Univerzitet u Beogradu
Matematički fakultet

Miroslav Marić

OPERATIVNI SISTEMI

Četvrto izdanje

Beograd, 2023.

dr Miroslav Marić

OPERATIVNI SISTEMI

Izdavač:	Univerzitet u Beogradu – Matematički fakultet Studentski trg 16, Beograd
Za izdavača:	dr Zoran Rakić, dekan
Izdavački odbor:	dr Zoran Petrović, predsednik dr Mirjana Đorić, dr Predrag Janičić, dr Miloš Arsenović, dr Bojan Arbutina
Recenzenti:	dr Dušan Tošić, dr Vladimir Filipović, dr Boris Delibašić
Lektura:	Ivana Kovačević
Obrada teksta:	Autor
Dizajn korica:	Božidar Radivojević
Štampa:	Donat Graf, Beograd
Tiraž:	300 primeraka

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.45(075.8)

МАРИЋ, Мирослав, 1978-
Operativni sistemi / Miroslav Marić. - 4. izd. - Beograd : Univerzitet, Matematički fakultet, 2023 (Beograd : Donat Graf). - 352 стр. : ilustr. ; 24 cm

Tiraž 300. - Bibliografija: str. [353].

ISBN 978-86-7589-174-1

a) Оперативни системи
COBISS.SR-ID 85485065

ISBN 978-86-7589-174-1

© 2023. Autor i Univerzitet u Beogradu – Matematički fakultet
Sva prava su zadržana. Nijedan deo ove publikacije ne može biti reprodukovan niti smešten u sistem za pretraživanje ili transmitovanje u bilo kom obliku, elektronski, mehanički, fotokopiranjem, snimanjem ili na drugi način, bez prethodne pismene dozvole autora.

SADRŽAJ

SADRŽAJ 3

PREDGOVOR 9

17

1. UVOD U OPERATIVNE SISTEME 13

1.1. Operativni sistemi 13

1.2. Osnovni koncepti operativnih sistema 15
1.2.1. Funkcije jezgra operativnog sistema 16
1.2.2. Sistemski pozivi 17
1.2.3. Korisničko okruženje 20
1.2.4. Drajveri 22

1.3. Arhitekture operativnih sistema 23
1.3.1. Monolitni sistemi 23
1.3.2. Slojeviti sistemi 25
1.3.3. Sistemi zasnovani na mikrojezgru 26
1.3.4. Hibridni sistemi 27
1.3.5. Sistemi zasnovani na egzojezgru 28

[redacted]

45

2. UPRAVLJANJE PROCESIMA 55

2.1. Procesi 55
2.1.1. Proces u memoriji 57

2.2. Stanja procesa 58

2.3. Kontrolni blok procesa ... 61

2.4. Niti ... 64

2.5. Redovi procesa ... 70

2.6. Planeri ... 73

2.7. Višeprocesorski sistemi ... 74

2.8. Raspoređivanje procesa ... 80

2.9. Algoritmi planiranja ... 82
2.9.1. FCFS algoritam ... 83
2.9.2. SPF algoritam ... 85
2.9.3. Algoritam sa prioritetima ... 88
2.9.4. Kružni algoritam ... 90
2.9.5. Redovi u više nivoa ... 93
2.9.6. Redovi u više nivoa sa povratnom vezom ... 95

2.10. Upravljanje procesima u sistemima za rad u realnom vremenu ... 96
2.10.1. Raspoređivanje procesa u sistemima za rad u realnom vremenu ... 98

45

3. KONKURENTNOST I SINHRONIZACIJA PROCESA ... 105

3.1. Kritična sekcija ... 109

3.2. Rešenja zasnovana na aktivnom čekanju ... 111
3.2.1. Striktna alternacija ... 113
3.2.2. Dekerov algoritam ... 113
3.2.3. Pitersonov algoritam ... 116
3.2.4. Lamportov (pekarski) algoritam ... 116
3.2.5. Hardverska rešenja ... 118

3.3. Rešenja za zaštitu kritične sekcije bez aktivnog čekanja ... 122
3.3.1. Semafori ... 122
3.3.2. Upotreba semafora ... 128
3.3.2.1. Problem proizvođača i potrošača ... 128
3.3.2.2. Problem filozofa koji večeraju ... 130
3.3.2.3. Problem čitača i pisača ... 133
3.3.3. Kritični regioni ... 139

3.3.4. Monitori 143
3.3.4.1. Rešenje problema filozofa koji večeraju uz pomoć monitora 149

24

4. ZAGLAVLJIVANJE 155

4.1. Mere za sprečavanje zaglavljivanja 161
4.1.1. Prevencija čekanja i držanja 161
4.1.2. Eliminisanje nemogućnosti prekidanja 162
4.1.3. Prevencija kružnog čekanja 162

4.2. Mere izbegavanja 164
4.2.1. Bankarev algoritam 164

4.3. Detekcija zaglavljivanja 173
4.3.1. Oporavak od zaglavljivanja 178

30

5. UPRAVLJANJE MEMORIJOM 185

5.1. Upravljanje memorijom pri monoprogramiranju 191

5.2. Upravljanje memorijom pri multiprogramiranju 194
5.2.1. Prebacivanje 195
5.2.2. Particije 197
5.2.3. Statičke particije 198
5.2.4. Dinamičke particije 200
5.2.5. Fragmentacija 202
5.2.6. Algoritmi za dodelu memorije 205

5.3. Straničenje 207
5.3.1. Hardverska podrška – tabela stranica 211
5.3.2. Asocijativna memorija 211

5.4. Segmentacija 213
5.4.1. Tabela segmenata 214
5.4.2. Karakteristike segmentacije 215

16

6. VIRTUELNA MEMORIJA 221

6.1. Prekrivači 222

6.2. Dinamičko punjenje ... 223

6.3. Straničenje na zahtev ... 224
6.3.1. Zamena stranice ... 228

6.4. Algoritmi za izbacivanje stranice ... 230
6.4.1. Algoritam zasnovan na slučajnom izbacivanju ... 230
6.4.2. Beladijev optimalni algoritam ... 231
6.4.3. FIFO algoritam ... 232
6.4.4. Algoritam druge šanse ... 233
6.4.5. Algoritam sata ... 234
6.4.6. LRU algoritam ... 235
6.4.7. NRU algoritam ... 236
6.4.8. Modifikacija NRU algoritma ... 236
6.4.9. LFU i MFU algoritmi ... 237

30

7. FAJL SISTEM ... 241

7.1. Interfejs fajl sistema ... 241
7.1.1. Atributi fajlova ... 242
7.1.2. Operacije nad fajlovima ... 244
7.1.3. Struktura fajlova ... 245
7.1.4. Tipovi fajlova ... 246

7.2. Direktorijumi ... 250
7.2.1. Organizacija direktorijuma – jedan nivo ... 250
7.2.2. Organizacija direktorijuma – dva nivoa ... 251
7.2.3. Organizacija direktorijuma u strukturu stabla ... 251
7.2.4. Apsolutne i relativne putanje ... 252
7.2.5. Deljenje fajlova, aciklični graf ... 254

7.3. Prava pristupa ... 255

7.4. Struktura i implementacija fajl sistema ... 258
7.4.1. Implementacija fajlova ... 259
7.4.1.1. Neprekidna alokacija ... 259
7.4.1.2. Povezane liste ... 260
7.4.1.3. Indeksirana alokacija ... 262
7.4.2. Implementacija direktorijuma ... 264
7.4.2.1. Lista ... 265
7.4.2.2. Heš tabela ... 265

7.4.3. Upravljanje slobodnim prostorom 267
7.4.4. Performanse 268
7.4.5. Virtuelni fajl sistem 268
7.4.6. Sigurnost 269

25

8. UPRAVLJANJE ULAZNO-IZLAZNIM UREĐAJIMA 273

8.1. Hardverske komponente 274

8.2. Interfejs uređaja 276
8.2.1. Tehnika prozivanja 278
8.2.2. Prekidi 280
8.2.3. Direktan memorijski pristup – DMA 284

8.3. Drajveri 286

8.4. Softver za upravljanje koji ne zavisi od uređaja 288
8.4.1. Planiranje ulazno-izlaznih operacija 288
8.4.2. Baferovanje 288
8.4.3. Obrada grešaka 291
8.4.4. Keširanje 291
8.4.5. Spuler 293

8.5. Interfejs ka korisničkim procesima 294

[redacted]

PREDGOVOR

Ova knjiga predstavlja udžbenik za predmet Operativni sistemi koji se drži na drugoj godini studijskog programa Informatika i na četvrtoj godini studijskog programa Matematika, modul Računarstvo i informatika na Matematičkom fakultetu, Univerziteta u Beogradu. Knjiga može biti korisna i drugim čitaocima koji žele da se upoznaju sa osnovnim principima funkcionisanja operativnih sistema.

Operativni sistemi su sada već tradicionalna oblast u odnosu na veoma brz razvoj računarskih sistema. Komponente računarskih sistema se veoma brzo razvijaju, ali su principi njihovog funkcionisanja slični, tako da se u teoriji operativnih sistema pojavljuju isti principi, problemi, ali i rešenja. U ovoj knjizi pažnja je posvećena osnovnim principima funkcionisanja operativnih sistema bez ulaženja u detalje implementacije, jer su one veoma podložne promenama, dok su principi suština koja ne zastareva. Za pisanje ovog udžbenika korišćene su neke od najpopularnijih knjiga o operativnim sistemima: [1], [2], [3] i [4].

U uvodnoj glavi prikazana je uloga operativnih sistema u okviru računarskih sistema i definisani osnovni pojmovi vezani za njih. Opisana je uloga jezgra operativnog sistema, sistemskih poziva i drajvera. Zatim su opisane različite arhitekture operativnih sistema, kratak istorijski prikaz razvoja operativnih sistema i bitnih koncepata, a na kraju su pomenuti neki od operativnih sistema koji su ostavili dubok pečat u istoriji računarstva.

Druga glava je posvećena procesima. U njoj je prvo definisan pojam procesa, zatim način na koji operativni sistemi vode evidenciju o procesima, definisane su niti i objašnjen način izvršavanja procesa i niti na višeprocesorskim sistemima. Objašnjena je funkcija planera i prikazani tradicionalni algoritmi za raspoređivanje procesa. Poslednji deo ove glave posvećen je upravljanju procesima u sistemima za rad u realnom vremenu.

U trećoj glavi razmatra se problem konkurentnosti i sinhronizacije procesa. Definiše se kritična sekcija, a zatim se prikazuju metode za njenu zaštitu, uz osvrt na sinhronizaciju procesa. Prvo se obrađuju metode za zaštitu kritične sekcije zasnovane na aktivnom čekanju – softverske i hardverske, a zatim i sofisticiranije metode bez aktivnog čekanja.

Četvrta glava posvećena je problemu zaglavljivanja. U njoj se prvo definišu neophodni uslovi da do zaglavljivanja dođe, a onda se analiziraju metode za sprečavanje, izbegavanje i detekciju zaglavljivanja. Na kraju ove glave prikazane su metode za oporavak sistema posle zaglavljivanja.

Peta i šesta glava posvećene su upravljanju memorijom. U petoj glavi se analiziraju karakteristike različitih vrsta memorije. Zatim se objašnjavaju načini upravljanja memorijom pri monoprogramiranju i pri multiprogramiranju. Prikazani su koncepti statičkih i dinamičkih particija, straničenje i segmentacija. Šesta glava posvećena je konceptu virtuelne memorije. Prikazani su pristupi poput: prekrivača, dinamičkog punjenja i straničenja na zahtev. Posebna pažnja posvećena je algoritmima za izbacivanje stranice.

U sedmoj glavi objašnjava se pojam fajl sistema. Prikazani su atributi, tipovi, struktura fajlova i direktorijuma kao i operacije nad njima. Analizirana je implementacija fajl sistema, fajlova i direktorijuma.

Osma glava je posvećena upravljanju ulazno-izlaznim uređajima. Opisani su ulazno-izlazni uređaji, zatim interfejs kojim se oni vezuju za računarski sistem, rutine za obradu prekida, drajveri, softver koji ne zavisi od uređaja i na kraju interfejs ka korisničkim procesima uz pomoć kojeg se obezbeđuju usluge.

U devetoj glavi obrađene su računarske mreže. Prikazana je klasifikacija računarskih mreža i kratka istorija. Zatim je opisana slojevita arhitektura mreže, ISO OSI i TCP/IP referentni modeli. Objašnjeni su UDP i TCP protokoli. Poslednji deo ove glave posvećen je bezbednosti računarskih mreža.

Na kraju svakog poglavlja navedena su pitanja koja bi mogla da služe za proveru znanja.

Koristim priliku da se zahvalim svima koji su čitali ovaj rukopis i svojim komentarima doprineli da on bude bolji. Posebnu zahvalnost dugujem dr Dušanu Tošiću koji je od samog početka strpljivo i pažljivo nadgledao rad na rukopisu. Na brojnim korisnim sugestijama želeo bih da se zahvalim recenzentima dr Vladimiru Filipoviću i dr Borisu Delibašiću. Zahvalnost dugujem Aleksandru Đeniću na tehničkoj i stručnoj pomoći, kao i Dušanu Džamiću na savetima i pomoći oko ilustracija. Ivani Kovačević se zahvaljujem na pomoći oko jezičkih konstrukcija i lekturi. Na sugestijama i komentarima koji su doprineli kvalitetu rukopisa zahvalan sam mr Đuri Mišljenoviću, Marku Šošiću, Milošu Stankoviću i Vladimiru Kuzmanoviću. Zahvaljujem se i mojim studentima na svim pitanjima, komentarima, primedbama i diskusijama koje su mi pomogle u pisanju ove knjige.

U četvrtom izdanju knjige ažurirani su podaci tako da odgovaraju promenama koje su se dogodile posle objavljivanja prvog, drugog i trećeg izdanja i ispravljene su uočene greške.

Beograd, januar 2023.

Miroslav Marić

1. UVOD U OPERATIVNE SISTEME

1.1. OPERATIVNI SISTEMI

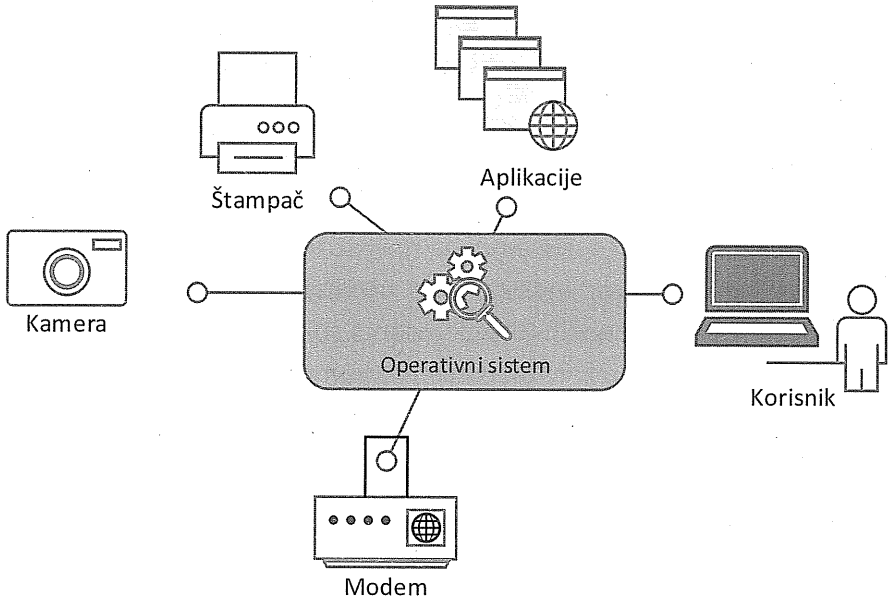
Operativni sistem je jedna od najvažnijih komponenti savremenog računarskog sistema. Računarski sistem se sastoji od tehničkog i programskog sistema računara, odnosno od hardvera i softvera. On funkcioniše po principima koje je postavio Džon fon Nojman (*John von Neumann*) sredinom dvadesetog veka.

Tehnički sistem računara – hardver čine uređaji računarskog sistema, pre svih procesor i radna (primarna, glavna) memorija koji predstavljaju srce tehničkog sistema računara, a zatim i ulazno-izlazni uređaji itd. Komponente računarskog sistema su magistralama povezane u celinu. Iako se hardver i softver veoma brzo razvijaju, arhitektura računara, organizacija procesora i način funkcionisanja su približno isti iz generacije u generaciju. Razlike su u karakteristikama komponenti, tj. veličini, brzini, kapacitetu itd. ali i ceni koja je višestruko niža u odnosu na prve računare.

Programski sistem računara čine aplikativni i sistemski softver. *Aplikativni softver* je najviši sloj u hijerarhiji i čine ga programi koji služe za obavljanje specifičnih zadataka na računaru (programi za obradu teksta, programi za reprodukciju multimedijalnih sadržaja, programi za matematička izračunavanja itd.). *Sistemski softver* predstavlja skup programa koji bi trebalo da pruže što udobniji interfejs (okruženje) za korišćenje hardvera i da omoguće izvršavanje aplikativnih programa. Praktično, sistemski softver predstavlja sponu između hardvera i aplikativnog softvera, tj. korisnika.

Operativni sistem je deo sistemskog softvera koji je odgovoran za upravljanje računarskim resursima koji treba da obezbedi što bolje uslove za korišćenje računara (slika 1.1.). Operativni sistem predstavlja sloj softvera koji je naslonjen na hardver i trebalo bi da ga učini upotrebljivim, da pri tome prikrije razlike između različitih hardverskih komponenti istog

tipa i da korisniku omogući što veću udobnost za rad. To je program koji se sve vreme izvršava, prati i nadgleda funkcionisanje sistema. Suština operativnih sistema je da obezbede okruženje u kojem će korisnici imati mogućnost da što jednostavnije pokreću i izvršavaju programe, a da se hardver koristi što efikasnije. Pri tome, operativni sistem bi trebalo da zaštiti hardver od direktnog pristupa korisnika, odnosno korisničkih programa.

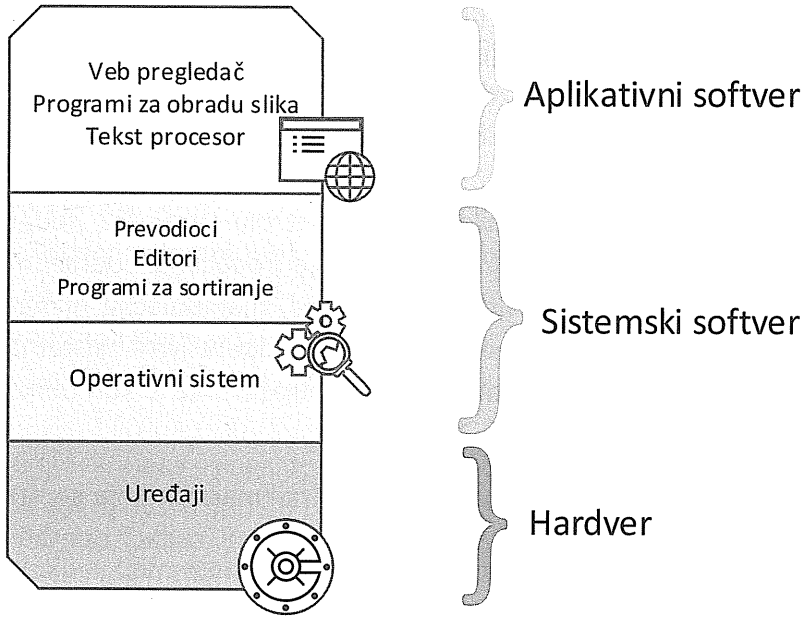


Slika 1.1. Operativni sistem

Sistemski softver, osim operativnih sistema najčešće sadrži prevodioce, editore, programe za sortiranje, alate itd., odnosno programe koji nisu esencijalno bitni za funkcionisanje sistema, ali se bez njih teško može pružiti dodatna udobnost korisnicima, a pri tome nisu deo aplikativnog softvera. Na slici 1.2. prikazan je odnos komponenti računarskog sistema.

Operativni sistemi spadaju u najkomplikovanije i najveće programe, koji se obično programiraju duži vremenski period uz angažovanje većeg broja programera, ali se često ne doživljavaju na takav način. Naime, korisnici

zbog lakoće korišćenja, ali i ne previše visoke cene ovakve vrste softvera (neki su i besplatni), često ne uočavaju njihovu kompleksnost. Operativni sistemi se doživljavaju kao nešto što se podrazumeva uz hardver i na šta kupovinom računara korisnici imaju pravo. Vremenski period između isporuke dva operativna sistema, od strane istog proizvođača, obično nije mali. Proizvođači operativne sisteme često ne programiraju od početka, već se oni nadograđuju na temelje postavljene u nekoj od prethodnih verzija.



Slika 1.2. Računarski sistem

1.2. Osnovni koncepti operativnih sistema

Računarstvo se razvija velikom brzinom, ali su osnovni principi funkcionisanja zajednički za sve generacije računarskih sistema. Savremeni računari se sastoje od jednog ili više procesora, memorije, većeg broja ulazno-izlaznih uređaja, komunikacione opreme itd. Međutim, da bi ovaj hardver imao upotrebnu vrednost potreban je

softver koji će ga učiniti upotrebljivim za korisnike različitih profila. Osnovni zadaci operativnih sistema su da omoguće što efikasniju realizaciju sledećih aktivnosti:

- Upravljanje procesima;
- Upravljanje memorijom;
- Upravljanje ulazno-izlaznim uređajima;
- Upravljanje podacima;
- Upravljanje mrežama.

Upravljanje procesima, odnosno programima u izvršavanju, podrazumeva njihovo kreiranje, izvršavanje, dodeljivanje resursa procesima, sinhronizaciju, donošenje odluka o tome koji proces će imati priliku da se izvršava na procesoru u određenom trenutku itd. *Upravljanje memorijom* se odnosi na raspoređivanje procesa u okviru radne memorije, a *upravljanje ulazno-izlaznim uređajima* na kontrolu i transfer podataka između uređaja i ostatka sistema. *Upravljanje podacima* podrazumeva čuvanje podataka, vođenje evidencije o njima, manipulaciju njima itd. Poslednja funkcija i danas neizostavni deo operativnih sistema je *upravljanje mrežama* koje podrazumeva umrežavanje i komunikaciju između računara.

Operativni sistemi predstavljaju staru oblast računarstva koja se razvija uporedo sa napretkom računara. Kada su osnovni zadaci operativnih sistema u pitanju, najveća razlika u odnosu na prve operativne sisteme je u upravljanju mrežama. Naime, računarske mreže nisu postojale u današnjem obliku u trenutku začetka prvih generacija operativnih sistema tako da rad sa njima nije bio jedan od prioriteta.

1.2.1. Funkcije jezgra operativnog sistema

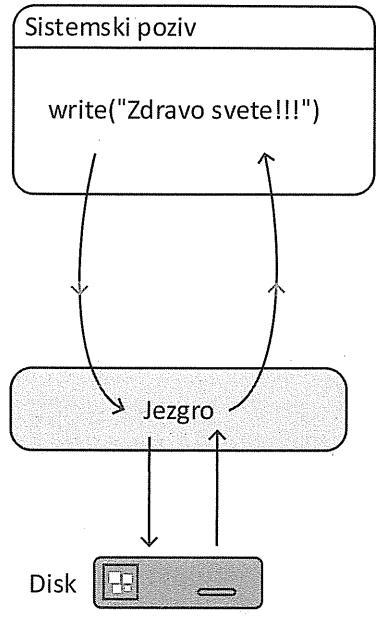
Jezgro (*kernel*) je deo operativnog sistema u koji su smeštene najvažnije funkcije koje obezbeđuju osnovne servise operativnog sistema. Ono je odgovorno za funkcionisanje sistema i ima zadatak da upravlja hardverskim i softverskim resursima na najnižem nivou. To je deo softvera koji se prvi učita u radnu memoriju, pri pokretanju računarskog

sistema, i ostaje u njoj do završetka rada odnosno isključivanja sistema. Jezgro se obično nalazi u posebnom delu radne memorije i stalno je aktivno. To je srce operativnog sistema i najniži je sloj u hijerarhiji računarskog sistema koji nije hardverski (ili apstrakcija hardvera). U njemu su definisana pravila i dozvole kojima se reguliše funkcionisanje celokupnog sistema. U sistemu se, osim jezgra, mogu izvršavati aplikativni i sistemski programi. Sistemski programi ne moraju biti deo jezgra operativnog sistema.

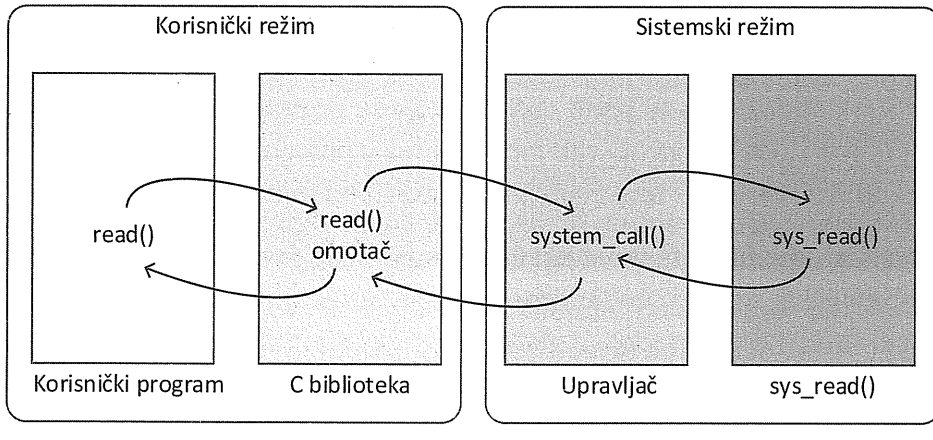
1.2.2. Sistemski pozivi

Usluge koje operativni sistem može da pruži aplikativnim programima ostvaruju se uz pomoć *sistemskih poziva.* Programi uz pomoć sistemskih poziva komuniciraju sa jezgrom i pomoću njega dobijaju mogućnost da izvrše osetljive operacije u sistemu. Praktično, sistemski pozivi su skup funkcija koji predstavlja interfejs ka operativnom sistemu. Na primer, aplikativni program može pristupiti hardveru (hard disku, štampaču itd.) ili uraditi neku operaciju sa procesima jedino korišćenjem odgovarajućeg sistemskog poziva (slika 1.3.). Sistemski pozivi su implementirani tako da dozvole samo operacije koje ne mogu biti štetne po računarski sistem. Sistemskim pozivom se jasno definiše koje su dozvoljene operacije kada je odgovarajuća usluga operativnog sistema u pitanju. Na ovakav način se pristup hardveru štiti od potencijalno štetnih operacija korisnika.

Procesori savremenih računarskih sistema imaju mogućnost rada u bar dva različita režima rada: *korisničkom* (*user mode*) i *sistemskom* (*supervisor, kernel mode*). U sistemskom režimu moguće je izvršiti sve instrukcije, dok je broj instrukcija koje je dozvoljeno izvršiti u korisničkom režimu redukovan. Naime, instrukcije za osetljive operacije poput pristupa ulazno-izlaznim uređajima, zaštićenim delovima memorije itd. moguće je izvršiti samo u sistemskom režimu rada procesora. Na slici 1.4. prikazan je primer koji ilustruje poziv funkcije za čitanje (*read*) iz C biblioteke koja se sistemskim pozivom (*sys_read*) iz sistemskog moda obraća hardveru.



Slika 1.3. Sistemski poziv



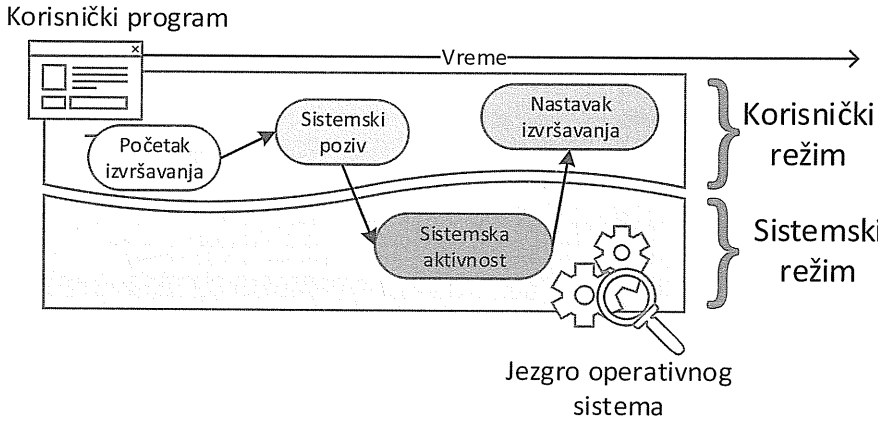
Slika 1.4. Promena režima rada

Aplikativni programi se veći deo vremena izvršavaju u korisničkom režimu, dok je sistemski režim predviđen za posebno osetljive operacije koje izvodi operativni sistem. Pri korišćenju sistemskog poziva prelazi se iz korisničkog u sistemski režim i dalju kontrolu preuzima operativni

sistem. Ključni deo operativnog sistema koji reaguje u ovakvim situacijama je jezgro.

Sistemski pozivi koriste jezgro da bi omogućili različite servise operativnog sistema. Svi programi, često uključujući i sistemske, funkcionišu na nivou iznad jezgra u korisničkom režimu rada. Sistemske aktivnosti, koje se pokreću sistemskim pozivima poput pristupa hardveru, obavljaju se na nivou jezgra, odnosno u sistemskom režimu rada. Zbog osetljivosti poslova kojima se jezgro bavi, ono se obično učitava u poseban, zaštićeni deo memorije i time čuva od neželjenih promena.

Pri dizajniranju operativnih sistema često se teži da se više aktivnosti odvija u korisničkom režimu umesto u sistemskom jer se na taj način povećava stabilnost sistema. U tom slučaju, eventualne greške pri izvršavanju programa ili problemi uglavnom ne mogu da ugroze funkcionisanje sistema.



Slika 1.5. Korisnički i sistemski režim rada

Kada aplikativni programi izvrše sistemski poziv, parametri sistemskog poziva se postave na predviđene lokacije u memoriji. Zatim, menja se režim rada u sistemski u kojem su, za razliku od korisničkog režima, dozvoljene sve operacije koje procesor može da uradi (slika 1.5.). Tada jezgro preuzima kontrolu i na osnovu parametara sistemskog poziva izvršava željenu operaciju. Po završetku operacije režim rada se ponovo

prebacuje u korisnički, a rezultati se vraćaju programu koji je izvršio sistemski poziv.

Neki procesori podržavaju više različitih režima rada koji se mogu iskoristiti pri dizajniranju operativnih sistema. U situacijama kada procesori ne podržavaju dovoljno režima, oni se mogu realizovati na nivou operativnih sistema.

1.2.3. Korisničko okruženje

Mnogi korisnici operativni sistem doživljavaju kroz *korisničko okruženje*. Ono ima zadatak da olakša korišćenje ostalih delova operativnog sistema, a i celokupnog računarskog sistema. Razvoj korisničkih okruženja pratio je razvoj računara jer su poboljšanja karakteristika računarskih sistema kao i povećavanja njihove kompleksnosti obično povlačila i zahteve za olakšavanje njihovog korišćenja.

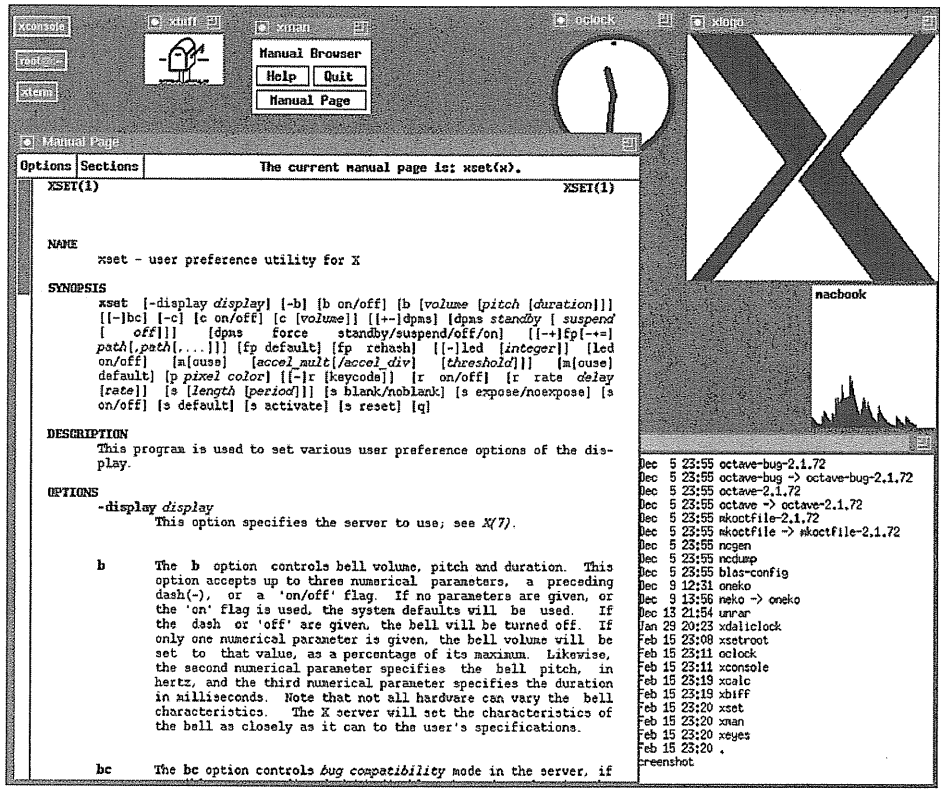
Korisnička okruženja se mogu podeliti na tekstualna i grafička. Međutim, pošto se uz pomoć karaktera mogu improvizovati grafički elementi u okviru ekrana, ispravnija podela bi bila na linijska i ekranska.

```
geogebra@alas:~$ top
top - 00:47:16 up 52 days,  7:48,  3 users,  load average: 0.00, 0.01, 0.05
Tasks: 174 total,   1 running, 170 sleeping,   0 stopped,   3 zombie
Cpu(s):  1.6%us,  0.2%sy,  0.0%ni, 98.1%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  32895892k total,  7698136k used, 25197756k free,  2254120k buffers
Swap: 16000736k total,   530444k used, 15470292k free,  4063052k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2068 apache    20   0 2047m 106m 4836 S    7  0.3 109:13.35 httpd
 2066 apache    20   0 2047m 107m 4864 S    3  0.3  32:31.14 httpd
21397 root      20   0 20280 1736 1316 S    2  0.0   0:00.08 auth
    9 root      20   0     0    0    0 S    0  0.0  16:58.24 rcu_sched
10912 dovecot   20   0  9456  576  268 S    0  0.0   1:10.32 anvil
21165 mr13049   20   0 20232 5872 3692 S    0  0.0   0:00.37 nano
21267 dovecot   20   0 11844 1356 1008 S    0  0.0   0:00.03 auth
21395 root      20   0 18756 2240 1172 S    0  0.0   0:00.02 config
    1 root      20   0  4352   28    0 S    0  0.0   0:56.40 init
    2 root      20   0     0    0    0 S    0  0.0   0:01.12 kthreadd
    3 root      20   0     0    0    0 S    0  0.0  19:17.77 ksoftirqd/0
    5 root       0 -20     0    0    0 S    0  0.0   0:00.00 kworker/0:0H
    7 root      RT   0     0    0    0 S    0  0.0   1:23.66 migration/0
    8 root      20   0     0    0    0 S    0  0.0   0:00.00 rcu_bh
   10 root      RT   0     0    0    0 S    0  0.0   0:50.09 migration/1
   11 root      20   0     0    0    0 S    0  0.0   5:41.71 ksoftirqd/1
   12 root      20   0     0    0    0 S    0  0.0   0:00.00 kworker/1:0
geogebra@alas:~$
```

Slika 1.6. Linijsko korisničko okruženje

Pod *linijskim korisničkim okruženjima* podrazumevaju se konzole, terminali, komandne linije itd. koje omogućavaju da se operativnim sistemom upravlja kucanjem tekstualnih komandi – linija teksta. Ovakva okruženja su se među prvima pojavila u računarstvu.

Komandni interpreter je najvažniji deo linijskog korisničkog okruženja i njegova uloga je da naredbe i podatke koje korisnik unese u tekstualnom obliku prepozna i naloži operativnom sistemu izvršavanje odgovarajućih operacija (slika 1.6.).



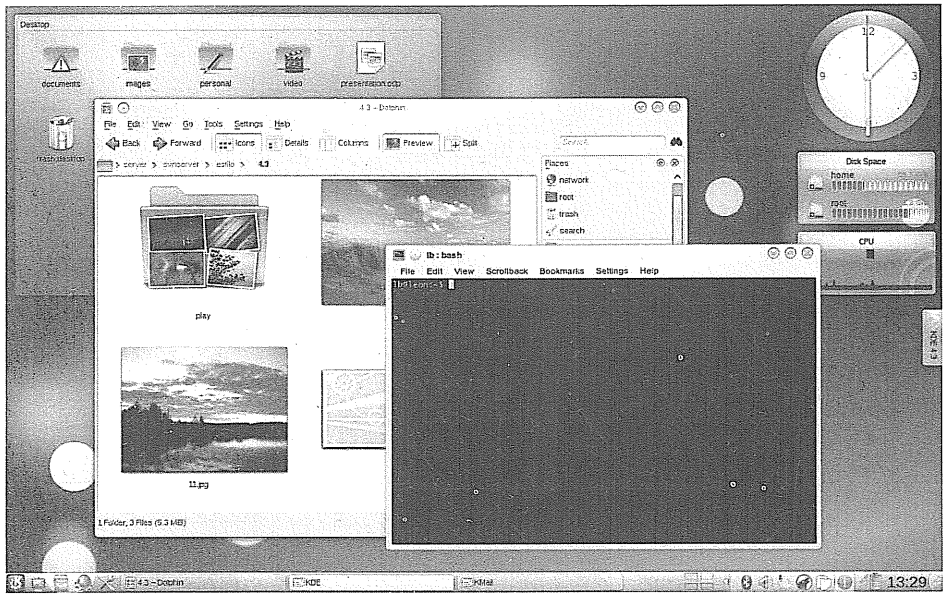
Slika 1.7. Ekransko korisničko okruženje

Za razliku od linijskih, *ekranska korisnička okruženja* pružaju mogućnost da se operativnim sistemom upravlja korišćenjem cele površine ekrana (slika 1.7.). Osim uz pomoć tastature, komande i manipulacije sa podacima se mogu izvoditi uz pomoć miša i sličnih ulaznih uređaja. U poslednje vreme veoma su popularni korisnički interfejsi koji

omogućavaju da se komande zadaju dodirima prstiju po ekranima osetljivim na dodir.

Radna površina je osnovni deo ekranskog korisničkog okruženja na kome su aplikacije i podaci predstavljeni vizuelnim elementima. Ovi elementi na različite načine reaguju na komande koje se zadaju uz pomoć miša, tastature itd.

Savremeni operativni sistemi obično korisnicima pružaju mogućnost da koriste obe vrste okruženja (slika 1.8.).

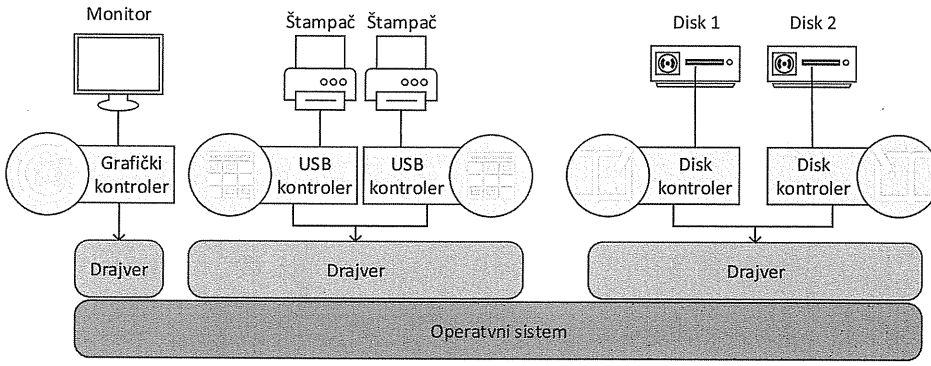


Slika 1.8. Ekransko i linijsko korisničko okruženje u istom sistemu

1.2.4. Drajveri

Ulazno-izlazni uređaji imaju svoje kontrolere koji se koriste pri komunikaciji sa ostalim delovima sistema. Sa druge (softverske) strane nalaze se *drajveri* (*drivers* – upravljački programi) koji se nadograđuju na kontrolere i omogućavaju komunikaciju, odnosno upravljanje ulazno-izlaznim uređajima. Oni se programiraju tako da za različite tipove iste vrste uređaja (štampača, hard diskova, tastatura itd.) definišu jedinstven skup dozvoljenih instrukcija. Različiti uređaji mogu zahtevati posebne

naredbe ili parametre za izvršavanje, a zadatak drajvera je da apstrahuju ove razlike i naprave uniformni interfejs. Na taj način se rad sa ulazno-izlaznim uređajima dosta olakšava jer se izbegava pisanje posebnih programa za svaki tip uređaja, već se jasno definisanim skupom funkcija omogućava pisanje univerzalnih programa za uređaje iste klase. Na slici 1.9. ilustrovana je pozicija drajvera u računarskom sistemu. Drajveri su detaljnije obrađeni u delu 8.3.



Slika 1.9. Drajveri

1.3. ARHITEKTURE OPERATIVNIH SISTEMA

Postoje različiti pristupi pri projektovanju operativnih sistema. Razlike se najčešće ogledaju u skupu funkcija koje će se nalaziti u jezgru i njihovoj organizaciji (strukturi). U odnosu na tip jezgra postoje:

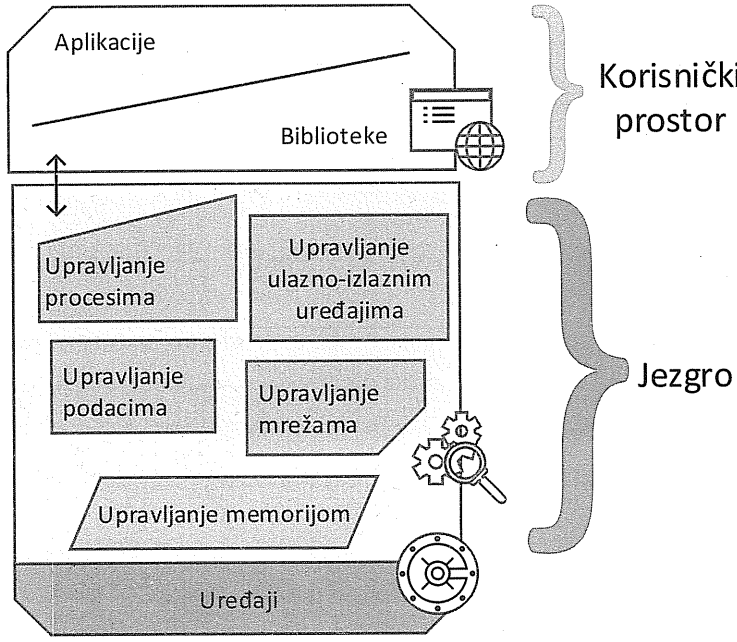
1. Monolitni sistemi;
2. Slojeviti sistemi;
3. Sistemi zasnovani na mikrojezgru;
4. Hibridni sistemi;
5. Sistemi zasnovani na egzojezgru.

1.3.1. Monolitni sistemi

Za arhitekturu operativnog sistema kaže se da je *monolitna,* tj. da ima monolitno jezgro (*monolithic kernel*), ukoliko se u jezgru nalaze svi servisi operativnog sistema zajedno sa drajverima integrisani u jedan program

(slika 1.10.). Pri tome, svi delovi se pokreću u istom trenutku, izvršavaju u sistemskom režimu i u istom delu memorije. Prilikom pokretanja ovakvog operativnog sistema jezgro se u memoriju učitava u celosti kao jedan izvršni program. Funkcije jezgra mogu jedna drugu pozivati bez ograničenja. Iako se čini da su ovakvi sistemi jednostavni, oni mogu biti vrlo složeni.

Veoma velika povezanost funkcija na niskom nivou obično doprinosi brzini i efikasnosti ovakvih jezgara. Monolitno jezgro predstavlja jedinu apstrakciju nad hardverom tako da se aplikativnim programima na raspolaganje stavljaju sistemski pozivi kroz koje su implementirane usluge operativnog sistema.



Slika 1.10. Monolitni sistem

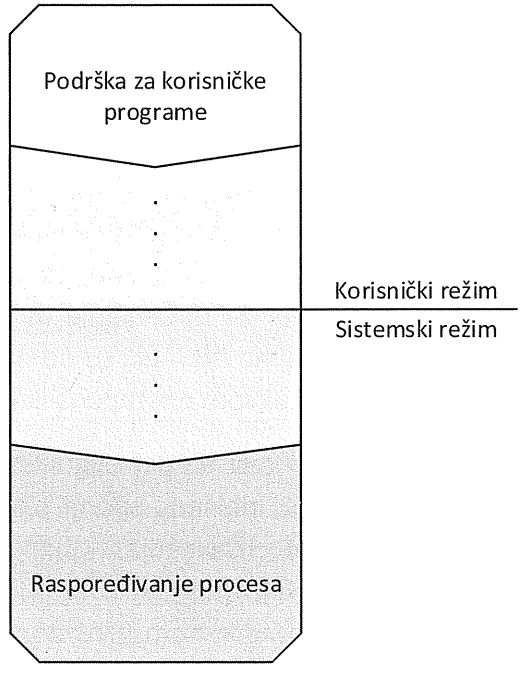
Jedna od najvećih mana monolitnih jezgara je loša otpornost na greške. U slučaju da dođe do greške u nekom od podsistema, dolazi do problema koji mogu uticati na ceo sistem i najčešće je jedino rešenje njegovo

ponovno podizanje. Neki od najpoznatijih sistema koji imaju monolitna jezgra su: MS-DOS, BSD, AIX, Windows 98 i GNU/Linux. Na Slici 1.10. ilustrovan je primer monolitnog operativnog sistema.

1.3.2. Slojeviti sistemi

Slojevita arhitektura podrazumeva da je operativni sistem izgrađen od zasebnih slojeva (celina) koji se nadograđuju jedan na drugi. Pri tome, svaki sloj ima određene funkcije koje su opisane kroz njegov interfejs ka višem sloju. Slojevi se implementiraju tako da mogu da koriste isključivo usluge prvog sloja ispod sebe. Iz tog razloga projektovanje slojeva je veoma zahtevan posao jer se mora voditi računa o raspodeli funkcija.

Najveći problem ovakvih sistema je neefikasnost. Naime, sistemski poziv prolazi kroz više slojeva a pri svakom prolazu se prosleđuju podaci, menjaju parametri itd. što dovodi do usporenja.



Slika 1.11. Slojeviti sistem

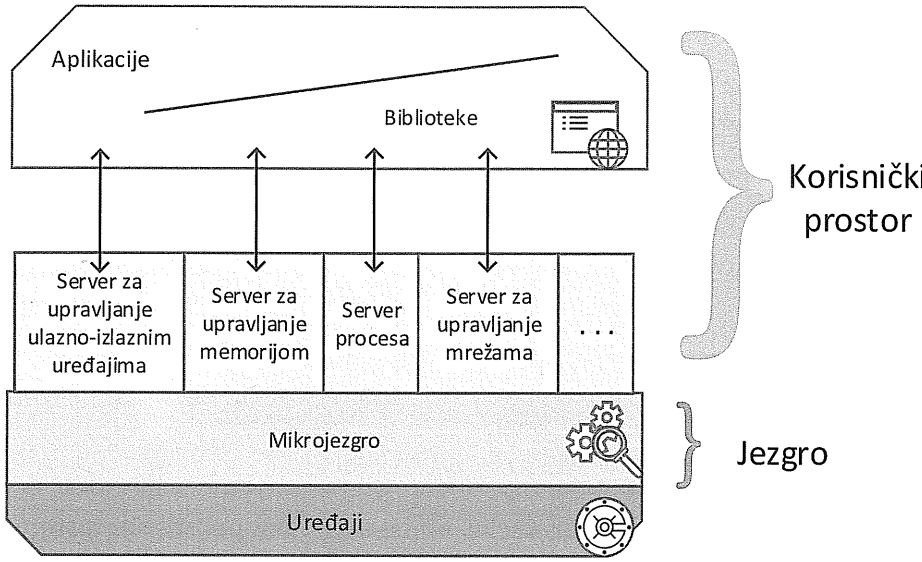
Pri tome, slojevi ne moraju biti u jezgru operativnog sistema već, zavisno od implementacije, određeni slojevi mogu funkcionisati u korisničkom režimu (slika 1.11.). Najniži sloj čini hardver ili njegova apstrakcija, na najvišem sloju je podrška za aplikativne programe a između može biti proizvoljan broj slojeva. U najpoznatije operativne sisteme sa slojevitom arhitekturom spadaju: THE, Multics itd.

1.3.3. Sistemi zasnovani na mikrojezgru

Za razliku od monolitnih sistema, a u većini implementacija i od slojevitih sistema, arhitektura koja se zasniva na *mikrojezgru* (*microkernel*) podrazumeva minimalno jezgro u kojem se nalaze samo najosnovnije funkcije. Deo funkcija koje bi u prethodnim slučajevima bile deo jezgra se izmeštaju u korisnički prostor i to u zasebne prostore u memoriji tako da sa bezbednijeg nivoa pristupaju jezgru. Na ovaj način se rešavaju problemi koji nastaju kada dođe do greške pri primeni neke od ovih funkcija, tako da se to ne odražava na funkcionisanje jezgra. Usluge operativnog sistema koje su izmeštene iz jezgra se njemu obraćaju kako bi ostvarile svoje ciljeve (prvenstveno pristup hardveru), a preko njega i komuniciraju sa drugim uslugama. Usluge koje obavljaju slične zadatke u okviru operativnih sistema se grupišu u procese koji se nazivaju serverski procesi, a često i samo *serveri*. Drajveri se često ne nalaze u mikrojezgru, već pripadaju odgovarajućim serverima. Takođe, server za upravljanje memorijom, server procesa, server za upravljanje mrežama, server za upravljanje ulazno-izlaznim uređajima itd. neki su od najvažnijih servera koji se mogu implementirati pri ovakvom pristupu (slika 1.12.).

Arhitekturu mikrojezgra odlikuje veći stepen sigurnosti u odnosu na monolitne sisteme, ali su oni obično sporiji. Naime, promene memorijskog prostora dovode do kašnjenja i manje propusnosti u poređenju sa sistemima sa monolitnim jezgrom. One su neophodne jer se serveri i jezgro ne nalaze u istom adresnom prostoru i ne izvršavaju u istom režimu. Često prosleđivanje poruka između malog jezgra i servera takođe dovodi do kašnjenja. Imajući u vidu navedene razloge, može se zaključiti da arhitektura zasnovana na mikrojezgru pruža veći stepen

sigurnosti i u odnosu na slojevite sisteme, ali da brzina nije odlika ni jednog ni drugog pristupa.

