

Programavimas Unix OS

Albertas Agejevas <albertas.agejevas@mif.vu.lt>

<http://www.mif.vu.lt/~alga/unix/unix.pdf>

Vilnius

2003

Turinys

1. Įvadas	4
1.1. Apie šį kursą	4
1.2. Tipografiniai susitarimai	4
2. Unix pradmenys	4
2.1. Istorija	4
2.2. Standartai	5
2.3. Unix ypatumai	6
2.3.1. Failų tipai	6
2.4. Filosofija	9
2.5. Kalbos pasirinkimas	11
3. Instrumentinės priemonės	12
3.1. man — pagrindinis informacijos šaltinis	12
3.2. Redaktoriai	12
3.3. gcc	13
3.4. make	14
3.5. indent	16
3.6. gdb	16
3.7. strace	16
3.8. automake, autoconf ir libtool	16
3.9. Konfigūracijos valdymo sistemos	18
3.9.1. Istorinė perspektyva	18
3.9.2. CVS	18
3.9.3. CVS alternatyvos	19
4. Perl, AWK ir sed	19
4.1. Šablonai	19
4.2. sed	20
5. Procesų valdymas	21
5.1. Kas yra procesas?	21
5.2. struct task_struct	21
5.3. Procesų API	21
5.3.1. fork(2)	21
5.3.2. exec*(3) šeima	22
5.3.3. wait(2) ir waitpid(2)	22
5.3.4. Procesų uid ir gid	22
5.3.5. Sesijos ir procesų grupės	23
5.3.6. Demonizacija	23
5.4. Signalai	24
5.5. Kaip gaudyti signalus	24
5.5.1. signal(2)	25
5.5.2. BSD semantika vs. SysV semantiką.	25
5.5.3. sigaction(2)	25

5.6. Kaip siųsti	26
6. Lizdai	26
6.1. Lizdai su ryšio nustatymu	27
6.2. Lizdai be ryšio nustatymo	28
6.3. Unix domeno lizdai	28
7. System V IPC	29
7.1. Pranešimai	30
7.2. Semaforai	30
7.3. Bendra atmintis	31
8. POSIX gijos (pthreads)	31
8.1. Gijų realizacijos Unix sistemose	32
8.1.1. Sun Solaris 2.x	32
8.1.2. LinuxThreads	32
8.2. NPTL	32
8.2.1. BSD pthreads	33
8.3. Gijų valdymo funkcijos	33
8.4. Mutexų valdymo funkcijos	34
8.5. Sąlygos kintamųjų valdymo funkcijos	34
8.6. Specifiniai gijų duomenys	35
9. Žodynėlis	36
Rodyklė	36
Literatūra	39

1. Įvadas

1.1. Apie šį kursą

Šis kursas skirtas kompiuterių mokslo trečio–ketvirto kursų studentams. Reikalingas programavimo principų supratimas bei C programavimo kalbos bei standartinės bibliotekos žinojimas. Pageidautina pažintis su Unix vartotojo aplinka.

Išklausęs šį kursą, studentas turėtų mokėti naudotis Unix programuotojo aplinkos įrankiais bei sugebėti rašyti programas panaudojančias Unix sistemų procesų valdymo, gijų, tinklo bei tarpprocesinio bendravimo paslaugas.

1.2. Tipografiniai susitarimai

Šiame konspekte `ls(1)`, `bind(2)` ir `sprintf(3)` pavidalo užrašai reiškia komandas, sistemes ir bibliotekines funkcijas. Skaičius skliausteliuose — man vadovo skyriaus numeris.

Interaktyvių sesijų iliustracijose riebiu šriftu spausdinamos vartotojo įvedamos komandos, o normaliu — programų spausdinami duomenys, pavyzdžiui:

```
root@fleksija:~# id
uid=0(root) gid=0(root) groups=0(root)
```

2. Unix pradmenys

2.1. Istorija

Parengiant šį skyrių buvo remtasi [HHU], [BSD], bei [TAOUP].

Unix raida prasidėjo 1969 metais *AT&T Bell Labs*, kai Ken Thompson ant jau atgyvenusios PDP-7 mašinos assembleriu pradėjo kurti operacinę sistemą, besiremdamas *Multics* projekto patirtimi ir bandydamas apjungti sėkmingus sprendimus iš kitų operacinių sistemų.

1972–1974 metais Thompson kartu su Dennis Ritchie perrašė Unix'ą C kalba. Nuo to laiko prasidėjo šito garsaus dueto (t.y Unix ir C, o ne Thompsono ir Ritchie) pasaulio užkariavimas.

Unix perrašymas C kalba buvo revoliucinis žingsnis. Unix buvo kaipmat perkelta į keletą kitų architektūrų, o tai atvėrė naujas galimybes: vienoda aplinka besikeičiant aparatūrai. Iki tol praktiškai su kiekvienu aparatūros atnaujinimu vartotojai gaudavo ir visiškai naują aplinką.

Kadangi tuo metu AT&T buvo telefonų kompanija, dėl antimonopolinių įstatymų jai buvo draudžiama uždirbinėti pinigų iš programų kūrimo, todėl į Unix nebuvo žiūrima kaip į verslo šaltinį. 1975 m. AT&T pasiūlė Unix kartu su išeities tekstais universitetams už labai mažą kainą. 1979m. Version Seven licencija universitetams kainavo \$100, tuo tarpu kai valstybinėms ir komercinėms įmonėms licencija kainavo \$21000.

Apie 1977 m. UCB (Kalifornijos universitetas Berklyje) pasiūlė savo Unix variantą, BSD (*Berkeley Software Distribution*). 1978 UCB nusipirko VAX'ą¹ ir parašė virtualios

¹DEC VAX (Virtual Address eXtention) — vienas iš pirmųjų, ir bene populiariausias, minikompiuteris su 32 bitų adresų erdve ir aparatine atminties apsauga.

atminties posistemę jam. Po to UCB gavo kontraktą iš DARPA² ir sukūrė tinklo palai-
kymą, greitą failų sistemą, bei padarė kitus patobulinius. 1983 išėjus sistema 4.2BSD
užtemdė net komercinės USL Unix versijas.

Po kurio laiko (1986) AT&T inkorporavo Berklio patobulinius į savo sistemą (Sys-
tem III ir System V), ir komercinis pasaulis vėl persimetė prie AT&T versijos. Bet BSD
kūrimas nesustojo. 1991 metais Berkli išvalė BSD išeities tekstus nuo AT&T kodo ir
išleido rezultata Net/2 pavadinimu. Po to sekė 4.4BSD, atsišakojo Net/Free/OpenBSD
projektai, bylinėjimasis su AT&T, ir galų gale išėjo 4.4BSD-Lite, kuri yra dabartinių BSD
sistemų pagrindas.

1984 m. buvęs MIT (Masačusetso technologijos instituto) dirbtinio intelekto labo-
ratorijos darbuotojas Richard M. Stallman (RMS) užsibrėžė tikslą sukurti laisvą Unix
kloną – GNU sistemą. Pirmiausia buvo sukurti Emacs tekstų redaktorius, bazinės Unix
komandos, C kompiliatorius ir kiti programavimo įrankiai. GNU projektas stokojo tik
branduolio. Ši niša netikėtai buvo užpildyta atsiradus Linux.

Kaip visiškai atskiras projektas 1991 metais suomių studento Linus Torvalds buvo
pradėtas kurti Linux branduolys. Kai darbas buvo viešai išleistas su GNU GPL licencija,
aplink jį susidarė gyvybinga bendruomenė, kuri tobulina ir palaiko Linux branduolį. Su-
dėjus Linux kartu su GNU ir BSD vartotojo programomis išėjo visai funkcionalus Unix
klonas. Kai kurie žmonės sako, kad tai yra dabartiniu metu progresyviausias Unix'as.
Neabejotina, kad jis sulaukia daugiausia spaudos ir IT industrijos gigantų dėmesio.

2.2. Standartai

Kas yra Unix? Operacinė sistema? Jei taip, tai kuri iš to daugelio atšakų ir variantų?

Vienas iš požiūrių yra tas, kad Unix — tai ne konkreti sistema, o apibendrinta sąsaja
— tiek vartotojo, tiek programinės. Prisilaikant tokios pozicijos, Linux irgi yra Unix.

Visgi kai kurios komercinės organizacijos teigia, kad Unix yra prekybinis ženklas ir
tik sertifikuotos sistemos gali būti taip vadinamos.

Yra dvi pagrindinės Unix šakos — BSD ir SysV. Jos tarpusavyje skiriasi daugeliu
smulkmenų. Kiekvienas gamintojas taip pat turėjo savo Unix'o versiją, ir tai sudarė ne-
patogumų. Buvo imtasi daugelio pastangų standartizuoti Unix. Ką dabar iš to turim, tai
POSIX standartai.

- POSIX.1 — taikomųjų programų API standartas.
- POSIX.2 — vartotojo aplinka, standartinės komandos.
- POSIX.1b — realaus laiko ir tarpprocesinio bendravimo interfeisai.
- POSIX.1c — lygiagreto programavimo interfeisai.

Yra ir daugiau POSIX standartų, bet kai kurie dar visai neratifikuoti, kiti neprisiję
gyvenime. Dar yra X/Open grupės standartai (XPG3, XPG4), kurie yra ANSI C ir POSIX
standartų patikslinimai.

Šiuolaikinė POSIX standartų redakcija yra neformaliai vadinama POSIX 2003, o ofi-
cialiai žinoma kaip *The Single UNIX Specification V. 3*, ISO/IEC 9945:2003 bei IEEE Std.
1003.1, 2003 redakcija. Ji yra laisvai prieinama Internetu [POSIX].

²DARPA (Defence Advanced Research Projects Agency) yra JAV kariškių agentūra, sukūrusi Interneto
pagrindus.

2.3. Unix ypatumai

Unix, ko gero, yra pirmoji sistema taip plačiai panaudojusi failo sąvoką. Praktiškai bet koks objektas operacinėje sistemoje yra arba procesas arba failas³. Kas turima omeny šiuo posakiu? Apžvelgsime, kokie yra galimi failų tipai. Bet pirma reikia suprasti, kas yra Unix failų sistema.

Visi failai „sudėti“ į hierarchinę katalogų struktūrą. Katalogų medis turi tik *vieną* šaknį. Nėra jokių A :, B :, C : skirtingiems įrenginiams. Kai norima prieiti prie failų sistemos kokiame nors įrenginyje, ta failų sistema turi būti prijungtama (primontuojama) prie kokio nors katalogo `mount (8)` komanda (t.y. `mount (2)` sisteminė funkcija). Prijungta failų sistema perdengia visą to katalogo, prie kurio ji buvo prijungta, turinį. Todėl dažniausiai failų sistemos montuojamos ant tuščių katalogų. Kartais sukuriamos direktorijos `/mnt / *` skirtingoms failų sistemoms laikinai montuoti.

Kaip yra saugomas failas failų sistemoje? Svarbiausia sąvoka yra *i-mazgas* (angl. *inode*, nuo „*index node*“, indeksinis mazgas). I-mazgas yra svarbiausias failo blokas, kuris laiko metainformaciją apie failą. I-mazge yra laikomi failo tipas, priėjimo teisės, savininkas, grupė, didysis ir mažasis įrenginio numeris (turi prasmės tik įrenginio failo tipui), paskutinio kreipimosi, paskutinio modifikavimo ir paskutinio i-mazgo modifikavimo laikai. Failo vardas nėra laikomas inode'e, bet laikomas kataloguose. Visi i-mazgai yra sunumeruoti ir failų sistema juos gali surasti pagal numerį⁴. Paprastai, maksimalus i-mazgų skaičius yra fiksuotas ir nustatomas failų sistemos sukūrimo metu.

Platesnį failų sistemos paaiškinimą galite rasti The Linux Kernel LDP knygoje [TLK].

2.3.1. Failų tipai

Taigi, štai kokie failų tipai yra Unix sistemose:

Paprastas failas yra visiems pažįstamas vardą turintis duomenų rinkinys.

Katalogas yra ne kas kita, kaip failas, laikantis i-mazgų numerių ir failų vardų lentelę.

Katalogai kuriami `mkdir (2)` sisteminė funkcija. Paprastomis failų operacijomis prie katalogo turinio prieiti negalima, tam yra specialios funkcijos `opendir (3)`, `readdir (3)`, `closedir (3)` ir t.t.

