. Eilės numeris, kaip jau buvo minėta, yra priskiriamas kiekvienam iš perduodamų duomenų oktetų. TCP protokole yra naudojamas pozityvaus patvirtinimo ir duomenų pakartojimo principas (Positive Acknowledgement and Retransmission, PAR). Tai reiškia, kad jei adresatas tvarkingai gauna duomenis, jis apie tai praneša siuntėjui. Jei siuntėjas kurį laiką negauna patvirtinimo, jis pasiunčia duomenis dar kartą, tačiau niekada nenaudojamas neigiamo patvirtinimo principas (NAK).

Duomenų perdavimo tarp dviejų kompiuterių eiga yra pateikta 6.8 paveiksle.

Duomenų srauto valdymui TCP protokolas naudoja specialią sąvoką, vadinamą “slenkančiu langu”. Langas yra duomenų kiekis, kurį TCP modulis gali išsiųsti (send window) arba priimti (receive window). Langų dydis faktiškai atspindi komunikacinių mazgų buferių būseną. Priėmimo langas rodo duomenų kiekį, kurį adresatas gali priimti, o siuntimo langas – duomenų kiekį, kurį siuntėjas gali išsiųsti, nelaukdamas patvirtinimo iš adresato. Adresatas (gavėjas) gali keisti siuntėjo lango dydį dviem būdais – patvirtindamas eilinio segmento gavimą arba nusiųsdamas naują lango reikšmę antraštės lauke Window kartu su patvirtinančiu segmentu.

6.9 paveiksle yra pateikta siuntėjo lango interpretacijos schema. Langas gali keisti savo dydį priklausomai nuo to, kuri kraštinė kinta (kairė ar dešinė):

- langas užsidarinėja, kai kairysis lango kraštas juda į dešinę. Tai vyksta siunčiant duomenis;
- langas atsidarinėja, kai dešinysis kraštas juda į dešinę. Tai vyksta tuo atveju, kai adresato buferis atsilaisvina;
- langas spaudžiasi, kai dešinysis kraštas juda į kairę. Nors tokia situacija nėra rekomenduojama, tačiau TCP modulis turi būti numatęs tokios situacijos apdorojimą.

Kai kairysis lango kraštas pasiekia dešinįjį – lango dydis tampa lygus nuliui ir duomenų siuntimas baigiasi.

Lango dydis, kurį adresatas siuntėjui pasiūlo yra vadinamas rekomenduojamu langu (offered window), kuris paprasčiausiu atveju yra lygus priimančiojo buferio laisvos vietos kiekiui. Naudodamas šią reikšmę siuntėjas apskaičiuoja naudotiną langą (usable window), kuris yra lygus rekomenduojamo lango dydžiui minus išsiustų, bet nepatvirtintų duomenų kiekiui. Tokiu

[11] RFC 793 “Transmission Control Protocol” rekomenduojamas MSL yra 2 minutės, tačiau realiai yra naudojamos 30 sek., 1 ar 2 minučių reikšmės.

būdu naudotinas langas yra lygus rekomenduojamam arba mažesnis. Neefektyvus patvirtinimo strategijos parinkimas gali sukelti vadinamąjį “aklo lango” sindromą (Silly Window Syndrome, SWS), kuris pasireiškia prastu duomenų perdavimo greičiu.

6.3.2.3 “Aklo lango” sindromas

Standartų laikymasis realizuojant TCP modulį įgalina jį naudoti bet kuriuose komunikaciniuose mazguose. Svarbiausias etapas TCP protokole yra segmento gavimo patvirtinimas, tačiau standartas neapibrėžia per kiek laiko turi būti gautas patvirtinimas. Korektiška, protokolo specifikacijos atžvilgiu, tačiau neoptimali modulio realizacija gali sukelti prastą viso komunikacinio kanalo darbą ir sukelia “aklo lango” sindromą.

Tarkime naudojant TCP protokolą yra perduodamas didelis failas, o segmentą sudaro 200 oktetų. Pradžioje perdavimo langas yra lygus 1000 oktetų, todėl siuntėjas iš karto perduoda 5 segmentus, po 200 oktetų. Gavęs pirmąjį segmentą adresatas nusiunčia patvirtinimą `ACK=201`. Gavęs šį pranešimą siuntėjas apskaičiuoja naują naudotiną langą lygų 1000-200=800.

Tarkime kyla situacija, kai siuntėjo buferyje yra 50 oktetų, o aplikacija nustato `PSH` požymį. Tokiu būdu TCP modulis nusiunčia 50 baitų, o vėliau likusius 150, nes langas yra lygus 1000-800-50=150. Po kurio laiko siuntėjas gauna 50 baitų patvirtinimą ir nustato lango dydį lygų 1000-950=50 ir vėl yra priverstas siųsti 50 oktetų segmentą, nors situacija to jau nebereikalauja.

Jei nagrinėsime toliau, tai pastebėsime, jog ši situacija periodiškai kartojasi, t.y. kas kurį laiką siuntėjas bus priverstas siųsti segmentą turintį nepagristai mažą duomenų kiekį. Jei virtualus kanalas perduoda daug duomenų ir ilgai gyvuoja, jis gali užteršti tinklus mažais duomenų paketais. Tačiau laimei tokios situacijos pakankamai lengva išvengti modifikuojant algoritmą:

1. priimanti pusė neturi prašyti mažų langų, t.y. gavėjas negali prašyti didesnio lango nei esamasis, tol kol negali padidinti jo iki segmento dydžio arba iki ½ priėmimo buferio dydžio;
2. siuntėjas privalo susilaikyti nuo duomenų siuntimo tol, kol nesukaups pilno segmento arba nesukaups duomenų, kurių kiekis yra didesnis nei ½ maksimalaus kada nors rekomenduoto lango dydžio.

6.3.2.4 Vangaus starto principas

Senosios TCP realizacijos nuo pat pradžių siųsdavo segmentus rekomenduojamo lango rėmuose. Tai sukeldavo sprogstamą duomenų srauto tinkle padidėjimą, kuris sukeldavo tinklo perpildymą, kurio pasėkoje dalis segmentų būdavo atmetami ir reikalaudavo pakartojimo.

Siekiant išvengti tokių situacijų buvo pasiūlytas vangaus starto (slow start) algoritmas. Jis remiasi principu, kad duomenų perdavimo pradiniame etape segmentai turi būti perduodami greičiu, proporcingu patvirtinimų gavimui.

