Borislav Đorđević, Dragan Pleskonjić, Nemanja Maček

Operativni sistemi: koncepti

Predgovor

Svaki složeni postupak, koji se može izraziti pomoću algoritma sa konačnim brojem elementarnih operacija, može se predati računaru na izvršenje. Jedan od ključnih delova računarskog sistema koji omogućava prenošenje intelektualnog rada sa čoveka na računar jeste sistemski softver. Sistemski sofver je posrednik između aplikativnog softvera i hardvera koji korisniku omogućava da lakše, jednostavnije i efikasnije iskoristi resurse računarskog sistema.

Glavni deo sistemskog softvera i osnova svakog računarskog sistema je operativni sistem. Sam značaj operativnih sistema ogleda se u količini ljudskih resursa, utrošenih na projektovanju i izgradnji, kao i mističnosti koja ih često okružuje. Poznavanje operativnih sistema je temelj znanja svih koji se ozbiljnije bave računarima i informatičkim tehnologijama. Kao posledica masovne upotrebe računara u gotovo svim oblastima ljudske delatnosti, na polju informatičkih tehnologija prisutan je trend stalnog napretka i čestih promena. Međutim, neki fundamentalni koncepti ostali su neizmenjeni - na tim konceptima zasnovana je i ova knjiga.

Knjiga uvodi čitaoce u osnove operativnih sistema, jasno definišući koncepte i algoritme korišćene pri projektovanju pojedinih delova savremenih operativnih sistema. Može se reći da ova knjiga predstavlja pokušaj demistifikacije operativnih sistema bez detaljnije analize i interpretacije izvornog koda. Koncepti i značajniji algoritmi obrađeni u ovoj knjizi uglavnom su zasnovani na implementacijama u postojećim besplatnim i komercijalnim operativnim sistemima. Naš cilj je da te koncepte i algoritme predstavimo ne vezujući se za konkretan operativni sistem ili konkretnu računarsku arhitekturu.

Pretpostavljamo da čitaoci koji žele da se bave izučavanjem ove materije poznaju osnove arhitekture računara, operativnih sistema i računarskih mreža sa aspekta prosečnog korisnika, kao i osnovne strukture podataka i osnove programiranja na jeziku C, koji je korišćen za ilustraciju značajnijih algoritama. Hardver, značajan za analizu pojedinih komponenti operativnih sistema, kao što su kontroleri ulazno-izlaznih uređaja, opisan je na adekvatnim mestima u samoj knjizi.

Napomenimo da je ovo prva knjiga koja se detaljnije bavi konceptima operativnih sistema na srpskom jeziku. Knjiga je prvenstveno namenjena osnovnom kursu iz nastavnog predmeta Operativni sistemi koji autori izvode na Višoj elektrotehničkoj školi u Beogradu, a mogu je koristiti i studenti koji nastavu iz Operativnih sistema pohađaju na drugim fakultetima ili višim školama, kao i svi koji žele da steknu neka osnovna znanja iz ove oblasti. Knjiga je nastala kao rezultat višegodišnjeg iskustva autora stečenog prilikom izvođenja nastave i praktičnog i teoretskog rada na projektovanju i implementaciji pojedinih komponenti operativnih sistema, računarskih mreža, zaštite i sigurnosti.

Sadržaj knjige

Sadržaj knjige je podeljen na dva dela. U prvom delu detaljno su objašnjeni koncepti savremenih operativnih sistema, dok su primerima ilustrovane implementacije na raznim operativnim sistemima, kao što su razne distribucije UNIX sistema, Linux, Microsoft MS-DOS, Windows NT, Windows 2000 i Windows XP.

Navodimo kraći opis poglavlja koja čine prvi deo knjige.

[1] Uvod u operativne siteme.

U ovom poglavlju je u kraćim crtama opisan istorijat operativnih sistema i navedene su njihove funkcije, karakteristike i poželjne osobine. Na osnovu različitih kriterijuma izvršena je podela operativnih sistema i dat je kraći pregled njihove strukture. Detaljnije informacije o funkcijama pojedinih slojeva u hijerarhijskom modelu operativnog sistema nisu predmet izučavanja u ovom poglavlju.

[2] Kernel i upravljanje procesima.

Poglavlje počinje opisom delova jezgra (kernel) operativnog sistema i hardverskim preduslovima za nadgradnju hardvera jezgrom operativnog sistema. Zatim se uvodi pojam procesa i struktura neophodnih za upravljanje procesom. U ovom poglavlju objašnjena su stanja u kojima se proces može proći u toku svog izvršenja i tranzicije u osnovnom i proširenom dijagramu stanja procesa. Ukratko su objašnjeni pojmovi procesorskog reda, zamene konteksta i raspoređivanja procesa, osnovne operacije koje se nad procesima mogu izvršiti i osnovni mehanizmi međuprocesne komunikacije. Na kraju poglavlja opisane su četiri vrste jezgra operativnog sistema.

[3] Raspoređivanje procesa i dodela procesora.

U višeprocesnim operativnim sistemima procesima se na osnovu nekog algoritma dodeljuje procesor na korišćenje. Na početku poglavlja objašnjeni su pojmovi raspoređivanja sa pretpražnjenjem i bez pretpražnjenja, a zatim su detaljnije predstavljeni značajniji algoritmi za dodelu procesora: *First Come First Served*, *Shortest Job First*, *Round Robin* i raspoređivanje na osnovu prioriteta. Ukratko je opisano i raspoređivanje u više procesorskih redova, u višeprocesorskoj okolini i u realnom vremenu.

[4] Sinhronizacija procesa

Svaki računarski sistem sastoji se od konačnog broja resursa. Procesi koriste resurse računarskog sistema - u slučaju da veći broj procesa želi da koristi isti resurs (na primer, deo operativne memorije), operativni sistem mora da obezbedi adekvatne mehanizme sinhronizacije. U ovom poglavlju opisani su mehanizmi koji obezbeđuju sinhronizaciju procesa: kritične sekcije, semafori i monitori. Predstavljene su hardverske i softverske realizacije kritičnih sekcija - algoritam

striktne alternacije, Dekker-Petersonov algoritam i pekarski algoritam. Takođe, na primeru problema ograničenog bafera, problema čitalaca i pisaca i problema večere filozofa ilustrovana je primena semaforskih tehnika u rešavanju klasičnih problema.

[5] Niti

U ovom poglavlju opisane su osnovne celine za izvršavanje koda pod savremenim operativnim sistemima - niti, odnosno laki procesi. Ukratko su navedeni osnovni višenitni modeli i prednosti korišćenja višenitnih programa.

[6] Zastoj

Poglavlje počinje definicijom zastoja i skupa uslova pod kojim se u sistemu pojavljuje zastoj. Dalje je opisano predstavljanje stanja u sistemu pomoću grafa dodeljenih resursa i metode upravljanja zastojem: prevencija, izbegavanje i oporavak od zastoja.

[7] Upravljanje memorijom

Operativna memorija smatra se jednim od najznačajnijih resursa svakog računarskog sistema. Bez operativne memorije proces ne može da obavi neku značajniju aktivnost. U ovom poglavlju najpre su navedeni osnovni ciljevi koje treba postići na nivou sloja upravljanja memorijom, a zatim su ukratko opisani pojmovi logičkog i fizičkog upravljanja memorijom, vezivanja adresa i razmene (*swap*). Navedene su i osnovne programerske tehnike upravljanja memorijom, poput dinamičkog vezivanja i tehnike preklapanja (*overlay*). Nakon toga su opisane metode kontinualne i diskontinualne dodele memorijskog prostora, kao što su straničenje i segmentacija.

[8] Virtuelna memorija

U ovom poglavlju najpre su opisane tehnike učitavanja stranica na zahtev (*demand paging*) i neke alternativne tehnike, kao što su *Copy on Write*, *prepaging* i memorijski mapirane datoteke. Zatim su opisani značajniji algoritmi za izbor žrtve pri zameni stranica: *First In First Out*, oprimalni algoriram, *Least Recently Used*, satni algoritam i frekvencijski algoritmi *Least Frequently Used* i *Most Frequently Used*. Na kraju poglavlja razmatra se efekat zasićenja.

[9] Ulazno-izlazni podsistem

Ulazno-izlazni podsistem obezbeđuje mehanizme za komunikaciju sa ulazno-izlaznim uređajima. Bez ulazno-izlaznog podsistema, korisnik procesima ne može zadati ulazne podatke niti može od procesa preuzeti rezultate obrade. Takođe, proces nije u mogućnosti da rezultate obrade sačuva na disku ili nekom drugom medijumu niti da ostvari komunikaciju sa drugim računarom. Na početku poglavlja navedene su osnovne funkcije i ciljevi projektovanja ulazno-izlaznog podsistema. Nakon toga je izvršena podela uređaja prema različitim kriterijuma i opisan je hardver koji je značajan za ulazno-izlazni podsistem.

Zatim su opisani servisi koje pruža ulazno-izlazni podsistem, proces prevođenja zahteva u ulazno-izlazne operacije i UNIX System V mehanizam tokova.

[10] Sekundarne memorije

Za razliku od operativne memorije, sadržaj sekundarnih memorija se ne gubi nakon isključivanja napajanja. Na sekundarnim memorijama smešten je operativni sistem, programi i podaci koji se obrađuju. Na početku poglavlja ukratko je opisana struktura diskova i postupci kojima se diskovi pripremaju za rad. Nakon toga su opisani nivoi keširanja diskova i algoritmi za raspoređivanje zahteva za rad sa diskovima (*First Come First Served*, *Shortest Seek Time First*, *SCAN*, *C-SCAN*, *LOOK* i *C-LOOK*). Zatim sledi opis RAID struktura koje se mogu iskoristiti za formiranje stabilnih podsistema sekundarne memorije.

[11] Sistemi datoteka

Za većinu korisnika sistem datoteka je nevidljiviji aspekt operativnog sistema koji obezbeđuje mehanizam za čuvanje i pristup datotekama. Na početku poglavlja definisani su osnovni pojmovi (datoteka, direktorijum i link) i opisani mehanizmi za zaštitu i deljenje datoteka. Nakon toga, čitalac se upoznaje sa pojmom sistema datoteka i metodama dodele prostora datotekama i upravljanja slobodnim prostorom u sistemima datoteka. Zatim je obrađena problematika pouzdanosti i efikasnosti sistema datoteka, a na kraju poglavlja je dat opis značajnijih sistema datoteka na UNIX/Linux i Windows operativnim sistemima.

[12] Distribuirani sistemi

Distribuirani sistemi predstavljaju kolekciju procesora, odnosno računara koji ne dele zajedinčku memoriju i sistemski časovnik. Na početku poglavlja objašnjeni su neki osnovni pojmovi i navedene karakteristike distribuiranih sistema. Nakon toga opisane su razlike između mrežnih i distribuiranih operativnih sistema. Zatim su opisani mrežno-komunikacioni podsistem i distribuirani sistemi datoteka.

[13] Sinhronizacija procesa u distribuiranim sistemima

U ovom poglavlju problem sihronizacije procesa i zastoja proširuje se sa centralizovane na distribuiranu okolinu. U poglavlju su opisani mehanizmi sinhronizacije procesa, atomske transakcije u centralizovanim i distribuiranim sistemima i rešavanje problema zastoja u distribuiranoj okolini.

[14] Zaštita i sigurnost

Zaštita se odnosi na kontrolu pristpupa resursima operativnog sistema. Za razliku od zaštite koja se odnosi na unutrašnjost samog sistema, sigurnost, po pravilu, uključuje uticaj interackije sistema sa spoljnim svetom. Na početku poglavlja opisani su domeni zaštite i matrica pristupa, a zatim aspekti sigurnosti i mehanizmi autentifikacije korisnika. Zatim je opisana metodika napada na sistem (*DoS*, *spoofing*, *sniffing*), programske pretnje (trojanski konji, klopke - *trap door*,

prepunjenje steka i bafera) i sistemske pretnje (crvi i virusi). Od tehnika za povećanje sigurnosti sistema opisane su mrežne barijere, sistem za detekciju napada i kriptografske metode zaštite. Na kraju poglavlja navedeni su rangovi sigurnosti prema TCSEC kriterijumu Ministarstva odbrane Sjedinjenih Američkih Država.

[15] Korisnički interfejs

Na vrhu hijerarhije slojeva operativnog sistema nalazi se korisnički interfejs, pomoću kog korisnici ostvaruju dvosmernu interakciju sa operativnim sistemom. Postoje dve vrste korisničkih interfejsa - alfanumerički i grafički, koji su u ovom poglavlju ukratko opisani. Poglavlje sadrži i kraći opis kontrole posla na sistemima sa paketnom obradom.

U drugom delu knjige opisani su značajniji operativni sistemi.

[16] Windows familija operativnih sistema

MS-DOS, Windows 3.x, Windows for Workgroups 3.11, Windows 95/98, Windows NT, Windows 2000, Windows XP, Windows 2003 Server i Longhorn.

[17] Linux

[18] Apple Mac OS X.

Zahvalnost

Autori se zahvaljuju svima koji su na bilo kakav način pomogli u realizaciji ove knjige. Posebnu zahvalnost autori duguju profesorima Abrahamu Silberschatzu i Slobodanu Obradoviću koji su svoje rukopise i materijale ustupili autorima na korišćenje.

Autori

Sadržaj

PREDGOVOR II
Sadržaj knjige III
Zahvalnost VI

SADRŽAJ 7

1. UVOD U OPERATIVNE SISTEME 1

1.1. Istorijat operativnih sistema 1

1.2. Definicija i funkcije operativnog sistema 6
Funkcije operativnog sistema 8

1.3. Karakteristike operativnih sistema 9
Poželjne osobine operativnog sistema 10
Ciljevi projektovanja operativnog sistema 11

1.4. Vrste operativnih sistema 12
Klasifikacija prema broju korisnika i procesa 12
Klasifikacija prema nameni i načinu obrade poslova 13
Klasifikacija prema funkcionalnim osobinama računarskog sistema 13
Vrste računarskog okruženja 17

1.5. Opšti pregled strukture operativnih sistema 18
Monolitni sistemi 20
Slojevita (hijerarhijska) realizacija 20
Sistemski pozivi 22
Mikrokernel arhitektura 23
Virtuelne mašine 23

2. KERNEL I UPRAVLJANJE PROCESIMA 25

2.1. Kernel 25
Delovi kernela 27

2.2. Pojam procesa 28

2.3. Predstavljanje procesa 29

Dijagram stanja procesa 30
Prošireni dijagram stanja procesa 33

2.4. Raspoređivanje procesa 34
Redovi čekanja na procesor 35
Planer poslova i dispečer sistema 36
Zamena konteksta procesa 38
Srednji nivo raspoređivanja procesa 38

2.5. Operacije nad procesima 39
Kreiranje procesa 40
Završetak procesa 42
Saradnja među procesima 42

2.6. Rutine za interprocesnu komunikaciju 44
Slanje poruka 45
Direktna komunikacija 46
Indirektna komunikacija 46
Semafori 47
Interprocesna komunikacija u klijent-server sistemima 48

2.7. Vrste kernela 49

3. RASPOREĐIVANJE PROCESA I DODELA PROCESORA 51

3.1. Uvodne napomene 51
Raspoređivanje sa pretpražnjenjem 52
Kriterijumi dodele procesora 53
Ispitivanje algoritama 54

3.2. Algoritmi za dodelu procesora 54
First Come, First Served (FCFS) 54
Shortest Job First (SJF) 55
Raspoređivanje na osnovu prioriteta procesa 59
Round Robin (RR) 60

3.3. Raspoređivanje u više procesorskih redova 63
Povratna sprega između redova čekanja 64
Raspoređivanje u višeprocesorskoj okolini 65
Rapoređivanje u realnom vremenu 66

4. SINHRONIZACIJA PROCESA 67

4.1. Problem sinhronizacije 67
Ilustracija na problemu proizvođač-potrošač 68

4.2. Kritična sekcija 70

Softverska realizacija kritične sekcije 71
Algoritam striktne alternacije 72
Realizacija kritične sekcije bez stroge alternacije 73
Dekker-Petersonov algoritam 74
Pekarski algoritam 74
Hardverska realizacija kritične oblasti 76

4.3. Semafori 78
Upotreba semafora 78
Proširena definicija semafora 80
Problem zastoja i zakucavanja 81
Binarni i brojački semafori 82

4.4. Klasični problemi sinhronizacije i semafori 83
Problem ograničenog bafera 83
Problem čitalaca i pisaca 84
Problem večere filozofa 86

4.5. Kritični regioni i monitori 87
Kritični regioni 88
Monitori 89

5. NITI 93

5.1. Laki i teški procesi 93
Prednosti upotrebe niti 94

5.2. Korisničke i kernelske niti - višenitni modeli 95
Višenitni model više u jednu 96
Višenitni model jedna u jednu 96
Višenitni model više u više 97

5.3. Osobine niti i rad sa nitima 98
Prekidanje niti 98
Upravljanje signalima 98
Gomila niti 99
Podaci specifični za nit 99

5.4. Vrste niti 99
Pthreads 99
Solaris 2 niti 100
Windows 2000 niti 101
Linux niti 102

6. ZASTOJ 103

6.1. Zastoj 103

Sistemski model 103
Uslovi pod kojima nastupa zastoj 104
Graf dodeljenih resursa 105

6.2. Metodi upravljanja zastojom - prevencija 107
Prevencija zastoja 107

6.3. Izbegavanje zastoja 108
Bezbedno stanje 109
Graf dodele resursa za izbegavanje zastoja 110
Bankarski algoritam 111
Primer upotrebe bankarskog algoritma 113

6.4. Detekcija i oporavak od zastoja 114
Detekcija u slučaju da resursi imaju samo jednu instancu 115
Detekcija u slučaju da resursi imaju više instanci 115
Korišćenje algoritma za detekciju 117
Oporavak od zastoja 117

7. UPRAVLJANJE MEMORIJOM 119

7.1. Uvodne napomene 119
Ciljevi koje treba postići 120
Alokacija memorije 120
Vezivanje adresa 121
Logički i fizički adresni prostor 123
Zaštita memorije 124
Razmena (swap) 125

7.2. Programerske tehnike upravljanja memorijom 126
Dinamičko punjenje memorije programom 126
Dinamičko povezivanje 126
Tehnika preklapanja 127

7.3. Kontinualna alokacija memorije 129
Multiprogramiranje sa fiksnim particijama 129
Multiprogramiranje sa particijama promenljive veličine 129
Algoritmi za izbor prazne particije 132
Eksterna fragmentacija i kompakcija memorije 132

7.4. Straničenje 133
Hardverska podrška 137
Zaštita 138
Hijerarhijsko straničenje 139
Heš (hash) bazirane tabele stranica 140
Invertovane tabele stranica 141
Deljive stranice 142

7.5. Segmentacija **143**
Hardverska podrška 144
Zaštita i deljenje segmenata 146
Segmentacija sa straničenjem 146

8. VIRTUELNA MEMORIJA 149

8.1. Uvodne napomene **149**

8.2. Učitavanje stranica prema potrebi **150**
Neophodan hardver 150
Prebacivanje stranica sa diska u fizičku memoriju 151
Mogući problemi 153
Performanse DP tehnike 153

8.3. Alternativne tehnike učitavanja stranica **154**
CoW (Copy on Write) tehnika 154
Prepaging tehnika 155
Memorijski mapirane datoteke 155

8.4. Zamena stranica **156**
FIFO algoritam 158
Optimalni algoritam 160
LRU algoritam 161
LRU aproksimativni algoritmi 162
Druga šansa (satni algoritam) 163
NRU algoritam 164
Frekvencijski bazirani algoritmi - LFU i MFU 165

8.5. Raspodela okvira po procesima i efekat zasićenja **165**
Raspodela okvira 166
Globalna i lokalna zamena stranica 166
Efekat zasićenja 167

8.6. Dopunska razmatranja **169**
Veličina stranice 170
Uticaj na strukturu programa 170
Zaključavanje stranica 171

9. ULAZNO-IZLAZNI PODSISTEM 173

9.1. Funkcije i ciljevi projektovanja **173**
Funkcije ulazno-izlaznog podsistema 174
Ciljevi projektovanja ulazno-izlaznog podsistema 174

9.2. Klasifikacija uređaja **176**
Časovnik i timer kao specijalna klasa uređaja 177

9.3. Hardver od značaja za ulazno-izlazni podsistem **178**
Magistrale 179
Kontroleri 179
Tehnika prozivanja (polling) 180
Prekidi 181
DMA (Direct Memory Access) 184

9.4. Uniformni interfejs ka aplikacijama **186**
Blokirajući i ne-blokirajući sistemski pozivi 188
UNIX i problem uniformnosti interfejsa 188

9.5. Servisi koje obezbeđuje ulazno-izlazni podsistem **189**
Raspoređivanje ulazno-izlaznih operacija 189
Baferovanje 189
Keširanje 190
Spool tehnika 191
Upravljanje greškama 192

9.6. Prevođenje zahteva u ulazno-izlazne operacije **192**

9.7. Mehanizam tokova **193**

9.8. Performanse ulazno-izlaznog podsistema **194**

10. SEKUNDARNE MEMORIJE 197

10.1. Struktura diskova **197**
Geometrija diskova 198
Savremeni disk uređaji 198
Karakteristike savremenih disk kontrolera 201
IDE i SCSI diskovi 203

10.2. Priprema diskova za rad **203**
Formatiranje diskova 204
Oporavak od oštećenih blokova 205
Kreiranje particija i sistema datoteka 205
Bootstrap rutina 206

10.3. Nivoi keširanja diskova **207**

10.4. Raspređivanje zahteva za rad sa diskom **210**
FCFS 210
SSTF 211
SCAN 212
C-SCAN 212
LOOK i C-LOOK 213
Izbor algoritma za raspoređivanje disk zahteva 214

10.5. RAID strukture - realizacija stabilnih sistema **215**
RAID nivoi 216

10.6. Priključivanje diskova **220**

10.7. Upravljanje swap prostorom **222**

10.8. Tercijalna memorija **223**
Flopi diskovi 224
CD-ROM uređaji 225
Magnetne trake 226

11. SISTEMI DATOTEKA 227

11.1. Pojam datoteke **227**
Logička struktura datoteke 229
Tipovi datoteka 229
Metode pristupa datotekama 230

11.2. Pojam direktorijuma **232**
Logička struktura direktorijuma 233

11.3. Linkovi (reference) **235**
Linkovi na direktorijume i problem beskonačnih petlji 236

11.4. Deljenje i zaštita datoteka **237**
Deljenje datoteka po mreži 237
Zaštita datoteka 238
Zaštita datoteka i direktorijuma na UNIX i Linux sistemima 238
Zaštita datoteka i direktorijumima na NTFS sistemu datoteka 240

11.5. Sistemi datoteka **241**
Realizacija sistema datoteka 242
Strukture podataka neophodne za realizaciju sistema datoteka 244
Aktiviranje sistema datoteka 245
Virtuelni sistem datoteka 247
Realizacija direktorijuma 248

11.6. Dodela prostora za datoteke **249**
Dodela kontinualnog prostora 249
Vezivanje blokova 250
Mapa datoteka 251
Metoda indeksnih blokova 253

11.7. Upravljanje slobodnim prostorom **254**

11.8. Pouzdanost sistema datoteka **256**
Vođenje dnevnika transakcija 257

Arhiviranje i rezervne kopije podataka 258

11.9. Efikasnost i performanse sistema datoteka 259
Keširanje 259
Defragmentacija 263

11.10. Značajniji sistemi datoteka (UNIX/Linux) 263
Opis konkretnih sistema datoteka 266

11.11. Značajniji sistemi datoteka (DOS/Windows) 268
FAT (File Allocation Table) 268
FAT32 269
NTFS 270

12. DISTRIBUIRANI SISTEMI 273

12.1. Uvod u distribuirane sisteme 273
Karakteristike distribuiranih sistema 275

12.2. Tipovi mrežno-orijentisanih operativnih sistema 275
Mrežni operativni sistemi 276
Distribuirani operativni sistemi 276

12.3. Mreža i komunikacija 278
Komunikacije 281
Komunikacioni protokoli 284

12.4. Detekcija greške i oporavak 287

12.5. Distribuirani sistemi datoteka 288
Mapiranje datoteka i trasparentnost 288
Pristup udaljenim datotekama, keširanje i replikacija 289

13. SINHRONIZACIJA PROCESA U DISTRIBUIRANIM SISTEMIMA 293

13.1. Distribuirana sinhronizacija procesa 293
Međusobno isključenje 295

13.2. Atomske transakcije u centralizovanim uslovima 296
Pojam transakcije 296
Konkurentne atomske transakcije 297

13.3. Atomske transakcije u distrubuiranim uslovima 299
2PC protokol (two-phase commit protocol) 299
Konkurentne atomske transakcije u distribuiranim uslovima 301
Protokoli za zaključavanje 302

Timestamp protokoli u distrubuiranim uslovima 303
Šema poretka vremenskih oznaka 303

13.4. Upravljanje zastojima u distribuiranim uslovima 304
Izborni algoritmi 306

14. ZAŠTITA I SIGURNOST 307

14.1. Domeni zaštite i matrice prava pristupa 307
Matrica prava pristupa 309
Implementacija matrice prava pristupa 312

14.2. Aspekti sigurnosti 313

14.3. Autentifikacija korisnika 315
Čuvanje lozinke na disku sistema 315

14.4. Metode napada - programske i sistemske pretnje 317
Napad na sistem 317
Programske pretnje 318
Sistemske pretnje 319

14.5. Tehnike za povećanje sigurnosti sistema 320
Mrežne barijere 321
Detekcija napada 322

14.5. Kriptografija 323
Simetrični kriptoalgoritmi 324
Kriptoalgoritmi sa javnim ključem 324
Autentifikacija 325
Digitalni potpis 325

14.6. Rangovi sigurnosti 326
Uobičajeni kriterijum 328

15. KORISNIČKI INTERFEJS 329

15.1. Uvodne napomene 329

15.2. Komandni interpreteri 331
Osnovne funkcije komandnog interpretera 331
Poželjne osobine komandnog interpretera 332
Dodatne funkcije komandnih interpretera 332

15.3. Kontrola posla 334

15.4. Grafički korisnički interfejs 335

16. WINDOWS FAMILIJA OPERATIVNIH SISTEMA 337

16.1. Istorijat Microsoft operativnih sistema 337
MS-DOS 338
Microsoft Windows 3.x 341
Windows for Workgroups 3.11 343
Microsoft Windows 95/98 344
Razvoj NT tehnologije 347

16.2. Windows NT 348
Arhitektura Windows NT operativnog sistema 349
Sloj apstrakcije hardvera (HAL) 349
Kernel 350
Windows NT egzekutiva 350
Podsistemi okruženja 353
Sigurnosni podsistem 356

16.3. Windows 2000 357
Principi dizajna 358
Arhitektura Windows 2000 operativnog sistema 358
Kernel, procesi i niti 359
Komunikacija između procesa 361
Rukovanje prekidima i zamkama 361
Egzekutiva 362
Podsistemi okruženja i sigurnosni podsistem 366
Sistem datoteka 367
Upravljanje diskovima i tolerancija otkaza 369
Umrežavanje - mrežni interfejsi, protokoli i razrešavanje imena 371
Umrežavanje - pristup udaljenim datotekama 373

16.4. Windows XP 374
Arhitektura Windows XP operativnog sistema 374
Unapređenja u Windows XP operativnom sistemu 375

16.5. Windows 2003 Server 376

16.6. Longhorn 377
Nove tehnologije 378
Novi korisnički interfejs 378
Arhitektura Longhorna 379
Sloj osnovnih servisa operativnog sistema 379
Sloj prezentacije - Avalon 381
Sloj upravljanja podacima - WinFS 382
Sloj komunikacija - Indigo 384

17. LINUX 385

17.1. Šta je Linux ? 385

GNU/Linux i Open Source Software 386
Linux distribucije 386

17.2. Opšti pregled Linux sistema 387

17.3. Jezgro Linux sistema 387
Struktura Linux sistema 388
Modularni kernel 389

17.4. Značajni delovi jezgra 390
Upravljanje procesima 391
Vrste procesa 392
Kreiranje procesa - fork i exec 393
Procesi i niti 394
Dodeljivanje procesora procesima 394
Komunikacija između procesa 395
Upravljanje memorijom 396
Ulazno - izlazni sistem 396
Sistemi datoteka i aktivno UNIX stablo 397

17.5. Osnovni servisi Linux sistema 398

18. MAC OS X 403

18.1. Istorijat 403
A/UX 404

18.2. Uopšteno o Mac OS X operativnom sistemu 405
Darwin - temelj Mac OS X 405
Grafički podsistemi 405
Upravljanje podacima i informacijama 406
Sistemi datoteka 407
Umrežavanje 409
Sigurnost 409
Mac OS X Server 409

18.3. Arhitektura Mac OS X operativnog sistema 410
XNU - kernel 410
Osnovni servisi 412
Aplikativni servisi 412
Aplikativna okruženja 413
Korisnički interfejs 414

LITERATURA 415

1

Uvod u operativne sisteme

Sadržaj poglavlja:

1.1. Istorijat operativnih sistema
1.2. Definicija i funkcije operativnog sistema
1.3. Karakteristike operativnih sistema
1.4. Vrste operativnih sistema
1.5. Opšti pregled strukture operativnih sistema

1.1. Istorijat operativnih sistema

Računarski sistem je složeni skup tehničkih uređaja (hardvera), programa koji zadaju instrukcije hardveru i podataka koji se obrađuju. Ljudi koji koriste računare, bilo kao operateri, programeri ili projektanti, u radu se pridržavaju različitihih formalnih i neformalnih procedura. Mnoge procedure mogu se pretočiti u programe, tako da ih obavljaju računari, a ne ljudi.

Prevodioci (*compilers*) omogućavaju ljudima da programe pišu na jeziku koji razume čovek, a zatim da taj program prevedu na mašinski jezik, odnosno jezik koji razume mašina. Povezivači (*linkers*) i punioci (*loaders*) omogućavaju sprezanje više mašinskih programa u jednu programsku celinu. Ta celina se može više puta upotrebiti i uvek se smešta na razne lokacije u memoriji. Pri tome, potrebno je odgovoriti na sledeća pitanja:

- Kako se više programa može istovremeno smestiti u memoriju?

- Ko će izabrati koji će program biti u memoriji?
- Koji će program i kada koristiti procesor?
- Kako ćemo obezbediti da svaki program dobije resurse?

Odgovor na sva ova pitanja sadrži operativni sistem.

Istorijat operativnih sistema posmatraćemo prateći istorijat samih računara.

Računari prve generacije (1945-1955), čiju su osnovu činile vakuumske cevi (do 20.000 cevi po računaru), bili su ogromnih dimenzija i veoma skupi. Kao takve, koristila ih je, uglavnom, vojska. Ovi računari su bili jako spori, programiralo se na mašinskom jeziku, dok su simbolički jezici (uključujući i asembler), kao i operativni sistemi u to vreme bili nepoznati. Ljudi koji su radili na tim računarima radili su sve poslove, od programiranja do održavanja računara.

U prvoj generaciji računara, opsluživanje računarskog sistema bilo je potpuno prepušteno operateru, koji je morao da pripremi sve što je potrebno da se zadatak obrade može obaviti. Čovek je, dakle, imao punu kontrolu nad računarskim sistemom. Operater je bio u mogućnosti da sve potrebne radnje obavi na vreme, jer je sistem bio spor i izvršavao samo jedan program, odnosno obavljao samo jedan zadatak. Može se reći da je iskorišćenje računarskog sistema, odnosno njegovih najvažnijih resursa, centralnog procesora i centralne memorije, bilo slabo. Pretežni deo vremena trošio se na poslove operatera i ulazno-izlazne operacije, a mnogo manji deo vremena na rad centralnog procesora. Iako je ovakav sistem bio krajnje neefikasan, odnos tih vremena bio je u prihvatljivim granicama zbog relativno male brzine samog centralnog procesora.

Osnovu računara druge generacije (1955-1965) činili su tranzistori, tako da su računari postali manji, pouzdaniji i jeftiniji. Računare druge generacije su, osim vojske, kupovale i velike korporacije i univerziteti. Računari su bili smešteni odvojeno, u posebnim sobama, koje su se delile u tri funkcionalne celine: ulazna soba, centralni računar i izlazna soba. Programeri su pisali programe na papiru u programskom jeziku FORTRAN, zatim su se ti programi prenosili na bušene kartice, koje su se ostavljale u sobi sa ulaznim računarom (*input room*). Sistem operator je, zatim, uzimao bušene kartice i ubacivao u računar, i to prvo kartice sa FORTRAN prevodiocem, a potom bušene kartice sa programom koji treba izvršiti. Glavni računar je odrađivao posao, a rezultat se dobijao takođe na bušenim karticama, koje su se prenosile u prostoriju sa rezultatima (*output room*). Ovde se mnogo vremena trošilo na šetanju između raznih prostorija sa bušenim karticama. Operativni sistem kao zaseban pojam još uvek nije postojao.

Dalje se kao poboljšanje uvodi paketna, odnosno grupna obrada (*batch system*), bazirana na upotrebi magnetne trake kao mnogo bržeg uređaja u odnosu na bušene kartice. Kod paketne obrade u ulaznoj sobi sa poslovima se sakuplja jedna količina sličnih programa (na primer, svi programi koji zahtevaju FORTRAN prevodioc), koji se pomoću jeftinijeg računara (npr. IBM 1401) sa bušenih kartica prenose na magnetnu traku. Posle toga se magnetna traka prenosi u sobu sa glavnim računarom, odnosno sa

moćnijim i skupljim računarom predviđenim za izvršavanje programa (npr. IBM 7094). U glavni računar se učitava poseban program koji je zadužen da sa trake sa poslovima programe redom učitava i izvršava. Taj program se može smatrati pretkom operativnih sistema. Nakon izvršavanja programa, rezultati se snimaju na drugu magnetnu traku koju operater prenosi do trećeg računara, zaduženog za prebacivanje rezultata sa magnetne trake na bušene kartice. Manji računari (ulazno/izlazni) nisu direktno vezani za glavni računar, odnosno rade u *off-line* režimu.

U drugoj generaciji računarskih sistema povećava se brzina rada centralnog procesora, kapaciteti centralne i eksternih memorija, pojavljuju se nove i brže ulazno-izlazne jedinice. Programi se pišu u simboličkom mašinskom jeziku, asembleru ili u višem programskom jeziku (FORTRAN). Operater više nije u stanju da efikasno opslužuje računarski sistem, jer su njegove reakcije suviše spore. Jedino rešenje se moglo naći u prebacivanju niza kontrolnih funkcija sa operatera na sam računarski sistem, odnosno na posebne kontrolne programe. Dakle, funkcije opsluživanja i upravljanja sistemom podeljene su između operatera i kontrolnih programa. Ti programi se uključuju u određenim situacijama, kao što su, na primer, priprema programa za izvođenje, kontrola ulaznih i izlaznih uređaja i razni poslovi oko učitavanja programa i njegove pripreme za izvođenje. Dakle, u računarima druge generacije razlikuju se dve osnovne vrste programa: kontrolni programi i korisnički programi.

Računari treće generacije (1965-1980) prave ce od integrisanih kola (IC). Početkom 60-tih većina proizvođača računara proizvodi dve vrste računara: jednu bržu verziju (kao IBM 7094) i jednu slabiju (kao IBM 1401), što je skup poduhvat. Novi korisnici računara najpre žele slabije modele koji su jeftiniji, dok će im jači, brži i skuplji modeli biti potrebni tek nakon izvesnog vremena. IBM taj problem pokušava da razreši uvođenjem klase računara IBM System/360. To je serija kompatibilnih računara različitih snaga. Svaki od ovih računara je pogodan i za naučnu i za poslovnu primenu, pa je time podela računara na ove dve vrste nestala. Ovaj koncept su preuzeli i ostali proizvođači računara. Računari iz serije System/360 su radili pod operativnim sistemom OS/360, koji je bio veoma glomazan i prepun grešaka.

Sa razvojem discipline poznate pod imenom softversko inžinjerstvo, uvode se nove funkcije:

- multiprogramiranje,
- višestruke ulazno/izlazne operacije (*spool*) i
- podela računarskog vremena (*time-sharing*).

Kada neki program čeka na rezultate ulazno-izlaznih operacija, procesor je neiskorišćen, pa se javlja gubljenje procesorskog vremena. Ovaj problem nije toliko izražen kod programa koji ne zahtevaju često ulazno-izlazne operacije (na primer, naučno orijentisani programi), ali jeste kod poslovnih programa. Multiprogramiranje (*multiprogramming*) je tehnika kojom se postiže bolje iskorišćavanje procesora: memorija se deli na particije u kojima se učitavaju različiti programi, odnosno poslovi

(*jobs*). Dok neki program čeka na ulazno-izlaznu operaciju, procesor može izvršavati drugi program. Na taj način, ako imamo dovoljan broj programa u memoriji, procesor je u stalnoj upotrebi.

Spool (*Simultaneous Peripheral Operation On Line*) je tehnika koja omogućava kompenzaciju nedovoljene brzine ulazno-izlaznih uređaja upotrebom brzih uređaja kao što su trake, a naročito diskovi. Na taj način se omogućava izvršenje višestrukih ulazno-izlaznih operacija. Brzi uređaj prihvata sve sa ulaza, a zatim se ulaz ka procesoru realizuje sa brzog uređaja. Ulaz se realizuje prebacivanjem sadržaja bušenih kartica na disk (traku) pomoću posebnog uređaja, a bez korišćenja procesora. To znači da se disk paralelno puni novim poslovima (*jobs*) dok procesor izvršava programe u memoriji. Kada jedan program završi rad procesor na njegovo mesto može učitati drugi program sa diska. Slično, sve što se šalje na izlaz, prvo se prenosi na brzi uređaj, a zatim sa njega na spore periferale.

Podela vremena (*time-sharing*) je tehnika koja omogućava da svaki korsinik radi interaktivno sa računarom i to preko posebnog terminala, koji je istovremno i ulazni i izlazni uređaj za korisnika. Podela vremena je poseban oblik multiprogramiranja, gde svakom terminalu pripada dodeljeno procesorsko vreme. Posle isteka vremenskog kvantuma, odnosno dodeljene količine procesorskog vremena, porcesor se dodeljuje drugom terminalu. Ukoliko je terminal blokiran zbog čekanja na ulazno-izlazne operacije, procesor se i pre isteka kvanutuma dodeljuje drugom terminalu.

Kod treće generacije računara posebno treba istaći pojavu dva operativna sistemima, a to su MULTICS i UNIX. Projekat MULTICS (*MULTIplexed Information and Computing Service*) je neuspela ideja (MIT, Bell Labs, General Electric) da se napravi moćan računar i operativni sistem koji će biti u stanju da radi sa velikim brojem terminala. Osnovna ideja preuzeta je iz modela distribucije električne energije - u tom modelu, dovoljno je da svaki korisnik, koji poželi da koristi neki električni aparat, taj isti aparat samo priključi na električnu mrežu. Sličan model su pokušali da naprave i sa računarima: ideja je da u jednom gradu postoji moćan centralni računar, a da građani kod kuće imaju terminale kojima preko modema pristupaju glavnom računaru. Ovaj model se može smatrati pretečom računarskih mreža i Interneta. Drugi operativni sistem, UNIX je uprošćena varijanta MULTICS sistema, koja je doživela praktičnu realizaciju i ekspanziju do današnjih dana. Ken Thompson, jedan od naučnika i programera kompanije Bell Labs, koji je radio na razvoju projekta MULTICS, napisao je za računar PDP-7 mini verziju MULTICS sistema. Posle toga, nastao je UNIX (UNI = jedan , X = CS = *Computing Service*).

Računari treće generacije imali su naziv mini računari: prvi računar je DEC-ov (Digital Equipment Corporation) PDP-1, do tada najmanji i najjeftiniji računar. Koštao je tada samo $120.000.

U trećoj generaciji računarskih sistema, kontrolno-upravljačke funkcije se još više prebacuju na sam računar, zbog još većeg porasta brzina, veličine memorije, broja ulaznih i izlaznih jedinica, ali i zbog pojave multiprogramiranja. Dakle, čovek definitivno gubi mogućnost kontrole i upravljanja internom situacijom u računarskom

sistemu i sve što je moguće prebacuje se na računarski sistem, odnosno na pojedine sistemske programe. Skup svih tih programa naziva se jednim imenom: operativni sistem. Programer se oslobađa niza složenih rutinskih poslova i pruža mu se mogućnost većeg angažovanja na kreativnom delu posla. No, pored kontrolno-upravljačkih programa u računarima treće generacije razvijen je i čitav niz uslužnih programa, čiji je zadatak da dalje olakšaju i pojednostave upotrebu računarskih sistema. Zbog toga, prema nameni, možemo izvršiti podelu softvera na sistemski softver i korisnički, problemski ili aplikativni softver.

U četvrtoj generaciji računara (1980-1990) prvi put se pojavljuju personalni računari. Razvoj personalnih računara započet je pojavom LSI čipova, odnosno čipova visokog stepena integracije (*Large Scale Integration*). Računari su bili dovoljno jeftini, tako da su ih mogli priuštiti i više odseka iste firme ili univerziteta, dok su personalni računari postali dovoljno jeftini da ih mogu priuštiti i pojedinci. Poznatiji personalni računari su Spectrum, Commodore, Atari, zatim IBM PC, Apple Macintosh, itd. U prve operativne sisteme za personalne računare spadaju MS-DOS i UNIX.

Paralelno sa razvijanjem korisničkog softvera, razvija se i korisnički interfejs programa, odnosno dolazi do razvoja korisničkog okruženja. Na taj način se omogućava osobama koje nisu potkovane detaljnim poznavanjem računarskog sistema i samih programa da te programe uspešno koriste.

Pored klasičnih operativnih sistema javljaju se i dve nove vrste, a to su mrežni operativni sistemi i distribuirani operativni sistemi.

Karakteristika mrežnih operativnih sistema su računari povezani u mrežu. Ovi računari zadržavaju relativno visok stepen autonomije - svaki računar ima svoj operativni sistem, a u mogućnosti su da međusobno razmenjuju podatke pomoću odgovarajućih protokola. Operativni sistemi mogu biti različiti, potreban je samo zajednički protokol, odnosno zajednički jezik za komunikaciju. Korisnik jednog računara može se prijaviti na drugi, preuzeti neke datoteke itd. Korisnik zna da nije sam u mreži, odnosno svestan je različitih računara sa kojima komunicira preko mreže.

Distribuirani operativni sistemi su mnogo ozbiljnija varijanta u mrežnom okruženju, zato što osim deljenja i migracije datoteka i štampača omogućavaju i deljenje procesa, odnosno programa. Korisnici ovaj sistem vide kao jednoprocesorski sistem, ali se, u stvari, radi o operativnom sistemu namenjenom za rad sa više procesora koji su fleksibilno povezani preko mreže. To znači da postoji više računara povezanih u mrežu, ali samo jedan operativni sistem, koji upravlja svim resursima u mreži. U pravom distribuiranom sistemu korisnik ne treba da vodi računa o tome gde su njegovi njegove datoteke smeštene ili gde se izvršava njegov program - to je posao distribuiranog operativnog sistema. Distribuirani operativni sistem se, dakle, ponaša kao jedinstvena celina. Korisnik ne mora znati da je umrežen sa drugim računarima - on ceo sistem vidi kao jedan računar.

1.2. Definicija i funkcije operativnog sistema

Operativni sistem objedinjuje raznorodne delove računara u skladnu celinu i sakriva od korisnika detalje funkcionisanja ovih delova koji nisu bitni za korišćenje računara. Kao takav, operativni sistem:

- upravlja programima, podacima i delovima od kojih se računar sastoji (procesor, kontroleri, radna memorija), sa ciljem da oni budu što celishodnije upotrebljeni,
- kreira pristupačno radno okruženje za krajnjeg korisnika računara, tako što pretvara računar od mašine koja rukuje bitima, bajtima i blokovima u mašinu koja rukuje datotekama i procesima.

U opštem smislu, operativni sistem se može definisati kao skup programa koji upravljaju resursima računarskog sistema i pružaju interfejs ka korisniku.

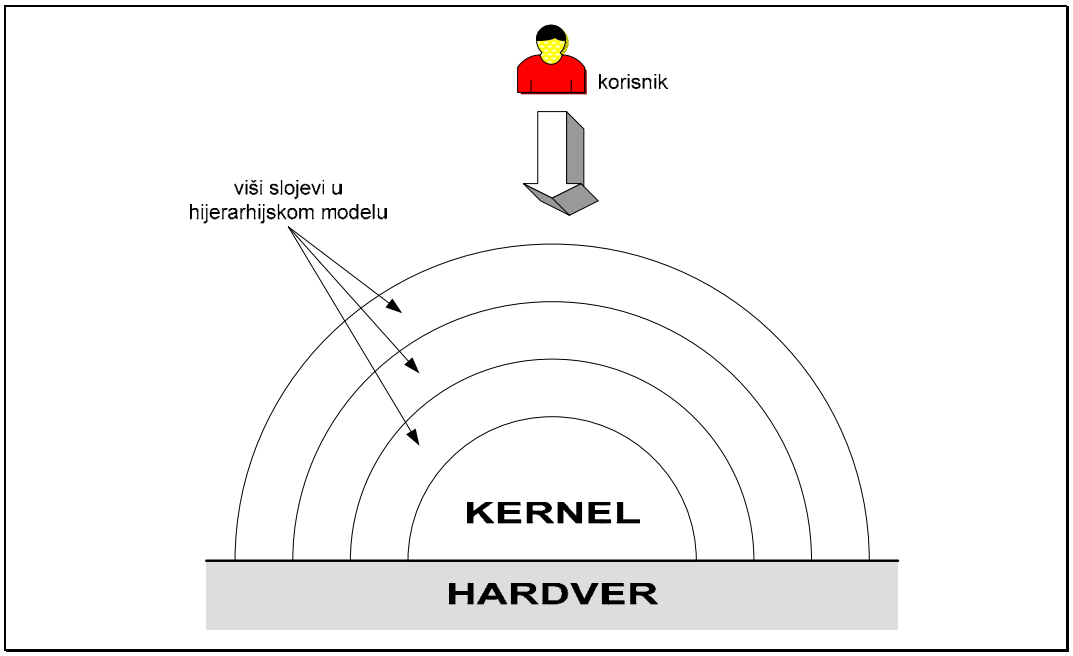
Prva funkcija operativnog sistema je upravljanje resursima računara (*resource management*). Pod pojmom resurs obuhvatićemo sve što je programu potrebno za rad. Resursi mogu biti hardverski (procesor, memorija, ulazno/izlazni uređaji) i softverski (programi, podaci, odnosno datoteke svih vrsta). Zadatak operativnog sistema jeste da vodi računa o resursima računara, odnosno da zadovolji potrebe programa, da prati koji program koristi koje resurse itd. Na primer, ukoliko dva korisnika višekorisničkog sistema istovremeno žele nešto da štampaju, operativni sistem je dužan da obezbedi dostupnost štampača programima tih korisnika i da obezbedi se podaci poslati na štampu ne mešaju.

Kako je arhitektura računara i hardvera krajnje komplikovana za programiranje za većinu korsinika, operativni sistem takođe obezbeđuje takozvanu virtuelnu mašinu, odnosno jedinstven pogled na računarski sistem sa tačke gledišta korisnika, nezavistan o konfiguraciji računara, konkretnom hardveru i samoj arhitekturi. Ova nezavisnost može biti izražena u većoj ili manjoj meri. Kao primer, možemo uzeti NEC PD765 kontroler za disketni uređaj (koristi se na personalnim računarima). Taj kontroler ima 16 komandi. Osnovne komande, READ i WRITE, zahtevaju da se prilikom njihovog pozivanja navede 13 parametara smeštenih u 9 bajtova. Prilikom pristupa disketnom uređaju treba voditi računa o tome, da li je motor uključen, zatim treba naći odgovarajuću stazu, pa sektor itd. To bi trebalo uraditi svaki put kada želimo nešto da pišemo ili da čitamo sa diskete. Zadatak operativnog sistema kao virtuelne mašine jeste da te stvari radi umesto nas i da nam pruža neke funkcije višeg nivoa apstrakcije radi lakšeg pristupa hardveru.

Ukratko rečeno, operativni sistem je skup sistemskih programa koji ima ulogu posrednika između korisnika računara i računarskog hardvera, čiji je cilj:

- da izvršava korisničke programe i olakša rešavanje korisničkih problema
- da korišćenje računarskog sistema učini podesnijim za korisnika i
- da omogući što efikasnije iskorišćenje računarskog hardvera.

Kao takav, operativni sistem je jedan od najvažnijih i najsloženijih delova računarskog sistema, koji se sastoji od više relativno nezavisnih celina. Treba imati na umu da svaki proizvođač računara ima svoje operativne sisteme, pa je teško dati opštu strukturu operativnog sistema. Na slici 1.1. prikazan je hijerarhijski model operativnog sistema.



Slika 1.1. *Hijerarhijski model operativnog sistema*

Pojedini delovi predstavljaju nivoe operativnog sistema. Hijerarhijski model ima sledeći smisao: na posmatranom nivou operativnog sistema mogu se zahtevati usluge samo od njegovih nižih nivoa, a nikako od viših. Najniži sloj je poznat kao jezgro operativnog sistema (*nucleus*, kernel).

Ne postoje čvrsto definisana pravila koja regulišu raspodelu funkcija operativnog sistema po nivoima. Zbog odnosa veličine operativnog sistema i operativne memorije, većina operativnih sistema ne može da stane u operativnu memoriju. Zato se u memoriji uvek nalaze samo najvažniji delovi operativnog sistema, takozvani rezidentni delovi, koji aktiviraju i završavaju korisničke programe, vrše dodelu memorije i datoteka i obavljaju operacije ulaza-izlaza. Rezidentni deo operativnog sistema mora obavezno podržavati mehanizam prekida, jer je on osnova višeprogramskog rada i komuniciranja računara sa spoljnim svetom. Deo operativnog sistema koji mora uvek biti prisutan u operativnoj memoriji obično se naziva jezgro ili nukleus. Funkcije koje koriste svi nivoi moraju se smestiti u jezgro operativnog sistema. Ostali delovi se ubacuju u memoriju kada su potrebni i izbacuju kada više nisu potrebni.

Funkcije operativnog sistema

Funkcije koje treba da obavlja operativni sistem mogu se izvesti na osnovu očekivanih funkcija jednog računarskog sistema, kao što su:

[1] Automatsko funkcionisanje računarskog sistema.

Operativni sistem mora da obezbedi funkcionisanje računarskog sistema bez operaterske intervencije, iz veoma prostog razloga što su ljudske intrevencije veoma spore u odnosu na računar. U aktivnosti koje operativni sistem treba da izvršava bez ljudske intervencije spadaju, na primer, punjenje memorije programom sa sekundarnih memorija i izvršavanje jednog ili više programa.

[2] Mogućnost planiranja i raspoređivanje poslova i postojanje jezika za upravljanje poslovima.

Pod planiranjem i raspoređivanjem poslova (*scheduling*) podrazumeva se određivanje koji se poslao koji je spreman za rad izvršavati, odnosno kome će se dodeliti glavni procesor. Radi kontrole automatske sekvence, operativni sistem uvodi kontrolne naredbe, koje služe za kontrolu rada celog računarskog sistema. Preciznije, uvodi se jezik za upravljanje poslovima (*job control language*), koji se intepretira preko operativnog sistema.

[3] Multiprogramiranje.

Multiprogramiranje je tehnika za pokretanje više programa na istom računaru istovremeno, tako što svaki dobije deo memorije, a procesor se dobija prema funkciji za raspoređivanje poslova.

[4] Eliminisanje zavisnosti I/O operacija.

Kako su I/O operacije veoma spore u odnosu na procesor, operativni sistem mora da izoluje I/O operacije od procesora, što se postiže upotrebom brzog medijuma za privremeno memorisanje svih I/O podataka. Taj uređaj je u prošlosti bila traka, a danas je to po pravilu disk. U cilju efikasnosti, I/O operacije po mogućstvu treba što više preklapati ili kombinovati sa drugim procesorskim poslovima. U tom kontekstu, realizovane su dve hardverske strukture: kanal (*channel*) i tehnika prekida (*interrupt*). Pod kanalom se podrazumeva uređaj koji kontroliše jednu ili više periferala, a sposoban je da obavlja transfer podataka između periferala i memorije bez intervencije glavnog procesora. Kanal po pravilu za takve transfere koristi usluge kvalitetne hardverske komponente koja se zove DMA kontroler (*Direct Memory Access*). Tehnika prekida je univerzalan metod za obaveštavanje operativnog sistema da se neka akcija, odnosno I/O komanda završila. Tehnika prekida obezbeđuje visoke performanse i izolovanost od I/O uređaja, DMA prenosu podatke po kanalu, a prekid odgovarajućim signalom obaveštava sistem da je transfer završen. Prekidnim signalom se prenosi upravljanje centralnim procesorom na neku drugu lokaciju (fiksni), pri

čemu se čuva prethodna vrednost programskog brojača, čime se omogućava da se prekinuti program nastavi.

Dakle, na osnovu same definicije operativnog sistema i očekivane funkcionalnosti jednog računarskog sistema zaključuje se da operativni sistem mora obavljati sledeće funkcije:

- upravljanje poslovima (sekvenciranje i raspoređivanje poslova) i interpretacija komandnog jezika,
- rukovanje ulazno/izlaznim operacijama,
- rukovanje greškama i prekidima,
- upravljanje resursima,
- višestruki pristup,
- zaštita resursa od zlonamernih napada, slučajnih grešaka korisnika i grešaka u korisničkim programima i samom operativnom sistemu.

Dodatno mogu se izdvojiti i sledeće dve funkcije:

- obezbeđivanje dobrog interfejsa za operatora i korisnika,
- obračun korišćenja računarskih resursa.

1.3. Karakteristike operativnih sistema

Karakteristike operativnih sistema su:

[1] Konkurentnost.

Konkurentnost (*concurrency*) je postojanje više simultanih, paralelnih aktivnosti. Primeri su preklapanje I/0 operacija i operacija izračunanja ili koegzistencija više programa u memoriji. Konkurentnost dovodi do problema prelaska sa jedne aktivnosti na drugu, probleme zaštite jedne aktivnosti od druge i sinhronizacije aktivnosti koje su međusobno zavisne.

[2] Deoba resursa.

Konkurentne aktivnosti mogu da zahtevaju deljenje (*sharing*) resursa ili informacija. Razlozi za deljenje resursa su višestruki. Pre svega, to je ukupna cena, jer je obezbeđivanje dovoljno resursa za svakog korisnika najčešće preskupo. Poželjno je nadovezivanje rada jednog korisnika na rad drugog korisnika, deljenje podataka, odnosno korišćenje istih podataka između različitih programa i otklanjanje redundanse resursa.

[3] Postojanje dugotrajne memorije.

Potreba za deljenjem programa i podataka implicira potrebu za trajnim skladištenjem podataka sa mogućnošću brzog pristupa (*long-term storage*). To omogućavaju uređaji masovne ili sekundarne memorije, bazirane uglavnom na magnetnim uređajima. Problemi koje je pri tome potrebno razrešiti su: obezbeđenje jednostavnog pristupa podacima, zaštita od štetnih uticaja bilo kakve vrste (zlonamernih ili nehotičnih) i zaštita od grešaka samog sistema.

[4] Nedeterminizam.

Operativni sistem mora biti deterministički orijentisan, što znači da kad izvršava isti program sa istim podacima mora da daje isti rezultat, bez obzira da li će to raditi danas, sutra, ili za mesec dana. Na drugoj strani, operativni sistem se mora okarakterisati nedertemnističkim ponašanjem (*nondeterminancy*), što znači da mora da odgovori na masu zahteva i događaja koji se mogu desiti na nepredvidiv način (zahtevi za resursima, greške u toku izvršavanja programa, prekidni signali perifernih uređaja). Operativni sistem ne može predvideti sve situacije koje mogu nastupiti i mora da bude spreman za sve moguće sekvence događaja.

Poželjne osobine operativnog sistema

U poželjne osobine operativnog sistema spadaju:

[1] Visoki nivo efikasnosti.

Efikasnost operativnog sistema se ne može precizno predstaviti pomoću jednog kriterijuma, već samo pomoću više kriterijuma, pri čemu značaj svakog kriterijuma u globalnoj predstavi zavisi od vrste i namene operativnog sistema. Kriterijumi su sledeći:

- srednje vreme između poslova,
- vreme neiskorišćenosti operativnog sistema,
- vreme prolaska (*turn-around time*) za *batch* poslove,
- vreme odziva za interaktivne sisteme (*response time*),
- iskorišćenost resursa (*resource utilisation*) i
- propusna moć (broj poslova po satu, odnosno ukupna veličina posla koji može biti obavljen interaktivno u nekom vremenu).

Uvešćemo na ovom mestu merilo efikasnosti (*efficiency*) e kao odnos vremena kada procesor radi korisne stvari i ukupnog vremena da bi se taj posao obavio: $e=t_{korisno}/t_{ukupno.}$ Pri tome, e se mora nalaziti u intervalu od 0 do 1 ($0<e<1$).

Uvešćemo, takođe, merilo gubitka vremena o (*overhead*) kao odnos vremena kada procesor radi poslove vezane za održavanje samog sistema (*housekeeping jobs*) i ukupnog vremena: $o=t_{održavanje}/t_{ukupno}$. Takođe važi $0<o<1$ i $e+o=1$.

[2] Visok nivo pouzdanosti.

Idealno, operativni sistem bi trebao da bude potpuno bez grešaka, ali to ne može biti realan slučaj. Merilo pouzdanosti je broj grešaka, odnosno srednje vreme između dve greške ili dva otkaza.

[3] Jednostavnost održavanja

Potrebno je omogućiti održavanje sistema, ali sa što manjim brojem ljudi koji će na održavanju raditi. Da bi se to postiglo, potrebna je modularna struktura, sa jasno definisanim interfejsima među modulima i, naravno, sa dobrom pratećem dokumentacijom.

[4] Prihvatljiva veličina.

Potreban prostor u memoriji za smeštanje operativnog sistema treba da bude manji, ali to sa povećanjem količine savremenih memorijskih sistema gubi na važnosti. Na drugoj strani, veliki sistemi su podložni greškama, ali i njihov razvoj traje duže.

Ciljevi projektovanja operativnog sistema

Ciljevi koji se postavljaju kod projektovanja operativnog sistema mogu se podeliti na sistemske i korisnički ciljeve. Što se korisničkih ciljeva tiče, operativni sistem treba da bude podesan za korišćenje, lak za učenje, pouzdan, bezbedan i brz. Što se sistemskih ciljeva tiče operativni sistem treba da bude jednostavan za projektovanje, implementaciju, održavanje, da bude fleksibilan, pouzdan, bez grešaka i efikasan.

Tradicionalno, operativni sistemi koji pripadaju prošlosti pisani su na asemblerskom jeziku, što je svakako imalo svoje prednosti i mane. Prednosti su svakako u kompaktnosti izvršnog koda koji je minimalan, a po performansama kod je svakako optimalan. Mane se ogledaju u velikoj količini programerskog vremena za realizaciju koda. Zahvaljući C jeziku i UNIX sistemu, operativni sistemi se danas pišu u višim programskim jezicima.

Implementacija u višem programskom jeziku donosi masu prednosti:

- izvorni kod se mnogo brže piše,
- izvorni kod je mnogo kompaktiniji,
- kod se mnogo lakše razume i lakše se otklanjaju greške (debug) i
- operativni sistem napisan na višem programskom jeziku se mnogo lakše prenosi-portira na drugu računasku arhitektru, odnosno drugu vrstu procesora.

Operativni sistemi se generišu da rade za više klasa računara, odnosno procesora. Sa takvim konceptom, operativni sistem mora da se konfiguriše za svaku klasu procesora, posebno.

Jedna od posebnih celina koja mora da se realizuje kod svih operativnih sistema, a takođe je veoma zavisna od hardvera i arhtekture jeste podizanje operativnog sistema (*booting*), odnosno pokretanje operativnog sistema punjenjem kernela sa diska u memoriju. U tom kontekstu, realizuje se poseban program za podizanje operativnog sistema (*bootstrap* program) fiksni program koji se obično čuva u ROM memoriji, ali može biti i na fiksnim pozicijama na disku, koji je sposoban da locira kernel, napuni kernel u memoriju i preda kernelu kontrolu, odnosno otpočne izvršenje kernelskog koda koji nastavlja dalje sa podizanjem operativnog sistema.

1.4. Vrste operativnih sistema

Postoje brojne podele operativnih sistema na osnovu različitih kriterijuma: prema broju korisnika i/ili procesa, prema načinu obrade poslova, prema distribuciji procesorske snage i ostalih resursa, prema nameni i funkcionalnim osobinama.

Klasifikacija prema broju korisnika i procesa

Prema broju korisnika, operativni sistemi se mogu podeliti na jednokorisničke (*single-user*) i višekorisničke (*multiuser*) sisteme.

Jednokorisnički sistemi, kao što im samo ime govori, obezbeđuju virtuelnu mašinu za samo jednog korisnika. To su ili računarski sistemi prilagođeni za jednu funkciju ili je reč o slabijim i jeftinijim konfiguracijama tipa mikroračunara. Uglavnom ih karakteriše jeftin hardver, solidna prateća programska podrška, jednostavan sistem datoteka, jednostavan I/O sistem, a u prošlosti i relativno niske performanse. Međutim, prilike su se izmenile, hardver mikroračunara je ostao jeftin, ali su se performanse drastično poboljšale, što je dovelo do realizacije vrlo kvalitetnih operativnih sistema koji su i dalje jednokorisnički ali višeprocesni, odnosno sposobni da obavljaju više simultanih aktivnosti.

Višekorisnički sistemi su kvalitetni operativni sistemi koji zahtevaju jače hardverske konfiguracije. Tipičan višekorisnički sistem je UNIX, koji obezbeđuje simultani pristup za više korisnika istovremeno, pri čemu korisnici pristupaju sistemu preko posebnih terminala.

Prema broju simultanih aktivnosti, odnosno prema broju procesa koji se mogu izvršavati paralelno ili kvaziparalelno, operativni sistemi se mogu podeliti na jednoprocesne (*singletasking, singleprocess*) i višeprocesne (*multitasking*, *multiprocess*).

Na osnovu kombinovanog kriterijuma mogu se izdvojiti tri vrste operativnih sistema:

- jednokorisnički jednoprocesni (*single-user*, *singletasking*), kao što je MS-DOS
- jednokorisnički višeprocesni (*single-user*, *multitasking*), kao što su OS/2 i MS Windows 3.1/ 9x/ME
- višekorisnički višeprocesni (*multiuser*, *multitasking*), kao što je UNIX, ali uslovno se mogu prihvatiti i MS Windows 2000/XP ukoliko obezbeđuju terminal servise (*terminal services*).

Klasifikacija prema nameni i načinu obrade poslova

Prema nameni, operativni sistemi se mogu podeliti na operativne sisteme opšte namene (*general purpose systems*), koji mogu da obavljaju razne poslove, kao što su obrada teksta i slike i operativne sisteme specijalne namene, koji, po pravilu, služe za kontrolu procesa.

Prema načinu obrade poslova, operativni sistemi se mogu klasifikovati kao:

[1] Sistemi sa grupnom obradom (*batch*)

Grupna (serijska, *batch*) obrada je takav način rada računara u kome korisnici predaju svoje poslove na izvršenje posredstvom ulaznih jedinica, i koji se zatim odvijaju jedan za drugim u nizu, pri čemu korisnik nema mogućnost komuniciranja sa svojim poslom.

[2] Interaktivni sistemi (*interactive systems*)

Interaktivne sisteme (nazivaju se još i *time-sharing* sistemi) karakteriše postojanje terminala za svakog korisnika, preko kojih korisnici zadaju poslove i komuniciraju sa svojim poslovima. Paralelnost u radu se postiže tako što se svakom korisničkom programu dodeljuje jedan kvantum vremena centralnog procesora, tako da je odziv na svaku poruku korisnika u granicama od nekoliko sekundi. Po isteku vremenskog kvantuma dodeljenog jednom procesu, on se prekida, bez obzira da li je završio sa radom ili ne, a procesor se dodeljuje sledećem procesu u redu čekanja.

[3] Kombinovani sistemi

Klasifikacija prema funkcionalnim osobinama računarskog sistema

Prema funkcionalnim osobinama računarskog sistema za koji su namenjeni, operativni sistemi se mogu podeliti na:

- operativne sisteme za *mainframe* sisteme,

- operativne sisteme za sisteme sa deljenjem vremena,
- operativne sisteme za desktop sisteme,
- operativne sisteme za višeprocesorske sisteme,
- mrežne operativne sisteme,
- distribuirane sisteme,
- udružene (*clustered*) sisteme,
- operativne ssiteme za upravljanje u realnom vremenu (*real-time*) i
- ručne (*hand-held*) sisteme.

Veliki računarski sistemi (*mainframe systems*) su prvi računarski sistemi na kojima su se pokretale mnoge komercijalne i naučne aplikacije. U svoje vreme bile su to fizički ogromne mašine kontrolisalne preko upravljačke konzole. Zajednički ulazni uređaji su bili čitači bušenih kartica i magnetne trake, a izlazni uređaji, takođe, bušene kartice, trake ili štampači. Da bi se ubrzao rad sistema kod koga je priprema posla bila vremenski kritična, vreme pripreme je smanjivano tako što su se slični poslovi udruživali u grupe koje bi se zajednički obrađivali. Uvodi se tehnika mutliprograming, a sa njom i automatsko sekvenciranje poslova, koje bez operatora prebacuje kontrolu sa jednog posla na drugi. Da bi operativni sistem mogao da funkcioniše, njegov fundamentalni deo mora uvek biti prisutan u memoriji (*resident monitor*), koji bira posao koji će se trenutno izvršavati. Po završetku posla ili nakon blokade zbog čekanja na ulazno-izlazne operacije, operativni sistem opet preuzima kontrolu i dodeljuje kontrolu drugom poslu.

Sistemi sa deljenjem vremena (*time-sharing systems*) nastali kao specijalna klasa velikih sistema u kojoj je svakom korisniku omogućena *on-line* komunikacija sa svojim poslom i operativnim sistemom. Vreme odziva se uvodi kao novi kriterijum za ocenu efikasnosti ove klase sistema. Opet imamo mutliprograming tehniku, što znači više poslova koji se u memoriji nalaze istovremeno i tehniku deljenja vremena, gde svaki posao mora da preda kontrolu drugom poslu posle isteka vremenskog kvantuma i sačeka svoje novo vreme.

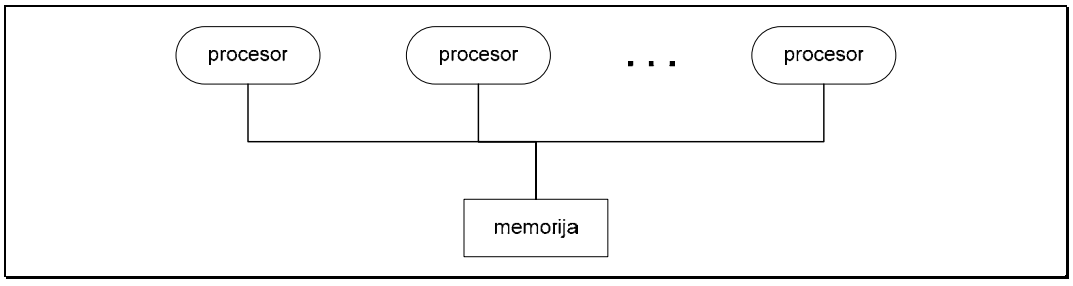
Ove sisteme podelićemo na dve posebne klase: upitni sistemi (*file interrogation systems*) i transakcioni sistemi (*transaction processing systems*). Kod upitnih sistema, kao što su razni informacioni sistemi koji rade sa velikim bazama podataka, veoma je bitno da vreme odziva za svaki postavljeni upit bude veoma kratko. Uobičajeno postoji dodatni sloj na vrhu operativnog sistema za upravljanje bazom podataka. Transakcioni sistemi su relativno slični upitnim, s tim što se kod njih često (na primer, nekoliko puta u sekundi) vrše modifikacije baze podataka. Primer su sistemi za rezervaciju karata i bankarsko poslovanje. Ažurnost baze podataka je od kritičnog značaja za transakcione sisteme.

Desktop sistemi (*desktop systems*) su mali računarski sistemi namenjeni za jednog korisnika, dostupni po pristupačnim cenama. Prve verzije desktop sistema imale

su skromne performanse i hardverom (na primer, mala količina memorije i diskovi manjih kapaciteta), ali se ta situaciija drastično promenila. Osim povećanja performansi, kod ovih sistema je, razvojem pogodnih grafičkih korisničkih interfejsa, razvijena ergonomičnost, odnosno prilagođenost korisniku. Za razliku od velikih računarskih sistema, ova klasa je veoma fleksibilna i prilagodljiva za nove tehnologije, nove procesore, nove memorije, nove multimedijalne i druge periferale. Brojna rešenja namenjenja velikim sistemima se primenju za ovu klasu računarskih sistema (kao što su, na primer, tehnike keširanja). Ovu klasu karakteriše realizacija više vrsta operativnih sistema (DOS, MS-Windows, Novell-Netware, UNIX, Linux).

Višeprocesorski sistemi (*multiprocessor systems*) su sistemi sa više procesora koju su čvsto povezani zatvorenim komunikacionim linijama, a nalaze se u istom kućištu. U literaturi se pominju pod imenom paralelni sistemi. Karakteristično za ovu klasu sistema je to da procesori dele istu memoriju i sistemski časovnik. Komunikacija i sinhronizacija između procesora se obično odvija preko deljive memorije. Prednosti višeprocesorkih sistema su povećanje brzine, ekonomičnost i povećanje pouzdanosti. Što se tiče povećanja brzine, N procesora bi u idealnom slučaju radilo N puta većom brzinom, ali zbog međuprocesorske sinhronizacije, to povećanje je manje od N. Ekonomičnost se postiže zato što se dele isti hardverski resursi, memorija, časovnik, DMA kanali, prekidni kontroleri, i korsiti jedan izvor napajanja. Povećanje pouzdanosti se postiže tako što će sistem nasatviti da radi i u slučaju otkaza jednog ili više procesora.

Postoje dva koncepta za realizaciju višeprocesorskih sistema, a to su simetrično multiprocesiranje (*symmetric multiprocessing*, *SMP*), prikazano na slici 1.2. i asimetrično multiprocesiranje (*asymmetric multiprocessing*). U slučaju simetričnog multiprocesiranja, svaki procesor izvršava istu kopiju operativnog sistema, pri čemu te kopije komuniciraju jedna sa drugom kad god je potrebno. Svi procesori su ravnopravni, odnosno nema odnosa gospodara-roba (*master-slave*). U idealnom slučaju, svakom procesoru se dodeljuje jedan proces i oni se izvršavaju nezavisno bez gubitaka performasi. Većina modernih operativnih sistema podržava SMP. U slučaju asimetričnog multiprocesiranja, svaki procesor ima dodeljen specifičan posao. Postoji jedan glavni procesor (*master*) koji potpuno kontroliše ceo sistem i koji dodeljuje poslove ostalim procesorima (*slaves*). Ova koncepcija se koristi, uglavnom, u ektremno brzim i velikim sistemima.

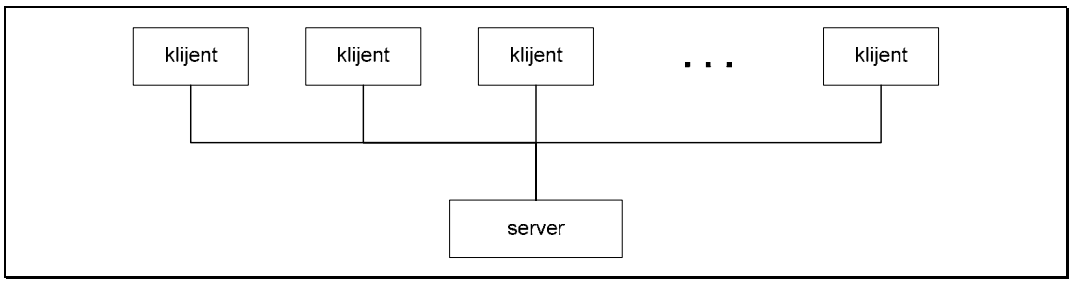


Slika 1.2. *SMP*

Mrežni operativni sistemi (*network OS*) obezbeđuju takvu okolinu u kojoj korisnici sa svojih lokalnih mašina mogu pristupiti resursima udaljenih mašina na dva načina: procedurom udaljenog prijavljivanja na sistem (*remote login*) ili razmenom datoteka sa udaljenim sistemom (*remote file transfer*). Posebnu, mnogo kvalitetniju klasu mrežnih operativnih sistema predstavljaju distribuirani sistemi, kod kojih se gubi koncept lokalne i udaljene mašine.

Distribuirani sistemi (*distributed systems*) predstavljaju kolekciju procesora odnosno računara koji ne dele zajedinčku memoriju i sistemski časovnik. Umesto toga, svaki procesor, odnosno računar ima sopstvenu lokanu memoriju, a međusobna komunikacija se ostvaruje putem mreže realizovane kao LAN ili WAN. Osim podataka, datoteka i štampača, distribuiraju se i procesi. Četiri glavne prednosti distribuiranih sistema su deljenje resursa, ubrzavanje izračunavanja, pouzdanost i komunikacije. Neophodno je naglasiti da distribuirane sisteme karekterišu tri vrste migracija: migracija podataka, migracija izračunvanja i migracija procesa.

Distribuirani sistemi zahtevaju mrežnu infrastrukturu i mogu biti realizovani ili kao klijent-server arhitektura prikazana na slici 1.3., ili kao ravnopravni računarski sistemi koji po mreži dele resurse (*peer-to-peer systems*). U klijent-server arhitekuri, postoje računari koje predstavljaju servere i računari koji koriste njihove usluge - klijenti (*clinets*). Postoje dve vrste servera: serveri za izračunavanje (*computing servers*), kojima klijenti šalju zahteve na obradu i serveri podataka (*file server*) koji služe za smeštanje datoteka. Treba svakako naglasiti i treću vrstu servera, a to su serveri za štampače (*print server*) koji omogućavaju da klijenti obave štampanje.



Slika 1.3. *Klijent-server arhitektura*

Udruženi sistemi (*clustered system*) se sastoje od udruženih računara, odnosno od dva ili više nezavisnih računara koji dele diskove (*strorage*) i čvrsto su povezani LAN mrežom. Svaki računar u udruženom sistemu se naziva čvor (*node*). Jedan nivo softvera za udruživanje se izvršava na svakom čvoru. Postoje dve vrste udruživanja: asimetrično i simetrično. U asimetričnom udruživanju (*asymmetric clustering*) jedan server (čvor) izvršava aplikaciju, dok ostali - prateći serveri (*monitoring servers*) prate rad glavnog servera u budnom ali neaktivnom stanju. U slučaju otkaza glavnog servera jedan od pratećih servera preuzeće njegovu ulogu. U simetričnom udruživanju (*symmetric clustering*), svi serveri su aktivni i izvršavaju aplikaciju čime se drastično poboljšavaju

performase, ali sistem mora da izdrži otkaz jednog ili više servera kao i u perthodnom slučaju.

Sistemi za upravljanje u realnom vremenu (*real-time systems*) praktično spadaju u klasu specijalnih operativnih sistema. Glavna karakteristika ovih sistema je davanje odziva u propisanom vremenskom intervalu, precizno definisana vremenska ograničenja u kojima moraju da se dogode odzivi. Po pravilu, glavni izvori podataka uglavnom su senzorski uređaji, kao i razni prekidači, A/D i D/A konvertori. To su sistemi koji kontrolišu aplikacije specijalne namene, kao što su: specijalni grafički sistemi, sistemi za medicinsku grafiku, sistemi za industrijsku kontrolu, kontrolisanje naučnih eksperimenata itd. Dele se na dve klase: čvrsti sistemi za rad u realnom vremenu (*hard real-time systems*) i meki sistemi za rad u realnom vremenu (*soft real-time systems*). Čvsti realni sistemi garantuju da će se svi kritični poslovi obaviti na vreme. Po pravilu, nemaju sekundarnu memoriju, odnosno diskove jer najčešće rade u teškim uslovima, a kompletan kod i operativnog sistema i aplikacija čuva se u ROM memoriji. Takođe, sam operativni sistem je limitiran tako da mnoge funkcije operativnih sistema opšte namene uopšte ne postoje. Meki realni sistemi su manje zahtevni, oni obezbeđuju dve klase procesa, a to su obični procesi i procesi u relanom vremenu koji imaju apsolutni prioritet. Praktično, oni su mešavina sistema za rad u realnom vremenu i operativnih sistema opšte namene. Meki realni sistemi su, praktično, sve vrste UNIX i Linux sistema, Windows 2000/XP.

Ručni sistemi (*handheld systems*) uključuju PDA uređaje (*Personal Digital Assistants*) i celularne telefone. Glavne karaktersitike ovih sistema su mobilnost (korisnici ih mogu nositi svuda sa sobom) i mogućnost priključivanja na mrežu kao što je Internet. Karakterišu ih veoma slabi hradverski resursi, kao što su mala količina memorije (512KB-8MB), procesori skromnih mogućnosti i mali ekrani. Te skromne komponente daju im skromne performanse i mogućnosti. Neki se mogu bežičnim putem priključiti na mrežu, što ih čini veoma atraktivnim.

Vrste računarskog okruženja

U nastavku teksta ukratko su opisane tri vrste računarskog okruženja (*computing environments*):

[1] Tradicinalno (*traditional computing*).

Pod tradicionalnim okruženjem podrazumevamo jednokorisničke računare ili sisteme sa deljenjem vremena, gde se korisnici preko svojih terminala povezuju na servere.

[2] Web-bazirano (*web-based computing*).

Web-bazirano okruženje zasnovano je na umrežavanju zasnovanom na principima globalne mreže (kao što je Internet) na kojoj postoje web serveri i klijenti u vidu PC računara i malih mobilnih sistema.

[3] Ugrađeno okruženje (*embedded computing*).

Ugrađeno okruženje je tipično za tvrde *real-time* sisteme koji se karakterišu nepostojanjem tastarure, monitora i diskova. Aplikacija koja se izvršava u realnom vremenu dobija ulazne informacije preko uređaja kao što su senzori, dok se statusne i izlazne informacije prikazuju na led diodama ili nekom malom displeju.

1.5. Opšti pregled strukture operativnih sistema

Upravljanje osnovnim resursima računarskog sistema obezbeđuje više funkcionalnih grupa programa:

- upravljanje procesorom ,
- upravljanje memorijom,
- upravljanje ulazom i izlazom,
- upravljanje podacima,
- upravljanje sekundarnom memorijom,
- umrežavanje,
- zaštita i
- korisnički interfejs.

Ponekad je teško precizno reći (usaglasiti) šta se podrazumeva pod delovima operativnog sistema, odnosno da li neki od kontrolnih programa pripadaju ili ne pripadaju operativnom sistemu. Prve četiri grupe su fundamentalne i kao takve su ukratko opisane u nastavku teksta.

Procesor je jedan od najvažnijih resursa računarskog sistema, mada u pojedinim konkretnim situacijama neki drugi resursi mogu biti kritičniji. Upravljanje procesorom se može podeliti na dva nivoa:

- nivo neposredne dodele procesora nekom programu, tj. predaje kontrole nad procesorom nekom programu i
- nivo razrešavanja prioriteta, odnosno odlučivanja koji od mogućih programa ima najveći prioritet da bi postao proces i da bi u nekom sledećem trenutku vremena dobio kontrolu nad procesorom.

Pod upravljanjem memorijom podrazumeva se upravljanje operativnom, odnosno glavnom memorijom računara, kojoj centralni procesor pristupa direktno, radi uzimanja instrukcija ili podataka. Na ovom nivou obavljaju se sledeće funkcije:

- realizacija određene strategije dodeljivanja memorije (redosled dodeljivanja memorije poslovima, statično ili dinamično dodeljivanje, principi dodeljivanja, itd.),
- dodeljivanje memorije (algoritmi izbora početne lokacije segmenta koji se dodeljuje),
- sprovođenje određene strategije oslobađanja memorije (ukrupnjavanje manjih oslobođenih delova memorije, oslobađanje delova memorije, redosled obrade zahteva za dodelu memorije).

Na nivou upravljanja uređajima realizuju se sledeće funkcije:

- obezbeđivanje nezavisnosti uređaja (programi treba da budu nezavisni od tipa uređaja koji se koriste za ulazno-izlazne operacije),
- obezbeđivanje efikasnog rada uređaja (pošto U/I operacije predstavljaju često usko grlo računarskog sistema, poželjno je da se one efikasno koliko god je to moguće preklapaju i izoluju od procesora),
- obezbeđivanje jedinstvenog koncepta analize rada svih uređaja računarskog sistema,
- realizacija određene strategije dodeljivanje uređaja (na primer, redosled dodeljivanja uređaja poslovima, način dodeljivanja uređaja: namenski, sa deobom, kao virtuelni, itd.),
- dodeljivanje uređaja (fizičko dodeljivanje uređaja, kontrolnih jedinica i kanala poslovima),
- realizacija određene strategije oslobađanja uređaja (na primer, uređaj se oslobađa tek kad se posao kojem je dodeljen završi).

Nivo upravljanja podacima treba da obezbedi softverska sredstva za organizovanje i pristupanje podacima na način koji odgovara korisniku računarskog sistema. Koncepcija upravljanja zavisi, pre svega, od vrste podataka i režima njihovog korišćenja. Na ovom nivou realizuje se sledeće funkcije:

- formiranje i brisanje osnovnih struktura podataka (datoteka),
- čitanje datoteka i upisivanje u datoteke,
- obezbeđivanje uslova za simboličko obraćanje datotekama (prema njihovom imenu),
- zaštita podataka od namernog i nenamernog uništavanja (usled otkaza sistema),
- zaštita podataka od neovlašćenog pristupa i korišćenja i
- deoba datoteka (podataka) između više poslova (korisnika).

U literaturi (Lister, Kvaternik), sloj upravljanja podacima se često objedinjuje sa slojem upravljanja sekundarnom memorijom. U tim slučajevima, ovaj sloj takođe

obavlja i funkcije organizacije memorijskog prostora na masovinim memorijama (najčešće diskovima).

U nastavku teksta opisane su dve koncepcije u dizajnu operatvnih sistema, a to su monolitna organizacija (*monolithic systems*) i slojevita organizacija (*layered systems*). Takođe uvodimo i treću, veoma savremenu koncepciju, a to je mikrokernel arhitektura.

Monolitni sistemi

U prošlosti monolitni sistemi su predstavljali najčešću organizaciju operativnih sistema. Ovaj način organizacije operativnih sistema dobio je naziv velika zbrka (*big mess*). Monolitni operativni sistem je realizovan kao skup procedura, koje se po potrebi mogu međusobno pozivati, bez ikakvih ograničenja. Korisnički programi koriste servise operativnog sistema na sledeći način: parametri sistemskog poziva se smeštaju u određena mesta, kao što su registri procesora ili stek, nakon čega sledi poziv kernela operativnog sistema (*kernel call*). Ova operacija prebacuje procesor iz korisničkog režima rada u sistemski režim rada i kontrolu predaje operativnom sistemu. U sistemskom režimu rada su dostupne neke komande procesora kojima se ne može pristupiti iz korisničkog režima. Posle poziva kernelu, operativni sistem preuzima kontrolu, na osnovu parametara poziva određuje koju sistemsku proceduru treba pozvati. Nakon izvršavanja procedure, kontrola se vraća korisničkom programu. Operativni sistem ima sledeću strukturu koja se sastoji iz tri osnovna skupa programa:

- glavni program koji obrađuje sistemske pozive,
- skup sistemskih procedura koje se pozivaju prilikom sistemskih poziva,
- skup pomoćnih procedura koje se koriste od strane sistemskih procedura.

Slojevita (hijerarhijska) realizacija

Kod slojevite realizacije (*layered systems*), operativni sistem se deli na različite slojeve. Slojevi su hijerarhijski organizovani: svaki sloj može da poziva samo funkcije nižih slojeva.

Prvi slojeviti operativni sistem, THE *(Technische Hogeschool Eindhove*n, E.W.Dijkstra), sastojao se iz šest slojeva. Slojevi od 0 do 3 predstavljaju jezgro operativnog sistema i rade u sistemskom režimu rada:

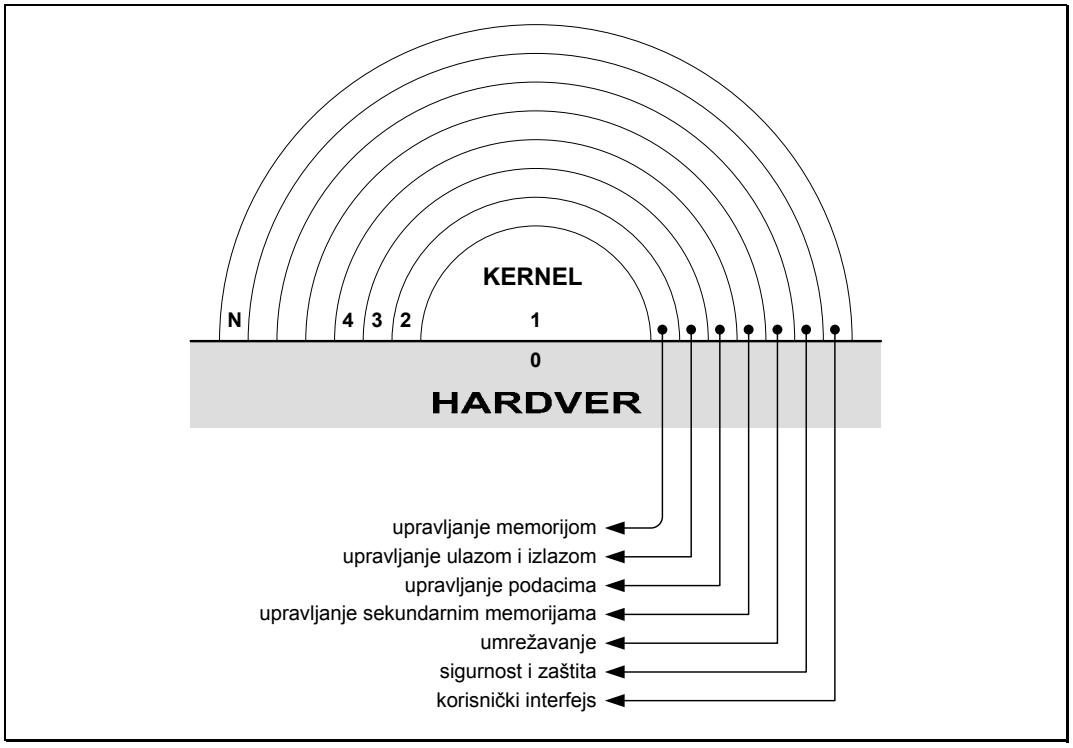
- nulti sloj upravlja procesorom, odnosno dodeljuje procesor različitim procesima,
- prvi sloj upravlja memorijom, odnosno dodeljuje potrebnu memoriju procesima,
- drugi sloj upravlja komunikacijom između različitih procesa i komandnog interpretera,
- treći sloj obavlja ulazno-izlazne operacije.

Na četvrtom sloju rade korisnički programi, koji se ne brinu ni oko dodele procesora, alokacije memorije, komandnog interpretera, ulazno-izlaznih operacija - sve te operacije obavljaju niži slojevi.

Znači:

- operativni sistem monolitne strukture se sastoji od skupa procedura bez ikakvog grupisanja ili hijerarhije;
- operativni sistem slojevite slojevite strukture se deli na više slojeva od kojih svaki ima tačno određenu funkciju (upravlja tačno određenim resursima) i oslanja se isključivo na funkcije nižih slojeva, kojima pristupa pomoću poziva sličnih sistemskim pozivima.

Generalno, slojeviti operativni sistem se deli na određeni broj nivoa (N), od koji se svaki gradi na vrhu prethodnog nivoa. Najniži nivo (*layer 0*) je hardver, a najviši nivo (*layer N*) je korisnički interfejs. Sa ovakvim modularnim konceptom, nivoi koriste funkcije i servise isključivo nižih nivoa. Jedna tipična slojevita organizacija je prikazana na slici 1.4.



Slika 1.4. *Slojevita organizacija operativnog sistema*

Slojeviti operativni sistem dozvoljava pozivanje (korišćenje) operacija stvaranja i uništavanja procesa samo iz sloja koji se u hijerarhiji nalazi iznad svih slojeva

operativnog sistema. To znači da je postojanje procesa isključivo vezano za korisnički sloj. Iako su svi procesi locirani u korisničkom sloju, oni su međusobno jasno razdvojeni zahvaljujući činjenici da svaki od procesa poseduje zaseban adresni prostor - korisnički prostor *(user spac*e). Na istom principu se zasniva i razdvajanje procesa i operativnog sistema. Ali, razdvojenost korisničkih prostora od adresnog prostora operativnog sistema, odnosno sistemskog prostora *(kernel spac*e), sprečava da se pozivi operacija operativnog sistema zasnivaju na korišćenju poziva potprograma. Zato je neophodno uvođenje posebnog mehanizma sistemskih poziva koji omogućuje prelazak iz korisničkog prostora u sistemski prostor radi poziva operacija operativnog sistema. Sistemski pozivi zahtevaju korišćenje specifičnih asemblerskih naredbi i zbog toga se sakrivaju unutar sistemskih potprograma. Neki od sistemskih potprograma, pored sistemskih poziva, sadrže i specifične obrade podataka, kao što je, na primer, pretvaranje brojeva iz znakovnog u binarni oblik i obrnuto kod formatiranog ulaza ili izlaza. Svaki od sistemskih potprograma je namenjen za pozivanje jedne od operacija operativnog sistema, namenjenih korisničkom sloju. Ovakve operacije se nazivaju sistemske operacije, kako bi se razlikovale od ostalih (internih) operacija operativnog sistema, namenjenih samo za koriršćenje unutar operativnog sistema. Sistemski potprogrami obrazuju sistemsku biblioteku. Prema tome, pozivanje sistemskih operacija se svodi na pozivanje potprograma iz sistemske biblioteke. Sistemska biblioteka se isporučuje uz operativni sistem kako bi se koristila u postupku linkovanja. U toku linkovanja njeni potprogrami se vezuju za objektni oblik korisničkog programa radi stvaranja izvršnog oblika korisničkog programa, koji tako postaje spreman za saradnju sa operativnim sistemom.

Zahvaljujući sistemskim potprogramima, odnosno sistemskoj biblioteci, operativni sistem predstavlja deo korisničkog programa, iako nije za njega direktno linkovan. Zato se može smatrati da obavljanje sistemskih operacija predstavlja sastavni deo izvršavanja korisničkog programa, odnosno pripada aktivnosti procesa, vezanog za ovo izvršavanje. To, uz istovremeno postojanje više procesa i nepredvidivost priključivanja, uzrokuje mogućnost da istovremeno postoji više procesa, koji su započeli, a nisu završili svoju aktivnost u okviru operativnog sistema (odnosno, čija aktivnost je zaustavljena unutar operativnog sistema).

Sistemski pozivi

Aplikacioni programi komuniciraju sa operativnim sistemom pomoću sistemskih poziva (*system calls*), odnosno preko operacija (funkcija) koje definiše operativni sistem. Sistemski pozivi se realizuju pomoću sistema prekida: korisnički program postavlja parametre sistemskog poziva na određene memorijske lokacije ili registre procesora, inicira prekid, operativni sistem preuzima kontrolu, uzima parametre, izvrši tražene radnje, rezultat stavlja u određene memorijske lokacije ili u registre i vraća kontrolu programu.

Sistemske pozive često podržava i hardver, tj. procesor, na taj način što razlikuje dva režima rada: korisnički režim (*user mode*) i sistemski režim (*supervisor mode*) rada. Korisnički programi mogu raditi isključivo u korisničkom režimu rada procesora, sistemski režim rada je predviđen za operativni sistem. Ako korisnički program pokuša da izvršiti neku operaciju koja je dozvoljena samo u sistemskom režimu rada, kontrola se predaje operativnom sistemu. Prilikom sistemskih poziva procesor prelazi iz korisničkog režima rada u sistemski, a vraća se u korisnički režim posle obrade poziva.

Sistemski pozivi obezbeđuju intefejs između programa koji se izvršava i operativnog sistema. Generalno, realizuju se u asembleskom jeziku, ali noviji viši programski jezici, poput jezika C i C++, takođe omogućavaju realizaciju sistemskog poziva. Program koji se izvršava može proslediti parametre operativnom sistemu na tri načina:

- prosleđivanjem parametara u registrima procesora,
- postavljanjem parametara u memorijskoj tabeli, pri čemu se adresa tabele prosleđuje u registru procesora i
- postavljanjem parametara na vrh steka (push), koje operativni sistem "skida" (pop).

Mikrokernel arhitektura

Mikrokernel predstavlja veoma savremen koncept u realizaciji savremenih operativnih sistema. Osnovna zamisao je napraviti minimalan, i pouzdan kernel visokih performansi, a sve ostale funkcije kernela potisnuti u takozvani korisnički prostor (*user space*). Korisnički moduli međusobno komuniciraju slanjem poruka (*message passing*).

U dobre osobine mikrokernel arhitekture spadaju:

- jednostavno proširivanje i optimizacija kernela,
- jednostavno dodavanje novih modula bez uticaja na bazični kernel,
- jednostavna prenosivost na drugu računarsku arhitekturu,
- veća pouzdanost (manje koda se izvršava u kernelskom režimu) i
- veća sigurnost.

Virtuelne mašine

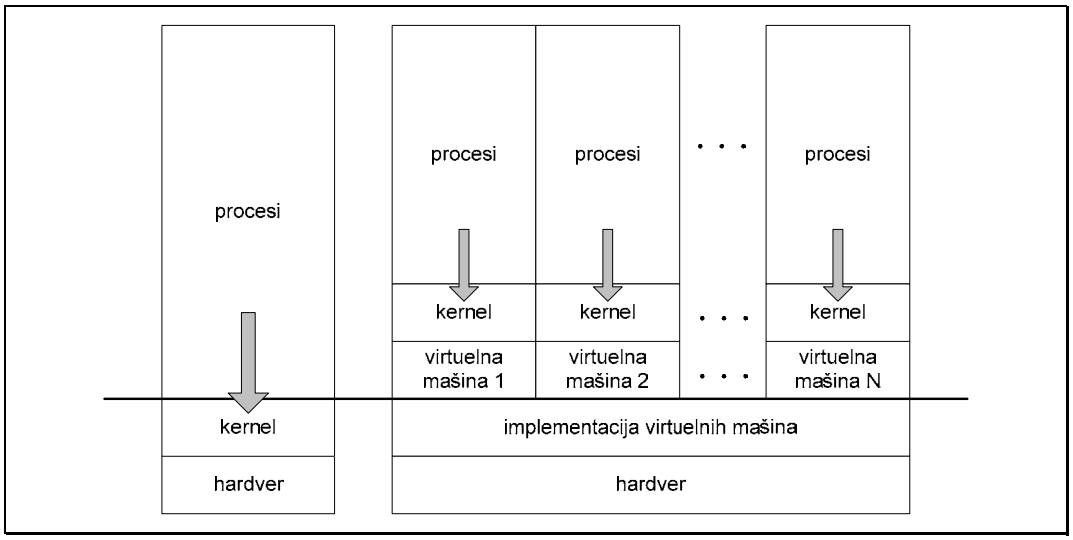
Iako smo operativni sistem definisali kao virtulenu mašinu, IBM je razvio posebnu softversku strukturu koja ima velike prednosti za realizaciju novih operativnih sistema i koja se takođe naziva virtulena mašina.

Strukturu virtuelnih mašina IBM definiše na sledeći način: na najnižem nivou se nalazi hardver, a iznad hardvera monitor virtuelnih mašina (*virtual machine monitor*), odnosno poseban sistem koji obezbeđuje niz virtuelnih mašina (tačnih kopija hardvera)**.** Zatim se na te virtuelne mašine mogu instalirati različiti operativni sistemi. Odgovarajući operativni sistemi primaju sistemske pozive korisničkih programa, a hardverske operacije koje šalju ti operativni sistemi prema svojim virtuelnim mašinama prihvata monitor virtuelnih mašina i realizuje ih u skladu sa hardverom ispod sebe.

Virtuelna mašina je bazirana na slojevitoj organizaciji i tretira realni hardver i realni kernel kao da su hardver za operativni sistem koji predstavlja. Virtulena mašina obezbeđuje identičan interfejs kao da je realni hardver ispod virtulene mašine, a ne čitav niz slojeva softvera.

Operativni sistem kreira iluziju o višestrukim procesima koji se izvršavaju na svom virtuelnom procesoru i svojoj virtuelnoj memoriji. Virtulene mašine obezbeđuju kompletnu zaštitu sistemskih resursa, pošto su sve virtulene mašine međusobno izolovane, i ne mogu direktno deliti resurse.

Koncept virtulene mašine prikazan je na slici 1.5.



Slika 1.5. *Virtuelne mašine*

Virteulne mašine su prefektan razvojni sistem za realizaciju novih operativnih sistema, jer se razvoj realizuje na virtulenoj mašini, a ne na realnoj, može lako da se kontroliše, mogu se ispravljati greške i nikako ne može doći do narušavanja normalnih sistemskih operacija, jer će sistem koji nudi virtulene mašine da radi bez obzira na stanje pojedinih virtulenih mašina.

Sistem virtulenih mašina se ne može lako realizovati zato što treba egzaktno simulirati hardver koga reprezentuje virtulena mašina.

2

Kernel i upravljanje procesima

Sadržaj poglavlja:

2.1. Kernel
2.2. Pojam procesa
2.3. Predstavljanje procesa
2.4. Raspoređivanje procesa
2.5. Operacije nad procesima
2.6. Rutine za interprocesnu komunikaciju
2.7. Vrste kernela

2.1. Kernel

Kernel, odnosno nukleus (*nucleus*) ili jezgro osnovni je deo svakog operativnog sistema. U hijerarhijskom (slojevitom) modelu, kernel je najbliži hardveru i najčešće se nadovezuje direktno na čist hardver. Kao takav, kernel predstavlja vezu, odnosno interfejs između hardvera i ostalih slojeva operativnih sistema.

Izuzetak predstavljaju Windows operativni sistemi u čijoj se osnovi nalazi NT arhitektura. U slojevitom modelu NT arhitekture ispod kernela se nalazi sloj abstrakcije hardvera (HAL - *Hardware Abstraction Layer*), koji omogućava operativnom sistemu da vidi različit hardver na isti način.

Kernel se, takođe, može definisati kao deo operativnog sistema koji omogućava programima da na siguran način pristupe hardveru. Kernel operativnog sistema nije neophodan za pokretanje i izvršavanje programa. Programi, odnosno procesi, mogu se

izvršavati na računaru bez kernela ukoliko je programer spreman da zaobiđe apstrakciju hardvera i usluge koje operativni sistem pruža i program napiše na niskom nivou, koristeći instrukcije mašinskog jezika. Ovaj način programiranja i izvršavanja programa predstavljao je normalnu pojavu na starijim računarskim sistemima. Nemogućnost prelaska iz jednog programa u drugi predstavljao je osnovni nedostatak ovakve realizacije - korisnik koji želi da prekine rad u jednom programu i pokrene drugi program morao je da resetuje računar. To se sprečavalo pomoću malih pomoćnih programa - *debuggera* i alata koji su punili memoriju programskim kodom. Ovi pomoćni programi, koji su omogućili prelazak iz jednog programa u drugi bez resetovanja računara, predstavljaju osnovu za dalji razvoj kernela operativnih sistema.

Osnovna funkcija kernela je upravljanje procesima, odnosno stvaranje okoline u kojoj mogu postojati procesi, dodeljivanje procesora procesima i obezbeđivanje mehanizama za interprocesnu komunikaciju. Kako se na jednom procesoru u jednom trenutku može izvršavati samo jedan proces (procesor je nedeljivi resurs), kernel određuje kada i na koje vreme će proces dobiti procesor. Ova pojava je poznata pod imenom multipleksiranje i predstavlja osnovu kvaziparalelnosti.

Da bi kernel ostvario svoju osnovnu funkciju, neophodno je da sam hardver ispuni određene preduslove, odnosno da na nivou hardvera postoje određene komponente koje omogućavaju nadogradnju hardvera kernelom. U te komponente spadaju:

[1] Mehanizam prekida

Mehanizam prekida obezbeđuje izvršenje kontrolnog programa (prekidne, tj. *interrupt* rutine), odnosno prebacivanje kontrole izvršavanja sa korisničkog na kontrolni program. Najmanje što mehanizam prekida treba da uradi jeste da sačuva vrednost programskog brojača prekinutog korisničkog programa i pokrene kontrolni program sa fiksne lokacije u memoriji. Kontrolni program dalje određuje izvor prekida i odgovara na određeni način.

[2] Zaštitni mehanizam adresiranja memorije

Zaštitni mehanizam adresiranja memorije sprečava pogrešno adresiranje, odnosno sprečava mogućnost da jedan proces upiše svoje prateće podatke u deo memorije koji je dodeljen drugom procesu. Ovaj mehanizam automatski čuva integritet procesa i podataka koji se nalaze u operativnoj memoriji.

[3] Privilegovani instrukcijski set

Privilegovani set instrukcija računara čine sve instrukcije koje su dostupne operativnom sistemu, ali ne i korisničkim programima. Ove instrukcije omogućavaju operativnom sistemu da maskira prekide, dodeli procesor drugom procesu, pristupi zaštićenim registrima u memoriji, izvrši ulazno-izlaznu operaciju ili zaustavi procesor. Za razliku od neprivilegovanih instrukcija koje se izvršavaju u korisničkom režimu, prilikom izvršavanja privilegovanih instrukcija operativni sistem se nalazi u kontrolnom režimu (*supervisory mode*). Korisnički

program ne može da izvrši privilegovanu instrukciju direktno, već isključivo pomoću sistemskih poziva. Korisnički program pomoću sistemskog poziva (*system call*) zahteva od operativnog sistema da izvrši privilegovanu instrukciju, nakon čega operativni sistem prelazi u kontrolni režim i izvršava tu instrukciju.

[4] *Real-time* časovnik

Pomoću *real-time* satnog mehanizma vrši se kontrola i evidencija potrošnje resursa računara za sve pojedinačne procese. Takođe, ovaj mehanizam se može koristiti za raspoređivanje i zakazivanje izvršenja raznih poslova.

Delovi kernela

Prema Listeru i Kvaterniku, jezgro *paper-model* operativnog sistema (teoretski model „na papiru“) može se podeliti na tri osnovne celine:

[1] Prvi nivo obrade prekida

Prvi nivo obrade prekida (*FLIH - First Level Interrupt Handler*) čine rutine za određivanje izvora prekida i iniciranje servisa, odnosno opsluživanje pojedinih vrsta prekida. FLIH odgovara na spoljašnje prekide i sistemske pozive. Posle izvršavanja koda koji čini ovaj deo operativnog sistema, prekid se smatra opsluženim, a dispečer dalje odlučuje kom će procesu dalje predati procesor na korišćenje.

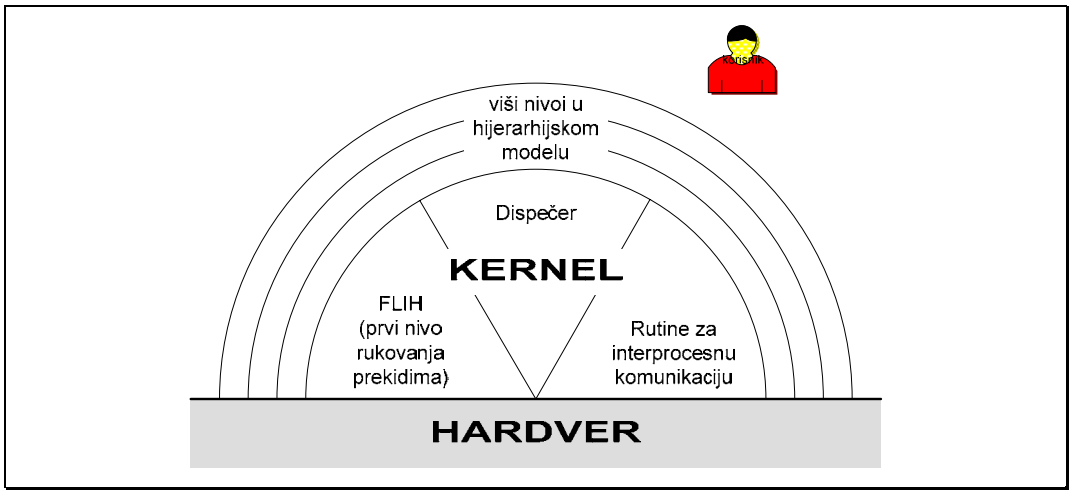
[2] Dispečer sistema (odnosno planer poslova niskog nivoa)

Dispečer (*dispatcher*, *low level scheduler*) je deo kernela koji dodeljuje procesor procesima. Pri tome, procesor se uvek dodeljuje na osnovu nekog algoritma, kao što je na primer *Shortest Job First* (prvo poslovi koji zahtevaju najmanje procesorskog vremena). Pre objašnjenja ovih algoritama opisan je način na koji se u operativnom sistemu predstavljaju procesi.

[3] Rutine za ostvarivanje interprocesne komunikacije.

Rutine za ostvarivanje interprocesne komunikacije su deo jezgra operativnog sistema koji obezbedjuje različite načine komunikacije među procesima. Postoji više načina ove komunikacije, kao što su slanje poruka (*send message*, *post message*), semaforske tehnike, korišćenje imenovanih cevi (*named pipes*, mehanizam karakterističan za UNIX sisteme) i korišćenje deljive memorije.

Na slici 2.1. prikazan je položaj kernela u hijerarhijskom modelu operativnog sistema.



Slika 2.1. *Kernel u hijerarhijskom modelu operativnog sistema*

2.2. Pojam procesa

Računarski sistem koji ima samo jedan procesor može da izvršava samo jednu instrukciju u jednom trenutku. Operativni sistem mora da obezbedi svim korisnicima računarskog sistema pristup procesoru, i ne sme dopustiti nikome da ostvari monopol nad njim. Da bi odradio taj zadatak, operativni sistem mora imati određene informacije o svakom korisniku i svakom programu u računaru, u bilo kom trenutku vremena.

Većina operativnih sistema upravlja procesima (*process*) i zadacima (*tasks*). Proces predstavlja jedan od najvažnijih koncepata operativnih sistema. Proces je program ili deo programa u stanju izvršavanja, zajedno sa svim resursima koji su potrebni za rad programa. Prosto rečeno, program je pasivan objekat, odnosno datoteka na disku. Kada se ta datoteka učita u memoriju, ona postaje proces, odnosno aktivan objekat koji ima svoje resurse, poput registara i memorije. To znači da da će tri korisnika, koji obavljaju neku aktivnost (na primer, izvršavaju korisničke aplikacije), biti predstavljena sa tri razna procesa. To takođe znači da će jedan program, koji je sam sebe razdelio na dva dela radi istovremenog (simultanog) izvođenja, biti predstavljen sa dva razna procesa. Sam operativni sistem je takođe sastavljen od niza procesa.

Svaki proces ima tri fundamentalne memorijska dela, odnosno sekcije:

- programska ili tekst sekcija koja se ne menja (*read only*) i koja sadrži programski kod,
- stek sekcija (*stack section*) koja sadrži privremene podatke (parametre za procedure, povratne adrese, lokalne promenljive),

- sekcija podataka (*data section*), koja sadrži globalne promenljive.

Kako se tekst sekcija ne menja, većina operativnih sistema formiraće zajednički tekst segment za sve procese koji nastaju posle pokretanja jednog programa. Ovaj model je efikasan i poželjan jer se na taj način štedi memorija. Naravno, svaki od tih procesa imaće posebnu stek sekciju i sekciju podataka.

Osim memorijskih sekcija, proces obuhvata i vrednost programskog brojača (*program counter*), kao i vrednost ostalih registara procesora koji su od interesa. Proces takođe obuhvata i ulazno-izlazne resurse koje eventualno koristi, kao što su datoteke i razne vrsta ulazno-izlaznih uređaja.

U otvorenoj literaturi, aktivnosti procesora se nazivaju različitim imenima: procesi (*processes*), poslovi (*jobs*) zadaci (tasks). U teoriji operativnih sistema, procesi i poslovi su relativno slični pojmovi. U slučaju UNIX operativnog sistema, posao se definiše kao grupa procesa koja radi u pozadini ili koja je suspendovana.

2.3. Predstavljanje procesa

Svaki program se sastoji od niza instrukcija i statičkog je karaktera. Za razliku od programa, proces je dinamička struktura i najjednostavnije se može opisati kao program u stanju izvršenja, odnosno kao niz aktivnosti koje nastaju kao posledica izvršenja programa. Deo procesa su i podaci koji tu aktivnost opisuju, odnosno podaci koji su neophodni za upravljanje procesom. Ove podatke generiše i koristi operativni sistem, odnosno dispečer, a u literaturi se pominju pod raznim imenima: kontrolni blok procesa (PCB - *Process Control Block*), vektor stanja ili *process descriptor*. Na primer, posmatrajmo višeprocesni operativni sistem sa dva aktivna procesa (P1 i P2). Proces P1 koji se izvršava, u jednom trenutku se dovodi u stanje čekanja na resurs. Nakon toga, nastavlja se izvršavanje procesa P2 koji posle nekog vremena takođe postaje blokiran. U međuvremenu, resurs neophodan za izvršenje procesa P1 se oslobađa i sada proces P1 može nastaviti sa izvršenjem. Da bi operativni sistem znao gde treba nastaviti izvršenje, svakom procesu se dodeljuju prateće informacije, odnosno jedinstveni kontrolni blok.

Kontrolni blok je deo operativne memorije, odnosno memorijska struktura koja sadrži osnovne informacije o procesu, koje operativni sistem koristi za upravljanje tim procesom. Zahvaljujući kontrolnom bloku, izvršenje programa se može prekidati i nastavljati više puta. U informacije iz kontrolnog bloka spadaju:

- ime ili jedinstveni identifikator procesa (PID),
- kontekst (okolina) procesa,
- prioritet procesa,
- informacije vezane za memoriju procesa,

- lista otvorenih datoteka,
- status zauzetih ulazno-izlaznih resursa i
- trenutno stanje procesa.

Uopšteno, kontekst procesa čine podaci koji se čuvaju prilikom oduzimanja procesora, a koje generiše sam hardver: programski brojač, vrednosti registara i pokazivači na deo memorije koji proces koristi. Deo kontrolnog bloka procesa u kome se čuva kontekst se još naziva i hardverski kontrolni blok procesa, odnosno *hardware process descriptor*. Šematski, kontrolni blok se prikazuje pridružen samom kodu, odnosno procesu, ali se u realnim operativnim sistemima nikad ne nalazi u istom memorijskom području.

Dijagram stanja procesa

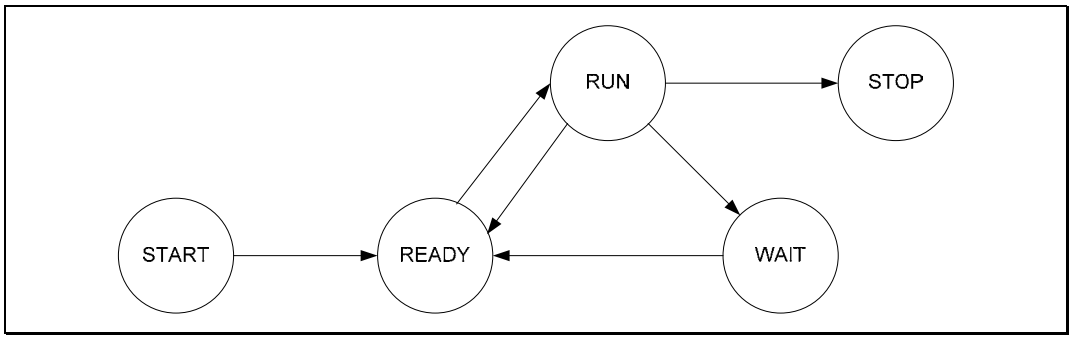
Proces se sastoji iz niza koraka koji slede jedan za drugim. Između dva koraka proces može biti prekinut, a njegovo izvršavanje se može nastaviti u nekom drugom trenutku vremena, na istom ili nekom drugom procesoru. Svi procesi koji uđu u računarski sistem prolaze kroz niz stanja tokom svog boravka u računaru. Jednostavno rečeno, stanje procesa (*process state*) opisuje ono što se u datom trenutku događa sa procesom. Provođenje procesa iz jednog stanja u drugo (*state transition*) obavlja operativni sistem.

Proces se može naći u nekoliko stanja (pet ili sedam, zavisno od konačnog automata koji je korišćen za opisivanje stanja) a sledeća tri su najznačajnija:

- stanje izvršavanja (RUN, RUNNING) - procesor izvršava istrukcije ovog procesa
- stanje čekanja na procesor (READY, RUNNABLE) - proces je dobio sve potrebne resurse osim pocesora, spreman je za rad i čeka da mu se dodeli procesor
- stanje čekanja na resurs (WAIT, UNRUNNABLE) - proces čeka na neki događaj (na primer, da se završi sa štampanjem), jer su za dalje izvršavanje procesa potrebni neki resursi koji trenutno nisu na raspolaganju. Takođe, proces može ući u stanje WAIT ukoliko čeka završtak neke ulazno-izlazne operacije rezultat nekog drugog procesa.

U ovom slučaju, radi jednostavnosti izlaganja, korišćen je konačni automat sa pet stanja. Dva dodatna pomoćna stanja su vezana za nastanak procesa, odnosno trenutak formiranja procesa (START) i prestanak, odnosno kraj izvršenja procesa (STOP). Proces u stanju STOP oslobađa sve resurse koje je zauzeo.

Proces se u svakom trenutku mora naći u jednom od ovih stanja, a tranzicije iz jednog u drugo stanje se vrše zavisno od događaja vezanih za proces, resurse koje proces koristi i ostatak računarskog sistema. Na slici 2.2. prikazan je dijagram stanja procesa, a nakon toga su opisane tranzicije, odnosno promene stanja.



Slika 2.2. *Dijagram stanja procesa*

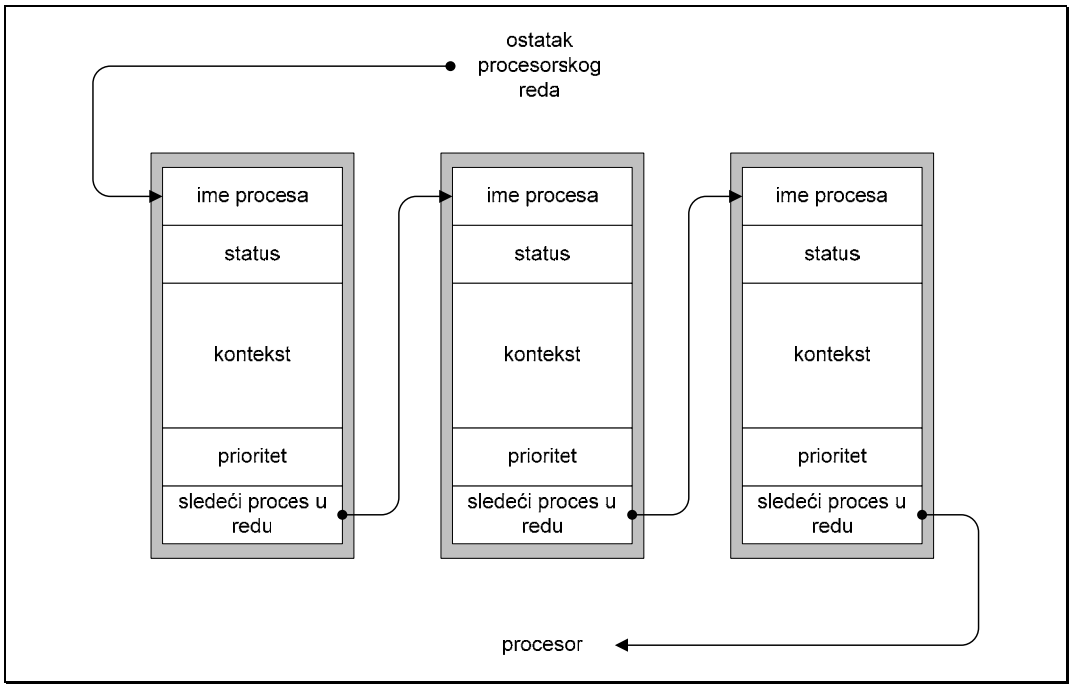
Prilikom upravljanja procesima, dispečer prati i ispituje stanja u kontrolnim blokovima procesa. Na primer, na osnovu toga, procesi se mogu dodati u red za čekanje na procesor. Dodavanje u red je simboličko - proces se fizički ne pomera sa jedne lokacije u drugu, već se samo menjaju podaci u kontrolnim blokovima i pokazivači pomoću kojih se formiraju redovi. Svaki proces počinje u stanju START, koje predstavlja neku vrstu pripreme, nakon čega se dovodi u stanje READY, odnosno u stanje čekanja na dodelu procesa u procesorskom redu. Ovo opisuje prvu tranziciju u dijagramu stanja. Posle nje, proces prolazi kroz dodatne tranzicije u konačnom automatu:

- READY-RUN: dodela procesora procesu koji je došao na početak reda za dodelu procesora. Proces prelazi iz stanja čekanja na procesor u stanje izvršavanja kada procesor prekine izvršenje tekućeg procesa, tako da može da otpočne izvršavanje novog procesa.
- RUN-READY: oduzimanje procesora procesu nakon isticanja perioda u kojem mu je procesor dodeljen. Dodatno, ukoliko je operativni sistem sa pretpražnjenjem (*pre-emptive*), procesor se može oduzeti i ukoliko naiđe proces višeg prioriteta. Ova tranzicija je moguća samo u višeprocesnim operativnim sistemima.
- RUN-WAIT: oduzimanje procesora procesu ukoliko je neki resurs koji je potreban za izvršenje procesa zauzet. Ova tranzicija je moguća i u jednoprocesnim i u višeprocesnim operativnim sistemima (na primer, print spooler ne može da nastavi štampanje dokumenata dok se u štampač ne doda papir). Takođe, ova tranzicija se može desiti u slučaju da proces čeka rezultat operacije (npr. izračunavanja) koje obavlja drugi proces što je tipično za odnos proizvođač-potrošač (engl. *producer-consumer*). Do ove tranzicije može doći i ako proces čeka neki odredjeni vremenski trenutak koji je unapred programiran i u kome se može nastaviti izvršavanje. Primer ove situacije je kad se npr. postave parametri sistema za arhivirenje (*backup*) ili antivirusnog alata da započnu arhiviranje ili skeniranje u odredjenom trenutku.

- WAIT-READY: proces se vraća na kraj procesorskog reda nakon oslobađanja resursa koji je neophodan za rad procesa. Napomena: tranzicija WAIT-RUN nije moguća u višeprocesnim operativnim sistemima. U takvim sistemima, proces se uvek prvo dovodi u stanje READY.
- RUN-STOP: proces završava sa radom (prirodno ili nasilno).

U stanju RUN se u jednom trenutku može naći najviše onoliko procesa koliko procesora ima na sistemu. Procesi koji se nalaze u stanju RUN su tekući procesi. U stanju RUN se proces ne može zadržati do svog završetka, zato što se u opštem slučaju posmatra višeprocesno okruženje u kome se procesor dodeljuje na određeno vreme, po nekom algoritmu. Na taj način se sprečava da jedan proces crpi resurse računara, dok drugi procesi čekaju u redu. Dodatno, ukoliko se operativni sistem karakteriše osobina pretpražnjenja, procesor se može oduzeti i u slučaju da naiđe proces višeg prioriteta, čime se obezbeđuje blagovremeno izvršavanje kritičnih procesa.

Svi procesi koji se nalaze u stanju READY mogu se povezati u red koristeći jednostavnu strukturu liste. Ova linearna lista, prikazana na slici 2.3. čini procesorski red u koji je uključen i tekući proces. Dispečer koristi procesorski red prilikom upravljanja procesima.



Slika 2.3. *Procesorski red*

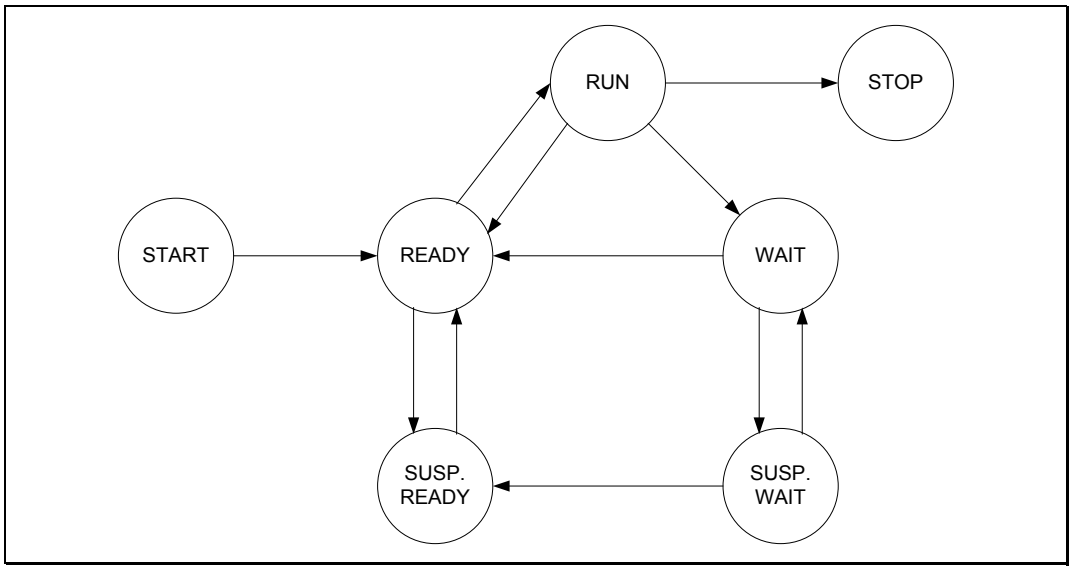
Prošireni dijagram stanja procesa

Na nekim operativnim sistemima uvodi se mogućnost privremenog prekida izvršenja procesa. Proces čije je izvršenje privremeno prekinuto je suspendovan proces, i kao takav, on prestaje da se takmiči za resurse.

Ovaj model automata karakterističan je za UNIX operativne sisteme. Osnovna pretpostavka je ta da korisnik koji inicira proces ima pravo da privremeno zaustavi njegovo izvršenje (*suspend*). Takođe, korisnik može u određenom momentu da nastavi izvršenje suspendovanog procesa (*revive*). Sam operativni sistem može, ukoliko postoji potreba, da određeni broj procesa dovede u suspendovano stanje i tako spreči pojavljivanje zastoja i spreči efekat zasićenja usled prevelike količine keširanih podataka. Takođe, suspendovanje procesa se može javiti kao posledica upotrebe virtuelne memorije, odnosno *swap* prostora. Izvršenje procesa se može prekinuti samo u stanjima WAIT i READY. Na taj način, formiraju se dva dodatna stanja:

- SUSPENDED-READY (u ovo stanje proces može doći ukoliko je suspendovan u stanju čekanja na procesor) i
- SUSPENDED-WAIT (u ovo stanje proces može doći ukoliko je suspendovan u stanju čekanja na resurs).

Na ovaj način se dobija novi konačni automat, sa sedam stanja i tri nove tranzicije, prikazan na slici 2.4:



Slika 2.4 *Prošireni dijagram stanja procesa*

U nastavku teksta ukratko su opisane nove tranzicije u proširenom dijagramu stanja:

- READY-SUSPENDED READY: dovođenje procesa iz stanja READY u suspendovano stanje. Proces prolazi kroz avu tranziciju ako: (1) na sistemu postoji previše procesa u stanju READY, (2) treba izbeći zastoj (*deadlock*) ili (3) ako korisnik eksplicitno suspenduje proces,
- WAIT-SUSPENDED WAIT: dovođenje procesa u suspendovano stanje iz stanja WAIT,
- SUSPENDED WAIT-SUSPENDED READY: resurs neophodan za dalje izvršenje procesa je slobodan, ali proces je i dalje suspendovan,
- SUSPENDED READY-READY: proces je odmrznut i dovodi se na kraj procesorskog reda. Ova tranzicija je moguća samo na eksplicitan zahtev korisnika,
- SUSPENDED WAIT-WAIT: proces je odmrznut, ali resurs neophodan za njegovo izvršenje nije oslobođen. Ova tranzicija je moguća samo na eksplicitan zahtev korisnika.

2.4. Raspoređivanje procesa

Kao što je već rečeno, multiprogramiranjem se vrši podela memorije na particije u koje se smeštaju različiti procesi. Multiprogramiranjem se se postiže bolje iskorišćenje procesora - ukoliko tekući proces čeka na resurs, procesor može izvršavati neki drugi proces koji se nalazi u stanju čekanja na procesor. Podela vremena (*time-sharing*) je poseban oblik multiprogramiranja koji omogućava da svaki korsinik radi interaktivno na računaru preko posebnog terminala kome pripada dodeljeno procesorsko vreme. Posle isteka vremenskog kvantuma, odnosno dodeljene količine procesorskog vremena, porcesor se dodeljuje drugom terminalu.

Na jednoprocesorskim računarskim sistemima se u jednom vremenskom trenutku može izvršavati samo jedan proces. Na višeprocesorskim sistemima se u jednom trenutku može izvršavati onoliko procesa koliko na sistemu ima procesora. Bez obzira na vrstu računarskog sistema (jednoprocesorski ili višeprocesorski sistem), ukoliko se na sistemu izvršava višeprocesni operativni sistem (kao što su Windows 2003 Server ili UNIX), veći broj procesa istovremeno čeka na procesor.

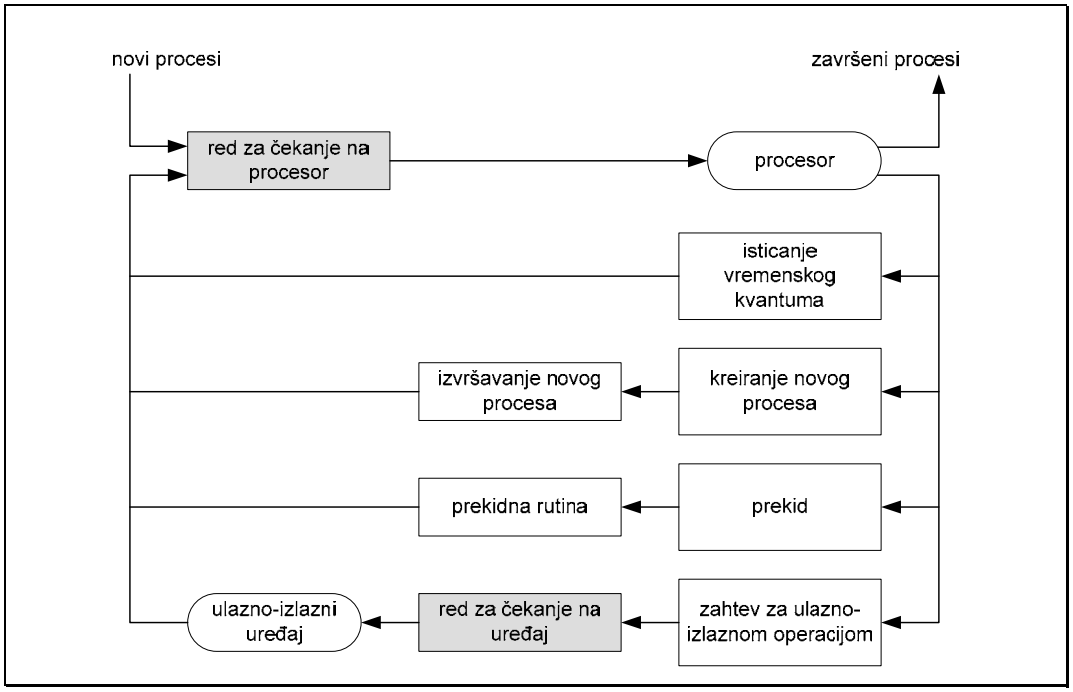
Operativni sistem treba da obezbedi odgovarajući mehanizam za dodelu procesora različitim procesima. Delovi tog mehanizma su redovi čekanja na procesor i planeri poslova.

Redovi čekanja na procesor

Proces u svom životu prolazi kroz nekoliko redova čekanja (*scheduling queues*). Nakon kreiranja, proces se ubacuje u takozvani red čekanja za poslove (*job queue*), koji obuhvata sve postojeće procese na sistemu. Procesi se mogu nalaziti u raznim stanjima i raznim lokacijama (u memoriji, na disku, delimično u memoriji, delimično na disku). Svi procesi koju su spremni za rad i nalaze se u operativnoj memoriji čuvaju se u redu čekanja na procesor, odnosno u redu čekanja spremnih procesa (*ready queue*). Redovi čekanja na procesor se po pravilu realizuju kao povezane liste, formirane od kontrolnih blokova procesa sa definisanim redosledom izvršavanja procesa. Redosled se specificira preko zaglavlja liste (*queue header*), koje sadrži informacije o početnom i poslednjem kontrolnom bloku u listi, i pokazivača na sledeći kontrolni blok.

Opertivni sistem takođe uvodi i poseban red čekanja za svaki ulazno-izlazni uređaj (*I/O queue*). Svaki red čekanja na uređaj sadrži povezanu listu kontrolnih blokova procesa koji taj uređaj zahtevaju.

Na slici 2.5 prikazana je tehnika za raspoređivanje procesa (*scheduling*). Prisutne su dve vrste redova čekanja (prikazane pravougaonicima): red čekanja na procesor i redovi čekanja na ulazno-izlazne uređaje. Svaki krug predstavlja resurs koji opslužuje red čekanja, a strelice pokazuju dijagram stanja.



Slika 2.5 *Tehnike raspoređivanja poslova*

Novi proces se inicijalno postavlja u red čeknja za spremne procese u kome čeka dodelu procesora, nakon čega napušta red i počinje da se izvršava. Proces koji se nalazi u stanju izvršavanja može:

- ostati bez procesora kada mu istekne vremenski kvantum,
- kreirati novi proces i čekati u blokiranom stanju da se novi proces izvrši,
- ostati bez procesora kada se dogodi prekid,
- postaviti I/O zahtev, nakon čega se prebacuje u red čekanja na ulazno-izlazni uređaj, odnosno postaje blokiran.

Proces se vraća u red čekanja na procesor sve dok se ne završi, posle čega oslobađa sve zauzete resurse.

Planer poslova i dispečer sistema

Kao što se vidi, proces u svom trajanju, prolazi kroz razna stanja i redove čekanja. Programi za raspoređivanje (*schedulers*) odlučuju o tome kada će proces ući u neki red čekanja ili napustiti taj red. U literaturi se ove komponente operativnog sistema pominju pod imenima:

- planer poslova (*job scheduler*, *long-term scheduler*, *high-level scheduler*) i
- dispečer (*dispatcher*, *short-term scheduler*, *low-level scheduler*).

Planer poslova, koji se u hijerarhijskom modelu nalazi iznad kernela, obavlja sledeće funkcije:

- deli poslove na procese,
- na osnovu određenih algoritama dodeljuje prioritete procesima i
- dovodi procese u red čekanja na procesor.

Planer poslova se, praktično, poziva samo kada se pojave novi procesi ili kada jedan ili više procesa završi aktivnosti. Planer poslova reguliše stepen multiprogramiranja, odnosno broj simulatnih procesa u memoriji. Od njega se očekuje da pravi dobru selekciju procesa koji će dobiti memoriju kako bi sistem što efikasnije funkcionisao. Planer poslova potiče sa *main-frame* sistema, koje karakteriše grupna obrada poslova.

Uspeh raspoređivanja procesora zavisi od osobina samih procesa. Izvršavanje jednog procesa se sastoji iz jednog ili više ciklusa izvršavanja na procesoru (*CPU burst*) i jednog ili više ciklusa čekanja na ulazno-izlazne operacije (*I/O burst*). Proces počinje ciklusom izvršavanja na procesoru, posle čega može više puta da menja ove cikluse ili stanja. Na osnovu resursa koje dominantno koriste, odnosno broja i trajanja ciklusa, procesi se mogu podeliti na:

- procese koje dominantno koriste procesor (*CPU bound*)

- procese koji dominantno koriste ulazno-izlazne operacije (*I/O bound*)

Najveći deo vremena izvršavanja I/O intenzivnih procesa otpada na ulazno-izlazne cikluse, dok se na upotrebu procesora i memorije troši relativno malo vremena. Sasvim je prirodno da će se ovi procesi relativno brzo blokirati čekajući na I/O operacije (ulazak u WAIT stanje). CPU intenzivni procesi generišu vrlo malo ulazno-izlaznih zahteva, a najveći deo svog vremena troše na upotrebu procesora. Planer poslova mora da napravi dobru mešavinu I/O i CPU internzivnih procesa. Ako većina procesa koju planer odabere intenzivno koristi ulazno-izlazne uređaje, procesor će biti slabo iskorišćen. Ako većina procesa koju planer odabere intenzivno koristi procesor, ulazno-izlazni uređaji će biti slabo iskorišćeni.

Zadatak dispečera sistema je da dodeljuje procesor procesima koji se nalaze u procesorskom redu. Dispečer dodeljuje procesor svaki put kad tekući proces pređe iz stanja RUN i stanje WAIT ili READY. Dispečer utvrđuje koji je proces najpovoljniji za dodelu procesora, odnosno koji je proces u stanju READY najvišeg prioriteta. Ukoliko se procesorski red formira sa inkrementalnim prioritetima (dinamički izmenljiva lista), najpovoljniji proces je prvi proces u redu. Ukoliko na sistemu postoji više procesorskih redova, odnosno po jedan red za svaki prioritet, uzima se prvi proces iz odgovarajućeg reda.

Najjednosntavnije rečeno, dispečer odlučuje koji će proces dobiti procesor, kada će dobiti procesor i koliko dugo će dobiti procesor. Dispečer najpre utvrđuje da li je potrebno procesor dodeliti tekućem procesu (čime bi se nastavilo izvršavanje istog) ili nekom drugom procesu iz reda. Ukoliko se procesor dodeljuje tekućem procesu, kontrola izvršavanja procesa se vraća na adresu koju čuva prekidni mehanizam kernela. Ukoliko se procesor dodeljuje nekom drugom procesu iz reda, dispečer najpre ažurira kontekst, odnosno okolinu tekućeg procesa u kontrolnom bloku, a zatim na osnovu konteksta novog procesa postavlja okruženje u kome se taj proces može izvršavati. Posle toga se kontrola izvršavanja prebacuje na ono mesto u novom procesu gde je izvršavanje bilo prekinuto.

U literaturi, planer poslova i dispečer se navode kao posebni programski moduli (Silbershatz) ili kao jedan modul (Lister, Kvaternik). U slučaju odvojenih modula, planer poslova niskog nivoa (*low level scheduler*, *short-term scheduler*) bira proces iz reda čekanja na procesor po nekom kriterijumu. Nakon toga, dispečer (u užem smislu reči) ažurira kontekst procesa, postavlja neophodno okruženje i predaje kontrolu nad procesorom izabranom procesu.

Učestalost pozivanja dispečera i planera poslova se, shodno njihovim funkcijama, znatno razlikuje. Zavisno od situacije, dispečer se poziva veoma često (na primer, na svakih 100 milisekundi) i od njega se očekuje veoma brz odziv, kako bi mogao adekvatno da odgovori na sukcesivne pozive. Vreme koje je potrebno dispečeru da zaustavi jedan proces i dodeli procesor drugom je poznato kao kašnjenje dispečera (*dispatch latency*).

Učestalost pozivanja planera poslova zavisi od srednjeg vremena između pojavljivanja novih procesa (u određenim situacijama može proći i po više minuta dok se ne pojave novi procesi u sistemu).

Na većini operativnih sistema planer poslova je minimalan. Na interaktivnim sistemima (*time-sharing*), kao što je UNIX, na kojima planer poslova ne postoji, svaki novi proces se direktno ubacuje u red čekanja na procesor. Stabilnost ovakvih sistema zavisi od toga da li su procesorska snaga i količina sistemske memorija dovoljni da zadovolje tekući broj interaktivnih korisnika.

Zamena konteksta procesa

Prilikom dodele procesora drugom procesu, odnosno zamene procesa koji se trenutno izvršava, dispečer:

- pamti stanje procesa koji se prekida, kao bi se proces mogao kasnije nastaviti i
- puni memoriju stanjem novog procesa kome se dodeljuje procesor.

Navedene operacije su poznatije pod nazivom zamena konteksta procesa (*context switch*). Kontekst procesa čine podaci koji se čuvaju prilikom oduzimanja procesora, a koji omogućuju nastavak izvršenja procesa, kao što su registri procesora, memorijske sekcije i lista otvorenih datoteka.

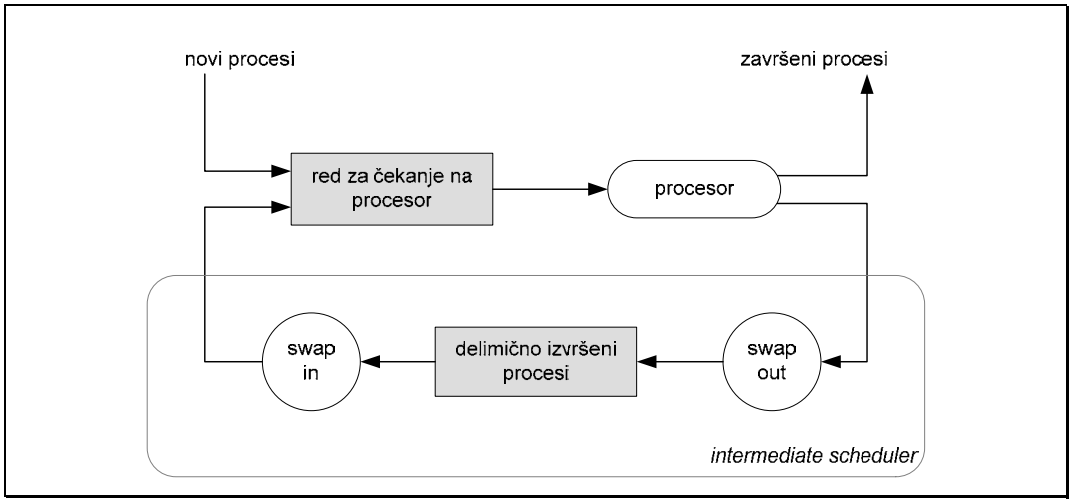
Prilikom zamene konteksta, poslednje stanje tekućeg procesa se čuva u njegovom kontrolnom bloku. Poslednje stanje procesa koji će se dalje izvršavati rekonstruiše se iz odgovarajućeg kontrolnog bloka, nakon čega se procesu predaje kontrola za nastavak izvršenja. Prebacivanje konteksta je čist gubitak vremena i predstavlja premašenje sistema (*overhead*), ali je isto tako neophodno radi omogućavanja multiprogramiranja. Premašenje zavisi od hardveskih performansi procesora i memorije, broja registara procesora koji se moraju sačuvati, optimalnih asembelskih instrukcija i načina korišćenja raznih memorijskih tehnika. Napomenimo da je ovo premašenje mnogo manje kod prebacivanja, odnosno zamene niti (*threads*).

Srednji nivo raspoređivanja procesa

Upotrebom *swap* tehnike, odnosno virtulene memorije, kod interaktivnih sistema se uvodi novi nivo raspređivanja procesa - srednji raspoređivač (*intermediate scheduler*). Blok dijagram srednjeg nivoa raspoređivanja prikazan je na slici 2.6.

Suština je u sledećem: svaki novokreirani proces odlazi u red čekanja, nakon čega dispečer odlučuje kome će da dodeli procesor. Neki procesi mogu biti suspendovani, odnosno privremeno prekinuti i upisani na disk (*swap space*), čime se oslobađa memorija za druge procese. Premeštanjem sa diska u memoriju, duspendovani procesi se vraćaju u red čekanja na procesor. Funkciju suspenzije procesa (*swap-out*),

funkciju povratka procesa u stanje spremnosti (*swap-in*) i izbor procesa za obe funkcije obavlja *intermediate scheduler*.



Slika 2.6. *Srednji nivo raspoređivanja poslova*

2.5. Operacije nad procesima

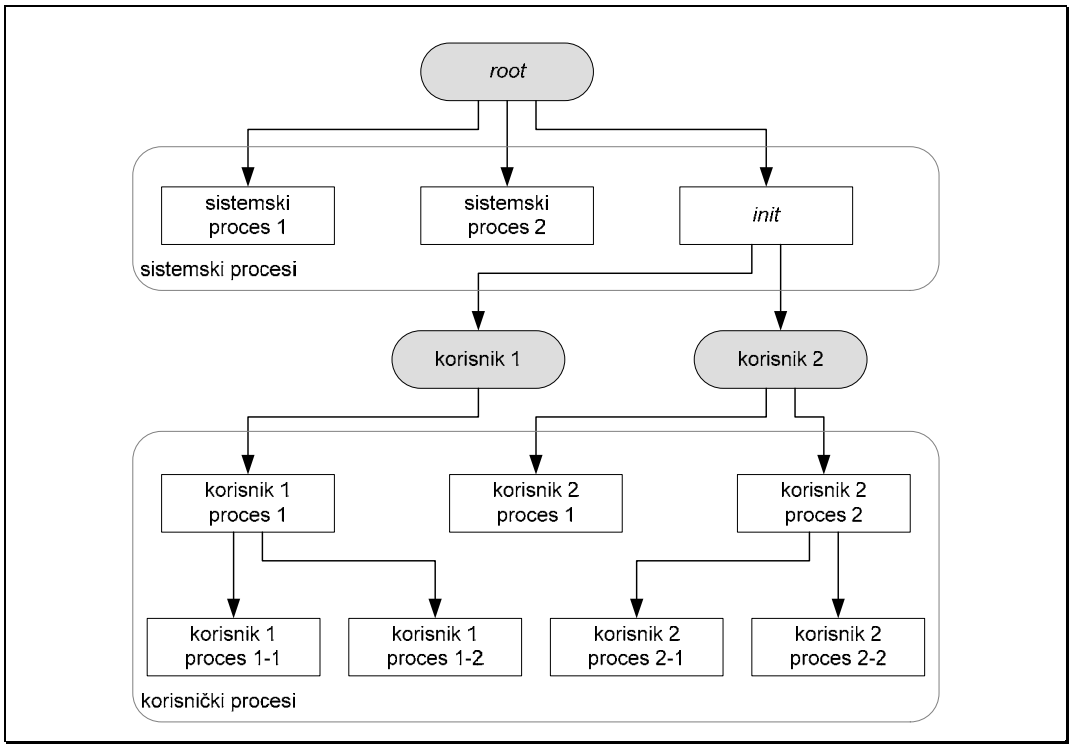
Ranije opisani koncept jezgra, uz korišćenje kontrolnih blokova procesa, dozvoljava operativnom sistemu da nad procesima izvrši sledeće operacije:

- kreiranje novog procesa (kreiranjem kontrolnog bloka i ažuriranjem procesne liste),
- kreiranje veza proces-proces roditelj,
- uništenje procesa (brisanjem kontrolnog bloka iz procesne liste),
- promena stanja procesa (obavljanje tranzicija u dijagramu stanja, kao što je dodela procesora),
- promena prioriteta procesa,
- sinhronizacija procesa i
- komunikacija između procesa.

U nastavku teksta opisane su operacije kreiranja procesa i formiranja stabla, uništenja procesa i konkurentnog izvršavanja procesa.

Kreiranje procesa

Proces u toku svog izvršenja može, odgovarajućim sistemskim pozivom, kreirati nove procese. Proces koji kreira nove procese je roditeljski proces (*parent process*), dok se proces koji je roditelj kreirao naziva dete proces (*child process*). Svaki proces koji je dete nekog drugog procesa može dalje kreirati nove procese - na taj način se formira hijerarhijska procesna struktura, odnosno stablo aktivnih procesa, koje je prikazano na slici 2.7.



Slika 2.7. *Stablo aktivnih procesa*

Odnos proces - proces roditelj se može opisati na osnovu načina deljenja resursa i načina izvršavanja.

Prema pitanju deljenja resursa, procesi roditelj i dete se mogu naći u sledećim relacijama:

- procesi roditelj i dete dele sve resurse
- procesi roditelj i dete dele podskup resursa proces roditelja
- procesi roditelj i dete ne dele resurse.

To znači da, nakon kreiranja, dete proces može tražiti od operativnog sistema nove resurse, deo resursa proces roditelja ili sve resurse koje pripadaju roditelju. Od resursa, svaki proces za svoj rad najčešće korsiti registre procesora, memorijske sekcije, datoteke i ulazno-izlazne uređaje. Posebno je osetljiv memorijski adresni prostor, za čije korišćenje se primenjuju sledeće tehnike:

- proces dete duplicira adresni prostor roditelja,
- adresni prostor deteta se generiše prema programu kojim se taj adresni prostor puni.

Prema načinu izvršavanja, proces se sa svojim roditeljem može naći u sledećim relacijama:

- proces roditelj nastavlja da se izvršava nezavisno i konkurentno sa detetom,
- proces roditelj se blokira i čeka dok proces dete ne završi svoje aktivnosti.

Osobine na relaciji poces - proces roditelj ilustrovaćemo na primeru UNIX operativnog sistema. Na UNIX sistemu svaki proces dobija univerzalni identifikator PID, a novi procesi se kreiraju pomoću sistemskog poziva fork. Proces dobija kopiju adresnog prostora roditelja, čime se omogućava lakša interprocesna komunikacija. Oba procesa (roditelj i dete) nastavaljaju izvršenje nakon fork sistemskog poziva, ali svaki od njih dobija različit PID: dete dobija PID=0, a roditelj PID različit od 0. Osim dodele internog PID broja, fork duplicira adresni prostor roditelja i dodeljuje ga detetu. Nakon fork sistemskog poziva adresni prostor oba procesa je napunjen istim programom (kodom), ali su identifikatori procesa različiti. Dete proces zatim izvršava sistemski poziv exec, koji puni program u adresni prostor dodeljen detetu i izvršava ga. Proces roditelj izvršava wait sistemski poziv čime se blokira dok proces dete proces ne završi svoje aktivnosti i ne vrati status roditeljskom procesu.

Sledeći C program ilustrovaće način funkcionisanja procesa pod UNIX sistemom, a primer odgovara slučaju komandnog interpretera */bin/sh* koji izvršava komandu *ls*. Isti kod izvršavaće i proces roditelj i proces dete. Roditelj kreira novi proces pomoću *fork* sistemskog poziva, koji proces roditelju i detetu dodeljuje različit PID. Proces dete čiji je PID 0, izvršava sistemski poziv *exec* koji će u dodeljni adresni prostor napuniti i izvšiti program *ls*. Proces roditelj čiji je PID različit od 0, čeka da se ls komanda završi (*wait* sistemski poziv). Tek kada proces dete završi aktivnosti, roditelj će izaći iz wait sistemkog poziva.

```
# include <stdio.h>
main (int argc, char *argv[]) {

   int pid;                          // fork
   pid = fork ();

   if (pid == 0)                     // dete proces
   {
      execlp ("/bin/ls", "ls", NULL);
```

```
    }

    else                              // proces roditelj
    {
       wait (NULL);
       /* proces dete završio svoje aktivnosti */
       exit (0);
    }
}
```

Završetak procesa

Proces završava svoje aktivnosti onda kada završi poslednju naredbu programa kojim je napunjen adresni prostor tog procesa. Posle toga, koristeći *exit* sistemski poziv, proces traži od operativnog sistema da ga uništi, odnosno obriše. Pri tome, proces dete može vratiti izlazne podatke proces roditelju pomoću *wait* sistemskog poziva. Potom, svi resursi koji su pripadali procesu, kao što su fizička i virtulena memorija, datoteke i ulazno-izlazni baferi, se oslobađaju. Ovo predstavlja takozvani normalan završetak rada po obavljanju svih predviđenih instrukcija.