Įrenginiai. Tai yra tokie objektai failų sistemoje, kurie patys duomenų nelaiko, bet suteikia sąsają su kažkokiais draiveriais sistemos branduolyje (Kaip PRN MS DOS'e). Pagal darbo su įrenginiu pobūdį, įrenginių failai skirstomi į nuoseklius (angl. *character device*) ir blokinius (angl. *block device*). Nuoseklaus įrenginio pavyzdžiu galėtų būti terminalo įrenginys, o blokinio — magnetinių diskų įrenginys. Iš pirmojo galima skaityti ir rašyti po vieną baitą, o pastarasis skaito ir rašo informaciją mažiausiai po vieną bloką, dažniausiai 512 baitų. Be to, blokiniuose įrenginiuose galima kreiptis į bet kurį bloką, o nuosekliuose įrenginiuose gali ir nebūti laisvo adresavimo galimybės.

³Iš konteksto iššokanti išimtis yra tinklo interfeisai. Jie turi vardus savo erdvėje, nors savo esmė yra panašūs į įrenginių failus. Šis trūkumas, visgi, yra ištaisytas *Plan 9* sistemoje.

⁴Visgi išorinis API surasti failą pagal jo i-mazgo numerį neegzistuoja. Galima gauti tik tai turimo failo i-mazgo numerį.

Įrenginio failas yra priskiriamas konkrečiam draiveriui pagal įrenginio tipą (nuoseklus arba blokinis) bei didįjį ir mažąjį įrenginio numerį, kurie laikomi įrenginio failo i-mazge.

Įprasta, kad visi įrenginių failai sudėti /dev kataloge. Be įrenginių, atitinkančių tikrus aparatinius įrenginius, yra ir „virtualūs“ įrenginiai, tokie kaip /dev/null bei /dev/zero. Pirmasis iš karto praneša apie failo pabaigą, kai bandoma iš jo skaityti, bei sėkmingai priima viską, kas į jį rašoma, o antrasis grąžina nulių seką, kai iš jo skaitoma.

```
alga@fleksija:~$ dd if=/dev/zero of=/dev/null bs=1K count=1024
1024+0 records in
1024+0 records out
1048576 bytes transferred in 0.002828 seconds (370787333 bytes/sec)
```

Ši komanda perkopijuoja 1 MB nulių. Šio kopijavimo greitis parodo branduolio vidines sąnaudas darbui su failais, kadangi jokio fizinio įvedimo-išvedimo ši komanda nevykdo.

Įrenginių failai kuriami komanda `mknod(1)`, bei sisteminė funkcija `mknod(2)`. Pavyzdžiui, minėti įrenginių failai gali būti sukurti tokiomis komandomis:

```
root@fleksija:~# mknod /dev/zero c 1 3
root@fleksija:~# mknod /dev/zero c 1 5
root@fleksija:~# ls -l /dev/null /dev/zero
crw-rw-rw-  1 root    root      1,   3 Aug 30 18:21 /dev/null
crw-rw-rw-  1 root    root      1,   5 Aug 30 18:21 /dev/zero
```

Kaip galima pastebėti, komanda `ls -l` išveda įrenginio failo tipą (b arba c) eilutės pradžioje, bei įrenginio didįjį ir mažąjį numerį vietoj failo dydžio.

Nuorodos. Simbolinė nuoroda (angl. *symbolic link*, *symlink*) tiesiog pratęsia kelią, kuriuo reikia kreiptis (gali būti failas ar katalogas). Pavyzdžiui, tarkime, yra tokia nuoroda:

```
alga@fleksija:~$ ls -l /usr/lib/X11
lrwxrwxrwx  1 root root 16 Sep 11 20:53 /usr/lib/X11 -> ../X11R6/lib/X11
```

Tada kreipiantis į failą /usr/lib/X11/xkb/symbols/lt iš tiesų bus kreipiamasi į /usr/lib/../../X11R6/lib/X11/symbols/lt, arba, normalizavus šį kelią, į /usr/X11R6/lib/X11/symbols/lt.

Simbolinės nuorodos kuriamos komanda `ln` su raktu `-s`. Pavyzdžiui, norint sukurti aukščiau parodytą nuorodą, reikia įvykdyti tokias komandas:

```
root@fleksija:~# ln -s ../X11R6/lib/X11 /usr/lib/X11
```

Programose simbolinėms nuorodoms kurti yra naudojama `symlink(2)` sisteminė funkcija.

Yra ir kitas nuorodų tipas, vadinamos kietos nuorodos (angl. *hard link*). Jos kuriamos komanda `ln` be rakto `-s`, bei sistemos funkcija `link(2)`. Šios nuorodos, skirtingai nuo simbolių, nėra ypatingas failo tipas, tai tiksliai nauji įrašai direktoriuose, rodantys į tą patį i-mazgą.

Paminėtina, kad ir sisteminė funkcija failo ištrynimui vadinasi `unlink(2)`. Ši funkcija pašalina failą iš katalogo, o jis ištrinamas iš disko tik tuo atveju, jei jo kietų nuorodų skaičius tampa lygus nuliui.

```
alga@fleksija:~$ echo turinys > pvz
alga@fleksija:~$ ls -l pvz*
-rw-rw-r-- 1 alga alga 8 Dec 13 22:19 pvz
alga@fleksija:~$ ln pvz pvz2
alga@fleksija:~$ ls -l pvz*
-rw-rw-r-- 2 alga alga 8 Dec 13 22:19 pvz
-rw-rw-r-- 2 alga alga 8 Dec 13 22:19 pvz2
alga@fleksija:~$ chmod o-r pvz2
alga@fleksija:~$ ls -l pvz*
-rw-rw---- 2 alga alga 8 Dec 13 22:19 pvz
-rw-rw---- 2 alga alga 8 Dec 13 22:19 pvz2
alga@fleksija:~$ rm pvz
alga@fleksija:~$ ls -l pvz*
-rw-rw---- 1 alga alga 8 Dec 13 22:19 pvz2
```

Failo kietų nuorodų skaičius yra rodomas `ls -l` komandos išvedamos informacijos antrame stulpelyje. Pastebėkime, kad visos to paties failo nuorodos turi bendrą priėjimo teisių ir priėjimo laiko informaciją, nes ji yra laikoma i-mazge.

Įvardinti konvejeriai (angl. *named pipes*). Kitaip vadinami FIFO. Konvejeris — virtualus kanalas, kurį vienas procesas gali atsidaryti rašymui, o kitas — skaitymui. Kol nebus kas nors įrašyta, skaitytojas blokuosis. Rašytojas taip pat blokuosis, kai užsipildys konvejerio buferis.

Įvardinti konvejeriai taip pat kuriami sistetine funkcija `mknod(2)` arba komanda `mknod(1)`:

```
alga@fleksija:~$ mknod fifo p
alga@fleksija:~$ ls -l fifo
prw-rw-r-- 1 alga alga 0 Dec 13 22:04 fifo
alga@fleksija:~$ echo "Laba diena" > fifo
```

Tada atskirame komandų interpretatoriuje galima paleisti procesą, kuris skaitytų iš šio įvardinto konvejerio:

```
alga@fleksija:~$ cat fifo
Laba diena
```

Komanda `echo "Laba diena"` pasibaigs tik tada, kai bus paleista komanda `cat fifo`.

Lizdai (sockets). Tai yra tarpprocesinio bendravimo ir tinklo API pasireiškimas. Šio tipo failai — `AF_UNIX` protokolų šeimos lizdai. Jų atsiradimą galima pailiustruoti šia Python programėle:

```
#!/usr/bin/env python
from socket import socket, AF_UNIX, SOCK_STREAM

sock = socket(AF_UNIX, SOCK_STREAM)
sock.bind("/tmp/lizdas")
sock.close()
```

Šią programą įvykdžius failų sistemoje atsiras lizdo failas:


```
alga@fleksija:~$ ls -l /tmp/lizdas
srwxrwxr-x    1 alga    alga
```

0 Nov 17 00:21 /tmp/lizdas

Plačiau apie lizdus bus kalbama 6 skyriuje.

Labiausiai failo sąvokos platumą pavaizduoja skirtingų virtualių failų sistemų egzistavimas. Tai yra `/proc` failų sistema, NFS, Portal failų sistema ir t.t. `/proc` failų sistema atrodo kaip paprastų tekstinių failų ir katalogų rinkinys, bet iš tiesų jos failai yra generuojami branduolio kai į juos kreipiamasi. Juose pateikiama pagal proceso ID sutvarkyta informacija apie procesus, o taip pat informacija apie branduolio skirtingų posistemų būklę. Per kai kuriuos `/proc` failų sistemos failus galima ir derinti tų posistemų veiklą į juos rašant. Linux 2.6 versijoje papildoma informacija iškelta į atskirą `/sys` failų sistemą.

NFS (Network File System) yra Sun firmos sukurta tinklinė failų sistema optimizuota greičiui, naudojanti UDP protokolą perdavimui, bei teisėtai susilaukusi tokių pravardžių, kaip No File Security, Nightmare File System ir taip toliau.

PortalFS yra BSD sistemų draiveris, leidžiantis vartotojui pateikti savo „failų sistemos“ kodo realizaciją ir tokiu būdu pateikti bet kokią informaciją per standartinį failų sistemos interfeisą.

2.4. Filosofija

Unix yra idėjiškai stipri ir konceptualiai vieninga tradicija. Panagrinėkime, ką apie ją kalba jos pradininkai [TAOUP]. Doug McIlroy, konvejerių išradėjas, apie Unix filosofiją rašė taip:

(i) Stenkis, kad kiekviena programa darytų vieną dalyką gerai. Naujam darbui kurk naujai, užuot painiojęs senas programas pridėdamas naujas galimybes.

(ii) Tikėkis, kad kiekvienos programos išvestis taps kitos, dar nežinomos, programos įėjimi. Nevelk išvesties nereikalinga informacija. Venk griežtų stulpelinių ar dvejetainių įvesties formatų. Nereikalauk interaktyvios įvesties.

(iii) Projektuok ir kurk programas, net operacines sistemas, kad jas galima būtų pabandyti kuo anksčiau, geriausiai — savaičių bėgyje. Nedvejodamas išmesk nerangias dalis ir pastatyk jas iš naujo.

(iv) Naudok įrankius, o ne nekvalifikuotą pagalbą, programavimo darbui palengvinti, net jei teks nukrypti nuo kelio tam, kad sukurtum įrankius, be to, tikėkis dalį jų išmesti, kai baigsi jais naudotis.

Vėliau jis tai apibendrino taip:

Štai Unix filosofija: rašyk programas, kurios daro vieną dalyką ir daro jį gerai. Rašyk programas, kurios dirba kartu. Rašyk programas, kurios apdoroja tekstinius srautus, nes tai universalioji sąsaja.

Rob Pike, C programavimo ekspertas, pateikia kiek kitą perspektyvą:

Taisyklė 1. Neįmanoma nuspėti, kur programa leis laiką. Greitaveikos siaurumos (bottlenecks) atsiranda stebinančiose vietose, taigi nebandyk spėlioti ir optimizuoti, kol neįrodei, kad toje vietoje tikrai yra siauruma.

Taisyklė 2. Matuok. Neoptimizuok greičiui, kol nematavai, ir net tada neoptimizuok, jei nei viena programos dalis neužgožia likusių.

Taisyklė 3. Gudrūs algoritmai lėti, kai n yra mažas, o n dažniausiai yra mažas. Kol nežinai, kad n dažnai bus didelis, nebūk gudrus. (Net jei žinai, kad n bus didelis, naudok taisyklę 2).

Taisyklė 4. Gudrūs algoritmai turi daugiau klaidų, nei paprasti, ir juos daug sunkiau įgyvendinti. Naudok paprastus algoritmus ir paprastas duomenų struktūras.

Taisyklė 5. Duomenys dominuoja. Jei pasirinkai teisingas duomenų struktūras ir gerai suorganizavai dalykus, algoritmai dažniausiai bus akivaizdūs. Duomenų struktūros, o ne algoritmai, yra svarbiausi programavime.

Taisyklė 6. Taisyklės 6 nėra.