Pradžioje lango dydis nustatomas lygus vienam segmentui, t.y. MMS, kurį nustato adresatas arba lygų reikšmei pagal nutylėjimą, dažniausiai 536 arba 512 baitų. Šis langas yra vadinamas perpildymo langu (congestion window). Kai siuntėjas gauna patvirtinimą jis padidina perpildymo lango dydį gautąja reikšme ir sekantį naudotiną langą parenka iš dviejų – iš perpildymo ir rekomenduojamo langų išrenka mažesnįjį. Lango kitimo formulę galime atvaizduoti sekančiu būdu:

$cwnd_0 = sz$
$cwnd_1 = cwnd_0 + (cwnd_0/sz)*sz = 2*cwnd_0 = 2*sz$
...
$cwnd_n = 2*cwnd_{n-1} = 2^n*sz$

, kur $cwnd_i$ – i-tojo lango, o sz – segmento dydis.

Kai perpildymo langas pakankamai išauga ir pasiekia efektyviausią duotajam kanalui reikšmę įsijungia kamščių pašalinimo mechanizmas.

6.3.2.5 Kamščių vengimas

Kamščiai gali kilti dėl įvairių priežasčių, tarkim prie šliuzo, skiriančio greitą ir lėtą tinklus arba duomenys yra multpleksuojami (dauginami) kanale, kurio talpa yra mažesnė jų sumai. Visais šiais atvejais prarandami paketai.

Algoritmai, padedantys išvengti tokių situacijų laikosi principo, kad duomenų praradimas dėl fizinio tinklo savybių (klaidų) yra santykinai maža (mažesnė nei 1%). Todėl yra manoma, kad bet koks patvirtinimo negavimas arba patvirtinimo dublikato gavimas yra sukeltas kamščio, kilusio kažkur tarp komunikacinių mazgų.

Vangaus starto ir kamščių vengimo mechanizmai yra nepriklausomi, tačiau realizuojami dažniausiai kartu, nes vienas kitą papildo. Jų realizacijai yra būtini du papildomi parametrai: perpildymo langas `cwnd` ir vangaus starto slenkstis `ssthresh`. Algoritmai remiasi sekančiomis taisyklėmis:

- pradinės `cwnd` ir `ssthresh` reikšmės yra nustatomos vieno segmento dydžiui ir 65535 baitų atitinkamai;
- maksimalus perduodamų duomenų kiekis, kurį gali perduoti siuntėjas, negali viršyti mažesniosios iš perpildymo ir rekomenduojamo langų reikšmių;
- kai atsiranda kamščio tikimybė, t.y. kyla time-out arba yra gaunamas patvirtinimo dublikatas, parametras `ssthresh` yra nustatomas lygus pusei einamojo lango, tačiau ne mažesnis nei du segmentai. Jei kamščio požymiu yra time-out – tai dar papildomai `cwnd` reikšmė yra nustatoma lygi vienam segmentui, t.y. įjungiamas vangus startas;
- kai siuntėjas gauna patvirtinimą, jis padidina `cwnd` reikšmę, tačiau nauja reikšmė `cwnd` priklauso nuo to. kokią procedūrą modulis atlieka – vangų startą ar šalina kamštį.

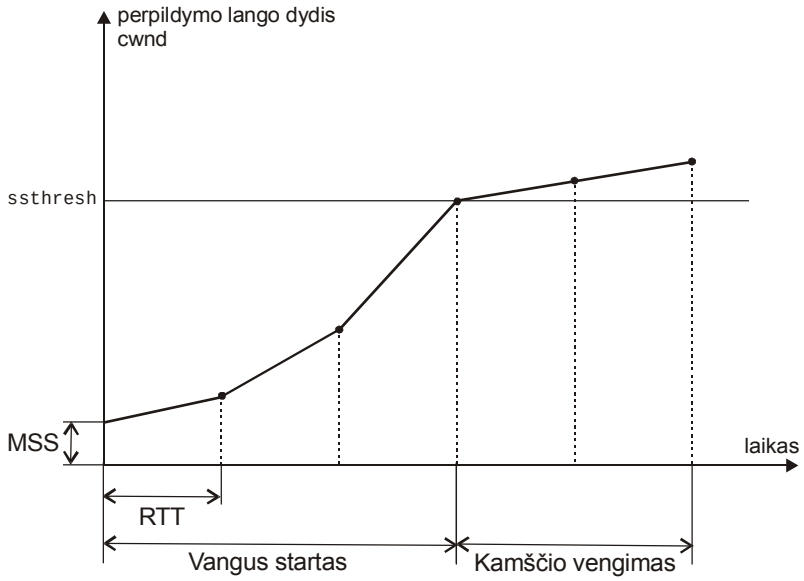
Jei `cwnd` reikšmė yra mažesnė ar lygi `ssthresh` reikšmei, tai TCP yra vangaus starto fazėje, kitu atveju – vengiama kamščių.

Vangus startas prasideda nuo vieno segmento nusiuntimo, po to siunčiami du, vėliau – keturi, aštuoni ir taip toliau eksponentiškai yra didinamas perdavimo langas. Kamščio vengimo metu nauja lango reikšmė yra apskaičiuojama taip:

$$cwnd_{n+1} = cwnd_n + 1/cwnd_n$$

Tokiu būdu, vengiant kamščių, nepriklausomai nuo to, kiek patvirtinimų jau yra gauta, lango augimas tampa lygus vienam segmentui per duomenų perdavimą pirmyn-atgal (Round Trip Time, RTT).

6.8 paveiksle yra pateiktas lango augimo grafikas nuo TCP seanso pradžios.

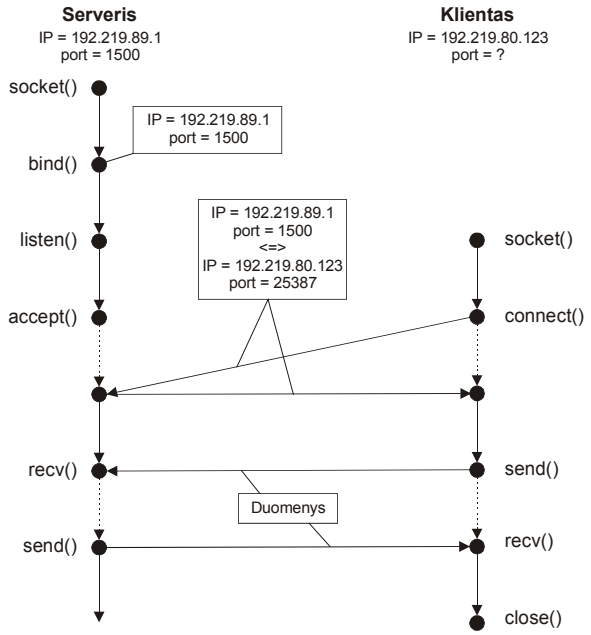


6.8 pav. Perpildymo lango dydžio kitimas vangaus starto ir kamščio vengimo metu.