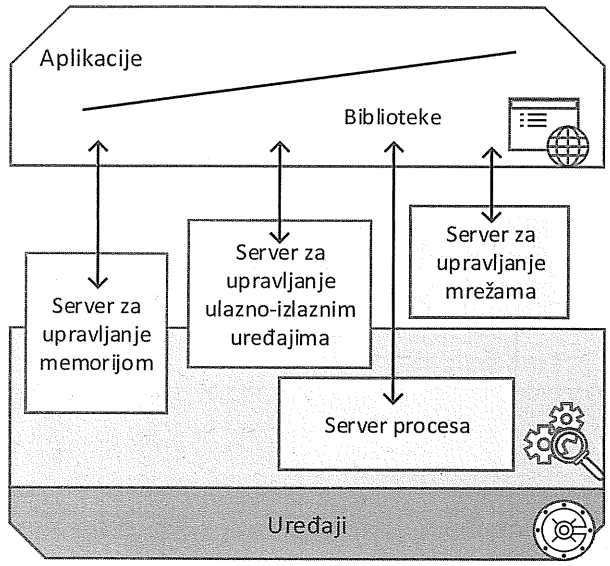


Slika 1.12. Sistem zasnovan na mikrojezgru

Primeri sistema koji su dizajnirani na principima mikrojezgra su: Mach, Minix, QNX i L4.

1.3.4. Hibridni sistemi

Hibridna arhitektura sistema, odnosno *hibridna jezgra* (*hybrid kernel*) predstavljaju kompromis između monolitne i arhitekture koja se zasniva na mikrojezgru (slika 1.13.). Neke veoma bitne, kao i funkcije koje se često izvršavaju, spuštaju se u jezgro kako bi se povećala brzina i efikasnost, ali se dobar deo funkcija zadržava u nivoima iznad jezgra. Kod ovakvog pristupa najvažnije je precizno odrediti koje funkcije treba spustiti u jezgro, a koje ostaviti van, jer će u suprotnom hibridni pristup ličiti na sistem sa monolitnim (u slučaju da se previše funkcija implementira u jezgro) ili mikrojezgrom (ako dosta važnih funkcija ostane van jezgra).



Slika 1.13. Hibridni sistem

Hibridna jezgra se koriste u većini komercijalnih operativnih sistema: Apple Mac OS X, Microsoft Windows NT 3.1, NT 3.5, NT 3.51, NT 4.0, 2000, XP, Vista, 7, 8, 8.1, 10 i sl.

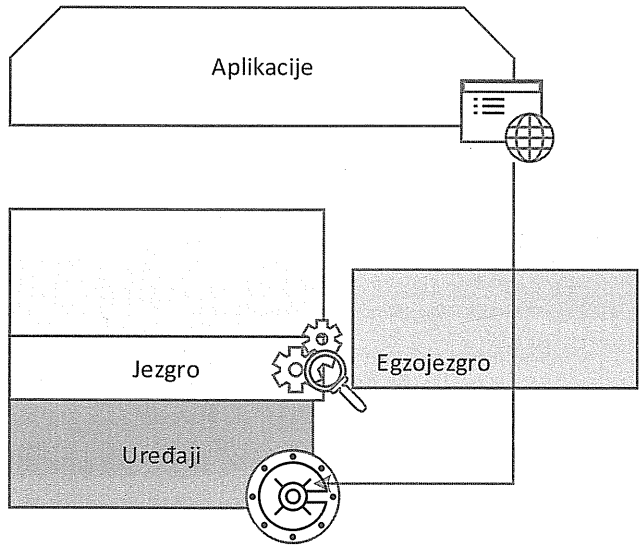
1.3.5. Sistemi zasnovani na egzojezgru

Tradicionalni pristup pri dizajniranju operativnih sistema podrazumeva da hardverske komponente ne budu vidljive korisničkim aplikacijama, odnosno da programi komuniciraju sa hardverom isključivo preko jezgra operativnog sistema. Na ovaj način olakšava se programiranje, ali se, sa druge strane, ograničavaju performanse. Suprotni pristup bi podrazumevao da se dozvoli da aplikacije direktno pristupaju hardveru, tj. da se jezgro potpuno zaobiđe. U tom slučaju bi funkcionisanje sistema zavisilo od aplikacija koje pišu programeri i aplikacije bi zavisile jedne od drugih.

Koncept *egzojezgra* (*exokernel*) predstavlja kompromis između ova dva suprotna pristupa (slika 1.14.). Ideja je da jezgro obezbedi osnovne resurse i da aplikacijama prepusti rad sa njima. Ovo se postiže premeštanjem apstrakcije iznad hardvera u posebne biblioteke koje

obezbeđuju minimalne apstrakcije uređaja. Na taj način se programeri mogu osloniti na odgovarajuću biblioteku ali im se ostavlja i mogućnost da implementiraju svoje biblioteke. Drugim rečima, programeri imaju slobodu u izboru nivoa apstrakcije kada je pristup hardveru u pitanju. Ovakav način rada može da doprinese znatnom ubrzanju i poboljšanju performansi. Sa druge strane, dodatna fleksibilnost za korisničke aplikacije može da dovede do smanjenja konzistentnosti i neurednosti koda.

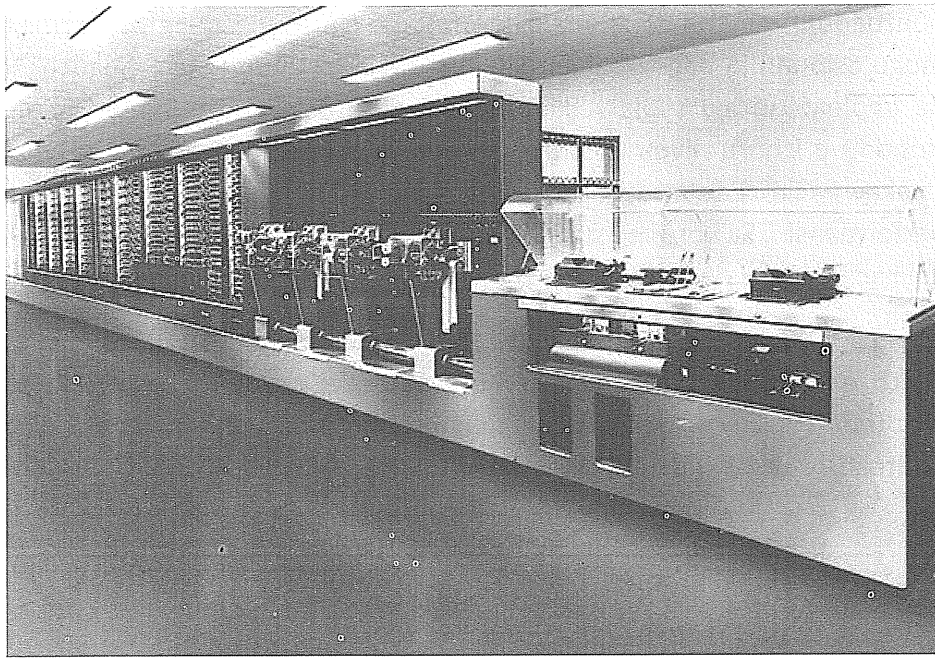
Pošto je veći deo funkcija izmešten iz egzojezgra, ono je obično manje od prethodno pomenutih tipova jezgara. Primeri sistema koji imaju arhitekturu zasnovanu na egzojezgru su: XOK, ExOS itd.



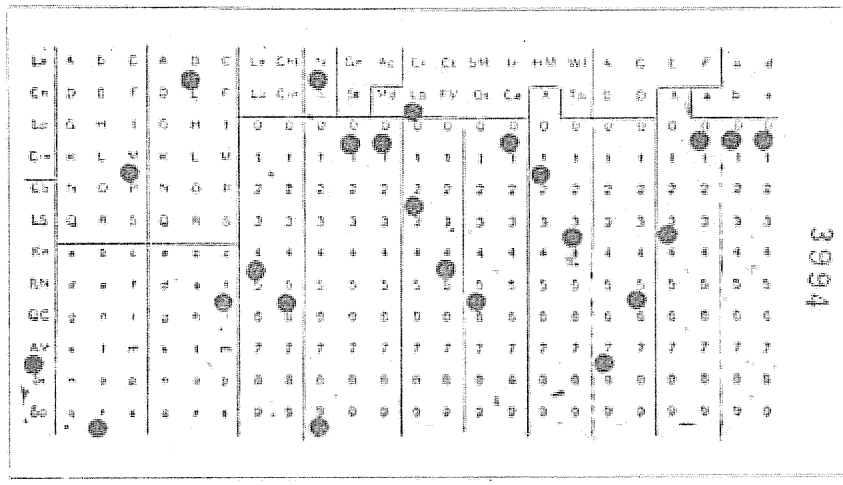
Slika 1.14. Egzojezgro

1.4. Razvoj operativnih sistema i istorijat

Prvi računari koji su se pojavili 40-tih godina dvadesetog veka bili su zasnovani na elektronskim (vakuumskim) cevima. Zbog velikog broja glomaznih elektronskih cevi (bilo ih je i do 20.000) ovi računari su bili ogromnih dimenzija, trošili su puno struje i veoma se grejali (slika 1.15.).



Slika 1.15. Računar Mark 1



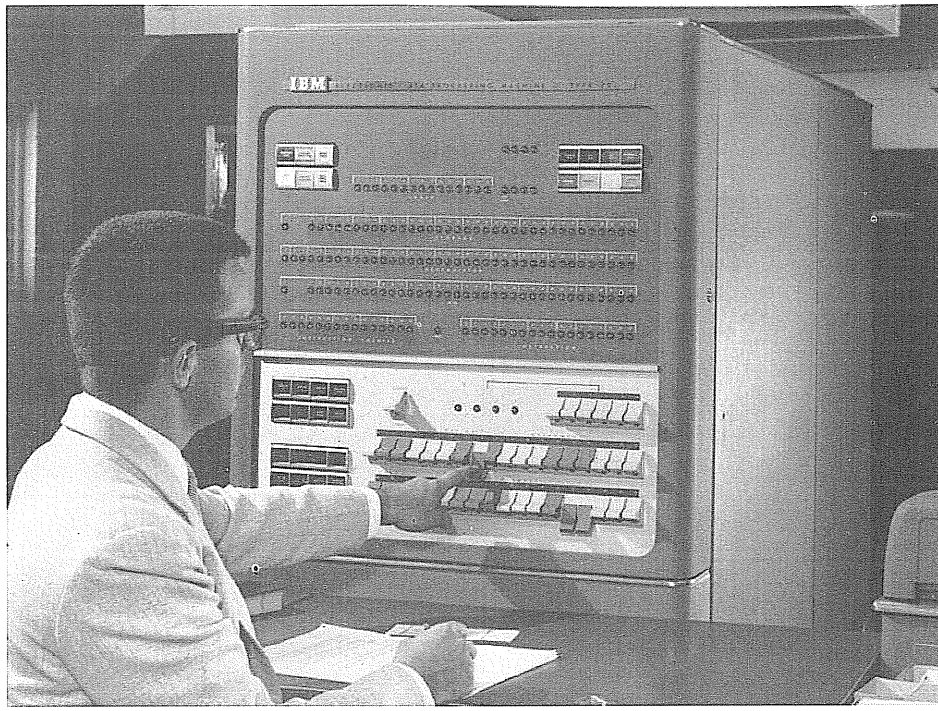
Slika 1.16. Holeritova – bušena kartica

Prvi računari su bili neefikasni i podložni kvarenju. Ulazne tehnologije su se zasnivale na bušenim karticama (slika 1.16.) i magnetnim trakama (slika 1.17.). Bušene kartice je koristio Herman Holerit (*Herman Hollerith*)

za potrebe popisa u Sjedinjenim Američkim Državama. Za programiranje, na prvim računarskim sistemima, koristio se mašinski ili simbolički jezik, a programeri su morali da imaju pripremljen program i podatke pre početka rada na računaru.

Slika 1.17. Jedinica za rad sa magnetnim trakama

Pošto su računari postajali sve moćniji, vreme potrebno za obavljanje zadatih instrukcija je postalo veoma kratko u odnosu na vreme potrebno da se izvrši promena programera koji koristi računar. Iz tog razloga dolazi do angažovanja obučenih operatera koji su bili zaduženi za upravljanje računarskim sistemima (slika 1.18.). Operater je bio osoba koja opslužuje računarski sistem i njegov zadatak je bio da pripremi sve što je potrebno da se zadatak može obaviti. Tadašnji sistemi su bili spori i imali su mogućnost izvršavanja samo jednog programa, tako da je operater bio u mogućnosti da sve potrebne radnje obavi na vreme. Najveći deo vremena trošio se na poslove operatera i ulazno-izlazne operacije, a mnogo manji deo na rad centralnog procesora.



Slika 1.18. IBM 704 i operater

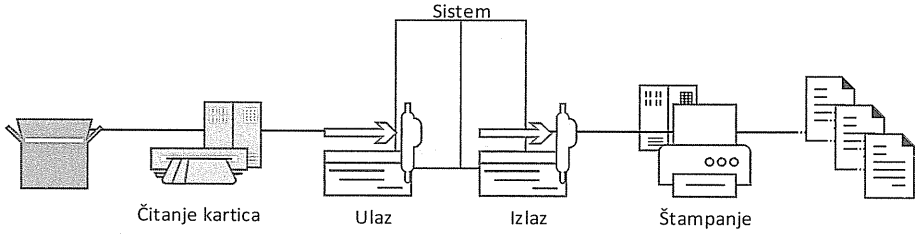
Pošto je ovakav način manipulisanja podacima bio veoma neudoban, ali i dovodio do slabe iskorišćenosti procesora, kao najvažnijeg (i najskupljeg) dela računarskog sistema, došlo je do potrebe da se sistemi učine efikasnijim. Prvi korak ka tom cilju 50-tih godina dvadesetog veka bilo je automatsko punjenje i pražnjenje memorije koje predstavlja i začetak operativnih sistema. U to vreme razvili su se programi koji su imali zadatak da vrše automatske konverzije binarnih i dekadnih podataka. Takođe, računarski centri počeli su da posvećuju pažnju čuvanju podataka kako se oni ne bi izgubili usled računarskih grešaka, trudili su se da prošire sistemske biblioteke kako bi sprečili pogrešno korišćenje resursa, uvedeni su automatski signali za potrebe komunikacije sa operaterom, uvedene su bezbednosne mere itd. Sve prethodno pomenute inovacije dovele su do pojave operativnih sistema. Osnovni programi koji se izvršavaju u pozadini a služe za planiranje i izvršavanje poslova i upravljanje računarskim resursima predstavljaju prethodnike korisnički orijentisanih

operativnih sistema. Sistem *GM-NAA I\O* koji je 1956. godine isprogramiran za računar *IBM 704* smatra se prvim operativnim sistemom. On je omogućavao automatizovano izvršavanje programa napisanih na simboličkom jeziku.

Sledeća generacija računara, zasnovana na tranzistorima, dovela je i do ekspanzije novih koncepata kada je računarstvo u pitanju. Jedan od koncepata koji postoji i danas je *paketna obrada* (*Batch processing*).

Ovakav način izvršavanja programa podrazumeva da se oni nadovezuju jedan na drugi, tj. da se blokovi kartica korisničkih programa ređaju jedan za drugim. Računari specijalizovani za ulazno-izlazne operacije bi se koristili za prenos podataka sa uredno složenih bušenih kartica (slika 1.19.) na magnetne trake. Potom bi se magnetna traka prenosila na računar specijalizovan za obradu sa posebno podržanim matematičkim operacijama.

Operativni sistem specijalizovanog računara vodio je računa o punjenju i pražnjenju memorije. Prvo bi sa ulazne trake učitavao program, a zatim program za izvršavanje učitanog korisničkog programa, onda izvršavao program i na kraju zapisivao rezultat na izlaznu magnetnu traku. Postupak bi se ponavljao sve dok se ne izvrše svi programi sa ulazne magnetne trake. Izlazna traka bi se na kraju prenosila na računar specijalizovan za ulazno-izlazne operacije kako bi se rezultati odštampali.

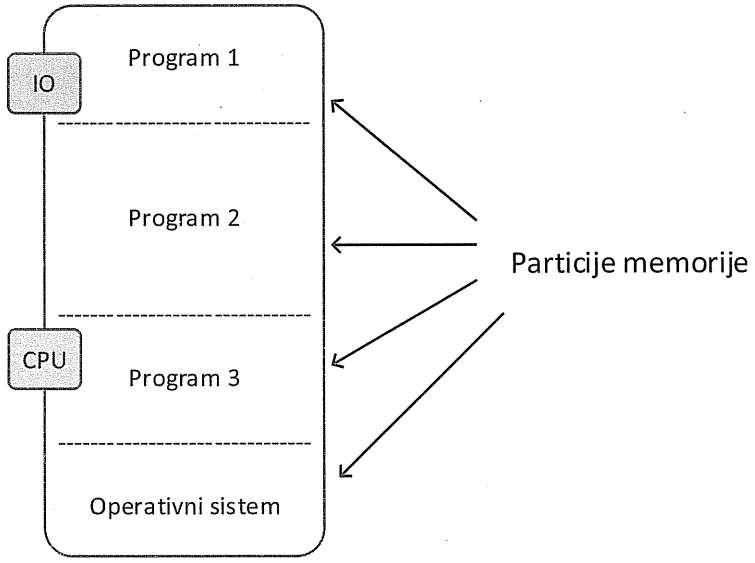


Slika 1.19. Paketna obrada

Ovakav pristup omogućavao je da se u trenutku izvršavanja jednog programa učitava sledeći koji je na redu. Smatra se da je 1962. godine

kompanija CSC (*Computer Science Corporation*) razvila sistem *EXEC II* koji je prvi podržavao obradu paketnog tipa a koristio se na računarima *UNIVAC 1107*.

Treća generacija računara zasnovana na integrisanim kolima, sredinom šezdesetih godina dvadesetog veka, dovela je do još većih razlika između komponenti računara. Dolazi do velikog nesklada kada su brzine u pitanju, prvenstveno između procesora i periferijskih uređaja. Periferijski uređaji su u to vreme bili mehanički uređaji i većina poslova je zahtevala saradnju ovih komponenti jer programi obično naizmenično rade sa ulazno-izlaznim uređajima i obrađuju podatke dok se program ne izvrši do kraja. Vremenske jedinice kojima su se merile brzine rada procesora bile su *ns,* diskova – *ms* a štampača – *s*. Iz tih razloga procesor je bio nedovoljno iskorišćen jer je često morao da čeka na sporije komponente. *Multiprogramiranje,* kao bitan koncept u razvoju operativnih sistema, pružalo je rešenje za gorenavedene probleme.

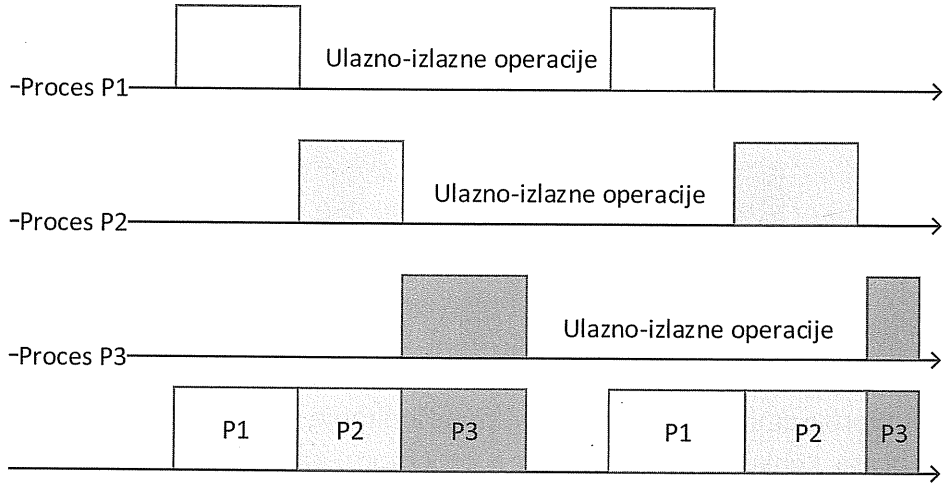


Slika 1.20. Programi u memoriji

Osnovna ideja multiprogramiranja je bila da se u radnu memoriju smesti više programa (procesa) kako bi se poboljšala iskorišćenost procesora (slika 1.20.). Memorija bi se podelila na delove (particije) u koje bi se

učitavali programi. Programi koji imaju potrebu da rade na procesoru bi se smenjivali na njemu tako da on uvek bude zaposlen, a dok se jedan program izvršava na procesoru drugi imaju mogućnost da izvršavaju ulazno-izlazne operacije (slika 1.21.).

Glavni cilj multiprogramiranja je maksimalno povećanje iskorišćenosti sistema, ali je sa druge strane poželjno da vreme izvršavanja programa bude što manje. Iz tog razloga, pri projektovanju operativnih sistema se morao napraviti balans kada su ove dve stvari u pitanju.

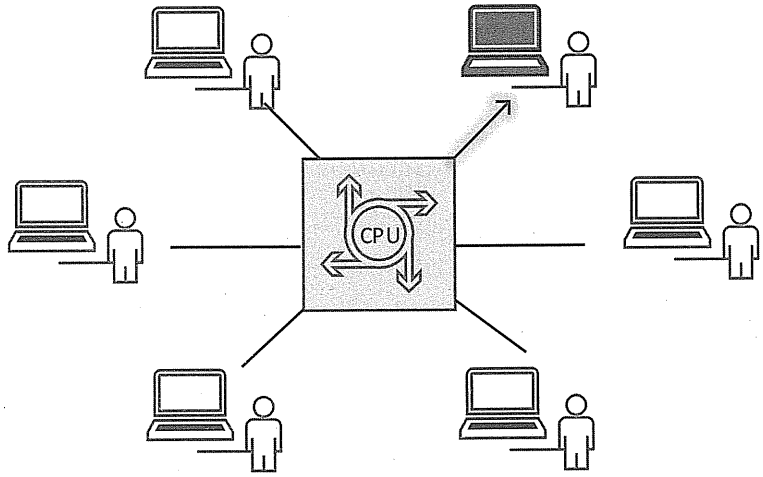


Slika 1.21. Multiprogramiranje

Na ideji multiprogramiranja zasnovani su još neki bitni koncepti u računarstvu poput deljenja vremena, multitaskinga i multiprocesiranja.

Deljenje vremena (*Time-sharing*) je koncept koji se zasniva na deljenju računara između više korisnika. Pošto je više korisnika zahtevalo rad na računaru, a procesori su bili dovoljno moćni da obrade zahteve, javio se problem deljenja procesora između više korisnika. Procesor nije bilo moguće podeliti fizički, pa se došlo na ideju da se podeli „vremenski" tako što bi svaki korisnik dobio određeno vreme u kojem bi imao procesor na raspolaganju (slika 1.22.). Po isteku dodeljenog vremena procesor bi dobio sledeći korisnik. Mali vremenski intervali koje korisnik često dobija

stvarali su iluziju da korisnik na raspolaganju ima procesor koji sve vreme radi samo za njega.

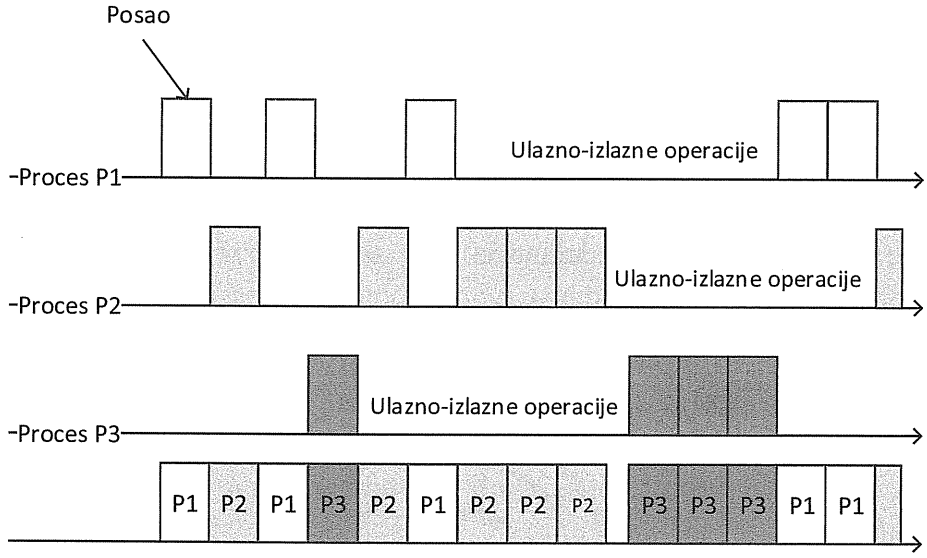


Slika 1.22. Deljenje vremena

Multitasking je „moderniji" pristup u odnosu na multiprogramiranje. Tačnije, on se odnosi na efikasniji način implementacije ideje multiprogramiranja. Izvorno multiprogramiranje podrazumeva da se više procesa konkurentno izvršava na procesoru, tako da se jedan proces izvršava na procesoru sve dok na red ne dođu ulazno-izlazne operacije kada se on prekida i procesor dodeljuje sledećem procesu koji na njega čeka. Multitasking podrazumeva da je jedinica izvršavanja na procesoru posao (*task*) koji ne mora nužno da obuhvata izvršavanje procesa na procesoru između dve ulazno-izlazne operacije (slika 1.23.). Takođe, ovaj pristup podrazumeva da se sistem organizuje na sličnim principima kao i kada je multiprogramiranje u pitanju tako što će proces po izvršavanju posla osloboditi procesor za sledeći posao procesa koji je na redu. Pojmovi multiprogramiranja i multitaskinga se često poistovećuju jer su dosta slični.

Multiprocesiranje ili multiprocesorski sistemi su pojam u računarstvu koji doživljava ekspanziju u poslednjoj generaciji računara. Multiprocesiranje se obično odnosi na postojanje više procesora u računarskom sistemu koji

hardverski omogućavaju istovremeno izvršavanje više poslova (različitih ili istih) procesa. Ono se često poistovećuje sa pojmovima multiprogramiranja i multitaskinga što nije pogrešno, ali se više odnosi na hardverske mogućnosti sistema.



Slika 1.23. Multitasking

1.5. ZNAČAJNI OPERATIVNI SISTEMI

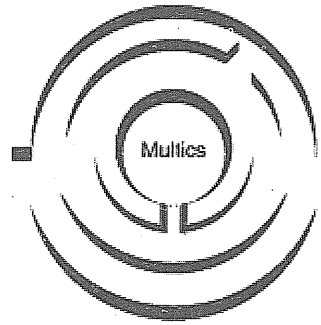
U narednom delu biće kratko opisani neki od najvažnijih operativnih sistema koji su obeležili razvoj računarstva. Neki od njih su bili samo dobra zamisao koja je inspirisala neke od kasnijih komercijalno uspešnijih sistema. Takođe, navedeni su i neki od savremenih operativnih sistema koji su zaslužili da budu pomenuti zbog svoje popularnosti i inovativnosti. U sisteme koji su obeležili razvoj računarstva spadaju:

- Multics;
- UNIX;
- Linux;
- DOS i Windows;
- Apple OS;

- Android.

1.5.1. Multics

Operativni sistem *Multics* (*Multiplexed Information and Computing Service*) je bio zajednički pokušaj najvećih kompanija i univerziteta u to vreme, poput MIT (*Massachusetts Institute of Technology*), General Electric i Bel laboratorija (*Bell Laboratories*), da naprave operativni sistem u koji bi bila implementirana sva najbolja, do tada poznata, rešenja. Učesnici u projektu su želeli da za moćne računare kreiraju operativni sistem koji će biti u stanju da radi sa velikim brojem terminala. Oni su budućnost računarstva videli tako da u svakom gradu postoji moćan centralni računar, a da građani u svojim domovima poseduju terminale uz pomoć kojih (modemom) pristupaju glavnom računaru.



Slika 1.24. Zvanični logo Multics-a

Na razvoju Multics sistema, od 1964. godine, bili su zaposleni veliki timovi stručnjaka i neki od najboljih programera tog vremena. U ovaj operativni sistem implementirani su svi najbolji do tada poznati algoritmi. Rezultat nije bio onoliko dobar koliko se očekivalo. Multics se u praksi pokazao veoma loše jer je puno procesorskog vremena trošio na donošenje odluka (na svoj rad), a malo ostavljao korisnicima. Osim toga, ovaj projekat je bio i veoma skup. Iz tih razloga Multics nikada nije ozbiljno zaživeo na računarima. Iako je u krajnjoj liniji bio neuspešan, ovaj operativni sistem je ostavio dubok trag u kasnijoj evoluciji operativnih sistema. Imajući u

vidu osnovnu koncepciju ovog sistema, on se može smatrati pretečom interneta. Na slici 1.24. može se videti logo operativnog sistema Multics.

1.5.2. Operativni sistemi UNIX familije

Krajem šezdesetih i početkom sedamdesetih godina dvadesetog veka, programeri iz Bel laboratorija (*AT&T Bell Laboratories, Bell Telephone Laboratories*) Ken Tompson (*Ken Thompson*) i Denis Riči (*Dennis Ritchie*), koji su među poslednjima napustili rad na sistemu Multics, sa svojim kolegama su odlučili da napišu operativni sistem za mali računar PDP-7. Ovaj sistem je bio uprošćena varijanta Multics-a i iz njega se razvio operativni sistem *UNIX* (*Uniplexed Information and Computing System*).

Ken Tompson je kreirao programski jezik *B* na osnovu kojeg je Denis Riči osmislio programski jezik *C* kako bi na njemu bio isprogramiran deo operativnog sistema. Naime, kritični delovi sistema su bili isprogramirani na asembleru, a ostatak na programskom jeziku *C*. U drugoj iteraciji kompletan UNIX je napisan na *C*-u što je značajno doprinelo njegovoj prenosivosti na različite sisteme. Pošto je kompanija Bel imala monopol nad telefonijom i iz tog razloga nije smela da se bavi računarstvom i nastavi rad na UNIX-u, doneta je odluka da se kompletan kôd ustupi univerzitetima i drugim kompanijama kako bi se dalje razvijao. Ovaj korak je verovatno bio presudan kako za sam UNIX tako i za razvoj operativnih sistema uopšte. Najpoznatije verzije UNIX-a implementirane su na Univerzitetu Berkli, sa oznakom BSD (*Berkeley Software Distributions*) koje su date na korišćenje američkim univerzitetima radi daljeg usavršavanja. U ovim verzijama operativnog sistema UNIX prvi put je implementirana podrška za umrežavanje. Ova mrežna rešenja prihvatili su i brojni proizvođači računara koji su razvijali sopstvene varijante operativnog sistema UNIX:

- SunOS – kompanije Sun Microsystems, zasnovan na sistemu BSD UNIX v4.2;
- System V UNIX – kompanije AT&T (*Bell*);
- XENIX – kompanije Microsoft.

Tabela 1.1. Verzije UNIX operativnog sistema

Operativni sistem	Datum	Operativni sistem	Datum
UNIX (V1)	1971.	BSD/OS 2.0	1995.
UNIX (V2)	1972.	OpenBSD 1.0	1995.
UNIX (V3)	1973.	IRIX 6.2	1996.
UNIX (V4)	1973.	NetBSD 1.2	1996.
UNIX (V5)	1974.	OpenBSD 2.0	1996.
UNIX (V6)	1975.	IRIX 6.4	1997.
1 BSD	1977.	BSD/OS 3.0	1997.
2 BSD	1978.	SCO UnixWare 7.0	1998.
3 BSD	1979.	Solaris 7	1998.
UNIX (V7)	1979.	BSD/OS 4.0	1998.
4 BSD	1980.	FreeBSD 3	1998.
4.1 BSD	1981.	NetBSD 1.4	1999.
Unix System III	1981.	True64 UNIX 5.0	1999.
Unix System IV	1982.	Solaris 8	2000.
SunOS 1.0	1982.	FreeBSD 4	2000.
Unix System V	1983.	SCO Open UNIX 8	2001.
4.2 BSD	1983.	OpenBSD 3.0	2001.
Unix System V Release 2	1984.	Solaris 9	2002.
UNIX (V8)	1985.	FreeBSD 5	2003.
SunOS 2.0	1985.	BSD/OS 5.0	2003.
UNIX (V9)	1986.	NetBSD 2.0	2004.
Unix System V Release 3	1986.	Solaris 10	2005.
4.3 BSD	1986.	FreeBSD 6	2005.
SunOS 3.0	1986.	NetBSD 3.0	2005.
Unix System V Release 3.2	1987.	OpenBSD 4.0	2006.
IRIX 1.0	1987.	NetBSD 4.0	2007.
Unix System V Release 4	1988.	FreeBSD 7	2008.
UNIX (V10)	1989.	NetBSD 5.0	2009.
SunOS 4.0	1989.	FreeBSD 8	2009.
IRIX 4.0	1990.	OpenBSD 5.0	2011.
OSF/1	1991.	Solaris 11	2011.
Solaris 2.0 (SunOS 5.0)	1992.	NetBSD 6.0	2012.
Unix System V Release 4.2	1992.	FreeBSD 9	2012.
4.4 BSD	1992	FreeBSD 10	2014.
BSD/OS 1.0	1993.	NetBSD 7.0	2015.
Novell UnixWare 1.1	1993.	FreeBSD 11	2016.
IRIX 5.3	1993.	OpenBSD 6.0	2016.
FreeBSD 1	1993.	FreeBSD 12.0	2018.
NetBSD 1.0	1994.	NetBSD 8.0	2018.
IRIX 6.0	1994.	NetBSD 9.0	2020
FreeBSD 2	1994.	FreeBSD 13.0	2021.
Novell UnixWare 2.0	1995.	OpenBSD 7.0	2021.

System V Release 4 (SVR4) je verzija UNIX-a koja je nastala 1988. godine kombinacijom dobrih rešenja iz verzija: Sun OS/BSD, AT&T System V Release 3 i XENIX operativnih sistema. Ova verzija je ubrzo postala polazna osnova za dalji razvoj UNIX sistema. Danas postoji veliki broj vrsta UNIX sistema koje su zasnovane ili na industrijskom standardu SVR4 ili na BSD distribuciji. One vrste UNIX sistema koje potiču od istog standarda međusobno su veoma slične.

X/Open konzorcijum je 1984. godine osnovalo nekoliko evropskih proizvođača UNIX-a, radi promovisanja otvorenih standarda u polju *IKT-a* (Informaciono komunikacionih tehnologija). Spajanjem *OSF* (*Open Software Foundation*) i X/Open-a, 1996. godine se formira *Otvorena grupa* (*The Open Group*), koja je i danas prepoznata kao sertifikaciono telo za UNIX.

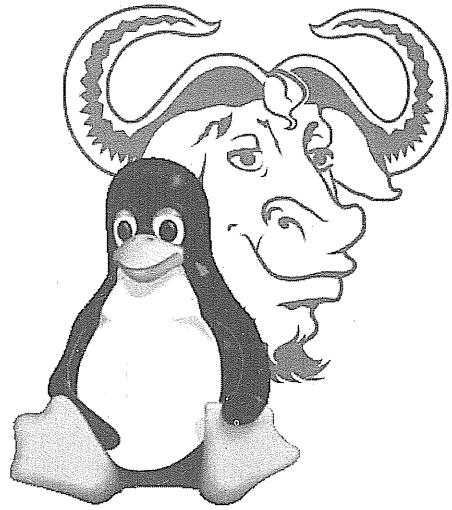
U tabeli 1.1. se mogu videti najpoznatije verzije UNIX-a kao i godine kada su predstavljene.

1.5.3. GNU/Linux

Ričard M. Stolman (*Richard M. Stallman*) je 1983. godine pokrenuo inicijativu koja je trebalo da dovede do stvaranja slobodnog operativnog sistema na osnovama UNIX-a. Ovaj projekat nazvan je *GNU* što je skraćenica od rekurzivnog naziva „GNU nije UNIX" (*GNU's Not UNIX*). Projekat je dobro napredovao u svim delovima osim kada je u pitanju bilo jezgro. GNU/HURD jezgro se dosta sporo razvijalo a prvo javno testiranje 1996. godine pokazalo je dosta nedostataka ovog jezgra.

Finski programer Linus Torvalds je 1991. godine započeo rad na izradi operativnog sistema za računare koji koriste procesor Intel 80386. Prvo je napisao jednostavan emulator za terminale koji je koristio za povezivanje sa UNIX sistemom na fakultetu, a onda je nastavio da ga razvija i poboljšava. Linus je bio inspirisan operativnim sistemom *Minix* koji je par godina ranije za potrebe svojih kurseva razvio Endru Tanenbaum (*Andrew Stuart „Andy" Tanenbaum*). Za razliku od Minix-a koji je imao mikrojezgro, prva verzija Linusovog jezgra (0.01), koja je u septembru 1991. godine

postala javno dostupna, bila je monolitna. Vrlo brzo, javno dostupna je postala i verzija 0.02 sa izvornim kodom. Linus Torvalds je tom prilikom pozvao sve programere da se pridruže projektu.



Slika 1.25. Logoi Linux-a i GNU-a

Linusovo jezgro je dobio ime *Linux* i spajanjem Linux jezgra i GNU alata kompletiran je operativni sistem *GNU/Linux* (slika 1.25.). Danas postoji veliki broj različitih distribucija GNU/Linux sistema a neke od najpoznatijih su: Ubuntu, Slackware, Debian, Fedora, SuSE i druge. Sve verzije operativnog sistema GNU/Linux odlikuje stabilnost, brzina i visok stepen bezbednosti. U poslednje vreme umesto GNU/Linux odomaćio se naziv Linux što proizvodi česte polemike između programera širom sveta. Pristalice GNU projekta insistiraju na tome da se naziv Linux može koristiti isključivo kada se govori o jezgru, a ne i o kompletnom operativnom sistemu.

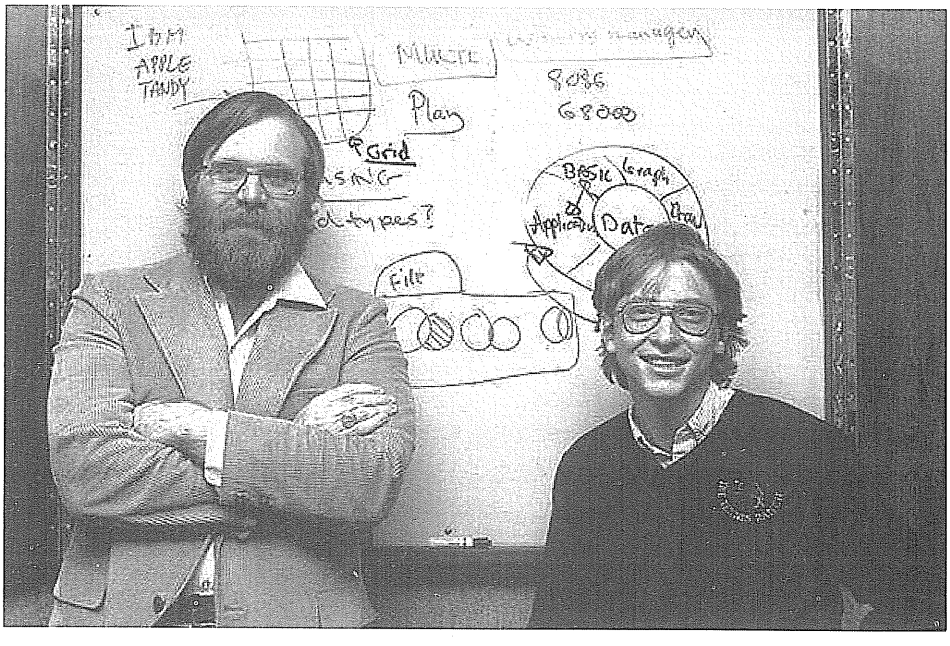
Jezgro Linux operativnog sistema je softver otvorenog koda sa licencom GNU GPL (*General Public Licence*, verzija 2.0), što znači da su korisnici slobodni da preuzimaju izvorni kôd i prave izmene, ali i da su u obavezi da korisnicima daju ista prava koja su i sami koristili ukoliko žele dalje da distribuiraju izmenjeni sistem. U tabeli 1.2. prikazane su najpoznatije distribucije operativnog sistema Linux.

Tabela 1.2. Verzije Linux operativnog sistema

Operativni sistem	Datum	Operativni sistem	Datum
Debian	1993.	Zentyal	2005.
Slackware	1993.	Linux Mint	2006.
SUSE Linux	1994.	Xubuntu	2006.
Red Hat Linux	1995.	openSUSE	2006.
Mandriva Linux	1998.	Oracle Linux	2006.
ELinOS	1999.	gOS	2007.
MontaVista Linux	1999.	Gobuntu	2007.
Red Flag Linux	1999.	Ubuntu Studio	2007.
VectorLinux	1999.	antiX	2007.
Yellow Dog Linux	1999.	SHR	2008.
Ututo	2000.	Super OS	2008.
Finnix	2000.	XBMC	2008.
Trustix	2000.	EasyPeasy	2008.
ClearOS	2000.	Kuki Linux	2008.
aLinux	2001.	Tor-ramdisk	2008.
CRUX	2001.	Netrunner	2009.
ALT Linux	2001.	Emdebian Grip	2009.
Freespire	2002.	Salix OS	2009.
Arch Linux	2002.	Google Chrome OS	2009.
Gentoo Linux	2002.	Tiny Core Linux	2009.
Lunar Linux	2002.	Chakra	2010.
Slax	2002.	MintPPC	2010.
Red Hat Enterprise	2002.	ArchBang	2010.
PCLinuxOS	2003.	Mageia	2010.
Kanotix	2003.	Porteus	2010.
MEPIS	2003.	Bodhi Linux	2011.
Fedora	2003.	elementary OS	2011.
Knoppix	2003.	OpenELEC	2011.
Puppy Linux	2003.	Ubuntu GNOME	2012.
CentOS	2003.	ESSPEE	2012.
KnoppMyth	2004.	Qubes OS	2012.
Foresight Linux	2004.	Instant WebKiosk	2012.
Ubuntu	2004.	Kali Linux	2013.
Scientific Linux	2004.	SolydXK	2013.
Pentoo	2005.	OpenMandriva Lx	2013.
Edubuntu	2005.	KaOS	2014.
Elive	2005.	Peach OSI	2014.
Parsix	2005.	AryaLinux	2015.
Kubuntu	2005.	Ubuntu MATE	2015.
Trisquel	2005.	GeckoLinux	2016.
Gnoppix	2005.	Parrot	2016.
Grml	2005.	Raspberry Pi OS	2020.

1.5.4. Operativni sistemi kompanije Microsoft

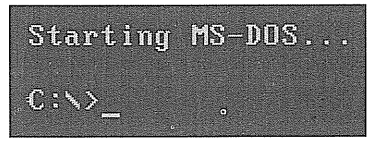
Operativni sistem CP/M, firme *Digital Research*, koji je sredinom sedamdesetih godina dvadesetog veka razvio Geri Kildal (*Gary Kildall*) bio je namenjen osmobitnim računarima koji su imali mikroprocesore Intel 8080 i Zilog Z80. Ovaj sistem je imao višeslojnu arhitekturu što je bio prvi slučaj da kod operativnih sistema postoji nivo apstrakcije koja ga deli u nezavisne celine.



Slika 1.26. Pol Alen i Bil Gejts, osnivači Microsoft-a

Kompanija Intel je 1981. godine proizvela mikroprocesor 8088 sa 16-bitnom arhitekturom. Iz tog razloga kompaniji IBM je bio potreban novi operativni sistem koji bi je podržao. Međutim, programeri su početkom osamdesetih godina dvadesetog veka smatrali da je 1 MB memorije više nego što će bilo kome biti potrebno, pa je firma Digital Research odlučila da ostane pri svojim 8-bitnim rešenjima i ne razvija 16-bitni operativni sistem. Za razliku od konkurencije, Bil Gejts (*Bill Gates*), koji je bio na čelu tada skromne firme *Microsoft* (slika 1.26.), imao je viziju: „računar na svakom radnom stolu i u svakom domu“. Kompanija IBM sa Microsoft-om

sklapa ugovor o kupovini 16-bitnog operativnog sistema *MS-DOS* (*Microsoft Disk Operating System*). Taj sistem je razvila kompanija *Seattle Computer Products* pod nazivom 86-DOS, ali ga je Microsoft otkupio, prilagodio potrebama sistema, licencirao pod novim imenom i prodao IBM-u. IBM je ovaj sistem instalirao na prvi personalni računar PC 1981. godine pod nazivom PC-DOS. Verzije namenjene isključivo IBM računarima Microsoft je nazivao PC-DOS, dok su verzije namenjene ostatku tržišta nosile oznaku MS-DOS. Primarni zadatak operativnog sistema DOS je bilo upravljanje diskovima. On nije imao grafički korisnički interfejs, već su se komande zadavale iz konzole, tj. iz linijskog korisničkog interfejsa (slika 1.27.) unošenjem teksta sa tastature.



Slika 1.27. MS-DOS

Microsoft kroz partnerstvo sa IBM-om, ali i samostalnim nastupom, počinje da osvaja tržište. Tada i kompanija Digital Research pokušava da se svojim sistemom DR-DOS uključi u trku. Međutim, dobar marketing i pravi trenutak za pojavljivanje operativnih sistema kompanije Microsoft bio je presudan za dobijanje bitke.

Tokom skoro dvadeset godina razvoja DOS-a pojavile su se mnoge verzije čiji se pregled može videti u tabeli 1.3.

Kada se 1984. godine pojavio Apple Macintosh, dolazi do naglog porasta interesovanja za grafičkim korisničkim interfejsom. Microsoft je to na vreme shvatio i snage usredsredio na kreiranje grafičkog interfejsa. U novembru 1985. pojavila se prva verzija Windows-a koji je bio grafička nadogradnja na DOS. Ova veza će opstajati sve do verzije Windows-a NT koji je potpuno nezavisan od DOS-a. Nakon toga, u okviru novih operativnih sistema ove familije, ostaje samo emulacija starog sistema koja je bila neophodna kako bi se omogućila kompatibilnost sa starim programima.

Tabela 1.3. Verzije DOS operativnog sistema

Operativni sistem	Datum	Operativni sistem	Datum
PC-DOS 1.0	08.1981.	MS-DOS 4.0	06.1988.
PC-DOS 1.1	01.1982.	PC-DOS 4.0	05.1988.
MS-DOS 1.25	01.1982.	MS-DOS 4.01	11.1988.
MS-DOS 2.0	03.1983.	MS-DOS 5.0	06.1991.
PC-DOS 2.1	10.1983.	MS-DOS 6.0	03.1993.
MS-DOS 2.11	03.1984.	MS-DOS 6.2	11.1993.
MS-DOS 3.0	08.1984.	MS-DOS 6.21	02.1994.
MS-DOS 3.1	11.1984.	PC-DOS 6.3	04.1994.
MS-DOS 3.2	01.1986.	MS-DOS 6.22	06.1994.
PC-DOS 3.3	04.1987.	PC-DOS 7.0	04.1995.
MS-DOS 3.3	08.1987.		

Radni naziv operativnog sistema sa grafičkim korisničkim interfejsom bilo je *Interface Manager*, jer je i zamišljen kao dopuna postojećem DOS-u. Na kraju je kao najadekvatnije ime izabran *Windows*, jer najbolje opisuje okvire ili računarske „prozore" koji predstavljaju temelj ovog operativnog sistema.

U principu, novonapravljeni operativni sistem nije bio najbolje rešenje jer je u tom trenutku mogao da se napravi bolji, koji podržava više memorije, multitasking itd. Međutim, Microsoft je ovim sistemom želeo da napravi brzi prelazni korak kako ne bi izgubio tržište. Sa druge strane, kompanija IBM je želela ozbiljniji operativni sistem, ali i ekskluzivnu podršku (drajvere) samo za njihov hardver. Iz tih razloga postepeno dolazi do razlaza ove dve kompanije. IBM razvija svoj operativni sistem OS/2, a Microsoft operativne sisteme Windows.

Od prve, ne tako uspešne verzije Windows-a, pa do danas predstavljeno je na desetine manje-više uspešnih verzija koje su kompaniju Microsoft promovisale u lidera kada su operativni sistemi na računarima u pitanju. U tabeli 1.4. se mogu videti sve verzije Windows operativnog sistema.

Poslednje verzije operativnog sistema kompanije Microsoft su redizajnirane i imaju potpuno novi pristup kada su grafički korisnički

interfejsi u pitanju. Najveću novinu predstavlja činjenica da se ovi sistemi mogu koristi na serverima, moćnim računarima, PC računarima ali i na tabletima i pametnim telefonima. Dakle, njihov interfejs se može koristiti kako na ekranima osetljivim na dodir tako i korišćenjem miša i tastature.

Tabela 1.4. Verzije Windows operativnog sistema

Operativni sistem	Datum	Operativni sistem	Datum
Windows 1.0	20.11.1985.	Windows 98	25.06.1998.
Windows 2.0	09.12.1987.	Windows 98 SE	05.05.1999.
Windows 2.10	27.05.1988.	Windows 2000	17.02.2000.
Windows 2.11	13.03.1989.	Windows ME	14.09.2000.
Windows 3.0	22.05.1990.	Windows XP	25.10.2001.
Windows 3.1	06.04.1992.	Windows Vista	08.11.2006.
Windows NT 3.1	27.07.1993.	Windows 7	22.07.2009.
Windows NT 3.5	21.09.1994.	Windows 8	26.10.2012.
Windows NT 3.51	30.05.1995.	Windows 8.1	18.10.2013.
Windows 95	24.08.1995.	Windows 10	29.08.2015.
Windows NT 4.0	24.08.1996.	Windows 11	05.10.2021.

1.5.5. Operativni sistemi kompanije Apple

Operativni sistemi kompanije *Apple* sa grafičkim korisničkim interfejsom pojavili su se 1984. Verzija 7.6 koja se pojavila 1996. godine je prva verzija koja je nosila naziv *Mac OS*, dok su se prethodne zvale sistemski softver i isporučivale uz hardver. Prve verzije sistema su bile namenjene isključivo Macintosh računarima. Međutim, kada je Apple napravio konfiguraciju računara sa PC hardverom, onda je i sistem prilagođen takvoj arhitekturi.

Prva izdanja operativnih sistema kompanije Apple su imala mogućnost da pokreću i izvršavaju samo jedan program (uz operativni sistem) u određenom trenutku. Tek kasnije je verzija System Software 5.0 dopuštala sistemu da pokreće više programa odjednom. Vremenom su realizovane mnoge verzije ovog proizvođača koje su nosile imena velikih mačaka (tabela 1.5.). Međutim, pošto su nazivi vrsta velikih mačaka iskorišćeni, Apple tim je odlučio da se, počevši od verzije 10.9 (*Mavericks*),

verzije operativnih sistema nazivaju po mestima u Kaliforniji. Mavericks je mesto u Severnoj Kaliforniji čuveno po surfovanju na velikim talasima.