Ketvirtąją taisyklę glausčiausiai išreiškė Unix sukūrėjas Ken Thompson:

Abejoti — naudok grubią jėgą. (*When in doubt, use brute force.*)

Naudinga taip pat pažvelgti, ką apie Unix stovyklą sakė žmonės iš kitų „kultūrų“. Joel Spolsky, garsus Windows programuotojas rašantis straipsnius apie programavimą ir programų inžineriją, pastebi, kad Unix tradicijoje stengiamasi palengvinti gyvenimą programuotojui, o Windows tradicijoje yra pirmiausia galvojama apie galutinį vartotoją [Spolsky]:

Tarkime, paimate Unix programuotoją ir Windows programuotoją ir duodate kiekvienam užduotį sukurti tą pačią programą vartotojui. Unix programuotojas sukurs komandinės eilutės ar tekstinės sąsajos branduolį, ir, vėliau, kaip pagerinimą, sukurs grafinę sąsają, kuri valdys branduolį. Tokiu būdu, pagrindinės programos funkcijos bus prieinamos kitiems programuotojams, kurie galės kviesti programą komandinėje eilutėje ir skaityti rezultatus tekstiniu pavidalu. Windows programuotojas bus linkęs pradėti grafinę sąsają, ir tik vėliau, kaip pagerinimą, pridės skriptų kalbą, kuri leis automatizuoti grafinę sąsają. Tai yra adekvatu kultūroje, kur 99,999% vartotojų nėra programuotojai ir nenori jais tapti.

Kaip priešpastatymą idealistinei Lisp „teisingo dalyko“ kultūrai žymus Lisp programuotojas Richard P. Gabriel pateikė karikatūrizuotą Unix kūrimo vertybių sistemą — „blogiau yra geriau“ [WIB]:

- Paprastumas — projektas (*design*) turi būti paprastas, tiek jo įgyvendinimas, tiek sąsaja. Svarbiau, kad būtų paprastas įgyvendinimas, o ne sąsaja. Projektuojant, paprastumas yra svarbiausias dalykas.
- Korektiškumas — projektas turi būti korektiškas visais stebimais aspektais. Svarbiau būti paprastam, nei korektiškam.

- Nuoseklumas (*consistency*) — projektas neturi būti per daug nenuoseklus. Nuoseklumas gali būti paaukotas vardan paprastumo kai kuriais atvejais, bet geriau tiesiog išmesti tas projekto dalis, kurios apdoroja retesnes sąlygas, negu įvelti įgyvendinimo sudėtingumą ar nenuoseklumą.
- Pilnumas — projektas turi padengti tiek svarbių situacijų, kiek yra praktiška. Visi sveikai numatomi atvejai turi būti padengti. Pilnumas gali būti aukojamas vardan bet kurios kitos vertybės. Tiesą sakant, pilnumas turi būti aukojamas, kada įgyvendinimo paprastumas yra pavojuje. Nuoseklumas gali būti aukojamas siekiant pilnumo, jei išlaikomas paprastumas, ypatingai bevertis sąsajos nuoseklumas.

Parafrazuojant šiuos principus, paprastas 90% sprendimas yra geriau už sudėtingą ir labai sunkiai įgyvendinamą 99,9% sprendimą. Jų autorius per daugelį metų taip ir neapsisprendė, ar „blogiau yra geriau“ yra tiesa ar ne.

2.5. Kalbos pasirinkimas

Unix šeimos sistemose ypač gausu instrumentinių priemonių. Dažniausiai tą patį tikslą galima pasiekti skirtingomis priemonėmis, taigi kiekvienam uždaviniui reikia pasirinkti tinkamiausią įrankį.

Iš visų programavimo kalbų Unix sistemoje, be abejo, svarbiausios yra C ir apvalkalo kalba. Tai yra skirtingi poliai — žemo lygio efektyvi sisteminė kalba ir aukšto lygio vartotojo komandų interpretatorius. Tarpe tarp jų skirtingas nišas ar ištisas sritis užima visokios interpretuojamos kalbos.

Sistemos programavimas, draiveriai, programos, reikalaujančios greičio yra rašomos C kalba. C privalumai — universalumas, portabilumas. C kompiliatorius yra praktiškai kiekvienoje Unix mašinoje. Galima rasti bibliotekų skirtingiausioms sritims, nuo XML manipuliavimo iki skaitinių algoritmų. Visgi, C yra ne geriausias įrankis mažoms pagalbinėms priemonėms dėl savo žemo abstrakcijos lygio, kompiliuojamos kalbos prigimties ir kitų priežasčių. Net didelėse programose, kuriose vykdymo greitis nėra svarbiausias reikalavimas, verta pagalvoti apie alternatyvas C, turint galvoje šios kalbos saugumo problemas, silpną moduliarizacijos laipsnį ir kitus trūkumus.

Priemonės, automatizuojančios keletą komandų paleidimą, manipuliacijos failais, atsarginių kopijų darymas, sistemos administravimo pagalbinės priemonės ir panašūs dalykai yra rašomi apvalkalo (angl. *shell*) skriptais. Apvalkalas yra ne tik komandinis vartotojo interfeisas, bet ir aukšto lygio kalbos interpretatorius, pateikiantis primityvus darbui su failais, procesais bei paprastoms manipuliacijoms teksto eilutėmis. Apvalkalo skriptais įmanoma padaryti labai daug dalykų, net labai sudėtingas programas, bet jei *shell* skriptas išauga ilgesnis, nei kokios 100 eilučių kodo, yra verta pamąstyti apie kitą kalbos pasirinkimą.

Tarp C/C++ ir apvalkalo skripto yra daugybė kitų interpretuojamų programavimo kalbų — Python, Perl, TCL/Tk, Ruby, AWK ir t.t. Jos visos turi savo stipriąsias ir silpnąsias puses, o taip pat stipriai persidengia savo taikymo sritimis. Pastaruoju metu Unix taikomųjų programų pasaulyje paplito ir Java, tačiau jina nesiūlo objektyvių pranašumų prieš, tarkime, Python.

Platesnės diskusijos šia tema siūlau ieškoti [TAOUP] ketvirtame skyriuje.

3. Instrumentinės priemonės

Šiame skyriuje kalbama apie programuotojo darbo įrankius Unix pavidalo sistemose.

3.1. `man` — pagrindinis informacijos šaltinis

Kaip jūs jau tikriausiai žinote, lengviausiai pasiekama dokumentacija apie beveik bet kurią komandą ar funkciją yra `man` vadovai. Patogumo dėlei tie vadovai suskaidyti į skyrius. Linux sistemose `man` vadovų skyriai yra tokie:

- 1 Vartotojo komandos (`ls(1)`);
- 2 Sistemos iškvietimai (`open(2)`);
- 3 Skirtingų API funkcijos (`fopen(3)`);
- 4 Įrenginiai ir draiveriai (`zero(4)`);
- 5 Failų formatai (`passwd(5)`);
- 6 Žaidimai (`fortune(6)`);
- 7 Kiti (`latin1(7)`, `ip(7)`);
- 8 Sistemos administravimo komandos (`mount(8)`).

Toks paskirstymas nėra taisyklė, skirtingose sistemose sekcijos 4, 5, 6 ir 7 būna sukaitytos, o taip pat kartais būna ir papildomų skyrių.

`man (1)` komanda naudotis yra pakankamai paprasta:

```
man komanda
```

Dažniausiai to pakanka, bet kartais norisi puslapio iš kitos sekcijos, negu duoda `man` — pvz. `mknod (2)` vietoj `mknod (1)`. Tada galima nurodyti sekciją:

```
man 2 mknod
man -S 2:3 mknod
```

3.2. Redaktoriai

Ko gero, svarbiausias programuotojo įrankis yra teksto redaktorius. Unix'e jų yra daug ir įvairių. Labiausiai paplitę, be abejo, yra `emacs` ir `vi`.

`vi` yra pirmas Unix'e vizualus (*full-screen*) redaktorius. Jo pranašumas prieš konkurentus yra tas, kad jis yra visose Unix sistemose. Ne be keistenybių, bet lengvas, greitas ir „išprogramuojantis į pirštus.“ Labai keistas naujokams. Bet vis vien rekomenduoju išmokti juo naudotis.

`Emacs` yra daugiau negu redaktorius. `Emacs` labiau yra Unix integruota programavimo aplinka. Didesnė dalis redaktoriaus logikos parašyta `Elisp`'u — `LISP` dialektu. Iš `Emacs` galima paleidinėti skirtingas programas: `make`, debuggerius, apvalkalą. Yra skirtingų programų `Elisp`'u — žiniatinklio naršyklė, `IRC` klientas, el. pašto klientas, naujienų skaityklė, failų tvarkyklė ir t.t. Turi integruotą dokumentacijos sistemą. Palaiko sintaksės spalvinimą ir specialius režimus begalei skirtingų failų tipų. `Emacs` turi labai lanksčią ir

visaapimančią vartotojo interfeisą, na o jei jo nepakanka, galima imtis kūrybos Elisp'u. Bet tikriausiai tai, ko jums reikia, jau yra parašyta.

Būtent Emacs redaktoriumi Jūsų autorius dabar rašo šitą konspektą.

Be šių dviejų priešingų programų, Unix'ams yra parašyta labai daug teksto redaktorių. Paprasčiausi būtų `pico`, jo laisvas klonas `nano`, `joe`, `jed`. Naujokams iš DOS/Windows pasaulio rekomenduočiau `mcedit`, kuris *labai* panašus į įprastus Norton Commander ar FAR vidinius redaktorių.

Be tradicinių redaktorių yra ir keletas IDE projektų. Yra perkelta net komercinių IDE. Bet tarp profesionalių programuotojų jie nėra paplitę ir tikriausiai niekada ir nebus.

3.3. gcc

C kompiliatorius Unix sistemose tradiciškai vadinamas `cc`, o GNU C kompiliatorius išskviečiamas komanda `gcc`. Kaip būtinas parametras kompiliatoriui turi būti perduotas išeities teksto failo (failų) vardas. Jei neperduoti kitų parametrų, išeities tekstas bus sukompiliuotas į vykdomąjį failą `a.out`.

```
[alga@peleda t]$ cat hello.c
main() {
    printf("Hello world!\n");
}
[alga@peleda t]$ gcc hello.c
[alga@peleda t]$ ls
a.out* hello.c
[alga@peleda t]$ ./a.out
Hello world!
```

Taip pat galima nurodyti tokius parametrus:

- c** Tik kompiliuoti, iš `failas.c` sukompiliuoja `failas.o` objektinį modulį
- o failas** Nurodo, kad galutinis produktas (objektinis modulis arba vykdomasis failas) turi būti pavadintas `failas`.
- O** Nurodo optimizacijos lygį. `-O` nurodo optimizuoti, `-O0` nurodo neoptimizuoti, o `-O2` nurodo stiprią optimizaciją.
- W** (gcc specifiška) Nurodo, kokie perspėjimai (*warnings*) turėtų būti išvedami. `-Wall` reiškia rodyti visus svarbius perspėjimus.
- pedantic** Nurodo pedantiškai sekti ANSI C reikalavimus ir išvesti perspėjimus neatitikimo standartui atveju.

```
[alga@peleda t]$ gcc -Wall -pedantic hello.c -o hello
hello.c:1: warning: return-type defaults to 'int'
hello.c: In function 'main':
hello.c:2: warning: implicit declaration of function 'printf'
hello.c:3: warning: control reaches end of non-void function
[alga@peleda t]$ ls
a.out* hello* hello.c
```

- S** Sustoti prieš asamblavimo stadiją. Assemblerinis tekstas išvedamas į failą `failas.s`

- E Sustoti po preprocesavimo. Preprocesuotas kodas išvedamas į standartinį išvedimą.
- I Nurodo *include path*, t.y. kelią, kuriame ieškomi antraščių (*header*) failai.
- L Nurodo *library path*, t.y. kelią, kuriame ieškomi bibliotekų failai.
- l Linkeriui perduodama opcija. Dažniausiai tai būna dinaminių bibliotekų vardai.

3.4. make

`make` yra programa, skirta automatizuoti kompiliavimo procesą. Kai projektą sudaro vos daugiau, nei pora failų, rankinis kompiliavimo komandų rašymas gali pakankamai įgrysti, o surašius jas į apvalkalo skriptą kaskart būtų be reikalo perkompiliuojami failai, kurie nepasikeitė.

`make` pagal tam tikrą taisyklių rinkinį, dalis kurių yra įsiūta į pačią programą, o dalį galima nurodyti, seka, kokie produktai (targets) priklauso nuo pasikeitusių išeities failų, ir perkompiliuoja tik tai, ko reikia. Programos veikimas reguliuojamas specialiu failu, vadinamu `Makefile` arba `makefile`. Jame nurodomi skirtingi kintamieji (makrokomandos), išeities tekstų failai, rezultatai ir taisyklės, kokiais žingsniais gauti rezultatus iš išeities failų.