6.4 Tinklo programiniai interfeisai

6.4.1 Programinis soketų interfeisas

Soketų IPC interfeisas yra nagrinėtas 3.14.1 skyriuje. Tai buvo pirmoji tinklo programinio interfeiso realizacija ir ji buvo įdiegta UNIX BSD sistemose. Ir pastaruoju metu soketų interfeisas yra labai paplitęs kuriant tinklines aplikacijas. Šiame skyriuje pateiktas pavyzdys yra 3.14.1 skyriuje nagrinėtų atvejų analogas, tačiau čia naudojamas tik vienas komunikacinis soketo domenas – `AF_INET`. Soketų programinis interfeisas, greta kitų yra palaikomas ir UNIX SV šeimos sistemose.



6.9 pav. Ryšio seanso schema naudojant soketų mechanizmą.

Naudojant TCP/IP adresavimo schemą yra taikomas dviejų lygių adresas: kompiuterio (IP) adresas ir proceso kompiuteryje numeris. Tai atsispindi `sockaddr` struktūroje:

```
struct sockaddr {
       short        sin_family;   // komunikacinis domenas AF_INET
       u_short             sin_port;     // proceso numeris
       struct in_addr      sin_addr;     // kompiuterio IP adresas
       char         sin_zero[8];
};
```

Proceso (porto) numeris turi būti iš anksto žinomas klientui.

6.9 paveiksle pateiktame pavyzdyje transportiniu protokolu yra TCP/IP, todėl prieš pradedant duomenų perdavimą reikia sukurti virtualų ryšio kanalą. Jei to nereikėtų, galima būtų naudoti UDP protokolą.

Pradžioje serveris rezervuoja porto numerį naudodamas komandą `bind(2)` ir praneša, kad yra pasiruošęs priimti klientus – `listen(2)`. Gavęs užklausą iš kliento, jis, naudodamas funkciją `accept(2)` sukuria naują socket objektą. Tam, kad serveris toliau priiminėtų klientus, konkrečiam klientui aptarnauti jis sukuria vaikinį procesą, kuris priima kliento pranešimus – `recv(2)` ir siunčia jam savus – `send(2)`.

Klientas neatlieka porto numerio rezervacijos savo pusėje, nes jam, šiuo atveju, nėra svarbu, kokį numerį jis gaus. Tai atlieka branduolys, kuris parenka laisvą numerį. Klientas nusiunčia užklausimą dėl ryšio sukūrimo serveriui – `connect(2)`, nurodydamas serverio IP adresą ir proceso (porto) numerį. Sukūrus virtualų kanalą (po “trigubo rankų paspaudimo”) klientas perduoda duomenis serveriui – `send(2)` ir priima – `recv(2)`.

Programos pavyzdys yra pateiktas šios knygos praktikos skyriuje. Jame yra panaudotos čia neaprašytos funkcijos, kurių aprašymą galima gauti `man(1)` puslapiuose.

6.4.2 Programinis TLI interfeisas

Transportinio lygmens interfeisas, TLI (Transport Layer Interface) suteikia vartotojo programoms interfeisą su transportiniais protokolais. TLI pirmą kartą buvo realizuotas UNIX SVR3 1986 metais. Šis API yra tampriai susijęs su STREAMS posisteme ir izoliuoja transportines operacijas nuo vartotojo programų. Daugumoje UNIX sistemų TLI funkcijos saugomos libnsl.a ir libnsl.so bibliotekose.

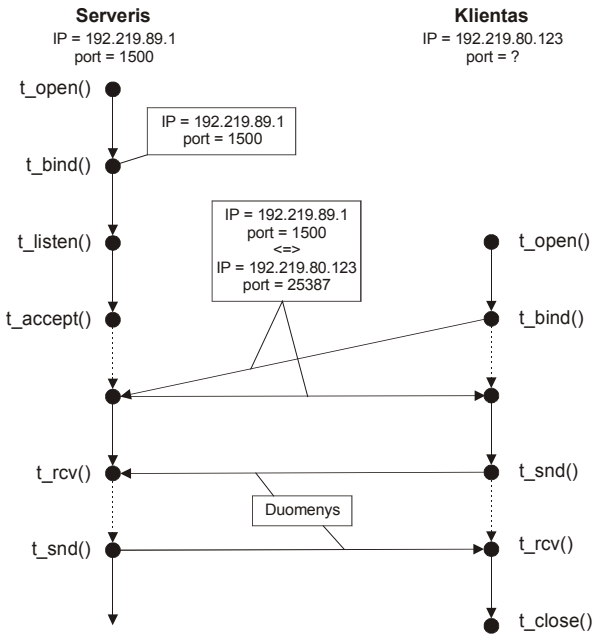
TLI darbo schema yra labai panaši į aukščiau nagrinėtą BSD soketų interfeisą ir priklauso nuo naudojamo protokolo – su išankstiniu ryšio nustatymu (TCP) arba be jo (UDP). Šios dvi schemos yra pateikiamos 6.10 ir 6.11 paveiksluose.