Proces može izazvati i nasilan završetak rada drugog procesa pomoću specijalnog sistemskog pozova abort. Po pravilu, proces roditelj može prekunuti izvršenje procesa koje je on kreirao, a to čini iz sledećih razloga:

- proces dete je prilikom korišćenja nekog resursa (memorije, sistemskog vremena) prekršilo kvotu,
- aktivnost koju dete proces obavlja više nije potrebna i
- proces roditelj je završio aktivnosti pre dece, što se smatra nenormalnom situacijom koju operativni sistemi ne dozvoljavaju - u tom slučaju se sva deca procesi moraju nasilno uništiti (*cascade termination*).

Saradnja među procesima

U odnosu na međusobnu zavisnost toka izvršavanja, procesi se mogu podeliti na:

- nezavine procese i
- kooperativne procese.

Proces je nezavistan ukoliko nema direktnog uticaja na izvršenje drugih procesa i ukoliko na njegovo izvršenje direktno ne utiču drugi procesi. Praktično, nezavisni su oni procesi koji ne dele nikakve podatke sa drugim procesima.

Kooperativni procesi su oni procesi koji jedni na druge međusobno utiču, a to su, praktično, svi procesi koji dele podatke ili bilo kakve resurse. Kooperativni procesi zahtevaju specijalan tretman i okolinu iz sledećih razloga:

- deljenje informacija: kako više procesa mogu biti zainteresovani za iste informacije, na primer za istu datoteku, mora se obezbediti okolina koja će omogućiti pravilan konkurentni pristup takvim podacima,
- ubrzavanje rada: ukoliko bi se neki posao podelio na više različitih delova i ako bi se ti delovi izvršavali konkurentno, došlo bi do ozbiljnog povećanja performansi. Ovo ima smisla samo kod višeprocesroskih sistema ili višestrukih ulazno-izlaznih kanala (kao što je slučaj sa RAID sistemom),
- modularnost: sistemske funkcije mogu modularno da se podele na odvojene procese i niti,
- pogodnost: korisnik može da obavlja više stvari istovremeno.

Konkurentno izvršavanje kooperatvnih procesa zahteva mehanizam koji će dozvoliti procesima da međusobno komuniciraju između sebe i da sinhronizuju svoje akcije. Problem sinhronizacije se može objasniti na čuvenom problemu proizvođač-potrošač (*producer-consumer*) koji predslavlja zajedničku paradigmu kooperativnih procesa. Proivođač je proces koji proizvodi informacije, a potrošač je proces koji ih troši, odnosno koristi. Ta dva procesa mogu da rade konkurentno, ukoliko postoji bafer koji će puniti proizvođač, a prazniti potrošač. Pravila sinhronizacije su intuitivno jasna: potrošač ne može uzeti ništa iz bafera, ukoliko proizvočač to prethodno nije stavio u bafer. Bafer takođe nameće svoja pravila. U idealnom slučaju, bafer je beskonačnog kapaciteta (*unbounded buffer*), odnosno ima beskonačan broj mesta i nikada se ne može napuniti. Ovakav slučaj je idealan i u praksi ne postoji. Realni baferi su ograničeni kapacitetom od N elemenata, odnosno informacija (*bound buffer*). Ograničenjem kapaciteta uvodi se novi limit, odnosno novo pravilo sinhronizacije: ukoliko je bafer pun, proizvođač ne može ubaciti informaciju dok potrošač najpre ne uzme nešto iz bafera. Bafer se pod pod operativnim sistemom može realizovati na dva načina:

- interprocesnom komunikacijom i
- pomoću deljive memorije.

U nastavku teksta ilustrovano je rešenje sa deljivom memorijom. Proizvođač i potrošač dele sledeće promenljive:

```
# define N 10
typedef struct {
   /* promenljive koje čine jedan element bafera,
      odnosno jednu informaciju */
} item;

item buffer[N]          // definicija bafera veličine N
int in = 0;             // prvo slobodno mesto
```

```
int out = 0;            // prvo zauzeto mesto
```

Deljivi bafer je realizovan kao cirkularna struktura koja se sastoji od N elemenata [0, N-1], sa dva pokazivača *in* i *out*. Pokazivači *in* i *out* određuju prvo slobodno i prvo puno mesto u baferu. Bafer je prazan kada je *in*=*out*, a pun kada (*in*+1) mod N = *out*.

U nastavku teksta dat je odgovarajući kod za proizvođačke i potrošačke procese.

Za proizvođača:

```
item next_produced;
while (1) {
   /* proizvođač proizvodi informaciju */
   while (((in+1)%N) == out);
   /* sinhronizacija - bafer je pun, proizvođač
      čeka da potrošač nešto uzme iz bafera */
   buffer[in] = next_produced;
   in = (in+1) % N;
}
```

Za potrošača:

```
item next_consumed;
while (1) {
   while (in == out);
   /* sinhronizacija - bafer je prazan, potrošač
      čeka se da proizvođač nešto stavi u bafer /*
   next_consumed = buffer[out];
   out = (out+1) % N;
   /* potrošač konzumira informaciju */
}
```

Ovo rešenje je korektno samo u slučaju da se u baferu mogu naći najviše N-1 informacija. Za slučaj bafera u kome se mogu naći svih N informacija mora se obezbediti dodatna sinhronizacija, što je objašnjeno u u poglavlju o sinhronizaciji procesa.

2.6. Rutine za interprocesnu komunikaciju

Rutine za interprocesnu komunikaciju se moraju implementirati u kernelu kako bi bile dostupne svim procesima i kako bi imale direktan pristup dispečeru. Interprocesna komunikacija se može ostvariti na više načina, kao što su semaforske

tehnike, slanje poruka, korišćenje deljive memorije (na primer, bafera) i imenovanih cevi.

Slanje poruka

U prethodnom primeru smo izložili kako dva procesa, proizvođač i potrošač, koriste deljivu memoriju (bafer), a sinhronišu svoje aktivnosti korišćenjem deljivih promeljivih *in* i *out*. Procesi takođe mogu komunicirati i slanjem poruka, čime se omogućava sinhronizacija aktivnosti bez upotrebe deljive memorije. Ovaj način komunikacije je veoma podesan u distribuiranim okruženjima gde procesi egzistiraju na različitim računarima.

Da bi procesi mogli da komuniciraju slanjem poruka, neophodno je u kernel implementirati sledeće dve operacije (primitive):

- slanje poruke (*send message*, *post message*) i
- prijem poruke (*receive message*).

Poruke koje se razmenjuju između procesa mogu biti fiksne ili promenljive veličine. Slanje poruka fiksne veličine je metod koji je jednostavniji za implemnetaciju ali manji fleksibilan, dok je slanje poruka promenljive veličine teže za implementaciju ali fleksibilnije.

Postoje različite šeme za realizaciju ovih primitiva, a sledeće dve su dominatne:

- sinhrono ili blokirajuće (*blocking*) slanje i primanje poruka i
- asinhrono ili neblokirajuće (*nonblocking*) slanje i primanje poruka.

Prilikom blokirajućeg slanja poruka (*blocking send*), proces koji šalje poruku se blokira dok drugi proces ili sanduče ne prime poruku. Ukoliko je slanje neblokirajuće (*non-blocking send*), proces koji šalje poruku nastavlja dalje aktivnosti ne čekajući potvrdu o prijemu. Prilikom blokirajućeg primanja poruka (*blocking receive*), proces koji prima poruku se blokira dok ne dobije poruku. Ukoliko je primanje neblokirajuće (*non-blocking receive*), proces će pokušati da primi poruku: ako je poruka stigla primiće je, ako nije stigla, rezultat je prazan niz (*null*). Proces nastavlja dalji rad bez blokade.

Moguće su različite kombinacije ovih metoda. Na primer, šema u kojoj su i slanje i primanje blokirajući se zove *rendezvous*.

Da bi dva procesa mogla da razmenjuju poruke, između njih mora postojati komunikacioni link, koji se može implementirati na različite načine (pomoću deljive memorije, magistrale, mrežne kartice i odgovarajućih protokola). U nastavku teksta ukratko su opisane logičke karakteristike komunikacionog linka ukoliko je on implementiran kao:

- direktna komunikacija ili

- indirektna komunikacija preko poštanskih sandučića.

Bilo da ju pitanju direktna ili indirektna komunikacija, poruke koje se razmenjuju, čuvaju se u privremenim redovima čekanja. Taj red čekanja je, praktično, memorijski bafer i može se realizovati kao bafer nultog, ograničenog ili neograničenog kapaciteta.

Veličina bafera nultog kapaciteta (*zero capacity*) za poruke je 0, što znači da slanje i primanje moraju biti blokirajući, odnosno da se radi o *rendezvous* šemi. Ukoliko je bafer ograničenog kapaciteta, slanje je neblokirajuće sve dok bafer nije pun. Kada se bafer napuni, pošiljaoc mora da sačeka da se mesto u baferu isprazni. Bafer neograničenog kapaciteta je idealan slučaj u kome primalac može primiti beskonačan broj poruka. U ovom slučaju, pošiljaoc se nikada ne blokira.

Direktna komunikacija

Procesi koji žele da komuniciraju moraju da imaju mogućnost imenovanja kako bi mogli međusobno jednoznačno da se identifikuju. U direktnoj šemi, procedure za slanje i primanje moraju eksplicitno da sadrže ime procesa sa kim žele konuniciraju. Takođe postoji i takozvana asimetrična varijanta adresiranja, u kojoj pošiljaoc mora da navede ime primaoca, dok primalac ne mora da navede ime pošiljaoca. U nastavku teksta su redom date moguće implementacije operacija za slanje poruke *message* procesu P, primanje poruke od procesa Q i primanje poruke od bilo kog procesa.

```
send (P, message)
receive (Q, message)     // simetrično adresiranje
receive (id, message)    // asimetrično adresiranje
```

Pravila direktne komunikacije su sledeća:

- link se uspostavlja automatski između para procesa koji žele da komuniciraju i
- jedan link se uspostavlja samo za dva procesa.

Nedostatak direktne komunikacije je limitirana modularnost definicija procesa, pri čemu promena imena procesa povlači ispitivanje svih procesnih definicija. Sve reference sa starim imenom moraju se pronaći i modifikovati u novo ime, a te situacije nisu uvek poželjne.

Indirektna komunikacija

Kod indirektne komunikacije poruke se šalju ili primaju preko poštanskih sandučića (*mailbox*) ili portova. Poštansko sanduče se može posmatrati kao objekat u kome operativni sistem ostavlja poruke drugih procesa. Svako sanduče ima jedinstvenu identifikaciju. Dva procesa mogu komunicirati samo ako dele sanduče. U indirektnoj

šemi, proces može komunicirati sa procesom preko više različitih sadučića. U ovoj šemi naredbe za slanje poruka u poštansko sanduče A i primanje poruka iz poštanskog sandučeta A su:

```
send (A, message)
receive (A, message)
```

U indirektnoj šemi osobine komunikacije su sledeće:

- link se uspostavlja između para procesa koji dele sanduče,
- linku se mogu pridružiti više od dva procesa i
- više različitih linkova može postojati između svakog para procesa, gde svaki link odgovara jednom sandučetu.

Analiziraćemo sledeću situaciju. Pretpostavimo da imamo tri procesa (P1, P2 i P3) koji dele isto sanduče A. Proces P1 šalje poruku u sanduče A, a procesi P2 i P3 proveravaju da li je neka poruka smeštena u sanduče. Postavlja se sledeće pitanje: koji će proces preuzeti upućenu poruku? Ovaj problem se može rešiti na više načina:

- dozvolom uspostavljanja linka preko poštanskog sandučeta isključivo između dva procesa,
- zabranom istovremenog primanja poruka od većeg broja procesa (samo jedan proces može obaviti prijem u jednom trenutku vremena) i
- adresiranjem poruke (navođenjem imena procesa primaoca).

Sandučići mogu biti vlasništvo procesa ili operativnog sistema. Da bi se omogućila indirektna komunikacija slanjem poruka, operativni sistem mora se obezbediti mehanizme za kreiranje novog sandučeta za svaki par procesa koji želi da komunicira, slanje i primanje poruka kroz sanduče i brisanje sandučeta.

Semafori

Semafor je celobrojna nenegativna promenljiva, čija vrednost štiti neki resurs i omogućava komunikaciju između procesa. Vrednost semafora određuje da li je resurs kog taj semafor štiti slobodan ili zauzet. Svaki semafor ima svoju početnu vrednost, a nad njim se mogu izvršiti dve nedeljive operacije, koje su radi jednostavnosti predstavljene kao *signal(s)* i *wait(s)*. Operacije *signal* i *wait* se ne mogu podeliti na više ciklusa, tako da dva procesa ne mogu istovremeno izvršavati ove operacije nad istim semaforom.

Resursi se štite semaforima na sledeći način: operacija *wait(s)* dekrementira vrednost semafora s samo ako je s>0. U slučaju da je s=0, proces koji izvršava ovu operaciju, ulazi u stanje čekanja na resurs (stanje WAIT, tranzicija READY-WAIT). Kada vrednost semafora s postane pozitivna, odnosno kada drugi proces oslobodi resurs

operacijom signal, operacija *wait* će dekrementirati vrednost semafora (s=s-1) i uvesti proces u kritičnu sekciju. Kritična sekcija (*critical section*) je deo koda u kome proces koristi resurs. Kada proces želi da prekine korišćenje resursa, on izvršava operaciju *signal(s)* i napušta kritičnu sekciju. Operacija *signal(s)* inkrementira vrednost semafora s kojim je resurs zaštićen (s=s+1) i kao takva oslobađa proces koji na taj resurs čeka iz stanja WAIT (tranzicija WAIT-READY).

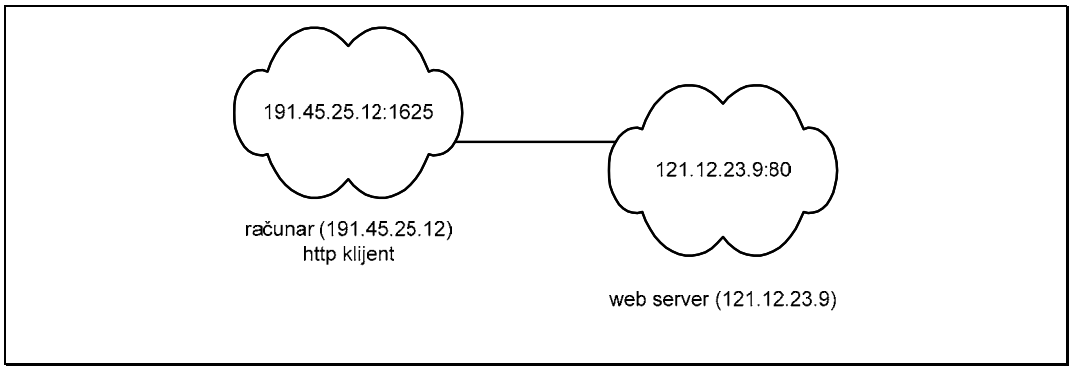
Operacije *wait(s)* i *signal(s)* respektivno uvode i oslobađaju resurse iz stanja čekanja na resurs i kao takve moraju poslati informaciju dispečeru o promeni u procesorskom redu. Iz tih razloga je neophodno da se ove operacije implementiraju u jezgru operativnog sistema kako bi imale direktan pristup dispečeru. U slučajevima izvršavanja operacije *signal(s)* nad semaforima na kojima na čeka ni jedan proces, kao i u slučajevima u kojima je vrednost semafora posle izvršene operacije *wait(s)* pozitivna, dispečer jednostavno nastavlja izvršenje tekućeg procesa, jer u procesorskom redu nije došlo do promene.

Vrednost semafora s u svakom trenutku može se odrediti formulom val(s)=c(s)+ns(s)-nw(s), gde je c(s) početna vrednost, ns(s) broj *signal* operacija i nw(s) broj *wait* operacija. Vrednosti semafora kojim je zaštićen resurs koji istovremeno može koristiti N procesa nalaze se na intervalu [0,N-1]. Za nedeljive resurse, odnosno resurse koje može koristiti samo jedan proces u jednom trenutku vremena je N=1. U nedeljive resurse spada većina perifernih uređaja, kao i datoteke otvorene u režimu za čitanje i pisanje. Za deljive resurse, odnosno resurse koje može koristiti veći broj procesa istovremeno je N>1. Na primer, ova vrednost može opisivati broj konekcija ostvarenih ka deljenom direktorijumu na nekom mrežnom serveru.

Interprocesna komunikacija u klijent-server sistemima

Korišćenjem različitih mehanizama, komunikacija među procesima se proširuje i na mrežne i distribuirane operativne sisteme. Procesi koji se izvršavaju na različitim računarima mogu da razmenjuju poruke preko mreže.

Jedan od najčešće korišćenih mehamizama jeste mehanizam priključaka (*socket*), patent proizvođača BSD UNIX sistema, koji je prihvaćen na svim operativnim sistemima sa mrežnom podrškom. *Socket* se definiše kao krajnja tačka komunikaije. Par procesa koji želi da komunicira preko mreže formira par priključaka, po jedan za svaki proces, na svakoj strani mreže. *Socket* se definiše pomoću dva parametra - IP:port, odnosno pomoću IP adrese računara na kome se priključak formira i odgovarajućeg broja porta. Generalno, *socket* koristi klijent-server arhitekturu, zato što server osluškuje portove analizirajući da li je stigao neki zathev od klijenata na tom portu. Serveri pružaju odgovarajuće servise (telnet, ftp, http) koji osluškuju i prenose podatke na tačno definisanim portovima. Na primer, telnet servisu odgovara port 23, ftp je na portu 21 itd. Znači, za svaki servis i za svaki par procesa koji žele da komuniciraju formira se par priključaka, preko kojih se razmenjuju poruke. Na slici 2.8. prikazan je ovaj način komunikacije:



Slika 2.8. *Mehanizam priključaka (*socket*)*

Računar želi da uspostavi *web* sesiju sa serverom. Klijent proces na strani računara kreira svoj *socket* na portu 1625 i preko tog priključka komunicira sa tačno definisanim *web* priključkom (port 80) na strani *web* servera.

U klijent-server okruženju uvodi se apstrakcija klasičnog poziva procedure pozivom udaljene procedure (RPC, *Remote Procedure Call*). RPC omogućava procesu koji se izvršava na jednom računaru da pozove proceduru na drugom računaru. RPC funkcioniše na sledeći način: najpre se na serverskoj i na klijentskoj strani definišu strukture poznate pod imenom stub između kojih se preciznim slanjem poruka i parametara (*marshaling parameters*) ostvaruje komunikacija. Najpre klijentski proces pošalje zahtev svom stubu, koji priprema parametre i preko mreže ih šalje podatke serverskom stubu. Serverski stub realizuje izvršenje procedure u lokalu, a rezultate priprema i šalje stubu na klijentskoj strani, koji će ih na kraju vratiti procesu koji je inicirao RPC.

2.7. Vrste kernela

Jezgra operativnih sistema se mogu klasifikovati u četiri kategorije:

[1] Monolitna jezgra

Monolitni kernel obezbeđuje potpuni set hardverskih apstrakcija, odnosno virtuelni interfejs visokog nivoa. Hardver se apstrahuje pomoću sistemskih poziva koji se izvršavaju u kontrolnom režimu, odnosno primitiva kojima se implementiraju servisi operativnog sistema, poput upravljanja procesima i upravljanja memorijom. Iako su sistemski pozivi realizovani u obliku posebnih modula, sam kernelski kod je usko integrisan - moduli se izvršavaju u istom adresnom prostoru tako da greška u jednom od njih može prouzrokovati krah sistema. S druge strane, ukoliko je implementacija korektna, monolitni kernel može biti vrlo efikasan. Većina poznatih monolitnih kernela (Linux, FreeBSD)

dozvoljava da se u toku izvršavanja kernel nadograđuje modulima, čime se povećavaju same mogućnosti kernela, dok se količina koda u memorijskom prostoru kernela svodi na minimum. Tradicionalni UNIX (BSD) i Linux kernel su primeri monolitnih kernela.

[2] Mikrokernel

Mikrokerneli obezbeđuju minimalni set hardverskih apstrakcija, odnosno krajnje jednostavnu virtuelnu mašinu. Sistemski pozivi mikrokernela omogućavaju pristup osnovnim funkcijama operativnog sistema kao što je upravljanje procesima i nitima, dok se veća funkcionalnost ostvaruje pomoću specijalnih aplikacija (servera), čiji kod nije integrisan sa samim kodom kernela. Znači, osnovni cilj koji mikrokernel treba da postigne jeste razdvajanje implementacija servisa operativnog sistema od osnovnih funkcija. Zbog nezavisnosti samog koda, otkaz specijalne aplikacije (server) neće prouzrokovati krah celog sistema - aplikacija se može pokrenuti ponovo, nezavisno od ostatka sistema. Mikrokernel arhitektura je prisutna u sledećim operativnim sistemima: AIX, GNU Hurd, Minix, MorphOS, QNX, RadiOS, BeOS.

[3] Hibridna jezgra (modifikovana mikrokernel arhitektura)

Hibridna jezgra su veoma slična mikrokernelima i mogu se shvatititi kao mikrokerneli koji sadrže delove koda koji nisu neophodni. Smeštanjem u sam kernel, obezbeđuje se brži rad dodatnog koda (kod se brže izvršava u kontrolnom režimu nego u korisničkom režimu). U operativne sisteme koji koriste hibridni kernel spadaju i Microsoft Windows NT/2000/XP/2003 Server, Mac OS X, DragonFly BSD i ReactOS.

[4] Egzokernel

Egzokernel ne obezbeđuje nikakvu apstrakciju hardvera. Umesto toga, obezbeđuju se specijalne biblioteke putem kojih korisnički procesei mogu ostvariti direktan ili skoro direktan pristup hardveru. Motivacija ovakvog pristupa leži u pretpostavci da sam programer može najbolje da odredi kako program treba da koristi hardver. Na primer, programer može da zahteva da se procesu dodele tačno određene memorijske lokacije ili blokovi diska na korišćenje. Pri tome egzokernel samo proverava da li je određeni resurs slobodan ili ne i ne vrši nikakvu apstakciju. Egzokerneli su jako mali jer je njihova funkcija ograničena na multipleksiranje resursa (na primer, dodelu procesora) i osnovni vid zaštite. Operativni sistemi koji koriste egzokernel nazivaju se i vertikalno stuktuiranim operativnim sistemima. Trenutno, razvoj egzokernela je još uvek u fazi istraživanja, tako da se egzokerneli ne upotrebljavaju u komercijalno dostupnim operativnim sistemima.

3

Raspoređivanje procesa i dodela procesora

Sadržaj poglavlja:

3.1. Uvodne napomene
3.2. Algoritmi za raspoređivanje procesa
3.3. Raspoređivanje u više procesorskih redova
3.4. Raspoređivanje u višeprocesorskoj okolini i realnom vremenu

3.1. Uvodne napomene

Osnova tehnike multiprogramiranja je naizmenična dodela procesora većem broju aktivnih procesa. Na taj način se obezbeđuje maksimalno iskorišćenje procesora, jer u svakom trenutku postoji proces koji će se izvšavati (u krajnjem slučaju to može biti i proces koji ne radi ništa - *idle* proces). U ovom poglavlju uvešćemo osnovne koncepte i predstaviti nekoliko algoritama za dodelu procesora.

Na jednoprocesorskim sistemima u jednom trenutku se izvršava samo jedan proces, dok svi drugi procesi moraju da sačekaju da procesor postane slobodan i da dođu na red (kvaziparalelno izvršavanje procesa). Na višeprocesorskoj arhitekturi se veći broj procesa može izvršavati istovremeno, odnosno paralelno (po jedan na svakom procesoru).

Ideja multiprogramiranja je relativno jasna i jednostavna. Svaki proces ostaje u stanju izvršenja dok mu ne istekne vremenski kvantum ili dok ne dođe u situaciju da

mora da čeka na neki događaj, kao što je, na primer, završetak ulazno-izlazne komande. Na prostim sistemima procesor ne radi ništa dok proces čeka. Na kvalitetnijim sistemima, memorija se puni većim brojem procesa, tako da kad aktivni proces mora da čeka, operativni sistem mu oduzima procesor i dodeljuje ga nekom drugom procesu.

Dodela procesora po nekom algoritmu je jedna od fundamentalnih funkcija operativnog sistema. Sličan princip se primenjuje i za druge resurse, kao što su ulazno-izlazni uređaji: veći broj zahteva se smešta u red za čekanje, a zatim se raspoređuju po nekom kriterijumu.

Raspoređivanje sa pretpražnjenjem

Procesor se dodeljuje drugom procesu pod sledećim okolnostima:

[1] kada proces pređe u stanje čekanja na resurs (čeka na završetak I/O operacije koju je inicirao),

[2] kada proces roditelj čeka proces dete da završi aktivnosti,

[3] prilikom tranzicije RUN-STOP, odnosno kada tekući proces završi aktivnosti.

[4] prilikom tranzicije RUN-READY,

[5] prilikom tranzicije WAIT-READY (u slučaju da je proces koji je napustio stanje WAIT i prešao u stanje ready višeg prioriteta od procesa koji se trenutno izvršava).

Jasno je da pod okolnostima [1], [2] i [3] novi proces mora biti izabran iz reda čekanja na procesor i aktiviran. Raspoređivanje koje se dešava pod ovim okolnostima naziva se rapoređivanje bez prekidanja izvršenja procesa, odnosno raspoređivanje bez pretpražnjenja (*nonpreemptive scheduling*). U slučaju raspoređivanja bez pretražnjenja, procesor se može oduzeti samo od procesa koji je završio svoje aktivnosti ili čeka na neki resurs.

U slučaju raspoređivanja sa pretpražnjenjem (*preemptive scheduling*), procesor se može oduzeti procesu koji nije završio svoje aktivnosti i nije blokiran. Jedna od fundamentalnih hardverskih komponenti neophodna za pretpražnjenje jeste tajmer, koji peroidično postavlja prekidni signal (prekidanje procesa se u literaturi pominje i kao pražnjenje). Prednosti ovakvog raspoređivanja su očigledne - na primer, procesor se može jako brzo dodeliti procesu višeg prioriteta koji zahteva izvršenje. Pretpražnjenje ima uticaj na konstrukciju kernela operativnog sistema. Po pravilu, da ne bi nastao haos, proces koji je pomoću sistemskog poziva prešao u kernelski režim se ne može prekidati sve dok je u tom režimu rada. Ovo je karakteristično za UNIX, ali i za većinu drugih operativnih sistema. Problemi koji se mogu javiti kod raspoređivanja sa pretpražnjenjem vezani su za moguću nekonzistentnost zajedničkih podataka (proces koji je otpočeo ažuriranje podataka je prekinut, a kontrola dodeljena drugom procesu koji koristi iste podatke).

Kriterijumi dodele procesora

Različiti algoritmi za raspoređivanje procesa imaju različite osobine i mogu favorizovati određenje klase procesa u odnosu na druge. Da bi odlučilo koji će se algoritam izabrati, moraju se znati osobine algoritama i karakteristike procesa. Većina algoritama za raspoređivanje procesa je optimizovana po pitanju nekog od sledećih kriterijuma:

[1] Iskorišćenje procesora (*CPU utilization*).

Po ovom kriterijumu procesor treba da bude zauzet, odnosno da radi nešto bez prestanka. Iskorišćenje procesora varirati od 0 do 100%. Na realnim sistemima, iskorišćenje ide od 40% na umereno opterećim sistemima, do 90% na veoma opterećenim sistemima.

[2] Propusna moć sistema (*throughput*).

Definiše se kao broj procesa izvršen u jedinici vremena. Za dugačke procese ovaj odnos može biti 1 proces na sat, za kratke transakcione procese može biti na primer 10 procesa u sekundi. U svakom slučaju, propusna moć zavisi od vrste procesa i brojnih hardverskih i sistemskih osobinama.

[3] Vreme potrebno za kompletiranje procesa (*turnaround time*).

To je ukupna količina vremena potrebna da se izvrši pojedinačni proces. Ovo vreme se računa od trenutka kreiranja do završetka procesa, a uključuje i vreme potrebno da proces uđe u red čekanja, vreme provedeno u redu čekanja, vreme korišćenja procesora i izvršenja ulazno-izlaznih operacija.

[4] Vreme čekanja (*waiting time*).

To je ukupno vreme koje proces provede u redu čekanja na procesor (*ready queue*). Potrebno je naglasiti da proces do kraja svog izvršavanja može više puta da bude u ovom redu čekanja, kao i da na ovo vreme direktno utiču algoritmi za rasproređivanje.

[5] Vreme odziva (*response time*).

To je vreme potrebno da se nakon slanja zahteva pojave prvi rezultati izvršenja procesa. Ovo vreme je veoma bitno kod interaktivnih sistema - proces koji otpočne sa radom treba relativno brzo da prikaže neke početne rezultate na ekranu, zatim da ponovo izvrši neka izračuvanja ili ulazno-izlaznu operacije, a onda opet da prikaže rezultate, i tako redom.

Zavisno od tipa, odnosno namene sistema, algoritmi se optimizuju za određene kriterijume, pri čemu se nekada forsiraju minimalne ili maksimalne vrednosti, a nekada srednje vrednosti. Na primer, analize pokazuju da je za interaktivne sisteme značajnije minimizirati maksimalno vreme odziva a ne srednje vreme, itd.

Ispitivanje algoritama

Postavlja se pitanje kako odabrati algoritam za raspoređivanje procesa u nekom posebnom sistemu. Prvi problem je, svakako, postavititi glavni kriterijum ili kombinaciju kriterijuma, kao što su, na primer:

- maksimizovanje iskorišćenosti procesora uz ograničenje da maksimalno vreme odziva bude 1 sekund i
- maksimizovanje propusne moći, tako da srednje vreme izvršavanja bude lienarno proporcionalno ukupnom vremenu izvršavanja.

Kada se defniše kriterijum, obavlja se analiza različitih algoritama, po pravilu putem simulacije. Za simulaciju je potrebno radno optrećenje (*workload*), koje može biti sintetičko ili realno. Opterećenje se formira snimanjem svih procesa jednog realnog sistema u određenom periodu vremena u statističku datoteku (*trace file*). Za simulator je bitna simualacija reda čekanja koji može biti izveden u jednom ili u više nivoa. Svaki red ima svoj kapacitet, vremensku raspodelu nailaska procesa, i algoritam koji se primenjuje za raspoređivanje procesa. Kada simulator obavi svoje, dobijaju se inforamcije o uspešnosti jednog algoritma na datom statističkom uzorku.

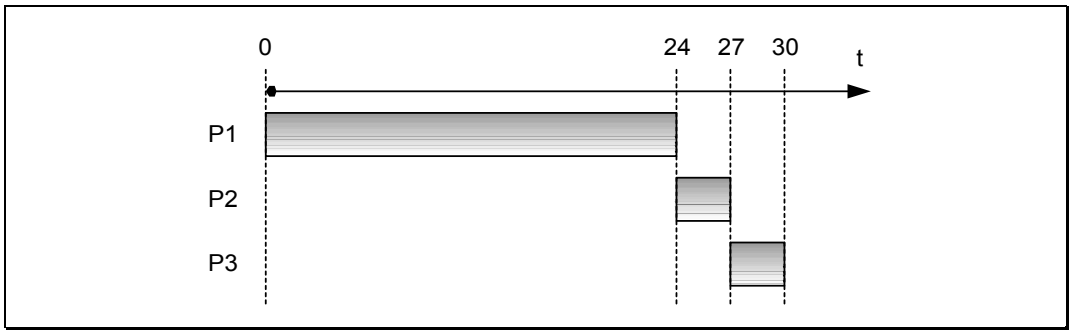
3.2. Algoritmi za dodelu procesora

U nastavku teksta ukratko su opisani najznačajniji algoritmi za dodelu procesora.

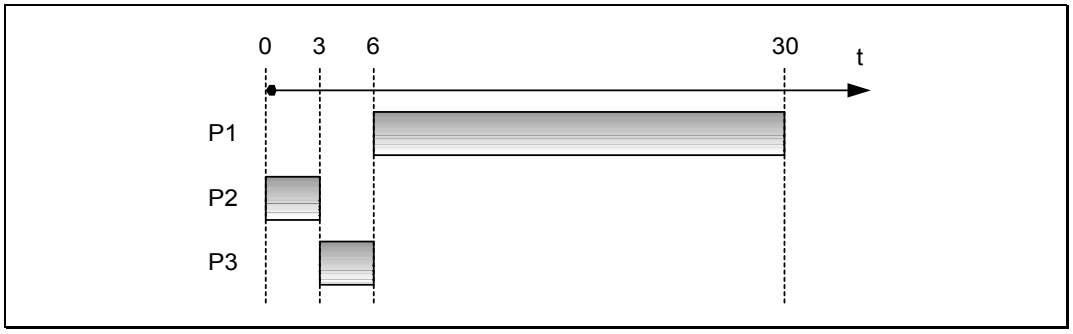
First Come, First Served (FCFS)

FCSC (prvi došao, prvi uslužen) je najprostiji algoritam za raspoređivanje procesora - procesi dobijaju procesor onim redom kojim su pristizali u red čekanja. Red čekanja funkcioniše po standardnom FIFO principu: kontrolni blok procesa koji je ušao u red čekanja na procesor stavlja se na kraj liste, dok se procesor uvek dodeljuje onom procesu koji je na početku liste.

Algoritam je lak i za razumevanje i za implementaciju. Međutim, srednje vreme čekanja (*waiting time*) za ovaj algoritam je krajnje dugačko. Uzmimo sledeći primer u kome imamo tri procesa P1, P2 i P3 čija su vremena izvršavanja 24, 3 i 3msec, respektivno. Pretpostavimo da procesi nailaze u sledećem poretku: P1, P2, P3. Gantogram aktivnosti ove sekvence dat je na slici 3.1. Vreme čekanja (*waiting time*) za procese P1, P2 i P3 je 0, 24 i 27msec respektivno, a srednje vreme čekanja (0+24+27)/3=17msec. Međutim, promenom redosleda nailaska procesa (slika 3.2.) situacija se drastično menja. Vremena čekanja za procese P1, P2 i P3 (slika 3.2) su 6, 0 i 3msec respektivno, a srednje vreme čekanja je t_w=(6+0+3)/3=3msec.



Slika 3.1. *Gantogram aktivnosti br.1 (FCFS algoritam)*



Slika 3.2. *Gantogram aktivnosti br.2 (FCFS algoritam)*

Na osnovu ovih primera se zapaža da srednje vreme čekanja zavisi kako od dužine trajanja procesa tako i od sekvence njihovog nailaska u sistem. Kod FCFS algoritma moguća je pojava konvoj efekta - svi procesi čekaju da se završi jedan proces koji dugo traje. Slična sitacija prisutna je i u saobraćaju: veliko sporo vozilo na uzbrdici prati veći broj manjih, ali bržih vozila, koji ne mogu da ga preteknu, na primer zbog pune linije.

FCFS se obično izvodi bez pretpražnjenja, odnosno prekidanja procesa, jer ne poštuje priorite procesa već samo vreme dolaska u red čekanja. To znači da se proces koji je dobio izvršava u potpunosti ili do trenutka blokiranja zbog čekanja na ulazno-izlaznu operaciju. Zbog toga je FCFS je krajnje nepodesan za interaktivne sisteme.

Shortest Job First (SJF)

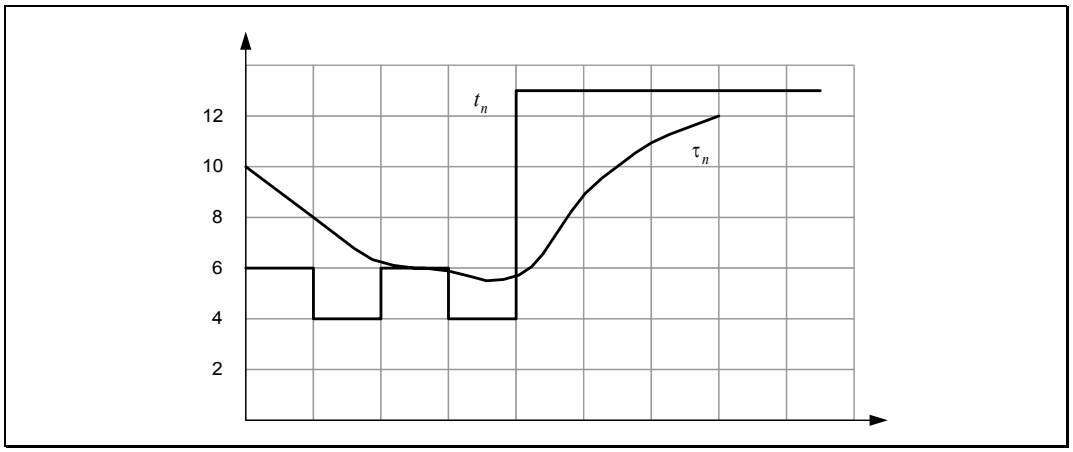
Suština algoritma *Shortest Job First* (prvo najkraći posao) jeste u sledećem: za sve procese u redu čekanja procenjuje se vreme potrebno za izvršavanje, a procesor se dodeljuje onom procesu kome treba najmanje vremena za izvršenje, odnosno procesu koji bi najkraće trajao. Na one procese iz reda koji imaju ista procenjena vremena izvršavanja primenjuje se FCFS algoritam.

Osnovni nedostatak SJF algoritma je u tome što sistem ne može unapred znati koliko će odgovarajući procesi trajati, odnosno ne može tačno proceniti trajanje procesa u celini. Predikcija trajanja procesa se zasniva na činjenici da se proces sastoji iz više ciklusa korišćenja procesora (*CPU burst*), koji se prekidaju ulazno-izlaznim operacijama. Procena trajanja se obavlja tako što se proceni trajanje sledećeg ciklusa korišćenja procesora na osnovu prethodnih ciklusa korišćenja procesora. Zato se algoritam preciznije može nazvati i najkraći sledeći CPU ciklus (*shortest next CPU burst*).

Jedan od načina da se obavi procena trajanja sledećeg ciklusa korišćenja procesora je određivanje eksponencijalne srednje vrednosti na bazi svih prethodnih CPU ciklusa. U sledećoj formuli τ_{n+1} je procenjena vrednost trajanja sledećeg CPU ciklusa, a t_n i τ_n realno trajanje i procenjena vrednost trajanja prethodnog CPU ciklusa. Uvešćemo koeficijenat α, takav da je $0 \leq \alpha \leq 1$. Formula za procenu trajanja sledećeg ciklusa korišćenja procesora je:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\,\tau_n$$

Formula kombinuje prvu prethodnu vrednost t_n i istoriju procesa koja je sublimisana u τ_n, dok koeficijent α određuje relativnu težinu istorije procesa, odnosno poslednjeg poznatog ciklusa. Ilustrovaćemo formulu sledećim primerom: ako je α=0.5, t_0=6, t_1=4, t_2=6, t_3=4, t_4=t_5=t_6=13 i τ_0=10 vremenskih jedinica, onda su procenjena vremena τ_1=8, τ_2=6, τ_3=6, τ_4=5, τ_5=9, τ_6=11, τ_7=12, kao što je prikazano na slici 3.3.



Slika 3.3. *Procena trajanja sledećeg ciklusa korišćenja procesora*

Postoje dve varijante SJF algoritma: sa pretpražnjenjem (*preemptive*) ili bez pretpražnjenja (*non-preemptive*). Ove dve varijante se ponašaju različito u slučaju da se novi proces pojavi u redu čekanja dok se tekući proces izvršava. Varijanta SJF bez pretpražnjenja će uvek završiti tekući proces bez obzira kakav se novi proces pojavio u redu čekanja. Kod varijante SJF sa pretpražnjenjem, ukoliko je vreme potrebno za

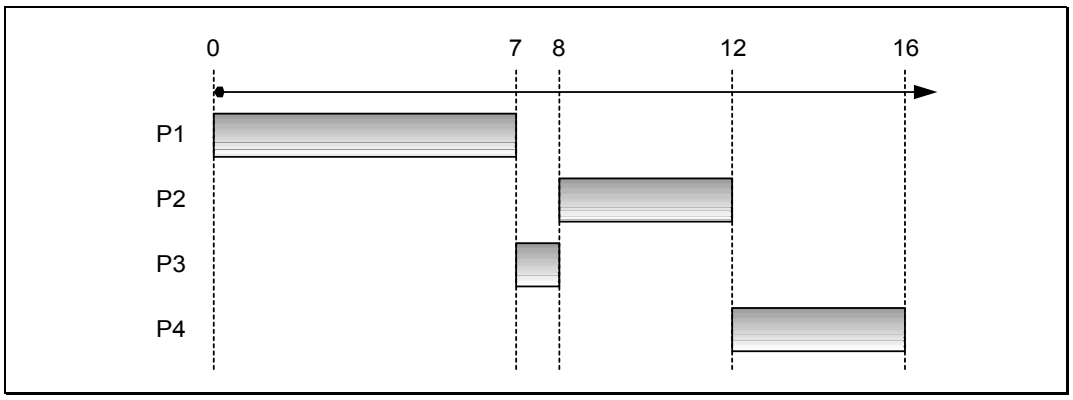
izvršenje novog procesa kraće od vremena potrebnog za završetak aktivnosti tekućeg procesa, procesor će biti dodeljen novom procesu. SJF sa pretpražnjenjem se naziva raspređivanje sa najmanjim preostalim vremenom (*shortest-remaining time first*).

U nastavku teksta data su dva primera koja ilustruju dodelu procesa korišćenjem prethodno opisanih varijanti SJF algoritma. Za SJF veoma je bitno znati dve informacije: vreme nailaska procesa u red čekanja (*arrival time*) i vreme potrebno za izvršenje procesa (*CPU burst time*). Kod FCFS je bitan samo jedan parametar, a to je vreme nailaska procesa u red.

U prvom primeru prikazan je SJF bez pretpražnjenja. Pretpostavimo da imamo četiri procesa čija su vremena nailaska u red čekanja i vremena izvršavanja data u sledećoj tabeli:

Proces	**Vreme nailaska u sistem**	**Vreme izvršavanja**
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Na slici 3.4. prikazan je gantogram aktivnosti dodele procesora.



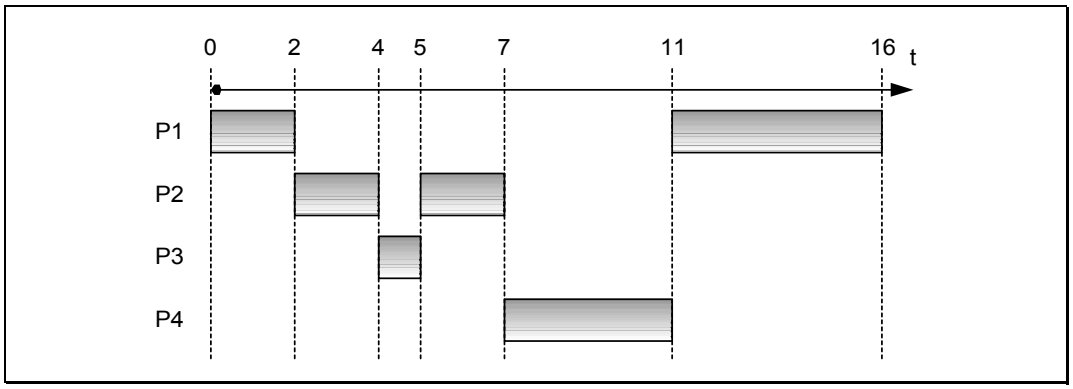
Slika 3.4. *Gantogram aktivnosti (*non-preemptive *SJF)*

Algoritam je bez pretpražnjenja, što znači da će se svaki proces izvršti do kraja. U prvom trenutku u redu čekanja se nalazi samo proces P1 čije izvršenje traje 7 vremenskih jedinica. U toku izvršavanja procesa P1 u red pristižu preostala tri procesa. Kada proces P1 završi svoje aktivnosti, SJF algoritam odabira proces P3 zato što je najkraći. Nakon završetka aktivnosti procesa P3 (t=8), u redu čekanja ostaju dva procesa sa jednakim vremenima izvršavanja, tako da se na njih primenjuje FCFS algoritam koji za dodelu procesora najpre odabira proces P2, a zatim P4. Vremena

čekanja za pojedine procese su: t_w(P1)=0, t_w(P2)=8-2=6, t_w(P3)=7-4=3, t_w(P4)=12-5=7, a srednje vreme čekanja t_w=(0+6+3+7)/4=4. U slučaju FCFS, srednje vreme čekanja bi bilo t_w=(0+5+7+7)/4=4.75.

U drugom primeru prikazan je SJF sa pretpražnjenjem. Takođe, pretpostavljamo situaciju identičnu prethodnoj: na sistemu se u trenucima 0, 2, 4 i 5 formiraju četiri procesa čija su vremena izvršenja 7, 4, 1 i 4 vremenske jedinice respektivno.

Algoritam je sa pretpražnjenjem, tako da se u slučaju pojave novog procesa proverava koliko je vremena ostalo tekućem procesu, a koliko je potrebno novom za izvršenje. Po potrebi, procesor se prazni i ustupa novom kraćem procesu. U prvom trenutku u redu čekanja se nalazi samo proces P1 (vreme izvršavanja 7 vremenskih jedinica), koji počinje da se izvršava. U trenutku t=2 pojavljuje se proces P2, čije je vreme izvršenja 4 vremenske jedinice, što je manje od vremena preostalog za izvršenje procesa P1 (5 vremenskih jedinica). Zbog toga se procesor u trenutku t=2 prazni i predaje procesu P2. U trenutku t=4 pojavljuje se novi proces P3. Opet se obavlja analiza preostalih vremena: P1 zahteva još 5 ciklusa, P2 još 2 ciklusa, a P3 kao novi proces samo 1 ciklus, što će izazvati novo pražnjenje nakon koga P3 dobija procesor. U trenutku t=5, P3 je završio aktivnosti, ali je istovremeno naišao proces P4. Dalje raspređivanje se vrši na sledeći način: u t=5, P1 ima zaostalo vreme od 5 ciklusa, P2 od 2 ciklusa, a novi proces P4 zahteva 4 ciklusa, tako da će procesor biti dodeljen procesu P2. U trenutku t=7 proces P2 završava aktivnosti, pa nastupa novo raspoređivanje: P1 ima zaostalo vreme od 5 ciklusa, tako da se procesor dodeljuje procesu P4 koji zahteva 4 ciklusa. P4 koristi procesor do završetka svojih aktivnosti, posle čega se on u trenutku t=11 dodeljuje procesu P1. Na slici 3.5 prikazan je gantogram aktivnosti dodele procesora:



Slika 3.5 *Gantogram aktivnosti (*preemptive *SJF)*

Vremena čekanja za pojedine proces su t_w(P1)=11-2=9, t_w(P2)=5-4=1, t_w(P3)=0, t_w(P4)=7-5=2, a srednje vreme čekanja t_w=(9+1+0+2)/4=3.

U svakom slučaju, SJF je optimalan po pitanju srednjeg vremena čekanja, što naročito važi za varijantu sa pretpražnjenjem koja se u svakoj sitaciji prilagođava najkraćem procesu.

Raspoređivanje na osnovu prioriteta procesa

Kod prioritetno-orijentisanih algoritama, svakom procesu se dodeljuje prioritet, posle čega algoritam bira proces sa najvećim prioritetom. Prioritet je, po pravilu, celobrojna vrednost, a najčešće se koristi konvencija po kojoj manji broj znači veći prioritet i obrnuto (vrednost 0 predstavlja maksimalni prioritet). U slučaju procesa sa jednakim prioritetom, odluka se donosi po FCFS principu.

SJF je specijalan slučaj prioritetno-orijentisanih algoritama za raspoređivaje: prioritet je obrnuto proporcionalan trajanju sledećeg CPU ciklusa procesa. To znači da duži procesi imaju manji prioritet i obrnuto.

Prioritetno-orijentisani algoritmi mogu biti realizovani sa pretpražnjenjem (*priority preemptive scheduling*) - tekući proces u tom slučaju prekida izvršenje ukoliko se pojavi proces većeg prioriteta (provera se odvija uvek kada se pojavi novi proces). Ukoliko je algoritam realizovan bez pretpražnjenja (*priority non-preemptive scheduling*), proces koji je dobio kontrolu ne prekida izvršenje, bez obzira što se pojavo proces većeg prioriteta.

Prioriteti mogu biti:

- interno definisani, na osnovu resursa koje proces zahteva i odnosa CPU i ulazno-izlaznih aktivnosti prisutnih u procesu i
- eksterno dodeljeni (programer može favorizovati određene procese po nekim eksternim kriterijumima).

Glavni problem prioritetno-orijentisanih algoritama je blokiranje niskoprioritetnih procesa. Procesi niskog prioriteta mogu veoma dugo (neograničeno) da čekaju na procesor. Na primer, kod jako opterećenih sistema, novi procesi neprestano pristižu u sistem, tako da se može dogoditi da procesi visokog prioriteta potpuno blokiraju nisko prioritetne procese. Problem zakucavanja niskoprioritetnih procesa se razrešava povećanjem prioriteta sa vremenom provedenim u redu čekanja na procesor. Preciznije rečeno, rezultujući prioritet se formira na osnovu:

- početnog prioriteta, koji proces dobija kada uđe u red čekanja na procesor i
- vremena provedenog u redu čekanja (*aging*).

Kako sa vremenom provedenim u redu čekanja rezultujući prioritet raste, niskoprioriteni procesi vremenom dobijaju veću šansu za dodelu procesora. U svakoj prioritetnoj šemi definiše se zona početnih prioriteta i vremenski interval ažuriranja prioriteta.

Ako prioritete dinamički određuje operativni sistem, dobro je uzeti u obzir sledeća razmatranja:

- Procesima koji intenzivno koriste ulazno-izlazne uređaje treba dodeliti viši prioritet, jer su ovakvi procesi često blokirani, a procesor im treba u kraćim intervalima.
- Procesima koji intenzivno koriste procesor i operativnu memoriju treba dodeliti niži prioritet. Procesor se može bolje iskoristiti ukoliko se najpre dodeli procesima koji intenzivno koriste ulazno-izlazne uređaje. Ovi procesi će procesor koristiti relativno kratko, a zatim se blokirati, posle čega se nesmetano mogu izvršavati procesi iz grupe procesa koji intenzivno koriste procesor. U obrnutom slučaju, *I/O bound* procesi bi dugo čekali na procesor i zauzimali mesto u operativnoj memoriji, a ulazno-izlazni sistem bi ostao prilično besposlen.

Round Robin (RR)

Round Robin algoritam za raspoređivanje je specijalno dizajniran za interaktivne sisteme. Round Robin se može posmatrati kao FCFS algoritam sa pretpražnjenjem, odnosno sa pravilnim vremenskim prebacivanjem između procesa. RR funkcioniše na sledeći način: najpre se definiše kratak vremenski interval (vremenski kvantum - *time slice*), reda veličine 10-100 msec i kružni red čekanja na procesor. Svaki prispeli proces ubacuje na kraj liste. Od sistema se očekuje da generiše prekidni signal po isteku vremenskog kvantuma, nakon čega se procesu oduzima kontrola nad procesorom i predaje sledećem procesu iz kružne liste. U okviru jednog vremenskog kvantuma moguće su sledeće situacije:

[1] Proces je završio aktivnost pre isteka vremenskog kvantuma - proces tada oslobađa procesor, a RR algoritam uklanja proces iz kružne liste i predaje kontrolu sledećem procesu.

[2] Proces nije završio aktivnosti, ali mora da prekine izvršenje kada mu istekne vremenski kvantum. RR algoritam postavlja prekinuti proces na kraj reda čekanja, a kontrolu predaje sledećem procesu iz liste

[3] Proces se blokirao zbog čekanja na ulazno-izlazne operacije. Blokirani proces oslobađa procesor, a RR algoritam predaje kontrolu sledećem procesu iz liste. Proces prelazi u red čekanja na kraj RR liste tek nakon povratka u stanje READY.

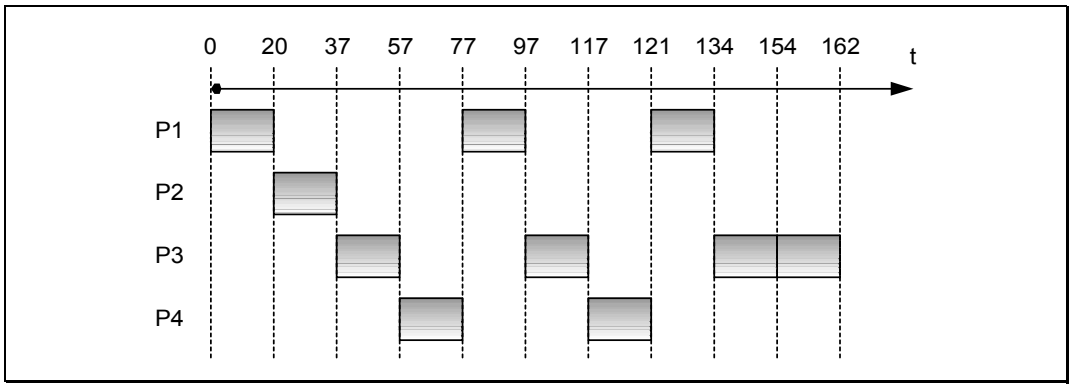
RR algoritam je fer prema procesima - svi dobijaju procesor na korišćenje ravnomerno i ravnopravno. Ako imamo n procesa u redu čekanja, svakom procesu pripada 1/n procesorskog vremena. Takođe, ako je vremenski kvantum dužine q, nijedan proces u stanju READY neće čekati više od (n-1)q vremena na sledeću dodelu procesora. Međutim, srednje vreme odziva RR algoritma je po pravilu jako dugo. Takođe, treba imati u vidu da prebacivanje konteksta između dva procesa unosi

premašenje u vidu nekorisnog vremenskog kašnjenja. Ovo premašenje je jedan od glavnih činilaca prilikom izbora veličine kvantuma. Što je kvantum manji, procesor će biti ravnomernije raspodeljen, ali će i prebacivanje konteksta između procesa biti češće, tako da će premašenje sistema biti veće. Ukoliko je kvantum veći, RR algoritam konvergira ka FCFS algoritmu.

U nastavku teksta dat je primer rapoređivanja procesora Round Robin algoritmom sa veličinom kvantuma 20 vremenskih jedinica. Pretpostavimo da su četiri procesa P1, P2, P3 i P4 došla u red, skoro u istom trenutku i da je formirana kružna lista P1->P2->P3->P4. Neka su vremena izvršavanja ovih procesa 53, 17, 68 i 24 vremenskih jedinica, respektivno.

Procesor najpre dobija proces P1. Po isticanju kvantuma, procesor mu se oduzima i predaje sledećem procesu iz kružne liste, a to je P2, dok P1 odlazi na kraj kružne liste. Proces P2 će se završiti pre isteka kvantuma, tako da će se u trenutku t=37 procesor dodeliti sledećem procesu iz liste (procesu P3). Proces P2 se kao završen briše iz liste. Procesu P3 su potrebna četiri kvantuma, a procesu P4 sva. U t=57, P3 završava svoj kvantum i ustupa mesto za P4, potom P4 ustupa mesto P1, pa ovaj P3 i tako redom. U t=121 proces P4 završava svoje aktivnosti i ispada iz liste, a u t=134 takođe i P1. Poslednja dva kvantuma pripadaju procesu P3.

Gantogram aktivnosti dodele procesora u slučaju RR algoritma prikazan je na slici 3.6.



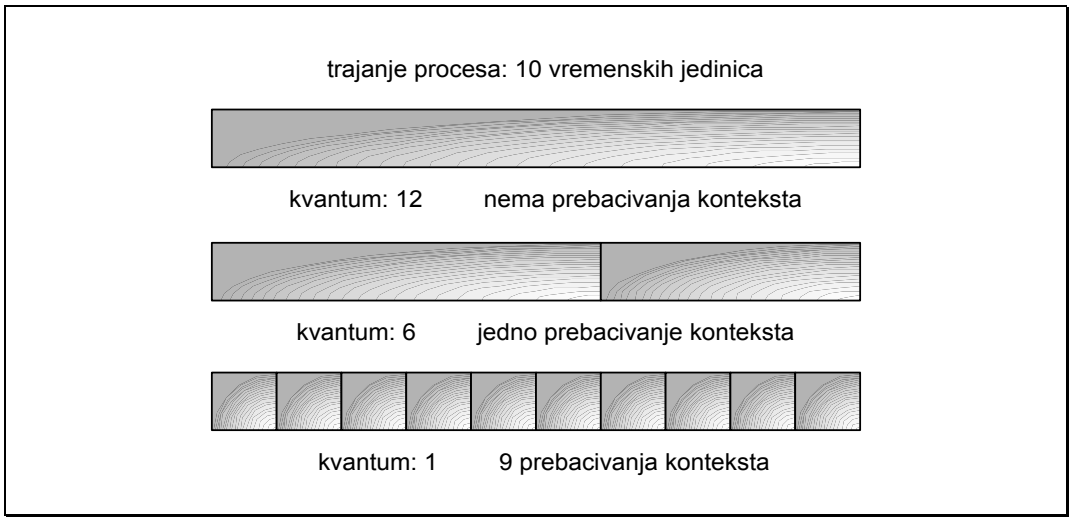
Slika 3.6. *Gantogram aktivnosti (Round Robin)*

Generalno gledano, RR ima najbolje vreme odziva (*response time*), a veće srednje vreme završetka procesa (*turnaround time*) u odnosu na SJF, čak i u slučaju zanemarivanja vremena potrebnog za prebacivanje konteksta procesa.

Performanse RR algoritma zavise od veličine vremenskog kvantuma. Kao što je već je rečeno, u slučaju većih vremenskih kvantuma RR konvergira ka FCFS algoritmu. U slučaju veoma malih vremenskih kvantuma, RR algoritam se naziva deljenje procesora (*CPU sharing*), jer tada svaki proces radi brzinom jednakom 1/n brzine

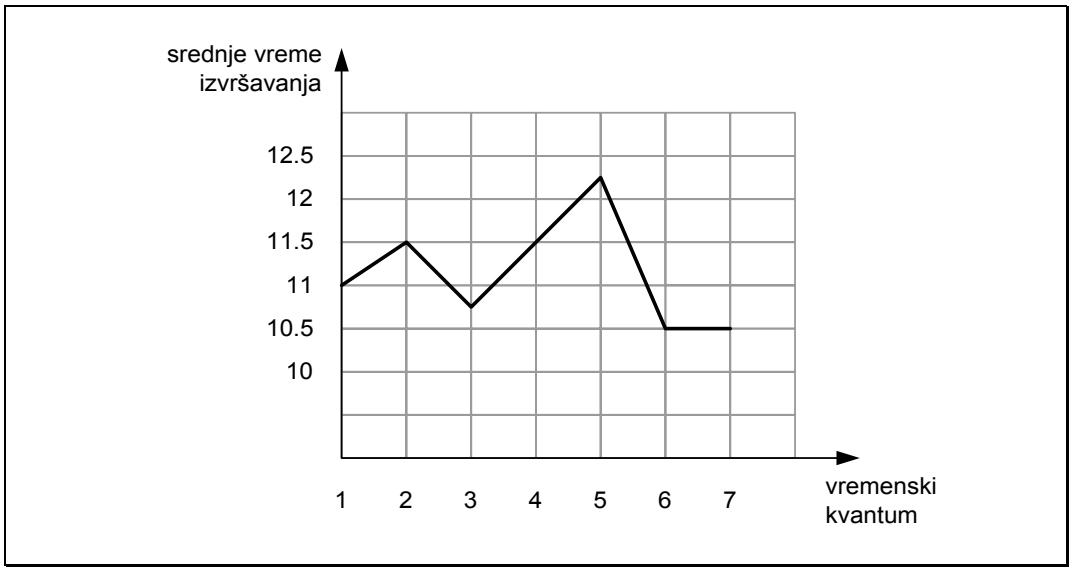
realnog procesora. Ova brzina predstavlja idealan slučaj - pretpostavlja se da dispečer obavlja prebacivanje konteksta trenutno. Realno, dispečer je realizovan softverski i izvršava se kao zaseban proces, i kao takav unosi kašnjenje pri svakom prebacivanju konteksta. Realna brzina izvršavanja svakog procesa je manja od 1/n brzine realnog CPU i opada sa porastom učestanosti prebacivanja konteksta.

Obavićemo analizu štetnog uticaja veoma malih vremenskih kvantuma: za proces koji traje 10 jedinica vremena i vremenski kvantum veličine 12 jedinica vremena nema prebacivanja konteksta. Ali, ako vremenski kvantum traje 6 vremenskih jedinica, tada će broj prebacivanja konteksta biti 1, a ako vremenski kvantum traje jednu jedinica vremena, tada će broj prebacivanja konteksta biti 9 (slika 3.7).



Slika 3.7. *Štetni uticaj malih vremenskih kvantuma*

Sledeći primer će pokazati da se srednje vreme izvršavanja procesa za male vrednosti vremenskog kvantuma ne popravlja obavezno sa povećanjem vremenskog kvantuma. Pretpostavljamo da su vremena potrebna za izvršenje procesa 6,3,1 i 7 vremenskih jedinica i da svi procesi pristižu u sistem u trenutku t=0. Primer ukazuje na to da za izabrani niz procesa vremenski kvantumi veličine 2, 4 i 5 povećavaju vreme izvršenja umesto, da ga smanjuju.



Slika 3.8 *Zavisnost srednjeg vremena izvršavanja od veličine vremenskog kvantuma*

3.3. Raspoređivanje u više procesorskih redova

Ova klasa algoritama za raspoređivanje se odnosi na sisteme i sitacije u kojima se procesi mogu klasifikovati u različite grupe. Na primer, procesi se mogu podeliti na interaktivne procese, odnosno procese koji rade u prvom planu (*foreground*), i procese koji rade u pozadini (*background*). Ove dve klase procesa imaju različite zahteve za vreme odziva i zbog toga se koriste različiti algoritmi za raspoređivanje procesora. Po pravilu, interaktivni procesi imaju osetno veći prioritet u odnosu na procese koji rade u pozadini.

Algoritmi za raspoređivanje u više redova (*multilevel queue scheduling*) dele red čekanja u više razdvojenih redova, kao što je prikazano na slici 3.9. Procesi se ubacuju u odgovarajući red čekanja na osnovu osobina procesa, kao što su veličina memorije, prioritet i tip procesa. Svaki red čekanja ima poseban nezavistan algoritam za raspoređivanje koji odgovara njegovim potrebama. Na primer, može se napraviti poseban red čekanja za interaktive procese, koji se opslužuje po RR algoritmu i poseban red čekanja za procese koji rade u pozadini, koji se opslužuje po FCFS algoritmu.

Slika 3.9. *Raspoređivanje u više procesorskih redova*

U slučaju više redova čekanja, kada je u pitanju selekcija procesa između različitih redova, primenjuje se prioritetna šema sa pretpražnjenjem. U primeru sa slike 3.9 imamo četiri reda čekanja koji su sortirani po opadajućem prioritetu, a svaki red čekanja ima apsolutni prioritet u odnosu na niže redove. Procesi iz četvrte klase moraju da sačekaju ili da daju prednost procesima iz viših klasa, a ako su dobili procesor, oslobodiće ga u slučaju da se pojavi proces iz više klase.

Pored apsolutnog prioriteta, može se uvesti i RR tehnika između različitih redova, što znači da svaki red dobija procesor na neko vreme, a u tom vremenu selektuju se procesi iz tog reda, po algoritmu koji je definisan za taj red. Na primer, u slučaju imamo dva reda (interaktivi red i red za procese koji rade u pozadini), sistem može dodeliti 80% procesorskog vremena interaktivnom redu i 20% vremena pozadinskom redu.

Povratna sprega između redova čekanja

U prethodno definisanoj šemi, procesi pripadaju svojim redovima čekanja od nastanka do završetka aktivnosti. Ponekad se zahteva dodatna fleksibilnost koja će omogućiti prelazak procesa iz jednog u drugi red. Ova fleksibilnost se postiže primenom algoritama sa povratnom spregom između redova (*multilevel feedback queue*). Ideja je da se procesi na osnovu svojih karakteristika (vezanih za korišćenje procesora i ulazno-izlaznih uređaja) razdvoje u različite redove. Ako proces zahteva puno procesorskog vremena, treba ga ubaciti u niskoprioritetni red čekanja, dok interaktivne procese i procese koji dominantno koriste ulazno-izlazne uređaje treba staviti u redove čekanja visokog prioriteta. Takođe, uvodi se i vremenska komponenta koja spečava zakucavanje - ako proces čeka isuviše dugo, prebacuje se u red višeg prioiteta.

Sledeći primer ilustruje raspoređivanje u tri reda čekanja, obeleženih od 0 do 2. Red 0 je najvišeg prioriteta - najpre se izvršavaju procesi iz reda 0, a zatim procesi iz

reda 1 i reda 2. Redovi funkcionišu po različitim algoitmima, procesima iz reda 0 i 1 se procesor dodeljuje po RR algoritmu sa kvantumom 8 i 16 respektivno, dok se procesima iz reda 2 procesor dodeljuje po FCFS algoritmu. Povratna sprega između redova je realizovana na sledeći način: svaki proces koji uđe u sistem se najpre ubacuje u red 0 i ako ne završi sve aktivnosti u kvantumu reda 0, prebacuje se na kraj liste prvog reda, gde dobija šansu da u većem kvantumu obavi posao. Ako i tada ne obavi posao prebacuje se u red 2 sa FCFS algoritmom. Na ovaj način su favorizovani kratki procesi koji će se brzo izvršiti i osloboditi procesor. Samo veoma dugi procesi upadaju u red 2, gde će se izvršavati samo kada u redu 0 i redu 1 nema procesa.

U principu, parametri koji definišu redove čekanja sa povratnom spregom su: broj redova, algoritam za raspoređivanje za svaki red posebno i metode za određivanje reda u koji će proces ući po kreiranju i kada proces može da pređe u red višeg, odnosno nižeg prioriteta. Ovo rešenje primenjuju praktično svi savremeni operativni sistemi.

Raspoređivanje u višeprocesorskoj okolini

Raspoređivanje procesa na višeprocesorskim sistemima je daleko kompleksnije, jer zahteva ozbiljnu sinhronizaciju procesora. Kao i u slučaju algoritama primenljivih na jednoprocesorske sisteme, svakako da postoji više šema i da njihova primena zavisi od mnogo faktora.

Pretpostavićemo da su procesori funkcionalno identični, odnosno da svaki procesor može da izvršava bilo koji proces iz reda čekanja. U tom slučaju može se obezbediti poseban red čekanja za svaki procesor, ili konstruisati zajednički red čekanja. U prvom slučaju jedan procesor može biti potpuno besposlen ukoliko je odgovarajući procesorski red prazan, a drugi procesor može biti maksimalno opterećen. Zato se umesto zasebnih formira zajednički red čekanja, koji obezbeđuje ravnomerno opterećenje svih procesora.

U slučaju zajedničkog reda čekanja, operativni sistem može koristiti više tehnika za raspoređivanje:

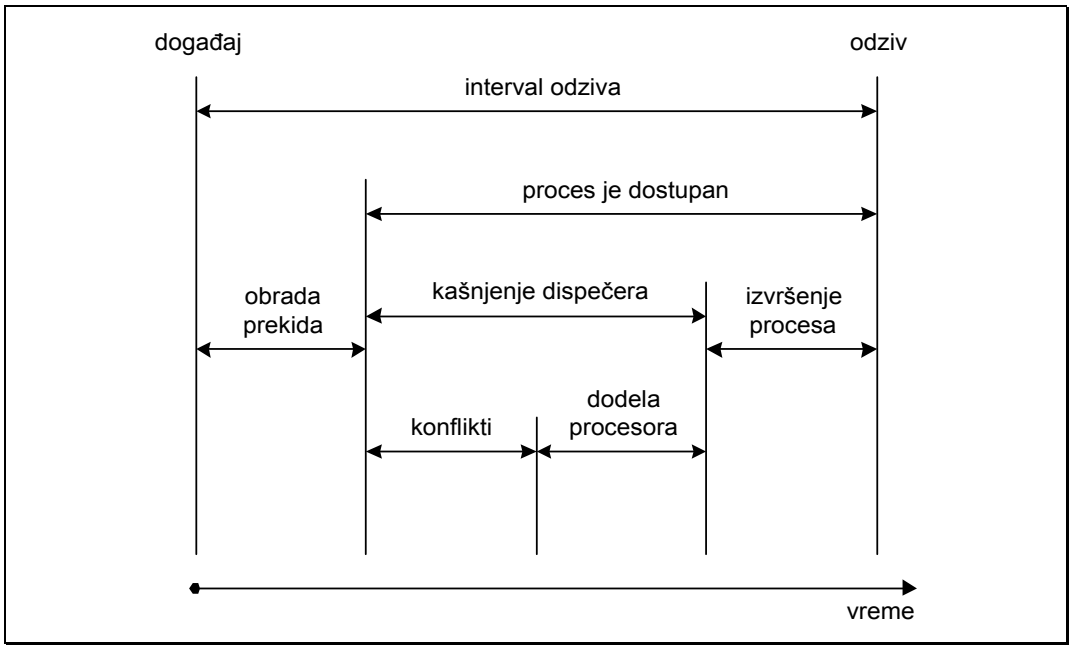
[1] svaki procesor ispituje zajednički red i odabira procese koje će izvršavati. Opasnost postoji prilikom pristupa zajedničkim podacima. Takođe, dva procesora ne smeju da preuzmu isti proces. Ovaj način multiprocesiranja poznat je kao simetrično multiprocesiranje.

[2] jedan procesor, koji je proglašen glavnim, određuje koje će procese izvršavati drugi procesori (međuprocesorski *master-slave* odnos). Ovo multiprocesiranje se naziva asimetričnim: *master* procesor izvršava kernelski kod i ulazno-izlazne operacije, dok ostali procesori izvršavaju korisnički kod koji im definiše *master* procesor. Navedeni sistemi nisu efikasni jer sve ulazno-izlaznr operacije izvršava samo jedan procesor.

Rapoređivanje u realnom vremenu

Operativni sistemi u realnom vremenu dele se na dve klase: čvste (*hard*) i meke (*soft*). Kod čvrstih *real-time* operativnih sistema se zahteva izvršavanje kritičnih poslova u garantovanom vremenskom roku, odnosno intervalu. Generalno, kod njih se proces kreira sa informacijom za koje vreme mora biti obavljen ili za koje vreme mora da izvrši ulazno-izlaznu operaciju. Raspoređivač mora da obezbedi procesu da se izvrši u svim zahtevanim rokovima ili da odbaci proces kao nemoguć za izvršenje (rezervacija resursa). Raspoređivač tačno zna koliko odgovarajući sistemski pozivi traju, što zbog velike varijacije u vremenu potrebnom za čitanje i upis podataka na medijume nije moguće u sistemima sa sekundarnom memorijom (a samim tim i sa virtuelnom memorijom).

Meki *real-time* operativni sistemi su manje restriktivni - njih odlikuje veći prioritet izvršavanja kritičnih procesa. Većina savremnih operativnih sistema pored interaktivnih funkcija ubacuju i osobine mekih realnih sistema, odnosno kritične procese. Implementacija takvih osobina zahteva da se raspoređivač pažljivo projektuje. Prvo, sistem mora da ima prioritetno raspređivanje i da realni procesi imaju maksimalni prioritet. Prioritet realnih procesa se ne menja u vremenu. Drugo, dispečersko kašnjenje mora da bude veoma malo. Na slici 3.10. je prikazan interval u kome se čeka na odziv. Nakon događaja slede vreme za procesiranje prekida i kašnjenje dispečera, koji poziva realni proces da odgovori na događaj. A da bi se dispečersko kašnjenje učinilo što manjim, neophodno je da sistemski pozivi imaju mogućnost pretpražnjenja.



Slika 3.10 *Interval odziva*

4

Sinhronizacija procesa

Sadržaj poglavlja:

4.1. Problem sinhronizacije
4.2. Kritične oblasti
4.3. Semafori
4.4. Klasični problemi sinhronizacije i semafori
4.5. Kritični regioni i monitori

4.1. Problem sinhronizacije

Kooperatvni proces može da utiče na druge procese ili da trpi uticaj drugih procesa. Po pravilu, do toga dolazi usled deljenja podataka koji su smešteni na odgovarajućim memorijskim lokacijama ili u datotekama. Moguća posledica konkuretnog pristupanja zajedničkim podacima je nekonzistentnost tih podataka. U tom slučaju, promenljive koje formiraju podatak nemaju odgovarajuće vrednosti. Na primer, posmatrajmo dva procesa P1 i P2 koji žele da privremeno sačuvaju neku vrednost na istoj memorijskoj lokaciji A. Neka se događaji odvijaju po sledećem scenariju:

[1] Proces P1 proverava da li je memorijska lokacija A slobodna. Lokacija A je slobodna. Proces P1 je obavešten da je lokacija A slobodna.

[2] Proces P2 proverava da li je memorijska lokacija A slobodna. Lokacija A je slobodna. Proces P2 je obavešten da je lokacija A slobodna.

[3] Proces P1 upisuje podatak na memorijsku lokaciju A.

[4] Proces P2 upisuje drugi podatak na memorijsku lokaciju A.

[5] Proces P1 čita POGREŠAN podatak sa memorijske lokacije A !

Da bi se izbegle slične situacije, operativni sistem mora da obezbedi mehanizme za očuvanje konzistentnosti podataka, odnosno mehanizme konkurentnog izvršavanja procesa.

Ilustracija na problemu proizvođač-potrošač

U poglavlju 2 (kernel i procesi), definisali smo paradigmu proizvođača i potrošača. Predloženo rešenje sa ograničenim baferom kapaciteta N elemenata i zajedničkim kontrolnim promenljivim *in* i *out* funkcionisalo je za N-1 elemenata u baferu. Pretpostavimo da želimo da izmenimo kod tako da bafer radi sa punim kapacitetom. To se može postići uvođenjem nove deljive promenljive *counter*, koja opisuje broj zauzetih elemenata u baferu. Početna vrednost promenljive *counter* je 0. Promenljiva *counter* se inkrementira svaki put kada proizvođač ubaci novi element u bafer, a dekrementira svaki put kada potrošač uzme jedan elemenat iz bafera. U nastavku teksta dat je modifikovani kod za proizvođački i potrošački proces.

Definicija dodatnih promenljivih.

```
item buffer[N]          // definicija bafera veličine N
int in = 0;             // prvo slobodno mesto
int out = 0;            // prvo zauzeto mesto
int counter = 0;        // broj zauzetih elemenata
```

Proizvođač:

```
item next_produced;
while (1) {
   /* proizvođač proizvodi informaciju */
   while (counter == N)
   /* bafer je pun, proizvođač čeka
      da potrošač nešto uzme iz bafera */
   buffer[in] = next_produced;
   in = (in+1) % N;
   counter++;
}
```

Potrošač:

```
item next_consumed;
while (1) {
   while (counter==0);
   /* bafer je prazan, potrošač čeka
```

```
        da proizvođač nešto stavi u bafer /*
    next_consumed = buffer[out];
    out = (out+1) % N;
    counter--;
    /* potrošač konzumira informaciju */
}
```

Iako je kod proizvođačkog i potrošačkog procesa na prvi pogled korektan, ovi procesi se mogu izvršavati konkurentno samo ako se operacije *counter*++ i *counter*-- izvršavaju atomski ili nedeljivo, odnosno bez prekidanja. Nekorektne situacije nastaju kao posledica implementacije operacija inkrementiranja i dekrementiranja promenljive counter. Po pravilu, operacije *counter*++ i *counter*-- izvršavaju se na sledeći način: najpre se vrednost promenljive *counter* upiše u regitar procesa, zatim se vrednost registra inkrementira ili dekrementira i na kraju vrati u promenljivu counter. U nastavku teksta dat je kod za operacije *counter*++ i *counter*-- na simboličkom mašinskom jeziku. Registri *register1* i *register2* su lokalni CPU registri i mogu biti isti.

Operacija *counter*++:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

Operacija *counter*--:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Na primer, pretpostavimo da je vrednost promenljive *counter* u početnom trenutku 5, i da proizvođački i potrošački proces obavljaju svoje operacije *counter*++ i *counter*-- konkurentno. Nakon izvršenja ovih operacija vrednost promenljive *counter* može biti 4, 5 ili 6, ali jedina ispravna vrednost je 5. Praktično, izvršava se šest mašinskih instrukcija , ali u slučaju da globalne operacije koje ih sadrže nisu atomski definisane, potencijalni prekidi i izbor procesa koji će nastaviti posle prekida mogu dovesti do preklapanja (*interleaving)* odnosno do mešavine ovih instrukcija, a samim tim do različitih konačnih rezultata. Jedna vrsta takvog preklapanja je:

naredba	izvršava	instrukcija	vrednost
T0	proizvođač	register1 = counter	register1 = 5
T1	proizvođač	register1 = register1 + 1	register1 = 6
T2	potrošač	register2 = counter	register2 = 5
T3	potrošač	register2 = register2 – 1	register2 = 4
T4	proizvođač	counter = register1	counter = 6
T5	potrošač	counter = register2	counter = 4

Ova sekvenca izvršavanja je sasvim moguća i za posledicu ima pogrešnu vrednost promenljive *counter* (4). Promena redosleda naredbi T4 i T5 takođe dovodi do nekorektne situacija, odnosno pogrešne vrednosti promenljive *counter* (6). U oba slučaj greška nastupa zato što oba procesa konkurentno pristupaju zajedničkoj promenljivoj.

Znači, ukoliko više procesa pristupa i modifikuje iste podatke konkurentno, krajnja vrednost zajedničkih podataka zavisi od sekvence instrukcija koje tim podacima pristupaju. Ova sekvenca se naziva stanje trke (*race condition*) i zavisi od prekidnih signala i načina raspređivanja procesa. U trci se pobednik obično ne zna, tako da je krajnji rezultat sekvence neizvestan, što se ne sme dozvoliti. Stanje trke se može sprečiti ukoliko se obezbedi mehanizam sinhronizacije koji će dozvoliti pristup zajedničkom resursu samo jednom od konkurentnih procesa u jednom vremenskom trenutku.

4.2. Kritična sekcija

Kritična sekcija (*critical section*) je deo koda u kome proces pristupa ili modifikuje zajedničke podatke, kao što su memorijske promenljive, tabele ili datoteke. Svaki operativni sistem može dozvoliti samo jednom procesu da bude u svojoj kritičnoj sekciji. Dok je jedan proces u svojoj kritičnoj sekciji, ni jedan drugi proces ne sme da uđe u svoju kritičnu sekciju. Prema tome, izvršavanje kritičnih sekcija procesa je međusobno isključivo u vremenu (*mutual exclusion*).

Problem kritične sekcije se rešava definicijom protokola po kome će procesi sadrađivati međusobno, uz poštovanje pravila međusobne isključivosti za kritične sekcije. Svaki proces mora od sistema tražiti dozvolu da uđe u kritičnu sekciju. To se obezbeđuje pomoću specijalnog koda - ulazne sekcije (*entry section*). Kada prođe ulaznu sekciju, proces može ući u svoju kritičnu sekciju. Kada je završi, proces mora objaviti svim ostalim procesima da više nije u kritičnoj sekciji, što se postiže delom koda koji se zove izlazna sekcija (*exit section*).

```
do {
```

```
      entry section
         critical section
      exit section
         /* ostatak koda koji ne pripada
            kritičnoj sekciji */
   } while (1);
```

Određivanje dela programskog koda koji predstavlja kritičnu oblast je zadatak programera.

Kritična oblast se može realizovati:

- softverski,
- hardverski,
- pomoću operativnog sistema (sistemskim pozivima) i
- višim programskims strukrurama za sinhronizaciju (monitorima).

Softverska realizacija kritične sekcije

Kritična oblasti se realizuje softverski, bez pomoći operativnog sistema. Prilikom razvoja programa ne uzimaju se u obzir pretpostavke o brzini i o broju procesora na sistemu. Algoritam mora garantovati svakom procesu mogućnost ulaska u svoju kritičnu sekciju.

Pretpostavke i softverskog rešenja kritične sekcije su:

- međusobno isključenje koje garantuje da se unutar kritične oblasti u jednom trenutku može naći najviše jedan proces,
- proces koji je izvan svoje kritične oblasti ne sme sprečiti druge procese da uđu u svoje kritične oblasti (jedino proces unutar kritične oblasti može sprečiti ostale procese da uđu u kritičnu oblast),
- proces ne sme neograničeno dugo da čeka da uđe u svoju kritičnu oblast i
- proces se sme neograničeno dugo ostati u svojoj kritičnoj oblasti (mora da izađe i omogući drugim procesima da uđu u svoje kritične oblasti).

Sledeći primeri ilustruju probleme softverske realizacije kritične oblasti. Svi programi su dati pomoću pseudo-koda na C jeziku. Krenimo od najprostijeg slučaja dva konkurentna procesa P0 i P1.

Algoritam striktne alternacije

U algoritmu striktne alternacije (*strict alternation*) zajednička promenljiva *turn* određuje koji proces može ući u svoju kritičnu sekciju. Samo onaj proces čija je vrednost indeksa jednaka vrednosti promenljive *turn* može ući u svoju kritičnu sekciju. Ulazana sekcija je izvedena kao *while* struktura koja čeka da promenljiva *turn* postane jednaka indeksu procesa. Kada proces uđe u svoju kritičnu sekciju i obavi svoje aktivnosti, prelazi se na izlaznu sekciju koja promenljivoj *turn* dodeljuje vrednost indeksa drugog procesa.

Kod procesa P0 je:

```
do {
   while (turn != 0) ;          // ulazna sekcija
      /* kritična sekcija */
   turn = 1;                    // izlazna sekcija
      /* ostatak koda */
} while (1);
```

Kod procesa P1 je:

```
do {
   while (turn != 1);           // ulazna sekcija
      /* kritična sekcija */
   turn = 0;                    // izlazna sekcija
      /* ostatak koda */
} while (1);
```

Generalizovano, kod procesa Pi je:

```
do {
   while (turn != i) ;          // ulazna sekcija
      /* kritična sekcija */
   turn = j;                    // izlazna sekcija
      /* ostatak koda */
} while (1);
```

Rešenje zadovoljava uslov međusobnog isključenja (*mutual exclusion*): samo jedan proces može da uđe u svoju kritičnu sekciju. S druge strane, P1 ne može ući u svoju kritičnu sekciju ako P0 ne prođe svoju kritičnu sekciju i postavi vrednost promenljive *turn* na 1, tako da može doći do blokiranja sistema. Ovde se poštuje stroga alternacija procesa po pitanju ulaska u svoje kritične oblasti, tj. strogo se može dešavati samo sekvenca P0->P1->P0->P1…. Nedostatak ovog rešenja je *polling* tehnika ulazne sekcije: stalno se proverava stanje promenljive *turn*, što troši procesorsko vreme. Rešenje je nepovoljno u slučaju dva procesa od kojih je jedan znatno sporiji u odnosu

na drugi. Brži proces će dugo čekati da spori proces dodeli odgovarajuću vrednost promenljivoj *turn* i time mu omogući ulazak u kritičnu sekciju.

Realizacija kritične sekcije bez stroge alternacije

Osnovni nedostatak prethodnog algoritma je sam princip striktne alternacije. Algoritam ne sadrži dovoljno informacija o procesima - konkretno, jedina informacija koja figuriše u algoritmu tiče se procesa koji ima pravo da uđe u kritičnu sekciju, bez obzira da li proces to želi ili ne. Stroga alternacija se može eliminisati uvođenjem binarne promenljive *flag* za svaki proces posebno. Ako je flag[i]=1, proces Pi je spreman i želi da uđe u svoju kritičnu sekciju. Ako je flag[i]=0, proces Pi ne želi da uđe u svoju kritičnu sekciju, jer njegovo izvršenje trenutno ne zavisi od zajedničkih podataka.

Inicijalno, za dva procesa P0 i P1 uvode se dve kontrolne promenljive *flag*[0] i *flag*[1] sa početnom vrednošću 0.

```
int flag[2];
flag[0] = flag[1] = 0;          // boolean
```

Kod procesa Pi je:

```
do {
   flag[i] = 1;
   while (flag[j]);             // ulazna sekcija
      /* kritična sekcija */
   flag[i] = 0;                 // izlazna sekcija
      /* ostatak koda */
} while (1);
```

Rešenje je, na prvi, pogled korektno, jer zadovoljava međusobno isključenje, a procesi van kritičnih sekcija ne utiču jedan na drugog. Proces prvo postavlja svoj fleg na 1, što znači da želi da uđe u svoju kritičnu sekciju, a zatim proverava da li je drugi proces postavio svoj fleg na 1. Ako jeste, proces čeka u petlji da drugi proces izađe iz svoje kritične sekcije i postavi svoj fleg na 0. Međutim, sledeća sekvenca dokazuje da je ovaj algoritam pogrešan:

[1] P0 postavlja flag[0]=1, posle čega se kontrola dodeljuje procesu P1,

[2] P1 postavlja flag[1]=1,

[3] oba procesa upadaju u beskonačnu petlju, jer su oba flega postavljena na 1 i nijedan ne može da uđe u svoju kritičnu sekciju.

Ovo rešenje otpada kao pogrešno.

Dekker-Petersonov algoritam

Kombinacijom prethodnih rešenja, Dekker i Peterson su definisali prihvatljivo softversko rešenje kritične oblasti, zasnovano na promeljivim *turn* i *flag*. Promenljive *flag* ukazuju da proces želi da u uđe u svoju kritičnu sekciju, dok promenljiva *turn* određuje proces koji ima prednost ulaska i time obezbeđuje mehanizam međusobnog isključenja. Proces Pi najpre postavlja vrednost promenljive *turn* na j (turn=j), odnosno daje šansu drugom procesu (Pj) da uđe u svoju kritičnu sekciju. Proces Pi čeka na ulazak u kritičnu sekciji samo ako su ispunjeni sledeći uslovi:

[1] proces Pj želi da uđe u svoju kritičnu sekciju (flag[j] =1)

[2] procesu Pj je data šansa za ulazak u kritičnu sekciju (turn = j).

U svim drugim situacijama proces Pi ulazi u svoju kritičnu sekciju. Kod procesa Pi je:

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn = j);
      /* kritična sekcija */
   flag[i] = 0;
      /* ostatak koda */
} while (1);
```

Svi uslovi su zadovoljeni:

- poštuje se međusobno isključenje: zbog pomoćne promenljive *turn*, u kritičnu sekciju će ući samo jedan proces, i to onaj koji ima prednost. Proces koji nema prednost ne odustaje, ali čeka na red.
- nema striktne alternacije: neka se P1 zaglavi u ostatku koda koji ne pripada kritičnoj sekciji dovoljno dugo, tako da proces P0 krene da izvršava *do-while* petlju iz početka. Tada je flag[1]=0, flag[0]=1, turn=1. Vrednost logičkog izraza *while* petlje procesa P0 biće 0, što znači da pa proces P0 može ponovo ući u kritičnu sekciju a da ne čeka na proces P1.
- nema zaglavljivanja u beskonačnoj petlji, jer ne može biti istovremeno turn=0 i turn=1, odnosno ne postoji mogućnost da oba procesa beskonačno dugo ostanu unutar *while* petlje.

Pekarski algoritam

Prethodno rešenje, koje se odnosi na slučaj sa dva procesa, mora se generalizovati za realan slučaj sa N procesa u sitemu. Algoritam koji omogućava tu

generalizaciju je poznat pod imenom pekarski algoritam (*bakery algorithm*). Algoritam se oslanja na poštovanje reda, čime se sprečava haos (red se poštuje u bankama, poštama, pa i pekarama - otud naziv algoritma). Pravilo je jasno, svaki kupac u pekari dobija broj (*ticket*), a radnik uslužuje kupce redom, počev od onog sa najmanjim brojem. Slično pravilo primenjuje se i na procese: svakom procesu se dodeljuje broj. Pekarski algoritam ne garantuje da će dva procesa uvek dobiti različite brojeve. Ako dva procesa dobiju isti broj, odlučiće ime, odnosno indeks procesa (*process ID*). Pravila pekarskog algoritma su sledeća:

- sekvenca brojeva se generiše u rastućem redu i dodeljuje procesima (na primer: 1,2,3,3,3,3,4,5...),
- procesi se opslužuju na osnovu vrednosti dodeljenih brojeva, počev od procesa kome je dodeljen najmanji broj i
- ako dva procesa, Pi i Pj, dobiju iste brojeve, prvo se opslužuje proces sa manjim indeksom (ako je i<j, tada se prvo opslužuje Pi) .

Definišimo i leksikografski poredak dva elementa (*ticket*, *process ID*) sledećom formulom:

(a) $(a,b) < (c,d)$ ako je $a < c$ ili ako je $a = c$ i $b < d$,

(b) $max\ (a_0, \ldots, a_{n-1})$ je broj a_k, takav da je $a_k \geq a_i$ za svako $i \in [0, n-1]$

Definisaćemo zajedničke podatke za pekarski algoritam

```
int choosing[n];          // boolean
int number[n];
```

Za svaki proces se definiše binarna promenljiva *choosing*[i], koja služi za sinhronizaciju prilikom dobijanja broja, i celobrojna promenljiva *number*[i], koja sadrži broj koji je proces Pi dobio i koji mu definiše prioritet. Ni jedan proces ne može ući u kritičnu sekciju dok neki proces dobija broj. Svi podaci su inicijalno postavljeni na 0.

Pekarski algoritam funkcioniše na sledeći način:

```
do {
   choosing[i] = 1;
   number[i] = max (number[0], … ,number[n-1]) + 1;
   choosing[i] = 0;
   for (j=0; j<n; j++) {
         while (choosing[j]);
         while ( (number[j] != 0) &&
                 ((number[j], j) < (number[i], i)) );
   }
      /* kritična sekcija */
   number[i] = 0;
      /* ostatak koda */
```

```
} while (1);
```

Proces Pi najpre dobija broj, a dok dobija broj svi kandidati za ulazak u kritičnu sekciju moraju da čekaju jer je vrednost choosing[i] =1. Proces Pi dobija najveći broj u tom trenutku, uvećan za 1, što mu garantuje najniži prioritet. Kada dobije broj, proces Pi oslobađa fleg choosing[i]=0 i pristupa ulaznoj sekciji. U ulaznoj sekciji, proces ostaje sve dok ne postane proces sa najmanjim brojem, odnosno najvišim prioritetom, što se proverava u petlji (number[j], j) < (number[i], i)). Proces zatim ulazi u kritičnu sekciju, a nakon izlaska iz nje, anulira svoj broj i više ne učestvuje u takmičenju za ulazak u kritičnu sekciju.

Hardverska realizacija kritične oblasti

Problem kritične sekcije se na sistemima sa jednim procesorom može rešiti blokadom prekidnih signala (*interrupt disable*). U tom slučaju, proces koji uđe u kritičnu sekciju se ne može prekinuti, tako nema potrebe za sinhronizacijom. To rešenje nije dobro, jer remeti sistemski časovnik. Kako zabrana i ponovna dozvola prekidnih signala traju dugo, još veći problem se javlja ukoliko je okruženje višeprocesorsko. To znači da je potrebno naći alternativno rešenje za realizaciju kritičnih sekcija.

Kritična oblast može se implementirati na nivou harvera, ako postoji procesorska instrukcija koja to podržava Većina mikroprocesora obezbeđuje instrukcije pomoću kojih se može pročitati i izmeniti memorijska reč ili razmeniti sadržaj dve memorijske reči. Osobina ovih instrukcija je nedeljivost, odnosno atomarni način izvršavanja, što znači da se te instrukcije ne mogu prekinuti u toku izvršavanja.

Naredba *TestAndSet (&target)* radi sledeće:

[1] čita vrednost korisničke promenljive *target*,

[2] kopira tu vrednost u korisničku promenljivu *rv* i

[3] postavlja vrednost promenljive *target* na true (1).

Kako je *TestAndSet* mašinska naredba procesora, ona se izvršava kao jedna operacija koja ne može biti prekinuta. Operacija *TestAndSet* garantuje, da promenljivama *target* i *rv* istovremeno može pristupiti samo jedan procesor, što znači da se ova operacija ne može biti paralelno izvršena na više procesora. U praksi *rv* predstavlja neki registar procesora, a *target* neku memorijsku lokaciju.

Sada, kada imamo hardversku podršku za realizaciju kritične oblasti, može se koristiti sledeći algoritam - proces pomoću promenljive *lock*, čija je inicijalna vrednost 0, štiti svoju kritičnu sekciju.

```
int lock = 0;
```

Kod za sve procese je:

```
do {
   while (TestAndSet(lock));
      /* kritična sekcija */
   lock = 0;
      /* ostatak koda */
} while (1);
```

Promenljivoj *lock* se inicijalno dodeljuje vrednost 0. Prvi proces koji želi da uđe u svoju kritičnu sekciju postavlja vrednost *lock* na 1 i nastavlja sa izvršenjem, odnosno ulazi u svoju kritičnu sekciju. Prilikom postavljanja vrednosti promenljive *lock* proces ne može biti prekinut. Svi ostali procesi, čekaće u *while* petlji dok je lock=1. Kada proces napusti svoju kritičnu sekciju, postaviće *lock* na 0, što će omogućiti drugim procesima da uđu u svoju kritičnu sekciju. Hardver garantuje da nikada dva procesa ne mogu pročitati vrednost lock=0, zato što je operacija *TestandSet* nedeljiva - samo jedan proces može da pročita vrednost promenljive *lock* i da je neposredno posle toga postavi na 1.

Instrukcija *swap* je takođe nedeljiva, a obavlja razmenu vrednosti dve memorijske promenljive preko privremenog registra procesora. Vrednost promenljive *a* se najpre upiše u regitar *temp*, zatim se vrednost promenljive *b* upiše u promenljivu *a*, a vrednost registra *temp* u promenljivu *a*, čime *a* i *b* praktično razmene vrednosti.

Problem međusobnog isključenja se može razrešiti i pomoću instrukcije swap. Kod za sve procese je:

```
do {
   key = 1;
   while (key == 1) swap(lock, key);
      /* kritična sekcija */
   lock = 0;
      /* ostatak koda */
} while (1);
```

Promenljivoj *lock* se inicijalno dodeljuje vrednost 0. Prvi proces koji želi da uđe u svoju kritičnu sekciju, deteketovaće ovu vrednost, a zatim će instrukcijom *swap* razmeniti vrednosti promenljivih *lock* i *key* (lock =1, key=0). Proces ulazi u svoju kritičnu sekciju. Svi ostali procesi će detektovati vrednost lock=1 i čekaće da proces napusti svoju kritičnu sekciju i postavi vrednost promenljive *lock* na 0. Nakon toga, neki drugi proces može ući u svoju kritičnu sekciju.

4.3. Semafori

Dosadašnja rešenja nije lako primeniti na kompleksnije slučajeve. U tom kontekstu uvodi se novi metod za sinhronizaciju procesa - semafor.

Semafor je celobrojna promenljiva nenegativne vrednosti koja štiti neki resurs i omogućava komunikaciju između procesa (mehanizam međusobnog isključenja ili sinhronizacije aktivnosti). Vrednost semafora određuje da li je resurs kog taj semafor štiti slobodan. Ukoliko je vrednost semafora val(s)=0, resurs je zauzet. Resurs je slobodan ukoliko je vrednost semafora val(s)>0. Svaki semafor ima svoju početnu vrednost, a nad njim se mogu izvršiti dve nedeljive operacije, odnosno primitive, koje su radi jednostavnosti predstavljene kao *signal(s)* i *wait(s)*. Nedeljivost ovih operacija se ogleda u sledećim činjenicama:

- *signal* i *wait* se ne mogu podeliti na više ciklusa i
- dva procesa ne mogu istovremeno izvršavati ove operacije nad istim semaforom.

Kao što je već rečeno, operacija *wait(s)* dekrementira vrednost semafora samo ako je s>0. U slučaju da je s=0, proces koji izvršava ovu operaciju čeka da vrednost semafora postane pozitivna (odnosno da drugi proces izvrši operaciju *signal* nad semaforom), nakon čega će dekrementirati vrednost semafora. Operacija *signal(s)* inkrementira vrednost semafora (s=s+1). Simbolički, operacije *signal* i *wait* se mogu predstaviti na sledeći način:

```
wait(s): when (s > 0) do decrement s;
signal(s): increment s;
```

Upotreba semafora

U nastavku teksta opisana je moguća upotreba semafora za rešavanje problema kritične sekcije (međusobnog isključenja) i sinhronizacije aktivnosti procesa.

Problem kritične sekcije može se rešiti upotrebom binarnog semafora *mutex* (*mutual exclusion*), čija je inicijalna vrednost 1:

```
semaphore mutex;     /* inicijalno, mutex = 1 */
```

Sledeći kod, koji važi za sve procese na sistemu, obezbediće zaštitu kritičnih sekcija:

```
do {
   wait(mutex);
      /* kritična sekcija */
```

```
    signal(mutex);
       /* ostatak koda */
} while (1);
```

Ulazna sekcija svih procesa razrešava se operacijom *wait*. Samo jedan proces proći će semafor *mutex*, oboriti mu vrednost na 0 i ući u svoju kritičnu sekciju. Svi drugi procesi čekaju na semaforu. Kada proces napusti kritičnu sekciju, oslobodiće semafor komandom signal(mutex).

Saradnju između procesa čije su aktivnosti sinhronizovane obezbeđuje semafor *proceed* čija je inicijalna vrednost 0. Mehanizam sinhronizacije se najjednostavnije može objasniti sledećim primerom: posmatraju se dva procesa P1 i P2. Procesu P1 je u tački S1 neophodna informacija koju proces P2 obezbeđuje u tački S2. Proces P1 ne može da nastavi dalje izvršenje u tački S1 dok proces P2 ne prođe tačku S2 i obezbedi neophodnu informaciju.

Programiranje sinhronizacije može se obaviti na sledeći način:

Proces P1	Proces P2
...	...
...	...
wait(proceed)	**S2**
S1	**signal(proceed)**
...	...
...	...

- program koji se izvršava u procesu P1:

```
/* proces P1 čeka na infomaciju
   koju obezbeđuje proces P2 */
wait (proceed);
/* proces P1 nastavlja izvršenje */
```

- program koji se izvršava u procesu P2:

```
/* proces P2 obezbeđuje infomaciju
   neophodnu za dalje izvršenje procesa P1 */
signal (proceed);
```

gde je proceed ime semafora kojim se obezbeđuje sinhronizacija.

Proširena definicija semafora

Osnovni nedostaci softverske i hardverske realizacije navedene metode zaštite kritične sekcije semaforima su:

- ignorisanje prioriteta procesa - može se desiti da najprioritetniji proces uđe u kritičnu oblast, tek posle mnoštva neprioritetnih procesa,
- mogućnost da proces bude zaposlen čekanjem (*busy waiting*). Do ove pojave dolazi ukoliko se proces neprestano vrti u while petlji i ne radi ništa korisno, već samo proverava vrednost neke promenljive kako bi saznao da li može ući u svoju kritičnu sekciju ili ne. Iako ne radi ništa korisno, proces troši procesorsko vreme. Kako se u kritičnoj sekciji u jednom trenutku može naći samo jedan proces, veliki broj procesa koji čeka na ulazak u kritičnu sekciju opteretiće procesor. Na primer, posmatrajmo situaciju sa 1.000 procesa od kojih se jedan nalazi u kritičnoj sekciji, a ostali čekaju na ulazak u svoje kritične sekcije. To znači da imamo jedan proces koji radi nešto korisno i 999 procesa koji čekajući na troše procesorsko vreme.

Ovi nedostaci se mogu izbeći ukoliko se kritična oblast realizuje pomoću sistemskih poziva kojima se proces može blokirati. Proces koji čeka da uđe u kritičnu oblast postaje blokiran (*blocked*), sve dok ga neki drugi proces ne oslobodi blokade. U kontekstu semaforskih tehnika, proces koji čeka na semaforu čija je vrednost 0 blokira se i kao takav ne troši procesorsko vreme.

Redefinisaćemo pojam semafora i operacije *signal* i *wait* i time ukloniti mogućnost da proces bude zaposlen čekanjem. Ideja je da se za svaki semafor uvede poseban red čekanja (semaforski red). Svi procesi koji izvršavaju operaciju *wait* nad semaforom čija je vrednost negativna ili jednaka nuli ($val(s) \leq 0$) se pomoću sistemskog poziva *sleep* blokiraju i prevode u semaforski red. Procesor se oslobađa i predaje nekom drugom procesu koji nije blokiran. Proces nastavlja svoje izvršenje nakon sistemskog poziva *wakeup* koji ukida blokadu procesu. Blokada se može ukinuti samo ako je proces prvi u semaforskom redu ili ima najviši prioritet na semaforu, a neki drugi proces obavi *signal* operaciju nad tim semaforom.

Semafor u proširenom kontekstu definiše kao struktura (zapis) čiji su elementi celobrojna vrednost semafora (koja može biti i negativna) i lista pokazivača na procese koji čekaju na semafor (semaforski red).

```
typedef struct {
   int value;
   struct process *L;
} semaphore;
```

Vrednost semafora s može biti:

- $val(s) > 0$ (resurs je slobodan)

- val(s)=0 (resurs je zauzet, ali je semaforski red prazan)
- val(s)<0 (resurs je zauzet, u semaforskom redu postoje procesi koji čekaju na taj semafor)

Operacija *wait* dekremetira vrednost semafora i ako je ona nakon toga negativna, stavlja proces u semaforski red i blokira proces:

```
wait(S) {
   S.value--;
   if (S.value < 0) {
      /* dodaj proces u semaforski red */
      sleep();
   }
}
```

Operacija signal inkrementira vrednost semafora i ako je ona posle toga negativna ili jednaka nuli uklanja jedan od procesa iz liste (izvršava *wakeup* sistemski poziv i prebacuje proces u red čekanja za spremne procese):

```
signal(S) {
   S.value++;
   if (S.value <= 0) {
      /* ukloni proces P iz semaforskog reda */
      wakeup(P);
   }
}
```

Napomene:

(a) Ovo je prvo predloženo rešenje koje koristi servise operativnog sistema (sistemaski pozivi sleep i wakeup). Sva ranije pomenuta rešenja su čisto programerska ili hardverska i kao takva se ne oslanjaju na usluge operativnog sistema.

(b) Zapaža se da u proširenoj definiciji vrednost semafora može biti negativna, što ukazuje na to da veći broj procesa čeka na semaforu. U takvim slučajevima, operacija signal po nekom kriterijumu "budi" neki od "uspavanih" procesa.

(c) U proširenoj definiciji semafora operacije wait i signal moraju ostati nedeljive.

Problem zastoja i zakucavanja

Loša implementacija semafora može dovesti do situacije u kojoj jedan ili više procesa beskonačno dugo čekaju na događaje koji se nikada neće dogoditi. Takvi procesi se nalaze u situaciji koja se zove zastoj (*deadlock*, *deadly embrace*).

Neka su P0 i P1 procesi koji redom izvršavaju sledeće operacije nad semaforima S i Q, čije su inicijalne vrednosti 1:

Proces P0	Proces P1
`wait(S)`	`wait(Q)`
`wait(Q)`	`wait(S)`
`...`	`...`
`signal(S)`	`signal(Q)`
`signal(Q)`	`signal(S)`

Ova sekvenca događaja dovešće do zastoja. Kada proces P0 izvrši operaciju *wait(S)*, vrednost semafora S postaje 0, a to se isto dešava i sa semaforom Q, čiju će vrednost P1 postavi na 0. Procesi P0 i P1 će se nakon toga naći u stanju zastoja: P0 čeka na semaforu Q, čiju vrednost može uvećati proces P1 naredbom *signal(Q)*. Proces P1 ne može da izvrši operaciju *signal(Q)*, jer čeka na semaforu S, čiju vrednost može da uveća proces P0. Nijedna signal operacija se neće izvršiti, jer oba procesa čekaju na semaforima.

Do zakucavanja (*starvation*) dolazi ukoliko procesi beskonačno dugo čekaju u semaforskom redu. Ova situacija je moguća ukoliko se procesi iz semaforskog reda uklanjaju po LIFO (*Last In First Out*) principu.

Binarni i brojački semafori

Binarni semafori (*binary semaphore*) su semafori čija vrednost može biti 0 ili 1. Vrednosti brojačkih semafora (*counting semaphore*) nisu limitirane na ovaj način, ali se ovi semafori teže implementiraju od binarnih. Sledeći primer ilustruje realizaciju brojačkog semafora S pomoću dva binarna semafora S1 i S2 (promenljive tipa binary-semaphore) i jedne celobrojne promenljive C:

Strukture podataka i inicijalizacija vrednosti:

```
binary-semaphore S1, S2;
int C;

S1 = 1; S2 = 0;
C = initial value of semaphore S;

wait(S) {
   wait(S1);
      C--;
      if (C < 0) {
         signal(S1);
         wait(S2);
```

```
      }
   signal(S1);
}

signal(s) {
   wait(S1);
      C++;
      if (C <= 0) signal(S2);
   else signal(S1);
}
```

4.4. Klasični problemi sinhronizacije i semafori

U nastavku teksta opisana je moguća primena semaforskih tehnika za rešavanje sledećih problema:

- proizvođač-potrošač (problem ograničenog bafera),
- problem čitalaca i pisaca i
- problem večere filozofa.

Problem ograničenog bafera

Sledeći primer ilustruje rešavanje problema proizvođač-potrošač upotrebom semaforskih tehnika. Sinhronizacija se obezbeđuje sledećim brojačkim semaforima, čije vrednosti pripadaju intervalu [0,N-1], gde je N kapacitet bafera:

- *item_available* (potrošač ne može uzeti ništa iz bafera, ukoliko proizvočač to prethodno nije stavio u bafer) i
- *space_available* (ukoliko je bafer pun, proizvođač ne može ubaciti informaciju).

Dodatno, uvodi se *mutex* semafor *buffer_ready* kojim se bafer štiti kao nedeljivi resurs. Na taj način se sprečava mogućnost zabune sa pokazivačima koja nastaje ukoliko jedan proces čita informacije iz bafera, dok drugi proces nešto upisuje u bafer.

Podsećamo na promenljive koje dele proizvođač i potrošač.

```
typedef struct {} item;  // struktura elementa bafera
item buffer [N]          // definicija bafera
int in = 0;              // prvo slobodno mesto
int out = 0;             // prvo zauzeto mesto
```

Inicijalizacija semafora:

```
item_available = 0;
space_available = N;
buffer_ready = 1;
```

Proizvođač:

```
do {
   /* proizvođač proizvodi informaciju */
   item next_produced;
   wait (space_available);
   wait (buffer_ready);
      buffer[in] = next_produced;
      in = (in+1)%N;
   signal (buffer_ready);
   signal (item_available);
} while (1);
```

Potrošač:

```
do {
   item next_consumed;
   wait (item_available);
   wait (buffer_ready);
      next_consumed = buffer[out];
      out = (out+1)%N;
   signal (buffer_ready);
   signal (space_available);
   /* potrošač konzumira informaciju */
} while (1);
```

Problem čitalaca i pisaca

Konkurentni procesi mogu da dele objekte podataka. U zavisnosti od toga šta procesi žele da rade sa podacima, izdvajamo sledeće dve kategorije procesa:

- procese čitaoce (*readers*), koji žele samo da čitaju deljive objekte i
- procese pisce (*writers*), koji žele da menjaju sadržaj objekata.

Dva procesa čitaoca mogu istovremeno da čitaju sadržaj deljenog objekta, dok proces koji piše ne sme dozvoliti konkurentni pristup deljenom objektu. To znači da dva pisca ne mogu istovremeno da menjaju sadržaj zajedničkog objekta i da nema

čitanja dok pisac ne obavi svoje aktivnosti. Za razliku od bafera koji ima N elemenata, ovde je reč o jednom objektu, čiji se sadržaj čita ili modifikuje.

Problem čitalaca i pisaca može se rešiti pomoću dva *mutex* semafora:

- *write*, koji služi za međusobno isključenje procesa pisaca i čitaoca i
- *counter_ready*, koji obezbeđuje međusobno isključenje pristupa kontroloj promenljivoj *read_count* (promenljiva *read_count* čuva informaciju o broju aktivnih čitalaca).

Inicijalne vrednosti semafora i promenljive read_count su: counter_ready=1, write=1, readcount=0.

Kod procesa pisaca je:

```
wait(write);
   /* proces modifikuje sadržaj objekta */
signal(write);
```

Pisac će nešto upisati pod uslovom da niko od čitaoca ili drugih pisaca nije aktivan. Kod procesa čitaoca je:

```
wait(counter_ready);
   readcount++;
   if (readcount == 1) wait(write);
signal(counter_ready);

/* proces čita sadržaj objekta */

wait(counter_ready);
   readcount--;
   if (readcount == 0) signal(write);
signal(counter_ready):
```

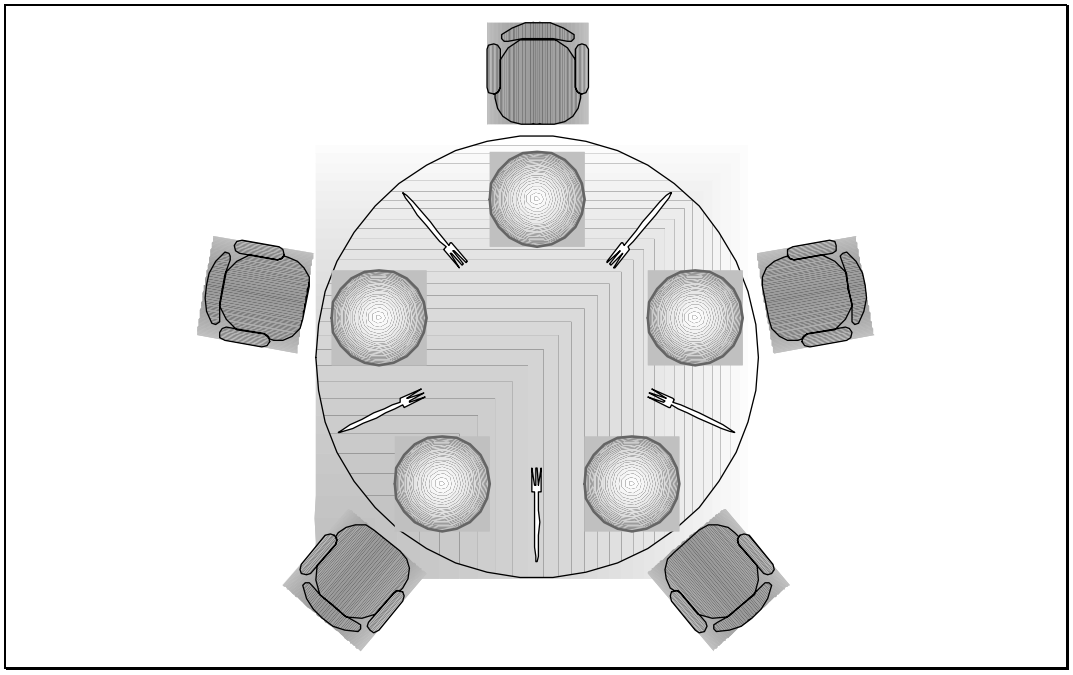
Proces čitalac proverava koliko ima aktivnih čitalaca u sistemu, a pomoću semafora *counter_ready* kontroliše promenljivu *readcount*. Ukoliko je vrednost promenljive *readcount* nakon inkrementiranja jednaka 1, on je jedini čitaoc i mora eventualno sačekati na pisca - *wait(write)*. U protivnom, on nije prvi i može slobodno da čita. Proces završava čitanje i dekrementira vrednost promenljive readcount. Ako je čitaoc poslednji, odnosno jedini, izvršiće operaciju signal(write) i time osloboditi objekat za pisce.

Problem večere filozofa

Analizirajmo slučaj pet filozofa koji sede za okruglim stolom. Ispred svakog filozofa nalazi se po jedan tanjir sa špagetama. Između svaka dva tanjira postavljena je po jedna viljuška. Filozofi su svoj život posvetili razmišljanju. Ako postanu gladni, oni jedu špagete, pa ponovo razmišljaju. Problem je u sledećem: da bi neki filozof mogao jesti, potrebne su mu dve viljuške. Zato, kada filozof postane gladan, najpre pogleda da li može uzeti obe viljuške i ako može, onda jede (za to vreme njegovi susedi ne mogu jesti). Ako ne može, filozof misli i čeka dok obe viljuške ne postanu slobodne. Kada završi sa jelom, filozof spušta obe viljuške i nastavlja da misli.

Ovaj problem može da se reši na sledeći način: svaki filozof predstavlja proces, a svaka viljuška semafor. Definisaćemo zajedničke podatke - pet semafora kojima se štite viljuške (*fork [i]*), a čiju ćemo vrednost postaviti na 1:

```
semaphore fork[5];
```



Slika 4.1. *Problem večere filozofa*

Kod koji rešava problem večere filozofa, za svakog filozofa posebno je sledeći:

```
philosopher (i) {
   do {
```

```
    wait(fork[i])              /* jedna viljuška*/
    wait(fork[(i+1)%5])        /* druga viljuška*/
       /* filozof jede */
    signal(fork[i]);
    signal(fork[(i+1)%5]);
    /* filozof misli */
} while (1);
```

Rešenje je dobro samo na prvi pogled - zastoj je moguć ako svi filozofi ogladne odjedanput i uzmu viljušku sa leve strane. Tada niko neće razrešiti drugu *wait* operaciju, odnosno niko neće doći do druge viljuške. Zastoj se može sprečiti ukoliko se:

- dozvoli da najviše četiri filozofa sednu za sto
- dozvoli da filozof uzme viljuške samo ako su obe slobodne
- koriste asimetrična rešenja - na primer, da neparni filozofi (prvi, treći i peti) uzmu prvo svoje leve viljuške a zatim desne, a parni drugi i četvrti prvo desne pa leve.

4.5. Kritični regioni i monitori

Iako su semafori dobro rešenje problema sinhronizacije procesa, ozbiljne greške koje nehotično može napraviti programer mogu dovesti do zastoja u sistemu. Na primer, programer može nehotično zameniti redosled operacija *signal* i *wait* nad semaforom koji štiti kritičnu sekciju:

Ispravno:

```
wait(mutex)
/* kritična sekcija */
signal(mutex)
```

Pogrešno:

```
signal(mutex)
/* kritična sekcija */
wait(mutex)
```

U ovakvoj situaciji veći broj procesa će moći da uđe u svoje kritične sekcije istovremeno, što je protivno principu međusobnog isključenja. Programer, takođe, može zameniti jednu operaciju drugom kao u sledećem primeru:

```
wait(mutex)
   /* kritična sekcija */
wait(mutex)
```

Umesto oslobađanja semafora, operacijom signal, nastupa zastoj.

Kritični regioni

Ovakve greške se mogu izbeći korišćenjem konstrukcija karakterističnih za više programske jezike, koje se nazivaju kritični regioni (*critical region*), odnosno uslovni kritični regioni (*conditional critical region*). Osim njih, definisana je i fundamentalna konstrukcija za sinhronizaciju procesa - monitor.

Kritični region zahteva promenljivu v tipa T, koja je zajednička za veći broj procesa, što se simbolički obeležava na sledeći način:

```
v: shared T;
```

Promenljivoj v se može pristupati samo unutar regiona naredbi S, uslovno, sa sledećom sintaksom:

```
region v when (B) S;
```

Ova konstrukcija ima sledeće značenje: dok se naredba S izvršava, nijedan drugi proces neće pristupati promenljivoj v. Izraz B je logički izraz, koji upravlja radom u kritičnom regionu. Kada proces uđe u kritični region, izraz B se ispituje i ukoliko je tačan, naredba S se izvršava. Ako izraz nije tačan, međusobno isključenje se napušta i odlaže, dok B ne postane tačan.

Kritični regioni štite od grubih, nehotičnih programerskih grešaka u sinhronizaciji. U nastavku teksta dat je primer rešavanja problema ograničenog bafera pomoću kritičnih regiona. Zajednički podaci su:

```
struct buffer {
   int pool[n];
   int count, in, out;
}
```

Kod za prozvođački proces koji u deljivi bafer ubacuje element *next_produced* je:

```
region buffer when (count < n) {
   pool[in] = next_produced;
   in = (in+1)%n;
   count++;
}
```

Kod za potrošački proces koji iz deljivog bafera uklanja element *next_consumed* je:

```
region buffer when (count > 0) {
   nextc = pool[out];
   out = (out+1)%+n;
   count--;
}
```

Zajedničke promenljive su zaštićene unutar kritičnog regiona i samo im odgovarajući proces može pristupati.

Monitori

Monitori predstavljaju takođe konstrukciju visokog nivoa koja služi za sinhronizaciju. Monitori omogućavaju programeru da resurse posmatra kao objekte (struktura monitora podseća na instance klasa kod objektno orijentisanog programiranja). Svaki monitor se sastoji iz:

- promenljivih koje opisuju deljeni resurs, odnosno objekat, čije vrednosti definišu stanje monitora. Lokalne promenljive vidljive su samo unutar monitora,
- skupa procedura i funkcija kojima se pristupa objektu, odnosno varijablama monitora i
- dela programa koji inicijalizuje objekat, a koji se izvršava samo jednom, prilikom kreiranja objekta.

```
monitor ime_monitora {

   /* deklaracije deljenih promenljivih */

   P1 (...) { /* definicija procedure/funkcije P1 */ }
   P2 (...) { /* definicija procedure/funkcije P2 */ }
   Pn (...) { /* definicija procedure/funkcije Pn */ }

   {
      /* inicijalizacija monitora */
   }

}
```

Konstrukcija monitora rešava problem sinhronizacije, odnosno dozvoljava da samo jedan proces bude aktivan u monitoru.

Na primer, jednostavan bafer koji se koristi prilikom rešavanju problema proizvođač-potrošač može se predstaviti monitorom koji čine:

- baferski prostor i pokazivači na prvo slobodno i prvo zauzeto mesto,

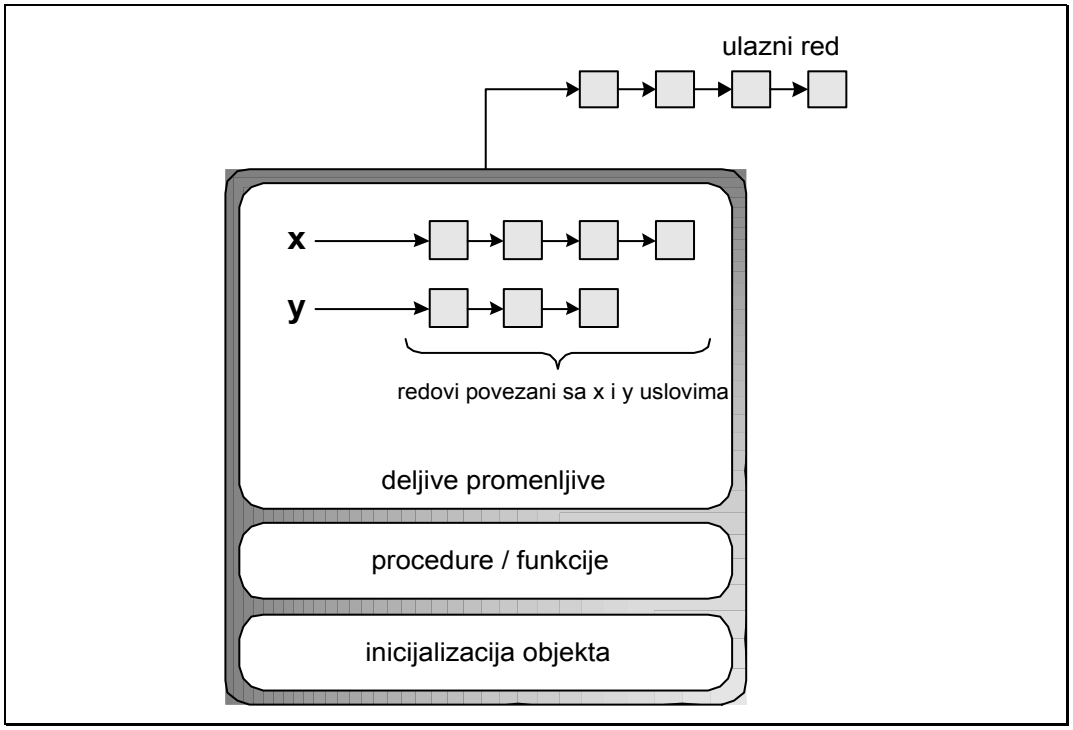
- procedure za dodavanje elemenata u bafer i uzimanje elemenata iz bafera i
- procedure za inicijalizaciju pokazivača na početak bafera.

U nekim slučajevima, ukoliko su potrebni dodatni sinhronizacioni mehanizmi, programer mora sam da definiše promenljive kojima će dodatni problem razrešiti. Ukoliko je dodatni sinhronizacioni mehanizam neka uslovna konstrukcija, definišu se tzv. uslovne promenljive:

```
condtion x, y;
```

Jedine opracije koje mogu da se izvode nad ovim promenljivim su *signal* i *wait*. Operacija x.wait() će blokirati proces koji je tu operaciju pozvao. Operacija x.signal će ukinuti blokadu sa tačno jednog blokiranog procesa. Ako nema blokiranih procesa, operacija *signal* ne radi ništa, za razliku od operacije *signal* na semaforu, koja uvek inkrementira vrednost.

Struktura monitora sa uslovnim varijablama prikazana je na slici 4.2.



Slika 4.2. *Monitor*

Rešićemo problem večere filozofa pomoću monitora. Najpre se definiše struktura podataka *state*, koja predstavlja trenutno stanje filozofa. Svaki filozof se može naći u

jednom od sledeća tri stanja: misli, gladan je ili jede. Kada misli ne treba mu ništa, kade je gladan pokušava da jede, a ne misli. Filozof može da jede samo ako njegovi susedi ne jedu. Dodatno, uvode se i uslovne promenljive *state[i]* koje će blokirati filozofa ukoliko je gladan, a ne može da dođe do obe viljuške.

Definisaćemo sledeći monitor *dp*:

```
monitor dp {

   enum {thinking, hungry, eating} state[5];
   condition self[5];

void pickup(int i) {
   state[i] = hungry;
   test[i];
   if (state[i] != eating) self[i].wait();
}

void release(int i) {
   state[i] = thinking;
   test((i+4) % 5);
   test((i+1) % 5);
}

void test(int i) {
   if ((state[i] == hungry) &&
   (state[(i+4)%5] != eating) &&
   (state[(i+1)%5] != eating)) {
      state[i] = eating;
      self[i].signal();
   }
}

void init() {
   for (int i = 0; i < 5; i++)
   state[i] = thinking;
   }
}
```

Najpre se u monitoru svi filozofi dovedu u stanje *think*, dakle svi misle, niko nije gladan, niko ne jede. Svaki filozof mora obaviti sledeće operacije:

```
dp.pickup(i);
   /* filozof jede */
dp.release(i)
```

Svaki filozof, pre nego što počne jesti, mora da pozove proceduru *pickup* iz monitora *dp*. U okviru te procedure, filozof će ili doći do obe viljuške ili će biti blokiran operacijom self [i].wait. Ako dođe do obe viljuške, filozof jede, a zatim ih spusti na sto. Posle toga, poziva se procedura release, koja oslobađa obe viljuške, a po potrebi i blokirane filozofe.

5

Niti

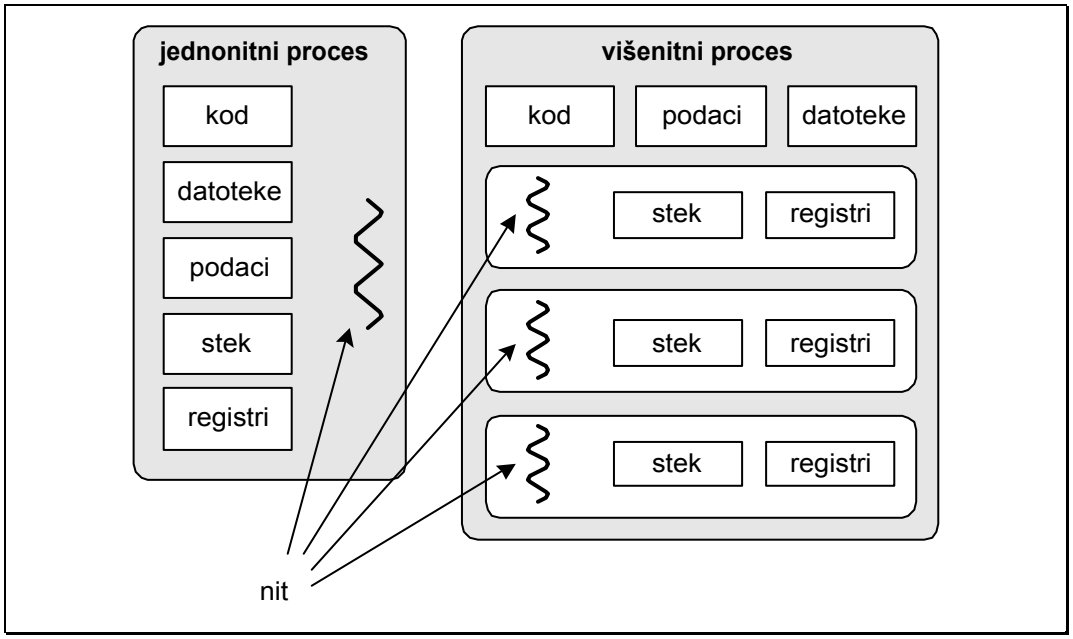
Sadržaj poglavlja:

5.1. Laki i teški procesi
5.2. Korisničke i kernelske niti - višenitni modeli
5.3. Osobine niti i rad sa nitima
5.4. Vrste niti

5.1. Laki i teški procesi

Niti (*threads*), odnosno laki procesi (LWP - *lightweight process*), predstavljaju osnovne celine za izvršavanje koda pod savremenim operativnim sistemima. Niti predstavljaju programsku celinu koja treba da obavi jedan posao. Niti (jedna ili više) pripadaju jednom tradicionalnom, odnosno teškom (*heavyweight*) procesu. U tradicionalnom kontekstu, jedan težak proces ima svoj programski brojač i druge procesorske registre, tri memorijske sekcije (kod, podaci i stek) i ulazno-izlazne resurse kao što su datoteke i uređaji. Niti kao laki procesi i delovi jednog istog procesa, imaju svoje unikatne resurse i zajedničke resurse sa ostalim nitima istog procesa. Od unikatnih resursa imaju poseban identifikator niti (*thread ID*), posebnu vrednost programskog brojača, kao i vrednosti drugih registara procesora i poseban stek. Sve ostalo (kod sekcija, sekcija podataka, otvorene datoteke, signali) su zajednički resursi za sve niti jednog istog procesa. Uz pomoć ovog savremenog koncepta, proces sa svojim nitima može obavljati više poslova simultano, pri čemu je prebacivanje konteksta sa jedne niti na drugu daleko brže i efikasnije od prebacivanje konteksta između teških procesa (čuvaju se i rekontruišu samo registri procesora i stek - sve

ostalo je zajedničko). Na slici 5.1. je ilustrovana razlika između tradicionalnog jednonitnog procesa (*singlethreaded*) i višenitnog procesa (*multithreaded*).



Slika 5.1. *Jednonitni i višenitni procesi*

Prednosti upotrebe niti

Osim efikasnosti i veće brzine odziva, motivacija za upotrebu niti leži u zahtevima savremenih sistema zasnovanih na Internetu, gde na primer jednom *web-serveru* stiže veliki broj različitih zahteva od korisnika. Mnogi moderni softverski paketi su višenitni - programi za prikazivanje *web* stranica (*web browser*) po pravilu imaju minimalno dve niti, od kojih jedna prikazuje hipertekst na ekranu, dok druga prima podatke sa mreže. Tekst procesori imaju jednu nit za prikazivanje grafike, drugu nit za unos karaktera sa tastature, treću nit za izvršavanje provere jezičke i gramatičke ispravnosti unetog teksta, itd.

Korišćenje višenitnog koncepta ima svoje prednosti, koje se ogledaju u sledećem:

[1] Smanjenje vremena odziva.

Višenitna tehnika omogućava interaktivnim aplikacijama da nastave rad, čak i u slučaju da je deo programa blokiran ili da izvršava neku dugotrajnu operaciju. Na primer višenitni web prikazivač, može da nastavi interakciju sa korisnikom na jednoj niti, dok druga nit simultano izvršava punjenje neke velike slike sa Interneta.

[2] Efikasno deljenje resursa.

Kao što je definisano, niti dele memoriju i sve ostale resurse koji pripadaju istom procesu. Deljenje kod segmenta veoma je korisno, jer omogućava da jedna aplikacija ima veliki broj niti koje se izvršavaju koristeći isti adresni prostor.

[3] Ekonomičnost.

Ogleda se u deljenju prostora i resursa i uštedi vremena koje, takođe, drastično utiče na performanse. Niti se kreiraju mnogo brže od procesa, a prebacivanje konteksta između niti je takođe brže od prebacivanja konteksta između procesa. Na primer, kreiranje procesa na sistemu Solaris 2 je 30 puta sporije od kreiranja niti, dok je prebacivanje konteksta 5 puta sporije.

[4] Bolje iskorišćenje višeprocesorske arhitekture.

Bilo koje niti se mogu istovremeno izvršavati, svaka na različitom procesoru. Na primer, ukoliko imamo samo jedan proces u sistemu, jednonitna koncepcija dozvoljava izvršavanje tog procesa samo na jednom procesoru, dok su ostali procesori neiskorišćeni.

5.2. Korisničke i kernelske niti - višenitni modeli

Kako jedan proces može da se izvršava i u koriničkom i u kernelskom režimu rada, potrebno je obezbediti istu podršku i na korisničkom nivou za korisničke niti (*user threads*), i na nivou kernela za kernelske niti (*kernel threads*).

Podrška za korisničke niti se realizuje preko biblioteke za rad sa korisničkim nitima (*user thread library*). Ova biblioteka obezbeđuje podršku za kreiranje niti, raspored izvršavanja niti i upravljanje nitima, ali bez uticaja kernela. Najznačajnije vrste korisničkih niti su: POSIX Pthreads, Mach C-threads i Solaris UI-threads.

Kernelske niti imaju podršku direktno od operativnog sistema. Konkretno, kernel izvršava operacije kreiranja niti, raspoređivanja izvršavanja niti i upravljanja nitima u prostoru kernela. Kernelske niti se po pravilu sporije kreiraju, a upravljanje njima unosi veće vremensko premašenje u odnosu na upravljanje korisničkim nitima, ali generalno kernel ozbilnije i efikasnije upravlja svojim nitima nego biblioteka sa rad sa korisničkim nitima.

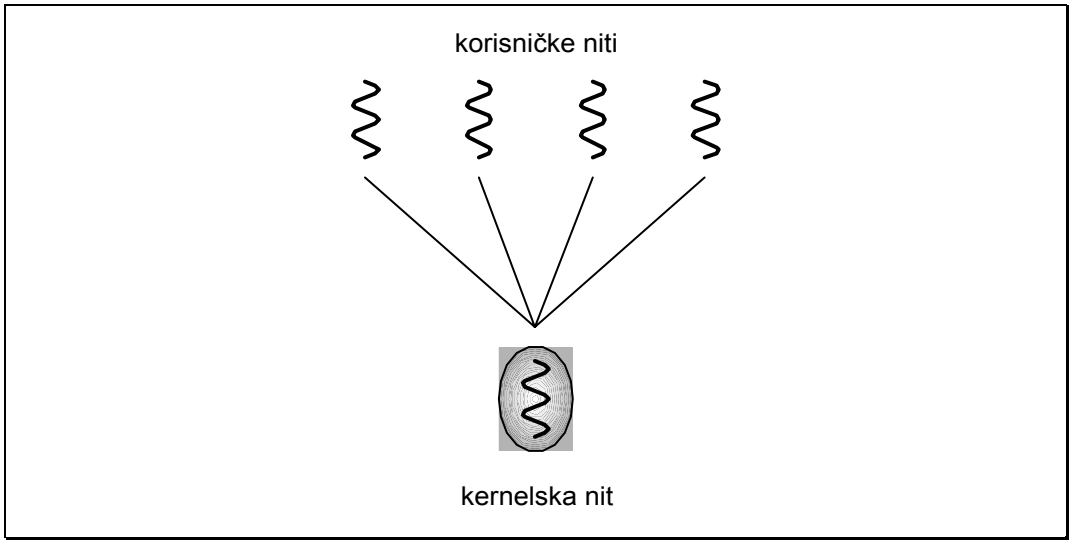
Mnogi sistemi podržavaju obe vrste niti, a zavisno od toga kako se korisničke niti mapiraju u kerneleske, postoje tri glavne koncepcije, odnosno tri višenitna (*multithreading*) modela:

- model više u jednu (*many-to-one*),
- model jedna u jednu (*one-to-one*) i

- model više u više (*many-to-many*).

Višenitni model više u jednu

Kod ovog modela više korisničkih niti se mapira u jednu kernelsku nit, kao što je prikazano na slici 5.2.

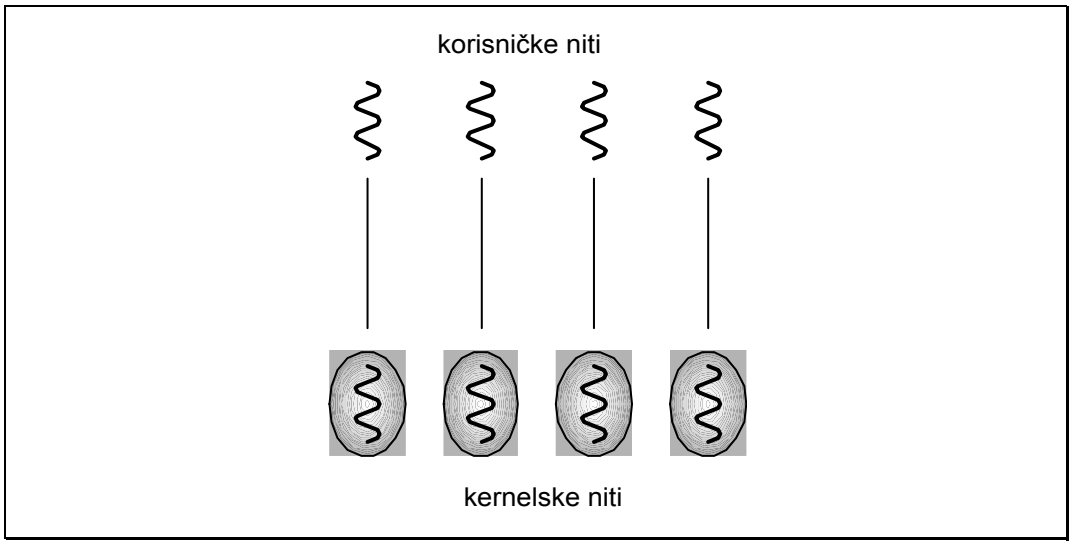


Slika 5.2. *Višenitni model više u jednu*

Upravljanje nitima se odvija na korisničkom nivou i može biti efikasno, ali ima nedostatke. Ceo proces, odnosno sve njegove niti će se blokirati ako neka od niti obavi bilo koji blokirajući sistemski poziv. Takođe, s obzirom na to da samo jedna nit može pristupati kernelu u jednom trenutku, u kernelskom modu neće moći da se iskoristi višeprocesorska arhitektura. Ovaj model koristi sistem operativni sistem Solaris 2 i biblioteka Green threads.

Višenitni model jedna u jednu

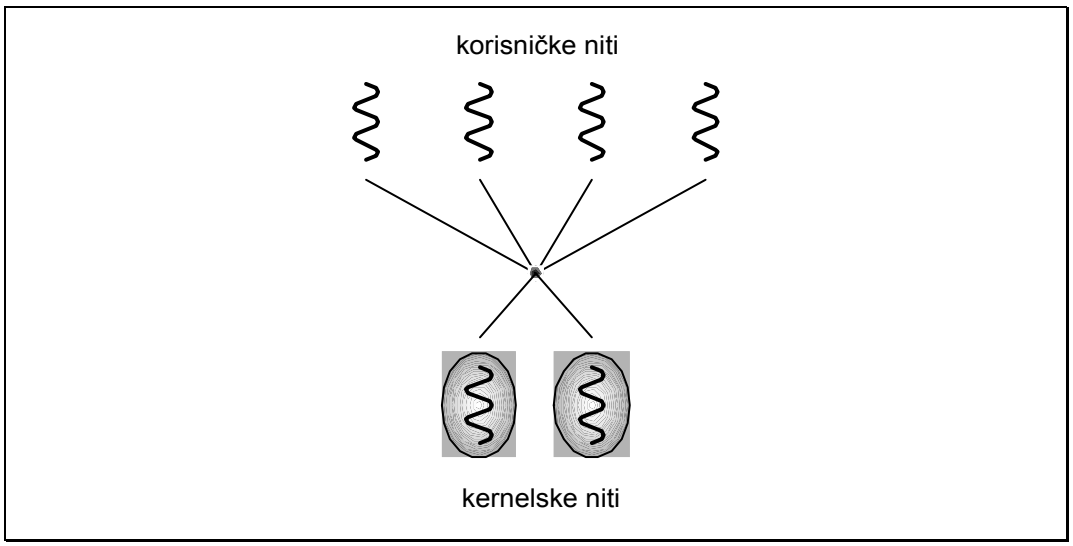
U ovom modelu svaka korisnička nit se mapira u jednu kernelsku nit, što je prikazano na slici 5.3. Ovaj model obezbeđuje mnogo bolje konkurentno izvršavanje niti, dozvoljavajući da druge niti nastave aktivnosti u slučaju kada jedna nit obavi blokirajući sistemski poziv. Takođe se omogućava da se više kernelskih niti izvršavaju paralelno na višeprocesorskoj arhitekturi. Kako se velika količina vremena gubi na kreiranje i održavanje kernelskih niti, mnogi operativni sistemi ograničavaju maksimalan broj kerenlskih niti. Prikazani model je karakterističan za operativne sisteme Windows NT, Windows 2000/XP i OS/2.



Slika 5.3. *Višenitni model jedna u jednu*

Višenitni model više u više

U ovom modelu više korisničkih niti se mapira u manji ili isti broj kernelskih niti, pri čemu mapiranje zavisi od operativnog sistema a naročito od broja procesora.



Slika 5.4. *Višenitni model više u više*

Ovo je najkompleksniji i najkvalitetniji model jer može da se prilagodi i aplikaciji i procesorskom okruženju. Ovaj višenitni model je prisutan na sistemima Solaris-2, IRIX, HP-UX i True64 UNIX.

5.3. Osobine niti i rad sa nitima

U višenitnoj arhitekturi menja se semantika sistemskih poziva za kreiranje i rad sa procesima. Mora postojati poziv za kreiranje teškog procesa koji će dupliciratti sve niti i njihove adresne prostore, kao i poziv za kreiranje jedne jedine niti.

Prekidanje niti

Prekidanje niti je aktivnost koja prekida izvršavnje niti pre njenog prirodnog završetka. Nit koja će biti prekinuta obično se naziva ciljna nit (*target thread*). Prekidanje ciljne niti može se dogoditi po dva scenarija:

- asinhroni prekid (*asynchronous cancellation*): aktivnost niti se odmah prekida bez obzira na stanje u kome se nalazi i
- odloženi prekid (*deferred cancellation*): ciljna nit proverava periodično da li treba da prekine aktivnost i ako treba to će učiniti u povoljnom trenutku kad obavi neku celinu.

Iako je odloženo prekidanje novije i kvalitetnije, takođe je i teže za implemetaciju, tako da većina operativnih sistema primenjuje asinhrono prekidanje.

Upravljanje signalima

Signal se koristi na UNIX sistemima da označi procesu da se desio neki događaj. Signali obično služe za prekidanje, suspedovanje i ponovno pokretanje procesa ili niti. Proces može primiti signal sinhrono ili asinhrono, zavisno od razloga zbog koga he signal upućen. Bez obzira kako je signal generisan, moraju se poštovati sledeća pravila:

- signal se generiše kao posledica dešavanja nekog događaja,
- generisani signal se upućuje procesu i
- kada proces primi signal, mora se obaviti određena akcija, odnosno rutina za obradu primljenog signala (*signal handling*). Akcija se ne obavlja ukoliko proces ignoriše signal.

Signal upućen nekom višenitnom procesu zavisno od situacije može se proslediti:

- samo onoj niti kojoj je namenjen,

- svim nitima procesa,
- samo nekim nitima procesa i
- samo onoj niti procesa čija je namena da prima i obrađuje sve signale za proces.

Gomila niti

Ideja iza gomile niti (*thread pool*) je sledeća: kada se kreira proces, napravi se više niti i postave se u gomulu u kojoj svaka nit čeka na neki posao. Ovo se tipično realizuje na serverima na kojima se kreiraju procesi sa gomilom niti. Kada server dobije neki zahtev, on taj zahtev dodeli nekoj niti na obradu, koja obavi posao i ponovo se vraća na gomilu, gde čeka novi posao. Naravno, ako su sve niti iz gomile zauzete, server će sačekati da neka postane slobodna. Prednosti gomile niti su:

- brže servisiranje zahteva - ne čeka se na kreiranje novih niti i
- limitiranje maksimalnog broja niti koje mogu postojati na serveru u jednom trenutku, čime se sprečava degradacija performansi sistema zbog prevelikog broja niti.

Broj niti u gomili se heuristički određuje na osnovu broja procesora, količine memorije i broja očekivanih klijentskih zahteva.

Podaci specifični za nit

Već je rečeno da niti dele sekciju podataka u jednom procesu. Međutim, jedna nit iz specifičnih potreba može kreirati privatnu kopiju delova sekcije podaka - ti podaci se nazivaju podacima specifičnim za nit.

5.4. Vrste niti

U nastavku teksta ukratko su implementacije višenitnih koncepata na poznatijim operativnim sistemima.

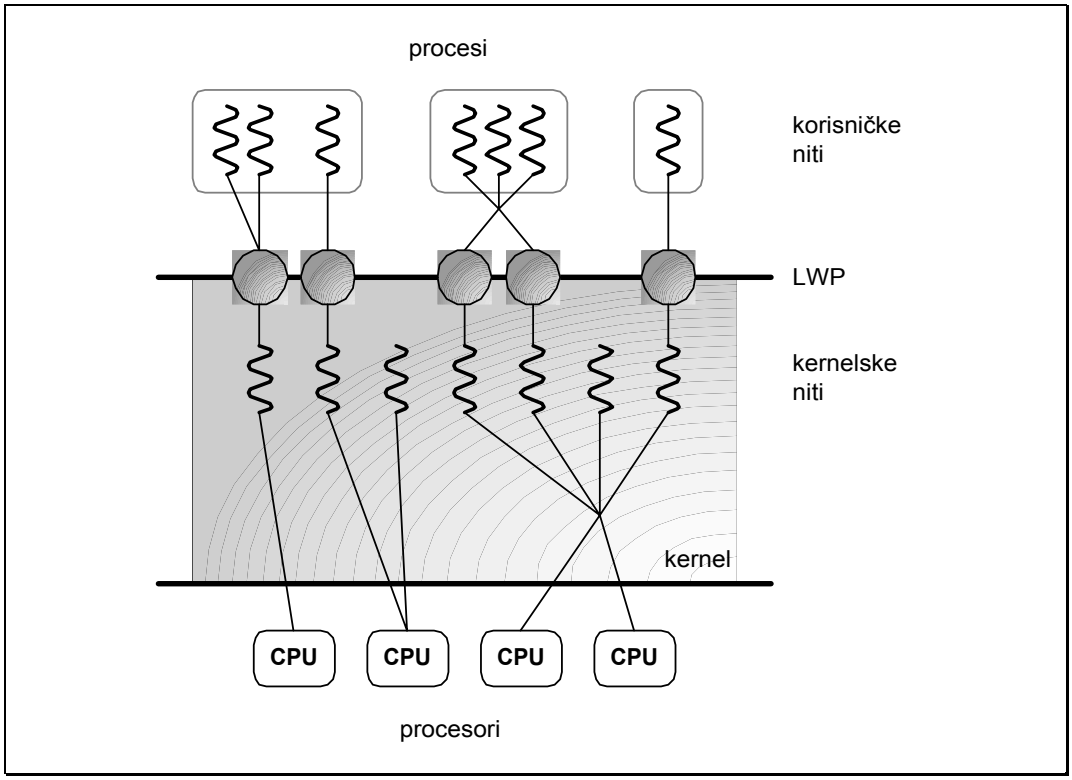
Pthreads

Pthreads upućuju na POSIX standard (IEEE 1003.1.c) koji definiše pravila za kreiranje niti i njihovu sinhronizaciju. Ovaj standard daje specifikaciju za niti, a realizacija je potpuno slobodna i prepuštena samom operativnom sistemu. Na korisničkom nivou postoji API (*Aplication Programing Interface*) i Pthread biblioteka,

koja programeru omogućava kreiranje niti. Podrška za Pthreads postoji u UNIX, ali ne i u MS Windows sistemima.

Solaris 2 niti

Solaris 2 je verzija UNIX sistema koja podržava i korisničke i kernelske niti. Solaris 2 podržava Phtreads API za korisničke niti. Solaris 2 definiše međunivo za niti koji spaja korisničke i kernelske niti (model više u više) na sledeći način: između korisničke i kernelske niti nalazi se laki proces (LWP) koji ih spaja, pri čemu svakom LWP odgovara tačno jedna kernelska nit. Solaris 2 niti su prikazane na slici 5.5.



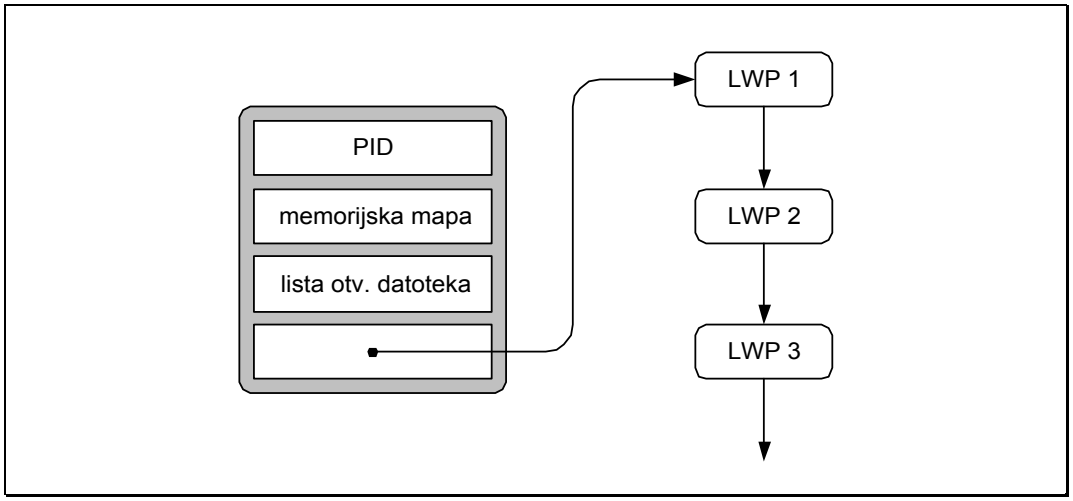
Slika 5.5. *Solaris 2 višenitni model*

Sledeće strukture podataka se vezuju za korisničke i kernelske niti i lake procese koji ih povezuju:

- Svaka korisnička niti ima svoj identifikator, registarski set, stek i prioritet. Nijedna od ovih struktura nije kernelska - sve pripadaju korisničkom adresnom prostoru.

- Svaki LWP ima registarski set za korisničku nit za koju je vezan, kao i memorijske i statističke strukture koje se nalaze u kernelskom adresnom prostoru.
- Svaka kernelska niti ima kopiju registara, ukazivač na LWP, prioritet i stek.

Sam proces na Solaris 2 sistemu ima kontrolni blok koji sadrži identifikator procesa (PID), memorijsku mapu, prioritet, listu ostvorenih datoteka i ukazivač na listu kernelskih niti povezanih sa nitima samog procesa, kao što je prikazano na slici 5.6.



Slika 5.6. *Solaris 2 proces*

Windows 2000 niti

Windows 2000 primenuje Win32 API. Svaki proces Windows aplikacija može imati jednu ili više niti. Windows 2000 koristi višenitni model jedna u jednu. Međutim, postoji i *fiber* biblioteka koja omogućava primenu šeme više u više. Svaka nit na Windows 2000 sistemu deli virtuleni adresni prostor sa svojim procesom. U svakoj niti se nalaze sledeće informacije:

- identifikator niti,
- registarski set,
- korisnički stek i kernelski stek i
- privatni podaci

Linux niti

Linux sistemi obezbeđuju tehnike za rad sa nitima, počev od kernela 2.2. Umesto termina nit, Linux koristi termin posao (*task*). Osim sistemskog poziva *fork*, koji služi za kreiranje novih procesa, postoji i sistemski poziv *clone* koji praktično kreira Linux niti. I *fork* i *clone* kreiraju decu procese od roditelj procesa. Osnovna razlika između ovih sistemskih poziva je u adresnom prostoru novih procesa koje kreiraju:

Sistemski poziv *fork* kreira za dete proces kopiju adresnog prostora procesa roditelja, dok kod *clone* sistemskog poziva dva procesa postaju niti koje dele isti adresni prostor.

6

Zastoj

Sadržaj poglavlja:

6.1. Sistemski model i osobine zastoja
6.2. Metode upravljanja zastojem - prevencija
6.3. Izbegavanje zastoja
6.4. Detekcija i oporavak od zastoja

6.1. Zastoj

U višeprocesnoj okolini više procesa se mogu međusobno takmičiti za konačan broj resursa. Kada proces zahteva resurs, a resurs nije raposloživ, proces ulazi u stanje čekanja na resurs (WAIT) i postaje blokiran. Blokirani proces može zauvek ostati u tom stanju ukoliko potrebni resurs ostane neraspoloživ. Ova situacija je moguća ukoliko je resurs prethodno dodeljen na korišćenje drugom procesu koji u toku vremena, takođe, prelazi u stanje čekanja na drugi resurs koji nije raspoloživ. Takva situacija naziva se zastoj, odnosno blokada (*deadlock, deadly embrace*). Pojava je sasvim moguća, osobito pri radu sa savremenim višenitnim procesima i treba je izbeći. Sistem koji je doveden u stanje zastoja se mora oporaviti.

Sistemski model

Sistem se sastoji od konačnog broja resusra koje koriste konkurentni procesi (memorijski prostor, procesor, datoteke, ulazno-izlazni uređaji). Svaki resurs se može

sastojati od jedne ili više identičnih instanci. Ako, na primer, server ima dva procesora, tada je procesor resurs sa dve instance. Ako resurs ima više instanci, a proces zahteva jednu, alokacija bilo koje slobodne instance će zadovoljiti proces.

Proces mora zahtevati resurs pre korišćenja, a takođe mora otpustiti resurs nakon korišćenja. Jedan proces može zahtevati više različitih resursa, a teorijski, sve moguće resurse koje poseduje sistem. U normalnom režimu rada, proces može da koristi resurs samo na sledeći način:

[1] Zahtev (*request*). U ovoj fazi, proces zahteva resurs. Ako zahtev za dodelom resursa ne može da se ispuni trenutno, proces mora da čeka dok se resurs ne oslobodi.

[2] Korišćenje (*use*). U ovoj fazi, proces je dobio resurs i može ga slobodno koristiti.

[3] Oslobađanje (*release*). Nakon korišćenja resursa, proces mora da oslobodi resurs.

Skup procesa je u zastoju, ako svaki proces drži resurs, a traži na korišćenje novi resurs koga drži neki drugi proces iz skupa. U toj sitaciji niko ne oslobađa svoje resurse, a traži nove - skup tih procesa postaje zaglavljen. Na primer, sistem ima jedan štampač i jednu traku. Proces P1 je dobio traku, a P2 je dobio štampač. Ukoliko P1 zatraži štampač a da prethodno nije oslobodio traku, a P2 zatraži traku a da prethodno nije oslobodio štampač, tada procesi P1 i P2 ulaze u zastoj.

Procesi koji su u zastoju, nikada ne završavaju svoje aktivnosti i nikada ne oslobađaju resurse.

Uslovi pod kojima nastupa zastoj

Zastoj nastupa ako su istovremeno ispunjena sledeća četiri uslova:

[1] Međusobno isključenje.

Samo jedan proces u jednom trenutku može koristiti resurs ili jednu njegovu instancu. Drugi proces koji zahteva taj isti resurs, odnosno instancu, mora da čeka dok se resurs ne oslobodi.

[2] Nema pretpražnjenja.

Resurs se ne može nasilno oduzeti i predati drugom procesu, sve dok proces koji ga koristi ne završi posao i ne oslobodi resurs.

[3] Uslov zadržavanja resursa i čekanja na drugi (*hold and wait*).

Proces mora da drži jedan resurs u korišćenju, a da istovremeno čeka na dobijanje resursa koga koristi neki drugi proces.

[4] Kružno čekanje (*Circular wait*).

Mora postojati skup procesa {*P*0, *P*1, …, *P*0}, koji čekaju na resurse u sledećem kružnom poretku: P0 čeka na resurs koga drži proces P1, P1 čeka na resurs koga drži proces P2, …, Pn–1 čeka na resurs koga drži proces Pn i Pn čeka na resurs koga drži proces P0.

Naglasimo još jedanput da zastoj nastupa samo pod uslovom da su sva četiri uslova istovremeno ispunjena, kao i da ova četiri uslova nisu sasvim nezavisna - na primer, uslov 2 je sadržan u uslovu 4.

Graf dodeljenih resursa

Situacije zastoja se mogu prikazati preciznije pomoću grafova dodeljenih resursa (*resource allocation graph*). Graf se sastoji iz skupa objekata i skupa strelica E. Skup objekata se sastoji iz dva skupa:

- skup svih aktivnih procesa u sistemu, P={P1, P2.....Pn} i
- skup svih raspoloživih resursa R={R1, R2.....Rm}.

Strelica zahteva (Pi->Rj) ukazuje da proces Pi zahteva jednu instancu resursa Rj i da trenutmo čeka na nju. Strelica alokacije (Rj->Pi) ukazuje na to da je resurs Rj dodeljen procesu Pi.

Graf dodeljenih resursa se konstruiše tako što strelica zahteva ubacuje u graf uvek kada proces traži resurs, a strelica alokacije uvek kada se resurs dodeli procesu.

Sa grafa se lako može uočiti stanje zastoja:

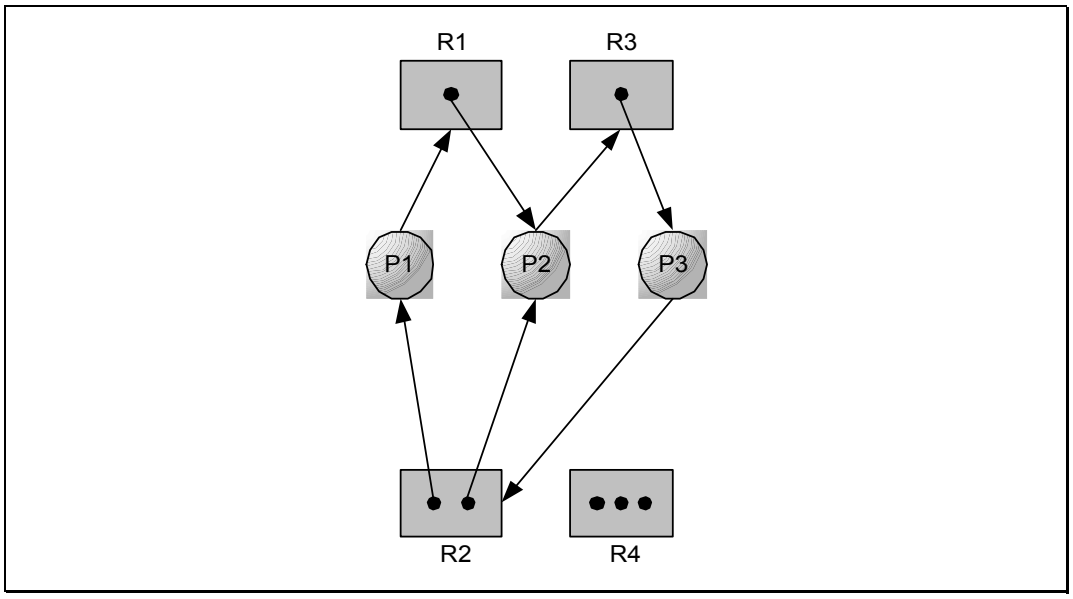
- Ako graf ne sadrži kružni tok, zastoja sigurno nema.
- Ako graf sadrži najmanje jedan kružni tok, zastoji su mogući. Ako svi resursi koji pripadaju kružnom toku sadrže tačno jednu instancu, zastoj se dogodio. Ako resursi u kružnom toku sadrže više instanci, može se dogoditi da zastoja i nema.

Analizirajmo graf na slici 6.1.

Skupovi objekata su P={P1, P2, P3} i R={R1, R2, R3, R4}. Pri tome, resursi R1, R2, R3 i R4 redom imaju 1,2,1 i 3 instance.

Skup strelica je E={P1->R1, P2->R3, R1->P2, R2->P2, R2->P1, R3->P3, P3->R2}.

Stanja procesa su sledeća: P1 koristi jednu instancu resursa R2, a čeka na jednu instancu resursa R1. P2 koristi jednu instancu resursa R1 i R2, a čeka na jednu instancu resursa R3. P3 koristi jednu instancu resursa R3, a a čeka na jednu instancu resursa R2



Slika 6.1. *Graf dodeljenih resursa - uočavanje kružnih tokova*

Na ovom grafu postoje dve kružne putanje koje mogu izazvati zastoj:

[1] P1->R1->P2->R3->P3->R2->P1

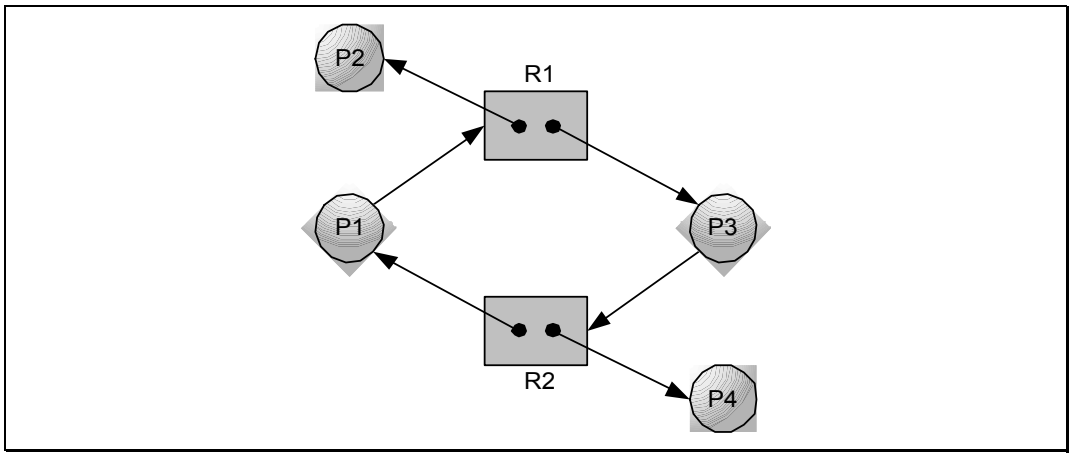
[2] P2->R3->P3->R2->P2

Zaključak je da su procesi P1, P2 i P2 najverovatnije u zastoju, jer

- proces P2 čeka na resurs R3, koga drži proces P3,
- proces P3 traži resurs R2, koga drže procesi P1 i P2,
- P2 je blokiran sa procesom P3 oko resursa R3 i
- proces P1, koji bi tada jedini mogao da oslobodi resurs R2, je blokiran zbog procesa P2 koji drži resurs R1.

Međutim, kružni tok na grafu ne znači obavezno zastoj. Na grafu sa slike 6.2. može se uočiti kružni tok P1->R1->P3->R2->P1. Međutim, zastoja nema jer oba resursa imaju više instanci. Na primer, proces P4 može da završi svoj posao i da oslobodi jednu instancu resursa R2, koja se može dodeliti procesu P3, čime se prekida krug i eliminiše zastoj.

Zaključci su: ako graf dodeljenih resursa nema kružnih tokova, sistem sigurno nije u stanju zastoja. Ako kružni tokovi postoje, sistem može, ali i ne mora biti u stanju zastoja, što zavisi od broja instanci resursa u kružnom toku.



Slika 6.2 *Graf dodele resursa - kružni tok bez zastoja*

6.2. Metode upravljanja zastojem - prevencija

Principijalno, problem upravljanja zastojem može se rešavati na tri načina:

- prevencijom ili izbegavanjem zastoja (*deadlock prevention and avoidance*), odnosno metodama koje obezbeđuju da sistem nikada ne uđe u stanje zastoja,
- detekcijom i oporavkom (*deadlock detection and recovery*), odnosno metodama koje dozvoljavaju sistemu da uđe u stanje zastoja, ali to stanje detektuju i oporavljaju sistem i
- ignorisanjem problema zastoja. Većina opertivnih sistema, uključujući i UNIX, se pretvara da se te pojave ne dešavaju i ne koriste prethodne dve metode.

Prevencija zastoja

Kao što je već rečeno, zastoj se dešava ako su četiri uslova istovremeno ispunjena. Metode prevencije obezbeđuju neispunjenje najmanje jednog od četiri uslova, potrebnih za pojavu zastoja. Analizirajmo sva četiri uslova:

[1] Međusobno isključenje.

Međusobno isključenje je obavezan uslov za sve nedeljive resurse, odnosno resurse može da koristi samo jedan proces u jednom vremenskom trenutku. Štampač je, na primer, nedeljiv resurs. Međutim, princip međusobnog isključenja ne treba primenjivati na deljive resurse, kao što su, na primer, datoteke koje se ne menjaju. Jedan deljivi resurs može koristiti veći broj procesa istovremeno. Prema

tome, jedan metod prevencije zastoja je izbegavanje uslova međusobnog isključenja za sve deljive resurse. Za nedeljive resurse to nije moguće.

[2] Uslov zadržavanja resursa i čekanja na drugi.

Ova situacija se može izbeći ukoliko: (a) svaki proces mora da traži i alocira sve svoje resurse pre početka izvršavanja ili (b) proces može da traži resurs samo pod uslovom da ne drži ni jedan drugi, što znači da proces može koristiti samo jedan resurs u jednom trenutku (čime se značajno smanjuje korišćenje ulazno-izlaznih resursa.

[3] Nema pretpražnjenja.

Prema ovom uslovu, samo proces koji koristi resurs može taj resurs osloboditi - resurs se ne može nasilno oduzeti i predati drugom procesu. Ovaj uslov se može poništiti korišćenjem sledećeg protokola: proces koji traži novi resurs, koji trenutno ne može da dobije na korišćenje, otpušta sve zauzete resurse, a zatim prelazi u stanje WAIT. Proces može preći u stanje READY tek kada povrati sve resurse koje je posedovao i resurse koje je tražio na korišćenje, ali ih nije mogao dobiti. Dakle, proces ne može preći u stanje WAIT sa zauzetim resursima. Alterantivno, ako proces zahteva neke resurse, najpre se proverava da li su isti raspoloživi i ako jesu dodeljuju se procesu. Ako resursi nisu raspoloživi, proverava se da li su dodeljeni nekom drugom procesu koji, takođe, čeka na dodatne resurse. Ukoliko jesu, resursi se nasilno oduzimaju od blokiranog procesa i dodeljuju procesu koji ih traži.

[4] Kružno čekanje.

Cirkularno čekanje se može izbeći na sledeći način: svakom resursu se dodeli broj N iz skupa prirodnih brojeva N=F(Ri), a procesi mogu zahtevati resurse u strogo rastućem redu. Na primer, ako je proces zahtevao resurs Ri, nakon toga proces može zahtevati resurs Rj samo ako je F(Rj)>F(Ri), tako da poslednji proces Pn u nizu ne može da se vrati unazad i traži resurs koga drži prvi proces P0. Na ovaj način se eliminiše kružno čekanje.

6.3. Izbegavanje zastoja

Izbegavanje zastoja je moguće ukoliko operativni sistem ima unapred poznate informacije o procesima i resurima koje će oni zahtevati prilikom svog izvršavanja.

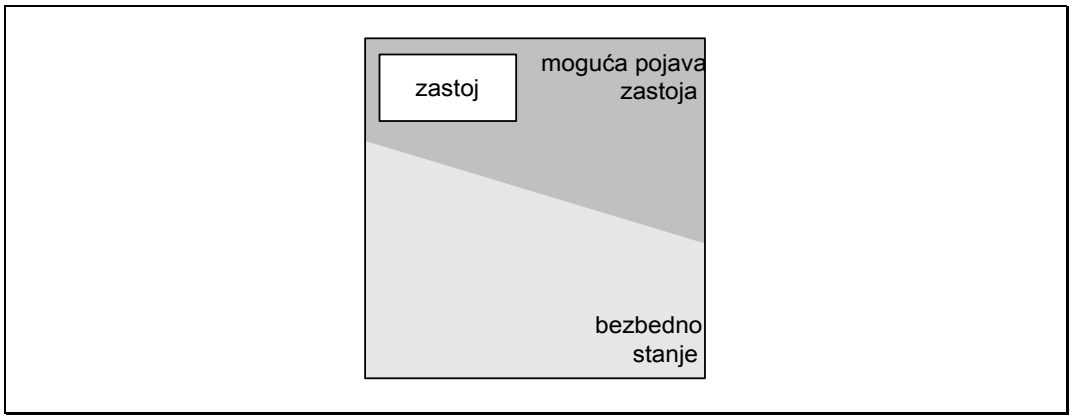
Za razliku od prethodnog algoritma koji uvodi brojna ograničenja procesima u zahtevanju resura, algoritam za izbegavanje zastoja zahteva od procesa dodatne informacije o tome koji će mu svi resursi biti potrebni i u kom redosledu. Sistem koji zna šta procesi žele može da napravi redosled opsluživanja zahteva, odnosno redosled uvođenja procesa u stanje čekanja na resurs, tako da se izbegne zastoj. Svaki novi

zahtev za resurse nateraće sistem da analizira resurse koji su trenutno raspoloživi i resurse koji su trenutno dodeljeni, kao i buduće zahteve svih procesa. Nakon toga sistem odlučuje da li će resurs dodeliti procesu koji je izdao tekući zahtev, ili će uvesti taj proces u stanje WAIT, kako bi se izbegao zastoj.

Najprostija šema izbegavanja zastoja zahteva od svakog procesa da deklariše najveći broj potrebnih resursa svakog tipa koji je procesu potreban. Sa informacijom o maksimalnim potrebnim resursima moguće je konstruisati algoritam koji će sprečiti stanje zastoja. Taj algoritam dinamički ispituje trenutno stanje dodeljenih resursa, tako da osigura da sistem nikada ne uđe u stanje cirkularnog čekanja. Pod stanjem dodeljenih resursa podrazumeva se stanje definisano brojem raspoloživih, dodeljenih i maksimalno traženih resursa u jednom trenutku vremena.

Bezbedno stanje

Kada proces traži neki resurs, sistem mora da proceni da li će dodela tog resursa ostaviti sistem u bezbednom stanju (*safe state*). Stanje je bezbedno ako sistem može može alocirati resurse svakom procesu u nekom poretku i još uvek izbeći zastoj. Bezbedno stanje je stanje bez zastoja. Stanje koje nije bezbedno ne mora biti zastoj, ali može težiti ka zastoju - u tom stanju, procesi mogu lako dovesti sistem u zastoj.



Slika 6.3 *Bezbedno stanje*

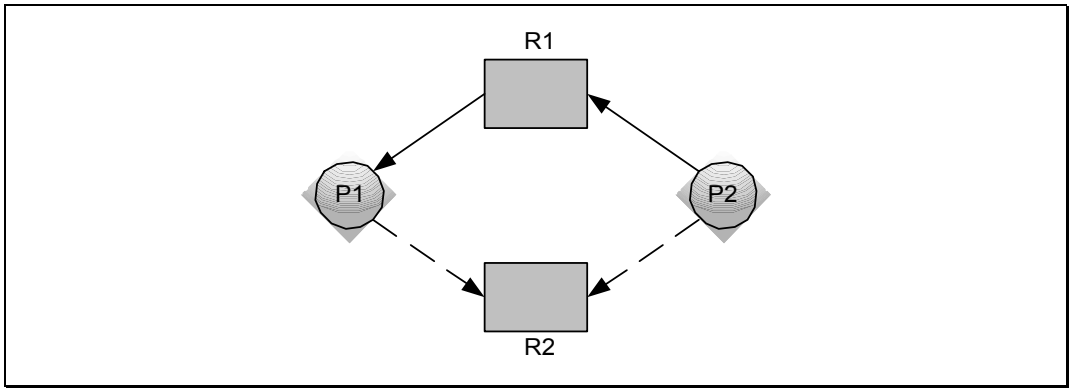
Formalnije i preciznije, sistem je u bezbednom stanju ako postoji bezbedna sekvenca svih procesa. Sekvenca <P1, P2, ..., Pn> je bezbedna ako za svaki proces Pi, u svakom ternutku, resursi koje Pi može još tražiti, mogu da se zadovolje iz trenutno raspoloživih resursa, uključujući resurse koji pripadaju svakom procesu Pj, za koje je j<i, odnosno za sve procese koji su pokrenuti pre procesa Pi. Ako resursi koje traži proces Pi nisu trenutno raspoloživi, tada će Pi čekati da prethodni procesi Pj završe svoje poslove i oslobode resurse.

Na primer, posmatrajmo sistem sa 12 traka i tri procesa. Neka je u trenutku t0 proces P0 dobio 5 traka, a procesi P1 i P2 po dve, tako da su tri trake ostale slobodne. Neka su maksimalne i trenutne potrebe procesa P0, P1 i P2 redom 10,4 i 9, odnosno 5,2 i 2 trake.

Sekvenca <P1, P0, P2> gde je P1 prvi proces koji se pojavio pa potom P0, a na kraju P2, nalazi se u stabilnom stanju. P1 kao prvi proces može zadovoljiti svoje makimalne potrebe, a to su još dve trake, da završi posao i da oslobodi sve četiri trake, tako da potom P0 može da zadovolji svoje maksimalne potrebe i da oslobodi svih svojih 10 traka, što će zadovoljiti proces P2 kao poslednji proces. Ali da je P2, na primer, uzeo u trenutku t0 jednu traku više, sistem bi bio u nestabilnom stanju. P1 bi završio svoje aktivnosti i vratio 4 trake, što ne bi bilo dovoljno da P0 zadovolji svoje maksimalne potrebe (5 traka) - ako P0 zatraži svoju maksimalnu potrebu za resursima, doći će do zastoja.

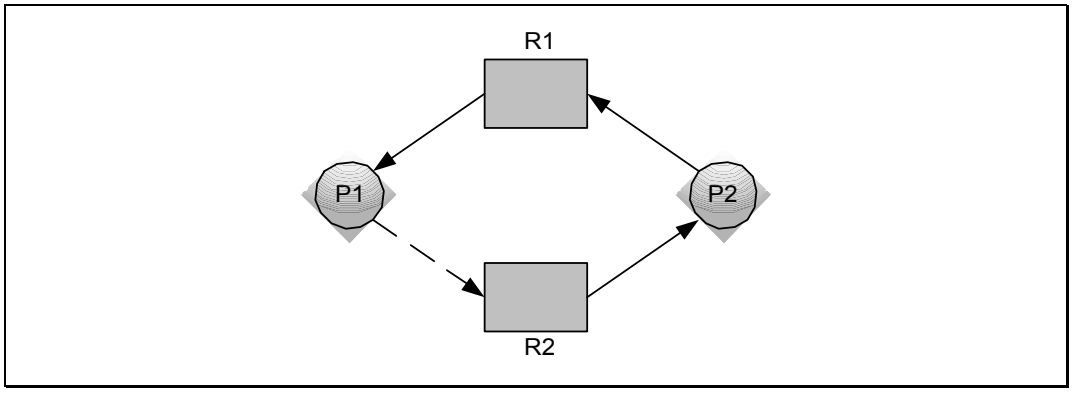
Graf dodele resursa za izbegavanje zastoja

U dodatku postojećih strelica, kao što su strelica zatheva i strelica dodele, uveščemo i treću kategoriju strelica, a to je strelica mogućih zahteva (*claim edge*). Ova strelica (Pi->Rj) znači da proces može zahtevati resurs Rj u nekom budućem trenutku. U grafu ćemo je predstaviti isprekidanom linijom. U trenutku u kome proces zahteva resurs, mogući zahtev preći će u realni zahtev za resurs, a strelica mogućeg zahteva biće zamenjena strelicom zahteva. Slično, proces oslobodi resurs, strelica dodele Rj->Pi prelazi u strelicu mogućih zahteva Pi->Rj. Naravno, svi mogući zahtevi za korišćenjem resursa moraju biti poznati pre nego što proces otpočne izvršavanje. To će omogučiti da se nacrta graf sa mogućim zahtevima procesa, a na bazi tog grafa se odlučuje da li će sistem dozvoliti zahtev ili ne. Ukoliko se ispunjenjem zaheva formiraju kružni tokovi, sistem se može dovesti u nebezbedno stanje pa se procesu ne ispunjava zahtev. Ova situacija je ilustrovana slikom 6.4.



Slika 6.4. *Graf dodele resursa za izbegavanje zastoja*

Vidimo da je resurs R2 slobodan, ali da postoje mogući zahtevi od procesa P1 i P2. Ukoliko bi proces P2 izdao zahtev za korišćenjem R2, sistem to ne bi smeo da dozvoli, jer bi se tada pojavio kružni tok. Ako i P1 zatraži R2, dolazi do zastoja, što je ilustrovano na slici 6.6.



Slika 6.5. *Moguća pojava zastoja*

Bankarski algoritam

Prethodni algoritam sa mogućim strelicama nije podesan za resurse koji imaju više instanci. Algoritam za izbegavanje zastoja koji se primenjuje u tim situacijama poznat je pod imenom bankarski algoritam (*Banker's Algorithm*). Algoritam je dobio takvo ime jer je primenljiv u bankarstvu, odnosno omogućava da bankar koji izdaje keš može da zadovolji sve potencijalne zahteve svojih štediša.

Bankarski algoritam se može primeniti ukoliko su ispunjeni sledeći uslovi:

- resursi imaju uglavnom više instanci,
- svaki proces unapred mora deklarisati najveći broj instanci svakog resursa koji želi da koristi,
- kada proces zahteva resurse, sistem procenjuje da li će posle toga ostati u stabilnom stanju. Ako ostaje u stabilnom stanju, proces će dobiti resurs, a u protivnom proces mora da sačeka da drugi procesi oslobode neke resurse i
- proces koji dobije resurse mora da ih vrati u nekom konačnom vremenu.

Da bi se primenio bankarski algoritam potrebno je definisati određene strukrure podataka. Neka je n broj procesa, a neka m broj resursa. Uvodimo sledeće struture podataka u vidu nizova ili matrica:

(a) vektor rapoloživosti: *available[j]*, $j \in [0,m]$. Ako je available[j]=k, tada je k instanci resursa Rj raspoloživo,

(b) matrica makimalnih zahteva: *max[n,m]*. Ako je max[i,j] = k, tada proces Pi može tražiti najviše k instanci resursa Rj,

(c) matrica alokacije: *allocation[n,m]*. Ako je allocation[i,j] = k, tada je proces Pi trenutno dobio k instanci resursa Rj.

(d) matrica potreba: *need[n,m]*. Ako je need[i,j] = k, tada proces Pi može tražiti još k instanci resursa Rj.

Sa ovakvom defnicijom struktura podataka, važi sledeća formula:

need[i,j] = max[i,j] – allocation[i,j].

Uvešćemo još neke definicije. Ako su X i Y dva vektora od n elemenata, onda je $X \leq Y$ ako je $X[i] \leq Y[i]$ za svako $i \in [1,n]$. Takođe ćemo vrste matrica alokacije i matrice potrebe označiti kao jednodimenzione vektore *allocation[i]* i *need[i]* koji predstavljaju sve resurse koje proces Pi može dobiti ili tražiti.

Neka su *work* i *finish* vektori dužine m i n, respektivno. Bankarski algoritam određuje da li se sistem nalazi u bezbednom stanju.

[1] Inicijalizacija:

work = available (opisuje vektor raspoloživih resursa)

finish[i] = 0 za $i \in [1,n]$ (opisuje završetak procesa i)

[2] Pronalaženje procesa Pi koji može da zadovolji svoje potrebe, odnosno procesa za koji važi:

(2.a) finish[i] = 0

(2.b) need[i] ≤ work

Ako nema takvih procesa idi na korak 4.

[3] Procesu se dodeljuju svi potrebni resursi, nakon čega on završava aktivnost i vraća ih u sistem.

work = work + allocation[i] (oslobađanje resursa)

finish[i] = 1

Ići na korak 2.

[4] Ako je finish[i] = 1 za sve i, tada je sistem u stabilnom stanju

Sistem je u stabilnom stanju ako svi procesi mogu uspešno da se završe. Ovaj algoritam zahteva $m \cdot n^2$ operacija kako bi proverio da li je sistem u stabilnom stanju.

Pretpostavimo da neki proces zahteva resurse. Definisaćemo algoritam za dodelu resursa. Uvodimo najpre vektor $request_i$, kao vektor trenutnih zahteva procesa Pi. Ako je $request_i[j]=k$, tada proces Pi traži k instanci resursa Rj. Pre ulaska u sam algoritam za izbegavanje zastoja, operativni sistem proverava:

[1] Da li proces traži više od onoga što je specificirao kao maksimalan zahtev ($request_i \geq need_i$)?

[2] Da li su resursi raspoloživi ($request_i \leq available$)?

Ukoliko proces traži više od onoga što je specificirano kao maksimalan zahtev, prijavljuje se greška i odbija se zahtev. Ukoliko su resursi zauzeti, operativni sistem uvodi proces u stanje čekanja. U suprotnom, sistem primenjuje zamišljenu dodelu resursa (kvazi-dodela) procesu Pi po sledećoj formuli:

$available = available - request_i$ (broj raspoloživih resursa)

$allocation_i = allocation_i + request_i$ (broj alociranih resursa)

$need_i = need_i - request_i$ (potrebe procesa)

Sistem ispituje stanje sistema nakon zamišljene alokacije. Ukoliko stanje ostane stabilno, sistem će procesu Pi dodeliti resurse, a alokacija postaje stvarna. U protivnom proces mora da čeka, a zamišljena alokacija se ne obavlja - resursi se vraćaju u staro stanje.

Napomena: bankarski algoritam je teorijski jako dobro zamjišljen. Realno gledano, većina programera ne može unapred odrediti koji će resursi biti neophodni za izvršenje određene aplikacije.

Primer upotrebe bankarskog algoritma

U sistemu imamo pet procesa (P0 do P4) i tri resursa, A, B i C, koji redom imaju po 10, 5 i 7 instanci. U trenutku t_0 raspoložive su po tri instance resursa A i B i dve instance resursa C, odnosno available=(3,3,2). Stanje sistema u trenutku t_0 opisano je sledećom tabelom, pri čemu se stanje matrice potreba need definiše kao need=max-allocation:

vekt.	**Allocation**			**max**			**need**		
res.	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Pretpostavimo da proces P1 izdaje zahtev za korišćenjem resursa $request_1$=(1, 0, 2). Primenićemo algortiam zahtevanja resursa.

Najpre se proverava da li je $request_1 \leq need_1$. Ovaj uslov je ispunjen, jer je (1,0,2) ≤ (1,2,2). Zatim se proverava da li je $request_1 \leq available$. Proces P1 ispunjava ovaj uslov, jer je (1,0,2) ≤ (3,3,2). Nakon toga se obavi kvazi dodela resursi po zahtevu procesu P1. Stanje sistema nakon kvazi dodele opisano je sledećom tabelom, pri čemu je nakon dodele vektor available=(2,3,0).

vekt.	**Allocation**			**max**			**need**		
res.	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	1	0	7	5	3	7	4	3
P1	3	0	2	3	2	2	0	2	0
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Primenom bankarskog algoritma može se dokazati da postoji sekvenca <P1, P3, P4, P0, P2> koja će zadovoljiti uslove stabilnosti, odnosno da sistem nakon dodele resursa ostaje u bezbednom stanju. To znači da će sistem ispuniti zahtev za dodelu resursa procesu P1.

Međutim, sistem neće odobriti:

- zahtev procesa P4, $request_4$=(3,3,0), jer je zahtev veći od raspoloživih resursa i
- zahtev procesa P0, $request_0$=(0,2,0), jer nakon kvazi dodele sistem više neće biti u stabilnom stanju.

6.4. Detekcija i oporavak od zastoja

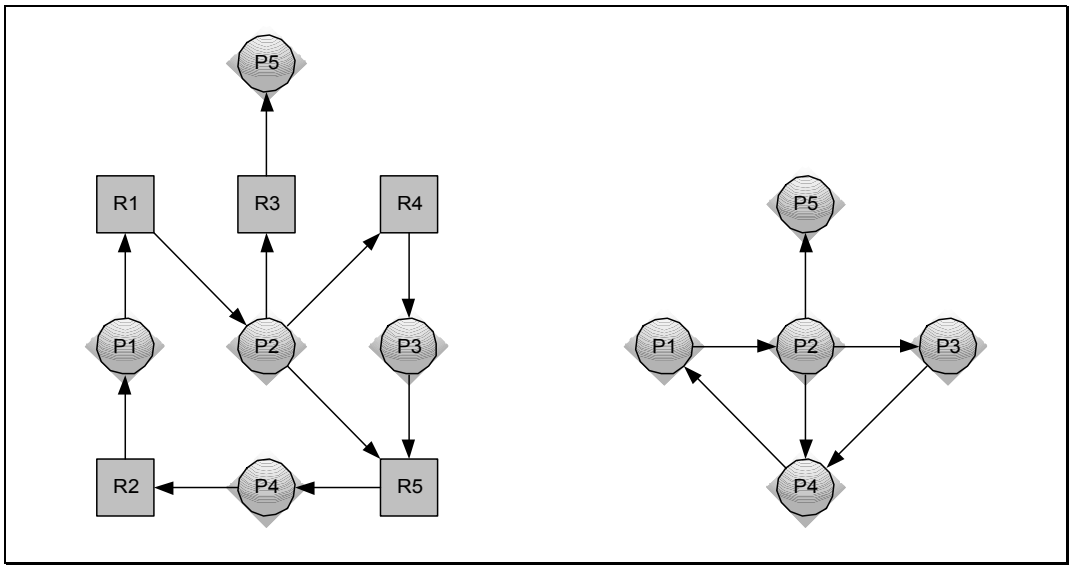
Ako sistem ne primenjuje algoritme za prevenciju ili izbegavanje zastoja, tada su zastoji sasvim mogući, čak i vrlo verovatni. U tom slučaju sistem mora da obezbedi dve stvari:

- algoritam za detekciju zastoja, koji treba da ispita da li se zastoj dogodio i
- algoritam za oporavak iz stanja zastoja.

Detekcija u slučaju da resursi imaju samo jednu instancu

Ako svi resursi imaju samo jednu instancu, primenjuje se algoritam za detekciju zastoja koji koristi specijalan graf čekanja na resurse (*wait-for graph*). Graf čekanja se konstruiše od grafa dodeljenih resursa, kao što je prikazano na slici 6.6. Pravila konstrukcije grafa su sledeća:

- čvorovi grafa su samo procesi (nema resursa) i
- strelice se crtaju samo između procesa koji čekaju jedan drugog za resurse.



Slika 6.6. *Graf čekanja na resurse*

Postojanje zastoja utvrđuje se ispitivanjem postojanja kružnih tokova: ako kružni tok postoji, to je zastoj. Dakle, algoritam se periodično poziva da ispita postojanje kružnih tokova, za šta je potrebno n^2 operacija, pri čemu je n broj strelica u grafu čekanja.

Detekcija u slučaju da resursi imaju više instanci

Algoritam za detekciju pomoću grafa čekanja nije pogodan za korišćenje ukoliko resursi imaju više instanci. U tom slučaju koristi se algoritam sličan bankarskom. Najpre ćemo uvesti sledeće strukture podataka:

(a) Vektor rapoloživosti: available[j], $j \in [0,m]$. Ako je available[j]=k, tada je k instanci resursa Rj raspoloživo.

(b) Matrica alokacije: allocation[n,m]. Ako je allocation[i,j]=k, tada je proces Pi trenutno dobio k instanci resursa Rj.

(c) Matrica trenutnih zahteva request[n,m]. Matrica ukazuje na trenutne zahteve procesa. Ako je request[i,j]=k, tada proces Pi trenutno traži k instanci resursa Rj.

Algoritam funkcioniše na sledeći način:

Neka su work i finish vektori dužine m i n, respektivno.

[1] U ovom koraku svi procesi koji imaju bilo kakve resurse se obeležavaju kao nezavršeni, a u protivnom ispadaju iz algoritma.

work = available

for i = 1,2, …, n

if $allocation_i \neq 0$ then finish[i]=0;

else finish[i]=1.

[2] U ovom koraku nalazi se proces koji može da zadovolji sve svoje potrebe i završi aktivnost, odnosno proces za koji važi:

(2.a) finish[i]=0

(2.b) $request_i \leq work$

Ako nema takvih procesa, idi na korak 4.

[3] U ovom koraku se resursi procesa vraćaju u sistem.

work = work + $allocation_i$

finish[i]=1

Idi na korak 2.

[4] Ako je finish[i]=0, za bilo koje $i \in [1,n]$, tada je sistem u stanju zastoja. Preciznije, ako je finish[i]=0, tada je proces Pi u zastoju.

Algoritam zahteva m x n^2 operacija da bi ustanovio da li je sistemu stanju zastoja.

Ilustraciju algoritma ćemo prikazati na primeru. Posmatrajmo sistem sa pet procesa (P0 do P4) i tri resursa: A (7 instanci), B (2 instance), C (6 instanci). U trenutku t_0 nema raspoloživih resursa, odnosno available=(0,0,0). Stanje sistema u trenutku t_0 opisano je sledećom tabelom:

vekt.	**allocation**			**request**		
res.	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	0
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

Kada se primeni algoritam za detekciju, može se dokazati da postoji sekvenca <P0, P2, P3, P1, P4> koja dovodi do rezultata finish[i]=1 za sve procese, što znači da sistem neće biti doveden u stanje zastoja. Interesantno je da sistem u trenutku t_0 nema raspoloživih resursa, ali da se počevši od procesa koji najmanje traže (P0 i P2) može obaviti dodela resursa bez dovođenja do zastoja. Međutim, dovoljno je da proces P2 traži jednu instancu resursa C i sistem će biti doveden u stanje zastoja. Dokazaćemo to primenom prethodno opisanog algoritma za detekciju: proces P0 ima najmanje prohteve, i ako bi završio svoje aktivnosti i vratio resurse, stanje raspoloživih resursa bi bilo (0,1,0) nakon čega nemamo proces koji može da zadovolji svoje potrebe. Znači sistem je u stanju zastoja - procesi P1, P2, P3, i P4 su zaglavljeni.

Korišćenje algoritma za detekciju

Učestanost pozivanja algoritma za detekciju zastoja zavisi od toga učestanosti pojave zastoja i prosečnog broja procesa zahvaćenog zastojem. Sa jedne strane, zastoj se mora detektovati na vreme jer su svi procesi (a samim tim i resursi koje procesi koriste) zaglavljeni, ali kako detekcija konzumira dosta vremena, algoritam se ne sme pozivati često.

Oporavak od zastoja

Posle detekcije zastoja potrebno je izvršiti oporavak. Dve metode za oporavak od zastoja koje se najčešće koriste su:

- prekid izvršenja procesa u zastoju i
- nasilno oduzimanje resursa od procesa koji su u zastoju.

Eliminacija zastoja nasilnim prekidanjem izvršenja zaglavljenih procesa odvija se na više načina, a u svakom slučaju se osobađaju svi resursi koje je zauzimao proces koji se prekida. Postoje dve tehnike:

[1] Prekid izvršenja svih zaglavljenih procesa.

Ova procedura traje dugo, a ima i sekundarne efekte, kao što je poništavanje svega što su procesi postigli u toku normalnog izvršenja..

[2] Prekid izvršenja procesa do razbijanja kružnog toka.

Ova metoda je svakako bolja, ali treba obaviti pravi izbor procesa, kao i redosled prekidanja procesa. U najboljem slučaju, prekida se izvršenje samo jednog procesa. Prilikom izbora procesa za prekidanje u razmatranje treba uzeti sledeće faktore:

- prioritet procesa,
- koliko dugo proces već radi i koliko mu je vremena ostalo do završetka aktivnosti,
- koje resurse proces koristi,
- koje resurse proces traži da kompletrira aktivnost i
- da li su to interaktivni ili grupni procesi.

Eliminacija zastoja oduzimanjem resursa je tehnika koja ne prekida izvršenje procesa. Određenom procesu (ili procesima) u zastoju se oduzimaju resursi. Najpre se bira proces kome će se oduzeti resursi (*select a victim*). To se mora učiniti sa najmanje gubitaka, a kriterijumi izbora su različiti: na primer, koliko resursa pirpadaju procesu, koliko dugo proces radi itd. Proces kome oduzimamo resurse ostaće i dalje blokiran. Potrebno je sačuvati bezbedno stanje procesa kako bi se kasnije mogao nastaviti, ali to nije uvek jednostavno. Takođe, treba izbegavati zakucavenje (*starvation*) nekih procesa - izvršenje procesa koji se često bira kao žrtva za oduzimanje resursa će veoma dugo trajati.

7

Upravljanje memorijom

Sadržaj poglavlja:

7.1. Uvodne napomene
7.2. Programerske tehnike upravljanja memorijom
7.3. Kontinualna alokacija memorije
7.4. Straničenje
7.5. Segmentacija

7.1. Uvodne napomene

Memorija je, kao i procesor, fundamentalni deo modernih računarskih sistema. Memorija se sastoji iz niza memorijskih reči od kojih svaka ima jedinstvenu adresu. Prilikom izvršavanja procesa, procesor na osnovu vrednosti programskog brojača (PC, *program counter*) čita instukcije iz memorije (*fetch*). Pročitane instrukcije u toku izvršenja dodatno mogu zahtevati čitanje operanada ili upisivanje podataka na druge memorijske lokacije. Slobodno se može reći da proces bez memorije ne može gotovo ništa da uradi - u većini slučajeva, ne može ni da se izvrši, tako da se sloj upravljanja memorijom smešta najbliže jezgru.

U ovom poglavlju obrađeni su osnovni koncepti i metode upravljanja memorijom. Ove metode mogu biti krajnje jednostavne, ali i veoma složene, poput straničenja i segmentacije, a svaka ima svoje prednosti i nedostatke. Svakako, izbor metode za upravljanje memorijom u velikoj meri zavisi i od hardverske podrške, odnosno od procesorske arhitektuture.

Ciljevi koje treba postići

U nastavku teksta nabrojani su značajniji ciljevi koje treba postići na nivou sloja upravljanja memorijom:

[1] Alokacija memorije, odnosno dodela memorije procesima.

[2] Razdvajanje fizičkog i logičkog adresnog prostora programa i vezivanje adresa (prevođenje relativnih relokatibilnih adresa u fiksne).

[3] Logička organizacija memorije, odnosno razdvajanje neizmenljivih segmenata (modula i procedura) od segmenata sa promenljivim sadržajem, odnosno podacima. Razdvajanjem segmenata programa postiže se:

(a) nezavistan rad na segmentima koji čine program,

(b) zaštita segmenata sa malim premašenjem u vidu listi za kontrolu pristupa segmentima (u višeprocesnoj okolini procesi ne smeju menjati vrednosti memorijskih lokacija koje pripadaju drugim procesima ako za to nemaju dozvolu),

(c) deljenje segmenata između većeg broja procesa (u slučaju da proces ima dozvolu da izmeni vrednost memorijskih lokacija koje koristi drugi proces, operativni sistem mora da obezbedi kontrolisani pristup deljenom memorijskom području). Višeulazne (*re-entrant*) procedure se, na primer, realizuju pomoću deljive memorije.

[4] Relokacija, koja obuhvata kompakciju, odnosno defragmentaciju operativne memorije (vezivanje određenog broja diskontinualnih memorijskih blokova u jedan kontinualan nefragmentisan prostor) i *swap* (suspendovanje procesa na disk).

[5] Podrška za dinamičko punjenje memorije programom i dinamičko vezivanje.

Alokacija memorije

U operativnoj memoriji se, pored korisničkih procesa, smeštaju i rezidentni deo operativnog sistema (na primer, kernel), kao i neke tranzijentne komponente. Da bi u višeprocesnoj okolini obezbedio stabilan i pouzdan rad sistema, neophodno je što efikasnije alocirati različite delove memorije. Memorija se deli na najmanje dve particije od kojih je jedna (najčešće niži deo) namenjena rezidentnom delu operativnog sistema (*kernel space*) a druga particija (viši delovi) korisničkim procesima (*user space*). Obično se u najnižem delu memorije nalazi tabela prekidnih rutina.

U korisničkom adresnim prostoru nalazi se više procesa koji formiraju red čekanja na procesor. Takođe, postoje i procesi koji nastoje da uđu u red čekanja, obično sa diska. Da bi proces ušao u red čekanja na procesor, neophodno je da najpre dobije

potrebnu memoriju. Glavni problem kod upravljanja memorijom je dodela slobodne memorije procesima koji su u ulaznom redu (alokacija memorije).

Tehnike za dodelu memorije procesima se grubo mogu podeliti na dve vrste:

[1] Kontinualna alokacija (*contignous allocation*).

I logički i fizički adresni prostor procesa sastoje se iz kontinualnog niza memorijskih adresa, pri čemu memorijske particije koje se dodeljuju procesima po veličini mogu biti jednake ili različite.

[2] Diskontinualna alokacija (*discontignous allocation*).

Generalno gledano, fizički adresni prostor procesa nije realizovan kao kontinualan niz memorijskih adresa. Diskontinualna alokacija obuhvata metode: straničenja, segmentacije i straničenja sa segmentacijom.

Vezivanje adresa

Po pravilu, program se nalazi na disku kao izvršna binarna datoteka (*binary executable*). Program se sa diska mora učitati u memoriju, unutar adresnog prostora novokreiranog procesa. Zavisno od metode upravljanja memorijom koja se koristi, proces se u toku izvršavanja može više puta pomerati na relaciji disk-memorija. Kolekcija procesa na disku, koja čeka povratak u memoriju i nastavak izvršenja, naziva se ulazni red (*input queue*).

Na sistemima na kojima je prisutno multiprogramiranje, veći broj procesa deli operativnu memoriju računara. Programer koji piše izvorni kod koji će se izvršavati u takvoj okolini ne može unapred znati koji će procesi biti prisutni u memoriji prilikom izvršenja programa, niti koje će memorijske lokacije biti slobodne. Programer ne može unapred odrediti fiksne memorijske lokacije za smeštaj programa, i zato koristi relativne ili simboličke adrese (zavisno od programskog jezika i konkretne potrebe). Dužnost operativnog sistema je da prevede relativne, odnosno relokatibilne adrese u fiksne prilikom učitavanja programa u memoriju, odnosno da izvrši vezivanje adresa.

Pre samog izvršavanja, korisnički program obično prolazi kroz više faza, tokom kojih se memorijske adrese predstavljaju na različite načine. Prevodilac (*compiler*) prevodi izvorni kod programa i vezuje (*bind*) simboličke adrese iz izvornog koda za relativne, odnosno relokatibilne adrese, koje punilac (*loader*) pretvara u apsolutne prilikom učitavanja programa u memoriju. Na primer, prevodilac vezuje promenljivu count za lokaciju na adresi 14 u odnosu na početak programskog modula, koju punilac pretvara u apsolutnu adresu 74704. Vezivanje adresa se može shvatiti kao transformacija, odnosno prevođenje adrese sa "jezika" jednog adresnog prostora u drugi.

Vezivanje instrukcija i podataka na memorijske adrese obavlja se u sledećim fazama (slika 7.1):

[1] Vreme prevođenja (*compile time*).

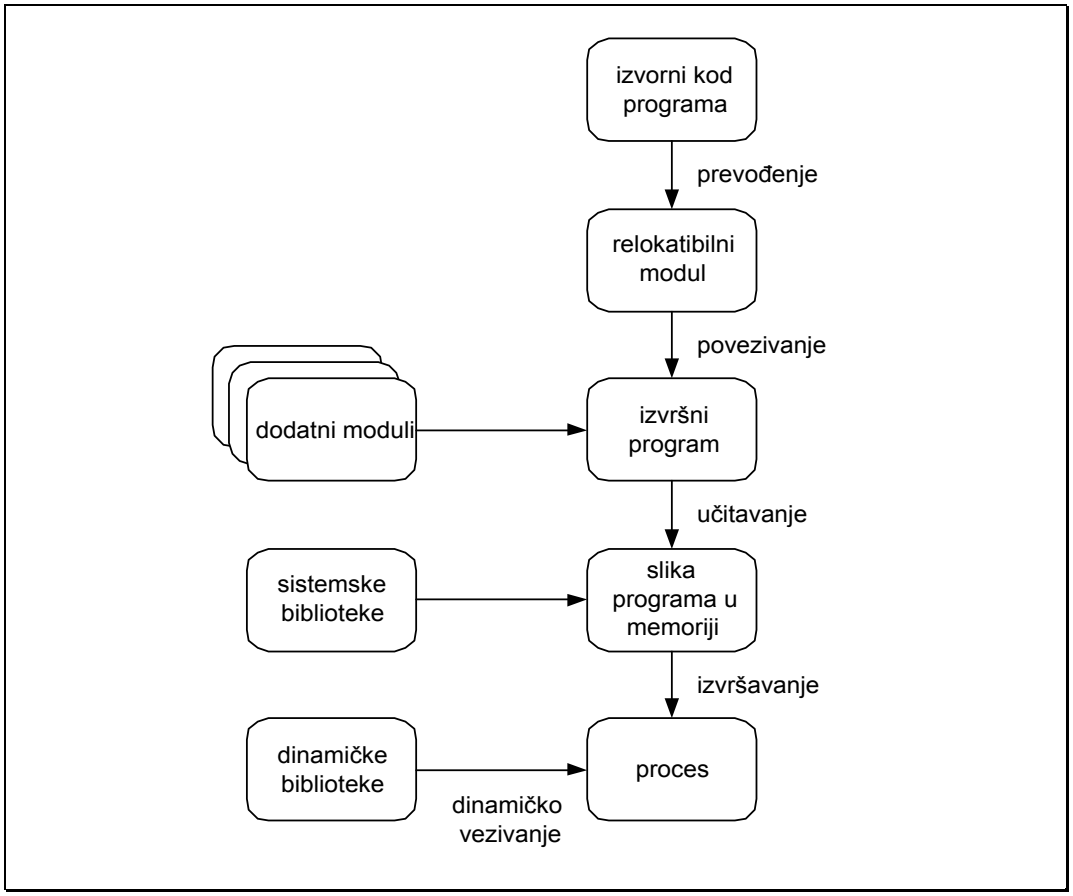
Kako obično nije poznato gde će proces koji će izvršavati generisani kod biti smešten u memoriji, prevodilac generiše relativne, a ne apsolutne adrese. Program se kasnije može smestiti bilo gde u memoriji.

[2] Vreme punjenja u memoriju (*load time*).

U ovoj fazi povezivač (*linker*) i punilac na bazi relokatibilnog koda generišu apsolutne adrese i pune memoriju programom. Povezivač može povezati korisnički program sa drugim relokatibilnim modulima (*object modules*).

[3] Vreme izvršavanja (*execution time*).

Za vreme izvršavanja, proces se može pomerati sa jednog segmenta na drugi (uključujući i disk, ukoliko se koristi virtuelna memorija).

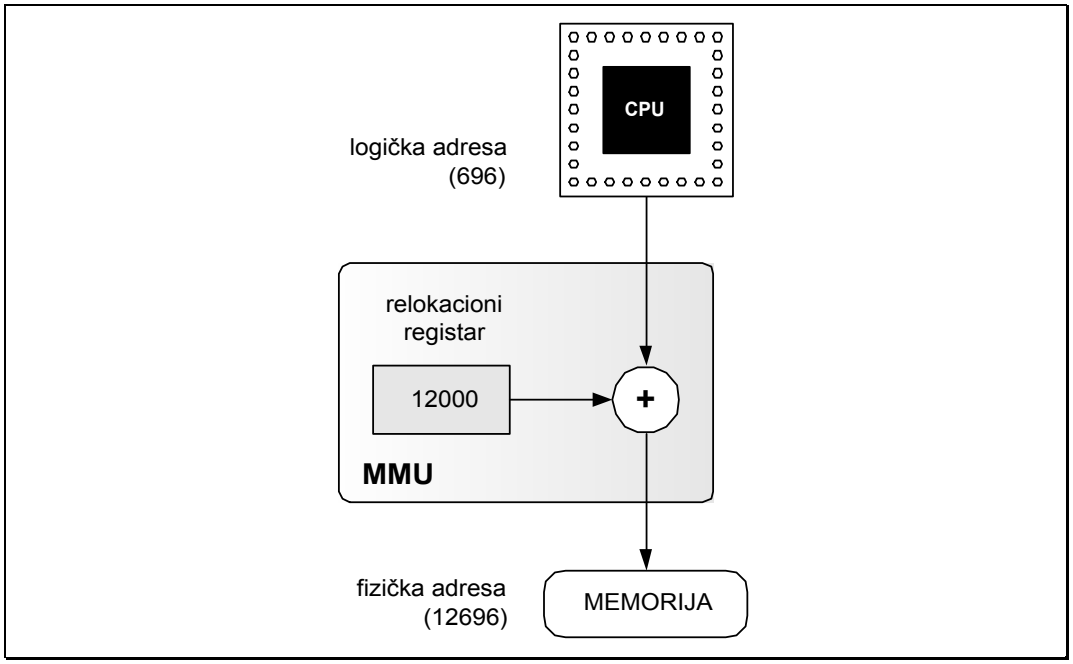


Slika 7.1. *Faze koje prethode izvršavanju programa*

Logički i fizički adresni prostor

Adresa koju generiše procesorka instrukcija je logička adresa, a adresa same memorijske jedinice je fizička adresa. Logičke i fizičke adrese su potpuno iste u fazi prevođenja i u fazi punjenja, ali se razlikuju u fazi izvršavaja programa (logičke adrese se u fazi izvršavanja nazivaju i virtuelnim adresama). Skup svih logičkih adresa koje generiše program naziva se logički ili virtualni adresni prostor, a skup svih fizičkih adresa koje njima odgovaraju naziva se fizički adresni prostor.

Mapiranje virtuelnog adresnog prostora u fizički obavlja hardverski uređaj koji se naziva MMU (*Memory-Management Unit*) - jedinica za upravljanje memorijom. Šeme mapiranja mogu biti veoma kompleksne, ali je zbog jednostavnosti izlaganja na slici 7.2. prikazana najjednostavnija šema koja koristi relokacioni registar.



Slika 7.2. *Mapiranje pomoću relokacionog registra*

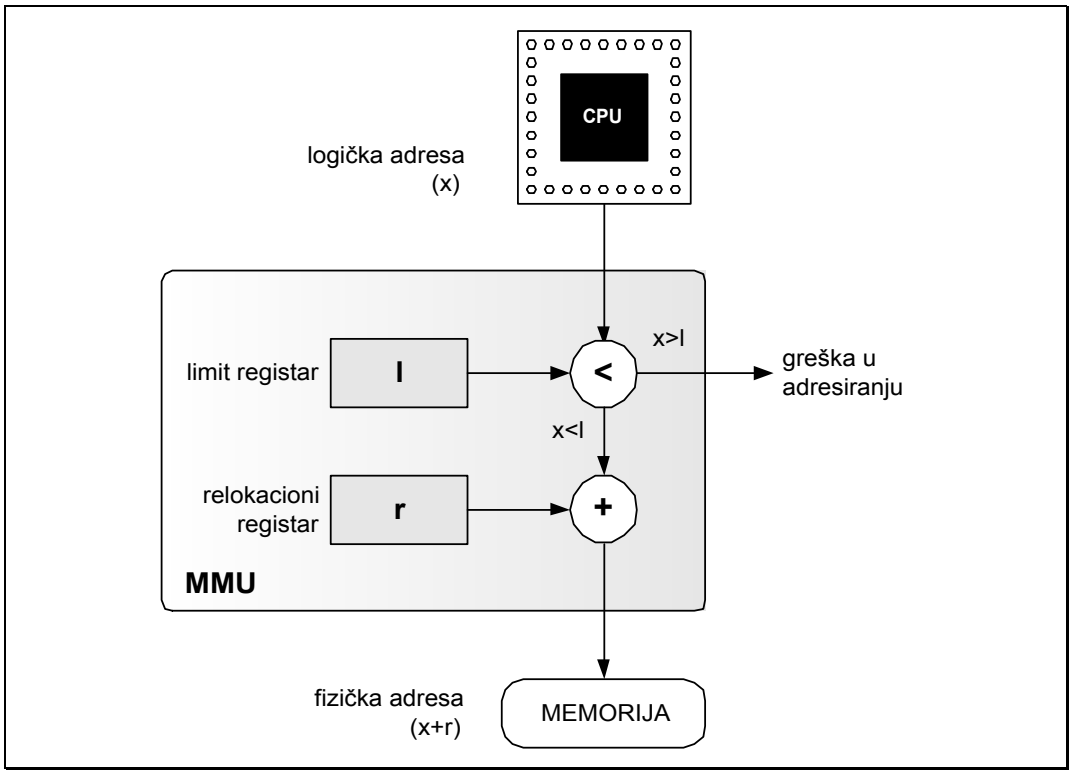
Relokacioni registar definiše adresu fizičkog početka programa. Svaka logička adresa koju generiše program se sabira sa vrednošću relokacionog registra, i tako se dobija fizička adresa. Korisnički program uvek počinje od nulte adrese i ne treba da vodi računa o svom fizičkom prostoru, osim o gornjoj granici programa (max). Logički adresni prostor koji se nalazi u opsegu [0, max] se mapira u opseg [R+0,R+max], gde je R vrednost relokacionog registra, odnosno fizička adresa početka programa.

Zaštita memorije

Zaštita operativnog sistema od korisničkih procesa i međusobna zaštita korisničkih procesa po pitanju pristupa memorijskim sekcijama.se može realizovati pomoću dva registra:

- relokacionog registra, koji sadrži najnižu adresu procesa i
- limit registra, koji sadrži najveći opseg logičkih adresa procesa.

Jedinica za upravljanje memorijom (MMU) mapira svaku logičku adresu procesa dinamički, tako što najpre proverava da li je logička adresa manja od vrednosti limit registra, i ako jeste, dodaje vrednost relokacionog registra (slika 7.3).



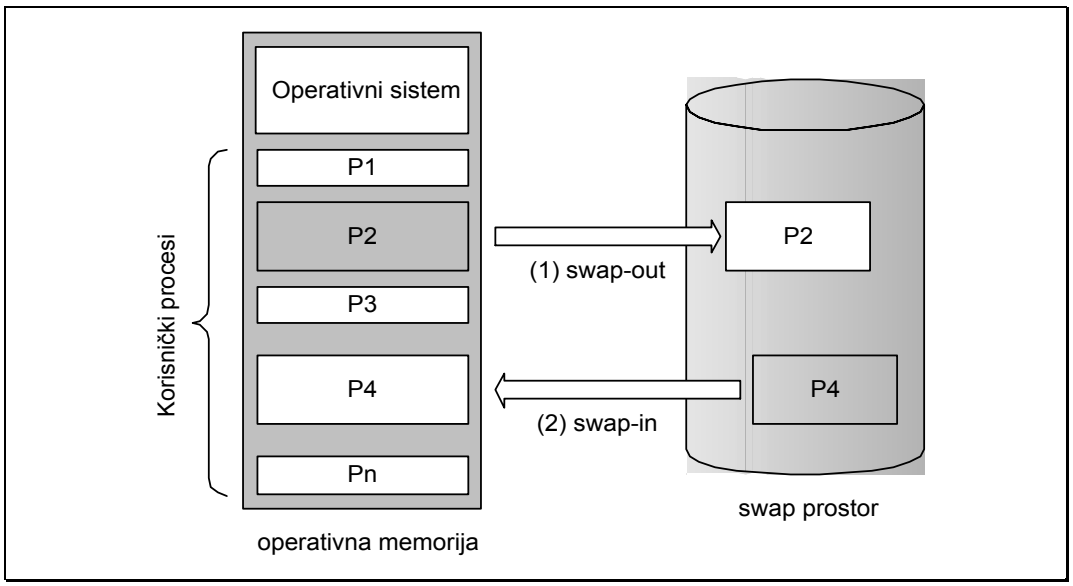
Slika 7.3. *Relokacioni i limit registar*

Neophodno je naglasiti da su relokacioni i limit registar dva registra procesora, koji se pune onda kada proces dobija procesor na izvršenje. Iz konteksta procesa dispečer čita vrednosti ova dva registra i postavlja ih prilikom aktiviranja procesa. Pomoću ova dva registra, procesi uspešno štite svoje memorijske sekcije, jer je svaka logička adresa relativna u odnosu na nulu i mora da prođe obradu preko ova dva registra.

Šema sa relokacionim i limit registrom obezbeđuje efektivan način da se veličina memorija koja pripada operativnom sistemu menja dinamički, odnosno da se delovi koda koji nisu potrebni trenutno izbace ili da se ponovo dovedu u memoriju kada zatrebaju. Ti delovi operativnog sistema se nazivaju tranzijentni kod, za razliku od rezidentog koda koji je uvek u memoriji.

Razmena (swap)

Proces se mora prilikom izvršavanja nalaziti u operativnoj memoriji. Postoje situacije kada se proces može privremeno prebaciti iz memorije na disk, kako bi se oslobodila memorija. Oslobođena memorija puni se drugim procesom, kao što je ilustrovano na slici 7.4. Posle izvesnog vremena, proces se može vratiti sa diska u memoriju kako bi nastavio izvršavanje.



Slika 7.4. *Swap*

Razmena (*swap*) se koristi u prioritetnim šemama za raspoređivanje procesa, gde se procesi visokog prioriteta čuvaju u memoriji, a svi procesi niskog prioriteta se upisuju na disk i čekaju da se oslobodi memorija. Ova varijanta razmenjivanja se naziva *roll out, roll in*. Proces koji se razmenjuje mora biti potpuno osobođen aktivnosti - ne sme da radi, niti da čeka kraj neke ulazno-izlazne operacije (stanja READY i WAIT).

Tehika razmene zahteva postojanje tri komponente:

- prostor na disku (*swap space*), na koji će se smeštati "uspavani" procesi,
- mehanizam *swap-out* koji prebacuje proces iz memorije na disk i

- mehanizam *swap-in* koji vraća "uspavani" proces sa diska u memoriju.

Najveći deo vremena u ciklusima razmene otpada na transfer podataka između memorije i diska. Trajanje jedne razmene zavisi od količine podataka za transfer i karakteristika diskova i pratećeg hardvera. Kako je to vreme ogromno u odnosu na performanse savremenih procesora, ne preporučuje se često korišćenje tehnike razmenjivanja.

Swap postoji na svim modernim operativnim sistemima i to u različitim modifikovanim varijantama. Razmena celih procesa se retko koristi - uglavnom se razmenjuju manji delovi memorije (na primer, stranice). Tehnika razmenjivanja se tesno povezuje sa tehnikom virtulene memorije.

7.2. Programerske tehnike upravljanja memorijom

Kao što je već rečeno, operativni sistem je dužan da obezbedi odgovarajuću podršku za rad sa operativnom memorijom. Međutim, prilikom projektovanja programa koji zbog veličine ne mogu stati celi u memoriju, programeri koriste odgovarajuće tehnike kojima se postiže prividno povećanje i bolje iskorišćenje memorijskog prostora.

Dinamičko punjenje memorije programom

U dosadašnjim diskusijama podrazumevalo se da se ceo program smešta u memoriju, pri čemu je veličina procesa limitirana količinom fizičke memorije. Memorija se može efikasnije iskoristiti ukoliko se primene tehnike dinamičkog punjenja programa. Suština dinamičkog punjenja (*dynamic loading*) je u smeštaju samo potrebnih delova programa u memoriju, pri čemu se odgovarajuće rutine učitavaju u memoriju samo kada ih program pozove. Da bi to bilo moguće, sve rutine programa se čuvaju na disku u relokatibilnom formatu. Kada se rutina pozove iz programa, tada se proverava da li je ona već u memoriji i ako nije, poziva se punilac da je učita u memoriju.

Prednosti dinamičkog punjenja su u tome što rutine koje nisu trenutno potrebne ne zauzimaju mesto u memoriji, što je zgodno za velike programe. Takođe, dinamičko punjenje ne zahteva specijalnu podršku od operativnog sistema - programer mora sam da projektuje svoje programe da koriste principe dinamičkog punjenja. Operativni sistem može pomoći programeru tako što obezbeđuje biblioteku za dinamičko punjenje.

Dinamičko povezivanje

Prilikom statičkog povezivanja, sistemske biblioteke se tretiraju kao i svi drugi objektni moduli koji se kombinuju sa korisničkim programom u jednistvenu izvršnu

verziju kojom se puni memorija. Neki operativni sistemi podržavaju jedino koncept statičkog povezivanja. Koncept dinamičkog povezivanja (*dynamic linking*) je sličan konceptu dinamičkog punjenja. Kao što se punjenje modula odlaže (memorija se puni modulima po potrebi), isto se postupa i sa povezivanjem (*link*) u vreme izvršavanja - rutina iz sistemske biblioteke se puni po potrebi u vreme izvršenja. Kod statičkog likovanja svaki poziv sistemske bibioteke zahteva da se kod kompletne rutine bibioteke kopira u kod programa i to za svaki poziv pojedinačno, čime raste veličina programa i na disku a i u memoriju kada dođe vreme izvršavanja. Kod dinamičkog povezivanja napravi se takozvani *stub*, kao parče koda za svaki poziv sistemkse biblioteke. Stub služi da pokaže kako da se locira odgovarajuća sistemska rutina i kako da se napuni, ako nije već u memoriji. Preko stuba, rutina koja se jedanput dovede u memoriju se može koristiti više puta. Dakle, za svaki poziv iste funkcije u programu imamo samo jedno memorijsko punjenje i zauzeće. Dimamičko punjenje omogućava laku izmenu sistemskih biblioteka, a da stari programi ne moraju da se prevode ponovo (*shared library*).

Za razliku od dinamičkog punjenja, dinamičko povezivanje traži podršku na nivou operativnog sistema, u kontekstu da istu sistemsku rutinu koja je u memoriji mogu koristititi više procesa.

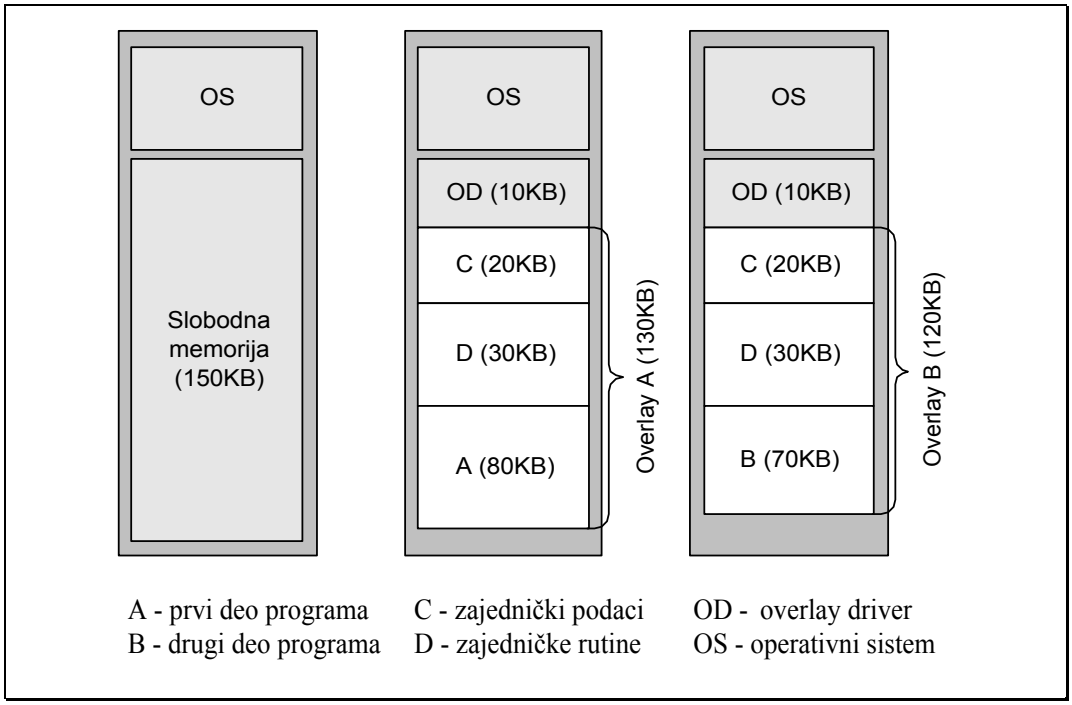
Tehnika preklapanja

Da bi se omogućilo izvršenje procesa koji je veći od same fizičke memorije, koristi se tehnika preklapanja (*overlay*). Ideja preklapanja sadržana je u tome da se u memoriji čuvanju samo oni delovi programa koji su potrebni u tom trenutku. Kada drugi delova programa budu potrebni, oni će se učitati u memoriju umesto onih delova koji više nisu potrebni.

Ilustrovaćemo ovo na jednostavnom primeru programa sačinjenog od dve relativno nezavisne komponente koje se izvršavaju jedna za drugom. Programer može podeliti sam izvorni kod programa na dva dela. Pretpostavimo da je veličina programskih komponenti sledeća: prvi deo (80KB), drugi deo (70KB), zajednički podaci (20KB), zajedničke rutine (30KB). Za izvršavanje programa potrebno je najmanje 200KB slobodne memorije (ne računajući rezidentne delove operativnog sistema), što znači da program ne može da se izvršava na hipotetičkom sistemu sa 150KB operativne memorije. Međitim, prilikom izvršavanja prvog dela programa, kod drugog dela ne mora biti učitan u memoriju. Takođe, drugi deo programa se može izvršavati ukoliko prvi deo nije prisutan u memoriji. Zato ćemo definisati dve *overlay* komponente:

- *Overlay* A, koji se sastoji od zajedničkih podataka i rutina i koda za prvi deo programa, što iznosi 130KB.
- *Overlay* B, koji se sastoji od zajedničkih podataka i rutina i koda za drugi deo programa, što iznosi 120KB.

Programu se mora dodati *overlay* drajver koji upravlja *overlay* tehnikom (pretpostavimo da on zauzima 10KB memorije). U prvoj fazi imamo *overlay* drajver i *overlay* A što zauzima 140KB i može se izvršiti u 150KB. Kada *overlay* A, završi svoje, drajver će učitati u memoriju *overlay* B preko njega. U drugoj fazi u memoriji su prisutni *overlay* drajver i *overlay* B, što zauzima 130KB i takođe može da se izvrši na sistemu sa 150KB memorije (slika 7.5).



Slika 7.5. *Overlay tehnika*

Tehnika preklapanja usporava rad samog programa, jer se delovi programa učitavaju u memoriju u više iteracija, ali se njome postiže ono što drugačije ne bi bilo moguće. Tehnika preklapanja može biti korisna na sistemima sa ograničenim resursima.

Overlay drajveri su uključeni u neke programske jezike (na primer, Borland Pascal i Clipper), tako da rad sa tehnikom preklapanja ne zahteva posebnu podršku operativnog sistema. Konkretnu strukturu *overlay* delova definiše programer, koji najbolje poznaje svoj program.

7.3. Kontinualna alokacija memorije

Ukoliko se koristi kontinualna alokacija memorije, i logički i fizički adresni prostor procesa sastoje se od kontinualnog niza memorijskih adresa. Prosto rečeno, svaki proces dobija jedan kontinualni deo memorije.

Multiprogramiranje sa fiksnim particijama

Jedan od najstarijih i najprostijih metoda alokacije memorije je podela cele fizičke memorije na više delova fiksne veličine, pri čemu se u jednom delu može naći samo jedan proces. U ovakvoj organizaciji, stepen multiprogramiranja je jednak broju memorijskih particija. Ovaj metod je korišćen u sistemu IBM OS/360. Cela memorija se izdeli na više delova, a proces se smešta u onaj deo koji je veći ili jednak memoriji koju zahteva proces. Svi procesi se stavljaju u red čekanja (*input queue*) koji može biti jedinstven za ceo sistem ili poseban za svaku particiju.

Višestruki redovi čekanja se formiraju obično za opsege veličina (Q=1KB, Q=2KB, Q=4KB, itd). Ukoliko za proces koji je došao na red nema dovoljno memorije, uzima se sledeći manji proces iz liste. U slučaju višestrikih redova čekanja, veći broj malih procesa može čekati u redu za male particije, dok su velike particije neiskorišćene. U tom slučaju ima dovoljno memorije, ali se ne koristi. Sa jednog stanovišta bolji je jedinstveni red čekanja, jer ako nema mesta u memoriji u redu čekanja za particiju koja odgovara veličini procesa, procesu se dodeljuje veća particija. Dva procesa nikako ne mogu biti smeštena u jednoj particiji.

Obe vrste reda čekanja pate od interne fragmenatcije - delovi memorije koji su veći od veličine procesa potpuno neiskorišćeni.

Multiprogramiranje sa fiksnim particijama korišćeno je u sistemima sa grupnom obradom (*batch*). Za interaktivne sisteme, ovaj metod je krajnje nepogodan zbog postojanja većeg broja procesa promenljive veličine (obično manjih) koji se pojavljuju po nekom slučajnom rasporedu. Kao takva, ova metoda se više ne koristi, ali je vredna pomena jer predstavlja osnovu za definisanje nove metode - multiprogramiranja sa particijama promenljive veličine.

Multiprogramiranje sa particijama promenljive veličine

Umesto fiksnih particija, memorija se dinamički deli, a svaka šupljina (*hole*), odnosno slobodni kontinualni deo memorije se može iskoristiti za smeštanje procesa, pod uslovom da je dovoljno velika. Šupljine se dinamički kreiraju i nestaju, zajedno sa procesima, mogu biti bilo gde u memoriji i imati bilo koju veličinu, što odgovara procesima na interaktivnim sistemima. Kada proces naiđe u sistem, traži se šupljina

dovoljno velika da zadovolji proces. Sav prostor koji proces ne zauzme od cele šupljine, predstavlja novu šupljinu u koju se može smestiti drugi proces.

Alokacija memorije je dinamička - memorija se sastoji iz procesa i šupljina, a operativni sistem dinamički vodi evidenciju o zauzetosti memorije na jedan od sledećih načina:

[1] Bit mape (bit maps).

Memorija se deli na delove iste veličine, a u krajnjem slučaju to su bajtovi. Svakom delu dodeljujemo po jedan bit na sledeći način: 1 označava da je taj deo zauzet, a 0 da je slobodan, ili obrnuto. Tako dolazimo do niza nula i jedinica koji predstavlja bit mapu memorije. Postavlja se sledeće pitanje: koje veličine treba da budu ovi delovi memorije? Ako je veličina delova 32 bita (4 bajta), onda za bit mapu koristimo 1/33 od ukupne memorije. Ako su delovi manji, bit mapa je veća (ako je veličina delova 16 bita, onda za bitnu mapu koristimo 1/17 od ukupne memorije: od 512MB na administraciju otpada oko 30MB), a pretraživanje slobodne memorije (niz nula) je sporo. Ako su delovi veći, dobijamo lošu iskorišćenost memorije. Neka su delovi veličine 4KB - ako proces zauzme 1KB od nekog dela, 3KB ostaje neiskorišćeno, ali ne može biti alociran od strane drugog procesa jer je formalno zauzeto.

[2] Povezane liste (*linked lists*).

Povezane liste gradimo od slogova sledeće strukture: prvo polje sloga označava tip memorije (P znači da se radi o procesu, a H da se radi o slobodnoj memoriji). Drugo polje sadrži početnu adresu dela memorije koju opisuje dati slog. Treće polje označava dužinu opisane memorije, a četvrto sadrži pokazivač na sledeći slog. Upravljanje memorijom se odvija na sledeći način: prilikom zauzimanja memorije u povezanoj listi se traži rupa (slog tipa H) dovoljne veličine. Ako se odgovarajuća rupa nađe, umesto H se upisuje P, a eventualno se ubacuje i novi čvor tipa H (ukoliko proces ne zauzima celu rupu, ostaće neka manja rupa). Kada proces A završi sa radom, oslobođa se zauzeta memorija. Tada se, ukoliko se i ispred i iza procesa A nalazi neki drugi proces (a ne slobodna memorija), jednostavno stavlja H umesto P. Ako je ispred ili iza procesa A postojala šupljina, vrši se ažuriranje liste, čime se dobija veća šupljina.

[3] Sistem udruženih parova - drugova (*buddy system*).

Za upravljanje memorije koristi se po jedna lista za blokove slobodne memorije veličine 2^n bajtova (1, 2, 4, ... do ukupne veličine memorije). Uzmimo, na primer, da imamo 1MB memorije, tada ćemo imati 21 listu (2^0=1, 2^2=2 ... 2^{20}=1MB). Na početku rada, cela memorija je prazna, tako da u listi za rupe veličine 1MB imamo jedan slog, dok su ostale liste prazne. Proces A veličine 70KB se, nakon nailaska u sistem, može smestiti u particiju veličine najmanje 128KB (mora biti stepen od 2). Na žalost, lista koja sadrži particije te veličine je prazna. Zato se particija veličine 1MB deli na dva dela od 512KB (ovi delovi se zovu drugovi – *buddy*). Zatim se prvi deo deli na dva dela od 256KB, pa još jednom, na dva dela od 128KB. U prvu particiju se smešta proces. Posle toga lista od 128Kb sadrži dva čvora (jedan je zauzet – P, a drugi slobodan – H), dok liste od 256KB i 512KB sadrže po jedan slobodan čvor:

```
P (128KB)    {proces A:70KB, neiskorišćeno:58KB}
H (128KB)
H (256KB)
H (512KB)
```

Sada je vidljiv i nedostatak ovog sistema: proces veličine 70KB zauzima čitavu particiju od 128KB, pri čemu 58KB memorije ostaje neiskorišćeno (unutrašnja fragmentacija). Proces B veličine 35KB se nakon nailaska u sistem može smestiti u particiju veličine 64KB. Lista za takve particije je prazna, tako da se prva slobodna particija od 128KB deli na dva dela veličine 64KB. Proces B se smešta u jednu particiju, dok druga ostaje prazna. Sada imamo po jedan čvor u listama za particije veličine 128KB, 256KB i 512KB i dva čvora u listi za particije veličine 64KB:

```
P (128KB)    {proces A:70KB, neiskorišćeno:58KB}
P (64KB)     {proces B:35KB, neiskorišćeno:29KB}
H (64KB)
H (256KB)
H (512KB)
```

Novi proces C veličine 80KB ubacujemo tako što delimo pariciju veličine 256KB na dva dela od 128KB: u prvi ubacimo proces, a drugi ostaje prazan.

```
P (128KB)    {proces A:70KB, neiskorišćeno:58KB}
P (64KB)     {proces B:35KB, neiskorišćeno:29KB}
H (64KB)
P (128KB)    {proces C:80KB, neiskorišćeno:48KB}
H (128KB)
H (512KB)
```

Posmatrajmo sada stanje memorije: tri procesa ukupne veličine 70+35+80=185KB zauzimaju 128+64+128=320KB memorije, dok 135KB ostaje neiskorišćeno kao posledica unutrašnje fragmentacije.

Susedne rupe iste veličine predstavljaju drugove, dok susedni procesi to nisu. Kada neki proces završi rad, particija u kojoj se proces nalazio postaje rupa. Rupa se spaja sa susedom, odnosno svojim drugom, ukoliko on postoji, čime se formira duplo veća. Važno je da veličina novoformirane rupe bude neki stepen broja dva - ukoliko to nije slučaj, spajanje se ne vrši.

Algoritmi za izbor prazne particije

Alokacija novog procesa zavisi od trenutne slike memorije. Po pravilu, postoji više šupljina koje se mogu iskoristiti za smeštaj procesa, a u tom kontekstu postoji i više algoritama za alokaciju šupljina. Objasnićemo sledeća tri:

[1] Prvo podudaranje (*first-fit*).

Procesu se dodeljuje prva dovoljno velika šuplina. Pretraživanje se odvija ili od početka liste, ili se nastavlja prethodno pretraživanje (*next fit*). Pretraživanje staje kada se nađe prva odgovarajuća šupljina bez obzira što ima i boljih rešenja, ali je vreme pretraživanja minimalno, jer se ne pretražuje cela lista.

[2] Najbolje podudaranje (*best-fit*).

Procesu se dodeljuje najmanja šupljina koja je dovoljno velika za smeštaj procesa. Pretraživanje je intenzivno, pretražuje se cela lista, osim ako nije uređena po veličini u rastućem ili opadajućem poretku. Od postojeće šupljine nakon smeštaja procesa ostaje najmanja moguća šupljina.

[3] Najgore podudaranje (*worst-fit*).

Procesu se dodeljuje najveća moguća šupljina, što zahteva pretraživanje cele liste. Međutim, kod ove tehnike, od postojeće šupljine nakon smeštaja procesa ostaje najveća moguća šupljina koja je povoljna za smeštanje drugih velikih procesa.

Simulacije pokazuju da su *first-fit* i *best fit* algoritmi bolji i po performansama i po iskorišćenosti memorije od metode najgoreg podudaranja. Što se tiče prvog i najboljeg podudaranja, prvo je svakako brže, a koje je bolje, diskutabilno je. Naravno, sve zavisi od raspodele procesa u vremenu.

Kod multiprogramiranja sa particijama promenljive veličine prisutna je i eksterna fragmentacija - slobodna memorija se ne može iskoristiti za smeštaj procesa ako su preostale šupljine male u odnosu na proces. Problem eksterne fragmentacije može biti ozbiljan. Jedan način za rešenje ovog problema je suspendovanje aktivnog procesa i prebacivanje na disk (*swap-out*), čime se oslobađa memorija koju je proces zauzeo i formira duplo veća šupljina po principu spajanja drugova. Drugi metod koji se koristi je mnogo ozbiljniji i naziva se kompakcija memorije.

Eksterna fragmentacija i kompakcija memorije

Prilikom dodele memorije procesima javljaju se dve vrste fragmentacije memorije:

[1] Unutrašnja (interna) fragmentacija

Unutrašnje fragmentacije se javlja ukoliko se procesu dodeli memorijska particija veća od memorije koju zahteva proces. Preostala memorija je neupotrebljiva.

[2] Spoljašnja (eksterna) fragmentacija.

Memorijski prostor nije kontinualan i kao takav ponekad ne može da zadovolji zahtev procesa, iako je ostalo dovoljno slobodne memorije.

Eksterna fragmentacija se može smanjiti kompakcijom memorije. Cilj kompakcije je da se ispremešta sadržaj memorije kako bi se dobile veće šuplljine, odnosno kontinualan prazan prostor. Kompakcija nije uvek moguća - na primer, ako je relokacija programa statička u fazi prevođenja i punjenja, kompakcija nije moguća. Kompakcija je moguća ako je relokacija programa dinamička i radi se u vreme izvršavanja.

Kompakcija svakako unosi degradaciju performansi sistema, jer sistem mora da prekida procese i da ih privremeno prebacuje na disk, što je vremenski zahtevno. Najprostiji algoritam za kompakciju potiskuje procese ka vrhu memorije, a šupljine se pojavljuju u nižim delovima memorije.

Problem eksterne fragmenatcije se takođe može rešiti primenom tehnika diskontinualne alokacije memorije.

7.4. Straničenje

Straničenje je metoda sa hardverskom podrškom na nivou procesora koja se koristi u svim operativnim sistemima i na svim računarskim arhitekturama. Gotovo da ne postoji ni jedan savremeni procesor, koji hardverski ne podržava straničenje.

Funkcionisanje metode straničenja prikazano je na slici 7.6.

Fizička memorija, ondnosno fizički adresni prostor se izdeli na blokove fiksne veličine koji se nazivaju fizičke stanice ili okviri (*page frames*). Logički adresni prostor se takođe izdeli na blokove istih veličina koji se nazivaju logičke stranice (*pages*). U daljem tekstu pod terminom stranice podrazumevaćemo logičku a pod terminom okvir fizičku stanicu. Veličine stranica su po pravilu stepen broja 2, najčešće u opsegu 512B-8KB, mada mogu biti i veće - na primer, 16MB. *Swap* prostor na disku se takođe deli na stranice koje po veličini odgovaraju memorijskim stranicama.

Metod straničenja funkcioniše na sledeći način. Svakoj logičkoj stranici odgovara jedna fizička a korespodencija između njih se čuva u tabeli stranica (*page table*). Na ovaj način kontinualni logički prostor procesa može da se razbaca svugde po memoriji, na nivou granularnosti od jedne stanice.

Svaka logička adresa koju generiše procesor se deli na dva dela:

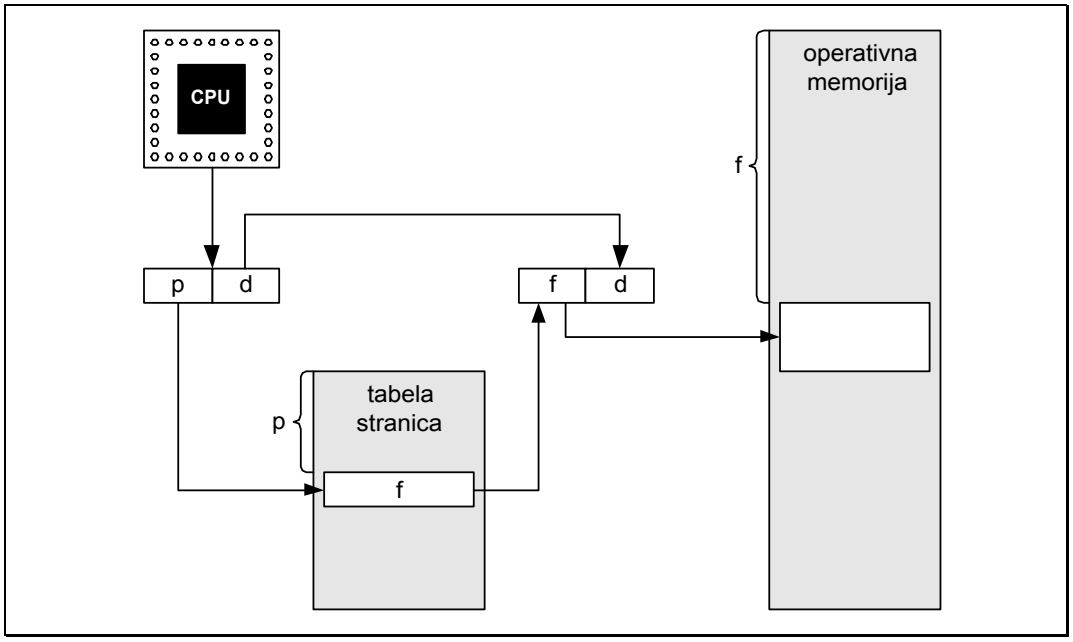
- Broj stranice (p, *page number*)

  Koristi se kao indeks u tabeli stranica koja sadrži baznu adresu okvira. Bazna adresa predstavlja viši deo adrese.

- Pomeraj unutar stranice (d, *page offset*)

  Definiše položaj u odnosu na samu stranicu i u kombinaciji sa baznom adresom (čisto sabiranje) definiše punu fizičku adresu koja se šalje memorijskoj jedinici. Naglasimo da je osnovna adresibilna jedinica bajt i da je pomeraj isti i za logičku i fizičku adresu.

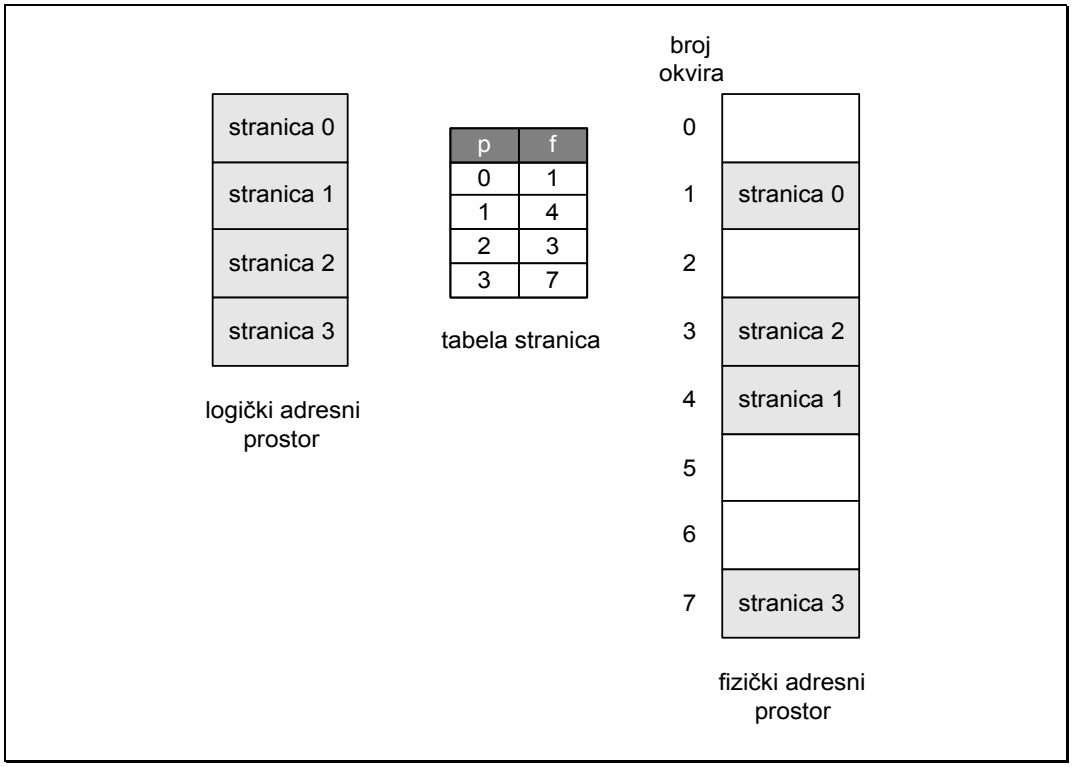
Ako je veličina logičkog adresnog prostora 2^m, a veličina stanice 2^n, tada viši deo adrese dužine m-n definiše broj stranice, dok najnižih n bita adrese predstavljaju pomeraj unutar stranice.



Slika 7.6. *Straničenje*

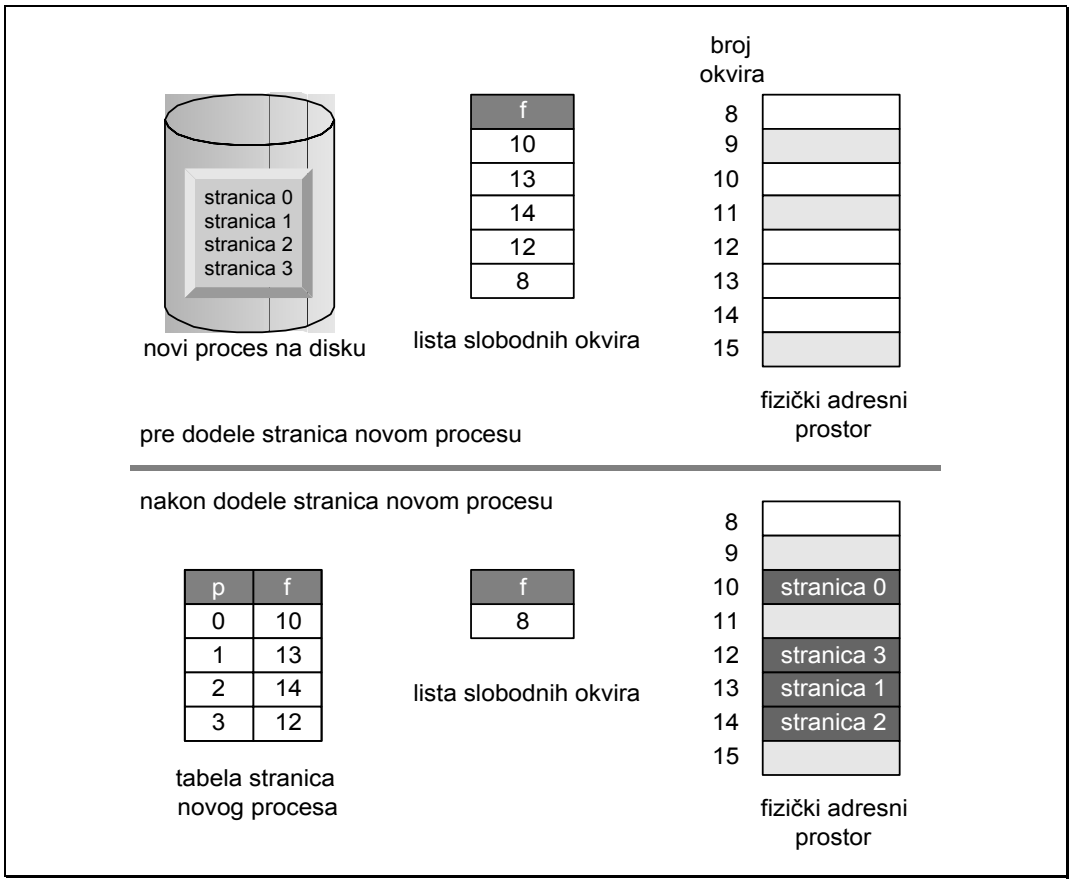
Tehniku straničenja ilustrovaćemo sledećim primerom: zamislimo da imamo memoriju veličine 32B. Definišimo 8 okvira veličine 4B, što znači da je m=6, a n=2. Uzmimo korisnički proces koji zauzima 4 logičke stranice, sa logičkim adresama: stranica 0 (0-3), stranica 1 (4-7), stranica 2 (8-11) i stranica 3 (12-15). Interpretirajmo neke logičke adrese. Logička adresa 0, ima broj logičke stranice 0 i pomeraj 0. Pomoću broja logičke stranice ulazimo u tabelu stranica i nalazimo da je ona smeštena u okvir broj 1. I tako redom: logička stranica 1 je smeštena u okvir 4, logička stanica 2 u okvir 3 i logička stranica 3 u okvir 7 (slika 7.7). Na primer, logička adresa 5 se mapira na

sledeći način: adresa 5 je logička stranica 1 sa pomerajem 1, a na osnovu tabele stranica to okvir 4 sa pomerajem 1, adresa je 16+1=17.



Slika 7.7. *Primer straničenja*

Proces koji ulazi u stanje izvršavanja mora da dobije potrebnu memoriju, tako da se za dati proces preračunava koliko mu stranica memorije treba. Svaka stanica mora da se mapira u okvir. Ako proces zatheva n stranica, tada se alocira n okvira, koji se pune procesom, pri čemu se mapiranje stranica-okvir upisuje u tabelu stranica (kao na slici 7.8., gde se alokacija vrši na osnovu liste slobodnih okvira). Kao posledica, raspored zauzetih i slobodnih stranica je slučajan.



Slika 7.8. *Dodela stranica procesu na osnovu tabele slobodnih okvira*

Svako straničenje predstavlja dinamičku relokaciju, a tabela stranica predstavlja relokacioni registar za svaki okvir fizičke memorije. Na ovaj način, straničenje potpuno eliminiše eksternu fragmentaciju, jer se svaki okvir može iskoristiti za smeštaj dela bilo kog procesa. Međutim, može se pojaviti izvesni stepen interne fragmentacije ukoliko proces zahteva količinu memorije različitu od celog broja stranica. Kako se procesu mora dodeliti ceo broj stanica deo jednog okvira će ostati neiskorišćen. U optimalnom slučaju gubitak je 0, dok se najveći gubitak javlja ukoliko proces zahteva količinu memorije jednaku celom broju stranica uvećanom za jedan. Tada se zbog jednog bajta gubi ceo okvir. Uprošćeno rečeno, srednji gubitak memorije po procesu iznosi ½ stranice, što ukazuje da stranice treba da budu što manje. S druge strane, smanjenje veličine stranica dovodi do povećanja tabele stranica, jer je svaka stranica opisana jednim zapisom u tabeli. Kao posledica, povećava se kašnjenje prilikom mapiranja, odnosno pretraživanja. U slučaju korišćenja *swap* tehnike sa stranicama, poželjnije su veće stranice zato što je rad sa diskovima efikasniji kada su transferi veći.

Generalno gledano, veličine procesa i struktura podataka rastu u vremenu, što za posledicu ima porast količine operativne memorije i veličine stranica. Tabele stranica su, uglavnom, 32-bitne, mada teže ka 64-bitnim strukturama. Na primer, ako je tabela stranica 32 bitna a veličina stranice 4KB, može se adresirati $2^{32}*4*2^{10}$=16TB fizičkog adresnog prostora.

Značajan aspekt straničenja je jasno razdvajanje korisničkog pogleda na memoriju i aktulene fizičke memorije. Korisnik svoj deo memorije doživljava kao kontinualni prostor iako su stranice raznih procesa razbacane po memoriji. Mapiranje logičkog i fizičkog prostora je zadatak operativnog sistema i transparentno je za korisnika. Operativni sistem mora da prati koji su okviri slobodni, a koji su dodeljeni i to kom procesu. Sve se to čuva u tabeli okvira (*frame table*) u kernelu, u kojoj je svaki okvir opisan jednim slogom. Na drugoj starni, operativni sistem mora za svaki proces da generiše tabelu stranica koja se odnosi samo na njegove stranice i koja definiše mapiranje za taj proces.

Hardverska podrška

Iako cela priča o staničenju zvuči dobro, straničenje realizovano pomoću struktura u operativnoj memoriji, ili, još gore, u kombinaciji memorija-disk je jako sporo i neefikasno. Ukoliko su veličine stranica male, broj stranica će biti veliki, a samim tim će biti velika i tabela okvira. Takođe, veliki broj procesa povlači i veliki broj tabela stranica za procese. Održavanje svih tih struktura, u šta spada čitanje i ažuriranje, zahteva veliku količinu vremena koja predstavlja čisto premašenje. Na svu sreću, hardver samih procesora drastično pomaže u realizaciji tehnika straničenja.

Svaki operativni sistem ima sopstevenu šemu za realizaciju tabela stranica. Pored glavne tabele, većina operativnih sistema ima za svaki proces posebnu tabelu. Idealno mesto za smeštaj tabele stranica su registri procesora kao najbrže komponente, ali je broj registara nedovoljan. Zato se tabele stranica, koje su relativno velike, čuvaju u operativnoj memoriji, pri čemu obično postoji poseban par registara:

- PTBR (*page-table base register*) - ukazivač na tabelu stranica,
- PRLR (*page-table length register*) ukazivač na veličinu tabele stranica.

Svaka korisnička memorijska referenca zahteva da se pre pristupa konkretnom podatku izvrši čitanje tabele stranica, što, praktično, duplo umanjuje performanse memorijskih ciklusa. Ovaj problem je rešen na nivou samih mikroprocesora, uvođenjem specijalne keš struktura - TLB (*translation look-aside buffers*). TLB je veoma brza komponenta, realizivana na principu asocijativne memorije. Svaki ulaz u TLB sastoji se od dva dela: ključa (*key*, *tag*) i vrednosti. Kod asocijativne memorije svi izlazi istovremeno upoređuju svoj ključ sa ulaznim ključem i ako postoji identifikacija, asocijativna memorija vraća vrednost. U ovom slučaju, ključ je logička adresa stranice, a vrednost je fizička adresa okvira koji joj odgovara. Asocijativna memorija je brza, ali skupa, tako da je broj ulaza relativno mali (obično od 64 do 1024).

TLB se koristi sa tabelama stranica kao keš za najčeće korišćene memorijske reference. Ukoliko se referenca pronađe u TLB (*TLB hit*), adresa okvira se dobija gotovo trenutno i prelazi se na pravu memorijsku referencu. Ali, ako se dogodi promašaj (*TLB miss*), tada se par vrednosti za tekuću referencu mora dovesti u TLB bafer. TLB se ponaša kao i svi keš sistemi: brzo radi u slučaju pogotka (*cache hit*), a osetno sporije u slučaju promašaja (*cache miss*). Statistički, sistem sa visokim odnosom pogodaka radi mnogo brže nego sistem koji ne koristi TLB. Povremeno se neki elementi izbacuju iz TLB da bi se upisali novi (LRU algoritam).

U nekim TLB struktruma se koristi ASID (*adress-space idenifiers*) identifikacija kao dodatak TLB informacijama, koja je unikatna za svaki proces i služi za brže pronalaženje informacija o tekućem procesu. Istovremeno, ASID identifikacija je dobra jer omogućava da se u TLB kešu čuvaju informacije za više procesa u jednom trenutku. Bez ASID identifikacije, ceo TLB mora se briše (*flushing*) čim dođe novi proces, da se ne bi mešale informacije o mapiranju. Ako ASID tekućeg proceasa ne odgovara ni jednom u TLB kešu, to je promašaj, a ako odgovara, onda se dalje upoređuju ključevi.

Vreme pristupa u sistemu sa TLB je različito i zavisi da li je pogodak ili promašaj. Pretpostavimo da je vreme asocijativnog pretraživanja (*associative lookup*) $t_{AL}=\varepsilon$ vremenskih jedinica (najčešće, nano sekundi). Pretpostavimo da je vreme trajanja jednog memorijskog cilkusa t_{MC} vremenskih jedinica (najčešće, mikro sekundi). Vreme memorijskog ciklusa u slučajevima pogodka i promašaja u TLB kešu su:

$$t_{HIT} = \varepsilon + t_{MC}$$

$$t_{MISS} = t_{MC} + \varepsilon + t_{MC}$$

Definišimo α kao odnos broja pogodaka i promašaja u TLB kešu (*hit ratio*). Efektivno vreme pristupa (*effective access time*) je:

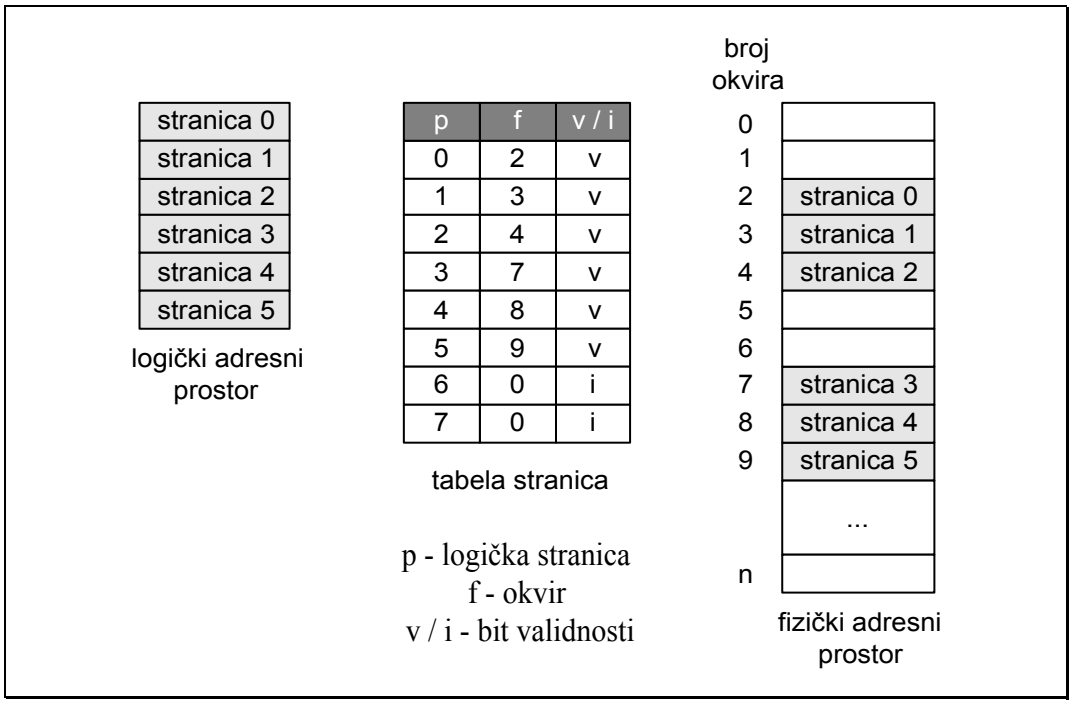
$$t_{EFF} = \alpha(t_{MC} + \varepsilon) + (1 - \alpha)(2t_{MC} + \varepsilon) = 2t_{MC} + \varepsilon - \alpha t_{MC}$$

Zaštita

Zaštita memorije u sistemu straničenja postiže se uvođenjem novih bitova koji imaju specijalno značenje u tabelu stranica. Dva tipična bita su bit validnosti v, i bit za zaštitu od upisa rw (odnosno ro).

Bit validnosti v ukazuje da je vrednost u tabeli stranica važeća, ispravna i da se okvir može se koristiti za mapiranje. To se ponekad koristi da se nekom programu zabrani pristup određenim logičkim adresama, kao što je prikazano na slici 7.9.

U primeru sa slike, svaka referenca programa koja pripada logičkim stranicama 6 i 7 (logičke adrese veće od 12287, pri veličini stranice 2KB), završiće se kao ilegalan zahtev, zato što su te stranice u tabeli proglašene kao nevažeće (invalid bit).



Slika 7.9. *Bit validnosti*

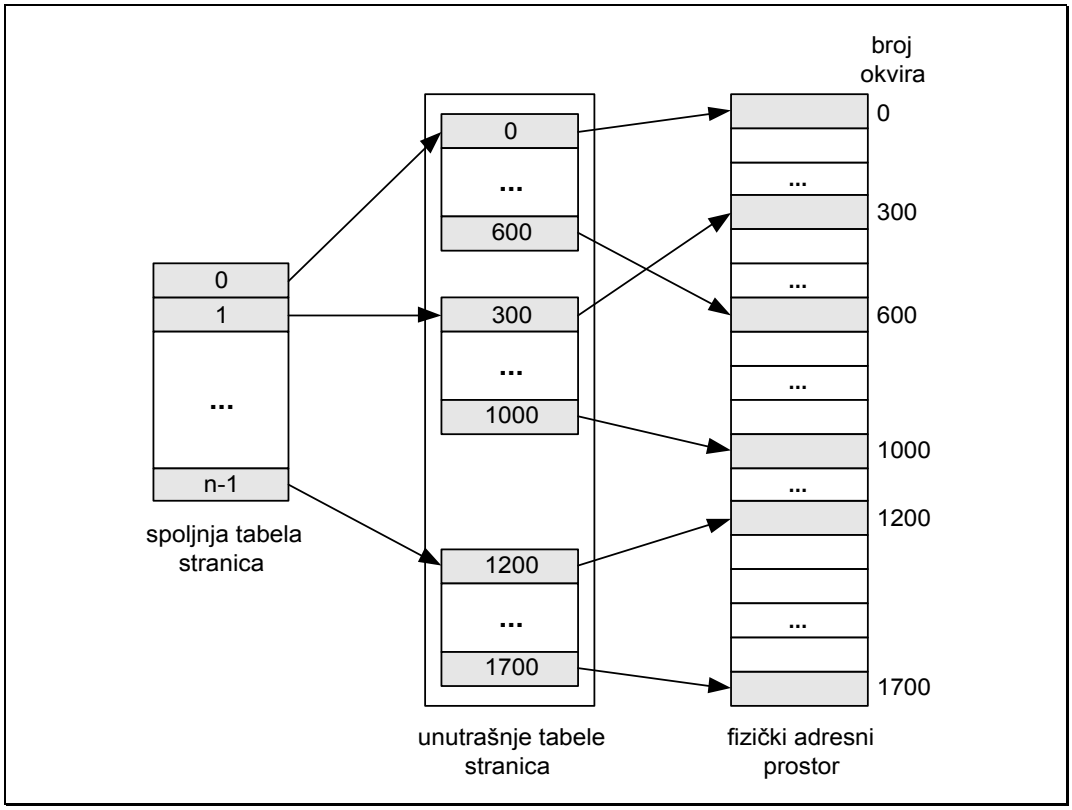
Bit rw određuje da li je stranica sa punim pristupom koji uključuje čitanje i pisanje ili je stranica zaštićena, pa se može samo čitati. Veoma je poželjno zaštititi stanice koje sadrže kod programa čime se sprečava nehotično prepisivanje.

Hijerarhijsko straničenje

Većina savremenih računara podržava širok logički adresni prostor, (2^{32} ili 2^{64}). U takvim situacijama, tabele stranica su jako velike. Na primer u 32 bitnom sistemu, sa stranicama veličine 4KB (2^{12}), tabela stranica se sastoji od milion ulaza ($2^{32} / 2^{12} = 2^{20}$). Ako se svaki ulaz sastoji od 4 bajta, tada tabela stanica obuhvata 4MB. Međutim, ne treba svakom procesu milion ulaza, odnosno ceo logički adresni prostor, tako da se tabela stranica može podeliti na više manjih delova. Najjednostavniji primer je dvonivovsko straničenje, prikazano na slici 7.10.

Kod ove metode postoji spoljna tabela stranica koja ukazuje na stranice u kojima su smeštene prave tabele stranica. Uzmimo opet primer 32-bitnog sistema sa stranicama veličine 4KB. U normalnom kontekstu (jednonivovsko straničenje) 12 nižih bita predstavljaju pomeraj, a 20 viših adresu stranice. Ukoliko se koriste dva nivoa tabela, 20 bita se mogu podeliti na bite koji ukazuju na spoljne i unutrašnje tabele u proizvoljnom odnosu (na primer, 10-10). Tada će logička adresa imati sledeću interpretaciju: najviših 10 bita predstavljaju indeks kojim se iz spoljne tabele odabira

jedna od unutrašnjih stranica sa pravim tabelema. Sledećih 10 bita adrese predstavlja poziciju u odabranoj untrašnjoj tabeli iz koje se čita ukazivač na odgovarajući okvir.



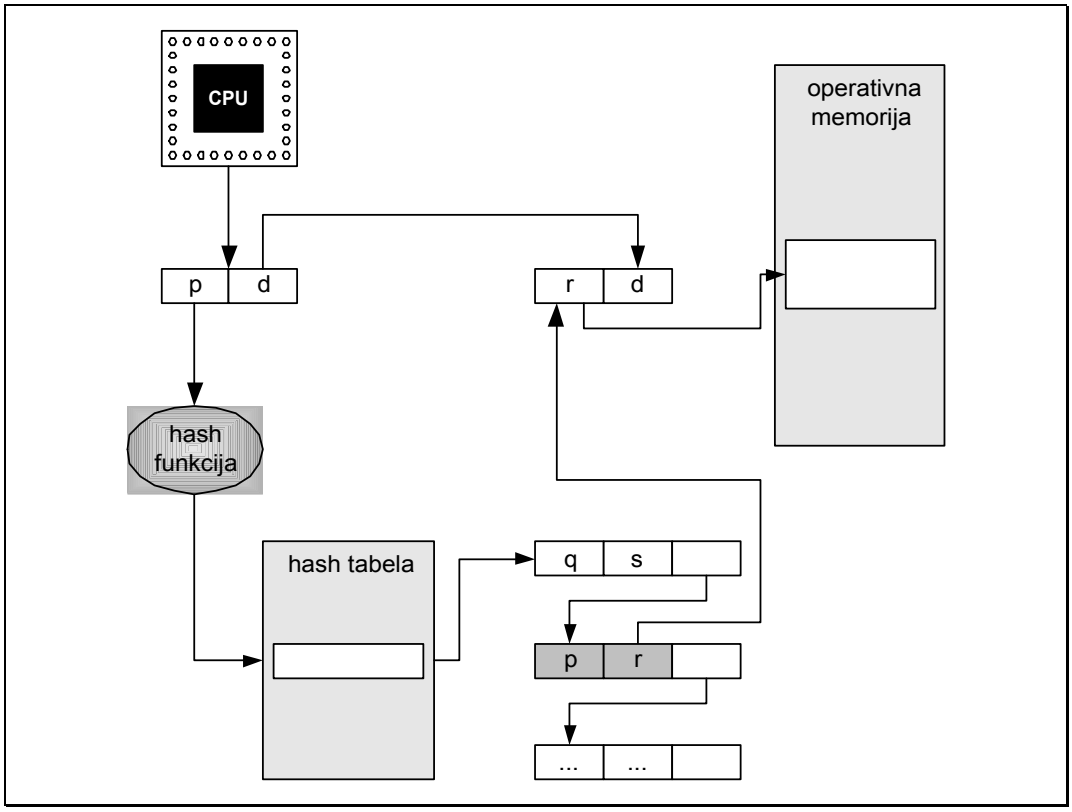
Slika 7.10. *Dvonivovsko straničenje*

Heš (hash) bazirane tabele stranica

Hijerarhijski nivoi su krajnje nepraktični za 64 bitne sisteme, jer se zahteva veliki broj nivoa. Heš je mehanizam koji se često koristi u keš sistemima zato što omogućava brzo pretraživanje. U tom slučaju koriste se heširane tabele podataka, gde heš algoritam izračunava vrednost virtuelnog broja stranice. Svaki ulaz u heš tabelu sadrži povezanu listu elemenata koji imaju istu vrednost heš funkcije. Svaki heš element ima tri polja:

- virtuelni broj stranice,
- vrednost mapirane fizičke stranice i
- ukazivač na sledeći elemenat u listi.

Svaka povezana lista ima precizno definisano mesti u heš tabeli. Heš šema je prikazana na sledećoj slici.



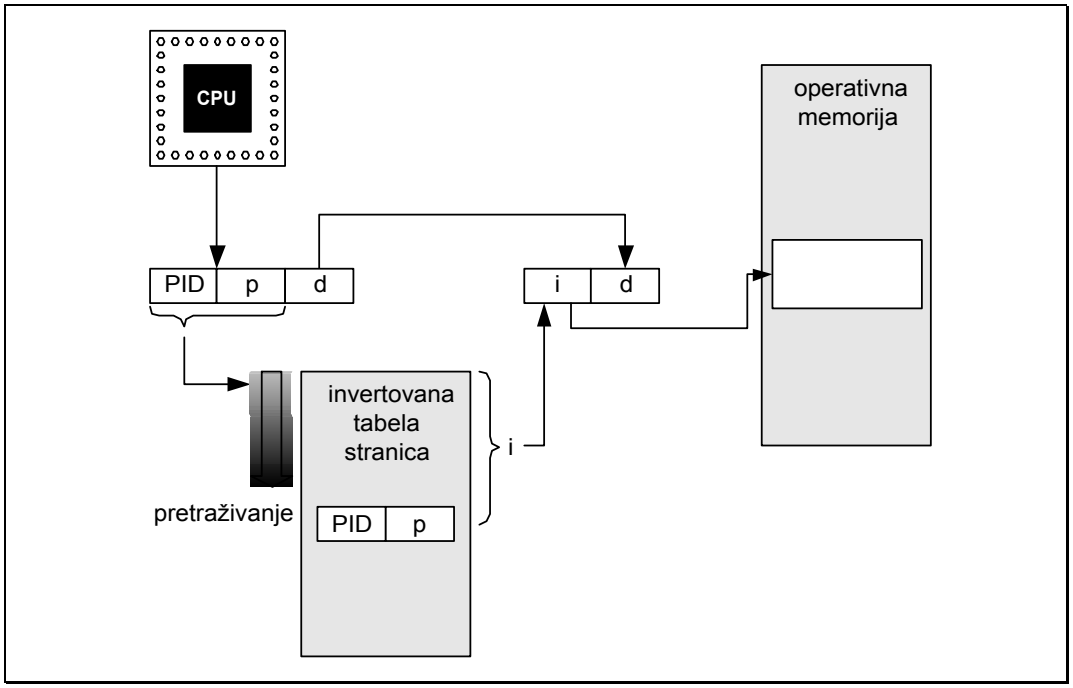
Slika 7.11. *Heš bazirane tabele stranica*

Heš šema funkcioniše na sledeći način: na bazi logičke adrese izdvoji se p deo i propušta kroz heš funkciju koja generiše virtuleni broj stranice. Na bazi tog broja ulazi se u heš tabelu i čita se lista povezanih elemenata koja sadrži vredsnosti keširanih parova. Ukoliko se virtuelni broj poklapa sa nekim od virtuelnih brojeva iz povezane liste, tada je to pogodak u kešu i dobijamo vrednost fizičke stranice. U protivnom, to je promašaj, pa se u povezanu listu upisuje novi član povezane liste. Heš funkcija selektuje samo odgovarajući deo tabele i pretraživanje je osetno ubrzano.

Invertovane tabele stranica

Svaki proces ima dodeljenu tabelu stranica, a tabele stranica mogu biti velike. Problem pojavljivanja većeg broja velikih tabela rešava se metodom invertovanih tabela stranica. Invertovana tabela ima jedan ulaz za svaki okvir ili fizičku stranicu memorije, koji se sastoji od virtuelene adrese stranice koja joj je pridružena, ali i sa informacijom

o procesu koji ju je dobio. Na taj način, u sistemu imamo samo jednu tabelu stranica, a ne više tabela za svaki proces pojedinačno.

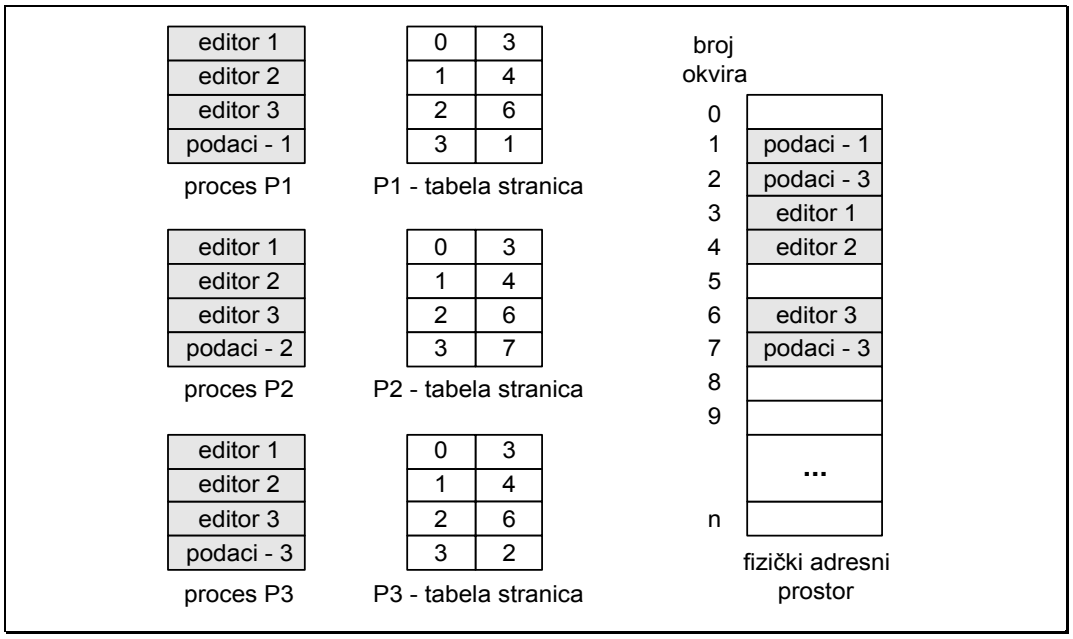


Slika 7.12. *Invertovane tabele stranica*

Zapaža se da CPU u logičkoj adresi, pored virtuelne adrese (p+d), mora da pruži informaciju o procesu PID, sa kojim se obavlja pretraživanje u tabeli u kojoj se takođe nalazi PID procesa. Pretraživanje može biti intenzivno, jer mora da se pretraži cela tabela, pa se, po pravilu, uvodi heš funkcija.

Deljive stranice

Jedna od prednosti straničenja je mogućnost deljenja zajedničkog koda između većeg broja procesa, što je veoma povoljno za interaktivne sisteme. Uzmimo za primer sistem na kome više korisnika istovremeno izvršava isti tekst editor. Kada ne bi bilo deljenja, svaki korisnički proces, pored sekcija podataka morao bi da učita za sebe posebnu kopiju koda tekst editora u memoriju. Ako je kod editora re-entrant tipa, što znači da se nikada ne modifikuje u toku izvršavanja, deljenje stranica se može iskoristiti vrlo efikasno: jednu kopiju koda u memoriji koristiće svi procesi - njihove tabele stranica ukazivaće na isti memorijski prostor u kome se nalazi tekst editor. Primer na slici ilustruje slučaj za tri korisnika.



Slika 7.13. *Deljive stranice*

Svakom procesu potrebne su tri stranice za kod editora teksta (ed1, ed2, ed3) i jedna stranica za privatne podatke. Operativni sistem će tri stranice edtora koje su na fizičkim lokacijama 3, 4 i 6 učiniti dostupnim svim procesima, tako što će njihove logičke stanice mapirati u isti memorijski prostor (3,4,6). Četvrta stanica je privatna i mora biti unikatna za svaki proces.

Ova tehnika je pogodna za slučajeve često korišćenih programa, kao što su tekst editori, prevodioci, grafički sistemi, sistemske biblioteke i baze podataka.

7.5. Segmentacija

Prethodno razmatrani slučajevi odnose se na korisničke procese, odnosno programe, koji zahtevaju kontinualni memorijski prostor. Međutim, sama struktura programa nije kontinualna, jer se programi logički dele na više nezaviznih celina. Na primer, sam kod se sastoji od glavnog programa, procedura, funkcija i modula. Podaci se takođe sastoje od više celina kao što su tabele, polja, stek, promenljive itd. Svaki od ovih logičkih segmenata dobija ime pomoću koga se referencira i može nezavisno da se učita u memoriji. Po pravilu, programer definiše logičke segmente u izvornom programu, a prevodioc na osnovu toga kreira memorijske segmente.

Segmentacija je metoda upravljanja memorijom koja podržava logički korisnički pogled na memoriju. Logički adresni prostor se sastoji kolekcije segmenata, a svaki segment ima jedinstveno ime i dužinu. Logička adresa se se sastoji od dva dela:

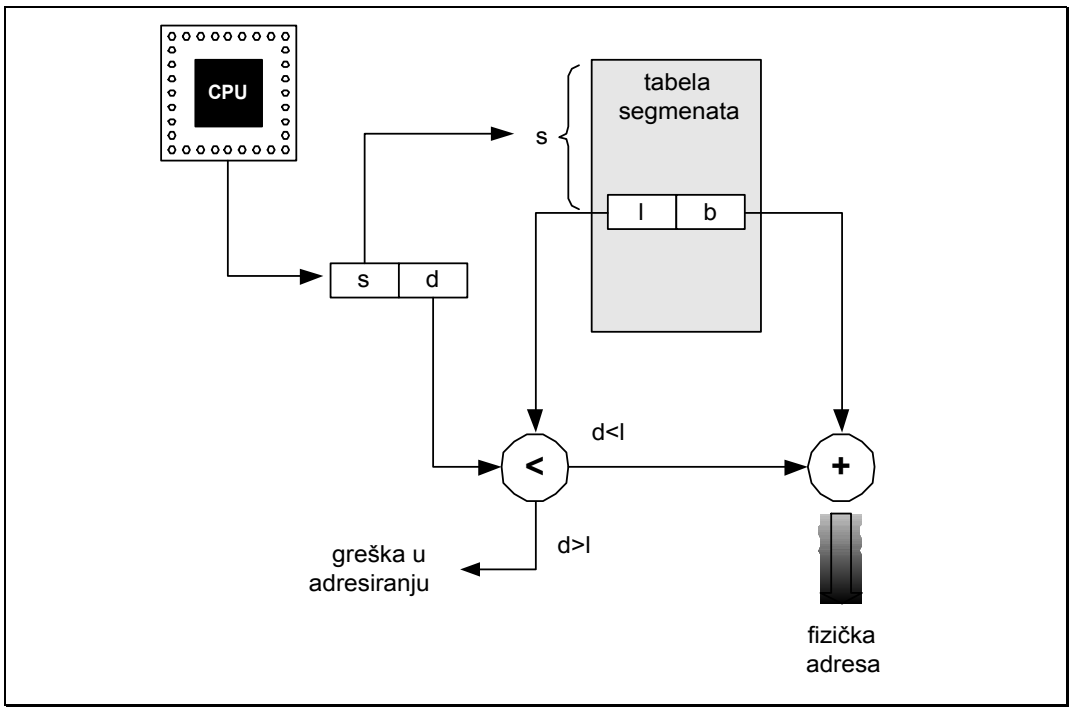
- imena segmenta (umesto imena segmenta obično se zadaje broj koji predstavlja identifikator segmenta) i
- pomeraja unutar segmenta.

Podsetimo da je kod straničenja logička adresa jednodimenziona, pri čemu razdvajanje delova adrese i transliranje obavlja hardver.

Kod segmenatacije je, kao i kod multiprogramiranja sa particijama promenljive dužine, prisutna eksterna fragmentacija. Slobodna memorija se ne može iskoristiti za smeštaj segmenta ukoliko ne postoji dovoljno velika šupljina, bez obzira na ukupnu količinu slobodne memorije. Problem eksterne fragmentacije, koja zavisi od veličine segmenata i raspodele nailaska procesa može se umanjiti kompakcijom memorije.

Hardverska podrška

Korišćenje segmenata je ilustrovano na slici 7.14.



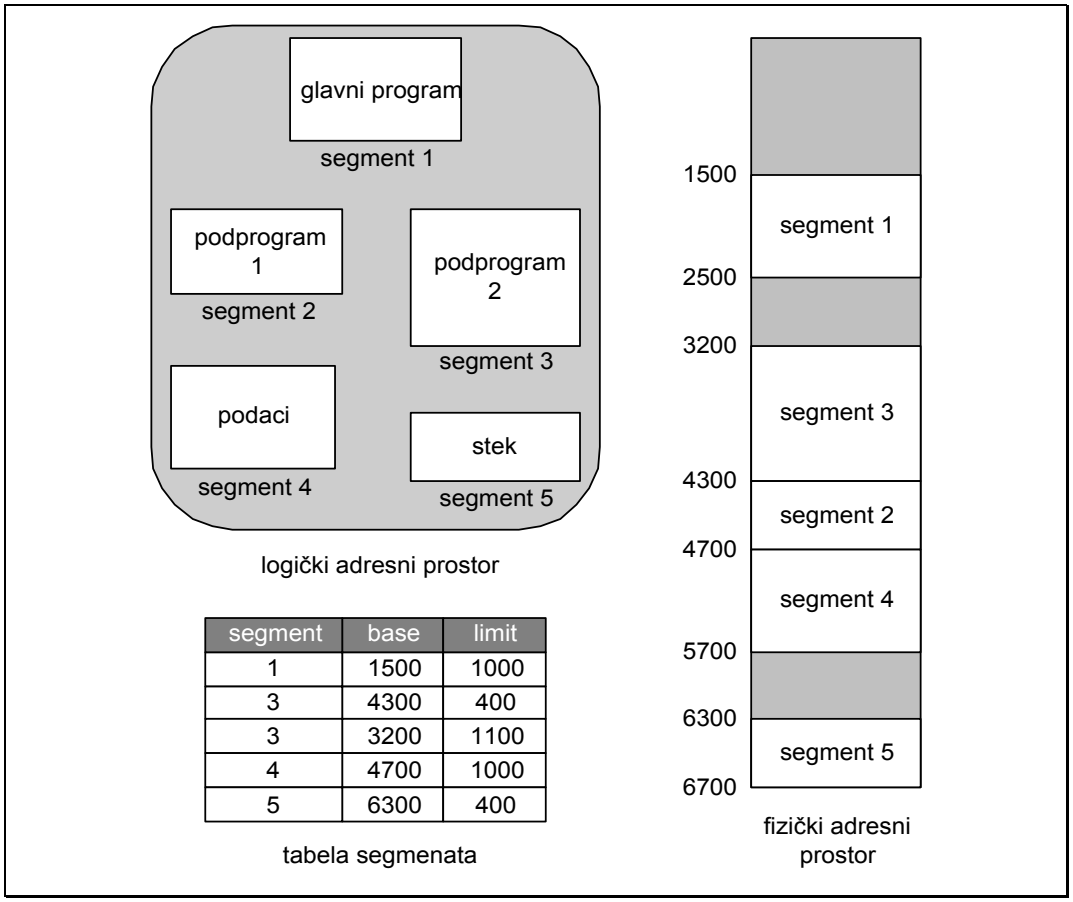
Slika 7.14. *Segmentacija*

U metodi segmentacije, korisničke adrese su dvodimezione, ali se moraju translirati u jednodimenzione fizičke adrese. Ovo mapiranje se obavlja preko tabele segmenata a takođe ima podršku u hardveru samih mikroprocesora. Svaki ulaz u tabeli segmenata opisuje tačno jedan segment, a sadrži dva parametra:

- baznu adresu segmenta, koja defniše početnu fizičku adresu segmenta u memoriji,
- limit segmeta, koji definiše dužinu segmenta.

Broj segmenta s se koristi kao ulaz u tabelu segmenta, iz koje se čitaju bazna adresa segmenta i limit. Pomeraj d svakako mora biti manji u odnosu na limit segmenta.

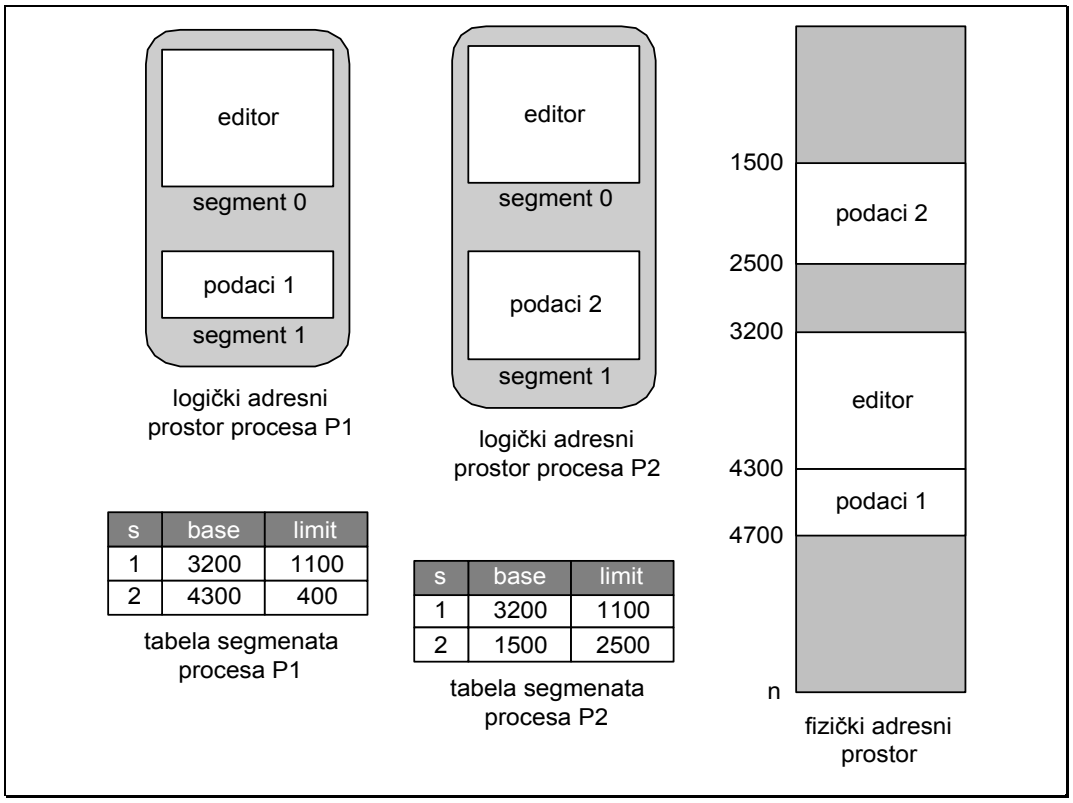
Na slici 7.15. prikazan je slučaj segmentacije za 5 segmenata, čije su definicije date su tabeli segmenata.



Slika 7.15. *Primer segmentacije*

Zaštita i deljenje segmenata

Tabela segmenata, pored definicije segmenta u fizičkoj memoriji može sadržati i neke definicije zaštite od kojih se najčešće koristi zaštita od modifikacije. Segmenti sa kodom mogu biti definisani kao nepromenljivi (*read only*). Slično kao kod metode straničenja, i segmenti koji se ne menjaju se mogu deliti i na taj način se obavlja efikasna ušteda memorije. Na slici je prikazano deljenje segmenata za slučaj dva procesa koji koriste isti editor teksta. Svaki proces ima svoju tabelu segmenata, a segment koji sadrži kod editora, biće zajednički.



s	base	limit
1	3200	1100
2	4300	400

tabela segmenata procesa P1

s	base	limit
1	3200	1100
2	1500	2500

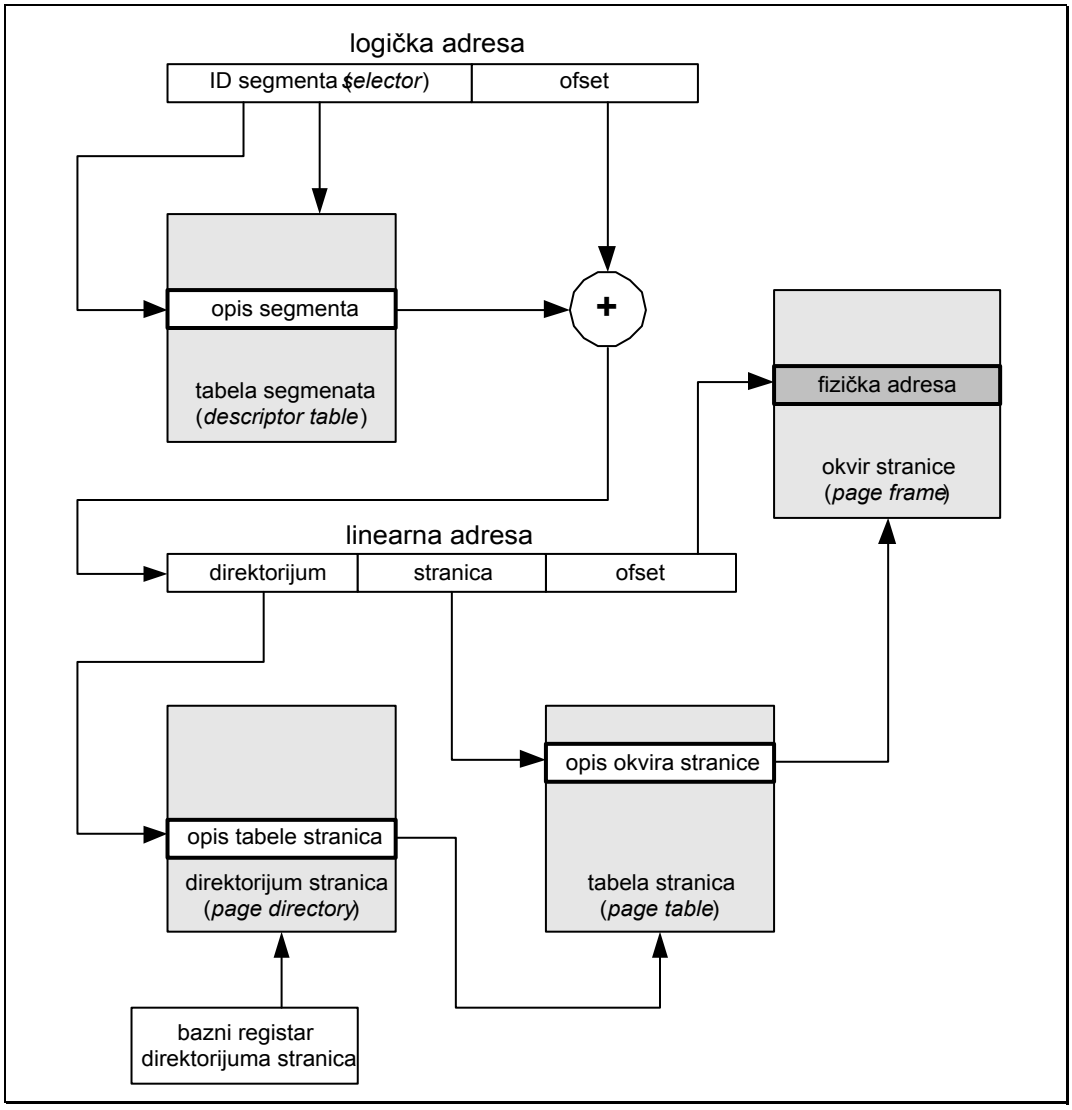
tabela segmenata procesa P2

Slika 7.16 *Deljenje segmenata*

Segmentacija sa straničenjem

Najpopularnije serije procesora Intel (80x86 i Pentuim) i Motorola (68000) imaju ugrađenu podršku i za segmentaciju i za straničenje, tako da omogućavaju primenu kombinovanih metoda diskontinualne alokacije čime je omogućena podela procesa na fizički diskontinualne logičke celine. Pri tome, straničenje poništava eksternu

fragmentaciju segmenata. Procesor Intel 80386 koristi segmenaciju sa straničenjem, koja je prikazana na slici 7.17. Logička adresa se sastoji iz identifikatora segmenta (*selector*) i pomeraja u okviru segmenta (*offset*). Iz tabele segmenta se čita adresa logičke stranice u direktorijumu stranica. Straničenje je realizovano u dva nivoa: spoljna tabela se zove direktorijum stranica (*page directory*) a unutrašnja, jednostavno, tabela stranica (oba dela adesese su 10 bitna).



Slika 7.17. *Segmentacija sa straničenjem*

8

Virtuelna memorija

Sadržaj poglavlja:

8.1. Uvodne napomene
8.2. Učitavanje stranica prema potrebi
8.3. Alternativne tehnike učitavanja stranica
8.4. Zamena stranica
8.5. Raspodela okvira po procesima i efekat zasićenja
8.6. Dopunska razmatranja

8.1. Uvodne napomene

Virtuelna memorija je tehnika koja dozvoljava izvršavanje procesa čiji delovi mogu biti smešteni na sekundarnim memorijama, odnosno diskovima. Virtuelna memorija formira apstrakciju u vidu logičke memorije, koju čine operativna i sekundarna memorija i razdvaja korisničku logičku memoriju od fizičke. Program više nije ograničen količinom raspoložive fizičke memorije, tako da programeri mogu da pišu programe bez formiranja *overlay* komponenti podrazumevajući ekstremno veliki adresni prostor. Pri tome, virtuelna memorija omogućava izvršenje programa većih od same fizičke memorije. Koncept virtuelne memorije dozvoljava smeštaj osetno većeg broja procesa u memoriju (konkretno, delova procesa), čime se povećavaju iskorišćenje i propusna moć procesora, a bez povećanja vremena odziva (*response time*) i vremena izvršavanja (*turnaround time*). Takođe, očekuje se manji broj ulazno-izlaznih operacija vezanih za punjenje ili zamenu korisničkih programa u memoriji, što znači da bi sami programi trebalo da rade brže. Virtuelna memorija omogućava deljenje datoteka i

memorije između različitih procesa na isti način kao kod deljenja stranica, što omogućva uštedu memorije i povećanje performansi.

Virtuelna memorija se najčešće realizuje tehnikom učitavanja stranica prema potrebi (DP, *demand paging*), a može se upotrebljavati i tehnika učitavanja segmenata prema potrebi (*demand segmentation*), koja je dosta složenija, kao i kombinovana tehnika.

8.2. Učitavanje stranica prema potrebi

Sistem sa učitavanjem stranica prema potrebi (DP, *demand paging*) veoma je sličan straničenju sa razmenjivanjem (*swap*). I memorija i prostor na disku koji se koristi za razmenjivanje su izdeljeni na stranice. Procesi su smešteni na disku u *swap* prostoru i prebacuju se u memoriju kada dođu na red za dodelu procesora.

Prvi sistemi koji su koristili *swap* tehniku, prebacivali su isključivo cele procese iz *swap* prostora u fizičku memoriju i obrnuto. Za razliku od njih, DP sistem koristi specifičnu tehniku za razmenivanje: u fizičku memoriju se ne prebacuje ceo proces, nego samo stranice koje se trenutno traže, i to najčešće samo ona koja je neophodna. Na taj način se izbegava nepotrebno čitanje sa diska i smanjuje potrebna količina fizičke memorije. DP može prebaciti i više stranica u memoriju na osnovu pretpostavke o stranicama koje će biti potrebne procesu. DP je takozvani "lenji razmenjivač" (*lazy swapper*), jer nikada ne prebacuje stranice u memoriju, osim ako se stranice ne zahtevaju. Proces koji obavlja razmenu naziva naziva se *pager*, a sistemi koji koriste DP - *paging* sistemi.

Neophodan hardver

Hardver neophodan za realizaciju DP tehnike je unija hardvera potrebnog za straničenje i razmenjivanje. Sledeće dve komponente su apsolutno neophodne:

[1] Tabela stranica

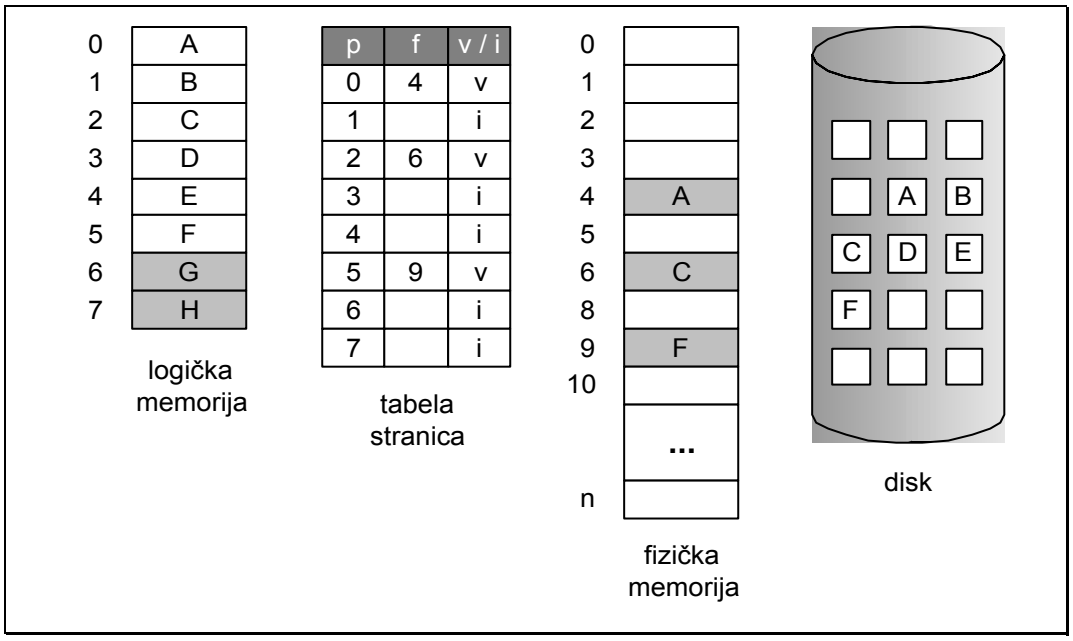
Eksplicitno se zahteva prisustvo bita validnosti u tabeli stranica. Pomoću bita validnosti, koji se čuva u tabeli stranica, može se opisati trenutni položaj stranice. Vrednost bita v (*valid*), ukazuje da se logička stranica nalazi u memoriji. Vrednost bita i (*invalid*), ukazuje da se stranica ne nalazi u memoriji, već na disku u *swap* prostoru. Vrednost i takođe može označavati takođe da je to stranica koja ne pripada adresnom prostoru diska.

[2] Sekundarna memorija

Služi za smeštaj svih stranica koje nisu prisutne u memoriji. Uređaj koji se, po pravilu, koristi je disk.

Prebacivanje stranica sa diska u fizičku memoriju

DP je ilistrovan na slici 8.1, na kojoj je prikazan izgled logičkog adresnog prostora hipotetičkog procesa, kao i odgovarajuće fizičke memorije i *swap* prostora.



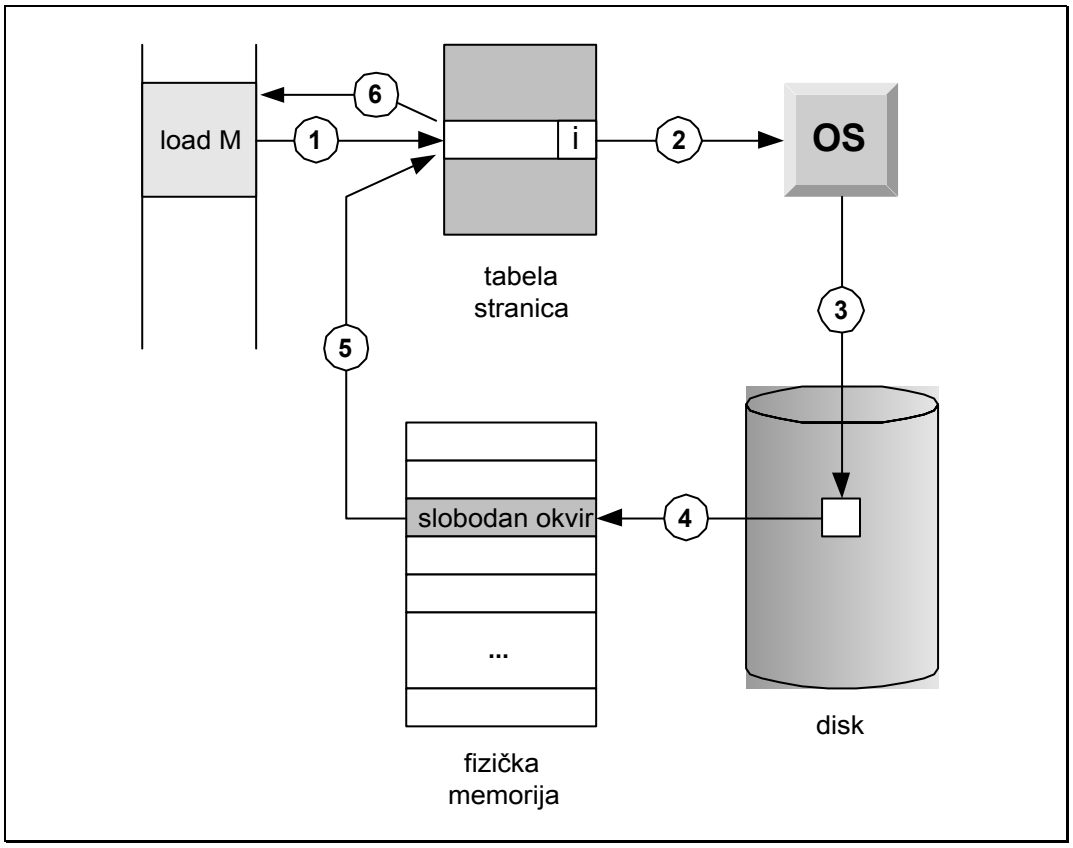
Slika 8.1. *Demand paging sistem*

Analizirajmo sledeće dve situacije:

[1] proces se izvršava ili pristipa stranici koja je u memoriji i

[2] proces pokušava da pristupa stranici koja nije u memoriji nego na disku.

U prvom slučaju, izvršavanje programa teče normalno. U drugom slučaju, proces pristupa logičkoj stranici čija je vrednost bita validnosti i, što izaziva prekidni signal PF (*page-fault trap*). Operativni sistem tada poziva rutinu za opsluživanje PF, koja ima zadatak da stranicu sa diska prebaci u memoriju. Prebacivanje stranice sa diska u memoriju obavlja se u više koraka, kao što je prikazano na slici 8.2.



Slika 8.2. *Prebacivanje stranice sa diska u memoriju*

[1] Referenca (load M) je prouzrokovala prekid PF, jer je prilikom čitanja stranice u tabeli detektovan *invalid* bit.

[2] Operativni sistem poziva sistemsku rutinu za obradu PF. Ukoliko referenca nije validna, proces se prekida jer sadrži pogrešnu instrukciju. Ukoliko je referenca validna, PF započinje učitavanje stranicu u memoriju.

[3] PF rutina pronalazi stranicu na disku, u prostoru za razmenjivanje.

[4] PF rutina traži slobodan okvir u fizičkoj memoriji, nakon čega prebacuje stranicu sa diska u pronađeni slobodan okvir.

[5] PF rutina ažurira tabelu stranica. Na ulazu koji je napravio PF prekid, upisuje se adresa okvira i poništava *invalid* bit (postavlja se v bit).

[6] Prekinura instrukcija koja je uzrokovala PF prekid se izvršava iz početka, s tim što sada ima sve što joj treba u memoriji.

Postupak se ponavlja za svaku stranicu procesa koja nije u memoriji, pri čemu svaki PF prekid učitava samo jednu stranicu sa diska.

DP tehnika koja učitava stranicu u memoriju isključivo posle referenciranja, pri čemu se svaki put prebacuje samo jedna stranica, naziva se čista DP tehnika (*pure demand paging*). Teorijski, jedna instrukcija bi mogla da napravi više PF prekida, ali se takve stvari retko dešavaju zato što procesi uglavnom ispoljavaju lokalitet reference.

Mogući problemi

Glavni problem DP tehnike predstavljaju procesorske instrukcije za pomeranje ili kopiranje blokova podataka sa jedne lokacije na drugu, naročito u slučaju preklapanja izvorišnog i odredišnog bloka, kao i instrukcije sa autoinkrementom ili autodekrementom. Ove instrukcije ne smeju da se izvršavaju ponovo od početka nakon opsluživanja PF prekida, jer podaci koji se obrađuju gube integritet. U tom smislu, primenjuju se druge tehnike za razrešenje ovog problema.

Još jedan problem koga treba pomenuti je nedostatak slobodnih okvira, odnosno slobodne fizičke memorije. PF rutina tada najčešće obavlja algoritam za zamenu stranica: jedan okvir se izabere iz memorije, a stranica koja se u njemu nalazi prebaci na disk, čime se oslobađa fizička memorija. Algoritmi za zamenu stranica opisani su u daljem tekstu.

Performanse DP tehnike

Performanse DP tehnike, zavise od brojnih faktora, kao što su verovatnoća da se dogodi PF greška (p, $p \in [0,1]$), trajanje memorijskog ciklusa (t_{MA}) i trajanje obrade PF prekida (t_{PF}).

Efektivno vreme pristupa memoriji ukoliko se koristi DP tehnika je:

$$t_{EA} = (1 - p)\, t_{MA} + p\, t_{PF}.$$

Razmotrimo detaljno od čega zavisi trajanje obrade PF prekida. Uprošćeno, komponente obrade PF prekida su: servisiranje PF prekida, čitanje stranice i ponovno izvršenje procesa koji je izazvao PF. Detaljnije, PF rutinu čini sledeća sekvenca aktivnosti:

[1] prelazak u operativni sistem izazvan prekidnim signalom PF,

[2] čuvanje konteksta prekinutog procesa u registrima procesora,

[3] određivanje da li je referenca legalna i ako jeste određivanje lokacije stranice na disku,

[4] čitanje stranice sa diska koje izaziva čekanje (PF rutina prelazi u stanje WAIT),

[5] dodela procesora drugom procesoru, jer tekući proces (PF rutina) čeka na resurs,

[6] spašavanje konteksta tekućeg procesa nakon prekidnog signala koji znači da je čitanje stranice sa diska završeno,

[7] korekcija tabele stranica,

[8] čekanje da se procesor ponovo dodeli procesu koji je izazvao PF i

[9] obnova konteksta procesa.

Dodatno, ukoliko neka stranica mora prethodno da se izbaci iz fizičke memorije radi oslobađanja okvira, proces punjenja stranice u operativnu memoriju može sadržati još jednu ili dve ulazno-izlazne operacije. Stranica koja se žrtvuje ima svoju sliku na disku, ali se mora ponovo upisati na disk ukoliko je njen sadržaj izmenjen. Dakle, pored jednog ciklusa za punjene stranice (*swap-in*) može se dogoditi da imamo i još jedan ciklus upisa žrtvovane stranice na disk (*swap-out*).

8.3. Alternativne tehnike učitavanja stranica

Korišćenje DP tehnike značajno olakšava procedure kreiranja i izvršavanja procesa. Proces može da otpočne izvršenje ako se ona stranica procesa koja sadrži njegovu prvu instrukciju nalazi u memoriji. U nastavku teksta opisane su alternativne tehnike za učitavanje stranica, kao i pojam memorijski mapiranih datoteka.

CoW (Copy on Write) tehnika

Prilikom kreiranja procesa na UNIX sistemu fork sistemski poziv duplira adresni prostor procesa roditelja. Posle toga, potrebno je formirati i kopirati sve stranice koje pripadaju roditelju. U slučaju da proces dete otpočne izvršavanje, stranice se mogu formirati tehnikom DP. Međutim, DP tehnika se može izbeći korišćenjem tehnike slične deljenju stranica, koja omogućava brzo kreiranje procesa i minimizira broj novih strana koje će se kreirati za proces dete. Ovu tehniku, poznatu pod imenom kopiranje na ciklus upisa (*CoW, Copy-on-Write*), koristi većina savremenih operativnih sistema.

CoW tehnika funkcioniše na sledeći način: stranice proces roditelja se inicijalno ne kopiraju, niti se nova memorija dodeljuje proces detetu. Procesi roditelj i dete inicijalno dele sve stranice, koje se označavaju kao CoW. Ako bilo koji proces pokuša da modifikuje stranicu, najpre se kreira kopija stranice koja se dodeljuje tom procesu. Nakon toga proces može da modifikuje svoju kopiju.

Prepaging tehnika

Ukoliko se koristi čista DP tehnika, prilikom pokretanja procesa dolazi do velikog broja PF grešaka. Ukoliko je poznata lista stranica koje će proces zahtevati, što je siguran slučaj za suspendovanim procesom u *swap* prostoru, može se primeniti *prepaging* tehnika. *Prepaging* učitava više stranica u memoriju odjedanput, za razliku od čiste DP koja učitava samo jedni stranicu.

Tehnika se izvodi na sledeći način: operativni sistem prati radno stanje svih aktivnih procesa. Kada se proces suspenduje, na disku se osim stranica čuva i njegov radni skup (*working set*). Radni skup procesa se može obnoviti nakon izlaska iz suspendovanog stanja.

Postavlja se pitanje: šta je brže - prepaging tehnika ili čisti DP, zato što učitavanje svake stranice zahteva jednu I/O operaciju, a pitanje je da li će sve one koristiti. Pretpostavimo da se ukupno s stanica učitava odjedanput, a da je α procenat od tih s stranica koji će se zaista koristiti. Prepaging se isplati samo ako je vreme za prepaging s stranica manje od vremena za izvršnje (1-α) x s pojedinačnih PF grešaka, a to je slučaju kada α teži ka 1.

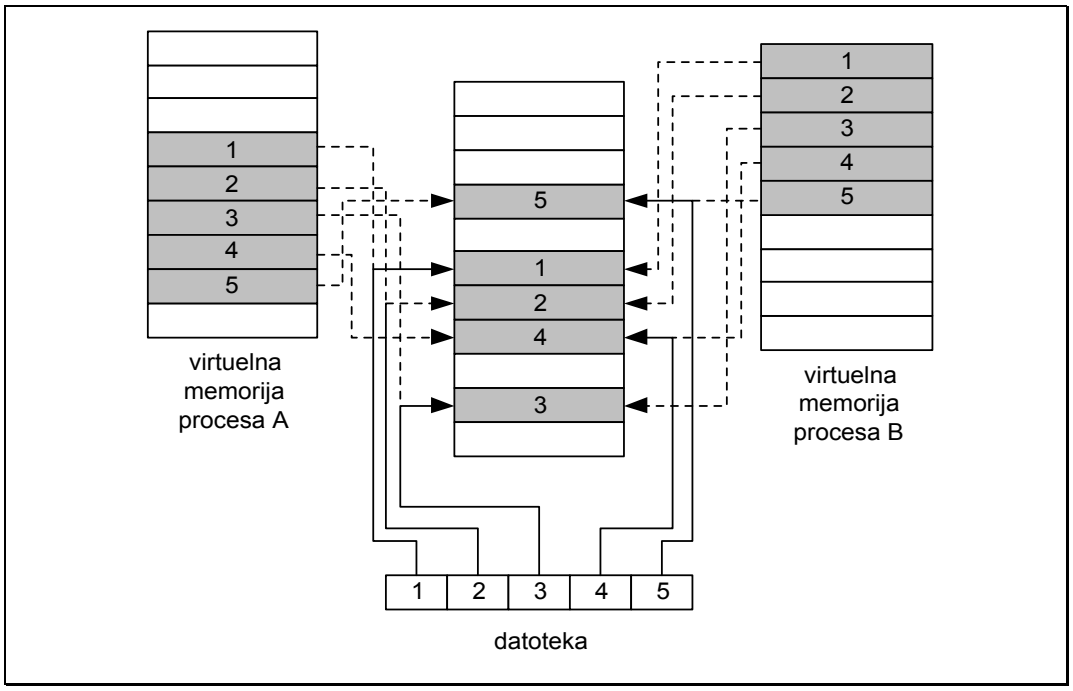
Memorijski mapirane datoteke

Iako se datoteke čitaju pomoću standardnih sistemskih poziva za rad sa sistemima datoteka, koncept virtuelne memorije dozvoljava korisnicima da datotekama pristupaju i preko memorijskih referenci.

To se postiže pomoću memorijski mapiranih datoteka, tako što se deo virtuelnog memorijskog prostora dodeli datotekama - slika stranice na disku predstavlja deo datoteke, a ne stranicu u *swap* prostoru. Inicijalni pristup datoteci se odvija preko DP tehnike, koja će izazvati PF prekid i kao rezultat te greške, pročitaće se deo datoteke u memoriju. Sledeći delovi datoteke dobijaju se na isti način, a ne preko read()sistemkog poziva.

Neki operativni sistemi imaju posebnu podršku za memorijski mapirane datoteke, dok neki sve datoteke obrađuju samo na taj način.

Više procesa može deliti datoteke u memoriji baš na ovaj način, tako što se datoteka pročita u memoriju, pomoću DP tehnike, a svaki proces podesi svoj adresni prostor tako da ukazuje na datoteku, kao što je prikazano na slici 8.3. Na ovaj način, upis koji uradi bilo koji proces, automatski je vidljiv svim ostalim procesima.

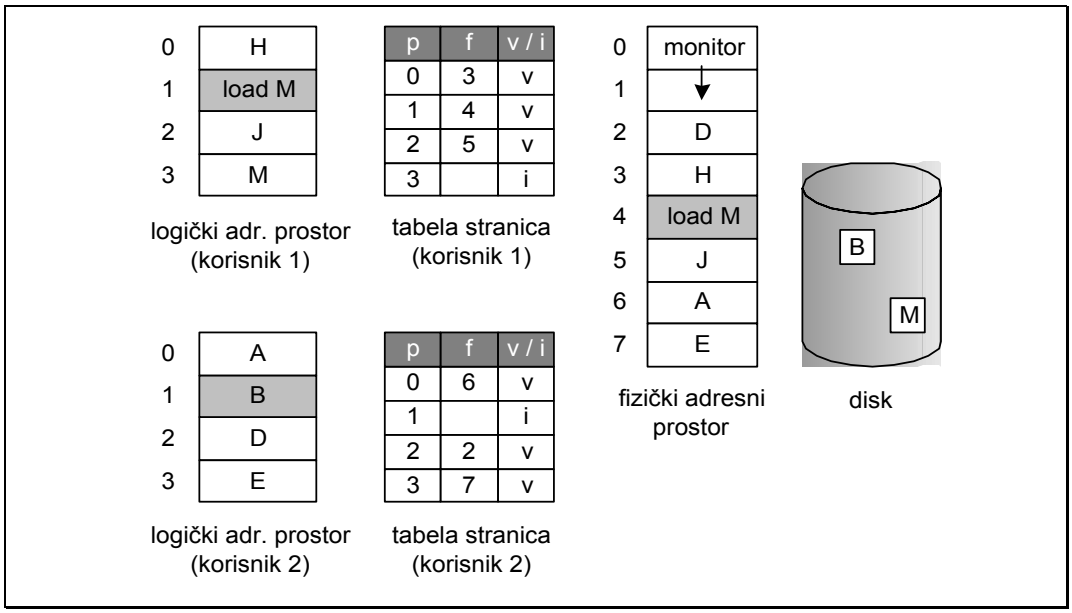


Slika 8.3. *Memorijski mapirane datoteke*

8.4. Zamena stranica

Već je rečeno da proces može da se izvršava ako se samo deo procesa nalazi u memoriji. To dovodi do povećanja broja aktivnih procesa u memoriji, a samim tim i do povećanja stepena multiprogramiranja, odnosno pre-alokacije (*over-allocation*) i iskorišćenja procesora. Pojave koje redovno prate sisteme sa virtuelnom memorijom su iznenadna pojavljivanja velikog broja PF prekidnih signala i potpuno zauzeće cele fizičke memorije.

Na slici 8.4. prikazana su dva procesa sa po 4 logičke stranice i memorija sa 8 okvira, koja je trenutno popunjena. Proces P1 izvršava instrukciju load M koja ima referencu za stranicu 3. Stranica 3 se ne nalazi u fizičkoj memoriji, nego na disku, tako da nastupa PF prekidni signal. Kako nema slobodnih okvira, jedan mora da se žrtvuje i da se oslobodi za stranicu M.



Slika 8.4. *Pre-alokacija*

Zamena stranica, prikazana na slici 8.5., funkcioniše na sledeći način:

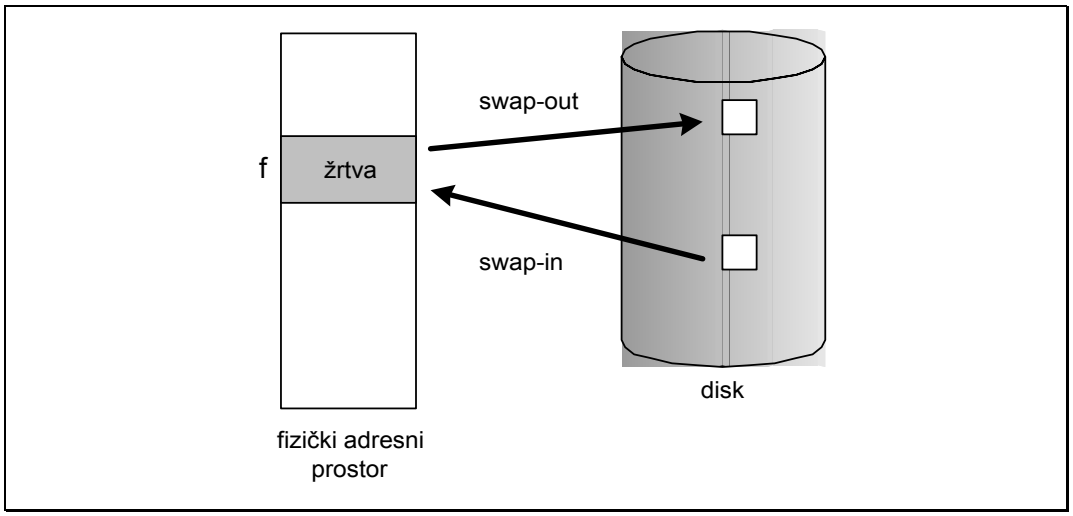
[1] Najpre se bira okvir u koji će biti smeštena željena stranica. Ako ima slobodnih okvira, uzima se jedan od njih. Ako nema nijednog slobodnog, pomoću algoritama za zamenjivanje stranica bira se žrtva, odnosno okvir u kome se nalazi stranica koja će biti zamenjena.

[2] Žrtvovana stranica se upisuje na disk ukoliko je sadržaj okvira modifikovan u odnosu na postojeću sliku na disku. Ovaj korak zahteva da se u tabelu stranica uvede bit čistoće (*dirty flag*), koji opisuje da li je okvir modifikovan, ali znatno povećava performanse, jer se sadržaj neizmenjenih okvira ne prebacuje na disk.

[3] Žrtvovani okvir se u tabeli stranica obeležava bitom i (*invalid*) kao nevalidan.

[4] Željena stranica se učitava u oslobođeni okvir.

[5] Tabela stranica se ažurira i nova stranica se obeležava kao validna bitom validnosti.

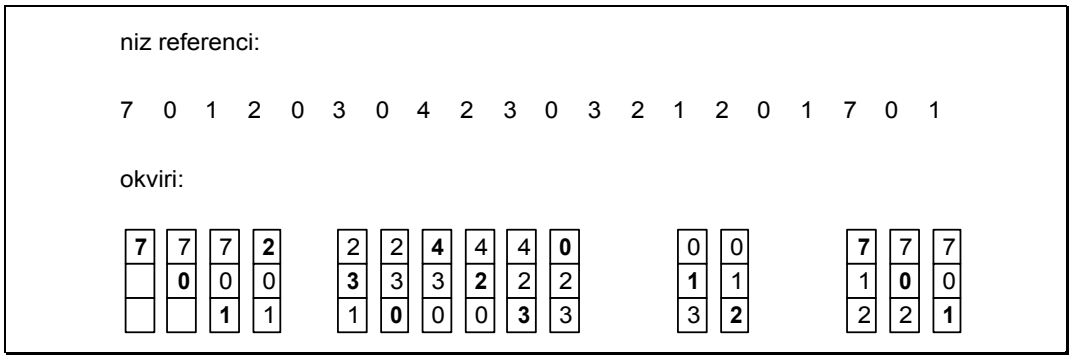


Slika 8.5. *Zamena stranica*

Na ovaj način se rešava problem pre-alociranja memorije, ali se performanse procesa u slučaju pojave PF prekidnog signala znatno smanjuju jer se pri svakoj zameni stranica mogu javliti jedan ili dva ciklusa korišćenja diska. Zato se broj PF prekida mora održati na što manjem nivou kako se ne bi degradirale performanse. Iako je potpuno jasno da se povećanjem fizičke memorije, odnosno broja okvira, smanuje verovatnoća pojave PF grešaka, treba naći onaj algoritam koji će za datu sekvencu referenci da postigne najmanju verovatnoću pojave PF grešaka. Algoritmi za izbor žrtve opisani su u nastavku poglavlja. Prilikom ispitivanja ovih algoritama potrebno je definisati ulaznu sekvencu referenci, i na datoj veličini fizičke memorije analizirati broj PF grešaka.

FIFO algoritam

Jedan od najprostijih algoritama je FIFO algortitam, koji prilikom razmene bira najstariju stranicu kao žrtvu. Kao parametar se, po pravilu, ne koristi vreme punjenja okvira stranicom, već se formira FIFO red na čiji se kraj stavljaju stranice prilikom učitavanja u memoriju. Prilikom zamene, memoriju napušta stranica koja se nalazi na početku reda. FIFO algoritam je ilustrovan sledećim primerom: posmatrajmo memoriju sa tri okvira koja su inicijalno prazna i niz memorijskih referenci, kao što je prikazano na slici 8.6.

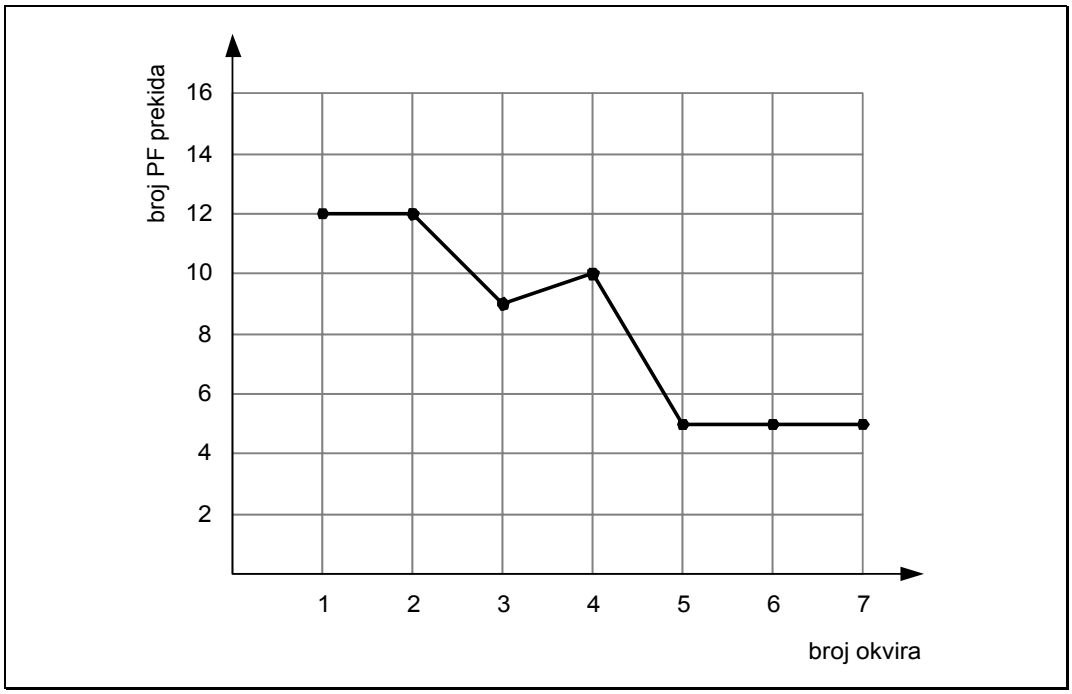


Slika 8.6. *Primer FIFO algoritma*

Prve tri reference napraviće tri uzastopne PF greške i popuniti sva tri okvira. Pojava četvrte reference usloviće PF grešku, koja mora da izazove zamenu stranice. Kao žrtva, bira se okvir 0 u kome se nalazi najstarija stranica (stranica 7). Peta referenca ne izaziva PF jer se stranica 0 nalazi u drugom okviru u memoriji. Šesta referenca izaziva novi PF i novu zamenu, a kao žrtva bira se okvir 1 koji sadrži stranicu 0. Sadržaj memorije prikazan je samo za reference koje izazivaju PF grešku. FIFO algoritam proizvodi 15 PF prekida za datu sekvencu.

FIFO je lak i za razumevanje i realizaciju, ali nije optimalan jer kao kriterijum koristi samo vreme punjenja okvira stranicom, ali ne i broj referenci ka pojedinim stranicama. Kao anomalija, češće korišćene stranice se mogu izbaciti vrlo lako umesto stranica koje se retko koriste.

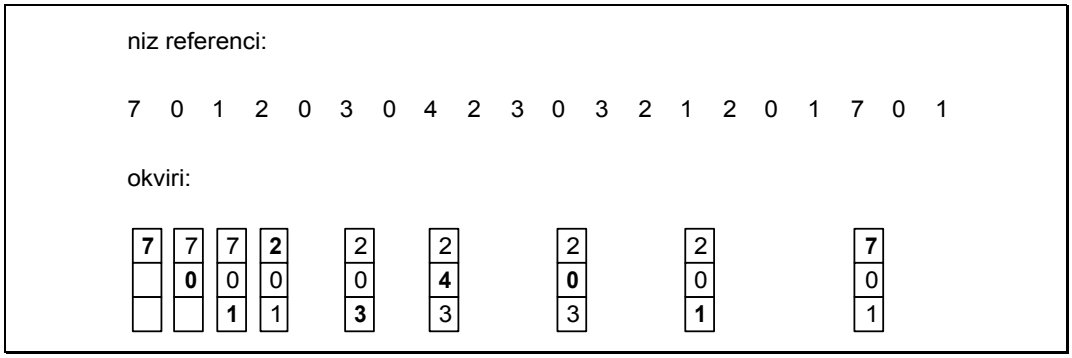
Takođe, ilustrovaćemo i *Belady*-jevu anomaliju koja je karakteristična za FIFO algoritam, a koja se odnosi na smanjenje broja PF grešaka sa povećanjem broja okvira, bez obzirana ulaznu sekvencu referenci. Ako primenimo FIFO algoritam na niz refernci: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 i variramo broj okvira, dobićemo grafik prikazan na slici 8.7. Umesto strogog smanjenja broja PF grešaka sa povećanjem broja okvira, analiza pokazuje da FIFO algoritam izaziva veći broj PF grešaka u slučaju memorije sa 4 okvira u odnosu na memoriju sa tri okvira.



Slika 8.7. Belady-*jeva anomalija*

Optimalni algoritam

Optimalni algoritam (OPT, MIN) je onaj algoritam koji postiže najmanji mogući broj PF grešaka i nikada ne dovodi do *Belady*-jeve anomalije. Prilikom zamene žrtvuje se ona stranica koja se neće koristiti najduže vremena, čime se maksimalno odlaže pojavljivanje sledeće PF greške.



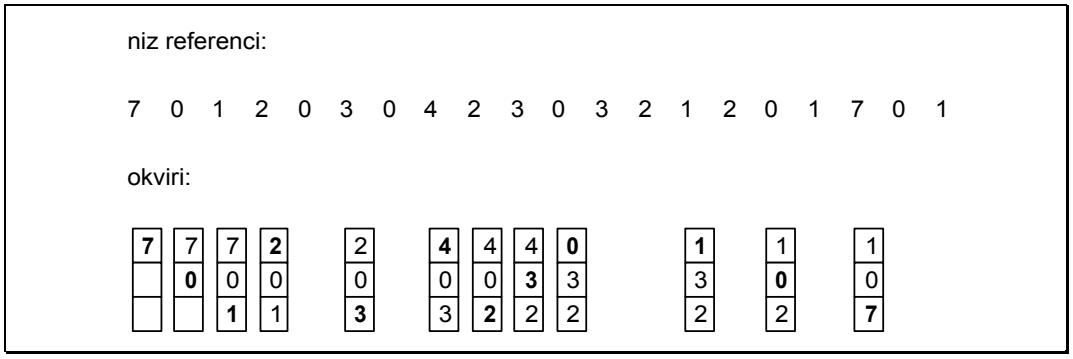
Slika 8.8. *Optimalni algoritam*

Na primer, ako će stranica A biti potrebna posle 100 instrukcija, a stranica B posle 1000 instrukcija, onda je bolje žrtvovati stranicu B, jer se izazivanje sledeće PF greške odlaže na duže vreme.

Na slici 8.8. prikazan je optimalni algoritam, koji za sekvencu korišćenu u primeru sa slike 8.6. proizvodi 9 PF grešaka. Međutim, za razliku od FIFO algoritma koji svoj kriterijum formira na istoriji referenci, koja je uvek poznata, OPT algoritam svoj kriterijum formira na budućnosti. Kao takav, OPT algoritam mora unapred da poznaje sve procese, što nije moguće, naročito ukoliko se radi o interaktivnim sistemima. OPT je najbolji mogući algoritam, ali se, na žalost, ne može implementirati, jer ne postoji način da operativni sistem dođe do potrebnih informacija o redosledu stranica koje će u budućnosti biti potrebne (predikcija je apsolutno nemoguća u interaktivnim sistemima).

LRU algoritam

Za razliku od samog optimalnog algoritma, aproksimacije istog se mogu implementirati. LRU algoritam (*least recently used*) kao kriterijum koristi vreme poslednje upotrebe stranice, što je podatak iz prošlosti, tako da je implementacija samog algoritma moguća. Po LRU algortitmu, zamenjuje se stranica koja najduže nije korišćena. LRU svakoj stranici dodeljuje vreme poslednjeg korišćenja, koje se ažurira sa sa svakim pristupom stranici.



Slika 8.9. *LRU algoritam*

Na slici 8.9. prikazan je LRU koji za istu sekvencu korišćenu u prethodnim primerima izaziva 12 PF grešaka (OPT izaziva 9, a FIFO 15). LRU se veoma često koristi kao jako dobar algoritam. Međutim za njegovu realizaciju često se koristi pomoć hardvera. Algoritam se može realizovati na sledeća dva načina:

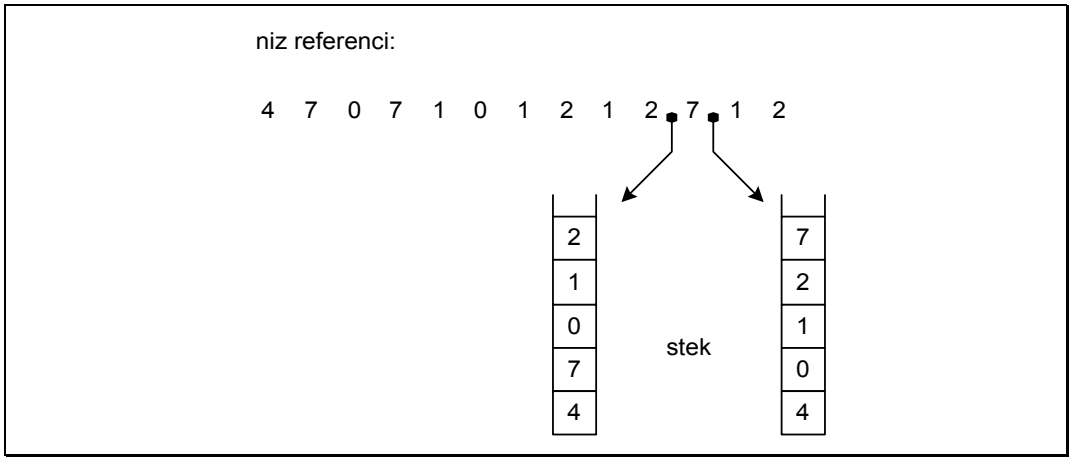
[1] Realizacija pomoću brojača.

Procesor ima brojač memorijskih instrukcija. Kada se računar pokrene, brojač se resetuje na nulu. Posle svake memorijske instrukcije vrednost brojača se

povećava za jedan. Svaki okvir ima svoj interni brojač. Svaki put kada se pristupi okviru, sadržaj brojača instrukcija se kopira u interni brojač okvira. Kada dođe do PF prekida, algoritam pregleda sve interne brojače i bira stranicu čiji je broj najmanji (što znači da najduže nije korišćena). Premašenje ovog načina realizacije čine interni brojači okvira kojima se pristupa prilikom svakog pristupa okviru radi upisivanja vrednosti brojača instrukcija.

[2] Realizacija pomoću steka.

Slično kao kod FIFO algoritma, formira se stek koji opisuje redosled pristupanja stranicama. Nakon PF prekida, žrtvuje se stranica sa vrha steka, a na njeno mesto se stavlja stranica kojoj se pristupa. Ova realizacija je veoma spora jer se pri svakom pristupu memoriji stek mora ažurirati. Tipična sekveca pomeranja kroz stek pikazana je na slici 8.10.



Slika 8.10. *LRU - realizacija pomoću steka*

LRU kao i OPT ne ispoljava *Belady*-jevu anomaliju. Svi algoritmi koji ne ispoljavaju ovu anomaliju nazivaju se stek algoritmi i zahtevaju solidnu hadversku asistenciju.

LRU aproksimativni algoritmi

Neki sistemi obezbeđuju hadversku podršku za LRU algoritam, obično u formi bita reference (*reference bit*, R bit) koji se nalazi u tabeli stranica. Za svaki ulaz u tabeli stranica postoji bit reference koji se setuje svaki put kada se stranica odnosno okvir referencira. Vrednost R bitova se inicijalno postavlja na 0. Brisanje R bita nakon određenog vremena obavlja se pomoću tajmerskih prekida. R bitovi obezbeđuju informaciju o korišćenju konkretnog okvira, ali ne i o redosledu korišćenja okvira.

Da bi se dobila informacija o redosledu korišćenja stranica, uvode se dodatni biti koji se koriste u kombinaciji sa referentnim bitom. Uzima se jedan bajt koji se dodeljuje svakom ulazu u tabelu stranica čija je inicijalna vrednost jednaka 0. Uvek kada se dogodi tajmerski prekid, operativni sistem pomera referentni bit u najviši bit dodeljenog bajta, pri čemu se svi ostali bitovi pomeraju u desno, a najniži bit otpada. Ako bajt sadrži vrednost u bitima 00000000, znači se tom okviru nije pristupilo za vreme trajanja poslednjih 8 tajmerskih intervala. Vrednost 11111111 znači da se okviru pristupilo tačno pre 8 tajmerskih intervala a sve ostalo je nešto između. Broj bitova i način postavljanja i pomeranja bitova može da varira.

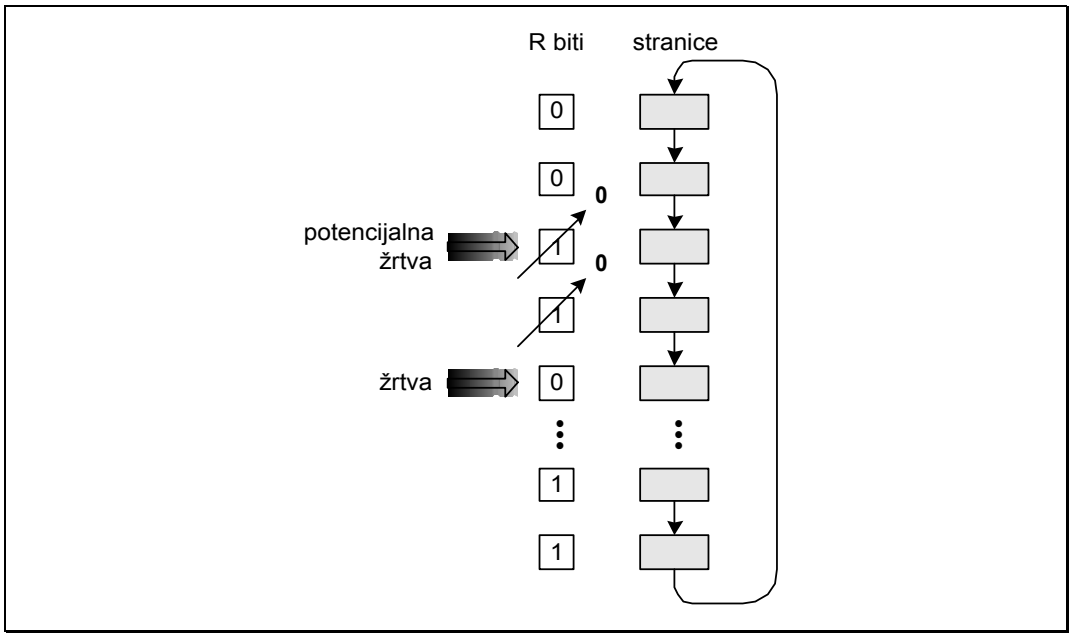
Druga šansa (satni algoritam)

Algoritam zamene stranica poznat pod imenom druga šansa (*Second Chance*) je varijanta FIFO algoritma koja u obzir uzima i korišćenost stranica. Ovaj kombinovani kriterijum predstavlja pokušaj izbegavanja nedostatka FIFO rešenja. Kao pomoć, koristi se vrednost R (*reference*) bitova (1 ako smo pristupili stranici, 0 ako nismo). Potencijalna žrtva je stranica na kraju reda čekanja. Ako je za tu stranicu R=0, to znači da se stranica dugo nalazi u memoriji, a da joj nismo pristupili. Dakle, stranica očigledno nije potrebna, tako da se može žrtvovati. Ako je R=1, to znači da je stranica dugo u memoriji, ali i da je nedavno korišćena. R bit te stranice biće resetovan posle provere, a sama stranica biće prebačena na početak reda opsluživanja. Pretraga se nastavlja: sledeća potencijalna žrtva je opet stranica na kraju reda. Ako je stranica korišćena (R=1), dobiće još jednu šansu (algoritam je tako dobio ime), ako nije, žrtvovaće se. Algoritam nastavlja rad dok se ne pronađe žrtva.

Ako su sve stranice korišćene, algoritam će resetovati vrednost svih R bitova i početi ponovo analizu. Kako je R=0 za sve stranice, žrtvuje se najstarija. To znači da algoritam sigurno nalazi okvir za smeštaj nove stranice. Ako je R=0 za sve stranice, ovaj algoritam se ponaša isto kao i FIFO.

Jedna od mogućih realizacija ovog algoritma koristi kružni red čekanja, tako da se ovaj algoritam alternativno naziva i satnim algoritmom (*clock*).

Po ovom algoritmu sve stranice koje popunjene stavljamo u kružnu listu, a ukazivač za zamenu stoji po FIFO principu na poziciji stranice koja je najranije učitana. Ako je R=0, stranica se izbacuje, a na njeno mesto se učitava nova stranica. Ako je R=1, stavimo R=0 i pomeramo na sledeću stranicu, odnosno tražimo dalje. Na ovaj način izbegavamo stalno pomeranje opisa stranica sa početka reda na kraj. Na slici 8.11, ukazivač (strelica) ukazuje na potencijalnu žrtvu. Ako je R=1, R bit se briše, a lista se dalje pretražuje, sve dok se ne pojavi R=0. Svi R biti koji su postavljeni na 1 se u tom prolasku kroz listu brišu. Ako su svi R bitovi postavljeni, algoritam će napraviti pun krug, nakon čega se pretvara u običan FIFO algoritam.



Slika 8.11. *Druga šansa - realizacija pomoću kružnog reda za čekanje*

NRU algoritam

Algoritam NRU (*not recently used*) zahteva da se svakom okviru pridruže dva bita: R (*reference bit*), koji označava da li smo pristupili tom okviru, i M (*modified*), koji označava da li smo menjali sadržaj tog okvira. Ova dva bita se mogu implementirati na nivou operativnog sistema, ukoliko nisu implementirani u hardveru. Na početku rada procesa svi R i M bitovi dobijaju vrednost 0. Na osnovu vrednosti R i M bitova okvire delimo u sledeće klase:

[1] R=0, M=0. Stranica nije skoro ni korišćena ni modifikovana, i kao takva predstavlja idealnu žrtvu. Ovo je najbolji slučaj za zamenu.

[2] R=0, M=1. Stranica nije skoro korišćena, ali je modifikovana. Kao takva, nije pogodna za zamenu, jer mora da se upiše na disk, što će izazvati dva I/O ciklusa.

[3] R=1, M=0. Stranica je skoro korišćena, ali nije modifikovana. Najverovatnije će se koristiti ponovo.

[4] R=1, M=1. Stranica je skoro korišćena i modifikovana. Verovatno će se ponovo koristiti. Stranica nije pogodna za zamenu, jer mora da se upiše na disk, što će izazvati dva I/O ciklusa

Svaki okvir spada u jednu od ove četiri klase. Kada dođe do zamene, stranice se analiziraju na osnovu pripadnosti određenoj klasi. Postupak je sličan kao i kod satnog

algoritma: najpre se bira stranica koja se nalazi u što optimalnijoj klasi za zamenu. Znači, prvo se pretražuje klasa (0,0), zatim klasa (0,1) i tako redom. Prilikom pretraživanja, kružna lista se može puta više puta. Kao što se vidi, prednost za zamenu se daje stranicama koje nisu modifikovane, jer će one napraviti samo jedan I/O ciklus.

Prednost NRU algoritma je jednostavnost implementacije. Nedostatak je u tome što ne pravi dovoljno finu razliku između stranica iste klase, odnosno ne vodi se računa o broju pristupa, niti o broju menjanja sadržaja okvira.

Frekvencijski bazirani algoritmi - LFU i MFU

Prethodno opisani algoritmi su kriterijum formirali na osnovu vremena punjenja okvira stranicom sa diska ili na osnovu vremena korišćenja stranice. Ni jedan od tih algoritama ne uzima u obzir koliko puta je stranica bila referencirana. Definisaćemo dva algoritma koji kriterijum formiraju na osnovu frekvencije korišćenja stranice:

[1] LFU (*least frequently used*)

LFU algoritam zamenjuje stranicu koja ima najmanji broj referenci u vremenu. Pri tome, svaka stranica koja se učita se određeno vreme mora štititi od samog algoritma, jer kada se pojavi njena frekvencija je jednaka 1, što to je minimum. Kao takva, stranica bi bila potenicjalna žrtva i ne bi dobila šansu da se takmiči sa drugim stranicama. Algoritam je loš za slučajeve u kojime se neke stranice intenzivno koriste u nekom kratkom vremenskom intervalu, nakon čega se uopšte ne koriste. Takve stranice dugo ostaju u memoriji.

[2] MFU (*most frequently used*)

MFU algoritam zamenjuje stranicu koja ima najveći broj referenci. Algoritam smatra da su te stranice odavno učitane u memoriju i da procesima više nisu potrebne, dok su stranice sa malim brojem referencu relativno nove i potrebne.

Frekvencijski bazirani algoritmi imaju svoje prednosti i mane ali su relativno teški za realizaciju.

8.5. Raspodela okvira po procesima i efekat zasićenja

Najprostija strategija alokacije fizičke memorije koja se može primeniti je čisti DP - kad god se pojavi potreba, stranica sa učitava u memoriju. Kada nema više slobodnoh okvira, vrši se zamena - na osnovu prethodno opisanih algoritama bira se stranica koja se izbacuje iz memorije, čime se oslobađa okvir. Postoje brojne modifikacije ove osnovne strategije.

Raspodela okvira

Postavlja se sledeće pitanje: kako rasporediti m okvira na n procesa?

Minimalni broj okvira po procesu zavisi od konkretne procesorske arhitekture, a najmanji je dva, zato što instrukcija koja se sastoji od više bajtova može da bude na granici između dve stranice. Maksimalni broj okvira koji proces može da dobije je ograničen veličinom fizičke memorije.

Prema metodi jednake raspodele, svakom procesu se dodeljuje jednaka količuna okvira m/n. Prema metodi proporcionalne raspodele, okviri se dodeljuju procesima na osnovu osobina procesa (kao što, na primer, veličina procesa). Svakom procesu pripašće sledeći broj okvira:

$$a_i = \frac{s_i}{\sum_{i=1}^{n} si} m .$$

Pri tome je m - ukupan broj okvira fizičke memorije, s_i - veličina procesa p_i, n - broj procesa i S - ukupna virtuelna memorija, jednaka zbiru veličina svih n procesa ($S = \sum_{n}^{i=1} s_i$).

Osnovni nedodatak ovih metoda je u tome što ne uzimaju u obzir prioritet procesa pri raspodeli okvira.

Globalna i lokalna zamena stranica

Problem karakterističan za metoda jednake i proporcionalne raspodele je zamena stranica u slučaju da nema slobodnih okvira. Prethodno opisani algoritmi za zamenu stranica nisu vodili računa o tome kom su procesu okviri dodeljeni na korišćenje. Kada se to uzme u obzir, mogu se defnisati dve vrste zamene stranica:

[1] Globalna zamena.

Algoritmu se dozvoljava da odabere bilo koji okvir fizičke memorije, što znači da jedan proces može žrtvovati tuđu stranicu (odnosno poslati je u *swap* prostor). Broj okvira dodeljenih procesu može se menjati u vremenu. Na broj PF prekida utiču i drugi procesi.

[2] Lokalna zamena.

Algoritmu se dozvoljava da odabere bilo koji okvir dodeljen procesu koji želi da taj okvir napuni svojom stranicom. Prosto rečeno, jedan proces može žrtvovati isključivo svoje stranice. Broj okvira dodeljenih procesu se ne menja u vremenu.

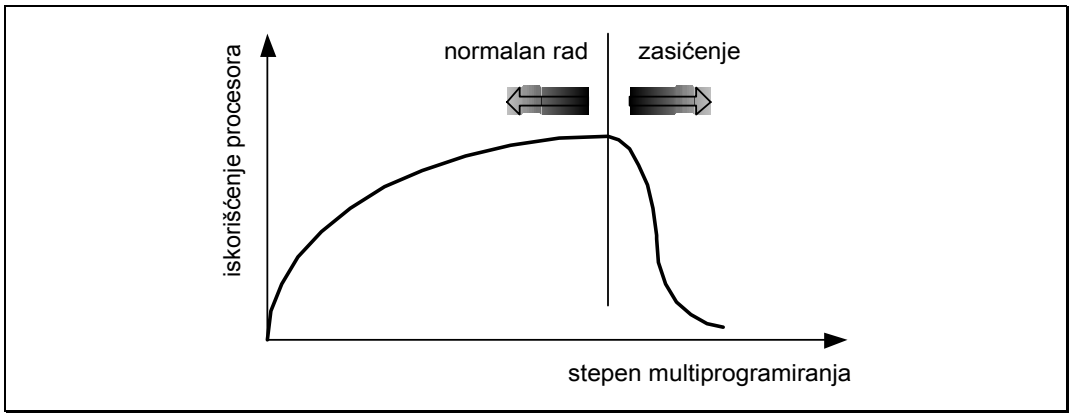
U slučaju lokalne razmene proces sam diktira svoje PF prekide (drugi procesi nemaju uticaj na broj PF prekida koje taj proces izaziva).

Lokalna razmena ima svoju manu, jer blokiran proces ne može ustupiti svoje okvire drugim procesima, što znači da aktivan proces mora žrtvovati sopstvene stranice. Zbog toga globalna razmena daje bolje rezultate.

Efekat zasićenja

Proces zahteva operativnu memoriju kao resurs. Ukoliko broj raspoloživih okvira proces padne ispod minimalne vrednosti, proces se mora dovest u stanje WAIT, jer resurs nije slobodan, pa proces ne može da se izvršava. Ukoliko je broj okvira raspoloživih za proces veoma mali, proces može da se izvršava, ali sa pojavom veoma čestih PF grešaka. Pri tome, proces veoma često ili stalno zamenjuje svoje stranice, što uključuje veliki ulazno-izlaznih operacija sa diskom. Pojava čestog razmenjivanja stranica, koja nastaje kao posledica visokog stepena multiprogramiranja, naziva se efekat zasićenja (*thrashing*). U takvim situacijama javljaju se ozbiljni problemi u vidu degradacije performansi.

Razmatrajmo sledeći slučaj: operativni sistem prati stepen iskorišćenja procesora. Ako je taj stepen previše mali, operativni sistem povećava stepen multiprogramiranja, tako što aktivira nove procese. Ukoliko se koristi globalna razmena stranica svako uvođenje novog procesa povećava broj PF grešaka. To rezultuje u povećanom broju ulazno-izlaznih operacija, što još više smanjuje stepen iskorišćenja procesora. Ukoliko operativni sistem ne prepozna zasićenje, mogao bi da nastavi sa uvođenjem novih procesa, što bi dovelo do katastrofalnog pada performansi i iskorišćenja procesora. Efekat zasićenja prikazan je na slici 8.12.



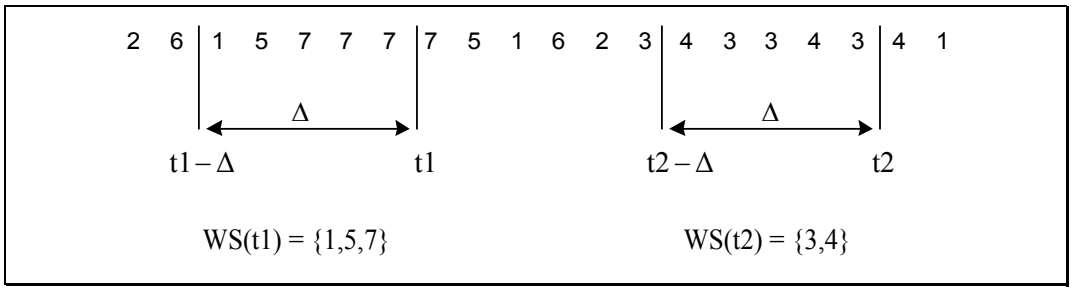
Slika 8.12. *Efekat zasićenja*

Stepen iskorišćenja procesora raste sa povećanjem broja procesa do kritične tačke u kojoj počinje naglo da opada zbog učestalih *swap-in* i *swap-out* operacija. Zasićenje se može sprečiti ukoliko operativni sistem prestane da povećava stepen multiprogramiranja.

Efekat zasićenja se može ograničiti korišćenjem lokalne ili prioritetne razamene. Takođe, da bi se zaštitili od zasićenja, proces mora unapred obezbediti informaciju broju okvira koje će tražiti u toku svog izvršavanja. Uvešćemo model lokalnosti u svaki proces, koji će nam pomoći da objasnimo efekat zasićenja.

Lokalnost predstavlja skup stranica koje proces koristi zajedno u jednom intervalu vremena. Proces može menjati svoje lokalnosti a one se mogu preklapati. Na primer, svaki poziv potprograma napraviće novu lokalnost, a takođe kad se izađe iz tog potprograma lokalnost se menja. Trashing efekat nastupa kada je suma lokalnosti za sve aktivne procese veće od fizičke memorije.

Uvodimo novi model koga ćemo nazvati radni model (*working-set model*). Prozor radnog skupa Δ (*working-set window*) je vreme u kome se izvrši određeni broj instrukcija, a sve stranice koje su referencirane u tom periodu, predstavljaju radni skup (*working-set*) za Δ.



Slika 8.13 *Radni model*

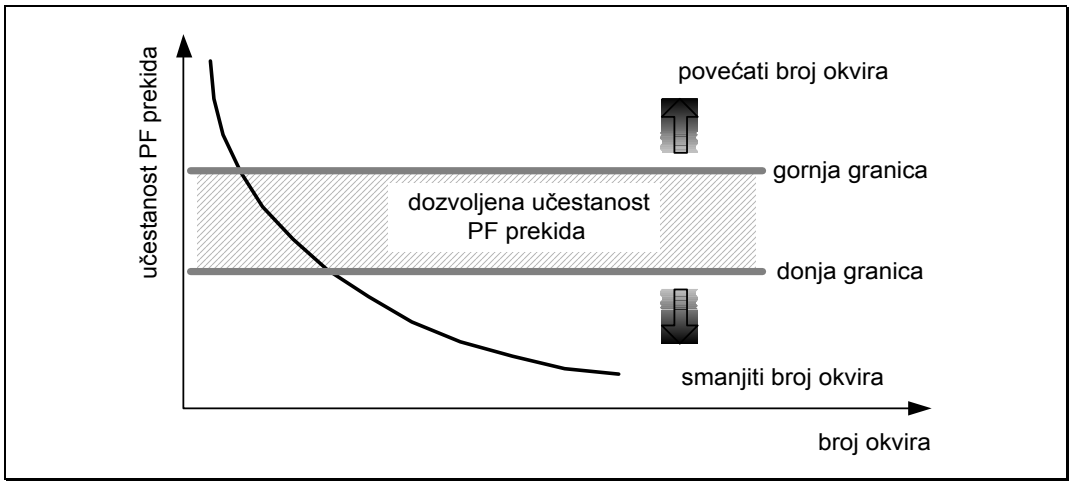
Na slici 8.13 data su dva radna skupa za po 5 memorijskih referenci. WS(t1) obuhvata stranice 1,5 i 7, a WS(t2) stranice 3 i 4. Potrebno je odrediti veličinu vremenskog prozora. Ako je prozor suviše mali, ne opisuje dovoljno lokalnost procesa, a ako je preveliki, tada se preklapa više lokalnosti.

U celoj ovoj priči najznačanija je veličina radnog skupa D, koja predstavlja zahtev svih procesa za okvire jednom trenutku. Definišimo veličinu radnog skupa procesa kao unkupan broj stranica koje proces traži u vremenskom prozoru Δ. Veličinu radnog skupa procesa Pi označićemo sa WSS_i. Ukupna veličina radnog prostora u sistemu D je:

$$D = \sum_{i=1}^{n} WSS_i .$$

Ako je D > m, nastupiće efekat zasićenja. Operativni sistem prati veličinu D i kada D dođe do veličine sistemske memorije, novi procesi se neće uvoditi, a pojedini procesi moraju da se suspenduju na disk, kako bi se smanjila veličina radnog skupa.

Efekat zasićenja se može sprečiti limitiranjem broja PF prekida koje jedan proces izaziva u okvirima dozvoljenog opsega (broj PF prekida je veliki u slučaju zasićenja). Tipičan grafik broja PF grešaka za jedan proces u funkciji broja okvira dat na slici 8.14:



Slika 8.14. *Broj PF prekida procesa u funkciji broja okvira*

Najpre se postavljanjem donje i gornje granice definiše opseg dozvoljenog broja PF prekida za jedan proces. Zatim se prati broj, odnosno frekvencija izazivanja prekida. Ako je broj PF grešaka procesa manji od donje granice, tada je procesu dodeljeno više okvira, pa se poneki može oduzeti i dodeliti drugim procesima. Ako je broj PF grešaka veći od gornje granice, procesu nije dodeljen dovoljan broj okvira, tako da može da ih dobije još. Ako se broj PF grešaka svakog procesa održava u dozvoljenom opsegu, između ove dve granice, efekat zasićenja se neće dogoditi.

8.6. Dopunska razmatranja

Upotreba virtuelne memorije ima svoje prednosti, ali je implementacija relativno komplikovana. Između ostalog, projektant operativnog sistema mora odrediti optimalnu veličinu stranice, na osnovu uticaja velikog broja faktora, odrediti optimalne tehnike za učitavanje stranica i algoritme zamene, obezbediti implementaciju *swap* tehnike i metoda za zaključavanje stranica i sprečiti efekat zasićenja. Takođe, od programera se očekuje da koriguje stil programiranja u skladu sa konceptima virtuelne memorije. U

suprotnom, programi mogu izazvati veliki broj PF prekida, što dovodi do degradacije performansi.

Veličina stranice

Na izbor veličine stranice utiču mnogi fakori. Faktori koji utiču da stranice budu veće su:

- veličina tabele stranica (što su stranice veće, tabela stranica je manja, pa je manje i premašenje sistema),
- ulazno-izlazni ciklusi *swap* tehnike (manji broj većih transfera podataka traje kraće od većeg broja manjih transfera) i
- TLB efikasnost (poželjno je da sav radni prostor jednog procesa može da stane u TLB procesora, što se postiže samo ako su stranice veće).

Faktrori koji utiču da stranice budu manje su:

- bolje iskorišćenje memorije i
- manja interna fragmentacija.

Uticaj na strukturu programa

Virtuelna memorija zahteva da programer vodi računa o načinu programiranja. Uzmimo za primer sledeća dva programa koji će na različiti način dovesti do istog efekta, ali sa različitim brojem PF prekida, a samim tim i različitim vremenom izvršavanja. Uzmimo matricu veličine 1024x1024 elemenata i napunimo je nulama. Pretpostavimo da je veličina memorijske stranice 2K - svaka vrsta matrice biće smeštena u jednu stranicu.

Izvorni kod prvog programa je:

```
int X[i][j] = new int[1024][1024];
for (j = 0; j < X.length; j++)
   for (i = 0; i < X.length; i++)
      X[i,j] = 0;
```

Program najpre upisuje nule u sve elemente jedne kolone, pa onda prelazi na sledeću kolonu. Kako su vrste matrice smeštene po principu jedna vrsta u stranici, program će napraviti 1024^2 grešaka.

Izvorni kod drugog programa je veoma sličan:

```
int X[i][j] = new int[1024][1024];
for (i = 0; i < X.length; i++)
```

```
for (j = 0; j < X.length; j++)
   X[i,j] = 0;
```

Program najpre upisuje nule u sve elemente jedne vrste, pa onda prelazi na sledeću vrstu. Kako su vrste matrice smeštene po principu jedna vrsta u stranici, program će napraviti samo 1024 grešaka, odnosno svakoj stranici će pristupiti samo jednom.

Ovaj jednostavan primer ilustruje značaj prilagođavanja programera konceptu virtuelne memorije.

Zaključavanje stranica

Objasnićemo na sledećem primeru kada je zaključavanje stranica potrebno. Proces je odabrao svoju stranicu kao bafer za ulazno-izlazne operacije i poslao komandu disku. Dok ulazno-izlazni uređaj sprema podatke i prebacuje ih, proces prelazi u stanje čekanja na resurs. Drugi proces postaje aktivan i prelazi u stanje RUN. Drugi proces u toku svog rada može izavati PF grešku, koja može dovesti do dodele bafera nekom drugom procesu.

Ova situacija se može sprečiti zabranom baferovanja ulazno-izlaznih operacija u korisničkoj memoriji. Podaci se prebacuju u kerneske I/O bafere, koji nisu dostupni algoritmima za zamenu stranica u toku korisničkih procesa. Međutim, posle transfera na relaciji uređaj-kernelski bafer, stranica se mora kopirati iz kerneskog prostora u korisnički, što umanjuje performanse.

Kao alternativa može se koristiti mehanizam za zaklučavanje stranica (*page lock*) koji zaključanoj stranici ne dozvoljava da bude dodeljena nekom drugom procesu. Jednostavno rečeno, drugi procesi ne mogu koristiti zaključane stranice.

9

Ulazno-izlazni podsistem

Sadržaj poglavlja:

9.1. Funkcije ulazno-izlaznog podsistema
9.2. Klasifikacija uređaja
9.3. Hardver od značaja za ulazno-izlazni podsistem
9.4. Uniformni interfejs ka aplikacijama
9.5. Servisi koje obezbeđuje ulazno-izlazni podsistem
9.6. Prevođenje zahteva u ulazno-izlazne operacije
9.7. Mehanizam tokova
9.8. Performanse ulazno-izlaznog podsistema

9.1. Funkcije i ciljevi projektovanja

Posmatrajmo teorijski model operativnog sistema. Do sada smo definisali pojam i funkcije jezgra operativnog sistema i sloja upravljanja memorijom. To znači da smo obezbedili okolinu u kojoj se mogu formirati procesi koji međusobno mogu komunicirati i kojima može biti dodeljena određena količina memorija na korišćenje. Međutim, u ovom stadijumu razvoja operativnog sistema, korisnik procesima ne može zadati ulazne podatke, niti može od procesa preuzeti rezultate obrade. Takođe, proces nije u mogućnosti da rezultate obrade sačuva na disku ili nekom drugom medijumu niti da ostvari komunikaciju sa drugim računarom. To znači da u operativni sistem treba ugraditi podsistem koji će obezbediti komunikaciju sa ulazno-izlaznim uređajima.

Većina uređaja koji se priključuju na računar može se svrstati u neku opštu kategoriju. Opšte kategorije su formirane prema nameni uređaja, tako da se mogu

izdvojiti uređaji za dugotrajno skladištenje podataka (kao što su diskovi i trake), uređaji za prenos podataka (kao što su mrežne kartice i modemi) i uređaji koji obezbeđuju interfejs ka korisniku (kao što su monitori, tastature i miševi). Ovi uređaji se međusobno razlikuju u mnogim aspektima. Osim namene, uređaje karakteriše i skup operacija koje se nad podacima mogu izvesti, brzina prenosa podataka na relaciji računar-uređaj, količina podataka kojima se istovremeno pristupa (jedinična količina transfera), način pristupa podacima, deljivost uređaja i skup stanja koja se mogu smatrati greškama.

Operativni sistem mora da obezbedi podršku za rad sa širokim spektrom ulazno-izlaznih uređaja čije karakteristike variraju kako sa vrstom uređaja, tako i sa konkretnim modelom određenog proizvođača hardvera. Jasno je da je zbog raznovrsnosti samih uređaja otežana konstrukcija ulazno-izlaznog podsistema koji karakterišu izvestan nivo uniformnosti i izolacija karakteristika specifičnih za konkretne uređaje. Definisaćemo osnovne funkcije i ciljeve ulazno-izlaznog podsistema.

Funkcije ulazno-izlaznog podsistema

Ulazno-izlazni podsistem je deo operativnog sistema koji obavlja sledeće dve funkcije:

[1] upravlja i kontroliše ulazno-izlazne uređaje i operacije koje ti uređaji izvršavaju i

[2] obezbeđuje što jednostavniji interfejs prema korisniku i prema ostatku sistema.

Ciljevi projektovanja ulazno-izlaznog podsistema

Kontrola ulazno-izlaznih uređaja je jedno od glavnih interesovanja projektanata operativnog sistema. Zbog velikih razlika u samim uređajima, potrebno je u sistem implementirati različite metode kontrole uređaja, a sama kompleksnost upravljnja uređajima mora biti sadržana i sakrivena u kernelu.

U savremenim tehnologijama prisutna su dva konfliktna trenda: standardizacija softverskih i hardverskih interfejsa i povećanje broja ulazno-izlaznih uređeja različitih karakteristika. Kao rezultat, podrška za konkretne uređaje ili grupe srodnih uređaja sadržana je u programima za upravljanje ulazno-izlaznim uređajima - veznicima, odnosno drajverima (*drivers*). Drajveri predstavlaju unikatni interfejs ulazno-izlaznog podsistema, kao što sistemski pozivi predstavljaju univerzalni interfejs između aplikacija i operativnog sistema.

Navešćemo osnovne ciljeve razvoja ulazno-izlaznog podsistema:

[1] Nezavisnost uređaja

Postoje dva aspekta nezavisnosti uređaja: različiti modeli uređaja istog tipa moraju se sa aspekta programa jednako posmatrati. Na primer, program za rad sa

datotekama ne sme se opteretiti činjenicom da se kao CD-ROM uređaj koristi TEAC CDR52E (konkretan model određenog proizvođača). Jednostavno rečeno, program pristupa CD-ROM uređaju bez obzira da li je kao uređaj korišćen određen model TEAC ili LG CD-ROM uređaja. Drugi, zahtevniji cilj je nezavisnost programa od konkretne vrste uređaja. Na primer, mora se obezbediti da program na što istovetniji način može pročitati podatke sa diska ili CD-ROM uređaja. Jasno je da ova nezavisnost ne sme dovesti do problema kao što je slanje izlaznih podataka na ulazni uređaj.

Posledica: programi ne rade sa realnim, konkretnim uređajima, već sve ulazno-izlazne operacije obavljaju pomoću virtuelnih uređaja. Primeri virtuelnih uređaja su tokovi (*streams*) i datoteke (karakteristično za UNIX i Linux sisteme). Programer jednostavno preusmerava ulaz i izlaz na tokove koji su prethodno povezani sa realnim uređajima. Povezivanje se obavlja komandama operativnog sistema na osnovu informacija koje su dobavljene od korisnika. Nezavisnost uređaja se posle toga može lako postići. Na primer, ulazni tok OUTPUT vezan za CD-ROM uređaj može se lako vezati za disk ili traku.

[2] Nezavisnost karakter koda

Očigledno je nepoželjno pisati takve programe koji od korisnika zahtevaju detaljno poznavanje karakter koda korišćenog u konkretnim uređajima. Ulazno-izlazni podsistem mora preuzeti brigu o prepoznavanju konkretnog karakter koda i predstavljanju podataka korisniku u standardnoj formi.

Posledica nezavisnosti karakter koda je postojanje unutrašnjeg karakter koda, odnosno uniformne predstave svih karaktera u samom uređaju, i mehanizama za prevođenje karakter koda, koji su obično realizovani u vidu translacione tabele ili manjeg programa.

[3] Efikasnost

Svaki uređaj je znatno sporiji od operativne memorije i samog procesora, što znači da su ulazno-izlazne operacije usko grlo računarskog sistema i da je poželjno obaviti ih što efikasnije.

Posledica: ulazno-izlazni podsistem mora obezbediti konkurentnost u radu sa uređajima.

[4] Uniformni interfejs ka aplikacijama i korisniku

U interesu jednostavnosti poželjno je omogućiti uniformno rukovanje uređajima. Zbog raznovrsnosti uređaja, ovaj cilj je teško postići. Moguće rešenje ovog problema, koje se koristi na UNIX i Linux sistemima, objašnjeno je kasnije u ovom poglavlju.

9.2. Klasifikacija uređaja

Ulazno-izlazni uređaji se razlikuju u mnogim aspektima. Karakteristike variraju kako sa vrstom uređaja, tako i sa konkretnim modelom određenog proizvođača hardvera. Izvršićemo klasifikaciju uređaja prema sledećim kriterijumima:

[1] Namena uređaja

Prema nameni, uređaji se mogu klasifikovati u sledeće opšte kategorije:

- uređaji za dugotrajno skladištenje podataka (*storage devices*), odnosno masovni memorijski medijumi, kao što su diskovi i trake,
- uređaji za prenos podataka (*transmission devices*), kao što su mrežne kartice i modemi i
- uređaji koji obezbeđuju interfejs ka korisniku (*human-interface devices*), kao što su monitori, tastature i miševi.

[2] Smer transfera

Prema smeru transfera uređaji se mogu podeliti na ulazne (miš, skener), izlazne (štampač) i ulazno-izlazne (mrežna kartica).

[3] Jedinična količina transfera

Prema jediničnoj količini transfera, odnosno minimalnoj količini podataka za čitanje i upis, uređaji se dele na blok i karakter uređaje. Kao specijalna klasa mogu se izdvojiti mrežni uređaji.

Jedinična količina transfera blok uređaja je blok podataka. Interfejs za blok uređaje sadrži sve elemente potrebne za pristupanje diskovima i drugim blok-orijentisanim uređajima. Po pravilu, aplikacije ne koriste ovaj interfejs direktno, već preko intrefejsa za sistem datoteka, koji predstavlja viši nivo u hijerarhijskom modelu operativnog sistema. Međutim, aplikacija može direktno pristupiti blok uređaju ukoliko za to postoji potreba. Takav način pristupa naziva se sirovi ulaz-izlaz (*raw I/O*).

Jedinična količina transfera karakter uređaja je bajt. Za neke uređaje (kao što je na primer tastatura) stroga definicija jedinične količine transfera se prošuruje na celu liniju (*line editing*).

Mrežni uređaji na prvi pogled pripadaju klasi karakter-orijentisanih uređaja. Međutim, njihove performase i karakteristike se znatno razlikuju od karakteristika klasičnih karakter uređaja, tako da se mrežni uređaji izdvajaju kao posebna klasa.

[4] Metod pristupa

Po metodi pristupa, uređaji se dele na uređaje sa sekvencijalnim pristupom (kao što su modem i magnetna traka) i uređaje sa direktnim ili slučajnim prisitupom (kao što su diskovi i CD-ROM uređaji). Sekvencijalni uređaji prenose podatke fiksnim redosledom koji određuje sam uređaj, dok je kod uređaja sa direktnim pristupom redosled prenosa podataka proizvoljan i određuje ga aplikacija, odnosno korisnik.

[5] Deljivost uređaja

Prema deljivosti, uređaji se mogu klasifikovati kao deljivi i nedeljvi. Deljive uređaje može koristiti veći broj procesa istovremeno. Deljivi uređaji su, na primer, diskovi (veći broj korisnika prijavljenih na udaljeni UNIX server istovremeno koristi disk servera preko sistema datoteka). Nedeljive, odnosno posvećene uređaje karakteriše to da ih u jednom trenutku vremena može koristiti samo jedan proces. Primer nedeljivog uređaja je tastatura.

[6] Brzina uređaja

Prema brzini, uređaji se grubo mogu podeliti na brze i spore. Ova podela je relativna jer brzina varira od nekoliko bajtova u sekundi (tastatura sa izuzetno brzim operaterom) do više gigabajta u sekundi (gigabitna mrežna kartica). Uređaji se prema ovom kriterijumu teško mogu klasifikovati, jer brzina uređaja iste klase znatno varira sa modelom konkretnog proizvođača.

[7] Mogućnost upisa

Prema mogućnosti upisa, uređaji se dele na uređaje koji služe za čitanje i pisanje (kao što su diskovi), uređaje koji služe isključivo za čitanje (CD-ROM uređaji) i uređaje koji služe isključivo za pisanje (grafička kartica).

Časovnik i timer kao specijalna klasa uređaja

Većina računara ima hardverski časovnik i tajmer koji obezbeđuju tri osnovne funkcije:

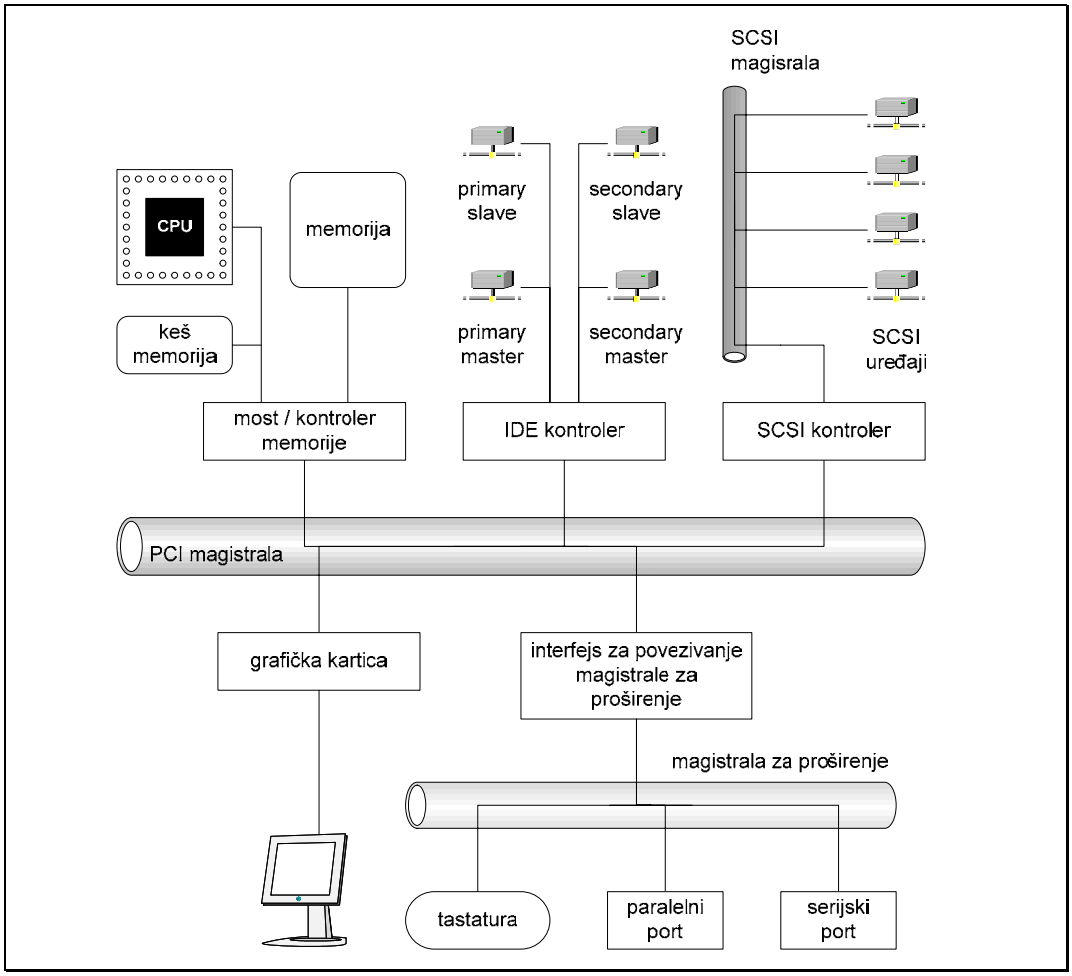
- prikazivanje tekućeg vemena,
- prikazivanje proteklog vremena i
- tajmerski okidač za operaciju X u trenutku T.

Časovnik i tajmeri predstavljaju specijalnu klasu ureađaja, tako da im se pristupa pomoću specifičnih sistemskih poziva. Tajmeri koji mere proteklo vreme ili obezbeđuju okidanje su programibilni intervalski tajmeri. Oni se programiraju tako da nakon određenog vremena generišu prekidni signal. Kao takvi, koriste se za realizaciju tehnike pretpražnjena (*preemption*) pri raspoređivanju procesora. Pretpražnjenje funkcioniše na sledeći način: tajmer generiše prekidni signal, izvršenje tekućeg procesa se prekida, a

zatim se poziva dispečer da na osnovu algoritma za raspoređivanje procesora odabere proces iz procesorskog reda i preda mu procesor na korišćenje.

9.3. Hardver od značaja za ulazno-izlazni podsistem

Uređaji komuniciraju sa računarskim sistemom tako što šalju signale preko žičnih ili bežičnih transmisionih medijuma, a na računar su vezani preko odgovarajućeg priključka. Princip povezivanja uređaja objasnićemo na primeru PC arhitekture, koja je prikazana na slici 9.1.



Slika 9.1. *PC arhitektura*

Magistrale

Ukoliko više uređaja deli zajednički skup žica, sa strogo definisanim protokolom koji specificira skup poruka koje se mogu poslati, veza sa računarom se naziva magistrala (*bus*). Serijska veza uređaja kod koje je samo jedan uređaj priključen na sistem, a svaki uređaj ima priključak za povezivanje sledećeg, naziva se *daisy-chain* i funkcioniše slično kao i magistrala.

PCI sistemska magistrala spaja procesorsko-memorijski podsistem sa brzim ulazno-izlaznim uređajima. Specijalna magistrala koja se naziva *expansion bus* (magistrala za proširenje) sadrži serijske i paralelne portove na koje se priključuju spori uređaji kao što su tastature, miševi i štampači.

Kontroleri

Na slici 9.1. prikazana su dva SCSI diska priključena na SCSI kontroler. Kontroler je kolekcija elektronike koja radi na tri načina: kao port, kao magistrala ili kao uređaj. Na primer, serijski kontroler je port. Grafička kartica je kontroler koji se na priključuje na PCI magistralu, a radi kao uređaj. SCSI kontroler se sa jedne strane priključuje na PCI magistralu, a sa druge strane formira novu magistralu (SCSI *bus*) na koju mogu da se povežu do 15 SCSI uređaja.

Svaki kontroler ima jedan ili više registara. Upisom i čitanjem vrednosti ovih registara obavlja se komunikacija između procesora i kontrolera, tako da procesor može izdati komande kontroleru kako bi se, na primer, obavio neki transfer podataka na izlazni uređaj.

Registrima kontrolera se može pristupiti na dva načina, zavisno od toga da li su memorijski i ulazno-izlazni prostor razdvojeni. Nekim uređajima se može pristupiti na oba načina, zavisno od operacije koju je potrebno obaviti. Tipičan primer je grafička kartica, koju ćemo iskoristiti za ilustraciju ovih tehnika.

[1] Ukoliko su memorijski i ulazno-izlazni prostor razdvojeni, procesor ima posebne instrukcije za upis i čitanje podataka sa određene adrese na kontroleru. Na primer, grafička kartica ima određeni broj ulazno-izlaznih portova kojima se mogu obaviti osnovne upravljačke operacije, a kojima se pristupa posebnim instrukcijama.

[2] Ukoliko se ulazno-izlazni prostor može tretirati kao memorija, transferi se mogu obaviti običnim instrukcijama za rad sa memorijom (memorijski mapirane ulazno-izlazne operacije). Tipičan primer je memorija na grafičkim karticama koja služi za smeštaj slike.

Tipičan ulazno-izlazni port se sastoji iz četriri vrste registara:

[1] Kontrolni registar

Služi za postavljanje režima rada uređaja, kao što su, na primer, brzina porta i tip komunikacije. U ovaj registar procesor isključivo upisuje podatke.

[2] Statusni registar

Ovaj registar opisuje status komande koja se izvršava, odnosno opisuje da li je komanda izvršena, da li podatak spreman, da li je nastupila greška, itd. Procesor iz statusnog registra isključivo čita podatke.

[3] Registar podataka za ulazni režim (*data-in*).

Data-in registar služi za čitanje podataka sa ulaznih uređaja. Procesor iz data-in registra isključivo čita podatke.

[4] Registar podataka za izlazni režim (*data-out*).

Data-out registar služi za upis podataka na izlazni uređaj. Iz ovog registra procesor isključivo čita informacije.

Upisivanjem i čitanjem podataka iz odgovarajućih portova procesor obavlja ulazno-izlazne operacije i kontroliše rad uređaja.

Tehnika prozivanja (polling)

Kompletan protokol za interakciju između računara i kontrolera može biti dosta komlikovan. U principu računar i kontroler funkscionišu po principu proizvođač-potrošač, koji se u ovom slučaju realizuje pomoću dva bita: *busy* i *command ready*.

- *Busy* bit u statusnom registru opisuje stanje kontrolera. Ukoliko je bit setovan, kontroler je zauzet, odnosno nešto radi. Ukoliko je bit resetovan, kontroler je slobodan i spreman da prihvati novu komandu.
- Računar ukazuje kontroleru na prisustvo nove komande preko *command-ready* bita u komandom registru. Kada računar setuje ovaj bit to znači da je kontroler dobio komandu i da izvršavanje može da počne.

Handshake tehniku objasnićemo na primeru slanja jednog bajta na serijski kanal, koje se odvija u sledećim koracima:

[1] Računar čita statusni registar. Čitanje se ponavlja sve dok vrednost *busy* bita ne bude 0.

[2] Računar setuje *write* bit u komandnom registru i upisuje bajt za serijski kanal u *data-out* registar.

[3] Računar setuje *command-ready* bit.

[4] Setovani *command-ready* bit je znak kontroleru da treba da otpočne izvršenje komande. Kontoler setuje *busy* bit.

[5] Kontroler čita svoj komandni registar, detektuje *write* komandu, zatim čita *data-out* registar i organizuje ciklus za slanje sadržaja *data-out* registra na serijski kanal.

[6] Ukoliko je sve u redu, kontroler briše *command-ready* bit, zatim briše *error* bit u statusnom registru koji ukazuje na pojavu greške, i briše *busy* bit koji ukazuje da je komanda završena. Počev od kraja koraka 4 računar stalno proverava da li je *busy* bit 0, kako bi znao kada je komanda završena.

Ovakva petlja mora da se ponovi za svaki ciklus. U koraku 1 i koraku 6 imamo tehniku prozivanja (*polling*), koja se takođe naziva "zaposlen čekanjem" (*busy-waiting*). Računar ponavlja čitanje statusnog registra i analizira *busy* bit sve dok vrednost bita ne postane 0. Za brze uređaje ovo traje relativno kratko, ali za spore veoma dugo.

Tehnika prozivanja se na mnogim procesorima obavlja u petlji sastavljenoj od tri instrukcije:

- čitanje statusnog registra
- ekstrakcija i analiza *busy* bita
- povratak u petlju ako je vrednost *busy* bita 1, izlaz iz petje ako je vrednost *busy* bita 0.

U slučaju sporih uređaja procesor ostaje zaposlen čekanjem u krajnje dugačkim petljama, umesto kog bi mogao da radi pametnije stvari. Međutim, za sada je busy bit jedini način da računar odredi da li kontroler slobodan ili da li je komanda završena.

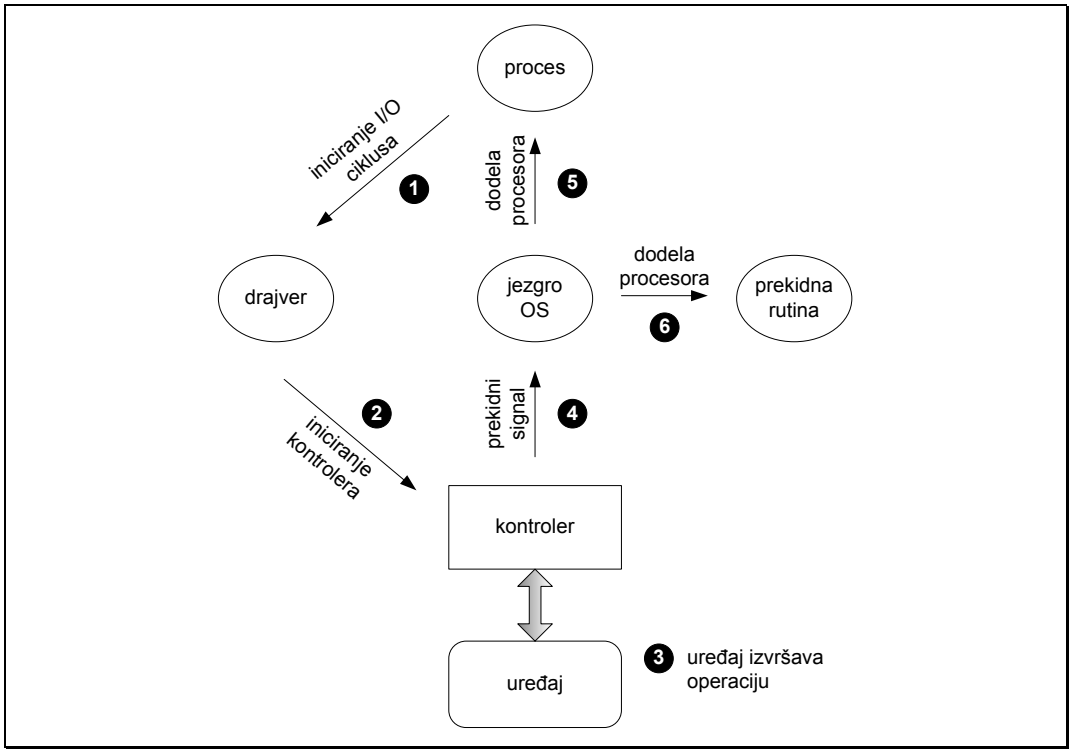
Prekidi

Osnovni nedostatak tehnike prozivanja je trošenje procesorskog vremena na petlju prozivanja. Nedostatak se može ukloniti uvođenjem hardverskog mehanizma koji omogućava ureađaju da označi procesoru kada je komanda izvršena. Ovaj mehanizam se naziva prekid (*interrupt*).

Prekidni sistem funkcioniše na sledeći način: kada uređaj završi operaciju, kontroler šalje prekidni signal procesoru preko prekidne linije (IRQ - *interrupt request line*). Procesor će završti tekuću instrukciju, posle čega se prekida sekvencijalno izvršavanje programa, kako bi se obradio prekid. Posle obrade prekida, procesor nastavlja izvršavanje programa.

U principu, prekinuti proces mora da nastavi izvršenje (da li odmah nakon obrade prekida, ili neki drugi put - to zavisi od raspoređivanja procesora). Prekinuti proces može nastaviti izvršenje ukoliko se pre obrade prekida sačuva kontekst procesa, odnosno vrednosti programskog brojača (PC, *program counter*), registara procesora i pokazivači na deo memorije koji proces koristi. Kada se sačuva stanje procesa, poziva se rutina za obradu prekida (*interrupt handler*), koja određuje uzrok prekida i opslužuje prekid. Posle toga se može nastaviti izvršenje prekinutog procesa.

Obrada prekida je prikazana na slici 9.2. Proces najpre inicira ulazno-izlazni ciklus koji se prosleđuje drajveru, nakon čega drajver inicira kontroler (koraci 1 i 2). Uređaj zatim izvršava operaciju. Nakon završetka operacije (korak 3) kontroler postavlja prekidni signal (4) koji u prekida izvršenje tekućeg procesa i predaje kontrolu rutini za obradu prekida (korak 5). Nakon završene prekidne rutine, obnavlja se kontekst prekinutog procesa (korak 6) i predaje kontrola prekinutom procesu.



Slika 9.2. *Obrada prekida*

Ovakav mehanizam omogućava procesoru da odgovori na svaki asinhroni događaj. U modernim operativnim sistemima, tehnika prekida uključuje brojne sofisticirane osobine:

- mogućnost odlaganja (*deffer*) obrade prekida dok je proces u kritičnoj sekciji,
- brzu i efikasnu tehniku određivanja uređaja koji je postavio prekid (ne može se svaki uređaj analizirati kako bi se odredio uzročnik prekida) i
- omogućeno gneždenje prekida (*interrupt nesting*) i višeslojni prekidni sistem, koji će razlikovati prioritet prekidnih signala i rešavati prekide po prioritetu.

Sve ove kvalitetne funkcije realizuje hardverski uređaj koji se naziva prioritetni prekidni kontroler (PIC, *priority interrupt contoller*). Većina procesora umesto jedne ima dve posebne linije za prekidne signale.

[1] Nemaskirajuća (*non-masking*) linija

Prekidni signal poslat po nemaskirajućoj liniji uvek može da prekine izvršenje tekućeg procesa. Nemaskirajuća linija se koristi za slanje prekidnih signala usled kritičnih hardverskih grešaka, kao što su greške u memoriji.

[2] Maskirajuća (*masking*) linija

Prekidni signal poslat po maskirajućoj liniji ne prekida izvršenje procesa sve dok se proces na primer nalazi u kritičnoj sekciju. Maskirajuća linija se koristi za slanje prekidnih signala koji potiču od normalnih operacija i standardnih grešaka koje se javljaju na ulazno-izlaznim uređajima.

Prekidni sistem obično funkcioniše na sledeći način. Relativno je mali broj uređaja koji mogu postaviti prekidni sigmal, a za svaki od njih postoji posebna prekidna rutina, koja se nalazi u memoriji. Informacije o adresama svih prekidnih rutina se čuvaju u tabeli prekidnih vektora (*interrupt vector table*), gde svaki ulaz sadrži adresu jedne prekidne rutine. Na ovaj način se smanjuje vreme potrebno za određivanje uređaja koji je postavio prekidni signal. Tačno se zna koja je klasa uređaja vezana za koju liniju PIC kontrolera, tako da na osnovu aktivne linije, PIC okida procesor i daje mu informaciju u vidu vektora, odnosno ukazivača na ulaz u prekidnu vektor tabelu. Na ovaj način nema prozivanja uređaja, a sve informacije obezbeđuje PIC.

U tabeli prekidnih vektora za Intel Pentuim procesore, prekidni vekrori 0-31 su nemaskirajući i koriste se u slučaju ozbiljnih sistemskih grešaka. Neki prekidni vektori su rezervisani za kasniju upotrebu. Prekidni vektori od 32-255 su maskirajući i tipično ih koriste ulazno-izlazni uređaji. Opisaćemo neke od značajnijih nemaskirajućih prekida:

prekid	**Opis**
0	divide error - greška u deljenju
6	invalid opcode - neispravan kod operacije
7	device not available - uređaj nije prisutan
11	segment not present - segment nije prisutan
12	stack fault - greška u radu sa stekom
14	page fault - PF prekid
16	floating-point error - greška u radu sa pokretnim zarezom
18	machine check - provera sistema

Prekidni sistem, po pravilu, primenuje prekidne prioritetne nivoe, a to je mehanizam koji omogućava da se odloži opsluživanje prekida niskih prioriteta, dok se

opslužuju prekidi visokih prioritetata, naravno bez maskiranja. Takođe, kad naiđe prekid višeg prioriteta, prekida se opsluživanje prekida nižeg prioriteta, pa se opslužuje prekid višeg prioriteta (pretpražnjenje - *preemption*).

Na modernim operativnim sistemima prisutan je veliki broj prekida. U vreme podizanja sistema (*booting*) ispitivanjem magistrale se određuje da li su uređaji prisutni i ako jesu postavlja se odgovarajuća prekidna rutina u tabeli prekidnih vektora. Za vreme ulazno-izlaznih ciklusa uređaj postavlja prekidni signal uvek kada zahteva servisiranje, odnosno u slučaju kada je komanda gotova ili kada se dogodila neka greška. Prekidni mehanizam se koristi i za rukovanje izuzecima (*exception*), kao što su PF greška, deljenje nulom, pristup zaštićenim adresama i greška u parnosti. U svim slučajevima prekida se izvršavanje tekućeg procesa, a zatim se na bazi vektora iz tabele prekidnih vektora čita adresa odgovarajuće servisne rutine.

Još jedan primer korišćenja prekidnog sistema je specifična realizacija sistemskih poziva - takozvanih softverskih prekida (*trap*). Specijane instrukcije izazivaju isti efekat kao da se dogodio hardverski prekid. čuvaju se registri procesora, pa se skače na sistemski poziv preko tabele prekidnih vektora.

Neophodno je naglasiti da se sve prekidne rutine izvršavaju u kernelskom modu.

DMA (Direct Memory Access)

Kada uređaj pripremi podatke (jedan ili više bajtova), procesor treba da preuzme svaki bajt iz kontrolera. Pri tome, procesor svaki put proverava statusni bit, što znači da se izvršavaju dva I/O ciklusa. Ova tehnika čitanja podataka se naziva programirani ulaz-izlaz (PIO).

U slučaju masovnih transfera podataka sa uređajima kao što su diskovi, PIO tehnika unosi veliki gubitak vremena. Kako bi se transfer podataka mogao obaviti na relaciji kontroler - operativna memorija i obrnuto, specijalno je projektovan uređaj koji se naziva DMA kontroler (*direct memory access*). DMA ime signale za upravljenje transferom sa kontrolera, a ima i signale kojima upravlja memorijskim ciklusima. Da bi se inicirao DMA transfer, DMA se mora programirati, odnosno mora se definisati početna memorijska adresa, blok bajtova koje treba da prenese i smer transfera. I/O kontroler se ne navodi kao izvorište jer je DMA fizički povezan sa njim. Na primer, za PC arhitekturu, matična ploča sadrži više DMA kanala i svaki ima svoj I/O kontroler.

Uprošćeno, objasnićemo jednostavan prenos podataka na relaciji disk-memorija preko DMA kontrolera. Transfer se obavlja u sledećim koracima:

[1] Proces nalaže drajveru da obavi prenos podataka sa diska u bafer koji se nalazi na memorijskoj adresi X.

[2] Drajver nalaže disk kontroleru da izvrši transfer C bajtova sa diska u bafer na memorijskoj adresi X.

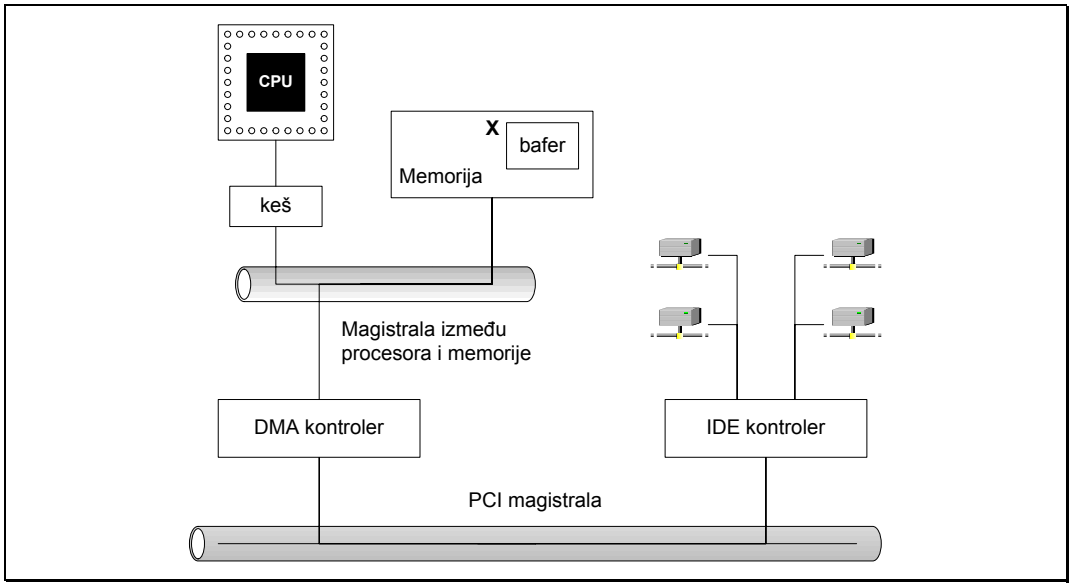
[3] Disk kontroler inicira DMA transfer.

[4] Disk kontroler redom šalje bajtove podataka DMA kontroleru.

[5] DMA kontroler prenosi podatke u bafer, DMA uvećavajući memorijsku adresu X i smanjujući broj bajtova C nakon svakog prenešenog bajta.

[6] Kada je prenešen poslednji bajt (C=0), DMA šalje prekidni signal procesoru obaveštavajući ga o završenom transferu

Blok šema sistema sa DMA kontrolerom prikazana je na slici 9.3.



Slika 9.3 *DMA kontroler*

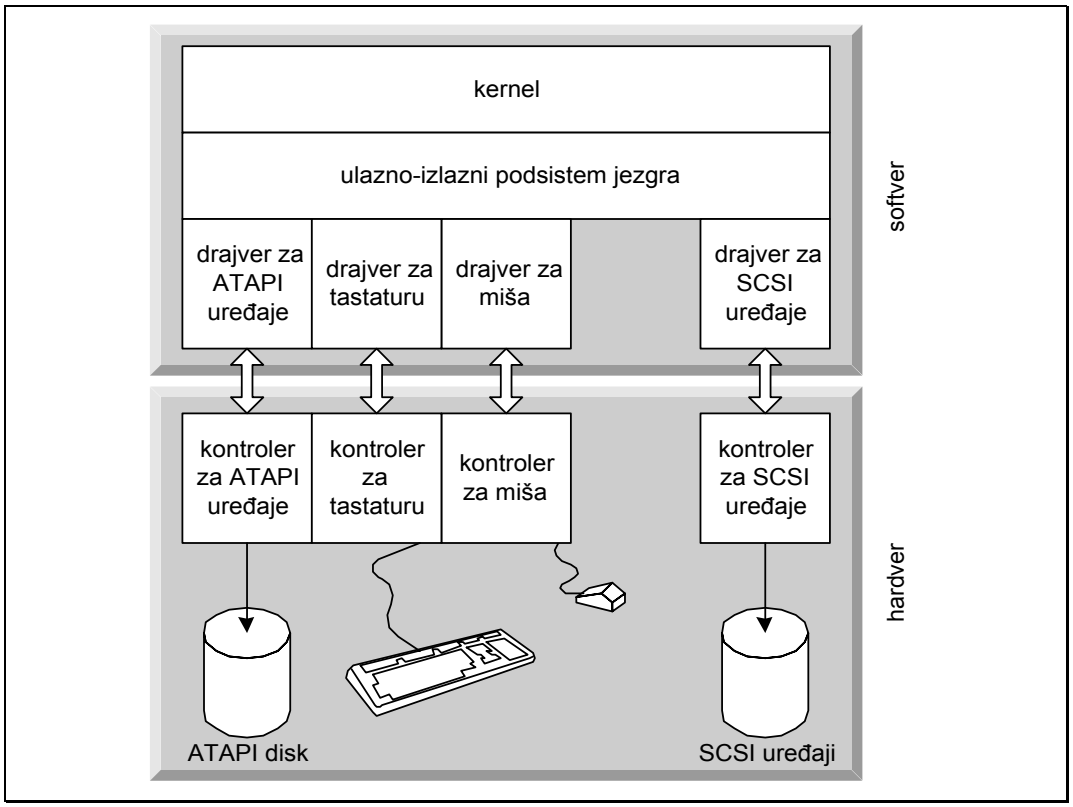
Handshake protokol između DMA i I/O kontrolera odvija se preko para linija, signalima DMA-request i DMA-acknowledge. I/O kontroler inicira DMA ciklus samo kada ima spreman podatak postavljanjem signala DMA-request. DMA posle primanja ovog signala zauzima memorijsku magistralu, a zatim organizuje memorijski ciklus u kome se bajt iz I/O kontrolera prenosi u memoriju. Po okončanju ovog ciklusa, DMA kontroler postavlja signal DMA-acknowledge, što znači kraj jednog DMA ciklusa. DMA automatski inkrementira adresu memorijskog bafera za sledeći DMA ciklus, a smanuje broj bajtova za prenos. DMA postavlja prekidni signal procesoru kada prenese sve bajtove.

Kako je memorijska magistrala zajednička za CPU i DMA, ona može pripadati samo jednom od tih resursa u jednom trenutku vremena. Postoje hardverski signali za njihovu sinhronizaciju, pri čemu procesor može svoje zahteve zadovoljiti iz keš memorije. Pojava koja se pri tome dešava na relaciji procesor-DMA je krađa ciklusa (*cycle stealing*): DMA čeka da magistrala bude slobodna i zauzima je, a za to veme

procesor mora da čeka ukoliko želi memoriju i tako gubi cikluse. Bez obzira na ovu krađu, DMA potpuno rasterećuje procesor od transfera na relaciji memorija-uređaji, čime se osetno povećavaju performanse sistema. Neke računarske arhitekture koriste fizičke adrese za DMA, dok druge koriste virtuelne adrese (DVMA - *direct virtual memory access*). DVMA obavlja i adresno mapiranje i može da obavi transfer između dva memorijski mapirana uređaja bez upotrebe memorije i procesora.

9.4. Uniformni interfejs ka aplikacijama

Jedan od ciljeva koje treba postići pri projektovanju operativnog sistema je formiranje uniformnog interfejsa ka aplikacijama i korisniku. Definisaćemo strukture i interfejse operativnog sistema koji omogućavaju tretiranje ulazno-izlaznih uređaja na standardni, jedinstveni način. Problem uniformnosti uređaja se rešava metodama apstrakcije i višeslojnosti softvera u okviru ulazno-izlaznog podsistema, kao što je prikazano na slici 9.4.



Slika 9.4. *Apstrakcija uređaja i drajveri*

Srodni uređaji, kao što su, na primer, diskovi, grupišu se u opšte klase uređaja. Svakoj klasi uređaja pristupa se preko standardnog skupa funkcija, koji se naziva interfejs. Dakle, interfejs je standardni skup funkija preko kojih se obraćamo istoj klasi ulazno-izlaznih uređaja. Razlike koje postoje među uređajima iste klase sakrivene su u specijalnim modulima jezgra operativnog sistema koji se nazivaju veznici ili drajveri (*device drivers*). Drajveri postoje za svaki uređaj, ali se korisniku predstvljaju kao jedan univerzalni interfejs.

Iznad svih drajvera nalazi se sloj ulazno-izlaznog podsistema jezgra koji je praktično nezavistan od hardvera - svi detalji vezani za konkretan harver prepušteni su drajverima. Ova slojevita organizacija olakšava posao dizajnerima operativnog sistema, a takođe i dizajnerima uređaja i kontrolera. Proizvođači uređaja, po pravilu, za svoje uređaje pišu drajvere za razne oprativne sisteme, tako da se njihovi uređaji mogu priključiti bez intervencije projektanata samog operativnog sistema. Ono što je za prozvođače uređaja nezgodno je to što za svaki operativni sistem moraju da napišu poseban drajver za konkretan uređaj.

Da ponovimo: srodni uređaji su grupisani u klase, a drajverima je izvršena apstrakcija karakteristika konkretnih uređaja. Programeru se dalje nudi standardni set sistemskih poziva za rad sa ulazno-izlaznim uređajima, kao što su: pozivi za blokovski orijentisan transfer podataka, pozivi za rad sa karakter uređajima, pozivi za rad sa memorijski mapiranim datotekama, pozivi za mrežne funkcije itd. Deo jezgra zadužen za rad sa ulazno-izlaznim uređajima koordiniše široku kolekciju servisa koji su na raspolaganju aplikacijama i ostalim delovima kernela, kao što su:

- upravljanje imenima datoteka i uređaja (*name space*),
- kontrola pristupa datotekama i uređajima,
- kontrola operacija,
- dodela uređaja procesima na korišćenje,
- raspoređivanje ulazno-izlaznih operacija,
- baferovanje, keširanje i *spool* tehnika,
- kontrola statusa uređaja i
- konfiguracija i i inicijalizicija drajvera.

Pri tome, jezgro operativnog sistema čuva informacje o stanju uređaja pomoću različitih struktura podataka koji se nalaze u delu memorije koji pripada jezgru. Tipične strukture koje opisuju stanje uređaja su tabela otvorenih datoteka i strukture koje čuvaju informacije o mrežnim konekcijama.

Većina operativnih sistema ima mehanizam za transparentno prosleđivanje komandi drajveru iz aplikacije (*escape sequence*). Na UNIX sistemima se prosleđivanje komandi drajveru vrši pomoću sistemskog poziva ioctl() koji zahteva tri argumenta: drajver, odnosno uređaj koji se opslužuje tim drajverom, komandu koju uređaj treba da

obavi, ukazivač na stukturu podataka u memorji koja služi za komunikaciju sa drajverom (na primer adresa memorijskog bafera).

Blokirajući i ne-blokirajući sistemski pozivi

Sistemski pozivi dozvoljavaju izbor između blokirajućih i neblokirajućih ulazno-izlaznih operacija. Proces koji izdaje blokirajući sistemski poziv za izvršavanje operacije se blokira i čeka da se operacija izvrši. Proces oslobađa procesor i vraća se u red čekanja na procesor tek onda kada se operacija završi.

Neki procesi šalju ne-blokirajuće sistemske pozive. Proces se posle izdavanja takvog poziva ne blokira, nego nastavlja da radi. Ne-blokirajući sistemski poziv vraća kontrolu procesu vrlo brzo, obično sa pojavom prvih podataka. Po pravilu, ulazno-izlazne operacije se tada preklapaju sa nekom drugom operacijom, ali proces ne koristi podatke pribavljene ulazno-izlaznom operacijom sve dok se ta operacija ne završi. Jedan od primera je komandni interpreter kao proces koji obezbeđuje korisnički interfejs - podaci se primaju sa tastature i istovremeno prikazuju na ekranu. Međutim, komandni interpreter inicira zadatu komandu tek kada se svi podaci prime sa ulaznog uređaja (odnosno kada operater završi unos komande preko tastature). Idealnu primenu ovakvi sistemski pozivi nalaze kod više-nitnih procesa: ukoliko jedna nit obavi sistemski poziv koji je blokirajući za niti (a samim tim i za jedno-nitne) i blokira se, procesor se dodeljuje drugoj niti istog procesa. Posmatrano u celini, proces ne gubi procesor, tako da se u kontekstu više-nitnog procesa ovakav poziv smatra ne-blokirajućim.

Alternativa ne-blokirajućih sistemskih poziva su asinhroni sistemski pozivi. Asinhroni sistemski pozivi posle iniciranja ulazno-izlazne operacije odmah vraćaju kontrolu procesu koji ih je pozvao. Asinhrona ulazno-izlazna operacija će se završiti u nekom budućem trenutku vremena, što će se označiti upisom odgovarajućeg podatka u neku memorijsku strukturu ili putem prekidnog signala. Napravimo razliku između ne-blokirajućeg read() i asinhronog read() sistemkog poziva. Ne-blokirajući poziv će se obratiti procesu sa aktulenim brojem bajtova prenetim do tog trenutka, dok asinhroni ne vraća nikavu informaciju o transferu.

Dakle, blokrajući sistemski pozivi blokiraju proces do završetka operacije, ne-blokirajući vrlo brzo vraćaju kontrolu procesu, a asinhroni trenutno vraćaju kontrolu procesu.

UNIX i problem uniformnosti interfejsa

Objasnićemo moguće rešenje problema uniformnosti izvršavanja operacija na primeru UNIX operativnog sistema.

Uređaji su na UNIX i Linux sistemima predstavljani datotekama. Svaki uređaj je predstavljen jedinstvenom specijalnom datotekom na direktorijumu /dev aktivnog

UNIX stabla. Pomoću te datoteke mogu se izvršiti neke razumne operacije sa uređajem. Na primer, štampač je predstavljen datotekom /dev/lp0, a prvi SCSI disk datotekom /dev/sda. Korisnik jednostavno može iskopirati neku datoteku u datoteku /dev/lp0 i time inicirati štampanje sadržaja te datoteke. To znači da se štampanje datoteke i kopiranje datoteke na drugi disk mogu obaviti istom komandom, zadavajući različite argumente, čime se postiže razumni nivo uniformnosti.

9.5. Servisi koje obezbeđuje ulazno-izlazni podsistem

Jezgro operativnog sistema obezbeđuje više servisa koji se odnose na ulazno-izlazne operacije. U značajnije servisime spadaju raspoređivanje ulazno-izlaznih operacija, baferovanje, keširanje, *spool* tehnika (*spooling*) i upravljanje greškama.

Raspoređivanje ulazno-izlaznih operacija

Pod raspređivanjem ulazno-izlaznih operacija (*I/O scheduling*) podrazumeva se obezbeđivanje dobrog poretka izvršavanja ulazno-izlaznih operacija u cilju postizanja optimalnih performansi. Poredak u kome su zahtevi za izvršavanjem ulazno-izlaznih operacija pristigli u sistem je po pravilu strogo stohastički i gotovo nikada nije optimalan. Na primer, kada se svi prispeli zahtevi za rad sa diskom u jednom trenutku analiziraju, može se odrediti optimalan raspored koji će smanjiti premašenje u vidu vremena potrebnog za pozicioniranje glava, čime će se povećati performanse sistema.

Raspoređivanje operacija se zasniva na redu čekanja koji se definiše za svaki uređaj posebno, a vrši se na osnovu nekog kriterijuma. Kriterijum mora biti definisan tako da se u obzir uzmu prioriteti, ali i da se spreči pojava zastoja i zakucavanja.

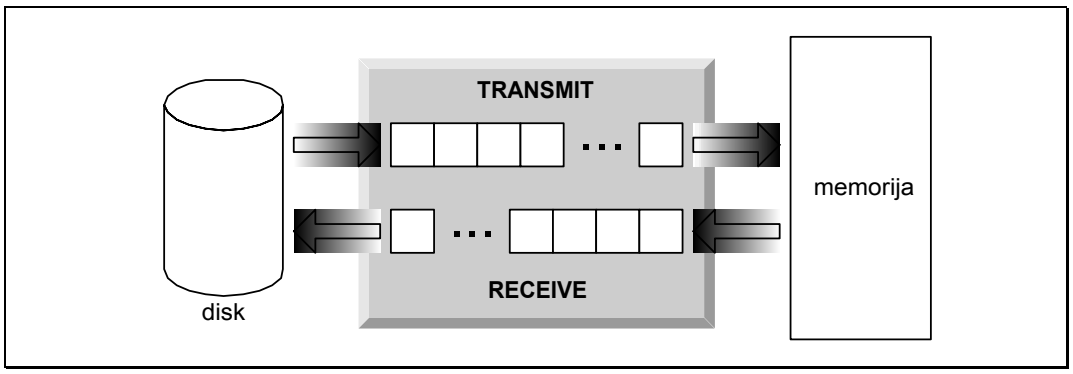
Baferovanje

Bafer (*buffer*) je deo memorije koji funkcioniše na principu proizvođač-potrošač i služi za čuvanje privremenih podataka prilikom transfera između dva uređaja ili između uređaja i aplikacije. Baferovanje (*buffering*) se izvodi iz sledećih razloga:

[1] Usklađivanje različitih brzina između potrošača i proizvođača.

Zbog velike razlike u brzini, ulazno-izlazne operacije mogu bti preklopljene, ali se tada bafer mora podeliti na dva dela, koje altrenativno koriste proizvođač i potrošač. Takav sistem se naziva dvostruko baferovanje (*double-buffering*). Dvostruko baferovanje je jedna od tehnika koja se koristi pri keširanju hard diskova (slika 9.5). Smisao je u sledećem: postoje dva bafera (TRANSMIT i RECEIVE) koji se koriste za baferovanje podataka na relaciji memorija-disk i disk-memorija, respektivno, čime se premošćava razlika u brzini između

memorije i diska. Tehnika dvostrukog baferovanja je, na primer, implementirana u Microsoft SmartDrive.



Slika 9.5. *Primena dvostrukog baferovanja u keširanju hard diskova*

Specijalna varijanta dvostrukog baferovanja se može koristiti u slučaju jednosmernog učestalog prenosa velike količine podataka (*rapid burst of I/O*), čime se osigurava da proces čeka samo u slučaju da su oba bafera puna. Tehnika se može proširiti uvođenjem višestrukog baferovanja (*multiple buffering*).

[2] Prilagođavanje različitih veličina trasfera podataka

Baferi se veoma često koriste u mrežnim okruženjima, gde se velike poruke dele na manje pakete pre slanja. Paketi se primaju u bafer u nedeterminističkom redosledu (što zavisi od pojave grešaka na mreži, kolizije ramova itd.), nakon čega se iz bafera rekonstruiše početna poruka.

[3] Održavanje semantike kopriranja

Ovaj razlog ćemo objasniti na sledećem primeru: korisnikči proces izdaje sistemski poziv za upis, odnosno slanje podataka na neki uređaj. Ukoliko je poziv neblokirajući ili asinhroni, proces dobija mogućnost da u toku slanja izmeni podatke koje želi da pošalje uređaju. To znači da proces može upisati nešto drugo, a ne ono što je inicijalno zadato. Da bi se to izbeglo, odnosno da bi se očuvala semantika kopiranja, operativni sistem definiše bafere u jezgru operativnog sistema u koje kopira korisnički bafer, pa tek onda preda kontrolu korisničkom procesu.

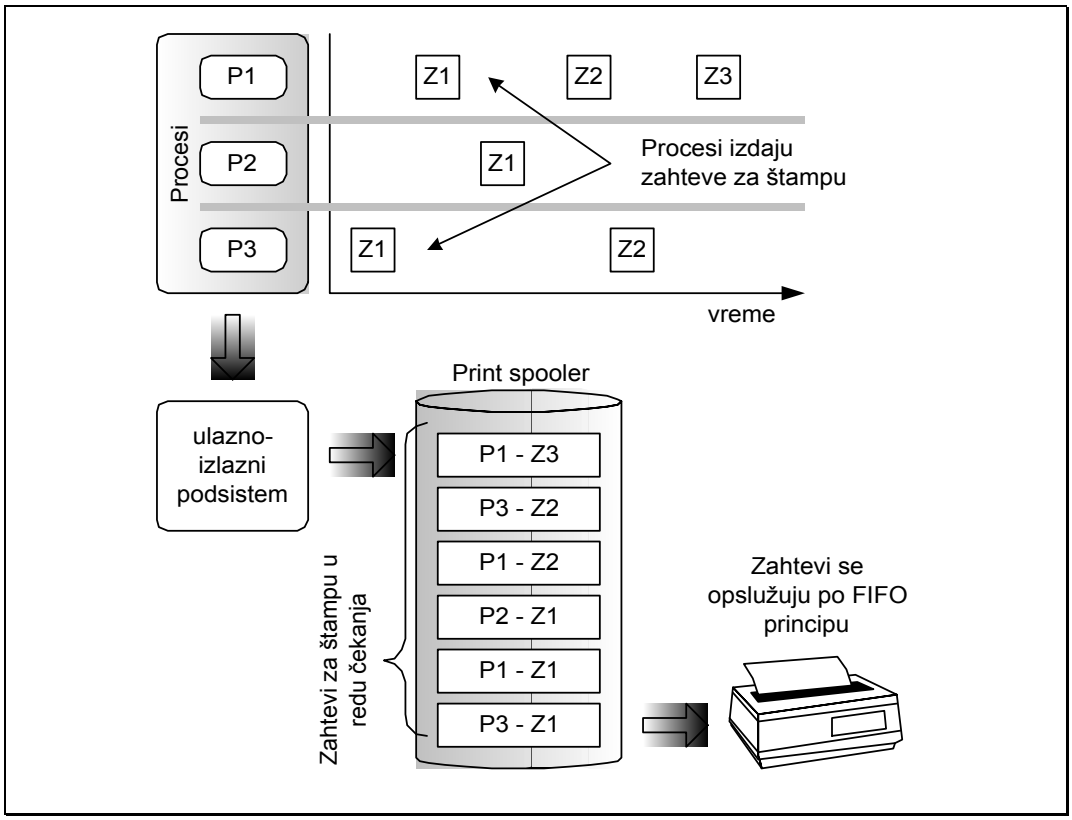
Keširanje

Keš (*cache*) je region brze sistemske memorije koji čuva kopiju podataka, obično sa diska. Pristup podacima u kešu je znatno brži u odnosu na pristup podacima na ulazno-izlaznim uređajima. Keširanje (*caching*) je tehnika kopiranja delova diska u keš memoriju, čime se osetno povećavaju performanse disk I/O sistema.

Najjednostavnije rečeno, razlika između keša i bafera je u tome što bafer čuva trenutno aktuelne podatke, a keš čuva bilo koju kopiju sa diska. Isti memorisjki prostor se može koristiti i za baferovanje i za keširanje. Po pravilu, keširanje je jako značajno i realizuje se u više nivoa, pri čemu se koriste različite tehnike za popunu i zamenjivanje podataka u kešu.

Spool tehnika

Spool je bafer koji privremeno čuva izlazne podatke namenjene nekom nedeljivom uređaju (kao što je štampač). *Spool* tehnika omogućava istovremeni pristup uređajima koji nisu deljivi na sledeći način: procesi upisuju podatke namenjene uređaju na disk, a operativni sistem upravlja tim *spool* baferom tako što opslužuje jedan po jedan zahtev. Na primer, svaki proces koji želi nešto da pošlje na štampu ostavlja svoj zathev u *spool* bafer na disku. Proces koji radi u pozadini upravlja *spool* baferom i obavlja štampu jednog po jednog zahteva (slika 9.6).



Slika 9.6. *Print spooler*

Korišćenje *spool* tehnike ima svoje prednosti:

- proces relativno brzo obavlja postavljanje svog zahteva u spool bafer, a nakon toga je slobodan da dalje obavlja svoje aktivnosti i
- nedeljivi uređaji se koriste kao prividno deljivi, što omogućava većem broju procesa da istovremeno koriste uređaj.

U protivnom, procesi bi morali najpre da čekaju da uređaj postane slobodan da bi uopšte mogli da ga koriste, a zatim bi ponovo postali blokirani čekajući da uređaj završi operaciju koju su inicirali.

Upravljanje greškama

Uređaji i ulazno-izlazni transferi mogu doživeti otkaz iz mnogo razloga, koji mogu biti privremene prirode (na primer, preopterećenje mreže) ili trajne prirode (fizički defekti diska). U slučaju privremenih grešaka, operativni sistem može da se oporavi sam, tako što će ponoviti operaciju (*retry* u slučaju diska, resend() u slučaju mreže). U slučaju trajnih grešaka ovakve pomoći nema, a operativni sistem jedino može da obavesti korsnika da se dogodila greška. Obaveštavanje korisnika obezbeđuju sistemski pozivi koji vraćaju status izvršenja operacije, odnosno izveštaj o uspešnosti. U slučaju greške, sistemski poziv mora precizno da opiše vrstu i lokaciju greške koja se dogodila.

9.6. Prevođenje zahteva u ulazno-izlazne operacije

Prevođenje zahteva u ulazno-izlazne operacije objasnićemo na primeru čitanja datoteke sa diska. Korisnik zadaje ime datoteke koju želi da pročita, a zatim operativni sistem prevodi logičko ime u fizičku adresu prvog bloka datoteke na uređaju (disku). Ignorisaćemo sam proces prevođenja imena u fizičku adresu prvog bloka datoteke i obraditi slučaj blokirajućeg čitanja jednog bloka sa diska. Čitanje se odvija u sledećim koracima:

[1] Proces šalje blokirajući sistemski poziv read().

[2] Sistemski poziv proverava da li su ulazni parametri korektni, i ako jesu nastavlja dalje. Ukoliko je zahtev za čitanje, proverava se stanje keša. Ako zahtev može tako da se zadovolji, podaci se vraćaju procesu iz keša, čime se ciklus čitanja završava, pratktično bez upotrebe samog uređaja.

[3] Ako podaci nisu u kešu, izvršiće se pravi ulazno-izlazni ciklus, što znači da proces mora da se blokira. Pre blokade se poziva drajver (preko poziva potprograma ili preko poruka poslatih jezgru).

[4] Drajver alocira kernelski bafer za prijem podataka i ubacuje zahtev za izvršenjem operacije u red čekanja. Po nekom kriterijumu zahtev dolazi na red. Tada drajver programira odgovarajući kontroler.

[5] Kontroler obavlja ulazno-izlaznu operaciju i odgovarajući transfer podataka.

[6] Trasfer se odvija ili u PIO režimu ili preko DMA kontrolera.

[7] Po okončanju transfera kontroler postavlja prekidni signal koji se obrađuje.

[8] Drajver prima prekidni signal koji ukazuje da je transfer završen, obrađuje status i obaveštava jezgro da je posao završen.

[9] Jezgro prebacuje podatke u korisnički prostor, a potom ukida blokadu sa procesa, odnosno prevodi proces u stanje čekanja na procesor (tranzicija WAIT-READY u dijagramu stanja procesa).

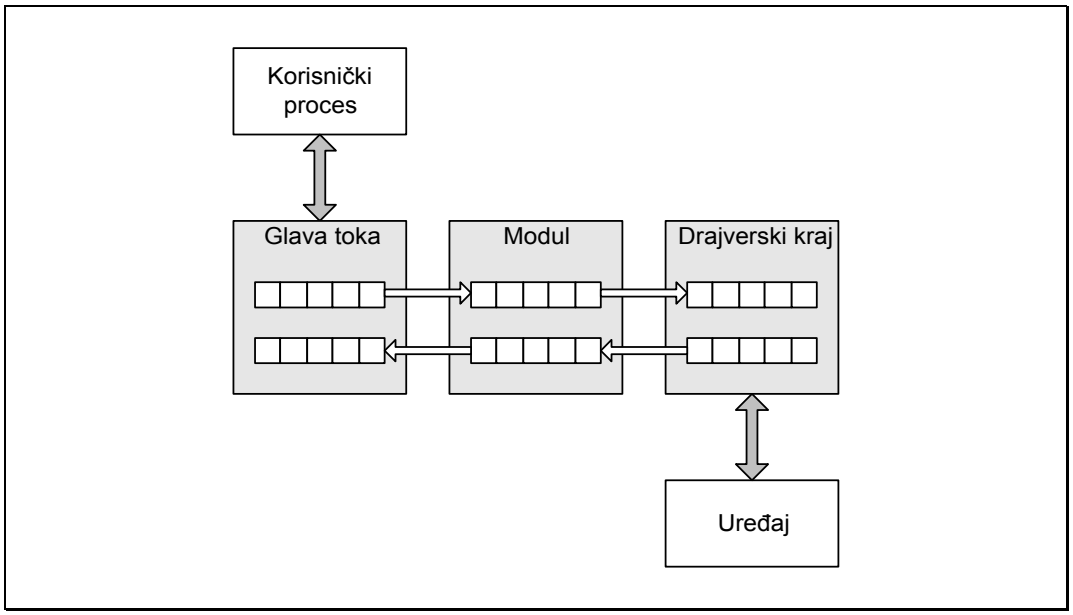
[10] Proces nastavlja sa izvršenjem aktivnosti kada mu dispečer dodeli procesor na korišćenje.

9.7. Mehanizam tokova

Na UNIX System V operativnom sistemu uveden je mehanizam tokova. Jedan tok (*stream*) predstavlja full-duplex konekciju između drajvera i korisničkog procesa, čime je omogućena komunikacija na toj relaciji. Tipična struktura tokova prikaza je na slici 9.7.

Tok se sastoji iz glave (*stream head*), koja komunicira sa korisničkom procesom, drajverskim krajem, koji kontroliše uređaj i proizvoljnim brojem modula između njih. Svaki od ovih delova sadrži par redova čekanja: red čekanja za čitanje (*read queue*) i red čekanja za upis (*write queue*). Za trasfer podataka između redova koristi se sistem prosleđivanja poruka.

Moduli se u strukturu toka mogu ubaciti (*push*) pomoću ioctl() sistemskog poziva. Kako moduli međusobno rzamenjuju poruke, moglo bi doći do prepunjenja nekih modula, tako da se uvodi kontrola toka između redova čekanja. Bez ove kontrole, red prihvata sve poruke i neposredno ih šalje sledećem modulu, ondosno redu sledećeg modula bez baferovanja. Ukoliko se primenjuje kontola toka, red baferiše svaku poruku i ne prihvata novu ukoliko je bafer pun.



Slika 9.7 *Tokovi*

Korisnički proces upisuje podatke na uređaj preko write() ili putmsg() sistemskih poziva. Bez obzira koji se sistemski poziv koristi, glava strukture toka kopira podatke u poruku i prosleđuje je sledećem modulu i tako sve do drajvera. Proces čitanja ide obrnuto, a koriste se sistemski pozivi read() ili getmsg(). Ciklusi ulazno-izlaznih operacija sa tokovima su asinhroni ili ne-blokirajći, osim u slučaju kada korisnički proces komunicira sa glavom strukture toka. Ovo veoma povoljno utiče na performanse, jer se proces blokira (odnosno prelazi u stanje WAIT) samo dok komunicira sa glavom strukture toka, dok je komunikacija između svih ostalih modula asinhrona.

Tokovi predstavljaju modularni, odnosno inkremntalni način za realizaciju drajvera i mrežnih protokola. Na primer, isti mrežni moduli se mogu korsititi za različite mrežne kartice, pri čemu će se razlikovati samo drajverski krajevi. Na primer, Solaris i UNIX System V svoj *socket* mehanizam realizuju preko tokova.

9.8. Performanse ulazno-izlaznog podsistema

Performanse ulazno-izlaznog podsistema znatno utiču na opšte performanse sistema. Po pravilu, kao posledica ulazno-izlaznih operacija javlja se povećanje broja prebacivanja konteksta procesa (procesi se često blokiraju, pa se procesor dodeljuje novim). Takođe, kao posledica se javlja i povećanje broja prekida, čije opsluživanje utiče negativno na opšte performanse sistema. Iako savremeni sistemi mogu da

upravljaju ogromnim brojem prekidnih signala u jednici vremena, treba imati na umu da obrada svakog prekida zahteva tri faze: čuvanje konteksta procesa, izvršavanje prekidne rutine i obnovu konteksta procesa. Dodatno, performanse sistema obara i učestalo kopiranje istih podataka između kernelskih i korisničkih bafera. Mrežni saobraćaj izaziva takođe veliki broj prebacivanja konteksta procesa i prekida.

Mnogi sistemi uvode posebne procesore za opsluživanje ulazno-izlaznih kanala. Pri tome, jedan procesor opsužuje veliki broj periferijskih uređaja, a operativni sistem sve to vidi kao jedan ulazno-izlazni port.

Performansi ulazno-izlaznog podsistema mogu se povećati ukoliko se:

- balansirano koristi procesor, memorija i ulazno-izlazni podsistem, jer preopterećenje u jednom delu dovodi do besposlenosti u drugom delu,
- koriste DMA kontrolera i keš mehanizami,
- smanji broj prekida povećanjem jedinične količine transfera ili uvođenjem tehnike prozivanja (ukoliko se zaposlenost čekanjem može minimizovati),
- smanji broj prebacivanja konteksta procesa i
- smanji broj kopiranja istih podataka na putu od aplikacije do I/O uređaja.

Generalizovano, performanse se mogu uvećati na sledećim nivoima:

[1] Nivo aplikativnog softvera

Kod aplikativnog softvera je fleksibilan i najčešće bezopasan po sistem. Greške koje se mogu načiniti razvojem algoritama za povećanje performansi ulazno-izlaznog podsistema na nivou aplikacije ne mogu dovesti do havarije sistema. Razvoj performansno-orijentisanih algoritama na ovom nivou ne zahteva ponovo punjenje memorije drajverima niti obaranje sistema. Međutim, na ovom nivou se ne može direktno koristiti funkcionalnost jezgra niti jedna struktura jezgra, kao što su niti, poruke unutar kernela itd. Takođe je prisutan i veliki broj prebacivanja konteksta procesa.

[2] Nivo drajvera (jezgro operativnog sistema)

Na ovom nivou se mogu koristiti sve strukture podataka jezgra, tako da se mogu postići bolje performanse. Međutim, jezgro je veoma složen softver u kome važe stroga pravila čije narušavanje može dovesti do ozbiljnih grešaka u sistemu.

[3] Nivo hardvera (uređaj i kontroler)

Na ovom nivou se performanse mogu najviše uvećati specijalnim hardverskim implementacijama, kao što je, na primer, čist hardveski RAID kontroler. Međutim, na nivou hardvera je jako teško nastupiti sa programerske tačke gledišta, jer se obično sve radi u asembleru. Razvoj traje dugo i zahteva specijalnu opremu za testiranje.

10

Sekundarne memorije

Sadržaj poglavlja:

10.1. Struktura diskova
10.2. Priprema diskova za rad
10.3. Nivoi keširanja diskova
10.4. Raspoređivanje zahteva za rad sa diskom
10.5. RAID strukture - realizacija stabilnih sistema
10.6. Priključivanje diskova
10.7. Upravljanje swap prostorom
10.8. Tercijalne memorije

10.1. Struktura diskova

U ovom poglavlju objašnjena je struktura sekundarne i tercijalne memorije, a glavni akcenat je stavljen na diskove i njihove preformanse. Pristup podacima na sekundarnim memorijama može biti direktan ili sekvencijalan. Metod pristupa podacima i brzina samog uređaja određuju i njegovu namenu. Na primer, diskovi se koriste kao osnovni uređaj sekundarne memorije - na njima se nalaze operativni sistem, programi i podaci koji se obrađuju. Pristup podacima na diskovima je direktan, a sami diskovi su dovoljno brzi uređaji. Za razliku od njih, pristup podacima na magnetnim trakama je sekvencijalan, tako da se one koriste isključivo za arhiviranje podataka.

Geometrija diskova

Disk uređaj se sastoji iz rotacionih kružnih ploča koje rotiraju oko zajedničke ose. Površine ploča su presvučene magnetnim materijalom. Svaka površina ima pridruženu glavu za čitanje i pisanje koja čita ili upisuje podatke sa magnetnih ploča. Za većinu disk uređaja, samo jedna glava može biti aktivna u jednom trenutku vremena. Glave se linearno pokreću pomoću sopstvenog servo-sistema, čime im je, uz rotaciju ploča, omogućen pristup svim delovima magnetne površine. Kompletnim uređajem upravlja kontrolna eletronika diska.

Procesor računara i disk komuniciraju putem disk kontrolera (*disk controller*). Disk kontroleri različitih diskova pružaju isti interfejs ka ostatku računara i time pojednostavljuju pristup podacima na disku, tj. oslobađaju računar potrebe za poznavanjem načina rada i kontrolom elektro-mehanike diska. U ostale funkcije kontrolera spada i baferovanje podataka koje treba upisati na disk, keširanje diskova i automatsko obeležavanje neispravnih sektora diska.

Površina diska je podeljena u koncentrične prstenove - staze (*tracks*), a svaka staza je dalje podeljena na sektore (*sectors*). Broj sektora po stazi je kod starijih diskova najčešće isti za sve staze, dok noviji diskovi dele spoljašnje staze na veći broj sektora, čime obezbeđuju jednaku magnetnu površinu za sve sektore. Tipična količina podataka koja se može upisati u jedan sektor je 512 bajtova, i to je najmanja količina podataka koja se može upisati na disk ili pročitati sa diska.

Sve površine magnetnih ploča su jednako podeljene na staze i sektore. To znači da se glave za čitanje i pisanje na svim pločama diska u jednom vremenskom trenutku nalaze na istim stazama. Ekvidistantne staze svih ploča čine jedan cilindar (*cylinder*). Datoteke koje nisu smeštene u okviru jednog cilindra su fragmentisane - pomeranje glava sa jedne staze na drugu prilikom čitanja ovakvih datoteka unosi kašnjenje. Performanse diska se mogu uvećati smeštanjem datoteke u okviru jednog cilindra kad god je to moguće.

Geometrija diska je u opštem slučaju određena brojem magnetnih površina (odnosno glava za čitanje i pisanje), cilindara i sektora, i čuva se u posebnoj memorijskoj lokaciji sa baterijskim napajanjem - CMOS RAM. Operativni sistem čita vrednosti koje opisuju geometriju diska prilikom podizanja sistema ili inicijalizacije drajvera. Dalje se trodimenzionalnim adresiranjem (*head, cylinder, sector*) može ostvariti pristup svim delovima diska u cilju pristupa postojećim podacima ili dodele prostora za nove podatke. Na primer, podatak koji je upisan na drugu površinu, u stazu 3, na sektoru 5 može se adresirati uređenom trojkom (*head, cylinder, sector*)=(2,3,5).

Savremeni disk uređaji

Razvoj mehaničkih delova savremenih disk uređaja usmeren je ka povećavaju performansi. Što se tiče mehanike, zapaža se:

[1] Smanjivanje vremena poziociniranja (seek time)

[2] Povećanje rotacione brzine

Povećanjem rotacione brzine smanjuje se rotaciono kašnjenje, a samim tim i povećava brzinu čitanja/upisa sa magnetnog mediuma.

Po pitanju oblika podataka i gustine magnetnog medijuma zapaža se:

[1] Povećanje gustine magnetnog mediuma

Stalno povećanje gustine magnetnog medijuma ima za posledicu povećanje kapaciteta, povećanje brzina čitanja/upisa sa magnetnog mediuma, pad cena po GB disk prostora;

[2] Zonska tehnika.

Magnetna tehnologija sve više napreduje, pa je gustina staze sve veća što iziskuje potrebu efikasnijeg korišćenja magneta. Unutrašnje staze, staze bliže centru imaju manju površinu pa samim tim i manje magneta. U tom kontekstu, za razliku od prvih generacija diskova, uvodi se tehnika zona na disku, gde se cilindri grupišu u zone iste gustine (BZR = *bit zone recording*). Na taj način se povećava i kapacitet diska kao i brzina čitanja sa mediuma, ali kontrolna eletronika diska ima mnogo složeniji zadatak da upravlja promenljivom gustinom staza, odnosno kompleksnijim oblikom podataka na disku;

[3] Alternativni sektori za upravljanje defektima

Alternativni sektori se koriste za zamenu fabričkih ili novonastalih defekata na disku. Savremeni disk uređaji omogućavaju 100% nazanačenog kapaciteta i pored značajnog broja defekata na mediumu koji se pojavljuju bilo u fabrikaciji uređaja, bilo naknadno u toku rada disk uređaja. Da bi se omogućilo 100% iskorišćenje kapaciteta, savremeni diskovi imaju rezervne regione na mediumu (*spare regions*). Taj prostor se rezerviše na kraju staza, cilindera i zona kako bi se defektivnim blokovima našla zamena.

Po pitanju kontrolne elektronike diska stalno se povećavaju performanse i uvode nove funkcije:

[1] Upotreba procesora solidne snage na disku

Kontrolna ploča dika je bazirana na sopstvenom procesoru, koji izvršava softver zadužen za kompletno upravljanje svim delovima disk uređaja. Za savremene diskove, ugrađeni procesori su tipične jačine 32 bitnog CPU kao što je Intel-80386. Pored sopstvenog procesora sve je veća količina RAM memorije prisutna na kontrolnoj ploči diska, koja ima više namensku ulogu.

[2] Keširanje na disk uređaju (*On-Board Cache*)

U savremenim disk uređajima prisutna je relativno velika količina RAM memorije, reda veličine više MB, koja obavlja dve glavne funkcije. Prva i

osnovna funkcija je usklađivanje brzine transfera disk mediuma i disk interfejsa (*speed-matching buffer*), a tu funkciju imaju praktično svi disk uređaji i stari i novi. Druga moderna funkcija je disk keširanje na nivou disk uređaja. Moredni disk uređaji koriste različite keš tehinke. Keš memorija se deli na ceo broj keš linija koji se nazivaju keš-segmenti. Veličina segmenta kod nekih uređaja odgovara veličini staze, ali kod većine je deo staze. Postoje keš algoritmi koji dinamički određuju broj segmenata i njihovu veličinu kako bi što bolje ostvarili odnos pogodaka u kešu. Funkcinisanje disk keširanja se bazira na tehnici čitanja unapred (*read-ahead*). Na svako zahtevano čitanje sa diska, pored zahtevanih podataka, čitanje se nastavlja. Prekid čitanja se može završtiti iz više razloga, ali mnogi algoritmi će agresivno nastaviti čitanje sve dok ne popune segment. Neki algoritmi staju u slučaju da se naiđe na kraj staze ili cilindra, a neki će prekinuti čitanje ako naiđu novi zahtevi za disk.

Za svaki pristup disku se obavlja analiza keša. U slučaju ciklusa čitanja, može se dogoditi totalni pogodak, delimični pogodak ili totalni promašaj. U slučaju ciklusa upisa, ukoliko dođe do pogotka u kešu, podaci se ažuriraju ili im se poništva validnost, a neki diskovi imaju asinhronu tehniku upisa (*write-back*), koja je opasna jer disk nema bateriju i u slučaju gubitka napajanja podaci nestaju, a postoje različite tehike za smanjenje šanse za gubitak. U slučaju potpuno popunjenog keša, mora se odlučiti koji će segment biti zamenjen i obično je to neka LRU tehnika. Potrebno je istaći osobine koje se malo koriste kao što su "*read-on-arrival*" i "*write-on-arrival*" koji čitaju ili pišu odmah posle pozicinirania i aktiviranja glave bez rotacionog kašnjenja, a logički redosled se podešava u disk baferu. Retko se koriste, jer su efikasne samo za transfere koji su jednaki veličini staze, dok su za male ili jako velike transfere neefikasni.

[3] Logičko-fizičko mapiranje blokova na disku (LBN->PBN)

Tradicionalno mapiranje jednodimenzione adrese LBN (*Logical Block Number*) u trodimenzionu fizičku adresu adresu PBN (*cylinder/head/sector*) obavlja se počevši od LBN=0 koje se pridružuje prvom bloku na početnom cilindru i onda se LBN blokovi dodaju sukcesivno, sve do punog kapaciteta diska. Međutim brojni faktori (kao što su *Bit Zone Recording* i upravljanje defektima) utiču na dosta izmena i osetno povećanje složenosti u mapiranju.

Kao što je već opisano BZR omgućava povećanje performansi i kapaciteta savremenih diskova, ali je zato firmveru koji upravlja diskom prilično otežan posao. Za svaki LBN se najpre mora odrediti zona na disku kojoj pripada, a onda se u datoj zoni saglasno njenoj gustini određuje PBN (*cylinder/head/sector*).

Upravljanje defektrima se odvija na 2 načina: tehnikom klizanja (*slipping*) i tehnikom ponovne dodele LBN adrese (*relocation*). Tehika klizanja se obavlja na dva nivoa: klizanjem na nivou bloka (*sector slipping*) i klizanjem na nivou staze (*track slipping*) i ta tehika se obalja jedino na nivou fizičkog formatiranja celog disk uređaja (*low-level formating*). U procesu formatiranja, preskače se defektini sektor i ažurira LBN->PBN tabela i tabela defekata, tako da su svi sektori logički

sekvencijalni. Stazno klizanje je isto, stim što se preskačke cela staza u slučaju da sadrži defete i ažuriraju se tabele. Ovakvo formatiranje se radi prilikom fabrikacije diska ili kada se detektuju defekti na disku, ali tada se neminovno uništava sadržaj na disku. Mnogo bolja metoda u slučaju diska sa podacima je ponovna dodela LBN, kada se svakom defktivnom bloku nadje alternativa, ažurira se lista loših blokova i svaka sledeća referenca se upućuje na novu alternativnu adresu.

Iz svega izloženog se zaključuje da je LBN->PBN relativno kompleksna operacija koja uključuje kao prvo identifikaciju zone, potom uticaj pomeranja (*skew*) na nivou staze ili cilindra i na kraju analizu tabela loših blokova i korekciju na bazi njih.

[4] Obrada komandi u redu čekanja (*Command Queuing*)

Moderni diskovi podržavaju obradu komandi u redu čekanja, što znači da disk može da primi više komandi u redu čekanja, a zatim organizuje njihov optimalni redosled izvršavanja i eventualnu kombinaciju komandi. Interni disk raspoređivač (*scheduler*) može mnogo moćnije da radi nego bilo koji eksterni raspoređivač iz tog razloga što najbolje poznaje trenutni položaj sopstvenog servo-sistema i karakteristike svojih mehaničkih i baferskih komponenti.

Karakteristike savremenih disk kontrolera

Savremene disk kontroleri, namenjene za PC arhitekturu, po pitanju složenosti, funkcionalnosti i performansama mogu se podeliti u tri funkcionalne grupe.

[1] Kompaktni kontroleri

Bazirani su na savremenom stepenu razvoja integrisanih kola kada imamo pojavu jako složenih i kvalitetnih periferijskih kontrolera u vidu jednog integrisanog kola (*on-chip design*). Njihova velika prednost je jednostavnost, pouzdanost i niska cena. Kompaktni disk kontroleri zahtevaju kvalitetna periferijska IC kola koja poseduju dovoljnu funkcionalnost da samostalno upravljaju periferalom (one chip design). Takva IC kola su dostupna i realna i sa takvim izborom periferijskog IC kola, realizacija kompaktnog kontrolera zahteva još elementarnu dekodersku logiku za selekciju kao i bafersku logiku za spregu periferijskog IC kola sa sistemskom magistralom. Pri tome se može upotrebiti baferska memorija za usklađivanje između sistemske magistrale i periferala. Usklađivanje se odnosi na nivou brzine i na nivou veličine podataka. Glavne karakteristike kompaktnih kontrolera jesu te da nemaju poseban lokalni procesor za upravljanje kontrolerom i ne poseduju veliku bafersku memoriju na kontrolerskoj kartici.

[2] Baferski kontroleri (*FIFO architecture*)

Baferski kontroleri su optimalna ili kompromisna klasa za većinu operativnih sistema i većinu aplikativnih primena kada je u pitanju kompromis između

performansi koje postižu i cene koštanja. Baferski kontroleri su bolji od kompaktnih kontrolera mada su i skuplji. Sadrže relativno veliku baferskу memoriju koja je organizovana na FIFO principu. Upotreba velike baferske memorije omogućava izuzetno dobro usklađivanje brzina diska i sistemske magistrale, kao jednu redukovanu vrstu keširanja diska koja se obezbeđuje tehnikom čitanja unapred (*read-ahead*). FIFO arhitektura uključuje u sebi jednostavnije lokalne procesore za upravljanje FIFO baferom i radom kontrolera. FIFO kontroler ima malo vreme procesiranja komande (*controller command overhead*) jer ne primenjuje složene keš algoritme koji intenzivno pretražuju gde su podaci, da li su validni itd. U odnosu na keš kontrolere (*full caching controller*) imaju daleko manje memorije na kartici, tipično od 32K ili 64K. Konstruktori ovakvog tipa kontrolera, ističu superiornost baferskih kontrolera nad konkurencijom kada su u pitanju sekvencijani pristupi disku. Cena im je daleko niža od potpunih keš kontrolera (*full caching*) kontrolera.

[3] Keš kontroleri (*full caching architecture*)

Predstavljaju najsloženije i najkvalitetnije kontrolerske kartice. Ispoljavaju kvalitetne osobine po pitanju performansi i to zahvaljujući boljim lokalnim procesorima kao i složenom algoritmu koji koriste za keširanje diska. Cena im je relativno visoka. Keš arhitektura uključuje kvalitetnije lokalne procesore (CPU), 16 bitne ili 32 bitne, veliku keš memoriju, mali FIFO bafer za spregu sa sistemskom magistralom. Keš arhitektura dosta komplikuje i poskupljuje dizajn, ali poboljšava performanse sistema. Algoritam za upravljanje keš memorijom je relativno složen. Veličina keš memorije može biti relativno velika i reda je više megabajta.

Savremeni trenutak, odnosno prošle dekada dale su velika konkurentska takmičenja između raznih disk intefejsa i disk kontrolera. Neki opšti zaključci koji bi se mogli postaviti su:

- Od brojnih disk interfejsa na sceni su opstale dve klase: SCSI i ATA
- Kompaktni kontroleri su veoma retki, praktično više ne postoje.
- Baferski (FIFO) kontoleri su se pokazali kao optimalna arhitektura za slučaj pojedinačnih disk uređaja. U kombinaciji sa ugrađenim disk keširanjem samog operativnog sistema, daju mnogo bolje rezultate nego potpuni keš kontroleri, a cena im je daleko niža.
- Potpuni keš kontoleri su našli svoje mesto ali u kombinaciji sa RAID tehnikom, tako da kombinuju dve sofisticirane funkcije: disk keširanje i RAID kao tehniku za povećanje brzine, kapaciteta i pouzdanosti. Pored svih RAID funkcija, svaki RAID kontroler poseduje moćan CPU, obično RISC i veliku količinu keš memorije tako da predstavlja u suštini puni keš kontroler.

IDE i SCSI diskovi

IDE i SCSI diskovi su danas najčešće korišćene klase diskova na PC računarima. IDE uređaji (*Integrated Drive Electronics*) su dobili naziv po elektronici integrisanoj na samom uređaju. Ovoj klasi uređaja pripadaju relativno jeftini diskovi solidnih performansi - kapaciteta do 160GB, sa brzinama okretanja ploča od 5.400-7.200 obrtaja u minuti. Kontroleri za IDE uređaje su ugrađeni na matičnim pločama računara i pružaju interfejs ka računaru pri brzinama od 33-133 Mbit/s. Realna brzina čitanja i pisanja na same magnetne površine znatno je manja, tako da na kontroleru postoji bafer u koji se podaci smeštaju pre upisa na sam disk - na ovaj način se sprečava da performanse sistema značajno padnu prilikom rada sa diskovima. Svaki IDE kontroler ima dva kanala - primarni (*primary*) i sekundarni (*secondary*), a na svaki kanal se mogu vezati najviše dva uređaja u odnosu master-slave. Uređaji vezani na različite kanale mogu da primaju ili šalju podatke računaru istovremeno. Na jednom kanalu može biti samo jedan uređaj aktivan u jednom trenutku vremena. Svaki IDE uređaj ima preklopnike koje treba podesiti u željeni režim rada - master ili slave pre vezivanja na kontroler. Administrator sistema određuje način na koji će uređaji biti vezani na IDE kontroler - uređaje treba vezati tako da se performanse sistema održe na najvišem mogućem nivou.

SCSI uređaji predstavljaju profesionalnu klasu širokog spektra uređaja - diskova, CD-ROM uređaja, traka, skenera itd. Kontroler za SCSI diskove nije integrisan na matičnim pločama i kupuje se odvojeno. Elektronika na SCSI kontroleru je komplikovanija, interfejs ka računaru je brži (do 320Mbit/s), a na kontroler je moguće vezati od 7 do 15 uređaja, u zavisnosti od kontrolera. SCSI uređaji se ne nalaze u master-slave odnosu, već se na kontroler vezuju prema prioritetima. Prioritet svakog uređaja određen je njegovim identifikacionim brojem koji se postavlja preko preklopnika na uređaju. Princip je sledeći: prioritet uređaja sa ID=0 je najviši i treba ga dodeliti sistemskom disku, a prioritet uređaja sa ID=15 je najniži. Identifikacioni broj ID=7 rezervisan je za SCSI kontroler.

10.2. Priprema diskova za rad

Priprema diskova za rad obuhvata sledeće administrativne postupke:

- formatiranje diskova
- kreiranje particija
- kreiranje sistema datoteka

Formatiranje diskova

Formatiranje diska je proces upisivanja oznaka koje predstavljaju granice staza i sektora na magnetni medijum, čime se uvodi red u magnetni haos neformatirane površine. Disk koji nije formatiran ne može da se koristi.

Terminologija može biti zbunjujuća za korisnike DOS/Windows operativnih sistema - pod formatiranjem se na DOS i Windows sistemima podrazumeva proces kreiranja sistema datoteka. Taj proces je u literaturi poznat i pod imenom formatiranje visokog nivoa (*high level formatting*). Na primer, neka hipotetički korisnik u DOS-u ili u komandnoj liniji Windows operativnog sistema zada sledeću komandu:

```
A:> format c:
```

Ovom komandom se inicira kreiranje sistema datoteka na primarnoj particiji prvog diska - ovim se ne formatira disk na niskom nivou. Formatiranje diska na niskom nivou (*low level formatting*) je priprema magnetne površine diska za rad.

U prošlosti su se novi diskovi isporučivali potrošačima potpuno prazni - disk se sastojao samo od koncentričnih krugova magnetnog materijala, tako da je pre kreiranja sistema datoteka na njemu, morao da se formatira. Proces formatiranja niskog nivoa, odnosno fizičko formatiranje, obavlja funkciju deljenja koncentričnog kruga na sektore. Svaki sektor se formira kao struktura koja ima zaglavlje (*header*) u kome piše adresa sektora, potom sledi 512 bajtova podataka i na kraju završni slog (*trailer*) u kome se obično navodi ECC (*error correction code*), odnosno informacija za detekciju i oporavak od greške. Prilikom svakog upisa u taj sektor, ECC bajtovi se ponovo preračunavju i upisuju na disk.

Većina diskova koji se danas proizvode su fabrički formatirani na niskom nivou, tako da ovaj proces nije potrebno ponavljati. U nekim slučajevima se to i ne preporučuje, jer sam proces može biti komplikovan zbog zonskog zapisa (odnosno, promenljive gustine medijuma) i nekih specijalnih funkcija kao što je automatska zamena loših blokova.

Diskovi koje treba formatirati se obično isporučuju sa specijalnim softverom koji je za to namenjen. Što se tiče ATA tehnologije, u prošlosti su se za formatiranje koristili disk menadžeri (kao što je *Seagate Disk Manager*). Ovi programi ne mogu da se koriste za formatiranje novijih ATA diskova, osim ako ih proizvođač ne isporuči za rad sa konkretnim modelom (modelima) diskova. SCSI interfejs ima veoma inteligentnu komandu za fizičko formatiranje, koja potpuno transparetno formatira ceo disk i remapira defektivne blokove. Proces traje dugo, naročito kod velikih diskova i po pravilu svaki SCSI kontroler ima u svom firmveru (*firmware*) program za formatiranje diskova.

U principu, fizičko formatiranje se ne radi, osim slučaju da se pojave novi fizički oštećeni blokovi. Ove površine treba izbegavati prilikom upisa podataka, jer su podaci

upisani u njih najčešće nepovratno izgubljeni. Kvalitetniji diskovi imaju ugrađen mehanizam za automatsko obeležavanje neispravnih sektora, ali je glavna logika ugrađena u sisteme datoteka. Pomenućemo još jednom: broj fizički neispravnih blokova ima tendenciju rasta, kao i verovatnoća daljeg gubitka podataka. Disk čija je površina fizički oštećena treba što pre zameniti ispravnim diskom.

Oporavak od oštećenih blokova

Novi modeli diskova zbrinjavaju fabrička oštećenja raznim tehnikama remapiranja, tako da korisnik od proizvođača dobija disk koji je praktično bez defekata. Međutim, ukoliko oštećenja ipak postoje, programi za logičko formatiranje će markirati te blokove kao neispravne i oni se neće koristiti. Problem nastaje u slučaju da se nova oštećenja pojave u toku rada. U tom slučaju postoji više mogućnosti. Jedan od načina je da se očuvaju podaci, a da se preko specijalnih programa (*scandisk* na DOS/Windows sistemima, *fsck* na UNIX/Linux sistemima) proširi lista defekata. Druga metoda je remapiranje koje zahteva ili da se disk fizički formatira ili da se iskoriste specijalni programi za remapiranje defekata bez gubitka podataka. U svakom slučaju, proces remapiranja se može obaviti na dva načina:

[1] Korišćenjem rezervnih-alternativnih sektora

Za svaki defektivan sektor nađe se alternativni sektor i tom alternativnom se dodeli ista adresa kao defektivnom, koji zauvek ispada iz igre. Kada se sistem obrati defektivnoj adresi, unutrašnji kontroler diska automatski translira adresu na nivu poziciju. Ova metoda obara performanse jer algoritmi koji raspoređuju zahteve za rad sa diskovima gube pravu informaciju o cilindru. Zbog toga, većina diskova na svakom cilindru ostavlja altrenativni prostor koji se može iskoristiti za remapiranje, a tehnika remapiranja se uvek trudi da remapira sektor sa alternativnim na istom cilindru.

[2] Korišćenjem tehnike klizanja sektora (*sektor slipping*)

Kiizanje sektora je alternativa rezervnim sektorima. Fukcioniše na sledeći način: prilikom zamene defektivnog bloka alternativom, svi ostali sektori između njih klizaju na gore. Na primer, neka je sektor 17 defektivan, sektor 203 rezervni, a prvi sektor pre alternative 202. Prilikom oporavka, sektor 202 se pomera u rezervni (203), 201 u 202, 200 u 201, a na kraju defektivni sektor 17 u 18. Na ovaj način relativni položaj sektora ostaje isti.

Kreiranje particija i sistema datoteka

Da bi se formatirani disk mogao koristititi pod bilo koji operativnim sistemom, obavlja se dalja priprema diska u dve faze. U prvoj fazi disk se mora izdeliti na particije u kojima mogu da se kreiraju sistemi datoteka.

Na disku se može kreirati samo jedna particija (*partition*), a po potrebi disk se može podeliti i na nekoliko delova - particija, koje se ponašaju kao odvojeni diskovi. Veći broj particija se kreira ukoliko je na računar sa jednim diskom potrebno instalirati više operativnih sistema. Pri tom, svaki operativni sistem koristi svoju particiju, a po potrebi može da čita i piše datoteke na druge, ukoliko podržava rad sa sistemima datoteka koji su na tim particijama kreirani. Bez particija, instalacija većeg broja operativnih sistema bila bi moguća samo na sistemima sa dva ili više diskova. Deljenje diska na particije se često vrši i na računarima na kojima je instaliran samo jedan operativni sistem - na taj način se lako mogu razdvojiti sistemske i korisničke datoteke, čime se pojednostavljuju postupci arhiviranja datoteka.

Informacije o svim particijama diska čuvaju se u prvom logičkom sektoru, odnosno u prvom sektoru prve staze sa prve površine diska. Ovaj sektor je poznat pod imenom *Master Boot Record* (MBR) i njemu BIOS pristupa svaki put kad se računar uključuje. MBR sadrži mali program koji očitava particionu tabelu, proverava koja je particija aktivna, i očitava prvi sektor aktivne particije (*boot sector*). U *boot* sektoru se nalazi mali program čijim pokretanjem započinje *boot-strap* rutina, odnosno punjenje RAM memorije operativnim sistemom.

Originalni koncept particionisanja diskova na PC računarima dozvoljavao je najviše četiri particije po jednom disku, što se ubrzo pokazalo kao nedovoljno. Razlozi su sledeći: nemoguće je na samo četiri particije instalirati više operativnih sistema (naročito ako neki od njih zahtevaju dodatne particije, kao što su *swap* i *boot*), *boot manager*, i odvojiti particiju za korisničke podatke.

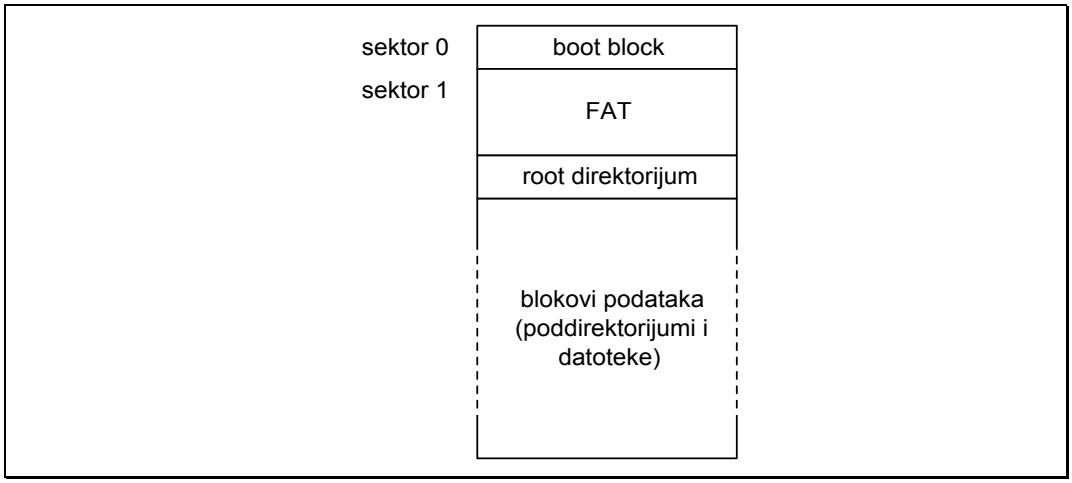
Problem je rešen uvođenjem *extended* particije, koja služi kao okvir u kome se mogu kreirati nekoliko logičkih particija. Logičke particije se ponašaju kao primarne, ali se razlikuju po načinu kreiranja. Na ovaj način se maksimalni broj particija po disku uvećava. Informacije o logičkim particijama se vode u *boot* sektoru *extended* particije, koji se još naziva i extended partition table. Na disku može postojati najviše jedna extended particija.

Znači. podela diskova na particije može biti jednostavna ili složena. Nakon podele diska na particije, prelazi se na drugu fazu - formatiranje visokog nivoa (logičko formatiranje), odnosno kreiranje samih sistema datoteka. Programi za logičko formatiranje kreiraju kontrolne strukture pomoću kojih se upravlja sistemom datoteka. Kontrolne strukture i način dodele datoteka je različit za sve sisteme datoteka.

Bootstrap rutina

Proces podizanja operativnih sistema sa diska zahteva da na sistemu postoji program inicijalni program koji će otpočeti podizanje operativnog sistema (*bootstrap* program). Za većinu operativnih sistema, inicijalno podizanje se izvršava kroz više faza, pri čemu je početak koda u ROM memoriji. Taj kod obično nije dovoljan i teško

se menja, tako da se dopunjuje programima na disku. Na primer, na MS-DOS operativnom sistemu, kod koji podiže operativni sistem se naziva *boot block*.



Slika 10.1. *Boot block na FAT sistemu datoteka*

Kada se računar uključi, BIOS pokreće seriju testova hardvera nakon čega kreće proces podizanja sistema. Podizanje operativnog sistema započinje u ROM memoriji računara, a nastavlja se sa diska, sa kog se najpre pročita prvi sektor. Kod upisan u prvom sektoru diska (*master boot record*) najpre identifikuje aktivnu particiju u particionoj tabeli, a zatim izvršava kod upisan u boot sektoru aktivne particije, koji dalje puni memoriju kernelom. U boot sektoru se nalazi mali program koji je zadužen da pokrene punjenje RAM memorije operativnim sistemom. Delovi koda kojim se memorija puni u toj ranoj fazi nalaze se na fiksnim područjima diska, a ne u sistemima datoteka, zato što u toj fazi nema kernela, pa nemamo podršku za sistem datoteka. Rana faza podizanja operativnog sistema se završava učitavanjem jezgra.

10.3. Nivoi keširanja diskova

Savremeni trenutak karakteriše izuzetni brz razvoj hardvera, osobito procesora i memorijskih komponenti čije performanse rastu, a cene padaju. Ta pojava uslovljava intenzivnu upotrebu distribuirane procesorske snage i memorije, odnosno disk keširanja na raznim nivoima.

Postoji više nivoa disk keširanja:

[1] Keširanje na nivou operativnog sistema (*file caching*)

Svaki operativni sistem poseduje sopstveno keširanje (*built-in file caching*). Keširanje na ovom nivou naziva se *file-caching* zato što keš sadrži kopije datoteka. Projektanti operativnih sistema, svoje algoritme keširanja čuvaju u tajnosti kao zaštićene patente. Sa druge strane, veliki broj naučnih radova je posvećen poboljšanju disk keširanja u raznim pravcima.

[2] Keširanje na nivou disk kontrolera (*HBA level caching*)

Na početku 90-tih godina bilo je više realizovanih punih keš kontolera (*full-caching*), ali su baferski kontroleri u kombinaciji sa disk keširanjem na nivou operativnog sistema davali mnogo bolje rezultate, a za mnogo nižu cenu. Koncept keširanja na nivou disk kontrolera se, praktično, ne koristi za klasične disk kontrolere, ali je nezaobilazni deo u najkvalitetnijim i najsloženijim disk kontrolerima kao što su RAID kontroleri.

[3] Keširanje na nivou disk uređaja (*disk drive level caching*)

Diskovi postaju vrlo inteligentni uređaji, koji pored ogromnih poboljšanja u mehaničkim karakteristikama imaju solidne procesore i veliku količinu memorije koja služi za disk keširanje. Disk uređaj, preciznije disk keš memorija, optimalno je mesto za tehniku predikovanog čitanja jer niko ne poznanje bolje svoj servo-sistem i raspored podataka na mediumu, kao sam disk uređaj.

[4] RAID keširanje (*caching in RAID*)

RAID je izvanredan koncept na polju povećanja kapaciteta, performansi, a pre svega pouzdanosti. Veoma snažan uticaj na RAID performanse ima disk keširanje na samom RAID kontroleru, tako da svaki RAID kontroler pored RAID funkcionalnosti predstavlja i potpuni keš kontroler.

[5] Keširanje na nivou aplikacije (*application level caching*)

Svi pomenuti keš nivoi su po prirodi opšte namene, generalni. Najkvalitetniji su svakako adaptivni keš algoritmi, koji svoju organizaciju mogu da prilagode trenutnim disk zahtevima. Međutim, svaka kvalitetnija aplikacija, bilo da reč o bazama podataka ili o određenim vrstama video servera ili Internet servera ima veoma specifičan rad sa diskom, ima svoje datoteke i strukture od izuzetne vaznosti za sopstvene performanse. Zahvaljući dobrom poznavanju sopstvenih potreba u radu sa diskom, kvalitena aplikacija rezerviše svoj memorijski prostor i kešira disk saglasno svojim potrebama, po pravilu osetno bolje nego generalni disk keš na nivou operativnog sistema (*OS level cache*).

Tesno povezana sa disk keširanjem su razna poboljšanja postojećih sistema datoteka i algoritmi za raspoređivanje zahteva za rad sa diskom.

Pomenućemo više savremenih tehnika koje su, uglavnom, implementirane na operativnom sistemu UNIX.

[1] Stazno-bazirani pristup disku.

Ideja stazno-baziranog pristupa disku (*Track-based disk access*, odnosno *Track-alligned Extents*) je u modifikaciji sistema datoteka koja će usloviti pristup disku budu u formi cele staze (*track*). Time se izbegava rotaciono kašnjenje, a preskaču se i gubici u vremenu zbog promene staze ili cilindra (*track crossing overheads*).

[2] Rešavanje problema malih datoteka.

Klasični UNIX sistem datoteka se modifikuje u kontekstu udruživanja meta podataka sistema datoteka sa direktorijumskim blokovima i blokovima samih datoteka kako bi se povećela performanse u radu sa malim datotekama (*Embedded I-nodes and Explicit Grouping*).

[3] *Disk Shuffling*

Najčešće korišćeni podaci grupišu se oko centra diska smajujući vreme pozicioniranja.

[4] *Freeblock Scheduling*

Tehnika je bazirana na pretpostavci da će se podaci pročitani pri rotaciji, koji nisu zahtevani, kasnije koristiti.

Algoritmi za raspoređivanje disk zahteva su oblast koja se takođe intenzivno razvija. U radu sa diskovima, svi višeprocesni operativni sistemi primenuju tehiku obrade u redu čekanja koja omogućava zadavanje više komandi istovremeno. Izvšavanje komandi će biti prema algoritmu koji će dinamički obraditi prispele zahteve i odrediti redosled izvrešenja komandi. Počelo se davno sa prvim algoritmom FCFS (*First Come First Served*) a iza njega su nastupile brojne modifikacije. Među prvim modifikacijima bio je SSTF (*Shortest Seek Time First*), algoritam koji uzima samo vreme pozicioniranja kao kriterijum za narednu disk komandu. Bolja varijanta je, svakako, bio SCAN baziran na lift principu. Kod SCAN algoritnma servo sistem se fiktivno pomera naviše i na tom putu se obrađuju svi pripeli zahtevi. Postoje razne varijacije SCAN algoritma koje se odnose na smer kretanja servo sistema. Vrlo savremeni algoritam je SATF (*Shortest Access Time First*), mada je takođe poznat pod nazivom SPTF (*Shortest Positioning Time First*). Algoritam računa ukupno mehaničko kašnjenje koje ima dve komponente, vreme pozicioniranja i veme rotacionog kašnjenja, pri čemu je vreme rotacije osetno veće u odnosu na poziciniranje. Algoritam ima složenije preračunavanje, ali daje veoma dobre rezultate. Jedan od poslednjih i trenutno najboljih algoritama SPCTF (*Shortest Positioning –w /Cache Time First*). Algoritam radi u kombinaciji sa ugrađenim disk keširanjem na samom disk uređaju i odluku donosi na bazi stanja u kešu, rotacije i pozicioniranja.

10.4. Raspređivanje zahteva za rad sa diskom

Operativni sistem mora da obezbedi efikasno iskorišćenje hardvera, a ulazno-izlazni uređaji predstavljaju usko grlo računarskog sistema po pitanju performansi. Uzmimo na primer magnetne diskove - vreme pristupa kod diska zavisi od sledećih vremenskih komponenti:

- vremena pozicioniranja glava za čitanje i pisanje sa tekuće pozicije na zahtevani cilindar,
- vremena rotacionog kašnjenja, odnosno vremena potrebnog da se ispod magnetne glave postavi zahtevani blok podataka (direktno zavisi od brzine okretanja rotacionih površina diska) i
- brzine transfera podataka sa magnetnog medijuma, koja zavisi od gustine medijuma i brzine okretanja rotacionih površina diska.

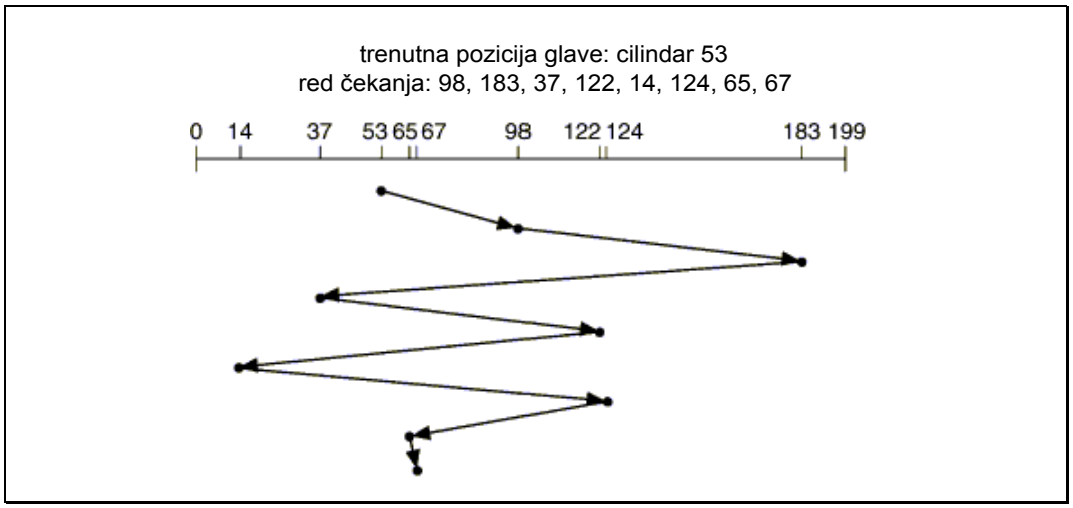
Brzina disk transfera (*bandwidth*) je količnik ukupnog broja prenetih bajtova i ukupnog vremena koje obuhvata ove tri komponente. U višeprocesnoj okolini, u jednom trenutku vremena postoji veliki broj zahteva za rad sa diskom. Pravilnim raspređivanjem ovih zahteva (*disk scheduling*), ukupno vreme pozicioniranja ili rotacionog kašnjenja se može smanjiti. Svaki zahtev koji je upućen disku sadrži sledeće informacije: da li se zahteva operacija čitanja ili pisanja, adresu bloka na disku, adesu memorijskog bafera i broj bajtova koje treba preneti.

Više zahteva može stići istovremeno, a u jednom trenutku disk može obraditi samo jedan. Postoje više algoritama koji će obaviti raspoređivanje zahteva za rad sa diskom.

FCFS

Algoritam FCFS (*first come, first served*) je najjednostavniji algoritam, koji zahteve prosleđuje u onom redosledu u kome su stigli. FCFS obezbeđuje krajnje fer odnose prema prispelim zahtevima, ali i znatno loše performanse. Uzmimo sledeći primer: neka je trenutna pozicija glava za čitanje i pisanje na cilindru 57, a u red čekanja za disk zahtevi pristižu po sledećem redu: 98, 183, 37, 122, 14, 124, 65, 67.

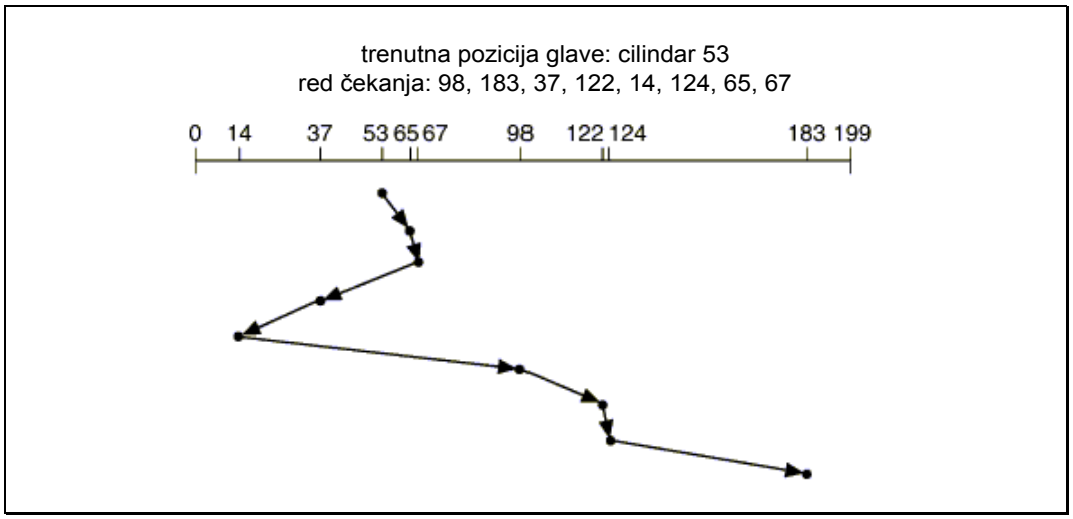
Performanse zavise od ukupnog pomeraja glava za čitanje i pisanje pri opsluživanju zahteva iz reda, a savršeno je jasno da su performanse slabije sa većim pomeranjem. Relativne performanse izrazićemo ukupnim brojem cilindara koje glave za čitanje i pisanje prelaze pri opsluživanju zahteva. Za slučaj sa slike 10.2., ukupni pomeraj glava diska iznosi 640 cilindara.



Slika 10.2. *FCFS algoritam*

SSTF

SSTF algoritam (*shortest seek time first*) opslužuje zahteve na sledeći način: od prispelih zahteva najpre se uzima onaj koji će izazvati najmanji pomeraj glava (*seek time*). Uzmimo isti primer koji je iskorišćen za ilustraciju FCFS algoritma. SSFT će opslužiti zahteve prema sekvenci prikazanoj na slici 10.3, a ukupni pomeraj glava diska iznosi 236 cilindara.

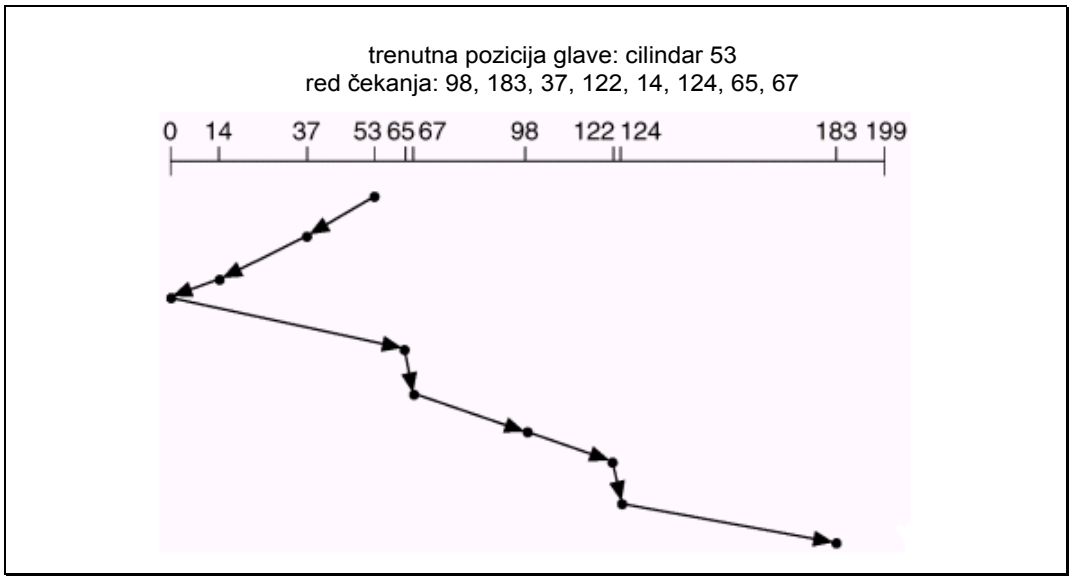


Slika 10.3. *SSTF algoritam*

Algoritam podseća na SJF (*shortest job first*) algoritam za raspoređivanje procesora i optimalan je po pitanju vremena pozicioniranja. Međutim, kod SSTF algoritma prisutan je problem zakucavanja (*stravation*). Glave mogu ostati veoma dugo u jednoj zoni opslužujući zahteve koji unose male pomeraje, tako da zahtevi čiju su cilindri daleko od trenutne pozicije mogu dugo čekati u redu.

SCAN

SCAN algoritam radi na principu lifta koji se naizmenično kreće od prizemlja do vrha zgrade. Algoritam naizmenično pomera glave od početka do kraja diska i unazad i opslužuje zahteve koji se nalaze na tekućem cilindru. Na ovaj način se rešava problem zakucavanja.



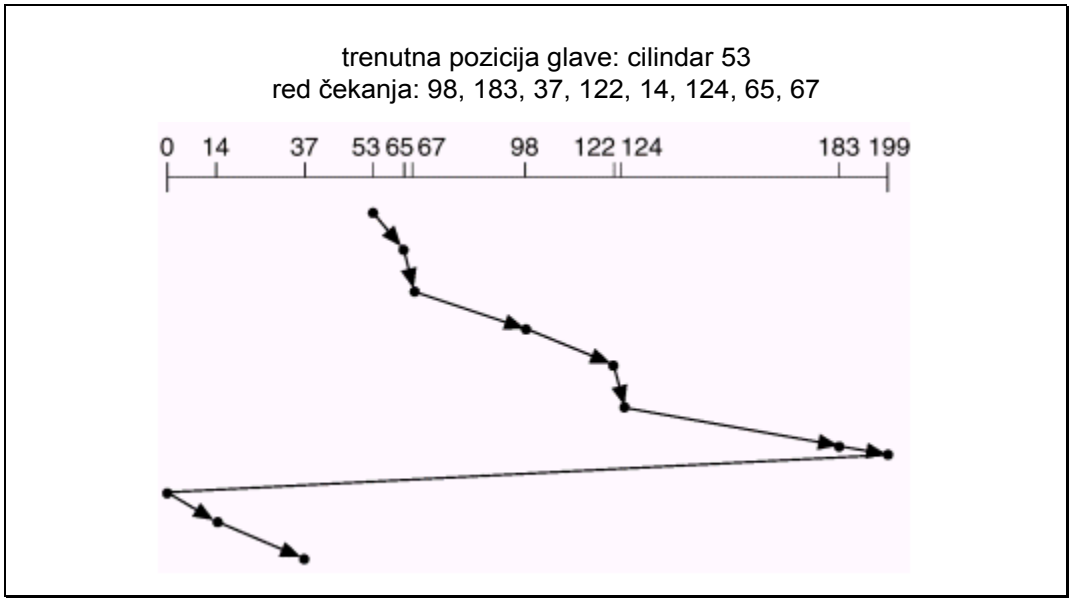
Slika 10.4. *SCAN algoritam*

Po performansama, SSTF može nadjačati SCAN algoritam, što nije slučaj za ovu sekvencu zahteva (SSTF će uneti ukupni pomeraj od 236, a SCAN od 208 cilindara). Prilikom obrade zahteva, SCAN daje prednost unutrašnjim cilindrima u odnosu na periferne.

C-SCAN

Kružni SCAN algoritam (C-SCAN, *circular* SCAN) je varijanta SCAN algoritma koja razrešava problem favorizovanja unutrašnjih cilindara. Izmena se sastoji u tome da se zahtevi opslužuju samo u jednom smeru. Kada glave dođu do poslednjeg cilindra,

pomeraju se na početak, ne opslužujući zahteve na tom putu. Posle toga se nastavlja opsluživanje zahteva od početnog do krajnjeg cilindra.

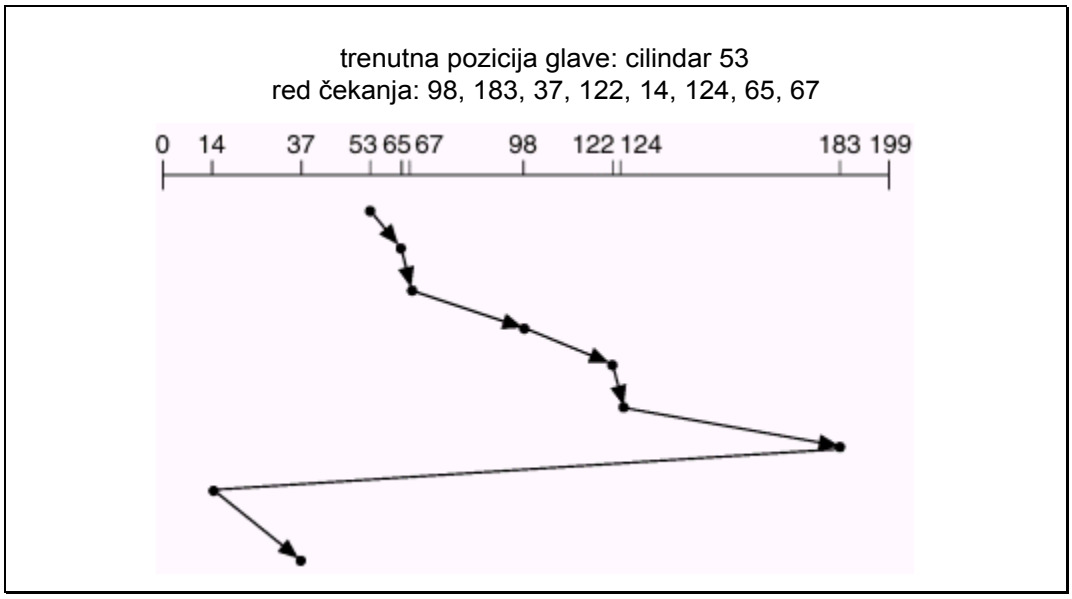


Slika 10.5. *C-SCAN algoritam*

LOOK i C-LOOK

Algoritmi LOOK i C-LOOK su modifikacije SCAN i C-SCAN algoritama. Glave se ne pomeraju do kraja ili početka diska, nego do poslednjeg zahteva koji se nalazi u redu čekanja u tom smeru. Pri tome, LOOK opslužuje zahteve u oba smera, a C-LOOK samo u rastućem smeru do poslenjeg zahteva u redu, nakon čega se vraća na zahtev najbliži početku diska, kao što je prikazano na slici 10.6.

Algoritmi su ime dobili po tome što "gledaju" na kom se cilindru nalazi poslednji zahtev u tom smeru i kreću se dog cilindra. U praksi se umesto SCAN algoritama uvek koriste LOOK algoritmi.



Slika 10.6 *C-LOOK algoritam*

Izbor algoritma za raspoređivanje disk zahteva

Kako izabrati najbolji algoritam za rapoređivanje disk zahteva? Jasno je da su svi algoritmi bolji od FCFS, ali je teško odrediti koji je od njih najbolji, jer performanse samih algoritama zavise od optrećenja, odnosno prispelih zahteva za rad sa diskom. Kružne varijante SCAN i LOOK algoritama imaju mnogo bolju raspodelu opsluživanja i nemaju problem zakucavanja (starvation), koji je prisutan pri radu sa SSTF algoritmom. C-LOOK je najbolje rešenje za jako opterećene sisteme.

Neophodno je naglasiti da su prethodno pomenuti algoritmi zastareli. Modernije varijate ovih algoritama minimiziraju obe mehaničke komponente - i pozicinirаnje i rotaciono kašnjenje (pri čemu rotaciono kašnjenje ima dominantan uticaj na performanse savremenih diskova). Jedan takav algoritam je SATF (*shortest access time first*), koji radi na principu SSTF algoritma, ali pri odabiru sledećeg zahteva iz reda računa obe mehaničke komponente. Najsavremeniji algoritmi uzimaju u obzir i keširanje na samom disk uređaju. C-LOOK algoritam u kombinaciji sa ugrađenim disk keširanjem daje najbolje rezultate, što potvrđuju brojne simulacije iz otvorene literature.

Poželjno je da se program za raspoređivanje zahteva za rad sa diskom realizuje kao poseban modul kako bi se mogao lako zameniti.

10.5. RAID strukture - realizacija stabilnih sistema

Upis na disk može da se završi na 3 načina:

- uspešno okončan upis - svi sektori su uspešno upisani na disk,
- delimični otkaz - otkaz je nastupio u sredini transfera, neki su sektori dobro upiani, a neki su oštećeni i
- potpuni otkaz ništa nije upisano jer je ciklus upisa odmah otkazao.

Sistem se može lako opraviti od otkaza ukoliko se koristi tehnika vođenja dnevnika transakcija (*journalling*) u kombinaciji sa keširanjem i NVRAM memorijom. RAID strukture koje koriste princip ogledala i parnosti takođe mogu pomoći u realizaciji stabilnih sistema.

Koliko god da su savremeni diskovi brzi i kvalitetni, po performansama daleko zaostaju za savremenim procesorima. Jedna od brojnih tehnika kojima se povećaju perfomanse sekundarne memorije je RAID (*redundant array of inexpensive disks*). RAID tehnikom se više diskova na koje se paralelno upisuju podaci mogu spojiti u jedan veliki logički disk čije su performanse znatno veće. Pored toga, RAID može obezbediti solidnu pouzdanost jednog takvog sistema uvođenjem redundanse. U prošlosti RAID sistemi su bili jeftina altrenativa velikim skupim diskovima (RAID sistem je činio veći broj manjih jeftinih diskova), a danas predstavljaju sinonim za brzinu i pouzdanost. Osnovne karakteristike RAID sistema su:

[1] Paralelizam i konkurentnost operacija

RAID uvećava performanse paralelizmom i konkurentnim operacijama. Paralelizam je tehnika deljenja podataka između različitih diskova. Minimalna jedinica podataka koja se može smestiti kontinualno na jednom disku je traka (*stripe unit*). U slučaju malih traka, jedan veći logički blok fizički se smešta na više diskova. U slučaju velikih traka, mali zahtevi se opslužuju sa jednog diska, što može dovesti do paralelnog izvršenja više nezavisnih disk operacija u istom trenutku (konkurentnost operacija).

Minimalno deljenje je deljenje na nivou bita (*bit-level stripping*), gde svaki bit ide na poseban disk. Sledeći nivo podele je na nivou bajta (*byte-level stripping*), gde svaki bajt ide na poseban disk. Karaktersitike bit i bajt nivoa deljenja su da u svakom pristupu logičkom disku učestvuju svi diskovi u paraleli, čime se postiže maksimalna brzina transfera. Međutim, konkurentnost celog RAID sistema je jednaka 1, jer na ovaj način RAID opslužuje samo jedan zahtev u jednom trenutku. Sledeći slučaj je deljenje na nivou bloka (*block-level striping*). Blokovi se dele između različitih fizičkih diskova na sledeći način: logički blok i smešta se na disk po formuli (i mod n) + 1, gde je n broj diskova u RAID strukturi. Veličina bloka za deljenje je stepen broja 2 - može biti manja od veličine bloka na disku, može biti jednaka 512 bajtova ili umnožak od 512 bajtova. Ukoliko su

blokovi veći, situacija je povoljnija u oba slučaja - postiže se paralelizam operacija pri velikim transferima a konkurentnost za male zahteve.

[2] Povećanje pozdanosti uvođenjem redundanse

Svaki disk ima svoju pouzdanost koja se meri srednjim vremenom otkaza (obično 100.000 časova). Verovatnoća otkaza RAID strukture koju čini N diskova uvećava se N puta. Problem pouzdanosti se rešava uvođenjem redundanse, odnosno čuvanjem dodatnih informacija koje obezbeđuju mogućnost potpunog povratka podatak u slučaju otkaza jednog diska. Najprostija tehnika koja se može koristiti je tehnika ogledala (*mirroring*, *shadowing*), odnosno uvođenje identične kopije svakog diska. Svaki logički disk se sastoji od dva fizička diska, tako da se u slučaju otkaza jednog, podaci mogu pročitati sa drugog diska. Podaci će nestati samo u slučaju da oba diska istovremeno otkažu. Srednje vreme otkaza jednog ogledala diskova zavisi od dva faktora: srednjeg vremena otkaza pojednačnih diskova (t_F) i srednjeg vremena potrebnog za oporavak od otkaza jednog diska (t_R). U principu, umesto otkazalog diska stavlja se novi disk, kojim se ogledalo opravlja - svi podaci sa zdravog diska se kopiraju na novi. U ovoj situaciji do otkaza ogledala može doći usled otkaza zdravog diska u toku kopiranja podataka. Srednje vreme za koje par u ogledalu gubi podatke (t_{MF}) jednako je: $t_{MF} = t_F^2 / 2t_R$.

Osnovni nedostatak tehnika ogledala je što se za jedan logički disk koriste dva diska, čime iskorišćenje kapaciteta pada na 50%. Druga tehnika koja se koristi za povećanje pozdanosti je tehnika parnosti (*parity*).

RAID nivoi

Kao što je već rečeno, deljenje podataka na trake povećava performasne, ali smanjue pouzdanost. Tehnika ogledala povećava pouzdanost, ali smanjuje iskorišćenost i performanse upisa. Uvođenje parnosti je dobar kompromis. Postoji šest osnovnih RAID nivoa koji se razlikuju po performansama, pouzdanosti i ceni. RAID nivoi su prikazani na slici 10.7. U šemi na slici, oznaka P (*parity*) ukazuje na disk parnosti ili bitove za korekciju greške, a C (*copy*) predstavlja kopiju diska u ogledalu. Svi nivoi su ilustrovani na primeru RAID sistema sačinjenog od četiri diska.

[1] RAID 0 (slika 10.7-a)

RAID 0 predstavlja RAID konfigraciju u kojoj je *stripe* jedinica na nivou jednog ili više blokova podataka (*block-striping*). RAID 0 je veoma brza konfiguracija - ima najbolje performnse čitanja i upisa, ali kako nema nikavu redundansu, otkaz bilo kog diska znači gubitak svih podataka.

[2] RAID 1 (slika 10.7-b)

RAID 1 predstavlja RAID konfigraciju u kojoj svaki disk ima svoje ogledalo. RAID 1 ima najveći utrošak prostora i najgore performanse upisa.

[3] RAID 2 (slika 10.7-c)

RAID 2 predstavlja RAID konfigraciju koja je poznata pod nazivom RAID sa memorijskim stilom korekcije (*memory style error correcting code ECC organisation*). Memorije imaju ECC algoritam koji za svaki bajt ima 3 ekstra bita, potrebna za detekciju i korekciju jednobitnih grešaka. RAID 2 ima organizaciju deljenja podataka na bit ili bajt nivou, a bez obzira na broj diskova podataka, potrebna su još tri diska za ECC koja mogu sačuvati podatke u slučaju otkaza bilo kog diska. RAID 2 je dobar po pitanju paralelizma, bolji je od RAID 0 po pitanju utroška diskova, ali se praktično ne koriiti.

[4] RAID 3 (slika 10.7-d)

RAID 3 predstavlja RAID konfigraciju koja ima organizaciju deljenja podataka na bit ili bajt nivou, a po pitanju redundanse prvi put se uvodi parnost za diskove (*bit-interleaved parity*). Ideja je u sledećem: za razliku od memorije kod koje je jako teško odrediti tačnu poziciju grešaka, kod diskova se tačno zna gde je nastupila greška, a tamo gde nema geške ti bitovi su dobri. Prema tome, za oštećeni bit, dovoljan je jedan bit da čuva parnost, i na bazi te parnosti može da se rekonstruiše oštećeni bit. Za sve diskove podataka dovoljan je jedan disk parnosti. Naravno da izračuvanje i upis parnosti osetno smanjuju performanse ciklusa upisa u odnosu na RAID-0, ali su one mnogo bolje nego kod RAID 1 i RAID 2, a keširanje može drastično da poveća performanse upisa.

[5] RAID 4 (slika 10.7-e)

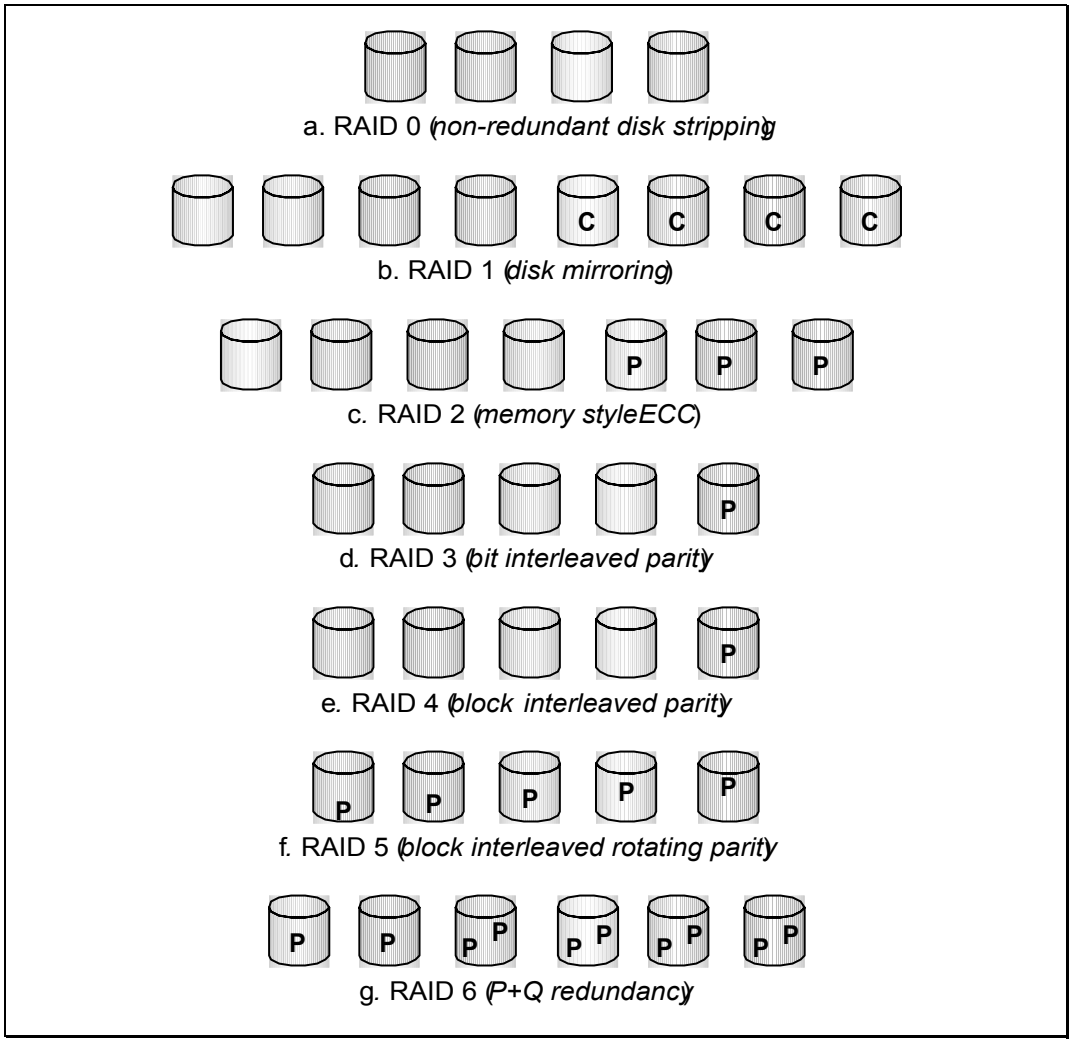
RAID 4 predstavlja RAID konfiguraciju koja ima organizaciju deljenja podataka na blok nivou, a po pitanju redundanse koristi parnost za diskove na nivou bloka (*block-interleaved parity*). Za N blokova (sa N diskova) dovoljan je jedan blok parnosti, koji je jednak veličini *stripe* jedinice. RAID 4 ima dobre osobine, kao što su kombinacija paralelizma i konkurentnosti, ali i dve velike mane. U svakom ciklusu upisa učestvuje i disk parnosti, što dovodi do preopterećenja ovog diska koji postaje usko grlo u sistemu. Međutim najgori slučaj za RAID 4 su mali ciklusi upisa, koji praktično obuhvataju samo jedan disk - zbog parnosti, moraju se čitati sve ostale *stripe* jedinice, pročitati stara parnost, a zatum izračunati i ponovo upisati nova parnost. Mali cilkusi upisa izazivaju četiri vrste ulazno-izlaznih ciklusa (*read-modify-write*).

[6] RAID 5 (slika 10.7-f)

RAID 5 (*block-interleaved distributed parity*, *rotating parity array*) predstavlja RAID konfiguraciju veoma sličnu RAID 4. U ovom slučaju ne postoji jedan disk parnosti, nego svih N+1 diskova predstavljaju i diskove podataka i diskove parnosti. Parnost se upisuje u levom simetričnom rasporedu. RAID 5 od osnovnih struktura predstavlja najbolju kombinaciju: poseduje paralelizam, konkurentnost, dobar je za velike upise, a svi diskovi su ravnomerno opterećeni. Jedino ostaje i dalje problem malih upisa (*read-modify-write*) koji se delimično razrešava preko keša na RAID nivou.

[7] RAID 6 (slika 10.7-g).

RAID 6 (*P+Q redundancy scheme*) predstavlja jedinu RAID kombinaciju koja može razrešiti prolem u slučaju otkaza više od jednog diska (ukoliko se koriste prethodne šeme, osim RAID 1, svi podaci se gube ako istovremeno otkažu bar 2 diska). Da bi postigao takav stepen pouzdanosti, RAID 6 nastaje uvođenjem dodatnog diska (Q) i *Read-Solomon* koda u RAID 5 šemu.

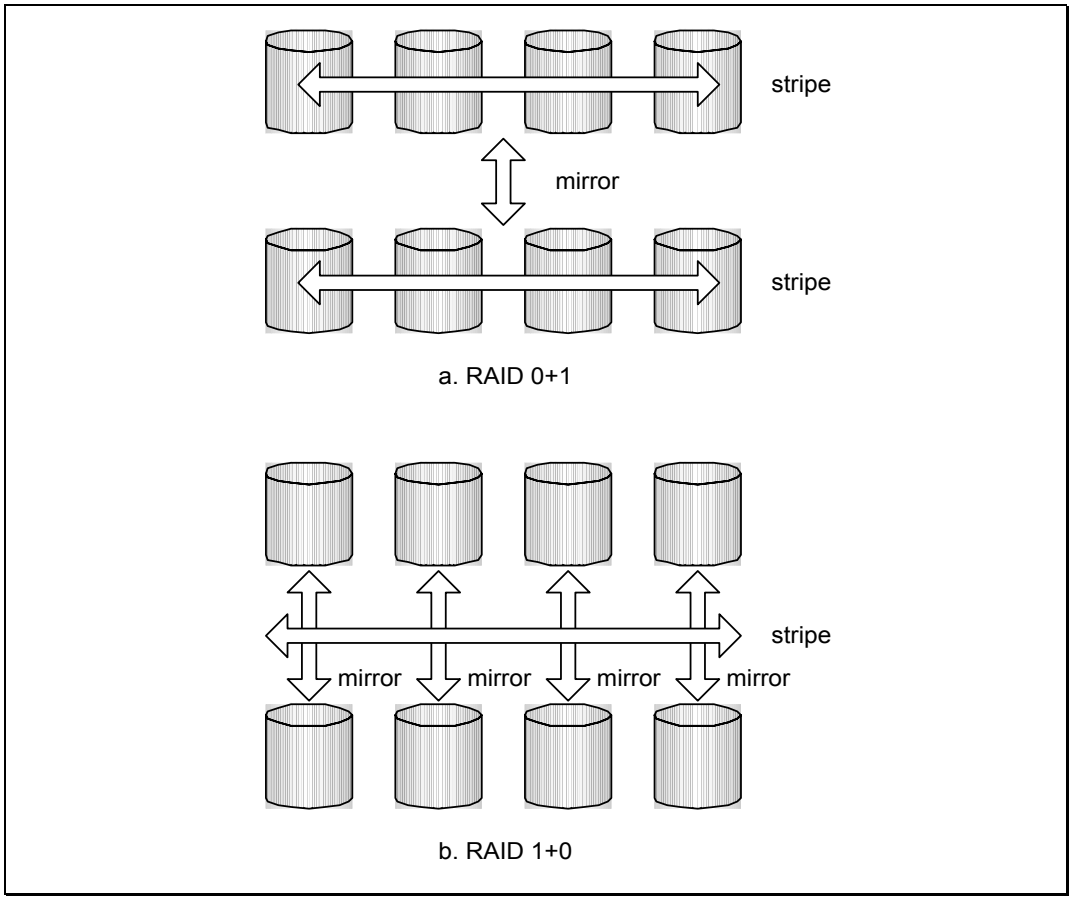


Slika 10.7. *RAID nivoi*

[8] RAID (0+1) i RAID (1+0).

Ovo su dve veoma kvalitetne kombinacije tehnike 0 i 1, prikazane na slici 10.8. RAID 0 obezbeđuje visoke performanse, a RAID 1 visoku pouzdanost. Međutim,

kombinacija ostaje i dalje skupa jer udvostručava broj diskova. U kombinaciji 0+1, skup diskova deli podatke, a potom se sve *stripe* jedinice u celini kopiraju u svoje ogledalo. Druga kombinacija 1+0 znači da svaki disk ima svoje ogledalo a onda se podaci dele između ogledala. 1+0 je teorijski bolji u odnosu na 0+1. Kod 0+1 u slučaju da jedan disk otkaže cela grupa stripe jednica je neraspoloživa na 2 mesta, dok je ona u ogledalu raspoloživa. U slučaju 1+0 otkaz jednog diska utiče na pouzdanost samo te stripe jedince koja ima samo jednu raspoloživu kopiju.



Slika 10.8. *RAID 0+1 i RAID 1+0*

Ako disk otkaže, vreme za opravak može biti značajno i zavisi od RAID konfiguracije. Oporavak je najbrži u slučaju RAID 1, kad se podaci na novi disk upisuju sa ogledala. U slučaju RAID nivoa sa parnošću, vreme oporavka je značajno veće. RAID 0 se koristi u aplikacijama gde su performanse kritične, a pouzdanost nije previše bitna. S druge strane, RAID 1 se korsiti gde je pozdanost od izuzetne važnosti. RAID 1+0 i 0+1 se koriste tamo gde su i brzina i perfomanse jako kritične, kao što su, na primer, baze podataka. Za veoma velike sisteme zbog osetno manjeg zahteva u broju

diskova, koristi se RAID 5, a u slučaju zahteva za visokom pouzdanošću koristi se RAID 6.

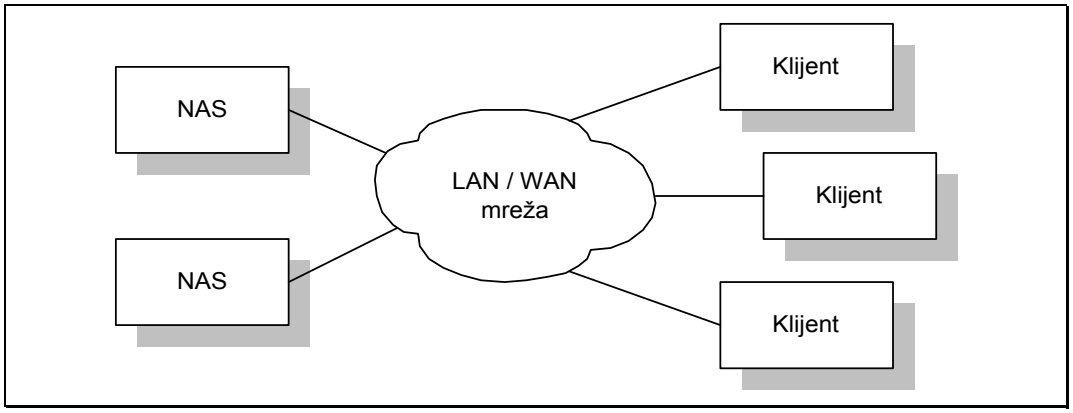
Prilikom projektovanja RAID sistema, projektant treba da odgovori na brojna pitanja:

- koji će RAID nivo da se primeni ,
- koliko diskova će biti u RAID sistemu,
- koliko grupa parnosti će se napraviti i
- kolika će biti veličina stripe jedinice.

10.6. Priključivanje diskova

Diskovi se na računarske sisteme priključuju na dva načina. Jedan način je povezivanje preko ulazno-izlaznog porta (*host-attached storage*) i koristi se u manjim računarskim sistemima. Drugi način je povezivanje preko mreže (NAS, *network-attached storage*) i koristi se u velikim distribuiranim sistemima.

U slučaju NAS, specijalnom sistemu diskova klijenti pristupaju preko mreže, kao što je prikazano na slici 10.9. Klijenti koriste rpc (*remote procedure call*) pozive. Osnovni nedostatak NAS načina povezivanja je prenos velike količine podataka preko mreže, što loše utiče na performanse ali i na optrećenost mreže.



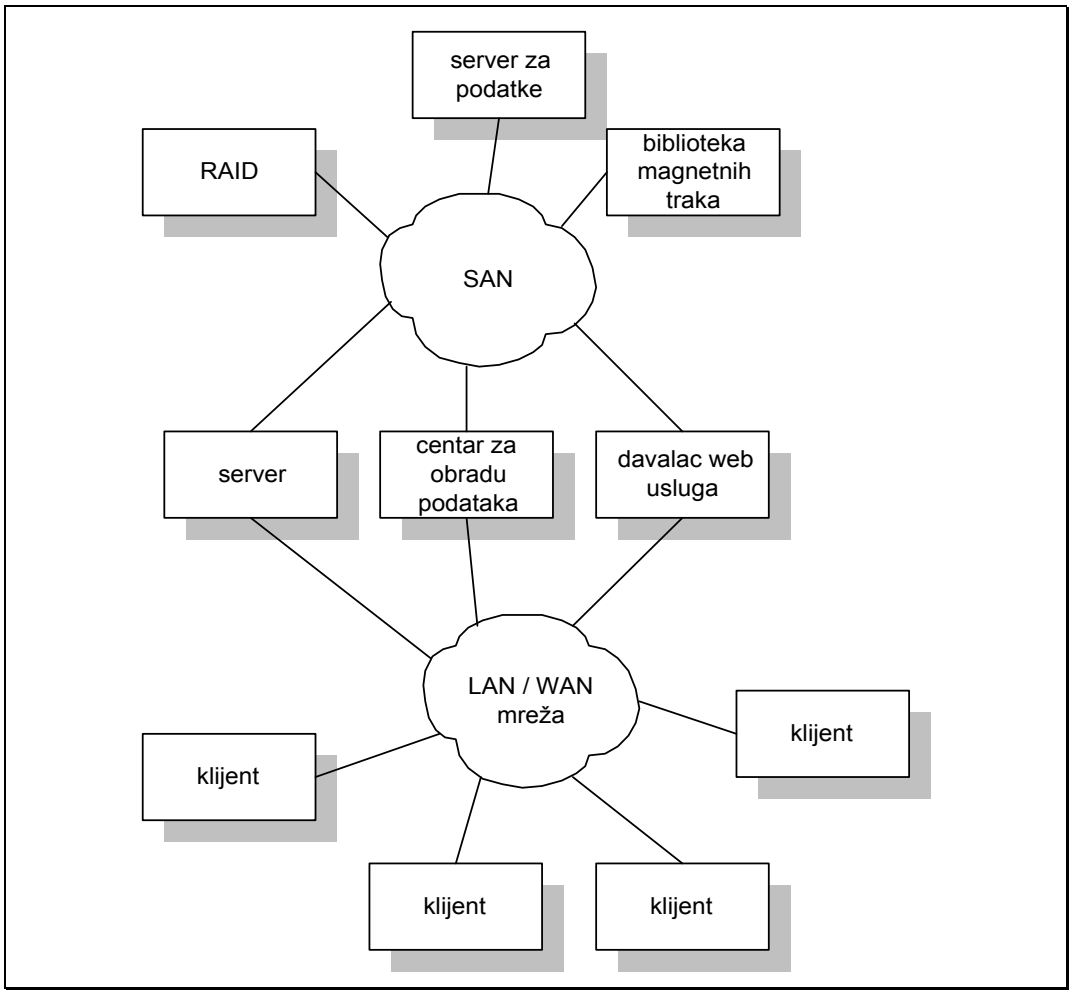
Slika 10.9. *NAS - Network Attached Storage*

Povezivanje diskova preko ulazno-izlaznog porta (*host-attached storage*) izvodi se u različitim tehnologijama. U ove tehnologije spada ATA, koja podržava dva diska po kanalu kontrolera u *master-slave* relaciji, SCSI (*small computer system interface*) i FC (*fibre chanell*). SCSI port je realizovan kao magistrala na koju se može priključiti

15 različitih uređaja po kontroleru. FC predstavlja serijsku arhitekturu visokih performansi, baziranu ili na optičkom kablu ili na četvorožilčnom kablu. Postoje dve varijante za FC tehnologije:

- SAN (*storage area network*), koji omogućava priključivanje velikog broja računara i diskova; SAN ima 24-bitni adresni prostor i radi kao prebacivač i
- arbitrirana petlja (*arbitred loop*, FC-AL) na koju se može priključiti 12 uređaja.

SAN se realizuje kao privatna mreža između servera i diskova priključenih na mrežu. U ovom konceptu, klijenti ne napadaju direktno diskove po mreži, već se za usluge preko uobičajenih mreža (LAN/WAN) obraćaju serveru SAN sistema. Konfiguracija je krajnje fleksibilna i optimalna.



Slika 10.10. *SAN - Storage Area Network*

10.7. Upravljanje swap prostorom

Upravljanje *swap* prostorom i efikasno upravljanje virtuelnom memorijom spadaju u značajnije zadatake operativnog sistema. Način korišćenja *swap* prostora zavisi od samog operativnog sistema, a prvenstveno zavisi od toga kako se realizuje tehnika upravljanja virtuelnom memorijom, odnosno da li se zamenjuju celi procesi (*swaping systems*) ili samo fiksni delovi procesa, odnosno stranice (*paging systems*). Veličina *swap* prostora zavisi od veličine fizičke memorije, od veličine virtuelne memoriji i od načina funkcionisanja virtuelne memorije. *Swap* prostor može biri realizovan na dva načina:

[1] *Swap* kao datoteka u postojećem sistemu datoteka

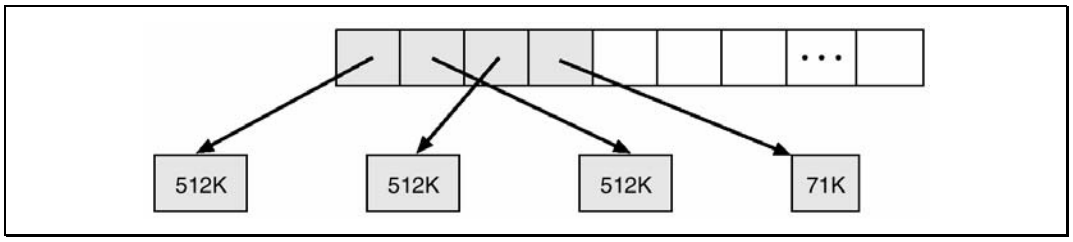
U ovom slučaju *swap* prostor je velika datoteka o kojoj, što se tiče kreiranja i dodele prostora, kompletnu brigu vodi sistem datoteka. Program koji upravlja *swap* prostorom jedino treba da kontroliše unutrašnji sadržaj datoteke. Ova šema je jednostavna za implementaciju i nudi veću fleksibilnost administracije - veličina *swap* datoteke se lako može povećati, što u slučaju *swap* particije nije tako jednostavno. Međutim, ovaj način realizacije *swap* prostora nije efikasan jer se za upravljanje *swap* datotekom koriste rutine sistema datoteka. Takođe, postoji mogućnost fragmentacije *swap* datoteke. Keširanje u ovom slučaju može dosta povećati performanse.

[2] *Swap* kao posebna particija

U ovom slučaju *swap* prostor je posebna particija na disku pa samim tim nema nikakve veze sa sistemom datoteka i može se organizovati na poseban način. Program koju upravlja *swap* prostorom, vodi kompletan posao oko dodele, pronalaženja i oslobađanja delova *swap* prostora, što je svakako optimalnije nego u prethodnom slučaju.

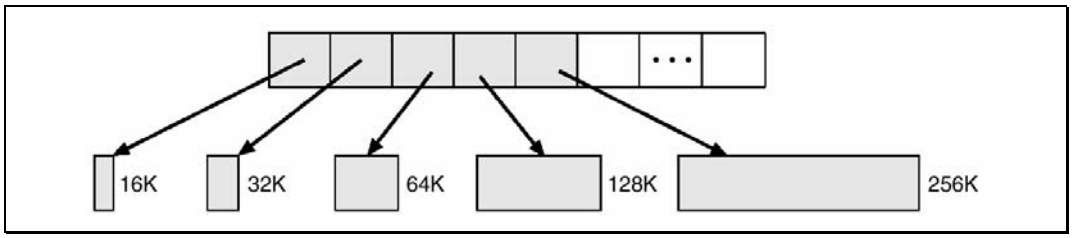
Neki sistemi (Linux, Solaris 2) dozvoljavaju korišćenje obe metode.

Na primeru BSD UNIX 4.3 operativnog sistema prikazaćemo jedan primer upravljanje *swap* prostorom. Svaki proces nakon pokretanja dobija svoj prostor u *swap* području. Već je rečeno da se UNIX proces sastoji od tri sekcije - tekst segmenta, segmenta podataka i stek segmenta. BSD UNIX 4.3 poseduje dupli mehanizam za upravljanje *swap* prostorom. Posebno se upravlja tekst segmentima, a posebno segmentima podataka. Kod upravljana *swap* prostorom za tekst segmente, mapa sadrži blokove fiksne veličine. Tekst segment je fiksne veličine u vremenu, pa se *swap* prostor za tekst segment dodeljuje u većim fiksnim blokovima od 512KB, osim za poslednji elemenat tekst segmenta, koji se alocira u rezoluciji od 1K, kao što je prikazano na slici 10.11.



Slika 10.11. *BSD UNIX 4.3 Swap za tekst segment*

Mapa za segmente podataka je komplikovanija, zato što segment podataka može da menja svoju veličinu u vremenu. Mapa sadrži blokove promeljive veličine u stepenima broja 2. Za svaki indeks i, blok na koju ukazuje ulaz i u mapi je veličine 2^i x16KB, a maksimalna veličina je 2MB, kao što je prikazano na slici 10.12. Kada proces poveća svoj segment podataka, swap sistem mu dodeli sledeći blok koji duplo veći. Po ovoj šemi, mali procesi zauzimaju male blokove, što smanjuje fragmentaciju. Blokovi velikih procesa se nalaze brzo zato što je mapa ima mali broj ulaza.



Slika 10.12. *BSD UNIX 4.3 Swap za segment podataka*

10.8. Tercijalna memorija

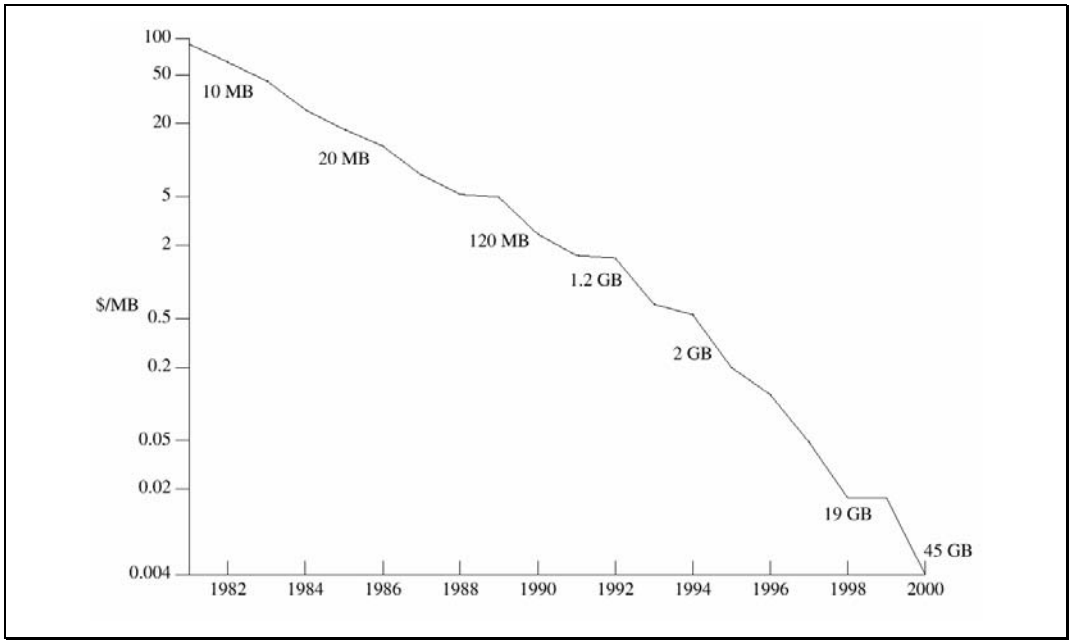
Osnovne karakteristike tercijalne memoriju se niska cena, velki kapaciteti i izmenjivost medijuma. Koriste se po pravilu za arhiviranje podataka u formi arhive ili sistema datoteka. Za rad sa tercijalnim memorijama koriste se dodatne operacije i sistemski pozivi koji nisu karakteristični za diskove.

U tercijalne memorije spadaju sledeći uređaji:

- diskete, odnosni magnetni diskovi kapciteta reda veličine 1MB,
- magneto-optički diskovi, koji imaju površinu od magneta na koju se upisuje i sa koje se čita pomoću laserskih zraka,
- optički diskovi, koji imaju površinu od nemagnetnog optičkog materijala na koju se takođe upisuje pomoću laserskih zraka,

- WORM uređaji (*write once read many*) - podaci se upisuju jednom, a čitaju više puta (CD-R, DVD-R) i
- trake, odnosno magnetni uređaji solidnog kapaciteta sa sekvencijalnim pristupom, koji su jeftiniji, ali i osetno sporiji od diskova.

Performanse tercijalne memorije određuju se na osnovu brzine transfera i vremena pristupa. Izmenljivi diskovi imaju brzine transfera od 0.25MB/sec do nekoliko MB/sec, dok se brzine traka nalaze u opsegu od 0.25MB/sec do 30MB/sec. Vreme pristupa diskova je uglavnom uniformno, dok sekvencijalna priroda trake unosi raznolikost u vremenu pristupa koje zavisi od tekuće i od željene pozicije. Po pitanju pouzdanosti, optički urađaji su u prednosti u odnosu na magnetne. Što se tiče cena sekundarne i tercijalne memorije, zapaža se trend velikog rasta kapaciteta i osetnog pada cena. Na slici 10.13. prikazan je grafikon pada cena diskova u jednom periodu.



Slika 10.13. *Pad cena diskova u periodu 1982-2000*

Flopi diskovi

Disketni uređaj (*Floppy Disk Drive*) je uređaj koji koristi izmenjljive medijume (diskete) za skladištenje podataka. Disketa se sastoji se od fleksibilne membrane koja je obostrano premazana tankim slojem magnetne supstance i odgovara jednoj rotacionoj ploči hard diska. Glave za čitanje i pisanje se nalaze u disketnom uređaju. Ovim je

omogućeno sledeće: jedan uređaj može da pristupi različitim medijumima (disketama), a disketi snimljenoj na jednom uređaju može se pristupiti pomoću drugog uređaja.

Diskete se dele na staze i sektore poput hard diskova, pri čemu dve ekvidistantne staze na različitim stranama diskete čine jedan cilindar. Broj staza i sektora je znatno manji nego na hard diskovima. Kontroler za disketne uređaje je integrisan na matičnoj ploči - pomoću njega se na sistem mogu vezati najviše dva uređaja (kojima se u DOS-u i Windows-u pristupa preko logičkih diskova A: i B:). Dve vrste disketnih uređaja predstavljaju standard za PC računare:

- 5.25 inča, kapaciteta 360KB sa dvostrukom gustinom zapisa (*double density*) i 1.2MB sa visokom gustinom zapisa (*high density*). Ovi uređaji se više ne upotrebljavaju
- 3.5 inča, kapaciteta 720KB (*double density*), 1.44MB (*high density*) i 2.88MB (*extended density*).

Pri tome se više različitih formata disketa može pročitati na istom uređaju - 3.5 inčni drajv kapaciteta 1.44MB omogućava pristup disketama 720KB i 1.44MB.

CD-ROM uređaji

CD-ROM uređaj koristi izmenljive medijume u vidu tankih diskova s visokim stepenom refleksije. Podaci se upisuju na površinu diska u vidu malih rezova, poređanih u spiralu s početkom u centru. Prilikom čitanja podataka, CD-ROM uređaj upućuje laserski zrak duž spirale, koji se može reflektovati na dva načina, zavisno od površine s koje se refleksija vrši (glatka površina ili rez). Ovim je omogućeno jednostavno kodiranje bitova, a samim tim i podataka. Ovo je osnovni princip - postupak se svodi na preciznu optiku i elektromehaniku i daleko je komplikovaniji.

U poređenju sa hard diskovima, CD-ROM uređaji pripadaju klasi sporih uređaja. Srednje vreme pristupa podacima (*average seek time*) standardnog IDE diska ne prelazi 15 milisekundi, što je nekoliko desetina puta manje u odnosu na CD-ROM uređaje. CD-ROM uređaj nije prijatno koristiti kao "live" sistem datoteka ukoliko se na njemu nalazi veća količina malih datoteka raspoređenih po direktorijumima. CD-ROM je jako koristan uređaj za korišćenje prilikom instalacije softvera - kapacitet medijuma je relativno visok (650-700MB), a brzina nije presudna za instalaciju.

Postoji nekoliko sistema datoteka koji se mogu koristiti za čuvanje podataka na CD-ROM medijumima. Najčešće je korišćen minimalni sistem datoteka specificiran internacionalnim standardom ISO 9660. Većina operativnih sistema prepoznaje ovaj sistem datoteka, a podaci upisani na CD-ROM medijum softverom za narezivanje u jednom operativnom sistemu su prenosivi na druge operativne sisteme i druge računare. Na primer, slike u JPEG formatu ili Video CD narezani pod Linux-om su vidljivi pod Windows operativnim sistemom, ali se Windows izvršni programi ne mogu pokrenuti pod Linux-om bez emulatora.

Magnetne trake

Uređaj za snimanje na magnetne trake (*tape drive, streamer*) koristi trake, slične audio kasetama. Traka je po prirodi sekvencijalni medijum - stroga sekvencijalnost čini traku sporom i krajnje neprijatnom za rad sa sistemima datoteka. S druge strane, trake su jeftine za proizvodnju, odnosno kupcu se mogu isporučiti trake velike dužine, a samim tim i relativno velikog kapaciteta (2GB-20GB i veće) po pristupačnim cenama. Veliki kapacitet ih čini pogodnim za backup i arhiviranje, a to su postupci koji ne zahtevaju brze uređaje i direktan pristup medijumu.

11

Sistemi datoteka

Sadržaj poglavlja:

11.1. Pojam datoteke
11.2. Pojam direktorijuma
11.3. Linkovi (reference)
11.4. Zaštita i deljenje datoteka
11.5. Sistemi datoteka
11.6. Dodela prostora za datoteke
11.7. Upravljanje slobodnim prostorom
11.8. Pouzdanost sistema datoteka
11.9. Efikasnost i performanse sistema datoteka
11.10. Značajniji sistemi datoteka (UNIX/Linux)
11.11. Značajniji sistemi datoteka (DOS/Windows)

11.1. Pojam datoteke

Kao što je već rečeno, potreba za deljenjem programa i podataka implicira potrebu za trajnim skladištenjem podataka sa mogućnošću brzog pristupa. To omogućavaju masovni memorijski uređaji (sekundarne memorije), bazirani uglavnom na magnetnim uređajima. Podaci se na sekundarnim memorijama mogu organizovati korišćenjem različitih mehanizama. Sistem datoteka (*filesystem*) radi sa nestrukturiranim celinama i te celine ne interpretira. Sistem za upravljanje podacima (*data management system*) radi sa strukturiranim celinama, ali ih ne interpretira. Sistem

za upravljanje bazama podataka (*database management system*) radi sa strukturiranim celinama koje interpretira.

Za većinu korisnika sistem datoteka je nevidljiviji aspekt operativnog sistema koji obezbeđuje mehanizam za čuvanje i pristup datotekama i programima koji pripadaju operativnom sistemu ili korisnicima. Jednostavno rečeno, sistem datoteka se sastoji od dva dela: kolekcije datoteka i kolekcije direktorijuma, odnosno kataloga koji obezbeđuju informacije o datotekama. Takođe, uvodi se i pojam particije, odnosno strukture koja omogućava deljenje diskova na više sisetma datoteka.

Datoteka (*file*) je za korisnika kolekcija povezanih informacija, odnosno logička celina sa značenjem, a za operativni sistem objekat koji se čuva na sekundarnoj memoriji. Definisaćemo datoteku kao organizovani skup podataka koji ima svoje ime i koji se prema određenom prostornom rasporedu čuva na sekundarnoj memoriji.

Datoteku osim sadržaja i imena opisuju i dodatni atributi, koji se čuvaju u kontrolnom bloku datoteke. Kontrolni blok datoteke je najčešće direktorijumska struktura, ali se implementirati i kao zasebna tabela (na primer, indeksni čvor u *second extended* sistemu datoteka). U značajnije atribute datoteka spadaju:

- tip datoteke, pomoću koga operativni sistem preliminarno određuje vrstu datoteke i po potrebi je povezuje sa nekom aplikacijom,
- opis prostornog rasporeda blokova koji čine datoteku ili ukazivač na lokaciju prvog bloka datoteke,
- tekuća veličina datoteke,
- informacije o vlasništvu i pravima pristupa, odnosno atributi koji regulišu kontrolu pristupa datoteci (kao što su kontrola čitanja i modifikacije datoteke) i
- vreme i datum.

Definišimo operacije koje se mogu obaviti nad datotekama.

[1] Kreiranje datoteke

Prilikom kreiranja datoteke, najpre se dodeljuje prostor na disku za smeštaj nove datoteke, a zatim se infomacije o datoteci upisuju u direktorijum u koji će datoteka logički biti smeštena.

[2] Čitanje podataka iz datoteke u memorijski bafer

[3] Upis podataka u datoteku

[4] Pozicioniranje unutar datoteke (file seek)

Ova operacija predstavlja pozinioranje u okvirima datoteke za neki sledeći ciklus čitanja ili upisa. Realizuje se na osnovu informacija o primenjenoj alokacionoj tehnici i informacija o datoteci koje se čuvaju u direktorijumu ili nekoj drugoj meta-strukturi kojom je ta datoteka opisana. Pozicioniranje unutar datoteke ne

uključujue ulazno-izlazni ciklus, jer nema transfera na relaciji disk - memorijski bafer.

[5] Brisanje datoteke

Ova operacija ukljanja kontrolni blok datoteke iz direktorijuma u kome je datoteka logički smeštena i time oslobađa zauzeti prostor na sekundarnim memorijama. Sadržaj datoteke se najčešće fizički ne briše, tako da se uz pomoć specijalnog softvera može povratiti (*undelete*). Ovaj problem predstavlja potencijalnu slabu tačku i može se iskoristiti za nedozvoljenu eksploataciju sistema.

[6] Odsecanje datoteke (*truncate*):

Postojeća datoteka se smanjuje, čime se oslobađa prostor na disku. Svi ostali atributi datoteke se čuvaju.

Rad sa datotekama se može ubrzati uvođenjem operacija otvaranja (*open*) i zatvaranja (*close*) datoteka, koje uz pomoć odgovarajućih memorijskih struktura ubrzavaju pristup datotekama. Operativni sistem, po pravilu, čuva globalnu sistemsku tabelu otvorenih datoteka (*system wide open-file table*) i posebnu tabelu otvorenih datoteka za svaki proces (*per process open-file table*). Prilikom otvaranja datoteke, u direktorijumskoj strukturi se traži kontrolni blok datoteke, koji se posle toga upisuje u glavnu sistemsku tabelu. Dalji pristup datotekama se realizuje pomoću ovih tabela, a kontrolni blok datoteke se ažurira na disku po zatvaranju datoteke.

Logička struktura datoteke

U najjednostavnijem slučaju, datoteka nema svoju logičku strukturu i predstavljena je kao kolekcija reči, odnosno bajtova.

U jednostavnije logičke strukture spadaju strukture zapisa, pri čemu jedan zapis u datoteci može biti fiksne ili promenljive dužine. Primer zapisa je linija u tekstualnoj datoteci. Složenije strukture predstavljaju formatirani dokumenti. Datoteka sa logičkom strukturom se može simulirati umetanjem kontrolnih karaktera u datoteke bez logičke strukture (na primer, umetanjem *Line Feed* i *Carriage Return* kontrolnih karaktera u tekst dobijamo jednostavni tekstualni dokument).

Tipovi datoteka

Najjednostavnije, datoteke se mogu podeliti na izvršne datoteke (odnosno prevedene i povezane programe) i datoteke sa podacima koje mogu biti tekstualne i binarne.

Na većini operativnih sistema, posle kreiranja, datokeka osim imena dobija i tip. Pomoću tipa, operativni sistem može preliminarno da odredi vrstu datoteke i da je

poveže sa nekom aplikacijom. Tip datoteke se može realizovati produženjem (ekstenzijom) imena datoteke, što je slučaj za DOS/Windows operativne sisteme. U sledećoj tabeli navedene su neke od značajnijih ekstenzija koje se koriste na tim operativnim sistemima.

Ekstenzija	**vrsta datoteke**
exe, com	binarna izvršna datoteka
Obj	prevedeni izvorni kod
lib, dll	statičke i dinamičke biblioteke
c, cpp, asm	izvorni kod
Bat	izvršna datoteka (*batch* program)
Txt	tekstualna datoteka
doc, rtf	tekstualni dokument kreiran tekst procesorom (kao što su Word i Open Office Writer)
zip, rar, arj	arhive
jpeg, bmp, gif	rasterske slike
mpeg, wmv	multimedijalni sadržaji

Tip datoteke se alternativno može zadati nekim karakterističnim zapisom. Ovaj metod zadavanja tipa se primenjuje na UNIX/Linux sistemima - magični brojevi koji se nalaze na samom početku datoteke određuju njen tip. Iz komandnog interpretera UNIX ne prepoznaje ekstenziju (datoteka sa ekstenzijom txt bez problema može biti arhiva ili izvršna datoteka). Međutim, Linux u grafičkom okruženju preliminarno određuje vrstu datoteke na osnovu ekstenzije i otvara datoteku asociranom aplikacijom.

Metode pristupa datotekama

Opisaćemo tri različite metode pristupa datotekama.

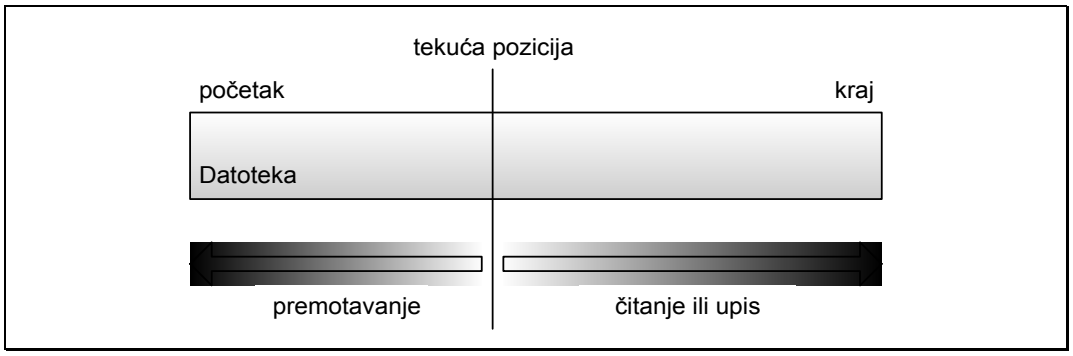
[1] Sekvencijalni metod pristupa

Informacije se prosleđuju u tačnom redosledu, jedna iza druge, uvek u odnosu na vrednost tekućeg ukazivača CP (*curent pointer*). Posle svakog pristupa datoteci, vrednost tekućeg ukazivača se ažurira. Sekvencijalni pristup zahteva da postoji mogućnost da se datoteka premota na početak, tako da vrednost tekućeg pokazivača bude nula (CP=0). Neki editori teksta i prevodioci koriste ovaj metod pristupa.

Znači, u operacije koje se mogu izvesti ukoliko je pristup datoteci sekvencijalan spadaju:

- čitanje sledećeg bloka,
- upis u sledeći blok i

- pozicioniranje na početak.



Slika 11.1. *Sekvencijalni metod pristupa datotekama*

[2] Direktni metod pristupa

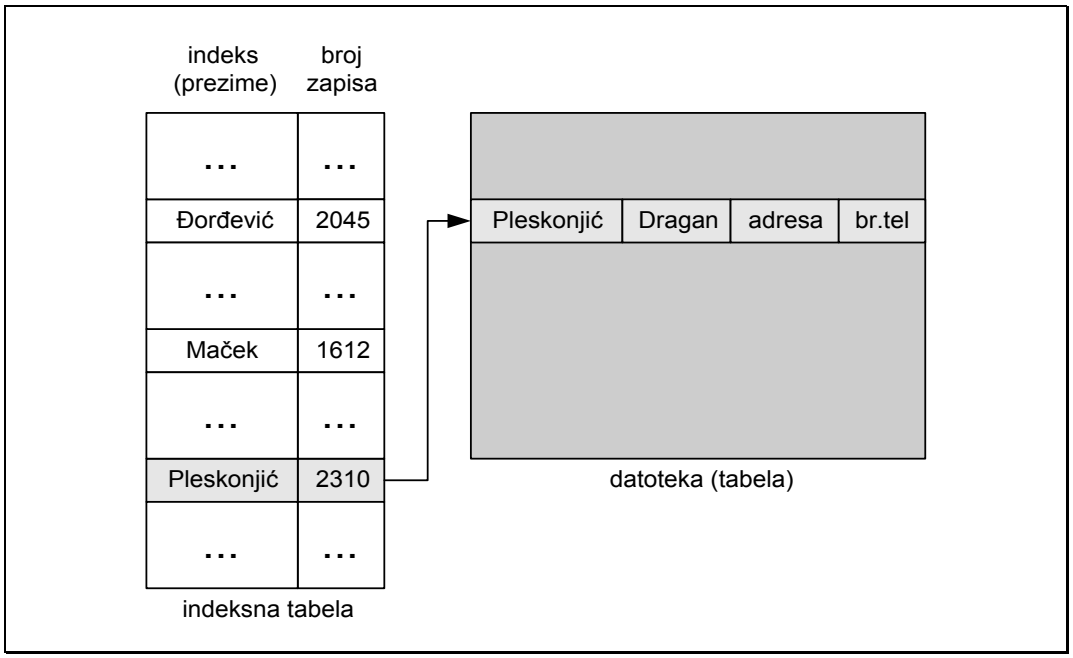
Omogućava pristup bilo kom delu datoteke, tako što se najpre odredi njegova pozicija na disku, a zatim pristupi podacima. Direktan pristup omogućava korisniku da pristupi krajnjem bloku datoteke bez čitanja prethodnog sadržaja. Pristup datotekama na disku je direktan, a pristup podacima na traci sekvencijalan.

U operacije koje se mogu izvesti ukoliko je pristup datoteci direktan spadaju:

- čitanje n-tog bloka,
- upis u n-ti blok,
- pozicioniranje na n-ti blok,
- čitanje sledećeg bloka i
- čitanje prethodnog bloka.

[3] Metod pristupa pomoću indeksnih datoteka

Ovaj metod pristupa se koristi za pristup odgovarajućem slogu u bazi podataka (slika 11.2). Svakoj datoteci (tabeli) pridružena je indeksna datoteka, uređena po nekom kriterijumu, pomoću koje se prilikom čitanja brzo može naći odgovarajući zapis. Prilikom upisa novog sloga u datoteku, ažurira se i indeksna datoteka.



Slika 11.2. *Pristup slogu pomoću indeksne datoteke*

11.2. Pojam direktorijuma

Na jednom disku se može smestiti veliki broj datoteka, tako da se one radi preglednosti i kontrole pristupa grupišu u posebne strukture koje se nazivaju direktorijumi. Direktorijum je struktura koja sadrži kontrolne blokove svih datoteka koje su u njemu logički smeštene. Svi direktorijumi sa svojim datotekama obrazuju sistem datoteka.

Operacije koje se mogu izvesti nad direktorijumima su:

[1] Prikazivanje sadržaja direktorijuma ("listanje" direktorijuma)

[2] Pretraživanje direktorijuma

Datoteka se u okviru tekućeg direktorijuma pronalazi tako što se upoređuju kontrolni blokovi datoteka sa zadatim imenom.

[3] Promena imena datoteke

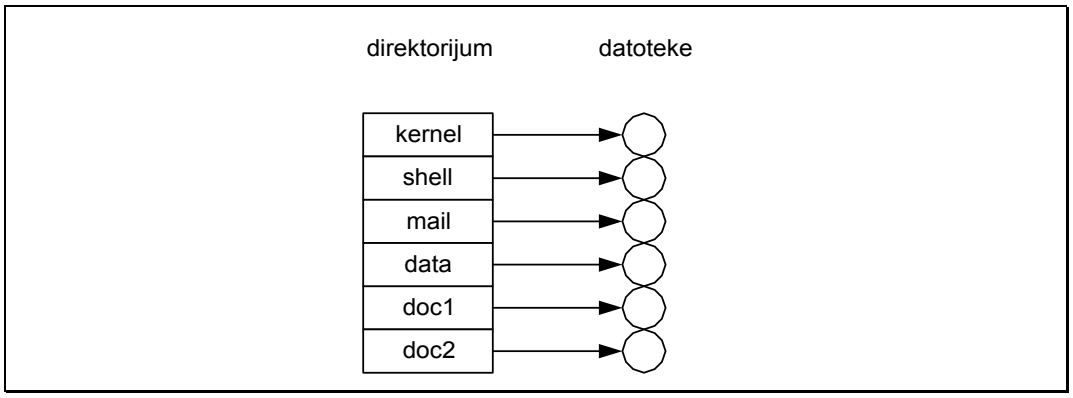
[4] Kreiranje i brisanje datoteka i poddirektorijuma u okviru tekućeg direktorijuma (navedene su u spisku operacija koje se mogu izvesti nad datotekama, ali ih zbog celovitisti navodimo i ovde)

Direktorijumi se u okviru sistema datoteka moraju implementirati tako da obezbede efikasno pronalaženje postojećih datoteka i prostora za nove, što se postiže upotrebom tehnikama kao što je heš i B+ stabla.

Logička struktura direktorijuma

Direktorijumska struktura implementirana u sistemu datotaka može biti jednonivovska, dvonivovska ili višenivovska (direktorijumsko stablo).

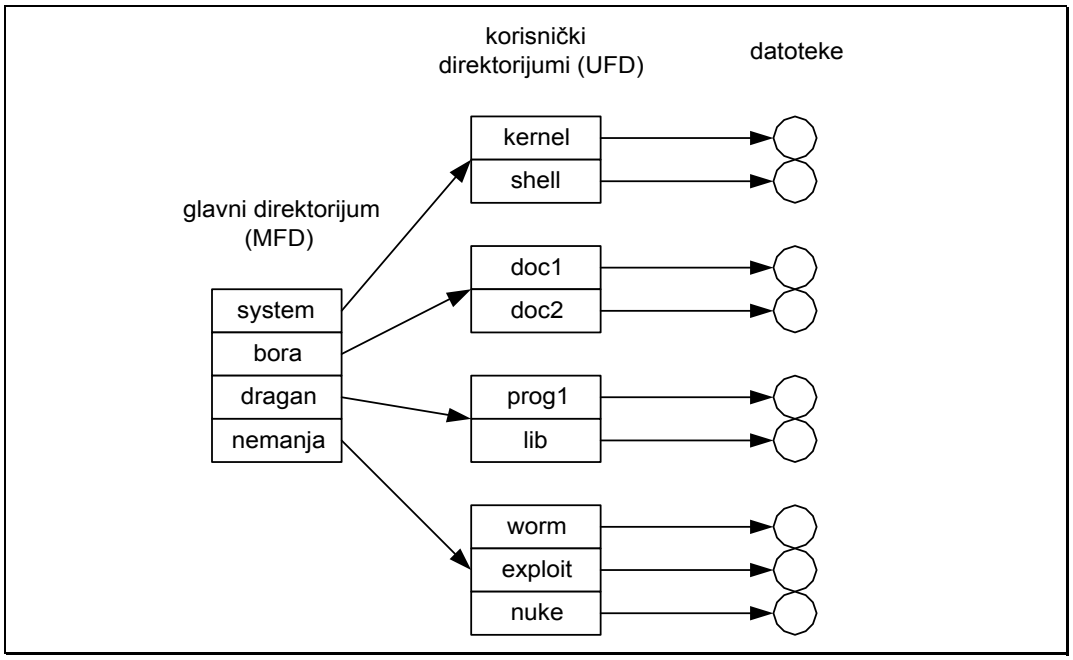
Jednonivovska struktura je najprostija - na sistemu datoteka postoji jedan direktorijum u kome se nalaze sve datoteke. Osnovni nedostaci ovakvog načina organizacije su nepreglednost i problem imenovanja datoteka koje pripadaju različitim korisnicima, jer sve datoteke u direktorijumu moraju da imaju različita imena.



Slika 11.3 *Jednonivovska struktura direktorijuma*

Problem imenovanja datoteka može se rešiti uvođenjem dvonivovske strukture direktorijuma, koja je prikazana na slici 11.4.

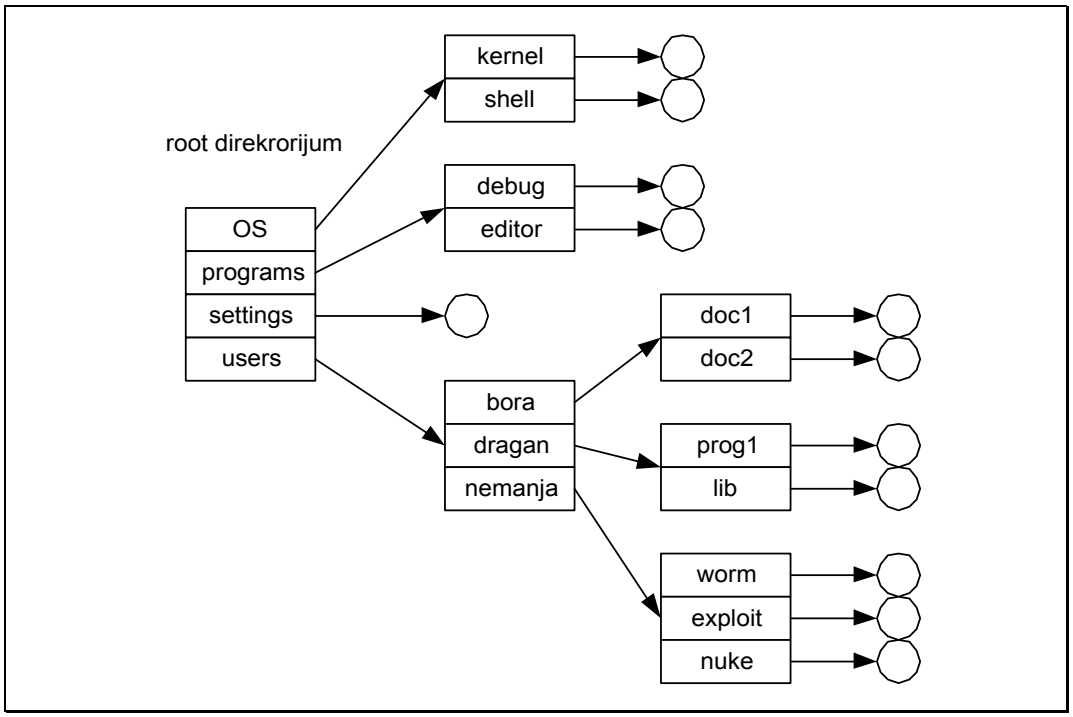
Dvonivovsku strukturu karakteriše skup direktorijuma podeljenih na dva nivoa. Na prvom nivou se nalazi glavni direktorijum (MFD - *master file directory*), a na drugom poseban diretorijum za svakog korisnika (UFD - *user file directory*). Ovakva struktura dozvoljava različitim korisnicima da kreiraju istoimene datoteke (dve datoteke sa istim imenom mogu se nalaziti na sistemu datoteka pod uslovom da su smeštene u različite direktorijume). Pretraživanje je brže, jer svaka datoteka ima svoju putanju (*path*), odnosno direktorijum u kome se nalazi. Međutim, ova struktura ne rešava problem grupisanja i imenovanja datoteka za pojedinačne korisnike.



Slika 11.4. *Dvonivovska struktura direktorijuma*

Nakvalitetniju direktorijumsku strukturu predstavlja stablo direktorijuma, prikazano na slici 11.5.

Direktorijum najvišeg nivoa je početni ili korenski direktorijum (*root*). U svakom direktorijumu se mogu kreirati datoteke i poddirektorijumi (subdirectory) koji predstavljaju grane tog stabla. Dubina stabla, odnosno broj nivoa u direktorijumskoj strukturi, određena je konkretnom implementacijom sistema datoteka, i u krajnjem slučaju može biti neograničena. Stablo rešava sve probleme grupisanja i imenovanja datoteka, odnosno logičke organizacije sa aspekta korisnika. Svaka datoteka ima svoje ime i putanju, koja može biti apsolutna, ako je izražena u odnosu na početni direktorijum ili relativna ako je izražena u odnosu na bilo koji drugi direktorijum osim početnog.



Slika 11.5 *Direktorijumsko stablo*

11.3. Linkovi (reference)

Datoteke se na većini sistema datoteka mogu linkovati, odnosno referencirati drugim imenima, sa iste ili različite pozicije u direktorijumskom stablu. Razlozi za korišćenje linkova su lakše pretraživanje i korišćenje datoteka i ušteda prostora na disku.

Neki sistemi datoteka podržavaju više vrsta linkova. Na UNIX sistemima se mogu kreirati dve vrste linkova na datoteke: hard link i simbolički link (*symbolic link*).

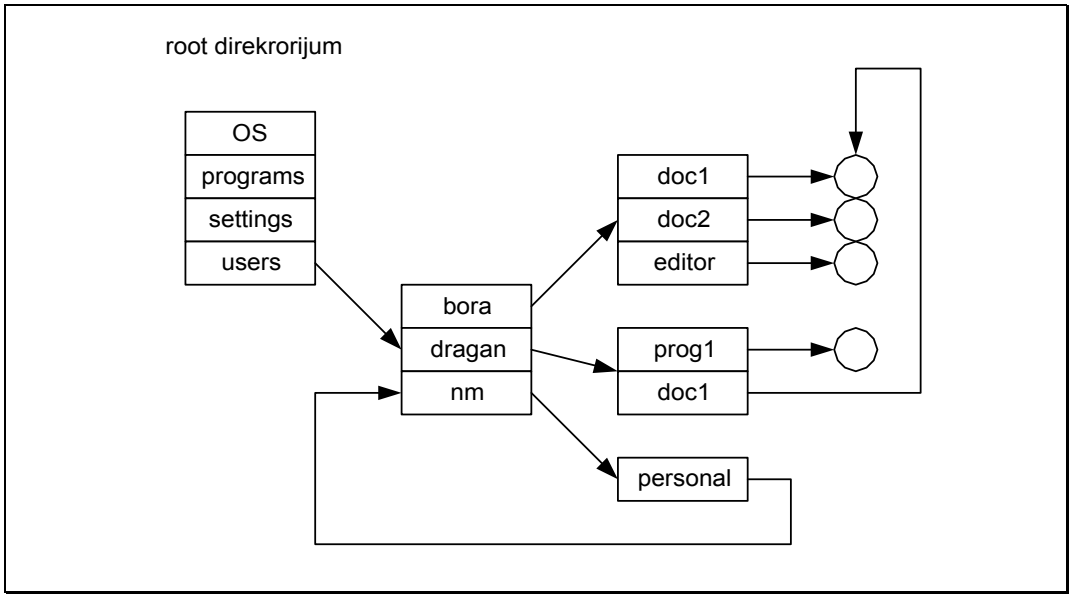
Kada korisnik pozove datoteku po imenu operativni sistem prevodi simboličko ime datoteke koje je naveo korisnik u interno ime, koje koristi operativni sistem. Zbog posebne interne reprezentacije, korisnici mogu datotekama dodeliti veći broj alternativnih imena. Hard link je jedno od tih imena. Link i original imaju isti i-node, tako da se moraju nalaziti na fizički istom sistemu datoteka. Ne mogu se linkovati datoteke sa mrežnog sistema datoteka (NFS), direktorijumi niti nepostojeće datoteke. Datoteka sa hard linkovima se ne može obrisati sa diska sve dok se ne uklone svi hard linkovi koji upućuju na tu datoteku.

Simbolički link je prečica ka objektu u sistemu datoteka, odnosno zaseban objekat sistema datoteka koji koristi jedan i-node i jedan blok podataka u kom je zapisana lokacija originalnog objekta. Kao takav, simbolički link zauzima određen prostor na disku i jedno mesto u i-node tabeli. Simbolički link pruža veću fleksibilnost jer dozvoljava referenciranje direktorijuma, nepostojećih datoteka i datoteka koje se nalaze na drugom sistemu datoteka (uključujući i NFS).

Na Windows sistemima hard linkovi ne postoje. Postoje prečice (*shortcuts*), veoma slične simboličkim linkovima na UNIX sistemu. Svaka prečica se realizuje kao zasebna datoteka sa ekstenzijom *lnk* i može se obrisati nezavisno od originala. Prečica može ukazivati na postojeće i nepostojeće datoteke i direktorijume na lokalnim ili mrežnim diskovima. Na NTFS 5 sistemima datoteka simbolički linkovi se realizuju putem *reparse point*a. Akcija usmerena ka *reparse point*u preusmerava se na definisani drajver. Simbolički link je specijalna vrsta *reparse point*a koji se upotrebljava da bi se izvršila redirekcija NTFS procesa na određenu datoteku ili direktorijum.

Linkovi na direktorijume i problem beskonačnih petlji

Problem koji se može javiti pri deljenju direktorijuma je pojava kružnih putanja u direktorijumskoj strukturi, odnosno beskonačnih petlji iz kojih formalno nema izlaska. Na primeru sa slike 11.6, korisnik pokušava da pročita sadržaj direktorijuma /users/nm/personal i ulazi u beskonačnu petlju.



Slika 11.6. *Pojava beskonačne petlje*

Linkovanje datoteka ne može izazvati ulazak u beskonačnu petlu, ali deljenje direktorijma može. Zato neki sistemi ne dozvoljavaju kreiranje linkova na direktorijime. U slučaju da su ovakvi linkovi dozvoljeni, opasnost od pojavljivanja krugova se uklanja uvođenjem algoritama koji pre svakog kreiranja linka na direktorijum kontrolišu da li se formira krug ili ne. Na osnovu toga se dozvoljava ili zabranjuje formiranje linka. Dodatno, uvodi se i algoritam koji prati brisanje datoteka iz linkovanih direktorijuma (*garbage collection*).

11.4. Deljenje i zaštita datoteka

Poseban kvalitet u sistemu datoteka predstavlja mogućnost deljenja datoteka i direktorijuma, što znači da različiti korisnici mogu različitim imenima referencirati isti prostor na disku. Na taj način se štedi prostor na disku, a sve promene koje jedan korisnik obavi su vidljive svima. Pri tome, svako ima utisak da radi sa svojom datotekom.

Deljenje datoteka se mora obavljati po strogo definisanim pravilima kojima su obuhvaćeni mehanizmi zaštite datoteka. Pored toga, mora se uspostaviti sinhronizacija višeprocesnog istovremenog obraćanja datoteci po algoritmu pisci-čitaoci. To znači da u jednom trenutku više procesa može da čita datoteku, ali samo jedan da vrši upis ili izmenu.

Jedan od problema koji karakteriše deljenje datoteka je problem konzistentnosti podataka. Ovaj problem se javlja u slučaju da veći broj procesa otvori istu datoteku, a zatim jedan proces upiše ili izmeni deo te datoteke. Problem je rešiv na više načina. Definisaćemo najpre pojam sesije (*file session*) kao skup svih akcija koje proces obavlja nad datotekom u intervalu od otvaranja do zatvaranja datoteke. Pretpostavimo da je više procesa otvorilo datoteku (odnosno da postoji više sesija) i da jedan ili više procesa želi da izvrši upis. Prema AFS semantici, promene koje izvrši jedan proces nisu odmah vidljive iz ostalih sesija. Kada se datoteka zatvori, izmene su vidljive samo iz novih sesija (odnosno sesije koje su otvorene nakon zatvaranja datoteke). Prema neizmenljivoj semantici (*immutable*), datoteka koja se proglasi za deljivu se ne može više modifikovati. Pristup datoteci je omogućen samo u režimu čitanja (read-only). UNIX je u pogledu deljenih datoteka krajnje fleksibilan - sve promene koje obavi jedan proces su trenutno vidljive u svim ostalim sesijama.

Deljenje datoteka po mreži

U distribuiranim uslovima, datoteke se dele po mreži. Direktorijum sa svojim kompletnim sadržajem je najmanja jedninica podataka koja se može deliti preko mreže. Na primer, Windows protokol za deljenje datoteka dozvoljava administratoru da proglasi direktorijum deljivim. UNIX obavlja deljenje datoteka preko NFS sistema

datoteka koji, takođe, može da deli direktorijum sa kompletnim sadržajem (svi poddirektorijumi i datoteke), ali ne i pojedinačne datoteke.

Zaštita datoteka

U višekorisničkom okruženju potrebno je uvesti kontrolu pristipa datotekama, odnosno specificirati ko može da obavi određene skupove operacija nad datotekom. Opšte rečeno, kontrola pristupa se obavlja u odnosnu na sledeće operacije: čitanje sadržaja datoteke, upis, izvršavanje, dodavanje sadržaja na kraj datoteke (*append*), brisanje datoteke, prikazivanje atributa datoteke.

Svaka datoteka treba da bude nezavisna u pogledu definisanja prava pristupa. Skup definicija prava pristupa za jednu datoteku naziva se tabela kontrole pristupa - ACL (*access controll list*), a svaki zapis u njoj određeje prava konkretnog korisnika u odnosu na tu datoteku. U tabeli kontrole pristupa treba regulisati prava za svakog korisnika koji ima potrebu da pristupi datoteci, tako da sama tabela može biti dosta velika. U tom kontekstu, tabele se kondenzuju tako što se korisnici grupišu u određene vlasničke kategorije ili grupe korisnika kojima se dodeljuju prava pristupa. Prava pristupa dodeljena grupi korisnika prenose se na sve korisnike koji toj grupi pripadaju.

Zaštita datoteka i direktorijuma na UNIX i Linux sistemima

UNIX u odnosu na datoteku definiše tri vlasničke kategorije:

[1] Vlasnik (*owner*)

Vlasnik korisnik koji je kreirao datoteku ili direktorijum, a u slučaju promene vlasničkih odnosa, korisnik kome je vlasništvo dodeljeno.

[2] Grupa (*group*)

Vlasnička kategorija grupa se odnosi na korisničku grupu kojoj je datoteka formalno priključena. Sistem administatori kreiraju grupe i definišu pripadnost korisnika grupama. Datoteke i direktorijumi mogu formalno pripadati samo jednoj grupi. Grupno pravo se primenjuje na sve korsnike koji pripadaju istoj grupi kojoj pripada i datoteka. Administrator sistema naknadno može promeniti pripadnost objekta grupi.

[3] Ostali (*others, public*)

Kategoriji ostatka sveta pripadaju svi korisnici koji nisu ni vlasnik objekta, niti pripadaju grupi kojoj objekat pripada.

Prava pristupa za svaku vlasničku kategoriju se eksplicitno dodeljuju svakom objektu prilikom kreiranja, a kasnije se mogu promeniti. Potpuni skup prava za svaku

vlasničku kategoriju čine pravo čitanja (r - *read*), pravo upisa (w - *write*) i pravo izvršavanja (x - *execute*).

Značenje pristupnih prava različito je za datoteke i direktorijume. U odnosu na datoteku, pristupna prava imaju sledeće značenje:

- read (r) - korisnik može pročitati sadržaj datoteke,
- write (w) - korisnik može modifikovati sadržaj datoteke i
- execute (x) - korisnik može izvršavati datoteku, pod uslovom da se radi o shell programu ili o datoteci u binarnom izvršnom formatu.

Pristupna prava u odnosu na direktorijum su malo složenija i imaju sledeće značenje:

- read (r) - korisnik može pročitati sadržaj direktorijuma,
- write (w) - korisnik može modifikovati sadržaj direktorijuma, odnosno dodavati nove datoteke i brisati postojeće, kreirati i brisati poddirektorijume i
- execute (x) - korisnik se može pozicionirati na direktorijum, prikazivati potpuni listing sadržaja i pretraživati direktorijum.

Interpretiraćemo pristupna prava dat1:

```
$ ls -l dat1

-rw-r--r--   1   u1   g1   509   Mar 10   17:21   dat1
```

Prvih deset karaktera predstavljaju tip datoteke i prava pristupa. Kao što je već pomenuto, prvi karakter ne spada u prava pristupa već određuje tip datoteke. U ovom slučaju to je (-), što znači da je reč o običnoj, regularnoj datoteci. Znak (-) na bilo kojoj drugoj poziciji, ukazuje na suspenziju prava. Sledeća tri karaktera određuju prava koja vlasnik ima nad datotekom. Vlasnik datoteke dat1 je korisnik u1, što se vidi iz trećeg polja. Prava korisnika u1 u odnosu na datoteku su rw-, što znači da je može čitati i modifikovati, ali je ne može izvršavati. Sledeći skup od tri karaktera definiše prava pristupa grupe. U slučaju datoteke dat1, koja pripada grupi g1, grupno pravo je r--, što znači da svi korisnici koji pripadaju grupi staff mogu da pročitaju datoteku, ali je ne mogu modifikovati ni izvršavati. Poslednji skup od tri karaktera predstavlja prava pristupa za sve korisnike koji nisu ni vlasnik niti pripadaju istoj grupi kojoj datoteka pripada. U ovom slučaju, pravo za ostatak sveta je r--, što znači da svaki korisnik koji pripada ovoj vlasničkoj kategoriji može čitati datoteku, ali je ne može modifikovati niti izvršavati.

Zaštita datoteka i direktorijumima na NTFS sistemu datoteka

Administrator Windows NT/2000/XP/2003 sistema kreira korisničke grupe i u njih učlanjuje korisnike i druge grupe. Svakom objektu, odnosno datoteci i direktorijumu (fascikla, odnosno *folder*, u Windows terminologiji) na NTFS sistemu datoteka se prilikom kreiranja dodeljuje grupa informacija koja se tiče kontrole pristupa (*Security Descriptor*). Security descriptor sadrži informacije o dozvolama za pristup resursu (NTFS dozvole) i o vlasniku resursa.

NTFS dozvole se dodeljuju ili oduzimaju korisnicima i grupama, pri čemu svaki korisnik ili grupa predstavlja jedan *Access Control Entry* (ACE), a skup svih dozvola tabelu *Access Contol List* (ACL). Grupe i korisnici su u ovoj tabeli predstavljeni numeričkim vrednostima SID (*Security Identifiers*) koje ih jednoznačno identifikuju. NTFS dozvolama se kontroliše pristup objektima isključivo na NTFS volumenima (NTFS dozvole se ne mogu dodeljivati objektima na FAT sistemu datoteka), a dozvole važe bez obzira da li korisnik pristupa resursu preko mreže ili lokalno. Prava pristupa data grupi prenose se na grupe i korisnike koji su u tu grupu učlanjeni.