Pademonstruokim pavyzdžiu. Turime nedidelę programėlę. Jos išeities tekstą sudaro tokie failai:

```
[alga@peleda lines]$ ls *.[ch]
balls.h  handlers.c  hiscore.c  lines.h    rules.c
child.c  handlers.h  hiscore.h  menu.c     rules.h
child.h  hello.c     lines.c    menu.h     smallballs.h
```

Jos `Makefile`’as yra pateiktas iliustracijoje 15 puslapyje.

Taigi, kaip galite atspėti pažiūrėję į šį pavyzdį, makrokomandos apibrėžimas atrodo kaip

```
vardas = reikšmė
```

Makrokomandos panaudojimas atrodo, kaip

```
$(vardas)
```

Na, o taisyklė atrodo taip:

```
tikslas: priklausomybės
→ komanda
→ komanda
→ ...
```

Be to, net tradicinis BSD `make` palaiko neišreikštines taisykles (kaip ta `.c.o`, t.y. kaip iš `*.c` failo gauti `*.o`), kintamųjų pakeitimą pagal šabloną (kaip `$(SRC:.c=.h)`), o GNU `Make` turi dar daugiau galingų savybių, kaip patobulinti šablonai ir vidinės funkcijos (kaip ta `$(shell ...)`).

Daugiau informacijos apie GNU `Make`, jos parametrus, sintaksę, vidines funkcijas ir kita galite sužinoti `make info` puslapyje: **info make**.

```

# Makefile for lines
#
# Kaip jau pastebėjote, komentaras yra bet kas nuo
# simbolio '#' iki eilutės pabaigos
#
# → reiškia TAB simbolį. 8 tarpai čia netiks!

# Čia yra makrokomandų apibrėžimai
CC      = gcc
CFLAGS  = $(shell gtk-config --cflags ) -g -Wall
LIBS    = $(shell gtk-config --libs )
SRC      = lines.c rules.c child.c handlers.c \
           menu.c hiscore.c
HDR      = $(SRC:.c=.h)
OBJS    = $(SRC:.c=.o)

# Čia yra pirmoji taisyklė. Ji nurodo, kad
# targetas 'all' priklauso nuo targetų tags ir lines
all: tags lines

lines: $(OBJS)
→ $(CC) $(LIBS) $(OBJS) -g -o lines

.c.o: *.c $(HDR)
# Štai šitą taisyklę make žino pats:
#→ $(CC) $(CFLAGS) -c *.c

# Nurodom, kad visi išeities teksto failai priklauso
# nuo visų antraščių
$(SRC): $(HDR)

# Šita taisyklė skirta išvalyti visokias šiukšles
clean:
→ rm -f *.o lines *~ core TAGS

# TAGS yra duomenys vi ir emacs redaktoriams apie
# tai, kur rasti skirtingų C funkcijų ir kintamųjų
# apibrėžimus
tags: TAGS
TAGS: $(SRC) $(HDR)
→ etags $(SRC) $(HDR)
# EOF

```

1 pav. Makefile pavyzdys

3.5. indent

`indent` yra gana paprasta programėlė, kurios vienintelis tikslas – gražinti C išeities kodų formatavimą. Atsirado BSD sistemose, GNU, kaip visada, siūlo savo patobulintą versiją, kuri pilnai emuliuoja ir originaliosios elgesį.

`indent` turi labai daug opcijų kiekvienam kodo formatavimo aspekto nustatymui, bet yra ir keletas iš anksto paruoštų režimų. Opcijos `-orig`, `-gnu` ir `-kr` įjungia atitinkamai originalaus BSD `indent`'o, GNU ir K&R formatavimo stilių.

3.6. gdb

Tai yra debuggeris (angl. *debugger*) su patogia komandinės eilutės sąsaja.

Nors GDB interfeisas ir gali būti neįprastas programuotojui su Windows grafinių debuggerių patirtimi, bet savo funkcionalumu GDB nenusileidžia kitiems debuggeriams. Ir jo komandinis interfeisas, beje, yra patogus ir efektyvus, kai su juo susipažįsti.

Norint efektyviai naudotis debuggeriu reikia, žinoma, kad debuginama programa būtų sukompiliuota su opcija `-g`. Nors, iš debuggerio galima išlupti naudos net jei sukompiliuota buvo ir be `-g`. Pagal nutylėjimą į vykdomuosius failus visgi įtraukiama simbolių informacija, bet ji gali būti „atsegama“ paleidžiant komandą `strip`. Kitais žodžiais, su `-g` jūs gausite visus simbolių vardus ir eilučių numerių informaciją, be jos gausite funkcijų vardus, o po `strip` komandos iškvietimo galėsite operuoti tikrai adresais.

Elementariausias debuggerio pritaikymas — pomirtinė analizė. Kai „lūžta“ kokia nors programa, yra sugeneruojamas `core` failas. Tai yra programos atminties `dump`'as. Priklausomai nuo apvalkalo nustatymų, `core` failas gali būti ir negeneruojamas. Maksimalus sukuriama `core` failo dydis yra reguliuojamas apvalkalo komanda `ulimit`.

3.7. strace

Šis įrankis trasuoja visus debuginamos programos sistemos kvietimus ir išveda juos gražiu pavidalu. Parodo simbolines argumentų ir grąžinamų rezultatų reikšmes. Nepakeičiamas įrankis, jei kokia nors programa „lūžta“ dėl nepaaiškinamų priežasčių, o jos išeities tekstų nėra.

`strace` komandos išvedamos informacijos pavyzdį galite surasti iliustracijoje 17 puslapyje.

Naudinga opcija yra `-o failas`, kuri nukreipia visą išvedamą informaciją į failą. Tokiu būdu programos išvedami pranešimai nepaskęsta `strace` informacijos sraute. `strace` turi dar daug visokių opcijų išvedamos informacijos kontroliavimui.

`strace` praverčia, jei norime, pavyzdžiui, sužinoti, kokius failus atidarinėja programa. Kokia tvarka yra nuskaitomi konfigūracijos failai, ar kas nors panašaus. Tada tiesiog `strace` rezultatus perfiltruojam `grep` komanda:

```
alga@peleda:~/unix$ strace -o log true  
alga@peleda:~/unix$ grep open log | less
```

3.8. automake, autoconf ir libtool

[AAL] Šitie įrankiai yra skirti palengvinti portabilių programų rašymą. Šio įrankių rinkinio panaudojimas juokais vadinamas „autokonfiskacija“. Idealiai, „autokonfiskuota“


```

[alga@peleda unix]$ strace true
execve("/bin/true", ["true"], [/* 57 vars */]) = 0
brk(0) = 0x8049cd4
open("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=19688, ...}) = 0
mmap(NULL, 19688, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40015000
close(3) = 0
open("/lib/libc.so.6", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0755, st_size=4223971, ...}) = 0
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200\204"... , 4096) = 4096
mmap(NULL, 1025596, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x4001a000
mprotect(0x4010d000, 30268, PROT_NONE) = 0
mmap(0x4010d000, 16384, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0xf2000) = 0x4010d000
mmap(0x40111000, 13884, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40111000
close(3) = 0
mprotect(0x4001a000, 995328, PROT_READ|PROT_WRITE) = 0
mprotect(0x4001a000, 995328, PROT_READ|PROT_EXEC) = 0
munmap(0x40015000, 19688) = 0
personality(PER_LINUX) = 0
getpid() = 1338
brk(0) = 0x8049cd4
brk(0x8049d0c) = 0x8049d0c
brk(0x804a000) = 0x804a000
open("/usr/share/locale/locale.alias", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=2174, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40015000
read(3, "# Locale name alias data base.\n#"... , 4096) = 2174
brk(0x804b000) = 0x804b000
read(3, "", 4096) = 0
close(3) = 0
munmap(0x40015000, 4096) = 0
open("/usr/share/locale/lt/LC_ALIAS", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/share/locale/lt/LC_COLLATE", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=29524, ...}) = 0
mmap(NULL, 29524, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40115000
close(3) = 0
open("/usr/share/locale/lt/LC_CTYPE", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=10428, ...}) = 0
mmap(NULL, 10428, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40015000
close(3) = 0
_exit(0) = ?

```

2 pav. strace panaudojimo pavyzdys

programa kompiliuojasi bet kurioje Unix pavidalo platformoje. `autoconf` ir `auto-make` pagal specifikacijas sugeneruoja `configure` skriptą, kuris išanalizuoja reikiamas sistemos savybes, patikrina, ar egzistuoja reikiamos bibliotekos, ir sugeneruoja `Makefile`'us, atitinkančius aplinką ir `config.h`, perduodantį programai `#define`'us, pranešančius apie atrastas aplinkos ypatybes.

3.9. Konfigūracijos valdymo sistemos

Konfigūracijos valdymas yra svarbi veikla bet kokiame programiniame projekte. Kartais prireikia grįžti prie ankstesnės versijos, jei kokie nors eksperimentai nepavyksta. Jei projekte dirba keletas žmonių, jiems reikia derinti savo veiksmus ir daryti savo darbo dalį taip, kad ji integruotųsi su kolegų darbu. Kai išleidžiama laida, programos vystymasis nuskuba tolyn, o surastos klaidos turi būti taisomos tiek stabilioje laidoje, tiek vėliausioje vystomoje versijoje. Visais šiais atvejais padeda konfigūracijos valdymo sistemos, kitaip dar vadinamos išeities tekstų valdymo sistemomis arba versijų valdymo sistemomis.

3.9.1. Istorinė perspektyva

Senais laikais Berklyje atsirado konfigūracijos valdymo sistema SCCS, vėliau ją pakeitė RCS. Tai yra klasikinės versijų kontrolės sistemos, veikiančios štai koku principu. Yra saugykla, kurioje laikomi versijuoti failai. Jei programuotojas nori padaryti pakeitimą tokiam failui, jis pasiima jį iš sistemos (*check out*), pamodifikuoja ir vėl grąžina į saugyklą (*check in*). Kol programuotojas dirba su failu, failas saugykloje yra užblokuotas (*locked*), t.y. joks kitas programuotojas negali pasiimti (*check out*) to failo. Tokiu būdu kiekvieną failą kiekvienu metu modifikuoja tik vienas žmogus. Grąžinant failą į konfigūracijos valdymo sistemą jis užregistruojamas kaip nauja versija. Sistema leidžia išsiimti peržiūrai bet kokią failo versiją, o taip pat leidžia greitai pasiimti visų pasikeitusių failų vėliausias versijas, kitais žodžiais susisynchronizuoti su versijų valdymo sistemos saugykla.

Dažniausiai versijuoti failai laikomi taupiu formatu, kai laikoma viena pagrindinė failo versija, o kitos versijos laikomos tiesiog kaip pakeitimai nuo pagrindinės.

3.9.2. CVS

CVS (Concurrent Versions System) atsirado apie 1991 metus ir pirmiausia buvo realizuota, kaip apvalkalo skriptų rinkinys RCS sistemos pagrindu. Vėliau apvalkalo skriptai buvo pakeisti C programomis, bet pradinė logika išliko.

CVS veikimas remiasi prielaida, kad net jei keli žmonės dirba ties vienu failu, jų modifikavimai retai konfliktuoja. CVS neblokuoja failų, su kuriais dirbama. Tiesiog prieš grąžinant failus į repozitoriją, jų pakeitimai yra suliejami su pakeitimais, padarytais po paskutinio synchronizavimo. Jei visgi įvyksta konfliktas, jis paliekamas spręsti programuotojui, kuris synchronizuoja savo darbinę failų kopiją su sistemos saugykloje esančia vėliausia versija.

Be to, CVS turi ir kitų naudingų ir plačiai naudojamų savybių, kaip darbas per tinklą, versijų šakos ir t.t.

Šiuo metu daugelis atviro kodo projektų naudoja CVS serverius su anonimine prieiga grupinio darbo organizavimui ir šviežiausio kodo platinimui.

3.9.3. CVS alternatyvos

CVS sistema turi ir pastebimų trūkumų. Pagrindiniai jų — versijuotų failų neįmanoma pervadinti neprarandant jų istorijos, šakų sinchronizavimas ar suliejimas yra neintuityvus, kruopštus ir skausmingas netgi patyrusiems CVS vartotojams. Šie CVS trūkumai yra sprendžiami CVS alternatyvose — Subversion ir Arch.

Subversion yra sistema labai panaši į CVS, kuri leidžia kopijuoti ir pervadinti versijuotus failus, bei turi labai efektyvias operacijas darbui su šakomis. Arch, savo ruožtu, yra naujomis koncepcijomis turtinga sistema, leidžianti lengvą sinchronizaciją tarp šakų, net tarp skirtingų saugyklų.