TLI naudoja laisvą adresavimo formatą, t.y. šis interfeisas adresavimo formatų apibrėžimą palieka žemesnio lygmens STREAMS sluoksniams. Tai leidžia vieną ir ta patį interfeisą naudoti įvairių protokolų šeimoms. Adreso nustatymui TLI naudoja bendrą `netbuf` sturktūrą:

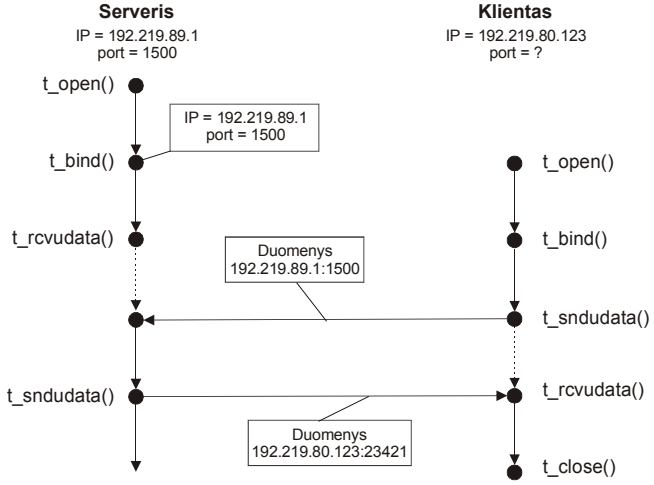
```
struct netbuf {
       unsigned int maxlen;
       unsigned int len;
       char *buf;
};
```

, kur `buf` – duomenų buferis, `maxlen` – maksimalus buferio dydis, `len` – duomenų buferyje kiekis. Ši struktūra yra naudojama ne tik adreso, tačiau ir kitos informacijos, pvz. protokolo opcijų ir pačių duomenų perdavimui. Ši struktūra yra žymiai sudėtingesnių duomenų struktūrų, naudojamų duomenims perduoti, dalis. Siekiant palengvinti dinaminį struktūrų patalpinimą buferyje yra pateikiamos dvi TLI bibliotekos funkcijos: `t_alloc(3N)` struktūros patalpinimui ir `t_free(3N)` – buferio atlaisvinimui:

```
#include <tiuser.h>
char *t_alloc(int fd, int struct_type, int fields);
int t_free(char *ptr, int struct_type);
```



6.10 pav. TLI interfeiso funkcijų darbo schema protokolui su išankstiniu ryšio sukūrimu.



6.11 pav. TLI interfeiso funkcijų darbo schema protokolui be išankstinio ryšio sukūrimo.

Kur argumentas `struct_type` nurodo, kokio tipo struktūra yra naudojama ir jis gali įgyti reikšmes, pateiktas 6.7 lentelėje.

6.7 lentelė. TLI interfeise naudojamos `struct_type` reikšmės.

Reikšmė	**Duomenų struktūra**
`T_BIND`	`struct t_bind`
`T_CALL`	`struct t_call`
`T_DIS`	`struct t_discon`
`T_INFO`	`struct t_info`
`T_OPTMGMT`	`struct t_optmgmt`
`T_UNITDATA`	`struct t_unitdata`
`T_UDERROR`	`struct t_uderr`

Kadangi TLI naudojamos ir 6.7 lentelėje pateiktos funkcijos turi ne po vieną `netbuf` lauką, argumentas `fields` nurodo kokius konkrečiai buferius reikia patalpinti į struktūrą. Galimos `fields` argumento reikšmės yra pateikiamos 6.8 lentelėje.

6.8 lentelė. TLI mechanizmo struktūrų laukų reikšmės.

Lauko reikšmė	Inicijuojami struktūrų laukai
T_ALL	Visi laukai
T_ADDR	Laukas addr struktūrose t_bind, t_call, t_unitdata, t_uderr
T_OPT	Laukas opt struktūrose t_call, t_unitdata, t_uderr, t_optmgmt
T_UDATA	Laukas udata struktūrose t_call, t_unitdata, t_discon

netbuf dydis maxlen (t.y. adreso ilgis, perduodamų duomenų kiekis ir pan.) priklauso nuo transportinio lygmens. Todėl iškviečiant t_alloc(3N) argumentu yra perduodamas įrenginio failo deskriptorius fd.

Ryšio seanso pirmuoju etapu yra funkcijos t_open(3N) iškvietimas, kuri, kaip ir sisteminė komanda open(2) grąžina failo deskriptoriaus numerį, kurį vėliau galima naudoti duomenų perdavimui:

```
#include <tiuser.h>
#include <fcntl.h>
int t_open(const char *path, int oflags, struct t_info *info);
```

, kur path – specialaus įrenginio failo vardas, pvz. /dev/tcp arba /dev/udp, oflags – failo atidarymo požymiai, tokie pat kaip ir open(2) funkcijoje. Naudojant info lauką programa gauna informaciją apie transportą, kuri, savo ruožtu, turi tokius laukus:

addr — Nustato maksimalų adreso ilgį. -1 reiškia, kad adreso ilgis neribojamas, -2 reiškia, kad duotoji programa neturi galimybės nustatyti protokolo adresą. TCP protokolas nustato šią reikšmę lygią 16.

options — Nustato opcijų lauko dydį. -1 reiškia, kad opcijų lauko dydis nėra ribojamas, -2 – protokolo opcijos nėra palaikomos.

tsdu — Nurodo maksimalų duomenų paketo dydį (Transport Service Data Unit, TSDU). 0 – protokolas nepalaiko paketinio perdavimo, -1 – reikšmė neribojamas, -2 – paprastų duomenų perdavimas nėra atliekamas. TCP protokolas atlieka nestruktūrinių duomenų perdavimą, todėl ši reikšmė yra lygi 0.

etsdu — Maksimalus ekstrinių duomenų paketo dydis (Expedited Transport Service Data Unit, ETSDU). 0 reiškia, kad protokolas nepalaiko paketinių duomenų perdavimo, -1 – dydis nėra ribojamas, -2 – protokolas nepalaiko ekstrinių duomenų perdavimo.

connect — Kai kurie protokolai palaiko duomenų perdavimą kartu su ryšio sukūrimo užklausa. Šis laukas nurodo maksimalų tokių duomenų kiekį. Nei TCP, nei UDP tokio mechanizmo neturi.

discon — Kai kurie protokolai palaiko duomenų perdavimą kartu su ryšio nutraukimo užklausa. Šis laukas nurodo maksimalų tokių duomenų kiekį. Nei TCP, nei UDP tokio mechanizmo neturi.

servtype — Nurodo transportinio mechanizmo tipą: T_COTS žymi perdavimą su virtualaus kanalo sukūrimu, T_COTS_ORD – perdavimą su virtualaus kanalo sukūrimu ir perduodamų duomenų eiliškumo kontrole, T_CLTS – perdavimą be išankstinio kanalo sukūrimo. TCP protokolas yra T_CONTS_ORD, o UDP – T_CLTS.