```
C:\doc>cacls usersguide.txt

C:\doc\usersguide.txt   BUILTIN\Administrators:F
                        HACK\nmacek:F
                        HACK\bora:R
                        HACK\dragan:R
                        NT AUTHORITY\SYSTEM:F
                        BUILTIN\Users:C
```

Postoje dve vrste dozvola koje se mogu dodeliti objektima fajl sistema: standardne i specijalne. Svaka standardna dozvola (*High-Level Permission*) predstavlja sistemski predefinisan skup atomarnih dozvola. Atomarne dozvole omogućavaju korisniku da nad objektima fajl sistema izvrši određenu akciju. Standardne dozvole kao kombinacije atomskih dozvola omogućavaju korisniku da izvrši određeni skup akcija nad objektima sistema datoteka. U standardne dozvole spadaju:

[1] *Read*

Minimalna dozvola, koja omogućava korisniku da vidi sadržaj objekta.

[2] *Write*

Dozvola za upis novog sadržaja u objekat. Izmena sadržaja objekta je moguća ukoliko je korisniku pored ove dozvole data i dozvola *Read*, koja omogućava korisniku pregledanje sadržaja.

[3] *Read and Execute*

Ova dozvola predstavlja proširenje dozvole *Read* - korisniku se daje mogućnost zaobilaska restrikcija na roditeljskim granama i izvršavanja datoteke.

[4] *Modify*

Predstavlja uniju standardnih dozvola *Read and Execute* i *Write*, i atomarne dozvole za brisanje objekta.

[5] *Full Control*

Uključuje sve atomarne dozvole. Korisnik kome je data dozvola *Full Control* ima sva prava nad objektom - uključujući mogućnost dodele i oduzimanja NTFS dozvola i preuzimanja vlasništva.

[6] *List Folder Contents*

Dozvola koja se može dati samo nad direktorijumom. Ova dozvola omogućava korisniku da vidi sadržaj direktorijuma.

Za razliku od standardnih, specijalne dozvole predstavljaju korisnički definisan skup atomarnih dozvola koji se može dodeliti nekom objektu.

NTFS uključuje mogućnost nasleđivanja dozvola sa direktorijuma koji se u hijerarhiji stabla nalazi na višem nivou, odnosno propagacije dozvola na poddirektorijume i datoteke. U tom smislu, dozvole se mogu svrstati u eksplicitno dodeljene (dozvole koje su direktno priključene samom objektu) i nasleđene.

Svaki korisnik može biti član više grupa. Dozvole nad jednim objektom sistema datoteka mogu biti dodeljene većem broju grupa čiji je dati korisnik član. Efektivne dozvole korisnika formiraju se na sledeći način:

- različite dozvole dodeljene grupama kojima korisnik pripada se sabiraju,
- različite nasleđene i eksplicitno dodeljene dozvole se sabiraju i
- zabrana dozvole nadjačava dodelu.

11.5. Sistemi datoteka

Prilikom rada na računaru, datoteke koje čine sam operativni sistem, korisničke programe i radne verzije dokumenata čuvaju se na diskovima, zbog solidnih performasi kojima se diskovi karakterišu. Datoteke koje se ne koriste često i koje predstavljaju rezervnu kopiju podataka (*backup*) čuvaju se na izmenljivim memorijskim medijumima, kao što su CD i DVD medijumi i magnetne trake.

Diskovi se mogu podeliti na više delova. Delovi diska se nazivaju particije (*partitions*), parčići (*slices*), minidiskovi ili volumeni (*volumes*). Svaki deo diska predstavlja granice u čijim se okvirima može kreirati sistem datoteka. Operativni sistem podržava i drugi oblik formiranja sistema datoteka - kreiranje logičkog volumena od više diskova.

Da bi disk ili particija mogli da se iskoriste za skladištenje podataka, potrebno je na njima kreirati sisteme datoteka. Sistem datoteka je skup metoda i struktura podataka koje operativni sistem koristi za čuvanje datoteka. Sistem datoteka čine: zaglavlje (u kome se nalazi najmanje podataka, ali su ti podaci neophodni za funkcionisanje sistema datoteka), strukture za organizaciju podataka na medijumu (meta podaci) i sami podaci, odnosno datoteke i direktorijumi. Zaglavlje i meta-podaci čine premašenje sistema, ali bez njih sistem datoteka ne može da funkcioniše.

Realizacija sistema datoteka

Prilikom realizacije sistema datoteka, definiše se:

[1] Logička struktura sistema datoteka

Logička struktura je način predstavljanja sistema datateka korisniku. Pod ovim se podrazumeva definicija datoteka, direktorijuma, njihovih atributa i operacija dozvoljenim nad njima, odnosno sve što predstavlja logičku sliku sistema datoteka.

[2] Fizička struktura sistema datoteka

Fizičku strukturu sistema datoteka čine strukture podataka na disku koje služe za skladištenje podataka. U ove strukture spadaju, na primer, blokovi (UNIX, Linux) i klasteri - *clusters* (FAT sistem datoteka).

[3] Preslikavanje logičke strukture sistema datoteka u fizičku

Pod ovim preslikavanjem se podrazumeva uspostavljanje veze između sadržaja konkretne datoteke ili direktorijuma i struktura na disku. Na primer, preslikavanje određuje da se sadržaj datoteke mydoc.txt koja se nalazi na direktorijumu c:\1, nalazi u blokovima 15,16 i 25.

Po pravilu, sistem datoteka se realizuje u više nivoa, odnosno slojeva (*layers*), koji su prikazani na slici 11.7. Pri radu sa objektima sistema datoteka - na primer, sa datotekama, potrebno je preći put od imena datoteke do konkretnog fizičkog bloka na disku. Pri tome se prolazi kroz sledeće nivoe:

[1] Nivo logičkog sistema datoteka

Na najvišem nivou, kom pristupaju korisnički programi, nalazi se logički sistem datateka (*logical file system*). Na ovom nivou se upravlja meta podacima, odnosno svim strukturama podataka vezanim za datoteku osim njenog sardžaja. U meta podatke spadaju kontolni blok datoteke (FCB - *file control block*), položaj datoteke na disku i direktorijumska struktura. Za svaki sistem datoteka definiše se specifičan skup meta podataka. Na primer, u FAT sistemu datoteka meta podatke čine zaglavlje, dve FAT tabele (*file allocation table*) i koreni direktorijum, dok su na UNIX/Linux sistemima datoteka metapodaci realizovani kroz tabelu indeksnih čvorova i *superblock*.

[2] Nivo organizacije datoteka

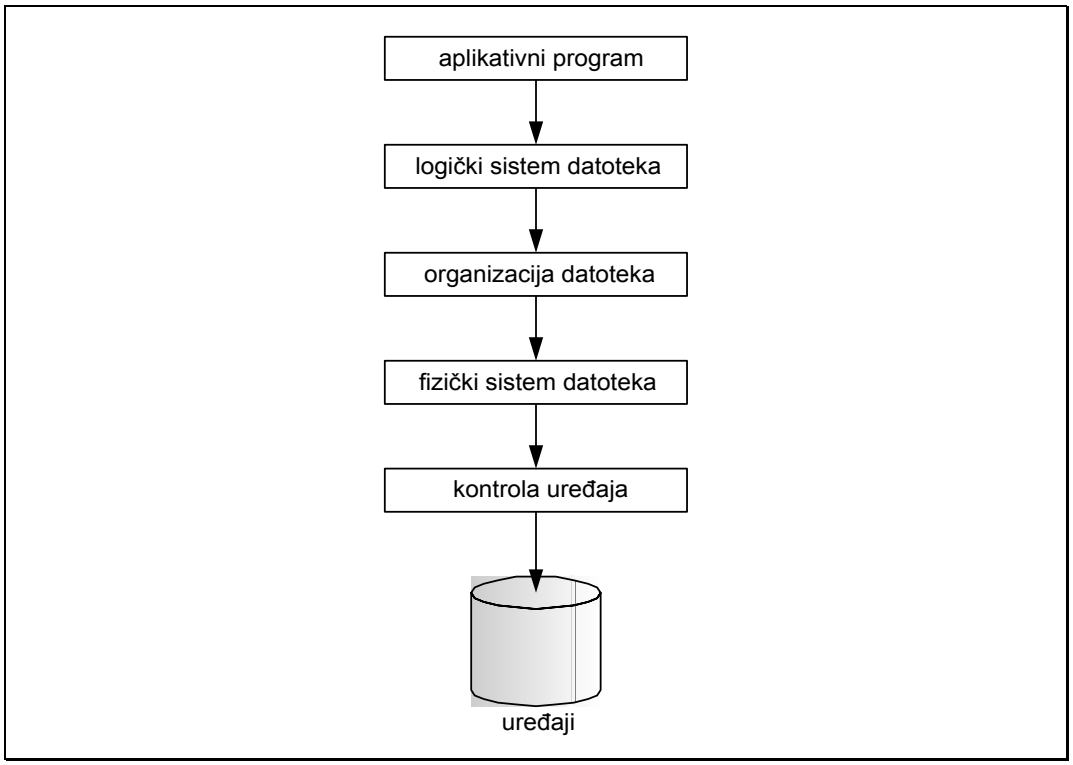
Na sledećem nivou nalazi se modul za organizaciju datoteka (*file organisation module*), koji vodi evidenciju o adresama logičkih blokova datoteka. Modul na osnovu zahteva za pristupom određenom delu datoteke određuje logičku adresu na kojoj se taj deo nalazi.

[3] Nivo fizičkog sistema datoteka

Sledeći nivo je osnovni, odnosno fizički sistem datoteka (*basic file system*), koji drajveru šalje osnovne komande za rad sa diskom. Na ovom nivou se logičke adrese trasformišu u fizičke, trodimenzionalne adrese.

[4] Nivo kontrole uređaja

Na najnižem nivou smeštena je kontrola uređaja, koju čine drajveri za uređaj i rutine za obradu prekida. Na ovom nivou podaci se prebacuju između diska i memorije. Adresiranje je fizičko - adrese su trodimenzionalne i definišu lokaciju podataka na konkretnom disku.



Slika 11.7. *Slojevita organizacija sistema datoteka*

Strukture podataka neophodne za realizaciju sistema datoteka

Kao što je već rečeno, da bi se realizovao sistem datoteka, potrebne su brojne strukture podataka na disku:

[1] BCB (*boot control block*)

BCB sadrži informacije koje se potrebne za otpočinjanje procesa podizanja jednog operativnog sitema. BCB je obično prvi blok sistema datotka

[2] Kontroni blok partcije (PCB, *partition control block*)

Kontrolni blok particije sadrži informacije o sistemu datoteka, kao što su veličina bloka, ukupni broj blokova, broj slobodnih blokova sa ukazivačem na listu slobodnih blokova, listu slobodnih kontrolnih blokova datoteka i ukazivač na tu listu.

[3] Kontrolne strukture za alokaciju datoteka

Ovim strukturama se određuje konkretan sadržaj datoteke, odnosno logički blokovi u kojima je smešten sadržaj. U ove strukture spadaju tabela indeksnih čvorova, odnosno *i-node* tabela (UNIX), FAT tabela (FAT) i MFT (NTFS).

[4] Direktorijumske strukture koj sadrže kontrolne blokove datoteka

[5] Kontrolni blokovi datoteka (FCB - *file control block*)

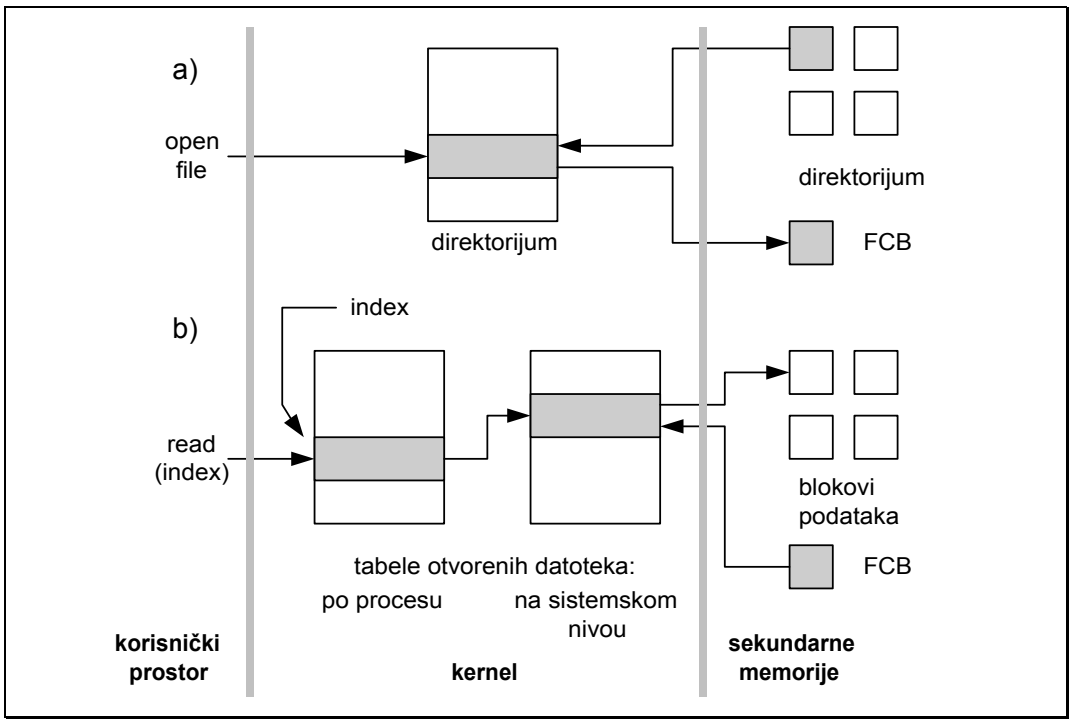
Kontrolni blok sadrži atribute datoteka i ukazivače na alokaciju datoteke. Na UNIX sistemima, kontrolni blok se sastoji iz dva dela: fileinfo strukture u direktorijumu, koja sadrži ime datoteke i indeksnog čvora u kome se nalaze svi atributi i prostorni raspored datoteke na disku. Na NTFS sistemu, svaka datoteka čuva svoj kontrolni blok u MFT tabeli. Tipičan FCB prikazan je na slici 11.8:

Slika 11.8. *Kontrolni blok datoteke*

Takođe, u operativnoj memoriji se definišu strukture koje ubrzavaju rad sistema datoteka. U memorijske strukture podataka spadaju:

- tabela otvorenih datoteka na sistemskom nivou (*sistem wide open-file table*), koja sadži kontrolne blokove za svaku otvorenu datoteku i
- tabela otvorenih datoteka po procesu (*per proces open-file table*), koja sadži ukazivač na glavnu tabelu i druge infomacije, kao što je tekući ukazivač na datoteku.

Prilikom otvaranja datoteke (slika 11.9.a), njen kontrolni blok se iz direktorijuma upisuje u glavnu memorijsku tabelu. Pri svakom pristupu datoteci (slika 11.9.b), iz tabele procesa se preko glavne tabele otvorenih blokova pronalaze blokovi datoteke na disku, a zatim se njima pristupa. Bitno je da pretraživanje blokova datoteke ide, uglavom, kroz memoriju, što osetno povećava performanse sistema.



Slika 11.9. *Memorijske strukture podataka*

Aktiviranje sistema datoteka

Svaki sistem datoteka pre korišćenja mora da se aktivira. Kod nekih operativnih sistema ovaj postupak je transparentan i ne zahteva učestvovanje korisnika. Drugi

operativni sistemi zahtevaju da korisnik sa odgovarajućim privilegijama eksplicitno odredi koji će sistemi datoteka biti automastski aktivirani, a koji ne, i koje sisteme datoteka ostali korisnici naknadno mogu aktivirati. Dve osnovne metode aktiviranja sistema datoteka su:

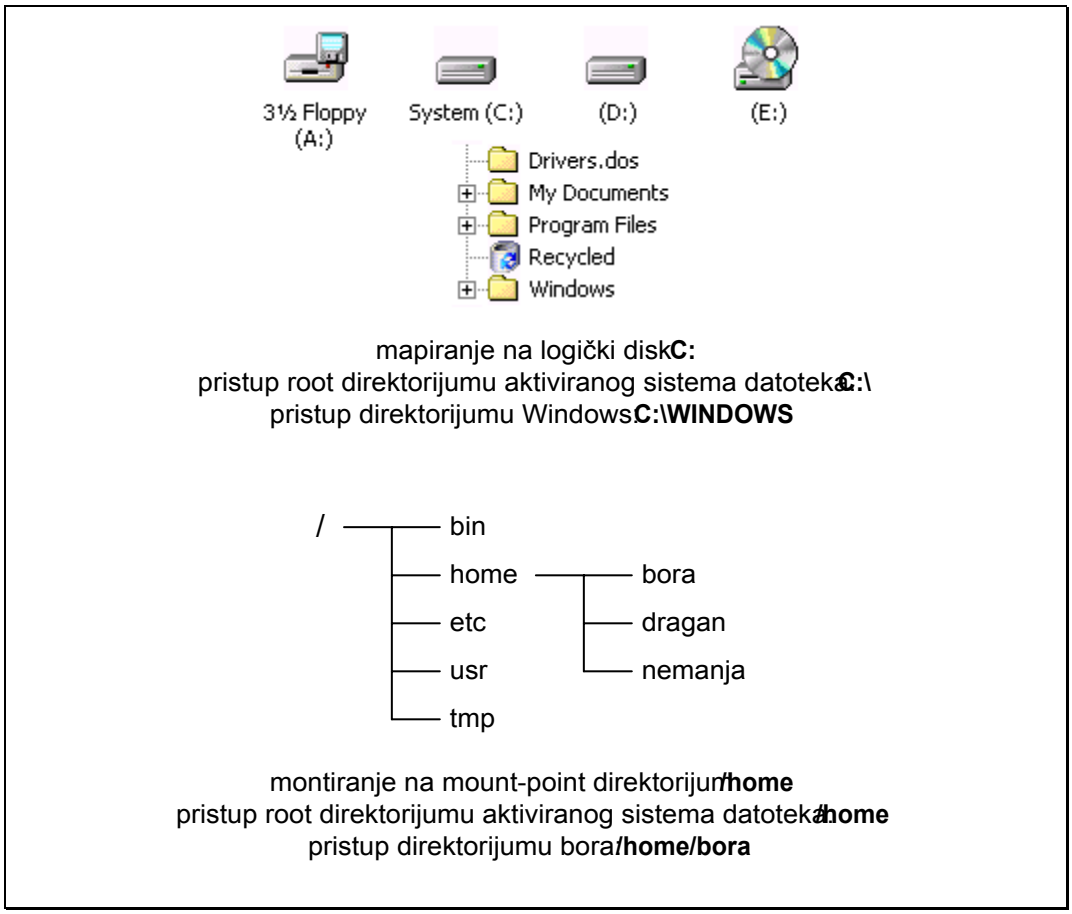
- mapiranje na logičke diskove, pri čemu se koreni direktorijum sistema datoteka poklapa sa korenim direktorijumom logičkog diska i
- montiranje na drugi direktorijum, pri čemu se koreni direktorijum sistema datoteka poklapa sa direktorijumom na koji je montiran.

Na primer, DOS i Windows 9.x automatski aktiviraju sve FAT sisteme datoteka i mapiraju ih na logičke diskove. Logički diskovi A: i B: su rezervisani za prvi i. drugi flopi disk, dok se svi ostali dodeljuju hard diskovima, CD-ROM uređajima i mrežnim diskovima, pri čemu je C: najčešće dodeljen prvoj particiji prvog hard diska. Aktivirani sistemi datoteka se nekim komandam mogu naknadno montirati na drugi direktorijum (komandom APPEND), ali se to ne preporučuje. Korisnici DOS i Windows 9.x operativnih sistema ne mogu deaktivirati sisteme datoteka. Takođe, NTFS sistemi datoteka nisu vidljivi iz DOS-a, osim ako se ne koriste specijalni programi za pristup NTFS sistemu iz operativnog sistema koji nije zasnovan na NT tehnologiji (na primer, NTFSDOS firme SysInternals).

Windows 2000/XP/2003 dozvoljava korisnicima da sisteme datoteka aktiviraju i deaktiviraju po želji, pod uslovom da im je dodeljeno pravo da to rade. Prilikom aktiviranja, korisnik sa administratorskim privilegijama može da mapira volumen na logički disk ili da ga montira na neki drugi direktorijum. Vidljivi su FAT i NTFS sistemi datoteka.

Na UNIX/Linux sistemima logički diskovi ne postoje - sistemi datoteka se montiraju (*mount*) na odgovarajuće *mount-point* direktorijume, i na taj način se formira aktivno UNIX stablo. Korisnici dalje pristupaju tim sistemima datoteka navigacijom po stablu i pozicioniranjem na *mount-point* direktorijume. Aktiviranje i deaktiviranje sistema datoteka po pravilu obavlja *root* korisnik.

Na slici 11.10. prikazani su postupci mapiranja na logičke diskove i montiranja na *mount-point* direktorijume.



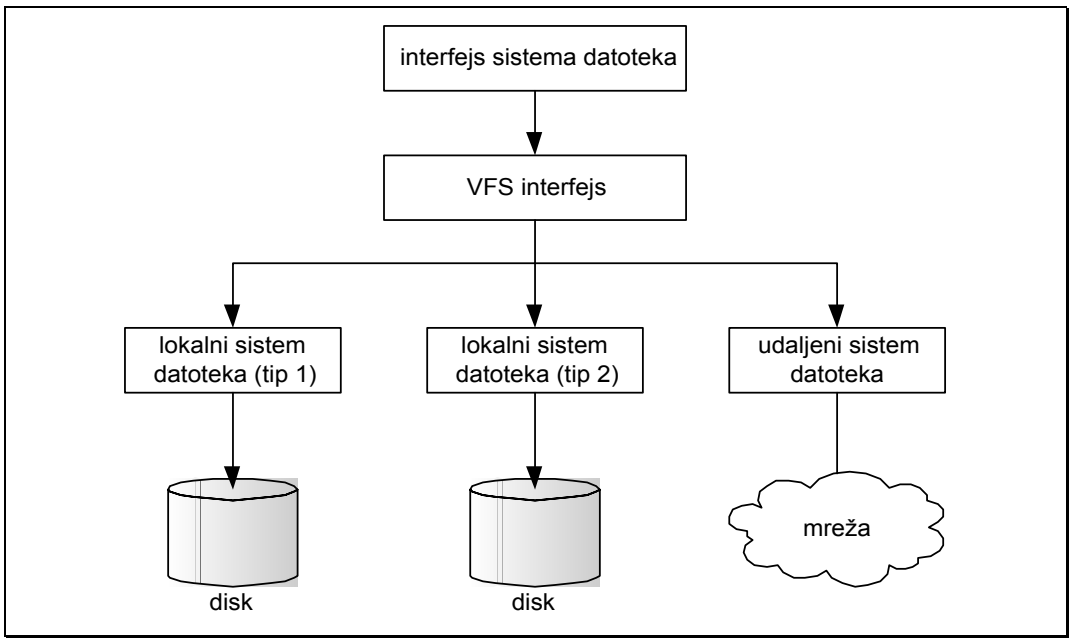
Slika 11.10. *Aktiviranje sistema datoteka*

Virtuelni sistem datoteka

Većina operativnih sistem podržava rad sa različitim sistemima datoteka. Na primer, Linux podržava rad sa velikim brojem Linux-specifičnih sistema datoteka (minix, ext, ext2, ext3, reiserFS, JFS, XFS) i nekim sistemima datoteka koji pripadaju drugim operativnim sistemima (FAT, FAT32, NTFS, HPFS).

Iz razloga mnogovrsnosti, uvodi se novi logički koncept - virtuelni sistem datoteka (VFS - *virtual filesystem*). VFS je objektno-orijentisani način realizacije sistema datoteka, koji omogućava korisniku da na isti način pristupa svim datotekama, bez obzira kom sistemu datoteka pripadaju. Korisnik se, putem sistemskih poziva, odnosno API-ja, obraća virtuelnom sistemu datoteka, a VFS obrađuje zahrev i upućuje poziv za rutine odgovarajućeg sistema datoteka. Na isti način, VFS omogućava

formiranje jednog logičkog sistema datoteka od više fizičkih diskova. VFS je ilustrovan slikom 11.11.



Slika 11.11. *Virtuelni sistem datoteka*

Realizacija direktorijuma

Direktorijumi sadrže kontrolne blokove dataoteka i odgovorni su za brzinu pretraživanja, čime direktno utuču na performanse. Postoje dve osnovne šeme za realizaciju direktorijuma:

[1] Linearna lista.

Ova šema se karakteriše smeštanjem novih kontrolnih blokova u redosledu koji je identičan redosledu kreiranja datoteka. Šema je jednostavna, ali pretraživanje traje dugo ukoliko se u direktorijumu nalazi veliki broj datoteka i poddirektorijuma. Pretraživanje se ubrzava uvođenjem raznih tehnika, kao što su balansirana staba (*B-trees*).

[2] Linearna lista sa hash tabelom.

Ova šema funkioniše kao linearna lista, ali se za upravljanje direktiorijumom korsiti hash tabela, koja osetno ubrzava pretraživanje. Heš kolizija se lako razrešava i kao takva nije opasna. Glavni problem predstavlja to što je hash

tabela fiksne veličine, tako da je broj datoteka u direktorijumu ograničen brojem ulaza u heš tabelu.

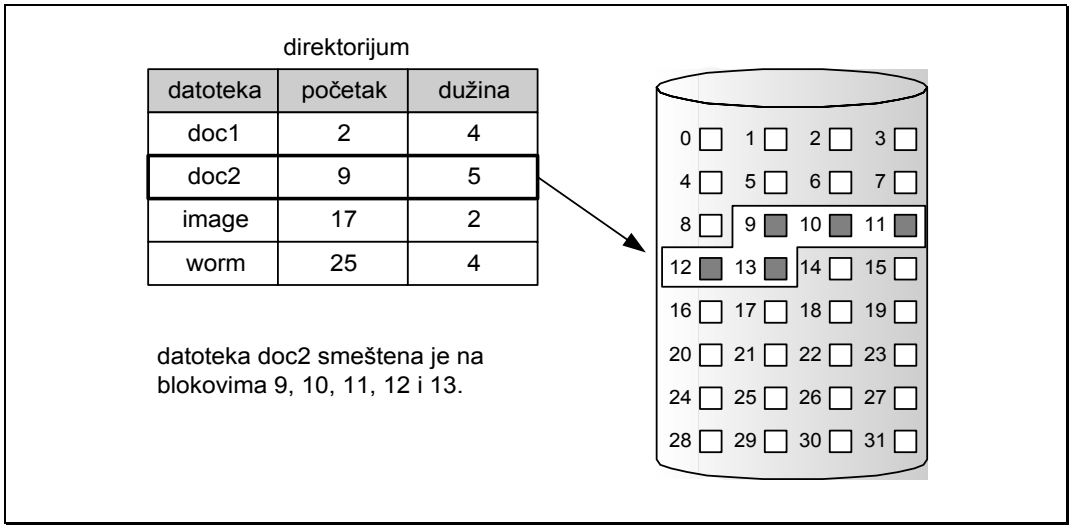
11.6. Dodela prostora za datoteke

Blokovi diska se datotekama mogu dodeliti na tri načina, odnosno korišćenjem sledećih metoda alokacije:

- dodela kontinualnog prostora,
- vezivanje blokova,
- mapa datoteka i
- metoda indeksnih blokova.

Dodela kontinualnog prostora

Svaka datoteka zauzima kontinualan prostor na disku - datotekama se dodeljuju uzastopni blokovi, kao što je prikazano na slici 10.12.



Slika 11.12. *Dodela kontinualnog prostora*

Ova metoda je prosta je za realizaciju. U kontrolnom bloku datoteke dovoljno je navesti adresu početnog bloka datoteke i ukupan broj dodeljenih blokova. Pristup bilo kom delu datoteke je direktan. Metoda dodele kontinualnog prostora unosi veoma malo kašnjenje jer se sadržaj datoteke čita odmah nakon čitanja njenog kontrolnog bloka.

Takođe, premašenje u pogledu utroška prostora na disku za meta strukture je zanemarljivo malo, jer se informacije koje opisuju datoteku (adresa početnog bloka i ukupan broj blokova) nalaze u kontrolnom bloku datoteke.

Metoda dodele kontinualnog prostora podseća na tehniku multiprogramiranja sa particijama promenljive veličine (MVT). U tom kontekstu, prisutan je problem eksterne fragmetacije koja se može regulisati kompakcijom prostora na disku (odnosno defragmentacijom).

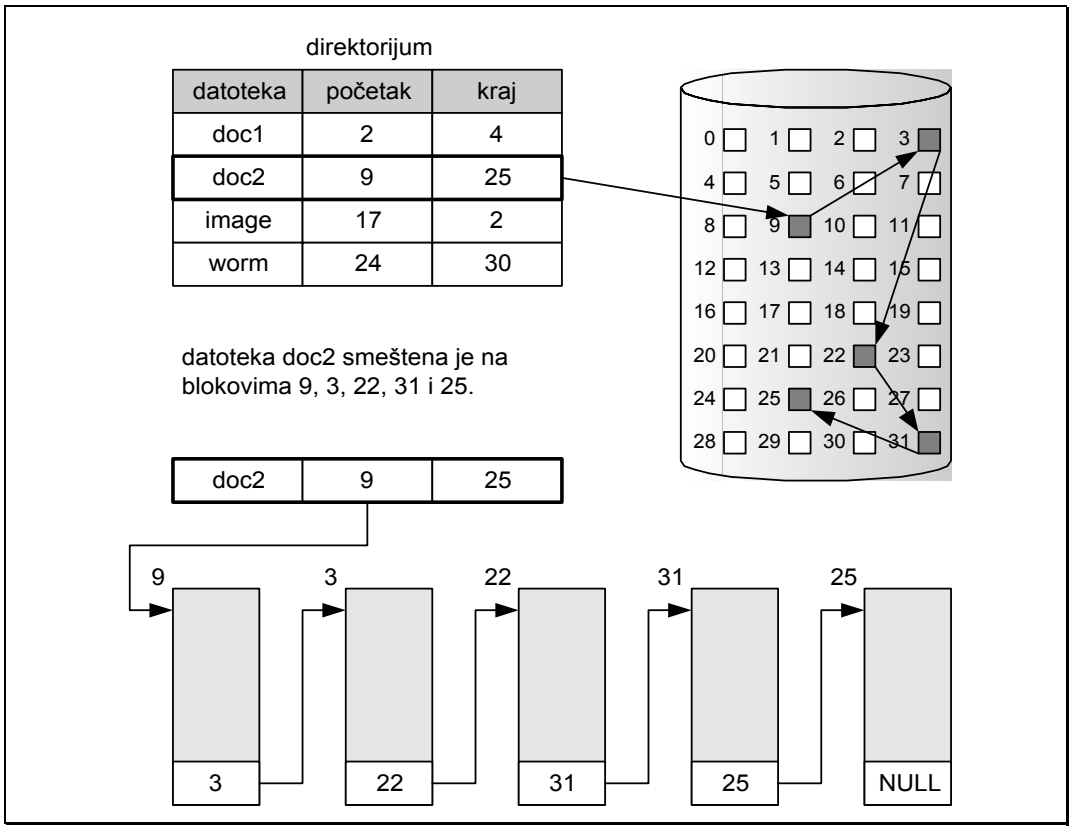
Najveći problem ove metode je to što se datoteci odmah mora dodeliti ukupan adresni prostor na disku. Posle toga, datoteka ne može da raste. U idealnim uslovima (relativno nepromenljiv sadržaj sistema datoteka, veličina nove datoteke se može odrediti pre kreiranja), metoda je pogodna za upotrebu. Međutim, u realnim uslovima, koji uključuju upis novih datoteka čija veličina nije unapred poznata, metoda se u opštem slučaju ne može primeniti. Iz tih razloga, značaj ove metode je čisto teorijski.

Većina operativnih sitema koristi modifikovanu metodu dodele kontinualnog prostora. Datotekama se dodeljuje kontinualan prostor na nivou *extent*-a. *Extent* je celina sastavljena od kontinualnih blokova na disku. Svakoj datoteci se može dododeliti ceo broj ovih celina, koje na disku ne moraju biti konsekventne.

Vezivanje blokova

Blokovi dodeljeni datoteci mogu biti proizvoljno razbacani po celom disku. I ova metoda se može jednostavno realizovati - nekoliko bajtova svakog bloka (32 do 64 bita) odvoji se za smeštanje pokazivača na sledeći blok (slika 11.13). Poslednji blok sadrži null vrednost. Kako svaki blok sadrži pokazivač na sledeći, u kontrolni blok datoteke smešta se samo informacija o početku i kraju datoteke, odnosno broj prvog i poslednjeg bloka datoteke.

Metoda vezivanja blokova je po pitanju performansi ubedljivo na poslednjem mestu, jer je pristup datotekama strogo sekvencijalan, što znači da se pre čitanja nekog bloka datoteke moraju pročitati svi prethodni blokovi. Takođe, pouzdanost ove metode je relativno mala - defekt jednog bloka učiniće nepristupačnim ostatak datoteke. Pouzdanost se može uvećati uvođenjem redundantnih inverznih pokazivača. Eksterna fragmentacija ne predstavlja problem, zato što svaki blok diska može da se dodeli bilo kojoj datoteci.



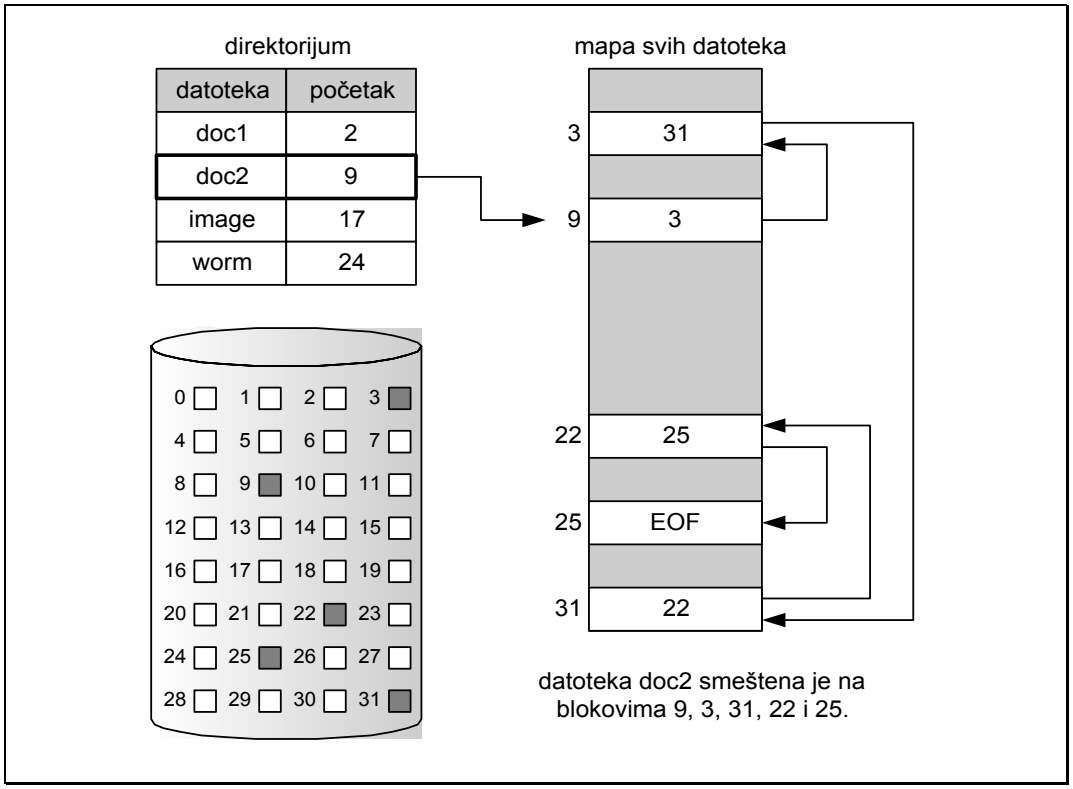
Slika 11.13. *Vezivanje blokova*

Mapa datoteka

Nedostaci metode vezivanja blokova mogu se otkloniti uvođenjem struktura koje će sadržati povezane liste, odnosno prostorni raspored datoteke. Ove liste su, u opštem smislu, promenljive veličine i kao takve nisu pogodne za čuvanje u kontrolnom bloku datoteke. Znači, potrebno je uvesti centralizovanu tabelu u kojoj će na neki način biti opisan prostorni raspored svih datoteka na disku (slika 11.14). Ova tabela se zove mapa datoteke (*file map*) i sadrži najmanje dva atributa: broj bloka i pokazivač na sledeći blok. Objasnićemo metodu mape datoteka na primeru FAT sistema datoteka.

FAT sistem datoteka je dodelu prostora razrešio vrlo specifično, korišćenjem FAT tabele. FAT sistem datoteka u suštini predstavlja kombinivanu metodu kontinulane dodele prostora na nivou *extent*-a i mape datoteka. Sistem datoteka se prilikom kreiranja deli na *extent*-e koji se nazivaju klasteri (*cluster*). U okvirima jednog klastera datoteci je dodeljen kontinualan prostor. Datoteci se može dodeliti ceo broj klastera u proizvoljnom rasporedu, ali jedan klaster ne mogu koristiti dve datoteke.

Neiskorišćen deo klastera zove se *slack* i predstavlja gubitak. Svi klasteri jedne datoteke čine njenu povezanu listu. Povezane liste svih datoteka čuvaju se u FAT tabeli (koja predstavlja mapu datoteke) na sledeći način: svaki klaster diska opisan je jednim zapisom u FAT tabeli, u kome je naveden broj sledećeg klastera dodeljenog datoteci. Kraj datoteke je klaster u čijem slogu u FAT tabeli stoji oznaka za kraj datoteke - EOF (*end of file*). Kontrolni blok datoteke samo broj početnog klastera, dok se svi ostali klasteri određuju na osnovu FAT tabele.



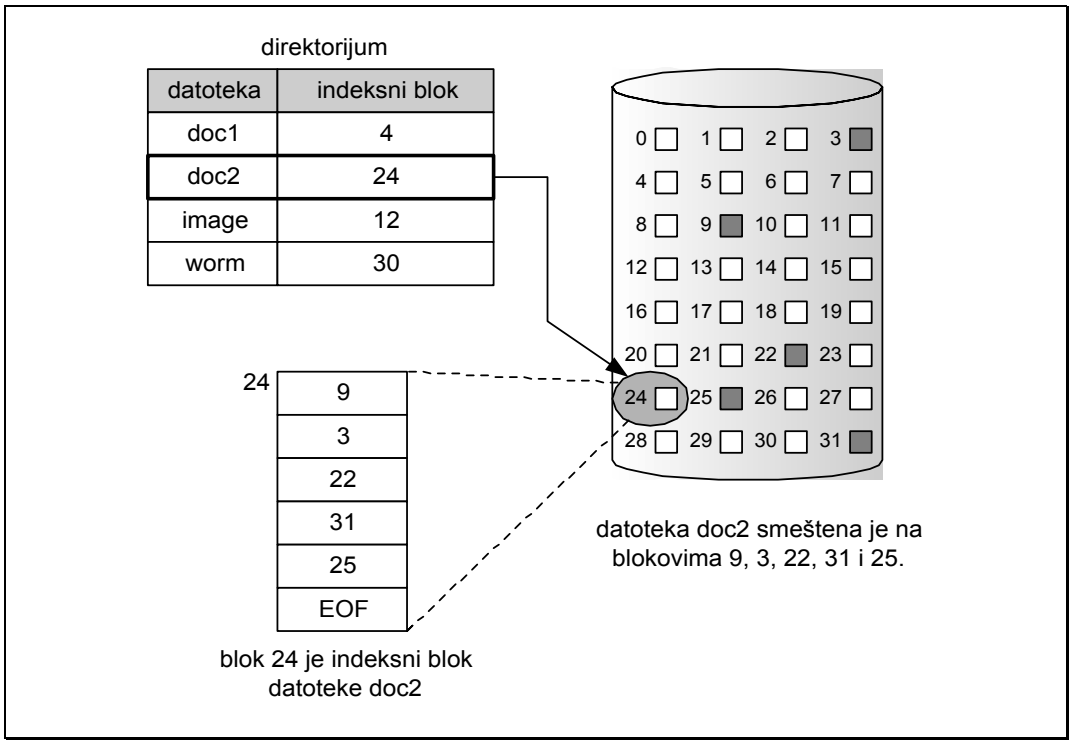
Slika 11.14. *Mapa datoteke*

Mapa datoteke predstavlja premašenje operativnog sistema, odnosno gubitak prostora na disku, ali su performanse ove metode veće u odnosu na metodu vezivanja blokova (radi se sa pokazivačima koji se nalaze u jednoj tabeli). Potencijalne greške u mapi mogu napraviti veliki broj grešaka u sistemu datoteka. Na FAT sistemu datoteka, formira se identična rezervna kopija tabele, koja se redovno ažurira. Na taj način, opšta pouzdanost sistema datoteka se uvećava.

Metoda indeksnih blokova

Svakoj datoteci se dodeljuje indeksni blok koji sadrži sve informacije o prostornom rasporedu datoteke, odnosno o blokovima koji su dodeljeni datoteci (slika 11.15). Zahvaljujući indeksnim blokovima, svi ukazivači na blokove podataka koji pripadaju datoteci smešteni na jednom mestu. Kraj datoteke označen je slogom EOF u indeksnom bloku. U kontrolnom bloku datoteke se nalazi adresa odgovarajućeg indeksnog bloka. Prednosti ove metode su sledeće:

- nema eksterne fragmentacije i
- pristup datoteci je direktan.



Slika 11.15. *Indeksni blokovi*

Potencijalan problem predstavlja dodela prostora velikim datotekama, čiji se prostorni raspored ne može opisati jednim indeksnim blokom. Ovaj problem se može rešiti na dva načina:

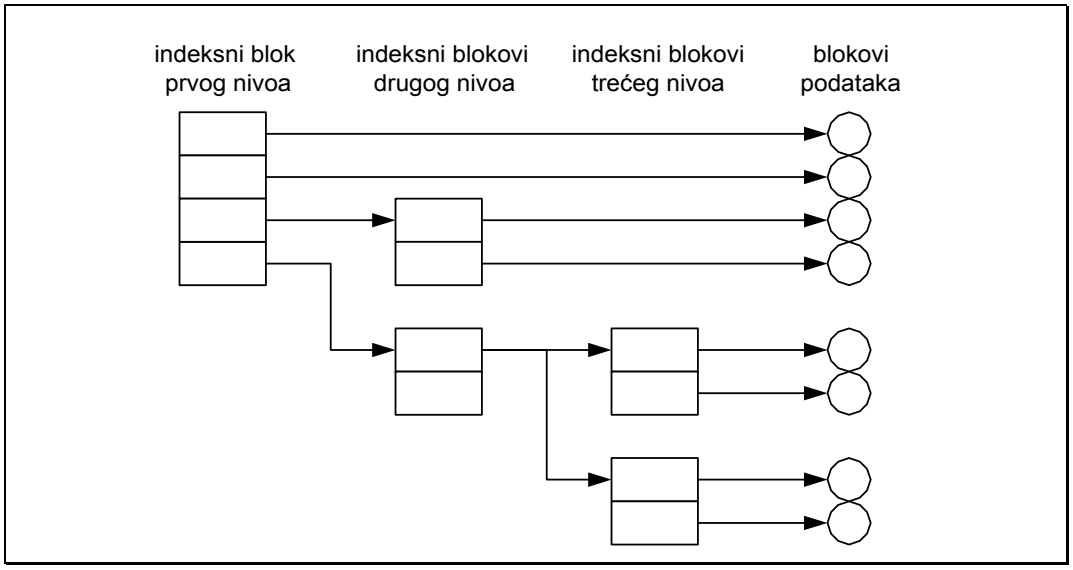
[1] Vezivanjem indeksnih blokova

Za opis prostornog rasporeda datoteke koristi se veći broj indeksnih blokova, pri čemu poslednji zapis u indeksnom bloku predstavlja ukazivač na sledeći indeksni

blok. U kontrolnom bloku datoteke stoji ukazivač na prvi indeksni blok, a kraj datoteke označen je zapisom EOF u poslednjem indeksnom bloku.

[2] Višenivovskim adresiranjem

Kontrolni blok datoteke ukazuje na jedan indeksni blok prvog nivoa, koji ukazuje na n indeksnih blokova drugog nivoa. Indeksni blokovi drugog nivoa adresiraju blokove datoteke. Broj nivoa se po potrebi može povećati. Višenivovsko adresiranje prikazano je na slici 11.16. Na UNIX sistemima prisutno je tronivovsko adresiranje blokova podataka.



Slika 11.16 *Višenivovsko adresiranje*

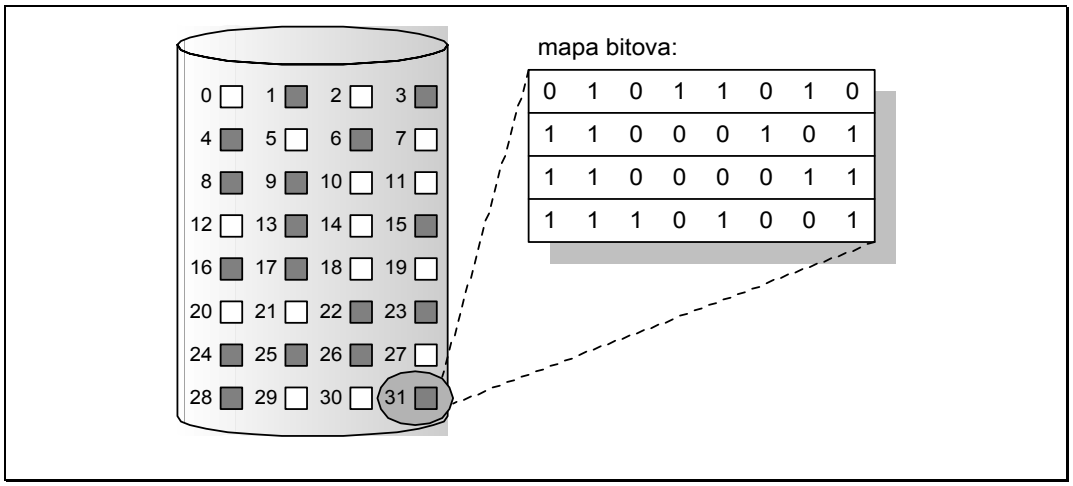
Metoda indeksnih blokova po pitanju performasi nadmašuje prethodno pomenute metode, ali zato unosi najveći utrošak prostora na disku za meta strukture. Takođe, broj indeksnih blokova je ograničen, čime je dodatno ograničen i broj datoteka koje se mogu kreirati na sistemu datoteka. U slučaju mape datoteka, ovo ograničenje ne postoji.

11.7. Upravljanje slobodnim prostorom

Slobodni prostor na disku se dodeljuje datotekama prilikom kreiranja i povećavnja datoteka. Upravljanje slobodnim prostorom značajno utiče na performanse sistema datoteka. Postoje dve osnovne metode za upravljanje slobodnim prostorom:

[1] Mape bitova

Za svi blok diska uvodi se bit koji vrednostima 0 i 1 opisuje da li je taj blok slobodan ili zauzet. Mapa bitova se formira kao struktura podataka na disku, odnosno kao niz svih prethodno pomenutih bitova i kao takva opisuje koji su blokovi slobodni u sistemu datoteka (slika 11.17). Mape bitova zahtevaju formiranje posebne meta strukture na disku, ali se uz pomoć memorijskih struktura za keširanje relativno brzo pretražuju. U tom slučaju, mora se voditi računa o koherenciji između mape na disku i mape u memoriji.



Slika 11.17 *Mape bitova*

[2] Povezane liste

Liste se formiraju na sledeći način: početak liste ukazuje na prvi slobodni blok, a svaki slobodni blok ukazuje na sledeći (slika 11.18).

Metoda ne koristi dopunske strukture za vođenje evidencije o listi slobodnih blokova (osim pokazivača), ali su performanse znatno manje u odnosu na mape bitova. Metoda se može modifikovati na dva načina, čime se uvećavaju i performanse:

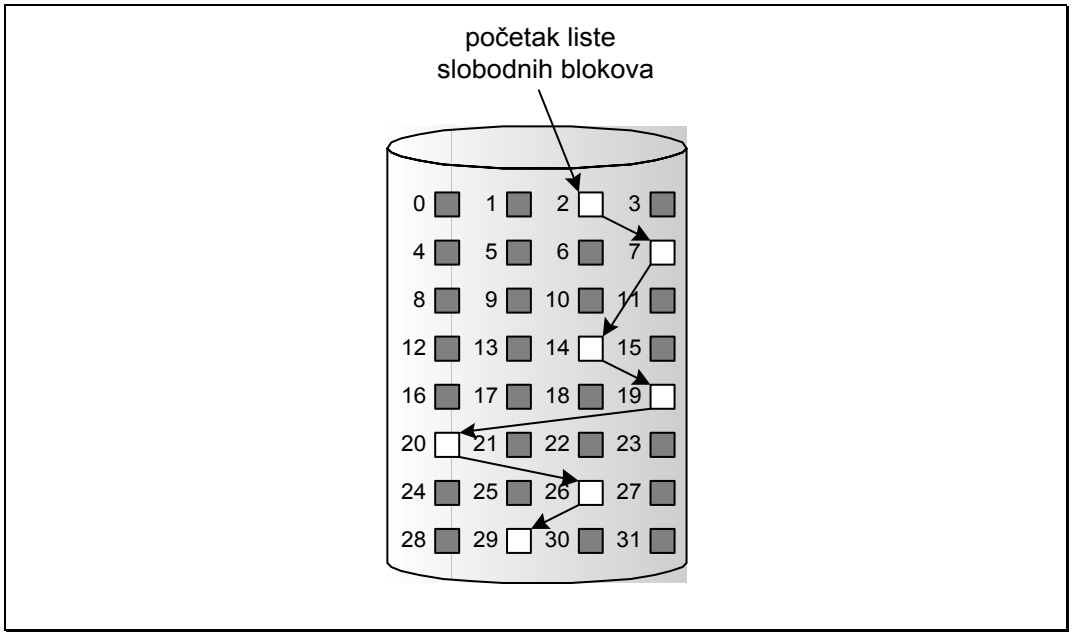
- Grupisanjem ukazivača na slobodne blokove

  U slobodnom bloku se umesto ukazivača na sledeći blok čuvaju ukazivači na sledećih N slobodnih blokova. Poslednji blok ukazuje na sledećih N slobodnih blokova i tako redom. N je ceo broj određen količnikom veličine bloka (512bajtova) i veličine ukazivača.

- Uvođenjem informacije o broju uzastopnih slobodnih blokova

  U slobodnom bloku se čuva ukazivač na sledeći slobodni blok i informacija o broju uzastopnih slobodnih blokova nakon njega. Na ovaj način se jednim

slobodnim blokom može definisati veliki broj slobodnih blokova, pod uslovom da nisu mnogo razbacani.



Slika 11.18 *Povezane liste slobodnih blokova*

11.8. Pouzdanost sistema datoteka

Sistem datoteka je kompleksna struktura podataka, koju osim datoteka i direktorijuma čine i meta strukture kao što su zaglavlje i kontrolni blokovi datoteka. Zbog same kompleksnosti, na pouzdanost sistema datoteka utiče veliki broj činilaca. Na primer, pouzdanost sistema datoteka direkto zavisi od pouzdanosti površine diska na kome se nalazi, ali i od mogućnosti operativnog sistema da oporavi nekonzistentan sistem datoteka nakon nasilnog zaustavljanja sistema.

Oštećenja sistema datoteka se mogu podeliti na:

[1] Fizička oštećenja

Fizička oštećenja sistema datoteka odnose se na neispravnost površine diska i mogu da prouzrokuju nepovratan gubitak podataka. Preliminarna sigurnost se postiže prijavljivanjem defekata na površini diska prilikom kreiranja sistema datoteka. Na ovaj način se pouzdanost sistema može povećati. Međutim, praksa pokazuje da broj fizički neispravnih blokova ima tendenciju rasta, i da je dalji

gubitak podataka moguć. Zato se disk čija je površina fizički oštećena treba što pre zameniti ispravnim diskom.

[2] Logička oštećanja

Logička oštećenja se odnose na neispravnosti u meta strukturama, i najčešće nastaju kao posledica keširanja diskova, odnosno kao posledica informacija koje se čuvaju u *write-behind* kešu. Ukoliko sistem u potpunosti ne isprazni *write-behind* keš u periodima slabe aktivnosti diska i prilikom deaktiviranja sistema datoteka, deo informacija neće biti upisan na medijum. Ovakve situacije se javljaju prilikom nasilnog zaustavljanja sistema, koje može biti prouzrokovano nestankom struje. Manja logička oštećenja se u većini slučajeva mogu otkloniti.

U oba slučaja, posledice zavise od podataka koji su upisani u fizički oštećen blok, odnosno od tipa informacije u *write-behind* kešu koja nije upisana na disk. Ukoliko se radi o podacima koji čine zaglavlje sistema datoteka, sistem datoteka gubi integritet, a postoji mogućnost i da će ceo sistem datoteka biti neupotrebljiv. Ukoliko se radi o meta strukturama, kao što su FAT ili *i-node* tabela, korisnik neće moći da pristupi datoteci ili direktorijumu, iako sami podaci postoje na disku. Ukoliko se radi o datotekama, delovi datoteka mogu biti nepovratno oštećeni. Pouzdanost sistema datoteka može se povećati korišćenjem neprekidnog izvora napajanja (UPS), dnevnika transakcija, i naprednih tehnika kao što je *disk mirroring* (RAID level 1).

Vođenje dnevnika transakcija

Gubitak integriteta se najčešće javlja kao posledica nasilnog zaustavljanja sistema, odnosno promena u objektima sistema datoteka koje nisu blagovremeno ažurirane u meta strukturama, i može za posledicu imati gubitak podataka. Opasnost od gubitka podataka umanjuje se uvođenjem dnevnika transakcija koji prati aktivnosti vezane za promenu meta struktura i samih objekata sistema datoteka. Dnevnik (*journal*) se ažurira pre promene sadržaja objekata i prati relativne promene u sistemu datoteka u odnosu na poslednje stabilno stanje. Transakcija se zatvara po obavljenom upisu i može biti ili u potpunosti prihvaćena ili odbijena. U slučaju oštetećenja, sistem datoteka se može lako rekonstruisati povratkom na stanje poslednje prihvaćene transakcije.

Iako je sam način organizacije i vođenja dnevnika specifičan za konkretan sistem datoteka, zavisno od informacija koji se čuvaju u dnevniku razlikuju se dva opšte metode vođenja dnevnika transakcija:

[1] Praćenje svih promena u sistemu datoteka

Prate se promene u meta strukturama i datotekama. Pouzdanost sistema datoteka se znatno uvećava, ali se performanse umanjuju kao posledica velike redundanse u ciklusima upisa.

[2] Praćenje promena u meta strukturama

Pre ili posle upisa ili izmene datoteka, u dnevnik se upisuju promene načinjene u meta strukturama. Ovaj metod karakteriše manja redundansa i veća brzina rada. Potpuna sinhronizacija objekata sistema datoteka i meta struktura se može postići ukoliko se najpre upiše ili izmeni sadržaj datoteke, a tek nakon toga ažuriraju meta strukture. U suprotnom se ne može garantovati sinhronizacija objekata sistema datoteka i meta struktura.

Arhiviranje i rezervne kopije podataka

Arhiviranje je proces kreiranja kopije značajnih datoteka na drugom medijumu u značajnim trenucima vremena. Arhiva obavlja jednostavnu ali korisnu funkciju - služi za povratak izgubljenih podataka u sistem datoteka. Arhiviranje je preventivna mera - u slučaju havarije sistema, korisnici sa arhivnim kopijama značajnih datoteka osobođeni su frustracije gubitka podataka i vremena koje bi utrošili za njihovo novo kreiranje. Dodatno, ukoliko su značajne datoteke arhivirane, uklanjanje suvišnih datoteka sa diska je relativno lak i brz proces - svi korisnički podaci mogu se izbrisati, nakon čega se korisni jednostavno prenose sa arhiva u sistem datoteka.

Jedina sličnost između sistema datoteka i arhiva je u njihovom sadržaju - obe strukture sadrže iste vrste objekata, odnosno datoteke i direktorijume. Arhiva, za razliku od sistema datoteka, relativno prosta struktura podataka nad kojom se mogu izvršiti dve osnovne operacije - dodavanje datoteka u arhivu i povratak datoteka iz arhive. Složene operacije nad datotekama nikad se ne vrše u arhivi, nego u sistemu datoteka. Pristup podacima u arhivi je strogo sekvencijalan: nove datoteke mogu se dodati isključivo na kraj arhive, a postojeće je teško ili nemoguće obrisati.

Za razliku od arhive, neophodno je da rezervna kopija podataka (*backup*) obuhvati podatke koji obezbeđuju integritet celog sistema ili neke funkcionalne celine, kao što je na primer baza podataka ili sistem za elektronsku poštu. *Backup* obično obuhvata veliku količinu podataka, tako da je kreiranje iste relativno dug proces, i kao takav se zbog nedostatka motivacije ne radi često (izuzetak predstavlja relativno kratak period nakon havarije sistema, u kom ljudi shvataju značaj kopije podataka, ali ga i brzo zaboravljaju). U svakom slučaju, imati bilo koju kopiju datoteke je mnogo bolje nego nemati nikakvu. Proces kreiranja rezervne kopije treba automatizovati, odnosno deklarisati podatke koji trebaju biti smešteni u backup i vreme kada se rezervna kopija može kreirati. Posle toga planer poslova pokreće odgovarajući program koji kreira rezervnu kopiju. Po pravilu, optimalna strategija predstavlja kombinaciju punog (*full*) i inkrementalnog (*incremental*) *backup*a, čime se obezbeđuje potpun integritet podataka, a vreme potrebno za kreiranje kopije i broj upotrebljenih medijuma minimizira.

Po pravilu, backup obuhvata podatake koji su unikatni, značajni, ili se ne mogu rekonstruisati:

- korisničke podatke koje korisnici ne mogu sami arhivirati,

- aplikacije koje nije jednostavno reinstalirati (ukoliko, na primer, sadrže složene konfiguracione datoteke),
- baze podataka i
- sistemske konfiguracione datoteke.

Backup ne treba da obuhvati softver koji se može lako reinstalirati i velike *log* datoteke koje opisuju statistiku aktivnosti sistema.

11.9. Efikasnost i performanse sistema datoteka

Efikasnost sistema datoteka, odnosno efektivno iskorišćenje prostora na disku je problem kom se u prošlosti pridavao veliki značaj, zbog malih kapaciteta diskova.

Na primer, FAT tabela je čist gubitak prostora u FAT sistemu datoteka. Veličina FAT tabele zavisi od veličine i broja ukazivača, a broj ukazivača zavisi direktno od veličine diska, ali je obrnuto proporcionalan veličini klastera. S druge strane, interna fragmentacija je veća ukoliko su klasteri veći. Problem efikasnosti je očigledan: ukoliko je veličina klastera smanji, gubi se prostor na uvećanje FAT tabele. Ako se veličina klastera uveća, gubi se prostor kao posledica interne fragmentacije (dve datoteke ne mogu biti smeštene u jednom klasteru). Problem efikasnosti razrešavaju sistem administratori, koji mogu odabrati veličinu klastera prilikom kreiranja FAT sistema datoteka, zavisno od veličine i broja datoteka koje će na taj sistem biti smeštene. Po pravilu, koriste se manji klasteri (na primer, 8KB) bez obzira na veličinu FAT tabele.

Diskovi koji se danas proizvode su dovoljno veliki, tako da projektanti operativnih sistema mogu zanemariti problem efikasnosti i prvenstveno se mogu posvetiti povećanju performansi. Na performanse sistema datoteka može se uticati na više načina. Jedna od tehnika koja značajno utiče na performanse sistema datoteka je keširanje.

Keširanje

Keš (*cache*) je međumemorija kojom se premošćava jaz između spore elektromehanike diska i brze operativne memorije. Keš postoji kao hardverska komponenta na procesorima i novijim modelima diskova. Međutim, operativni sistem u cilju povećanja performansi implementira softversko keširanje putem straničnog keša (*page cache*) i keša za datoteke (*file cache*).

Keširanje datoteka se obavlja u ciklusima čitanja i upisa. Keširanjem ciklusa čitanja se sve datoteke kojima se čisto pristupa prenose u keš u operativnoj memoriji. Sledeći pristup tim datotekama biće znatno brži jer se datoteke čitaju iz memorije, a ne sa diska. Keširanje ciklusa čitanja nije destruktivno. Keširanje ciklusa upisa predstavlja

problem jer je efikasno jedino u slučaju *write-back* tehnike (tehnike odloženog upisivanja). Ovaj način keširanja uvećava performanse jer se podaci iz keša prenose na disk u periodima niske aktivnosti korisnika, ali u opštem slučaju umanjuje pouzdanost. Informacije koje se nalaze u *write-behind* kešu, a koje se ne prenesu na disk pre zaustavljanja sistema mogu prouzrokovati logička oštećenja sistema datoteka, a u nekim slučajevima i gubitak podataka. Da bi se to sprečilo, *write-back* keširanje se koristi u kombinaciji sa dnevnikom transakcija.

Postoje četiri glavna pravca razvoja disk keširanja:

[1] Podela keš memorije na više logičkih celina po različitim kriterijumima (*cache splitting*)

Prema ovoj metodi, sve reference se po nekom kriterijumu dele na lokalne (*Temporal*) i prostorne (*Spatial*), pa samim tim se i keš memorija deli na takve delove (*Temporal Cache*) i (*Spatial Cache*). Kriterijumi za podelu mogu biti različiti. Gotovo po pravilu, keš se deli na minimalno tri funkcionalna dela a to su:

- Keš podataka (*data cache*).

  To je najveća komponenta keša i sadrži nedavno korišćene delove datoteka. Veoma bitni parametri ovog dela keša su veličina i algoritam za alokaciju blokova. Za razliku od memorijskog keširanja, gde algoritam izvršava hardver, upravljanje kešom podataka obavlja softver što mu daje brojne dodatne mogućnosti. Dok se kod memorijskog keširanja koriste 3 bazične hardverske šeme (direktno mapirana, set-asocijativna i puna-asocijativna), dotle se kod disk keširanja koriste hash algoritmi. Tipičan algoritam je hash algoritam baziran na modulu N, gde je N broj blokova u u keš baferu. Algoritam je realizovan na operativnom sistemu UNIX, baziran na baferskim zaglavljima koji opisuju stanje u baferu i samih baferskih blokova. Alokacija i pretraživanje baferskih elemenata je po hash algoritmu baziranom na modulu N, a mora se poštovati sinhronizacija tipična za višeprocesnu okolinu (lock i unlock procedure). Na starijim operativnim sistemima, količina RAM memorije dodeljena za keš bafer je fiksna, a kod modernih operativnih sistema veličina keš bafera je adaptibilna prema memorijskom optrećenju sistema. Veličina keš bafera raste kada broj procesa opadne, odnosno zauzeće memorije opadne, a smanjuje se sa povećanjem opterećenja sistema.

- Keš meta-podataka (*metadata cache*).

  Područje meta-podataka je relativno malo, ali vrlo značajno. Sama struktura područja meta-podataka je sasvim različita u odnosu na same datoteke, pa se vrlo dugo koristi razdvajanje keša za podatke i medata keša. Za operativni sistem UNIX, područje meta-podataka je tabela indeksnih čvorova. Za svaku otvorenu datoteku kreira se u kernelskoj memoriji struktura poznata pod nazivom *in-core inode* koja sadrži sve informacije za živu datoteku u višeprocesnoj okolini. Po zatvaranju datoteke *in-core inode* više nije potreban, ali se može čuvati u

memoriji zato što neki proces može ponovo da otvori tu istu datoteku. Sa tom idejom *in-core i-node* područje prerasta u *i-node* keš, koji sadrži *in-core i-nod*ove živih datoteka, kao i većine nedavno otvorenih datoteka. Slična priča važi za NTFS, gde se u kešu meta-podataka čuvaju atributi nedavno otvorenih datoteka. Kod FAT sistema datoteka, keš meta-podataka sadrži delove FAT tabele.

- Keš direktorijumskih blokova (*directory cache*).

  Direktorijumi su specijalne datoteke koje sadrže informacije o datotekama (FCB). Da bi se došlo do neke datoteke, mora se pročitati direktorijumski blok koji sadrži FCB za tu datoteku, a jedan direktorijumski blok sadrži FCB za više datoteka. Praksa je pokazala da je veoma korisno uvesti novu komponentu keša koja će sadržavati nedavno korišćene direktorijumske blokove i ta komponenta naziva se keš direktorijumskih blokova.

  Sve tri komponente imaju različitu veličinu, različit *hash* algoritam za alokaciju/pretraživanje, različitu šemu za zamenu podataka u punom kešu i različitu realizaciju ciklusa upisa. Naravno, ubedljivo je najveći keš podataka, ali keš meta-podataka i direktorijumski keš, iako su osetno manji, drastično povećavaju performanse

[2] Različite metode predikovanog čitanja (read-ahead)

Pored zahtevanih podataka u keš se upisuju predikovani podaci. Nezaobilazni faktor savremenih disk uređaja je upravo tako realizovan keš na samom disk uređaju. Predikcija se obavlja na nivou cele ili dela staze. Predikovani keš na samom disku je vrlo dobra osnova za predikovano čitanje u keš na nivou operativnog sistema, što znači da disk ima već spremne podatke u svojim keš segmetima, a data keš treba da odluči hoće li to prebaciti kod sebe ili ne.

Najsavremenija rešenja kombinuju oba keša da se međusobno dopunjuju ali da se ne dupliraju po podacima, što znači da keš u RAM memoriji i keš na disku sadrže unikatne podatke koji se mogu koristiti iz oba keša, a ne isključivo iz glavnog data keša, to je princip ekskluzivnosti keša.

[3] Različite metode razmenjivanja podataka u kešu (*cache replacement*)

Tehnika razmenivanja podataka u kešu (*Cache Replacemnet Policy*) obavlja se na različite načine. Počevši od pseudo LRU (*least recently used*) algoritma sa LRU bitima, imamo razne LRU modifikacije sa vremenskim varijablama dodeljenim svakom bloku u kešu, ili sa upotrebom *victim* keša ili čak i *victim* evidencije (*victim tracer*). Princip *victim* keša je da se pored glavnog keša ubaci mala keš memorija (*Victim Cache*) koja će selektivno sadržavati u sebi podatke koji moraju napustiti glavni keš zbog razmene (*evicted data*). Ideja je da ono što je izbačeno, može uskoro zatrebati. Podesnim keš algoritmom i upotrebom male *victim* keš memorije se osetno poboljšavaju performanse. Osnovni nedostatak LRU algoritama je što uzimaju samo jedan podatak, a to je vreme poslednje

refernce, a ne prati frekvenciju korišćenja blokova, tako da neki vrući blokovi mogu lako ispasti iz keša.

Drugi parametar koji se može koristiti kao kvalifikator za izbacivanje blokova je frekvencija korišćenja blokova u kešu i ti algorimi se nazivaju LFU (*least frequently used*). Jedan od najboljih algoritama je LRFU (*least recently-frequently used*) koji kombinuje oba parametra, nedavnost korišćenja i frekvenciju korišćenja sa ponderišućim koeficijentom koji favoriše prvi ili drugim parametar.

[4] Različite metode tehnike odloženog upisa (*write-back*)

Kada je reč o ciklusu upisa, najčešće korišćena tehika je tehnika naknadnog upisa (*write back*), a prateća tehnika koja se pri tome koristi je *journaling*, odnosno tehnika vođenja dnevnika transakcija. Koncept je da se svaki upis na disk prihvati u bafer za upis (*write buffer*) a da se kasnije upiše na disk. *Write back* je vrlo ozbiljna stvar, jer pored osetnog poboljšanja performansi koju nudi za cikluse upisa, nosi veliki rizik od gubitka podataka, pa je suštinska stvar smanjiti taj rizik na minimum. To se, praktično, postiže upotrebom *Non-Volatile* RAM memorije koja ima baterijsko napajanje, ali je prilično skupa. Jedno veoma zanimljivo rešenje je da se umesto *NV-RAM* memorije koristi disk, koji je u suštini magnet i čuva podatke nakon nestanka napajanja. Projekat je poznat pod nazivom "*Disk Caching Disk*".

Svaki od prethodno pomenutih delova keša ima sopstevnu *write* šemu, pri čemu je bitno naglasiti da su greške u kešu meta-podataka ili u direktorjumskom kešu fatalne, mogu oštetiti mnogo datoteka odjedanput, dok se greške u data kešu reflektuje samo u jednoj datoteci. Mnogi stariji operativni sistemi za keš meta-podataka imaju sinhroni upis na disk i ta tehnika se naziva *write-through*.

Veliki je broj naučnih radova vezanih za problematiku *write-back*. Pojava koja se pri tome neminovno dešava je razlika izmedju sadržine diska i write bafera i to se u stručnoj literaturi zove *Destage*. Postoji veliki broj algoritama za *Destage*. Kod tehnike upisa javljaju se praktično dve odvojene akcije upisa: upis u područje meta-podataka (*metadata update*) i upis u područje podataka. Postoje više različitih tehnika koje se razlikuju po načinu realizacije obe vrste upisa. Pomenućemo neke od njih.

- *Flushing* tehnika

  Tehnika je tipična za UNIX operativni sistem. Keš funkciniše kao običan *write back* keš, a na svakih 30 sekundi izvršava se proces *sync* koji sve blokove iz keša koji su se promenili (*dirty*) upisuje na disk

- Upis na zatvaranje (*write on close*)

  Kod ove tehnike, upis na disk iz *write-back* keša se odvija kada proces zatvori datoteku. Tehnika je krajnje nepovoljna za kratkotrajne procese, koji brzo zatvaraju datoteke.

- LFS (*Log structured File System*)

  Ova tehnika podrazumeva potpuno novi koncept sistema datoteka, gde ceo disk izgleda kao veliki *log*. Disk deli na segmente, relativno velike, u koje se pomoću velikih transfera upisuju i datoteke i metadata podaci. Na prvi pogled deluje moćno za ciklus upisa, ali LFS imaju problem sa ciklusom čitanjem kao i sa čišćenjem *loga*, jer broj segmenata je relativno mali, puni segmenti zbog brisanja datoteka postaju delimično popunjeni segmenti pa se kompaktuju u pune a oslobađaju se novi segmenti.

- *Soft Updates* tehnika

  Ova tehnika garantuje ispravan poredak operacija u području meta-podataka, tako da u slučaju oštećenja broj potencijalnih grešaka bude minimalan i može lako da se ispravi.

- Tehnika vođenja dnevnika transakcija (*journaling*)

  Dnevnik transakcija omogućava upravljanje upisnim baferom - svi prispeli zahtevi za upis prihvataju se u bafer dnevnika. Sadržina bafera se može upisatu na disk (ne u sistem datoteka, već u dnevnik) kada se bafer napuni. Na taj način se jednim velikim upisom može obuhvati mnogo malih zahteva za upis. Kako će se ujednačavati stanje između dnevnika i sistema datoteka, zavisi od konkretne realizacije. Poenta je u što manjem radu sa diskom - ažuriranje sistema datoteka prema sadržaju dnevnika mora uneti što manje pozicioniranja i mehaničkih kašnjenja. Journaling omogućava brz i kvalitetan oporavak sistema datoteka uz minimalna oštećenja u slučaju havarije.

Defragmentacija

Fragmentisana datoteka je datoteka koja prilikom upisa na disk nije smeštena u strogo sekvencijalni niz blokova. Čitanje fragmentisanih datoteka u opštem slučaju traje duže, s obzirom na to da je potrebno više puta pomeriti glave za čitanje i pisanje na samom disku. Fragmentacija predstavlja problem na sistemima sa lošom tehnikom keširanja diskova, teško se može izbeći, a mora biti svedena na neki prihvatljivi minimum. Problem se rešava defragmentacijom sistema datoteka, odnosno aranžiranjem datoteka u sistemu datoteka u kontinualne sekvence blokova. Većina sistema datoteka koristi specifične programe za defragmentaciju, a pre same defragmentacije preporučuje se izrada rezervne kopije podataka.

11.10. Značajniji sistemi datoteka (UNIX/Linux)

UNIX posmatra svaki sistem datoteka kao nezavisnu hijerarhijsku strukturu objekata (direktorijuma i datoteka) na čijem se vrhu nalazi *root* direktorijum (/).

Objektima se pristupa pomoću relativne ili apsolutne putanje i imena objekta. U objekte UNIX sistema datoteka spadaju:

- regularne datoteke,
- direktorijumi (mogu se posmatrati kao specijalne datoteke koje sadrže objekte sistema datoteka, uključujući i poddirektorijume),
- hard linkovi (alternativna imena datoteka),
- simbolički linkovi (prečice, odnosno datoteke čiji su sadržaji putanje i imena objekata na koji upućuju),
- blok i karakter specijalne datoteke (opisuju uređaje, odnosno drajvere u kernelu. Korišćenjem ovih datoteka mogu se vršiti ulazno-izlazne operacije na uređajima koje opisuju) i
- imenovani pipeline.

Osnovna struktura svih domaćih UNIX/Linux sistema datoteka je slična. UNIX sistem datoteka čine:

- zaglavlje (*superblock*),
- tabela indeksnih čvorova (*i-node* tabela),
- blokovi sa podacima (*data blocks*),
- direktorijumski blokovi (*directory blocks*) i
- blokovi indirektnih pokazivača (*indirection block*).

Superblok je zaglavlje sistema datoteka i sadrži informacije o sistemu datoteka u celini, kao što su njegova veličina, tip i zastavica čistoće (*dirty flag*). U superbloku se nalazi zaglavlje *i-node* tabele i zaglavlja listi slobodnih indeksnih čvorova i slobodnih blokova za podatke. Superblokovi svih aktivnih sistema datoteka se keširaju u RAM memoriji računara i periodično se upisuju na disk.

U okviru dela sa podacima nalaze se blokovi podataka, direktorijumski blokovi i blokovi indirektnih pokazivača. Sadržaj regularnih datoteka čini određeni broj blokova podataka. Direktorijumi se posmatraju kao specijalne datoteke čiji sadržaj čine direktorijumski blokovi, koji sadrže jedan deo kontrolnih blokova datoteka (*file-info* strukture). Direktorijumski blokovi su tabele sastavljene od određenog broja *file-info* struktura. Svaki objekat koji se nalazi u tom direktorijumu (*directory-entry*) predstavljen je jednom *file-info* strukturom. Svaka *file-info* struktura sadrži ime objekta kog predstavlja i broj indeksnog čvora kojim je taj objekat u potpunosti opisan. Na ovaj način je omogućeno kreiranje hard linkova.

Indeksni čvor (*i-node*) je osnovna struktura UNIX sistema datoteka koja u potpunosti opisuje jedan objekat. Svaki UNIX sistem datoteka ima tabelu indeksnih čvorova. I-node je drugi deo kontrolnog bloka datoteke i sadrži sve informacije o objektu koji opisuje osim imena:

- tip objekta (npr. regularna datoteka, direktorijum ili simbolički link) i pristupna prava za tri vlasničke kategorije,
- broj hard linkova na dati objekat,
- *user ID*, odnosno ID korisnika koji je vlasnik objekta,
- *group ID*, odnosno ID grupe korisnika kojoj objekat pripada,
- veličinu objekta izraženu u bajtovima,
- vreme zadnjeg pristupa objektu (*access time*) u UNIX vremenskom formatu,
- vreme zadnje modifikacije objekta (*mod time*) u UNIX vremenskom formatu,
- vreme zadnje modifikacije indeksnog čvora objekta (*i-node time*) u UNIX vremenskom formatu,
- listu direktnih pokazivača na blokove sa podacima, koja je dovoljna da se adresiraju prvih 10-12 blokova podataka koji čine početak datoteke (broj zavisi od tipa sistema datoteka) i
- listu indirektnih pokazivača (lista pokazivača na jednostruke, dvostruke i trostruke indirektne blokove).

Svaka datoteka u sistemu datoteka koristi jedan indeksni čvor. Indeksni čvor (i-node) je struktura podataka koja potpuno opisuje jednu datoteku ili direktorijum i zauzima oko 128 bajtova prostora na disku. Broj indeksnih čvorova se ne menja dinamički - npr. ukoliko je na sistemu datoteka ostalo pet slobodnih i-node čvorova, korisnik će moći da kreira najviše pet novih datoteka, bez obzira na slobodni prostor. Sistem datoteka neće kreirati automatski još pet slobodnih i-node čvorova. Gustina indeksnih čvorova, kao i veličina bloka mogu se definisati prilikom kreiranja sistema datoteka.

Kompromis između veličine *i-node* tabele i brzine rada nastao je u originalnoj verziji UNIX sistema datoteka. Naime, većina UNIX datoteka je relativno male veličine. Umetanjem prvih 10-12 pokazivača na blokove sa podacima u *i-node*, tabela indeksnih čvorova će, takođe, biti relativno mala. Na taj način, manje datoteke se mogu potpuno opisati indeksnim čvorom. Prilikom alokacije prostora za veće datoteke, koristi se dodatni blok pokazivača na blokove podataka (*single indirection block*). Za još veće datoteke, dodatni prostor se može alocirati korišćenjem dvostrukih i trostrukih indirektnih pokazivača. Ova metoda dinamičke alokacije prostora je efikasna i kao takva se sa malim modifikacijama koristi u najnovijim verzijama UNIX i Linux sistema datoteka. Na primer, maksimalne veličine datoteka adresirane pomoću direktnih i 32-bitnih indirektnih pokazivača na ufs sistemima datoteka iznose 16GB (ukoliko je sistemski blok veličine 1KB) i 64TB (ukoliko je sistemski blok veličine 8KB).

Neki UNIX sistemi datoteka dozvoljavaju kreiranje rupa u datotekama (*hole*). Rupe u datoteci ne zauzimaju prostor na disku, a sistem datoteka ih simulira određenim brojem nula. Na ovaj način se smanjuje broj upotrebljenih blokova podataka za datoteke koje u svom binarnom obliku imaju velike nizove nula (male binarne datoteke,

deljene biblioteke i neke baze podataka). Rupe se implementiraju upisivanjem specijalnih vrednosti na mestima adresa blokova podataka u indirektnim pokazivačima. Te vrednosti znače da ni jedan blok podataka nije upotrebljen za taj deo datoteke, već da se na tom mestu nalazi rupa.

Opis konkretnih sistema datoteka

U značajnije sisteme datoteka, koji su domaći za UNIX i Linux sisteme, spadaju:

[1] Minix

Minix je najstariji, i verovatno najpouzdaniji domaći UNIX sistem datoteka. Maksimalna veličina minix sistema datoteka je 64 MB, a imena datoteka ne mogu biti duža od 30 karaktera.

[2] Xia

Xia je modifikovana varijanta minix-a, ukida limite na veličinu sistema datoteka i broj karaktera u imenu datoteke, ali ne donosi nove mogućnosti. Retko se koristi, ali se smatra da je pouzdan

[3] ext2

Linux second extended filesystem, jako popularan sistem datoteka visokih performansi. Ovaj sistem datoteka je nekoliko godina predstavljao podrazumevani izbor pri instalaciji Linux operativnog sistema

[4] ext3

Linux third extended filesystem, koji se ukratko može definisati kao ext2 proširen dnevnikom transakcija. Potpuno je kompatibilan sa prethodnom verzijom (ext2), tako da se nadogradnja ostvaruje jednostavnim kreiranjem dnevnika. Za funkcionisanje ext3 sistema datoteka neophodna je podrška na nivou kernela. Ext3 podržava tri režima vođenja dnevnika koji na različite načine utiču na pouzdanost i performanse sistema.

- *Journal* je režim praćenja svih promena u sistemu datoteka, kako u meta-data oblasti, tako i u objektima-datotekama, čime se pouzdanost sistema datoteka znatno uvećava. Redundansa koju ovaj režim rada unosi je velika.
- *Ordered* je režim praćenja promena u meta-data oblasti, pri čemu se promene u objektima sistema datoteka upisuju pre ažuriranja i-node tabele. Ovo je podrazumevati režim rada dnevnika, koji unosi manju redundansu i garantuje potpunu sinhronizaciju objekata sistema datoteka i meta-data oblasti.
- *Writeback* je režim praćenja promena u meta-data oblasti, pri čemu se i-node tabela može ažurirati pre upisa promena u objekte sistema datoteka. Ovo je najbrži režim rada, ali ne garantuje konzistenciju meta struktura, odnosno sinhronizaciju objekata sistema datoteka i meta struktura, što može dovesti do

neprijatnih situacija u sistemu datoteka, kao što su pojave nove i stare verzije datoteke itd.

Dnevnik transakcija umanjuje performanse sistema, pri čemu je pad performansi pri praćenju celokupne aktivnosti u sistemu datoteka (*journal*) znatno veći od pada performansi pri praćenju aktivnosti meta podataka (*ordered* i *writeback*).

[5] ReiserFS

Jedan od prvih transakcionih sistema datoteka je ReiserFS. ReiserFS značajno povećava performanse pri radu sa malim datotekama, koje su kod ostalih journaling sistema datoteka veoma slabe. Brojni testovi pokazuju da je ReiserFS 8 do 15 puta brži od ext2, pri radu sa datotekama manjim od 1KB. Dodatno, ReiserFS razrešava problem interne fragmentacije, čime se povećava efikasnost iskorišćenja diskova. Ovako visoke perfromase pri radu sa malim datotekama ReiserFS postiže na osnovu optimizovanog balansnog stabla (B+ stablo, jedno po sistemu datoteka) i dinamičke alokacija indeksnih čvorova (za razliku od fiksne alokacije koju koristi ext2). Dodatno, ReiserFS koristi promenljivu veličinu sistemskog bloka, a male datoteke upisuje u roditeljski direktorijum, zajedno sa svojom *fileinfo* strukturom. U dnevniku se ažuriraju samo promene u meta strukturama. ReiserFS se ponaša loše pri radu sa šupljim datotekama (*sparse files*), a takođe radi sporije sa velikim datotekama u odnosu na ext2.

[6] XFS

XFS je proizvod kompanije Silicon Graphics, Inc, razvijen početkom devedesetih godina. Predstavlja robusni, puni 64-bitni sistem datoteka, prvenstveno namenjen za SGI IRIX, ali je kasnije prenešen i na Linux. U XFS sistem datoteka uvedene su alokacione grupe, odnosno linearni regioni jednake veličine, koji se definišu za svaki disk. Svaka alokaciona grupa ima svoju *i-node* tabelu i listu slobodnog prostora. Alokacione grupe su nezavisne i mogu učestvovati u paralelnim ulazno-izlaznim operacijama, čime se omogućava paralelizam ulazno-izlaznih operacija na istom sistemu datoteka. Interno, svaka alokaciona grupa koristi efikasna B+ stabla, koja čuvaju informaciju o zonama slobodnog prostora i o slobodnim indeksnim čvorovima. XFS optimizuje dodelu slobodnog prostora, metodom odložene alokacije (*delayed allocation*), što je kritično po pitanju performansi upisa. Kao i ReiserFS, XFS u dnevniku transakcija vodi evidenciju o izmenama meta struktura. Dnevnik se može realizovati na više načina, a najčešće se formira kao zaseban sistem datoteka. XFS je brz pri radu sa velikim datotekama.

[7] JFS

JFS je puni 64 bitni transakcioni sistem datoteka koji je namenjen prvenstveno za IBM servere, ali je, takođe, prenešen i na Linux. Kao i svi *journaling* sistemi, obezbeđuje visoku pouzdanost, brz oporavak sistema datoteka i visoke performanse. Kao i ReiserFS i XFS, tako i JFS prati promene samo meta podataka. JFS je sistem datoteka zasnovan na *extent*-ima (odnosno sekvencama kontinualnih blokova koji se dodeljuju datoteci), što mu omogućava da

fleksibilno upravlja datotekama na disku. Stablo je bazirano na B+ strukturi. JFS koristi različite veličine za sistemske blokove (512, 1024, 2048 i 4096 bajtova), što omogućava prilagođenje sistema datoteka prema potrebama. Tabela indeksnih čvorova i liste slobodnih blokova se održavaju dinamički. JFS sadržaj malih direktorijuma upisuje u njihov indeksni čvor, što uvećava performanse pri radu sa malim direktorijumima. Veliki direktorijumi se realizuju u formi optimizovanog B+ stabla, koje optimizuje pretraživanje, unos novih objekata i brisanje. JFS se dobro snalazi i sa šupljim (*sparse files*) i sa gusto popunjenim datotekama (*dense files*). Kao puni 64 bitni sistem datoteka, predstavlja odličan izbor za realizaciju sistema datoteka velikih kapaciteta.

11.11. Značajniji sistemi datoteka (DOS/Windows)

Počev od MS-DOS operativnog sistema (*Disk Operating System*), zaključno sa Windows 2003 Server familijom, Microsoft je u svoje operativne sisteme ugrađivao podršku za FAT, FAT32 i NTFS sisteme datoteka. FAT i FAT32 sistemi datoteka su vidljivi iz svih DOS/Windows operativnih sistema, a mogu se aktivirati i pod UNIX/Linux sistemima u režimu čitanja i pisanja. NTFS je vidljiv isključivo iz NT zasnovanih Windows operativnih sistema, kao što su Microsoft Windows NT, 2000, XP i 2003 Server. Ovaj sistem datoteka se može aktivirati i pod DOS i Windows 98/ME operativnim sistemima, ali su za to potrebni specijalni uslužni programi (kao što je ntfsdos, firme Sysinternals). Linux dozvoljava aktiviranje NTFS sistema datoteka u režimu čitanja, dok je podrška za upis još uvek u fazi razvoja i kao takva ne preporučuje se za korišćenje

FAT (File Allocation Table)

FAT je jednostavan sistem datoteka zasnovan na principu mape datoteka (realizovane u vidu FAT tabele sa 16-bitnim adresiranjem). FAT je zbog ograničenja po pitanju ukupne veličine sistema datoteka zadovoljavao potrebe većine korisnika u periodu kada su kapaciteti diskova na XT (8086, *extended technology*) i AT (80286, *advanced technology*) računarima bili reda veličine 10-20MB. Naime, FAT ima ograničenje na 16.535 klastera, što će reći da se veličinom klastera praktično određuje i maksimalna veličina sistema datoteka. Naime, korisnik je mogao dobiti sistem datoteka najvećeg kapaciteta samo ako koristi klastere veličine 64KB, što unosi značajne gubitke kroz internu fragmentaciju (*slack*).

Svaka datoteka na FAT sistemu datoteka ima svoje ime, datum i vreme, i osnovne atribute. Ime datoteke se može zadati isključivo korišćenjem 7-bitnog ili 8-bitnog ASCII ili ANSI standarda. Ime se zadaje u formatu 8.3 - 8 karaktera za ime, a 3

za ekstenziju, kojom je ujedno i određen tip datoteke. U atribute kojima se datoteke opisuju spadaju:

- H (*hidden*) - datoteka je skrivena, komanda dir je neće prikazati prilikom izlistavanja sadržaja direktorijuma),
- S (*System*) - sistemska datoteka,
- R (*Read-only*) - datoteka je samo za čitanje i
- A (*Archive*) - dodeljuje se datotekama nakon kreiranja i određuje koje će datoteke biti uključene u listu prilikom formiranja rezervne kopije podataka (*backup*). Objekti koji nemaju ovaj atribut neće biti arhivirani prilikom izrade rezervne kopije. Atribut A se uklanja nakon arhiviranja, a ponovo postavlja nakon promene sadržaja objekta.

Pristupna prava i vlasnički odnosi nisu regulisani, tako da se na nivou sistema datoteka ne može izvršiti nikakva kontrola pristupa.

Pouzdanost sistema je relativno niska: postoje dve identične FAT tabele (sadržaj jedne se ažurira na osnovu sadržaja druge). Međutim, ove tabele su fizički smeštene jedna do druge na disku. Ukoliko se prilikom fizičkog oštećenja diska ošteti jedna FAT tabela, verovatnoća je velika da će i druga biti oštećena. Na FAT sistemu datoteka ne postoji dnevnik transakcija, tako da je upotreba *write-back* keširanja krajnje rizična.

FAT sistem datoteka se danas koristi pri formatiranju disketa. Potpuni pristup FAT sistemu datoteka, odnosno aktiviranje u režimu čitanja i pisanja, može se ostvariti iz gotovo svih poznatijih operativnih sistema (DOS, sve verzije Windows-a, UNIX i Linux).

FAT32

Sa porastom kapaciteta diskova, Microsoft je ponudio FAT32 kao proširenje FAT sistema datoteka. Osnovna promena u odnosu na stari FAT je 32-bitno adresiranje, čime je omogućeno kreiranje većih sistema datoteka sa prihvatljivom veličinom klastera.

FAT32 se upotrebljava kao sistem datoteka na Windows 95/98 operativnim sistemima. Neki korisnici *dual-boot* računara, na kojima su instalirani Windows NT/2000/XP i Linux, koriste posebnu particiju sa FAT32 sistemom datoteka za dvosmernu razmenu datoteka između ta dva operativna sistema. Razlog je u tome što Windows nema podršku za rad sa domaćim Linux sistemima datoteka, i što se aktiviranje NTFS sistema datoteka pod Linuxom u režimu čitanja i pisanja ne preporučuje (podrška za NTFS pod Linuxom još uvek je u fazi razvoja).

NTFS

Sa pojavom Windows NT 3.1 operativnog sistema Microsoft je predstavio novi sistem datoteka - NTFS (*New Technology File System*). NTFS je evoluirao iz HPFS sistema datoteka (*High Performance File System*) koji pripada operativnom sistemu IBM OS/2. NTFS je u verziji koja se isporučivala uz Windows NT 4.0 znatno napredovao u smislu stabilnosti i fleksibilnosti. Trenutna verzija, NTFS 5.1 isporučuje se uz Windows Server 2003 i opisana je u nastavku teksta. Za razliku od FAT i FAT32 sistema datoteka, potpun lokalni pristup podacima na NTFS sistemu datoteka je moguć isključivo iz Windows 2000/XP/2003 operativnih sistema. Korisnici Windows NT 4.0 na kom je instaliran Service Pack 5 moći će da pristupe nekim datotekama. NTFS se može aktivirati u režimu čitanja (i pisanja, na svoju odgovornost) i pod Linux sistemom, montiranjem na aktivno UNIX stablo.

Navodimo neke osnovne podatke vezane za NTFS:

- maksimalna veličina NTFS volumena kreće se od 2-16TB pod Windows 2003 Server operativnim sistemom.
- maksimalna veličina datoteke određena je veličinom volumena.
- NTFS sistemi datoteka se ne mogu kreirati na disketama.

Navešćemo ukratko i neke prednosti NTFS u odnosu na FAT i FAT32 sisteme datoteka:

- otpornost u slučaju otkaza (*fault tolerance*) korišćenjem RAID tehnike. Ukoliko operativni sistem otkrije neispravan deo na disku, iskoristiće ispravnu kopiju podataka i načiniti novi sektor koji će se nadalje koristiti umesto oštećenog,
- veća skalabilnost ka većim particijama, odnosno manji pad performansi sa rastom kapaciteta volumena,
- mogućnost kreiranja aktivnog direktorijuma (hijerarhijske baze distribuirane na domen kontrolerima),
- mogućnost kompresije datoteka i direktorijuma u cilju smanjenja zauzetog prostora na diskovima,
- mogućnost šifrovanja datoteka s ciljem povećanja sigurnosti,
- kontrola pristupa pomoću određenih skupova pristupnih prava dodeljenih ovlašćenim korisnicima i grupama,
- praćenje aktivnosti na sistemu datoteka, koje omogućava brz oporavak od logičkih oštećenja,
- disk kvote, kojima se kontroliše zauzetost sistema datoteka od strane korisnika. Korisnicima se može na upotrebu dodeliti određeni prostor na disku koji nikako ne mogu prekoračiti i

- funkcionalnost montiranja sistema datoteka na direktorijume (poput UNIX *mount*-a).

Osnovna struktura NTFS sistema datoteka je volumen (*volume*), zasnovan na logičkoj disk particiji. Volumen može zauzeti deo diska, ceo disk, a može se i rasprostrati na više diskova. Svi meta podaci, kao što su informacije o volumenu, smešteni su regularnoj datoteci. NTFS koristi klastere (*clusters*) kao osnovne tj. ishodišne jedinice alokacije diska. Klaster je grupa sektora diska koji je stepen broja dva. S obzirom da je veličina klastera manja nego kod za 16-bitnog FAT sistema, interna fragmentacija je smanjena. Klasteri se na NTFS sistemu adresiraju korišćenjem logičkih brojeva klastera *(LCN - logical cluster numbers)* kao disk adresa.

NTFS sistem datoteka, po svojoj strukturi, podseća na relacionu bazu podataka koja organizuje podatke na principu B+ stabla. Sam Microsoft je objavio vrlo malo podataka o strukturi ove baze. Prostor imena (NTFS *name space*) je organizovan hijerarhijom direktorijuma, pri čemu *index root* sadrži vrh (koren) B+ stabla. Datoteka u NTFS sistemu datoteka nije jednostavan tok bajtova (*byte stream*) kao što je to u MS-DOS sistemu, već je to struktuirani objekat koji se sastoji od atributa. Svaka datoteka je opisana pomoću jednog ili više zapisa u području koje je smešteno u okviru posebne datoteke - *Master File Table* (MFT). MFT predstavlja prvu liniju svake NTFS particije, i ima svoju rezervnu kopiju. Ova redundansa osigurava oporavak sistemskih datoteka u slučaju oštećenja. Pokazivači u boot sektoru volumena opisuju lokaciju na kojoj se nalazi MFT. Svaka datoteka na jednom NTFS volumenu ima svoj jedinstveni 64-bitni identifikator (*unique ID*) koji se zove referenca datoteke (engl. *file reference*).

Datoteke i direktorijumi su objekti NTFS sistema datoteka i slične su strukture (NTFS tretira direktorijum kao specijalnu datoteku). Svaki objekat ima svoje ime i sadržaj. Sadržaj datoteke su konkretni podaci. Sadržaj direktorijuma čine indeksi relativnih lokacija, veličina direktorijuma i bitmapirana reprezentacija strukture direktorijuma. NTFS podržava Unicode imena datoteka, dužine do 255 karaktera. Svaki objekat u sistemu datoteka osim imena opisuju i:

- Standardni atributi

  Ove atribute NTFS nasleđuje sa FAT sistema datoteka. Standardni atributi su H (*hidden*), S (*system*), R (*read-only*) i A (*archive*).

- Prošireni atributi (*extended atributes*)

  U navedene atribute spadaju razni atributi koje dodaju programi pomoću kojih su datoteke kreirane. Na primer, Microsoft Word dokumentu pridružuje ime autora (*author*), naslov (*title*) i predmet (*subject*).

- Sigurnosni opis (*security descriptor*)

  Sigurnost NTFS volumena je izvedena iz Windows objektnog modela, karakterističnog za Windows 2000/XP/2003. Svaki fajl objekat ima sigurnosni opis (*securiy descriptor*) koji je smešten u MFT. Ovaj atribut sadrži pristupni žeton vlasnika datoteke i pristupnu kontrolnu listu (*access control list*) koja

utvrdjuje privilegije koje su dodeljene svakom korisniku koji ima pristup datoteci.

Sve promene na strukturi sistema datoteka koje se izvrše unutrašnjim transakcijama se prate na posebnom mestu. Transakcioni sistem ne garantuje da će svi korisnički podaci biti oporavljeni posle pada sistema ili greške, zapravo su meta podaci sistema datoteka neoštećeni i reflektuju neko konzistentno stanje pre pada.

Jedna od osnovnih razlika NTFS 5 sistema datoteka u odnosu na starije verzije je realizacija simboličkih linkova putem tačaka redirekcije (*reparse point*). *Reparse point* je specijalni podatak kojim se identifikuje drajver uređaja; akcija usmerena kao *reparse point*u preusmerava se na definisani drajver. Simbolički link je specijalna vrsta *reparse point*a kojim se izvršava redirekcija na određenu datoteku ili direktorijum.

12

Distribuirani sistemi

Sadržaj poglavlja:

12.1. Uvod u distribuirane sisteme
12.2. Tipovi mrežno-orijentisanih operativnih sistema
12.3. Mreže i komunikacija
12.4. Detekcija greške i oporavak
12.5. Distribuirani sistemi datoteka

12.1. Uvod u distribuirane sisteme

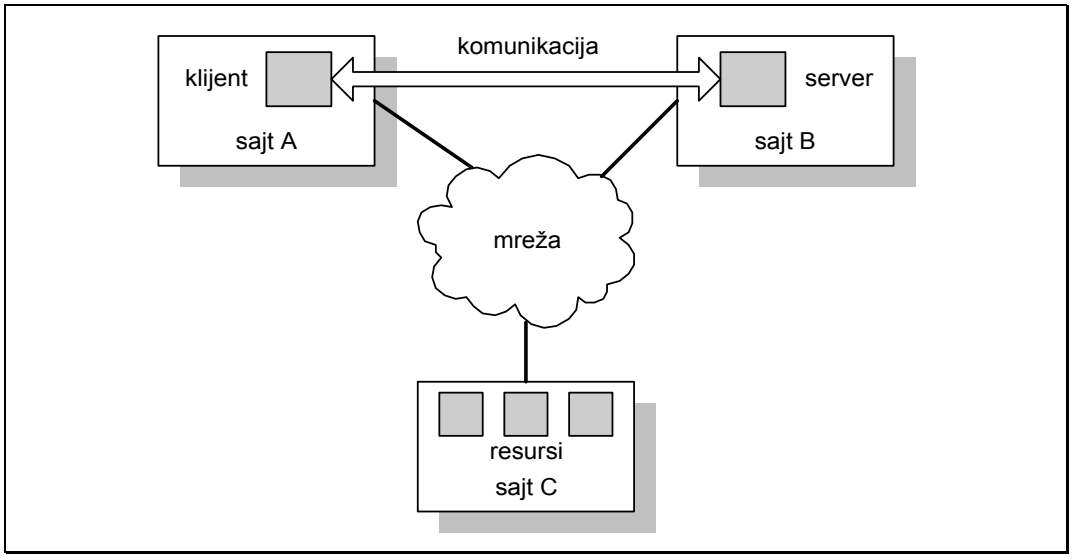
Distribuirani sistemi (*distributed systems*) predstavljaju kolekciju procesora odnosno računara koji ne dele zajedinčku memoriju i sistemski časovnik. Umesto toga, svaki procesor, odnosno računar ima sopstvenu lokanu memoriju, a međusobna komunikacija se ostvaruje putem mreže LAN ili WAN tipa. U najprostijem slučaju, distriburiani sistem se sastoji od servera, klijenata i sekundarnih memorija rapostranjenih na raznim mestima. Osnovna namena distribuiranog sistema je da obezbedi efikasno i pogodno deljenje resursa.

Procesori u distribuiranom sistemu razlikuju se po svojoj snazi i funkcijama koje obavljaju - distribuirani sistem može činiti kolekcija ručnih (*handheld*) sistema, PC računara i veoma moćnih (*mainframe*) računara. Osim podataka, datoteka i štampača, distribuiraju se i procesi, tako da distribuirani sistemi obezbeđuju korisnicima visoke performanse, deljivost podataka i visoku pouzdanost. Kao i kod centralizovanih, tako se i kod distribuiranih operativnih sistema moraju obezbediti mehanizmi sinhronizacije

procesa i komunikacije između procesa i rešiti problem zastoja. Pri tome se mogu pojaviti problemi koji nisu karakteristični za centralizovane operativne sisteme.

Sa stanovišta jednog procesora drugi procesori i nihovi resursi su udaljeni resursi (*remote resources*), dok su njegovi sopstveni resursi lokalni (*local resources*). Procesori, odnosno računarski sistemi u distribuiranom sistemu se nazivaju sajt (*site*), čvor (*node*) ili računar (*computer*, *machine*, *host*). U ovoj knjizi će se, uglavnom, koristiti termin sajt.

Tipičan distribuirani sistem prikazan je na slici 12.1.



Slika 12.1. *Tipičan distribuirani sistem*

Distribuirani sistemi zahtevaju mrežnu infrastrukturu i mogu biti realizovani kao klijent-server sistemi ili kao ravnopravni računarski sistemi koji po mreži dele resurse. Klijent-server arhitekuru karakteriše postojanje računara koje predstavljaju servere (serveri podataka, serveri za izračunavanje i serveri za štampu) i računara koji koriste njihove usluge. Udruženi sistemi (*clustered system*) se sastoje od udruženih računara, odnosno od dva ili više nezavisnih računara koji dele diskove (*strorage*) i čvrsto su povezani LAN mrežom. Svaki računar u udruženom sistemu se naziva čvor (*node*). Jedan nivo softvera za udruživanje se izvršava na svakom čvoru. Postoje dve vrste udruživanja: asimetrično i simetrično. U asimtetričnom udruživanju (*asymmetric clustering*) jedan server (čvor) izvršava aplikaciju dok ostali - prateći serveri (*monitoring servers*) prate rad glavnog servera u budnom ali neaktivnom stanju. U slučaju otkaza glavnog servera jedan od pratećih servera preuzeće njegovu ulogu. U simetričnom udruživanju (*symmetric clustering*) svi serveri su aktivni i izvršavaju aplikaciju, čime se drastično poboljšavaju performase, ali sistem mora da izdrži otkaz jednog ili više servera kao i u perthodnom slučaju.

Karakteristike distribuiranih sistema

Distribuirane sisteme karakterišu sledeće osobine:

[1] Transparentnost

Distribuirani sistem korisniku treba da izgleda kao konvencionalni, centralizovani sistem.

[2] Otpornost na greške

Distribuirani sistem treba da nastavi funkcionisanje u slučaju bilo kog otkaza. Ako bilo koji sajt u distrubuiranom sistemu otkaže, ostali sajtovi mogu nastaviti i završiti započeti posao. Od distribuiranog sistema se očekuje da detektuje otkaz i da pronađe sajt koji će zameniti onoga ko je ispao iz igre.

[3] Skalabilnot

Sa povećanjem zahteva, sistem treba lako da prihvati dodavanje novih računara i resursa.

[4] Deljenje resursa

Distribuirani sistemi obezbeđuju mehanizme za deljenje datoteka, obradu informacija u distribuiranim bazama podataka, deljenje štampača i specijalizovanog hardvera.

[5] Ubrzavanje izračunavanja

Ubrzavanje izračunavanja može da se obavi takozvanim deljenjem opterećenja (*load sharing*). Proces se izdeli na celine koje se obrađuju na posebnim računarima u mreži. Nakon toga, rezultati parcijalnih izračunavanja se spajaju.

Može se izvesti zaključak da su distribuirani sistemi naslednici *main-frame* sistema koji moćne i skupe računare zamenjuju mrežom radnih stanica i personalnih računara.

12.2. Tipovi mrežno-orijentisanih operativnih sistema

Mrežno-orijentisani operativni sistemi se prema svojim karakteristikama mogu podeliti na mrežne operativne sisteme i distribuirane operativne sisteme.

Mrežni operativni sistemi

Mrežni operativni sistemi obezbeđuju takvu okolinu u kojoj korisnik može pristupiti udaljenim resursima na dva načina: procedurom prijavljivanja na odgovarajuću udaljeni računar i transferom datoteka sa udaljene mašine na sopstvenu.

[1] Procedura prijavljivanja na udaljeni računar (*remote login*)

Predstavlja značajnu funkciju mrežnog operativnog sistema koji omogućava korisnicima da se prijave na različite mašine na mreži. Internet obezbeđuje telnet servis za tu svrhu, koji omogućava korisniku da postane lokalni korisnik na udaljenoj mašini.

[2] Transfer datoteka sa udaljene mašine (*remote file transfer*)

Značajni servis ftp (*file transfer protocol*) omogućava korisnicima da obave transfer datoteka sa jednog računara na drugi. Prenos se obavlja pomoću lokalnog korisnika formalno prijavljenog na udaljenom računaru, pri čemi se može koristiti univezalni korisnik *anonymous*. Naime, potrebno je obaviti kvazi-login proceduru nakon koje može doći do uzimanja datoteka (*get*) sa udaljenog sistema, slanja datoteka na udaljeni sistem (*put*), listanja sadžaja direktorijuma na udaljenom računaru (*ls*) i pozicioniranja na direktorijum na udaljenom računaru (*cd*).

Distribuirani operativni sistemi

U distribuiranim operativnim sistemima korisnik pristupa udaljenim resursima kao da su lokalni.

Distribuirane operativne sistema karakterišu tri vrste migracija: migracija podataka, migracija izračunavanja i migracija procesa, koje kontroliše sam operativni sistem.

[1] Migracija podataka

Postoje dve alternativne metode koje se koriste za deljenje podataka. Po jednoj metodi, korisnik koji želi da pristupi udaljenoj datoteci dobija kopiju datoteke. Originalna datoteka se zamenjuje kopijom ukoliko korisnik bilo šta izmeni. Ova metoda je karakteristična za AFS (*Andrew File System*), pri čemu se automatsko kreiranje kopije realizuje ftp protokolom. Prema drugoj metodi prenosi se samo zahtevani deo datoteke. Ako korisnik zahteva novi deo datoteke, inicira se novi transfer po mreži. Ovaj postupak podseća na DP tehniku (*demand paging*), karakterističnu za rad sa virtuelnom memorijom, a koristi se kod NFS sistema datoteka (*Network File System*), SMB (*Samba*) i novijih varijanti AFS. Prvi metod je bolji u slučaju da je korisniku potreban veći deo datoteke, jer su manje većih transfera efikasnije nego više manjih. Drugi metod je bolji u slučaju da korisnik zahteva manji deo datoteke.

[2] Migracija izračunavanja

Uzmimo slučaj procesa koji treba da otvori više datoteka smeštenih na različitim računarima u mreži i da na osnovu njih obavi razna izračunavanja, odnosno obradu. Umesto da kreira kopiju svake datoteke na svojoj lokalnoj strani, pa da vrši obradu nad njom, proces može inicirati obradu na udaljenoj strani, a zatim pokupiti rezultate. Takva migracija obrade može da se realizuje na korišćenjem rpc poziva (*remote procedure call*) i slanjem poruka. Proces P šalje poruku udaljenom računaru. Udaljeni računar kreira novi proces Q koji će obaviti obradu i poslati rezultate procesu P. Oba procesa se mogu izvšavati konkurentno.

[3] Migracija procesa

Migracija procesa je proširenje migracije izračunavanja. Ideja je u sledećem: proces kreiran na lokalnoj strani ne mora se izvršavati strogo na svom računaru. Ceo proces ili neki njegovi delovi mogu se izvršavati na drugim računarima. Ova šema donosi brojne pogodnosti:

- balansirano opterećenje (*load balancing*) - migracijom procesa računari u mreži se mogu ravnomerno opteretiti,
- ubrzavanje obrade (*computation speedup*) - ako se proces podeli na više delova koji se konkurentno izvršavaju na različitim mašinama, vreme izvršavanja procesa se smanjuje,
- hardversko preferenciranje (*hardware preference*) - proces može imati karaktersitike koje zahtevaju izvršavanje na specijalizovanom hardveru, a ne na svojoj lokalnoj mašini
- softversko preferenciranje (*software preference*) - proces može zahtevati sofver koji je raspoloživ na nekom drugom računaru,
- pristup podacima (*data access*) - nekada je mnogo brže migrirati proces na računar na kome se nalaze podaci koje proces zahteva, nego kopirati podatke lokalno da bi ih proces obrađivao.

Postoje dve altrenativne metode migracije procesa. Prema prvoj metodi, lokalni računar pokušava da sakrije od korisnika činjenicu da je proces migriran, tako da korisnik nije svestan šta distribuirani sistem radi u pozadini. U drugom slučaju, lokalni računar eksplicitno traži dozvolu od korisnika da migrira proces.

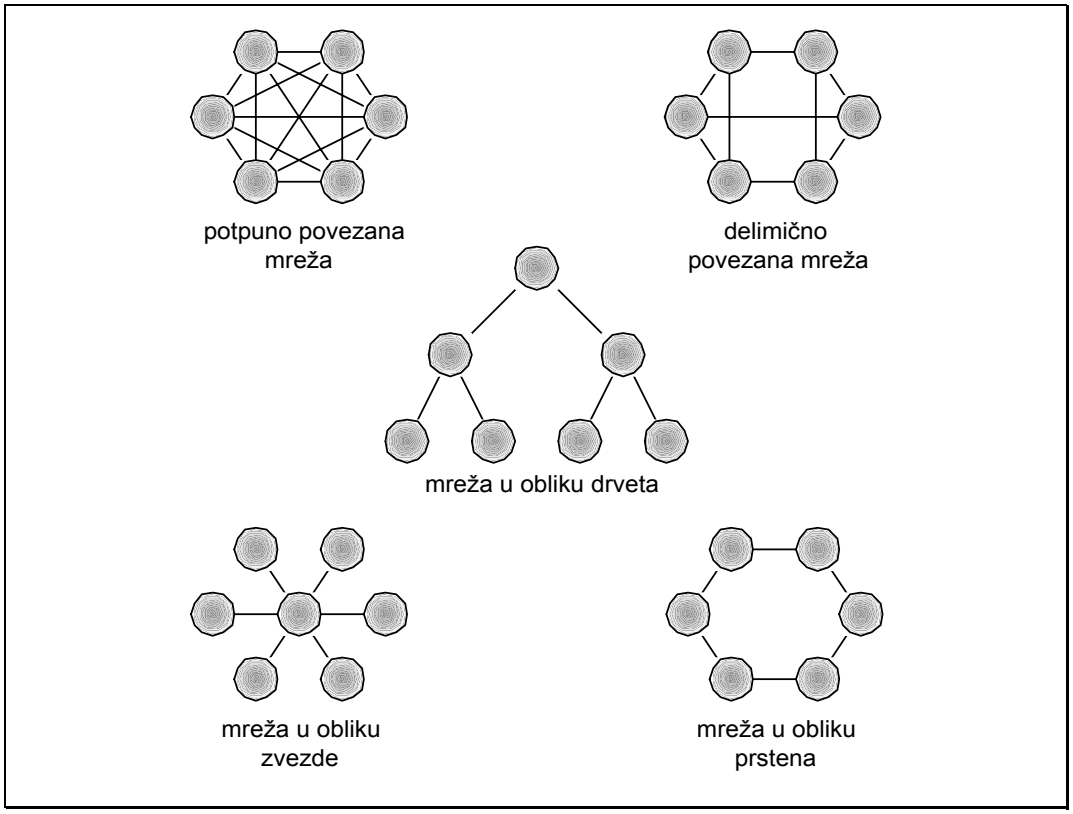
Tipičan primer distribiranih migracija je *web*-server. Prvo, imamo migraciju podataka između servera i klijenta. Dalje, prisutna je i migracija obrade: klijent inicira obradu u distribuiranoj bazi a zatim dobija rezultate od servera. Migracije procesa je postignuta putem Jave: Java aplet se šalje sa servera na računar na kome će se proces izvršavati.

12.3. Mreža i komunikacija

Osnova za realizaciju distribuiranih sistema je mreža. Razmotrićemo osnovne topologije mreža, od kojih svaka ima svoje prednosti i mane. Sledeći kriterijumi se uzimaju u obzir prilikom poređenja mrežnih instalacija:

- troškovi instalacije, odnosno troškovi fizičkog povezivanja računara u mrežu),
- troškovi komunikacije, odnosno vremenski i novčani troškovi slanja poruke sa sajta A na sajt B i
- raspoloživost, odnosno mogućnost pristupa podacima u slučaju otkaza neke veze ili sajtova.

Na slici 12.2. prikazano je šest različitih mrežnih topologija.



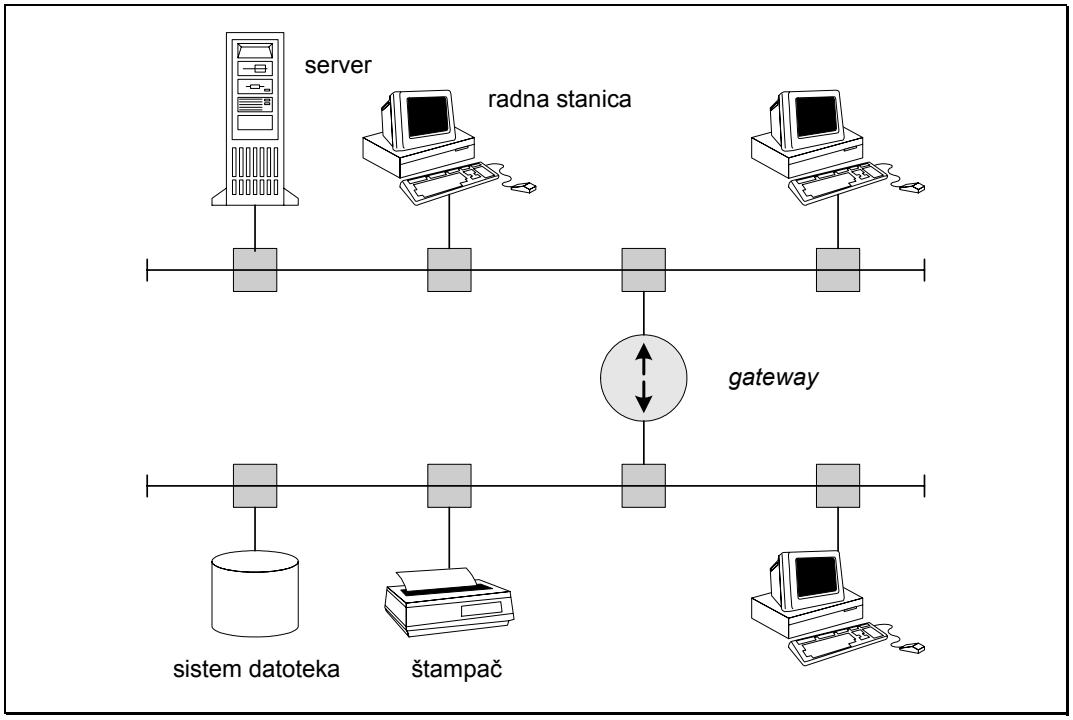
Slika 12.2. *Različite mrežne topologije*

Svaki krug predstavlja jedan sajt ili čvor mreže, a linija između sajtova predstavlja direktnu fizičku vezu između sajtova. U slučaju potpuno povezane mreže

(*fully connected network*), svaki sajt ima direktnu fizičku vezu sa svim preostalim sajtovima. Sa povećanjem broja sajtova u mreži broj veza raste po kvadratnoj funkciji. Ove topologija je jako skupa i nepraktična za neke velike sisteme, dok za male sisteme ovakva topologija ima smisla. Sve preostale mreže su delimično povezane (*partially connected network*). Kod njih važi pravilo da postoje direktne fizičke veze između nekih čvorova, ali ne između svih. U slučaju da ne postoji direktna veza između dva čvora, poruka se mora rutirati kroz sekvencu komunikacionih veza, što smanjuje ukupne troškove, ali povećava komunikacione troškove.

Na slici su prikazane različite vrste računarskih mreža, kao što su mreža u formi stabla (*tree-structured network*), mreža u obliku prstena (*ring*), mreža u obliku zvezde (*star*). Svaka mrežna struktura ima svoje karakteristike. Na primer, ako nastupi otkaz u mrežnom stablu, mreža se može izdeliti na više podmreža, između kojih se ne može ostvariti komunikacija. U strukturi zvezde komunikacija je najbrža, ali ako otkaže čvor u centru, cela mreža ispada iz igre. Struktura prstena ima najmanju brzinu komunikacije ali je otporna na otkaze pojedinih veza.

Dva osnovna tipa mreža su LAN i WAN. LAN mreže povezuju računare koji su distribuirani na malom prostoru, kao što su jedna zgrada ili više susednih zgrada i generalno se koriste u kancelarijskom okruženju.

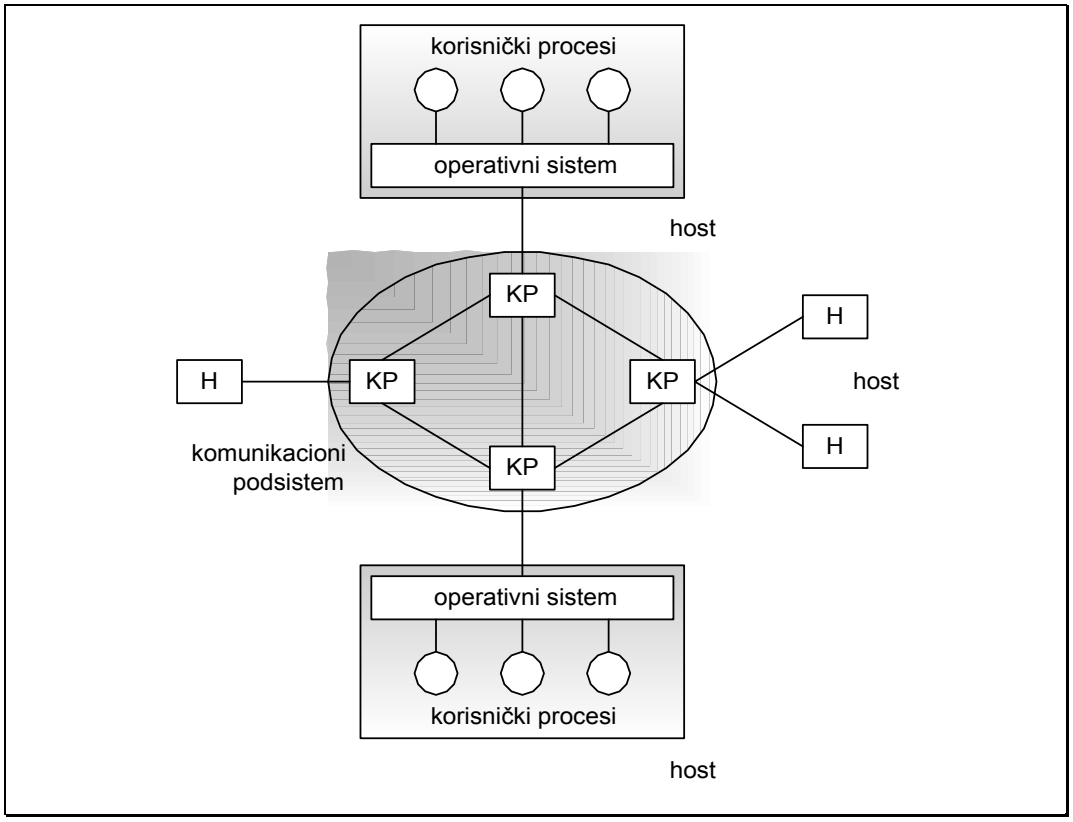


Slika 12.3. *Tipična LAN mreža*

Svi sajtovi u LAN mreži su relativno blizu tako da komunikacioni linkovi, po pravilu, imaju visoke brzine transfera i manju verovatnoću greške u komunikaciji. Sajtovi se povezuju kvalitetnim kablovima, kao što su UTP i optički kablovi, a mrežne strukture koje se najčeće koriste su prstenovi ili zvezde. Brzina komunikacije varira između 1Mb/sec do 1Gb/sec. Dve najčešće korišćene varijante su 100BaseTX i 1000baseTX Ethernet sa brzinom od 100 i 1000Mb/sec.

Tipična LAN mreža je prikazana na slici 12.3. Sastoji od računara različite procesorske snage, različitih deljivih periferijskih uređaja kao što su štampači, diskovi ili trake. Po pravilu, jedan ili više rutera (*gateways*) spajaju fizički razdvojene mreže.

WAN mreže čini znatno veći broj računara i mrežnih uređaja u odnosu na LAN. WAN mreže su distribuirane na širem geografskom prostoru, što izaziva pojavu da su neke veze spore i nepouzdane.



Slika 12.4. *Tipična WAN mreža*

Tipični linkovi su telefonske linije, mikrotalasni linkovi i satelitski kanali, i oni se kontrolišu preko specijalnog komunikacionog procesora. Komunikacioni procesor je odgovoran za definisanje interfejsa pomoću koga će priključeni sajtovi komunicirati

preko mreže i za obavljanje transfera podataka između sajtova. Tipična WAN mreža je prikazana na slici 12.4. Oznaka KP na slici 12.4 odnosi se na komunikacioni procesor, a oznaka H na host računar.

U lokalu, sajtovi su povezani LAN mrežama, dok se povezivanje na velike distance odvija preko sistema rutera. Ruteri na jednoj strani prihvataju LAN konekciju, a na drugoj strani se preko telefonskih linija i modemskih uređaja spajaju sa sličnim ruterom. Za modemske komunikacije koriste se dva protokola, stari uucp UNIX protokol i novi univerzalni PPP protokol..

Komunikacije

Komunikacija u računarskim mrežama zahteva da se prethodno obezbede sledećih pet mehanizama.

[1] Imenovanje računara na mreži i mehanizmi razrešavanja imena u adrese

Dva procesa koji žele da komuniciraju preko mreže moraju međusobno da se identifikuju. Svaki proces na lokalnom računaru ima jedinstven identifikator procesa, a svaki računar ima ime koje je jedinstveno u toj mreži. Svaki proces na mreži postaje unikatan ukoliko se opiše pomoću sledeća dva podatka <*hostname, PID*>. Kako računari mnogo lakše i brže rade sa brojevima, osim simboličkih imena računara uvode se i brojne vrednosti, odnosno IP adrese. Ime računara se može razrešiti u IP adresu.

Neophodno je da svaki računar zna imena i IP adrese računara sa kojima treba da komunicira. Tabela parova ime računara - IP adresa može se čuvati lokalno. Razrešavanje imena na osnovu lokalnih tabela je nezgodno u slučaju velikog broja računara na mreži. Zato su uvedene brojne tehnike, odnosno servisi koji obavljaju tu funkciju. Distribuirane baze podataka sa tabelama parova <ime, adresa> nalaze se na više računara na mreži, koji se nazivaju serveri imena (*nameserver*). Jedna od tehnika je DNS (*domain name system*). Radi lakše obrade mapiranja, računari se grupišu u logičke celine - domene, koji mogu biti hijerarhijski. U kontekstu domena, računari se na Internetu označavaju kao logičko ime sastavljeno iz više delova. Po pravilu, puna kvalifikovana imena počinju unikatnim imenom računara na lokalnoj mreži i svim domenima kojima računar pripada (na primer fwall2.hack.co.yu znači računar fwall2 na domenu hack.co.yu). Po pravilu, DNS razrešava adresu po komponentama imena u obrnutom redosledu. Svaka komponenta imena je domen koji ima svoj server imena, koji na bazi imena vraća adresu servera imena odgovornog za to ime. Naš primer bi se razrešio u četiri koraka: (1) kernel prvo upućuje zahtev domenu yu da mu vrati adresu servera imena za domen co.yu, (2) server imena yu vraća IP adresu imena servera co.yu, (3) kernel potom upućuje zahtev serveru imena co.yu za adresu imena servera hack.edu.yu, (4) od servera imena hack.edu.yu koji sadrži računar od interesa dobija se IP adresa računara fwall2.hack.co.yu.

Cela procedura izgleda komplikovana i očekivano je da dugo traje, ali upotrebom keširanja razrešavanje imena može znatno da se ubrza.

[2] Strategije rutiranja

Za ostvarivanje komunikacije između dva hosta najčešće je potrebno više uzastopnih fizičkih veza, a komunikacija se može ostvariti preko više alternativnih putanja. Svaki host mora imati svoju tabelu rutiranja (*routing table*), koja sadrži sve moguće putanje koje se mogu koristiti za komunikaciju sa drugim hostovima. Tabele mogu sadržavati informacije o brzini i transportnim troškovima različitih komunikacionih putanja. Tabele su podložne promenama, bilo ručno ili programski.

Postoje tri vrste rutiranja:

- fiksno rutiranje: putanja od hosta A do hosta B je uvek fiksna, specificira se unapred po kriterijumu optimalnih komunikacionih troškova i ne menja se sve dok putanja ne doživi hardverski otkaz. Fiksno rutiranje je loše u slučaju preopterećenja mreže.
- virtuelno rutiranje: putanja od hosta A do hosta B je fiksna samo za jednu sesiju, druga sesija može izabrati neku drugu putanju.
- dinamičko rutiranje: putanja od hosta A do hosta B je dinamička, određuje se uvek pre slanja svake poruke, i svaki put se može odabrati neka druga putanja.

Svaka šema ima svoje prednosti i mane. Fiksnim i virtuelnim rutiranjem poruke se prenose u strogom redosledu nailaska, dok kod dinamičkog rutiranja to nije slučaj. Zato se poruke moraju označavati brojevima. Dimamičko rutiranje je najteže za realizaciju ali je najkvalitetnije. Po pravilu svi računari imaju fiksnu ili statičku putanju ka jednom specijalnom računaru koji se naziva ruter i dinamički rutira poruke ka ostatku mreže. Ruter je ili poreban računar sa odgovarajućim softverom ili je specijalizovan hardverski uređaj, a mora imati barem dve mrežne konekcije - jednu za lokalnu mrežu, a drugu za ostatak sveta.

[3] Uvođenje mrežnih paketa

Poruke promenljive veličine se razbijaju u delove fiksne dužine koji se nazivaju paketi ili okviri (*frames*), a koji se mogu slati na pouzdan ili nepouzdan način.

[4] Povezivanje procesa na mreži

Dva procesa koja žele da komuniciraju po mreži mogu se povezati na više načina. Postoje tri osnovne šeme za uspostavljanje veze između procesa.

- Komutacija veze (*circuit switching*). Permanentni fizički link se uspostavlja između dva procesa koja žele da komuniciraju. Dok traje komunikaciona sesija, niko drugi ne može da koristi taj link (situacija je slična telefonskom razgovoru - dok par govornika ne oslobodi liniju niko drugi ne može da je koristi).

- Komutacija poruka (*message switching*). Privremeni fizički link se uspostavlja između dva procesa samo za vreme prenosa jedne poruke. Ovo liči na poštanski sanduče sa pismima - svaka poruka liči na pismo koje ima adresu pošiljaoca i adresu primaoca. Više procesa može korisiti link istovremeno.
- Komutacija paketa (*packet switching*). Poruke su promenljive dužine i razbijaju se u veći broj paketa, a svaki paket može da se šalje pojedinačno. Za to vreme se koristi fizički link, pod uslovom da paket sadrži i izvornu i odredišnu adresu. Paketi stižu u proizvoljnom redosledu nakon čega se od njih formira originalna poruka.

  Sva tri rešenja imaju svoje prednosti i mane. Prva šema okupira vezu sve dok traje sesija, ali je za tu sesiju premašenje, odnosno gubitak vremena, najmanje. Sledeće dve šeme su višeprocesno orijentisane, ali imaju veće vreme premašenje, jer se za svaku poruku, odnosno paket, traži novi link.

[5] Problem sudara na mreži

Postoje situacije kada više hostova žele da prebacuju podatke preko istog linka istovremeno. Tade će jedan host morati da čeka. Njegova poruka se odbacuje, a on se obaveštava o tome. Nakon toga host ponovo šalje poruku. Ponovljeni pokušaji slanja iste poruke degradiraju performase.

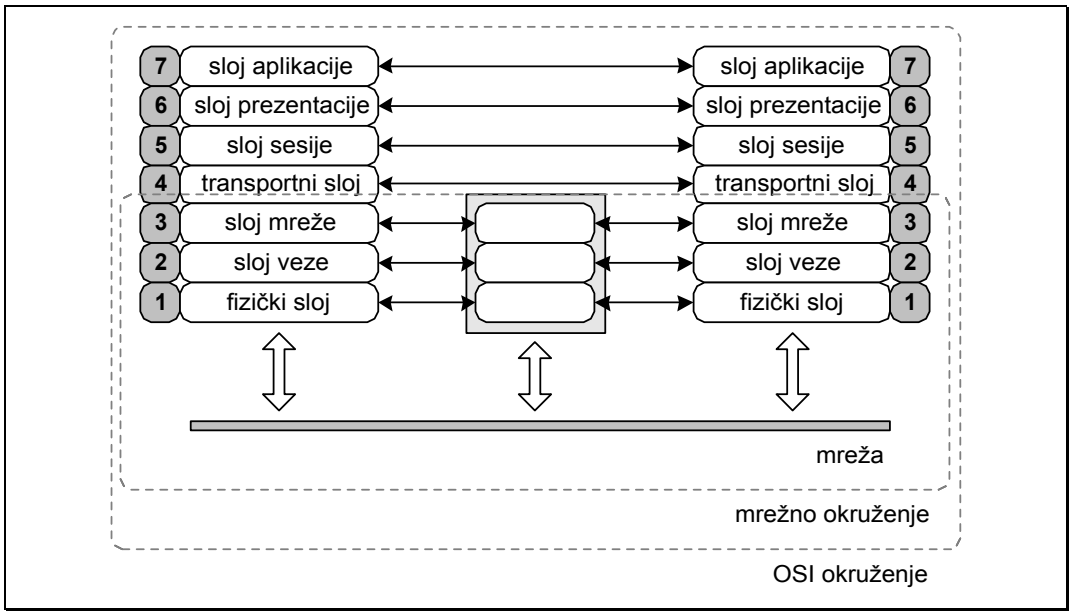
Višestruki pokušaji slanja poruke mogu se izbeći korišćenjem sledećih tehnika:

- Osluškivanjem medijuma i detekcijom kolizije (CSMA/CD). U ovom slučaju, pre slanja poruke, host mora da proveri da li se neka druga poruka šalje po linku preko signala nosioca. Ako je link slobodan, host šalje poruku, ako nije čeka. Ako se dva procesa pojave istovremeno na slobodnom linku, tada oba detektuju koliziju i oba se povlače sa linka, a posle nekog vremena oba će pokušati ponovo, ali verovatno u različito vreme. Loša strana ove tehnike je u tome što se na opterećenim mrežama može dogoditi veliki broj kolizija, što degradira performanse.
- Prosleđivanjem žetona (*token passing*). Jedinstvena poruka poznata pod nazivom žeton (*token*) kruži po mreži od računara do računara. Onaj računar koji dobije žeton ima pravo da koristi link. Ako računar ima poruku za slanje a dobije žeton, on će koristiti link, poslaće poruku i proslediti žeton dalje. Ako mu ne treba link, proslediće žeton odmah nakon primanja. Dobra strana ove tehnike je ravnopravnost hostova i održavanje performansi mreže na približno istom nivou, pri čemu će novi računar u mreži povećati vreme kruženja žetona.
- Prorezi za poruke (*Message slots*). Struktura koja sadrži određeni broj proreza za smeštanje poruka fiksne veličine kruži mrežom, odnosno u vidu poruke se prosleđuje svakom računaru. Svaki prorez može da primi poruku fiksne dužine, tako da hostovi moraju da razbiju svoje poruke na takve pakete koje će sadržati obe adrese (*source, destination*). Host koji ima poruku mora da sačeka da struktura stigne do njega i da ima u njoj bar jedan prazan prorez. Takođe host

tada proverava da li ima neka poruka za njega i ako ima poruke, uzima je i oslobađa slot.

Komunikacioni protokoli

Sva komunikacija u računarskim mrežama odvijaju se po određenim protokolima. Da bi se uprostila realizacija kvalitetnih mrežnih protokola, komunikacioni zadaci se dele na više nivoa (slojeva - *layers*), od kojih svaki nivo komunicira samo sa svojim ekvivaletnim nivoom, na drugoj strani veze. Svaki nivo ima svoj sopstveni protokol ili više logičkih segmenata, koji mogu biti realizovani hardverski ili softverski. Na slici 12.5. prikazan je ISO OSI model koji definiše sedam nivoa - slojeva.



Slika 12.5. *ISO model komunikacionih protokola*

[1] Fizički sloj (*Physical layer*)

Upravlja mehaničkim i električnim detaljima fizičkog prenosa podataka. Nivo je implementiran u hardveru mrežnih kartica.

[2] Sloj veze (*Data-link layer*)

Upravlja okvirima, odnosno delovima paketa fiksne dužine, uključujući detekciju i korekciju grešaka koje su se desile na fizičkom nivou.

[3] Mrežni sloj (*Network layer*)

Odgovoran je za konekciju i rutiranje paketa, uključujući upravljanje adresama odlazećih paketa, dekodovanjem adresa dolaznih paketa i podržavanje informacija o rutiranju. Ruteri rade na ovom sloju.

[4] Transportni nivo (*Transport layer*)

Odgovoran je za mrežne pristupe niskog nivoa i za prenose poruka između klijenata, uključujući podelu poruka na pakete, održavanje redosleda paketa i kontrolu toka.

[5] Nivo sesije (*Session layer*)

Realizuje sesije ili proces-proces komunikacione protokole.

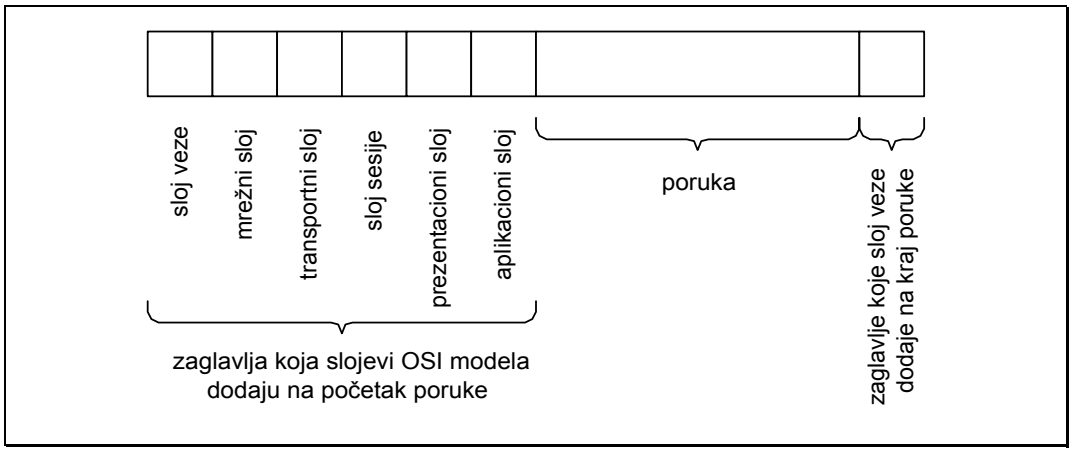
[6] Nivo prezentacije (*Presentation layer*)

Rešava problem razlika u formatu između različitih računara na mreži, uključujući konverziju karaktera.

[7] Nivo aplikacije (*Application layer*).

Interaguje direktno sa korisnicima i bavi se sa ftp, procedurama udaljenog prijavljivanja na sistem, elektronskom poštom itd.

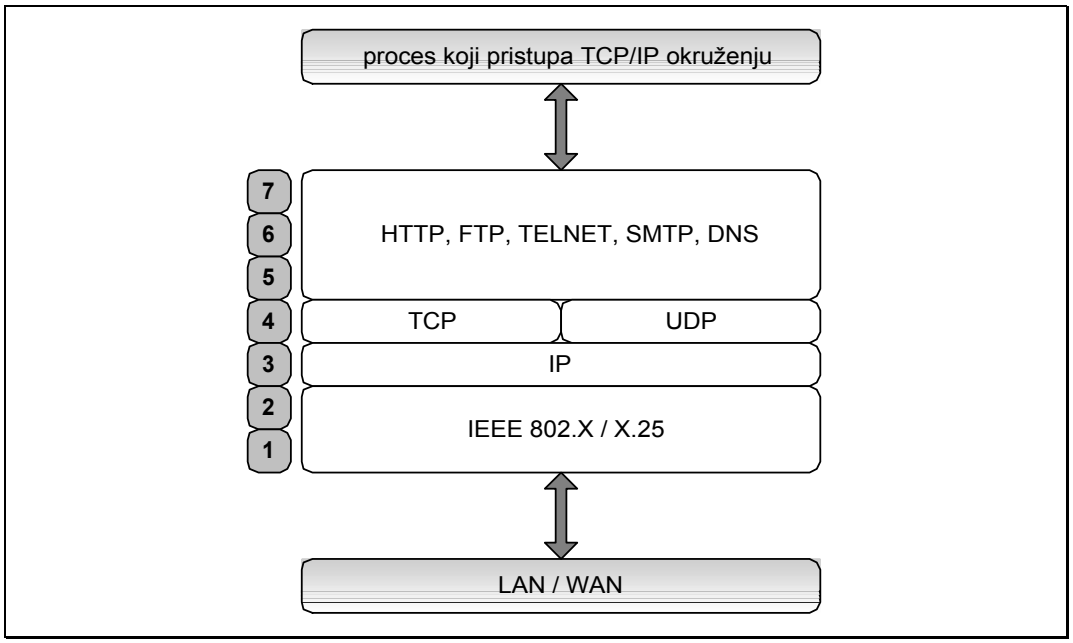
Svaki sloj logički komunicira sa ekvivaletnim nivoom na drugoj strani. Fizički, poruka putuje kroz slojeve počev od aplikacionog do fizičkog sloja (hardvera) a zatim po mreži dolazi do drugog sistema. Na drugom sistemu poruka se prenosi kroz slojeve, počev od fizičkog pa do aplikacionog. Tipična ISO poruka izgleda kao na slici 12.6.



Slika 12.6. *Izgled tipične ISO poruke*

Računari koji su povezani na Internet komuniciraju pomoću TCP/IP familije protokola. TCP/IP model čine sloj pristupa mreži, Internet sloj, transportni sloj i

aplikacioni sloj. TCP/IP model ne specificira sloj veze podataka i fizički sloj, već koristi različite protokole (SLIP, PPP) i tehnologije (*Ethernet*) na tom sloju. SLIP (*Serial Line Internet Protocol*) i PPP (*Point-to-Point Tunelling Protocol*) su adaptacije TCP/IP steka za računare koji su na mrežu povezani preko serijskog porta, odnosno modema.



Slika 12.7. *TCP/IP protokol stek*

Internet sloj odgovara sloju mreže u OSI modelu. Bavi se IP adresiranjem i rutiranjem paketa, čime obezbeđuje vezu između računara koji se ne moraju nalaziti na fizički istoj mreži. Na ovom sloju prisutan je IP (*Internet Protocol*) kao fundamentalni protokol koji obezbeđuje transfer informacija od računara do računara. Transportni sloj preuzima podatke sa višeg nivoa, po potrebi vrši segmentaciju podataka u datagrame ili uspostavljanje virtuelnih veza i prenosi podatke do destinacije koristeći mrežni sloj. Na transportnom sloju prisutni su TCP (*Transmission Control Protocol*) koji obezbeđuje pouzdanu vezu između dva procesa, otkriva i ispravlja greške i UDP (*User Datagram Protocol*) koji ne uspostavlja virtuelne veze niti obezbeđuje mehanizam za detekciju grešaka. Aplikacioni sloj omogućava aplikacijama, odnosno korisnicima da pristupe servisima Internet mreže.

Na aplikacionom sloju između ostalih prisutni su i sledeći protokoli:

- HTTP (*HyperText Transport Protocol*) - pristup Web stranicama,
- FTP (*File Transport Protocol*) - transfer datoteka,
- SMTP (*Simple Mail Transport Protocol*) - dolazeća pošta,

- POP3 (*Post Office Protocol*) - odlazeća pošta i
- DNS (*Domain Name System*) - razrešavanje imena u IP adrese.

TCP/IP model je prikazan na slici 12.7.

12.4. Detekcija greške i oporavak

U distribuiranim sistemima moguće su različite vrste hardverskih otkaza:

- otkaz sajta,
- otkaz linka i
- gubitak poruke.

Da bi sistem bio robustan, mora da ima mogućnost da detektuje bilo koji otkaz, da rekonfiguriše sistem tako da može da nastavi sa radom i da se oporavi ako se link ponovo uspostavi.

Kada je reč o detekciji greške, sistem funkcioniše na sledeći način: kada se greška dogodi, preko handshake procedure se ustanovi da li je otkazao link ili host. Pretpostavimo da sajt A i B imaju fizički link koji funkcioniše. U fiksnim intervalima oba sajta šalju poruke tipa <*I-am-up*>. Ako na primer sajt A ne primi ovu poruku u propisnom intervalu, onda se pretpostavlja da je ili sajt ili link otkazao ili da je poruka izgubljena. Ako i dalje nema poruke, tada sajt A šalje poruku <*Are-you-up*>. Ako i dalje nema odgovora, sajt A pošalje istu poruku preko druge putanje i ako mu se ovaj javi, tada je prva putanja otkazala. Ukoliko i preko druge putanje odgovora nema, tada se doazi do nepreciznog zaključaka:

- host B otkazao,
- prvi link otkazao,
- drugi link otkazao,
- poruka je izgubljena.

Dakle, jedan od ove četiri vrste otkaza se dogodio, ali se ne može ustanoviti tačno koji.

U slučaju da se dogodila greška i da je njena detekcija obavljena kao u opisanom slučaju, tada se mora otpočeti procedura za rekonfiguraciju sistema, na sledeći način:

- ako je direktni link između A i B otkazao, to mora da se objavi svim računarima u sistemu, kako bi se različite tabele rutiranja ažurirale,
- ako se pretpostavi da je host otkazao, svaki sajt u sistemu mora da dobije tu informaciju kako više ne bi pokušavao da ostvari konekciju sa tim sajtom,

- kada se link ili host oporave, moraju se ponovo vratiti u sistem. Link se unosi u tabele za rutiranje, a svi sajtovi se obaveštavaju da je prethodno otkazani sajt ponovo u fukciji.

12.5. Distribuirani sistemi datoteka

Distribuirani sistemi datoteka (*distributed file system* - DFS u daljem tekstu), predstavljaju distribuiranu realizaciju klasičnog sistema datoteka, u kome više korisnika dele datoteke i uređaje sekundarne memorije. Namena DFS je da omogući efikasno deljenje datoteka po mreži.

Da bi objasnili strukturu DFS, defnisaćemo termine kao što su servis, server i klijent.

- Servis je softverska celina koja se izvršava na jednom ili više računara i obezbeđuje određenu funkciju na mreži apriori nepoznatim korisnicima.
- Server je servis koji se izvršava na jednom računaru
- Klijent je proces koji poziva servis pomoću skup operacija (*primitiva*) koje formiraju takozvani klijentski interfejs.

Koristeći ovu terminologiju, kažemo da sistem datoteka obezbeđuje servis za datoteke klijentima. Klijent interfejs za servis datoteka sastoji se od skupa osnovnih operacija za datoteke, kao što su kreiranje datoteke, brisanje, čitanje, upis. DFS je sistem datoteka kod koga su servisi, klijenti i serveri raširenі svuda po mreži. Raspodela servisa i servera može biti veoma različita. Idealna kombinacija DFS jeste da bude totalno transparentan za korisnika. To znači da DFS korisniku izgleda kao klasični centralizovani sistem datoteka, a da korisnik ne zna gde su na mreži servisi, serveri i diskovi. To, takođe, znači da korisnik u DFS ne treba da posebno razlikuje lokalne datoteke sa svoje mašine od udaljenih datoteka.

Performansame DFS se mere ukupnim vremenom potrebnim da se zadovolji DFS zahtev. U konvencionalnim sistemima, to vreme je sastavljeno uglavnom od mehaničkih kašnjenja diska i prenosa podataka sa magnetnog medijuma uključujući veoma malo procesorskog vremena za obradu komande. U DFS pojavljuje se dodatno vreme koje ćemo nazivati kašnjenje po mreži. Ovo vreme uključuje vreme da se zahtev prosledi serveru, kao i vreme da server po mreži vrati željene podatke. Od DFS se očekuju solidne performanse - ne sme se dozvoliti da korisnik previše čeka na mreži.

Mapiranje datoteka i trasparentnost

Pod mapiranjem datoteka podrazumevaćemo mapiranje između između logičkih i fizičkih objekata. Korisnik se obraća datoteci po imenu, a sistem mapira tekstualno ime

u blokove na disku. U transparentnom DFS, mora postojati još jedan nivo apstrakcije: ime računara na kome se deljiva datoteka nalazi, koje zbog transparentnosti treba da se sakrije od korisnika. Uvešćemo dve osobine u pogledu mapiranja imena datoka kod DFS:

[1] Transparentnost lokacije

Transparentnost lokacije znači da ime datoteke ne otkriva ni jednu pojedinost oko njene fizičke lokacije.

[2] Nezavisnost lokacije

Nezavisnost lokacije znači da ime datoteke ostaje isto, odnosno da ne mora da se menja ako se promeni fizička lokacija datoteke.

Druga osobina zahteva dimaničko mapiranje datoteka, ali u praksi većina DFS koristi statičko mapiranje koje isključuje drugu osobinu. Prvu osobinu, uglavnom, poštuju svi. Obe osobine unose dosta kvaliteta u DFS.

Postoje tri šeme za mapiranje imena datoteka u DFS:

[1] kombinacija ime računara - ime datoteke *<host:filename>*

Ova šema je najprostija - ime datoteke se sastoji iz imena računara na kome se nalazi (*hostname*) i imena datoteke na tom računaru. Na taj način se kreira jedinstveno ime u DFS. Ova šema ne obezbeđuje ni transparentnost ni nezavisnost lokacije.

[2] NFS koncept

Po NFS konceptu (*network filesystem*), udaljeni direktorijumi se montiraju (*mount*) na lokalne direktorijume. Time se dobija stablo datoteka koje u potpunosti izgleda kao da se u celini nalazi na lokalnom računaru. Ovaj koncept obezbeđuje transparentnost, ali ne i nezavisnost lokacije.

[3] totalna integracija

Ovo je slučaj koji omogućava da sve datoteke u DFS imaju unikatna imena, a da se pri tome obezbede i transparentnost i nezavisnost lokacije. Ovu šemu nije lako realizovati, zato što distribuirani sistem obuhvata lokalne operativne sisteme koji su različiti i specifični (na primer UNIX ima veoma specijalne datoteke na direktorijumu /dev).

Pristup udaljenim datotekama, keširanje i replikacija

Razmatrajmo slučaj korisnika koji pristupa udaljenoj datoteci. Jedan od načina je mehanizam za udaljene servise (*remote-service mechanism*), u kome se zahtevi prosleđuju serveru, server čita svoju lokalnu datoteku i prebacuje je po mreži klijentu. Zahtevi se najčešće upućuju koristeći rpc pozive.

Da bi se ubrzao zahtev po mreži, koristi se keširanje koje je u DFS daleko složenije nego kod centalizovanih operativnih sistema. Kod centralizovanih sistema datoteka imamo jednu kopije datoteku i jedan keš za datoteke. Kod DFS postoji jedna glavna kopija datoteke koja se nalazi na serveru i keš za datoteke koji se nalazi na svakom računaru na mreži, uključujući i servere i klijente. Jedan ili više klijenata koji pristupaju datoteci na serveru dobijaju celu datoteku ili delove te datoteke u svoje lokalne keš memorije. Serverski keš takođe sadrži delove te datoteke, čime se obezbeđuju veće performanse.

Dakle, postojeći keš u DFS može da se nađe na tri lokacije:

- keš na serverima,
- keš na klijentima i
- DCD (*disk caching disk*) - tehnika keširanja diska drugim diskom.

DCD je tehnika koja nije prisutna na centralizovanim sistemima. Ideja je u sledećem: vreme potrebno da se podaci dobiju sa serverskog diska i mreže znatno je veće od vremena potrebnog da se isti podaci dobiju sa lokalnog diska. Sve što stigne sa mreže treba čuvati ili u lokalnom memorijskom kešu ili ako nema mesta, onda na lokalnom disku koji u ovom slučaju fukcioniše kao keš.

Ukoliko bi svi klijenti samo čitali datoteku, ne bi bilo skoro nikakvih problema u vezi sa keširanjem. Problem nastupa onda kada bilo ko od klijenata promeni nešto u datoteci. Promene se obavljaju u lokalnom kešu, tako da ostali korisnici tih promena nisu svesni. Ukoliko istu deljivu datoteku kešira više nezavisnih keš memorija, javlja se problem keš konzistencije, odnosno keš koherencije - svaka promena na bilo kojoj strani mora se reflektovati kod svih. DFS keširanje podseća na virtulenu memoriju, pri čemu se swap prostor ne nalazi na lokalnom disku nego na serveru.

Granularnost keša kod DFS varira od više blokova do celih datoteka. Koriste se različite tehnike za popunjavanje (*read-ahead*). Svaki keš u DFS je nezavistan po pitanju organizacije, popune i zamenjivanja. Međutim, po pitanju ciklusa upisa mora da zadovolji koherentnost keša.

Ciklus upisa je fundamentalni problem DFS keširanja. Već smo istakli da se kod centralizovanih keš memorija ciklus upisa rešava tehnikama *write through* (WT) i *write-back* (WB). Svaki klijent koji promeni datoteku u DFS, obavlja to prvo kroz svoj keš. Nakon toga se promene reflektuju u serverskom kešu, a na kraju na glavnu kopiju serverske datoteke (*master copy*). Najpouzdanija metoda je WT za oba keša (klijentski i serverski), ali su performase u tom slučaju najgore. *Write-back* tehnika ima odloženi upis i osetno poboljšava performanse, ali može biti veoma nepouzdana u smislu da promene nikada ne dostignu glavnu kopiju datoteke. Ako se, na primer, promene čuvaju na klijentskoj strani, može doći do otkaza klijenta, do otkaza linka ili do otkaza servera. Kombinacija *write-through* tehnike na klijentskoj i *write-back* na serverskoj strani predstavlja dobro rešenje.

Postoje brojne modifikacije *write-back* tehnike za slučaj DFS. Jedna alternativa je prebacivanje promena na server u trenutku kada taj blok mora da se zbog zamene izbaci iz keša na strani klijenta. Međutim, u toj alternativi, promene mogu ostati veoma dugo u kešu na strani klijenta. Tada se radi *flushing* - u defnisanim vremenskim intervalima proverava se lokalni keš i sve što je promenjeno šalje se na server. Reakcija serverskog keša je tada obično u *write-through* tehnici - glavna kopija datoteke se odmah ažurira. Posebno je osetljivo ažuriranje područja sa meta-podacima. Na primer, NFS za datoteke koristi *flushing* tehiku, a za meta-podatke sinhroni upis, koji svaku promenu prosleđuje na disk bez odlaganja.

Još jedna modifikacija *write-back* tehnike koja se koristi u AFS sistemima je upis na zatvaranje (*write-on-close*). Sve promene u datoteci se šalju na server tek kada klijent zatvori datoteku. Ovaj metod je dobar za u slučaju datoteka koje se otvaraju na relativno duži vremenski period, i koje se pre konačnog zatvaranja često modifikuju. Ako se većina datoteka otvara na kratko, modifikuje jedanput i odmah zatvori, ova tehnike nije od veće koristi.

Već smo naglasili problem keš konzistencije ili keš koherencije. Svaki klijent povremeno se obraća serveru radi provere validnosti podataka u svom kešu. Ukoliko su podaci u kešu validni, klijent nastavlja da ih koristi. Ako podaci nisu validni (neko od klijenata je promenio deo datoteke), klijent od servera traži poslednju verziju podataka. Postoje dve tehnike pomoću kojih klijent može proveriti validnost svojih podataka u kešu:

[1] Provera koju inicira klijent (*client-initiated aproach*)

Klijent inicira proveru validnosti podataka, odnosno konzistentnost sa glavnom kopijom datoteke. Provera može da se obavlja pre svakog pristupa, samo na početku ili u nekim periodičnim regularnim intervalima. Provera je čist gubitak vremena i utiče negativno na performanse.

[2] Provera koju inicira server (*server-initiated aproach*)

U ovoj tehnici server mora voditi strogu evidenciju o svakom klijentu. Server prati koji je klijent uzeo koji deo datoteke na čitanje, i koji je deo datoteke promenjen. Kada server detektuje konflikt, odnosno kada jedan klijent promeni deo datoteke, server obaveštava sve klijente koji koriste taj isti deo datoteke da preuzmu poslednju verziju podataka.

Ako uporedimo upotrebu keša u DFS i udaljenih servisa (*remote service*) koji isključuju keš, dolazimo do sledećih zaključaka: udaljeni servisi bez keša su prosti za realizaciju i veoma pouzdani, ali imaju niske performanse. Keš u DFS je relativno komplikovan za realizaciju, ali lokalni keš i keš na serveru kao i kombinacije keša na samom disku (*cachefs*) osetno smanjuju upotrebu diskova i mrežni saobraćaj. Naravno, problem keš konzistencije je uvek prisutan. Takođe, keš menja raspodelu broja i veličine zahteva po mreži u odnosu na udaljene servise. Kod keša imamo manji broj većih zahteva, a kod udaljenih transfera imamo veći broja manjih zahteva po mreži. Kako su mali transferi manje efikasni nego veliki transferi, prednost je na strani keša.

Keš je osetno bolji u slučajevima malog broja upisa (*infrequent writes*). U slučaju velikog broja upisa, performase opadaju zbog rešavanja problema keš koherencije. Udaljeni servisi se moraju koristiti na računarima bez diskova i na računarima sa malom količinom memorije.

Po pitanju ponašanja servera, svi servisi se dele na servise punog stanja (*stateful service*) i na servise bez stanja (*stateless service*). Ako server vodi evidenciju o svim klijentskim aktivnostima za neki servis, to je onda *stateful* servis. Na primer, servis datoteka je *stateful* ako server prati informacije o svim datotekama koje klijenti koriste. Ukoliko server ne prati informacije za neki servis, taj servis je bez stanja, što znači da se svaki zahtev obrađuje posebno i ne koriste se neki prethodni rezultati. *Stateful* server ima bolje performanse, ali *stateless* server bolje reaguje na otkaz u mreži. Na primer, ako server otkaže pa se regeneriše, *stateless* servis će se lakše oporaviti, zato što je svaki zahtev praktično nezavistan i opslužuje se iz početka. *Statefull* servis se oslanja na prethodne zahteve i rezultate i zato se teže oporavlja. Neki servisi, kao što je DFS u kom proveru validnosti inicira server, moraju bito *statefull*.

Replikacija datoteka se često koristi u DFS, a podrazumeva kreiranje više kopija iste datoteke i njihovo smeštanje na različite računare na mreži. Replikacija unosi dve velike prednosti: povećava pouzdanost i performanse pristupa čitanja, jer se uvek bira najbliža replika klijentu. Lokacije replika se skrivaju od korisnika, koji najčešće i nije svestan njihovog postojanja.

Glavni problem vezan za replike je ažuriranje: svaka promena mora da se upiše u sve replike, a to je vremenski intenzivan proces. Jedan od načina kojim se problem delimično može rešiti jeste uvođenje primarne replike za klijenta. Nakon izmene datoteke najpre se ažurira primarna replika, dok ostale dobijaju poruke da nisu više validne i da se moraju ažurirati najkasnije do sledećeg korišćenja.

13

Sinhronizacija procesa u distribuiranim sistemima

Sadržaj poglavlja:

13.1. Distribuirana sinhronizacija procesa
13.2. Atomske transakcije u centralizovanim uslovima
13.3. Atomske transakcije u distrubuiranim uslovima
13.4. Upravljanje zastojima u distribuiranim uslovima

13.1. Distribuirana sinhronizacija procesa

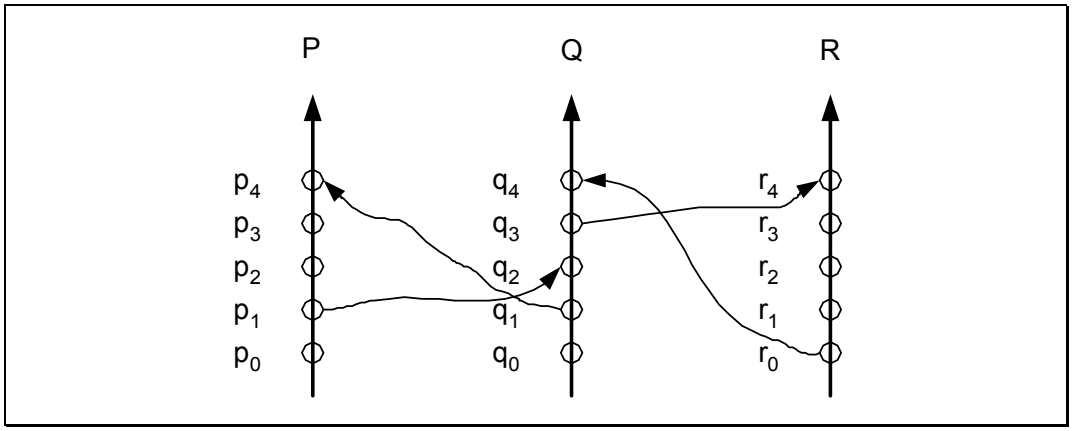
U prethodnim poglavljima opisana je sihronizacija procesa i problem zastoja za centralizovanu okolinu. Pod centralizovanom okolinom podrazumeva se jedan ili više čvrsto povezanih procesora, koji koriste istu memoriju i isti sistemski časovnik. U ovom poglavlju proširićemo problem sihronizacije procesa i zastoja na distribuiranu okolinu.

U distribuiranim sistemima na različitim računarima postoje različiti časovnici, što dovodi do prvog problema. Dva procesa koja se izvršavaju na različitim računarima ne mogu da odrede tačan redosled događaja, a veliki broj aplikacija zahteva da se poštuje poredak događaja. U distrbuiranim sistemima, svaki časovnik može da pokazuje različito vreme tako da nemamo jedinstveno vreme sistema. Umesto informacije o događaju koji se dogodio prvi, u distribuiranim sistemima uvodi se relacija "*happened-before*", što znači "dogodilo se pre". Ova relacija pruža samo delimičnu informaciju o

redosledu događaja. Ovu relaciju definisaćemo najpre na jednom procesu, a potom ćemo je proširiti na više procesa. Proces se izvršava sekvencijalno, uz poštovanje jasnih pravila vezanih za neke događaje. Na primer, poruka ne može da se primi a da je neko pre toga ne pošalje. Definišimo relaciju dogodilo se ranije (*happened before*):

- Relaciju *happened-before* označićemo sa ->.
- Ako su A i B događaji istog procesa i ako je događaj A izvršen pre događaja B, tada važi A -> B.
- Ako je događaj A slanje poruke, a događaj B primanje te poruke, pri čemu različiti procesi šalju i primaju poruku, tada je A -> B.
- Ako je A -> B i B ->C, tada je A ->C (implikacija).
- Ako dva događaja A i B nisu povezana relacijom *happened before*, kaže se da se oni izvršavaju konkurentno, ali ako je A ->B, tada se se A mora izvršiti pre B.

Uzmimo primer sa slike 13.1.



Slika 13.1. *Relacija happened before*

Događaji p1->q2, r0->q4, q3->r4, itd. su povezani relacijom happened before, p1->r4 je implikacija (p1->q2 i q2->r4), a neki događaji se odvijaju konkurentno (q0, p2), (r0, q3), (r0, p3). U slučaju konkurentnih događaja ne znamo koji se dogodio prvi.

Redosled događaja mora se odrediti bez fizičkih časovnika. Zato ćemo svakom fizičkom događaju dodeliti vremensku oznaku TS (*timestamp*) i definisaćemo relaciju *hapended before* za svaki par događaja A i B na sledeći način: ako je A->B, tada je TS događaja A manji od TS događaja B, tj. TS(A)<TS(B).

Vrednosti *timestamp* promenljive mogu se dodeliti procesima na osnovu logičkog časovnika - LC (*logical clock*). Za svaki proces Pi definiše se LC_i kao brojač događaja. Brojač događaja je nenegativna vrednost, monotono rastuća. Ako se događaj A desi pre događaja B, tada je $LC_i(A)<LC_i(B)$. Problem se javlja u slučaju dva procesa

na dve različite mašine, sa različitim procesorima i različitim taktovima. Proces P1 šalje poruku procesu P2 na drugoj mašini (događaj A), a proces P2 prima poruku (događaj B). Neka je $LC_1(A) = 200$ i $LC_2(B)=195$.

Situacija je moguća, ali ovo nije saglasno sa onim što smo hteli. Ako je A->B, tada $LC_1(A)$ mora biti manji od $LC_2(B)$. Problem se rešava uvođenjem korekcije LC vrednosti procesa primaoca poruke u slučaju ovakve nesaglasnosti. Proces primalac koriguje svoju LC vrednost tako što uzima LC vrednost procesa pošiljaoca uvećanu za 1. U konkretnom slučaju $LC_2(B)$ mora da se koriguje i da postane $LC_2(B)=201$.

Međusobno isključenje

U nastavku teksta objasnićemo moguće načine realizacije mehanizma međusobnog isključenja u distribuiranim uslovima. Pretpostavimo da se sistem sastoji od n procesa i da se svaki izvršava na različitom procesoru.

[1] Centralizovan pristup

Jedan od procesa u sistemu je određen da odobrava ulaz u kritičnu sekciju i naziva se koordinator. Svaki proces koji želi da uđe u kritičnu sekciju obraća se koordinatoru sa zahtevom (poruka *request*). Koordinator će analizirati da li je neki proces već u kritičnoj sekciji. Ako jeste, proces ide u red čekanja, a kada proces koji je bio u kritičnoj sekciji izađe, koordinator odabira jedan od procesa iz reda (obično po FIFO algoritmu). Samo onaj proces koji dobije od koordinatora odobrenje (poruka *reply*), može ući u svoju kritičnu sekciju. Posle izlaska iz kritične sekcije, proces šalje koordinatoru poruku *release*, što znači da je izašao iz kritične sekcije.

U slučaju otkaza, proces kordinator se mora zameniti drugim, što se obavlja posebnim algortimom za izbor koordinatora (*election*). Novi koordinator mora prozvati sve procese u sistemu da bi rekonstruisao listu čekanja.

[2] Puni distribuirani pristup

Rešenje izbegava centralizaciju i funkcioniše na sledeći način: kada proces Pi želi da uđe u svoju kritičnu sekciju, on generiše novi *timestamp*, TS, i šalje poruku *request* (Pi, TS) svim ostalim procesima u sistemu. Pošto u sistemu ima n procesa, proces Pi šalje 2(n –1) poruka, odnosno zahteva za ulaz u kritičnu sekciju. Poruka sadrži identifikaciju procesa Pi i njegov TS.

Proces Pj nakon primanja poruke *request* postupa na dva načina: odlaže slanje poruke *reply* za kasnije, ukoliko se nalazi u kritičnoj sekciji, odnosno odgovara trenutno porukom *reply*, ako nije u kritičnoj sekciji i nema nameru da uđe u nju. Ako proces Pj ima nameru da uđe u kritičnu sekciju, uporediće TS sa procesom koji traži dozvolu. Ako je TS(Pj)>TS(Pi), poslaće *reply* poruku trenutno, a u protivnom se slanje *reply* poruke odlaže.

Proces Pi može ući u svoju kritičnu sekciju tek kad primi *reply* poruke od svih drugih procesa u sistemu. Kada napusti svoju kritičnu sekciju, proces šalje *reply* poruku svim procesima koji su mu se obratili.

Dobre osobine ovog načina pristupa su u tome što nema zastoja i nema zakucavanja, jer se procesi sinhronišu na bazi TS i po FIFO principu.

Međutim, šema ima i neke loše osobine. Proces mora da poznaje sve procese u sistemu i da prati dodavanje i ukljanjanje procesa iz sistema. Ako bilo koji proces otkaže, cela šema puca. Da bi se izbegao kolaps, mora se pratiti stanje svih procesa u sistemu. Izvodi se zaključak da je ovaj protokol pogodan samo za male sisteme, sa malim brojem procesa.

[3] Slanje žetona (*token passing*)

Ovo rešenje se zasniva na specijalnoj poruci, žetonu (*token*), koji kruži između svih procesa u sistemu. Procesi obrazuju logički krug. Proces koji je dobio žeton, ima pravo da uđe u svoju kritičnu sekciju. Ako proces ne želi da uđe u svoju kritičnu sekciju, prosleđuje ga dalje.

Ako je kruženje žetona jednosmerno, nema zakucavanja. Ovo rešenje karakterišu dva moguća problema: uništenje žetona kao poruke (tada se generiše novi žeton) i otkaz nekog procesa (nakon toga se mora formirati novi logički krug procesa).

13.2. Atomske transakcije u centralizovanim uslovima

Međusobno isključenje omogućava da se kritične sekcije izvršavaju na atomski način, odnosno nedeljivo.

Pojam transakcije

Kolekciju instrukcija koje obavljaju jednu logičku funkciju zvaćemo transakcija. Transakcija obuhvata čitanje ili upis podataka, a završava se sa dve moguće operacije: *commit*, koja označava da je transakcija obavila svoje izvršavanje uspešno i *abort*, koja označava da je transakcija završila svoje izvršavanje neuspešno, zbog neke greške. Od transakcija se traži da se obavljaju atomski (transakcija se ili izvršava ili se ne izvršava). To znači da stanje sistema u slučaju greške ili prekida transakcije mora biti svedeno na stanje sistema pre izvršavanja transakcije. To se naziva povratak unazad (vraćanje trake unazad - *rollback*).

U opštem slučaju, na računarskim sistemima razlikuju se tri vrste memorija: memorija koja gubi sadržaj bez napajanja (*volatile storage*), kao što je RAM, memorija koja ne gubi sadržaj kad ostane bez napajanja (*non-volatile storage*), kao što su diskovi, i stabilna memorija (*stable storage*), koja nikada ne gubi sadržaj (RAID, *cluster*, NV-

RAM). Da bi se obezbedila atomska priroda transakcije mora se voditi evidencija o svim koracima upisa. To se obavlja preko *log* strukture (dnevnika) smeštene na stabilnoj memoriji. Ova metoda se naziva *write-ahead logging*. Svaki zapis u dnevniku opisuje jednu operaciju upisa u transakciji i sadrži sledeća polja:

- ime transakcije,
- ime polja za upis,
- stare podatke koji predstavljaju vrednost podataka pre operacije upisa,
- nove podaci koji predstavljaju vrednost nakon operacije upisa.

Pre nego što transakcija Ti počne izvršavanje u dnevnik se upiše zapis <Ti starts>. U toku svake transakcije, pre svake operacije upisa, mora se upisati jedan zapis u dnevnik. Poslednji zapis u dnevniku je <Ti commit>. Transakcija Ti izvršena na ovaj način može se poništiti, odnosno stanje se može vratiti u početno stanje ili u neko međustanje. Uvode se dve operacije za rad sa transakcijama koje koriste dnevnik:

- *undo* (Ti), koja vraća sve vrednosti koje je promenila transakcija na stare vrednosti,
- *redo* (Ti), koja postavlja sve vrednosti koje je transakcija promenila na nove vrednosti.

Ako se dogodi otkaz sistema zbog havarije, *log* tehika omogućava dve situacije:

- ako ne postoji zapis <Ti commit>, što znači da se transakcija Ti nije završila, izvršava se operacija *undo* (Ti),
- ako postoji zapis <Ti commit>, što znači da se transakcija Ti završila i da je treba imati u oporavljenom sistemu, izvršava se operacija *redo* .(Ti)

Da bi se ova pojava učinila još efikasnijim u slučaju otkaza i regeneracije sistema, u dnevnik se uvodi oznaka *checkpoint*. *Checkpoint* (kontrolna tačka) određuje sve transakcije završene na stabilnom medijumu. Sve što se u dnevniku nalazi pre kontrolne tačke je u redu, a sve što se nalazi iza nje treba obraditi na sledeći način:

- operacija *redo* (Tk) izvršava se za svaku transakciju Tk koja sadrži zapis <Tk commit>,
- operacija *undo* (Tk) izvršava se za svaku transakciju Tk koja ne sadrži zapis <Tk commit>.

Konkurentne atomske transakcije

Svaka transakcija mora da bude atomska. Transakcije se izvršavaju konkurentno sa svojim procesima, što znači da se mora obezbediti pravilan redosled izvršenja. Pravilan redosled izvršenja može se obezbediti izvršavanjem svake transakcije u celosti unutar kritične sekcije. Ovaj način izvršavanja je isuviše restriktivan i povlači strogo

serijsko, nekonkurentno raspoređivanje transakcija. Serijsko raspoređivanje ne može dovesti do konfliktnih odnosa, ali smanjuje performanse. Zato se uvode specijalne tehnike kojima se reguliše konkurentnost transakcija i sprečavaju konflikti.

Konkurentne operacije u transakcijama mogu biti u konfliktnom ili ne konfliktnom odnosu. Konkurentne operacije su u konfliktnom odnosu ukoliko se odnose na isti zapis, pri čemu je bar jedna operacija upis. Ukoliko transakcije preklapaju svoja izvršenja, performanse se povećavaju, ali su mogući konfliktni odnosi između operacija. Konflikti se moraju sprečiti - konfliktne operacije ne smeju se izvršavati istovremeno.

Transakcije se mogu preklapati samo ako se izbegnu konflikti između operacija, što se postiže na dva načina.

[1] Protokol za zaključavanje (*lock protocol*)

Transakcija Ti pre pristupa zapisu Q mora da zatraži odgovarajuće pravo zaključavanja. Transakcija pristupa zapisu isključivo nakon zaključavanja. Zapis se može zaključati na dva načina:

- deljivo (*shared lock*) - transakcija koja dobije pravo na deljivo zaključavanje zapisa Q dobija pravo čitanja zapisa Q, ali ne i pravo upisa zapisa Q,
- ekskluzivno (*exclusive lock*) - transakcija koja dobije pravo na ekskluzivno zaključavanje zapisa Q taj zapis može čitati i upisati.

[2] Poredak izvršavanja na bazi vremenske oznake (TS poredak)

Sinhronizacija između procesa može se ostvariti pomoću protokola zasnovanog na vremenskim oznakama. Svakoj transakciji dodeljujemo vremensku oznaku TS kao vreme kada transakcija počinje da se izvršava. Takođe, svakom zapisu Q, dodeljujemo dva vremenska parametra:

- W-timestamp (Q), koji predstavlja vreme poslednje transakcije koja je uspešno obavila upis u Q
- R-timestamp (Q), koji prestavlja vreme poslednje transakcije koja je uspešno obavila čitanje iz Q

Ova dva parmetara se stalno ažuziraju. TS protokol obezbeđuje da konfliktno čitanje i upis obave u TS poretku, koji isključuje preklapanje konflikta. Pretpostavimo da transakcija Ti pošalje zahtev read(Q). Moguće su dve situacije:

- ako je TS(Ti) ≥ W-timestamp(Q), tada je zahtev korektan, čitanje se obavlja, a nakon toga se ažurira R-timestamp(Q) ,
- ako je TS(Ti) < W-timestamp(Q), tada transakcija traži vrednost Q iz prošlosti. Ta vrednost je prepisana, tako da se čitanje odbacuje, a vrednost se može dobiti primenom metode *rollback* na transakciju Ti.

Pretpostavimo da je transakcija Ti pošalje zahtev write(Q). Moguće su tri situacije:

- ako je TS(Ti) < R-timestamp(Q), zahtev je nekorektan, jer transakcija pokušava da upiše nešto što je već trebalo da bude pročitano. Upis se odbacuje, a na transakciju Ti se primenjuje metoda *rollback*,
- ako je TS(Ti) < W-timestamp(Q), transakcija pokušava da upiše staru vrednost za Q. Upis se odbacuje, a na transakciju Ti se primenjuje metoda *rollback*,
- u svi ostalim slučajevima upis se obavlja.

Transakcija za koju se obavlja *rollback* dobija novu TS vrednost i restartuje se.

13.3. Atomske transakcije u distrubuiranim uslovima

Kao što je već rečeno, atomarna ili nedeljiva operacija se izvršava u celosti ili se ne izvršava uopšte. Atomarna operacija ne može se prekinuti, a zatim kasnije nastaviti. Mehanizmi koji obezbeđuju izvršenje atomarnih operacija u distrubuiranim sistemima složeniji su od mehanizama koji tu funkciju obavljaju u centralizovanim okruženjima. U distribuiranim sistemima u izvršenju jedne transakcije najčešće učestvuje više računara, odnosno sajtova. Transakcija se ili izvršava na svim sajtovima u celosti ili se izvršenje transakcije prekida na svim sajtovima. Otkaz bilo kog sajta ili komunikacionog linka izaziva blokadu atomske transakcije.

Izvršavanje atomske transakcije omogućava se uvođenjem koordinatora transakcija (*transaction coordinator*) na svakom računaru. Koordinator transakcija kontroliše izvršavanje svih transakcija koje je taj računar inicirao, odnosno:

- pokreće izvršavanje transakcije,
- razbija jednu transakciju u određeni broj podtransakcija i distribuira te podtransakcije ostalim sajtovima na izvršavanje,
- koordiniše izvršavanje transakcije, i u slučaju otkaza prekida izvršenje transakcije na svim sajtovima.

2PC protokol (two-phase commit protocol)

Da bi se obezbedila nedeljivost transakcije T, svi sajtovi na kojima ona treba da se izvršava moraju da prihvate udeo u njenom izvršavanju. Transakcija T će se izvršavati samo ako se svi sajtovi slože, a u protivnom se izvršenje prekida.

Da bi obavio transakciju T, koordinator najpre mora obaviti protokol izvršavanja (*commit protocol*). Najčešće se koristi varijanta ovog protokola (2PC, *two-phase*

commit protocol), koja je na osnovu izvršavanja u dve faze dobila ime. Definisaćemo 2PC za transakciju T, koju inicira sajt Si, sa transakcionim koordinatorom Ci.

[1] Faza 1.

U ovoj fazi koordinator transakcije šalje svim sajtovima od interesa poruku da želi da sa njima obavi transakciju T. Ci dodaje zapis <prepare T> u dnevnik i, po pravilu, ga upisuje na disk. Zatim Ci šalje poruku *prepare* (T) svim sajtovima koji treba da izvrše T. Sajt Sj koji dobije ovu poruku odlučuje da li će prihvatiti izvršavanje transakcije T na svojoj strani. Ako je odgovor ne, koordinator Cj sajta Sj upisuje u svoj dnevnik <no T> i šalje poruku *abort* (T) koordinatoru Ci. Ako je odgovor da, tada koordinator Cj upisuje u svoj dnevnik <ready T> i šalje poruku *ready* (T) koordinatoru Ci.

[2] Faza 2.

U ovoj fazi koordinator Ci na osnovu odgovora analizira da li su svi sajtovi voljni da rade transakciju T. Samo pod uslovom da su svi sajtovi koordinatoru Ci odgovorili sa *ready* (T), transakcija će otpočeti izvršenje. Koordinator Ci će u svoj dnevnik upisati poruku <commit T>, a zatim će svim sajtovima poslati poruku *commit* (T). Ova poruka je signal sajtovima da otpočnu sa izvršenjem odgovarajuće podtransakcije. Ako u toku izvršenja bilo koji sajt koji ima udela u izvršenju koordinatoru Ci pošalje poruku *abort* (T), transakcija se prekida. Posle prekida, koordinator Ci će u svoj dnevnik upisati poruku <abort T>, a zatim će poslati svima poruku *abort* (T), koja znači da se T prekida.

2PC protokol se završava kada svi sajtovi obave svoje aktivnosti, odnosno kada svaki sajt koordinatoru transakcije pošalje poruku *acknowledge* (T). Kada koordinator primi ovu poruku od svih sajtova, to znači da je transakcija T završena. Koordinator u svoj dnevnik upisuje zapis <complete T>.

Analizirajmo šta se dešava sa 2PC protokolom u slučaju otkaza. Najopasnija situacija je otkaz koordinatora - u tom slučaju može doći do zastoja.

[1] Otkaz sajta Sk koji učestvuje u transakciji T.

Pretpostavimo da je otkazao sajt Sk koji ima udeo u izvršenju transakcije T. Posle oporavka, sajt Sk mora da ispita svoj dnevnik. Moguće situacije su:

- dnevnik sadrži zapis <commit T>, što znači da je počeo da izvršava operaciju, pa je otkazao. Operacija se mora ponovo izvršiti - sajt izvršava *redo* (T),
- dnevnik sadrži zapis <abort T>. U tom slučaju operacija se prekida - sajt izvršava poništavnanje *undo* (T),
- dnevnik sadrži zapis <ready T>. U tom slučaju sajt mora da konsultuje glavnog koordinatora Ci i da ispita u kojoj se fazi nalazi transakcija T (zato što je pristao na saradnju a onda otkazao). Ukoliko je aktivan, koordinator Ci će obavestiti sajt Sk da li je transakcija T u izvršenju ili je prekinuta. Shodno s tim, sajt Sk obavlja *redo* (T), odnosno *undo* (T). Ukoliko glavni koordinator nije aktivan, sajt Sk

mora da pošalje upit *query-status* (T) svim sajtovima u mreži i da od njih dobije informaciju o stanju transakcije T. Posle toga sajt Sk obavlja *redo* (T), odnosno *undo* (T),

- zapis u dnevniku nema nijednu kontrolnu informaciju o transakciji T (abort, ready, commit). To znači da je Sk dobio poruku da se pripremi (prepare T) i da je pre odgovora otkazao. Zbog otkaza sajta Sk obustavljena je cela transakcija, tako da Sk obavlja *undo* (T).

[2] Otkaz glavnog koordinatora

U slučaju da koordinator otkaže u toku izvršavanja transakcije T sajtovi saučesnici moraju sami da odluče o sudbini te transakcije. U mnogim slučajevima, sajtovi koji učestvuju ne mogu sami da odluče da li da nastave izvršavanje ili da prekinu operaciju T, tako da moraju da čekaju da se oporavi koordinator.

Sajt koji više ne komunikacira sa koordinatorom transakcije ispituje svoj dnevnik i u skladu sa sadržajem dnevnika obavlja određenu aktivnost.

- Ako dnevnik sadrži zapis <commit T>, operacija se nastavlja.
- Ako dnevnik sadrži zapis <abort T>, operacija se prekida.
- Ako dnevnik sadrži zapis <ready T>, sajt je poslao poruku koordinatoru Ci da je spreman, ali nakon toga nije dobio informaciju da počne izvršavanje. Sajt Sk ovu situaciju može razrešiti na dva načina: čekanjem na oporavak glavnog koordinatora ili prekidom izvršenja transakcije T.

  Ukoliko dnevnici svih sajtova učesnika sadrže zapise <ready T>, čeka se oporavak glavnog koordinatora radi nastavka izvršenja transakcije. U ovom slučaju, prisutan je problem blokiranja transakcije T čekanjem na oporavak koordinatora. Ukoliko bar jedan od dnevnika sajtova učesnika ne sadrži zapis <ready T>, transakcija se može obustaviti.

[3] Otkaz mreže.

U slučaju otkaza mreže ili linka, koordinator pokušava da nađe alternativni link. Ukoliko ga nađe, transakcija T se nastavlja. U protivnom, slučaj se svodi na otkaz sajta Sk. Ako se link oproravi, situacija se razrešava kao da je sajt Sk otkazao, a u protivnom se cela transakcija prekida.

Konkurentne atomske transakcije u distribuiranim uslovima

Mnogi mehanizmi koje smo definisali za sinhronizaciju procesa moraju se prilagoditi ili redefinisati za distrubiranu okolinu. Svaki sajt mora da ima sopstveni mehanizam za upravljanje transakcijama koji pored sinhronizacije obavezno vodi i

dnevnik koji pomaže u slučaju opravaka od otkaza. Kao i u slučaju centralizovane okoline, definisaćemo protokole za zaključavanje i timestamp metodu.

Protokoli za zaključavanje

Obradičemo prvo slučaj bez replikacije datoteka. Svaki sajt ima sopstveni mehanizam za zaključavanje i otključavanje podataka (*lock manager*). Transakcija koja želi da zaključa zapis Q na sajtu Si šalje poruku o zaključavanju - *lock* (Si, Q). Ukoliko *lock manager* sajta Si ima uslove da zaključa Q, poslaće transakciji poruku da joj je zahtev ispunjen, a u protivnom transakcija mora da čeka. Posle toga transakcija odradi svoje, pa slanjem poruke *unlock* (Si, Q) otključa zapis Q. Ova šema je jednostavna za impementaciju, ali je rukovanje zastojima kompleksno.

U slučaju da postoji više replika datoteke u sistemu, protokoli za zaključavanje se mogu realizovati na više načina.

[1] Protokol za zaključavanje sa jednim koordinatorom

U celom distrubuiranom sistemu postoji samo jedan *lock manager* koji se nalazi na izabranom sajtu Sj. Bilo koji drugi sajt, koji želi da zaključa datoteku mora da se obrati glavnom koordirnatoru sa odgovarajućim *lock* zahtevom. Nakon toga će *lock manager* odlučiti da li će mu dozvolti da zaključa datoteku ili će morati da čeka.

Protokol je jednostavan za implementaciju, a i rukovanje zastojima je jednostavno, zato što se sve dešava na jednom mestu. Loše osobine ovog protokola su mogućnost pojave uskog grla u sistemu (svi sajtovi se obraćaju samo jednom) i ranjivost sistema (ako sajt na kome je koordinator otkaze, distribuirani sistem ostaje bez mehanizma za sinhronizaciju procesa).

Oba nedostatka se otklanjaju uvođenjem većeg broja *lock managera* na različitim računarima, koji kontrolišu pristup različitim podacima (protokol za zaključavanje sa više koordinatora). Šema smanjuje mogućnost pojave uskog grla i mogućnost kolapsa sistema usled otkaza, ali je mehanizam za upravljanje zastojima složeniji.

[2] Prokotol većine (*majority protocol*)

Na svakom sajtu postoji *lock manager* koji kontroliše lokalne datoteke, podatke u njima i njihove replike. Transakcija koja želi da zaključa zapis Q koji je repliciran na n različitih sajtova mora da pošaje *lock* zahtev za više od n/2 sajtova. Svaki sajt odlučuje hoće li da ispuni *lock* zahtev, a transakcija će dobiti dozvolu ukoliko većina sajtova pošalje pozitivan odgovor. Dobra osobina ovog protokola je ta što se poštuje većina a izbegava centralizovanu kontrola. Međutim, šema je komplikovana je za relazaciju, a i zastoji se teško kontorlišu.

[3] Pristrasni protokol (*biased protocol*)

Ovaj metod je sličan većinskom protokolu, s tim što zahtevi za deljivim zaključavanjem imaju prednost u odnosu na zahteve za ekskluzivnim zaključavanjem. I ovde postoji *lock manager* za svaki sajt, ali se deljivi i ekskluzivni zahtevi opslužiju drugačije.

Deljivi zahtevi za zaključavanjem zapisa Q (operacija čitanja), upućuju se samo jednom sajtu koji sadrži repliku zapisa Q. Ekskluzivni zahtevi za zaključavanjem zapisa Q (operacije čitanja i upisa) upućuju se svim sajtovima koji sadrži repliku zapisa Q.

Pristrasni protokol ima manje kašnjenje kod operacija čitanja u odnosu na većinski protokol, jer su zathevi deljivi. Međutim, prilikom upisa, transakcija mora da se obrati svakom sajtu koji sadrži repliku, što je lošije u odnosu na većinski protokol.

[4] Primarna kopija (*primary copy*)

Jedna od replika datoteke može se proglasiti primarnom replikom. Zahtev za zaključavanjem se prosleđuje samo sajtu na kome se nalazi primarna replika. Taj sajt dalje obezbeđuje konkurentnu kontrolu. Ako primarni sajt otkaže, datoteka će biti neraspoloživa, bez obzira što ostale replike postoje na drugim računarima.

Timestamp protokoli u distrubuiranim uslovima

Svaka transakcija ima vremensku oznaku (TS, *timestamp*) koja omogućava serijalizaciju, odnosno poredak izvršavanja. Jedinstvena TS vrednost može se generisati centralizovanom ili distrubuiranom metodom.

Dodela TS vrednosti je centralizovana ukoliko jedan sajt generiše i dodeljuje TS vrednosti na osnovu vrednosti svog logičkog brojača ili sopstvenog časovnika.

Dodela TS vrednosti je distribuirana ukoliko se jedinstvena TS vrednost formira udruživanjem jedinstvene lokalne TS vrednosti, koju svaki sajt generiše putem svog logičkog brojača, i identifikatora računara. Poredak udruživanja je bitan. Problem različitih brzina računara rešava se sinhronizacijom dva logička brojača pri slanju poruka između sajtova. Na primer, kada jedan sajt poseti drugi sajt sa manjim LC, tada se LC bržeg sajta sinhroniše, odnosno postavlja na LC vrednost sporijeg uvećanu za 1.

Šema poretka vremenskih oznaka

Ova šema kombinuje centralizovanu *timestamp* metodu sa 2PC protokolom u cilju obezbeđivanja poretka bez velikog broja kaskadnih *roll-back* operacija. Različiti zahtevi za čitanje i upis se baferuju, a zatim izvršavaju u poretku definisanom na sledeći način:

- zahtev $read_i(x)$ transakcije Ti za čitanje podatka x mora biti odložen ako postoji transakcija Tj koja treba da izvrši upis $write_j(x)$, a pri tome je TS(Tj) < TS(Ti).
- zahtev $write_i(x)$ transakcije Ti za upis podatka x mora biti odložen ukoliko postoji transakcija Tj koja treba da izvrši čitanje - $read_j(x)$ ili upis - $write_j(x)$ podatka x, a pri tome je TS(Tj) < TS(Ti).

13.4. Upravljanje zastojima u distribuiranim uslovima

Tehnike upravljanje zastojima u centalizovanoj okolinu, objašnjene u poglavlju 6, proširićemo na distribuirane uslove. Što se tiče prevencije zastoja, koriste se tehnika dodeljivanja brojeva resursa u rastućem redu. Bankarski algoritam se, takođe, može primenti na distribuiranu okolinu, ali zbog velikog broja operacija ima veliko kašnjenje. Definisaćemo dve nove metode koje koriste vremenske oznake.

[1] Metoda *wait-die*.

Ova metoda je zasnovana na tehnici bez pretpražnjenja, odnosno bez oduzimanja resursa. Proces Pi koji traži resurs dodeljen procesu Pj čekaće da proces Pj taj resurs otpusti samo ako je TS(Pi)<TS(Pj). Jednostavno rečeno, stariji proces čeka na mlađeg da otpusti resurs. U protivnom, tj. ako je TS(Pi)>TS(Pj), proces Pi se podvrgava *rollback* operaciji.

[2] Metoda *wound-wait*.

Ova metoda, suprotna metodi *wait-die*, zasnovana je na tehnici pretpražnjenja. Proces Pi će čekati na resurs dodeljen procesu Pj samo ako je Pi mlađi, odnosno ako je TS(Pi)<TS(Pj). U protivnom, Pj se podvrgava *rollback* operaciji, a resurs mu se oduzima i predaje procesu Pi. Dakle, stariji proces nikada ne čeka na mlađe

Za detekciju zastoja u centalozovanim sistemima korišćen je *wait-for* graf. U distribuiranoj okolini *wait-for* grafovi sajtova spajaju se u jedan zajednički graf, kao što je prikazano primerom na slikama 13.2 i 13.3.

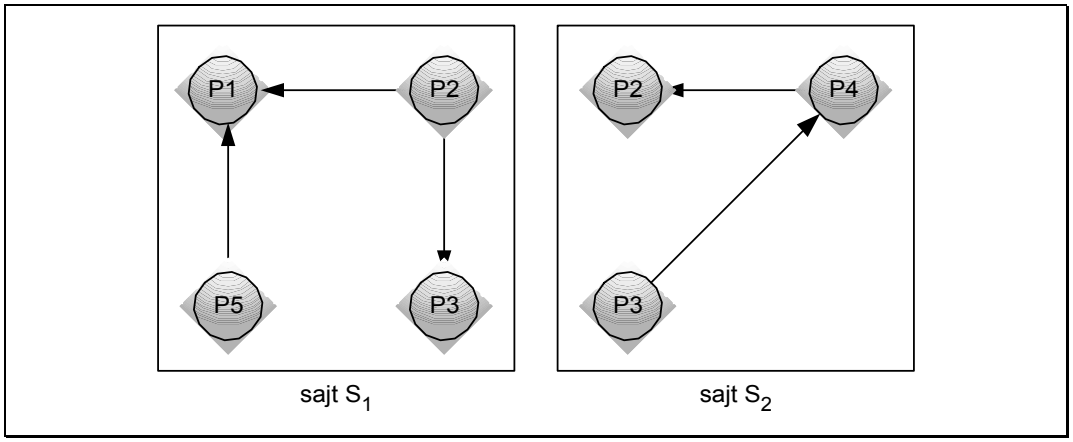
Detekcija zastoja u distribuiranim sistemima je složena procedura i može se obavljati centralizovano ili distribuirano.

[1] Centalizovani pristup

Problem zastoja rešava jedan sajt, na kome se izvršava proces koordinator za detekciju zastoja (*deadlock detection coordinator*). Taj proces treba da proceni da li je distribirani sistem u stanju zastoja ili ne. Ovaj proces traži od ostalih sajtova da mu dostave svoj *wait-for* graf, kako bi na osnovu toga konstruisao zajednički *wait-for* graf.

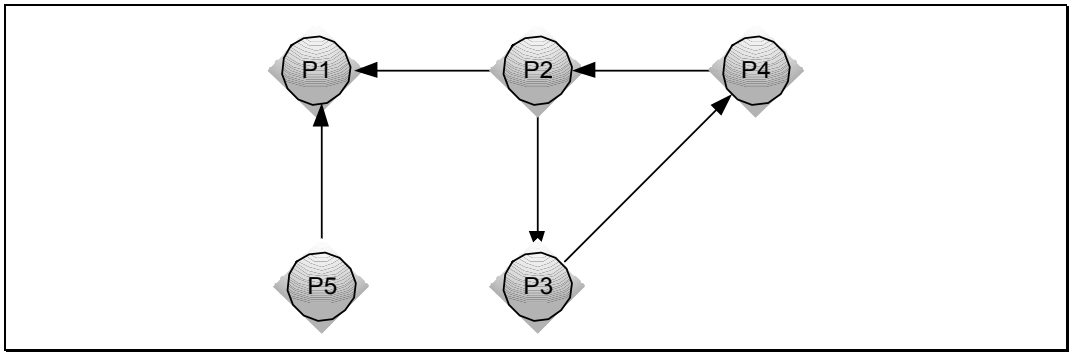
[2] Puni distrubuirani pristup

Svi sajtovi su zaduženi za detekciju zastoja. Svaki sajt konstruiše svoj modifikovani *wait-for* graf uključujući Pex čvor (eksterni čvor). Kada primeti mogućnot zastoja, sajt šalje poruku kritičnom sajtu sa kojim se može naći u stanju zastoja da oba sajta dodatno provere da li je sistem u zastoju.



Slika 13.2. *Wait-for grafovi sajtova S1 i S2*

Spajanjem pojedinačnih *wait-for* grafova dobija se zajednički *wait-for* graf. Ukoliko postoji bilo koji krug u pitanju je zastoj.



Slika 13.3. *Wait-for graf za sajtove S1 i S2*

Izborni algoritmi

U slučaju otkaza koordinatora, izbornim (*election*) algoritmima odabira se novi proces koordinator. Novi koordinator se bira između više kanditata na osnovu prioriteta. Pomenućemo dva takva algoritma.

[1] *bully* algoritam

Proces koji detektuje da nema koordinatora pokušava sebe da kandiduje za koordinatora, pa svim procesima višeg prioriteta šalje *election* poruku. U principu, proces svakome šalje poruku, a koordinator će postati proces najvišeg prioriteta koji to želi da postane.

[2] *ring* algoritam

Postoji i takozvani kružni algoritam (*ring*), koji je pogodan za mrežnu topologiju prstena. U ovim topologijama slanje velikog broja poruka nije pogodno, tako da se kandidat bira sa malim brojem razmenjenih poruka.

14

Zaštita i sigurnost

Sadržaj poglavlja:

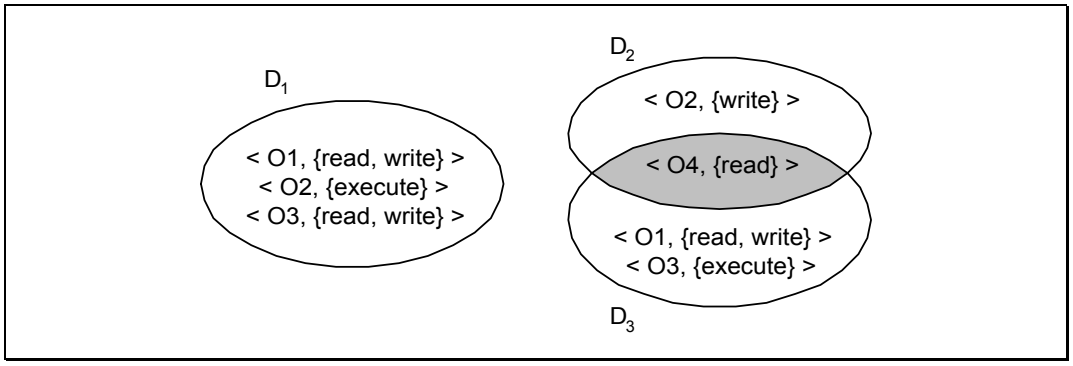
14.1. Domeni zaštite i matrice prava pristupa
14.2. Aspekti sigurnosti
14.3. Autentifikacija korisnika
14.4. Metode napada - programske i sistemske pretnje
14.5. Tehnike za povećanje sigurnosti
14.6. Kriptografske metode zaštite
14.7. Rangovi sigurnosti

14.1. Domeni zaštite i matrice prava pristupa

Zaštita se odnosi na kontrolu pristpupa programa, procesa i korisnika resursima operativnog sistema. Iako između pojma zaštite i pojma sigurnosti sistema postoji izvesna sličnost, zaštita se prvenstveno odnosi na unurašnjost sistema dok je sigurnost mnogo širi pojam.

Operativni sistem upravlja raznim objektima koji mogu biti hardverski (procesor, memorija, diskovi) i softverski (datoteka, program, semafor). Svaki objekat ima unikatno ime i može mu se pristupati kroz precizno definisani skup operacija. Problem zaštite svodi se na kontrolu pristupa objektima operativnog sistema: objektima mogu pristupati samo oni korisnici koji na to imaju pravo, odnosno koji su autorizovani i nad objektom mogu izvšiti samo operacije koje pripadaju dozvoljenom skupu operacija.

Svaki domen definiše skup objekata i sve operacije koje se mogu obaviti nad tim objektom. Mogućnost da se izvrši operacija nad objektom nazvaćemo pravo pristupa (*access right*). Domen je kolekcija prava pristupa koja su definisana parovima (ime objekta, skup prava).



Slika 14.1. *Domeni zaštite*

Na slici je prikazan sistem sa tri domena: D1, D2 i D3. Pravo pristupa <O4, read > je zajedničko za domene D2 i D3, što znači da proces koji pripada domenu D2 ili D3 može da izvrši operaciju čitanja nad objektom O4 (na primer, ukoliko je objekat O4 datoteka, proces je može otvoriti u režimu čitanja).

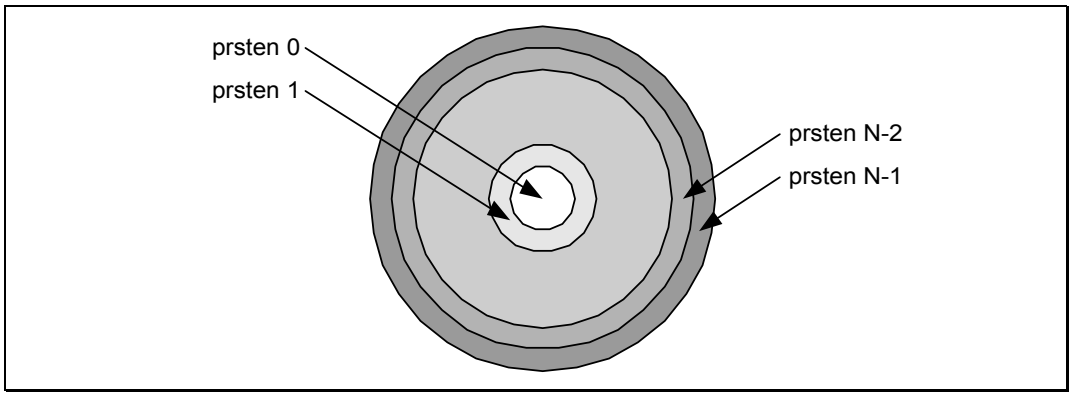
Alokacija procesa u domene može biti statička ili dinamička, a sam domen može da se realizuje na različite načine:

- svaki korisnik može biti domen,
- svaki proces može biti domen i
- svaka procedura može biti domen.

Svaki sistem koji ima dva režima rada (korisnički i sistemski) mora da ima najmanje dva domena: korisnički i sistemski domen.

Kod UNIX operativnog sistema, domeni su definisani na bazi korisnika (*domain* = UID). Prebacivanje domena može se realizovati preko sistema datoteka - svakoj datoteci može se dodeliti domenski bit (*setuid* - SUID bit). Ako se pokrene program sa postavljenim domenskim bitom, korisnik dobija identitet vlasnika datoteke; kada se program završi UID se resetuje, odnosno vraća na staru vrednost.

Kod Multics sistema, domeni zaštite su organizovani hijerarhijski u kružne strukture - prstenove, kao što je prikazano na slici 14.2. Svaki prsten predstavlja jedan domen. D0 je najprivilegovaniji domen - to je kernelski režim rada. Prava iz višeg domena uključena su u skup prava nižih domena, dok obrnuto ne važi.



***Slika 14.2.**. Hijerarhijska organizacija domena - prstenovi zaštite*

Matrica prava pristupa

Zaštita se može prikazati kao matrica pristupa (*access matrix*) u kojoj vrste predstavljaju domene a kolone predstavljaju objekte. Element matrice (i,j) predstavlja skup operacija koje proces iz domena Di može da izvrši nad objekatom Oj.

Domen	**Objekat**			
	Datoteka F1	Datoteka F2	Datoteka F3	Štampač
D1	read		read	
D2				print
D3		read	execute	
D4	read, write		read, write	

Matrica reguliše kontrolu pristupa procesa koji pripadaju različitim domenima nad objektima u sistemu. Međutim, u ovako definisanoj matrici procesi u određenim situacijama mogu preći iz jednog domena zaštite u drugi i time ostvariti veća prava nad objektom. U tom smislu uvodi se izvesna kontrola prelaska procesa iz jednog domena zaštite u drugi.

Naka je prebacivanje procesa iz jednog domena u drugi predstavljeno operacijom *switch*. Matrica se proširuje kolonama koje predstavljaju domene kako bi se mogle definisali moguće operacije prebacivanja iz jednog domena u drugi.

Domen	Objekat							
	F1	F2	F3	printer	D1	D2	D3	D4
D1	read		read			switch		
D2				print			switch	switch
D3		read	exec					
D4	read write		read write		switch			

Takođe, u nekim situacijama je potrebno izmeniti sadržaj matrice pristupa, odnosno dodeliti ili oduzeti pravo procesima jednog domena nad određenim objektom. U tom smislu uvode se sledeće operacije:

[1] Operacija *copy*

Operacijom *copy* kopira se pravo nad objektom, pri čemu odredišno polje pripada istoj koloni (procesima iz drugog domena daje se neko pravo pristupa nad tim objektom). Zvezdicom (*) označavamo pravo kopiranja, odnosno mogućnost da proces iz odgovarajućeg domena kopira pravo u drugi domen, odnosno u drugo polje iste kolone. Postoje tri varijante kopiranja prava:

- kopiranje prava - proces u drugom domenu dobija kopiju prava i kopiju prava kopiranja; dato pravo se ne oduzima od procesa koji obavlja operaciju *copy*;
- transfer prava - proces u drugom domenu dobija kopiju prava i kopiju prava kopiranja; kopirano pravo se oduzima od procesa koji obavlja operaciju *copy*;
- limitirano kopiranje - proces u drugom domenu dobija kopiju prava, ali ne dobija pravo kopiranja.

Sledeće tabele ilustruju operaciju kopiranja prava read nad objektom F_2 iz domena D_2 u domen D_3.

Domen	Objekat		
	F1	F2	F3
D1	execute		write*
D2	execute	read*	execute
D3	execute		

Domen	**Objekat**		
	F1	F2	F3
D1	execute		write*
D2	execute	read*	execute
D3	execute	read	

[2] Pravo vlasništva (*owner*)

U matricu je potrebno uvesti mehanizam koji omogućava dodavanje novih prava ili ukidanja postojećih. Ove operacije nad objektom mogu izvesti procesi iz domena koji ima pravo vlasništva nad tim objektom (*owner*). Na primer, ako je u polju (i,j) postavljeno pravo *owner*, tada proces se iz domena Di može ukidati ili postavljati prava nad objektom j (izmena je vidljiva u koloni j).

Na primer, iz sledećih tabela se vidi da procesi u domenu D_2 imaju pravo *owner* nad objektima F_2 i F_3 i da kao takvi mogu dodeliti ili oduzeti prava procesima iz drugog domena nad tim objektima. Konkretno, pravo *write* nad objektom F_3 oduzeto je domenu D_1 i dato domenu D_3.

Domen	**Objekat**		
	F1	F2	F3
D1	owner, exec		write
D2		read*, owner	owner, write*
D3	exec		

Domen	**Objekat**		
	F1	F2	F3
D1	owner, exec		
D2		owner	owner, write*
D3		read	write

[3] Pravo kontrole u domenu (*control*)

Operacije kopiranja, dodele i oduzimanja prava modifikuju sadržaj određene kolone u matrici. U matricu se uvodi i pravo kontrole u domenu (*control*) kojim je omogućena promena prava po vrsti. Pravo kontrole se može dodeliti samo objektima koji predstavljaju domene (na primeru sledeće tabele, to su poslednje četiri vrste u kojima su opisani domeni D_1-D_4). Ako je u polju (i,j) dato pravo *control*, proces koji pripada domenu Di može ukloniti bilo koje pravo dato domenu Dj (pravo u vrsti Dj).

Domen	Objekat							
	F1	F2	F3	printer	D1	D2	D3	D4
D1	read		read			switch		
D2				print			switch	switch control
D3		read	exec					
D4	read write		read write		switch			

Implementacija matrice prava pristupa

Matrica pristupa može se na sistemu implementirati na četiri načina, zavisnosti od skupa domena/objekata koji su konkretnom matricom opisani:

[1] Globalna tabela

Prvi i najprostiji slučaj je realizacija matrice pristupa pomoću globalne tabele koja se sastoji od skupa uređenih trojki (domen, objekat, skup prava). Pre nego što proces iz domena Di izvrši operaciju Sk nad objektom Oj, u globalnoj tabeli se traži odgovarajuća uređena trojka (Di, Oj, S), takva da Sk pripada skupu prava S. Ukoliko se takva trojka nađe, operacija se izvršava. U suprotnom, sistem odbija da izvrši operaciju.

Prednost ove metode je centralizacija zaštite na nivou sistema, a nedostatak veličina tabele - pretraživanje globalne tabele unosi veliko vremensko premašenje.

[2] Lista za kontrolu pristupa objektima

Matrica pristupa može se implementirati i pomoću liste za kontrolu pristupa objektima (*access list*). Posebna lista kontrole pristupa formira se za svaki objekat sistema i odgovara jednoj koloni matrice pristupa. Listu čini skup uređenih parova (domen, skup prava) - u listi su opisani svi domeni koji nad tim objektom imaju neka prava, a domeni bez prava se ne uključuju. Lista se može dopuniti listom podrazumevanih prava (*default*). Jednostavno rečeno, lista opisuje operacije koje procesi koji pripadaju različitim domenima mogu izvršiti nad tim objektom.

Liste za kontrolu pristupa su korisniku najpodesnije, jer vlasnik objekta može nad tim objektom jednostavno dodeliti ili oduzeti prava određenim domenima. Pri određivanju ukupnih prava domena moraju analizirati svi objekti.

[3] Lista sposobnosti domena

Treći način implementacije matrice pristupa su liste sposobnosti domena. Lista sposobnosti (*capability list*) formira se za svaki domen i odgovara jednoj vrsti matrice prava pristupa. Listu čini skup uređenih parova (objekat, pravo pristupa) - u listi su opisani svi objekti nad kojima taj domen ima neka prava. Jednostavno rečeno, lista sposobnosti jednog domena opisuje operacije koje procesi tog domena mogu izvršiti nad različitim objektima.

S korisničke tačke gledišta, liste sposobnosti nisu najpodesnije za korišćenje, ali su pogodne za lokalizaciju inforamcija pri analizi prava domena.

[4] Mehanizam ključeva (*lock-key*)

Mehanizam ključeva je kompromis prethodna dva načina implementacija matrice pristupa. Svakom objektu se dodeli lista bravica (*lock*), a svakom domenu lista ključeva (*key*). Ključevi i bravice su jedinstveni nizovi bitova. Proces iz domena može pristupiti objektu samo ako njegov ključ odgovara jednoj od bravica objekta.

Ovaj mehanizam je fleksibilan i efektivan, zavisno od veličine ključeva. Prava se mogu jednostavno oduzeti izmenom bitova koji čine bravicu.

Neki sistemi (na primer, MULTICS) koriste kombinovanu metodu liste za kontrolu pristupa objektina i liste mogućnosti. Prilikom prvog pristupa procesa objektu, proverava se lista za kontrolu pristupa. Ukoliko proces ima prava da pristupi objektu, objektu se pripisuje sposobnost. Prilikom daljeg referenciranja objekta, lista za kontrolu pristupa se ne proverava. Sposobnost se ukida prilikom zadnjeg pristupa objektu..

14.2. Aspekti sigurnosti

Za razliku od zaštite koja se odnosi na kontrolu pristupa i unutrašnjost sistema, sigurnost obuhvata više aspekata i po pravilu uključuje uticaj interackije sistema sa spoljnim svetom. Probijanje, odnosno rušenje sigurnosti sistema može biti slučajno (nehotične prirode) ili namerno (svesno) - naravno, sistem je mnogo lakše zaštiti od slučajnog napada (na primer, pogrešnog korišćenja) nego od namernog.

Sigurnost obuhvata zaštitu sistema od:

- neautorizovanog pristupa podacima i resursima,
- zlonamerne (maliciozne) modifikacije podataka,
- zlonamernog uništenja podataka i

- sprečavanja da se sistem legitimno koristi (*denial of service*).

Apsolutna sigurnost ne postoji, ali se sigurnosne mere uvek primenjuju i to na sledeća četiri nivoa:

[1] Fizički nivo

Računari i mrežna oprema moraju biti fizički obezbeđeni od vatre, vode, provalnika itd. Prva linija odbrane je računarskog sistema je zaštita računara i mrežne opreme kao što su serveri, ruteri itd. Ako hardver nije fizički dostupan neovlašćenim osobama, šanse da dođe do slučajnih ili namernih oštećenja se umanjuju.

[2] Ljudski faktor

Za sistem administratore potrebno je izabrati poverljive i ozbiljne ljude. Potrebno je naglasiti da uspeh napada na sistem često zavisi od ljudskog faktora. Na primer, programer može slučajno ili namerno ostaviti *backdoor* u programu, a administrator mreže port na ruteru. U oba slučaja, napadač može lako ostvariti pristup sistemu.

[3] Mrežni nivo

Korisnici računara danas intenzivno koriste mrežu za autentifikaciju i transfer podataka na relaciji server - radna stanica. Sigurnost na mrežnom nivou svodi se na obezbeđivanje udaljenog pristupa resursima za legitimne korsinike sistema, zaštitu resursa od neautorizovanog pristupa, zlonamerne modifikacije i/ili uništenja podataka i sprečavanje ulaska virusa ili drugih zlonamernih podataka i progama sa mreže u sistem. Na ovom nivou zaštite zahteva se da svaki korisnik koji pristupa umreženoj radnoj stanici ili serveru ima važeće korisničko ime na mreži i odgovarajuću lozinku.

[4] Nivo operativnovnog sistema

Operativni sistem mora zaštiti samog sebe i sistem u celini od slučajnog ili namernog oštećenja. Sigurnost na ovom nivou uključuje mehanizme zaštite (koji obuvataju kontrolu pristupa na nivou sistema datoteka) i autentifikacije korisnika; na ovom nivou zaštite zahteva se da svaki korisnik koji pristupa sistemu ima važeće korisničko ime na sistemu i odgovarajuću lozinku.

Napomenimo da će redovno kreiranje rezervnih kopija značajnih podataka minimizirati vreme oporavka u slučaju oštećenja, gubitka podataka i raznih drugih havarija. Takođe, potrebno je napraviti plan restauracije koji identifikuje kritične podatke i opisuje zaštitne mere koje je potrebno preduzeti u slučaju havarije ili proboja sigurnosti kako bi se brzo i sa minimalnim gubicima obezbedilo normalno funkcionisanje sistema.

14.3. Autentifikacija korisnika

Jedan od glavih problema sigurnosti operativnih sistema je autentičnost korisnika - operativni sistem na neki način mora da zna da li se radi o pravom, ovlašćenom korisniku ili ne. U tom smislu, svaki korsinik pre korišćenja sistema mora sebe da identifikuje. Generalno, autentifikacija se može obaviti na tri načina:

- specijalnim hardverom, kao što je ključ ili ID kartica,
- navođenjem poverljivih informacija, kao što je lozinka i
- biološkim atributima korisnika (*biometrics*), kao što su otisak prsta, snimak mrežnače oka (*retina scan*) i potpis.

Dodatno, korisnici se preko Internet mreže mogu autentifikovati pomoću sertifikata (na primer, X.509 sertifikatima).

Identifikacija korisnika pomoću poverljivih informacija je najčešće korišćen metod autentifikacije. Korisnik se, najpre, predstavi sistemu, odnosno identifikuje svojim imenom (*name ID*), a sistem zatim traži potvrdu, odnosno zahteva da korisnik navede odgovarajuću lozinku. Ako uneta vrednost lozinke odgovara vrednosti koja se nalazi na sistemu, operativni sistem smatra da je korisnik prošao autentifikaciju. Osim lozinke koja se koristi za prijavljivanje na sistem, lozinka se može pridružiti svakom objektu sistema, kao što je datoteka.

Lozinke su ranjivo mesto i kao takve su jedan od omiljenih objekata koje zlonamerni napadači koriste za razbijanje sigurnosti. Lozinke se mogu:

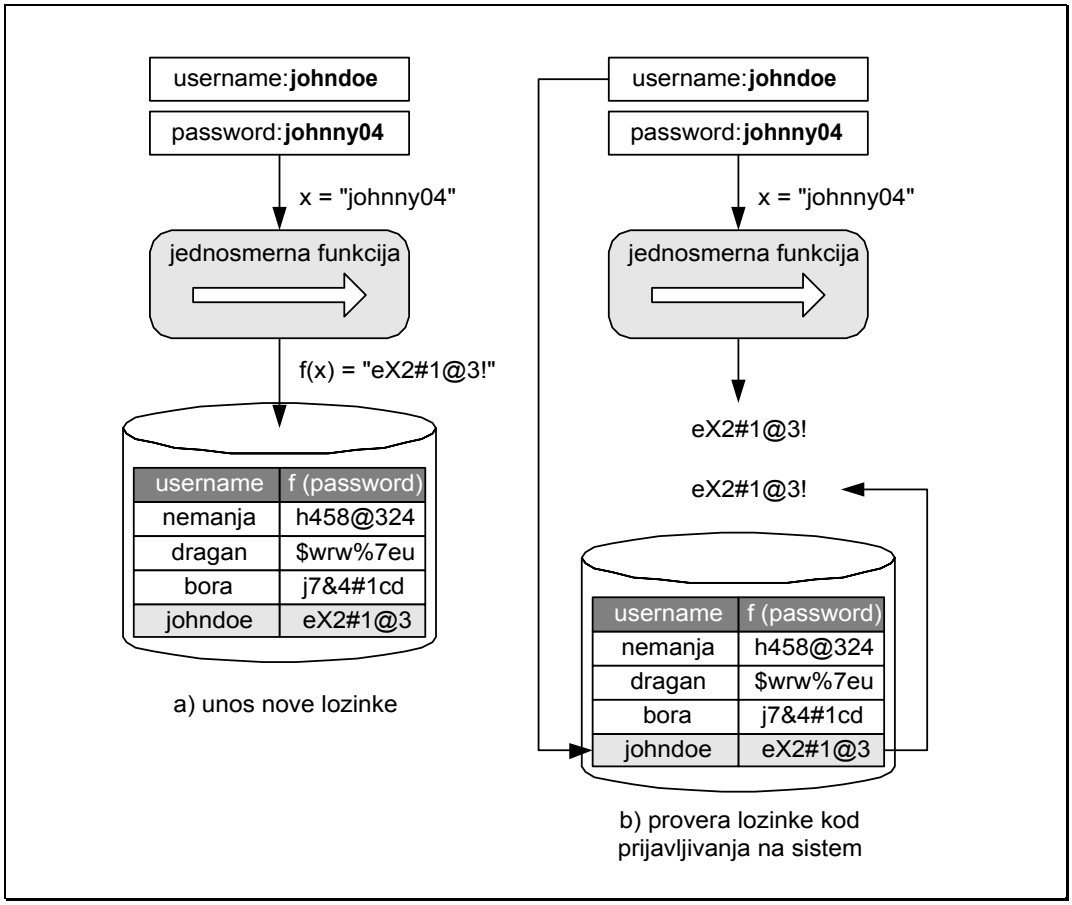
- jednostavno pogoditi, ukoliko su kratke ili jednostavne. Na primer, ukoliko je korisničko ime ppetar, lozinka pera se lako pogađa. S druge strane, programi za razbijanje lozinki koji koriste *brutal force* princip mogu se vrlo uspešno iskoristiti za pribavljanje lozinki koje su kratke. Potrebno je da lozinke budu složene, tj. da se sastoje od većeg broja alfanumeričkih karaktera i specijalnih znakova (npr. Re75Hab), poželjno je da se često menjaju, i da se pri promeni lozinke ne navode one koje su već korišćene,
- nehotično otkriti neznancu. Na primer, student gleda iza leđa dok profesor unosi lozinku i
- namerno, odnosno ilegalno proslediti neovlašćenim korisnicima u cilju ostvarivanja nekakve dobiti.

Čuvanje lozinke na disku sistema

Poseban problem predstavlja čuvanje informacije o lozinkama na disku računarskog sistema. Uljezi mogu doći do tih informacija, a onda mogu saznati korisničke lozinke uključujući i lozinke povlašćenih korisnika, kao što su sistem

administratori. U tom kontekstu, informacije o lozinkama se šifruju, odnosno obrađuju jednosmernim funkcijama (kao što su *hash* funkcije). Proces smeštanja lozinke na disk obavlja se na sledeći način (slika 14.3):

- korisnik unosi novu lozinku,
- operativni sistem računa vrednost jednosmerne funkcije sa lozinkom kao parametrom funkcije: realvalue=f(realpwd) i
- dobijena vrednost se smešta na disk u odgovarajuću tabelu; tabelu čine parovi (korisničko ime, vrednost funkcije).



Slika 14.3. *Jednosmerne funkcije i čuvanje lozinki*

Prilikom autentifikacije, korisnik navodi korisničko ime i lozinku. Sistem računa vrednost iste jednosmerne funkcije, a kao parametar navodi lozinku koju je korisnik uneo: value=f(pwd). Potom, sistem u tabeli traži vrednost funkcije koja odgovara

navedenom korisničkom imenu i upoređuje je sa dobijenom vrednošću. Ukoliko je f(realpwd)=f(pwd), sistem smatra da je korisnik prošao autentifikaciju (slika 14.3).

Napadač može doći do tabele u kojoj su opisani parovi (korisničko ime, vrednost funkcije), ali na osnovu tih vrednosti ne može rekonstruisati lozinke, jer su funkcije jednosmerne: vrednost y=f(x) može se lako izračunati, dok je $x=f^{-1}(y)$ teško računljivo ili uopšte neizračunljivo. Ovako zaštićeni sistemi napadaju se kriptoanalitičkim ili *brutal-force* metodama. Ove metode respektivno zahtevaju dobru matematičku osnovu, odnosno veliku procesorsku snagu, ali uspešan napad u oba slučaja garantuje sticanje privilegija povlašćenih korisnika. Jedna od metoda koja se u praksi pokazala vrlo uspešnom je ucenjivanje i iznuđivanje informacija od korisnika čija nam je lozinka potrebna (*rubber hose* metoda).

14.4. Metode napada - programske i sistemske pretnje

Jedan od uslova za uspešnu administraciju sistema, odnosno mreže, je razumevanje prirode potencijalnih pretnji i napada na sistem, odnosno mrežu.

Napad na sistem

Sistem se može napasti na mnogo načina; u najčešće korišćene vrste napada na sistem spadaju neautorizovani pristup i eksploatacija poznatih slabosti programa.

Najčešće korišćene metode eksploatacije slabosti programa su DoS, spoofing i sniffing. Administrator može najbolje zaštititi mrežu od napada ove vrste korišćenjem mrežne barijere. Dodatno, svi mrežni servisi koji nisu pouzdani moraju se isključiti ili zameniti alternativnim servisima (na primer, telnet se može zameniti ssh paketom).

[1] *Denial of Service*

DoS (*Denial of Service*, odbijanje usluga) kao napad izaziva prestanak rada servisa ili programa, čime se drugima onemogućava rad sa tim servisima ili programima. DoS napad se može izvršiti na mrežnom sloju slanjem zlonamernih datagrama kojima se izaziva raskid mrežne konekcije. Ovi napadi se mogu izvršiti i na aplikacionom sloju, slanjem specijalnih komandi programu, što kao posledicu ima prestanak rada programa.

[2] *Spoofing*

Napadač prati IP adrese u IP paketima i predstavlja se kao drugi računar. Kako DNS ne proverava odakle dolaze informacije, napadač može da izrši *spoof* napad dajući pogrešnu informaciju (ime računara od poverenja) DNS servisu. Najbolja zaštita od ovog napada je sprečavanje rutiranja datagrama sa neispravnim izvorišnim adresama.

[3] *Sniffing*

Sniffing (njuškanje) je metod u kome se specijalnim programima (*sniffer*) presreću TCP/IP paketi koji prolaze kroz određeni računar i po potrebi pregleda njihov sadržaj. Kako se kroz mrežu obično kreću podaci koji nisu šifrovani, snifer lako može doći do poverljivih informacija.

Programske pretnje

Program koji je napisao jedan korisnik (programer) a kojim se služe drugi korisnici predstavlja potencijalnu pretnju.

[1] Trojanski konj (*trojan horse*)

Trojanski konj je ilegalni segment koda podmentut u kod programa s ciljem da promeni funkciju ili ponašanje originalnog programa. Na primer, u tekst editor može biti podmetnuta rutina koja pretražuje otvorenu datoteku, i u slučaju da pronađe željenu sekvencu, kopira datoteku na mesto dostupno programeru koji je napisao taj editor. Specijalna varijanta trojanskog konja inje program koji oponaša proceduru prijavljivanja na sistem ili mrežu; programi ovakvog tipa presreću login proceduru i stvaraju sledeći scenario:

- napadač pokreće *login emulator* - na ekranu je prikazan *login prompt* identičan realnom koji čeka da korisnik unese korisničko ime i lozinku,
- korisnik unosi korisničko ime i lozinku,
- trojanski konj smešta uneto korisničko ime i lozinku u neku datoteku dostupnu napadaču, a zatim obaveštava korisnika da je pogrešno uneo lozinku,
- trojanski konj predaje kontrolu pravoj proceduri prijavljivanja na sistem,
- korisnik smatra da je uneo pogrešnu lozinku, unosi je ponovo i prijavljuje se na sistem i
- napadač proverava datoteku i prijavljuje se na sistem pod tuđim imenom.

Pretnje ovakvog tipa mogu se ukloniti ukoliko se procedura prijavljivanja na sistem poziva specijalnom kombinacijom tastera (kao što je Ctrl-Alt-Del) koja se ne može iskoristiti za pozivanje lažne *login* procedure.

[2] Klopka (*trap door*)

Autor programa može slučajno ili namerno ostaviti prazna mesta u svom kodu (*trap door*) - uljez koji zna za ta mesta može da podmetne svoj kod i time ostvari neku dobit. Takođe, autor programa može modifikovati deo koda tako da se modifikacija ne može jednostavno primetiti. Na primer, zaokruživanje iznosa transakcije na neku celobrojnu vrednost u određenim trenucima vremena predstavlja klopku ukoliko se ostatak zaokruživanja prenosi na račun programera.

Klopke se teško otkrivaju, jer je potrebno analizarati celokupan kod sumnjivog programa.

[3] Prepunjenje steka i bafera

Prepunjenje steka ili bafer je najčešće upućivan napad sa mreže pri pokušaju neautorizovanog pristupa sistemu. Autorizovani korisnici takođe mogu da iskoriste ovu vrstu napada kako bi prevarili sistem i ostvarili veća prava od onih koje imaju. Po pravilu, napadač korisiti grešku u programu, odnosno nedovoljno kontrole po pitanju razdvajnja steka, podataka i koda. Tada napadač šalje više ulaznih podataka nego što program očekuje i isprobava ranjivost programa na sledeći način:

- prepunjava ulazno poje, argumente komande linije ili ulazni bafer sve dok ne dođe do steka,
- prepisuje važeću adresu u steku adresom svog koda, koji se smešta u stek u sledećom koraku i
- puni deo steka svojim kodom, koji, na primer, izvršava neku komandu (kopira neke podatke ili pokreće komandni interpreter).

U slučaju uspešnog napada, umesto nedovoljno zaštićenog programa izvršiće se ilegalan kod ubačen prekoračenjem bafera.

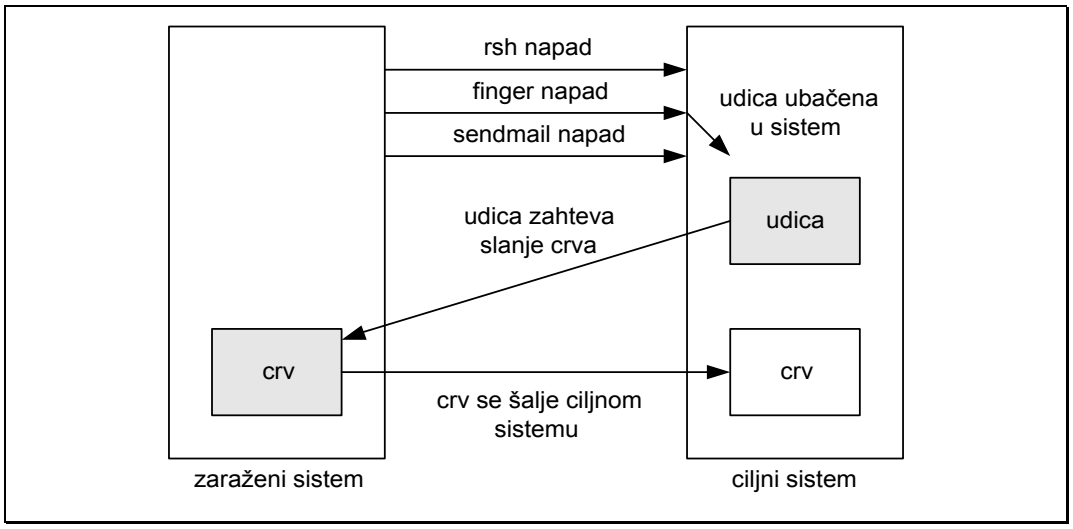
Sistemske pretnje

Mnogi operativni sistemi obezbeđuju mehanizam pomoću kog procesi mogu kreirati druge procese. U takvoj okolini moguće je zlonamerno korišćenje datoteka i sistemskih resursa. Dve metode kojima se to može postići crvi i virusi.

[1] Crvi

Crv je proces koji koristi mehanizam umnožavanja radi degradacije sistemskih performansi. Crvi su u mrežnom okruženju jako opasni jer mogu da se prošire i blokiraju celu mrežu.

Tako je 1988. godine Morisov crv napravio veliku štetu po UNIX sistemima. Crv se sastoji od glavnog programa koji koristi pokušava da otkrije lozinke korisnika i udice (*grappling hook*). Zaraženi sistem tehikom bafer-stek prekoračenja, pokušava da ubaci udicu na drugi čist sistem na mreži korišćenjem slabosti programa rsh, finger i sendmail (koji su tom periodu bili kritični po pitanju sigurnosti). U slučaju da crv uspe da progura udicu, udica šalje zahtev zaraženom računaru, u kome traži kopiju crva. Crv se kopira. Posle toga, obe mašine nastavljaju dalje da šire crve po mreži.



Slika 14.4. *Morisov crv*

[2] Virusi

Drugi slučaj sistemske pretnje je virus. Za razliku od crva koji se umnožavaju i smanjuju performanse sistema, virusi mogu da obave razne destruktivne akcije, kao što je brisanje datoteka, diskova u celini itd. Dok je crv samostalan program, virusi su fragmenti koda koji se ubacuju u druge legitmne programe i kao takvi predstavljaju jedan od glavnih problema personalnih računara. Po pravilu se dobijaju skidanjem zaraženih programa sa mreže ili preko elektronske pošte. Najbolji način za zaštitu od virusa je korišćenje dobrog antivirusnog softvera čija se lista antivirusnih definicija stalno osvežava.

14.5. Tehnike za povećanje sigurnosti sistema

Jedan od načina da se poveća opšta sigurnost sistema je periodično proveravanje mogućih sigurnosnih rupa u sistemu. U tom smislu potrebno je proveriti:

- da li postoje kratke lozinke ili lozinke koje se lako pogađaju,
- da li postoje opasni programi sa domenskim (SUID) bitom,
- da li postoje neautorizovani programi u sistemskim direktorijuma,
- da li postoji neočekivani proces koji se veoma dugo izvršava,
- da li postoji neodgovarajuća zaštita za direktorijume,
- da li postoji neodgovarajuća zaštita za sistemske direktorijume,

- da li postoje opasni ulazi u programskoj putanji i
- da li postoje promene u ček-sumama sistemskih programa.

Mrežne barijere

Umreženi računari nalaze se u većoj opasnosti od neumreženih jer se suočavaju sa nepoznatim korisnicima svuda po Internetu. U Internet okruženju, računari koji se nalaze na lokalnoj mreži mogu se smatrati poverljivim, dok sve svi ostali računari smatraju nepoverljivim. Komunikacija između ova dva sigurnosna domena može se ograničiti, odnosno kontrolisati, pomoću mrežnih barijera (firewall).

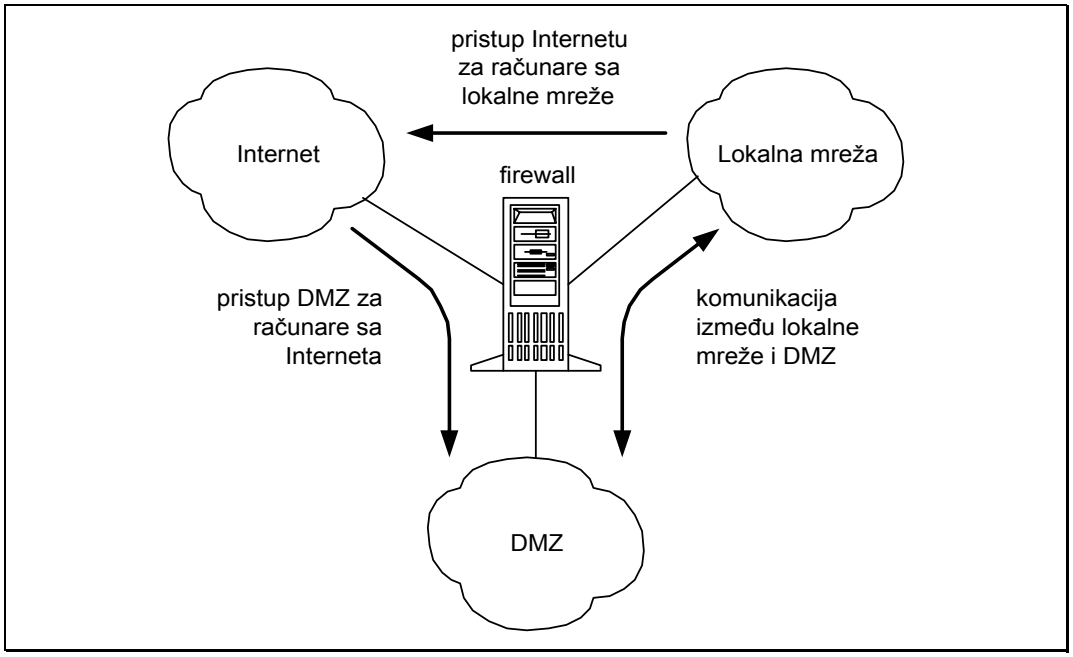
Mrežnim barijerama kreiraju se kontrolne tačke bezbednosti na granicama privatnih mreža. Na ovim kontrolnim tačkama mrežne barijere ispituju sve pakete koji prolaze između privatne mreže i Interneta. Mrežne barijere ispituju i posle toga odobravaju ili odbijaju svaki pokušaj povezivanja privatnih mreža i spoljnih mreža kao što je Internet. Jednostavno rečeno, firewall filtrira saobraćaj na relaciji lokalna mreža - Internet. Mrežne barijere koriste tri osnovne metode zaštite na mrežnom, transportnom i aplikacionom sloju.

- Na mrežnom sloju firewall filtrira pakete, odnosno odbacuje TCP/IP pakete neautentifikovanih računara. Mrežna barijera dozvoljava jedino prolaz IP paketa sa ispravnom IP adresom iz spoljne mreže. Prilikom kontrole saobraćaja iz interne mreže ka ostatku sveta neke mrežne barijere vrše maskiranje IP adresa. Neke mrežne barijere prevode interne IP adrese u validne IP adrese (*Network Address Translation*), dok druge zamenjuju sve interne adrese sa adresom mrežne barijere, čime se sprečava adresiranje računara u internoj mreži. Maskiranjem IP adresa uklanja se mogućnost praćenja spolja.
- Na transportnom sloju mrežna barijera dozvoljava ili zabranjuje pristup TCP/IP portovima zavisno od izvorišnih i odredišnih IP adresa. Na taj način se vrši kontrola pristupa TCP servisima.
- Na aplikacionom sloju proksi serveri prihvataju zahteve za pristup određenoj aplikaciji koji dalje upućuju ka odredištu ili blokiraju. Proksi servisi uspostavljaju konekciju na visokom aplikacionom nivou, čime se prekida konekcija na mrežnom sloju između računara u privatnoj mreži i računara iz spoljne mreže.

Kao mrežna barijera može se koristiti skup uređaja i servera od kojih svaki obavlja samo jednu od navedenih funkcija. Na primer, ruter obavlja filtriranje paketa, dok se proksi server nalazi na posebnom računaru unutar mreže.

Opasnost od napada sa Interneta može se znatno smanjiti korišćenjem dva nivoa zaštite. Osnovna ideja sastoji se u sledećem: jedan *firewall* štiti Web server od napada sa Interneta, ali dozvoljava pristup Internet servisima koje pruža taj deo mreže. Drugi firewall sa jačom bezbednosnom politikom ne dozvoljava pristup privatnoj mreži sa

Interneta i skriva identitete računara iz privatne mreže. Najveći broj *firewall* uređaja i softverskih paketa dozvoljava primenu različitih bezbednosnih politika na svakom interfejsu mrežne barijere. Na taj način jednom mrežnom barijerom sa tri interfejsa može se postići funkcionalnost dva firewall uređaja. U suštini, primenom ove tehnike mreža se deli na tri domena. Prvi domen je Internet, kao krajnje nepoverljivi i nesigurni domen, sledeći domen je polusigurna mreža DMZ (*demilitarized zone*) i treći, najsigurniji domen su lokalni računari. Po pravilu, konekcije su dozvoljene sa interneta ka DMZ i sa lokalnih računara ka Internetu, ali nisu dozvolene sa Interneta ili DMZ ka lokalnim računarim (osim u slučaju kontrolisane komunikacije).



Slika 14.5. *Mrežna barijera i podela mreže na domene*

Mrežne barijere ne mogu da:

- isfiltriraju validan paket koji u sebi nosi napad na sistem tipa prepunjenja bafera i
- spreče *spoofing* napada jer IP adresa paketa pripada opsegu koji može da prođe kroz *firewall*.

Detekcija napada

IDS (*Intrusion Detection System*) bavi se detekcijom napada i reakcijom na pokušaj napada na sistem. Metode detekcije mogu se klasifikovati u dve grupe:

[1] Detekcija zasnovana na oznaci

IDS analizira mrežni saobraćaj i traži se karakterističan uzorak koji otkriva napad, kao što je tipična sekvenca vistrukog pogrešnog logovanja.

[2] Detekcija anomalije

IDS otkriva anomalije u sistemu, kao što su sistemski pozivi sa velikom količinom ulaznih podataka (koji ukazuju na pokušaj napada tipa prepunjenja bafera).

U razvijene metode koje se koriste za detekciju napada spadaju praćenje sigurnosnih događaja (*auditing*), korišćenje softvera koji registruje promene u značajnim datotekama (kao što je /etc/passwd na UNIX sistemima) i praćenje sistemskih poziva.

Praćenje sistemskih poziva je jedna od novijih metoda zasnovana na anomaliji. Smisao metode je u sledećem: od svakog programa se unapred traži da definiše sekvence sistemskih poziva koje će pozivati pri normalnom izvršenju. Sekvenca se formira u vidu tabele koja se čuva u operativnoj memoriji za vreme prvog izvršenja programa. Prilikom ponovnog izvršavanja, sistem proverava se da li je sekvenca sistemskih poziva ista ili slična i ukoliko nije, pretpostavlja mogući napad.

14.5. Kriptografija

Šifrovanje obuhvata matematičke postupke modifikacije podataka takve da šifrovane podatke (šifrat) mogu pročitati samo korisnici sa odgovarajućim ključem. Proces šifrovanja matematičkom funkcijom uz određeni ključ transformiše otvoren tekst (*plain text*), odnosno originalnu poruku ili datoteku, u zaštićen, šifrovan tekst (*ciphertext* - šifrat). U osnovi svih kriptoalgoritama leže matematički postupci:

- supstitucije, odnosno zamene delova originalne poruke (pojedinačno ili u grupama konstantne dužine) drugim slovima ili kombinacijom tih slova sa ključem
- permutacije, odnosno preuređenja originalne poruke po nekom algoritmu.

Kombinovanjem ovih procesa ostvaruje se visok stepen zaštite podataka. Transformacije šifrata u otvoreni tekst (dešifrovanje) je obrnut proces - matematičkom funkcijom uz određeni ključ šifrovani podaci se transformišu u originalnu poruku ili datoteku. Šifrovani podaci su kao zaštićeni od neautorizovanog pristupa, jer korisnik bez odgovarajućeg ključa nema pristup podacima. Šifrovani podaci se mogu preneti preko nesigurnog kanala ili čuvati na disku koji nije obezbeđen od neautorizovanog pristupa.

Mehanizam šifrovanja čine sledeće komponenti:

- skup ključeva K,
- skup poruka M,
- skup šifrata C,
- funkcija šifrovanja E(M, K)->C, koja na osnovu ključa k i poruke m generiše šifrat c i
- funkcija dešifrovanja D(C,K)->M, koja na osnovu ključa k i šifrata c generiše poruku m.

Kriptoalgoritmi se dele na simetrične (isti ključ se koristi i za šifrovanje i za dešifrovanje podataka) i algoritme sa javnim ključem (podaci se šifruju javnim ključem a dešifruju privatnim).

Simetrični kriptoalgoritmi

Funkcija šifrovanja simetričnim kriptoalgoritmom E na osnovu ključa k i poruke m kreira šifrat c. Funkcija dešifrovanja D na osnovu istog ključa k i šifrata c kreira originalnu poruku m:

- E(m,k) = c, šifrovanje
- D(c,k) = m, dešifrovanje

Simetričnost algoritma dodatno implicira korišćenje iste funkcije za šifrovanje i dešifrovanje podataka. Ukoliko se poruka m šifruje funkcijom E pomoću ključa k, a zatim se istom funkcijom i istim ključem obavi dešifrovanje, dobiće se originalna poruka m, odnosno: E(m,E(m,k))=m.

Simetrični algoritmi su brzi i kao takvi se mogu koristiti za šifrovanje većih datoteka ili implementaciju u kripto sisteme datoteka. Najpoznatiji simetrični kriptoalgoritmi su DES (*Data Encryption Standard*), AES (*Advanced Encryption Standard*), IDEA, Blowfish i drugi.

Kriptoalgoritmi sa javnim ključem

Funkcija šifrovanja kriptoalgoritmom sa javnim ključem E na osnovu javnog ključa k1 i ulazne poruke m kreira šifrat c. Funkcija dešifrovanja D na osnovu privatnog ključa k2 i šifrata c kreira originalnu poruku m:

- E(p, k1) = c, šifrovanje javnim ključem i
- D(c, k2) = m, dešifrovanje privatnim ključem.

Javni ključ je poznat onim osobama sa kojima korisnik želi da komunicira, dok je tajni ključ poznat samo korisniku koji ga je kreirao. Najpoznatiji algoritam sa javnim ključem je RSA (patent kompanije RSA Data Security). Asimetrični algoritmi su sporiji

i primenjuju se za kreiranje digitalnih potpisa i šifrovanje ključeva simetričnih algoritama kojima su šifrovane datoteke.

Autentifikacija

Autentifikacijom se ograničava skupa potencijalnih pošiljaoca poruke. Autentifikacioni mehanizmi omogućavaju primaocu poruke da pomoću ključa proveri stvarni identitet pošiljaoca poruke.

Autentifikacioni mehanizmi zahtevaju postojanje sledećih komponenti:

- skup ključeva K
- skup poruka M
- skup autentifikatora A
- funkcija S: (K, M->A), koja na osnovu ključa k i poruke M generiše autentifikatora poruke
- funkcija V: (K, MxA-> boolean), koja na osnovu ključa k proverava autentifikatora poruke m,.

Digitalni potpis

Digitalni potpis (*digital signature*) je elektronska verziju potpisa, na osnovu koga se može identifikovati pošiljalac i dokazati verodostojnost poruke.

Digitalno potpisivanje se obavlja na sledeći način: najpre *hash* funkcija na osnovu ulaznih podataka bilo koje dužine kreira niz tačno određene dužine (*hash* vrednost, h) koji na jedinstven način definiše ulazni podatak. Pri tome se originalni podaci ne mogu odrediti na osnovu *hash* vrednosti, jer je funkcija strogo jednosmerna. Najčešće korišćena hash funkcija je MD5 (*Message Digest*).

Vrednost h se zatim šifruje privatnim ključem korisnika:

- H(m) = h1, kreiranje hash vrednosti,
- E(k1, h1) = s, potpisivanje hash vredosti privatnim ključem.

Korisnik šalje originalnu poruku i digitalni potpis primaocu. Primalac na osnovu poruke određuje hash vrednost h2 i dešifruje primljeni potpis javnim ključem pošiljaoca:

- H(m) = h2, kreiranje hash vrednosti i
- D(k2, s) = h1, provera primljenog potpisa javnim ključem.

Upoređivanjem vrednosti h1 i h2 proverava se identitiet pošiljaoca.

14.6. Rangovi sigurnosti

Nacionalni centar za sigurnost računara (*The National Computer Security Center* - NCSC, www.radium.ncsc.mil) osnovan je 1981. godine kao deo Nacionalne agencije za sigurnost (NSA) pri Ministarstvu odbrane SAD (DoD), kako bi pomogao u zaštiti svojine i ličnih podataka u računarskim sistemima vlade, korporacija i kućnih korisnika. NCSC je definisao nekoliko rangova, odnosno nivoa sigurnosti kojima se može ukazati na stepen zaštite komercijalnih operativnih sistema, mrežnih komponenti i aplikacija. Ovo sigurnosno rangiranje, zasnovano na Kriterijumu ocene pouzdanih računarskih sistema Ministarstva odbrane (*Trusted Computer System Evaluation Criteria* - TCSEC), definisano je 1983. godine, i uobičajeno se referira kao "narandžasta knjiga" ("the Orange Book"). Nivoi sigurnosti su sledeći:

- A1 - *Verified Design* (overen dizajn),
- B3 - *Security Domains* (domeni sigurnosti),
- B2 - *Structured Protection* (struktuirana zaštita),
- B1 - *Labeled Security Protection* (označena sigurnosna zaštita),
- C2 - *Controlled Access Protection* (zaštita kontrolisanim pristupom),
- C1 - *Discretionary Access Protection* (diskreciona zaštita pristupa) i
- D - *Minimal Protection* (minimalna zaštita).

TCSEC standard se sastoji od rangiranja ,,nivoa poverenja'', gde se viši nivoi grade na osnovu nižih nivoa dodavanjem rigoroznijih mera zaštite i zahteva za ispravnošću. Nijedan operativni sistem ne odgovara nivou A1. Iako je par operativnih sistema zaslužilo jedan od B-nivoa rangiranja, C2 se smatra dovoljnim i najvišim praktičnim rangom za operativni sistem opšte namene.

U julu 1995. godine, Microsoft Windows NT 3.5 (radna stanica i server) sa Service Packom 3 je bila prva verzija Windows NT koja je zaslužila C2 rangiranje. U martu 1999. godine, Windows NT 4 sa Service Packom 3 je dostigao E3 rangiranje koje je jednako C2 rangiranju u SADu. U novembru 1999. godine, Windows NT 4 sa Service Packom 6a postigao je C2 rangiranje i u samostalnoj (*stand-alone*) i u mrežnoj konfiguraciji.

Proces rangiranja zahteva nekoliko godina, tako da će uprkos tome što je Windows 2000 predat međunarodnim organizacijama za sertifikat sigurnosti, verovatno proći neko vreme pre nego što procena bude kompletirana. Ipak, osnovna sigurnosne arhitekture Windows 2000 je, ako ništa drugo, robusniji razvoj od onog u Windows NT 4, kao što se Windows NT 4 razvio iz Windows NT 3 implementacije. Windows 2000 će skoro sasvim sigurno postići isto rangiranje kao i Windows NT.

Navodimo ključne zahteve koje operativni sistem mora da ispuni kako bi dobio C2 rang sigurnosti:

- procedura sigurnog prijavljivanja na sistem (*secure logon facility)* - svi korisnici su jedinstveno identifikovani. Korisnicima se sme dozvoliti pristup računaru samo pošto su na neki način identifikovani (autentifikovani).
- diskreciona kontrola pristupa, koja omogućava vlasniku sredstva da odredi ko ima pravo pristupa sredstvu i šte može da uradi s njim. Vlasnik odobrava prava koja dozvoljavaju različite vrste pristupa korisniku ili grupi korisnika.
- praćenje sigursnosnih događaja (*security auditing*), koje omogućava sposobnost za otkrivanje i snimanje događaja vezanih za sigurnost ili svaki pokušaj kreiranja, pristupa ili brisanja sistemskih sredstava. *Logon identifiers* snimaju identitetete svih korisnika, što omogućuje lako praćenje i pronalaženje svakoga ko izvodi nedozvoljene (neautorizovane) akcije.
- zaštita ponovne upotrebe objekta, koja sprečava korisnike da vide podatke koje je drugi korisnik već obrisao ili od pristupa memoriji koju je već drugi korisnik koristio i oslobodio. Na primer, u nekim operativnim sistemima, moguće je kreirati novi dokument određene dužine i zatim ispitati sadržaj dokumenta kako bi videli podatke koji su zauzimali lokaciju na disku gde je dokument alociran. Ovi podaci bi mogli da budu osetljive informacije koje su bile ostavljene u dokumentu drugog korisnika, ali koje su bile izbrisane. Zaštita od ponovne upotrebe objekta sprečava ovu potencijalnu sigurnosnu rupu inicijalizacijom svih objekata, uključujući dokumente i memoriju, pre nego što su oni dodeljeni korisniku.

Windows NT takođe zadovoljava dva zahteva novoa B sigurnosti:

- Funkcionalnost pouzdanih putanja, koja sprečava programe tipa Trojanskog konja od mogućnosti da uhvate korisnička imena i lozinke, kada oni pokušavaju da se prijave na sistem. Funkcionalnost pouzdanih putanja u Windowsu NT dolazi u obliku njegovog Ctrl+Alt+Del *logon-attention* sekvence. Ova sekvenca pritisaka na tastere, koja je takoeđe poznata kao *secure attention sequence* (SAS), uvek prikazuje dijalog boks logovanja, tako da se Trojanski konj lako može prepoznati: Trojanski konj koji prikazuje lažno logovanje će biti premošten kad se pristupi SASu.
- *Trusted facility management,* koji zahteva podršku za funkcije odvojenih naloga za administrativne funkcije. Na primer, odvojeni nalog je obezbeđen za administriranje, korisnički nalozi zaduženi za podršku kompjutera i standardni korisnici.

Windows 2000 zadovoljava sve ove zahteve kroz svoj sigurnosni podsistem i vezne komponente.

Uobičajeni kriterijum

U januaru 1996. godine, SAD, Velika Britanija, Nemačka, Francuska, Kanada i Holandija su izdale zajednički razvijenu specifikaciju Uobičajenog kriterijuma informatičko tehnološke ocene sigurnosti (Common Criteria for Information Technology Security Evaluation -CCITSE). CCITSE, koji se obično referira kao Uobičajeni kriterijum (Common Criteria – CC), postaje prepoznatljiv međunarodni standard za ocenu sigurnosti proizvoda.

CC je fleksibliniji od TCSECovog rangiranja poverljivosti i ima strukturu bližu ITSECu nego TCSECu. CC uključuje koncept *Protection Profile* (PP) za sakupljanje zahteva sigurnosti u jednostavno specificirane i uporedive setove i koncept *Security Target* (ST) koji sadrži set sigurnosnih zahteva koji mogu biti napravljeni referenciranjem na PP.

15

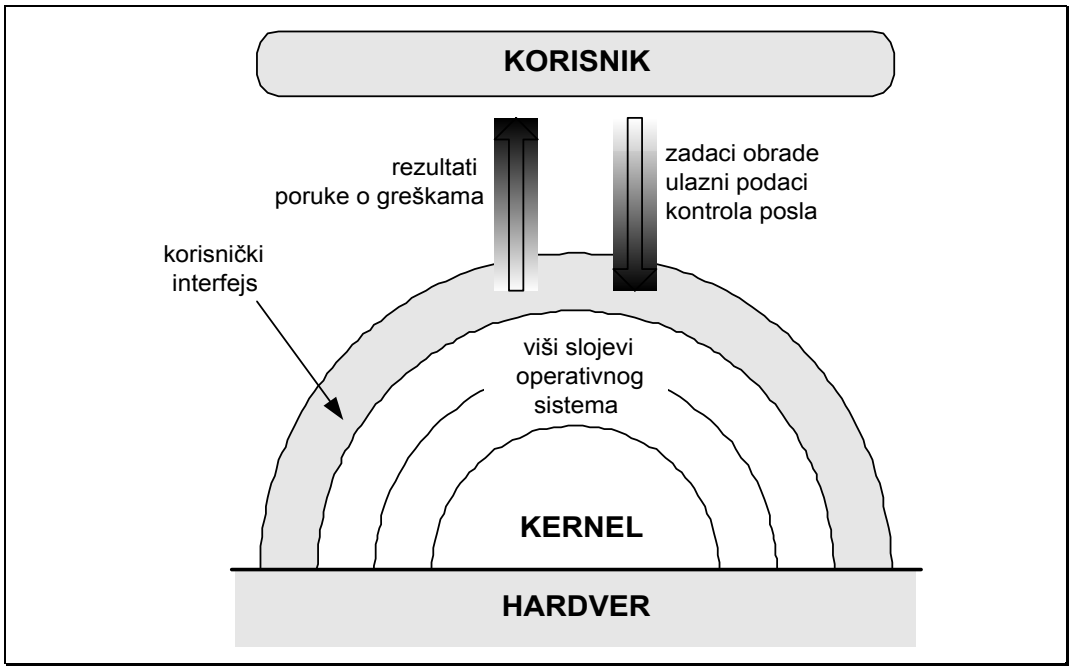
Korisnički interfejs

Sadržaj poglavlja:

15.1. Uvodne napomene
15.2. Komandni interpreteri
15.3. Kontrola posla
15.4. Grafički korisnički interfejs

15.1. Uvodne napomene

U prethodnim poglavljima opisane su osnovne komponente operativnog sistema opšte namene. Na vrhu hijerarhije slojeva operativnog sistema nalazi se korisnički interfejs, pomoću koga korisnici ostvaruju dvosmernu interakciju sa operativnim sistemom (slika 15.1). Korisnik operativnom sistemu saopštava koje poslove treba da obavi, a operativni sistem prikazuje rezultate obrade i poruke o greškama. Korisnika u užem smislu je osoba koja upotrebljava računar kao sredstvo za obavljanje nekog konkretnog zadatka; u širem smislu korisnici su i administratori, sistem inženjeri, programeri i operateri.



Slika 15.1. *Interakcija između korisnika i operativnog sistema*

Postoje dve osnovne vrste korisničkog interfejsa:

[1] Alfanumerički

Korisnik podatke unosi i vidi u obliku alfanumeričkih karaktera. Alfanumerički korisnički interfejs implicira postojanje jezika, jednostavnog ili kompleksnog, pomoću kog se može ostvariti komunikacija. Tip i karakteristike jezika zavise od vrste operativnog sistema. Na primer, jezik implementiran na sistemima paketne obrade (*batch systems*) razlikuje se od jezika implementiranog na interaktivnim sistemima. Razlog je u sledećem: korisniku interaktivnih sistema mora biti data mogućnost da interaktivno utiče na izvršenje nekog zadatka i odgovori na eventualne upite sistema. Korisnicima sistema paketne obrade ta mogućnost nije potrebna - oni jednostavno donose posao u sistem i sve potrebne parametre zadaju pre same obrade. Slično, jezik operativnog sistema IOS (operativni sistem specijalne namene, implementiran na Cisco ruterima i mrežnim barijerama) razlikuje se od jezika transakcionih ili upitnih sistema. Poželjne osobine svih jezika su razumljivost i jednostavnost korišćenja.

[2] Grafički

Grafičko radno okruženje (GUI, *graphical user interface*) korisnicima najčešće nudi potpun skup alata za rad sa datotekama, korišćenje i administraciju sistema. Grafičko radno okruženje po pravilu, opterećuje procesor i na nekim sistemima povećava rizik u smislu sigurnosti sistema. Kao takvo, na ozbiljnim serverima se

najčešće ne instalira, ali je zbog intuitivnosti i jednostavnosti korišćenja defacto standard za radne stanice.

15.2. Komandni interpreteri

Komandni interpreter je sistemski proces koji prihvata komande koje korisnik zadaje, zatim ih interpretira i potom izvršava. Po potrebi, komandni interpreter pokreće odgovarajuće programe. Na nekim sistemima, kao što su DOS i Windows, prisutan je samo jedan interpreter. Na UNIX i Linux sistemima postoji više različitih komandnih interpretera, a korisnici u toku rada po potrebi mogu preći iz jednog u drugi.

Osnovne funkcije komandnog interpretera

Tekst unet u odziv komandnog interpretera praćen pritiskom tastera Enter je komandna linija (*command line*). Komandni interpreter interpretira komandnu liniju, odnosno zahtev korisnika i izvršava određenu akciju. Komanda može biti interna, odnosno ugrađena u komandni interpreter (na primer, cd i dir na DOS i Windows sistemima, echo na Linux sistemima) ili eksterna, odnosno poseban program (na primer, deltree na DOS i Windows sistemima, rm na Linux sistemima). U opštem slučaju, komandna linija se sastoji od imena komande, opcija i argumenata. Ukoliko je komanda eksterna, ime komande referencira ime programa. Opcije i argumenti su parametri koje komandni interpreter prenosi komandi. Argumenti su najčešće obavezni i predstavljaju, na primer, ime neke datoteke ili direktorijuma. Opcijama korisnik utiče na izvršenje komandi.

Posle interpretacije komandne linije, komandni interpreter inicira izvršenje zadate komande. Ukoliko je komanda eksterna, komandni interpreter traži u sistemu datoteka izvršnu datoteku koja odgovara imenu komande, a posle toga pokreće program i prosleđuje mu argumente i opcije navedene u komandnoj liniji.

Izvršne datoteke koje odgovaraju imenu komande traže se na različitim mestima u sistemu datoteka, što zavisi od konkretnog operativnog sistema. Na DOS i Windows sistemima, izvršna datoteka (ekstenzije .bat, .com., exe) traži se u tekućem direktorijumu i u sistemskoj putanji, određenoj promenljivom PATH. Na Linux sistemima, izvršna datoteka (datoteka sa datim pravom pristupa x - *execute*) traži se isključivo u sistemskoj putanji, određenoj promenljivom PATH. U oba slučaja, ukoliko se datoteka ne nalazi u nekom od direktorijuma koji je naveden u sistemskoj putanji (ili u tekućem direktorijumu, za DOS/Windows), ime komande se mora zadati sa relativnom ili apsolutnom putanjom.

Poželjne osobine komandnog interpretera

Navodimo spisak poželjnih osobina komandnih interpretera.

- Neophodno je da imena komandi budu intuitivno vezana sa akcijom koju komande obavljaju, tako da ih korisnici mogu lako zapamtiti. Na primer, komanda za kopiranje datoteka treba da nosi ime copy ili cp, a datoteka za promenu tekućeg direktorijuma cd (*change directory*).
- Komandama koje zahtevaju parametre komandni interpreter treba prosleđivati podrazumevane vrednosti, u slučaju da korisnik ne navede ni jedan parametar. Za razliku od UNIX komande cp za kopiranje datoteke koja zahteva ime izvorišne i odredišne datoteke, DOS komanda copy dozvoljava korisniku da ne navede ime odredišne datoteke; u tom slučaju datoteka se kopira u istoimenu datoteku na tekućem direktorijumu.
- Format komande mora biti što jednostavniji. Redosled opcija ne sme biti od krucijalnog značaja, kao ni broj blanko karaktera između opcija i argumenata. S druge strane, ukoliko komanda kopira sadržaj jedne datoteke u drugu, redosled argumenata mora biti precizno određivati ime izvorišne i odredišne datoteke.
- Potrebno je omogućiti izvršavanje čestih aktivnosti koje zahtevaju unošenje većeg broja dugih komandnih linija jednom jednostavnom komandnom linijom, po mogućstvu bez parametara. Na primer, ukoliko korisnik radi na razvoju nekog programa, potrebno je da ima mogućnost da jednom komandom prevede sve izmenjene delove koda i prevede objektne module u izvršnu datoteku (kao što se obavlja komandom make na Linux sistemima).
- Korisniku mora biti jasno kada je sistem spreman da prihvati novu komandu, a kada je zauzet izvođenjem prethodno zadate komande.
- Sistem korisnike mora obavestiti o rezultatima izvršenja komande.

Napomenimo da neki sistemi korisnicima pružaju vrlo malo pomoći. Na primer, imena komandi na UNIX sistemima su kratka, kao i poruke kojima sistem obaveštava korisnike.

Dodatne funkcije komandnih interpretera

Osim interpretacije komandne linije i pokretanja programa, razni komandni interpreteri nude korisnicima dodatne funkcije, kao što su preusmeravanje ulazno-izlaznih operacija i rukovanje promenljivim.

[1] Preusmeravanje ulaza i izlaza

Operacije koje procesor izvršava ponekad zahtevaju izvršenje ulazno-izlaznih operacija. Na primer, korisnik pomoću tastature programu unosi podatke, a program posle izvršene obrade na ekranu prikazuje rezultat i po potrebi

obaveštenja o greškama. Komande koje izvršava komandni interpreter primaju podatke sa uređaja koji predstavlja standardni ulaz, rezultate izvršenja šalju na standardni izlaz, a poruke o greškama na standardni uređaj za greške.

Većina komandi podrazumevano koristi tastaturu kao standardni ulaz, a monitor kao standardni izlaz i uređaj za greške. Redirekcijom se ulaz i izlaz komande mogu preusmeriti na drugi uređaj. Na primer, komanda može čitati ulazne podatke iz datoteke a izlaz komande se može sačuvati kao datoteka ili odštampati radi kasnijeg čitanja. Standardni izlaz za greške se preusmerava ukoliko korisnik želi da sačuva rezultat izvršenja komande u nekoj datoteci radi kasnije analize grešaka (preusmeravanje se najčešće vrši na štampač ili u neku datoteku).

[2] Povezivanje komandi u pipeline

Pipeline funkcioniše na sledeći način: standardni izlaz komande sa jedne strane znaka koji predstavlja *pipe* (cev) postaje standardni ulaz komande sa druge strane znaka. *Pipeline* spregom komandi koje izvršavaju jednostavne, jasno definisane zadatke, mogu se izvršiti komplikovani poslovi.

[3] Zamena imena datoteka

Imena datoteka se mogu zameniti džoker karakterima (*joker*). Argument komande koji sarži džoker karakter zamenjuje se odgovarajućom listom datoteka shodno pravilima zamene. Komandni interpreter izvršava ovu zamenu pre izvršavanja same komande, odnosno pre pokretanja programa. Na primer, na UNIX i Linux sistemima džoker karakteri su * (menja bilo koji niz znakova proizvoljne dužine), ? (menja bilo koji, ali tačno jedan znak) i simbol opsega [beg-end] (menja tačno jedan znak koji pripada tom opsegu).

[4] Rukovanje promenjivim i kontrola okruženja

Okruženje je skup promenjivih čije vrednosti korisnici mogu menjati i na taj način prilagoditi radno okruženje svojim potrebama. Korisnici mogu definisati nove promenljive i brisati postojeće.

[5] Programiranje u komandnom interpreteru

Komandni interpreter se najčešće koristi za sekvencijalno izvršavanje zadatih komandi. Komandni interpreteri koji poseduju strukture za kontrolu toka i mogućnost rukovanja promenljivim mogu se koristiti kao programski jezici. Na taj način mogu se automatizovati razni administrativni zadaci koji zahtevaju od korisnika da unese veliki broj komandi.

[6] Aliasi

Alias je način dodele kraćeg imena pomoću kog se određena komanda ili niz komandi može pozvati iz komandnog interpretera. Na primer, komande mogu imati svoja duga imena i svoje skraćenice, koje se mogu koristiti umesto imena komande.

[7] Ponavljanje prethodno zadatih komandi

Poželjno je da komandni interpreter čuva u operativnoj memoriji ili upisuje u neku datoteku nekoliko prethodnih komadnih linija; time je omogućeno jednostavno ponavljanje prethodno zadatih komandi bez prekucavanja. Takođe je poželjno da editor komandne linije poseduje mogućnost izmene komande pre ponovnog izvršavanja.

15.3. Kontrola posla

Korisnici koji se prijavljuju na sistem obično sa sobom donose jedan ili više poslova. Posao (*job*) je skup procesa koje je potrebno obaviti kako bi se izvršio određeni zadatak. Na primer, posao obračunavanja plata može obuhvatati nekoliko procesa: unos podataka sa nekog medijuma, izračunavanje iznosa, formiranje izveštaja i štampanje. Kada korisnik donede posao, planer poslova visokog nivoa (*job scheduler*) deli poslove na procese kojima na osnovu određenih algoritama dodeljuje prioritete, a zatim dovodi procese u red čekanja na procesor.

Na sistemima sa paketnom obradom korisnici sve potrebne parametre zadaju pre same obrade. Kontrola posla se obavlja pomoću programa pisanim na jeziku za kontrolu posla (JCL - *job control language*) koji kontrolišu prolaz posla kroz računarski sistem. U kontrolišuće informacije koje sačinjavaju opis jednog posla spadaju:

[1] Obračunske informacije

Najčešće sadrže ime posla i identifikator korisnika. Ove informacije se koriste radi provere validnosti korisnika (da li je korisnik ovlašćen da koristi usluge sistema ili ne) i formiranja obračuna usluga na osnovu utrošenih resursa sistema.

[2] Informacije o raspoređivanju

Sadrže spisak maksimalnih količina različitih resursa, koje su neophodne za izvršenje posla. Ukoliko se posao može razdvojiti na nekoliko zasebnih celina, informacije o resursima se navode za svaku celinu. Na osnovu ovih informacija planer poslova visokog nivoa određuje koji će poslovi biti prihvaćeni, a koji odbijeni, donosi ispravne odluke rasporedu izvršavanja poslova s ciljem izbegavanja zastoja i dodeljuje prioritete procesima. Dodatno, informacije o raspoređivanju uključuju zahtev za izvršavanje procesa sa određenim prioritetima i zahtev da se posao izvrši odmah nakon završetka drugog posla.

[3] Informacije o korišćenju ulazno-izlaznog podsistema

Ove informacije uključuju zahteve za korišćenjem ulazno-izlaznih uređaja i datoteka. Dodatno, mogu se navesti i informacije o korišćenju specifičnog karakter koda i veličini bloka za ulazno-izlazne operacije.

[4] Proceduralne informacije

Ovaj deo opisa posla odnosi se na konretne zahteve korisnika, odnosno opisuje šta korisnik očekuje od operativnog sistema. Mogućnost opisivanja proceduralnih informacija u velikoj meri utiče na snagu jezika za kontrolu posla. Neki jezici za kontrolu posla podržavaju jednostavne sekvence direktiva, dok na drugim proceduralne informacije uključuju:

- makroe, koji omogućavaju definisanje složenog skupa akcija pomoću jedne komande,
- kontrolne strukture, koje omogućavaju izbor akcija koje će biti izvršene u sledećem koraku na osnovu rezultata izvršenja akcije u prethodnom koraku,
- informacije o paralelizmu, pomoću kojih se može specificirati paralelno izvršenje više delova posla, pod uslovom da je posao deljiv na manje celine koje su nezavisne i koje se mogu paralelno izvršavati.

15.4. Grafički korisnički interfejs

Na osnovu svega što je do sada rečeno u ovom poglavlju može se zaključiti da specifikacija posla ili zadavanje tačnih komandi nije uvek očigledno ili jednostavno. Uzroci su najčešće neprikladan izbor imena određenih komandi ili strukture jezika za kontrolu posla. Neke sisteme, kao što je UNIX, karakteriše dug vremenski period potreban za prilagođavanje korisnika; ponekada korisnici koji godinu ili više dana rade sa UNIX sistemom ne mogu da okarakterišu sistem kao jednostavan za korišćenje.

Za razliku od alfanumeričkog, grafički korisnički interfejs se karakteriše slikovitim prikazom na ekranu visoke rezolucije i kombinovanim korišćenjem miša i tastature. Snaga grafičkog korisničkog interfejsa leži u njegovoj ergonomičnosti i upotrebljivosti. Većina ljudi lakše pamti i shvata slike od teksta, tako da je sam proces učenja operativnog sistema i aplikacija koje se na njemu izvršavaju lakši. Konfigurisanje sistema iz komandne linije takođe nije jednostavno. Na primer, konfigurisanje štampača na Linux sistemima zahteva od korisnika da unese nekoliko komandi, sličnih sledećim:

```
# /usr/sbin/lpadmin -p HP1 -E -v usb:/dev/usb/lp0 -m
deskjet.ppd
# /usr/sbin/lpadmin -d HP1
# /usr/sbin/lpadmin -p HP1 -o job-quota-period=604800 -o
job-page-limit=100
# /usr/sbin/lpadmin -p HP1 -u allow:@hpusers
```

U grafičkom okruženju korisnik može dodati novi štampač, proglasiti ga podrazumevanim, odrediti kvotu i pristupna prava interakcijom sa odgovarajućim kontrolama na ekranu.

Takođe, grafičko okruženje omogućava korisniku da pokrene neke programe čije je izvršenje u tekstualnom režimu nemoguće ili nema nikakvog smisla (programi za obradu rasterskih slika ili stono izdavaštvo).

Većina operativnih sistema namenjenih radnim stanicama, kao što su UNIX, Linux, Microsoft Windows, IBM OS/2 Workplace Shell i Mac OS, sadrži grafički korisnički interfejs. Na UNIX i Linux sistemima, grafički inerfejs se obezbeđuje na sledeći način: X Window server obezbeđuje skup alata za implementaciju grafičkog interfejsa, a programi za upravljanje prozorima (*window manager*i), kao što su KDE i Gnome, koriste usluge X servera i obezbeđuju grafički interfejs prema korisniku. X Window sistem je naročito zanimljiv jer razdvaja funkcije izračunavanja i prikaza do te granice da program koji se izvršava na jednom računaru može koristiti ekran drugog računara sa kojim je na neki način povezan.

Većina grafičkih interfejsa je još uvek u fazi razvoja; osnovni ciljevi razvoja su intuitivnost, jednostavnost korišćenja, visoki nivo estetike i podrška za rad sa multimedijalnim sadržajima. Takođe, primetan je trend svojevrsne unifikacije grafičkih interfejsa.

16

Windows familija operativnih sistema

Sadržaj poglavlja:

16.1. Istorijat Microsoft operativnih sistema
16.2. Windows NT
16.3. Windows 2000
16.4. Windows XP
16.5. Windows 2003 Server
16.6 Longhorn

16.1. Istorijat Microsoft operativnih sistema

Umesto uvoda u ovo poglavlje, navešćemo spisak svih Microsoft operativnih sistema:

- MS-DOS
- Xenix
- Windows 1.0, 2.0, 3.0 i 3.1
- Windows for Workgroups 3.11
- Windows NT
- Windows 95, 98

- Windows Millenium Edition (ME)
- Windows 2000
- Windows XP
- Windows Server 2003
- Windows CE
- Windows PPC
- Windows WM
- Longhorn (kodno ime za novu verziju Windowsa)

MS-DOS

Kompanija Microsoft je pre više od dvadeset godine korisnicima PC računara ponudila operativni sistem pod imenom MS-DOS. MS-DOS (*Disk Operating System*) je kao jednokorisnički, jednoprocesni operativni sistem male hardverske zahtevnosti u to vreme bio pogodan za upotrebu na PC računarima. Koncept jednokorisničkog, jednoprocesnog operativnog sistema doveo je do krajnje jednostavnosti slojeva u hijerarhijskom modelu.

[1] Upravljanje procesorom

Pre svega, MS-DOS je strogo jednoprocesni sistem. Program pokrenut iz komandne linije izvršava se u prvom planu, bez obzira da li zahteva interakciju sa korisnikom ili ne. Primitivni oblik višeprocesnog rada može se ostvariti preko TSR programa (*terminate and stay resident*), odnosno programa koji nakon učitavanja ostaju smeštani u memoriji, s ciljem da nakon nekog događaja izvrše određenu akciju. Navodimo primere TSR programa, koji su isporučivani uz MS-DOS 6.2:

- VSafe, zaštitni programa kompanije Central Point Software (poznate po proizvodu PC Tools), za koji je Microsoft otkupio licencu. Program posle učitavanja kontroliše akcije izvršene nad diskovima (sprečava upis podataka u *boot* sektor i particionu tabelu, proverava validnost izvršnih datoteka na osnovu ček-sume, itd.)
- Undelete (u režimu Delete Sentry), takođe licenciran od kompanije Central Point Software, koji formira specijalan direktorijum u koji se smeštaju obrisane datoteke. Ovaj direktorijum može se smatrati pretečom Recycle Bin foldera na kasnijim Windows sistemima.

DOS Shell, program za upravljanje datotekama koji se isporučivao uz neke verzije MS-DOS sistema, poseduje primitivnu mogućnost zamene posla koji se izvršava - *Task Switching*. Poslovi u pozadini su neaktivni, tj. ne postoje procesi koji se izvršavaju u pozadini. Ovim se MS-DOS ne proširuje u višeprocesni

operativni sistem, jer je samo jedan proces aktivan, dok su drugi neaktivni (nema prebacivanja konteksta).

MS-DOS ne može da iskoristi prednost višeprocesorske arhitekture.

[2] Upravljanje memorijom

Svi programi se izvršavaju u realnom režimu rada (*real mode*). Ulazak u zaštićeni režim (*protected mode*) i korišćenje odgovarajućih instrukcija 80386/80486 procesora obezbeđuje se na nivou softvera. Interfejs za korišćenje zaštićenog režima procesorsa je kasnije "standardizovan" DOS/4GW interfejsom - većina programa koji su koristili zaštićene instrukcije koristili su ovaj interfejs kao podlogu za rad.

MS-DOS razlikuje sledeće tipove memorije:

- Osnovnu memoriju, kapaciteta 640KB, koja se koristiti za smeštanje procesa, TSR programa i drajvera. Ovaj limit potiče sa XT računara (8086), a može se proširiti preuzimanjem video memorije sa grafičkog adaptera (programom QEMM koji služi za upravljanje memorijom na 80386/80486 računarima), pri čemu je korisniku uskraćen grafički režim rada video adaptera. Svi programi zahtevaju postojanje osnovne memorije i u slučaju da nema dovoljno slobodne osnovne memorije, ne mogu se izvršiti.
- *Extended* memoriju, odnosno memoriju iznad 640KB. Podrška za *extended* memoriju isporučuje se u vidu HIMEM.SYS drajevera (*high memory*). Programi koji koriste *extended* memoriju zahtevaju određenu količinu osnovne memorije prilikom pokretanja, a kasnije im na raspolaganju stoji širi adresni prostor koji obezbeđuje *extended* memorija..
- *Expanded* memoriju, odnosno memoriju iznad 640KB koja radi na principu straničenja. Pošto je određen skup programskih paketa strogo zahtevao *expanded* memoriju za svoje izvršenje, u MS-DOS je uključen EMM386 drajver koji je u *extended* memoriji simulirao *expanded* memoriju (prema LIM standardu - Lotus, Intel, Microsoft).
- Više memorijske blokove (UMB, *upper memory block*), odnosno memoriju između 640KB i 1MB koja se može koristiti za smeštanje TSR programa.

Koncept virtuelne memorije MS-DOS sistemu je nepoznat.

[3] Ulazno-izlazni podsistem

Hardverska podrška je skromna - MS-DOS korisnicima nudi podršku za rad sa flopi i ATAPI hard diskovima i tastaturom. Softverska podrška za širi spektar hardverskih uređaja realizovana je u vidu *real-mode* drajvera, koje su nekada pisali proizvođači hardvera ili softvera. Na primer, da bi se koristio CD-ROM, prilikom podizanja sistema mora se učitati driver iz CONFIG.SYS konfiguracione datoteke, a zatim i Microsoft CD ekstenzija, koja se isporučuje u kasnijim varijantama MS-DOS sistema.

[4] Sistem datoteka

MS-DOS kao domaći sistem datoteka koristi FAT. Da odmah naglasimo, svi sistemi datoteka se aktiviraju prilikom podizanja operativnog sistema i kasnije se ne mogu deaktivirati. MS-DOS nema ugrađenu podršku za strane sisteme datoteka. Neki strani sistemi datoteka mogu se aktivirati pomoću specijalnih drajvera (SFS486.SYS drajver za Secure Filesystem, autora Petera Gutmanna ili NTFSDOS, kompanije SysInternals), dok se podacima na nekim sistemima datoteka može pristupiti pomoću programa specijalne namene (paket ltools, koji dozvoljava čitanje i upis na ext2 sistem datoteka).

FAT (*file allocation table*) je jednostavan sistem datoteka zasnovan na principu mape datoteka, realizovane u vidu FAT tabele sa 16-bitnim adresiranjem. Datotekama se prostor dodeljuje u vidu klastera (*cluster*) fiksne veličine. Veličina FAT sistema datoteka ograničena je brojem klastera (16535 klastera po sistemu datoteka) i veličinom klastera. Sistem datoteka najvećeg kapaciteta može se dobiti samo korišćenjem većih klastera veličine, ali su u tom slučaju gubici veći zbog interne fragmentacije (*slack*).

Svaka datoteka na FAT sistemu datoteka ima svoje ime, datum i vreme, i osnovne atribute. Ime datoteke se može zadati isključivo korišćenjem 7-bitnog ili 8-bitnog ASCII ili ANSI standarda. Ime se zadaje u formatu 8.3 - 8 karaktera za ime, a 3 za ekstenziju, kojom je ujedno i određen tip datoteke. Datoteke je opisana sledećim atributima: H (*hidden*, - skrivena datoteka), S (*System* - sistemska datoteka), R (*Read-only* - datoteka je samo za čitanje), A (*Archive*).

Pouzdanost sistema je relativno niska: postoje dve identične FAT tabele, pri čemu se sadržaj jedne ažurira na osnovu sadržaja druge. Pošto su tabele fizički smeštene jedna do druge na disku, velika je verovatnoća da će prilikom fizičkog oštećenja jedne FAT tabele i druga biti oštećena. Na FAT sistemu datoteka ne postoji dnevnik transakcija, tako da je upotreba *write-back* keširanja krajnje rizična.

[5] Kontrola pristupa i zaštita

MS-DOS je jednokorisnički sistem, tako da se korisnik na MS-DOS sistem ne prijavljuje ni na kakav način. Pristupna prava i vlasnički odnosi na FAT sistemu datoteka nisu regulisani, tako da se na nivou sistema datoteka ne može izvršiti nikakva kontrola pristupa. Zaštita datoteka može se ostvariti isključivo korišćenjem dodatnog softvera, najčešće kriptografskog (kao što su PGP, Norton Diskreet i SFS - Secure Filesystem).

[6] Mrežno okruženje

MS-DOS ne prepoznaje mrežu. Podrška za mrežni rad realizuje se preko drajvera za mrežne karte i odgovarajuće nadogradnje MS-DOS-a u mrežni operativni sistem (Novell Netware).

[7] Korisnički interfejs

Korisnički interfejs MS-DOS sistema je strogo alfanumerički. Komandni interpreter (izvršna datoteka COMMAND.COM, najčešće smeštena na root direktorijumu diska C), korisniku daje mogućnost zadavanja internih i eksternih komandi, pokretanja programa i elementarne redirekcije standardnog izlaza u datoteku. Pipe mehanizam je delimično funkcionalan (sprega izlaza neke komande sa programom more za prikazivanje sadržaja datoteka ekran po ekran). Komandni interpreter omogućava postavljanje okruženja (pomoću promenljivih) i *batch* obradu (.BAT datoteke). Kao nadogradnja COMMAND.COM komandnog interpretera raspoloživ je NDOS, sastavni deo paketa Norton Utilities, kompanije Symatec.

MS-DOS nema grafičko okruženje.

Osim Microsofta, DOS sisteme su proizvodile i druge softverske kompanije, poput Digital Research (DR-DOS) i IBM (PC-DOS).

Većina dugogodišnjih korisnika računara dobro je upoznata sa dobrim i lošim osobinama DOS operativnih sistema. U suštini, DOS dobija na funkcionalnosti ukoliko se na sistem instalira nekoliko uslužnih programa i Windows 3.x kao grafičko okruženje. Kao najveću prednost, pomenućemo veliku brzinu rada programa koji se izvršavaju (nema prebacivanja konteksta, zamene stranica, premašenja koje unose razni servisi višeprocesnih operativnih sistema). Jedna od programerskih šala s početka devedesetih godina će to najbolje ilustrovati: kako ćete od 486 računara najlakše napraviti XT računar? Otkucaćete win.

Microsoft Windows 3.x

Microsoft Windows 3.0 i 3.1 su, uslovno rečeno, operativni sistemi sa grafičkim korisničkim interfejsom. Teško je definisati ove verzije Windowsa kao potpune operativne sisteme, jer se one oslanjaju na sistemsku podlogu koju formira MS-DOS. Windows 3.0 i 3.1 ne mogu se instalirati ako na računaru prethodno nije instalirana odgovarajuća verzija MS-DOS sistema. Stoga, neki autori definišu Windows 3.x kao grafičko okruženje MS-DOS sistema (kao što, na primer, xfree i KDE čine grafičko okruženje Linux sistema). Navešćemo neke značajnije karakteristike Windows 3.x operativnih sistema.

[1] Upravljanje procesorom

Windows 3.x je jednokorisnički sistem sa delimičnim pretpražnjenjem. Više procesa mogu se izvršavati kvazi-paralelno, ali o trenutku kada se procesor oslobađa i dodeljuje drugom procesu ne odlučuje operativni sistem, nego sam proces. Ovaj način multiprogramiranja ima nekoliko nedostataka. Korisnik može pokrenuti veći broj aplikacija, koje će se izvršavati u prvom planu ili pozadini, ali otkaz jedne aplikacije može usloviti kolaps sistema.

[2] Upravljanje memorijom

Windows 3.x koristi 16-bitno adresiranje memorije i sposoban je da pokreće MS-DOS programe i Win16 programe. Win16 programi dele se na segmente, odnosno blokove memorije veličine između 16 bajtova i 64KB. Pri pisanju programa, programeri moraju voditi računa o ključnim rečima kao što su _near i _far, kako bi se obezbedilo ispravno adresiranje. Windows 3.x je prvenstveno korišćen kao DOS *multitasker*, ali je sa povećanjem broja Win16 aplikacija njegova upotreba dobila pravi smisao. Veliki broj aplikacija, prvenstveno napisanih za DOS (Microsoft Word, Word Perfect, AutoCad, itd.) prenešeno je na Windows, dok neki softverski paketi namenjeni Windows sistemu nisu imali svoj DOS ekvivalent (na primer, Corel DRAW).

Zajednički problem svih 16-bitnih verzija Windowsa je nemogućnost oslobađanja rezervisane sistemske memorije. Ovaj problem nastao je kao posledica pogrešne odluke u projektovanju: 16-bitni Windows zahteva od aplikacija da formiraju i brišu objekte u rezervisanoj sistemskoj memoriji. Sam Windows ne uklanja sistemske objekte iz memorije kako bi omogućio deljenje grafičkih objekata između programa. Ukoliko aplikacija zaboravi da obriše objekat, sistemska memorija biva izgubljena. Ukoliko sistem radi duže vreme bez prestanka (na primer, nekoliko nedelja), postoji mogućnost značajnog gubitka sistemske memorije.

U Windows 3.x uveden je koncept virtuelne memorije. Virtuelna memorija realizuje se pomoću *swap* datoteke, čija veličina može biti fiksna ili promenljiva.

[3] Ulazno-izlazni podsistem

Jedna od značajnijih prednosti Windows 3.x operativnih sistema je podrška za širi spektar hardvera. Između ostalog, skupu drajvera koje Microsoft isporučuje uz Windows 3.x pripadaju drajveri za SVGA grafičke adaptere, miševe, Sound Blaster kompatibilne zvučne kartice i štampače. Opseg važenja drajvera obuhvata sve aplikacije; na primer, ako je na sistem instaliran drajver za laserski štampač, sve aplikacije koje šalju zahteve za štampu koristiće taj drajver. Na MS-DOS sistemu, neki drajveri mogu biti univerzalni za sve aplikacije (na primer, drajver za specifičan sistem datoteka), dok su neki specifični za konkretan program (drajver za štampač za Microsoft Word for DOS ne implicira mogućnost štampe iz WordPerfecta). Ulazno-izlazni podsistem Windows 3.x sistema obuhvata *spool* tehniku za štampanje (*print manager*).

[4] Sistem datoteka

Što se tiče sistema datoteka, Windows 3.x koristi FAT sistem datoteka. Imena datoteka se i dalje zadaju u formatu 8.3. Kontrola pristupa na nivou sistema datoteka nije implementirana. Kako se u osnovi sistema nalazi DOS, svi sistemi datoteka se aktiviraju prilikom podizanja operativnog sistema i kasnije se ne mogu deaktivirati. U Windows 3.x nije ugrađena podrška za strane sisteme datoteka.

[5] Koncept dinamičkog povezivanja

Windows 3.x se u velikoj meri oslanja na funkcije i podatke smeštene u biblioteke sa dinamičkim povezivanjem (DLL, *dynamic link libraries*). Kada se malo bolje pogleda jedan Windows sistem, skoro sve što korisnik vidi postiže se pomoću biblioteka sa dinamičkim povezivanjem. DLL se zasniva na konceptu klijent-server komunikacije. Funkcije u DLL-u (server) dostupne su većem broju klijenata (aplikacijama ili drugim bibliotekama sa dinamičkim povezivanjem). DLL je sličan običnoj izvršnoj (*runtime*) ili statičkoj biblioteci, s tim što se funkcije i podaci prilikom dinamičkog povezivanja ne kopiraju iz DLL-a u izvršnu datoteku. Da podsetimo, u slučaju statičke biblioteke, funkcije i podaci prevode se u binarnu datoteku, a zatim ih povezivač kombinuje sa drugim modulima aplikacije i formira izvršnu datoteku. Zahtevani delovi biblioteke su u slučaju statičkog povezivanja integrisani sa aplikacijom. U slučaju dinamičkog povezivanja, funkcije i podaci se nalaze u posebnoj datoteci - DLL-u. DLL se mora isporučiti uz svaku aplikaciju koja koristi funkcije tog DLL-a. Pri tome, jedan DLL može opsluživati veći broj aplikacija.

[6] *True-type* tehnologija

Jedna od značajnijih novina uvedena u Windows 3.1 je tehnologija *true-type* fontova, patent kompanije Apple. Prethodne verzije Windows sistema, zaključno sa verzijom 3.0, koristile su rasterske fontove za prikazivanje teksta na ekranu i štampanje. Rasterski fontovi se formiranju na sledeći način: svako slovo je u određenoj veličini definisano bit-mapom. Ovaj pristup nije pogodan za grafičke operacije rotacije i skaliranja slova zbog neadekvatne rezolucije bit-mapa. *True-type* tehnologijom je u Windows 3.1 uveden koncept vektorskih fontova: svako slovo je definisano konturom (jednom ili više) koju čine prave i krive linije. Jedna od osnovnih prednosti tehnologije vektorskih fontova je u tome što su jednom konturom određene sve veličine slova. Nad tekstom formiranim od vektorskih fontova mogu se vršiti grafičke transformacije bez gubitka kvaliteta teksta.

Međutim, sledeća stvar ostaje nerazjašnjena: ako je Microsoft otkupio licencu za *true-type* tehnologiju, zašto nisu otkupljene i odgovarajuće licence za imena fontova? Na prmer, umesto fonta Helvetic, na Windowsu je prisutan Arial, a umesto Times, Times New Roman.

Windows for Workgroups 3.11

Na operativnom sistemu Windows for Workgroups 3.11 nećemo se detaljno zadržavati, jer ovaj sistem nasleđuje većinu osobina sa Windowsa 3.1. Značajno je da se pomene da je u ovu verziju Windowsa uveden koncept umrežavanja preko podrške za radne grupe (*workgroups*) i TCP/IP steka protokola koji se može naknadno

instalirati. Takođe, uz ovu verziju Windows sistema može se instalirati Internet Explorer 2.0 i prateći *mail* klijent.

Microsoft Windows 95/98

Za razliku od Windowsa 3.x, Windows 95/98 su operativni sistemi u pravom smislu te reči. I ove verzije se, kao i Windows 3.x, oslanjaju na sistemsku podlogu koju formira MS-DOS. Međutim, za razliku od prethodnih verzija DOS je integralni deo Windows 95/98 sistema. Navešćemo neke značajnije karakteristike Windows 98 operativnih sistema.

[1] Multiprogramiranje sa pretpražnjenjem

Više procesa mogu se izvršavati kvazi-paralelno, a o trenutku kada se procesor oslobađa i dodeljuje drugom procesu na korišćenje odlučuje operativni sistem, odnosno rutine za dodelu procesora (*preemtive scheduler*). Za razliku od Windows 3.x sistema, kod koje otkaz jedne aplikacije može usloviti kolaps sistema, Windows 98 je otporniji na otkaz aplikacija.

Kao i 16-bitne verzije, Windows 95/98 nije sposoban da iskoristi prednosti višeprocesorske arhitekture.

[2] Niti

U Win32 okruženju, svaki pokrenuti program sastoji se iz procesa, a svaki proces od jedne ili više niti. Fundamentalna razlika između 16-bitnih i 32-bitnih Windows sistema je u tome što 32-bitni Windows ne ograničava broj niti u procesu na jednu. Proces započinje izvršenje kao jedna nit, koja može stvoriti dodatne niti. Rutine za dodelu procesora dele procesorsko vreme aktivnim nitima, čime se stiče predstava o istovremenom izvršavanju većeg broja niti. Sekundarne niti su idealne za obavljanje poslova u pozadini (kao što je renumerisanje stranica u tekst procesoru). Sve niti se dodeljuju jednom procesoru na izvršenje, bez obzira na broj procesora u sistemu.

[3] Prioriteti procesa

Većina proces započinje izvršenje u klasi prioriteta NORMAL, i posle toga nema potrebu da menja klasu prioriteta. U toku izvršenja, proces može da promeni klasu prioriteta u IDLE, NORMAL, HIGH i REALTIME. Procesi koji se izvršavaju u klasama prioriteta HIGH i REALTIME mogu ozbiljno da umanje sposobnost sistema da odgovori na zahtev korisika, jer im dispečer sistema dodeljuje više procesorskog vremena u odnosu na procese koji se izvršavaju u klasi NORMAL.

[4] 32-bitno adresiranje

Pod Windows 98 operativnim sistemom mogu se pokrenuti MS-DOS, Win16 i Win 32 programi. Jedno od najznačajnijih poboljšanja Windows 98 operativnog sistema u odnosu na prethodne verzije je čisto 32-bitno adresiranje.

Da bi smo objasnili ukratko kako Windows 98 upravlja memorijom, objasnićemo najpre koncept 32-bitnog straničenja na 80386 i novijim procesorima. Četiri Intelova režima adresiranja za 32-bitne procesore su:

- realni režim (*real mode*),
- 16-bitni zaštićeni režim (*16-bit protected mode*),
- 16-bitni virtuelni 8086 režim i
- 32-bitni režim sa straničenjem (*32-bit paged mode*).

Win32 API napravljen je za procesore koji podržavaju 32-bitno adresiranje sa straničenjem. Ovaj način adresiranja je moguć samo na 80386 i kasnijim procesorima. Prethodne generacije Intelovih procesora nemaju ove karakteristike, tako da Windows 98 ne može da radi sa njima. Zbog karakteristika procesora 80386, 80486 i Pentium, Win32 programi koji rade pod Windows 98 sistemom koriste veličinu stranice 4KB. Intelovo 32-bitno dvonivovsko stranično adresiranje podrazumeva podelu adresa na tri dela:

- najnižih 12 bitova predstavlja indeks unutar stranice (2^{12}=4KB),
- 10 viših bitova predstavlja indeks u tabeli stranica (služe za izbor stranice iz tabele stranica) i
- 10 najviših bitova predstavlja indeks u direktorijumu stranica (služe za izbor tabele stranica iz direktorijuma stranica).

Za razliku od Windowsa NT, koji svakom procesu dodeljuje privatni adresni prostor, Windows 98 nije tako darežljiv. Ako bi se svakom procesu dodelio zaseban adresni prostor, premašenje po procesu iznosilo bi 4KB za direktorijum stranica i 4KB za svaku tabelu stranica. Umesto toga, Windows 98 formira jedan skup tabela stranica. Sve što vidi jedan proces zove se memorijski kontekst procesa. Za vreme promene konteksta procesa, Windows 98 ažurira tabele stranica i vrši zamene u tabeli koje se tiču privatnih stranica procesa. Deo adresnog prostora koji sadrži sistemske DLL-ove, zajedničke za sve procese, prilikom izmene konteksta procesa ne menja. Na taj način se obezbeđuje da svaki proces vidi svoje privatne podatke u jednom adresnom opsegu (4MB-2GB) i zajedničke podatke u drugom (2GB-4GB). Zbog kompatibilnosti sa Windows 3.x sistemima, 16-bitne aplikacije u Windows 98 okruženju nemaju svoj privatni adresni prostor, već rade u zajedničkom adresnom prostoru.

[5] Formati izvršnih datoteka

Definisaćemo formate izvršnih datoteka koje se mogu pokrenuti pod Microsoft Windows 98 operativnim sistemom.

- MZ (MS-DOS program) - relokaciona izvršna datoteka Marka Zbikovskog, definisana u operativnom sistemu MS-DOS 2.0. Može se reći da je Zbikovski ušao u legendu ugrađujući svoje inicijale u oznaku izvršne datoteke.
- NE (*new executable*, Win16) - nove izvršne datoteke, odnosno segmentno orijentisani Win16 program.
- PE (*portable executable*, Win32) - prenosive izvršne datoteke. Windows 98 i Windows NT koriste isti format Win32 izvršnih datoteka. U oba slučaja, Win32 programi se učitavaju kao memorijski mapirane datoteke (*memory mapped file*). Mehanizam se zasniva na tome da upravljač virtuelnom memorijom povezuje datoteku na disku sa opsegom memorijskih adresa. Stranice se čitaju sa diska samo onda kada se pristupa odgovarajućim memorijskim adresama.
- LE (*linear executable*, VxD) - linearne izvrpne datoteke, odnosno 32-bitni upravljački programi uređaja, uvedeni u Windowsu 3.0.

[6] Problem čišćenja sistemske memorije

Windows 98 poštuje pravilo Win16 programa da deli objekte, što može predstavljati potencijalan problem, jer Win16 program sa greškom može da potroši sistemske resurse i ugrozi rad Win32 programa. Windows 98 se brani od ovih situacija na nekoliko načina:

- brisanjem GDI objekata preostalih iz Win16 programa u slučaju da se ne izvršava nijedan Win16 program. Na ovaj način se poštuje pravo Win16 programa na deljenje objekata.
- pre kreiranja GDI objekata, Windows proverava da li objekat tog tipa već postoji. Ukoliko postoji, prethodno kreirani objekat se koristi.

[7] Ulazno-izlazni podsistem

U Windows 98 uključena je drajverska podrška za širok spektar hardverskih uređaja, uključujući Plug and Play i USB uređaje. Windows 95/98 obavlja automatsku detekciju instaliranog hardvera i instalira odgovarajući drajver.

[8] Sistem datoteka

U Windows 98 uključena je podrška za rad sa FAT i FAT32 sistemima datoteka. Kako se u osnovi sistema nalazi DOS, svi sistemi datoteka se aktiviraju prilikom podizanja operativnog sistema i kasnije se ne mogu deaktivirati. U Windows 98 nije ugrađena podrška za strane sisteme datoteka. U 32-bitnom režimu rada, Windows smešta datoteke na disk sa dugim imenom datoteke (*long filename*). Ove datoteke su iz DOS i 16-bitnih sistema vidljive kao datoteke sa imenom u formatu 8.3. Na primer, direktorijum C:\Program Files vidljiv je iz DOS sistema

kao direktorijum C:\PROGRA~1. Kao i u prethodnim verzijama Windowsa, kontrola pristupa na nivou sistema datoteka nije implementirana.

[9] Mrežni rad

Windows 98 radne stanice funkcioniču u mrežama organizovanim na konceptu radne grupe i domena. Ukoliko u mreži ima Windows 98 klijenata, kao domen kontroler može se koristiti Windows NT 4 Server. Windows 2000 Sever uz neka ograničenja podržava rad sa Windows 98 klijentima.

Razvoj NT tehnologije

Godine 1988. Microsoft je odlučio da razvije novu tehnologiju (*new technology - NT*), odnosno portabilni (prenosivi) operativni sistem koji podržava i OS/2 i POSIX API-je. Izvorno, NT je trebao da koristi OS/2 API kao svoje prirodno okruženje, ali je tokom razvoja izmenjen tako da koristi Win32 API, što je odražavalo popularnost Windowsa 3.0. Ostaje zabeležena zvanična izjava vlasnika kompanije Microsoft, data početkom devedesetih godina, da se odustaje od razvoja Windowsa i prelazi na OS/2. Prve verzije NT operativnog sistema, Windows NT 3.1 i Windows NT 3.1 Advanced Server, prihvatile su korisnički interfejs tada aktuelne verzije 3.1 16-bitnog Windows-a. Nakon toga, objavljene su verzije NT Workstation 4.0 i NT Server 4.0, zasnovane na korisničkom interfejsu Windows 95. Jedna od značajnijih promena načinjenih u NT 4.0 familiji operativnih sistema je pomeranje delova koda koji upravljaju grafičkim podsistemom i korisničkim interfejsom u kernel, što je u suprotnosti sa pravilima teorijskog operativnog sistema (*paper model OS*) i za posledicu ima smanjenje pouzdanosti samog sistema. Prema statistici, 1996. godine je prodato više NT server licenci, nego Unix licenci. Takođe, NT 4.0 je poslednja verzija koja je prenešena i na druge arhitekture, osim Intelove. Kao posledica stanja na tržištu, u verzijama koje slede (Windows 2000, Windows XP) ukinuta je podrška za sve procesore koji nisu Intel kompatibilni.

U nastavku teksta daćemo kratke opise za:

- Windows 2000,
- Windows XP,
- Windows Server 2003 i
- Longhorn.

U vreme dok pišemo ovu knjigu (kraj 2004. godine), Longhorn je kodno ime za novu verziju Windowsa i informacije koje ovde dajemo mogu se pokazati netačnim u vreme planiranog izlaska ovog operativnog sistema (2006. godine). Zato molimo čitaoce da informacije o Longhornu uzmu sa rezervom.

16.2. Windows NT

Windows NT 4.0 je operativni sistem za Intel x86, MIPS i Alfa platforme. Navodimo ključne ciljeve razvoja Windows NT 4.0 operativnog sistema:

[1] Portabilnost

Windows NT se može preneti na druge platforme uz ulaganje minimalnog napora. Kod Windows NT sistema je modularan, većim delom napisan na C i C++ programskim jezicima. NT sadrži sloj apstrakcije hardvera (HAL) koji rasterećuje operativni sistem poznavanja detalja vezanih za konkretan hardver.

[2] Proširivost, odnosno lako ostvarljiva podrška za nove uređaje.

[3] Pouzdanost

Otpornost sistema na greške u hardveru, aplikacijama i greške koje čine korisnici ostvaruje se izolacijom aplikacija od drugih aplikacija (Windows NT svakom procesu dodeljuje privatni adresni prostor, što nije karakteristično za Windows sisteme koji nisu zasnovani na NT tehnologiji), izolacijom aplikacija od kernela i korišćenjem transakcionog sistema datoteka i mehanizama za organizaciju sekundarne memorije sa redundansom (*disk mirroring*).

[4] Kompatibilnost

Kompatibilnost, odnosno mogućnost operativnog sistema da pokreće programe napisane za prethodne verzije operativnog sistema ili druge operativne sisteme, Windows NT ostvaruje odgovarajućim podsistemima okruženja. Windows NT može da izvršava MS-DOS, OS/2, Win16 , Win32 i POSIX aplikacije.

[5] Sigurnost

Zaštitu lokalnih i mrežnih resursa od neovlašćenog pristupa Windows NT obezbeđuje pomoću *Security Reference Monitor*a, žetona za pristup (*Access Token*), identifikacije korisnika i grupa i listi za kontrolu pristupa.

[6] Višeprocesorska podrška

Za razliku od Windows sistema koji nisu zasnovani na NT arhitekturi, Windows NT je sposoban da iskoristi prednosti višeprocesorske arhitekture.

[7] Visoke performanse

Brzi odziv sistema obezbeđen je dobrim iskorišćenjem sistemskih resursa. Visoke performanse sistema obezbeđuje 32-bitna arhitektura sistema, optimizacija kritičnih procesa (kao što su sistemski pozivi i rukovanje PF prekidima) i mrežno-komunikacioni elementi ugrađeni u privilegovane delove operativnog sistema.

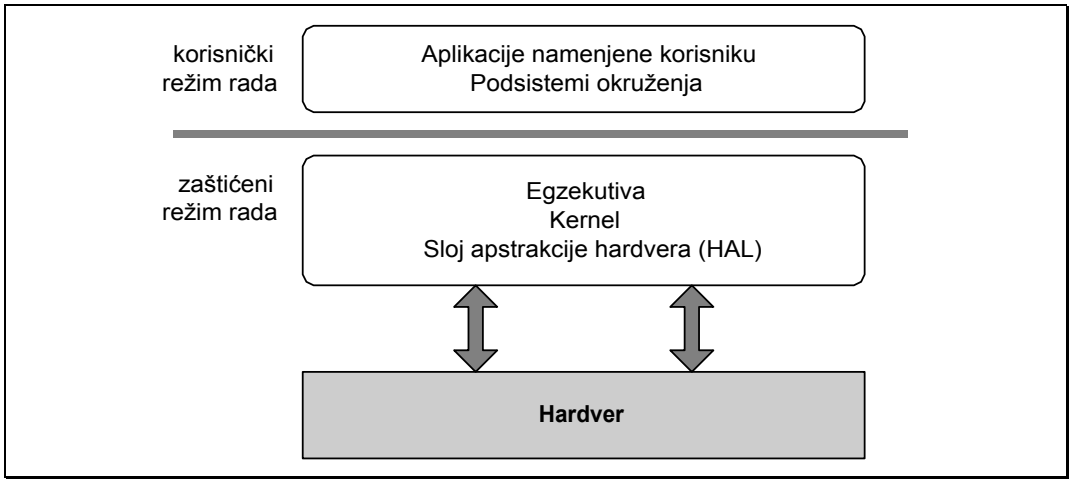
Arhitektura Windows NT operativnog sistema

Windows NT je višeprocesni operativni sistem sa pretpražnjenjem, sposoban da iskoristi prednosti višeprocesorske arhitekture.

Windows NT je slojeviti sistem koji se sastoji od modula. HAL, kernel i egzekutiva rade u zaštićenom režimu rada (*protected* mode), a kolekcija podsistema u korisničkom režimu rada (*user mode*). Podsistemi okruženja (MS-DOS podsistem, 16-bit Windows podsistem, 32-bit Windows podsistem, POSIX podsistem) emuliraju (oponašaju) različite operativne sisteme. Zaštitni podsistemi obezbeđuju sigurnosne funkcije.

Windows NT obezbeđuje 2GB adresnog prostora za aplikacije i 2GB za sam operativni sistem. U Windows NT ugrađena je podrška za nekoliko tipova datoteka, uključujući i NTFS. Mrežni servisi su integrisani u sistem.

Na slici 16.1. prikazana je arhitektura Windows NT sistema:



Slika 16.1. *Arhitektura Windows NT operativnog sistema*

Sloj apstrakcije hardvera (HAL)

Sloj apstrakcije hardvera (HAL - *Hardware Abstraction Layer*) rasterećuje operativni sistem poznavanja detalja vezanih za konkretan hardver. HAL je sloj posrednik između kernela i hardvera, koji višim slojevima operativnog sistema obezbeđuje podršku za rad sa procesorom i simetrično multiprogramiranje. Keš za instrukcije, keš za podatke i sve funkcije vezane za tajmer, prekide, konkretan *firmware* i rukovanje greškama, zavise od konkretne računarske arhitekture i kao takve se na nivou HAL-a prevode u API razumljiv višim slojevima operativnog sistema.

Kernel

Kernel Windows NT operativnog sistema obavlja sledeće funkcije.

- dodela procesora nitima,
- rukovanje hardverskim prekidima (kernel prosleđuje prekide odgovarajućim nitima i drajverima) i hardverski i softverski generisanim izuzetcima (kao što je pristup nepostojećoj memorijskoj lokaciji),
- sinhronizacija procesora niskog nivoa ,
- oporavak posle greške ili kvara u napajanju

Kernel obezbeđuje egzekutivu API-jem za pristup objektima kernela, kao što su događaji, *mutex*i, semafori i tajmeri.

Windows NT egzekutiva

Na slici 16.2 prikazana je egzekutiva (executive) Windows NT sistema.

Svaka komponenta egzekutive obezbeđuje skup API-ja ka višim slojevima operativnog sistema. Neki API-ji su dizajnirani tako da ih mogu pozvati procesi koji rade u korisničkom režimu rada; ostali API-ji su vidljivi isključivo iz zaštićenog režima, tj. iz ekutive.

[1] Upravljanje objektima

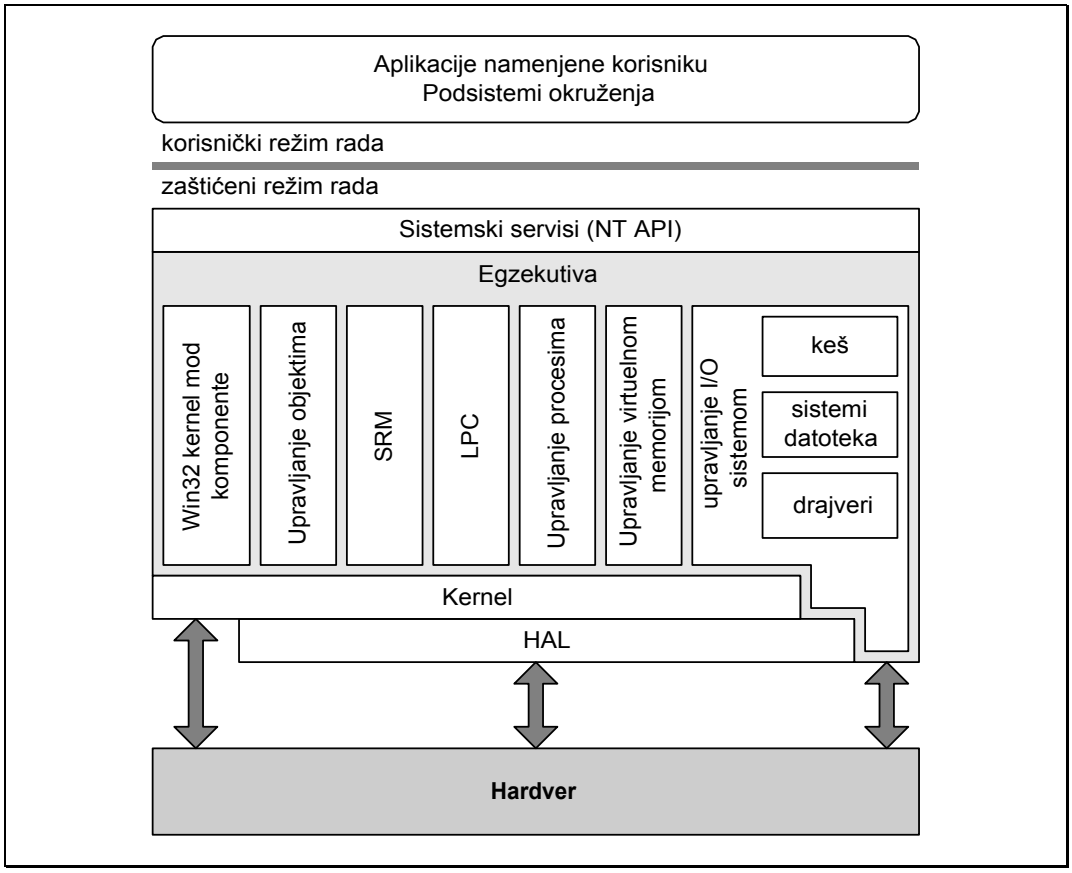
Objekat je reprezentacija konceptualne ili fizičke celine u sistemu. U objekte spadaju datoteke, direktorijumi, procesi, niti, objekti sinhronizacije (semafori, *mutex*i, događaji, tajmeri) i objekti virtuelne memorije. Može se reći da je gotovo sve nad čim NT radi ili sve što NT koristi za rad objekat.

Jedan objekat je u potpunosti opisan svojim: tipom, skupom akcija koje se nad njim mogu izvesti (npr. kreiranje, brisanje, promena neke vrednosti), atributima, relacijama sa drugim objektima i *security decriptor*om. Objekat se štiti identifikatorom vlasnika i listama za kontrolu pristupa, pomoću kojih se određenim korisnicima dozvoljava ili zabranjuje pristup objektu, odnosno prati pristup objektu (*audit*).

[2] Kontrola pristupa

Security Reference Monitor (SRM) obavlja kontrolu pristupa objektima, odnosno dozvoljava pristup objektima samo ovlašćenim korisnicima i obezbeđuje funkcionalnost praćenja pristupa objektu (*auditing*). Upravljač objekta koristi usluge koje pruža SRM - pre izvršenja neke akcije, upravljač objekta konsultuje SRM i tako određuje da li proces ima pravo da izvrši akciju koju želi nad objektom. SRM implementira C2 nivo sigurnosti:

- vlasnik objekta određuje ko može pristupiti objektu,
- identifikacija korisnika obavlja se pomoću jedinstvenog korisničkog imena i lozinke,
- mogućnost praćenja uspešnih i neuspešnih pokušaja pristupa objektima i
- sadržaj memorije i datoteka zaštićen je od neovlašćenog pristupa.



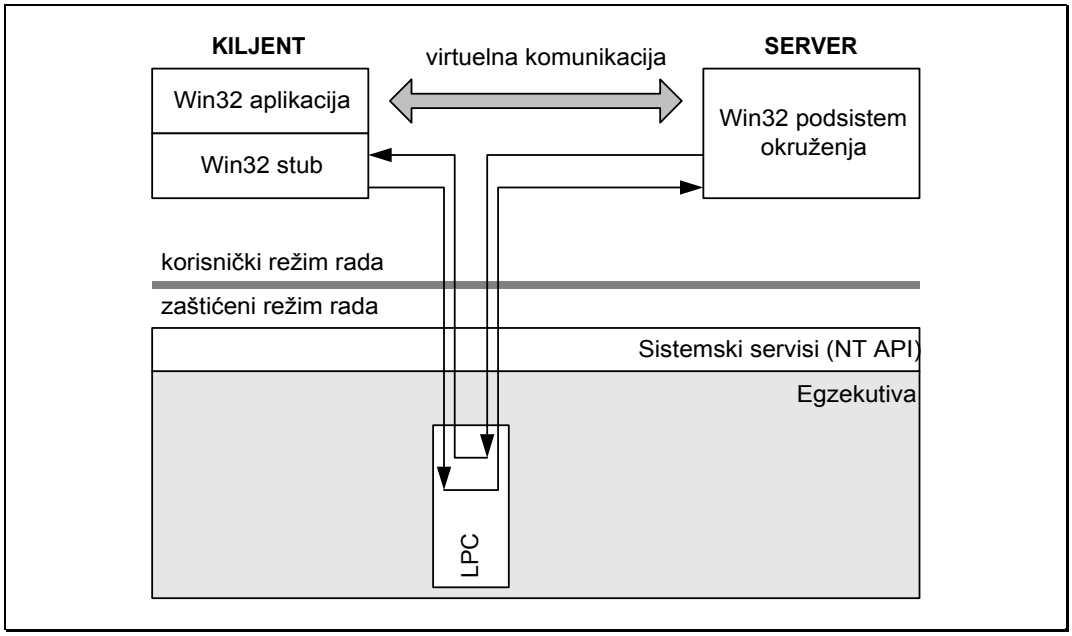
Slika 16.2. *Windows NT egzekutiva*

[3] Upravljanje procesima

Ovaj deo operativnog sistema (*process manager*) obezbeđuje usluge za kreiranje, brisanje i upotrebu niti i procesa. Raspoređivanje procesa, odnosno dodela procesora obavlja se na nivou kernela, a ne u egzekutivi. Proces je predstavljen odgovarajućim objektom, koji čine niti, virtuelni adresni prostor procesa i vidljivi resursi. NT ne vodi informacije o relacijama roditelji - deca između procesa.

[4] LPC

Lokalno pozivanje procedura (*Local Procedure Call - LPC*) je mehanizam koji obavlja prosleđivanje zahteva i rezultata između klijent i server procesa unutar jednog sistema (slika 16.3). Na ovaj način jedan podsistem može zahtevati usluge drugog NT podsistema.



Slika 16.3 *Windows NT LPC*

[5] Upravljanje virtuelnom memorijom

Windows NT obezbeđuje 2GB straničnog adresnog prostora za aplikacije i 2GB za sam operativni sistem. U sistemski adresni prostor smeštaju se rezidentni delovi operativnog sistema. Dizajn upravljača virtuelne memorije podrazumeva da osnovni hardver podržava mapiranje virtuelne memorije u fizičku i mehanizam straničenja. Kako je NT raspoloživ za nekoliko procesorskih arhitektura, parametri kao što je veličina stranice i mehanizmi zaštite stranica zavise od konkretnog tipa procesora.

Upravljač virtuelne memorije obavlja sledeće funkcije:

- implementira strategiju punjenja stranice na zahtev (demand paging, DP). Koristi se modifikacija DP algoritma sa lokalnošću reference (*clustering*),
- vodi evidenciju o slobodnim okvirima i veličini radnog skupa,
- implementira strategiju zamene stranica (određuje koja će stranica biti uklonjena iz operativne memorije),

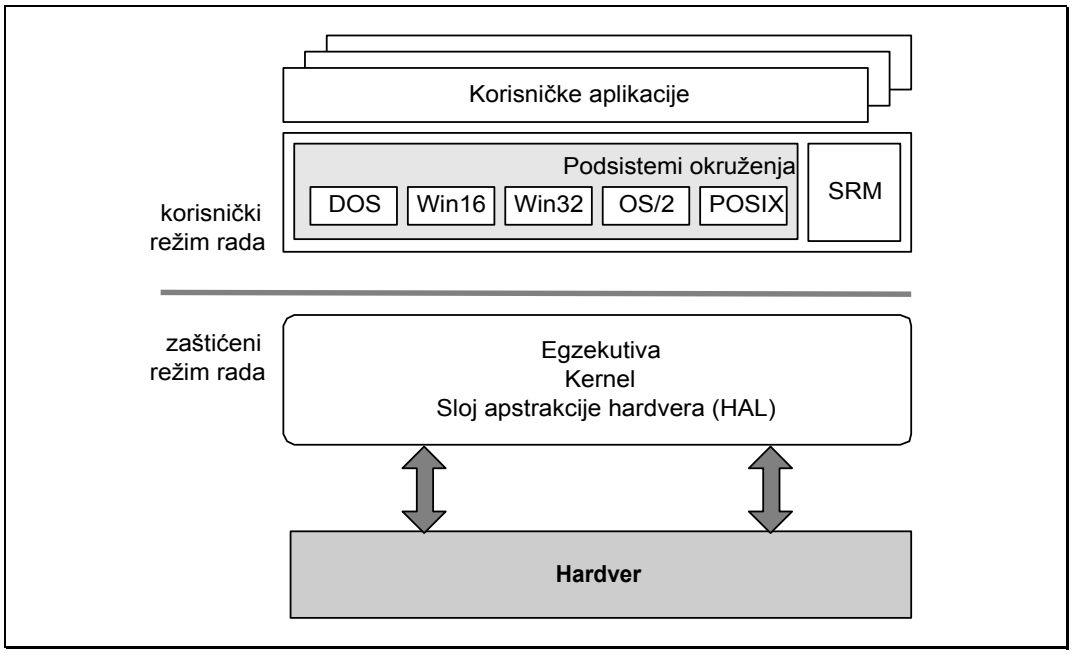
- implementira deljive memorijske segmente (koje koristi LPC za razmenu velikih poruka) i
- obezbeđuje podršku za memorijski mapirane datoteka.

[6] Upravljanje ulazno-izlaznim podsistemom

Ovaj deo egzekutive obezbeđuje podršku za razne uređaje, mrežu i sisteme datoteka (FAT; HPFS, NTFS i ISO9660 CD-ROM sistem datoteka). Drajveri se dinamički mogu napuniti i izbaciti iz memorije, pokrenuti i zaustaviti, bez zaustavljanja i ponovnog pokretanja sistema.

Podsistemi okruženja

Podsistemi okruženja su programi koji rade u korisničkom režimu rada i pružaju servise ka aplikacijama (slika 16.4). Ovi servisi simuliraju usluge koje pruža specifičan operativni sistem. Podsistemi okruženja implementirani su u Windows NT kao klijent-server model; aplikacija koja traži servis je klijent, a podsistem okruženja server.



Slika 16.4. *Podsistemi okruženja*

[1] Win32 podsistem.

Ovaj podsistem implementira Win32 API. Svaka Win32 aplikacija se izvršava u svom privatnom adresnom prostoru, odvojenom od adresnog prostora dodeljenog

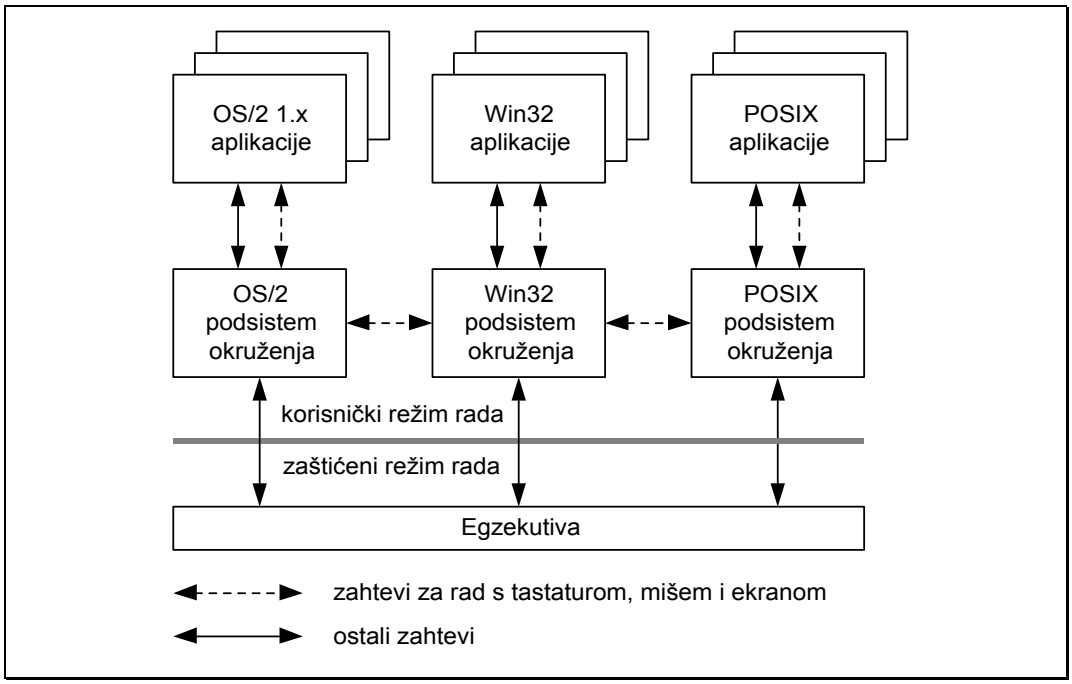
Win32 podsistemu. Win32 upravlja ulaznom (tastatura i miš) i izlazom (ekran). Ponaša se kao server za sve ostale podsisteme.

[2] OS/2 podsistem.

Omogućava rad aplikacijama namenjenim za OS/2 1.x (zaključno sa verzijom 1.3) koje rade u tekstualnom režimu rada. Svaka OS/2 aplikacija se izvršava u svom adresnom prostoru, odvojenom od adresnog prostora dodeljenog OS/2 podsistemu.

[3] POSIX podsistem.

Omogućava rad POSIX aplikacijama koje rade u tekstualnom režimu rada. Svaka POSIX aplikacija se izvršava u svom adresnom prostoru, odvojenom od adresnog prostora dodeljenog POSIX podsistemu.



Slika 16.5. *Win32. OS/2 i POSIX podsistemi okruženja*

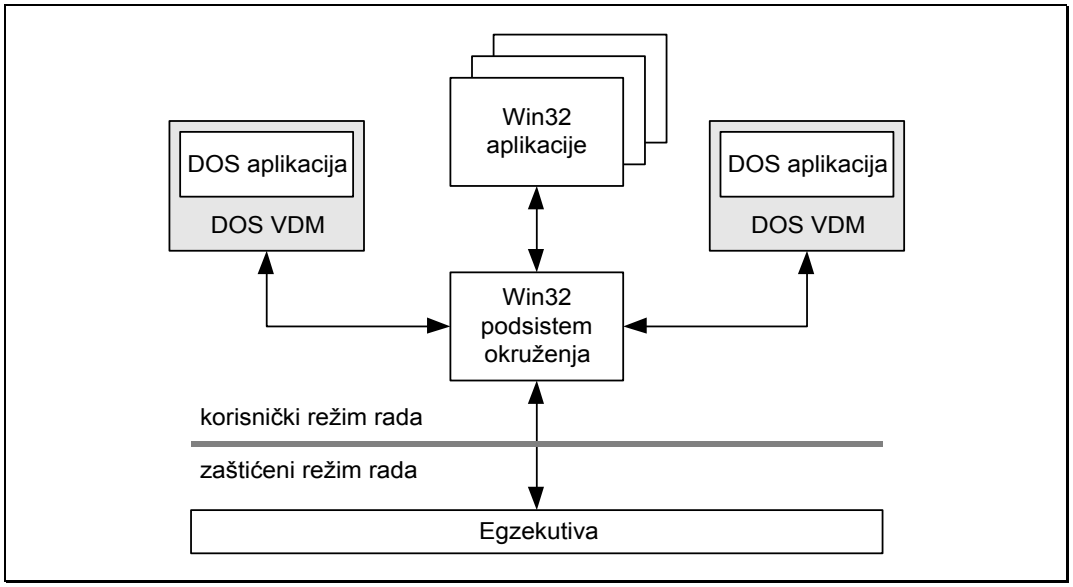
[4] MS-DOS okruženje.

Svaka DOS aplikacija se povezuje sa jednom virtuelnom DOS mašinom (VDM - *Virtual DOS Machine*), koja praktično simulira jedan PC i obezbeđuje:

- mehanizam za izvršenje Intel x86 instrukcija,
- simulirane DOS INT 21 servise,

- simulirane DOS BIOS prekidne servise i
- pristup standardnim uređajima PC računara.

DOS aplikacije se izvršavaju u privatnom adresnom prostoru.



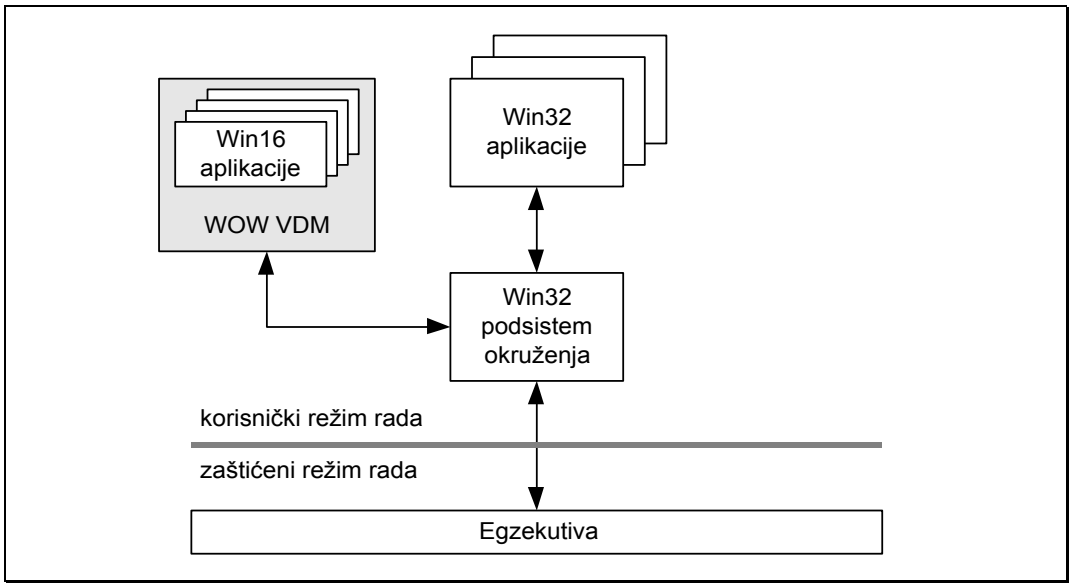
Slika 16.6. *MS-DOS okruženje*

[5] 16-bitno Windows okruženje

Podršku za sve Win16 aplikacije pruža jedna virtuelna DOS mašina - Windows on Windows (WOW, Win16 on Win32). Kao i VDM, WOW simulira PC i može se shvatiti kao virtuelna Windows mašina. WOW obavlja dvosmernu konverziju između Win16 i Win32 API-ja i poruka. Ono što se prevede izvršava se na Win32 podsistemu.

WOW se izvršava u zasebnom adresnom prostoru, odvojenom od ostalih procesa koji se izvršavaju u korisničkom režimu rada.

Svaka Win16 aplikacija dodeljuje se posebnoj niti WOW VDM procesa. Svaka WOW nit mora prepustiti kontrolu (shodno pravilima za izvršenje 16-bitnih Windows aplikacija, nasleđenih sa Windows 3.x sistema) pre nego što se dozvoli izvršenje druge WOW niti. Kolaps jedne Win16 aplikacije može izazvati kolaps cele virtuelne mašine (odnosno kolaps svih ostalih Win16 aplikacija), ali nema uticaj na Win32 aplikacije.



Slika 16.7. *16-bitno Windows okruženje*

Sigurnosni podsistem

Sigurnosni podsistem čine sledeće komponente:

[1] *Local Security Authority* (LSA)

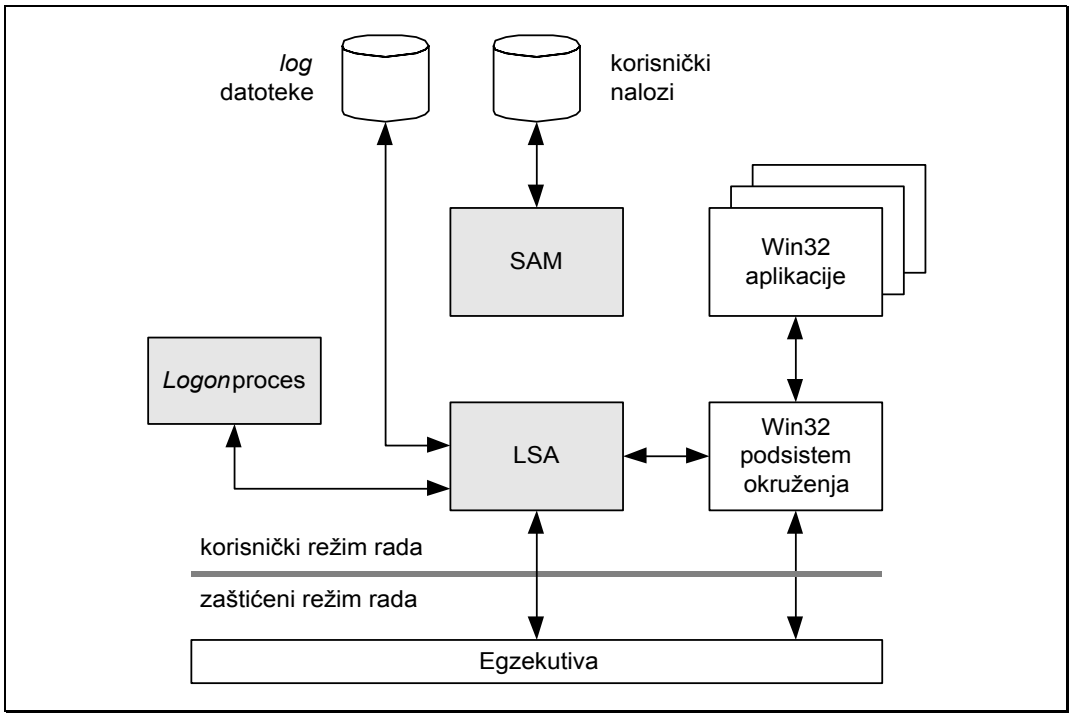
LSA upravlja sigurnosnom polisom računara. Generiše pristupne žetone (*access token*), odnosno informacije koje se dodaju procesima i koje određuju čemu korisnik može da pristupi. LSA upravlja polisom praćenja pristupa resursima. Komunicira sa *Security Reference Monitor*om u kernelu i vodi evidenciju o porukama vezanim za praćenje objekata koje šalje kernel.

[2] *Logon* proces

Prihvata zahteve za prijavljivanje na sistem; komunicira sa LSA kako bi odredio da li korisnik koji želi da se prijavi na sistem ima na to pravo ili ne.

[3] *Security Account Manager* (SAM)

Upravlja i bazom podataka u kojoj su opisani korisnici i grupe i obezbeđuje pristup toj bazi.



Slika 16.8. *Sigurnosni podsistem*

16.3. Windows 2000

Windows 2000, objavljen u februaru 2000. godine, je 32-bitni višeprocesni operativni sistem sa pretpražnjenjem za Intelove mikroprocesore. Ključni ciljevi razvoja ovog sistema su:

- portabilnost,
- sigurnost,
- uskladjenost sa POSIX-om,
- višeprocesorska podrška,
- proširivost,
- internacionalna podrška i
- kompatibilnost sa ranijim MS-DOS i MS Windows aplikacijama.

Windows 2000 koristi mikrokernel arhitekturu. U najznačajnija unapređenja Windows 2000 operativnog sistema u odnosu na Windows NT 4.0 spadaju:

- aktivni direktorijum, odnosno X.500/LDAP zasnovani direktorijumski servis,
- podrška za veći broj procesora i veću količinu operativne memorije i
- unapređenja mrežnog podsistema i sigurnosti.

Windows 2000 je raspoloživ u 4 verzije: Professional, Server, Advanced Server, National Server.

Principi dizajna

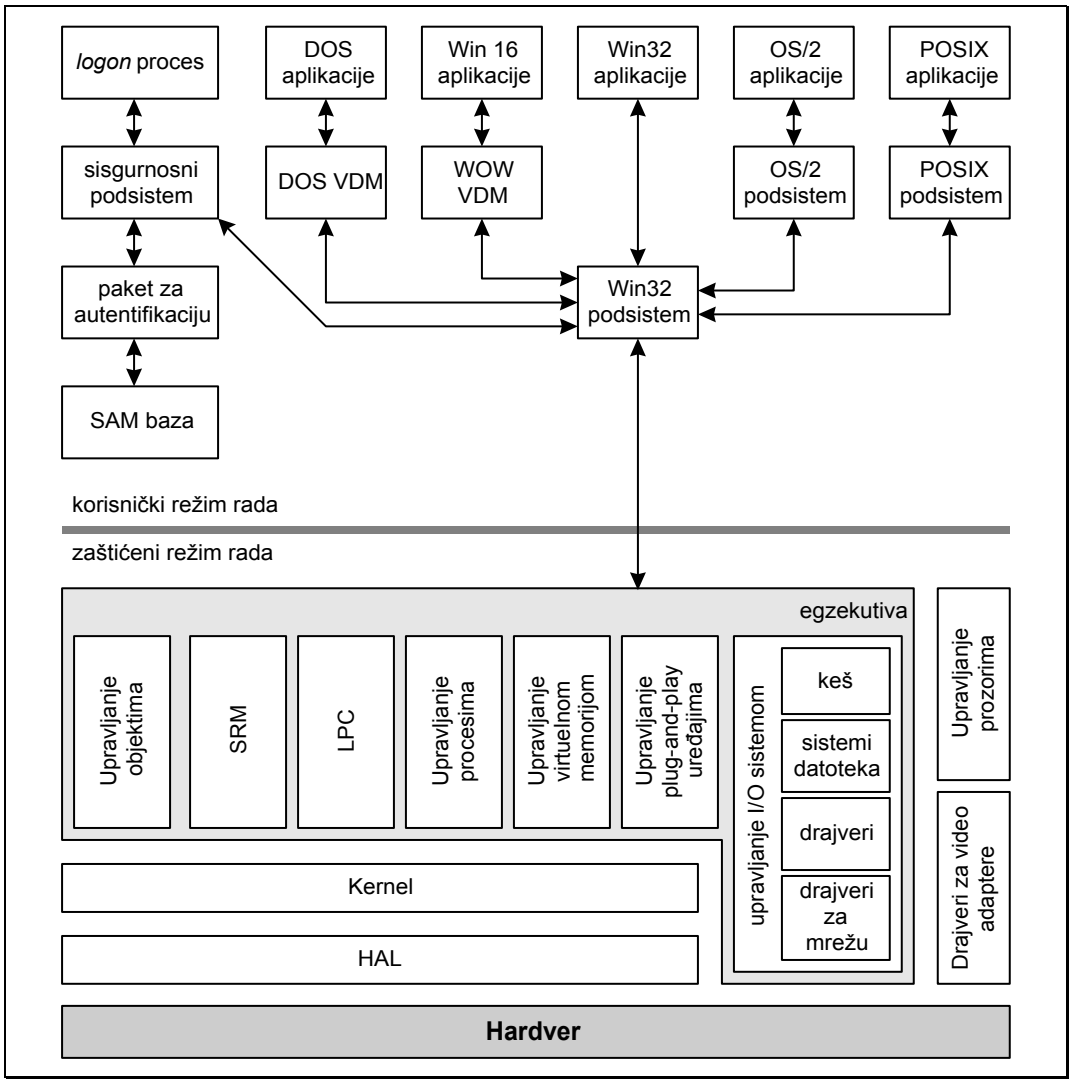
Proširivost i slojevita arhitektura je važan princip kod dizajna ovog operativnog sistema. Egzekutiva (*executive*) koja se izvršava u zaštićenom režimu, obezbeđuje osnovne sistemske usluge. Na vrhu egzekutive, nekoliko serverskih podsistema radi u korisničkom režimu. Modularna struktura omogućava da dodatni podsistemi okruženja budu dodavani bez uticaja na egzekutivu. Takođe, portabilnost je značajan princip, što znači da Windows 2000 može biti preseljen sa jedne arhitekture na drugu sa relativno malo izmena. Windows 2000 je napisan u programskim jezicima C i C++. Kod zavisan od procesora je izolovan u DLL biblioteke koje čine sloj apstrakcije harvera (*hardware abstraction layer* - HAL).

Pouzdanost je, takođe, važna karakterisitka Windowsa 2000. Ovaj operativni sistem koristi hardversku zaštitu virtuelne memorije i softverske metode zaštite za resurse operativnog sistema. U pogledu kompatibilnosti, aplikacije koje slede standard IEEE 1003.1 (POSIX) mogu biti kompajlirane i na Windowsu 2000 i pokrenute bez izmena izvornog koda.

Podsistemi Windowsa 2000 mogu komunicirati jedan sa drugim preko visoko performansnih mehanizama prosledjivanja poruka (*message passing*). Pretpražnjenje niskoprioritetnih niti omogućava sistemu da brzo odgovori na spoljne događaje. Ovaj sistem je takođe dizajniran za simetricno multiprocesiranje. Internacionalna podrška podrazumeva mogućnost različitih lokalnih postavke i podešavanja korišćenjem aplikativnog programskog interfejsa podrške za nacionalne jezike (*national language support* - NLS API).

Arhitektura Windows 2000 operativnog sistema

Windows 2000 je slojeviti sistem koji se sastoji od modula. Postoji takozvani zaštićeni režim rada (*protected* mode) u kome rade HAL, kernel i egzekutiva. Sledeći režim rada je takozvani korisnički režim (*user mode*) u kome radi kolekcija podsistema. Podsistemi okruženja (MS-DOS podsistem, 16-bit Windows podsistem, 32-bit Windows podsistem, POSIX podsistem) emuliraju (oponašaju) različite operativne sisteme. Zaštitni podsistemi obezbeđuju sigurnosne funkcije.



Slika 16.9. *Arhitektura Windows 2000 operativnog sistema*

Kernel, procesi i niti

Kernel je osnova (temelj) za egzekutivu i za podsisteme. On se nikad ne smešta izvan memorije i njegovo izvršenje nikada ne biva pretpražnjeno. Četiri osnovna zadatka kernela su:

- dodela procesora nitima,
- rukovanje prekidima i izuzetcima,

- sinhronizacija procesora niskog nivoa i
- oporavak posle greške ili kvara u napajanju.

Kernel Windows 2000 operativnog sistema je objektno orijentisan i koristi dva skupa objekata:

- dispečer objekat koji upravlja poslovima dodele procesora tj. rasporedjivanjem, kao i sinhronizacijom (dogadjaji, mutex-i, semafori, niti i tajmeri) i
- objekti upravljanja (asinhroni pozivi procedura, prekidi, notifikacija o napajanju, procesi i profili).

Svaki proces ima virtuelno memorijsko adresno podrucje, podatke (kao što je bazni prioritet) i afinitet ka jednom ili vise procesora. Niti su izvršne jedinice raspoređivane od strane kernelovog dispečera. Svaka nit ima svoje vlastito stanje, uključujući prioritet, procesorski afinitiet i obračunsku informaciju. Nit može biti u jednom od šest stanja: u stanju spremnosti (*ready*), pripravnosti (*standby*), izvršavanja (*running*), na čekanju (*waiting*), u stanju tranzicije (*transition*) i stanju završetka ili prekida rada (*terminated*).

Dispečer koristi 32-nivovsku šemu prioriteta da bi odredio redosled izvršenja niti. Prioriteti su podeljeni u dve klase:

- klasa koja se izvršava u realnom vremenu (*real-time class*) koja sadrži niti sa prioritetima rangiranim od 16 do 31 i
- varijabilna klasa koja sadrži niti koje imaju prioritete od 0 do 15.

Strategija prioriteta Windowsa 2000 karakteriše se veoma dobrim vremenom odgovora (*response time*) za interaktivne niti. Na primer, kvantum raspoređivanja procesa u klasi NORMAL_PRIORITY_CLASS pomerenih iz pozadine u prvi plan, uvećava se za neki faktor, čime se obezbeđuju solidne performanse izvršenja najvećeg broja interaktivnih niti.

Takođe, omogućeni su "skokovi" na izvršenje niti koje koriste ulazno-izlazne uređaje kako bi se ulazno-izlazni uređaji držali zauzetim (zbog male brzine transfera). Kernel dinamički podešava prioritet niti zavisno od toga da li prvenstveno koriste ulazno-izlazne operacije ili procesorsko vreme.

Akcija rasporedjivanja (dodele) se dešava kada nit uđe u stanje čekanja na procesor ili stanje čekanja na resurs, kada se nit terminira ili kada aplikacija promeni prioritet niti ili dodeljeni procesor.

Niti koje se izvršavaju u realnom vremenu imaju prioritetni pristup centralnom procesoru, ali Windows 2000 ne garantuje da će *real-time* niti startovati sa izvršenjem u bilo kojim posebno određenim vremenskim granicama. Ovo je poznato kao „meko realno vreme" (engl. *soft realtime*).

Vlakno (engl. *fiber*) je kod koji radi u korisničkom režimu i koji biva raspoređen, saglasno algoritmu definisanom od strane korisnika. Samo jedno "vlakno" se može

izvršavati u jednom trenutku vremena, čak i na višeprocesorskom hardveru. Windows 2000 sadrži vlakna da bi olakšalo portovanje nasleđenih Unix aplikacija koje su napisane za *fiber execution model*.

Komunikacija između procesa

Win32 aplikacije ostvaruju interprocesnu komunikaciju deljenjem kernel objekata. Da bi se sinhronizovao konkurentni pristup deljenim objektima od strane niti, kernel obezbeđuje objekte sinhronizacije, kao što su semafori i *mutex*i.

Jedno alternativno značenje interprocesne komunikacije je prosleđivanje poruka, koje je posebo popularno za Windows GUI aplikacije. Jedna nit šalje poruku drugoj niti ili prozoru, pri čemu se sa porukom mogu poslati i podaci.

Svaka Win32 nit ima svoj vlastiti ulazni red is koga nit dobija poruke. Ovo je mnogo pouzdanije nego deljivi ulazni red 16-bitnog prozora zato što sa posebnim redovima jedna zakočena aplikacija ne može blokirati ulaz za ostale aplikacije.

Rukovanje prekidima i zamkama

Kernel upravlja prekidima pomoću rutina za opsluživanje prekida ili pomoću internih rutina u kernelu. Prekidi se predstavljaju pomoću objekata (*interrupt object*) koji sadrže sve informacije neophodne za upravljanje prekidom. Broj i vrste prekida različiti zavise od konkretne procesorske arhitekture. Radi obezbeđivanja nužne portabilnosti, prekidni dispečer (*interrupt dispatcher*), koji je u teoretskom modelu operativnog sistema prikazan prvim nivoom obrade prekida (*FLIH - first level interrupt handler*), organizuje prekide u standardne skupove. Prekidima se dodeljuju prioriteti na osnovu kojih se kasnije vrši opsluživanje prekida. Od 32 prekidna nivoa, osam rezerviše kernel za internu upotrebu, a ostalih 24 predstavljaju hardverske prekide koji se opslužuju preko HAL-a.

Kernel koristi specijalnu tabelu (*interrupt dispatch table*) pomoću koje svakom prekidu dodeljuje odgovarajuću prekidnu rutinu. Ukoliko je računar višeprocesorski, posebna tabela se kreira za svaki procesor. Prekidna rutina se izvršava samo ukoliko je višeg prekidnog prioriteta od od onog koji je trenutno dodeljen procesoru, a u suprotnom čeka da kernelska nit smanji prekidni prioritet procesora. Na ovaj način omogućava se izvršavanje sistemskih funkcija kao što su sinhronizacija niti sa ulazno-izlaznim operacijama.

Kernel obezbeđuje rukovanje zamkama (*trap handling*) kada su izuzeci (*exceptions*) i prekidi (*interrupts*) generisani od strane hardvera ili softvera. Izuzetke koji ne mogu biti obrađeni od strane rukovaoca zamkama (*trap handler*), obrađuje kernelov dispečer izuzetaka (*exception dispatcer*). Dispečer prekida u kernelu rukuje prekidima tako što pozove prekidnu servisnu rutinu kao što je na primer rukovaoc

uređajem (*device driver*) ili internu kernel rutinu. Kernel koristi *spin locks* koji se nalaze u globalnoj memoriji da bi postigli višeprocesorsko međusobno isključenje.

Egzekutiva

Navodimo značajnije funkcije egzekutive (*executive*) Windows 2000 sistema:

[1] Upravljanje objektima

Windows 2000 koristi objekte za sve svoje usluge (servise) i entitete; upravljač objekata nadzire korišćenje objekata:

- generiše *handle* objekta,
- proverava sigurnost i sigurnosna ograničenja i
- vodi računa tj. prati i evidentira koji procesi koriste koje objekte.

Manipulacija objektima se obavlja standardnim skupom metoda, a to su: *create, open, close, delete, query name, parse* i *security*.

Egzekutiva Windowsa 2000 dozvoljava da svakom objektu bude dato ime, koje može biti stalno tj. permanentno ili privremeno. Imena objekata su struktuirana kao imena putanja u MS-DOS operativnom sistemu ili Unixu.

Proces ostvaruje pristup određenom kernel objektu pozivanjem funkcije koja otvara *handle* ka tom objektu. *Hande* je jedinstven za proces. Proces može zatvoriti odgovarajući *handle* pozivanjem CloseHandle funkcije. Operativni sistem može obrisati objekat ako broj procesa koji ga koriste padne na 0.

Proces može dobiti *handle* objekta ukoliko kreira novi objekat ili deli postojeći objekat sa drugim procesom. Windows 2000 obezbeđuje tri načina za deljenje objekata između procesa:

- otvori postojeći objekat (proces daje objektu ime prilikom kreiranja, a zatim drugi proces otvara objekat po tom imenu),
- proces nasleđuje *handle* objekta od procesa roditelja i
- dobije *handle* duplikat od drugog procesa (duplikat se kreira funkcijom DuplicateHandle).

Svaki objekat je zaštićen listom za kontorlu pristupa (*access control list*).

[2] Kontrola pristupa (*Security Reference Monitor*)

Objektno orijentisana priroda Windowsa 2000 omogućava korišćenje uniformnog mehanizma za obavljanje provere pristupnih prava u vreme izvršenja, kao i evidentiranje i reviziju kontrole za svaki entitet u sistemu. Uvek kada proces otvori *handle* (rukovaoc) za neki objekat, monitor sigurnosti

proverava sigurnosni žeton procesa i pristupnu kontrolnu listu (*access control list*) da vidi da li proces ima potrebna prava.

[3] Upravljanje procesima

Ovaj deo operativnog sistema (*process manager*) obezbeđuje usluge za kreiranje, brisanje i upotrebu niti i procesa. Problemi kao što su relacije roditelji – deca ili hijerarhija procesa su ostavljeni posebnim podsistemima okruženja koji su vlasnici procesa.

Proces se kreira CreateProcess rutinom koja puni memoriju eventualno potrebnim bibliotekama sa dinamičkim povezivanjem. Na taj način se kreira primarna nit (*primaty thread*). Dodatne niti mogu biti kreirane korišćenjem CreateThread funkcije. Svaki DLL ili izvršni fajl koji se napuni u adresni prostor procesa identifikuje se pomocu *instance handle*-a.

[4] Lokalno pozivanje procedura

Lokalno pozivanje procedura (*Local Procedure Call - LPC*) je mehanizam koji obavlja prosleđivanje zahteva i rezultata između klijent i server procesa unutar jedne mašine. Ovo se posebno koristi za zahteve za uslugama od različitih Windows 2000 podsistema.

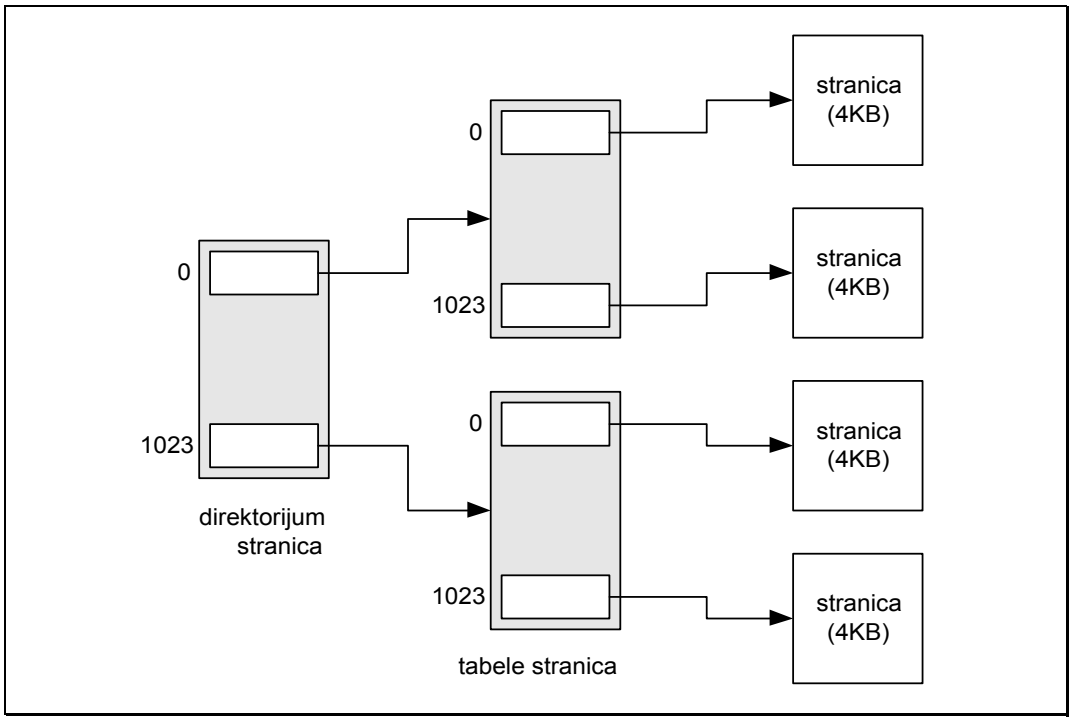
Kada se kreira LPC kanal, jedna od tri tipa tehnika za prosleđivanje poruka može biti korišćen:

- Prvi tip koji je pogodan za male poruke, do 256 bajtova; red čekanje poruka porta se koristi kao međuspremište i poruke se kopiraju iz jednog procesa u drugi.
- Drugi tip omogućava da se izbegne kopiranje većih poruka tako što pokazuje na objekt memorijske sekcije u deljivoj memoriji koja je kreirana za kanal
- Treći metod poznat pod nazivom brzi LPC (*quick LPC*) je bio koričćen od strane grafičkog displeja Win32 podsistema.

[5] Upravljanje virtuelnom memorijom

Dizajn upravljača virtuelne memorije (*Virtual Memory Manager, VM manager*) podrazumeva da osnovni harver koji "leži" ispod operativnog sistema podržava mapiranje virtuelne memorije u fizičku, mehanizam straničenja, transparentno pripajanje keš memorije na višeprocesorskim sistemima i višestruke pokazivače u tabeli stranica (*page table*) na iste memorijske stanice (*virtual addressing aliasing*). VM manager u Windowsu 2000 koristi upravljačku šemu baziranu na straničenju sa stranicom veličine 4 KB. Dodela memorije procesu obavljaja se u dva koraka:

- najpre se rezerviše deo adresnog prostora procesa,
- zatim se procesu dodeljuje prostor u Windows 2000 *paging* datoteci.

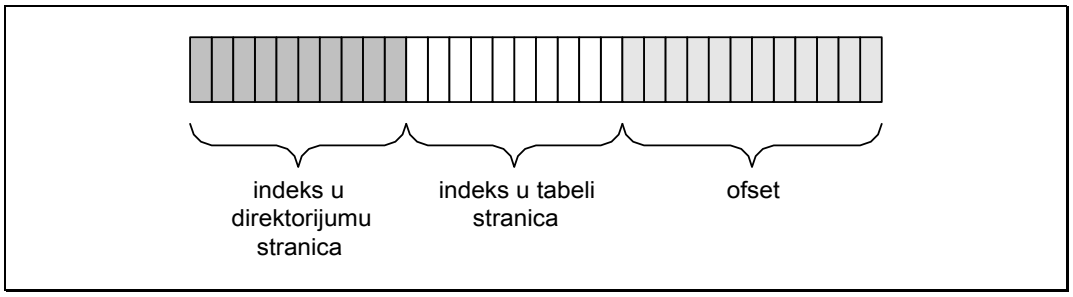


Slika 16.10. *Skica virtuelne memorije*

Windows 2000 koristi dvonivovsko straničenje, pri čemu se svakom procesu dodeljuje privatni adresni prostor, direktorijum stranica i odgovarajući broj tabela stranica:

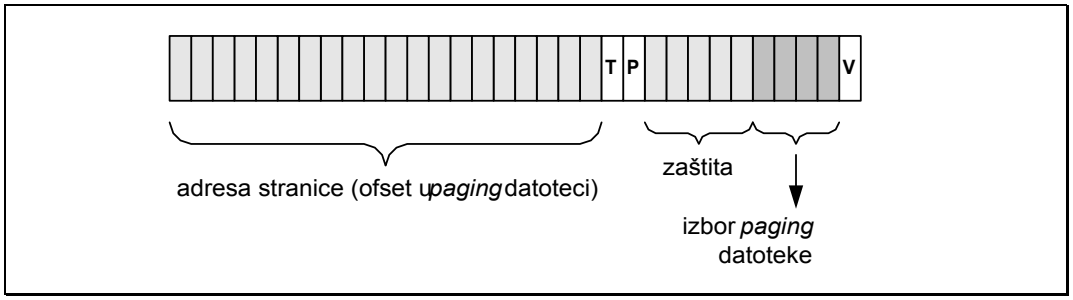
- svaki proces ima direktorjum stranica (*page directory*) koji sadrži 1024 zapisa veličine 4 bajta,
- svaki zapis u direktorijumu stranica pokazuje na odgovarajuću tabelu strana (*page table*) i
- svaka tabela stranica sadrži 1024 zapisa veličine 4 bajta koji pokazuju na okvire veličine 4KB u memoriji.

Jedan desetobitni ceo broj može reprezentovati sve vrednosti od 0 do 1023, i prema tome može odabrati bilo koji zapis (*entry*) u direktorijumu ili tabeli stranica. Ovo svojstvo se koristi za prevođenje virtuelnih u fizičke adrese. Prevođenje se obavlja na sledeći način: najviših 10 bitova virtuelne adrese biraju zapis u direktorijumu stranica i određuju tabelu stranica. Sledećih 10 bitova su indeks u tabeli stranica na osnovu kog se bira okvir u fizičkoj memoriji. Najnižih 12 bitova predstavljaju ofset u stranici koja je smeštena u odabranom okviru.



Slika 16.11. *Prevođenje virtuelnih u fizičke adrese*

Svaka strana može biti u jednom od šest stanja: ispravno (*valid*), "nulovano" (*zeroed*), slobodno (*free*), "u pripravnosti" (*standby*), modifikovano (*modified*) i loše (*bad*).



Slika 16.12. *Zapis u tabeli stranica*

Objasnićemo ukratko i strukturu tabele stranica u *paging* datoteci. Svaki zapis u tabeli stranica sadrži 20 bitova koji predstavljaju adresu stranice, odnosno ofset u *paging* datoteci, 4 bita za izbor *paging* datoteke, 3 bita koja određuju stanje stranice i 5 bitova za zaštitu stranice.

Izvršnim datotekama i memorijski mapiranim datotekama se ne dodeljuje prostor u *paging* datoteci jer su već smeštene na disku.

[6] Upravljanje Plug-and-Play uređajima

Plug-and-Play (PnP) manager se koristi da prepozna i podesi promene u harverskoj konfigiraciji. Kada se dodaju novi uređaji (npr. PCI ili USB), PnP *manager* napuni odgovarajući drajver. Ovaj deo egzekutive vodi evidenciju o resursima koje koriste pojedinačni uređaji.

[7] Egzekutiva i upravljanje ulazno-izlaznim podsistemom

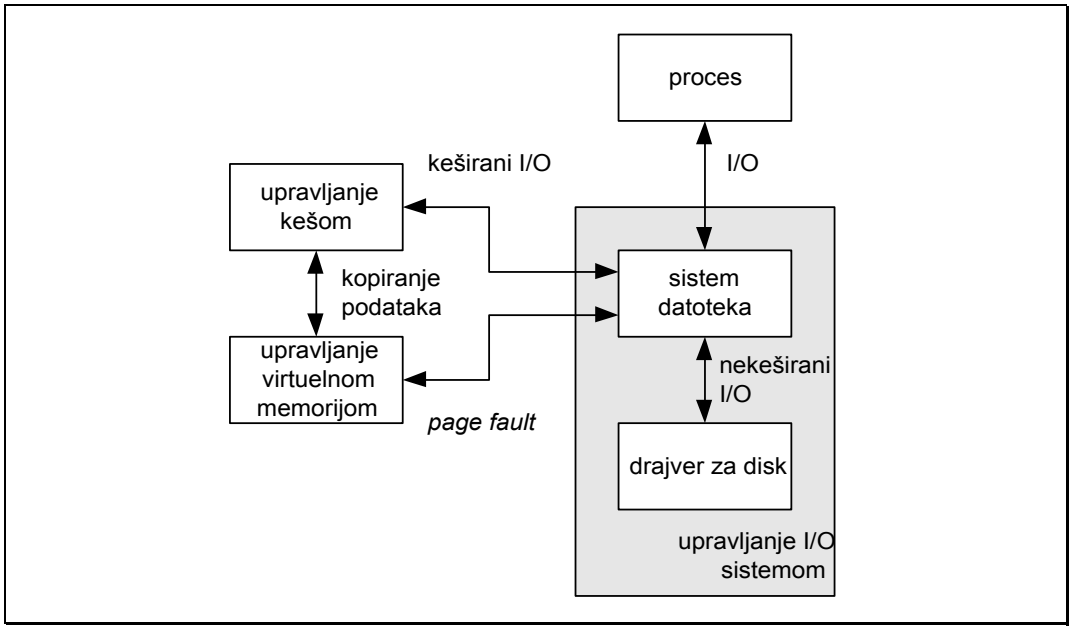
Upravljač ulazno-izlaznog podsistema (*I/O manager*) je odgovoran za:

- sisteme datoteka,

- upravljanje kešom,
- upravljanje drajverima uređaja i
- upravljanje mrežnim drajverima.

On vodi računa o tome koji su sistemi datoteka aktivirani i upravlja baferima za ulazno-izlazne zahteve (*I/O requests*). Upravljač ulazno/izlaznog podsistema radi sa upravljačem virtuelne memorije (*VM Manager*) da obezbedi memorijski mapiran fajl U/I. On takođe upravlja Windows 2000 keš upravljačem koji rukuje keširanjem za ceo ulazno/izlazni sistem. Podržava i sinhrone i asinhrone operacije, obezbeđuje tajmaute za drajvere i ima mehanizme koji omogućavaju da jedan drajver pozove drugi.

Navodimo bez daljeg objašnjenja skicu ulazno-izlaznog podsistema za datoteke:



Slika 16.13. *Ulazno-izlazni podsistem za datoteke*

Podsistemi okruženja i sigurnosni podsistem

Ovaj podsistem čine procesi koji rade u korisničkom režimu i nalaze se u sloju koji koristi usluge egzekutive, sa ciljem da omoguće izvršavanje programa razvijenih za drugi operativni sistem. Windows 2000 koristi Win32 podsistem kao glavno operativno kruženje. Win32 se koristi da pokrene sve procese. On takodje obezbeđuje sve mogućnosti i funkcije tastature, miša i grafičkog displeja.

MS-DOS okruženje je ostvareno pomoću Win32 aplikacije koja se zove *Virtual DOS Machine (VDM)*, a proces radi u korisničkom režimu. Prema njemu se mehanizmi straničenja memorije i dispečer odnose kao prema bilo kojoj Windows 2000 niti.

16-bitno Windows okruženje je omogućeno pomoću WOW VDM (*Windows on Windows*), koja se može shvatiti kao virtuelna Windows mašina. WOW obavlja dvosmernu konverziju između Win16 i Win32 API-ja i poruka. Ono što se prevede izvršava se na Win32 podsistemu.

POSIX podsistem je dizajniran da izvršava POSIX aplikacije sledeći POSIX.1 standard koji je baziran na Unix modelu.

OS/2 podsistem omogućava izvršavanje OS/2 aplikacija.

Podsistem za prijavljivanje (*logon*) i sigurnosni podsistem (*Security Subsystem*) autentifikuju korisnike koji se prijavljuju na Windows 2000 sisteme. Od korisnika se zahteva da imaju ime naloga i lozinke. Paket za autentifikaciju, autentifikuje korisnike uvek kada pokušaju da pristupe nekom objektu u sistemu. Kao podrazumevani paket za autentifikaciju, Windows 2000 koristi Kerberos.

Sistem datoteka

Windows 2000 koristi NTFS kao domaći sistem datoteka. Osnovna struktura NTFS sistema datoteka je volumen (*volume*), zasnovan na logičkoj disk particiji. Volumen može zauzeti deo diska, ceo disk, a može se i rasprostrati na više diskova. Osnovne jedinice za dodelu prostora na disku su klasteri (*clusters*), odnosno grupe koje čine 2^n sektora diska, koje se adresiraju korišćenjem logičkih brojeva klastera. S obzirom da je veličina klastera manja nego kod za 16-bitnog FAT sistema, interna fragmentacija je smanjena..

NTFS sistem, po svojoj strukturi, podseća na relacionu bazu podataka koja organizuje podatke na principu balansiranog B+ stabla. Prostor imena (NTFS *name space*) je organizovan hijerarhijom direktorijuma, pri čemu *index root* sadrži vrh (koren) B+ stabla.

Datoteka u NTFS sistemu datoteka je struktuirani objekat koji se sastoji od atributa. Svaka datoteka je opisana pomoću jednog ili više zapisa u području koje je smešteno u okviru posebne datoteke - *Master File Table* (MFT). MFT predstavlja prvu liniju svake NTFS particije, i ima svoju rezervnu kopiju. Ova redundansa osigurava oporavak sistemskih datoteka u slučaju oštećenja. Pokazivači u boot sektoru volumena opisuju lokaciju na kojoj se nalazi MFT. Svaka datoteka na jednom NTFS volumenu ima svoj jedinstveni 64-bitni identifikator (*unique ID*) koji se zove referenca datoteke (engl. *file reference*).

Datoteke i direktorijumi su objekti NTFS sistema datoteka i slične su strukture (NTFS tretira direktorijum kao specijalnu datoteku). Svaki objekat ima svoje ime i sadržaj. Sadržaj datoteke su konkretni podaci. Sadržaj direktorijuma čine indeksi

relativnih lokacija, veličina direktorijuma i bitmapirana reprezentacija strukture direktorijuma. NTFS podržava Unicode imena datoteka, dužine do 255 karaktera. Ovaj način imenovanja doprinosi jednostavnijoj internacionalizaciji operativnog sistema. Svaki objekat u sistemu datoteka osim imena opisuju i:

- standardni atributi, koje NTFS nasleđuje sa FAT sistema datoteka,
- prošireni atributi (*extended atributes*) koje dodaju programi pomoću kojih su datoteke kreirane i
- sigurnosni opis (*security descriptor*).

Sigurnost NTFS volumena je izvedena iz Windows objektnog modela, karakterističnog za Windows 2000/XP/2003. Svaki fajl objekat ima sigurnosni opis (*securiy descriptor*) koji je smešten u MFT. Ovaj atribut sadrži pristupni žeton vlasnika datoteke i pristupnu kontrolnu listu (*access control list*) koja utvrdjuje privilegije koje su dodeljene svakom korisniku koji ima pristup datoteci.

Sve promene na strukturi sistema datoteka koje se izvrše unutrašnjim transakcijama se prate na posebnom mestu. Transakcioni sistem funkcioniše na sledeći način:

[1] Pre nego se struktura podataka izmeni, transakcija upisuje *log* zapis koji sadrži *undo* i *redo* operacije

[2] Nakon što se struktura podataka promeni, *commit* zapis se upisuje u *log* da naznači da je transakcija bila uspešna.

[3] U slučaju pada tj. greške na sistemu, podatkovne strukture fajl sistema mogu biti oporavljene u konzistentno stanje procesiranjem *log* zapisa.

Ova šema ne garantuje da će svi korisnički podaci biti oporavljeni posle pada sistema ili greške, zapravo su meta podaci sistema datoteka neoštećeni i reflektuju neko konzistentno stanje pre pada. *Log* je smešten u treću *metadata* datoteku na početku volumena. Funkcija praćenja promena (*logging*) je obezbeđena preko Windows usluge *log file service*.

Jedna od mogućnosti NTFS sistema datoteka je i kompresija datoteka. U svrhu kompresije tj. „sabijanja“ datoteke radi smanjenja potrebnog prostora za smeštanje, NTFS deli podatke iz datoteke na jedinice kompresije (engl. *compression unit*) koje su u susštini blokovi od 16 kontinuiranih klastera. Za „raštrkane“ i „oskudne“ datoteke, tj. datoteke čiji se podaci nalaze u više manjih delova na različitim mestima na disku ili koji imaju kontinuirane blokove istovetnih podataka, NTFS koristi drugu tehniku da uštedi prostor:

- klasteri koji sadrže sve nule u suštini nisu alocirani na disku,
- umesto toga, praznine (engl. *gap*) su zadržane u sekvenci virtuelnih klastera smeštenih u MFT zapisu datoeke,

- kada se učitava fajl, ako se nađe praznina u virtuelnim klasterima, NTFS će samo napuniti nulama taj deo pozivačeovog bafera.

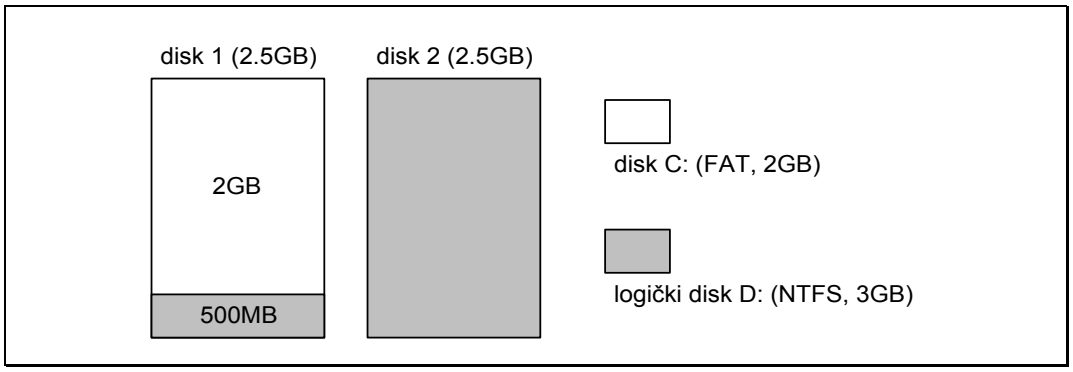
Još jedna značajna karakteristika NTFS sistema datoteka jesu tačke redirekcije (*reparse point*), odnosno 16KB podataka koji imaju formu taga kojim se identifikuje drajver uređaja. Akcija usmerena ka *reparse point*u preusmerava se na definisani drajver. Kao takav, *reparse point* se može iskoristiti da obezbedi funkcionalnost montiranja sistema datoteka. Dodatno, Windows 2000 realizuje simboličke linkove kao specijalnu vrstu *reparse point*a kojim se izvršava redirekcija NTFS procesa na određenu datoteku ili direktorijum.

Upravljanje diskovima i tolerancija otkaza

FtDisk (*fault tolerant disk driver*) je disk drajver koji obezbeđuje nekoliko načina da se više SCSI diskova kombinuje u jedan logički volumen tolerantan na otkaze.

[1] *Volume set*

Jedan način kombinovanja većeg broja diskova je logička konkatenacija, sa ciljem dobijanja većeg logičkog volumena. U Windows terminologiji, ovakva konkatenacija poznata je pod imenom *volume set* i može se formirati od najviše 32 diska ili particije.



Slika 16.14. *Volume Set*

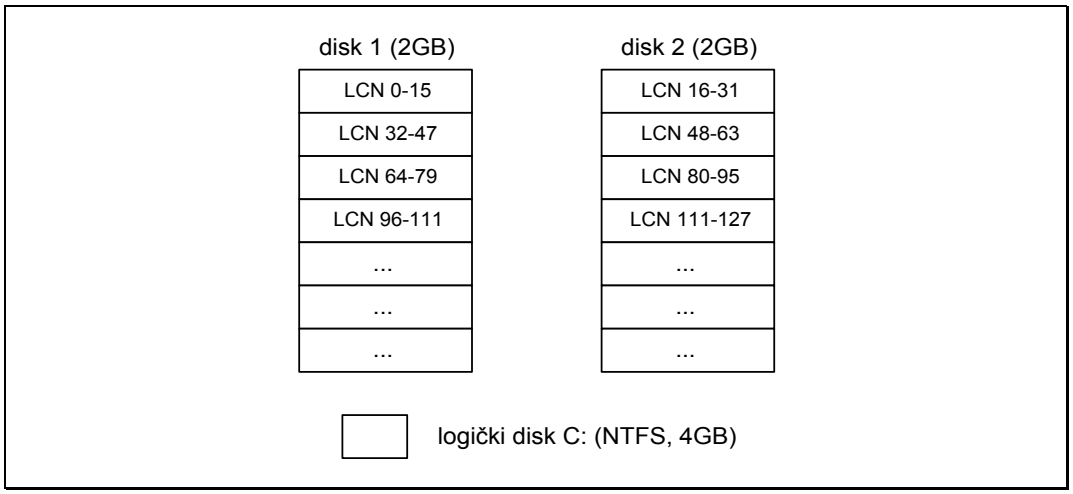
NTFS fajl sistem pristupa setu diskova istom metodom kojom pristupa i običnim particijama, odnosno diskovima - preko logičkih brojeva klastera (*LCN - logical cluster number*). Pri tom, FtDisk drajver pretvara broj logičkog klastera seta u odgovarajući broj sektora na određenom disku koji čini taj set. To znači da se *Volume Set* na kome se nalazi NTFS fajl sistem se može proširiti novim diskovima ili particijama bez gubitka podataka. Na sledećoj slici prikazan je set formiran od dva diska veličine 2.5GB. Disk 1 je izdeljen na dve particije, veličine 2GB i 500MB, koja je iskorišćena za kreiranje seta sa drugim diskom. Krajnji

korisnik vidi dva volumena, veličine 2GB (dostupan FAT ili NTFS fajl sistem) i veličine 3GB (*volume set*, dostupan isključivo preko NTFS fajl sistema).

[2] *Stripe set*

Veći broj diskova, odnosno particija se, takođe, može kombinovati formiranjem *stripe set*a, odnosno raspoređivanjem logičkih klastera na više diskova po round robin algoritmu. Ovaj metod rsapodele logičkih klastera se naziva i RAID nivo 0 (*redundant area of inexpressive disks*) ili *disk striping*. Naziv potiče od logike koja stoji iza samog postupka, odnosno deljenja volumena na manje delove (trake, *stripe*). FtDisk koristi veličinu trake (*stripe*) od 64KB: prvih 64KB logičkog volumena čuvaju se na prvom disku iz seta, sledećih 64KB na drugom, i tako redom, dok svi diskovi ne "prilože" 64KB za stripe set. Zatim se ponovo kreće od prvog diska. Ovaj metod fizičke raspodele podataka na više diskova koji mogu da vrše paralelni transfer značajno uvećava performanse, ali su u slučaju otkaza jednog diska podaci nepovratno izgubljeni.

Slika 16.15. ilustruje formiranje *Stripe Set*a veličine 4GB od dva diska veličine 2GB.

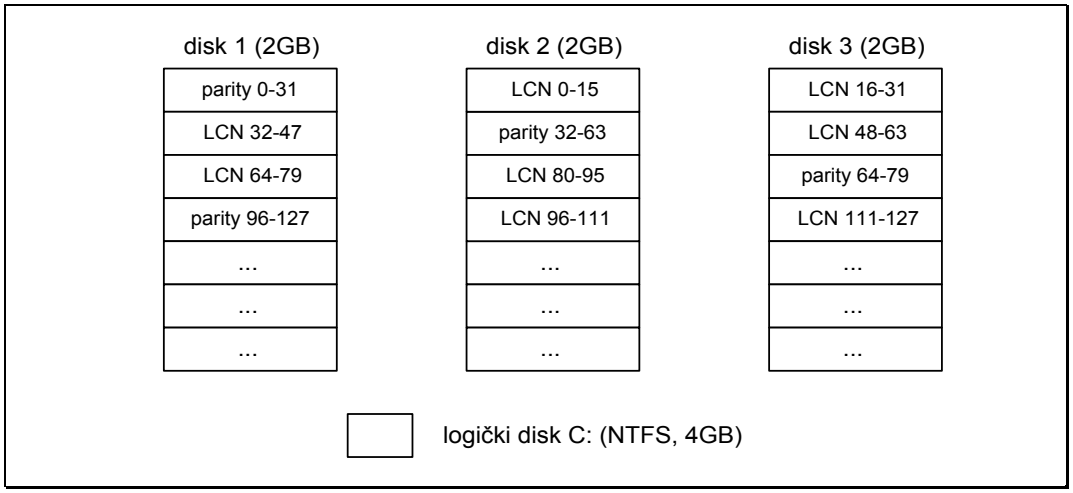


Slika 16.15. *Stripe Set*

[3] *Stripe set* sa proverom parnosti

Varijacija prethodno opisane tehnike je *stipe set* sa proverom parnosti, poznatija kao Raid nivo 5. Na primer, ukoliko set čine četiri diska, za tri trake sa podacima formira se traka parnosti na jednom od diskova, koja sadrži podatke dobijene ekskluzivnom ili operacijom podataka iz ostalih traka. Trake parnosti se po round robin algoritmu upisuju formiraju na svim diskovima koji čine set. Ukoliko jedan od diskova otkaže, podaci na njemu se mogu u potpunosti rekonstruisati izvršavanjem eksluzivno ili operacije nad podatacima sa ostalih diskova. Može se

zaključiti da ova provera parnosti značajno povećava pouzdanost seta, dok je premašenje koje unosi po pitanju kapaciteta diskova jednako kapacitetu jednog diska i kao takvo obrnuto proporcionalno broju diskova koji čine set. Par performansi koji unosi provera parnosti je skoro zanemarljiv zahvaljujući round robin alokaciji traka parnosti. Da bi se formirao ovakav set, potrebna su najmanje tri diska, odnosno particije. Slika 16.16. ilustruje formiranje stripe seta od tri diska veličine 2GB. Efektivni kapacitet dobijenog volumena je 4GB.



Slika 16.16. *Stripe Set sa proverom parnosti*

[4] *Mirror Set*

Za razliku od *stripe set*ova čija je osnovna namena značajno povećanje performansi i kapaciteta, *mirror set* se odlikuje značajnim povećanjem pouzdanosti volumena. Kako se identični podaci upisuju na oba diska, odnosno particije koje čine set, efektivni kapacitet volumena jednak je kapacitetu jednog diska. Performanse se mogu uvećati prilikom čitanja sa diska, tako što diskovi međusobno mogu podeliti radno opterećenje. Pouzdanost se dodatno može povećati vezivanjem diskova za različite kontrolere (u tom slučaju, set je imun na otkaz jednog kontrolera).

Umrežavanje - mrežni interfejsi, protokoli i razrešavanje imena

Windows 2000 podržava obe vrste umrežavanja, tzv. *peer-to-peer* i klijent-server i poseduje odgovarajući skup alata za upravljanje mrežom.

Da bi opisali umrežavanje na Windowsu 2000, pomenućemo dva mrežna interfejsa:

- NDIS (*Network Device Interface Specification*) - odvaja mrežni adapter od transportnog protokola tako da bilo koji od njih može biti izmenjen bez uticaja na onaj drugi i
- TDI (*Transport Driver Interface*) - omogućava bilo kojoj komponenti nivoa sesije da koristi bilo koji transportni mehanizam.

Windows 2000 implementira transportne protokole kao drajvere koji mogu biti učitani tj. napunjeni (*loaded*) i ispražnjeni (*unloaded*) iz sistema dinamički.

Pomenućemo neke od značajnijih protokola vezanih za umrežavanje pod Windows 2000 operativnim sistemom. *Server message block (SMB*) protokol se koristi za slanje UI (ulazno/izlaznih) zahteva preko mreže. *Network basic Input/Output system (NetBIOS)* je hardverska apstrkacija mrežnog interfejsa. Koristi se da uspostavi logička imena na mreži, uspostavi logičke veze sesija između logičkih imena na mreži i podrži pouzdani transfer podataka za sesije preko NetBIOS zahteva ili SMB-ova. *NetBEUI (NetBIOS Extended User Interface)* je bio osnovni protokol za Windows 95 *peer-to-peer* umrežavanje i za Windows for Workgroups. Koristi se i pod Windows 2000 sistemom ukoliko postoji potreba za deljenjem resursa sa prethodno pomenutim mrežama. Windows 2000 koristi TCP/IP skup protokola za povezivanje sa širokim spektrom operativnih sistema i harverskih platformi. *PPP (Point-to-Point Tunneling Protocol)* se koristi za komunikaciju *Remote Access Server* modula koji se izvršavaju na Windows 2000 mašinama koje su povezane preko Interneta. *NWLink* protokol se koristi za povezivanje NetBIOSa sa *Novell NetWare* mrežama. *Data Link Control protokol (DLC)* se koristi za pristup IBM *mainframe* računarima i HP štampačima koji su direktno povezani na mrežu. Windows 2000 sistemi mogu komunicirati sa Macintosh računarima preko *Apple Talk* protokola ako neki od Windows 2000 Servera na mreži ima pokrenut *Windows 2000 Services for Macintosh* paket.

U Windows 2000 ugrađena je podrška za aplikacije koje rade kroz:

- imenovani NetBIOS. NetBIOS aplikacije mogu komunicirati preko mreže korišćenjem protokola NetBEUI, NWLink ili TCP/IP,
- „imenovane cevi“ (*named pipes*). Imenovane cevi su mehanizam sa uspostavom veze za razmenu poruka koje su imenovane preko univerzalne konvencije imenovanja (*uniform naming convention* - *UNC*),
- *mailslot*-ove. *Mailslot*ovi su mehanizmi bez uspostave veze za razmenu poruka koji se koriste za *broadcast* aplikacije, kao što su nalaženje komponente na mreži,
- Windows priključke (*sockets*). *Winsock* ili *windows socket API* je interfejs na nivou sesije koji obezbeđuje standrdan interfejs za više transportnih protokola koji mogu imati različite adresne šeme i
- pozive udaljenih procedura. Windows 2000 RPC mehanizam sledi široko korišćeni *Distributed Computing Environment* standard za RPC poruke, tako da su programi pisani da koriste Windows 2000 RPC-ove vrlo portabilni. RPC

poruke se šalju korišćenjem NetBIOS-a ili Winscok-a na TCP/IP mrežama ili mehanizma imenovanih cevi na LAN Manager mrežama. Windows 2000 obezbeđuje Microsoft *Interface Definition Language* za opis imena procedura, argumenata i rezultata.

Umrežavanje - pristup udaljenim datotekama

U Windowsu 2000 aplikacija može koristiti njegov U/I API da pristupi datotekama na udaljenom računaru kao da su oni lokalni što je obezbeđeno time da udaljeni računar izvršava MS-NET server. *Redirector* je objekat na klijent strani koji prosleđuje U/I zahteve udaljenim datotekama gde se oni ispunjavaju od strane servera. Da bi se postigle performanse i sigurnost, redirektori i serveri se izvršavaju u kernel režimu.

Pristup udaljenoj datoteci se obavlja u sledećim koracima:

[1] Aplikacija poziva U/I manager da bi zahtevala da datoteka bude otvoren (podrazumevamo da je ime fajla u standardnom UNC formatu)

[2] U/I manager izgrađuje paket U/I zahteva

[3] U/I manager prepoznaje da je to pristup udaljenoj datoteci i pozova drajver zvani *Multiple Universal Naming Convention Provider (MUP)*

[4] MUP šalje paket U/I zahteva asinhrono svim registrovanim redirektorima

[5] Redirektor koji može da zadovolji zahtev, odgovara MUP-u. Da bi se izbegla potreba da se svi redirektori pitaju isto pitanje i u budućnosti, MUP koristi keš da se podseti koji redirektor rukuje kojom datotekom.

[6] Redirektor šalje mrežom zahtev udaljenom sistemu

[7] Mrežni drajvovi udaljenog sistema primaju zahtev i prosleđuju ga server drajveru

[8] Server drajver prenosi zahtev odgovarajućem sistemskom drajveru odgovornom za pristup lokalnoj datotekci

[9] Odgovarajući drajver uređaja (*device driver*) biva pozvan da bi se pristupilo podacima

[10] Rezultati bivaju vraćeni serverskom drajveru, koji šalje podatke nazad redirektoru koji ih je zahtevao.

16.4. Windows XP

U oktobru 2001. godine objavljen je Windows XP, kao nadogradnja Windows 2000 Proffesional operativnog sistema za radne stanice, odnosno zamena za Windows 98/Millenium Edition. Pre svega, treba reći da je Windows XP napisan na stabilnoj osnovi izvornog koda Windows 2000 operativnog sistema. Kao takav, nasleđuje opšte principe dizajna: prenosivost, proširivost, sigurnost, uskladjenost sa POSIX-om, kompatibilnost sa ranijim MS-DOS i MS Windows aplikacijama i internacionalnu podrška. Kao i njegov prethodnik, i Windows XP je 32-bitni operativni sistem sa pretpražnjenjem koji koristi mikrokernel arhitekturu i potpuno zaštićen memorijski model, što ga čini najpouzdanijim NT sistemom. Iako može solidno da radi na samostalnom računaru, odnosno na računaru koji nije umrežen, Windows XP je, takođe, i prirodni klijent za Windows 2000 mrežu. Najznačajnija unapređenja Windows-a XP primetna su u grafičkom interfejsu, koji se odlikuje krajnje intuitivnim, ergonomičnim dizajnom, mrežnom podsistemu i podršci za uređaje. Takođe, u odnosu na Windows 2000 odlikuje se i većim performansama, pouzdanošću i boljom zaštitom. Raspoloživ je u dve verzije: Windows XP Home Edition i Windows XP Professional Edition.

Arhitektura Windows XP operativnog sistema

Windows XP je, kao i prethodne verzije, slojeviti operativni sistem sastavljen od modula. U zaštićenom režimu rada izvršavaju se HAL, kernel i egzekutiva. U korisničkom režimu izvršava se kolekcija podsistema: podsistemi okruženja (MS-DOS podsistem, 16-bit Windows podsistem, 32-bit Windows podsistem, POSIX podsistem) koji emuliraju različite operativne sisteme, i zaštitni podsistemi koji obezbeđuju sigurnosne funkcije.

Kernel se nikad ne smešta van operativne memorije i njegovo izvršenje nikada ne biva pretpražnjeno. Kernel dodeljuje procesor nitima, rukuje prekidima i izuzecima, obavlja sinhronizaciju procesora niskog nivoa i zadužen je za oporavak posle greške ili kvara u napajanju.

Egzekutiva Windows XP operativnog sistema upravlja objektima, procesima, virtuelnom memorijom, ulazno-izlaznim podsistemom i Plug-and-Play uređajima, obavlja kontrolu pristupa (Security Reference Monitor) i prosleđuje zahteve i rezultate između klijent i server procesa unutar jedne mašine (LPC).

Podsistem okruženja čine procesi koji rade u korisničkom režimu rada sa ciljem da omoguće izvršavanje programa. Kao glavno operativno okruženje Windows XP koristi Win32 podsistem; Win32 se koristi za pokretanje svih procesa. On takodje obezbeđuje sve mogućnosti i funkcije tastature, miša i grafičkog displeja. MS-DOS okruženje je ostvareno pomoću VDM (*Virtual DOS Machine*), a 16-bitno Windows okruženje pomoću WOW VDM (*Windows on Windows*), koja se može shvatiti kao

virtuelna Windows mašina. Svaki DOS program izvršava se na zasebnoj VDM, a više 16-bitnih Windows programa se izvršava u okviru jedne WOW VDM. WOW obavlja dvosmernu konverziju između Win16 i Win32 API-ja i poruka. Ono što se prevede izvršava se na Win32 podsistemu.Podsistem za prijavljivanje (*logon*) i sigurnosni podsistem (*Security Subsystem*) autentifikuju korisnike koji se prijavljuju na Windows XP sisteme. Paket za autentifikaciju, autentifikuje korisnike uvek kada pokušaju da pristupe nekom objektu u sistemu.

Windows XP koristi NTFS kao domaći sistem datoteka, a dodatno uključuje i podršku za FAT, FAT32, ISO9660 i UDF.

Unapređenja u Windows XP operativnom sistemu

U nastavku teksta ukratko su opisana neka unapređenja Windows XP operativnog sistema u odnosu na svoje prethodnike (informacije su preuzete sa *web*-sajta kompanije Microsoft). Unapređenja su klasifikovana po kategorijama.

[1] Pouzdanost i sigurnost

- *System Restore*, pomoću kog se sistem jednostavno može povratiti u prethodno stanje bez gubitka podataka. *System Restore* automatski kreira presečne tačke na koje se sistem može vratiti.
- Povratak na pouzdane drajvere (*Driver Rollback*) - prilikom nadogradnje ili instalacije novih drajvera za hardver, Windows XP kreira kopiju postojećih stabilnih drajvera. Ukoliko se novi ili nadograđeni drajveri ne pokažu kao pouzdani, administrator može vratiti stare.
- Mrežna barijera ka Internetu (*Internet Connection Firewall*) štiti korisnika od najčešćih vrsta napada.
- Višekorisnički pristup datotekama na kripto fajl sistemu (EFS - *Encryption File System*).
- Podrška za SmartCard autentifikacioni mehanizam.

[2] Održavanje i administracija

- Povećana kompatibilnost sa korisničkim aplikacijama - veliki broj aplikacija koje se nisu mogle pokrenuti pod Windows 2000, mogu se izvršavati pod Windows XP operativnim sistemom. Ukoliko aplikacija nije namenjena za Windows XP sistem, korisnik može inicirati njeno izvršenje u Windows NT 4.0 ili Windows 95/98/ME kompatibilnom režimu.
- Ugrađena podrška za najnoviji hardver - Windows XP Professional podržava poslednje standarde i trendove razvoja hardvera, uključujući UDF 2.01 standard za čitanje DVD diskova, kreiranje FAT32 fajl sistema na DVD-RAM

medijumima, IrDA (*Infrared Data Association*), Universal Serial Bus (USB) i IEEE 1394. Takođe, integrisana je i podrška za snimanje CD medijuma.

- Automatska nadogradnja (*Automatic Update*) instaliranog operativnog sistema, kao i dinamička nadogradnja komponenti operativnog sistema u toku instalacionog procesa. U oba slučaja, operativni sistem ili sam instalacioni proces odgovarajuće zakrpe preuzimaju sa Interneta.
- Jednostavno objavljivanje informacija (datoteka i direktorijuma) na Internetu preko WebDAV protokola

[3] Umrežavanje

- Udaljeni desktop (*Remote Desktop*), pomoću kog korisnici mogu kreirati virtuelnu desktop sesiju sa udaljenog računara, ukoliko je na njemu instaliran Windows 95 ili noviji operativni sistem. Udaljeni desktop se može posmatrati kao grafički telnet servis.
- Podrška za IEEE 802.1x standard, odnosno za bežične mreže.

16.5. Windows 2003 Server

Windows 2003 Server je serverska verzija Microsoft operativnog sistema koja je naslednik Windows 2000 Servera i predstavlja korak u evoluciji Microsoft serverskih operativnih sistema. U suštini, može se posmatrati kao Windows XP nadograđen serverskim funkcijama, iz koga su isključene neke od multimedijalnih funkcija XP operativnog sistema (zvuk, ubrzanja 3D grafike, teme) u cilju povećanja stabilnosti.

Proizvod je inicijalno bio nazvan Windows .NET Server 2003 u cilju promovisanja *enterprise framework .NET* tehnologije. U ovom unapređenom Microsoft serveru, performanse ASP.NET-a (naslednika aktivnih serverskih strana, tj. *Active Server Pages*) su unapređene i tešnje integrisane u sistem.

Međutim, kao posledica straha od zbunjivanja tržišta u pogledu toga šta .NET znači i odgovarajući na neke kritike, Microsoft je uklonio .NET iz imena ovog servera. Nakon ovog poteza .NET pripada isključivo .*NET framework*u.

Windows Server 2003 je lansiran na tržište u Aprilu 2003. godine. Od brojnih unapređenja, navedimo samo neka:

- unapređenja aktivnog direktorijuma (kao što je mogućnost da se obriše klasa iz *scheme*),
- unapređenja vezana za administraciju grupnih polisa,
- unapređenja vezana za administraciju diskova (kao što je mogućnost da se izvrši arhiviranje otvorenih datoteka, korišćenjem starih verzija otvorenih datoteka - *Shadow copy*).

Windows 2003 Server se u mreži može ponašati kao radna stanica, domen kontroler i *member* server koji obavlja jednu ili više funkcija:

- server za datoteke i DFS server (distribuirani sistema datoteka),
- server za štampu (sa podrškom za MS-DOS, Windows, UNIX, Linux, Mac OS i Novell Netware klijente),
- terminal server,
- server za udaljeni pristup i VPN,
- server koji pruža razne TCP/IP servise (kao što su DHCP i DNS),
- web server i server za elektronsku poštu.

Windows 2003 Server dolazi je raspoloživ u nekoliko verzija koje su prilagođene različitim veličinama i zahtevima određenih poslovnih okruženja.

- Standard Edition je namenjen serverima u LAN mrežama srednje veličine. Omogućava standardne servise deljenja datoteka po mreži, deljenje štampača i centralizovanu administraciju mrežnih resursa. Podrška za simetrično multiprogramiranje ograničena je na sisteme sa četiri procesora.
- Enterprise Edition podržava osmoprocesorske sisteme sa najviše 32GB operativne memorije i osmostruke klastere, kao i 64-bitne platforme sa 64GB memorije.
- Datacenter Edition se isporučuje isključivo uz specijalizovane računare. Podržava računare sa 32 procesora i 64GB operativne memorije. Verzija za 64-bitne Itanium procesore podržava kreiranje 128 particija unutar jednog računara sa po 64 procesora i 512GB memorije po particiji. Podržani su osmostruki klasteri.
- Small Business Server je zamišljen kao kompletno serversko rešenje za mreže manjih preduzeća. Uz sam operativni sistem isporučuje se i Exchange server za elektronsku poštu, SQL server i još neki proizvodi.
- Web Edition je varijanta Windows 2003 Server operativnog sistema za namenske Web servere.

16.6. Longhorn

Napominjemo da je, u vreme dok pišemo ovu knjigu (kraj 2004. godine), Longhorn kodno ime za novu verziju Windowsa i da se informacije koje ovde navodimo mogu pokazati netačnim u vreme planiranog izlaska ovog operativnog sistema (2006. godine). Zato molimo čitaoce da informacije o Longhornu uzmu sa rezervom.

Longhorn bi trebao da bude sledeći operativni sistem iz porodice Windowsa koji će naslediti Windows XP i Windows Server 2003. Prema onome što je trenutno poznato ovaj operativni sistem bi trebao da donese brojna unapređenja i neke „revolucionarne“ koncepte.

Nove tehnologije

Ključne nove tehnologije Longhorn operativnog sistema, koje Microsoft označava sintagmom "The Pillars of Longhorn" su:

- Temelji - razvoj osnove strukture operativnog sistema u cilju integracije novih koncepata, kao što su *.NET framework* i *trusted computing*;
- Avalon - novi podsistem korisničkog interfejsa i API baziran na XML-u, .NET-u i vektorskoj grafici koji će omogućiti bolje korišćenje 3D kompjuterskog grafičkog hardvera i Direct3D tehnologija;
- Indigo - servisno orijentisan sistem poruka koji treba da omogući da programi interoperiraju kao deo .NET okruženja;
- WinFS (Windows File System) - kombinovana relaciona baza podataka i sistem datoteka, koji će omogućiti prezentovanje objekata i njihovih međusobnih relacija, za razliku od trenutnog prikazivanja hijerarhije u vidu datoteka i direktorijuma. Zasnovan na sledećoj verziji SQL Servera (kodno ime Yukon) i NTFS sistema datoteka.
- WinFX - novi API koji treba da omogući pristup novim funkcijama, zamenjujući Win32 API.

Novi korisnički interfejs

Potpuno redizajniran korisnički interfejs (kodno ime Aero) estetski je superioran u odnosu na interfejse korišćene u prethodnim verzijama Widnowsa. Najvidljiviji dodatak je *sidebar*, tj. područje sa strane ekrana koje prikazuje dinamičke informacije o prozorima koji se trenutno izvršavaju u prvom planu. Može se posmatrati kao proširenje linije zadataka (*taskbar*) koja je prisutna u dosadašnjim verzijama Windows sistema.

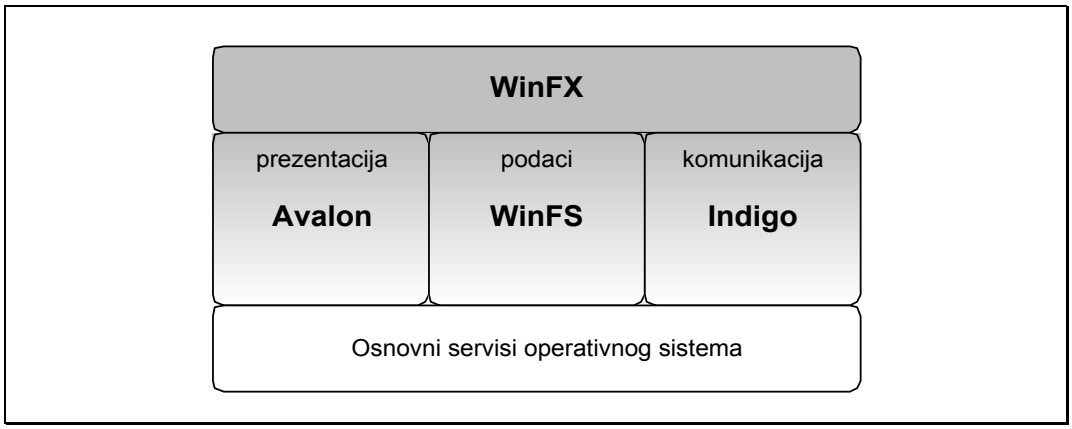
Aero je zahtevan po pitanju grafičkog podsistema, a konkretni zahtevi definišu se posebno za tri različita grafička režima. Aero Glass grafički režim uključuje potpun skup Longhorn vizuelnih efekata i punu podršku za 3D grafiku i animaciju, i kao takav zahteva AGP 8X ili PCI Express 16-lane bus grafički adapter sa 64 MB memorije (poželjno 128 MB). Aero koristi znatno manju količinu hardverske akceleracije i kao takav zahteva AGP 4X ili PCI Express 8-lane bus grafički adapter sa 32 MB memorije. Classic je ekvivalent korisničkog interfejsa Windowsa 2000. U ovom režimu Longhorn koristi softverske tehnike za prikaz grafičkih elemenata. Ukoliko grafički adapter

računara ne ispunjava zahteve za pokretanje Aero Glass ili Aero režima, koristi se Classic.

Novi interfejs komandne linije nosi kodno ime Monad, koje potiče iz Monadologije (Leibniz). Filozofija monadologije je zasnovana na pretpostavci da se sve sastoji od manjih elemenata, pri čemu je Monad najmanji.

Arhitektura Longhorna

Na slici 16.17. prikazana je arhitetktura Longhorna:



Slika 16.17. *Arhitektura Longhorn operativnog sistema*

Longhorn čini nekoliko slojeva: sloj osnovnih servisa operativnog sistema, sloj prezentacije (Avalon), sloj upravljanja podacima (WinFS) i komunikacioni sloj (Indigo). Pristup funkcijama operativnog sistema obavlja se pomoću WinFX API-ja.

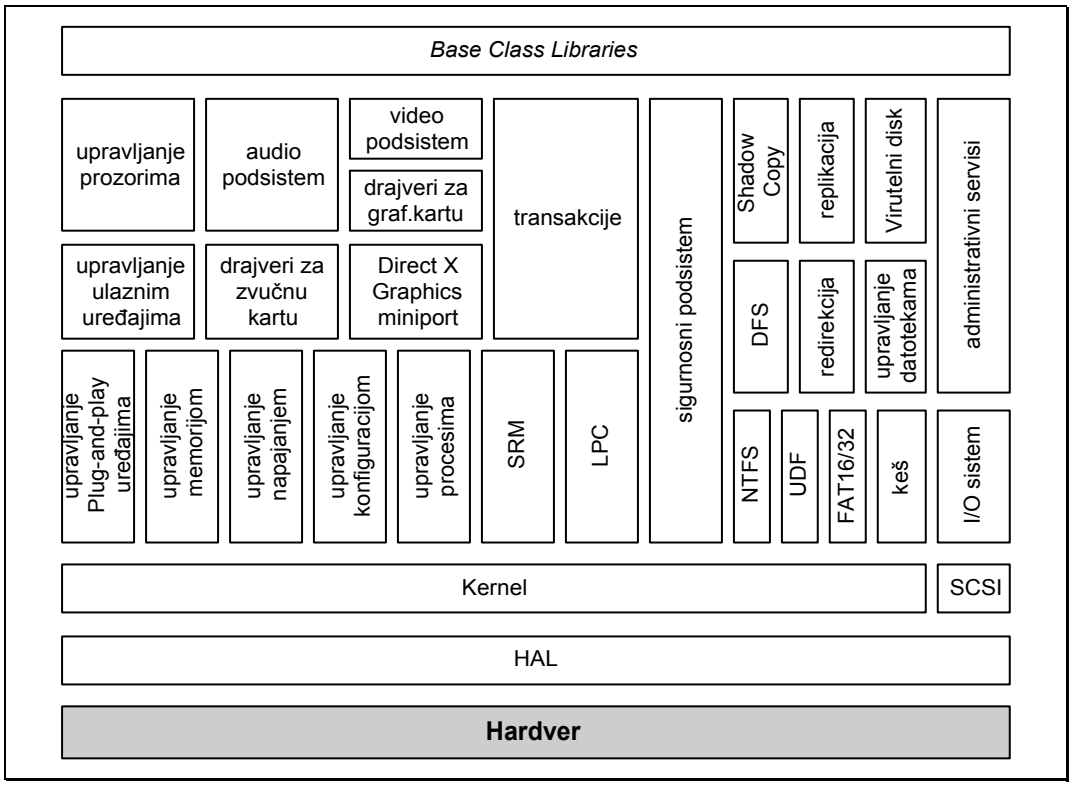
Iako na prvi pogled koncept izgleda jednostavan, realna situacija je daleko složenija. Analizirajmo redom strukuru svih slojeva.

Sloj osnovnih servisa operativnog sistema

Kao i u prethodnim verzijama na najnižem nivou nalazi se sloj apstrakcije hardvera (HAL). Na funkcije HAL-a oslanjaju se svi ostali slojevi. Ovo je jedini sloj operativnog sistema koji sadrži komponente koje se izvršavaju u zaštićenom režimu rada.

Izdvojićemo u dve grube celine komponente koje nisu vezane za mrežu i mrežni podsistem.

[1] Podsistem nevezan za mrežu



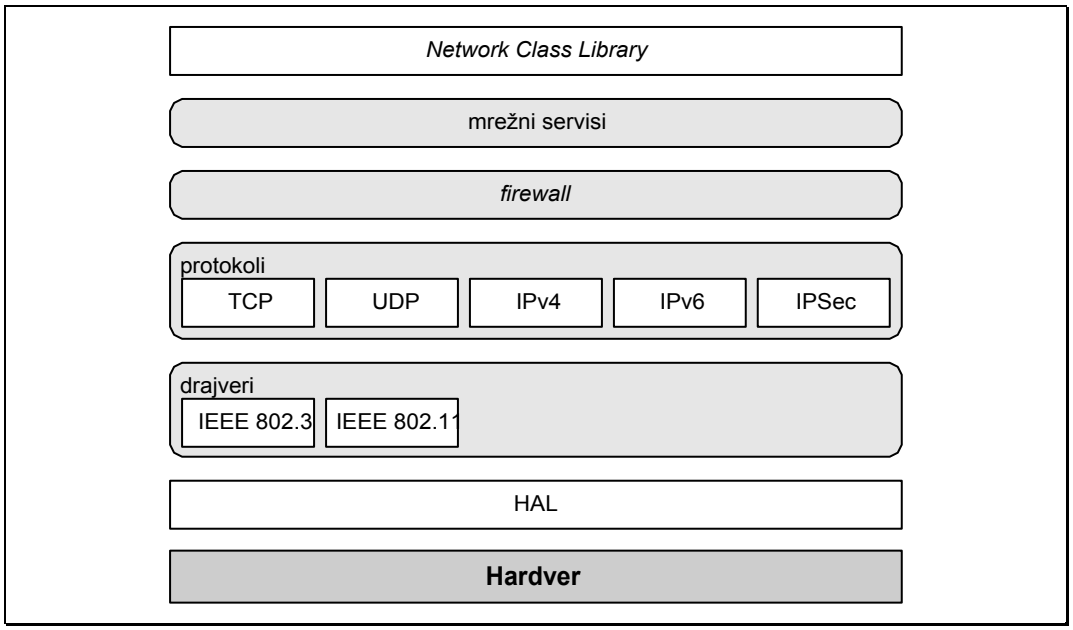
Slika 16.18. *Osnovni servisi nevezani za mrežu*

Kao što je prikazano na slici 16.18, kernel i SCSI drajver su najbliži HAL-u, a iznad njih se nalaze:

- komponente koje ulaze u sastav egzekutive Windowsa NT/2000 predstavljaju egzekutivu (upravljanje memorijom, procesima, napajanjem, konfiguracijom, Plug-and-play uređajima). Iznad "egzekutive" nalaze se transakcioni podsistem, osnovni grafički podsistem (koji je u odnosnu na Windows NT pomeren iznad kernela), audio drajveri i komponente za upravljanje ulaznim uređajima,
- sigurnosni podsistem,
- komponente kojima se upravlja sistemima datoteka i sekundarnim memorijama (podrška za FAT, FAT 32 i NTFS sisteme datoteka, DFS, upravljanje kešom i razni servisi - *Virtual Shadow Copy* servis, servis replikacije datoteka i servis virtuelnih diskova) i
- ulazno-izlazni podsistem sa pratećim administrativnim servisima.

Na najvišem nivou nalaze se biblioteke osnovnih usluga operativnog sistema (*Base Class Libraries*) koje obezbeđuju funkcionalnost upravljanja memorijom, zaštite, izvršavanja koda itd.

[2] Mrežni podsistem



Slika 16.19. *Mrežni servisi*

Na najnižem nivou nalazi se HAL čije usluge koriste generalizovani drajveri za mrežne adaptere (802.3, 802.11). Na višem nivou nalaze se protokoli (IPv4, IPv6, TCP, UDP, IPSEC). Sledeći nivo čine mrežni servisi koji se oslanjaju na mrežnu barijeru (*firewall*). Na vrhu podsistema nalaze se biblioteke mrežno-komunikacionih usluga (*Network Class Libraries*) koje obezbeđuje operativni sistem.

Sloj prezentacije - Avalon

Kao što je već rečeno, Avalon je novi podsistem korisničkog interfejsa i API baziran na XML-u, .NET-u i vektorskoj grafici koji će omogućiti bolje korišćenje 3D kompjuterskog grafičkog hardvera i Direct3D tehnologija. Jedan od ciljeva pri projektovanju Avalona je zbližavanje korisničkog interfejsa, dokumenata i multimedijalnih sadržaja. Ovakav podsistem zahteva snažniji grafički podsistem računara.

Korisnički interfejs se sastoji od elemetanta kao što su: paneli (određuju izgled stranice i mogu se koristiti kao kontejneri za druge elemente), kontrole (omogućavaju

interakciju sa korisnikom), konture i oblici (elementi vektorske grafike) i dokumenti. Elementi imaju svojstva, tako da se njihov izgled i ponašanje mogu prilagoditi. Elementi su sposobni da izazovu događaje (*events*). Ilustrovaćemo XAML kod korisničkog interfejsa koji u interakciji sa korisnikom izaziva događaj:

```
<FlowPanel xmlns=
 "http://stoner.org/download/public" >
 <Button Background="Green" Click="EventYes">
  Yes
 </Button>
 <Button Background="Red" Click="EventNo">
  No
 </Button>
</FlowPanel>
```

Pritiskom na odgovarajuće dugme na ekranu korisnik izaziva izvršenje funkcija EventYes() i EventNo().

Sloj upravljanja podacima - WinFS

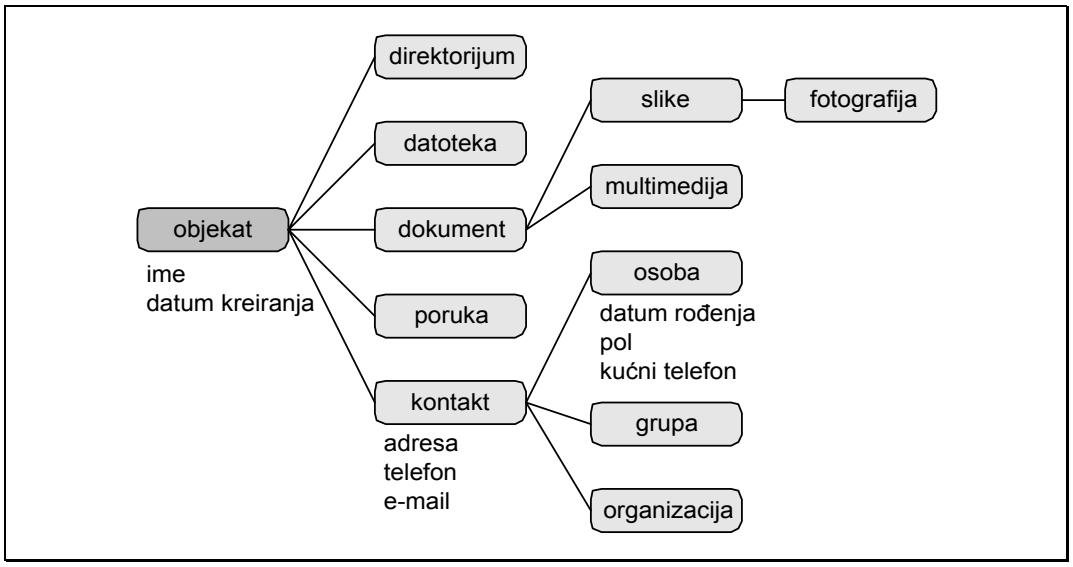
WinFS upravlja podacima koristeći relacionu bazu podataka (SQL Server) i sistem datoteka (NTFS), čime je omogućeno prezentovanje objekata i njihovih međusobnih relacija.

Razlozi za uvođenje koncepta upravljanja podacima pomoću relacione baze podataka su: pad cena diskova, povećanje količine podataka koje prosečni korisnici poseduju i čest gubitak podataka (prouzrokovan nemogućnošću korisnika da pronađe željene podatke). Dodatno, ovim konceptom se omogućava kvalitetni mehanizam deljenja podataka između aplikacija.

Razjasnićemo složenost koja je ugrađena u WinFS, poređenjem sa NTFS sistemom datoteka.

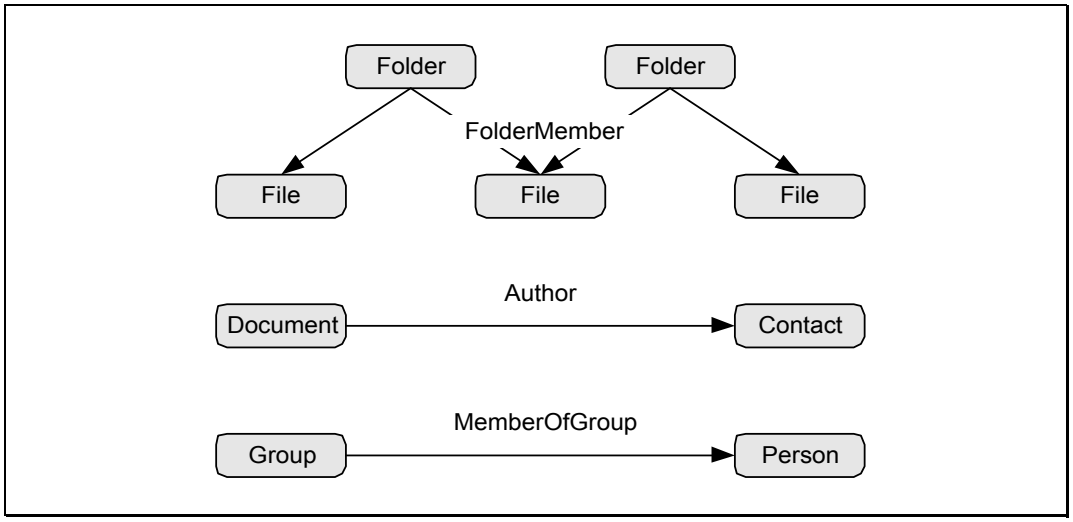
U objekte NTFS sistema datoteka spadaju datoteke i direktorijumi organizovani u hijerarhijsku strukturu stabla. Objekti su opisani standardnim skupom atributa (kao što su ime i datum), a datoteke dodatno mogu biti opisane specijalnim skupom atributa koji određuje aplikacija kojom je datoteka kreirana.

Za razliku od NTFS sistema datoteka, WinFS čuva instance objekata. Svaki objekat odgovarajućeg tipa opisan je svojstvima karakterističnim za taj tip. Korisnik može kreirati nove objekte ili promovisati datoteke na NTFS sistemu datoteka u objekte. Skup svojstava kojim se opisuje objekat nekog tipa određen je šemom tipa (*schema*), a po potrebi može uključivati i podatke (*data stream*).



Slika 16.20. *WinFS objekti i njihova svojstva*

Instance objekata su međusobno povezane relacijama, kao što je prikazano na slici 16.21. Na ovaj način je, na primer, omogućeno da jedna datoteka pripada većem broju direktorijuma (relacija FolderMember između objekata File i Folder), da se iskaže autorstvo (relacija Author između objekata Document i Contact) i članstvo u grupama (relacija MemberOfGroup između objekata Contact i Group).



Slika 16.21. *Relacije između objekata*

Aplikacija pristupa WinFS objektima korišćenjem WinFS API-ja. Pristup se može ostvariti i korišćenjem SQL upita (ADO.NET). Datotekama na NTFS sistemu datoteka pristupa se pomoću poziva funkcija Win32 API-ja za rad sa datotekama.

Sloj komunikacija - Indigo

Indigo je servisno orijentisan sistem poruka koji treba da omogući da programi interoperiraju kao deo .NET okruženja. Indigo je skalabilan i obezbeđuje mehanizam za kreiranje i upravljanje servisno-orijentisanih sistema, odnosno unificiran interfejs za kreiranje i izvršavanje aplikacija. Ideja na kojoj je zasnovan Indigo je sledeća:

Port je lokacija u mrežnom prostoru. Port ima jedan ili više transportnih kanala (kao što su HTTP, TCP, SMTP) i obezbeđuje URI za sve adrese. Poruke se prenose kroz port putem kanala (*Channel*) koji obezbeđuju mehanizam razmene poruka (kao što su Request/Reply i datagram) i opciono obezbeđuju sigurnost i transakcione mehanizme.

17

Linux

Sadržaj poglavlja:

17.1. Šta je Linux ?
17.2. Opšti pregled Linux sistema
17.3. Jezgro Linux sistema
17.4. Značajni delovi jezgra
17.5. Osnovni servisi Linux sistema

17.1. Šta je Linux ?

Jedna od poslednjih varijanti UNIX operativnih sistema, čiji je razvoj započeo Linus Torvalds 1991. godine na Unverzitetu u Helsinkiju, je Linux. Torvalds je svoj operativni sistem koji objedinjuje oba standarda, SRV4 i BSD, objavio na Internetu i podsticao druge programere širom sveta da se priključe njegovom daljem razvoju. Ubrzo, Linux je postao veoma popularan među računarskim entuzijastima, koji su tražili alternastivno rešenje za postojeće operativne sisteme za PC računare (DOS, Windows). Linux je svojom koncepcijom, stabilnog a jeftinog operativnog sistema doživeo veliku ekspanziju i popularnost. Simbol Linux sistema je mali pingvin (Tux).

Linux je prvobitno namenjen 32-bitnim Intel x86 mikroprocesorima (počevši od 80386), na kojima može funkcionisati kao radna stanica (*workstation*) ili kao server. Jezgro Linux sistema je kasnije modifikovano i prilagođeno procesorima koji ne

pripadaju Intel x86 klasi, među kojima treba istaći Intel IA-64, DEC Alpha, SUN SPARC/UltraSPARC, Motorola 68000, MIPS, PowerPC i IBM mainframe S/390. Može se konstatovati da današnji Linux u odnosu na bilo koji operativni sistem podržava najširi spektar procesora i računarskih arhitektura.

GNU/Linux i Open Source Software

Veliki deo komponenti Linux operativnom sistemu dodali su nezavisni programeri i programeri GNU projekta (www.gnu.org), koji pripada slobodnoj softverskoj fondaciji (FSF - *Free Software Foundation*). Svi GNU/Linux operativni sistemi koriste Linux kernel kao fundamentalni deo koji kontroliše interakciju između hardvera i aplikacija, i GNU aplikacije kao dodatne komponente operativnog sistema.

Linux je raspoloživ kao besplatan operativni sistem pod GNU GPL licencom (*GNU General Public License*), što važi i za neke druge vrste UNIX sistema, kao što su FreeBSD i NetBSD. Linux je softver sa otvorenim izvornim kodom (*Open Source*), što znači da mu je izvorni kod javno raspoloživ i može biti modifikovan tako da odgovara specifičnim potrebama. Linux se može slobodno distribuirati među korisnicima. Ovakav koncept je potpuno suprotan konceptu komercijalnog softvera, gde izvorni kod nije dostupan i svaki korisnik mora da plati licencu za korišćenje. Komercijalni softver je baziran na autorskim pravima (*copyright laws*), koja preciziraju limite koje korisnici softvera imaju u odnosu na izvorni kod, korišćenje i dalje distribuiranje softvera. Linux se besplatno može preuzeti sa različitih web-sajtova.

Linux distribucije

Brojne profitne i neprofitne organizacije čine Linux raspoloživim u formi distribucija, odnosno različitih kombinacija kernela, sistemskog softvera i korisničkih aplikacija. Većina distribicuja sadrži kolekciju CD/DVD medijuma na kojima se nalaze operativni sistem, izvorni kod, detaljna dokumentacija, kao i štampana uputstva za instalaciju i upotrebu sistema. Cene ovakvih distribucija su u većini slučajeva simbolične, osim ako se u distribuciji nalazi komercijalan softver ili je distribucija specifične namene.

Osnovna komponenta svake Linux distribucije je kernel operativnog sistema. Osim kernela i sistemskog softvera, u distribuciji se nalaze i instalacioni alati, softver za podizanje operatvnog sistema (*boot loader*), razne korisničke aplikacije (kancelarijski paketi - *office suite*, softver za obradu bit-mapiranih slika) i serverski paketi. Većina distribucija je, poput Windows sistema, grafički orijentisana prema korisniku, dok su neke distribucije namenjene za sistemske administratore i programere familijarne sa tradicionalnim UNIX okruženjem.

U poznatije Linux distribucije spadaju: Debian GNU/Linux, Linux Mandrake, Red Hat Linux, Slackware Linux i SuSE Linux.

17.2. Opšti pregled Linux sistema

Linux je višekorisnički, višeprocesni operativni sistem sa potpunim skupom UNIX kompatibilnih alata, projektovan tako da poštuje relevantne POSIX standarde. Linux sistemi podržavaju tradicionalnu UNIX semantiku i potpuno implementiraju standardni UNIX mrežni model.

Linux operativni sistem sastoji se od kernela, sistemskog softvera, korisničkih aplikacija, programskih prevodioca i njihovih odgovarajućih biblioteka (GCC - GNU C *Compiler* i C biblioteka za Linux) i dokumentacije. Sadržaj konkretne Linux distribucije definisan je sadržajem instalacionih medijuma, koji u slučaju nekih Linux sistema uključuju razne FTP sajtove širom sveta.

Kernel je jezgro operativnog sistema - on omogućava konkurentno izvršavanje procesa, dodeljuje im memoriju i druge resurse i obezbeđuje mehanizam za ostvarivanje usluga operativnog sistema. Kernel štiti korisničke procese od direktnog pristupa hardveru - procesi pristupaju hardveru korišćenjem sistemskih poziva kernela, čime se obezbeđuje jedna vrsta zaštite između samih korisnika. Sistemski programi koriste kernel u cilju implementacije različitih servisa operativnog sistema.

Svi programi, uključujući i sistemske, funkcionišu na nivou iznad kernela, što se naziva korisnički režim rada, dok se sistemske aktivnosti poput pristupa hardveru obavljaju na nivou kernela, odnosno u sistemskom režimu rada (*supervisory mode*). Razlika između sistemskih i aplikativnih programa je u njihovoj nameni: aplikacije su namenjene za razne korisne aktivnosti (kao što su obrada teksta i slike), dok su sistemski programi namenjeni za rad sa sistemom i administraciju.

17.3. Jezgro Linux sistema

Tri osnovne verzije Linux kernela su početna verzija, verzija 1.x i verzija 2.x. Početna verzija 0.01, koju je 1991. godine kreirao Linus Torvalds, podržavala je samo Intel 80386 kompatibilne procesore, mali broj hardverskih uređaja i Minix sistem datoteka. Mrežni servisi nisu imali kernelsku podršku. Verzija 1.0, nastala u martu 1994. godine, uključivala je podršku za standardne TCP/IP mrežne protokole, BSD-kompatibilni socket interfejs za mrežno programiranje i drajversku podršku za mrežne kartice. Ova verzija je dodatno podržavala ext i ext2 sisteme datoteka, široku klasu SCSI disk kontrolera, kao i brojne hardverske uređaje. Verzija 1.2 (mart 1995) je poslednja verzija Linux kernela namenjena isključivo PC arhitekturi. U verziji 2.0 (jun 1996) uvedena je podrška za više arhitektura (Motorola i Intel procesori, Sun Sparc i PowerMac sistemi), kao i podrška za višeprocesorsku arhitekturu (SMP). Dodatno, poboljšano je upravljanje memorijom i uvećane se performanse TCP/IP protokol steka, a ugrađena je i podrška za unutrašnje kernelske niti (internal kernel threads). Kernel je

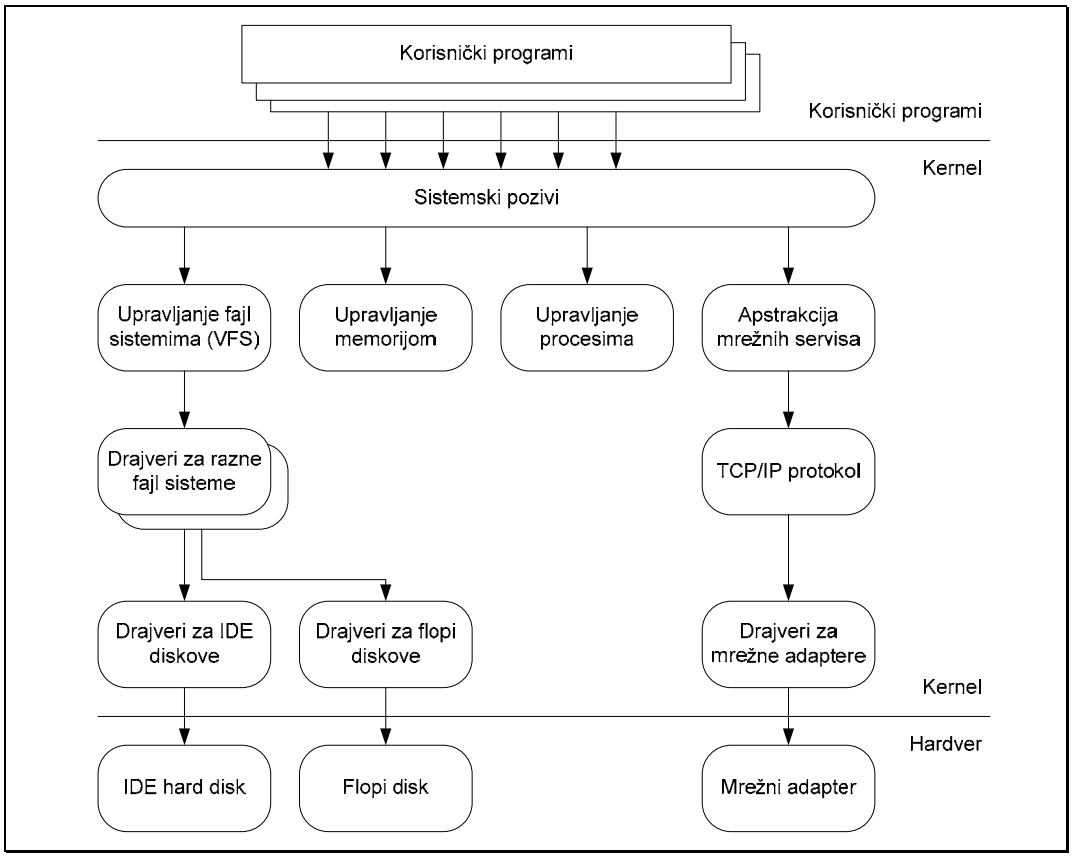
modularizovan, odnosno uvedena je mikro-kernel struktura sa izmenljivim drajverskim modulima (loadable kernel modules), a standardizovan je i konfiguracioni interfejs.

Struktura Linux sistema

Osnovu Linux sistema čine kernel, sistemske biblioteke i sistemski programi. Kernel je odgovoran za najznačajnije funkcije operativnog sistema. Dve osnovne karakteristike kernela su:

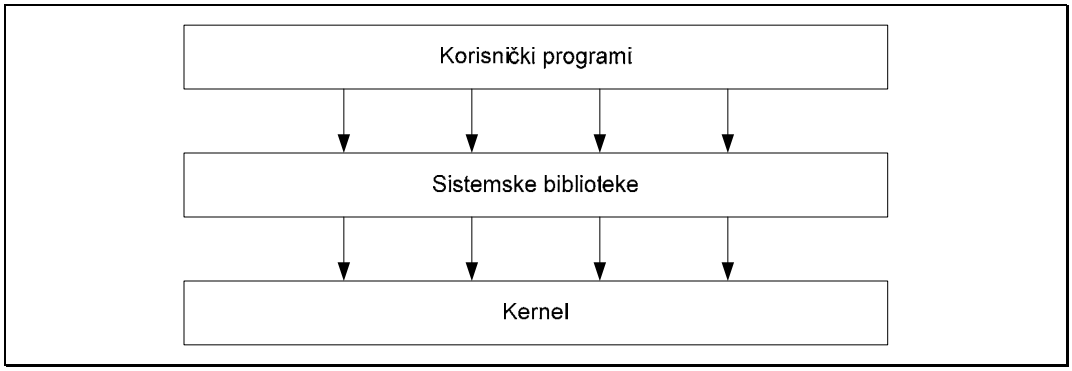
- kernel kod se izvršava u kernelskom modu u kome je jedino moguće pristupati svim komponentama hardvera,
- kompletan kernel kod i sve kelnel strukture podataka se čuvaju u istom adresnom prostoru (*monolithic*).

Kod većine UNIX sistema, aplikacije se preko sistemskog poziva direktno obraćaju kernelu, kao što je prikazano na slici 17.1.



Slika 17.1. *Jezgro UNIX sistema*

Kod Linux sistema, sistemski pozivi se upućuju kernelu preko sistemskih biblioteka, koje definišu standardni set funkcija preko kojih aplikacije komuniciraju sa kernelom. Ovaj metod komunikacije sa kernelom prikazan je na slici 17.2.



Slika 17.2 *Struktura Linux sistema*

Sistemski programi izvršavaju specifične upravljačke poslove, kao što je konfigurusanje mrežnih uređaja i protokola, punjenje kernelskih modula itd.

Modularni kernel

Moduli kernela su delovi kernelskog koda koji može da se prevede, napuni u memoriju ili izbaci iz memorije nezavisno od ostatatka kernela. Kernelski moduli implementiraju drajvere za hardverske uređaje, novi sistem datoteka, mrežne protokole, itd. Moduli omogućavaju raznim programerima da napišu i distribuiraju drajvere koji ne moraju da prođu GPL licencu.

NAPOMENA: struktura Linux kernela koji nije modularan odudara od teorijskog koncepta jezgra. Približavanje teorijskom konceptu je moguće ukoliko je jezgro modularno. Tada se upotrebom modula određene funkcije kernela (kao što su funkcije mrežne barijere) mogu izbaciti na korisnički nivo, dok u kernelu ostaju samo osnovne funkcije.

Moduli kernela omogućavaju *micro-kernel* arhitekturu, odnosno realizaciju minimalne stabilne konfiguracije kernela bez dodatnih drajvera. Potrebni drajveri pune se u memoriju kao moduli kernela. Module Linux kernela čine tri komponente:

[1] Upravljanje modulom

Komponenta omogućava punjenje modula u kernelsku memoriju i komunikaciju modula sa ostatkom kernela, proveru da li je modul u memoriji i da li se koristi i izbacivanje modula iz memorije (pod uslovom da se modul ne koristi).

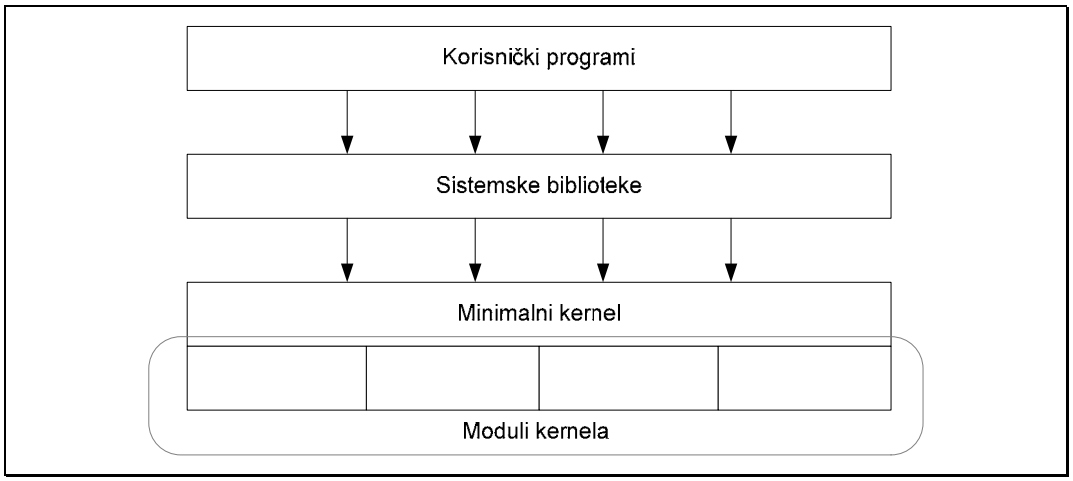
[2] Registracija drajvera

Komponenta omogućava modulu da objavi ostatku kernela da je novi drajver u memoriji i da je raspoloživ za korišćenje. Kernel održava dinamičku tabelu drajvera, koji se pomoću posebnog seta programa mogu napuniti ili izbaci iz memorije u svakom trenutku.

[3] Rezolucija konflikata

Mehanizam rezolucije konflikata služi da spreči hardverske konflikte, tako što omogućava drajveru da rezerviše hardverske resurse (IRQ, DMA, ports) i time spreči druge drajvere ili autoprobe funkciju da ih koriste.

Na slici 17.3. prikazana je struktura modularnog Linux kernela.



Slika 17.3. *Modularni Linux kernel*

17.4. Značajni delovi jezgra

Linux kernel čine sledeće značajne komponente:

- upravljanje procesima,
- upravljanje memorijom,
- upravljanje sistemima datoteka (VFS),
- apstrakcija mrežnih servisa,
- podrška za hardverske uređaje,
- podrška za različite sisteme datoteka i

- podrška za TCP/IP.

Kritične komponente Linux kernela su upravljanje procesima i upravljanje memorijom. Komponenta za upravljanje memorijom kontroliše dodeljivanje memorije i swap prostora procesima, kernelskim komponetama kao i bafersko keširanje. Komponenta za upravljanje procesima kreira procese i omogućava višeprocesni rad (*multitasking*), dodeljujući procesor procesima po odgovarajućem algoritmu.

Na najnižem nivou, kernel sadrži podršku u vidu drajvera za razne hardverske uređaje. S obrzirom da postoji veliki broj različitih vrsta hardverskih uređaja, broj drajvera je takođe veoma veliki. Za hardverske uređaje iste vrste (npr. diskove), koji vrše sličnu funkciju, ali se razlikuju u načinu softverske kontrole, formirane su opšte klase drajvera na sledeći način: svaki član klase pruža isti interfejs prema ostatku kernela, čime obezbeđuje podršku za operacije koje su karakteristične za tu vrstu uređaja, a hardver opslužuje na odgovarajući način. Na primer, svi disk drajveri pružaju isti interfejs prema ostatku kernela i svi imaju operacije tipa inicijalizacije drajva, čitanja podataka iz određenog sektora i upisa podataka u određeni sektor.

Takođe, neki softverski servisi koje kernel podržava imaju slične osobine, čime se omogućava njihova apstrakcija u klase. Na primer, različiti mrežni protokoli se apstrakuju u jedan programski interfejs koji se naziva "BSD *socket library*". Drugi primer je virtuleni sistem datoteka (VFS - *virtual filesystem*), koji apstrakuje operacije u sistemu datoteka, pri čemu svaki tip sistema datoteka obezbeđuje specifične implementacije raznih operacija. Zahtev za korišćenjem sistema datoteka koji šalje korisnik prolazi kroz VFS sloj, koji isti prosleđuje na odgovarajući drajver za konktertni sistem datoteka.

Upravljanje procesima

Kernel Linux operativnog sistema upravlja pokretanjem i uništenjem procesa i dodelom resursa sistema procesima, poput procesorskog vremena i operativne memorije.

Pod UNIX sistemom sve informacije koje operativni sistem mora čuvati da bi kontrolisao jedan proces predstavljaju kontekst tog procesa. Pod Linux operativnim sistemom, svaki proces je u potpunosti opisan identitetom, okolinom, i kontekstom.

Identitet procesa obuhvata sledeće informacije:

[1] Identifikator procesa (*Process ID* - PID)

Linix kontroliše proces pomoću jedinstvenog identifikatora.

[2] Akreditivi (*Credentials*).

Svaki proces pripada jednom korisniku koji ima svoj user ID i jedan ili više grupnih IDs, koji određuju prava pristupa procesu u radu sa datotekama.

[3] Ličnost (*Personality*).

Ova informacija se ne koristi kod drugih UNIX sistema, a Linux svakom procesu dodeljuje lični identifikator koji može imati uticaja za neke sistemske pozive.

Okolina procesa se nasleđuje od procesa roditelja (odnosno od procesa koji je izazvao kreiranje datog procesa). U okolinu procesa spadaju vektor argumenata koje proces roditelj prosleđuje programu i vektor okoline, odnosno lista promenljivih koje definišu okolinu procesa (*environment*).

Kontekst procesa je stanje procesa u datom trenutku vremena. Kontekst procesa čine sledeće komponente:

- kontekst za raspoređivanje (*scheduling context*), koji služi za efikasnu suspenziju ili ponovni start procesa. Obuhvata sve CPU registre, prioritet procesa i kernelski stek procesa,
- statistički kontekst, koji sadrži informacije o resursima koje proses koriste u jednom trenutku, kao i kompletnu upotrebu resursa za vreme trajanja jednog procesa (*accounting information*),
- tabela datoteka (*file table*), tj. polje ukazivača na kernelske strukture datoteka**,**
- kontekst sistema datoteka (*file-system context*),
- tabela za upravljanje signalima (*signal-handler table*), koja definiše ukazivače na programe koji se pozivaju nakon određenog signala i
- kontekst virtulene memorije (*virtual-memory context*), koji potpuno opisuje korišćenje memorije od strane procesa.

Vrste procesa

Komunikacija na relaciji komandni interpreter - korisnik se najčešće odvija po principu izvršavanja jednog zadatka u jednom momentu; korisnik zadaje komandu, shell pokreće program i prikazuje izlaz na ekranu. Korisnik čeka da program završi s radom i da *shell* preuzme kontrolu, odnosno da se na ekranu pojavi odziv, pre nego što zada drugu komandu. Procesi koji zahtevaju interakciju sa korisnikom se moraju izvršavati u prvom planu (*foreground*).

Procesi se u pozadini (*background*) izvršavaju se konkurentno sa svim sistemskim i korisničkim procesima. Procesi se izvršavaju u pozadini ukoliko su vremenski zahtevni, ukoliko dominantno koriste procesorsko vreme ili ukoliko ne zahtevaju interakciju sa korisnikom. Kvazi paralelno ili paralelno sa procesima u pozadini izvršava se i proces koji je u prvom planu, a sa kojim korisnik može da ostvari interakciju. Ilustrovaćemo primerom pokretanje procesa u pozadini.

```
$ heavyduty &
[1] 123
```

```
$ tar cf /home /dev/rmt0
$ [1] + Done heavyduty &
```

Kada korisnik zada komandu heavyduty, proces koji je vremenski zahtevan počinje da se izvršava u pozadini. Komandni interpreter odmah vraća odziv, tako da korisnik može da izvrši neku drugu komandu koja zahteva interakciju. Pri tom, korisnik ne može da ostvari interakciju sa procesom koji je inicirala komanda heavyduty, jer se on izvršava u pozadini. Kada proces u pozadini završi svoj rad komandni interpreter na ekranu prikazuje poruku kojom obaveštava korisnika o tom događaju.

Specijalna grupa procesa koji se izvršavaju u pozadini su *daemon* procesi. *Daemon* procesi su aktivni samo u određenim sistuacijama, a inače su neaktivni Na primer, lpd se pokreće pri podizanju sistema s namenom da prihvata zahteve za štampu. Ukoliko nema zahteva, lpd se izvršava ali je neaktivan. Kada stigne zahtev za štampu lpd postaje aktivan dok se zahtev ne odštampa.

Proces koji kreira podproces je proces roditelj (*parent*). Proces roditelj obezbeđuje okolinu koja je neophodna za normalno izvršenje procesa, i može preći u suspendovano stanje za vreme izvršenja podprocesa. Osim procesa init svi procesi imaju svoje roditelje. Proces dete (*child*) je proces koji pokreće proces roditelj. Na primer, ukoliko korisnik u komandnom interpreteru bash zada komandu vi, onda je proces koji vi kreira proces dete komandnog interpretera. Procesi siročići (*orphan*) mogu nastati u svim situacijama kada se nasilno prekine izvršenje proces roditelja. Ukoliko korisnik pokrene komandu iz terminala u grafičkom okruženju i zatvori prozor pre nego što komanda završi svoj rad, proces koji je komanda kreirala postaje proces bez roditelja, odnosno siroče (*orphan*). Da bi se održala hijerarhijska struktura procesa, init kao glavni proces usvaja sve siročiće a zatim prekida njihovo izvršenje. *Zombie* (*defunct*) je proces koji je izgubio vezu s proces roditeljem. *Zombie* ostaje izgubljen u sistemu - jedini resurs koji troši jeste jedno mesto u tabeli procesa, i kao takav ne obara performanse sistema. Ovakav proces se ne može uništiti standardnim metodama, već se uklanja prilikom sledećeg podizanja operativnog sistema.

Kreiranje procesa - fork i exec

Objasnićemo postupak kreiranja procesa na Linux sistemu, pod pretpostavkom da korisnik radi u komandnom interpreteru (*shell*). Novi proces se kreira ukoliko korisnik želi da koristi editor vi ili C prevodioc. Program koji je pokrenut kreira proces, čiji je proces roditelj komandni interpreter. Proces roditelj, u ovom slučaju komandni interpreter, obezbeđuje okolinu koja je neophodna za normalno izvršenje procesa, i prelazi u stanje čekanja (WAIT) za vreme izvršenja podprocesa. Kada podproces normalno ili nasilno završi svoj rad proces roditelj preuzima kontrolu i komandni prompt se vraća korisniku. Ukoliko se podproces smesti u pozadinu, komandni interpreter ne čeka kraj izvršenja procesa u suspendovanom stanju. Komandni prompt

se odmah vraća, a korisnik može zadati drugu komandu i na taj način pokrenuti drugi proces koji će se izvršavati paralelno ili kvazi-paralelno sa prethodnim.

Prilikom kreiranja procesa Linux, kao i UNIX, razdvaja operacije kreiranja procesa i izvršenja programa u resursima novokreiranog procesa. Kada korisnik zada komandu (na primer vi), izvršava se sistemski poziv fork koji kreira novi proces, odnosno izvršava sledeće operacije:

- rezervisanje slobodnog mesta u tabeli procesa
- dodelu jedinstvenog identifikatora procesa (PID)
- kopiranje konteksta proces roditelja
- slanje identifikatora proces roditelju, čime se ostvaruje direktna kontrola podprocesa

Nakon toga, komandni interpreter kao proces roditelj izvršava sistemski poziv exec programa vi, čime se program shell i prateći podaci menjaju programom vi i adekvatnim podacima.

Procesi i niti

Linux koristi istu internu reprezentaciju za procese i niti - nit (*thread*) je jednostavno novi proces koji deli adresni prostor roditelja. Za razliku od novog procesa koji pomoću sistemskog poziva fork formira novi kontekst sa jedinstvenim adresnim prostorom, nit nastaje pomoću sistemskog poziva clone, koji kreira novi kontekst, ali dozvoljava novom procesu da deli adresni prostor roditelja.

Dodeljivanje procesora procesima

Linux koristi dva algoritma za dodelu procesora procesima:

[1] *time-sharing* algoritam

Ovo je algoritam za korektno raspoređivanje između procesa (*fair preemptive scheduling*). Dodela se vrši na osnovu prioriteta procesa koji definiše korisnik i kredita (efektivni prioritet) koji raste s porastom vremena čekanja na procesor po sledećoj rekurzivnoj formuli:

$$kredit = \frac{kredit}{2} + prioritet$$

[2] *real-time* algoritam

Apsolutni prioriteti procesa su mnogo značajniji od ravnomerne raspodele. Linux je ipak soft real-time operativni sistem.

Koji će se algoritam primeniti zavisi od klase u kojoj se proces nalazi (FIFO ili *round-robin*). Trenutna pozicija procesa u svakoj od klasa određuje se na osnovu prioriteta, što znači da će se izvršavati onaj proces koji ima najviši prioritet, a u slučaju da su prioriteti isti, izvšava se proces koji je najduže čekao. U FIFO klasi, procesi nastavljaju da rade sve dok ne završe rad ili ne uđu u blokirano stanje, dok u *round-robin* klasi, svaki proces radi dok mu ne istekne vremenski kvantum (*time-slice*), posle čega prekida rad i i odlazi na kraj liste za čekanje.

Počevši od kernala 2.0, Linux podržava SMP, što znači da se različiti procesi ili niti mogu izvršavati paralelno na posebnim procesorima. Da bi se obezbedila procesorska sinhronizacija kernela u SMP okruženju, samo jedan CPU može izvršavati kod u kernel modu.

Komunikacija između procesa

Komunikacija između procesa obuhvata obaveštavanje procesa o događaju i prenos podataka s jednog procesa na drugi.

[1] Slanje signala

Kao i UNIX sistem, Linux informiše procese u korisničkom režimu o događaju putem signala. Signali se mogu posmatrati kao kratke poruke specijalnog značenja koje se šalju procesima, koje procesi dalje prihvataju ili ignorišu. Signali se koriste za uništenje, privremeni prekid i nastavak izvršenja procesa. Na primer, kombinacija tastera <Ctrl-C> može da uništi proces koji više nije pod kontrolom. Kada korisnik pritisne <Ctrl-C>, aktivnom procesu se šalje signal prekida INT, nakon čega se uništavaju aktivni proces i svi podprocesi koje je on inicirao.

Signali se mogu klasifikovati u dve osnovne kategorije: signali za kontrolu procesa (TERM, KILL, HUP, INT, QUIT), koji se mogu koristiti bez obzira na trenutni komandni interpreter i signali za kontrolu posla (STOP, CONT), koji se mogu koristiti samo ako komandni interpreter podržava kontrolu posla. Podrazumevana akcija koja se izvršava kao posledica signala za kontrolu procesa je uništenje procesa. Podrazumevana akcija koja se izvršava kao posledica signala za kontrolu posla (izuzev signala CONT) je zaustavljanje, odnosno suspenzija procesa. Osim signala KILL, STOP i CONT, proces može da "uhvati" (*catch*) signal, odnosno da izvrši neku drugu akciju umesto podrazumevane. Proces koji hvata signal može da odluči koju će akciju da izvrši kao posledicu datog signala. Na primer, proces koji primi TERM signal može regularno da završi svoj rad (da najpre završi obradu podataka i upiše rezultate na disk, a zatim da prekine izvršenje). Ukoliko proces ne ignoriše ili ne hvata signal, izvršava se podrazumevana akcija.

[2] Deljiva memorija

Procesi koji rade u zaštićenom režimu (*kernel mode*) umesto signala, koriste specijalnu vrstu deljive memorije (wait.queue struktura) za interprocesnu komunikaciju.

[3] Pipe mehanizam

Za prosleđivanje podataka između procesa koristi se *pipe* mehanizam (mehanizam cevi, odnosno tekuće trake), koji omogućava jednosmernu razmenu podataka putem komunikacionog kanala koji proces nasleđuje od roditelja.

Upravljanje memorijom

Upravljanje memorijom obuhvata upravljanje operativnom (RAM) memorijom i upravljanje virtuelnom memorijom.

Upravljanje operativnom, odnosno fizičkom memorijom se bavi dodeljivanjem (alokacijom) i oslobađanjem stranice (*pages, normal extent*), grupe stranica (*large extent*) i malih memorijskih blokova (*small extent*). Upravljanje fizičkom memorijom se obavlja po sistemu drugova (*Buddy heap*). Cela fizička memorija se deli na udružene blokove čije su veličine stepeni broja 2. Blokovi se prema potrebi alokacije dalje razbijaju na manje blokove ili se parovi udružuju u veće celine.

Sistem virtuelne memorije povećava ukupan adresni prostor koji je dostupan procesima - sistem kreira stranicu virtuelne memorije na zahtev, upravlja punjenjem te stanice u fizičku memoriju sa diska i povratkom stranice na disk u swap prostor. Kada stranica mora da napusti memoriju i ode na disk, izvršava se takozvani page-out alogoritam, koji je na Linux sistemu realizovan LFU konceptom (*Least Frequently Used* - najređe korišćen). Novi virtuleni adresni prostor se formira nakon kreiranja novog procesa sistemskim pozivom fork i nakon izvršavanja novog programa sistemskim pozivom exec.

Regioni virtulene memorije obuhvataju: fizičke stanice (*frame*) i *swap* prostor na disku.

Ulazno - izlazni sistem

Linux deli uređaje u tri klase: blok uređaje (poput diskova i CD-ROM uređaja), karakter uređaje (poput štampača) i mrežne uređaje. Svaki uređaj je predstavljen specijalnom datotekom (*device node, device file*) koja se nalazi u direktorijumu /dev root sistema datoteka. Kada korisnik upisuje podatke u datoteku koja predstavlja neki uređaj ili čita iz te datoteke, vrši se neka ulazno-izlazna operacija, odnosno sistem šalje ili prima podatke sa uređaja koji je predstavljen tom datotekom. Time se ukida potreba za postojanjem posebnih programa (a samim tim i posebnom metodologijom

programiranja ulazno-izlaznih operacija) neophodnih za rad sa uređajima. Na primer, korisnik može da odštampa tekst na štampaču jednostavnom redirekcijom standardnog izlaza na datoteku /dev/lp1 koji predstavlja štampač:

```
# cat izvestaj.txt > /dev/lp1
```

Ova komanda će korektno odštampati datoteku izvestaj.txt ukoliko je ona u obliku koji štampač razume (npr. tekstualna datoteka). Međutim, nije preporučljivo da više korisnika istovremeno šalju datoteke na štampač pomoću redirekcije izlaza, jer se ovim zaobilazi red za čekanje za štampač (*print spooler*). Poseban program, lpr (*line printer*) obezbeđuje da datoteke poslate na štampu čekaju u redu, i prosleđuje ih štampaču tek kad je štampanje prethodne datoteke završeno. Slični programi postoje za većinu uređaja, tako da korisnici uglavnom ne koriste specijalne datoteke.

Direktorijum /dev nastaje prilikom instalacije Linux sistema i u njemu se nalaze sve specijalne datoteke, bez obzira na to da li je uređaj instaliran na sistem ili ne - postojanje datoteke /dev/sda ne znači da je na sistem instaliran SCSI disk. Postojanje svih datoteka olakšava proces instalacije novog hardvera, tj. oslobađa administratora sistema potrebe za kreiranjem specijalnih datoteka sa korektnim parametrima.

Sistemi datoteka i aktivno UNIX stablo

Linux sistemi datoteka koriste hijerarhijsku strukturu stabla i semantku UNIX sistema datoteka. Interno, kernel sakriva detalje i upravlja različitim sistemima datoteka preko jednog nivoa apstrakcije, koji se naziva virtuleni sistem datoteka VFS.

Aktivno Linux stablo datoteka čini jedan ili više sistema datoteka koji su montirani na odgovarajuće direktorijume preko kojih im se pristupa. Osnovu aktivnog stabla datoteka čini korenski sistem datoteka (*root filesystem*), čiji koreni (*root*) direktorijum ujedno predstavlja i koreni direktorijum aktivnog stabla datoteka. Zavisno od hardverske konfiguracije i odluke administratora sistema, struktura aktivnog Linux stabla može biti jednostavna (aktivno stablo realizovano jednim sistemom datoteka), ili složena (aktivno stablo realizovano većim brojem sistema datoteka - root, /boot, /var, /usr, /home ...).

Filesystem Hierarchy Standard (FHS) definiše organizaciju aktivnog UNIX stabla i podelu stabla na nekoliko sistema datoteka specifične namene koje treba kreirati na odvojenim particijama ili diskovima. Datoteke su u aktivnom stablu grupisane prema nameni - sve komande se nalaze na jednom mestu, administrativni alati na drugom, dokumentacija na trećem, itd. Većina Linux distribucija sa manjim odstupanjem prati ovaj defacto standard, koji nije obavezan.

17.5. Osnovni servisi Linux sistema

Proces init se pokreće kao prvi proces na svakom Linux sistemu i to je poslednja akcija koju kernel obavlja prilikom podizanja sistema. Kada se pokrene, init nastavlja proces podizanja operativnog sistema, obavljajući razne inicijalne procedure kao što su provera i akitviranje sistema datoteka i pokretanje servisa (*daemons*).

Potpun skup aktivnosti koje će proces init obaviti zavisi od načina podizanja samog Linux sistema. Proces init obezbeđuje koncept jednokorisničkog režima rada (*single user mode*) u kome niko ne može da se prijavi na sistem osim administratora (*root*), koji u ovom režimu može da koristi komandni intepreter isključivo na konzoli. Osim ovog režima rada, koji se koristi u svrhe administarcije ili oporavka sistema, init obezbeđuje i standardni višekorisnički režim rada (*multiuser mode*). Neki režimi rada Linux sistema se generalizuju i nazivaju nivoima izvršenja (*runlevel*) - jednokorisnički i višekorisnički režim predstavljaju dva nivoa izvršenja, a osim njih postoje i dodatni nivoi izvršenja, poput nivoa sa grafičim okruženjem (X) i nivoa sa mrežnim servisima. Linux dozvoljava do 10 nivoa izvršavanja, 0-9, od kojih su samo neki u upotrebi. Nivo izvšenja 0 definiše se kao gašenje sistema (*system halt*), nivo izvršenja 1 kao jednokorisnički režim, a nivo izvršenja 6 kao ponovno podizanje sistema (*system reboot*). Ostali nivoi definisani su u datoteci /etc/inittab za konktretnu Linux distribuciju, pri čemu definicije mogu biti različite za različite distribucije.

U stanju normalnog izvršenja, proces init obezbeđuje funkcionisanje procesa getty, koji korisnicima omogućava proceduru prijavljivanja na sistem (*login*) i usvaja procese siročiće (*orphan*), odnosno procese čiji su roditeljski procesi nestali, čime se održava integritet rodbinskog stabla procesa.

Prilikom zaustavljanja Linux sistema, proces init redom obustavlja sve druge procese, deaktivira sve aktivne sisteme datoteka i zaustavlja procesor, a dodatno može obaviti i druge aktivnosti za koje je konfigurisan.

Navodimo neke značajnije servise Linux sistema:

[1] Prijavljivanje sa terminala

Funkciju prijavljivanja preko serijkih linija i sistemske konzole (bez grafičkog okruženja) obezbeđuje proces getty. Proces init kao proces roditelj pokreće posebnu instancu programa getty za svaki terminal na kome je dozvoljeno prijavljivanje na sistem. Dalje, getty čita korisničko ime i o tome izvršava proces login, koji proverava unešeno korisničko ime i lozinku. Ako su korisničko ime i lozinka korektni, login pokreće proces shell koji prihvata i izvršava korisničke komande. Proces init će otpočeti novu instancu procesa getty kada proces komandnog interpretera završi aktivnost, odnosno kada se korisnik odjavi sa sistema ili ako se login proces završi nekorektnim prijavljivanjem. Evidenciju o *login* procedurama ne vodi kernel, već to obavljaju sistemski programi.

[2] syslog

Kernel i mnogi sistemski programi generišu razna upozorenja i poruke o greškama, koje se upisuju u datoteke, tako da se mogu pregledati nakon izvesnog vremena. Program koji obavlja funkciju upisivanja poruka u datoteke je syslog. Program se može konfigurisati da poruke aranžira u različite datoteke na osnovu stepena značajnosti i procesa koji je poruke generisao. Na primer, poruke koje generiše kernel se kao vrlo značajne poruke koje mogu ukazivati na ozbiljne probleme u sistemu upisuju u posebnu datoteku.

[3] Periodično izvršavanje komandi

Većina korisnika, uključujući sistem administratore često ima potrebu za periodičnim izvršavanjem neke komande. Na primer, sistem administrator može da zakaže periodično izvršenje komande koja čisti direktorijume sa privremenim datotekama (/tmp i /var/tmp), čime sprečava prepunjenje diska. Funkciju periodičnog izvršenja komandi obezbeđuje servis cron. Svaki korisnik može u svojoj crontab datoteci definisati komande koje želi periodično da izvršava i vreme kada te komande treba izvršiti. Cron demon vodi računa o izvršenju komandi u specificiranom vremenu.

Servis koji je sličan cron servisu je at, ali za razliku od cron servisa komandu pokreće samo jednom, u specificiranom trenutku.

[4] Grafički korisnički interfejs

Linux ne ugrađuje korisnički interfejs u kernel - korisnički interfejs se implementira na korisničkom nivou. Koriste se dve vrste interfejsa, alfanumerički i grafički, čime se postiže veća fleksibilnost sistema. Nedostatak ovakvog uređenja je relativno složena implementacija različitih korisničkih interfejsa za svaki program, što operativni sistem čini težim za učenje i korišćenje.

Primarno grafičko okruženje koje se koristi u Linux sistemima je X Window System (skraćeno - X). X ne implementira korisnički interfejs, već obezbeđuje alate pomoću kojih grafički korisnički interfejs može da se implementira. Najčešće korišćeni window menadžeri na Linux sistemima su: KDE (*K Desktop Environment*) i Gnome.

[5] Umrežavanje

Umrežavanje je čin povezivanja dva ili više računara relativno visokog stepena autonomije u cilju omogućavanja njihove međusobne komunikacije. Iako proces povezivanja računara u mrežu i konfigurisanja mrežnog okruženja može biti komplikovan, krajnji rezultat je veoma koristan - korisnici mogu koristiti deljene mrežne resurse poput štampača, direktorijuma i *up-link*ova, a takođe se mogu omogućiti i mehanizmi centralizovane autentifikacije i administracije. Većina osnovnih servisa Linux sistema, poput sistema datoteka, štampanja i arhiviranja podataka može se realizovati pomoću mrežnih funkcija operativnog sistema.

[6] Prijavljivanje sa mreže

Procedura prijavljivanja sa mreže razlikuje se od klasične procedure prijavljivanja na sistem, koja zahteva postojanje posebne fizičke serijske linije za svaki terminal na kom je potrebno omogućiti prijavljivanje i rad. Za svakog korisnika koji se prijavljuje na sistem preko mreže, postoji posebna virtuelna mrežna konekcija. Broj tih konekcija teorijski može biti neograničen, tako da nije moguće izvršavati poseban getty proces za svaku virtuelnu konekciju.

Za potrebe prijavljivanja sa mreže, umesto gomile getty procesa, postoji glavni demon proces (*master daemon, wrapper*) koji osluškuje sve nadolazeće zahteve. Ukoliko se pojavi zahtev za prijavljivanje na sistem, wrapper kao proces roditelj pokreće novi proces (telnet ili rlogin *daemon*), sličan getty procesu, koji dalje upravlja pokušajem prijavljivanja, dok glavni demon nastavlja da osluškuje nove zahteve sa mreže.

[7] Deljenje datoteka i mrežni sistemi datoteka

U značajnije servise UNIX i Linux sistema spada i deljenje datoteka na mreži. Mrežni sistemi datoteka funkcionišu na sledeći način: zahtevi za operacijama nad datotekama i direktorijumima šalju se po mreži na računar na čijim se diskovima sistem datoteka nalazi, a korisnik ima utisak da se sve datoteke nalaze na lokalnom sistemu datoteka. Na ovaj način se omogućava deljenje datoteka na krajnje jednostavan način, pošto se ne zahteva nikakva modifikacija korisničkih programa.

Najčeše korišćeni tip mrežnih sistema datoteka je NFS (*Network File System*), koji je razvila kompanija Sun Microsystems. Drugi servis, koji omogućava pristup Linux mrežnim sistemima datoteka sa MS Windows radnih stanica, je Samba (http://www.samba.org). Po istim protokolima je moguće deliti i štampače.

[8] Elektronska pošta

Elektronska pošta je najpopularniji metod komunikacije putem računara. Svaki korisnik ima prijemno poštansko sanduče (*incoming mailbox*) u vidu datoteke u specijalnom formatu, koja se po pravilu nalaze na direktorijumu /var/spool/mail. Kada neko šalje poštu, specijalan program koji omogućava prijem i slanje pisama locira prijemno poštansko sanduče i dodaje pismo u mailbox datoteku. Ukoliko se prijemno poštansko sanduče nalazi na drugom računaru, pismo se najpre šalje preko mreže ka destinacionom računaru, čiji će mail program dalje pismo distribuirati u odgovarajuće sanduče.

Poštanski sistem čine dve osnovne vrste programa: MTA - *Mail Transfer Agent* (na primer sendmail), koji vrše isporuku pošte u lokalno poštansko sanduče i prosleđuju poštu udaljenim računarima, i MUA - *Mail User Agent* (pine, mutt ili elm), koje služe korisnicima za čitanje i slanje pošte.

[9] Štampanje

Jedan štampač u jednom vremenskom trenuntku može da koristi samo jedan korisnik, ali je krajnje neekonomično ne dozvoliti upotrebu štampača u različitim vremenskim trenucima većem broju korisnika. Zato štampačima upravlja program koji implemetira red čekanja za dati štampač (*printer queue*): svi zahtevi za štampu se postave u red čekanja i kada štampač obavi jedan posao upravljački program mu automatski šalje sledeći. Na ovaj način se omogućava korišćenje štampača većem broju korisnika pri čemu su korisnici oslobođeni organizacije reda čekanja za štampač.

Program koji implementira red čekanja čuva kopiju koju treba štampati na disku print servera, čime se omogućava aplikativnom programu da preda zahtev za štampu u red i nakon toga nastavi svoje aktivnosti. Disk je relativno brz uređaj u odnosu na štampač, tako da predaja zahteva u red traje kratko. Nakon predaje zahteva, aplikativni program ne mora da čeka da zahtev bude odštampan, a takođe ne mora ni da kontroliše proces štampe, tako da nastavlja s daljim radom. Ovakav način organizacije štampanja je jako povoljan, zato što korisnik može da izda zahtev za štampanjem jednog dokumenta, zatim otvori drugi dokument i pošalje ga na štampu bez ikakvog čekanja.

18

Mac OS X

Sadržaj poglavlja:

18.1. Istorijat
18.2. Uopšteno o Mac OS X operativnom sistemu
18.3. XNU - kernel
18.4. Mac OS X s programerske tačke gledišta
18.5. Softver dostupan za Mac OS X

18.1. Istorijat

Steve Jobs, zaposlen u kompaniji Apple, 12. oktobra 1988. godine predstavio je prvi NeXT računar, koji je radio pod operativnim sistemom NEXTSTEP 0.8. Računar je namenjen prvenstveno institucijama koje se bave istraživanjima, poput univerziteta i laboratorija. Godinu dana kasnije objavljena je stabilna verzija operativnog sistema - NEXTSTEP 1.0, čiju osnovu čine Mach 2.5 i 4.3BSD. NEXTSTEP grafičko okruženje zasnovano na Postscriptu, a kao osnovni programski jezik koristi se objektno orijentisani C. NEXTSTEP 2.0 predstavljen je 18. septembra 1990. U ovu verziju je, između ostalog, uključena podrška za CD-ROM uređaje, kolor monitore, NFS i drajvere za uređaje koji se dinamički učitavaju u memoriju. Zatim slede verzije 2.1 i 3.0, 3.1 i 3.2, i NEXTSTEP 486, namenjen PC arhitekturi. Poslednja verzija NEXTSTEP operativnog sistema objavljena je u februaru 1995. godine. Godinu dana ranije, NeXT i Sun definisali su OpenStep platformu kao kompromis nekoliko API-ja i *framework*a; korišćenjem ove platforme moguće je implementirati operativni sistem prilagođen konkretnim potrebama. NeXT implementacija nazvana je OPENSTEP i

može se smatrati naslednikom NEXTSTEP operativnog sistema. OPENSTEP je objavljen u verzijama 4.0 (jul 1996), 4.1 (decembar 1996) i 4.2 (januar 1997).

U međuvremenu, kompanija Apple je nekoliko puta neuspešno pokušala da realizuje operativni sistem koji bi na tržištu mogao da se takmiči sa konkurencijom (najjači konkurent u to vreme bio je Microsoft Windows 95). Pomenućemo Pink OS (na čijem radu je učestvovala i kompanija IBM) i operativni sistem sa kodnim imenom Copland kao neuspešne pokušaje; takođe, kupovina BeOS operativnog sistema je u jednom trenutku predstavljala moguće rešenje. Sa nešto više uspeha započet je projekat Rhapsody, zasnovan na NeXT operativnom sistemu i delimično na Copland tehnologiji. Rhapsody, koji predstavlja značajno unapređenje u odnosu na postoji Mac OS, objavljen je u septembru 1997 i maju 1998, oba puta kao razvojno izdanje (*developer release*).

Mac OS X, zasnovan na Mac OS i Rhapsody, je nakon toga najavljen kao osnovni pravac daljeg razvoja Apple operativnih sistema. Mac OS X je prvi put objavljen 1999 godine, kao Mac OS X Server 1.0 (razvojna verzija izdanja za radne stanice) i kao Darwin 0.1. Mac OS X beta ja objavljen godinu dana kasnije. U vreme dok pišemo ovu knjigu, Mac OS X je objavljen u četiri komercijalne verzije:

- Mac OS X 10.0 (Cheetah, mart 2001),
- Mac OS X 10.1 (Puma, septembar 2001),
- Mac OS X 10.2 (Jaguar, avgust 2002) i
- Mac OS X 10.3 (Panther, oktobar 2003).

A/UX

Nevezano za Mac OS X, treba reći da je Apple razvijao i svoju verziju UNIX sistema, nazvanu A/UX (prva verzija A/UX objavljena je 1988. godine). A/UX je sistem usklađen sa POSIX standardom, zasnovana na AT&T System V i BSD4.2/4.3 UNIX sistemima. Pomenućemo neke od značajnijih elemenata ugrađenih u A/UX: mehanizam tokova, TCP/IP skup protokola, kontrola posla, NFS, Yellow Pages, štampanje, X Window sistem, kompatibilnost sa System V i BSD standardima i usklađenost sa POSIX-om. Još značajnija je činjenica da je A/UX kombinovao razne elemente Macintosh i UNIX operativnih sistema, što je omogućavalo pokretanje Macintosh aplikacija, kao što je Finder. A/UX je poslednji put objavljen 1995. godine. u verziji 3.1.1.

18.2. Uopšteno o Mac OS X operativnom sistemu

Mac OS X je operativni sistem koji kombinuje snagu UNIX-a i jednostavnost Mac OS-a. Delovi softvera koji čini Mac OS X mogu se grupisati u nekoliko logičkih celina, koje se oslanjaju na osnovu koju čini Darwin.

Darwin - temelj Mac OS X

Darwin se može posmatrati kao operativni sistem u celini ili kao skup tehnologija koje čine temelj Mac OS X. Darwin 7.0.x (koji odgovara Mac OS X verzijama 10.3.x) sastoji se od preko 250 paketa. Većina paketa su Apple paketi, a takođe ima i BSD u GNU paketa; Apple je iskoristio prednost postojećeg *open source* softvera integrišući u Darwin neke značajnije pakete, kao što su apache, gcc, kerberos, mysql, postfix i samba, zatim editori emacs i vim, standardni UNIX skup alata za rad sa datotekama (cp, mv, ls, gnutar), bash komandni interpreter i skript jezici Perl, PHP, tcl, Ruby i Python. Izvorni kod nekih *open source* paketa (kao što je gcc) je modifikovan i prilagođen platformi. Originalni Apple paketi uključeni u Darwin najčešće su licencirani *Apple Public Source* licencom (APSL).

Darwin se može okarakteristi kao *soft real-time* višeprocesni operativni sistem sa pretpražnjenjem koji podržava simetrično multiprogramiranje. Namenjen je za PowerPC i x86 platforme, s tim što je moguće izgraditi kernel koji će sadžiti izvršni kod za obe platforme u jednoj datoteci. Modularni dizajn dozvoljava dinamičko učitavanje drajvera za uređaje i podrške za rad sa sistemima datoteka. U Darwin je ugrađena podrška za rad sa TCP/IP mrežnom arhitekturom (uključujući IPv4, IPv6 i IPSec protokole).

Grafički podsistemi

Osnovni delovi Mac OS X grafičkog podsistema su:

[1] Quartz

Quartz je skup 2D grafičkih tehnologija koje čine osnovu za rad sa rasterskom i vektorskom grafikom. Sastoji se od dva osnovna dela: Quartz 2D biblioteke i Quartz Compositor sistema koji upravlja prozorima i ekranskom prezentacijom Quartz 2D, OpenGL i QuickTime sadržaja. Quartz Compositor je "vlasnik" svih piksela na ekranu koji u skladu sa zahtevima konkretne aplikacije određuje sadržaj prozora. Ovaj centralizovani pristup omogućava oslobađa aplikaciju problematike upravljanja prozorima.

[2] Quartz Extreme

Na sistemima sa odgovarajućim grafičkim adapterima, Quartz Compositor korišćenjem intergrisane OpenGL tehnologije preusmerava operacije formiranja konačne slike na ekranu (*onscreen rendering*) na GPU (*Graphics Processing Unit*) video adaptera. Na ovaj način, procesor se rasterećuje, a sama aplikacija dobija više procesorskog vremena; čime se povećavaju performanse sistema i smanjuje vreme odziva.

[3] OpenGL

Za rad sa 3D grafikom, u Mac OS X je implementiran OpenGL standard za formiranje slike na osnovu 3D objekata i tekstura. Kombinovanjem 2D i 3D grafike, omogućena je integracija specijalnih poboljšanja korisničkog interfejsa, kao što je Exposé.

[4] QuickTime

QuickTime 6, integrisan u Mac OS X obezbeđuje podršku za rad sa multimedijalnim sadržajem (video zapisom, zvukom i animacijom) i konverziju više od 250 različitih formata. QuickTime uključuje podršku za HTTP, RTP (*Real-Time Transport Protocol*) i RTSP (*Real-Time Streaming Protocol*), kao i za 3GPP (mobilna telefonija treće generacije), gde se može koristiti za prikaz datoteka koje su u skladu sa 3GPP standardom.

Napomenimo da je u Mac OS X implementiran *X Window System* čime je omogućeno izvršavanje X11 aplikacija. X11 aplikacije se izvršavaju konkurentno sa svim ostalim aplikacijam i koriste prednosti Mac OS X grafičkog podsistema koji obezbeđuje brz odziv i hardversku akceleraciju za 2D i 3D grafičke operacije.

Upravljanje podacima i informacijama

Mac OS X organizuje podatake prema sadržaju i na taj način sprečava nedeterminističko uređenje podataka na sekundarnim memorijama. Na primer, podešavanja vezana za sistem i aplikacije čuvaju se na određenom mestu u zavisnosti od toga da li su globalna ili vezana za konkretnog korisnika. Takođe, razni API-ji obezbeđuju determinističko smeštanje korisničkih podataka, odnosno datoteka (pod uslovom da su pravilno implementirani). Jedna od mogućnosti pogodnih za pojednostavljivanje procesa reinstalacije ili nadogradnje Mac OS X operativnog sistema je sinhronizacija konfiguracije (*personality*) računara. Konfiguracione informacije starog sistema mogu se snimiti na uređaj (kao što je iPod) ili .Mac nalog, a zatim i jednostavno preneti na novu instalaciju sistema. Apple je najavio proširenje sinhronizacionog seta koji za sada obuhvata relativno uski skup podataka.

Direktorijumsko stablo Mac OS X sistema po strukturi je slično stablu tradicionalnih UNIX sistema. Na primer, na Mac OS X postoje direktorijumi /usr, /etc (simbolički link na /private/etc) i /tmp (simbolički link na /private/tmp), ali i neki

specifični, kao što su /Applications (u kome se nalaze aplikacije koje rade u grafičkom okruženju), /Developer (Xcode razvojni alati), /Library, /Network, /System i /Users. U okviru ličnog direktorijuma korisnika nalaze se poddirektorijumi Desktop, Documents, Music, Movies, Pictures, Public i Sites koji dozvoljavaju korisniku da lične podatke uredi na odgovarajući način.

Sistemi datoteka

Kao većina modernih operativnih sistema, Mac OS X apstrahuje realne sisteme datoteka pomoću VFS sloja. VFS sloj XNU kernela zasnovan je na FreeBSD tehnologiji uz neke manje značajne izmene (na primer, FreeBSD kao sinhronizacioni mehanizam koristi *mutex*e, a XNU zaključava datoteke).

Kao podrazumevani domaći sistem datoteka na Mac OS X operativnom sistemu koristi se HFS+. HFS+ je naslednik HFS sistema datoteka (*Hierarchical File System*), koji je korišćen na Macintosh Plus i novijim modelima, zaključno sa Mac OS 8.1.

Arhitektura HFS+ i HFS sistema datoteka je slična, s tim što su u HFS+ sistemu prisutna neka značajnija unapređenja:

- blokovi podataka se alociraju sa 32-bita, umesto sa 16. HFS deli diskove na blokove jednake veličine; na jednoj HFS particiji može se naći najviše 2^{16} blokova. Korišćenjem 32-bitne alokacije, veličina bloka podataka se smanjuje, jer se prostor može izdeliti na 2^{32} blokova. Samim tim se smanjuje i neiskorišćeni prostor na disku, odnosno gubici.
- duga Unicode imena datoteka (do 255 karaktera).
- proširiv skup atributa kojim su opisane datoteke
- najveća veličina datoteke na HFS+ sistemu datoteka je 2^{63} bajta.
- kao dodatak uz System Folder ID (zadužen za pokretanje Apple operativnih sistema), HFS+ po potrebi čuva specijalnu datoteku koja omogućava podizanje drugih operativnih sistema sa HFS+ sistema datoteka. Lokacija i veličina datoteke opisane su u zaglavlju sistema datoteka na fiksnoj lokaciji, tako da se datoteka može jednostavno pronaći pri podizanju sistema.

HFS+ je *case preserving - case insensitive* sistem datoteka, što znači da pamti da li je korisnik uneo ime datoteke velikim ili malim slovima, ali između malih i velikih slova ne pravi funkcionalnu razliku. U HFS+ uključena je podrška za postavljanje kvota, zaključavanje na nivou bajta i kreiranje hard i simboličkih linkova. Za svaku pojedinačnu datoteku može se odrediti da li je potrebno da se sakrije njena ekstenzija ili ne.

Umesto simboličkih linkova mogu se kreirati aliasi. Aliasi su slični simboličkim linkovima u smislu da dozvoljavaju kreiranje višestrukih referenci na datoteku ili direktorijum. Međutim, ukoliko se referencirani objekat pomeri, za razliku od

simboličkog linka alias ne gubi vezu sa objektom. Veza se gubi u slučaju da se referencirani objekat zameni novim objektom. Ovo se u HFS+ sistemu datoteka može ostvariti jer svaka datoteka/direktorijum ima svoj jedinstveni identitet koji se čuva zajedno sa putanjom. Ukoliko je putanja neispravna, alias traži datoteku na osnovu identiteta i ažurira putanju ka datoteci. Korišćenje aliasa je moguće samo kroz Carbon ili Cocoa API-je.

HFS+ je sistem datoteka sa vođenjem dnevnika transakcija. Kao i većina journaling sistema datoteka, HFS+ prati samo promene u meta-strukturama (promene se registruju u datotekama .journal i .jornal_info_block). Sam proces praćenja promena i formiranja dnevnika implementiran je na VFS journaling sloju kernela, i kao takav je nezavistan od tipa sistema datoteka.

Ukratko ćemo opisati optimizacione mehanizme koje koristi HFS+. Prilikom otvaranja datoteke HFS+ obavlja *on-the-fly* defragmentaciju datoteka ukoliko su ispunjeni sledeći uslovi: datoteka je fragmentisana, veličina datoteke je manja od 20MB, datoteka nije već otvorena, datoteka nije *read-only* (može se otvoriti za čitanje i upis), sistem je funkcionalan duže od tri minuta.

Drugi optimizacioni mehanizam je "*Hot File Clustering*". *Hot space* veličine 0.5% kapaciteta sistema datoteka rezerviše se na početku područja podataka (iza područja sa meta-strukturama). Sistem prati aktivnosti i u ovo područje smešta datoteke za koje se na osnovu podataka u B-stablu (./hotfiles.btree) utvrdi da su *hot files* (izuzev dnevnika i datoteka kojima su opisane kvote). U hot space može se smestiti najviše 5000 datoteka, pri čemu sve moraju biti manje od 10MB.

Za smeštaj podataka na CD-ROM medijumima Apple koristi ISO9660 sitem datoteka sa specijalnim proširenjima. Hibridni HFS/ISO9660 CD-ROM medijumi sadrže validne HFS i ISO9660 sisteme datoteka, od kojih su na Mac OS X sistemu vidljiva oba, dok se na ostalim sistemima mogu pročitati samo podaci smešteni na ISO9660 sistemu datoteka. To ne znači da se na diskovima pojavljuje redundansa - obično su podaci koji trebaju biti vidljivi sa svih operativnih sistema smešteni na ISO9660, dok se na HFS sistemu datoteka kreiraju aliasi tih podataka.

Od ostalih lokalnih sistema datoteka, na Mac OS X sistemu mogu se koristiti:

- FAT12, FAT16, FAT32,
- NTFS (samo u režimu čitanja),
- UDF (*Universal Disk Format*) - sistem datoteka koji se koristi za smeštanje podataka na DVD-ROM medijumima, uključujući DVD-video i DVD-audio diskove,
- UFS - Darwin implementacija UFS sistema datoteka nije potpuno kompatibilna sa implementacijom na BSD UNIX sistemima i trenutno je podržana samo u NetBSD sistemima.

Od mrežnih sistema datoteka, u Mac OS X ugrađena je podrška za AFP (*Apple Filing Protocol*) kao osnovni protokol za deljenje datoteka na mreži, FTP (trenutno

samo u režimu čitanja), NFS, SMB (Samba 3.0) i WebDAV (kojim je omogućeno aktiviranje URL-a kao sistema datoteka; iDisk sa .Mac naloga dostupan je preko WebDAV protokola).

Zanimljivo je da u Mac OS X nije ugrađena podrška za rad sa ext2, ext3 i ReiserFS sistemima datoteka (Linux).

Umrežavanje

U Mac OS X integrisani su sledeći mrežni protokoli: TCP/IP, IPv4, IPv6, IPSec, BOOTP, DHCP, DNS, FTP, HTTP, IMAP, Kerberos, L2TP, LDAP, CUPS, NIS, NTP, PPP, PPTP, SLP, SMB/CIFS, SSH, SSL. Mac OS X obezbeđuje podršku za rad sa mrežnim servisima vezanim za datoteke (AFP, SMB/CIFS,WebDAV i NFS), sigurni udaljeni pristup preko VPN i interoperabilnost sa Aktivnim Direktorijumom. Mac OS X volumeni na Windows mreži vidljivi su preko SMB protokola. Bežični servisi dostupni su preko AirPort Extreme (IEEE 802.11g) i Bluetooth tehnologija, a moguć je i pristup standardnim Ethernet mrežama (10/100/1000Base-T tehnologije). Dodatno, komapnija Apple je definisala i u Mac OS X ugradila *Rendezvous* protokol pomoću kojih se mrežni uređaji mogu pronaći na mreži bez eksplicitnog navođenja IP adresa ili konfigurisanja DNS servera. Ova tehnologija je takođe poznata pod imenom *Zero Configuration Networking*.

Sigurnost

U Mac OS X operativnom sistemu postoje sigurnosni servisi i API-ji za autentifikaciju, šifrovanje, upravljanje lozinkama, sertifikatima, privatnim i javnim ključevima. Mac OS X uključuje podršku za Kerberos, OpenSSL, i PAM (*Plugable Authentification Modules* - izmenljive autentifikacione module). Većina ovih servisa je dostupna putem *Keychain Services* API-ja koji mogu koristiti sve aplikacije; moguće je na primer koristiti jednu lozinku za pristup lancu umesto različitih lozinki koje zahtevaju aplikacije.

U podrazumevanom stanju (*default*) zabranjeno je prijavljivanje *root* korisnika na sistem - *root* privilegije ostvaruju se korišćenjem programa *sudo*.

U relativno skora unapređenja sigurnosti spadaju FileVault (šifrovanje home direktorijuma korisnika) i Secure File Deletation (kojim se sprečava povratak obrisanih datoteka).

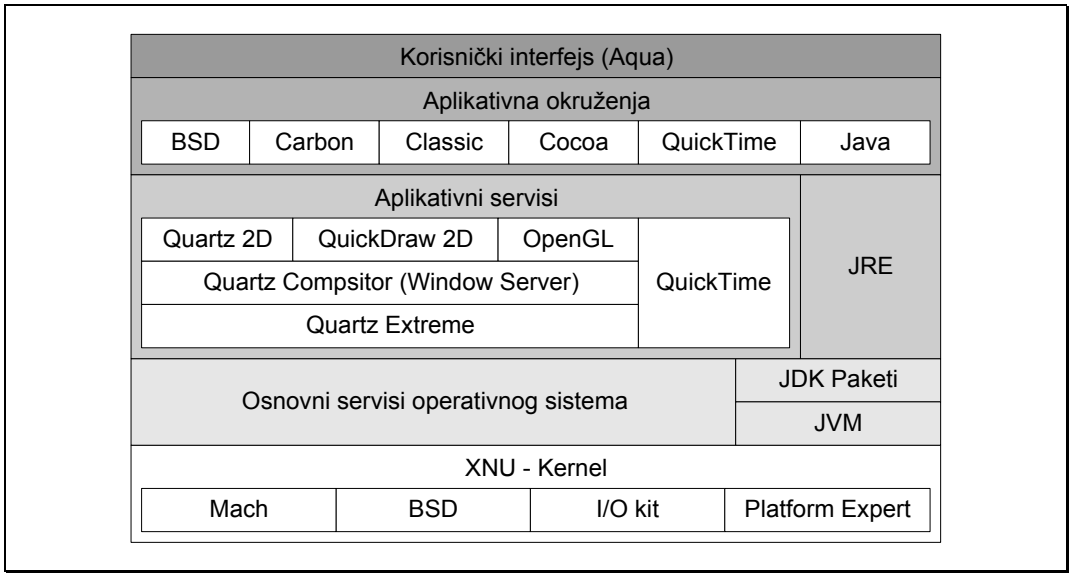
Mac OS X Server

U serversku varijantu Mac OS X operativnog sistema (Mac OS X Server) integrisani su servisi za deljenje datoteka, upravljanje klijentima, web i mail servisi.

Navodimo značajniji serverski softver koji se isporučuje uz Mac OS X Server: Apache web server, QuickTime Streaming Server, JBoss application server, Tomcat, Apache Axis, Samba, i MySQL. Mac OS X Server se isporučuje sa Server Admin aplikacijom koja obezbeđuje intuitivan interfejs za upravljanje (pokretanje i zaustavljanje) i nadgledanje mrežnih servisa (praćenje log datoteka, ukupne aktivnosti, aktivnosti mreže, procesora i slobodnog prostora na disku).

18.3. Arhitektura Mac OS X operativnog sistema

Na slici 18.1 prikazana je arhitektura Mac OS X operativnog sistema:



Slika 18.1. *Arhitektura Mac OS X operativnog sistema*

XNU - kernel

XNU, kernel Mac OS X operativnog sistema sastoji se od sledećih komponenti: Mach, BSD, I/O Kit, Platform Expert.

[1] Mach

XNU sadrži kod zasnovan na Mach arhitekturi koja je integrisana u neke značajnije sisteme. Razvoj Macha započet je na Carnegie Mellon University. Mach 3.0 sadrži prvu mikrokernel implementaciju; prethodni Mach kerneli bili su monolitni i sadržali su dosta BSD koda.

XNU Mach komponenta zasnovana je na Mach 3.0, ali je u ovom slučaju implementirana kao deo kernela, a ne kao mikrokernel. Mach komponenta kernela odgovorna je za:

- multiprogramiranje sa pretpražnjenjem, uključujući i kernelske niti (POSIX niti su na Mac OS X implementirane preko kernelskih niti),
- međuprocesnu komunikaciju,
- zaštićenu memoriju,
- upravljanje virtuelnom memorijom i
- upravljanje prekidima.

Dodatno, Mach komponenta upravlja ulazno-izlaznim operacijama sa konzole i obezbeđuje podršku za *real-time* aplikacije i otklanjanje grešaka pomoću ddb kernel debugger-a i implementacije *remote kernel debugging* protokola.

[2] BSD

XNU BSD komponenta zasnovana je na FreeBSD kodu (Darwin 7.x, odnosno Mac OS X 10.3.x koristi FreeBSD 5.x). Kao i Mach, BSD se izvršava kao deo kernela i odgovoran je za sledeće:

- model procesa,
- identifikatore korisnika, dozvole i osnovne sigurnosne polise ,
- POSIX API, BSD sistemske pozive,
- TCP/IP skup protokola, BSD priključke (*sockets*), mrežnu barijeru (*firewall*). XNU kernel koristi NKE mrežna proširenja (*Network Kernel Extensions*), radi kompatibilnosti sa 4.4BSD mrežnom arhitekturom,
- VFS i sisteme datoteka,
- System V IPC (interprocesna komunikacija),
- kriptografski *framework* i,
- sinhronizacione mehanizme (XNU koristi apstrakciju sinhronizacije zasnovanu na Mach *mutex*ima radi serijalizacije pristupa BSD delovima kernela).

[3] I/O kit

Sledeća komponenta XNU kernela je objektno orijentisani drajverski *framework* - I/O Kit. Kao programski jezik, I/O Kit koristi podskup C++ jezika u koji nije uključeno rukovanje izuzecima, višestruko nasleđivanje i korišćenje šablona (*templates*). Model drajvera koji obezbeđuje I/O kit između ostalog uključuje i podršku za različite familije uređaja (ATA, FireWire, PCI, USB), Plug-and-play tehnologiju, multiprogramiranje sa pretpražnjenjem, niti, SMP, zaštitu memorije, upravljanje napajanjem, dinamičko učitavanje, objektno orijentisanu apstrakciju uređaja koji mogu biti deljeni.

[4] Platform Expert

Platform Expert identifikuje platformu na kojoj se sistem izvršava i formira stablo priključenih uređaja. Stablo se formira tako što Platform Expert najpre puni memoriju drajverom za identifikovanu platformu; zatim se identifikuju sve prisutne magistrale i svi uređaji koji su na njih vezani. Dodatno, Platform Expert inicijalizuje korisnički interfejs koji se može koristiti u slučaju panike kernela (*kernel panic*).

Osnovni servisi

Sloj osnovnih servisa se u hijerarhijskom modelu nalazi iznad kernela. Na ovom sloju obezbeđuju se kritični sistemski servisi koji nisu vezani za grafički korisnički interfejs. Komponente ovog sloja su:

- OSServices - framework koji uključuje sistemske API-je (za upravljanje procesima, nitima, virtuelnom memorijom, sistemima datoteka...),
- CFNetwork - API za umrežavanje na korisničkom nivou koji uključuje nekoliko protokola (FTP, HTTP, LDAP, SMTP, ...),
- CarbonCore - kritični delovi Carbon aplikativnog okruženja i
- WebSevicesCore - API-ji za korišćenje Web servisa preko SOAP i XML-RPC.

Aplikativni servisi

Nalaze se iznad osnovnih servisa u hijerarhijskom modelu i uključuju servise koji čine Mac OS X grafičko okruženje.

Osnovna komponenta grafičkog podsistema na ovom sloju je Quartz, koji se sastoji od dva osnovna dela:

- Quartz 2D biblioteke za rad sa dvodimenzionalnom grafikom i
- Quartz Compositor sistema koji upravlja prozorima i ekranskom prezentacijom Quartz 2D, OpenGL i QuickTime sadržaja (*Window Server*).

Na sistemima sa odgovarajućim grafičkim adapterima, Quartz Compositor preusmerava operacije formiranja konačne slike na ekranu na procesor video adaptera, čime se rasterećuje procesor, a sama aplikacija dobija više procesorskog vremena (Quartz Extreme). Za rad sa 3D grafikom, u Mac OS X je implementiran OpenGL standard za formiranje slike na osnovu 3D objekata i tekstura. QuickTime 6, integrisan u Mac OS X obezbeđuje podršku za rad sa multimedijalnim sadržajem i konverziju formata.

Pomenućemo još neke značajnije komponente sloja aplikativnih servisa:

- Apple Events - mehanizam za komunikaciju između aplikacija,
- CoreGraphics - grafički *framework* korišćen za implementaciju Quartza,
- FindByContent - API koji obezbeđuje funkcije pretraživanja volumena i direktorijuma (traže se datoteke sa zadatim sadržajem) i
- Printcore - API koji obezbeđuje funkcionalnost podsistema za štampanje.

Aplikativna okruženja

Mac OS X he izveden iz različitih izvora (BSD, Mac OS) i kao takav sadrži nekoliko okruženja u kojima se aplikacije izvršavaju.

[1] BSD

Ovo okruženje je slično tradicionalnom BSD sistemu i kao takvo obezbeđuje BSD zasnovan POSIX API. Sastoji se od BSD izvršnog dela (*runtime*) i okoline za izvršavanje aplikacija. BSD okruženje zasnovano je na FreeBSD sistemu (Panther je na primer zasnovan na FreeBSD 5.0). Zbog usklađenosti sa UNIX-om, biblioteke i zaglavlja se nalaze na svojim tradicionalnim lokacijama: /usr/lib i /usr/include, respektivno.

[2] Carbon

Carbon je skup proceduralnih API-ja zasnovanih na Mac OS 9 API-jima. Carbon ne uključuje sve Mac OS 9 API-je; neki su izmenjeni radi usklađenosti sa OS X sistemom, a neki izostavljeni jer se na OS X sistemu ne mogu implementirati (zbog značajnih razlika između ova dva operativna sistema).

[3] Classic

Classic aplikativno okruženje je emulator kojim se ostvaruje kompatibilnost sa Mac OS 9 sistemom. Aplikacije napisane za Mac OS 9 sistem mogu se izvršavati u Classic okruženju na Mac OS X sistemu. Apple ne preporučuje razvoj aplikacija za ovo okruženje.

[4] Cocoa

Cocoa je objektno orijentisani API za razvoj aplikacija u objektnom C jeziku i Javi. Cocoa je značajno nasleđe NEXTSTEP sistema, velikim delom zasnovan na OpenStep *framework*u (činjenica je da razna imena u API-ju počinju sa ns). Apple alati za brz razvoj aplikacija podržavaju Cocoa okruženje. Cocoa aplikacija može pozvati Carbon API.

[5] Java

Ovo okruženje čine JDK (u komandnoj liniji i integrisan u Apple razvojno okruženje), izvršni deo (Hotspot VM, JIT) i kolekcija Java klasa (AWT, Swing, ...).

Korisnički interfejs

Aqua, korisnički interfejs prisutan na poslednjoj verziji Mac OS X operativnog sistema (Panther) po funkcionalnosti se može porediti sa Aerom na Longhornu. Aqua se oslanja na snagu i funkcionalnost grafičkih podsistema. Grafička aplikativna okruženja Carbon, Cocoa i Java podržavaju Aqua korisnički interfejs; Classic emultator ne podržava Aqua interfejs, tako da Mac OS 9 aplikacije koje se u njemu izvršavaju izgledaju kao i na Mac OS 9. U MacOS X implementiran je X Window server čime je omogućeno izvršavanje X11 aplikacija. Za upravljanje prozorima koji pripadaju X11 aplikacija zadužen je Aqua *window manager* (quartz-wm).

Literatura

[1] D. E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Edition, Addison-Wesley (1973).

[2] *Trusted Computer System Evaluation Criteria*. Department of Defence (1985).

[3] D. Grosshans, *File Systems Design and Implementation*, Prentice Hall (1986).

[4] P. Norton, *Inside the IBM PC*, Brady Books (1986).

[5] *iAPX 386 Programmer's Reference Manual*. Intel Corporation (1986).

[6] M. J. Bach, *The Design of the UNIX Operating System*, Prentice Hall (1987).

[7] *Apple Technical Introduction to the Macintosh Family*. Addison-Wesley (1987).

[8] R. Kvaternik, *Uvod u operativne sisteme, četvrto izdanje*, Informator, Zagreb (1988).

[9] S. J. Leffler, M. K. McKusick, M. J. Karels, J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley (1989).

[10] *i486 Microprocessor*. Intel Corporation (1989).

[11] A. S. Tannenbaum, *Structured Computer Organization, Third Edition*, Prentice Hall (1990).

[12] *i486 Microprocessors Programmer's Reference Manual*. Intel Corporation (1990).

[13] *Microsoft MS-DOS User's Guide and Reference*. Microsoft Press (1991).

[14] *Inside Macintosh, Volume VI*. Addison-Wesley (1991).

[15] B. Stroustrup, *Programski jezik C++*, Mikro knjiga (1991).

[16] H. M. Deitel, M. S. Kogan, *The Design of OS/2*, Addison-Wesley (1992).

[17] A. M. Lister, R. D. Eager, *Fundamentals of Operating Systems, Fifth Edition*, The Macmillan Press Ltd (1993).

[18] S. Mullender, *Distributed Systems, Second Edition*, ACM Press (1993).

[19] S. Rago, *UNIX System V Network Programming*, Addison-Wesley (1993).

[20] *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*, Corporation (1993).

[21] H. Custer, *Inside the Windows NT File System*, Microsoft Press (1994).

[22] S. Sengupta, C. P. Korobkin, *C++ Object-Oriented Data Structures*, Springer-Verlag (1994).

[23] M. A. Miller, *Internetworking: a Guide to Network Communication LAN to LAN, LAN to WAN, Second Edition*, M&T Books (1995).

[24] B. Schneier, *Applied Cryptography, Second Edition*, John Wiley & Sons, Inc. (1996).

[25] A. S. Tannenbaum, *Computer Networks, Third Edition*, Prentice Hall (1996).

[26] *Microsoft Windows NT Workstation Reference Kit*. Microsoft Press (1996).

[27] M. Brain, *Win32 System Services, Second Edition*, Prentice Hall (1996).

[28] M. K. McKusick, K. Bostic, M. J. Karels, *The Design and Implementation of the 4.3BSD UNIX Operating System*, John Wiley and Sons (1996).

[29] S. Jovanović, *UNIX, uvod u sistem, mrežno okruženje i administraciju*, Institut za nuklearne nauke "Vinča" (1996).

[30] A. S. Tannenbaum, A. S. Woodhull, *Operating System Design and Implementation, Second Edition*, Prentice Hall (1997).

[31] A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Inc. (1997).

[32] J. Richter, *Advanced Windows*, Microsoft Press (1997).

[33] J. Gray, *Interprocess Communication in UNIX*, Prentice Hall (1997).

[34] D. A. Solomon, *Inside Windows NT, Second Edition*, Microsoft Press (1998).

[35] J. Murray, *Inside Microsoft Windows CE*, Microsoft Press (1998).

[36] *FreeBSD Handbook*, The FreeBSD Documentation Project (1999).

[37] N. Rhodes, J. McKeehan, *Understanding the Linux Kernel*, O'Reily and Associates (1999).

[38] R. C. Leinecker, T. Archer, *Visual C++ 6 Biblija*, Mikro knjiga (1999).

[39] W. Stallings, *Operating Systems, Fourth Edition*, Prentice Hall (2000).

[40] W. Stallings, *Local and Metropolitan Area Networks*, Prentice Hall (2000).

[41] D. A. Solomon, M. E. Russinovich, *Inside Microsoft Windows 2000, Third Edition*, Microsoft Press (2000).

[42] S. Tate, *Windows 2000 Essential Reference*, New Riders (2000).

[43] *Microsoft Windows 2000 Server Resource Kit*, Microsoft Press (2000).

[44] M. Bar, *Linux Internals*, McGraw-Hill (2000).

[45] A. Danesh, *Red Hat Linux*, Mikro knjiga (2000).

[46] A. S. Tannenbaum, *Modern Operating Systems*, Prentice Hall (2001).

[47] S. McClure, J. Scambray, G. Kurtz, *Sigurnost na mreži*, Kompjuter biblioteka (2001)

[48] D. P. Bovet, M. Cesati, *Understanding the Linux Kernel*, O'Reily and Associates (2001).

[49] G. Mourani, *Securing and Optimizing Linux: The Ultimate Solution*, Open Network Architecture Inc. (2001).

[50] W. Stanfield, R. D. Smith, *Linux System Administration, Second Edition*, Sybex (2001).

[51] R. W. Smith, *Linux+ Study Guide*, Sybex (2001).

[52] J. Mauro, R. McDougall, *Solaris Internals: Core Kernel Architecture*, Prentice Hall (2001).

[53] T. Collings, K. Wall, *Red Hat Linux Networking & System Administration*, Hungry Minds Inc. (2002)

[54] A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts, Sixth Edition (Windows XP Update)*, John Wiley & Sons, Inc (2003).

[55] N. Ferguson, B. Schneier, *Practical Cryptography*, Weily Publishing Inc. (2003).

[56] S. Figgins, E. Siever, A. Weber, *Linux in a Nutshell, 4th Edition*, O'Reilly & Associates, Inc. (2003).

[57] S. Obradović, *Osnovi računarske tehnike i programiranja, četvrto izdanje*, Viša elektrotehnička škola (2003).

[58] B. Đorđević, D. Pleskonjić, N. Maček, *Operativni sistemi: UNIX i Linux*, Viša elektrotehnička škola (2004).