Subversion prisilaiko CVS modelio su viena centrine versijų saugykla, su kuria dirba visi programuotojai, o Arch siūlo kiekvienam programuotojui laikyti savo saugyklą, joje vykdyti darbą su versijuota sistema, bei kartais nuo karto keistis pakeitimais su kolegomis. Arch leidžia programuotojui, pavyzdžiui, savaitgalį gamtoje padirbėti, po kiekvieno loginio žingsnio išsaugant naują versiją, o parvažiavus į biurą sulieti savo pakeitimus po vieną su kanonine programos versija. Subversion ar CVS, šiuo atveju, verstų programuotoją visus savo savaitgalio pakeitimus registruoti kaip vieną versiją, kas rizikinga konfliktais.

Subversion išlaiko CVS darbo modelį ir turi panašias komandas, tai yra lengvai išmokstama žmogui, kuris moka dirbti su CVS, o Arch su CVS turi mažai bendro, išskyrus abstrakčius veikimo principus.

4. Perl, AWK ir sed

Dažnai prireikia pamanipuliuoti tekstu paprastesniais būdais, nei rankinis redagavimas tekstų redaktoriumi. Paprasčiausias tokio dalyko pavyzdys būtų registracijos failų (angl. *log files*) filtravimas, ataskaitų sudarymas. Kartais reikia atlikti kokius nors paprastus dalies formatuotos tekstinės informacijos išrinkimo, kokių nors žodžių pakeitimo veiksmus ir panašiai. Šio skyriaus antraštėje išvardintos priemonės kaip tik skirtos panašioms darbams.

Perl yra interpretuojama kalba, kuri buvo kuriama kaip pagerintas apvalkalas, bet apaugo daugeliu sintaksės savybių bei bibliotekų, ir tapo bendro pobūdžio interpretuojama kalba. Jai labiausiai tinkantys epitetai — eklektiška, intuityvi, nenuosekli. Perl kalbos motto – *There's more than one way to do it (TMTOWTDI)* (tai padaryti yra daugiau nei vienas būdas).

Perl yra sistemų administratorių kanoninis įrankis, bet yra ir programuotojų, kurie visas paprastesnes (iki 500 eilučių) programas rašo Perl'u.

Taigi, dabar panagrinėsim, kas per priemonės yra *sed*, *awk* ir *perl* ir su kuo jos valgomos. Pirmiausia, reikia suprasti šablono (angl. *pattern*, *regex*) sąvoką.

4.1. Šablonai

Šablonai (angl. *regex*, *regexp*, *regular expression*) yra labai galingas įrankis teksto apdorojimui. Tai yra išraiškos, kurių atitikčių galima ieškoti tekste, norint surasti kokią nors vietą, nustatyti kokio nors informacijos vieneto ribas faile. Regex šablonų sintaksė yra standartizuota POSIX standartų, bet skirtingose priemonėse gali skirtis metasimbolių pateikimo būdas. Pasviręs brūkšnyas (**) arba paverčia paprastus simbolius metasimboliais arba atvirkščiai metasimbolius verčia paprastais simboliais).

Žemiau pateikiama supaprastinta *regex* šablonų sintaksė.

A-Z, a-z, 0-9 ir kiti paprastos raidės reiškia kaip tik tai. Paprastas raides. Pavyzdžiui, šabloną `ana` tenkina žodis `banana`.

. Taškas yra šablonas, galintis reikšti bet kokią simbolį. Pavyzdžiui, šablonas `a . a` atitinka žodžius `banana`, `aaa`, bet ne `baa`, `manna`.

[A-Z123] Kvadratinuose skliaustuose išvardina simbolių aibę reiškia bet kokią vieną tos aibės simbolių. Brūkšniu galima nurodyti simbolių sekas jų pilnai neišrašant. Pavyzdžiui, `a[nm]a` atitinka žodžius `mama`, `banana`, bet ne `tata`.

[^A-Z345] Jei po kvadratinių skliaustų pirmas simbolis yra stogelis, visas šablonas reiškia ne išvardintų simbolių aibę, o jos papildinį.

[[:space:]], [[:alpha:]] ir kiti. Šitie daiktai reiškia skirtingas `<ctype.h>` apibrėžtas simbolių aibes.

(šablonas) taip pat yra šablonas.

(šablonas1|šablonas2) yra tenkinamas, kai tenkinamas šablonas1 arba šablonas2.

Šios šablonus gali sekti tokie „kvantoriai“:

šablonas? tenkinamas, jei šablonas pasikartoja vieną arba nulį kartų. Pavyzdžiui, `ab?a` atitiks eilutes `aa` ir `aba`.

šablonas* tenkinamas, jei šablonas pasikartoja nuo 0 iki ∞ kartų. Taigi, `b(an)*a` atitiks eilutes `ba`, `bana`, `banana`, `bananana` ir t.t.

šablonas+ tenkinamas, jei šablonas pasikartoja nuo 1 iki ∞ kartų.

šablonas{n,m}, kur n ir m yra sveiki skaičiai, yra tenkinamas, jei šablonas pasikartoja nuo n iki m kartų. Ir n ir m gali būti praleisti, praleidimas reiškia atitinkamai nulį ir begalybę kartų. Taigi, `b(an)2a` atitiks žodį `banana`.

Dar yra keletas metasimbolių, reiškiančių eilutės pradžią, eilutės pabaigą (`^` ir `$`), žodžio ribą, ne žodžio ribą ir panašius kontekstus (tiksliai sintaksė priklauso nuo kalbos, bet dažniausiai tai būna `\b` arba `\w`) ir keletas trumpinių skirtingoms simbolių klasėms (pavyzdžiui, `\s` Perl'e reiškia *whitespace*).

Išsamų šitos sintaksės aprašymą galite rasti `grep(1)` ir `perlre` man puslapiuose.

4.2. sed

`sed` reiškia *Stream EDitor*. Tai yra programa neinteraktyviam srautų redagavimui. Jei komandų eilutėje nenurodyti jokie failai, jis skaito iš standartinio įvedimo ir rašo į standartinį išvedimą.

Tipiška `sed` iškviatimo sintaksė yra tokia:

```
sed -e 'komanda' įvesties failai > išvesties_failas
sed skripto_vardas įvesties failai > išvesties_failas
```

Skripto failas gali būti sudarytas iš komandų, esančių skirtingose eilutėse, o komandos sintaksė yra tokia:

```
<adresas><komanda><parametrai>
```

Adresas gali būti eilutės numeris (numeruojama nuo vienetą), eilučių diapazonas nurodomas per kablelį, arba šablonas `/tarp slašų/`.

Pavyzdžiui, `2d` reikškia ištrinti antrą eilutę, o `3,5s/Windows/Unix/g` reikškia nuo trečios iki penktos eilutės pakeisti (*substitute*) šabloną `Windows` į eilutę `Unix`.

5. Procesų valdymas

5.1. Kas yra procesas?

Proceso sąvoka yra labai svarbi Unix sistemose. Beveik visi objektai sistemoje yra arba failai, arba procesai.

Procesas yra operacinės sistemos primityvas, apibrėžiamas labai skirtingai — nuo „savarankiškai vykdoma programa“ iki „tai, ką sukuria `fork(2)` arba `clone(2)`“.

5.2. `struct task_struct`

Linux sistemoje procesas yra vienareikšmiškai aprašomas `struct task_struct` struktūros. Jos yra dinamiškai išskiriamos branduolyje. `struct task_struct` apibrėžimą galima rasti `/usr/src/linux/include/linux/sched.h` faile.

Visos `task_struct` yra sujungtos į dvipusį sąrašą. Be to, visi pasiruošę vykdymui procesai taip pat sujungti į dvipusį sąrašą.

`task_struct` struktūroje laikoma visa proceso informacija: proceso identifikatorius `pid`, tėvinio proceso `pid` — `ppid`, procesų grupės ir sesijos `id` ir t.t., taip pat visa informacija apie teises:

```
uid_t uid,euid,suid,fsuid;
gid_t gid,egid,sgid,fsgid;
int ngroups;
gid_t groups[NGROUPS];
kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
```

Apie šiuos proceso atributus detaliau bus kalbama skyriuje 5.3.4.

Taip pat `struct task_struct` yra rodyklė į visą proceso atminties valdymo informaciją ir kiti tarnybiniai proceso atributai.

5.3. Procesų API

Nemaža dalis Unix procesų valdymo funkcijų yra ANSI C standartinės bibliotekos dalis. Tačiau likusi dalis yra pati Unix sisteminio programavimo esmė.

5.3.1. `fork(2)`

Ši funkcija sukuria naują procesą. Yra sukuriamas šia funkciją kviečiančio proceso kopija (vaikas), ir tėvas gauna kaip grąžinamą reikšmę vaiko `pid`’ą, o vaikas gauna 0. Nesėkmės atveju yra grąžinama -1 reikšmė. Taip atsitinka, jei sistemoje sukurta per daug procesų, pavyzdžiui, nekorektiškoje programoje per dažnai kviečiant `fork(2)`.

Linux sistemoje `fork(2)` yra realizuota kaip `clone(2)`, bendresnės funkcijos, kuri leidžia kurti ir gijas, ne tik pilnaverčius procesus, apvalkalas.

Dar egzistuoja funkcija `vfork(2)`. Tai yra `fork(2)` versija, kuri nekopijuoja iškvietusio proceso virtualios atminties, bet palieka vaikui dalintis tą pačią atmintį su tėvu. Ji buvo sugalvota kaip optimizacija. Mat, programose po `fork(2)` dažniausiai kviečiama `execve(2)` sistemos funkcija, paleidžianti naują programą vietoje kviečiančio proceso, o `fork(2)` sukeltas atminties kopijavimas brangiai kainuoja. Linux sistemoje naudojama *copy on write* semantika, kai po `fork(2)` funkcijos iškvietimo tėvas ir vaikas dalinasi tą pačią atmintį, ir tik kai vienas jų pabando į tą atmintį rašyti, sukuriami vieno rašomo puslapio kopija. Bet kokių atvejų, reikia atsiminti, kad iškvietus `vfork(2)` vaikiniame procese yra saugu kviešti tik `execve(2)` arba `exit(2)` funkcijas, nes kai kuriose sistemose kiti veiksmai gali sugadinti tėvino proceso atmintį.

5.3.2. `exec*(3)` šeima

Šios funkcijos paleidžia kitą programą vietoj einamosios. Visos šitos funkcijos yra praktiškai ekvivalenčios, tačiau turi skirtumų interfeise. Jos yra apvalkalai virš `execve(2)` sisteminės funkcijos.

Tipiškai kviečiama po `fork(2)`. Taip yra paleidžiama kita programa kaip naujas lygiagretus procesas.

5.3.3. `wait(2)` ir `waitpid(2)`

Užbaigęs savo vykdymą vaikas pasilieka procesų lentelėje kaip „zombie“ procesas, kol jo tėvas nenuskaito jo grąžinamo statuso. „Zombiai“ nėra pavojingi. Jie neužima atminties, tik tą vieną `struct task_struct`. Tačiau „zombių“ atsiradimas rodo, kad juos sukūrusi programa yra netvarkinga.

Kad „zombių“ neatsirastų, procesas paleidžiantis vaikus turi nuskaityti jų grąžinamus statusus funkcijomis `wait(2)` ir `waitpid(2)`.

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

```
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
```

Geriausia tiesiog sutvarkyti `wait(2)` iškvietimą iš `SIGCHLD` signalo apdorojimo procedūros. Apie signalus kalbėsime vėliau.

5.3.4. Procesų `uid` ir `gid`

Procesas turi du rinkinius vartotojo ir grupės identifikatorių: po realų ir po efektyvų. Realūs `uid` ir `gid` nurodo vartotoją, kuris paleido procesą, o `euid` ir `egid` gali būti kiti, jei programos paleidžiamasis failas turėjo nustatytus `setuid` ar `setgid` bitus. Tokiu atveju `euid` ir/arba `egid` reikšmės yra paleidžiamojo failo savininko ar grupės reikšmės.