Prieš pradedant duomenų perdavimą transportinis mazgas turi būti susietas su konkrečiu adresu, tai atlieka t_bind(3N) funkcija:

```
#include <tiuser.h>
int t_bind(int fd, const struct t_bind *req, struct t_bind *ret);
```

, kur argumentas fd adresuoja komunikacinį mazgą, req nurodo reikiamą adresą, ret – grąžina protokolo suteiktas reikšmes. Jei yra naudojamas protokolas su išankstiniu ryšio sukūrimu, programa turi iškviesti atitinkamą funkciją:

```
#include <tiuser.h>
int t_connect(int fd, const struct t_call *sndcall, struct t_call *rcvcall);
```

Jei protokolas naudoja išankstinį ryšio sukūrimo mechanizmą, tai serverio pusėje sekantis žingsnis bus t_listen(3N) funkcija:

```
#include <tiuser.h>
```

```
int t_listen(int fd, const struct t_call *call);
```

Gavęs kliento užklausą sujungimui serveris privalo iškviesti `t_accept(3N)` funkciją, kuri faktiškai priima kliento duomenis ir sukuria reikiamas duomenų struktūras:

```
#include <tiuser.h>
int t_accept(int fd, int connfd, struct t_call *call);
```

, kur `fd` yra transportinis mazgas priėmęs klientą, o `connfd` – naujas transportinis mazgas, kuris turės faktiškai aptarnauti klientą. Naujo transportinio mazgo sukūrimu turi pasirūpinti pati vartotojo programa, t.y. vėl iškviesti `t_open(3N)`, o `fd` gali toliau priiminėti klientus.

Po virtualaus kanalo sukūrimo duomenų apsikeitimui naudojamos sekančios funkcijos:

```
#include <tiuser.h>
int t_rcv(int fildes,char *buf, unsigned nbytes, int *flags);
int t_snd(int fildes,char *buf, unsigned nbytes, int flags);
```

Argumentu flags yra perduodamos šios reikšmės: `T_EXPEDITED` – perduodami ekstriniai duomenys, `T_MORE` – duomenys sudaro loginę seką, kurių pratesimas bus perduotas sekančia `t_snd(3N)` funkcija.

Protokolams be išankstinio ryšio sukūrimo yra naudojamos kitokios funkcijos:

```
#include <tiuser.h>
int t_rcvudata(int fildes,struct t_unitdata *unitdata, int *flags);
int t_sndudata(int fildes,struct t_unitdata *unitdata, int flags);
```

Sukurtas transportinis mazgas gali būti uždarytas (sunaikintas) naudojant sekančią funkciją:

```
#include <tiuser.h>
int t_close(int fd);
```

Klaida, kilusi TLI mechanizmo darbo metu yra išsaugoma kintamajame `t_errno`, o pranešimas vartotojams išvedamas naudojant funkciją

```
#include <tiuser.h>
void t_error(const char *errmsg);
```

6.4.3 RPC – aukšto lygio programinis tinklo interfeisas

Aukščiau nagrinėti tinklo interfeisai suteikdavo vartotojo programai galimybę tiesiogiai bendrauti su transportiniu protokolu, todėl daugumą programos kodo sudaro komunikacinių mazgų sukūrimui ir valdymui.

Nutolusių procedūrų iškvietimas – RPC (Remote Procedure Call) yra aukšto lygio interfeisas, kuris izoliuoja vartotojo programą nuo tinklo ryšio subtilybių.

Tradiciškai paprogramės (procedūros, funkcijos) yra naudojamos įvairių sudėtingų uždavinių struktūrizavimui.Dažniausiai naudojamos paprogramės yra surenkamos į bibliotekas, kurias gali naudoti įvairios programos dirbančios viename kompiuteryje ir tik vienos programos rėmuose. RPC atveju, procesas, dirbantis viename kompiuteryje, iškviečia procedūrą (procesą) dirbantį kitame kompiuteryje. Nors programuotojas visai nepastebi skirtumo, ar yra kviečiama bibliotekos funkcija vietiniame, ar nutolusiame kompiuteryje, tačiau, vietinio iškvietimo atveju duomenys yra perduodami steku ar bendrai naudojamos atminties srityse, tai RPC atveju procedūros argumentai ir atsakymas yra perduodami tinklu.

RPC žingsniai:

1. programa – klientas iškviečia lokalią procedūrą vadinamą stub. Klientas jai perduoda parametrus, o pastaroji grąžina rezultatą. Kliento programai atrodo, kad ji visą laiką dirba su vietine procedūra, tačiau pastaroji priima argumentus, performuoja juos į standartinę išraišką ir sudaro užklausą. Šis procesas yra vadinamas surinkimas (marshalling).
2. Užklausa tinklu perduodama nutolusiai sistemai. Tam stub gali panaudoti įvairiausius transportinius protokolus ir technikas, nagrinėtas praeituose skyriuose.
3. Serverio kompiuteryje viskas įvyksta atvirkščia seka: stub laukia užklausos, iš užklausos ištraukia argumentų reikšmes (unmarshalling).

4. Serverio stub procedūra, naudodama gautus argumentus iškviečia lokalią procedūrą.
5. Procedūra atlieka savo darbą ir grąžina stub atsakymą, o pastaroji, kaip ir kliento atveju, įvykdo surinkimą ir pasiunčia klientui atsakymą.
6. Kliento procedūra gauna atsakymą, jį išpakuoja (unmarshalling) ir perduoda atsakymą ją iškvietusiai programai.

Prieš klientui iškviečiant nutolusią procedūrą reikia susirišti su reikiamu kompiuteriu ir reikiamu procesu, todėl uždavinys skyla į dvi dalis:

- kompiuterio, su reikiamu serveriu suradimas ir
- reikiamo proceso kompiuteryje suradimas.

Reikiamo kompiuterio suradimas gali būti atliekamas įvairiai: galima tiesiogiai nurodyti kompiuterio adresą, arba galima sukurti kokį nors centrinį servisą, kuriame yra registruojami visi vietiniame tinkle prieinami RPC servisai.

Kiekviena RPC procedūra yra vyra identifikuojama unikaliais programos ir procedūros numeriais. Programos numeris nurodo procedūrų grupę, kurioje kiekviena procedūra turi savo unikalų numerį. Kiekvienai programai yra suteikimas ir versijos numeris, kuris leidžia nekeičiant programos numerio atlikti nežymius pakeitimus (tarkim, pridėti naują procedūrą).

Serverio pusėje yra naudojamas portmap(1M) arba rpcbind(1M) servisas, kuris naudoja visiems žinomą 111 portą. Šiame servise programa servisas užsiregistruoja.

Pradžioje kliento programa kreipiasi į serverio servisą portmap(1M), kuris arba grąžina reikiamo serviso numerį, arba tiesiogiai perduoda užklausą reikalaujamam servisui.