Tabela 1.5. Verzije Mac operativnog sistema

Operativni sistem	Datum	Operativni sistem	Datum
Mac System Software	01.1984.	Mac OS X 10.5 (Leopard)	10.2007.
System Software	09.1985.	Mac OS X 10.6 (Snow Leopard)	08.2009.
System Software 1.0	02.1986.	Mac OS X 10.7 (Lion)	07.2011.
System Software 2.0	01.1987.	OS X 10.8 (Mountain Lion)	07.2012.
System Software 5.0	10.1987.	OS X 10.9 (Mavericks)	10.2013.
System Software 6.0	04.1988.	OS X 10.10 (Yosemite)	10.2014.
System Software 7.0	06.1991.	OS X 10.11 (El Capitan)	09.2015.
Mac OS 7.6	01.1997.	macOS 10.12 (Sierra)	09.2016.
Mac OS 8.0	07.1997.	macOS 10.13 (High Sierra)	09.2017.
Mac OS 9.0	10.1999.	macOS 10.14 (Mojave)	09.2018.
Mac OS X 10.0 (Cheetah)	03.2001.	macOS 10.15 (Catalina)	10.2019.
Mac OS X 10.1 (Puma)	09.2001.	macOS 11 (Big Sur)	11.2020.
Mac OS X 10.2 (Jaguar)	08.2002.	macOS 12 (Monterey)	10.2021.
Mac OS X 10.3 (Panther)	10.2003.	macOS 13 (Ventura)	10.2022.
Mac OS X 10.4 (Tiger)	04.2005.		

iOS je operativni sistem koji kompanija Apple 2007. godine razvija za pametne telefone iPhone. Kasnije je podržao druge uređaje kompanije Apple (iPodTouch, iPad, iPad mini itd.). Naziv sistema je do juna 2010. godine bio iPhone OS da bi od tada postao samo iOS. Praktično iOS je mobilna verzija sistema OS X.

Epohalna novina koju je ovaj sistem doneo je korisnički interfejs koji se zasniva na multitač (*multitouch*) pristupu. Ovakav pristup podrazumeva da uređaj ima mogućnost da u istom trenutku prepoznaje više pritisaka ekrana. Interakcija sa ekranima osetljivim na dodir se zasniva na potezima prstiju.

Pre pojave iOS-a, telefoni nisu imali ekran osetljiv na dodir ili je on bio rezistivni i zahtevao relativno jako pritiskanje i nije imao mogućnost registrovanja više dodira u istom trenutku. Situacija se promenila ugradnjom kapacitivnih ekrana u iPhone uređaje. Usavršeno je skrolovanje (pomeranje sadržaja na ekranu) i omogućeno uveličavanje prstima (*pinch-to-zoom*) što su i bile glavne novine koje je Apple doneo ovim modelom. U tabeli 1.6. mogu se videti prvi nazivi i datumi objavljivanja novih osnovnih verzija operativnog sistema za mobilne uređaje kompanije Apple.

Tabela 1.6. Verzije iOS operativnog sistema

Operativni sistem	Datum	Operativni sistem	Datum
iPhone OS 1.0	29.06.2007.	iOS 9.0	16.09.2015.
iPhone OS 2.0	11.07.2008.	iOS 10.0	13.09.2016.
iPhone OS 3.0	17.06.2009.	iOS 11.0	19.09.2017.
iOS 4.0	21.06.2010.	iOS 12.0	17.09.2018.
iOS 5.0	12.10.2011.	iOS 13.0	19.09.2019.
iOS 6.0	19.09.2012.	iOS 14.0	16.09.2020.
iOS 7.0	18.09.2013.	iOS 15.0	20.09.2021.
iOS 8.0	17.09.2014.	iOS 16.0	12.09.2022.

1.5.6. Android

Android je jedan od najpopularnijih operativnih sistema za mobilne uređaje. Prvenstveno je namenjen za mobilne uređaje osetljive na dodir kao što su pametni telefoni i tablet računari. Zasnovan je na jezgru operativnog sistema Linux. Njegov razvoj je započela kompanija Android, koju je u početku samo finansijski podržavala kompanija Google da bi je 2005. godine i otkupila. Operativni sistem Android je predstavljen javnosti dve godine kasnije, a prvi telefon (*HTC Dream*) sa Android operativnim sistemom je prodat u oktobru 2008. godine.

Kompanija Google razvija Android u saradnji sa konzorcijumom *Open Handset Alliance*. Ovaj konzorcijum čine hardverske, softverske i telekomunikacione kompanije koje su pobornici otvorenih standarda za mobilne uređaje. Android je projekat otvorenog koda (*open-source*) što

znači da proizvođačima omogućava da ga menjaju i prilagođavaju potrebama svojih uređaja. Ovakav način licenciranja omogućio je veliku popularnost ovog sistema jer je privukao pažnju programera koji su dobili slobodu da razvijaju različite aplikacije.

Interakcija korisnika sa operativnim sistemom se odvija slično kao i kada je u pitanju iOS. Hardverske komponente pametnih telefona poput: akcelerometra, žiroskopa, senzora rastojanja itd. koriste se da bi neke aplikacije odgovorile na dodatne korisničke akcije. U tabeli 1.7. prikazane su verzije operativnog sistema Android dok se na Slici 1.28. mogu videti odgovarajući logoi.

Tabela 1.7. Verzije Android operativnog sistema

Operativni sistem	Datum	Operativni sistem	Datum
Android 1.0	23.09.2008.	Android 4.3 Jelly Bean	24.07.2013.
Android 1.1	09.02.2009.	Android 4.4 KitKat	31.10.2013.
Android 1.5 Cupcake	27.04.2009.	Android 5.0 Lolipop	03.11.2014.
Android 1.6 Donut	15.09.2009.	Android 5.1 Lolipop	09.03.2015.
Android 2.0 Eclair	26.10.2009.	Android 6.0 Marshmallow	05.10.2015.
Android 2.1 Eclair	12.01.2010.	Android 7.0 Nougat	22.08.2016.
Android 2.2 Froyo	20.05.2010.	Android 7.1 Nougat	04.10.2016.
Android 2.3 Gingerbread	06.12.2010.	Android 8.0 Oreo	21.08.2017.
Android 3.0 Honeycomb	22.02.2011.	Android 8.1 Oreo	05.12.2017.
Android 3.1 Honeycomb	10.05.2011.	Android 9.0 Pie	06.08.2018.
Android 3.2 Honeycomb	15.07.2011.	Android 10.0	03.09.2019.
Android 4.0 Ice Cream Sandwich	19.10.2011.	Androdi 11.0	08.09.2020.
Android 4.1 Jelly Bean	09.07.2012.	Android 12.0	04.10.2021.
Android 4.2 Jelly Bean	13.11.2012.	Android 13.0	15.08.2022.

Pitanja

1.1. Na kojim principima su zasnovani savremeni računarski sistemi?
1.2. Po čemu se razlikuju računarski sistemi danas od ranijih generacija?
1.3. Šta je hardver računarskog sistema?
1.4. Šta čini softver računarskog sistema?
1.5. Šta je aplikativni softver?
1.6. Navesti primere aplikativnih softvera.
1.7. Šta je sistemski softver?
1.8. Koji su osnovni zadaci operativnih sistema?
1.9. Čime bi operativni sistemi trebalo da upravljaju?
1.10. Objasniti odnos sistemskog softvera i operativnog sistema.
1.11. Šta su sistemski pozivi?
1.12. U kojim režimima mogu da rade procesori računarskih sistema?
1.13. Objasniti potrebu za promenom režima u toku izvršavanja programa.
1.14. Šta je jezgro operativnog sistema?
1.15. Od čega se sastoji jezgro operativnog sistema?
1.16. U kojem trenutku se jezgro operativnog sistema učitava u radnu memoriju?
1.17. Na koji način se izvršavaju sistemski pozivi?
1.18. Šta se postiže izvršavanjem što većeg broja aktivnosti u korisničkom režimu (umesto u sistemskom)?
1.19. Koja je uloga korisničkog okruženja?
1.20. Na koji način se mogu podeliti korisnička okruženja?
1.21. Koji je najvažniji deo linijskog korisničkog okruženja?
1.22. Šta su drajveri?
1.23. Koje uslove bi drajveri trebalo da zadovolje kada su u pitanju različiti tipovi iste vrste uređaja?
1.24. Kakvi pristupi postoje pri projektovanju operativnih sistema kada je skup funkcija koje će se nalaziti u jezgru u pitanju?
1.25. Objasniti pojam monolitnog jezgra.
1.26. Šta je mikrojezgro?

1.27. Koje osobine odlikuju arhitekturu sistema zasnovanih na mikrojezgru?
1.28. Šta je hibridna arhitektura jezgra?
1.29. Koji su glavni zadaci prilikom projektovanja hibridnih jezgara?
1.30. Koje su odlike egzojezgra?
1.31. Na kojoj tehnologiji su bili zasnovani računari prve generacije?
1.32. Koja je bila uloga operatera kod prvih računarskih sistema?
1.33. Koji sistem se smatra prvim operativnim sistemom?
1.34. Koji je novi koncept donela generacija računara zasnovana na tranzistorima?
1.35. Koje su glavne odlike paketne obrade podataka?
1.36. Koje su posledice uvođenja integrisanih kola kada su komponente računara u pitanju?
1.37. Objasniti koncept multiprogramiranja.
1.38. Ilustrovati multiprogramiranje primerom.
1.39. Kako izgleda životni vek procesa u sistemima koji podržavaju multiprogramiranje?
1.40. Koji su osnovni ciljevi multiprogramiranja?
1.41. Koja je osnovna ideja kada je deljenje vremena u pitanju?
1.42. Objasniti pojam multitaskinga.
1.43. U čemu je glavna razlika između multitaskinga i multiprogramiranja?
1.44. Šta se podrazumeva pod multiprocesiranjem?
1.45. Koji se operativni sistemi smatraju najznačajnijim u istoriji?
1.46. Šta je glavni doprinos operativnog sistema Multics kada je razvoj operativnih sistema u pitanju?
1.47. Koji su glavni razlozi za neuspeh operativnog sistema Multics?
1.48. Kako je nastao operativni sistem UNIX?
1.49. Ko je i iz kojih razloga osmislio programski jezik C?
1.50. Koji su glavni razlozi za veliku popularnost UNIX-a?
1.51. Koje su najpoznatije verzije operativnog sistema UNIX?
1.52. Iz kojih razloga je značajan rad Ričarda M. Stolmana?
1.53. Šta se podrazumeva pod pojmom Linux?
1.54. Ko je i kada kreirao Linux?

1.55. Ko je i u koje svrhe napravio operativni sistem Minix?
1.56. Prodiskutovati odnos GNU-a i Linux-a.
1.57. Koje su glavne prednosti GNU/Linux-a u odnosu na ostale operativne sisteme?
1.58. Koje su najpoznatije distribucije GNU/Linux operativnog sistema?
1.59. Prodiskutovati razloge za svojevremenu veliku popularnost operativnog sistema DOS.
1.60. Na čemu su bili zasnovani Windows operativni sistemi?
1.61. Šta je glavna odlika operativnih sistema kompanije Apple?
1.62. Na kakvim sistemima su prvi operativni sistemi kompanije Apple mogli da se koriste?
1.63. Koje novine je doneo iOS?
1.64. Kakvim uređajima je prvenstveno namenjen operativni sistem iOS?
1.65. Na kojoj osnovi je zasnovan operativni sistem Android?
1.66. Kakvoj vrsti uređaja je namenjen operativni sistem Android?
1.67. Koji su glavni razlozi za veliku popularnost operativnog sistema Android?
1.68. Analizirati trenutnu situaciju na tržištu kada su operativni sistemi u pitanju.

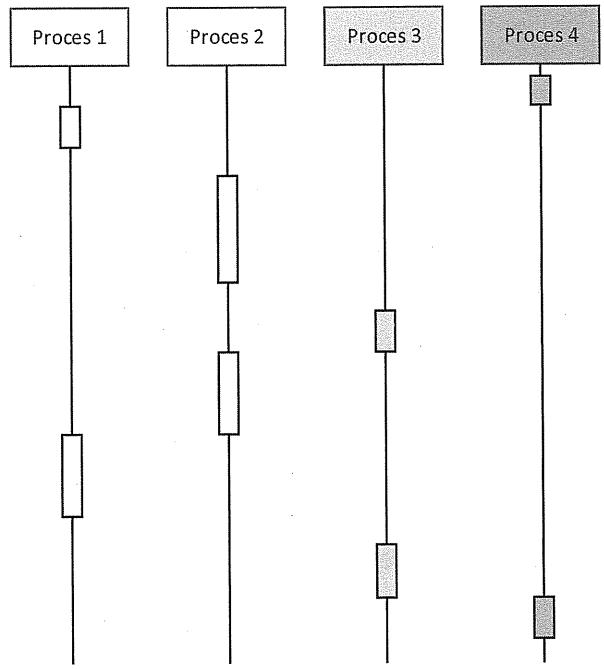
2. UPRAVLJANJE PROCESIMA

2.1. PROCESI

Jedan od ključnih koncepata, kada su operativni sistemi i računarski sistemi u pitanju, predstavljaju procesi. *Proces* se definiše kao program u izvršavanju. Kada programer napiše program (izvorni kôd) na nekom programskom jeziku, on je u stvari napisao tekst kojim se određeni algoritam implementira na tom jeziku. Izvorni kôd programa predstavlja niz instrukcija koje treba izvršiti i čuva se kao skup znakova (tekst). Prevođenjem napisanog izvornog programa na mašinski jezik, nastaje izvršni program (fajl na nekoj od sekundarnih memorija). Pokretanje izvršnog programa podrazumeva njegovo učitavanje u primarnu (radnu) memoriju računara i izvršavanje na procesoru. Na ovaj način pokrenuti program postaje proces. Proces je aktivan, a operativni sistem je dužan da vodi računa o njemu, tj. da mu obezbedi resurse koji su potrebni za izvršavanje. Da bi obavio zadatak zbog kojeg je pokrenut, procesu su potrebni resursi poput procesora, memorije, ulazno-izlaznih uređaja, fajlova itd.

Zadatak operativnih sistema je da obezbede da se procesi mogu efikasno izvršavati. Procesi mogu biti korisnički ili sistemski, a njihovo efikasno izvršavanje najčešće podrazumeva da se izvršavaju konkurentno i (ili) paralelno. Na sistemima čiji procesori imaju samo jedno jezgro, dodeljivanjem procesorskog resursa procesima koji se izvršavaju konkurentno, operativni sistem čini računarski sistem produktivnijim. Na taj način se stvara privid da se procesi izvršavaju u isto vreme, pa se često kaže da se izvršavaju pseudoparalelno. Upravljanje procesima na ovakav način predstavlja osnovni zadatak za operativne sisteme koji podržavaju multiprogramiranje i slične koncepte. Multiprogramiranje i srodni koncepti podrazumevaju učestalo smenjivanje procesa koji koriste procesor, pri čemu se svaki „po malo“ izvršava na procesoru, a

prebacivanje se izvršava toliko brzo i često da korisnici imaju utisak da se procesi izvršavaju istovremeno.



Slika 2.1. Pseudoparalelno izvršavanje procesa

Na slici 2.1. ilustrovana je paralelizacija, tj. pseudoparalelno izvršavanje četiri procesa na jednom procesoru. Kao i na slici, vremenski intervali koje procesi dobijaju za izvršavanje ne moraju biti jednaki i određuje ih operativni sistem.

Razvojem procesora i povećavanjem broja jezgara omogućeno je da se i više procesa na različitim jezgrima izvršava paralelno. Principi funkcionisanja su slični kao i u slučaju kada postoji samo jedno jezgro, ali su mogućnosti veće.

Osim izvršnog koda i podataka, bitne informacije o toku izvršavanja procesa sadrže i registri, a posebno programski brojač koji vodi računa o tome dokle se stiglo sa izvršavanjem procesa. Takođe, bitne informacije

za svaki proces su i podaci o otvorenim fajlovima, informacije o dozvolama i vlasniku procesa itd.

Zadatak operativnih sistema je da obezbede efikasne mehanizme za:

- Kreiranje i brisanje procesa;
- Upravljanje procesima;
- Komunikaciju između procesa;
- Sinhronizaciju procesa.

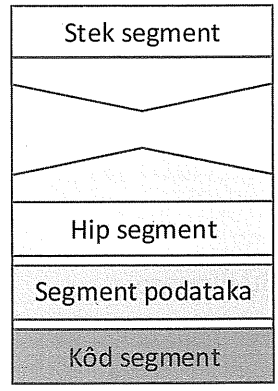
2.1.1. Proces u memoriji

Za razliku od programa koji je pasivan i opisuje sekvencu instrukcija koje treba izvršiti, proces je aktivan i izvršava se, korak po korak, po algoritmu koji je implementiran u programu. Programski brojač za svaki proces vodi računa o tome dokle se stiglo sa izvršavanjem, tj. koja je sledeća instrukcija. Instrukcije se nalaze u memoriji zajedno sa ostalim podacima koji su potrebni procesu za izvršavanje. Deo memorije namenjen za izvršavanje procesa se može organizovati na različite načine, a najčešći je prikazan u daljem tekstu.

Po pokretanju i učitavanju u memoriju programi, odnosno tada već procesi, obično raspolažu sa četiri dela memorije (slika 2.2.) koji se nazivaju segmenti i to su:

- Stek segment;
- Hip segment;
- Segment podataka;
- Kôd segment.

Stek segment (*Stack segment*) sadrži lokalne promenljive i parametre funkcija. *Hip segment* (*Heap segment*) je deo memorije u koji se smeštaju podaci koji se generišu u toku izvršavanja procesa, odnosno prostor koji se dinamički alocira u zavisnosti od potreba procesa. *Segment podataka* (*Data segment*) sadrži globalne promenljive, a instrukcije koje proces treba da izvrši nalaze se u *Kôd segmentu* (*Code segment, Text segment*).



Slika 2.2. Proces u radnoj memoriji

2.2. STANJA PROCESA

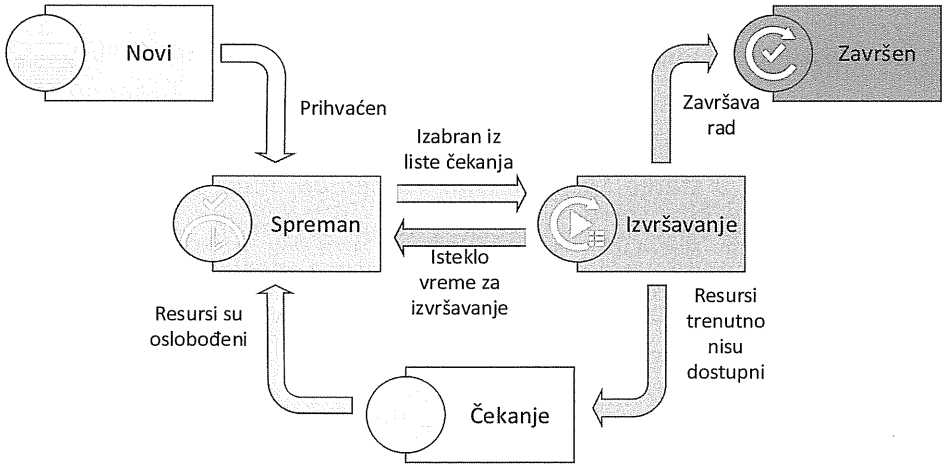
Operativni sistemi definišu skup stanja u kojima se, u zavisnosti od trenutne aktivnosti i situacije u sistemu, procesi mogu naći. Većina sistema podržava sledeća osnovna stanja:

- *Novi* – proces je upravo kreiran, operativni sistem je napravio dokumentaciju koja se odnosi na njega i on prelazi u spremno stanje.
- *Spreman* – proces čeka da operativni sistem donese odluku da mu bude dodeljen procesor.
- *Izvršavanje* – proces se izvršava na procesoru.
- *Čekanje* – proces se izvršavao ali je za njegov dalji rad potreban neki resurs koji nije slobodan tako da on čeka da se stvore uslovi da bi mogao da nastavi sa radom.
- *Završen* – proces je završio sa izvršavanjem i trebalo bi ga izbaciti iz sistema.

Na različitim sistemima nazivi stanja mogu varirati. Iz trenutnog stanja proces prelazi u neko drugo u sledećim situacijama (slika 2.3.):

1. Proces prelazi iz stanja *Spreman* u stanje *Izvršavanje* kada se procesor oslobodi a operativni sistem, po nekom kriterijumu, izabere njega iz liste spremnih procesa i dodeli mu procesor.

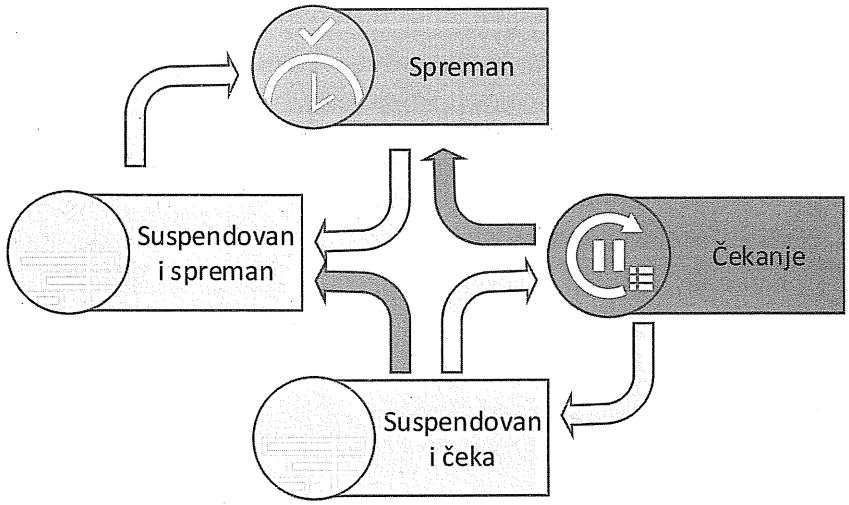
2. Proces prelazi iz stanja *Izvršavanje* u stanje *Čekanje* u trenutku kada su mu za dalje izvršavanje potrebni resursi koji trenutno nisu dostupni. Na primer, ako želi da odštampa rezultate, a štampač je u tom trenutku zauzet.
3. Proces prelazi iz stanja *Izvršavanje* u stanje *Spreman* kada istekne vreme koje mu je unapred određeno prilikom dodeljivanja procesora ili ako operativni sistem donese odluku da se proces prekine da bi neki drugi proces došao do procesora. U takvim situacijama proces ponovo prelazi u listu procesa koji konkurišu da dobiju procesor.
4. Proces prelazi iz stanja *Čekanje* u stanje *Spreman,* kada dođe do potrebnih resursa i spreman je za dalji rad. Pošto se u tom trenutku ne zna da li je procesor slobodan, proces se stavlja u listu spremnih procesa a operativnom sistemu se prepušta odluka kada će mu dozvoliti da ponovo dobije procesor.



Slika 2.3. Stanja procesa

Osim osnovnih stanja, operativni sistemi često podržavaju i suspendovana stanja u koje procesi mogu da pređu na zahtev korisnika ili operativnog sistema. Procesi koji su u stanju *Spreman* ili *Čekanje* se mogu suspendovati i oni onda prelaze u stanja *Suspendovan i spreman,* odnosno *Suspendovan i čeka* (slika 2.4.). Suspendovani proces oslobađa

resurse koje je zauzimao pre suspenzije i prestaje da konkuriše za druge resurse koji su mu potrebni za izvršavanje, ali i dalje ostaje proces. Često se suspendovani procesi prebacuju na disk do prestanka suspenzije, čime se oslobađa i deo radne memorije koji su zauzimali i na taj način ostavlja više prostora za druge procese.



Slika 2.4. Suspendovana stanja

Do suspendovanja spremnog procesa, odnosno do prelaska procesa iz stanja *Spreman* u stanje *Suspendovan i spreman,* najčešće dolazi zbog preopterećenosti sistema zbog velikog broja spremnih procesa, pa je privremeno suspendovanje nekog od procesa poželjno kako bi se operativnom sistemu olakšao rad. Takođe, do suspendovanja može doći i da bi se izbeglo zaglavljivanje sistema ili ukoliko korisnik želi da privremeno zaustavi izvršavanje procesa kako bi dobio mogućnost da proveri međurezultate.

Suspendovanje procesa koji je u stanju *Čekanje* se obično vrši da bi se oslobodili resursi kojima on raspolaže i time sprečilo zaglavljivanje ili ubrzao rad sistema. Kao što je već napomenuto, suspendovanjem procesa se oslobađaju i svi resursi koje je on posedovao čime se drugim procesima omogućava da do njih dođu.

Proces koji je u stanju *Suspendovan i čeka* prelazi u stanje *Suspendovan i spreman*, ako se stvore uslovi, tj. oslobode resursi na koje čeka (zbog kojih se našao u stanju *Čekanje*), tako da po povratku iz suspenzije može da bude u stanju *Spreman*.

Prekidanje suspenzije procesa koje inicira korisnik ili operativni sistem realizuje se njegovim prelaskom iz stanja *Suspendovan i čeka* u stanje *Čekanje* i iz stanja *Suspendovan i spreman* u stanje *Spreman*.

2.3. KONTROLNI BLOK PROCESA

Da bi operativni sistemi bili u mogućnosti da vode preciznu evidenciju o procesima, kao i da bi se olakšala implementacija multiprogramiranja, obično se koriste strukture podataka koje čuvaju informacije o svakom pokrenutom procesu. Ova dinamička struktura se često naziva *Kontrolni blok procesa* i sadrži najznačajnije podatke koji omogućavaju identifikaciju i upravljanje procesima.

U većini implementacija operativnih sistema kontrolni blok procesa sadrži sledeće informacije o procesima:

- *Jedinstveni identifikator procesa* (*Process ID – PID*) – broj koji proces dobija u trenutku pokretanja, tj. svog nastanka. Svi procesi na jednom sistemu imaju različite PID-ove, čime je sistemu omogućeno da ih razlikuje na krajnje jednostavan način.
- *Stanje procesa* – informacija o trenutnom stanju u kojem se proces nalazi. Na primer, proces se može naći u nekom od stanja koja su definisana u prethodnom odeljku: *Spreman, Izvršavanje, Čekanje, Suspendovan i spreman* itd.
- *Programski brojač* – čuva informaciju o sledećoj instrukciji koju proces treba da izvrši.
- *Sadržaj registara procesora* – vrednosti koje se nalaze u registrima kako bi proces posle gubitka prava da koristi procesor mogao da nastavi sa radom tamo gde je zaustavljen.

- *Prioritet procesa* – informacija o važnosti procesa u odnosu na ostale procese u sistemu.
- *Adresa memorije gde se nalazi proces* – pokazivači na adrese u memoriji gde se nalaze segmenti procesa.
- *Adrese zauzetih resursa* – kako bi operativni sistem imao informaciju o tome na kojim lokacijama treba da traži potrebne podatke.

Tabela 2.1. Klase podataka u kontrolnom bloku procesa

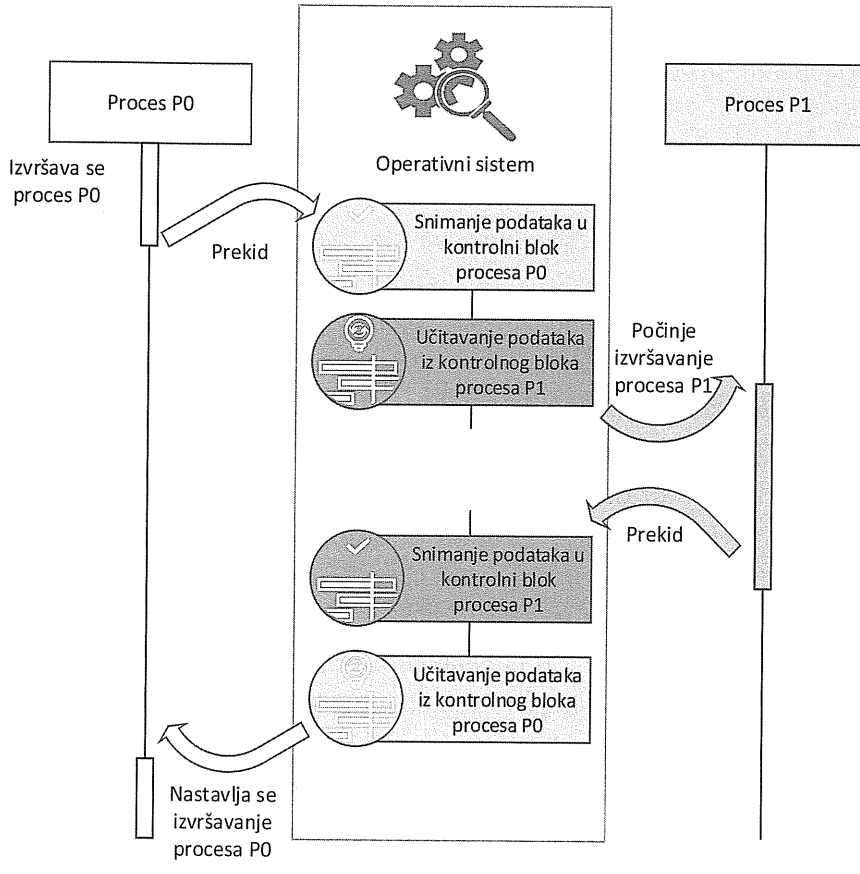
Procesi	Memorija	Fajlovi
PID	Pokazivač na tekst segment	Koreni direktorijum
Programski brojač	Pokazivač na segment podataka	Radni direktorijum
Registri	Pokazivač na stek segment	Fajl deskriptori
Status procesa	Pokazivač na hip	ID korisnika
Pokazivač na stek		ID grupe
Prioritet procesa		
Grupa kojoj pripada proces		
Vreme pokretanja procesa		
Signali		

U tabeli 2.1. mogu se videti neke od klasa podataka o kojima sistem može voditi računa. Generalno, polja koja čine kontrolni blok procesa se mogu razvrstati u one koji se odnose na procese, memoriju i fajlove.

Operativni sistemi obezbeđuju efikasne mehanizme za manipulaciju kontrolnim blokovima procesa kako bi se realizovali sledeći zadaci:

- Kreiranje kontrolnog bloka za novi proces;
- Uništavanje kontrolnog bloka procesa koji se izvršio;
- Menjanje stanja procesa;
- Menjanje prioriteta procesa;
- Izbor procesa za izvršavanje.

Prebacivanje konteksta predstavlja postupak kojim se proces koji se trenutno izvršava na procesoru prekida, pamte njegovi parametri, a zatim se umesto njega pokreće neki drugi proces i pri tome se učitavaju parametri drugog procesa.



Slika 2.5. Prebacivanje konteksta

Na slici 2.5. ilustrovan je primer prebacivanja (zamene) konteksta. Proces P_0 je u stanju izvršavanja, ali bi trebalo da se prekine i da proces P_1 krene da se izvršava. Prvo se stanje P_0 upisuje u odgovarajući kontrolni blok, zatim se učitava kontrolni blok novog procesa i on može da nastavi sa izvršavanjem. Programski brojač procesa P_1 dohvata sledeću instrukciju i on počinje da se izvršava. Kada se, iz nekog razloga donese odluka da postoji potreba da se P_1 prekine, a proces P_0 nastavi sa izvršavanjem,

upisuju se podaci o izvršavanju P_1 u njegov kontrolni blok, a učitavaju se podaci procesa P_0.

Modul koji je odgovoran da izvršava prebacivanje konteksta često se naziva *dispečer* (*Dispatcher*). On je zadužen za punjenje registara procesa, prebacivanje u korisnički režim rada i skok na odgovarajuću lokaciju u korisničkom programu kako bi se on nastavio.

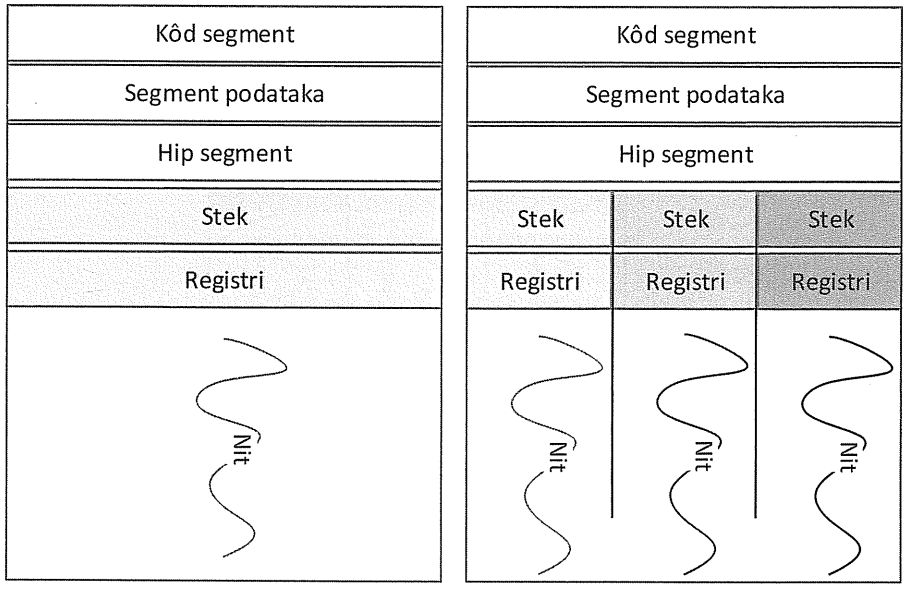
2.4. Niti

Tradicionalni pristup podrazumeva da procesi imaju svoj memorijski prostor, resurse i da imaju samo jednu jedinicu izvršavanja, tj. da se svi poslovi procesa izvršavaju sekvencijalno. Savremeni operativni sistemi obično podržavaju koncept *niti* (*threads*) koje predstavljaju osnovne jedinice za izvršavanje u okviru procesa. Niti su delovi jednog procesa i izvršavaju se korišćenjem resursa koji su njemu pridruženi. Osim resursa procesa kojem pripadaju, niti imaju i sopstvene resurse. Svaka nit poseduje svoje registre, programski brojač i stek, a razlikuje ih i jedinstveni identifikator (*thread ID – TID*). Kôd segment, segment podataka, podaci o otvorenim fajlovima itd. zajednički su za sve niti jednog procesa (slika 2.6.). Ovakvim pristupom smanjuje se zauzeti prostor i omogućava se da više niti obavlja različite zadatke u okviru jednog procesa. Takođe, korišćenjem niti pruža se mogućnost da se bolje iskoriste prednosti koje donosi višeprocesorska arhitektura. *Rad sa više niti* (*multithreading*) podrazumeva mogućnost operativnog sistema da podrži konkurentno ili paralelno izvršavanje više niti.

Na početku izvršavanja svaki proces dobija svoj memorijski prostor i kontrolnu (inicijalnu) nit. Ova nit ima zadatak da obavi potrebne inicijalizacije i kreira ostale niti koje su potrebne za izvršavanje procesa. S obzirom da niti imaju sve karakteristike procesa, pri čemu neke resurse dele sa drugim nitima, često se nazivaju i lakim (*lightweight*) procesima.

Upotreba niti može se ilustrovati na primeru veb pregledača. Naime, oni se mogu implementirati tako da koriste bar tri niti. Jedna nit služi za

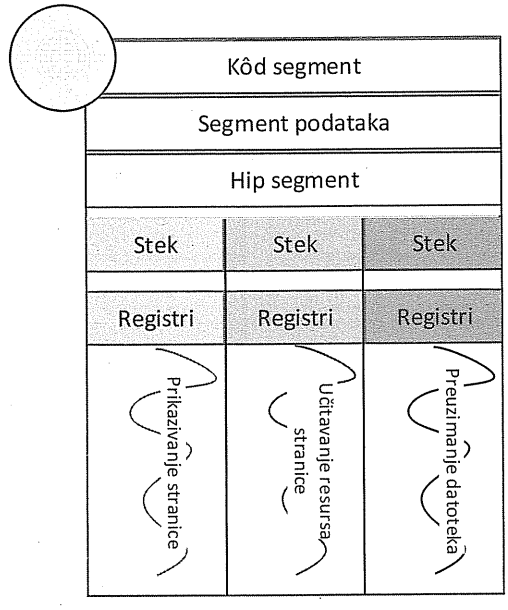
prikazivanje hiperteksta u okviru prozora, druga učitava podatke sa nekog servera, dok treća služi za preuzimanje podataka preko mreže (slika 2.7.). Pri tome, ove niti dele podatke kroz zajedničke resurse i istovremeno se mogu izvršavati na različitim jezgrima istog procesora. Slično, tekst procesori mogu imati nit koja prati pisanje teksta, nit koja je zadužena da u određenim vremenskim intervalima sačuva dokument nezavisno od ostalih aktivnosti i nit koja proverava pravopis.



Slika 2.6. Jedna nit procesa i više niti istog procesa

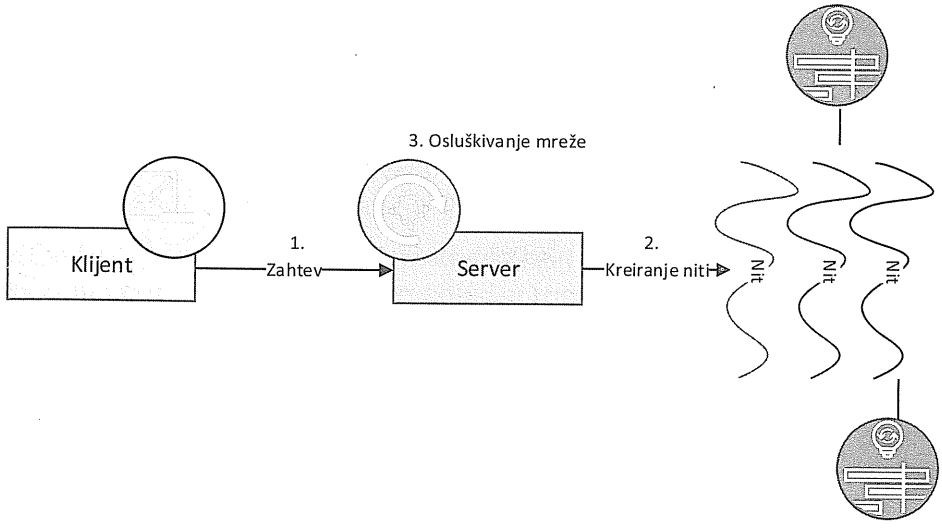
Korišćenje niti donosi mnoge prednosti. Prvenstveno omogućavaju značajne uštede memorijskog prostora i vremena. Niti dele memoriju i neke resurse koji pripadaju istom procesu tako da zauzimaju manje prostora nego kada su u pitanju nezavisni procesi. Takođe, iz istog razloga, niti se kreiraju mnogo brže od procesa, a i prebacivanje konteksta između niti istog procesa je brže od prebacivanja konteksta između procesa jer se prebacuju samo resursi koji su jedinstveni za nit – stek, registri i programski brojač. Niti pružaju mogućnost aplikacijama da nastave rad u situacijama kada se izvršavaju dugotrajne operacije koje bi, bez podele poslova procesa na niti, privremeno zaustavile izvršavanje

ostalih delova procesa. Slično, korišćenjem niti omogućava se rad procesa čiji su delovi potpuno blokirani. Razvojem i ekspanzijom višeprocesorskih sistema do izražaja naročito dolaze prednosti koje donosi korišćenje ovog koncepta jer više niti mogu da se izvršavaju istovremeno.



Slika 2.7. Niti veb pregledača

Na primeru serverske aplikacije, koja ima zadatak da opsluži veliki broj klijenata (klijentskih aplikacija) u kratkom vremenskom intervalu, mogu se dobro ilustrovati pogodnosti koje donosi korišćenje niti. Tradicionalni pristup, bez korišćenja niti, podrazumevao je da serverski proces u petlji čeka da se pojavi klijent, a onda se za svakog novog klijenta kreira novi proces kojem će prepustiti opsluživanje klijenta. Korišćenjem niti, problem se rešava na sličan način, ali se umesto različitih procesa koristi samo jedan u okviru kojeg se može kreirati veliki broj niti. Jedna nit procesa je zadužena da „osluškuje" mrežu, tj. da prihvata klijente pri čemu se za svakog novog klijenta kreira nova nit (slika 2.8.). Kreiranje niti je brže i zahteva manje prostora nego kada su u pitanju procesi, tako da ovakav pristup omogućava da se mnogo brže odgovori na zahteve, ali i da se opsluži mnogo veći broj klijenata.



Slika 2.8. Niti – serverska aplikacija

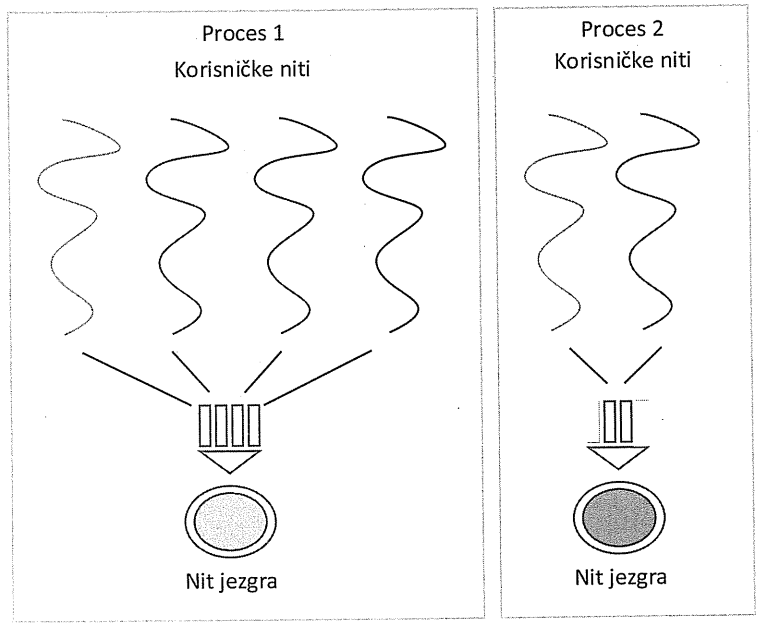
U zavisnosti od toga da li se nitima upravlja sa korisničkog ili sistemskog nivoa, niti se nazivaju korisničke ili niti jezgra. Pri tome, pristup procesoru i priliku da se izvršavaju imaju samo niti jezgra, tako da je potrebno napraviti odgovarajuću korespondenciju između korisničkih i niti jezgra. Ovo se postiže preslikavanjem (mapiranjem) korisničkih u niti jezgra. Najčešće podržana preslikavanja su:

- Preslikavanje više u jednu;
- Preslikavanje jedna u jednu;
- Preslikavanje više u više.

Preslikavanje više u jednu (*many-to-one*) podrazumeva da se više korisničkih niti, sve koje pripadaju jednom procesu, preslikaju u jednu nit jezgra (slika 2.9.). Broj niti jezgra je u ovom slučaju jednak broju procesa koji postoje u sistemu. Dakle, jezgro manipuliše isključivo sa procesima, tako da sve niti jednog procesa izvršava kroz jednu nit jezgra. Kod ovakvog pristupa, nitima se upravlja iz korisničkog režima bez uticaja jezgra operativnog sistema. Dakle, odluke o tome koja će se korisnička nit izvršavati donose se na korisničkom nivou. Međutim, jezgro operativnog sistema i dalje upravlja procesima i donosi odluke koja će se od niti jezgra izvršavati na procesoru. Obično se kreiranje, raspoređivanje i upravljanje

korisničkim nitima omogućava implementacijom posebnih biblioteka čija je osnovna funkcija da na korisničkom nivou pruže podršku za rad sa nitima.

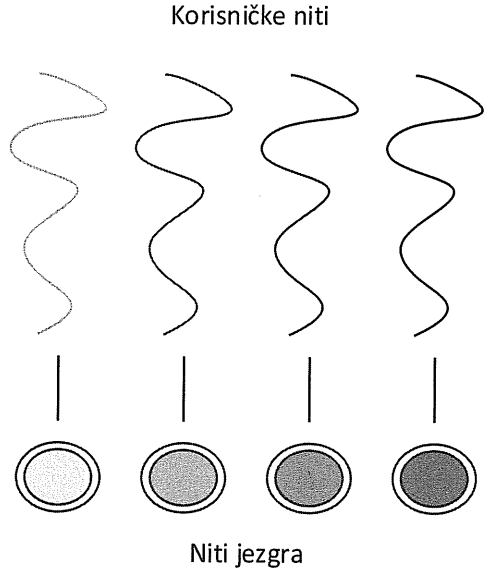
Glavni nedostatak ovakvog pristupa je u tome što, u situacijama kada se neka od korisničkih niti blokira, blokira se i odgovarajuća nit jezgra, tako da dolazi do blokiranja celog procesa, tj. svih niti koje ga čine. Pošto samo jedna nit može pristupiti jezgru u jednom trenutku, ovakvim pristupom ne mogu da se iskoriste prednosti višeprocesorske arhitekture.



Slika 2.9. Preslikavanje više korisničkih niti u jednu nit jezgra

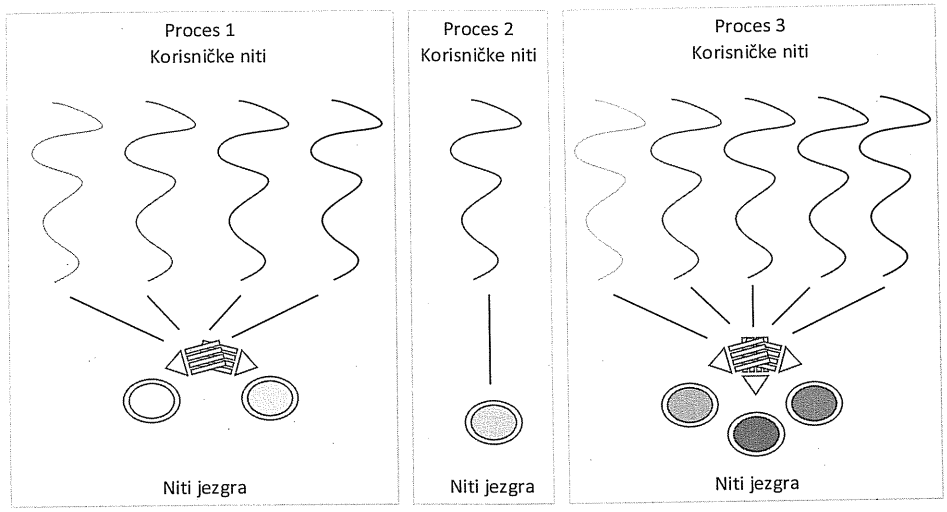
Preslikavanje jedna u jednu (*one-to-one*) podrazumeva da se svaka korisnička nit preslika u jednu nit jezgra (slika 2.10.). Pri tome, upravljanje nitima potpuno se prepušta jezgru operativnog sistema. Niti jezgra se kreiraju sporije i ima ih manje ali, za razliku od korisničkih niti, jezgro ima bolju podršku za rad sa ovim nitima, a nad njima ima i veći stepen kontrole. Ovakvim pristupom se rešavaju glavni problemi prethodnog modela. Naime, obezbeđuje se konkurentnije izvršavanje niti, jer postoji mogućnost da druge niti istog procesa, nastave aktivnosti u slučaju kada se jedna od njih blokira. Takođe, omogućava se da se više niti jezgra

izvršavaju paralelno na sistemima sa više procesora. Najveća mana ovakvog pristupa je ograničavanje broja niti jezgra a samim time i broja niti u korisničkom delu. Naime, zbog toga što se nezanemarljivo malo vremena i prostora troši pri stvaranju i održavanju niti jezgra, mnogi operativni sistemi imaju ograničenje dozvoljenog broja ovakvih niti.



Slika 2.10. Preslikavanje jedne korisničke niti u jednu nit jezgra

Preslikavanje više u više (*many-to-many*) je kombinacija (hibrid) prethodna dva pristupa jer podrazumeva da se, u zavisnosti od potreba procesa i mogućnosti sistema, korisničke niti jednog procesa preslikavaju u manji (ili isti u ekstremnom slučaju) broj niti jezgra (slika 2.11.). Broj niti, kao i način raspoređivanja korisničkih na niti jezgra, određuje se na korisničkom nivou, dok se upravljanje prepušta jezgru operativnog sistema. Broj niti jezgra koje će biti kreirane zavisi od konkretnog procesa, ali često i od broja procesora u sistemu. Naime, u sistemima sa više procesora prirodno je da se korisničke niti procesa preslikaju u više niti jezgra. Ovo je najkompleksniji model za implementiranje, ali je i najkvalitetniji jer se može prilagoditi potrebama procesa kao i karakteristikama sistema.

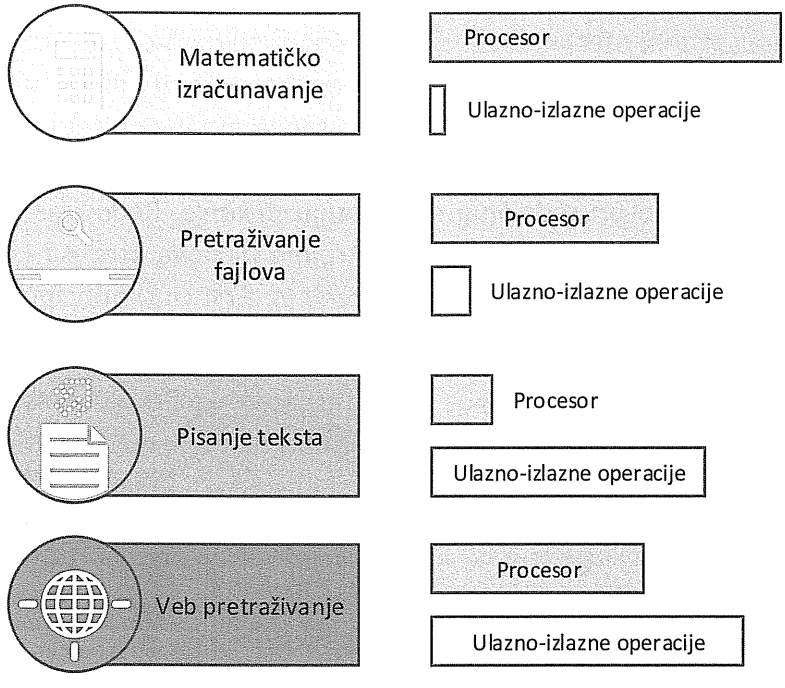


Slika 2.11. Preslikavanje više korisničkih niti u više niti jezgra

2.5. Redovi procesa

Jedan od najvažnijih zadataka operativnih sistema je da maksimalno povećaju efikasnost kada je izvršavanje procesa u pitanju. Između ostalog, ovo podrazumeva i da se procesor iskoristi na što bolji način, odnosno da se na njemu stalno izvršava neki od procesa, ali i da se pri tome procesi smenjuju. Prethodno navedeno, u stvari, predstavlja suštinu koncepta multiprogramiranja. Izvršavanje procesa se obično sastoji od naizmeničnog korišćenja procesora i čekanja na ulazno-izlazne operacije. Pri tome, da procesor ne bi bio besposlen tokom ulazno-izlaznih operacija procesa koji se izvršava na njemu, nekom drugom procesu se dozvoljava da koristi procesor. Kada procesi obave ulazno-izlazne operacije, onda im je ponovo potreban procesor do sledeće ulazno-izlazne operacije i tako do kraja izvršavanja. Na ovaj način se stvara utisak da se procesi izvršavaju paralelno, mada je pseudoparalelno adekvatniji izraz za opisanu situaciju. Naravno, ovakvo izvršavanje procesa može biti i paralelno u situacijama kada procesor ima više jezgara ili kada u sistemu postoji više procesora.