Šioms reikšmėms tikrinti ir modifikuoti yra tokie API:

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

uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);

int setgid(gid_t gid);
int setuid(uid_t uid);
int seteuid(uid_t euid);
int setegid(gid_t egid);
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);

/* Linux specific */
int setfsuid(uid_t fsuid);
int setfsgid(uid_t fsgid);
```

5.3.5. Sesijos ir procesų grupės

Siekiant įgyvendinti Unix darbų valdymo (*job control*) semantikas, procesai grupuojami į sesijas ir procesų grupes. Ir sesija ir procesų grupė turi „pagrindinį“ procesą, vadinamą sesijos lyderiu arba procesų grupės lyderiu. Sesijos ir procesų grupės identifikuojamos lyderio pid.

Supaprastinai žiūrint, sesija — visuma procesų, veikiančių ant vieno terminalo, t.y. paleistų iš vienos login sesijos. Visi jie turi vieną valdantį terminalą.

Procesų grupė, savo ruožtu, tai rinkinys procesų, paleistų viena apvalkalo komanda. Tarkime, jei sujungiame kelias programas į konvejerį, jos visos vykdomos vienoje procesų grupėje. Šis grupavimas reikalingas, norint pristabdyti ir vėl paleisti, arba nutraukti, visą „komandą“.

Veikiantys procesai gali būti pirmame plane (foreground) ir fone (background). Vienu metu tik viena procesų grupė gali būti pirmame plane, t.y. turėti priėjimą prie terminalo. Jei foninė procesų grupė bando kreiptis į terminalą, ji yra sustabdoma signalais SIGTTIN arba SIGTTOU⁵.

```
#include <unistd.h>

pid_t getpgrp(void);
pid_t getpgid(pid_t pid);
int setpgrp(void);
int setpgid(pid_t pid, pid_t pgid);
```

5.3.6. Demonizacija

Jei norime, kad procesas pasiliktų kaip demonas (fone besisukanti tarnyba, serveris), turime jį atjungti nuo valdančio terminalo ir einamos sesijos. Tai vadinama demonizacija ir pasiekama du kartus kviečiant `fork(2)` sistemos funkciją, kviečiant `setsid(2)`, pakeičiant einamąją direktoriją `/` ir uždaranč standartinius įvedimo/išvedimo failus.

```
int daemonize(void)
```

⁵Išties, šie signalai siunčiami tik jei nustatyta terminalo opcija TOSTOP. Žr. `stty(1)`

```

{
    if (fork())
        exit(0);
    chdir("/");
    umask(0);
    close(0);
    close(1);
    close(2);
    setsid();
    if (fork())
        exit(0);

    return 0;
}

```

Pirmasis `fork(2)` iškvietimas reikalingas, kad procesas nebebūtų jau sukurtos sesijos lyderis, tam kad kviečiant `setsid(2)` galima būtų sukurti naują sesiją. `setsid(2)` sukuria procesą, kuris neturi valdančio terminalo ir yra sesijos lyderis. Antrasis `fork(2)` iškvietimas sukuria procesą, kuris nebėra sesijos lyderis, tam, kad jam atidarius kokį nors terminalo įrenginį, tas įrenginys netaptų sesijos valdančiu terminalu.

5.4. Signalai

Signalai yra paprasčiausias tarpprocesinio bendravimo mechanizmas. Jis leidžia pranešti procesui apie kokį nors įvykį, tačiau neleidžia kartu perduoti jokios informacijos.

Pilną sistemoje palaikomų signalų sąrašą galite gauti iškvietę komandą `kill -l`.

Signalus galima siųsti tikrai savo (to paties `ruid`) procesams. Savaimė suprantama, root vartotojas gali siųsti signalus bet kokiems procesams.

SIGKILL signalas besąlygiškai sunaikina procesą. **SIGTERM** yra gražus būdas paprašyti programos baigtis. **SIGINT** yra signalas, siunčiamas, kai vartotojas paspaudžia Control-C kombinaciją (priklauso nuo terminalo nuostatų, keičiamų `stty(1)` komanda). Jei jis neapdorojamas, jis užbaigia procesą. Klaidų taisymui yra naudingas **SIGQUIT** signalas. Jis siunčiamas terminale paspaudus Control-\ kombinaciją, o jo nutylimoji procedūra nutraukia procesą sukurdamą `core` failą.

Signalai **SIGSTOP** ir **SIGTSTP** sustabdo (angl. *suspend*) procesą. **SIGSTOP** negali būti sugautas, o **SIGTSTP** gali. Ctrl-Z kombinacija siunčia procesui **SIGTSTP** signalą. Jis yra tam, kad procesas galėtų apsitvarkyti prieš sustodamas. Pavyzdžiui, sutvarkyti terminalo būseną ir išvaizdą.

SIGCHLD signalą procesas gauna, kai baigiasi koks nors vaikas. **SIGWINCH** praneša procesui, kad pasikeitė jo terminalo dydis (kai padidinamas arba pamažinamas `xterm(1)` langas). **SIGQUIT** priverčia procesą iškart išeiti ir išmesti `core` failą. Naudinga debugimui.

SIGSEGV, **SIGBUS**, **SIGILL**, **SIGFPE** yra programuotojo klaidos požymis. Atitinkamai tai yra nelegalus kreipimasis į atmintį (pavyzdžiui, sekant `NULL` nuorodą), priėjimo prie magistralės klaida, neteisinga instrukcija ir slankaus kablelio operacijų išimtis.

5.5. Kaip gaudyti signalus

Procesas gali apdoroti arba ignoruoti signalus. Kai kurių signalų nutylimoji procedūra (angl. *default handler*) užbaigia proceso vykdymą.

Signalai yra gaudomi pasinaudojant tradiciniu `signal(2)` API, arba pažangesniu `sigaction(2)` API.

5.5.1. `signal(2)`

Šiuo API galima priskirti apdorojimo procedūrą (angl. *handler*) signalui.

```
#include <signal.h>
void (*signal(int signal_num, void (*handler)(int)))(int)
```

Ši deklaracija atrodo sudėtingai tik dėl to, kad `signal(2)` ima kaip parametrus signalo numerį ir rodyklę į apdorojimo funkciją, bei grąžina senos funkcijos adresą, tai yra čia figūruoja dvi rodyklės į funkcijas. Kartais net oficialioj dokumentacijoje tai supaprastinama taip:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

`signal.h` faile apibrėžti tokie specialūs apdorojimo procedūros atvejai:

```
#define SIG_DFL void (*)(int)0
#define SIG_IGN void (*)(int)1
```

Tai yra, atitinkamai, nutylimoji procedūra ir signalo ignoravimas.

Beje, `SIGKILL` ir `SIGSTOP` signalų neįmanoma sugauti. Jie iškart nutraukia proceso vykdymą.

5.5.2. BSD semantika vs. SysV semantika.

SysV.2 sistemoje kai procesas pagaudavo signalą, branduolys deaktivavo signalo apdorojimo procedūrą ir ją paleisdavo. Kitam atėjusiam signalui buvo vykdomas `SIG_DFL` veiksmas. Norint sugauti kitą signalą, reikėjo vėl kviesti `signal(2)`. Šita semantika buvo stipriai kritikuojama, dėl nenusipėjamo elgesio (angl. *race condition*).

```
void signal_2_handler(int signum)
{
    signal(SIGINT, signal_2_handler);
    printf("SIGINT: ignoruoju signalą, važiuojam toliau!\n");
}
```

Nuo 4.2BSD (o taip pat POSIX standarte) yra naudojama kitokia semantika: kai sugaunamas signalas, branduolys nedeaktivuoja apdorojimo procedūros, bet užblokuoja procesus, siunčiančius naujas signalo kopijas.

Linux palaiko tiek SysV tiek BSD `signal(2)` semantikas. Jos gali būti pasirenkamos prijungiant skirtingus antraščių failus. Abu šitie variantai yra realizuoti panaudojant tą patį pažangesnį API, `sigaction(2)`.

5.5.3. `sigaction(2)`

`sigaction(2)` neturi `signal(2)` API dviprasmybių. Pagal nutylėjimą naudojama į BSD panaši semantika: branduolys blokuoja signalą, kol vykdoma jį apdorojanti funkcija. Be to, galima nurodyti bitų maskę signalų, kuriuos reikia blokuoti kokio nors signalo apdorojimo metu (`struct sigaction sa_mask` narys).

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);

int sigprocmask(int how, const sigset_t *set,
                sigset_t *oldset);

int sigpending(sigset_t *set);

int sigsuspend(const sigset_t *mask);

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

5.6. Kaip siųsti

Signalai procesams siunčiami kviečiant `kill(2)` sistemos funkciją. Jos sąsaja labai paprasta. Jai perduodami du parametrai: proceso, kuriam siųsti signalą, `pid` ir signalo numeris. Beje, `signal.h` apibrėžia `SIGKILL`, `SIGCHLD`, `SIGINT` pavidalo makrokomandas, taip kad signalų numerių prisiminti netenka. Be to, skirtingose sistemose jie gali skirtis.

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

int kill(pid_t pid, int sig);
```

Jei `pid` teigiamas, signalas siunčiamas vienam procesui. Jei jis lygus 0, signalas siunčiamas einamojo proceso grupei. Jei jis lygus `-1`, signalas siunčiamas visiems procesams, išskyrus pirmąjį (`init`). Jei jis mažesnis už `-1`, jis siunčiamas procesų grupei, kurios `PGID` yra `abs(pid)`.

6. Lizdai

BSD lizdai (angl. *BSD sockets*) yra tradicinis UNIX tinklo API, jau senai tapęs įprastu ne tik UNIX pavidalo sistemose. Kad ir MS Windows: Winsock biblioteka palaiko kiek savo Perteklingai ilgus BiKapitalizuotus funkcijų vardus, tiek ir tradicinį BSD lizdų API.

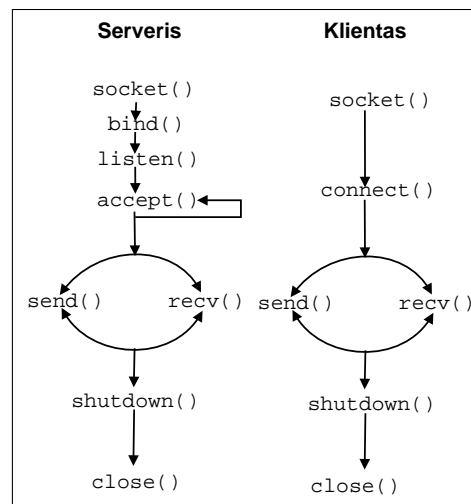
BSD sistemose ir Linux'e lizdai yra tiesiogiai palaikomi branduolio ir `libc`, o pavyzdžiui Sun Solaris sistemoje kompiliuojant reikia nurodyti papildomą biblioteką `-lsocket`.

Lizdų API palaiko daugelį transporto protokolų, teoriškai per lizdus prieinami visi protokolai, palaikomi konkretaus branduolio. Mums bus įdomūs TCP/IP šeimos protokolų bei lokalūs, *Unix domain*, lizdai.

Lizdai atsirado su 4.2BSD, o SysV siūlė kitą, alternatyvų, interfeisą, vadinamą TLI (Transport Level Interface). Visgi paplito BSD lizdai, ir buvo įtraukti į POSIX standarto 2001 m. redakciją.

Lizdai skirstomi į dvi rūšis — su ryšio nustatymu (virtualus kanalas, pvz. TCP) ir be ryšio nustatymo (datagramos, UDP). Priklausomai nuo lizdo tipo skiriasi ir jo vartojimas.

6.1. Lizdai su ryšio nustatymu



3 pav. Lizdai su ryšio nustatymu

Lizdo su ryšio nustatymu atveju, veiksmų seka tokia:

- Sukuriamas lizdo deskriptorius, kviečiant funkciją `socket(2)`. Jai kaip parametrai perduodami domenai (pvz., lokalus lizdas, IPv4, IPv6, IPX, ir kiti), lizdo tipas (virtualus kanalas, datagramos, paketai, tiesioginis priėjimas prie tinklo interfeiso ir t.t.), bei protokolas (IP, ICMP, TCP, UDP ir kiti (žr. `/etc/protocols`)).

Funkcija `socket(2)` tiesiog sukuria lizdą, kuriuo galima bandyti jungtis kur nors arba klausytis kokio nors portu, bet nenustato, kam konkrečiai tas lizdas bus naudojamas.

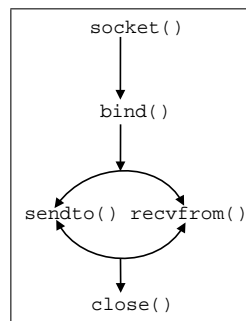
- Serverio atveju, reikia susieti lizdą su konkrečiu tinklo interfeiso portu `bind(2)` funkcija. Kaip parametrai šiai funkcijai perduodami lizdo deskriptorius ir adresas (priklauso nuo pasirinkto domeno ir protokolo). `bind(2)` „priskiria lizdai vardą“. Kai vienas lizdas prisirišo su `bind(2)` prie kokio nors adreso (IP atveju tai yra pora adresas:portas), kitas lizdas negali būti priřtas prie to paties adreso.
- kviečiant `listen(2)` funkciją yra parodomas ketinimas priimti prisijungimus prie lizdo (priřtu adresu). Kaip parametras šiai funkcijai yra perduodamas `backlog` — skaičius, kiek daugiausia prisijungimų bus priimama laukti, kol jie bus apdoroti.