Plačiau apie PRC naudojimą yra pateikta praktikos skyriuose.

7 Sistemos startavimas ir darbo pabaiga

UNIX yra sudėtinga sistema, todėl reikėtų mokėti ją startuoti (pakelti) ir baigti darbą (išjungti, nuleisti) tvarkingai ,jei norite, kad sistema toliau dirbtų tvarkingai.

Pradžioje, kai tiek techninė, tiek programinė įranga buvo standartizuota sistemos startavimas (pasikrovimas) buvo paprastas, tačiau dabar, kai UNIX sistemos sukasi ant techniškai nevienalyčių, įvairios konfigūracijos PC kompiuterių, pasikrovimo procedūros turi žaisti pagal Microsoft taisykles, .t. y. kreipti dėmesį į didelę aibę galimų konfigūracijų. Todėl visas sistemos pakilimo sudėtingumas yra susijęs su techninės įrangos įvairove, todėl visa tolesnė informacija gali ir tikrai skirsis nuo Jūsų naudojamos sistemos.

7.1 Sistemos startavimas

Šis procesas vadinamas “bootstrap” – įkrova. Jo metu neveikia įprasti sisteminiai servisai, todėl kompiuteris turi pats save prikelti iš numirusiųjų. Startavimo metu į atmintį yra pakraunamas branduolys, jis inicijuojamas ir paleidžiamas. Viso to pabaigoje sistema yra pasiruošusi aptarnauti vartotojus.

Sistemos startavimo trukmę lemia įvairūs veiksniai: klaidos konfigūraciniuose failuose, trūkstama ar netvarkinga techninė įranga, sugadintos failų sistemos ir pan.

Trumpai apžvelgsime startavimo procedūrą: kai įjungiamas kompiuteris jis perskaito pakrovimo kodą iš ROM. Šis kodas bando pakrauti sistemos branduolį. Pastarasis paleidžia sisteminį init procesą, kurio PID visada yra lygus 1. Tada yra patikrinamos ir sujungiamos (mount) failų sistemos bei paleidžiami sisteminiai servisai (daemons). Šiuos darbus atlieka startiniai skriptai, vadinami rc scripts (rc – run command), istoriškai likę nuo CTSS operacinės sistemos.

Paprastai sistemos startavimas yra vykdomas sekančiais žingsniais:

- branduolio pakrovimas ir inicijavimas. Branduolys yra programa, kuri yra pakraunama į atmintį ir paleidžiama. Branduolys yra saugomas `/unix`, `/vmunix` ar panašiame kataloge. Dauguma sistemų šį procesą realizuoja dviem žingsniais – pradžioje yra pakraunama maža pakrovimo programa, kuri vėliau pakrauna branduolį. Branduolys pratestuoja atmintį (sužino kiek yra atminties), dalį jos rezervuoja. Ši atminties dalis vartotojo procesams tampa neprieinama. Dažniausiai pasikraudamas branduolys išveda pranešimą apie visą ir vartotojui paliktą atminties kiekį.
- techninės įrangos suradimas ir konfigūravimas. Viena pirmųjų branduolio užduočių – nustatyti kokia mašinoje yra techninė įranga. Apie tai, kokios įrangos jis gali tikėtis yra pasakoma branduolio konfigūracijos metu. Branduolys paeiliui pertikrina visus nurodytus įrenginius ir į konsolę išveda informacija apie sėkmingą arba ne įrenginio konfigūraciją. Dažnai administratoriaus suteiktos informacijos branduoliui yra ne gana, todėl jis kreipiasi į tvarkykles siekiant gauti papildomos informacijos. Jei tvarkyklė nerasta arba neatsako, arba nėra įrenginio – jis yra ignoruojamas ir sistema dirba tarsi jo visai nebūtų.
- sisteminių procesų sukūrimas. Šio žingsnio metu vartotojo atminties srityje branduolys sukuria keletą procesų. Jie vadinami sisteminiais, nes yra sukuriami ne normaliu (fork) būdu. BSD sistemose yra sukuriami trys procesai: swapper (0), init (1) ir pagedaemon (2); System V sistemose: sched (0), init (1) ir kiti įvairūs atminties bei branduolio valdymo procesai; Linux'e nėra 0 proceso, jis turi init (1) ir kitus atminties bei branduolio valdymo procesus. Iš šių sisteminių procesų tik init (1) yra pilnavertis procesas, visi kiti yra branduolio dalys procesų pavidalu. Šiuo žingsniu branduolio vaidmuo startavimo procese baigtas. Dabar darbą perima init.
- operatoriaus pasirinkimas (tik single-user mode). Jei buvo pasirinkta single-user sistemos krovimasis – atsiranda root slaptažodžio reikalaujanti įvedimo eilutė. Jei paspausite ^D – sistema toliau krausis lyg niekur nieko. Įvedus root slaptažodį bus pateiktas shell'as su prijungta / failų sistema (kai kurios sistemos prijungia ir /usr failų sistemą). Dažniausiai single-user naudojamas fsck komandos paleidimui. Kai išeinama iš shell'o sistema pabando toliau pakrauti į multi-user mode.
- startinių skriptų paleidimas. Tai yra paprasti shell skriptai, kurie yra saugomi /etc kataloge. Juos paleidžia init procesas naudodamas tam tikrą algoritmą.
- multi-user inicijavimas. Norint, kad butų galima prisijungti, getty procesai turi būti paleisti. Tuo atskirai pasirūpina init. Jei sistema yra sukonfigūruota iš karto paleisti x-terminalo login'ą – init pasirūpina, kad būtų paleisti xdm, gdm arba dtlogin procesai. init taip pat pasirūpina startavimo lygiais.

7.1.1 PC startavimas

Specialiai UNIX pritaikytos mašinos turi protingus ROM'us, kurie tiksliai žino apie aparatūrinę įrangą, gali ją susitvarkyti, sukonfigūruoti naują ir pan. PC pradinė startavimo programa vadinama BIOS'u (Basic I/O System). Ji yra žymiai paprastesnė nei UNIX mašinų ROM'as. PC turi kelis BIOS: vieną sau, kitą grafiniam adapteriui ir dar vieną SCSI adapteriui. PC BIOS'as žino tik apie kelis įrenginius, t.y. dažniausiai IDE kontroleris, klaviatūra, serijiniai ir lygiagretūs portai.