Važno je napomenuti i da neki procesi više vremena troše na ulazno-izlazne operacije, dok drugi najviše vremena u toku svog izvršavanja koriste procesor (slika 2.12.).

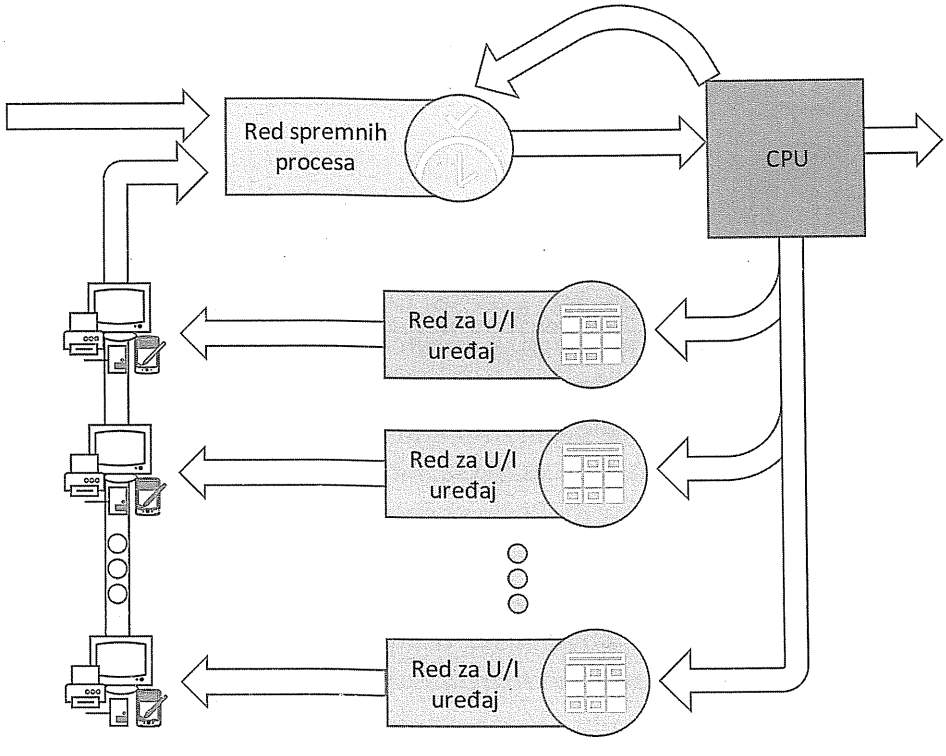


Slika 2.12. Potrošnja vremena različitih procesa

Imajući u vidu činjenicu da postoje različite potrebe procesa kada su procesor i ulazno-izlazni uređaji u pitanju, operativni sistemi obično od svih procesa koji bi trebalo da se izvrše, biraju podskup koji će se odmah učitati u memoriju i izvršavati. Posle početne selekcije, operativni sistem ima zadatak da određuje redosled kojim procesi dobijaju procesor i ostale uređaje i koliko vremena mogu da koriste dobijene resurse. Poslovi ovakvog tipa se obično poveravaju posebnim modulima operativnog sistema koji se nazivaju *planeri* (*schedulers*).

Da bi se olakšalo upravljanje procesima, uobičajeno je da se formira nekoliko redova procesa. Početni red, u koji se smeštaju pokrenuti procesi, naziva se *red poslova.* Ovaj red sadrži skup svih procesa u sistemu. *Red spremnih procesa* sadrži procese koji su izabrani po nekom

kriterijumu koji je određen na nivou operativnog sistema. Procesi u ovom redu su spremni za izvršavanje (nalaze se u stanju *Spreman*) i smešteni su u glavnu memoriju. Izbor procesa koji će ući u red spremnih trebalo bi da zavisi i od njihovih potreba, odnosno od procene da li će u toku svog izvršavanja više koristiti procesor ili će više vremena provoditi u radu sa ulazno-izlaznim uređajima, jer je poželjno da se napravi dobra selekcija procesa kako bi sistem bio produktivniji. Operativni sistemi vode računa i o redovima procesa koji čekaju na neke od uređaja (slika 2.13). Procesi koji se nalaze u ovim redovima su u stanju *Čekanje*. Redovi se obično implementiraju kao povezane liste kontrolnih blokova procesa koje ne moraju uvek biti organizovane po *FIFO* (*First In, First Out*) principu.

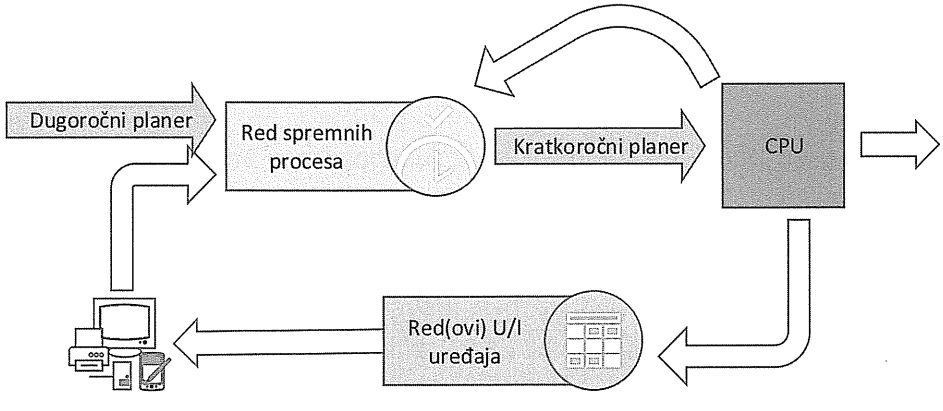


Slika 2.13. Redovi procesa koji čekaju na uređaje

2.6. Planeri

Kao što je već napomenuto, zadatak operativnog sistema je da omogući što veću produktivnost sistema, odnosno da organizuje kretanje procesa kroz različite redove i time omogući njihovo što brže izvršavanje. Odabir procesa i njihovo raspoređivanje po redovima, ali i organizaciju čekanja vrše planeri. Obično postoje bar dve vrste planera: dugoročni i kratkoročni (slika 2.14.). Uz njih se često implementira i srednjoročni planer.

Funkcija *dugoročnih planera* je da od skupa svih procesa koji bi trebalo da se izvrše (iz Reda poslova) izaberu one koji će da se aktivno uključe u sistem i počnu sa izvršavanjem. Drugim rečima, ovi planeri bi trebalo da naprave dobar odabir procesa za red spremnih procesa. S druge strane *kratkoročni planeri* imaju zadatak da donose odluke o tome koji će se od spremnih procesa izvršavati i koliko dugo će dobiti procesor. Kratkoročni planer se često naziva planer procesora.

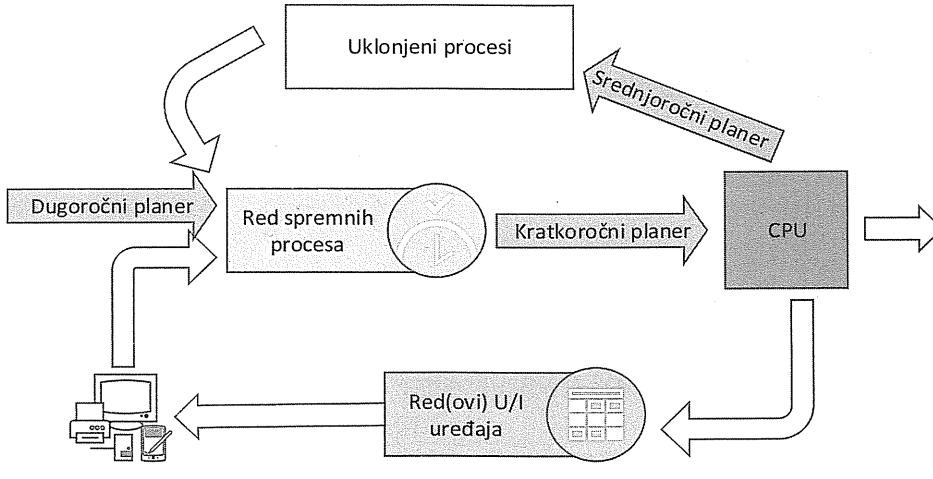


Slika 2.14. Dugoročni i kratkoročni planer

Zbog njihove funkcije u sistemu, kratkoročni planeri se često pozivaju, pa je potrebno da budu implementirani tako da što brže donose odluke. Sa druge strane, dugoročni planeri se ne pozivaju učestalo kao kratkoročni, tako da se može dozvoliti njihov nešto sporiji rad. U praksi, za dugoročne planere je važnije da naprave dobru kombinaciju poslova (strategiju) nego

da uštede na vremenu, dok je kod kratkoročnih planera brzina rada važnija od optimalnosti donete odluke.

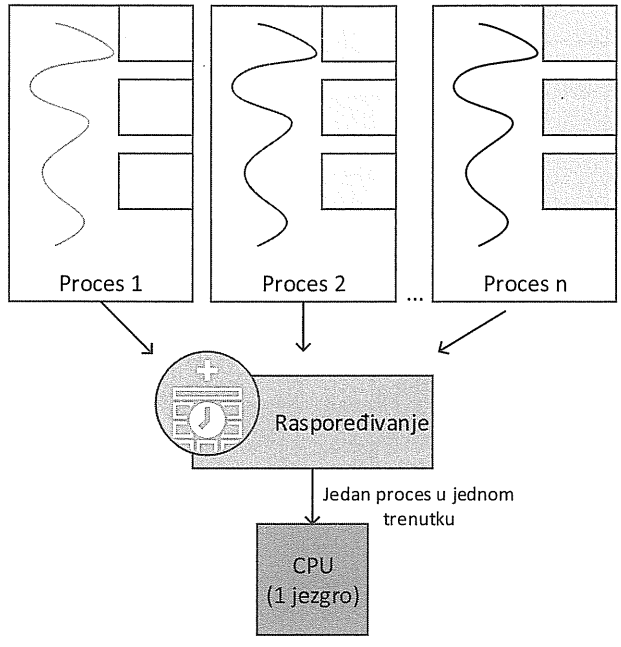
Ponekad se događa da je skup izabranih procesa u memoriji takav da jedni druge ometaju i zaglavljuju, čime umanjuju efikasnost celokupnog sistema. U takvim situacijama je, često, korisnije ukloniti (suspendovati) neke od procesa iz memorije, nego forsirati viši stepen multiprogramiranja nauštrb efikasnosti. Posle izvesnog vremena, kada se steknu uslovi za to, uklonjeni procesi se mogu ponovo vratiti u memoriju. Ova šema se naziva *prebacivanje* (*swapping*), a izvodi se sa ciljem da bi se poboljšala kombinacija izabranih poslova ili da bi se oslobodila memorija u slučaju zagušenja. Za zadatke ovakvog tipa, obično se na nivou operativnog sistema implementira poseban planer koji se naziva *srednjoročni planer* (slika 2.15.).



Slika 2.15. Srednjoročni planer

2.7. VIŠEPROCESORSKI SISTEMI

Višeprocesorski sistemi dozvoljavaju veću fleksibilnost kada je rad sa procesima i nitima u pitanju. Naime, planeri operativnih sistema koji podržavaju samo procese i izvršavaju se na jednoprocesorskim sistemima (slika 2.16.), imaju zadatak da vrše odabir procesa koji će se izvršavati i pri tome određuju trajanje njihovih aktivnosti na procesoru.

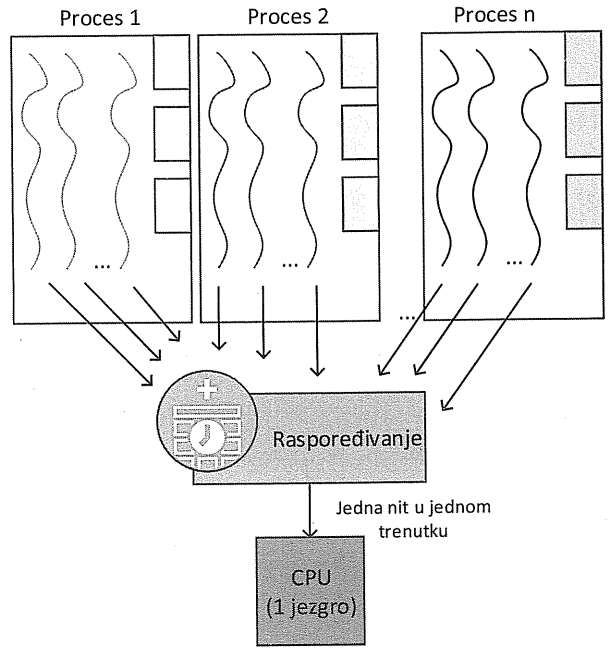


Slika 2.16. Raspoređivanje procesa na jednoprocesorskom sistemu

Ukoliko operativni sistemi podržavaju koncept niti i izvršavaju se na računaru sa jednim procesorom onda, slično kao i u prethodnom primeru, niti se mogu smenjivati stvarajući privid da se izvršavaju paralelno (slika 2.17.).

U sistemima sa više procesora, procesi se mogu izvršavati na različitim procesorima (jezgrima) paralelno, što sistem čini efikasnijim u odnosu na prethodno pomenute (slika 2.18).

Operativni sistemi koji podržavaju rad sa nitima, a izvršavaju se na višeprocesorskim sistemima (slika 2.19.) su najefikasniji, ali i najosetljiviji jer zahtevaju pažljivo planiranje. Naime, potrebno je obezbediti sinhronizaciju niti istog procesa koje se mogu izvršavati na različitim procesorima. Pri tome, treba voditi računa o tome koliko koja nit (i kada) treba da dobije mogućnost da se izvršava i omogućiti njihovu sinhronizaciju u situacijama kada nekoliko niti istog procesa treba da izvrši zadatke u određenom redosledu.

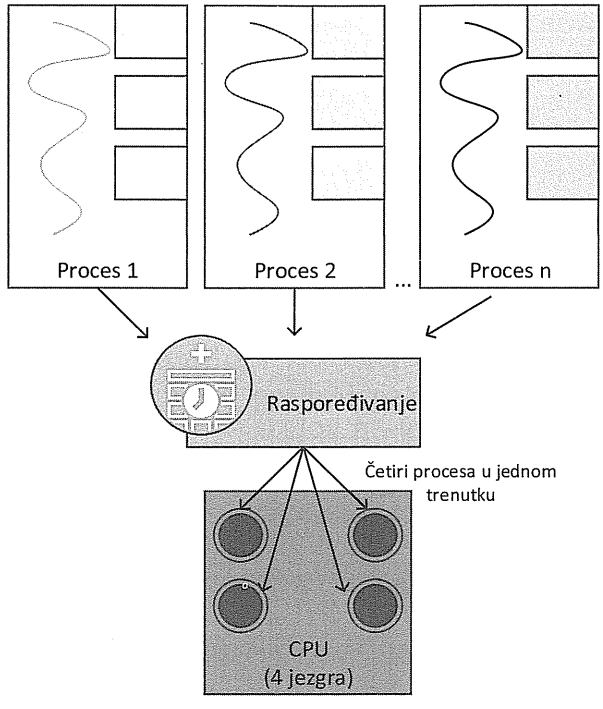


Slika 2.17. Raspoređivanje niti na jednoprocesorskim sistemima

Procesori savremenih računarskih sistema često imaju više jezgara, a u nekim situacijama i sistemi mogu imati više nezavisnih procesora. U daljem tekstu pod pojmom procesor podrazumevaće se različita jezgra jednog procesora, osim ako se ne naglasi drugačije. Višeprocesorski sistemi omogućavaju veću efikasnost sistema, ali zahtevaju kompleksnije algoritme za raspoređivanje. Kada su u pitanju metode raspoređivanja procesa na različite procesore, postoje dva pristupa:

- Simetrično multiprocesiranje;
- Asimetrično multiprocesiranje.

Simetrično multiprocesiranje podrazumeva da su procesori ravnopravni i da se procesi raspoređuju na bilo koji od njih u zavisnosti od primenjenog algoritma za raspoređivanje. Kod ovakvog pristupa svi procesori dele zajedničku memoriju, svi imaju pristup ulazno-izlaznim uređajima a operativni sistem ih tretira kao ravnopravne.

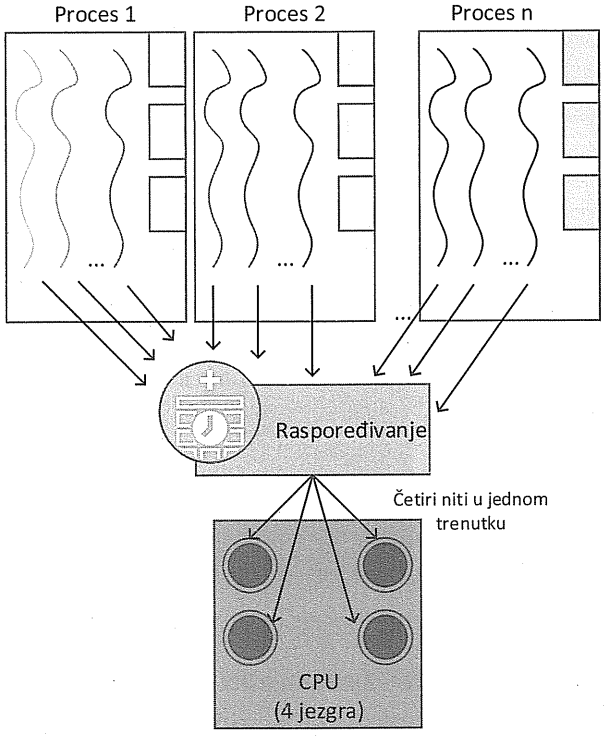


Slika 2.18. Raspoređivanje procesa na višeprocesorskom sistemu

Asimetrično multiprocesiranje se zasniva na ideji da neki procesori mogu biti zaduženi za određene funkcije. Na primer, jedan procesor se može proglasiti glavnim (nadređenim). Njemu se poverava izvršavanje koda jezgra operativnog sistema a samim time i vođenje računa o ulazno-izlaznim operacijama. Kod takve podele ostali procesori imaju zadatak da izvršavaju isključivo korisničke procese dodeljene od strane glavnog procesora. Sistemi koji podržavaju ovakav pristup su često neefikasni jer se dosta sporih operacija izvršava na jednom (glavnom) procesoru. Iz tog razloga, simetrično multiprocesiranje pokazalo se kao efikasnije u realnoj primeni.

Pri raspoređivanju procesa u višeprocesorskim sistemima mogu se implementirati rešenja koja podrazumevaju jedan zajednički red čekanja procesa za sve procesore ili različite redove čekanja procesa za svaki od

procesora. Kod oba pristupa, planeri vode računa o spremnim procesima i biraju sledeći koji će se izvršavati.



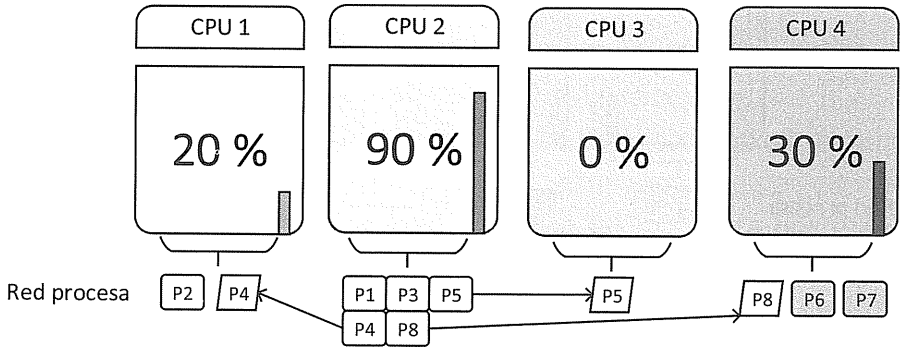
Slika 2.19. Raspoređivanje niti na višeprocesorskom sistemu

Pristup sa zajedničkim redom čekanja omogućava ravnomerno opterećenje procesora. Naime, pošto neki procesor postane slobodan, može mu se dodeliti sledeći proces za obradu. Ovaj pristup se relativno jednostavno implementira, ali pri tome treba voditi računa da se spreči da dva procesora dobiju isti proces za izvršavanje. Takođe, pristup sa zajedničkim redom sa povećanjem broja procesa može dovesti do stvaranja uskog grla.

Kod pristupa sa različitim redovima čekanja za svaki procesor može se dogoditi da procesi čekaju u redovima, a da pri tome postoje slobodni (besposleni) procesori. Ovakve situacije nisu poželjne i treba ih izbegavati a to se obično postiže balansiranjem opterećenja (*load balancing*).

Balansiranje opterećenja je postupak kojim se teži da se što ravnomernije podele poslovi između procesora. Najčešće se primenjuju dva načina balansiranja opterećenja: prenošenje migracije (*push migration*) i preuzimanje migracije (*pull migration*).

Prenošenje migracije podrazumeva da se iz reda procesa opterećenog procesora neki od procesa pošalju u redove besposlenih ili manje zauzetih procesora (slika 2.20.). Za potrebe prenošenja procesa potrebno je implementirati algoritam za određivanje opterećenja svih procesora, pronalaženje najmanje zauzetog procesora, izbor procesa koji će migrirati itd. Ovaj algoritam se može pozivati nakon što opterećenje nekog procesora pređe određenu granicu ili u određenim (unapred zadatim) vremenskim intervalima. U svakom slučaju treba napomenuti da vremenska složenost ovakvog algoritma nije zanemarljivo mala.

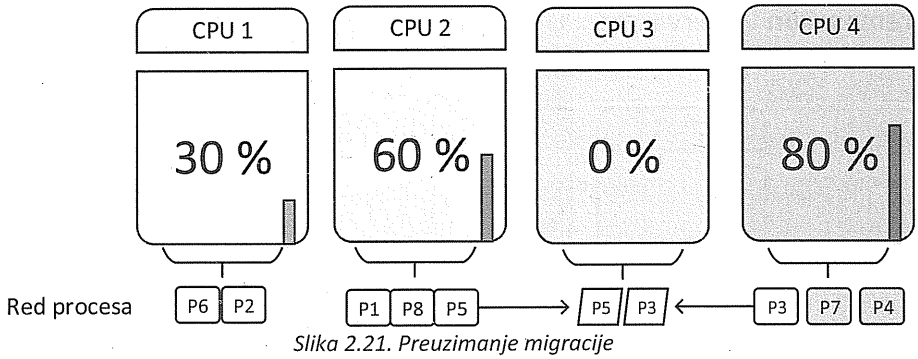


Slika 2.20. Prenošenje migracije

Preuzimanje migracije podrazumeva da se u red slobodnijeg procesora prebace procesi iz redova zauzetijih procesora (slika 2.21.). Za ovakav pristup je potrebno implementirati algoritam za pronalaženje slobodnijih i najzauzetijih procesora, kao i algoritam za odabir koji proces treba preuzeti. Obično se proces za migraciju bira od onih koji su visokog prioriteta ili procesa koji su se malo vremena izvršavali.

Balansiranje opterećenja obično povlači da se proces u celosti ne izvrši na jednom procesoru čime se povećava vremenska složenost izvršavanja. Naime, prilikom prebacivanja procesa sa jednog na drugi procesor, mora se isprazniti memorija prvog procesora a sadržaj premestiti na drugi što predstavlja izgubljeno vreme. Iz tog razloga procesu se može dodeliti *afinitet* (sklonost) ka određenom procesoru (*processor affinity*). Obično

se afinitet razvija prema procesoru na kojem je proces počeo da se izvršava. Afinitet procesa prema procesoru može biti slab (*soft*) ili jak (*hard*). *Slab afinitet* podrazumeva da operativni sistem omogućava ali i ne garantuje da će proces nastaviti izvršavanje na istom procesoru. Na primer, proces koji je počeo sa izvršavanjem na jednom procesoru može se prebaciti na drugi (u slučaju da je taj pogodniji) i razviti afinitet prema njemu. *Jak afinitet* podrazumeva da sistem dopušta da se procesu precizno odredi podskup procesora na kojima može da se izvršava. Mnogi sistemi podržavaju obe vrste afiniteta. Afiniteti su u direktnoj suprotnosti sa balansiranjem, tako da pri projektovanju sistema treba voditi računa o tome koji od ova dva pristupa treba da ima prednost.



Slika 2.21. Preuzimanje migracije

2.8. Raspoređivanje procesa

Pod raspoređivanjem procesa se podrazumeva donošenje odluka o toku izvršavanja (promeni stanja) procesa koji se nalaze u memoriji. Raspoređivanje bi trebalo da bude što bliže optimalnom, jer od njega zavisi efikasnost sistema. Dobro organizovano raspoređivanje procesa može u značajnoj meri poboljšati brzinu i produktivnost sistema.

Kod jednoprocesorskih sistema u jednom trenutku se tačno jedan proces može izvršavati, dok ostali spremni procesi čekaju da se procesor oslobodi i da budu određeni za izvršavanje. Kada su u pitanju višeprocesorski sistemi, primenjuju se slični mehanizmi za raspoređivanje, samo što sistem na raspolaganju ima više procesora, pa samim tim i više prostora za raspoređivanje. Dodatne mogućnosti pruža i rad sa nitima koje

omogućavaju da se delovi procesa izdvoje u posebne celine i izvršavaju pseoudoparalelno na jednom procesoru ili paralelno na višeprocesorskim sistemima.

Algoritmi prikazani u narednom delu će podrazumevati da se operativni sistem izvršava u okviru jednoprocesorskog sistema, a pojam procesa će se odnositi na jedan proces, jedan posao procesa (*task*) ili jednu nit, osim ako se drugačije ne naglasi.

Postoje različiti algoritmi raspoređivanja koji mogu favorizovati određene klase procesa u odnosu na druge, tako da je dobro poznavanje ovih algoritama bitno za odabir prave strategije pri dizajniranju planera operativnih sistema. Kvalitet algoritama se obično ocenjuje na osnovu kriterijuma kao što su:

- *Iskorišćenost procesora* – odnosi se na procenat koliko je procesor bio zaposlen u određenom vremenskom periodu.
- *Propusna moć* – mera koja predstavlja broj procesa koji se mogu završiti u jedinici vremena.
- *Vreme obrade* – vreme koje protekne od momenta pokretanja procesa do njegovog završetka. Konkretno, ovo vreme predstavlja sumu vremena koje proces provede u čekanju da se smesti u memoriju, čekanju u redu spremnih procesa, vremena tokom kojeg se izvršavao na procesoru i vremena tokom kojeg je obavljao ulazno-izlazne operacije.
- *Vreme čekanja* – ukupno vreme koje proces provede čekajući u redu spremnih procesa. Ovo vreme ne podrazumeva vreme izvršavanja procesa na procesoru kao i vreme koje je proces potrošio na ulazno-izlaznim uređajima.
- *Vreme odziva* – vreme koje protekne od prijavljivanja procesa do trenutka kada se proizvede prvi izlaz programa. Pri tome se ne računa i vreme potrebno da se taj izlaz prikaže na nekom od izlaznih uređaja. Ovo vreme obično daje veoma realnu sliku o situaciji u sistemu.

- *Količnik dužine vremena obrade i trajanja izvršavanja procesa.* Ovaj parametar može biti veoma koristan jer veoma dobro opisuje situaciju u sistemu.

Imajući u vidu prethodno definisane kriterijume, nameće se zaključak da se pod optimizacijom raspoređivanja procesa u sistemu podrazumeva pronalaženje rešenja koje odlikuje:

- Maksimalna iskorišćenost procesora;
- Maksimalna propusna moć;
- Minimalno vreme obrade;
- Minimalno vreme čekanja;
- Minimalno vreme odziva;
- Konvergencija količnika dužine vremena obrade i vremena izvršavanja procesa ka 1.

Najčešće je nemoguće istovremeno uticati na sve navedene parametre, tako da se, u zavisnosti od željenog rezultata, akcenat može staviti na neke od njih. Obično se za merenje kvaliteta rešenja koriste srednje vrednosti pomenutih parametara. Međutim, u nekim situacijama se akcenat stavlja na ekstremne vrednosti (globalne minimume i maksimume) nekih od parametara kako bi se rešenje stavilo u željene granice i usmerilo ka cilju.

2.9. Algoritmi planiranja

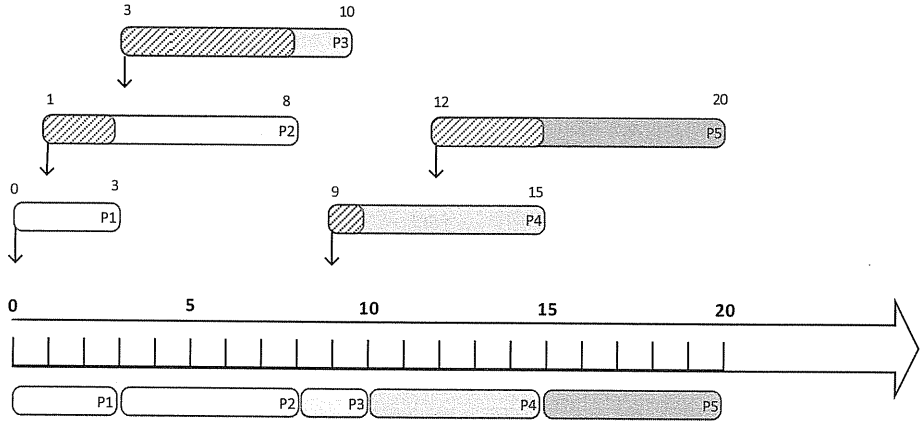
U narednom delu biće prikazani neki od tradicionalnih algoritama planiranja. Pri tome su polazne pretpostavke da sistem sadrži procesor sa jednim jezgrom. Takođe, važno je napomenuti da neki od algoritama imaju i verzije koje dozvoljavaju prekidanje procesa. *Prekidanje procesa* podrazumeva prebacivanje konteksta procesa koji se izvršava i učitavanje drugog procesa, na osnovu odluke planera, pre nego što je prvi proces iskoristio procesor onoliko koliko mu je bilo potrebno da obavi posao, da dođe do ulazno-izlaznih operacija ili da se izvrši do kraja. Ovakva mogućnost daje veću fleksibilnost pri upravljanju procesima.

2.9.1. FCFS algoritam

Algoritam za raspoređivanje procesa, koji se zbog svoje jednostavnosti i pravednosti obično prvi nameće je *FCFS* (*First Come, First Served*). Ovaj algoritam je zasnovan na ideji da se procesor dodeljuje procesima po redu kako su ga zatražili, tj. prijavili se za njegovo korišćenje. FCFS algoritam se jednostavno implementira korišćenjem FIFO (First In, First Out) strukture. Kada se proces pokrene i prijavi za korišćenje procesora, on se tada postavlja na kraj reda. U trenutku kada se procesor oslobodi, tada se prvi sledeći proces izbacuje iz reda i dobija pravo da koristi procesor. FCFS algoritam nema verziju koja bi dozvoljavala prekidanje jer bi to bilo u suprotnosti sa glavnim principom na kojem je zasnovan.

Prednost ovog algoritma je laka implementacija, ali je vreme čekanja često veliko. Takođe, ovakav pristup može dovesti do takozvanog „efekta konvoja", situacije kod koje dosta procesa čeka da se izvrši jedan vremenski zahtevan proces.

Na slici 2.22. je prikazan primer sistema u kojem se izvršava 5 procesa, dok je u tabeli 2.2. zadato vreme pristizanja procesa u sistem kao i vreme potrebno da se proces izvrši na procesoru.



Slika 2.22. FCFS algoritam

Tabela 2.2. sadrži i vrednosti početka i završetka izvršavanja procesa ako se primeni FCFS algoritam. Takođe, u tabeli se mogu videti i vremena obrade (T), čekanja (M) i količnik dužine vremena obrade i vremena izvršavanja (P) kada je ovaj primer u pitanju.

Tabela 2.2. FCFS algoritam

Proces	Vreme pristizanja	Zahtevano vreme	Vreme početka	Vreme završetka	T	M	P
P_1	0	3	0	3	3	0	1.0
P_2	1	5	3	8	7	2	1.4
P_3	3	2	8	10	7	5	3.5
P_4	9	5	10	15	6	1	1.2
P_5	12	5	15	20	8	3	1.6
Prosek					**6.2**	**2.2**	**1.74**

U tabeli 2.3. prikazan je primer reda koji sadrži tri mala i jedan veliki proces (P_2). Kao što ovaj primer pokazuje, procesima koji se dugo izvršavaju FCFS algoritam odgovara za razliku od procesa koji ne zahtevaju puno vremena, a prinuđeni su da čekaju da se pre njih izvrše neki koji se duže izvršavaju.

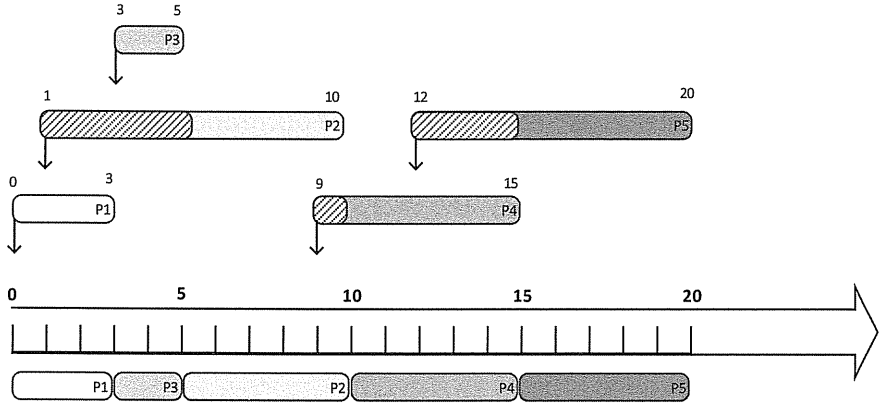
Tabela 2.3. Efekat konvoja

Proces	Vreme pristizanja	Zahtevano vreme	Vreme početka	Vreme kraja	T	M	P
P_1	0	1	0	1	1	0	1.0
P_2	0	100	1	101	101	1	1.01
P_3	0	1	101	102	102	101	102.0
P_4	0	1	102	103	103	102	103.0
Prosek					**76.75**	**51.0**	**51.75**

Ovaj algoritam se obično primenjuje na sistemima ili u situacijama koje ne dozvoljavaju prekidanje procesa već podrazumevaju da se procesima koji su dobili priliku da se izvršavaju na procesoru omogući da završe započeti posao bez obzira na situaciju u sistemu. Paketna (*Batch*) obrada je dobar primer primene FCFS algoritma u praksi.

2.9.2. SPF algoritam

SPF (*Shortest-Process-First*) algoritam se zasniva na ideji da se favorizuju procesi čiji je (sledeći) zahtev za procesorom (procesorskim vremenom) manji. Svakom procesu se pridružuje očekivano vreme za njegovo sledeće izvršavanje (posao) na procesoru, a procesor se dodeljuje redom procesima koji zahtevaju manje vremena. Ukoliko se dogodi da dva procesa imaju isto očekivano vreme, obično se primenjuje FCFS algoritam za odluku koji proces će dobiti prednost.



Slika 2.23. SPF algoritam

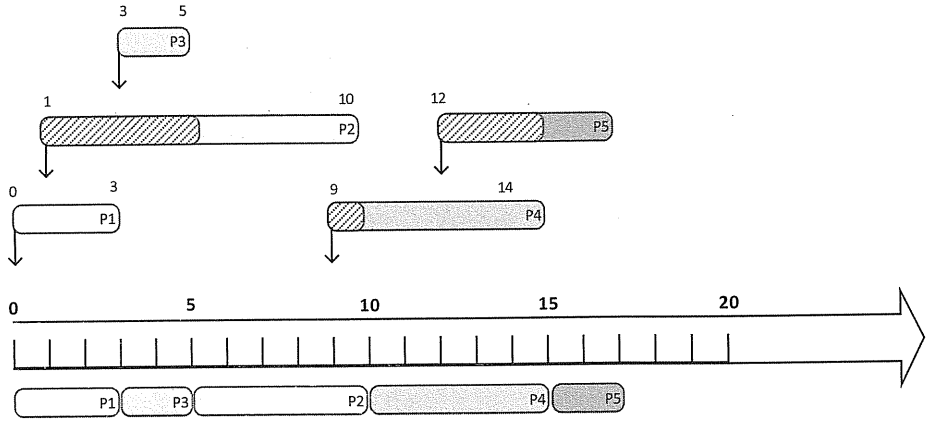
Tabela 2.4. Primer SPF algoritma

Proces	Vreme pristizanja	Zahtevano vreme	Vreme početka	Vreme kraja	T	M	P
P_1	0	3	0	3	3	0	1.0
P_2	1	5	5	10	9	4	1.8
P_3	3	2	3	5	2	0	1.0
P_4	9	5	10	15	6	1	1.2
P_5	12	5	15	20	8	3	1.6
Prosek					**5.6**	**1.6**	**1.32**

Na slici 2.23. može se videti ilustracija rada SPF algoritma, dok su u tabeli 2.4. prikazani početni parametri za 5 procesa koji se nalaze u sistemu u ovom primeru. U tabeli se mogu videti i vremena bitna za izvršavanje procesa kao i karakteristike predloženog algoritma.

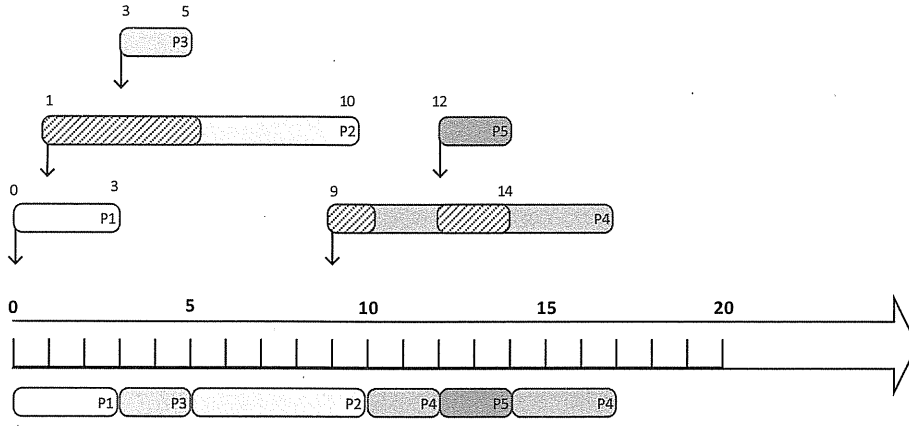
SPF algoritam se može implementirati u verzijama koje dozvoljavaju i koje ne dozvoljavaju prekidanje procesa koji se izvršavaju na procesoru. Razlika između ova dva pristupa je u tretmanu novih procesa koji imaju kraći sledeći zahtev za procesorskim vremenom u odnosu na proces koji se izvršava na procesoru. Naime, ukoliko se dozvoli prekidanje procesa, u situaciji kada se pojavi novi proces čija je sledeća operacija na procesoru vremenski manje zahtevna od one koja se trenutno izvršava, vrši se prebacivanje konteksta procesa koji se izvršava a novom procesu se dozvoljava da na procesoru izvrši operaciju.

Na slici 2.24. može se videti primer koji ilustruje rad SPF algoritma bez prekidanja. Kada se pojavi proces P_5 on će morati da sačeka da proces P_4 završi sa izvršavanjem, iako je njegovo vreme potrebno da se izvrši kraće. Naime, u trenutku kada se proces P_5 pojavi u sistemu sa potrebom da procesor koristi 2 vremenske jedinice, procesu P_4 preostaje još 3 vremenske jedinice da se izvrši do kraja.



Slika 2.24. SPF bez prekidanja

Ukoliko bi implementacija SPF algoritma dozvoljavala prekidanje, u situaciji koja je opisana u prethodnom primeru, proces P_5 bi prekinuo proces P_4 i preuzeo procesor, jer mu je manje vremenskih jedinica potrebno da se izvrši. Slika 2.25. ilustruje ovaj slučaj.



Slika 2.25. SPF sa prekidanjem

SPF algoritam je optimalan kada se posmatra srednje vreme čekanja za bilo koji skup poslova. Ovo tvrđenje je trivijalno dokazati, jer je očigledno da premeštanje kraćeg posla pre dužeg smanjuje vreme čekanja kraćeg posla više nego što povećava vreme čekanja dužeg.

Najveći problem, kada je ovaj algoritam u pitanju, predstavlja određivanje dužine sledećeg zahteva za procesorom, odnosno procena dužine trajanja sledeće aktivnosti procesa na procesoru (do ulazno-izlazne operacije ili završetka procesa). Kod dugoročnog planiranja, kao procena se može uzeti unapred zadata granica vremena potrebnog da se posao završi. U takvoj situaciji efikasnost sistema zavisi od mogućnosti da se realno proceni trajanje posla, jer manja vrednost očekivanog vremena potrebnog za njegovo izvršavanje povlači da se takvi poslovi favorizuju i da brže dođu na red za izvršavanje. Ovaj algoritam se teže primenjuje na kratkoročno planiranje jer nema pouzdanog načina da se precizno odredi trajanje sledeće aktivnosti procesa na procesoru. Pristup koji daje dosta dobre rezultate zasniva se na aproksimaciji vremena. Obično se za predviđanje dužine sledeće aktivnosti koristi *eksponencijalna sredina* prethodnih aktivnosti i prethodnih procena potrebnog vremena. Drugim rečima, koristi se pretpostavka da će sledeća aktivnost biti slične dužine kao prethodne i procenjene za taj proces.

Dužina trajanja sledeće aktivnosti procesa τ_{n+1} se može aproksimovati sledećim izrazom:

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

pri čemu je:

t_n – vreme koje je bilo potrebno da se izvrši n-ta aktivnost procesa na procesoru,

τ_n – očekivano vreme koje je bilo predviđeno za izvršavanje *n*-te aktivnosti procesa na procesoru.

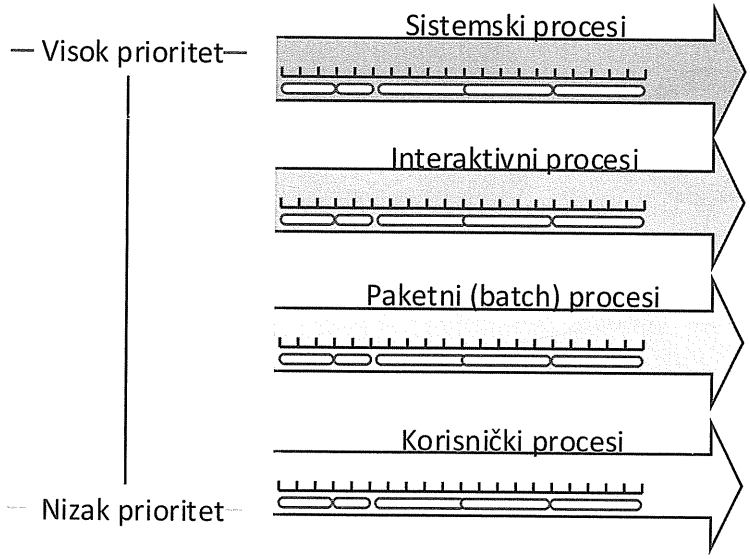
Parametar α reguliše uticaj procenjenog vremena i vremena koje je stvarno bilo potrebno za izvršavanje prethodne aktivnosti u proceni dužine sledeće aktivnosti. Praktično, ako α ima veću vrednost, tj. ako je bliža jedinici onda se više u obzir uzima prethodna procena a u suprotnom će pri novoj proceni prevagnuti stvarna dužina prethodne aktivnosti tog procesa. Ako je vrednost parametra 1/2 tada je aproksimativni izraz aritmetička sredina ove dve vrednosti.

2.9.3. Algoritam sa prioritetima

Algoritam sa prioritetima podrazumeva da se svakom procesu pridruži veličina koja predstavlja njegov prioritet i da se kao sledeći za izvršavanje bira onaj koji ima najviši prioritet (slika 2.26.). Ukoliko se dogodi da dva procesa imaju isti prioritet, bira se onaj koji je prvi došao u sistem. Iz prethodnog se može zaključiti da je SPF algoritam u stvari specijalan slučaj algoritma sa prioritetima gde se prioriteti procesa definišu tako da budu obrnuto proporcijalni očekivanom vremenu izvršavanja procesa čime se postiže da kraći procesi dobiju viši prioritet u odnosu na duže. Takođe, FCFS algoritam je specijalan slučaj algoritma sa prioritetima jer se vreme dolaska u sistem može uzeti kao prioritet pri rangiranju procesa.

Prioriteti se mogu definisati interno ili eksterno. Interno definisani prioriteti koriste neku merljivu veličinu za izračunavanje prioriteta procesa. Često se pri određivanju prioriteta u obzir uzimaju vremenske granice, memorijski zahtevi, broj otvorenih fajlova, prosečno trajanje

aktivnosti na procesoru, prosečno trajanje ulazno-izlaznih operacija kao i količnik dve poslednje navedene vrednosti. Na primer, procesi koji su više orijentisani ka ulazno-izlaznim operacijama bi trebalo da imaju viši prioritet u odnosu na procese koji više koriste procesor jer je njima procesor potreban na kraće vreme, a često blokirani čekaju na neki uređaj. U suprotnom, ovakvi procesi bi dugo čekali na procesor a ulazno-izlazni uređaji bi bili nezaposleni. Eksterni prioriteti se nameću na osnovu kriterijuma koji su spoljašnji u odnosu na operativni sistem odnosno, uglavnom su „političke" prirode. U tom slučaju, na prioritet procesa mogu uticati vremenski rokovi, važnost klijenta, vrednost projekta itd.



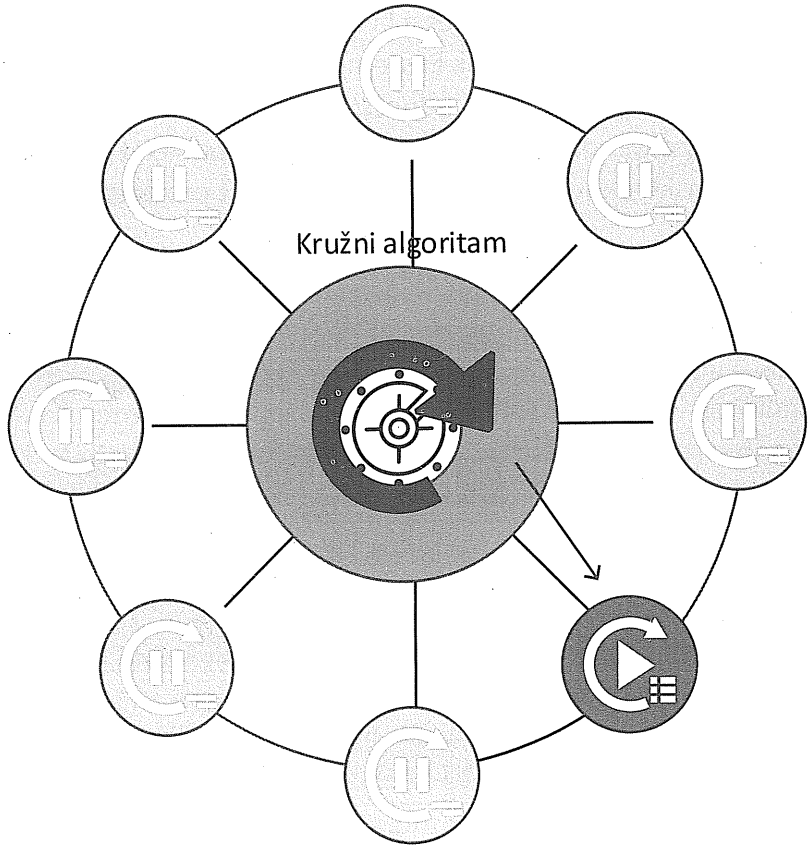
Slika 2.26. Prioritet procesa

Najveća mana ovog algoritma je potencijalno „izgladnjivanje" procesa. Naime, može se dogoditi da procesi nižeg prioriteta dugo (teoretski i beskonačno) čekaju na procesor jer u sistem stalno pristižu procesi višeg prioriteta. Ovaj problem se može rešiti povećavanjem prioriteta procesa sa porastom njegovog vremena provedenog u redu za čekanje.

Kao i kod prethodnog algoritma, za algoritam sa prioritetima postoji verzija sa i bez prekidanja. Prekidanje podrazumeva da se, ako se u

sistemu pojavi novi proces višeg prioriteta od procesa koji se trenutno izvršava, tekući proces prekine a proces višeg prioriteta počne da se izvršava.

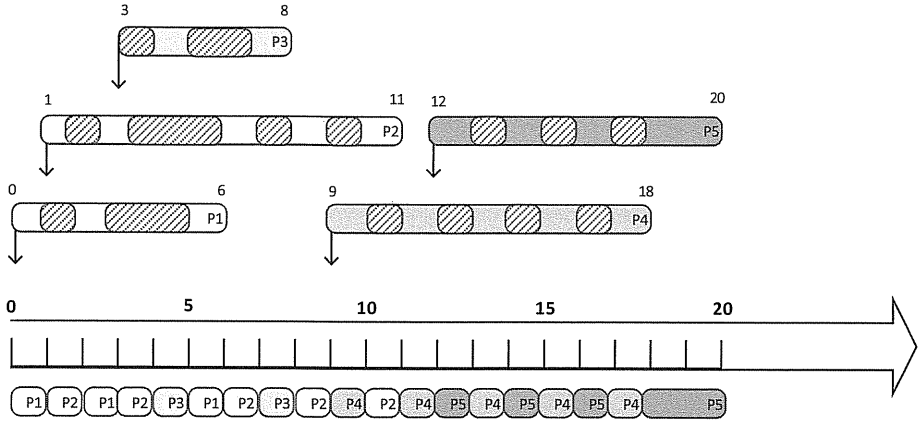
2.9.4. Kružni algoritam



Slika 2.27. Kružni algoritam

Kružni algoritam (*Round Robin*) je zasnovan na ideji za opsluživanje računara slično kao kada je deljenje vremena (*Time-sharing*) u pitanju. Naime, svaki proces dobija procesor na unapred zadati vremenski interval koji se naziva kvantum vremena (*time quantum*). Kada to vreme istekne ili ako se proces izvrši do kraja, proces se prekida a procesor na kvantum vremena dobija sledeći proces iz reda spremnih procesa (slika 2.27.). Prekinuti proces (u slučaju da se nije izvršio do kraja) se stavlja na kraj

reda spremnih procesa. Ako ima n spremnih procesa, a vremenski kvantum je dužine q, onda svaki proces dobija najmanje $1/n$ procesorskog vremena, najviše po q u jednom prolazu. Pri tome, nijedan proces ne čeka više od $(n-1)q$ vremena. Kao što se iz prethodnog može zaključiti, performanse ovog algoritma veoma zavise od kvantuma vremena. Ako je taj kvantum vremena veći od svih mogućih zahteva za procesorom (teži beskonačnosti), ovaj algoritam se ponaša kao FCFS algoritam. Nasuprot tome, ukoliko je vremenski kvantum veoma mali, kružni algoritam se naziva *deljenje procesora* i teoretski izgleda da svaki od n procesa u redu ima sopstveni procesor koji je n puta sporiji od procesora koji se nalazi u sistemu.