- Pats prisijungimas yra priimamas kviečiant funkciją `accept(2)`. Kaip parametrai jai perduodami serverio klausančio lizdo adresas ir rodyklė į `struct sockaddr`. Šitas lizdas nėra modifikuojamas, užtat kai priimamas prisijungimas, yra sukuriamas ir grąžinamas naujas lizdas, jau sujungtas su klientu ir `struct sockaddr` parametras yra užpildomas prie serverio prisijungusio kliento adresu.

Virtualus kanalas yra sukurtas ir galima pradėti bendravimą.

- Kliento atveju, `bind(2)`, `listen(2)` ir `accept(2)` funkcijų kviešti nereikia. Viena funkcija `connect(2)` priima kaip parametrus lizdo deskriptorių ir serverio adresą, prisijungia prie serverio ir automatiškai išskiria mūsų klientiniam lizdui adresą (pririša).
- Funkcijomis `send(2)` ir `recv(2)` yra siunčiami ir priimami duomenys. Šios funkcijos kaip parametrus priima lizdo deskriptorių, siuntimo/priėmimo buferį ir jo dydį, bei papildomas nuostatas.
- Baigus darbą, ryšys yra nutraukiamas funkcija `shutdown(2)`.
- Lizdo deskriptorius yra uždaromas funkcija `close(2)`, kaip paprastas failas.

6.2. Lizdai be ryšio nustatymo



4 pav. Lizdai be ryšio nustatymo

Lizdai, kuriuos norima naudoti be ryšio užmezgimo, taip pat kuriami `socket(2)` funkcijos, jai nurodžius kokį nors protokolą be ryšio užmezgimo. Prieš naudojant, lizdui būtina priskirti adresą iškviečiant funkciją `bind(2)`. Tada funkcijomis `sendto(2)/recvfrom(2)` yra siunčiami ir gaunami pranešimai. Nuo funkcijų `send(2)/recv(2)` šitos funkcijos skiriasi tuo, kad priima dar vieną argumentą — nuorodą į nutolusio lizdo adresą. Lizdas yra uždaromas funkcija `close(2)`. Jokio `shutdown(2)` nereikia, nes ryšys ir nebuvo nustatomas.

6.3. Unix domeno lizdai

Protokolų šeima PF_UNIX yra būdas procesams toje pačioje mašinoje efektyviai bendrauti. Lizdai gali būti anoniminiai arba susieti su lizdo tipo failu (srwxrwxrwx).

Jei lizdo adreso struktūroje `sun_path[0] == 0`, tai lizdas yra nesusijęs su failų sistema, ir jo adresas yra abstrakčioje adresų erdvėje. Tokiu atveju reikšmę turi visi likę `sun_path` baitai.

```
man 7 unix

#include <sys/socket.h>
#include <sys/un.h>

unix_socket = socket(PF_UNIX, type, 0);
error = socketpair(PF_UNIX, type, 0, int sv[2]);

#define UNIX_PATH_MAX    108

struct sockaddr_un {
    sa_family_t    sun_family;          /* AF_UNIX */
    char           sun_path[UNIX_PATH_MAX]; /* pathname */
};
```

Palyginus su TCP lizdais, Unix domeno lizdai yra žymiai efektyvesni, nes yra optimizuoti duomenų perdavimui vienos sistemos ribose, o be to leidžia perduoti iš vieno proceso į kitą atidarytus failų deskriptorius bei sužinoti proceso, su kuriuo susirišta, identifikatorių ir teises. Šioms papildomoms galimybėms prieiti naudojama `getsockopt(2)` funkcija.

7. System V IPC

IPC yra angliška santrumpa reiškianti *interprocess communication* — tarpprocesinį bendravimą. Tai yra visuma mechanizmų, kuriais informacija gali keistis toje pačioje sistemoje vykstantys procesai. Šiame skyriuje aptariami IPC mechanizmai pirmiausia atsirado System V AT&T Unix versijoje, todėl sutrumpintai vadinami SysV IPC.

System V IPC mechanizmai yra apžvelgiami `ipc(5)` vadovo skyrelyje. Jie skirstomi į šiuos tipus:

Pranešimai — pranešimų eilės formatuotiems duomenims persiųsti.

Semaforai — globalūs sistemos mastu kintamieji naudojami sinchronizacijai.

Bendra atmintis (angl. *shared memory*) — atminties segmentai įeinantys į kelių procesų virtualios atminties erdvę.

Visi šie mechanizmai yra adresuojami sistemoje unikaliais sveiko skaičiaus pavidalo identifikatoriais, kuriuos turi „žinoti“ visi tam tikrą bendrą resursą naudojantys procesai. Tokiems raktams kurti yra pagalbinių funkcijų `ftok(3)`. Jos argumentai — vykdomojo failo kelias bei sveikas skaičius — projekto identifikatorius, o grąžinama reikšmė — sugeneruotas raktas bendriems resursams.

IPC objektai sukuriami tik tada, kai juos atsidaro bent vienas procesas, ir yra sunaikinami, kai atsijungia visi procesai. Tai reiškia, pavyzdžiui, kad pranešimų eilės gali būti naudojamos duomenims perduoti tik tarp vienu metu vykstančių procesų.

Visų trijų mechanizmų atveju, naujas resursas sukuriamas kviečiant `..._get(2)` funkciją, o sunaikinamas `..._ctl(2)` funkcijos `IPC_RMID` komanda.

7.1. Pranešimai

Pranešimai yra pakankamai aukšto abstrakcijos lygio mechanizmas procesų sinchronizacijai ir duomenų perdavimui.

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

struct msgbuf {
    long mtype;      /* pranešimo tipas, turi būti > 0 */
    char mtext[1];   /* pranešimo duomenys */
};

int msgget(key_t key, int msgflg);
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz,
               long msgtyp, int msgflg);
```

Prieš naudojant pranešimų eilę, ją naudojantis procesas turi prisijungti prie eilės iškviesdamas `msgget(2)` ir gauti jos deskriptorių. Kiekvienoje eilėje gali būti skirtingų tipų pranešimai, todėl pranešimo struktūra be paties pranešimo turi ir `long` tipo identifikatorių, o pranešimus gaunančiai `msgrcv(2)` funkcijai reikia nurodyti pranešimo tipą kaip argumentą `msgtyp`. Jei šis argumentas lygus 0, bus nuskaitytas pirmas pranešimas eilėje neatsižvelgiant į jo tipą, jei jis teigiamas — bus nuskaitytas pirmas pranešimas su nurodytu tipu, o jei jis neigiamas, bus nuskaitytas pirmas eilėje esantis pranešimas su mažiausiu tipu, kurio reikšmė mažesnė už `msgtyp` absoliučiąją reikšmę.

7.2. Semaforai

Semaforas — sinchronizacijai naudojamas sveikas kintamasis, kurio reikšmė visada yra neneigiama. Procesai naudojantys semaforą gali atomiškai didinti ir mažinti semaforo reikšmę. Operacija, kuria bandoma pamažinti semaforo reikšmę žemiau 0, yra blokuojama, kol semaforo reikšmė nebus pakankamai padidinta.

System V IPC semaforų realizacija duoda priemonės operuoti semaforų aibėmis, o ne atskirais semaforais.

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

struct sembuf {
    short sem_num;   /* semaphore number: 0 = first */
    short sem_op;    /* semaphore operation */
    short sem_flg;   /* operation flags */
};

int semget(key_t key, int nsems, int semflg);
int semctl(int semid, int semnum, int cmd, ...);
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

Pirmiausia, semaforų aibė turi būti sukurta kviečiant `semget(2)` funkciją. Semaforų pradinės reikšmės turi būti nustatytos `semctl(2)` funkcijos pagalba. Ši funkcija

turi priemonės semaforų reikšmių nustatymui po vieną arba visų kartu (`cmd` reikšmės `IPC_SET` ir `SETALL`), bei semaforų reikšmių nuskaitymui (`IPC_STAT` ir `GETALL`).

Semaforų reikšmės keičiamos perduodant komandų rinkinį funkcijai `semop(2)`. Komandos perduodamos kaip `struct sembuf` tipo masyvas. Kiekvienoje masyvą sudarančioje struktūroje yra nariai, nurodantys semaforo numerį (`sem_num`), reikšmę, kuri turi būti pridėta prie semaforo reikšmės (`sem_op`), bei papildomas žymės (`sem_flg`). Jei `sem_op` reikšmė teigiama, ja padidinama semaforo reikšmė. Jei `sem_op` reikšmė neigiama, tokia reikšmė bus pamažinta semaforo reikšmė. Procesas gali blokuotis. Jei `sem_op` reikšmė lygi 0, procesas bus blokuojamas iki tol, kol semaforo reikšmė netaps lygi 0.

7.3. Bendra atmintis

Bendra atmintis — mechanizmas, leidžiantis vienoje mašinoje esantiems procesams labai efektyviai keistis dideliais duomenų kiekiais, arba tiesiog bendrai panaudoti kokius nors duomenis.

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

int shmget(key_t key, int size, int shmflg);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

Pirmiausia, bendros atminties segmentas turi būti sukurtas iškviečiant `shmget(2)` funkciją. Sistemoje unikalūs atminties segmento identifikatoriai perduodamas per `key` argumentą, o prašomas segmento dydis — per `size`. Realiai bus sukurtas atminties segmentas, kurio dydis bus suapvalintas aukštyn iki sveiko atminties puslapių skaičiaus. Unikali `key` reikšmės sudarymui yra rekomenduojama pasinaudoti `ftok(3)` funkcija.

Prieš naudojantis bendra atmintimi, ji turi būti prijungta prie proceso adresų erdvės `shmat(2)` funkcija. Jos `shmaddr` argumentas — adresas, kuriuo turi būti prijungtas bendros atminties segmentas. Jei šio argumento reikšmė yra `NULL`, sistema pati parinks adresą, kuriuo galima prijungti segmentą. Prijungto segmento adresas grąžinamas kaip šios funkcijos rezultatas.

Baigus naudotis bendros atminties segmentu jį reikia atjungti pasinaudojant `shmdt(2)` funkcija.

8. POSIX gijos (pthreads)

Šiame skyriuje skaitytojas bus supažindinamas su POSIX gijų API. Tikimasi, kad jis jau turi supratimą apie gijas ir kitas lygiagretaus programavimo sąvokas, ir bus pateikiami tik trumpi pačių šių sąvokų apibrėžimai.

Gijos yra keli valdymo srautai viename procese. Buitiškai šnekant, tai yra keli vykdomo kodo gabalai, turintys privačius stekus ir besidalinantys statiniais duomenimis.

Gijos naudojamos kaip mechanizmas išlygiagretinti uždavinį, kai turimi keli procesoriai. Gijos pasitarnauja ir programose vykdomose vieno procesoriaus — kai viena gija

blokuojasi dėl kokio nors resurso, kitos gali tęsti vykdymą. Panaudojant gijas pagerinamas programų su grafine sąsaja interaktyvumas.

Ilgą laiką Linux sistemoje nebuvo gijų palaikymo, ir buvo ginčijamasi, kad jų ir nereikia, nes Unix pavidalo sistemose proceso paleidimo kaštai tiek nedaug skiriasi nuo gijos kaštų, kad galima apsieiti ir su atskirais procesais.

Gijų samprata buvo pradėta vystyti 80-ųjų viduryje. POSIX.1c standarto pagrindu tapo Sun Solaris 2.x gijų realizacija.

Programuojant su gijomis reikia prisiminti, kad:

- `exit(3)` ir `exec*(3)` veikia visą procesą, o ne tik vieną giją;
- naudojant globalius duomenis būtina sinchronizacija.

8.1. Gijų realizacijos Unix sistemose

8.1.1. Sun Solaris 2.x

Sun Solaris sistemoje gijos yra betarpiškai susijusios su LWP (*lightweight processes*, lengvųjų procesų) sąvoka. LWP yra minimalus branduolio planuojamas (*scheduled*) vienetas. Jis turi įrašą branduolio procesų lentelėje, bet neturi savo atskiros virtualios atminties ir kitų proceso atributų.

Sun Solaris sistemoje gijos realizuotos per LWP, bei yra tvarkomos vartotojo režime, o ne branduolyje. LWP yra bibliotekų sukuriama ir dinamiškai priskiriama gijoms, kai jos pasiruošusios vykdymui. Užsiblokavusios gijos gali neturėti su jomis susietų LWP. Taip pat egzistuoja taip vadinamos surištos gijos, kurioms yra statiskai priskirtas LWP, bei jos gali turėti savo signalų stekus, alarmus, taimerius, bei tinka realaus laiko apdorojimui.

Solaris sistemoje kompiliuojant programas su gijomis, kompiliatoriui reikia perduoti opciją `-lthread`.

8.1.2. LinuxThreads

LinuxThreads dabar yra GNU C bibliotekos (nuo versijos 2.x) dalis, taigi yra visose Linux sistemose. Čia *visos* gijos yra branduolio valdomos esybės (LWP), realizuotos per `clone(2)` sistemos funkciją. Toks sprendimas ir sąlygoja svarbiausią šios realizacijos POSIX 1.c standarto neatitikimą. Pagal standartą, kai procesui siunčiamas signalas, jį gauna kolektyviai visos gijos, ir signalo apdorojimo procedūra leidžiama vienoje iš gijų, kurios neblokuoja to signalo. Linux'e neįmanoma pasiųsti signalo visoms gijoms bendrai, nes kiekviena gija branduolio supratimu yra atskiras procesas.

Norint pasinaudoti Linux gijomis, programą reikia kompiliuoti su raktu `-lpthread`.

8.2. NPTL

NPTL (*Native POSIX Threading Library*) yra Red Hat firmos vystoma nauja gijų bibliotekos realizacija, kuris pilnai atitiks POSIX standarto reikalavimus ir ilgainiui pakeis LinuxThreads realizaciją GNU C bibliotekoje. Ši gijų realizacija yra labai efektyvi, bet reikalauja branduolio modifikacijų. NPTL palaikymas buvo įgyvendintas Linux 2.5.x branduolio versijose.

8.2.1. BSD pthreads

BSD sistemose nurodoma `-lc_r`. Tai gali atrodyti keistai, bet viskas paprasta: standartinė biblioteka vadinasi `libc`, jos reenterabili versija — `libc_r`. Linkeris prisideda „lib“ prie bibliotekos vardo pats, taip kad lieka `c_r`.

8.3. Gijų valdymo funkcijos

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine)(void *), void *arg);
void pthread_exit(void *retval);
int pthread_join(pthread_t th, void **thread_return);
int pthread_detach(pthread_t th);
```