PC BIOS'as leidžia pasirinkti iš kurio įrenginio startuoti sistemą. Tai dažniausiai tėra arba pirmasis CD, arba pirmasis HDD, arba pirmasis FDD. Tik nedaugelis BIOS'ų leidžia pasirinkti startavima iš SCSI įrenginio.

Kai BIOS suranda iš kurio įrenginio startuoti sistemą ji pabando pakrauti pirmuosius 512K, kurie yra vadinami MBR (Master Boot Record). Ši programa nurodo iš kurio disko segmento reikia pakrauti antrąją pakrovimo programą (boot loader). Dažniausiai MBR tiesiog pasako, kad toliau reikia pakrauti pirmąjį disko segmentą, tačiau FreeBSD ir Linux turi sudėtingesnes MBR, kurios suteikia kelių operacinių sistemų ar branduolių pakrovimo servisą. Antroji programa, boot loader, pakrauna branduolį.

Linux MBR programa – lilo (Linux Loader) yra konfigūruojama su lilo komanda. Konfigūracinis failas yra /etc/lilo.conf. Ji, pagal nutylėjimą pakrauna /vmlinuz branduolį. lilo gali sėdėti MBR arba Linux sistemos root disko startavimo takelyje (boot record in root partition).

FreeBSD PC startavimo programa (boot loader) yra padalinta į dvi dalis: viena jų sėdi MBR, o kita FreeBSD root disko dalyje. Šios dalys turi būti instaliuotos atskirai. FreeBSD boot loader instaliuojamas su `boot0cfg` komanda. Startuojant MBR nuskanuoja visus diskus ieškodama segmentų, kurie atrodo kaip boot loader'iai. Tada ji pateikia pasirinkimo sąrašą. Antroji boot loader'io dalis leidžia pasirinkti branduolį, kurį reikėtų pakrauti. Informacija ji ima iš `/boot/loader.conf`, `/boot/loader.conf.local` ir `/boot/defaults/loader.conf fail•`.

7.1.2 single-user režimas

Solaris: naudojant originalią Sun mašiną reikia paspausti L1 (kai kada STOP) ir 'a' klavišus. Jei nėra šių klavišų, o naudojamas startavimo PROM'as, t.y. reikia surinkti boot komandą, tai surenkame boot –s.

6.1 lentelė. Solaris boot komandos variantai

Komanda	Funkcija
`boot`	Normalus sistemos startas
`boot /path_to_kernel`	Pakrauti alternatyvų branduolį
`boot –s`	Startuoti single-user režimą
`boot –r`	Perkonfigūruoti branduolį ir ieškoti naujų įrenginių
`boot –a /etc/system.bak`	Branduolys skaitys /etc/system.bak, o ne /etc/system
`probe-scsi`	Pateikia visų prijungtų SCSI įrenginių sąrašą

7.1.3 Startavimo programos (skriptai)

Pagrindinės užduotys, kurias atlieka startavimo programos yra šios:

1. nustatyti kompiuterio vardą;
2. nustatyti laiko zoną;
3. patikrinti failų sistemas (fsck);
4. prijungti failų sistemas;
5. ištrinti senus failus iš /tmp;
6. sukonfigūruoti tinklo interfeisus;
7. startuoti įvairius servisus ir tinklą.

7.1.3.1 System V startavimo programos

System V startavimo programų sistema yra labiausiai paplitusi UNIX sistemose. Šios sistemos nustato 7 darbo lygius, kurios nustato kokius servisus reikia pakrauti ir kokias programas paleisti:

- 0 lygis – sistema visiškai baigia darbą;
- 1 lygis (arba S) – žymi single-user režimą;
- 2 – 5 lygiai – multi-user režimai;
- 6 lygis – sistemos perkrovimas.

Pagal nutylėjimą sistema yra dažniausiai pakraunama 2 lygiu. RedHat turi 10 lygių, kurių 7 – 9 yra nenustatyti (undefined).

Failas `/etc/inittab` init procesui nurodo kaip elgtis kiekvieno lygmens atveju. Nors inittab formatai yra įvairūs, tačiau bendra idėja yra paleisti t.t. komandas.

Fizinės startavimo komandų kopijos yra saugomos /etc/init.d kataloge. Kiekvienas skriptas yra atsakingas už tam tikrą sistemos aspektą ar servisą. Skriptai supranta start ir stop argumentus, kai kurie supranta ir restart, kuris dažniausiai reiškia tą patį ką ir stop bei start. Paprastas startavimo programos pavyzdys:

```
#! /bin/sh
test –f /usr/local/sbin/sshd || exit 0
case "$1" in
   start)
      echo –n "Starting sshd"
      /usr/local/sbin/sshd
      ;;
   stop)
      echo –n "Stopping sshd"
      kill `cat /var/run/sshd.pid`
      ;;
   restart)
      echo –n "Stopping sshd"
      kill `cat /var/run/sshd.pid`
      echo –n "Starting sshd"
      /usr/local/sbin/sshd
      ;;
   *)
      echo "Usage: /etc/init.d/sshd start|stop|restart"
      exit 1
      ;;
esac
```

init procesui reikia papildomos informacijos apie tai, kurią programą kuriuo metu paleisti. Procesas neleidžia programų tiesiogiai iš init.d katalogo, o, priklausomai nuo darbo lygio kreipiasi į katalogą `/etc/rc`*`lygis`*`.d`, pvz antro lygmens darbo atveju skriptų bus ieškoma `/etc/rc2.d` kataloge. `/etc/rc`*`lygis`*`.d` katalogai paprastai turi tik simbolines nuorodas į `/etc/init.d` kataloge esančius failus. Simbolinės nuorodos prasideda raide 'S' arba 'K', po to eina skaičius ir gale serviso, dažniausiai fizinio failo, pavadinimas, pvz. S45sshd. Kai init eina iš žemesnio darbo lygio į aukštesnį (startuoja) jis paleidžia visas 'S' programas didėjančia numerių tvarka. Ir atvirkščiai, kai init eina iš aukštesnio į žemesnį darbo lygmenį (baigia darbą) jis paleidžia visas 'K' programas mažėjančia numerių tvarka.

Norint įterpti startavimo programą reikia į /etc/init.d katalogą nukopijuoti startavimo failą

```
# cp ~/sshd /etc/init.d/sshd
```

ir sukurti simbolinę nuorodą į reikiamo lygio katalogą (visi tinklo servisai dažniausiai paleidžiami antrame lygmenyje)

```
# ln –s /etc/init.d/sshd /etc/rc2.d/S45sshd
# ln –s /etc/init.d/sshd /etc/rc2.d/K25sshd
```

Kai kuriose sistemose reikia įdėti nuorodą ir į rc6.d katalogą.