Slika 2.28. Kružni algoritam sa vremenskim kvantumom 1

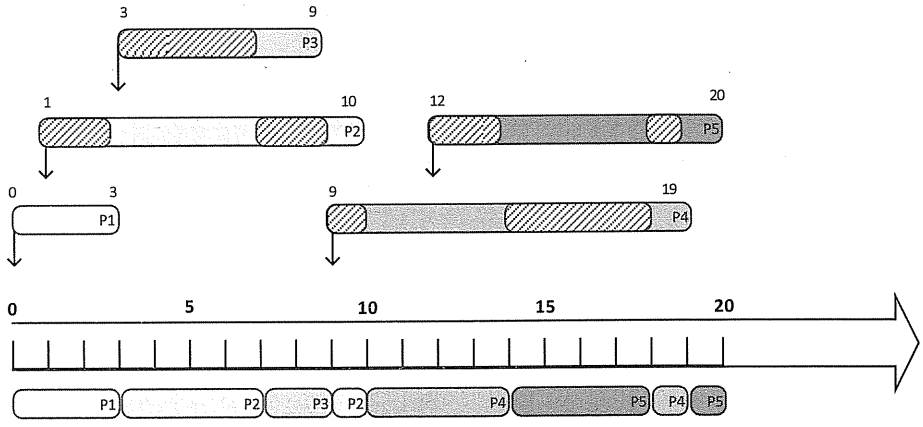
Na kraju svakog kvantuma vremena vrši se prebacivanje konteksta procesa. Kao što je ranije rečeno, prebacivanje konteksta je, iz ugla izvršavanja procesa ali i funkcionisanja sistema, potpuno nekoristan posao. Trajanje prebacivanja konteksta zavisi od brzine memorije, broja registara i postojanja specijalnih instrukcija. Poželjno je da se kvantum vremena izabere tako da bude znatno duži od vremena potrebnog za prebacivanje konteksta. Takođe, vreme obrade zavisi od kvantuma vremena, odnosno ono je kraće ako većina poslova svoju aktivnost na procesoru završava za jedan kvantum vremena. Izbor kvantuma vremena je veoma delikatan posao jer iz prethodnog sledi da je bolje imati duži

kvantum vremena, ali ako je kvantum preveliki onda algoritam više liči na FCFS algoritam.

Tabela 2.5. Primer kružnog algoritma

Proces	Vreme pristizanja	Zahtevano vreme	Vreme kraja	T	M	P
P_1	0	3	6	6	3	2.0
P_2	1	5	11	10	5	2.0
P_3	3	2	8	5	3	2.5
P_4	9	5	18	9	4	1.8
P_5	12	5	20	8	3	1.6
Prosek				**7.6**	**3.6**	**1.98**

Na slici 2.28. ilustrovan je način raspoređivanja 5 procesa čiji su zahtevi i vremena pristizanja zadati u tabeli 2.5. Pri raspoređivanju procesa je korišćen Kružni algoritam sa vremenskim kvantumom od 1 vremenske jedinice. U tabeli 2.5. se mogu videti i karakteristike algoritma kao i vremena izvršavanja procesa.



Slika 2.29. Kružni algoritam sa vremenskim kvantumom 4

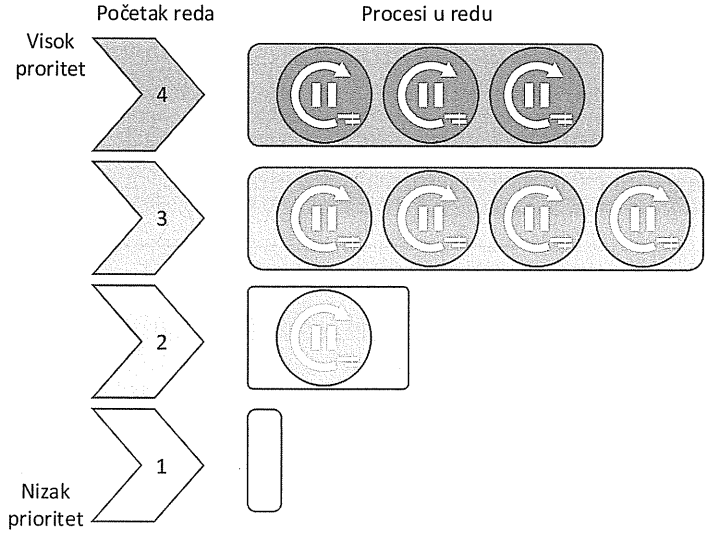
Na slici 2.29. i u tabeli 2.6. prikazani su detalji raspoređivanja procesa kao u prethodnom primeru ali uz vremenski kvantum od 4 vremenske jedinice. U ovom primeru, povećavanje vremenskog kvantuma doprinelo je boljim karakteristikama algoritma.

Tabela 2.6. Kružni algoritam sa vremenskim kvantumom 4

Proces	Vreme pristizanja	Zahtevano vreme	Vreme kraja	T	M	P
P_1	0	3	3	3	0	1.0
P_2	1	5	10	9	4	1.8
P_3	3	2	9	6	4	3.0
P_4	9	5	19	10	5	2.0
P_5	12	5	20	8	3	1.6
Prosek				**7.2**	**3.2**	**1.88**

2.9.5. Redovi u više nivoa

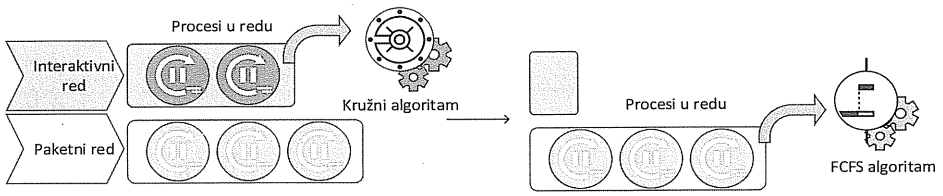
Redovi u više nivoa predstavljaju pristup koji je zasnovan na ideji da se red spremnih procesa podeli u više redova sa različitim prioritetima. Procesi koji pristižu svrstavaju se u odgovarajući red na osnovu nekog kriterijuma. Svaki red može imati sopstveni algoritam planiranja, a između samih redova postoji jasno definisana razlika u prioritetu (slika 2.30.). Proces koji je prvi u svom redu može dobiti procesor na korišćenje isključivo ako su redovi višeg prioriteta prazni.



Slika 2.30. Redovi u više nivoa

Najjednostavniji primer podrazumeva da postoje dva reda (slika 2.31.):

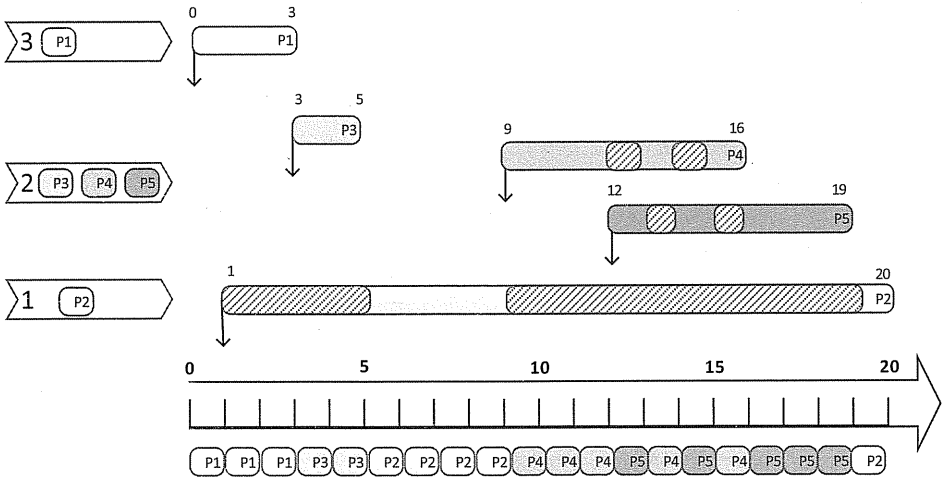
- Interaktivni (*Interactive*) i
- Paketni (*Batch*).



Slika 2.31. Interaktivni i paketni red

Pri tome, svaki red može imati svoj algoritam izbora, recimo u ovom primeru:

- Interaktivni red – Kružni algoritam
- Paketni red – FCFS algoritam.



Slika 2.32. Primer redova u više nivoa

Procesi iz interaktivnog reda bi pravo da koriste procesor dobijali na osnovu Kružnog algoritma. Ovaj algoritam bi se primenjivao dok se svi procesi iz prvog reda ne izvrše a onda bi se paketni red izvršio odabirom na osnovu FCFS algoritma.

Na slici 2.32. prikazan je malo kompleksniji primer koji ilustruje primenu algoritma na 5 procesa čiji su parametri zadati u tabeli 2.7. Svi redovi imaju Kružni algoritam raspoređivanja.

Tabela 2.7. Primer redova u više nivoa

Proces	Vreme pristizanja	Zahtevano vreme	Vreme kraja	T	M	P
P_1	0	3	3	3	0	1
P_2	1	5	20	19	14	3.8
P_3	3	2	5	2	0	1
P_4	9	5	16	7	2	1.4
P_5	12	5	19	7	2	1.4
Prosek				**7.6**	**3.6**	**1.72**

2.9.6. Redovi u više nivoa sa povratnom vezom

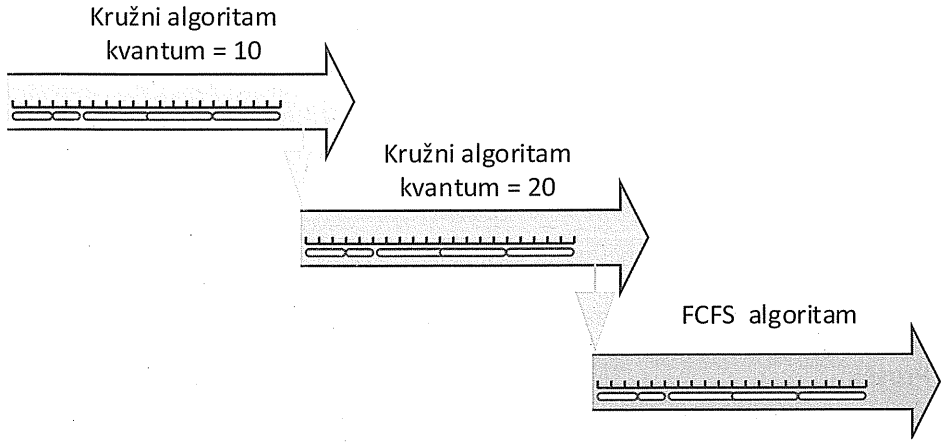
Redovi u više nivoa sa povratnom vezom predstavljaju modifikaciju prethodnog pristupa (Redovi u više nivoa). Naime, kod redova u više nivoa sa povratnom vezom dozvoljeno je kretanje procesa između redova. Ideja je da se u toku izvršavanja izdvoje procesi sa različitim trajanjem aktivnosti na procesoru tako da se procesi koji koriste previše procesorskog vremena premeste u redove sa nižim prioritetom. Ovakvim pristupom, procesi orijentisani više ka ulazno-izlaznim operacijama i interaktivni procesi se drže u redovima višeg prioriteta. Svaki proces u višim redovima dobija određeni kvantum vremena i kada ga iskoristi, premešta se u red na nižem nivou.

Na primer, ako postoje tri reda (slika 2.33.) sa sledećim algoritmima raspoređivanja:

1. Q_0 – Kružni algoritam sa kvantumom od 10 vremenskih jedinica;
2. Q_1 – Kružni algoritam sa kvantumom od 20 vremenskih jedinica;
3. Q_2 – FCFS algoritam.

U takvoj situaciji, kada se pojavi novi proces, on se smešta u prvi red Q_0. Tu dobija pravo da procesor koristi 10 vremenskih jedinica. Ako za to vreme proces ne završi sa izvršavanjem, premešta se u red Q_1. U Q_1 proces

dobija novih 20 vremenskih jedinica, a ako ni tada ne završi, prebacuje se u red Q_2 gde će ostati do kraja izvršavanja.



Slika 2.33. Primer redova u više nivoa sa povratnom vezom

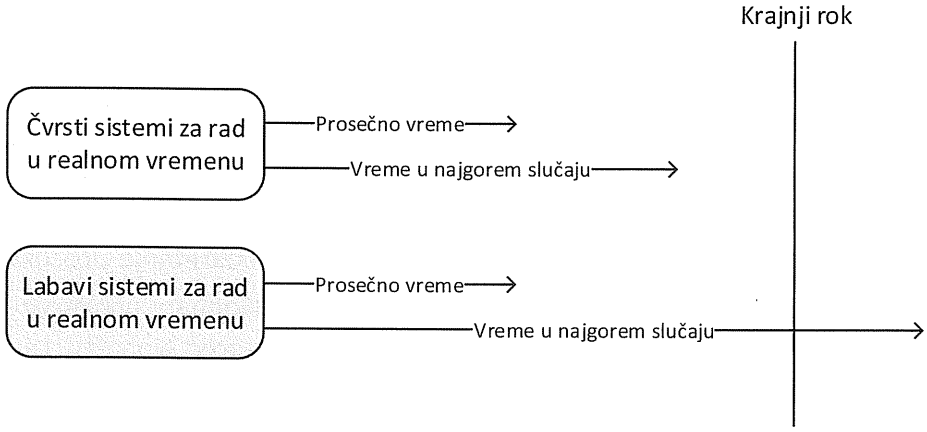
2.10. Upravljanje procesima u sistemima za rad u realnom vremenu

Sistemi za rad u realnom vremenu (*Real-time operating systems*) spadaju u operativne sisteme sa specijalnom namenom. Oni se obično projektuju za određenu primenu i koriste u situacijama kada postoje vremenska ograničenja za izvršavanje unapred definisanih poslova. U ovim sistemima je od suštinskog značaja da određeni poslovi budu obavljeni na vreme. U zavisnosti od toga do koje mere moraju da poštuju vremenska ograničenja, sistemi za rad u realnom vremenu se mogu podeliti na čvrste (*hard*) i labave (*soft*) (slika 2.34.).

Čvrsti sistemi za rad u realnom vremenu (*Hard real-time systems*) podrazumevaju da u sistemu postoji bar jedan proces koji ima čvrsti (fiksni) krajnji rok (*Hard deadline*) za izvršavanje koji se mora ispoštovati. Prekoračivanje zadatog roka za izvršavanje procesa obično dovodi do delimičnog ili potpunog otkaza čitavog sistema a rezultati dobijeni nakon isteka roka se odbacuju kao nekorisni. Ovakvi sistemi mogu se koristiti za

kontrolu: uređaja u avionima, odbrambenih sistema, avio saobraćaja, uređaja u vozilima, elektrana, komponenti pejsmejkera itd.

Labavi sistemi za rad u realnom vremenu (*Soft real-time systems*) podrazumevaju da je poštovanje zadatog roka za izvršavanje procesa poželjno, ali nije od presudnog značaja za funkcionisanje sistema. Naime, procesima se određuju rokovi (*Soft deadlines*) i njihovo prekoračenje utiče na efikasnost sistema ali ne ugrožava njegovo funkcionisanje. Drugim rečima, prekoračenje rokova se može tolerisati u izvesnim granicama. Brojni primeri ovakve vrste sistema mogu se videti u elektronskom bankarstvu, kod uređaja za rezervaciju avionskih karata, kod audio-video uređaja itd.



Slika 2.34. Sistemi za rad u realnom vremenu

Procesi u sistemima za rad u realnom vremenu mogu biti periodični i neperiodični (asinhroni). *Periodične procese* karakteriše ponavljanje posle tačno određenog vremenskog perioda. *Neperiodični procesi* obično imaju rok kada moraju da počnu ili da završe sa izvršavanjem (ili oba pomenuta ograničenja) ali mogu početi sa izvršavanjem u proizvoljnom trenutku između tih granica.

2.10.1. Raspoređivanje procesa u sistemima za rad u realnom vremenu

Zbog svoje specifičnosti sistemi za rad u realnom vremenu zahtevaju poseban način za raspoređivanje procesa. Pravljenje redosleda izvršavanja procesa se, kao i kod ostalih sistema, poverava planerima. Algoritmi za planiranje pri donošenju odluka najčešće uzimaju u obzir prioritete procesa, tako da prednost dobijaju procesi višeg prioriteta. Pri tome, algoritmi se mogu implementirati tako da dozvoljavaju ili ne dozvoljavaju prekidanje procesa. U situaciji kada je dozvoljeno prekidanje, pri ulasku u sistem procesa koji ima viši prioritet od onoga koji se trenutno izvršava, prekida se trenutni proces i počinje da se izvršava novi proces. Kada je verzija bez prekidanja u pitanju, proces koji je dobio priliku da se izvršava ne može biti prekinut. Prioriteti se mogu dodeliti statički ili dinamički.

Statičko određivanje prioriteta se obavlja u fazi implementiranja sistema i ovako dodeljeni prioriteti ne mogu se menjati u toku rada sistema. Prioriteti se obično određuju na osnovu procene važnosti procesa koja se dobija na osnovu kriterijuma kao sto su: učestalost pokretanja procesa, hitnost, vreme izvršavanja itd.

Dinamičko određivanje prioriteta podrazumeva da se oni mogu menjati u toku rada sistema. Njih određuju planeri na osnovu trenutnog stanja u sistemu, očekivanog vremena izvršenja procesa, krajnjih rokova za obavljanje procesa itd.

Smatra se da je sistem za rad u realnom vremenu *korektan* ako u tačno definisanim vremenskim ograničenjima daje odgovarajuće rezultate. U narednom delu teksta biće prikazana dva algoritma za raspoređivanje procesa u sistemima za rad u realnom vremenu, od kojih jedan ima statičko, a drugi dinamičko određivanje prioriteta.

RMS (*Rate Monotonic Scheduling*) *algoritam* za raspoređivanje procesa se primenjuje kod sistema za rad u realnom vremenu sa periodičnim procesima. On pretpostavlja da se u sistemu izvršava n procesa:

$P_1, \ldots, P_n$; pri tome se svaki od procesa P_i periodično pokreće (perioda T_i). Poznato je vreme njegovog izvršavanja C_i a pretpostavka je i da je krajnji rok zadatka kraći od njegovog perioda.

Kod RMS algoritma (statički) prioriteti se određuju na osnovu periode procesa tako da proces sa najkraćom periodom dobija najviši prioritet a ostali redom manje prioritete sve do procesa koji ima najdužu periodu koji dobija najniži prioritet. Ovaj algoritam dozvoljava prekidanje.

Ukoliko se ukupno opterećenje sistema računa na osnovu formule:

$$\sum_{i=1}^{n} \frac{C_i}{T_i},$$

RMS algoritam za planiranje je sigurno korektan odnosno ima mogućnost da sve procese izvrši poštujući zadate rokove, ukoliko je ispunjena sledeća nejednakost:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1).$$

Granična vrednost RMS algoritama (desni deo nejednakosti) predstavlja iskorišćenje procesora u najgorem slučaju kada je *n* procesa u sistemu. Iz te formule se može zaključiti da ovaj algoritam obezbeđuje relativno nizak nivo iskorišćenosti sistema. Naime, poželjno je da nivo iskorišćenosti bude što je moguće bliži 1 (100%), ali RMS algoritam nameće ograničenje u pogledu nivoa iskorišćenosti jer je iskorišćenost procesora umanjena u cilju garantovanja korektnosti. Osim toga, sistemi čiji je nivo iskorišćenosti veći od graničnog i manji od 1 nemaju jasno definisano pitanje korektnosti.

EDFS algoritam (*Earliest-Deadline-First Scheduling*) je dinamički algoritam koji najviši prioritet dodeljuje procesu koji je najbliži svom krajnjem roku za izvršavanje. Praktično, vremenom raste prioritet procesa koji čeka da se izvrši. Pošto je u pitanju dinamičko planiranje dodatno vreme će se

trošiti na algoritme koji bi planerima trebalo da pomognu u donošenju odluka.

Ovakav pristup garantuje održivost (korektnost) sistema pod uslovom da je opterećenje sistema manje od maksimalnog. Ukoliko opterećenje sistema u nekom trenutku bude veće od 1, to znači da će barem jedan zadatak premašiti krajnji rok.

Pitanja

2.1. Objasniti razliku između programa i procesa.
2.2. Koji su memorijski zahtevi procesa?
2.3. Koje su najbitnije informacije o toku izvršavanja procesa i gde se čuvaju?
2.4. Od čega zavise dužine vremenskih intervala koje procesi dobijaju za izvršavanje na procesoru?
2.5. Koje segmente obično procesi imaju na raspolaganju kada se učitaju u memoriju?
2.6. Šta je izvorni kôd programa?
2.7. Šta je izvršni program i kako on nastaje?
2.8. Koja stanja procesa obično podržavaju operativni sistemi?
2.9. Pod kojim uslovima procesi menjaju stanja?
2.10. Šta se događa sa suspendovanim procesima?
2.11. Iz kog razloga se najčešće suspenduju spremni procesi?
2.12. Zbog čega se najčešće suspenduju procesi koji čekaju?
2.13. Čemu služi kontrolni blok procesa?
2.14. Kako se mogu podeliti informacije o procesima koje obično čuva kontrolni blok procesa?
2.15. Koje mehanizme za manipulaciju kontrolnim blokovima procesa bi trebalo da obezbede operativni sistemi?
2.16. Objasniti kako funkcioniše prebacivanje konteksta.
2.17. Koji modul operativnog sistema je zadužen za prebacivanje konteksta?
2.18. Koje su osnovne klase podataka o kojima se vodi evidencija u kontrolnim blokovima procesa?
2.19. Koje su najčešće operacije koje se izvode sa kontrolnim blokovima procesa?
2.20. Objasniti koncept niti.
2.21. Šta je od podataka zajedničko za sve niti jednog procesa, a šta nije?
2.22. Koje su najvažnije prednosti koje donosi korišćenje niti?
2.23. Navesti primere upotrebe niti.

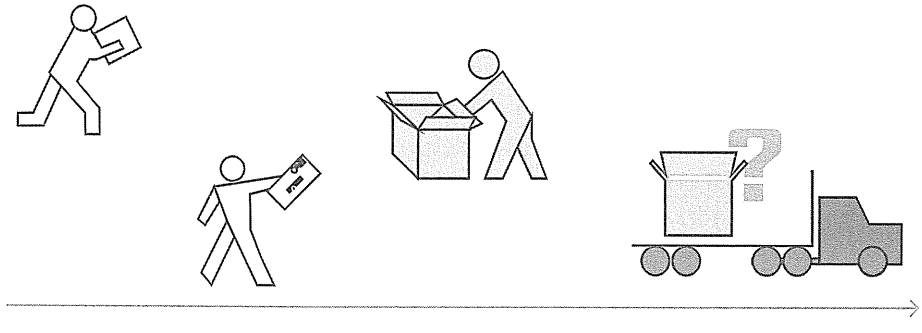
2.24. Na kojim nivoima se može pružiti podrška za rad sa nitima?
2.25. Na koji način se omogućava korespondencija korisničkih i sistemskih niti?
2.26. Koje su glavne odlike preslikavanja više u jednu?
2.27. Koja je najveća mana modela preslikavanja jedna u jednu?
2.28. Od čega zavisi broj niti jezgra koje će biti kreirane prilikom preslikavanja više u više?
2.29. Od čega se sastoji izvršavanje procesa u sistemima koji podržavaju multiprogramiranje i slične koncepte?
2.30. Koju ulogu imaju planeri u operativnim sistemima?
2.31. Šta je red spremnih procesa i šta se u njemu nalazi?
2.32. Od čega zavisi izbor procesa za red spremnih procesa?
2.33. O kojim redovima procesa operativni sistemi vode računa?
2.34. Koji se planeri najčešće implementiraju u okviru operativnih sistema?
2.35. Koja je osnovna funkcija dugoročnog planera?
2.36. Šta je glavni zadatak kratkoročnih planera?
2.37. Koja je uloga srednjoročnog planera?
2.38. Kakva se rešenja za raspoređivanje procesa mogu implementirati u višeprocesorskim sistemima?
2.39. Kakvi pristupi postoje kada je rešenje za raspoređivanje procesa u višeprocesorskim sistemima sa zajedničkim redom čekanja u pitanju?
2.40. Šta se podrazumeva pod balansiranjem opterećenja?
2.41. Iz kojih razloga se uvodi afinitet procesa prema procesoru?
2.42. Kakav može biti afinitet procesa prema procesoru?
2.43. Šta se podrazumeva pod raspoređivanjem procesa?
2.44. Koji su kriterijumi koji se posmatraju prilikom ocenjivanja algoritama raspoređivanja?
2.45. Koje su glavne odlike FCFS algoritma za raspoređivanje procesa?
2.46. Koja je osnovna ideja SPF algoritma za raspoređivanje procesa?
2.47. U čemu je razlika između verzija SPF algoritma sa i bez prekidanja?
2.48. Koji je najveći problem kada je realizacija SPF algoritma u pitanju?

2.49. Na koji način se obično vrši procena dužine sledeće aktivnosti procesa na procesoru?
2.50. Prodiskutovati algoritam sa prioritetima za raspoređivanje procesa.
2.51. Koje su mane algoritma sa prioritetima?
2.52. Na koji način se mogu definisati prioriteti kada je algoritam sa prioritetima u pitanju?
2.53. Kako se može rešiti problem „izgladnjivanja" procesa kod algoritma sa prioritetima?
2.54. Koja je osnovna ideja kružnog algoritma za raspoređivanje procesa?
2.55. Kako se ponaša kružni algoritam u zavisnosti od dužine kvantuma?
2.56. O čemu treba voditi računa prilikom određivanja dužine kvantuma vremena kod kružnog algoritma?
2.57. Prodiskutovati karakteristike pristupa zasnovanog na redovima u više nivoa za raspoređivanje procesa.
2.58. Koja je glavna ideja kada su redovi u više nivoa sa povratnom vezom za raspoređivanje procesa u pitanju?
2.59. Koji put pređe proces prilikom svog izvršavanja u sistemu u kojem je implementiran algoritam zasnovan na redovima u više nivoa sa povratnom spregom? Navesti primer.
2.60. Koje su glavne odlike sistema za rad u realnom vremenu?
2.61. Koje vrste sistema za rad u realnom vremenu postoje?
2.62. Kako se procesima u sistemima za rad u realnom vremenu mogu dodeliti prioriteti?
2.63. Koje su odlike RMS algoritma za raspoređivanje procesa u sistemima za rad u realnom vremenu?
2.64. Koje su odlike EDFS algoritma za raspoređivanje procesa u sistemima za rad u realnom vremenu?

3. KONKURENTNOST I SINHRONIZACIJA PROCESA

Konkurentno ili paralelno izvršavanje više procesa ili niti, osim prednosti koje donosi kada je efikasnost sistema u pitanju, može proizvesti dosta problema pri manipulaciji nad zajedničkim podacima. Naime, pri čestim promenama procesa (ili niti) koji koriste procesor, kao i u sistemima sa više procesora, nekontrolisano pristupanje zajedničkim podacima može dovesti do neželjenog krajnjeg rezultata. Deljeni podaci su veoma osetljivo mesto. Prekidanje procesa koji im je pristupao i propuštanje drugog, može dovesti do nekonzistentnosti podataka i neočekivanih grešaka. Takođe, do sličnih problema može doći i prilikom paralelnog izvršavanja procesa koji modifikuju iste podatke.

Situacija u kojoj krajnji rezultat zavisi od redosleda izvršavanja koraka različitih procesa koji manipulišu zajedničkim podacima (slika 3.1.), naziva se *trka za resurse* (*race condition*).



Slika 3.1. Trka za resurse

Najjednostavniji primer koji ilustruje trku za resurse je problem uvećavanja zajedničke promenljive. Neka je X promenljiva kojoj mogu pristupati različiti procesi koji žele da je uvećaju za 1:

$$X = X + 1;$$

Operacija promene vrednosti promenljive nije atomična operacija, tj. ne izvršava se „odjednom“ već kroz tri mašinske instrukcije, odnosno u tri koraka koja su navedena u primeru 3.1.

Primer 3.1. **Operacija uvećavanja promenljive**
1. Smestiti promenljivu *X* u registar *R*; 2. Uvećati registar *R* za 1; 3. Premestiti sadržaj registra *R* u promenljivu *X*;

Ukoliko dva procesa nastoje da obave ovakvu operaciju, može se dogoditi da jedan počne sa izvršavanjem koraka za uvećanje promenljive i da pre poslednje instrukcije bude prekinut, pri čemu drugi proces krene da se izvršava i da uvećava istu promenljivu. U tom slučaju će se skoro sigurno dogoditi da po završetku oba procesa promenljiva *X* nema očekivanu vrednost.

Primer 3.2. **Izvršavanje dva procesa**
Proces 1: Smestiti promenljivu *X* u registar *R*; Uvećati registar *R* za 1; Prekid procesa; **Proces 2:** Smestiti promenljivu *X* u registar *R*; Uvećati registar *R* za 1; Premestiti sadržaj registra *R* u promenljivu *X*; Prekid procesa; **Proces 1:** Premestiti sadržaj registra *R* u promenljivu *X*;

Primer 3.2. prikazuje situaciju u kojoj dolazi do prekida procesa prilikom povećavanja vrednosti promenljive, što dovodi do dobijanja neočekivane vrednosti po završetku izvršavanja procesa. Neka promenljiva *X* na početku ima vrednost 13 i konkurentno se izvršavaju dva procesa (Proces 1 i Proces 2) pri čemu oba žele da povećaju vrednost promenljive *X* za 1. Vrednost promenljive *X* nakon završetka ovih procesa bi trebalo da bude 15. Međutim, ako dođe do prekida prvog procesa (Proces 1) nakon što

poveća vrednost registra, on u tom trenutku još uvek nije upisao novu vrednost nazad u promenljivu. Tada drugi proces (Proces 2) učitava promenljivu *X* u svoj registar (koja i dalje ima vrednost 13), povećava njenu vrednost i upisuje je u promenljivu *X*. Nakon toga prvi proces nastavlja sa izvršavanjem, upisuje vrednost svog registra (14) u promenljivu *X*. Na kraju se dobija da je rezultat uvećavanja promenljive *X* za jedan od strane dva procesa 14, a ne 15, kao što se očekivalo.

Imajući u vidu prethodni primer, može se pretpostaviti da vrednost promenljive *X* (*X* ima vrednost 0 na početku) posle izvršavanja *k* procesa koji konkurentno ili paralelno, uvećavaju promenljivu *X* u ciklusu od *n* ponavljanja (primer 3.3.) verovatno neće biti *kn*.

Primer 3.3. **Izvršavanje i-tog procesa**
FOR ($j = 0; j < n; j$++) X++; **ENDFOR**

Primer potrošača i proizvođača se često koristi kao ilustracija trke za resurse (primer 3.4.). Dva procesa (proizvođački i potrošački) imaju zadatak da kontrolišu stanje u magacinu. Proces (funkcija) koji simulira rad proizvođača (proizvođač()) ima zadatak da povećava broj proizvoda koji se nalaze u magacinu u trenutku kada se novi proizvod donese u magacin. Analogno, proces koji simulira rad potrošača (potrošač()) umanjuje zajedničku promenljivu koja sadrži stanje u magacinu kada se proizvod iznese iz magacina.

Primer 3.4. **Proizvođač – potrošač**

```
Glavni program:
  //Prva slobodna pozicija u magacinu na koju se dodaje novi proizvod
  unutra = 0;
  //Pozicija najstarijeg proizvedenog proizvoda u magacinu
  van = 0;
  //Broj proizvoda u magacinu
  brojač = 0;
  //Proces proizvođač i proces potrošač se izvršavaju paralelno
  PARALLEL WHILE (true)
    //Procesi mogu biti pozvani u proizvoljnom trenutku
    proizvođač();
    potrošač();
  END PARALLEL WHILE

proizvođač()
  WHILE (true)
    Novi proizvod je proizveden;
    //Magacin je pun ako je brojač proizvoda u magacinu jednak broju
    //mesta u magacinu i treba sačekati
    WHILE (brojač == veličina_magacina)
      //Aktivno čekanje dok se ne oslobodi bar jedno mesto u magacinu
    ENDWHILE
    //Novi proizvod se dodaje na prvu sledeću slobodnu poziciju
    magacin[unutra] = novi_proizvod;
    //Pomera se indeks na sledeću poziciju
    unutra = (unutra + 1) mod veličina_magacina;
    //Uvećava se brojač koji govori koliko je proizvoda u magacinu
    brojač++;
  ENDWHILE
potrošač()
  //Ako je brojač koji broji koliko je proizvoda u magacinu jednak
  //0 to znači da je magacin prazan
  WHILE(brojač == 0)
    //Aktivno čekanje da se bar jedan proizvod proizvede
  ENDWHILE
  //Uzima se najstarije proizvedeni proizvod
  prodaja = magacin[van];
  //Povećava se brojač koji pokazuje na proizvod koji je
  //najduže u magacinu
  van = (van + 1) mod veličina_magacina;
  //Ažurira se broj proizvoda koji se nalazi u magacinu tako da
  //odgovara stvarnom broju tj. umanjuje se za jedan
  brojač--;
```

Kao i u prethodnim primerima, uvećanje i umanjenje za jedan nisu atomične operacije:

brojač++ :

1. *registar_1* = *brojač*
2. *registar_1* = *registar_1* + 1
3. brojač = registar_1

brojač-- :

1. *registar_2* = *brojač*
2. *registar_2* = *registar_2* – 1
3. *brojač* = *registar_2*

Neka je *brojač* = 8 i neka se, zbog prekidanja, koraci proizvođačkog i potrošačkog procesa za izmenu promenljive *brojač* izvršavaju sledećim redosledom:

1. Proizvođač: *registar_1* = *brojač* //*registar_1* = 8
2. Proizvođač: *registar_1* = *registar_1* + 1 //*registar_1* = 9
3. Potrošač: *registar_2* = *brojač* //*registar_2* = 8
4. Potrošač: *registar_2* = *registar_2* – 1 //*registar_2* = 7
5. Proizvođač: *brojač* = *registar_1* //*brojač* = 9
6. Potrošač: *brojač* = *registar_2* //*brojač* = 7

Jasno je da će doći do nekonzistentnosti, odnosno do razlike u broju stvarnih proizvoda u magacinu i vrednosti promenljive brojač. Naime, po završetku ovih operacija promenljiva brojač će imati vrednost 7 a u magacinu će regularno biti smešteno 8 proizvoda.

Rešavanje problema ovakve prirode podrazumeva da se implementira neka vrsta zaštite kojom bi se obezbedilo da osetljivim delovima može pristupiti samo jedan proces u jednom trenutku.

3.1. Kritična sekcija

Prethodni primeri ilustruju probleme do kojih može doći kada dva ili više procesa konkurišu za pristupanje deljenim podacima. Deo programa u

kojem se pristupa zajedničkim podacima naziva se *kritična sekcija*. Dovoljna garancija da ne dođe do neželjenih rezultata ovakvog tipa je da se obezbedi da kritičnu sekciju u jednom trenutku može da izvršava samo jedan proces. Međutim, rešavanje problema kritične sekcije na ovakav način nije trivijalno, naročito u sistemima gde su efikasnost a samim time i što viši stepen multiprogramiranja primarni.

Rešenje koje se prvo nameće u sistemima koji imaju samo jedan procesor je da se onemogući prekidanje procesa koji je u kritičnoj sekciji. Na ovaj način proces koji pristupa deljenim podacima neće biti prekinut pa samim tim i ne postoji mogućnost da neki drugi proces pristupi tim podacima. Rešenje sa isključivanjem prekida se ne primenjuje često, jer je suština implementacije multiprogramiranja zasnovana na sistemu prekida, pa bi njegovo često isključivanje dovelo do smanjenja efikasnosti sistema. Takođe, jedan od problema ovakvog pristupa je u tome što često nije jasno definisano šta kritična sekcija treba da obuhvati. Rešenje koje podrazumeva isključivanje prekida je pogodno primenjivati u situacijama kada je u kritičnoj sekciji mali broj instrukcija. Ovakvo rešenje se ne može koristiti u višeprocesorskim sistemima jer ne pruža garanciju da različiti procesi, koji se istovremeno izvršavaju, neće modifikovati deljene podatke.

Zbog prethodno navedenih nedostataka, rešenja sa isključivanjem prekida se vrlo retko primenjuju. Predložena su mnogobrojna rešenja od kojih će neka biti prikazana u narednom delu. Dobro rešenje za zaštitu kritične sekcije trebalo bi da zadovolji sledeće uslove:

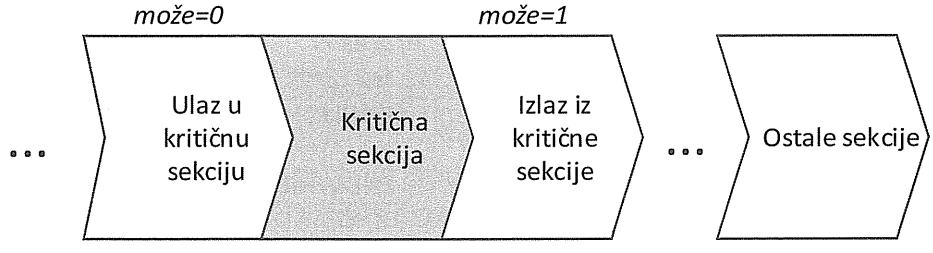
- *Uzajamna isključivost* – podrazumeva da dva procesa ne mogu u isto vreme da budu u kritičnoj sekciji, što je i ključni zahtev bez kojeg nema rešenja za zaštitu kritične sekcije.
- *Uslov progresa* – zahteva da proces koji nije u kritičnoj sekciji i ne želi da uđe u nju ne treba da ometa druge procese da u nju uđu.
- *Uslov konačnog čekanja* – podrazumeva da bi trebalo da postoji (razumna) granica koliko jedan proces može da čeka na ulazak u

kritičnu sekciju, odnosno da se ni pod kojim uslovima ne može dogoditi da neki od procesa neograničeno dugo čeka.

3.2. REŠENJA ZASNOVANA NA AKTIVNOM ČEKANJU

Veći broj rešenja za zaštitu kritične sekcije se zasniva na tzv. aktivnom čekanju, odnosno na čekanju u ciklusu (petlji) dok se ne stvore uslovi za ulazak procesa u kritičnu sekciju.

Može se steći utisak da je relativno jednostavno implementirati graničnike koji bi omogućili zaštitu kritične sekcije (slika 3.2.).



Slika 3.2. Zaštita kritične sekcije uz pomoć graničnika

Algoritam 3.1. prikazuje jedan predlog za rešenje zaštite kritične sekcije.

Algoritam 3.1. **Zaštita kritične sekcije (1)**	

Proces 1:
WHILE (*može* == 0)
//Aktivno čekanje
ENDWHILE
može = 0;
Kritična sekcija
može = 1;
	Proces 2:
	WHILE (*može* == 0)
	//Aktivno čekanje
	ENDWHILE
	može = 0;
	Kritična sekcija
	može = 1;

Ideja je da se aktivnim čekanjem u ciklusu sačeka da promenljiva *može* postane 1 (ako već nije), odnosno da ulaz u kritičnu sekciju bude slobodan. Po završetku rada u kritičnoj sekciji proces postavlja *može* na 1 čime drugom procesu omogućava da uđe u kritičnu sekciju ukoliko to želi.

Međutim, iako se čini da je ovo rešenje dobro, najvažniji uslov uzajamne isključivosti nije zadovoljen. Naime, može se dogoditi da jedan proces izvrši proveru da li je *može* jednako 1 i pošto utvrdi da jeste, krene da je postavi na 0. Ako taj proces bude prekinut pre nego što se *može* postavi na 0, drugi proces može videti da je ulaz u kritičnu sekciju slobodan i u nju ući.

Za potrebe rešenja koje bi trebalo da zadovolji uslov uzajamnog isključivanja uvode se dve promenljive *želi_1* i *želi_2* umesto promenljive *može* u prethodnom primeru. One se koriste da bi procesi mogli da najave da žele da uđu u kritičnu sekciju (algoritam 3.2.).

Algoritam 3.2. **Zaštita kritične sekcije (2)**	

Proces 1:
želi_1 = 1;
WHILE(*želi_2* == 1)
//Aktivno čekanje
ENDWHILE
Kritična sekcija
želi_1 = 0;
	Proces 2:
	želi_2 = 1
	WHILE(*želi_1* == 1)
	//Aktivno čekanje
	ENDWHILE
	Kritična sekcija
	želi_2 = 0;

Svaki proces najavi da želi u kritičnu sekciju i aktivno čeka dok i drugi to želi. Kada proces završi sa radom u kritičnoj sekciji, on svoju promenljivu *želi* postavlja na 0 čime drugi proces oslobađa od aktivnog čekanja i propušta ga.

Ovo rešenje zadovoljava uslov uzajamnog isključivanja, ali se može dogoditi da se procesi zaglave. Naime, ako prvi proces najavi da želi da uđe u kritičnu sekciju (postavi *želi_1* na 1) i u tom trenutku bude prekinut, drugi proces će tada postaviti *želi_2* na 1 i zaglaviti se u petlji jer je *želi_1* jednako 1. Po nastavku rada prvog procesa, on će se takođe zaglaviti u petlji jer je sada i vrednost promenljive *želi_2* jednaka 1.

Prethodna dva primera ilustruju da rešenje problema kritične sekcije nije trivijalno i da je veoma lako pogrešiti.

3.2.1. Striktna alternacija

Striktna alternacija (*strict alternation*) je jedno od rešenja za zaštitu kritične sekcije (algoritam 3.3.). Ona se zasniva na korišćenju promenljive (*na_redu*) kojom se određuje koji od dva procesa ima prednost. Na početku, prednost se daje prvom procesu, tj. *na_redu* = 1.

Algoritam 3.3. **Striktna alternacija**	

Proces 1:
WHILE(*na_redu == 2*)
//Aktivno čekanje
ENDWHILE
Kritična sekcija
na_redu = 2;
	Proces 2:
	WHILE(*na_redu == 1*)
	//Aktivno čekanje
	ENDWHILE
	Kritična sekcija
	na_redu = 1;

Procesi aktivno čekaju da dođu na red, tj. da promenljiva *na_redu* dobije vrednost njihovog broja (1 ili 2). Kada napusti kritičnu sekciju, proces promenljivoj *na_redu* postavlja vrednost broja drugog procesa koji tako dobija šansu da pristupi kritičnoj sekciji ukoliko to želi.

Od tri uslova koje bi trebalo da zadovoljava dobro rešenje za zaštitu kritične sekcije, ovo rešenje ispunjava samo uslov uzajamne isključivosti. Naime, ukoliko jedan proces ima prednost a ne želi da uđe u kritičnu sekciju, on će blokirati drugi proces i time onemogućiti progres, a kako to blokiranje može da bude na neodređeno vreme, jasno je da ni uslov konačnog čekanja nije zadovoljen. Takođe, mana ovog algoritma je i to što je on namenjen za situacije koje podrazumevaju učešće samo dva procesa. Sa druge strane, striktna alternacija je veoma pogodna za problem sinhronizacije kada procesi treba naizmenično da pristupaju deljenim resursima.

3.2.2. Dekerov algoritam

Dekerov algoritam predstavlja prvo kompletno rešenje za uzajamno isključivanje u slučaju kada dva procesa konkurišu za deljene podatke. Rešenje je prezentovao holandski matematičar Deker (*Theodorus Jozef Dekker*) a Edsger W. Dijkstra ga je opisao u [5]. Algoritam 3.4. prikazuje ovaj algoritam za zaštitu kritične sekcije.

Za rešenje su potrebne tri promenljive:

1. *želi_proces_1* kojom prvi proces najavljuje da želi da uđe u kritičnu sekciju;
2. *želi_proces_2* kojom drugi proces najavljuje da želi da uđe u kritičnu sekciju;
3. *na_redu* kojom se vodi računa o tome koji je proces na redu da uđe u kritičnu sekciju.

Na početku se jednom procesu daje prednost, odnosno promenljiva *na_redu* dobija vrednost 1 ili 2. Neka je *na_redu* jednako 1. Kada prvi proces želi da pristupi kritičnoj sekciji on to najavi tako što promenljivu *želi_proces_1* postavi na 1. Zatim se proverava da li postoji konkurencija, odnosno da li je i drugi proces zainteresovan za kritičnu sekciju.

Ako je *želi_proces_2* jednako 0, tj. ako drugi proces ne želi da uđe u kritičnu sekciju, onda je prvi proces slobodan da u nju uđe. Po završetku rada u kritičnoj sekciji proces označava da je završio sa korišćenjem kritične sekcije tako što postavlja promenljivu *želi_proces_1* na 0.

Ako drugi proces želi da uđe u kritičnu sekciju, tj. ako je *želi_proces_2* jednako 1 onda se proverava ko ima prednost a oba procesa ulaze u spoljašnju (WHILE) petlju u kojoj će ostati dok jedan od procesa ne postavi svoju želju na 0.

Ako je drugi proces na redu, onda prvi menja vrednost promenljive *želi_proces_1* na 0 čime omogućava drugom procesu da napusti spoljašnju (WHILE) petlju i uđe u kritičnu sekciju. Za to vreme prvi aktivno čeka u unutrašnjoj (WHILE) petlji dok drugi proces ne završi sa radom u kritičnoj sekciji i postavi *na_redu* na 1. Time prvom procesu omogućava izlazak iz unutrašnje petlje, a postavljanjem *želi_proces_2* na 0 i iz spoljašnje petlje. Prvi proces onda može da uđe u kritičnu sekciju.

Ovaj algoritam ne dopušta da dva procesa istovremeno uđu u kritičnu sekciju i ne dozvoljava da do zaglavljivanja dođe, što je najvažnije kada je u pitanju zaštita kritične sekcije. Međutim, može se dogoditi da jedan proces prođe dva puta zaredom čime je narušen princip pravednosti. Ako

se prvi proces nalazi u kritičnoj sekciji a drugi najavi ulazak, ukoliko je prvi proces brži on može još jednom da uđe u kritičnu sekciju. Naime, može da se dogodi da prvi proces izađe iz kritične sekcije i da najavi da hoće ponovo u nju, kao i da drugi proces izađe iz aktivnog čekanja (u unutrašnjoj petlji) ali da ne uspe da stigne da postavi želi_proces_2 na 1. U tom slučaju prvi proces, pri proveri (u spoljašnjoj petlji) da li drugi proces želi da uđe u kritičnu sekciju, vidi da on to ne želi (pošto nije stigao da promeni vrednost promenljive *želi_proces_2*) i ponovo ulazi u kritičnu sekciju. Pri tome, drugi proces nastavlja da čeka u svojoj spoljašnjoj petlji jer je sada vrednost promenljive *želi_proces_1* jednaka 1.

Algoritam 3.4. **Dekerov algoritam**

```
Proces 1:
  //Proces najavljuje da želi u
  //kritičnu sekciju
  želi_proces_1 = 1;
  //Ako to želi i drugi...
  WHILE(želi_proces_2 == 1)
    //Proverava se da li drugi
    //ima prednost
    IF (na_redu == 2)
      //Menja se želja
      želi_proces_1 = 0;
      //Proverava se da li drugi
      //ima prednost
      WHILE (na_redu == 2)
        //Aktivno čekanje
      ENDWHILE
      želi_proces_1 = 1;
    ENDIF
  ENDWHILE
  Kritična sekcija
  na_redu = 2;
  želi_proces_1 = 0;
```

```
Proces 2:
  želi_proces_2 = 1;
  WHILE(zeli_proces_1 == 1)
    IF (na_redu == 1)
      želi_proces_2 = 0;
      WHILE (na_redu == 1)
      ENDWHILE
      želi_proces_2 = 1;
    ENDIF
  ENDWHILE
  Kritična sekcija
  na_redu = 1;
  želi_proces_2 = 0;
```

Ograničenje ovog algoritma je i činjenica da je on namenjen isključivo za situacije kada dva procesa konkurišu za iste podatke.

3.2.3. Pitersonov algoritam

Pitersonov algoritam (*Gary L. Peterson*) [6] je predstavljen 1981. godine. Osnovna ideja ovog algoritma (algoritam 3.5.) je u tome da kada proces najavi da želi da uđe u kritičnu sekciju, „džentlmenski" ustupa prednost drugom procesu. Ova prednost važi samo ukoliko i drugi proces želi da uđe u kritičnu sekciju, a u suprotnom prvi proces ne čeka na drugi, već odmah prolazi u kritičnu sekciju.

Algoritam 3.5. **Pitersonov algoritam**

Proces 1:	**Proces 2:**

zeli_proces_1 = 1;
na_redu = 2;
WHILE(*zeli_proces_2* == 1 **AND**
na_redu == 2)
//Aktivno čekanje
ENDWHILE
Kritična sekcija
zeli_proces_1 = 0;
	zeli_proces_2 = 1;
	na_redu = 1;
	WHILE(*zeli_proces_1* == 1 **AND**
	na_redu == 1)
	//Aktivno čekanje
	ENDWHILE
	Kritična sekcija
	zeli_proces_2 = 0;

Da bi ušao u kritičnu sekciju, proces treba da najavi da to želi postavljanjem promenljive *želi_proces* na 1. Zatim on daje prednost drugom procesu i čeka dok drugi proces želi da uđe u kritičnu sekciju i ima prednost. Ovim je obezbeđeno da procesi naizmenično ulaze u kritičnu sekciju, ako to žele u isto vreme, bez obzira na njihovu brzinu. Mana ovog algoritma je da je primenljiv samo za dva procesa.

3.2.4. Lamportov (pekarski) algoritam

Uopštenje Pitersonovog algoritma za *n* procesa predstavlja *Lamportov algoritam* (*Leslie B. Lamport*) [7] koji se često naziva i *pekarski algoritam* (algoritam 3.6.). Algoritam je dobio ovakav „nadimak" jer je zasnovan na ideji koja se oslanja na redosled opsluživanja mušterija u pekari. Naime, kada uđe u pekaru, mušterija dobija broj koji je veći od brojeva koje su dobile prethodne mušterije. Mušterije se uslužuju redom počev od onih sa nižim brojevima.

Pošto se ne može garantovati da dva procesa neće dobiti isti broj jer je i dodela brojeva kritična sekcija, u slučaju da se to dogodi proces sa manjim indeksom se opslužuje prvi.

Za jednostavniji prikaz algoritma potrebno je uvesti sledeće oznake:

- $(A, B) < (C, D)$ ako $A < C$ ili $A = C$ i $B < D$.
- $\max(A_1, A_2, \ldots, A_n) = K$, pri čemu je K takvo da je $K \geq A_i$, $i = 1, 2, \ldots, n$.