Funkcija `pthread_create` sukuria naują giją. Jei viskas tvarkoj, grąžinama reikšmė 0, kitu atveju — klaidos kodas. Gijos identifikatorių įrašo į `*thread`. Funkcijos, kurią vykdys gija, adresas perduodamas `start_routine` parametru, o `arg` yra tai funkcijai perduodamas argumentas. `attr` yra nebūtinas parametras (jis gali būti NULL), struktūra su gijos atributais.

Funkcija `pthread_exit` baigia gijos vykdymą. `retval` yra gijos baigimosi statusas. Šios funkcijos iškvietimas ekvivalentus gijos funkcijos baigimuisi su `return retval`.

`pthread_join` laukia gijos baigimosi ir jos grąžintą statusą patalpina į `*thread_return`. Šia funkcija galima laukti tiksliai neatjungtų gijų pabaigos. `pthread_detach` padaro giją atjungtą, ko pasekoje negalima laukti jos pabaigos, bet visi gijos užimami resursai yra išlaisvinami iškart jai pasibaigus.

Darbai su šiais atributais taip pat yra šeima funkcijų:

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setattributes(pthread_attr_t *attr, int attributes);
int pthread_attr_getattributes(pthread_attr_t *attr, int *attributes);
```

Čia atributas gali būti:

- `detachstate` — nurodo, ar prie giją galima prisijungti, ar ji atjungta;
- `schedpolicy` — nurodo planavimo mechanizmą, taikomą gijai;
- `schedparam` — nurodo gijos planavimo prioritetą;
- `inheritsched` — nurodo, ar naudoti tėvinės gijos `shcedpolicy` ir `schedparam`;
- `scope` — ar gija yra planuojama branduolio ar bibliotekų;
- ir kiti.

LinuxThreads visos gijos yra valdomos branduolio, taigi `scope` reikšmė reiškianti planavimą bibliotekų lygyje, nėra įgyvendinta.

8.4. Mutexų valdymo funkcijos

Mutex angliškai yra sutrumpinimas nuo *mutual exclusion*, rus. взаимное исключение. Kadangi tinkamą lietuvišką terminą surasti sunku, kol kas naudosime anglišką žodį.

Mutexai yra tokie objektai, galintys būti laisvoje arba užrakintoje būsenoje. Vienu metu tik viena gija gali būti užrakinusi mutexą. Kitos gijos bandydamos užrakinti mutexą blokuosis, kol jis nebus atlaisvintas jį užrakinusios gijos. Tada viena iš jų užsirakins mutexą, o kitos liks užblokuotos.

Mutexai yra naudojami kritinės sekcijos apsaugai.

```
#include <pthread.h>

pthread_mutex_t fastmutex    = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex     = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t errchkmutex  = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;

int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Ko gero, įdomiausia čia yra tie statiniai inicializatoriai, kuriantys mutexus su atributais pagal nutylėjimą. Juos panaudojant nereikia kviesti `pthread_mutex_init(3)`!

LinuxThreads realizacijoje mutexai būna trijų rūšių — greiti, rekursyvūs ir tikrinantys. Jie skiriasi tuo, kaip apdoroja pakartotiną mutexo užrakinimą toje pačioje gijoje. Greitas mutex’as tokiu atveju užblokuos procesą iki mutexo atlaisvinimo (t.y. amžinai), klaidas tikrinantis grąžins klaidą EDEADLK, o rekursyvus iškart grįš, įsimindamas mutexo užrakinimo kartų skaičių. Jis turės būti atrakintas tiek pat kartų. Šioje realizacijoje `pthread_mutexattr_t` tipe tėra nurodomi tik šitie mutexo tipai.

Taigi, funkcija `pthread_mutex_lock` užrakina mutexą, `pthread_mutex_unlock` atrakina, o `pthread_mutex_trylock` užrakina, bet nesiblokuoja, o grąžina EBUSY, jei mutex’as jau užrakintas.

8.5. Sąlygos kintamųjų valdymo funkcijos

Sąlygos kintamieji (angl. *condition variable*) yra naudojami kartu su mutex’ais. Sąlygos kintamieji yra sinchronizacijos mechanizmas, leidžiantis gijoms laukti, kol koks nors predikatas taps teisingas, bei kitoms gijoms signalizuoti, kad tas predikatas tapo teisingas.

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Funkcija `pthread_cond_wait` užblokuoja kviečiančią giją iki sąlygos išsipildymo. Tuo pačiu, ji atblokuoja mutexą `mutex`.

Jei kokia nors giją vykdo `pthread_cond_signal`, viena laukianti gija atsiblokuoja, vėl užrakina atleistą mutexą ir vyksta toliau. `pthread_cond_broadcast` skiriasi nuo `pthread_cond_signal` tuo, kad atblokuoja visas sąlygos laukiančias gijas, o ne tik kurią nors vieną.

8.6. Specifiniai gijų duomenys

Dažnai prireikia globalių ar statinių kintamųjų, kurių reikšmės skirtingos skirtingose gijose. Tam skirtas specifinių gijų duomenų mechanizmas.

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *key,
                      void (*destr_function)(void *));
int pthread_key_delete(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *pointer);
void *pthread_getspecific(pthread_key_t key);
```

Specifiniai gijų duomenys gali būti traktuojami kaip `void*` tipo kintamųjų masyvai. Tų masyvų indeksai (raktai) yra bendri visoms gijoms, o reikšmės gali būti skirtingos skirtingose gijose. Naujai sukurtoje gijoje visi specifiniai duomenys inicializuojami `NULL` reikšmėmis.

9. Žodynėlis

Šiame skyriuje pateiktas šiame konspekte naudojamų sunkiau išverčiamų terminų bei jų angliškujų atitikmenų sąrašas.

antraščių failas header file

aparatura hardware

apdorojimo procedūra handler

apvalkalas shell

blokinis įrenginys block device

darbų valdymas job control

kietoji nuoroda hard link

i-mazgas inode, index node

įvardintasis konvejeris named pipe, FIFO

lizdas socket

nuoseklusis įranginys character device

(signalų) nutylimoji procedūra default handler

registracijos failas log file

simbolinė nuoroda symbolic link, symlink

sisteminė funkcija system call, syscall

šablonas regex, regular expression

tarpprocesinis bendravimas interprocess communication

Rodyklė

accept(2), 28
ANSI C, 5
apvalkalas, 11
Arch, 19
AWK, 11

bind(2), 27, 28

C, 11
clone(2), 22, 32
close(2), 28
closedir(3), 6
connect(2), 28
copy on write, 22
core, 16
CVS, 18

Emacs, 5, 12
exec*(3), 22, 32
execve(2), 22
exit(2), 22
exit(3), 32

fork(2), 21–24
ftok(3), 29, 31

Gabriel, Richard P., 10
gcc, 13
getsockopt(2), 29
GNU, 5

i-mazgas, 6
inode, 6
ipc(5), 29
jrenginio failai, 6

katalogas, 6
kill(2), 26

link(2), 7
Linux, 5
listen(2), 27, 28

Makefile, 14, 18
man(1), 12
McIlroy, Doug, 9
mkdir(2), 6

mknod(1), 12
mknod(2), 7, 8, 12
mount(2), 6
msgget(2), 30
msgrcv(2), 30

opendir(3), 6

Perl, 11
Pike, Rob, 9
POSIX, 5
pthread_cond_broadcast, 35
pthread_cond_signal, 35
pthread_cond_wait, 35
pthread_create, 33
pthread_detach, 33
pthread_exit, 33
pthread_join, 33
pthread_mutex_init(3), 34
pthread_mutex_lock, 34
pthread_mutex_trylock, 34
pthread_mutex_unlock, 34
Python, 11

readdir(3), 6
recv(2), 28
recvfrom(2), 28
Ritchie, Dennis M., 4
Ruby, 11

semctl(2), 30
semget(2), 30
semop(2), 31
send(2), 28
sendto(2), 28
setsid(2), 23, 24
shmat(2), 31
shmdt(2), 31
shmget(2), 31
shutdown(2), 28
sigaction(2), 25
signal(2), 25
socket(2), 27, 28
Spolsky, Joel, 10
Stallman, Richard M., 5

strace, 16
struct sockaddr, 28
stty(1), 23, 24
Subversion, 19
symlink(2), 7

Thompson, Ken, 4, 10
Torvalds, Linus, 5

ulimit, 16
unlink(2), 8

vfork(2), 22
vi, 12

wait(2), 22
waitpid(2), 22

Literatūra

- [POSIX] The Open Group. *The Single UNIX Specification, Version 3*. <http://www.unix-systems.org/version3/>
- [Чан97] Теренс Чан. «Системное программирование на C++ для Unix», BHV, Киев, 1997.
- [TAOUP] Eric S. Raymond. *The Art of Unix Programming* Addison-Wesley, 2003, ISBN 0131429019, 512 psl.
Laisvai prieinama internetinė versija: www.faqs.org/docs/artu/
- [AAL] Gary V. Vaughan, Ben Elliston, Tom Tromey, Ian Lance Taylor. *GNU Autoconf, Automake and Libtool*, <http://sources.redhat.com/autobook/>
- [TLK] David A Rusling. *The Linux Kernel*. 1997-1999
<http://www.linuxdoc.org/LDP/tlk/tlk.html>
- [HHU] Eric S. Raymond. *A Brief History of Hackerdom*. 1999
<http://www.tuxedo.org/~esr/writings/hacker-history/hacker-history-4.html>
- [BSD] Marshall Kirk McKusick. *Twenty Years of Berkeley Unix. From AT&T-Owned to Freely Redistributable*
<http://www.oreilly.com/catalog/opensources/book/kirkmck.html>
Tai yra straipsnis iš *Open Sources: Voices from the Open Source Revolution*
Edited by Chris DiBona, Sam Ockman & Mark Stone 1st Edition January 1999
ISBN 1-56592-582-3, 280 psl. <http://www.oreilly.com/catalog/opensources/>
- [Spolsky] Joel Splolsky. *Biculturalism*. 2003
<http://www.joelonsoftware.com/articles/Biculturalism.html>
- [WIB] Richard P. Gabriel. *Worse is Better*. 1991–2003
<http://www.dreamsongs.com/WorseIsBetter.html>