7.1.3.2 BSD startavimo programos

Nagrinėsime FreeBSD startavimo programas, kurios yra daugumos BSD sistemų atstovas. FreeBSD paleidžia `/etc/rc` programą, o po to visas `/etc/rc.`*`something`* programas. Jos yra paleidžiamos pagal iš anksto nustatytą tvarką nenaudojant jokių darbo lygių. Prieš tai init paleidžia tris konfigūracinius failus:

```
/etc/defaults/rc.conf
/etc/rc.conf
/etc/rc.conf.local
```

Šiuose konfigūraciniuose failuose gali būti nurodomos kitų startavimo programų vietos. Dažniausiai jie nurodo shell'o aplinkos kintamuosius, kuriuos naudoja vėliau startuosiančios programos.

7.2 Sistemos išjungimas

UNIX failų sistemų buferiai į diskus yra surašomi atsitiktinai; tai pagreitina I/O darbą, tačiau ir sukuria problemas netikėtai nutraukus sistemos darbą. Anksčiau UNIX sistemos buvo labai jautrios darbo nutraukimui; dabar jos yra atsparesnės, tačiau vis dėlto reikėtų stengtis baigti darbą gražiai. Staigus darbo nutraukimas gali baigtis failų sistemos taisymu arba net visišku duomenų praradimu.

Kai kuriose ne UNIX sistemose perkrovimas išsprendžia problemas, tačiau UNIX sistemose pradžioje reikia pagalvoti, o tik tada perkrauti. Perkrovimas dažniausiai reikalingas keičiant aparatūrinę įrangą. Kartais kai yra modifikuojami startavimo skriptai ir reikia įsitikinti ar sistema sėkmingai pakyla tenka taip pat perkrauti mašiną.

Sistemą išjungti galima keliais būdais:

1. išjungti energijos tiekimą mygtuku;
2. shutdown komanda. Tai yra saugiausias būdas. Šia komanda galima paprašyti sistemą šiek tiek palaukti baigiant darbą. Per tą laiką sistema siunčia pranešimus (a la wall) sistemos vartotojams apie darbo pabaigą. Tuo laikotarpiu nauji vartotojai negali prisijungti.
3. halt komanda. Tai yra paprastesnis ir grubesnis metodas. halt komanda įrašo įrašą į log'ą, nužudo visus nesisteminius procesus, palaukia kol failų sistemos susirašys savo buferius į diską ir nuleidžia branduolį.
4. reboot komanda. Ši komanda nuleidžia sistemą kaip halt, tačiau po to ją pakelia.
5. nusiųsti TERM signalą init procesui. Tai gali sukelti problemų, nes tada init tiesiog nužudo visus vaikinius procesus, servisus, tty'us ir grąžina sistemą į single-user režimą.
6. telinit komanda. Ji drastiškai pakeičia darbo lygį.
7. init nužudymas. Kai init procesui yra nusiunčiamas KILL signalas dauguma sistemų persikrauna, tačiau kai kurie branduoliai gali supanikuoti.

8 UNIX praktika

8.1 Praktikos pagrindai

8.1.1 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.

8.1.2 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>
```

8.1.3 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.

8.1.4 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.

8.1.5 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
```

8.1.6 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.

8.1.7 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).

8.1.8 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.

8.1.9 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

8.1.10 Pagrindinės komandos

8.1.10.1 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

8.1.10.2 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

8.1.10.3 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

8.1.10.4 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

8.1.10.5 Spausdinimas

8.6 lentelė. Spausdinimo komandos.

Komanda / sintaksė	Aprašymas
lpq / lpstat [options]	Išvesti informaciją apie spausdinamus darbus

lpr / lp [options] file lprm / cancel [options] pr [options] [file]	Spausdinti į nurodytą printerį Nutraukti spausdinimą arba išmesti darbą iš eilės Suformuoti failą spausdinimui

8.2 I/O valdymas

8.2.1 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

8.2.2 Failų (išvedimo / įvedimo) nukreipimas

8.2.2.1 /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
```

8.2.2.2 /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
```

8.2.3 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

8.3 Teksto apdorojimas

8.3.1 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/
```

8.4 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/*
```

8.4.1 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/
```

8.4.2 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 }
```

8.5 Kitos naudingos komandos

8.5.1 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

8.5.2 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

8.6 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.

8.6.1 Aplinkos kintamieji

8.6.1.1 /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`.

8.6.1.2 /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.

8.7 Shell programavimas

8.7.1 Shell programos (skriptai)

8.7.1.1 /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
```

8.7.1.2 /bin/csh

Programos interpretatoriaus eilutė yra ši:

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

8.7.2 Parametrų reikšmės

8.7.2.1 /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))
```

8.7.2.2 /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
```

8.7.3 Kintamieji

8.7.3.1 /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

8.7.3.2 /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

8.7.4 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

8.7.5 Here Document struktūra

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

8.7.6 Interaktyvus įvedimas

8.7.6.1 /bin/sh

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

8.7.6.2 /bin/csh

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

8.7.7 Funkcijos

```
ll(){ ls –la "$@"; }
```

8.7.8 Valdymo sakiniai

8.7.8.1 Sąlygos sakiniai

8.7.8.1.1 /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
```

8.7.8.1.2 /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
```

8.7.8.2 Ciklo sakiniai

8.7.8.2.1 /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
```

8.7.8.2.2 /bin/csh

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

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

8.7.9 Loginiai ir veiksmų operatoriai

8.7.9.1 /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

8.7.9.2 /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

8.8 C programavimas

8.8.1 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
```

8.8.2 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
```

8.8.3 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);
}
```

8.8.4 Procesų programavimas

8.8.4.1 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);
}
```

8.8.4.2 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	Kai pasiekiama riba procesas baigia

	dydis baitais, t.y. maksimalus brake adreso postūmis.	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
}
```

8.8.5 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();
}
```

8.8.6 IPC programavimas

8.8.6.1 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);
}
```

8.8.6.2 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);
}
```

8.8.6.3 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);
}
```

8.8.7 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);
```

8.8.8 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();
}
```

8.8.9 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);

}
```

8.8.10 Tinklo programavimas

8.8.10.1 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;
}
```

8.8.10.2 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;
}
```

8.8.10.3 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
```