Potrebno je kao globalne promenljive deklarisati celobrojni niz (*broj*[*n*]) i niz logičkih promenljivih (*uzima*[*n*]) i pri tome oba niza inicijalizovati na nulu.

Algoritam 3.6. **Lamportov (pekarski) algoritam**

```
Proces i:
  //Zaštita uzimanja broja
  uzima[i] = 1;
  broj[i] = max(broj[0], ..., broj[n – 1]) + 1;
  uzima[i] = 0;
  //Glavna petlja koja je graničnik kritične sekcije
  FOR (j = 0; j < n; j++)
    WHILE (uzima[j] == 1)
      //Aktivno čekanje da j-ti proces dobije broj
    ENDWHILE
    WHILE (broj[j] != 0 AND (broj[j], j) < (broj[i], i))
      //Aktivno čekanje da proces koji ima prednost završi
    ENDWHILE
  ENDFOR
  Kritična sekcija
  broj[i] = 0;
```

Uz pomoć promenljive *uzima*[*i*], štiti se dobijanje broja za *i*-ti proces (*broj*[*i*]) koji dobija vrednost koja je za jedan veća od trenutno najvećeg broja koju imaju ostali procesi (članovi niza *broj*). Zatim se redom (u FOR petlji) obilaze svi procesi, a proces *i* aktivno čeka da sa izvršavanjem u kritičnoj sekciji završe svi procesi (*j*) koji imaju bolji broj od njega (ili niži redni broj u slučaju kada su brojevi isti). Preciznije, proces *i* treba prvo da sačeka da *j*-ti proces završi sa uzimanjem broja ukoliko je u toj fazi, a zatim

i da, ukoliko *j*-ti proces ima prednost, sačeka da on završi sa pristupom kritičnoj sekciji i postavi *broj*[*j*] na nulu.

Ovo rešenje radi za proizvoljan broj procesa, zadovoljena su sva tri uslova koje treba da zadovolji dobro rešenje, a može da radi i na sistemima sa više procesora.

3.2.5. Hardverska rešenja

Hardverska rešenja za zaštitu kritične sekcije podrazumevaju upotrebu posebnih instrukcija procesora. Ideja je da se naprave mašinske instrukcije koje su u stanju da urade bar dve operacije bez mogućnosti prekida, tj. atomično. Najčešće se za potrebe zaštite kritične sekcije koriste sledeće tri instrukcije:

- TAS (Test And Set),
- FAA (Fetch And Add) i
- SWAP (zamena).

Instrukcija TAS operiše sa dve promenljive A = TAS(B) i funkcioniše tako što se vrednost koja se nalazi u promenljivoj B prebacuje u promenljivu A, a u promenljivu B se postavlja 1.

Instrukcija FAA takođe operiše sa dve promenljive. Njena sintaksa je FAA(A, B) i pri njenom korišćenju se vrednost promenljive B prebacuje u A, a vrednost $B + A$ u promenljivu B. Važno je napomenuti da se kod druge operacije u izrazu $B + A$ uzima stara vrednost promenljive A, pre promene vrednosti.

SWAP operiše sa dve promenljive SWAP(A, B) i rezultat je atomična zamena vrednosti promenljivih A i B. Primera radi zamena se, uz ili bez korišćenja pomoćne promenljive, sastoji od bar tri operacije. Zamena vrednosti promenljivih na tri načina ilustrovana je u primeru 3.5.

Postoje različita rešenja za zaštitu kritične sekcije korišćenjem ovakvih instrukcija, ali je u osnovi ideja svih slična. Obično postoji jedna promenljiva kojom se štiti ulaz u kritičnu sekciju, a procesi aktivno čekaju

da se ulaz oslobodi. Kada se to desi, onda atomičnost ovih instrukcija omogućava da tačno jedan proces uđe u kritičnu sekciju.

Primer 3.5. **Zamena vrednosti promenljivih**		
Sa pomoćnom promenljivom:	Bez pomoćne promenljive:	
pom = *A*; *A* = *B*; *B* = *pom*;	*A* = *A* – *B*; *B* = *A* + *B*; *A* = *B* – *A*;	*A* = *A* ^ *B*; *B* = *B* ^ *A*; *A* = *A* ^ *B*;

Algoritam 3.7. prikazuje jedno od rešenja za zaštitu kritične sekcije korišćenjem hardverske instrukcije TAS. Deklariše se globalna promenljiva *zauzeto* i postavi se na nulu, dok svaki proces ima lokalnu promenljivu *ne_može*.

Algoritam 3.7. **Zaštita kritične sekcije (TAS)**

```
Proces i:
  ne_može = 1;
  WHILE (ne_može == 1)
    ne_može = TAS(zauzeto);
  ENDWHILE
  [Kritična sekcija]
  zauzeto = 0;
```

Promenljiva *ne_može* se na početku izvršavanja procesa postavi na 1, a onda se u petlji stalno proverava da li je *zauzeto* postavljeno na 0, tj. da li je ulaz u kritičnu sekciju slobodan. Na ovaj način, kada se oslobodi ulaz (*zauzeto* postane 0) atomičnom operacijom će tačno jedan proces biti obavešten da može da uđe u kritičnu sekciju (*ne_može* će postati 0) i kritična sekcija će se zatvoriti za ostale procese jer TAS operacija *zauzeto* postavlja na 1.

Mana ovakvog rešenja je činjenica da je to rešenje zasnovano na hardverskoj podršci, odnosno da je potrebno da procesor ima ugrađenu tu instrukciju. Sa druge strane, ovo rešenje se bez bilo kakve modifikacije može primeniti na proizvoljan broj procesa, ali ne postoji garancija koliko će neki proces da čeka. „Izgladnjivanje“ procesa, tj. dugo čekanje procesa da uđe u kritičnu sekciju dok drugi procesi prolaze jer imaju više „sreće“, je moguće, ali se u praksi retko događa.

Da bi se obezbedila garancija da će proces konačno dugo čekati na ulazak u kritičnu sekciju, algoritam se mora modifikovati (algoritam 3.8.). Potrebno je deklarisati globalni niz *čeka[n]* čiji se članovi inicijalizuju na nulu i promenljivu celobrojnog tipa *zauzeto* koja se takođe postavlja na nulu.

Algoritam 3.8. **Zaštita kritične sekcije (TAS) – modifikacija**

```
Proces i:
  čeka[i] = 1;
  WHILE (čeka[i] == 1 AND TAS(zauzeto) == 1)
  //Aktivno čekanje dok je kritična sekcija zauzeta
  ENDWHILE
  čeka[i] = 0;
  Kritična sekcija
  //Traži se sledeći proces koji čeka
  j = (i + 1) mod n;
  //Treba ići dok se ne pronađe proces koji čeka ili ne obrne krug i dođe
  //do i-tog procesa
  WHILE (čeka[j] == 0 AND j != i)
    j = (j + 1) mod n;
  ENDWHILE
  IF (i == j)
    //Ako je obrnut ceo krug onda niko ne čeka na ulazak u kritičnu sekciju
    zauzeto = 0;
  ELSE
    //Inače se j-tom procesu prekida aktivno čekanje i on ulazi u
    //kritičnu sekciju
    čeka[j] = 0;
  ENDIF
```

Modifikacija osnovnog TAS algoritma se sastoji u tome da po završetku rada u kritičnoj sekciji, proces koji je završio pronađe sledeći od procesa koji čekaju i da ga „prozove“ da uđe u kritičnu sekciju tako što će mu promenljivu *čeka[j]* postaviti na 0 i time ga osloboditi iz aktivnog čekanja. Ukoliko takav proces ne postoji tj. ukoliko nema procesa koji čekaju da uđu u kritičnu sekciju, promenljiva *zauzeto* se postavlja na 0, čime se ulaz u kritičnu sekciju oslobađa.

Ovo rešenje je sporije od prethodnog jer se i drugi procesi proveravaju i nudi im se da pređu u kritičnu sekciju, a to iziskuje dodatnu potrošnju procesorskog vremena. Sa druge strane, ovaj algoritam je pravedan i garantuje da će svaki proces konačno dugo čekati u redu za pristup kritičnoj sekciji.

Slično kao i uz pomoć TAS operacije, zaštita kritične sekcije se može obezbediti korišćenjem SWAP atomične operacije (algoritam 3.9.).

Algoritam 3.9. **Zaštita kritične sekcije (SWAP)**

```
//Brava je globalna promenljiva inicijalizovana na 0
Proces i:
  ključ = 1;
  WHILE (ključ != 0)
    SWAP (brava, ključ);
  ENDWHILE
  Kritična sekcija
  brava = 0;
```

Brava je globalna promenljiva koja se postavlja na 0 što označava da je prolaz u kritičnu sekciju slobodan, odnosno da je brava bez ključa. Sa druge strane, procesi koji se nadmeću za pristup kritičnoj sekciji imaju promenljivu *ključ* čija vrednost 1 govori da ključ nije u bravi odnosno da čekaju da uđu u kritičnu sekciju. Procesi u petlji pokušavaju da kroz SWAP operaciju ubace ključ u bravu, odnosno da ugrabe trenutak kada je vrednost promenljive *brava* jednaka 0 pa da onda zamenom te vrednosti sa 1, koja se nalazi u promenljivoj *ključ,* dobiju uslov za izlazak iz ciklusa i pristupe kritičnoj sekciji. Istovremeno se brava „zaključava", odnosno promenljiva *brava* dobija prethodnu vrednost promenljive *ključ*, koja je bila 1. Na ovaj način se ulazak u kritičnu sekciju blokira sve dok proces, koji je u njoj, ne završi i postavi promenljivu *brava* na 0 čime se otvara mogućnost da tačno jedan, od procesa koji čekaju, uđe u kritičnu sekciju.

3.3. Rešenja za zaštitu kritične sekcije bez aktivnog čekanja

Sva prethodno prezentovana rešenja su se zasnivala na aktivnom čekanju, odnosno na tome da se u petlji proverava neki uslov i na taj način čeka da se ulaz u kritičnu sekciju oslobodi. Za razliku od ovakve vrste rešenja, postoje i rešenja koja su efikasnija, tj. ne sadrže cikluse koji bespotrebno troše procesorsko vreme. Ove metode se obično zasnivaju na tehnikama koje podrazumevaju zaustavljanje procesa, bez aktivnog čekanja i njihovo pokretanje u odgovarajućem trenutku. Generalno, ideja je da se proces koji ne može da uđe u kritičnu sekciju na neki način blokira pa da se, po sticanju uslova da uđe u kritičnu sekciju, „probudi", odnosno „prozove" i odblokira. Ovakva rešenja su bolja od pristupa sa aktivnim čekanjem, ali je njihova implementacija komplikovanija. Najpoznatiji metodi, kada je ovakav pristup u pitanju, zasnivaju se na korišćenju:

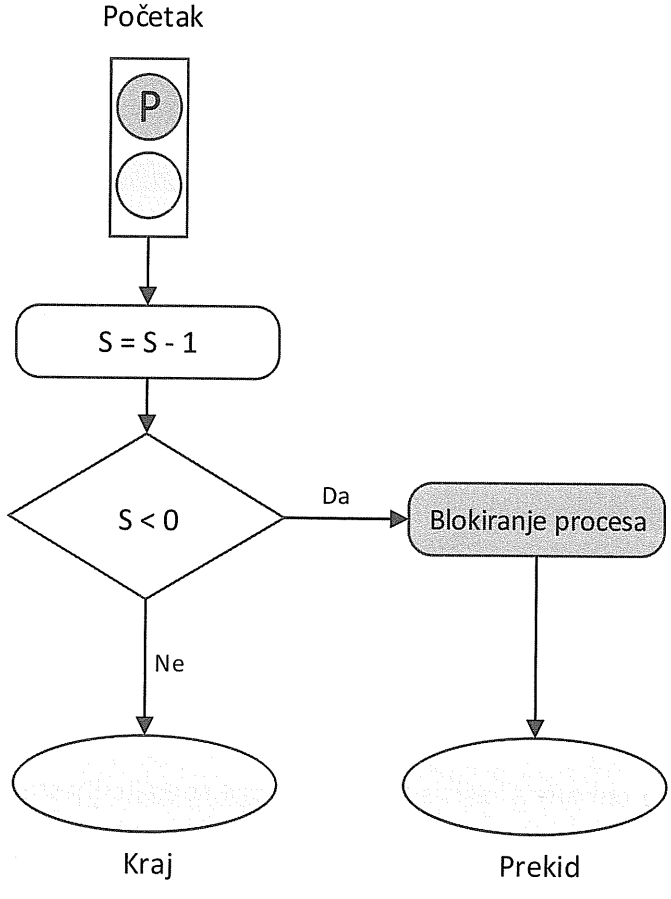
- Semafora;
- Kritičnih regiona;
- Monitora.

3.3.1. Semafori

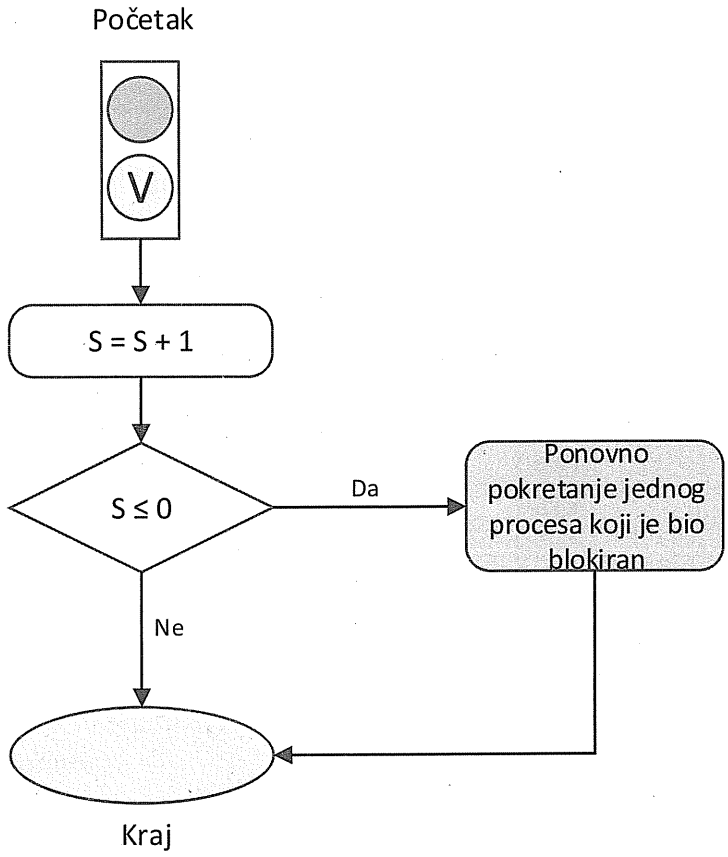
Sinhronizaciju procesa, korišćenjem semafora, osmislio je Edsger W. Dijkstra 1965. godine [8]. *Semafor* je apstraktni tip podataka, odnosno struktura koja može da blokira proces na neko vreme i da ga propusti kada se steknu određeni uslovi. Semafor se može definisati kao specijalna, celobrojna promenljiva nad kojom se, pored standardnih operacija kreiranja i oslobađanja, mogu izvesti samo još dve operacije: *P* i *V*. Ove operacije se u nekim implementacijama označavaju sa *up* i *down* odnosno *wait* i *signal*. Originalna Dijkstrina definicija semafora koristi oznake P i V koje predstavljaju početna slova reči: *Probeer* (pokušaj, probaj) i *Verhoog* (uvećaj, inkrementiraj), na holandskom jeziku. Sintaksa poziva ovih operacija je *P(S)* i *V(S)*, pri čemu je *S* semafor. Operacijom P(*S*) vrednost promenljive S se umanjuje za jedan a zatim se proverava da li je nenegativna i u slučaju da jeste proces (koji je izvršio tu operaciju) se

propušta dalje (obično u kritičnu sekciju). U suprotnom, proces se blokira i stavlja u red za čekanje. Analogno, operacijom V(*S*) se vrednost semafora povećava za jedan i ako postoje procesi koji čekaju blokirani na semaforu *S* (nova vrednost semafora nije pozitivna), propušta se tačno jedan od procesa da nastavi sa izvršavanjem (uđe u kritičnu sekciju), odnosno prekida se njegovo blokiranje.

Slika 3.3. prikazuje sintaksni dijagram operacije P dok se na slici 3.4. može videti dijagram operacije V.



Slika 3.3. Dijagram P operacije

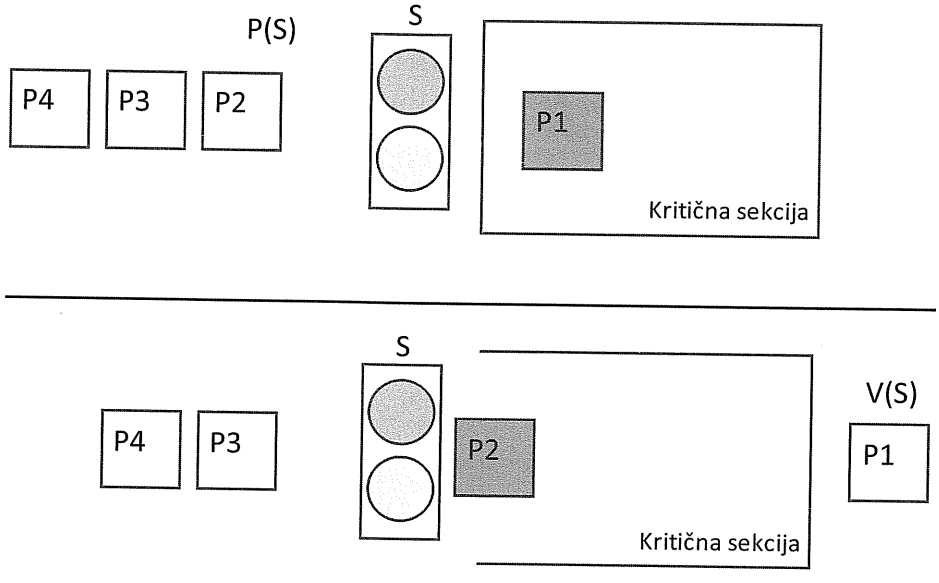


Slika 3.4. Dijagram V operacije

Ovako definisani semafori nazivaju se *opšti* ili *brojački* jer dozvoljavaju da semafor ima proizvoljnu nenegativnu celobrojnu vrednost. Ova vrednost određuje broj procesa koji se mogu propustiti kroz semafor. Sa druge strane, postoje i *binarni* semafori koji mogu samo imati vrednost 0 ili 1.

Korišćenjem semafora zaštita kritične sekcije se najjednostavnije postiže tako što se pre ulaza u kritičnu sekciju postavi binarni semafor (slika 3.5.). Proces koji želi da pređe u kritičnu sekciju testira semafor, pa ako je njegova vrednost 1, on prolazi u kritičnu sekciju, a semafor postavlja na 0. Ako je vrednost 0, to znači da je neki drugi proces u kritičnoj sekciji i da treba da sačeka. Po završetku rada u kritičnoj sekciji, proces izvršava operaciju V(*S*) i time, ukoliko neki od procesa čekaju blokirani na

semaforu, dozvoljava onome koji je na redu da uđe u kritičnu sekciju ili samo postavlja promenljivu *S* na 1 (algoritam 3.10.).



Slika 3.5. Princip rada binarnih semafora

Algoritam 3.10. **Primena semafora**

Proces i:
//Provera semafora
P(*S*);
Kritična sekcija
V(*S*);

Semafori se obično realizuju kao servis operativnog sistema. Za njihovu implementaciju se koriste strukture koje, osim promenljive (*S*), najčešće sadrže i pokazivač na listu koja bi trebalo da čuva podatke o procesima koji su blokirani na semaforu (primer 3.6.).

Primer 3.6. **Semafor**
semafor: **struktura** *vrednost*: **ceo broj**; *lista*: **lista<proces>;**

Pri takvoj realizaciji strukture semafor, operacije P i V se mogu opisati kao u algoritmu 3.11.

Algoritam 3.11. **Opis implementacije semafora**

```
P(S)
  //Semafor se umanjuje za jedan
  S.vrednost = S.vrednost – 1;
  IF (S.vrednost < 0)
    //Ako je Vrednost negativna znači da je drugi proces u kritičnoj sekciji
    // i da ovaj treba blokirati i ostaviti u redu za čekanje
    dodati trenutni proces u S.lista;
    blokirati proces;
  ENDIF

V(S)
  S.vrednost = S.vrednost + 1;
  IF (S.vrednost <= 0)
    //Skida se jedan proces sa liste i prekida njegovo blokiranje
    izbaciti proces iz S.lista;
    odblokirati taj proces;
  ENDIF
```

Na ovaj način se uz pomoć operativnog sistema eliminiše aktivno čekanje. Važno je napomenuti da rešenja koja se zasnivaju na aktivnom čekanju ne moraju uvek da budu loša po sistem i nepoželjna. Naime, ako je aktivno čekanje veoma kratko, ili ako retko do njega dolazi, onda se postavlja pitanje da li je potrebno uklanjati ga, odnosno da li je potrebno da se implementiraju algoritmi koji ga eliminišu. U principu, trebalo bi težiti da se ono ukloni, ali ne i u situacijama kada je cena uklanjanja viša od gubitka vremena koje proces provede u aktivnom čekanju.

Jedan od primera korišćenja semafora je i situacija kada je potrebno da se jedna funkcija izvrši pre druge (primer 3.7.). Neka su f_1 i f_2 funkcije pri čemu je potrebno da se prvo izvrši f_1 pa tek onda f_2. *S* je semafor koji se postavlja na 0.

Ukoliko drugi proces dođe do semafora pre nego što se završi funkcija f_1, on će tu biti blokiran dok se ne izvrši f_1 i zatim prvi proces otvori semafor.

Primer 3.7. **Primena semafora – redosled izvršavanja funkcija**	
Proces 1: f_1(); V(*S*)	**Proces 2:** P(*S*); f_2();

Interesantna je i implementacija brojačkih semafora uz pomoć binarnih koja je prikazana algoritmom 3.12.

Algoritam 3.12. **Implementacija brojačkog semafora korišćenjem binarnih**	

P(*C*)
//Ekskluzivni pristup
P(*S1*);
//Umanjenje brojačke
//promenljive
C--;
//Ako je C negativno semafor
//nije otvoren
IF (*C* < 0)
//Blokira se proces
V(*S1*);
P(*S2*);
ENDIF
V(*S1*);
	V(*C*)
	//Ekskluzivni pristup
	P(*S1*);
	//Uvećanje brojačke promenljive
	C++;
	//Ako C nije pozitivno postoje
	//procesi koji čekaju
	IF (*C* <= 0)
	//Oslobađanje jednog procesa
	//koji čeka
	V(*S2*);
	ELSE
	//Inače se oslobađa semafor
	// koji štiti ekskluzivni pristup
	V(*S1*);
	ENDIF

Za implementaciju su potrebna dva binarna semafora *S1* i *S2* i promenljiva *C* koja je celobrojnog tipa. Prilikom inicijalizacije algoritma vrednost semafora *S1* se postavlja na 1, semafora *S2* na 0, dok promenljiva *C* dobija vrednost *n*. Semafor *S1* se koristi da obezbedi ekskluzivni pristup instrukcijama u okviru aritmetičkih operacija nad promenljivom *C* brojačkog semafora, odnosno da obezbedi atomičnost ovih operacija. Semafor *S2* se koristi za čekanje procesa. Promenljiva *C* čuva vrednost

brojačkog semafora, odnosno određuje koliko procesa semafor može da propusti.

Kada je implementacija operacije P u pitanju, na početku se testira binarni semafor *S1* i ako je zatvoren, tada proces dobija ekskluzivno pravo da pristupi brojačkom semaforu i izvrši operaciju P. Proces prvo umanjuje vrednost promenljive *C* za jedan. Ako je *n* procesa već prošlo kroz semafor (vrednost *C* je manja od nule), onda ovaj proces ne može da prođe, tako da se oslobađa semafor *S1* i blokira se operacijom P(*S2*) na semaforu *S2*. U suprotnom (ako je *C* nenegativno), operacijom V(*S1*) se drugim procesima (koji čekaju na semaforu *S1* pokušavajući da izvrše P ili V operaciju brojačkog semafora) daje šansa da se odblokiraju.

Kod implementacije operacije V, pomoću P(*S1*) obezbeđuje se ekskluzivan pristup brojačkom semaforu. Promenljiva *C* se uvećava za jedan i ako njena vrednost nije pozitivna, to znači da neki proces čeka blokiran na semaforu *S2*. U tom slučaju se operacijom V(*S2*) propušta tačno jedan proces koji je blokiran. U suprotnom se sa V(*S1*) oslobađa semafor *S1* koji štiti ekskluzivni pristup operacijama brojačkog semafora.

3.3.2. Upotreba semafora

U narednom delu će biti prikazani primeri koji ilustruju korišćenje semafora. Odabrani primeri, uz manje modifikacije, obuhvataju široke klase problema koje se često sreću u realnim situacijama. Najpopularniji primeri u literaturi su:

- Problem proizvođača i potrošača;
- Problem filozofa koji večeraju;
- Problem čitača i pisača.

3.3.2.1. Problem proizvođača i potrošača

Problem potrošača i proizvođača je definisan na početku ovog poglavlja. Kao što je već opisano, problem je u istovremenom pristupu promenljivoj *brojač* koja bi trebalo da nosi informaciju o trenutnom stanju u magacinu, tačnije koliko se tačno proizvoda nalazi u njemu.

Primer 3.8. **Proizvođač – potrošač (korišćenjem semafora)**

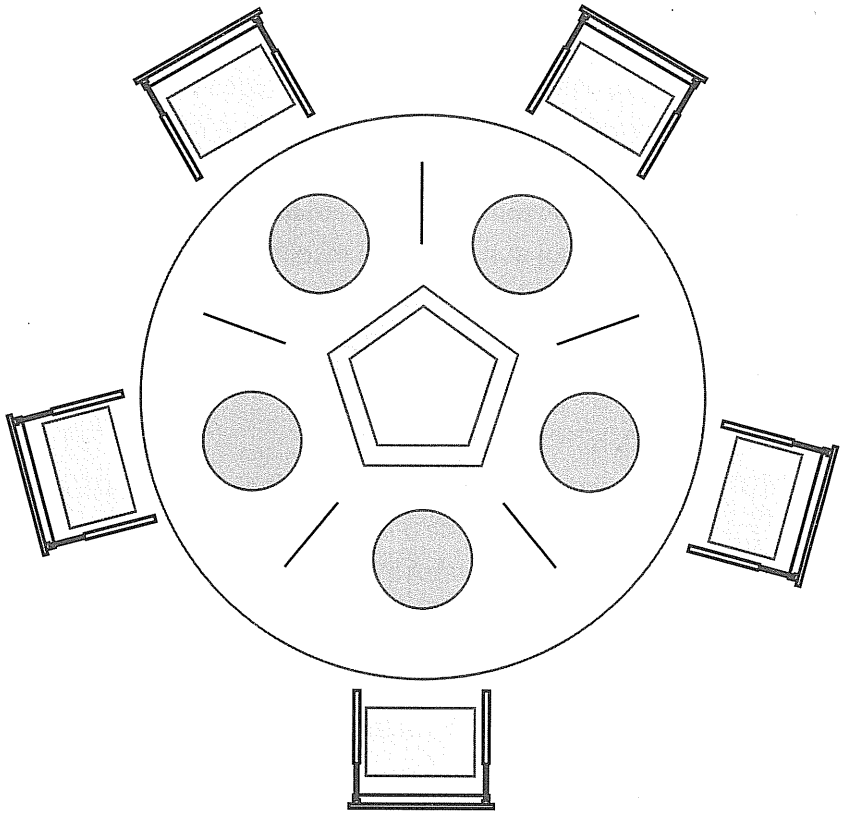
```
Glavni program:
  //Inicijalizacija magacina
  unutra = 0; van=0; brojač=0;
  //Inicijalizacija semafora
  mutex = 1; pun=0; prazan = veličina_magacina;

  //Proizvođački i potrošački proces se izvršavaju paralelno
  PARALLEL WHILE (true)
    proizvođač();
    potrošač();
  END PARALLEL WHILE

proizvođač()
  //Proverava da li ima mesta u magacinu
  P(prazan);
  //Zaštita ekskluzivnog pristupa magacinu
  P(mutex);
  //Novi proizvod se dodaje na prvu sledeću slobodnu poziciju
  magacin[unutra]=novi_proizvod
  //Pomera se indeks na sledeću poziciju
  unutra = (unutra + 1) mod veličina_magacina;
  //Uvećanje brojača za jedan
  brojač++;
  V(mutex);
  V(pun);

potrošač()
  //Provera da li ima proizvoda u magacinu
  P(pun);
  //Zaštita ekskluzivnog pristupa magacinu
  P(mutex);
  //Uzima se najstarije proizvedeni proizvod
  prodaja = magacin[van];
  //Pomera se indeks na sledeću poziciju
  van = (van + 1) mod veličina_magacina;
  //Umanjivanje brojača za jedan
  brojač--;
  V(mutex);
  V(prazan);
```

Za rešavanje problema proizvođača i potrošača korišćenjem semafora potreban je jedan binarni semafor koji bi trebalo da obezbeđuje rad sa magacinom, odnosno da ga štiti od istovremenog pristupa više procesa. Potrebna su i dva brojačka semafora koji će čuvati informaciju o broju praznih i punih mesta u magacinu. Naziv prvog semafora je *mutex* što je čest naziv za semafore koji se koriste za obezbeđivanje ekskluzivnog pristupa. Brojački semafori se nazivaju: *pun* i *prazan*. Semafor *pun* broji popunjena mesta u magacinu, dok prazna mesta broji semafor *prazan.* Iz ovoga sledi da bi *pun* i *prazan* u zbiru trebalo da imaju vrednost jednaku veličini magacina.

Primer 3.8. prikazuje detalje algoritma koji reguliše stanje u magacinu korišćenjem semafora. Proizvođački proces, prvo testiranjem brojačkog semafora *prazan,* proverava da li u magacinu ima prostora. Zatim pokušava da preko semafora *mutex,* koji obezbeđuje ekskluzivni pristup magacinu, u njega smesti novi proizvod. Ukoliko je neki proces već u magacinu, proces će sačekati blokiran na semaforu *mutex*. Slično prethodnom, potrošački proces proverava da li ima proizvoda u magacinu testiranjem brojačkog semafora *pun.* Nakon toga, potrošački proces preko binarnog semafora mutex pokušava da dođe do magacina kako bi uzeo proizvod.

3.3.2.2. Problem filozofa koji večeraju

Izvornu verziju *problema filozofa koji večeraju* (*Dining philosophers problem*) formulisao je Edsger W. Dijkstra 1965. godine (pod nazivom „*Dining Quintuple*") ali je najpoznatiju i najčešće korišćenu formulaciju problema predstavio Toni Hor (*Sir Charles Antony Richard Hoare*). Potrebno je simulirati život filozofa kojima treba obezbediti uslove da razmišljaju i jedu. Konkretno, za okruglim stolom sedi pet filozofa i pred svakim je tanjir, a između svaka dva je štapić (slika 3.6.). Da bi filozof mogao da jede potrebno je da ima dva štapića koja su mu najbliža. Filozofi ili razmišljaju ili jedu, a da bi mogli da jedu pokušavaju da dođu do dva najbliža štapića. Potrebno je simulirati ovakvu situaciju tako da vreme i broj obedovanja budu što ravnopravnije raspodeljeni.



Slika 3.6. Filozofi koji večeraju

Primer 3.9. prikazuje rešenje problema filozofa koji večeraju korišćenjem semafora. Za rešavanje ovog problema potrebno je deklarisati niz semafora *S*[*5*], promenljivu *mutex* za zaštitu ekskluzivnog pristupa i niz *stanje*[*5*] koji će za svakog filozofa čuvati informaciju o stanju u kojem se nalazi.

Na osnovu definicije problema jasno je da se procesi koji predstavljaju filozofe izvršavaju paralelno.

Primer 3.9. **Filozofi koji večeraju (korišćenjem semafora)**

```
Glavni program:
  //Paralelno izvršavanje procesa
  PARALLEL WHILE (true)
    filozof(0);
    filozof(1);
    filozof(2);
    filozof(3);
    filozof(4);
  END PARALLEL WHILE
//Procesi koji simuliraju život filozofa
filozof(i):
  WHILE (true)
    razmišlja();
    uzmi_štapiće(i);
    jede();
    vrati_štapiće(i);
  ENDWHILE
uzmi_štapiće(i):
  P(mutex);
  stanje[i] = GLADAN;
  test(i);
  V(mutex);
  P(S[i]);
vrati_štapiće(i):
  P(mutex);
  //Pošto je večerao filozof može da razmišlja
  stanje[i] = RAZMIŠLJA;
  //Testiraju se susedni filozofi kako bi dobili priliku da dođu do pribora
  test(levi);
  test(desni);
  V(mutex);
test(i):
  IF (stanje[i] == GLADAN AND stanje[levi] != JEDE AND
    stanje[desni] != JEDE)
    stanje[i] = JEDE;
    V(S[i]);
  ENDIF
```

Pri tome se funkcija koja simulira život svakog filozofa implementira tako da u beskonačnim petljama, oni razmišljaju i onda pokušavaju da dođu do štapića i jedu.

U procesu uzimanja štapića prvo se zaštiti proces promene stanja i izvrši provera, tj. test da li susedni filozofi jedu. Ova provera se vrši pozivom funkcije test(*i*) koja u slučaju da susedni filozofi ne jedu, ali i da je *i*-ti filozof gladan, otvara semafor pridružen filozofu *S*[*i*]. Po izlasku iz funkcije test(*i*) proverava se da li filozof ima uslove za jelo, tj. da li mu je ta funkcija dozvolila da prođe semafor. Ukoliko filozof prođe taj semafor, on onda može da jede. U slučaju da ne prođe semafor, filozof ostaje blokiran na njemu čekajući da ga neko od susednih filozofa propusti u trenutku kada bude vraćao štapiće.

Vraćanje štapića je proces u kojem se prvo menja stanje filozofa koji je završio sa jelom tako da on prelazi u stanje RAZMIŠLJA. Zatim se, pozivom funkcije test() za levog i desnog filozofa, od filozofa koji vraća štapiće, daje šansa da neki od njih dođe do štapića koji mu nedostaje, tačnije da se oslobodi sa semafora na kojem eventualno čeka.

3.3.2.3. Problem čitača i pisača

Problem čitača i pisača se odnosi na sinhronizaciju proizvoljnog broja procesa koji konkurentno pristupaju istom resursu, pri čemu neki od procesa žele samo da čitaju (čitači) dok drugi žele i da čitaju ali i da pišu (pisači). Čitači samo čitaju tako da više njih može da čita u isto vreme, dok su pisači zahtevniji jer zahtevaju ekskluzivni pristup, odnosno da tačno jedan može da piše, a da niko ne sme da čita dok traje pisanje. Zadatak je simulirati ovakve procese na što pravedniji način, a da pri tome pristup podacima može da ima: proizvoljan broj čitača ili tačno jedan pisač.

Za rešavanje ovog problema (primer 3.10.) potrebna su dva semafora: *može_da_piše* koji treba da obezbedi prilaz pisačima dok bi *mutex* trebalo da obezbedi ekskluzivan pristup promenljivoj *čitači* koja čuva informaciju o broju čitača.

Primer 3.10. **Čitač – pisač (korišćenjem semafora)**

```
Glavni program:
   čitači = 0;
   može_da_piše = 1;
   mutex = 1;
   PARALLEL WHILE (true)
      čitač();
      pisač();
   END PARALLEL WHILE

čitač()
   P(mutex);
   čitači++;
   //Da li je prvi čitač
   IF (čitači == 1)
      P(može_da_piše);
   ENDIF
   V(mutex);
   čitanje();
   P(mutex);
   čitači--;
   //Poslednji čitač
   IF (čitači == 0)
      V(može_da_piše);
   ENDIF
   V(mutex);

pisač()
   P(može_da_piše);
   pisanje();
   V(može_da_piše);
```

Funkcija (proces) koja simulira rad čitača (čitač()) na početku uvećava broj čitača (*čitači*++), a zatim proverava da li je on jedini čitač u tom trenutku tako što ispituje da li neko trenutno piše. Ako neko piše, semafor će biti zatvoren i čitač će ostati blokiran na njemu, u suprotnom čitač može da čita. Po završetku čitanja proces koji simulira rad čitača, smanjuje broj čitača za jedan i proverava da li je on poslednji. U slučaju da je u pitanju

poslednji proces koji je čitao, on otvara semafor i ukoliko ima pisača koji čekaju na ovom semaforu, omogućava jednom od njih da piše.

Implementacija funkcije koja simulira rad pisača (pisač()) je jednostavnija. Proveri se semafor *može_da_piše* koji vodi računa o dozvoli za pisanje. Ukoliko je taj semafor zatvoren, pisač će tu ostati blokiran. U suprotnom će nastaviti dalje zatvarajući prolaz za pisače, ali i za čitače koji se kasnije pojave. Posle obavljenog pisanja pisač otvara semafor i pruža mogućnost ostalim čitačima i pisačima, koji eventualno na njemu čekaju da pristupe podacima.

Predloženo rešenje favorizuje čitače. Naime, kada prvi čitač uđe u fazu čitanja, bilo koji čitač koji se kasnije pojavi, neće imati nikakvih prepreka da mu se pridruži, dok će pisači morati da sačekaju da svi čitači završe sa čitanjem i da poslednji otvori semafor kako bi mogli da počnu sa radom. Trebalo bi napomenuti da postoje situacije u kojima je ovakav „nedostatak" poželjan jer je potrebno favorizovati čitače.

Ovaj problem se može rešiti modifikacijom prethodnog algoritma koja je prikazana u primeru 3.11. Modifikacija podrazumeva uvođenje semafora *mutex_čitači* koji treba da obezbedi prilaz čitačima i semafora *mutex_2* koji treba da obezbedi ekskluzivan pristup novoj promenljivoj *pisači* koja čuva informaciju o broju pisača.

Čitač pokušava da prođe semafore koji su namenjeni čitačima, uvećava broj čitača i proverava da li je on jedini čitač u tom trenutku. Ukoliko već postoje čitači koji čitaju, on će im se pridružiti. Ako je on jedini čitač a neko piše, semafor *može_da_piše* će biti zatvoren i čitač će ostati blokiran na njemu, zatvarajući pri tome prolaz za ostale čitače na semaforima koje je prethodno prošao. U suprotnom, ukoliko je u pitanju jedini čitač, a nema pisača koji pišu, čitač će osloboditi prolaz ostalim čitačima i početi da čita. Po obavljenom čitanju, čitač umanjuje broj čitača za jedan i proverava da li je on poslednji čitač. Ukoliko jeste, otvara semafor *može_da_piše* kako bi omogućio pisačima da pišu ukoliko su za vreme čitanja ostali blokirani na tom semaforu.

Primer 3.11. **Čitač – pisač (modifikacija)**

```
Glavni program:
  pisači = 0;
  čitači = 0;
  može_da_piše = 1;
  mutex_čitači = 1;
  mutex_1 = 1;
  mutex_2 = 1;
  PARALLEL WHILE (true)
    čitač();
    pisač();
  END PARALLEL WHILE

čitač()
  P(mutex_čitači);
  P(mutex_1);
  čitači++;
  IF (čitači == 1)
    P(može_da_piše);
  ENDIF
  V(mutex_1);
  V(mutex_čitači);
  čitanje();
  P(mutex_1);
  čitači--;
  IF (čitači == 0)
    V(može_da_piše);
  ENDIF
  V(mutex_1);

pisač()
  P(mutex_2);
  pisači++;
  IF (pisači == 1)
    P(mutex_čitači);
  ENDIF
  V(mutex_2);
  P(može_da_piše);
  pisanje();
  V(može_da_piše);
  P(mutex_2);
  pisači--;
  IF (pisači == 0)
    V(mutex_čitači);
  ENDIF
  V(mutex_2);
```

Pisač se implementira tako da prvo pokuša da uveća broj pisača za jedan, a zatim proverava da li je on prvi pisač. Ukoliko jeste, testiranjem semafora *mutex_čitači* proveriće da li ima aktivnih čitača i ako ih nema, nastaviće dalje do semafora *može_da_piše*. U suprotnom, zatvoriće prolaz i za ostale pisače na semaforu *mutex_2*. Pisači koji nisu prvi preskočiće ovu proveru i nastaviti do semafora *može_da_piše*. Ako je semafor zatvoren, pisač će ostati blokiran na njemu, a u suprotnom kreće u pisanje. Po završetku pisanja pisač otvara semafor *može_da_piše* i omogućava drugom pisaču da pristupi podacima, ili čitaču ako pisača više nema.

Ovakvim rešenjem sprečava se stalni ulazak novih čitača jer se oni ostavljaju da čekaju na semaforu *mutex_čitači* ako se pojave nakon nekog od pisača. Međutim, upravo zbog toga dolazi do favorizovanja pisača. Naime, pisači koji se pojave nakon što je jedan aktivan u pisanju se propuštaju direktno na semafor *može_da_piše* čime zaustavljaju čitače dok svi ne završe sa pisanjem.

Rešenje ovog problema, a samim time i pravedno rešenje problema čitača i pisača, podrazumeva korišćenje zajedničkog semafora *red* koji ima zadatak da reguliše pravedan redosled čitanja, odnosno pisanja. Iz tog razloga, semafor *red* bi trebalo da propušta procese strogo prema redosledu u kojem su pristigli. Za potrebe implementacije ovakvog rešenja potrebna su još dva semafora: *mutex* i *može_da_piše*. Analogno prethodnim primerima, semafor *može_da_piše* ima zadatak da kontroliše pristup podacima dok se semafor *mutex* koristi kako bi obezbedio ekskluzivan pristup promenljivoj *čitači*.

Prethodno prikazana rešenja problema čitača i pisača bi trebalo da ilustruju da korišćenje semafora nije jednostavno i da je veoma lako napraviti grešku koju je teško uočiti. Takođe, male promene pozicije semafora mogu dovesti do potpuno drugačijeg ponašanja algoritama u kojima se koriste. Iz prethodno navedenih razloga semafore bi trebalo koristiti veoma oprezno.

Primer 3.12. **Čitač – pisač (pravedno rešenje)**

```
Glavni program:
   čitači = 0;
   može_da_piše = 1;
   mutex = 1;
   red = 1;
   PARALLEL WHILE (true)
      čitač();
      pisač();
   END PARALLEL WHILE

čitač()
   P(red);
   P(mutex);
   IF (čitači == 0)
      P(može_da_piše);
   ENDIF;
   čitači++;
   V(red);
   V(mutex);
   čitanje();
   P(mutex);
   čitači--;
   IF (čitači == 0)
      V(može_da_piše);
   ENDIF
   V(mutex);

pisač()
   P(red);
   P(može_da_piše);
   V(red);
   pisanje();
   V(može_da_piše);
```

3.3.3. Kritični regioni

Semafori su dosta dobro rešenje za zaštitu kritične sekcije, ali pri njihovom korišćenju treba biti veoma oprezan jer se malom greškom mogu napraviti veliki problemi. Naime, jasno je da pogrešna oznaka na početku ili kraju kritične sekcije može tu kritičnu sekciju potpuno otvoriti ili zatvoriti za sve procese.

Rešenje koje daje malo veću fleksibilnost predložili su Brinč Hansen (*Per Brinch Hansen*) i Toni Hor (*Sir Charles Antony Richard Hoare*) 1971. godine u vidu kritičnih regiona. *Kritični regioni* predstavljaju implementaciju zaštite pristupa kritičnoj sekciji na višem programskom jeziku. Kod ovog rešenja potrebno je definisati koju promenljivu deli više procesa i oznakom region obezbediti ekskluzivni pristup toj promenljivoj u okviru niza naredbi koje se nalaze u okviru tog regiona.

Sintaksa kritičnog regiona je prikazana u primeru 3.13.

Primer 3.13. **Kritični region**

```
//Definicija deljene promenljive proizvoljnog tipa
v: deljeni tip;
//Označava se region gde je promenljiva v zaštićena
REGION v
   naredba 1;
   .
   .
   .
   naredba n;
ENDREGION
```

Kritični regioni daju garanciju da, dok se izvršavaju naredbe u okviru njih, nijedan drugi proces ne može da im pristupa ukoliko su pridruženi istoj deljenoj promenljivoj. Takođe, deljenoj promenljivoj se može pristupati isključivo u okviru nekog od regiona gde je ona zaštićena.

Primer kojim se najjednostavnije može ilustrovati primena kritičnih regiona je uvećanje promenljive *v* za jedan (primer 3.14.). Promenljiva se prvo proglasi deljenom, a onda se označi region tj. operacija uvećanja gde

se garantuje ekskluzivni pristup i samim time atomičnost operacije uvećanja.

Primer 3.14. **Atomična operacija uvećavanja**
v: **deljeni ceo broj**; **REGION** *v*   *v* = *v* + 1; **ENDREGION**

Kritični regioni u potpunosti rešavaju problem kritične sekcije, ali je sinhronizacija procesa slaba tačka ovakvog pristupa. Malo napredniji koncepti podrazumevaju takozvane uslovne kritične regione, koji se aktiviraju samo u slučaju kada je određeni logički uslov tačan. Uslovne kritične regione uveo je Toni Hor 1971. godine. Primer 3.15. prikazuje sintaksu uslovnog kritičnog regiona. Ako je uslov ispunjen, proces ima pravo da pristupi regionu (ako neki drugi proces već ne pristupa promenljivoj *v*) i u njemu ima ekskluzivni pristup nad promenljivom *v*. Ako logički uslov nije ispunjen, ulazak procesa u kritični region se odlaže dok uslov ne bude zadovoljen i u kritičnom regionu pridruženom promenljivoj *v* ne bude drugih procesa.

Primer 3.15. **Uslovni kritični region**

v: **deljeni tip**;
REGION *v* **WHILE** (*uslov*)
  naredba 1;
  .
  .
  .
  naredba *n*;
ENDREGION

Primer 3.16. ilustruje primenu uslovnih kritičnih regiona za rešavanje problema potrošača i proizvođača.

Brinč Hansen je 1972. godine predložio još jednu modifikaciju koja pruža dodatne mogućnosti za sinhronizaciju time što dopušta da se privremeno prekine rad u kritičnoj sekciji i proveri novi uslov (korišćenjem naredbe AWAIT). Time se obezbeđuju dodatni mehanizmi za sinhronizaciju jer je

prethodno rešenje imalo samo jednu proveru uslova na početku kritičnog regiona.

Primer 3.16. **Proizvođač – potrošač (korišćenjem kritičnih regiona)**

```
bafer: deljena struktura
  magacin[n]: proizvod;
  unutra, van, broj: ceo broj;

proizvođač()
  REGION bafer WHILE (bafer.broj < n)
    //Stavljanje novog proizvoda u magacin
    bafer.magacin[unutra] = novi proizvod;
    bafer.unutra = (bafer.unutra + 1) mod n;
    bafer.broj++;
  ENDREGION

potrošač()
  REGION bafer WHILE (bafer.broj > 0)
    //Uzimanje proizvoda iz magacina
    proizvod = bafer.magacin[van];
    bafer.van = (bafer.van + 1) mod n;
    bafer.broj--;
  ENDREGION
```

Primer 3.17. **Kritični region (modifikacija)**

```
v: deljeni tip;
REGION v
  naredba 1;
  .
  .
  .
  naredba k;
  AWAIT (uslov)
  naredba k + 1;
  .
  .
  .
  naredba n;
ENDREGION
```

Primer 3.17. ilustruje primenu AWAIT naredbe. Naime, kada proces uđe u kritični region, on izvršava prvih *k* naredbi koje mogu biti i prazne (*k* = 0). Zatim se izračunava logička vrednost promenljive *uslov* pa ako je ona tačna izvršavaju se ostale naredbe počev od *k* + 1 pa do *n*-te. Ako je *uslov* netačan, onda proces napušta kritični region i odlaže svoje izvršavanje sve dok uslov ne postane tačan i da pri tome nijedan drugi proces nije u kritičnom regionu pridruženom promenljivoj *v*.

Korišćenje ovakvih regiona se može ilustrovati primerom koji rešava problem čitača i pisača (primer 3.18).

Primer 3.18. **Čitač – pisač (korišćenjem kritičnih regiona)**

```
v: deljena struktura
   čitači, pisači: ceo broj;
   zauzeto: logička promenljiva;
   //Prilikom inicijalizacije strukture promenljive dobijaju vrednost nula

čitač()
   REGION v
      AWAIT (v.pisači == 0);
      v.čitači++;
   ENDREGION
   čitanje();
   REGION v
      v.čitači--;
   ENDREGION

pisač()
   REGION v
      v.pisači++;
      AWAIT (v.čitači == 0 AND v.zauzeto == 0);
      v.zauzeto = 1;
   ENDREGION
   pisanje();
   REGION v
      v.pisači--;
      v.zauzeto = 0;
   ENDREGION
```

3.3.4. Monitori

Monitori predstavljaju najviši nivo apstrakcije kada je zaštita kritične sekcije bez aktivnog čekanja u pitanju. Naime, oni su konstrukcije programskih jezika u okviru kojih su implementirani mehanizmi za zaštitu kritične sekcije, ali i za sinhronizaciju. Korišćenje monitora predložili su Toni Hor (1974.) i Brinč Hansen (1975.).

Monitori predstavljaju konstrukcije programskog jezika, slične klasama u objektno orijentisanom programiranju, koje mogu da sadrže procedure i promenljive. Glavna odlika monitora je da, u okviru njega, u jednom trenutku može da bude aktivan samo jedan proces, čime se obezbeđuje zaštita bilo kojeg dela programa pa i kritične sekcije. Za potrebe sinhronizacije implementirane su specijalne uslovne (*condition*) promenljive. Nad ovakvim promenljivama su definisane dve operacije: wait i signal, čija sintaksa može da bude: *x*.wait(), odnosno *x*.signal(). Korišćenjem ovih promenljivih i operacija proces se može blokirati operacijom *x*.wait() i kasnije odblokirati signalom *x*.signal() iz drugog procesa. Ovim mehanizmima, kojima se obezbeđuje blokiranje procesa i kasnije ponovno pokretanje kada do potrebnog signala dođe, omogućeno je potpuno rešenje problema pristupa kritičnoj sekciji i sinhronizacije procesa.

Važno je napomenuti da uslovne promenljive monitora nisu ekvivalentne brojačkim semaforima jer kod njih čekanje procesa mora da nastupi pre slanja signala. Naime, ako neki proces čeka na signal, poslati signal će probuditi taj proces. U suprotnom, ako nijedan proces nije blokiran (ne čeka signal), poslati signal će biti izgubljen. Takođe, jedan poslati signal može probuditi tačno jedan proces koji na njega čeka, dok ostali blokirani procesi ostaju da čekaju naredne signale (slika 3.7.).

Primer 3.19. ilustruje sadržaj monitora. Monitori su strukture koje imaju svoje promenljive, procedure i kôd za inicijalizaciju.

Funkcije koje se nalaze u okviru monitora može pozvati tačno jedan proces, dok ostali procesi ne mogu pozvati nijednu funkciju iz monitora dok proces koji je pristupio monitoru ne završi sa radom u njemu. Sa

promenljivama u okviru monitora mogu raditi isključivo procedure iz monitora.

Primer 3.19. **Monitor**

```
monitor: klasa
  S, P: uslovna promenljiva;

  FUNCTION P1()
    ...
  END FUNCTION

  FUNCTION P2()
    ...
  END FUNCTION
  .
  .
  .
  FUNCTION Pn()
    ...
  END FUNCTION

  FUNCTION inicijalizacija()
    ...
  END FUNCTION
```

Monitori su karakteristika programskih jezika jer se implementiraju na tom nivou i pružaju više mogućnosti u odnosu na semafore koji se implementiraju na nivou operativnog sistema i koriste se na komplikovaniji način. Prednost monitora u odnosu na kritične regione leži u mehanizmima (wait i signal) koji olakšavaju sinhronizaciju.

Pristup kritičnoj sekciji se, uz pomoć monitora, veoma lako rešava jer je dovoljno kritičnu sekciju ubaciti u monitor. Prevodilac na poseban način markira monitore i o njima više ne brine programer.

Realizacija monitora uz pomoć semafora bi trebalo da dodatno pomogne u razumevanju kako funkcionišu monitori. Algoritam 3.13. prikazuje zaštitu funkcija u okviru monitora. Za ovakvu implementaciju potreban je semafor *mutex* koji obezbeđuje ekskluzivni pristup monitoru a samim tim

i za svaku funkciju iz monitora. On omogućava da se u svakom trenutku izvršava najviše jedna funkcija koja pripada monitoru. Potreban je još jedan semafor *procesi* koji se koristi za sinhronizaciju procesa koji su u monitoru i promenljiva *brojač_procesa* koja čuva informaciju o broju takvih procesa.

Slika 3.7. Monitori

Algoritam 3.13. **Realizacija monitora uz pomoć semafora**

```
mutex = 1;
brojač_procesa = 0;
procesi = 1;

//Semafor koji obezbeđuje ekskluzivan pristup funkcijama monitora
P(mutex);
...
Pi(...)
...
//Ako neki proces čeka da nastavi rad u monitoru
IF (brojač_procesa > 0)
   //Treba ga propustiti na semaforu
   V(procesi)
ELSE
   //Inače se oslobađa semafor koji štiti prilaz monitoru
   V(mutex)
ENDIF
```

Testira se semafor *mutex*, pa u slučaju da je slobodan proces, izvršava funkciju P_i pri čemu *mutex* semafor garantuje ekskluzivni pristup. Nakon izvršavanja funkcije, proverava se da li postoje procesi koji čekaju da nastave rad u monitoru i ako je to tačno, otvara se semafor na kojem oni

čekaju i propušta sledeći proces. Na ovaj način obezbeđuje se ekskluzivan pristup svakoj od funkcija monitora.

Algoritam 3.14. **Implementacija operacija monitora**

```
x: struktura
   //Semafor u monitoru na kojem će čekati procesi
   sem: semafor;
   //Brojač procesa koji čekaju na semaforu
   brojač: ceo broj;

wait()
   //Uvećava se broj procesa koji čekaju
   x.brojač++;
   //Ako postoje procesi koji su već blokirani na semaforu procesi
   IF (brojač_procesa > 0)
      //Propušta se jedan proces
      V(procesi);
   ELSE
      //Inače se oslobađa semafor koji štiti ekskluzivni pristup monitoru
      V(mutex);
   ENDIF
   //Proces se blokira na semaforu
   P(x.sem);
   //Kada se proces odblokira smanjuje se broj procesa koji čekaju
   x.brojač--;

signal()
   //Ako postoje procesi koji čekaju na signal blokirani na semaforu – u
   //suprotnom nije potrebno slati signal
   IF (x.brojač > 0)
      brojač_procesa++;
      //Propušta se jedan proces koji je bio blokiran na semaforu
      V(x.sem);
      //Suspendovanje procesa jer je upravo poslao signal i odblokirao drugi
      //proces
      P(procesi);
      brojač_procesa--;
   ENDIF
```

Funkcije wait i signal implementiraju se za uslovne promenljive na sledeći način. Potrebno je uslovne promenljive definisati tako da u sebi sadrže semafor *sem* koji će služiti za blokiranje procesa i organizovanje čekanja na semaforu i celobrojnu promenljivu *brojač* koja će čuvati informaciju o broju blokiranih procesa na semaforu *sem* (algoritam 3.14.).

Poziv *x*.wait() uvećava broj procesa koji čekaju na uslovnoj promenljivoj *x* a zatim, u slučaju da već postoji formirani red procesa koji čekaju da dobiju mogućnost da ponovo rade u monitoru, proziva jednog od njih da nastavi sa radom. U svakom slučaju proces koji je pozvao funkciju *x*.wait() se blokira na semaforu pridruženom uslovnoj promenljivoj *x* čekajući da ga neko propusti pozivom *x*.signal(), kada dođe na red. Sa druge strane, poziv *x*.signal(), u slučaju da postoje procesi koji čekaju blokirani na promenljivoj *x*, uvećavaju *brojač_procesa* koji govori o broju procesa koji žele da nastave rad u monitoru, otvara semafor *x.sem* propuštajući sledećeg koji je na redu na semaforu uslovne promenljive x i staje u red za pristup monitoru kada ponovo bude slobodan. Važno je napomenuti da se proces kada pošalje signal odmah blokira, kako bi probuđeni proces imao ekskluzivni pristup monitoru.

Primer korišćenja monitora može se ilustrovati na problemu potrošača i proizvođača (primer 3.20.).

Pun i *prazan* su uslovne promenljive koje služe za blokiranje u situacijama kada proizvođač pokušava da stavi novi proizvod u već pun magacin ili potrošač pokušava da uzme proizvod iz praznog magacina. Dakle, ako je magacin pun (*ima_ih* == *n*) onda se proces blokira (*pun*.wait()). U suprotnom se novi proizvod stavlja u magacin, uvećava brojač koji vodi računa o broju proizvoda u magacinu bez bojazni da će ga neko prekinuti jer je to nemoguće u okviru monitora. Na kraju se šalje signal *prazan*.signal() koji bi trebalo da obavesti procese koji eventualno čekaju, jer je magacin u nekom trenutku bio prazan, da se situacija promenila i da je sada bar jedan proizvod u njemu. Sa druge strane, proces potrošač se blokira (*prazan*.wait()) ukoliko je magacin prazan (*ima_ih* == 0) čekajući na prethodno pomenuti signal proizvođača da je bar jedan proizvod

smešten u magacin. Ukoliko su proizvodi na raspolaganju, potrošač uzima jedan i ažurira stanje. Na kraju se šalje signal proizvođaču (*pun*.signal()) da postoji slobodan prostor u magacinu. Na ovaj način, ukoliko ima spremnih proizvoda koji čekaju da budu smešteni u magacin (koji je u nekom trenutku bio popunjen), šalje se informacija da postoje slobodna mesta i da se bar jedan proizvod može smestiti u magacin.

Primer 3.20. **Proizvođač – potrošač (korišćenjem monitora)**

```
pun, prazan: uslovna celobrojna promenljiva;
ima_ih: ceo broj;

proizvođač()
  //Ako je magacin pun treba sačekati
  IF (ima_ih == n)
    pun.wait();
  ENDIF
  stavi novi;
  ima_ih++;
  //Šalje se signal da postoji bar jedan proizvod u magacinu
  prazan.signal();


potrošač()
  //Ako nema proizvoda treba sačekati
  IF (ima_ih == 0)
    prazan.wait();
  ENDIF
  uzmi novi;
  ima_ih--;
  //Šalje se signal da postoji slobodan prostor u magacinu
  pun.signal();
```

Redovi čekanja procesa na uslovnu promenljivu se obično organizuju po FCFS (First Come First Served) algoritmu, odnosno prednost će imati onaj proces koji je ranije blokiran na uslovnoj promenljivoj. Međutim, naprednije implementacije dozvoljavaju i drugačije pristupe. Toni Hor je 1974. godine uveo *prioritetno čekanje,* odnosno *x*.wait(*c*) gde je *c*

celobrojni izraz koji predstavlja broj prioriteta i koji se izračunava u momentu izvršenja operacije. Kada drugi proces izvrši *x*.signal(), obnavlja se proces sa najmanjom vrednošću broja prioriteta, a u slučaju više takvih procesa, primenjuje se FCFS algoritam za odluku koji proces će imati prednost.

Pri implementaciji monitora javlja se još jedna ozbiljna dilema. Naime, ako u trenutku kada neki proces P signalizira: *x*.signal(), a pri tome postoji proces Q koji je blokiran na uslovnoj promenljivoj *x*, tada treba doneti odluku da li da se suspendovanom procesu Q dozvoli da obnovi svoje izvršenje, a da proces P sačeka ili da se dozvoli obrnuto. U suprotnom bi procesi P i Q bili istovremeno aktivni u monitoru, što narušava uzajamnu isključivost koja je najvažniji uslov kada je zaštita kritične sekcije u pitanju. Dakle, postoje dve mogućnosti:

1. P čeka dok Q ne napusti monitor;
2. Q čeka dok P ne napusti monitor.

Pošto se proces P već izvršava u monitoru, druga mogućnost izgleda logičnije. Međutim, ako se dopusti da P nastavi, uslov za nastavljanje procesa Q ne mora da bude ispunjen kada P završi svoje izvršenje. Toni Hor je prihvatio prvi princip, dok je Brinč Hansen izabrao rešenje koje podrazumeva da kada proces P izvrši operaciju signal odmah napusti monitor tako da Q odmah posle signala nastavlja svoje izvršenje. Pri tome se insistira da poziv signal bude poslednje što se nalazi u okviru funkcije tako da u svakom slučaju proces P završi pre ponovnog pokretanja procesa Q. Ovaj pristup je uslovno slabiji od Horovog pošto na ovaj način proces može da signalizira najviše jednom.

3.3.4.1. Rešenje problema filozofa koji večeraju uz pomoć monitora

Imajući u vidu rešenje problema filozofa koji večeraju uz pomoć semafora, rešenje korišćenjem monitora je zasnovano na istoj ideji, ali je realizacija dosta olakšana zahvaljujući mehanizmima koje poseduju monitori (primer 3.21.).

Primer 3.21. **Filozofi koji večeraju (korišćenjem monitora)**

```
filozof[5]: monitor;
stanje[5]: {RAZMIŠLJA, GLADAN, JEDE};

uzmi_štapiće(i):
   stanje[i] = GLADAN;
   test(i);
   IF (stanje[i] != JEDE)
      filozof[i].wait();
   ENDIF

vrati_ štapiće(i):
   stanje[i] = RAZMIŠLJA;
   //Testira levog i desnog od sebe
   test(levi);
   test(desni);

test(i):
   IF (stanje[levi] != JEDE AND stanje[i] == GLADAN AND
      stanje[desni] != JEDE)
      stanje[i] = JEDE;
      filozof[i].signal();
   ENDIF
```

Pitanja

3.1. Koje su negativne posledice konkurentnog izvršavanja procesa?
3.2. Navesti primere trke za resurse.
3.3. Kako se u računarstvu definiše pojam kritične sekcije?
3.4. Šta bi trebalo da obuhvati kritična sekcija?
3.5. Koje uslove bi trebalo da zadovolji dobro rešenje za zaštitu kritične sekcije?
3.6. Koji su glavni nedostaci rešenja problema zaštite kritične sekcije koja podrazumevaju isključivanje prekida?
3.7. Prodiskutovati rešenja za zaštitu kritične sekcije koja se zasnivaju na aktivnom čekanju.
3.8. Napisati i analizirati algoritam striktne alternacije.
3.9. U kojim situacijama je pogodno koristiti algoritam striktne alternacije?
3.10. Napisati i prodiskutovati Dekerov algoritam za zaštitu kritične sekcije.
3.11. Koja je najveća mana Dekerovog algoritma?
3.12. Napisati i analizirati Pitersonov algoritam za zaštitu kritične sekcije.
3.13. Koji su glavni nedostaci Pitersonovog algoritma i na koji način se oni mogu otkloniti?
3.14. Prodiskutovati karakteristike Lamportovog (pekarskog) algoritma za zaštitu kritične sekcije.
3.15. Koje su najčešće instrukcije koje se hardverski implementiraju kako bi pomogle pri zaštiti kritične sekcije?
3.16. Napisati algoritam za zaštitu kritične sekcije koji se zasniva na korišćenju TAS instrukcije.
3.17. Koje su prednosti i mane algoritma za zaštitu kritične sekcije zasnovanog na korišćenju TAS instrukcije.
3.18. Na koji način se standardni algoritam za zaštitu kritične sekcije korišćenjem TAS instrukcije može modifikovati tako da pruža garancije da neće doći do izgladnjivanja nekog od procesa u sistemu?

3.19. Napisati algoritam za zaštitu kritične sekcije korišćenjem FAA instrukcije.
3.20. Napisati algoritam za zaštitu kritične sekcije korišćenjem SWAP instrukcije.
3.21. Koje su mane hardverskih rešenja za zaštitu kritične sekcije?
3.22. Na koji način funkcionišu metode za zaštitu kritične sekcije koje se ne zasnivaju na aktivnom čekanju?
3.23. Koje su najpoznatije metode za zaštitu kritične sekcije koje se ne zasnivaju na aktivnom čekanju?
3.24. Koji je osnovni princip na kojem funkcionišu semafori?
3.25. Kakvog tipa je promenljiva u okviru semafora?
3.26. Na kom nivou se implementiraju semafori?
3.27. Koji su glavni nedostaci semafora?
3.28. U čemu je razlika između brojačkih i binarnih semafora?
3.29. Na koji način se binarni semafori mogu iskoristiti za zaštitu kritične sekcije?
3.30. Prodiskutovati implementaciju brojačkih semafora korišćenjem binarnih.
3.31. Šta su kritični regioni i kako se implementiraju?
3.32. U čemu je prednost kritičnih regiona u odnosu na semafore?
3.33. Na koji način se rešava problem kritične sekcije kritičnim regionima?
3.34. Kako se rešava problem sinhronizacije procesa korišćenjem kritičnih regiona?
3.35. Šta su uslovni kritični regioni?
3.36. Koji su nedostaci kada je koncept kritičnih regiona u pitanju?
3.37. Objasniti koncept monitora.
3.38. Ko ima prava pristupa uslovnim promenljivama monitora?
3.39. Koje su prednosti monitora u odnosu na ostale koncepte za zaštitu kritične sekcije i sinhronizaciju procesa?
3.40. Na kojem nivou se implementiraju monitori?
3.41. Šta se događa sa poslatim signalima koje ni jedan proces ne čeka?

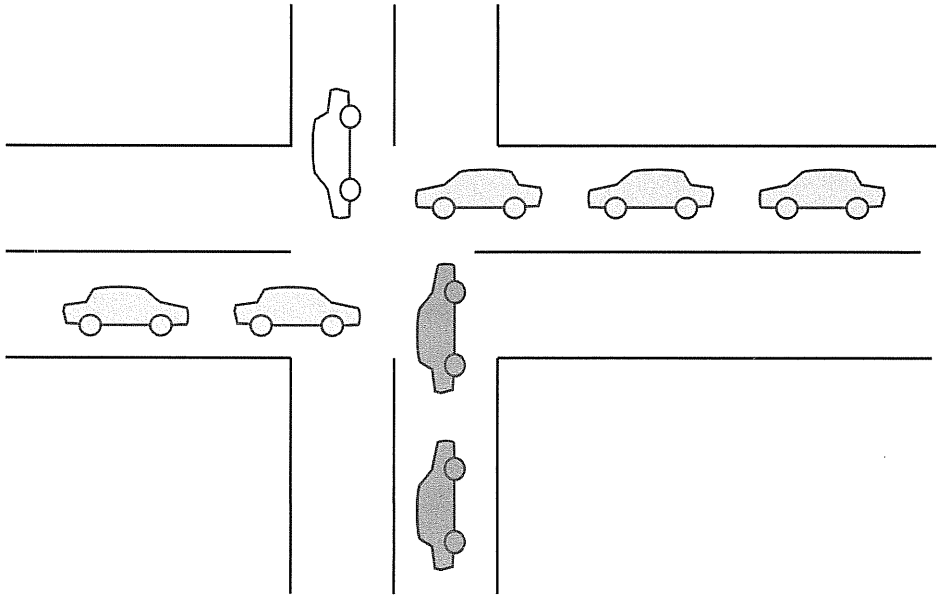
3.42. Analizirati situaciju u monitoru koja nastaje kada jedan proces pošalje signal drugome. Koji od njih treba da nastavi sa radom u monitoru?

3.43. Na koji način se obično organizuju redovi čekanja na uslovnu promenljivu monitora?

3.44. Analizirati rešenja problema filozofa koji večeraju koja se dobijaju korišćenjem semafora i monitora.

4. ZAGLAVLJIVANJE

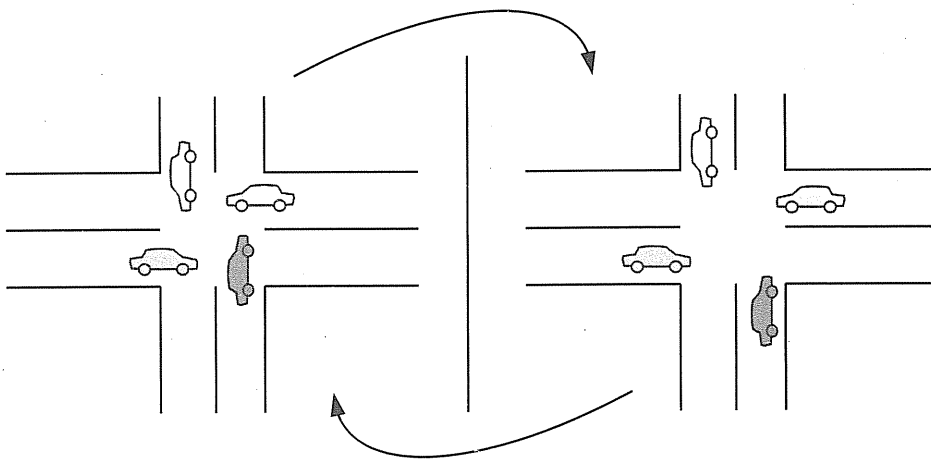
Situacija u kojoj dva ili više međusobno zavisnih procesa blokirani čekaju na resurse koje nikada neće dobiti naziva se *zaglavljivanje* (*deadlock*), odnosno uzajamno blokiranje, mrtvi čvor, mrtva brava i sl. Ovakvo zaglavljivanje je trajno i za posledicu često ima zaglavljivanje čitavog sistema. Takođe, ako sistem sadrži procese koji su zaglavljeni, smatra se da je i on zaglavljen. Slikoviti primeri zaglavljivanja mogu se naći u svakodnevnom životu. Na primer, ukoliko četiri kolone automobila istovremeno sa četiri strane uđu u raskrsnicu (slika 4.1.), pri tome poštujući pravilo desne strane, blokiraće jedni druge tako da će i ostali učesnici u saobraćaju biti zaglavljeni. Takođe, ako bi svaki od pet filozofa (primer iz prethodnog poglavlja), koji bi trebalo da večeraju, istovremeno uzeo po jedan štapić, došlo bi do zaglavljivanja. Nijedan od njih neće spustiti štapić, a pri tome će svaki čekati da to uradi filozof pored njega.



Slika 4.1. Zaglavljivanje

Pojmovi slični zaglavljivanju, koje je centralna tema ovog poglavlja, su i živo blokiranje (*livelock*) i izgladnjivanje (*starvation*). Situacije u kojima

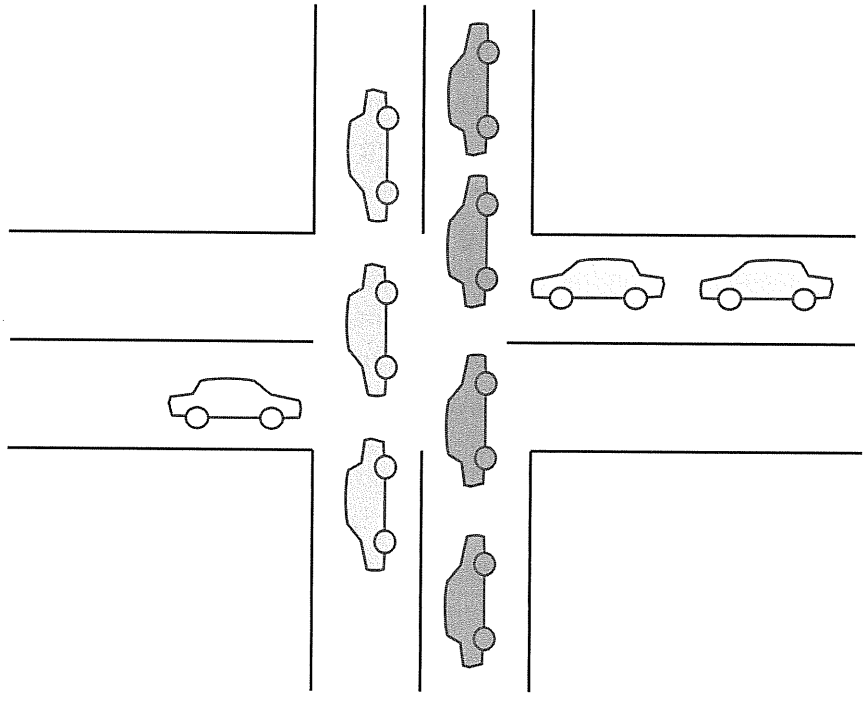
procesi nisu blokirani, ali nemaju napretka u izvršavanju, nazivaju se *živo blokiranje*. Primer ovakve situacije u realnom životu je scena u kojoj u uskom prolazu treba da se mimoiđu dve osobe pri čemu se obe „džentlmenski" istovremeno pomeraju u levu pa u desnu stranu pokušavajući da propuste jedna drugu. Pri tome, osobe ne napreduju, ali ni ne stoje u mestu. Analogno prethodnom primeru, gde je ilustrovano zaglavljivanje automobila u raskrsnici, živo blokiranje bi podrazumevalo situaciju u kojoj se automobili blokiraju u raskrsnici, ali se vraćaju unazad kako bi ponovo pokušali da prođu raskrsnicu. Međutim, pošto ovo rade istovremeno, ponovo se blokiraju i ponavljaju postupak (slika 4.2.).



Slika 4.2. Živo blokiranje

Izgladnjivanje se odnosi na situacije koje podrazumevaju da u sistemu generalno postoji napredak, ali da neki procesi dosta dugo ne napreduju. Na primer (slika 4.3.), ako su automobili koji se kreću u kolonama (vertikalno na slici) na putu sa prvenstvom prolaza, ostali neće imati mogućnost da uđu u raskrsnicu. Međutim, za razliku od zaglavljivanja i živog blokiranja, izgladnjivanje obično nije trajna situacija u sistemu. Naime, često se događa da posle određenog vremena (koje može biti proizvoljne dužine) procesi koji dugo čekaju dobiju priliku da nastave sa izvršavanjem. Kada svi automobili iz kolona, koje se nalaze na putu sa

prvenstvom prolaza iz primera (slika 4.3.) prođu raskrsnicu, stvoriće se uslovi da i ostali automobili prođu raskrsnicu i nastave svojim putem.

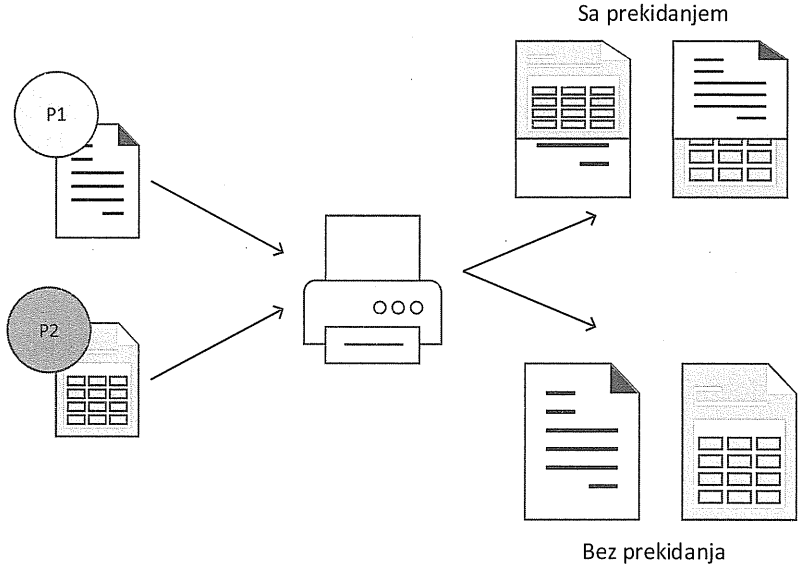


Slika 4.3. Izgladnjivanje procesa

U sistemima koji podržavaju multiprogramiranje i srodne koncepte lako se može dogoditi da dođe do zaglavljivanja. Ukoliko proces pri izvršavanju dospe u stanje *Blokiran* čekajući na resurse koje iz nekog razloga nikada neće dobiti, smatra se da je proces zaglavljen. Pošto se procesi izvršavaju (pseudo) paralelno i konkurišu za resurse čiji je broj ograničen, do zaglavljivanja obično dolazi kada procesi imaju resurse koji su drugima potrebni, a drugi od njih očekuju resurse koji su njima već dodeljeni. Pri tome procesi nemaju nameru da oslobode resurse pre nego što izvrše planirane zadatke. Resursi mogu biti hardverski poput diska, memorije, procesora, skenera itd. i softverski poput semafora, fajlova, slogova u bazama podataka itd. U obe grupe postoje oni koji se mogu i ne mogu „prekidati“, tj. koji se moraju iskoristiti do kraja kada se jednom do njih dođe. Za krajnji rezultat rada procesa nije važno kada će dobiti vreme da

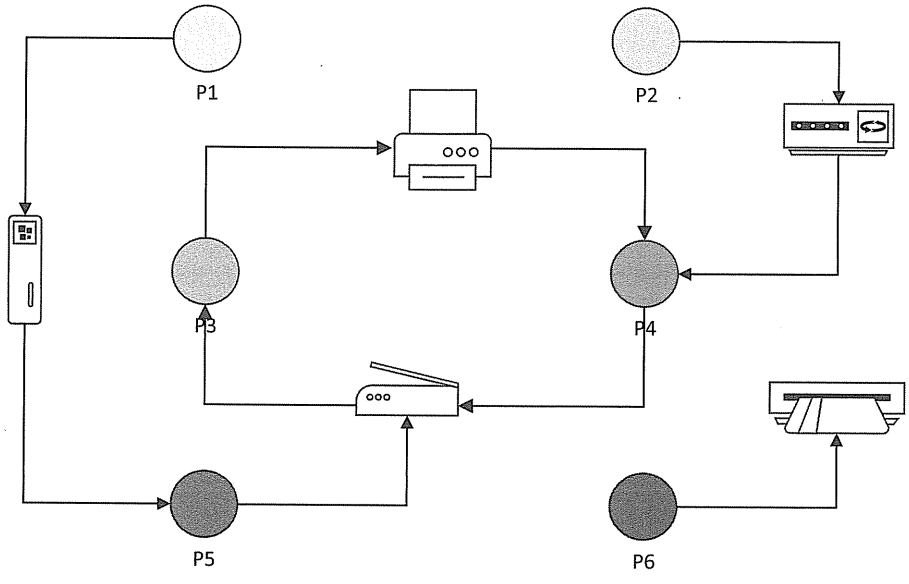
koriste procesor, da li će biti prekidani u toku svog rada i na kom će mestu dobiti deo memorije. Važno je da dobiju neophodnu količinu vremena i memorije za njihovo izvršavanje.

Do zaglavljivanja dolazi pri radu sa resursima koji, kada se dobiju, moraju biti iskorišćeni bez prekidanja, tj. oduzimanja tog resursa i predavanja drugom procesu pre nego što se potpuno iskoriste. Primer ovakve vrste resursa je kritična sekcija. Proces koji je dobio priliku da joj pristupi ne sme biti izbačen iz nje pre njenog završetka, jer bi dopuštanje drugom procesu da joj pristupi moglo da dovede do pogrešnog rezultata. Štampač je očigledan primer hardverskog resursa gde bi prekidanje jednog procesa, pre završetka štampanja i puštanje drugog da nešto odštampa, dovelo do neželjenih rezultata (slika 4.4.).



Slika 4.4. Rezultat štampanja sa i bez prekidanja

Računarski sistemi mogu imati i više instanci nekog resursa (više štampača, skenera, optičkih uređaja, čitača kartica itd.) koje su obično nezavisne jedna od druge, tako da će se u daljem tekstu one razmatrati kao zasebni resursi, osim ako se ne naglasi drugačije.



Slika 4.5. Kružno čekanje

Da bi došlo do zaglavljivanja, neophodno je da su u sistemu istovremeno ispunjena sledeća četiri uslova:

1. *Uzajamno isključivanje* (*Mutual Exclusion*) – podrazumeva da u jednom trenutku tačno jedan proces može da koristi resurs, dok ostali moraju da sačekaju njegovo oslobađanje.
2. *Čekanje i držanje* (*Wait and Hold*) – dok proces drži resurse sa kojima radi, može da zahteva nove koji su mu potrebni za dalje izvršavanje i da čeka one koji trenutno nisu raspoloživi.
3. *Nemogućnost prekidanja* (*No Preemption*) – podrazumeva da operativni sistem nema pravo da oduzme resurse koje je dodelio procesu i kasnije mu ih vrati, kako bi te resurse, u međuvremenu, dodelio drugim procesima. Dakle, resurse može osloboditi samo proces koji ih je dobio na korišćenje.
4. *Kružno čekanje* (*Currier wait*) – podrazumeva da u sistemu može da postoji lanac procesa koji čekaju jedni na druge, tj. na resurse koji su im dodeljeni, čineći pri tome krug (slika 4.5.). Dakle, svaki proces drži resurse koji su mu dodeljeni i čeka na druge koje drži

sledeći proces u nizu i tako do poslednjeg koji zahteva resurse koje drži prvi proces.

Navedene uslove je 1971. godine prvi put predstavio Edvard Kofman (*Edward G. Coffman, Jr.*) pa se iz tog razloga često nazivaju Kofmanovi uslovi [9].

Pošto je za zaglavljivanje potrebno da važe svi pomenuti uslovi, do zaglavljivanja neće doći ako se bilo koji od ovih uslova ukloni.

Postoje razni pristupi kada je u pitanju odnos prema zaglavljivanju, ali se na osnovu mera koje se primenjuju, mogu svrstati u sledeće četiri grupe:

- *Sprečavanje* (*Deadlock prevention*) – pristup koji podrazumeva sistemske mere koje se implementiraju u operativni sistem i kojima se isključuje mogućnost zaglavljivanja.
- *Izbegavanje* (*Deadlock avoidance*) – dinamičke mere koje se preduzimaju kako bi se sistem vodio kroz stanja koja obezbeđuju da ne dođe do zaglavljivanja. U odnosu na sprečavanje, izbegavanje je liberalnija mera, ali se i ovim pristupom garantuje da do zaglavljivanja neće doći.
- *Detekcija i oporavljanje* (*Deadlock detection and recovery*) – pristup koji dozvoljava da do zaglavljivanja dođe, ali ima mehanizme za otkrivanje da li je do zaglavljivanja došlo, tretman procesa koji su zaglavljeni i eventualno spašavanje dela rezultata koji su dobijeni pre zaglavljivanja u slučajevima kada je to moguće.
- *Nepreduzimanje bilo čega* (*Ignoring deadlock*) – mera koja je prihvatljiva na sistemima na kojima do zaglavljivanja retko dolazi ili u situacijama kada se ne radi o veoma važnim poslovima. Ovo je najjednostavnije rešenje koje ne pruža nikakve mogućnosti i uopšte ne vodi računa o zaglavljivanju. Kada do njega dođe, najčešće se problem rešava ponovnim pokretanjem procesa ili čitavog sistema u nadi da do zaglavljivanja neće doći.

4.1. MERE ZA SPREČAVANJE ZAGLAVLJIVANJA

Mere za *sprečavanje zaglavljivanja* su statičke, sistemske mere koje obično nisu algoritamski zahtevne, a za cilj imaju da u startu definisanjem pravila za izvršavanje procesa, eliminišu bar jedan od uslova koji dovode do zaglavljivanja. Najčešće su troškovi ovih algoritama mali, ali su zato ograničenja velika. Jasno je da se prvi Kofmanov uslov ne može ukloniti tako da se mere mogu primeniti na preostala tri uslova.

4.1.1. Prevencija čekanja i držanja

Glavna ideja za eliminisanje uslova *čekanje i držanje* je uvođenje ograničenja koje onemogućava da proces drži neke resurse dok traži nove. Obično se koriste dva pristupa za rešavanje ovakvih problema.

Prvi pristup podrazumeva da pre nego što počne sa izvršavanjem, proces zahteva da mu se dodele svi potrebni resursi i da u izvršavanje krene tek kada ih sve dobije. Loša strana ovakvog pristupa je to što za vreme svog izvršavanja proces drži sve potrebne resurse i time onemogućava druge procese da ih koriste u trenucima kada su njemu nepotrebni. Na primer, ako proces treba da pročita fajl sa fleš memorije, snimi ga na disk, modifikuje i na kraju odštampa, ovakvim pristupom proces bi na početku zahtevao sve potrebne resurse (fleš memorija, disk, štampač) i držao ih do kraja, tj. završetka štampanja. Štampač bi u tom slučaju bio nedostupan ostalim procesima i u trenucima dok proces prenosi podatke ili dok se oni modifikuju. Takođe, često na početku nije poznato koji će resursi biti potrebni procesu u toku izvršavanja, što dodatno otežava upotrebu ovakvog algoritma.

Drugi pristup rešava problem prethodnog jer podrazumeva da proces traži resurse u trenucima kada su mu potrebni, ali da pre svakog uzimanja novih resursa ima obavezu da one koje drži vrati operativnom sistemu kako bi operativni sistem doneo odluku da li će ih ponovo dodeliti njemu ili ih ustupiti nekom drugom procesu. Na ovaj način postiže se da proces ne drži nijedan resurs u trenutku kada traži nove. Kada je prethodni primer u pitanju, proces bi prvo zatražio fleš memoriju i disk kako bi

obavio transfer. Na kraju bi oslobodio te resurse, pa zatražio štampač. Štampač bi tako sve do poslednje operacije procesa (štampanja) bio na raspolaganju drugim procesima.

Iz prethodno navedenog jasno je da se primenom ovakvih mera eliminiše mogućnost da uslov čekanja i držanja bude ispunjen i da do zaglavljivanja ne može doći. Neke od mana ovakvog pristupa su slabije iskorišćenje resursa, ali i moguće „izgladnjivanje" procesa koji ne mogu da dođu na red jer su u svakom trenutku neki od potrebnih resursa nedostupni.

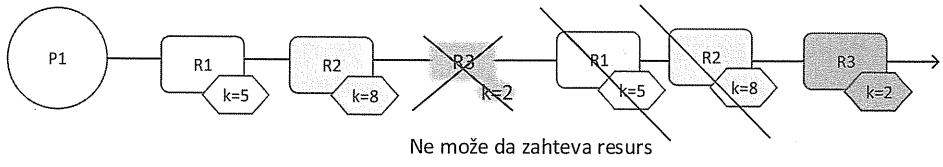
4.1.2. Eliminisanje nemogućnosti prekidanja

Mere kojima se uklanja uslov *nemogućnost prekidanja* se zasnivaju na ideji da se, u slučaju kada se potrebe procesa ne mogu zadovoljiti, otpuštaju svi resursi koje je držao. Naime, kada proces drži neke resurse a zahteva druge koji mu se ne mogu odmah dodeliti, on otpušta sve resurse koje drži i ustupa ih drugim procesima na korišćenje. Pri tome, oduzeti resursi se dodaju na listu onih koje proces čeka. Proces će biti ponovo pokrenut u trenutku kada povrati svoje stare resurse, ali i dobije nove koje je zahtevao. Jasno je da ovakav pristup sigurno ne dovodi do zaglavljivanja, ali i da se vrlo lako može dogoditi da zahtevniji procesi dugo čekaju da dođu u posed svih resursa koji su im potrebni.

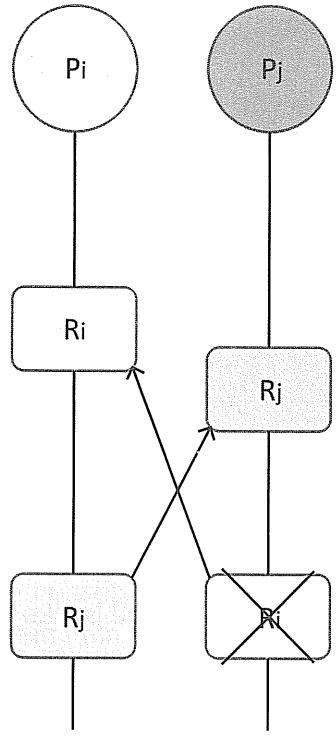
4.1.3. Prevencija kružnog čekanja

Da do *kružnog čekanja* ne dođe, obično se koriste rešenja koja podrazumevaju označavanje, tj. enumeraciju tipova (klasa) resursa. Ideja je da se svakom resursu dodeli jedan broj, a da proces koji drži određene resurse može da zahteva samo one resurse kojima je dodeljen veći broj od brojeva njegovih resursa. Na primer, neka je $R = \{R_1, ..., R_n\}$ skup svih resursa u sistemu. Svakom resursu se dodeljuje jedan ceo broj $F(R_i) = k$. Ako je R_i resurs sa najvećim brojem koji drži dati proces, u svom daljem izvršavanju taj proces može zahtevati samo resurse R_j takve da je $F(R_j) > F(R_i)$. Pri tome, proces resurse nižeg ranga (manje vrednosti dodeljenog broja) može zahtevati u trenutku kada otpusti sve resurse višeg ranga.

Slika 4.6. ilustruje dodelu resursa procesu po prethodno definisanim pravilima. Proces zahteva i dobija resurs čiji je broj 5, a zatim i resurs čiji je broj 8 (jer je 5 < 8). Međutim, tada proces ne može da zahteva resurs čiji je dodeljeni broj 2 jer je 2 manje od maksimuma brojeva resursa koje proces već drži. Iz tog razloga, proces mora da oslobodi resurse koje drži kako bi imao pravo da zatraži i (kada se steknu uslovi) dobije novi resurs da bi bio u mogućnosti da nastavi izvršavanje.



Slika 4.6. Pravo zahteva za resurse



Slika 4.7. Sprečavanje kružnog čekanja

Kod ovakvog pristupa javljaju se teškoće oko numerisanja, tj. dileme kako je najbolje napraviti enumeraciju. Takođe, jedan od problema je i potreba

za ponovnom enumeracijom svaki put kada se u sistemu pojavi novi resurs. I pored navedenih problema, primenom ovakvih mera se sigurno onemogućava kružno čekanje. Naime, neka proces P_i drži resurs R_i iz klase i, a proces P_j resurs R_j iz klase j (slika 4.7.) i pri tome važi da je $F(R_j) > F(R_i)$, onda do kružnog čekanja na resurse između ova dva procesa ne može doći jer bi to značilo da P_j traži nešto što je u klasi koja je ispod resursa iz klase koje drži.

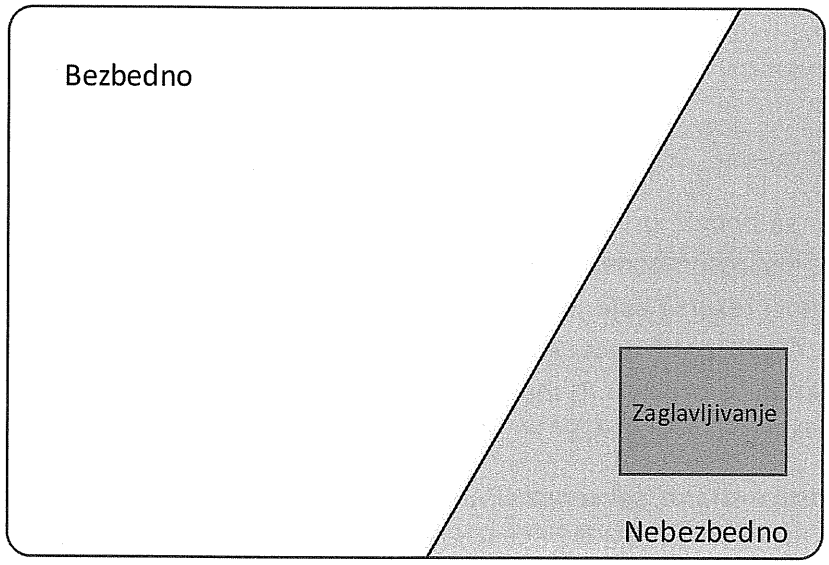
4.2. Mere izbegavanja

Za razliku od mera sprečavanja, koje u startu eliminišu neki od neophodnih uslova da do zaglavljivanja dođe, *mere izbegavanja* su dinamičke mere koje se preduzimaju u određenim momentima u zavisnosti od stanja u sistemu. Iz tog razloga troškovi algoritma su obično veći nego kada su mere sprečavanja u pitanju, ali se sa druge strane povećava efikasnost sistema jer se resursi koriste optimalnije.

4.2.1. Bankarev algoritam

Postoje različiti pristupi za izbegavanje zaglavljivanja. Jedan veoma efikasan, a pri tome i ne previše komplikovan, je *Bankarev algoritam* (*Banker's Algorithm*) [8], koji je 1965. godine predložio Edsger W. Dijkstra. Ime je dobio po principu davanja zajmova u bankama. Naime, korisnici traže od banke kredite kako bi mogli da realizuju svoje projekte. Banka ima ograničene resurse, a korisnici znaju kolika je maksimalna količina novca koja im je potrebna. Vremenom, korisnici traže određenu količinu novca (kredite), a banke procenjuju da li će im i u kojem trenutku izaći u susret. Novac banci mogu vratiti samo korisnici koji su uspeli da završe svoje projekte tako što su prethodno od banke dobili (kroz jedan ili više kredita) maksimalnu prijavljenu količinu novca. Vraćeni novac je onda, uz novac koji je ostao u banci, na raspolaganju svim ostalim korisnicima koji nisu završili svoje projekte. Međutim, ukoliko banka ne napravi dobru procenu može se dogoditi da ostane bez novca i da zbog toga neki od korisnika ne budu u mogućnosti da završe svoj projekat. Ta situacija je analogna zaglavljivanju pri radu sa procesima.

U skladu sa prethodnim, Bankarev algoritam podrazumeva da je pre početka izvršavanja svakog od procesa poznat maksimalni broj resursa svakog tipa koji će mu biti potreban. Ideja je da se dinamički ispituje pokušaj dodele resursa kako bi se obezbedilo da nikada ne dođe do kružnog čekanja. Stanje dodele resursa je definisano brojem raspoloživih, dodeljenih resursa i maksimalnim zahtevima procesa. Smatra se da je stanje sistema *bezbedno* ako sistem može u određenom redosledu da dodeljuje resurse kako bi svaki proces mogao da završi sa radom, tj. da ne dođe do zaglavljivanja, a da se pri tome svi procesi izvrše. Redosled kojim se vrši ovakva dodela resursa procesima naziva se bezbedna sekvenca.



Slika 4.8. Bezbedno i nebezbedno zaglavljivanje

Definicija: Sekvenca procesa $<P_1, P_2, ..., P_n>$ je *bezbedna* za tekuće stanje dodele resursa akko se potrebe svakog procesa P_i mogu zadovoljiti slobodnim resursima i resursima koje trenutno drže procesi P_j ($j < i$).

Ako potrebni resursi za proces P_i nisu odmah dostupni, tada proces P_i mora da sačeka dok se svi P_j ($j < i$) ne izvrše. Kada procesi P_j ($j < i$) završe sa radom, P_i dobija potrebne resurse, izvrši se, a zatim vrati sve resurse koje je koristio. Kada završi proces P_i tada P_{i+1} može da dobije potrebne resurse i tako dalje.

Definicija: Sistem je u *bezbednom stanju* akko postoji bezbedna sekvenca.

Važno je napomenuti da, ako stanje nije bezbedno, to ne znači da će sigurno doći do zaglavljivanja, već da postoji mogućnost da do njega dođe (slika 4.8.).

Primer 4.1. Neka sistem ima ukupno 10 instanci određenog resursa i neka su u tabeli 4.1. date informacije o zahtevima i trenutnom rasporedu dodeljenih instanci tog resursa procesima.

Tabela 4.1. Situacija u sistemu

	Maksimalne potrebe	Trenutno stanje
P_1	7	3
P_2	5	2
P_3	9	2

Sekvenca $<P_2, P_1, P_3>$ je bezbedna. Naime, ako se tri neraspoređena resursa dodele prvo procesu P_2, time će se zadovoljiti njegove potrebe tako da će on u nekom trenutku završiti sa radom i osloboditi svih 5 instanci resursa koje je imao. Od tih 5 instanci, 4 će zadovoljiti potrebe procesa P_1, pa će on na kraju otpustiti svih 7 instanci resursa koje je imao. Na kraju se procesu P_3 može dodeliti 7 resursa koji su mu potrebni od 8 koji su na raspolaganju.

Sa druge strane, sekvenca $<P_1, P_2, P_3>$ je nebezbedna, jer je očigledno da ako proces P_1 dobije sva tri raspoloživa resursa u prvom trenutku, to neće zadovoljiti njegove potrebe, a ostali procesi neće imati priliku da dobiju resurse kako bi zadovoljili svoje.

Osnovna ideja Bankarevog algoritma je da se resursi procesima dodeljuju tako da se sistem sprovodi kroz niz bezbednih stanja kako bi se izbegla mogućnost da dođe do zaglavljivanja. Kao što je već napomenuto, banke vode računa da novac rasporede tako da zadovolje potrebe što većeg broja svojih klijenata. One će se truditi da kreditiraju perspektivne klijente i ne bi trebalo da dopuste da se novac dodeli jednom ili grupi klijenata, a da pri tome ne ostane sredstava za ostale. Analogno, kada je Bankarev algoritam u pitanju, pri dodeli resursa vodi se računa o tome da sistem

uvek bude u bezbednom stanju, odnosno, po zahtevu procesa za dobijanje resursa vrši se provera da li postoji mehanizam kojim će se novododeljeni zajam kasnije (kada proces kojem je dodeljen završi svoj rad) vratiti i biti na raspolaganju drugim procesima.

Za prikaz ovog algoritma potrebno je definisati sledeće promenljive:

n – broj procesa,

m – broj resursa.

Niz *raspoloživo* dužine m ukazuje na broj raspoloživih resursa određenog tipa.

raspoloživo[j] = k, označava da ima k raspoloživih instanci resursa R_j.

Matrica *max* dimenzije $n \times m$ definiše maksimalne zahteve svakog procesa.

max[i,j] = l – proces P_i zahteva najviše l instanci resursa R_j.

Matrica *dodeljeno* dimenzije $n \times m$ definiše broj resursa svakog tipa koji je trenutno dodeljen svakom procesu.

dodeljeno[i,j] = h – proces P_i je trenutno u posedu h instanci R_j.

Matrica *potrebno* dimenzije $n \times m$ prikazuje preostale potrebe svakog procesa za resursima.

potrebno[i,j] = s – procesu P_i je potrebno s instanci resursa R_j.

Iz prethodnih definicija može se zaključiti da matrica *potrebno* predstavlja razliku matrica *max* i *dodeljeno*.

potrebno[i,j] = *max*[i,j] – *dodeljeno*[i,j].

Da bi se jednostavnije objasnili neki od algoritama, promenljive *dodeljeno*[i] i *potrebno*[i] označavaju nizove dužine m koji predstavljaju sve dodeljene i potrebne resurse za proces i. Drugim rečima, ovi nizovi su, u stvari, vrste odgovarajućih matrica.

Potrebno je definisati i nejednakost dva niza iste dimenzije. Tako je niz *A* „manji ili jednak" od niza *B* ako svaki njegov element ima manju ili bar jednaku vrednost od vrednosti odgovarajućeg elementa niza B:

$$A \leq B, \text{ako } \forall i \in \{1,2,\dots,n\}, A[i] \leq B[i].$$

Sa ovako definisanim promenljivama može se ilustrovati algoritam kojim se proverava da li je sistem u bezbednom stanju (algoritam 4.1.).

Neka su *radni* i *gotov* nizovi dužine *m* i *n*.

Algoritam 4.1. **Provera stanja**

```
//Inicijalizacija
radni = raspoloživo;
gotov[i] = 0, za i = 0, 1, ..., n – 1;
sledeći_proces_u_sekvenci:
//Provera da li postoji proces koji može biti zadovoljen raspoloživim
//resursima
IF (ne postoji i takvo da je gotov[i] == 0 i potrebno[i] ≤ radni)
   idi na kraj;
ENDIF
//Pronalazi se proces koji može biti zadovoljen raspoloživim resursima
Odaberi prvo i takvo da je gotov[i] == 0 i potrebno[i] ≤ radni;
//Radnom skupu se dodaju resursi procesa koji ima mogućnost da se izvrši
radni = radni + dodeljeno[i];
gotov[i] = 1;
idi na sledeći_proces_u_sekvenci;
kraj:
//Ako su svi procesi imali mogućnost sa se izvrše – stanje je bezbedno
IF (gotov[i] == 1 za svako i)
   bezbedno_stanje = true;
ELSE
   bezbedno_stanje = false;
ENDIF
```

Niz *radni* se inicijalizuje tako da bude jednak nizu raspoloživih resursa u prvom trenutku izvršavanja. Ovaj niz predstavlja niz resursa koji su na raspolaganju procesima. Niz *gotov*, koji nosi informaciju o tome da li *i*-ti proces ima uslova da se izvrši sa raspoloživim resursima, inicijalizuje se na

nulu. U koraku *sledeći_proces_u_sekvenci* se traži proces koji nije imao uslove da se izvrši (*gotov[i] == 0*), a u trenutnom radnom nizu postoje raspoloživi resursi koji su mu potrebni da bi se izvršio. Ako takav proces postoji, u sledećem koraku se niz *radni* proširuje sa resursima koje je *i*-ti proces držao jer se pretpostavlja da on može da završi sa radom i oslobodi dodeljene i svoje resurse. U tom slučaju se vrednost *gotov*[*i*] postavlja na 1 i ponovo, sada sa većim radnim nizom, traži proces koji može biti zadovoljen. Ako u koraku *kraj* ne postoji proces koji zadovoljava postavljene uslove, to znači da je algoritam završio sa radom i da treba proveriti da li je sistem u bezbednom stanju ili ne. Ukoliko se prolazom kroz niz *gotov* utvrdi da su svi procesi imali uslove da dobiju potrebne resurse, iskoriste ih i vrate, onda je sistem u bezbednom stanju. U suprotnom sledi da nije pronađena bezbedna sekvenca i da zbog toga sistem nije u bezbednom stanju.

Primer 4.2. Neka su raspoloživi resursi prikazani u tabeli 4.2.

Tabela 4.2. Raspoloživi resursi

A	B	C	D
3	1	1	2

Resursi koje drže procesi su dati u tabeli 4.3.

Tabela 4.3. Resursi koje drže procesi

	A	B	C	D
P_1	1	2	2	1
P_2	1	0	3	3
P_3	1	1	1	0

Maksimalne potrebe procesa su date u tabeli 4.4.

Tabela 4.4. Maksimalne potrebe procesa

	A	B	C	D
P_1	3	3	2	2
P_2	1	2	3	4
P_3	1	1	5	0

Potrebe procesa su date u tabeli 4.5.

Tabela 4.5. Potrebe procesa

	A	B	C	D
P_1	2	1	0	1
P_2	0	2	0	1
P_3	1	1	4	0

Dakle, ako se procesu P_1 dodeli ono što on traži (2, 1, 0, 1) on će, kada završi sa izvršavanjem, vratiti i resurse koje je posedovao pre dodele tako da će na raspolaganju biti 4 resursa tipa A i po 3 resursa tipa B, C i D. Proces P_2 će tada moći da se zadovolji, a potom i proces P_3.

Primer 4.3. U sistemu je dato 5 procesa P_1, P_2,..., P_5 i 3 tipa resursa: *A* (10 instanci), *B* (5 instanci), *C* (7 instanci). Neka je u tabeli 4.6. dato stanje u nekom trenutku.

Tabela 4.6. Stanje sistema

	Dodeljeno			***Max***			***Raspoloživo***		
	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3	3	3	2
P_2	2	0	0	3	2	2			
P_3	3	0	2	9	0	2			
P_4	2	1	1	2	2	2			
P_5	0	0	2	4	3	3			

Sadržaj matrice *Potrebno* (tabela 4.7.) je određen kao razlika matrica *Max* i *Dodeljeno*.

Tabela 4.7. Matrica Potrebno

	Potrebno		
	A	B	C
P_1	7	4	3
P_2	1	2	2
P_3	6	0	0
P_4	0	1	1
P_5	4	3	1

Sistem je u bezbednom stanju jer je sledeća sekvenca <P_2, P_4, P_5, P_3, P_1> bezbedna.

Kada sistem dobije zahtev za resursima od nekog procesa, on pokreće Bankarev algoritam kako bi doneo odluku da li će dodeliti resurse procesu, odložiti ili odbiti zahtev. Neka je *zahtev*[*i*] niz zahteva procesa P_i u nekom trenutku. Prema Bankarevom algoritmu, kada proces izvrši zahtev za resursima, preduzimaju se koraci opisani u algoritmu 4.2.

Ovaj algoritam je teorijski dobar, ali u praksi ima ozbiljna ograničenja od kojih je najveće to što se za sve procese unapred mora znati koji su njihovi maksimalni zahtevi, a to u realnim sistemima najčešće nije poznato.

Algoritam 4.2. **Dodela resursa**

```
//Provera da li su zahtevi u skladu sa najavljenim pre početka izvršavanja
IF (potrebno[i] < zahtev[i])
   Došlo je do greške jer proces prevazilazi svoje prethodno definisane
   maksimalne zahteve;
ENDIF
IF (raspoloživo < zahtev[i])
   Resursi nisu raspoloživi pa će Pi morati da sačeka;
ENDIF
//Sistem pokušava da dodeli tražene resurse procesu Pi modifikujući
//stanje na sledeći način:
raspoloživo = raspoloživo – zahtev[i];
dodeljeno[i] = dodeljeno[i] + zahtev[i];
potrebno[i] = potrebno[i] – zahtev[i];
IF (modifikovano stanje sistema je bezbedno)
   Transakcija se odobrava, a proces Pi dobija svoje resurse;
ELSE
   Pi će morati da sačeka zahtevane resurse iz zahtev[i] a sistem se vraća na
   staro stanje dodele resursa;
ENDIF
```

Primer 4.4. Ako je stanje u sistemu kao u primeru 4.2. i ako P_3 zahteva 2 resursa tipa C, onda se zahtev odbija (P_3 će morati da sačeka) jer u sistemu u tom trenutku nema dovoljno raspoloživih resursa tipa C.

Ako P_3 zatraži jedan resurs tipa C. Onda bi posle dodele tog resursa novo stanje u sistemu odgovaralo stanju prikazanom u sledećim tabelama.

Raspoloživi resursi su dati u tabeli 4.8.

Tabela 4.8. Raspoloživi resursi

A	B	C	D
3	1	0	2

Alocirani resursi su dati u tabeli 4.9.

Tabela 4.9. Alocirani resursi

	A	B	C	D
P_1	1	2	2	1
P_2	1	0	3	3
P_3	1	1	2	0

Maksimalni zahtevi su dati u tabeli 4.10.

Tabela 4.10. Maksimalni zahtevi

	A	B	C	D
P_1	3	3	2	2
P_2	1	2	3	4
P_3	1	1	5	0

Trivijalno je pokazati da postoji bezbedna sekvence $<P_1, P_2, P_3>$, što znači da je sistem i dalje u bezbednom stanju tako da se dodela odobrava i nastavlja sa sledećim zahtevom.

Ako pri stanju u sistemu koje je definisano u primeru 4.2. proces P_2 zahteva jednu raspoloživu instancu resursa B, treba izvršiti proveru da li bi sistem bio u bezbednom stanju ako bi se ovaj zahtev odobrio. Stanje koje bi nastalo ovom dodelom je prikazano u tabelama 4.11., 4.12. i 4.13.

Tabela 4.11. Raspoloživo

A	B	C	D
3	0	1	2

Tabela 4.12. Trenutno alocirani resursi procesa

	A	B	C	D
P_1	1	2	2	1
P_2	1	1	3	3
P_3	1	1	1	0

Tabela 4.13. Maksimalne potrebe procesa

	A	B	C	D
P_1	3	3	2	2
P_2	1	2	3	4
P_3	1	1	5	0

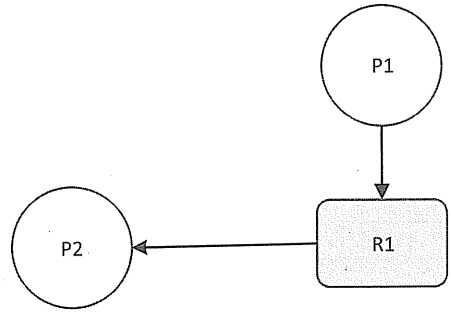
Pošto sistem nije u mogućnosti da procesima P_1, P_2 i P_3 obezbedi dovoljno resursa kako bi se izvršili do kraja, odnosno u takvom sistemu ne postoji bezbedna sekvenca, stanje je nebezbedno. Iz tog razloga zahtev procesa P_2 se odbija, a sistem vraća na stanje pre pokušaja dodele resursa B procesu P_2.

4.3. DETEKCIJA ZAGLAVLJIVANJA

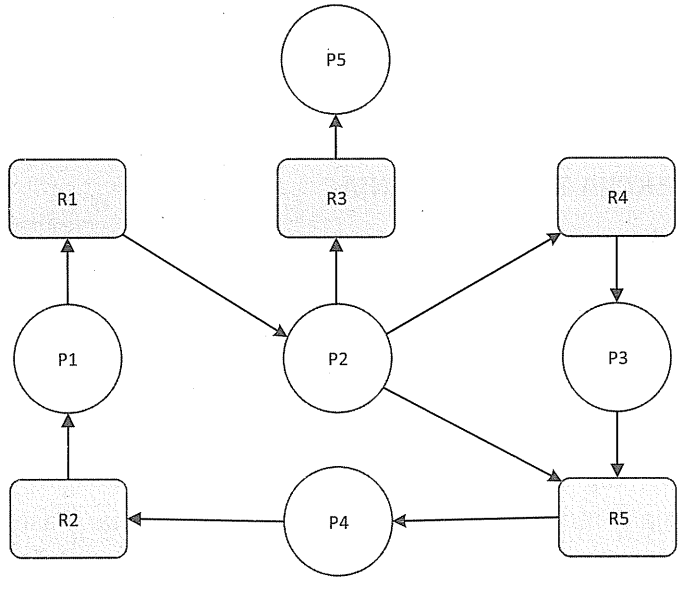
Detekcija zaglavljivanja je koncept koji ne garantuje da do zaglavljivanja neće doći, već dopušta da do njega dođe, a onda se primenjuju mere da se takva situacija otkloni. Otkrivanje da je do zaglavljivanja došlo može biti aktivno i pasivno. Aktivno otkrivanje je pristup u kojem se podrazumeva da se, korišćenjem algoritama za detekciju, proverava da li je došlo do zaglavljivanja. Učestalost provere zavisi od frekvencije zaglavljivanja i može varirati na različitim sistemima. Pasivan pristup podrazumeva da se provera vrši u situacijama kada se registruje da u sistemu nema aktivnosti ili da nešto nije u redu.

Stanje sistema se može modelovati korišćenjem grafova. Čvorovi ovakvih grafova su procesi (koji se obično predstavljaju krugovima) i resursi (predstavljeni pravougaonicima). Resursi sa više instanci se predstavljaju sa više tačaka u pravougaoniku. Grane grafova su u stvari oznake da li neki proces drži neki resurs što se označava strelicom od resursa ka procesu

dok strelica od procesa ka resursu označava da neki proces zahteva taj resurs, ali ga trenutno ne drži. Na slici 4.9. prikazana je situacija u sistemu sa dva procesa i jednim resursom. Proces P_1 zahteva resurs R_1, dok proces P_2 drži resurs R_1.



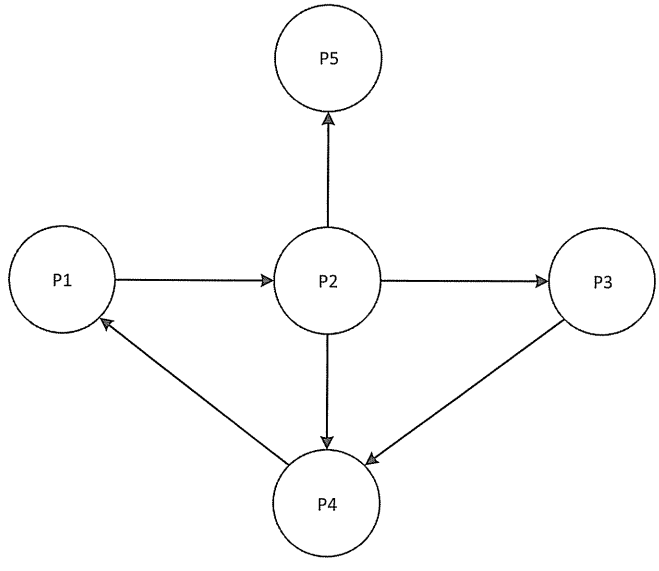
Slika 4.9. Primer stanja resursa i procesa u sistemu



Slika 4.10. Stanje sistema

Detekcija zaglavljivanja kod sistema u kojem svi resursi imaju po jednu instancu se svodi na pronalaženje ciklusa u grafu. Graf koji se posmatra nastaje kada se iz grafa koji opisuje stanje sistema uklone resursi, a grane povežu procese koji čekaju jedni na druge. Grana povezuje dva procesa ako jedan čeka na resurse drugog da bi se izvršio. Ukoliko u grafu postoji

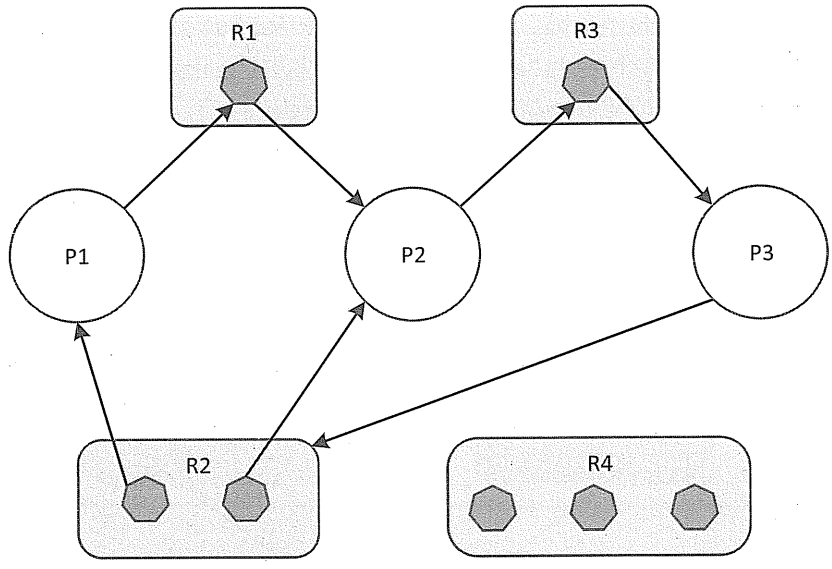
ciklus, to znači da postoji zaglavljivanje. Složenost ovakvog algoritma je $O(n)$ pri čemu je n broj grana grafa. Na slici 4.10. prikazan je primer stanja sistema a na slici 4.11. odgovarajući graf za detekciju zaglavljivanja.



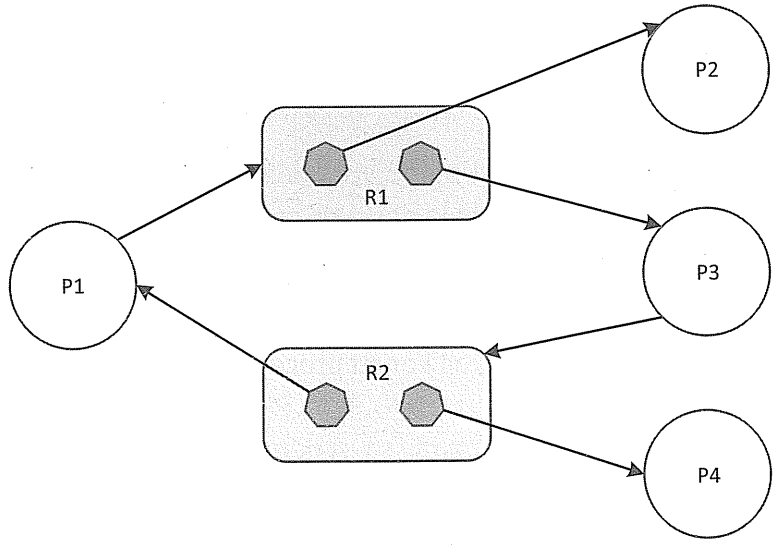
Slika 4.11. Detekcija ciklusa i zaglavljivanja

Kada su sistemi gde resursi mogu imati više instanci u pitanju, ciklus u grafu ne povlači obavezno da je došlo do zaglavljivanja.

Na slici 4.12. prikazan je primer grafa koji sadrži ciklus (R_2,P_2,R_3,P_3,R_2) i u zaglavljenom je stanju jer proces P_2 čeka na resurs R_3 kojeg koristi proces P_3, a koji čeka na resurs R_2 koji koristi proces P_2. Na slici 4.13. prikazan je primer grafa koji sadrži ciklus (P_1,R_1,P_3,R_2,P_1), ali pošto resurs R_2 ima dve instance, kada proces P_4 završi sa radom, oslobodiće jednu njegovu instancu koja će omogućiti procesu P_3 da se izvrši i ustupi resurs R_1 procesu P_1 kako bi se i on izvršio. Dakle, u ovom primeru nije došlo do zaglavljivanja.



Slika 4.12. Zaglavljeno stanje – više instanci resursa



Slika 4.13. Bezbedno stanje – više instanci resursa

Slično kao i u analizi prethodnog primera, algoritam za otkrivanje zaglavljivanja (algoritam 4.3.) se zasniva na traženju procesa koji mogu da budu zadovoljeni sa raspoloživim resursima. Ovaj algoritam je sličan algoritmu za proveru da li je stanje sistema bezbedno. Naime, ako se nađe

proces koji može biti zadovoljen, odnosno kojem se mogu obezbediti resursi da završi svoj posao, onda će on u trenutku kada završi, vratiti resurse koje je koristio. Tada i graf može da se redukuje njegovim izbacivanjem. Ako se ponavljanjem takvog postupka graf redukuje do kraja, tj. ako se izbace svi procesi, onda sistem nije zaglavljen. U suprotnom, zaključuje se da je došlo do zaglavljivanja, a procesi koji učestvuju u tom zaglavljivanju su među onima koji su ostali u grafu.

Algoritam 4.3. **Detekcija zaglavljivanja – više instanci resursa**

```
//Inicijalizacija
radni = raspoloživo;
FOR (i = 1; i <= n; i++)
   IF (dodeljeno[i] != 0)
      gotov[i] = 0;
   ELSE
      gotov[i] = 1;
   ENDIF
ENDFOR
sledeći_zadovoljeni_proces:
//Ispitivanje da li postoji proces koji nije zadovoljen a može da bude
IF (ne postoji i takvo da gotov[i] == 0 i zahtev[i] ≤ radni)
   idi na kraj;
ENDIF
//Pronalaženje procesa koji može biti zadovoljen sa raspoloživim resursima
Odaberi prvo i takvo da gotov[i] == 0 i zahtev[i] ≤ radni;
radni = radni + dodeljeno[i];
gotov[i] = 1;
idi na sledeći_zadovoljeni_proces;
kraj:
IF (gotov[i] == 0, za neko i, 1 ≤ i ≤ n)
   U sistemu je došlo do zaglavljivanja, procesi Pi za koje je gotov[i] == 0, su
   zaglavljeni;
ENDIF
```

Na početku niz *radni* sadrži resurse koji su na raspolaganju, odnosno nisu dodeljeni procesima. Zatim se traže procesi koji se mogu zadovoljiti resursima koji su raspoloživi. To znači da ti procesi nisu zaglavljeni i da postoje uslovi da se oni izvrše. Završetkom njihovog rada i oslobađanjem

resursa koje su koristili, može doći do stvaranja uslova da se i neki drugi procesi mogu izvršiti. Dakle, niz *radni* se može proširiti resursima koji su bili dodeljeni tim procesima. Na kraju se proverava da li su postojali uslovi da se svi procesi izvrše. Procesi koji na kraju nisu uspeli da „dobiju" resurse i da se izvrše (*gotov[i] == 0*) su zaglavljeni.

Vremenska složenost ovog algoritma je $O(m \times n^2)$ a njegova implementacija nije previše zahtevna.

Primer 4.5. Neka je stanje u sistemu opisano u tabeli 4.14.

Tabela 4.14. Stanje u sistemu u kojem postoji zaglavljivanje

	Dodeljeno			***Zahtev***			***Raspoloživo***		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P_1	0	1	0	0	0	0	0	0	0
P_2	2	0	0	2	0	2			
P_3	3	0	2	0	0	1			
P_4	2	1	1	1	0	0			
P_5	0	0	2	0	0	2			

U ovom sistemu je došlo do zaglavljivanja jer ne postoji dovoljno resursa da se zadovolje potrebe svih procesa. Naime, kada proces P_1 završi sa radom, on će ostaviti jedan resurs tipa B koji je nedovoljan da se podmire potrebe ostalih procesa koji će ostati zaglavljeni.

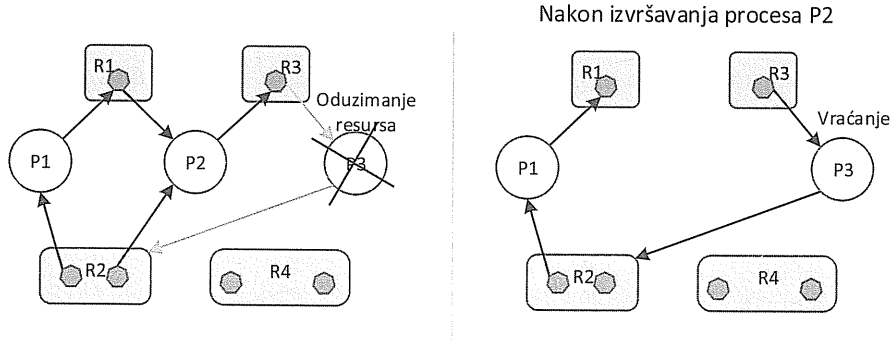
4.3.1. Oporavak od zaglavljivanja

Kada se utvrdi da je došlo do zaglavljivanja, sistem bi trebalo da interveniše i pomogne da se ovakva situacija prevaziđe. Najjednostavnija akcija koju operativni sistem može da uradi je da suspenduje blokirane procese, oduzme im resurse a odluku o njihovoj sudbini prepusti korisnicima. U praksi se više koriste potpuno automatizovane operacije kod kojih se ne očekuju intervencije korisnika. Najčešće se koriste dva pristupa, od kojih jedan podrazumeva prekidanje jednog ili više zaglavljenih procesa, a drugi oduzimanje resursa zaglavljenim procesima.

Algoritam za oporavak prekidanjem procesa, koji se prvi nameće, podrazumeva da se uklone svi blokirani procesi. Ovo rešenje je veoma

neefikasno jer se takvim postupkom gube rezultati do kojih su procesi došli dok se nisu zaglavili. Malo zahtevniji algoritam podrazumeva da se uklanja jedan po jedan od procesa u nadi da će se sistem izvesti iz zaglavljenog stanja. U tom slučaju, u svakom koraku, treba odabrati proces koji treba ukloniti. Ovo obično nije lak zadatak. Pri odabiru procesa, koji će biti prekinut, operativni sistem može uzimati neke od sledećih kriterijuma:

- Prioritet procesa – trebalo bi prvo prekinuti procese nižeg prioriteta.
- Koliko se dugo proces izvršava i koliko ima do završetka – ideja je da se prekidaju procesi koji su najkasnije počeli da se izvršavaju, a da se prekidanje onih koji su pri kraju (ako je to moguće odrediti) odlaže do poslednjeg trenutka.
- Koje resurse proces koristi i koji su mu potrebni da završi sa radom – teži se da se prekidaju procesi koji su veliki potrošači resursa ili procesi koji drže resurse koji su potrebni mnogim drugim procesima.



Slika 4.14. Oduzimanje resursa i kasnije vraćanje

Pristup koji podrazumeva oduzimanje resursa se zasniva na ideji da se određenim procesima oduzmu resursi kako bi se uz pomoć tih resursa otklonilo zaglavljivanje. Pri tome bi trebalo da se vodi računa o tome da se takvi procesi ostave u stanju koje je pogodno da kasnije mogu nastaviti sa radom. Trivijalno, proces se može vratiti na početak, ali je korisnije pronaći tačku u izvršavanju od koje on može nastaviti kada se steknu

uslovi i pri tome sačuvati rezultate izvršavanja pre oduzimanja resursa. U ovakvim situacijama trebalo bi voditi računa i o tome da se istom procesu ne oduzimaju resursi više puta, već da se poštuje neka vrsta pravednosti. Iako to nije očigledno, ponekad oduzimanje resursa procesu koji nije zaglavljen može dovesti do oslobađanja resursa koji mogu omogućiti da se sistem ne zaglavi. Na slici 4.14. prikazana je situacija u sistemu gde postoji zaglavljivanje. Problem se može rešiti suspendovanjem procesa P_3 i njegovim kasnijim pokretanjem kada proces P_2 završi sa radom.

Pitanja

4.1. Šta se podrazumeva pod pojmom zaglavljivanje?
4.2. Navesti primere zaglavljivanja.
4.3. Šta je živo blokiranje? Ilustrovati primerima.
4.4. U kakvim situacijama se može reći da je došlo do izgladnjivanja procesa?
4.5. Navesti primere izgladnjivanja procesa.
4.6. U kojem stanju se nalazi proces pre nego što se zaglavi?
4.7. Koliki je najmanji broj procesa koji mogu biti zaglavljeni?
4.8. Na kakvoj vrsti resursa obično dolazi do zaglavljivanja?
4.9. Koji uslovi treba da budu ispunjeni u sistemu kako bi došlo do zaglavljivanja?
4.10. Šta se podrazumeva pod čekanjem i držanjem resursa?
4.11. Objasniti pojam kružnog čekanja. Ilustrovati primerima.
4.12. Šta podrazumeva nemogućnost prekidanja?
4.13. Da li istovremena ispunjenost svih Kohmanovih uslova obavezno dovodi do zaglavljivanja?
4.14. Kakvi pristupi postoje kada je u pitanju tretman zaglavljivanja u odnosu na mere koje se primenjuju?
4.15. Kada je nepreduzimanje bilo čega kako bi se problem zaglavljivanja rešio opravdana mera?
4.16. U čemu je razlika između mera za sprečavanje i mera za izbegavanje zaglavljivanja?
4.17. Objasniti ideju detekcije i oporavka kada je zaglavljivanje u pitanju.
4.18. U kojem trenutku se primenjuju mere za sprečavanje zaglavljivanja?
4.19. Koji su glavni nedostaci mera za sprečavanje zaglavljivanja?
4.20. Na koji način se može eliminisati uslov čekanja i držanja?
4.21. Koji su nedostaci pristupa za sprečavanje zaglavljivanja koji podrazumeva da procesi pre početka izvršavanja zahtevaju dodelu svih potrebnih resursa?
4.22. Da li se svaki od Kohmanovih uslova može eliminisati primenom neke metode kako ne bi došlo do zaglavljivanja?

4.23. Na koji način se može eliminisati nemogućnost prekidanja?
4.24. U čemu je razlika između mera koje se primenjuju za eliminisanje uslova nemogućnosti prekidanja i uslova čekanja i držanja?
4.25. Kako se može eliminisati kružno čekanje?
4.26. Koji su glavni problemi kada je enumeracija procesa u pitanju?
4.27. Kada se primenjuju mere za izbegavanje zaglavljivanja?
4.28. Koja je osnovna ideja Bankarevog algoritma?
4.29. Koje informacije o procesima bi trebalo da budu poznate pre početka izvršavanja Bankarevog algoritma?
4.30. Na koji način se definiše bezbedna sekvenca?
4.31. Konstruisati primere bezbednih i nebezbednih sekvenci.
4.32. Kada se može reći da je sistem u bezbednom stanju?
4.33. Da li nebezbedno stanje sistema, u nekom trenutku, obavezno povlači zaglavljivanje?
4.34. Koji su ulazni podaci potrebni za izvršavanje Bankarevog algoritma?
4.35. Opisati Bankarev algoritam.
4.36. Na kojim principima funkcionišu mere za detekciju i otklanjanje zaglavljivanja?
4.37. Šta se podrazumeva pod detekcijom zaglavljivanja?
4.38. Kakvi pristupi postoje kada je učestalost provere da li je do zaglavljivanja došlo u pitanju?
4.39. Od čega zavisi učestalost provere da li je u sistemu došlo do zaglavljivanja?
4.40. Na koji način se može modelovati stanje u sistemu kada su u pitanju procesi i resursi?
4.41. Na šta se svodi detekcija zaglavljivanja u sistemima u kojim svi resursi imaju po jednu instancu?
4.42. Da li ciklus u grafu obavezno povlači da u sistemu postoji zaglavljivanje?
4.43. Konstruisati primere sistema koji imaju cikluse u grafu koji modeluje stanje sistema.
4.44. Koja je složenost algoritma za detekciju zaglavljivanja u sistemima u kojima svi resursi imaju po jednu instancu?

4.45. Koja je osnovna ideja algoritma za otkrivanje zaglavljivanja u sistemima sa više instanci istih resursa?
4.46. Koje se metode primenjuju za otklanjanje zaglavljivanja?
4.47. Koji su kriterijumi za odabir procesa koji će biti prekinut kada je algoritam za otklanjanje zaglavljivanja prekidanjem izabranih procesa u pitanju?
4.48. Na čemu se zasniva pristup za otklanjanje blokiranja oduzimanjem resursa?
4.49. O čemu treba voditi računa prilikom oduzimanja resursa procesima?
4.50. Da li za otklanjanje zaglavljivanja oduzimanjem resursa procesima, resurse isključivo treba oduzimati zaglavljenim procesima?

5. UPRAVLJANJE MEMORIJOM

Memorija predstavlja jedan od osnovnih resursa računarskog sistema. Svaka procesorska instrukcija se mora naći u memoriji, a za izvršavanje većine instrukcija potrebni su podaci iz memorije koji se moraju doneti (učitati) u procesor ili iz procesora prebaciti u memoriju. Iako današnji prosečni računari imaju nekoliko hiljada puta više memorije od najmoćnijih računara iz šezdesetih godina dvadesetog veka, programi postaju sve zahtevniji, a podaci kojima operišu postaju sve veći. Pošto je memorija ograničeni resurs, njeno efikasno korišćenje je jedan od osnovnih zadataka operativnih sistema. Ovim problemom se obično bavi poseban modul operativnog sistema koji se naziva *Menadžer memorije*.

Memorija je bilo koji fizički uređaj koji može privremeno ili trajno da čuva podatke. Postoji više različitih vrsta memorija. Na osnovu načina čuvanja podataka mogu se podeliti na memorije koje privremeno čuvaju podatke (dok je računar uključen) i memorije koje imaju mogućnost trajnog čuvanja podataka. Privremeno čuvanje podataka karakteriše se gubitkom podataka nakon isključivanja uređaja (računarskog sistema), dok kod trajnog čuvanja podataka to nije slučaj. Međutim, osnovna podela memorije obično se vrši na osnovu brzine pristupa. Po ovom kriterijumu memorije se mogu podeliti u četiri grupe:

- Registri;
- Keš memorija;
- Primarna memorija;
- Sekundarna memorija.

Na vrhu lestvice (slika 5.1.) nalaze se *registri*. To su memorijske ćelije ugrađene u procesor i u njima se nalaze podaci koje procesor trenutno obrađuje. Na primer, ako je potrebno izvršiti aritmetičku operaciju nad dva broja, ti brojevi se najpre smeštaju u registre. Registri su memorija koja privremeno čuva podatke.

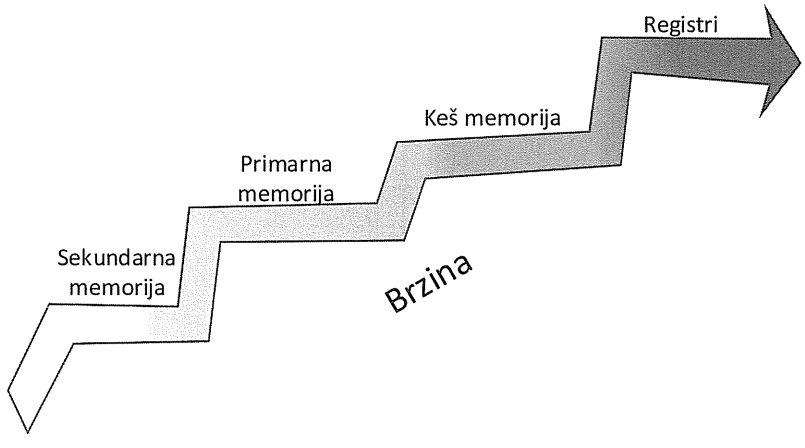
Keš memorija je veoma brza memorija koja čuva podatke koji se često koriste, a može predstavljati i bafer (videti deo 8.4.2.) između primarne memorije i procesora. Naime, keš memorija obično sadrži podatke i instrukcije koje se često koriste ili za koje se pretpostavlja da će biti potrebni u bliskoj budućnosti. Na ovaj način se smanjuje broj pristupa sporijim memorijama čime se povećava iskorišćenost procesora i poboljšava efikasnost sistema. Keš memorija se obično nalazi u samom procesoru, ali se delovi mogu naći i van njega. Keš memorija će biti detaljnije opisana u delu 8.4.4.

Primarna (glavna) *memorija* sadrži instrukcije i podatke sa kojima procesor trenutno operiše, pa se iz tog razloga još naziva *radnom memorijom*. Brzina ove memorije nije kao kod keš memorije, ali je značajno brža od sekundarne memorije. Posebna vrsta primarne memorije je RAM memorija (*Random Access Memory*). Ovaj naziv se odnosi na to da procesor može direktno da pristupi bilo kom delu RAM memorije, pri čemu vreme pristupa ne zavisi od lokacije i sadržaja podataka. U računarstvu se termin „memorija" obično odnosi na ovu vrstu memorije. To je memorija koja privremeno čuva podatke i bez koje računar ne može da funkcioniše. Druga vrsta primarne memorije je ROM memorija (*Read Only Memory*). Za razliku od RAM-a, ROM čuva podatke kada je računar isključen. Moguće je samo čitanje podataka iz ROM-a. Mnogi računari poseduju malu količinu ROM memorije koja sadrži kritične programe koji služe za pokretanje operativnog sistema. EEPROM memorija (*Electrically Erasable Programmable ROM*) takođe čuva podatke kada je računar isključen, ali za razliku od ROM-a podaci se mogu brisati i ponovo upisivati. Međutim, njihovo upisivanje oduzima mnogo više vremena nego kada je u pitanju RAM memorija. Stoga se EEPROM memorije koriste na isti način kao i ROM memorije, sa dodatnom funkcionalnošću da ih je moguće prepisati nakon izlaska iz fabrike. Termin „ROM" se često koristi za softver koji se čuva u EEPROM memoriji.

Sekundarna memorija ne gubi sadržaj nakon prestanka rada računara, pa se zbog toga koristi za trajno čuvanje podataka. Da bi procesor pristupio podacima iz sekundarne memorije, najpre je potrebno da se oni učitaju u

radnu memoriju. Zbog toga se sekundarna memorija naziva i spoljašnja (eksterna) memorija. Primeri sekundarne memorije su hard disk, CD-ROM, DVD itd. Vreme pristupa podacima iz ove memorije zavisi od njihove lokacije. Sekundarna memorija nije neophodna za rad računara.

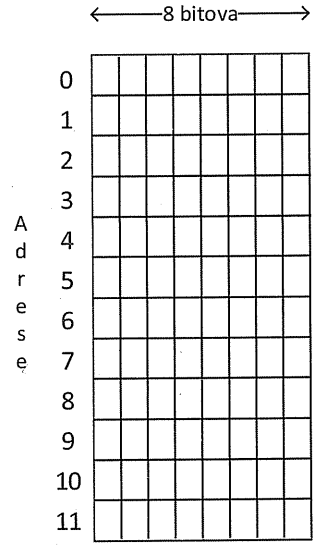
Memorija se sastoji od niza memorijskih ćelija koje mogu da sačuvaju najmanju količinu podataka – bit. Bitovi se organizuju u bajtove – grupe od po 8 bitova. Na većini računara ovo je najmanja adresibilna jedinica, što znači da svaki bajt ima svoju adresu i može mu se direktno pristupiti.



Slika 5.1. Brzina memorije

Memorijska reč određuje količinu podataka koje procesor može da obradi u jednom trenutku. Njena veličina je određena arhitekturom konkretnog računara, a u prošlosti je najčešće bila veličine 8, 16, 32 bita. U trenutku pisanja ovog teksta najzastupljeniji su računari sa 64-bitnom arhitekturom. Na primer, 64-bitni računar je u mogućnosti da jednom instrukcijom sabere dva 64-bitna broja (brojevi predstavljeni sa 64 binarnih cifara), dok će 32-bitni računar biti u stanju da isto to uradi sa brojevima veličine najviše 32 bita. Kod operacija sa pokretnim zarezom, veće memorijske reči dozvoljavaju veću preciznost. Često se, kada se govori o veličini memorijske reči, misli i na adresibilni prostor računara. Na primer, 32-bitni računar najčešće dozvoljava 32-bitne memorijske adrese, pa je u mogućnosti da podrži 2^{32}= 4.294.967.296 bajtova (4GB) memorije.

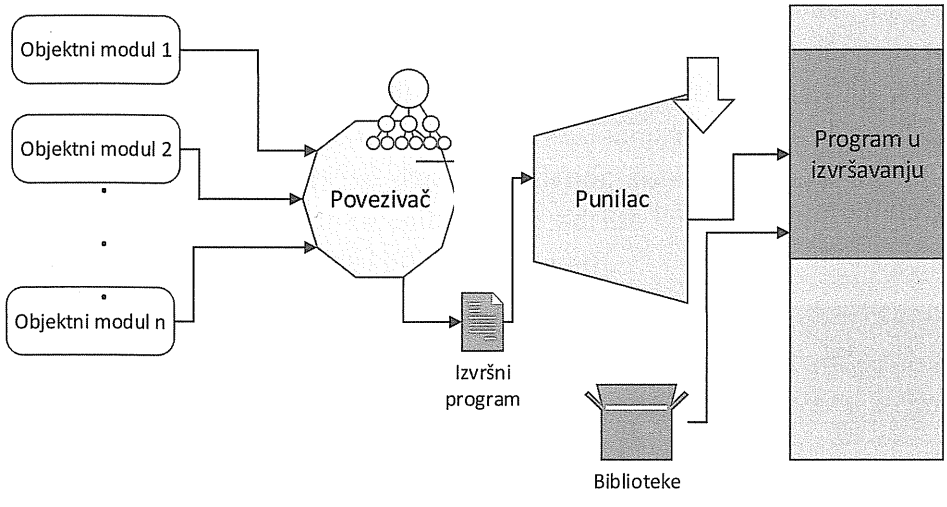
Radna memorija se može predstaviti kao matrica bitova. Zbog pojednostavljenja može se pretpostaviti da je ta matrica dimenzije *N* x 8, pa se memorija može shvatiti kao niz bajtova (slika 5.2.). Redni broj bajta zapisan u binarnom sistemu predstavlja adresu tog bajta. Na osnovu te adrese procesor može da pristupi proizvoljnom bajtu.



Slika 5.2. Memorija kao matrica bitova

Svaki program koji se izvršava mora se učitati u radnu memoriju (slika 5.3.). U trenutku pisanja programa nije poznato u kom delu memorije će program biti smešten. Zato se koriste simboličke adrese, koje odgovaraju imenima promenljivih u programskim jezicima. Kada se program učita u memoriju on mora da radi sa realnim fizičkim adresama. Proces prevođenja simboličkih u fizičke adrese se naziva povezivanje adresa (*address binding*) i može se obaviti u različitim trenucima. Ukoliko je za vreme prevođenja programa poznato u kom delu memorije će se učitati, onda se povezivanje adresa može obaviti u tom trenutku. U suprotnom (najčešćem) slučaju, prevodilac (kompilator) generiše *relativne adrese* u odnosu na početak dela memorije koja se dodeljuje procesu. Ove adrese se još nazivaju i relokatibilne adrese. Prevodilac prevodi izvorni kôd u objektni modul, a program se može sastojati iz više objektnih modula. Povezivač (*linker*) povezuje sve objektne module u izvršni program.

Prilikom učitavanja izvršnog programa, punilac (*loader*) preslikava relativne adrese u fizičke. Ako se program koji se izvršava može premeštati za vreme izvršavanja, tada se povezivanje adresa obavlja za vreme izvršavanja programa. Takođe, program može pozivati neke biblioteke pod određenim uslovima. Te biblioteke se ne učitavaju sa učitavanjem programa da ne bi nepotrebno zauzimale prostor. Kada se ostvare određeni uslovi, biblioteke se učitavaju u memoriju i pozivaju. I u ovom slučaju se povezivanje adresa obavlja za vreme izvršavanja programa.



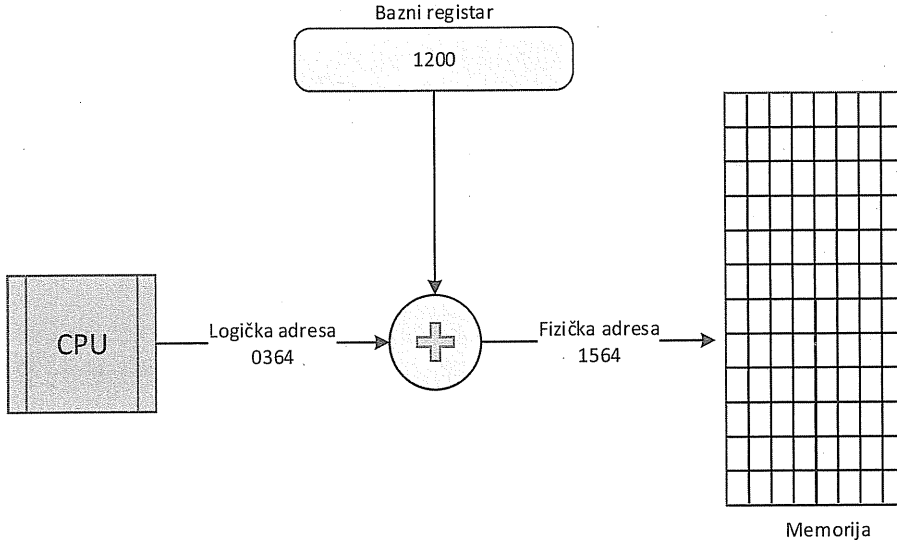
Slika 5.3. Izvršni program u memoriji

Tokom izvršavanja procesa, interakcija sa memorijom se odvija kroz niz čitanja ili pisanja u lokacije koje se nalaze na određenoj adresi. Procesor uzima instrukcije i podatke iz memorije ili smešta podatke u memoriju. U navedenim akcijama procesor ne manipuliše fizičkim adresama, već logičkim koje sam generiše. Skup svih logičkih adresa naziva se logički ili virtuelni adresni prostor. Svakoj logičkoj adresi odgovara fizička adresa.

Skup svih fizičkih adresa koje odgovaraju adresama logičkog adresnog prostora naziva se fizički adresni prostor. Ukoliko se povezivanje adresa obavlja za vreme prevođenja i punjenja programa, logički i fizički adresni prostori se podudaraju. To nije slučaj kada se povezivanje adresa obavlja

za vreme izvršavanja. Hardverski uređaj koji preslikava logičke u fizičke adrese naziva se *jedinica za upravljanje memorijom MMU* (*Memory-Management Unit*).

Jednostavni MMU se može predstaviti kao uređaj koji poseduje poseban registar koji se naziva *bazni registar*. MMU dodaje vrednost baznog registra na logičku adresu i tako generiše fizičku adresu (slika 5.4.). Korisnički program radi sa logičkim adresama koje on vidi u opsegu od 0 do *max*. Ako je sa *R* označena vrednost baznog registra, MMU preslikava logičke adrese u fizičke koje su u opsegu od *R* + 0 do *R* + *max*.



Slika 5.4. MMU uređaj

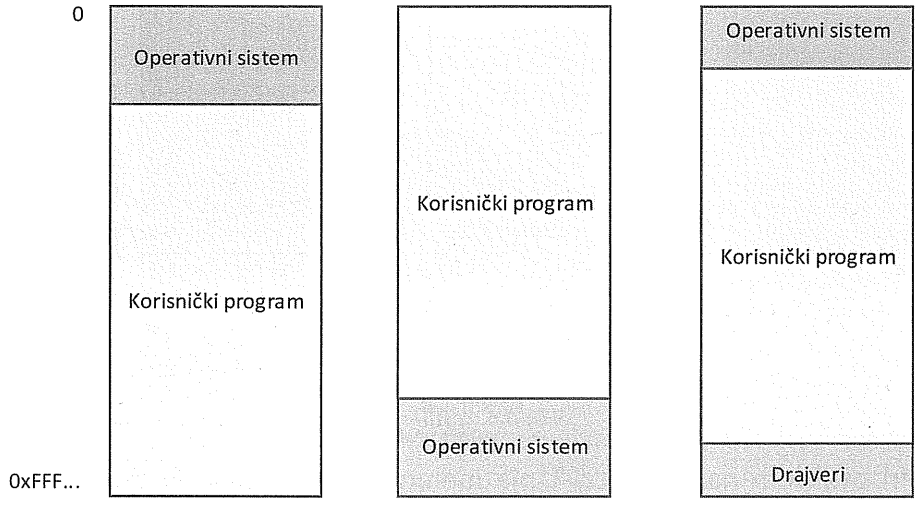
Prilikom upravljanja memorijom, operativni sistem bi trebalo da ispuni sledeća dva uslova:

1. Svaki proces mora imati dovoljno memorije za izvršavanje i ne sme pristupati memoriji nekog drugog procesa kao što drugi proces ne sme pristupati njegovoj memoriji.
2. Različite vrste memorija unutar računarskog sistema moraju biti korišćene tako da se svaki proces izvršava najefikasnije moguće.

Postoje različiti pristupi i principi za upravljanje memorijom, a neki od osnovnih će biti prikazani u narednom delu.

5.1. UPRAVLJANJE MEMORIJOM PRI MONOPROGRAMIRANJU

Monoprogramiranje je najjednostavniji pristup za upravljanje memorijom koji podrazumeva da se u memoriji izvršava samo jedan korisnički proces. Jedan deo memorije se odvaja za operativni sistem, dok ostatak memorije koristi proces koji se izvršava. Kod ovog pristupa postoje različite organizacije memorije (slika 5.5.). Jedna od njih podrazumeva da se operativni sistem učitava na niže adrese, a da procesor koristi više adrese memorije. Druga mogućnost je da se operativni sistem nalazi na vrhu memorije, a da niže adrese koristi proces. Poslednja mogućnost odnosi se na organizaciju kod koje se deo operativnog sistema učitava na niže adrese memorije, drajveri se učitavaju na adrese koje se nalaze na vrhu memorije, a ostatak služi za korisnički program.

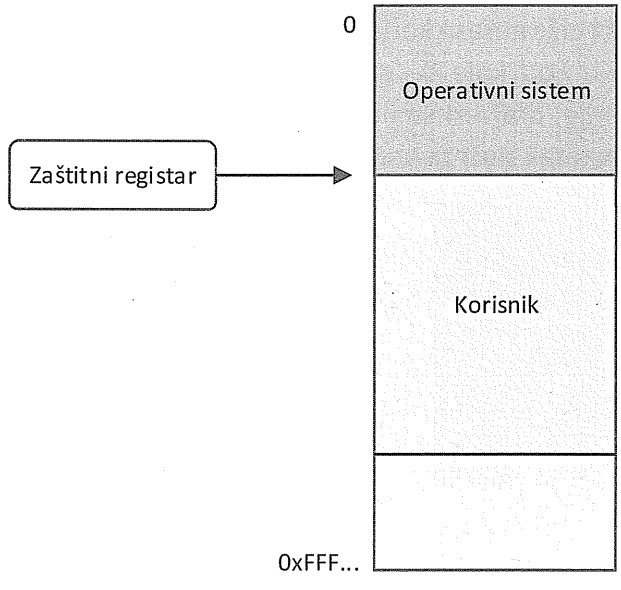


Slika 5.5. Monoprogramiranje

Prvi računarski sistemi i operativni sistemi mikroračunara su koristili ovakav vid upravljanja memorijom. Ovakvo upravljanje memorijom, i to treću organizaciju, imao je operativni sistem MS-DOS. Kod MS-DOS sistema, korisnik unosi komandu za pokretanje programa. Operativni

sistem kopira zahtevani program u memoriju i izvršava ga. Kada proces završi sa radom, operativni sistem štampa *„prompt"* karakter i čeka unos sledeće komande. Kada se unese sledeća komanda, operativni sistem učitava zahtevani program pri čemu prepisuje prethodni sadržaj memorije.

Učitavanje operativnog sistema u memoriju izvršava se prilikom uključivanja sistema. Tada se aktivira *punilac* (*bootstrap loader*) koji ima zadatak da prenese operativni sistem iz sekundarne memorije u radnu memoriju. Ovaj postupak se zove podizanje sistema. Po učitavanju operativnog sistema, korisnički program (odnosno proces) se puni na adrese odmah posle operativnog sistema, ili na drugi kraj memorije.

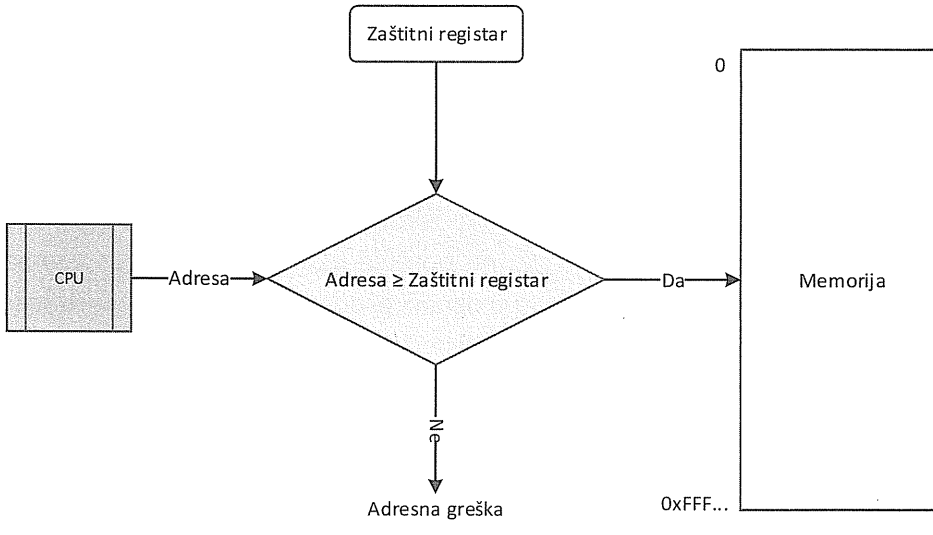


Slika 5.6. Zaštitni registar

Pri ovakvom načinu upravljanja memorijom potrebno je zaštititi kôd i podatke operativnog sistema od slučajnih ili namernih izmena od strane korisničkog programa. Ovakvu vrstu zaštite obično pruža hardver, a najjednostavniji pristup podrazumeva da se svaka adresa koju generiše korisnički program poredi sa sadržajem *zaštitnog registra* (slika 5.6.). Ukoliko je ona manja od sadržaja zaštitnog registra generiše se prekid

(pogrešna adresa) i operativni sistem preduzima odgovarajuće akcije (slika 5.7.). Obično je reakcija na takvu vrstu greške da se prekine korisnički program uz izdavanje odgovarajuće poruke o problemu.

Adresni prostor počinje od adrese 0, prva adresa u korisničkom programu nije 0, već prva adresa posle vrednosti u zaštitnom registru. Povezivanje instrukcija i podataka sa memorijskim adresama može se izvršiti ili u vreme prevođenja ili u vreme punjenja. Ako je vrednost zaštitne adrese poznata u vreme prevođenja, može se generisati apsolutni kôd, a adrese će se nizati počev od adrese koja se nalazi u zaštitnom registru. Problem pri ovakvom pristupu je u tome što promena vrednosti zaštitne adrese iziskuje ponovno prevođenje programa. Alternativa je da prevodilac generiše relokatibilni kôd. Tada se povezivanje odlaže do momenta punjenja. U tom slučaju, kod promene zaštitne adrese, potrebno je samo ponovno punjenje programa.



Slika 5.7. Zaštitna adresa

U oba slučaja, vrednost zaštitne adrese mora da bude fiksirana tokom izvršenja programa. U nekim slučajevima, u toku izvršavanja programa, potrebno je da se menja veličina dela memorije gde je smešten operativni sistem, a samim time i zaštitna adresa. Promena u vrednosti granične adrese zahteva izmenu sadržaja zaštitnog registra, ali i premeštanje

korisničke memorije na nove lokacije u odnosu na novu vrednost granične adrese. Ovakva operacija je vremenski veoma zahtevna jer se dosta vremena troši na kopiranje memorije.

5.2. Upravljanje memorijom pri multiprogramiranju

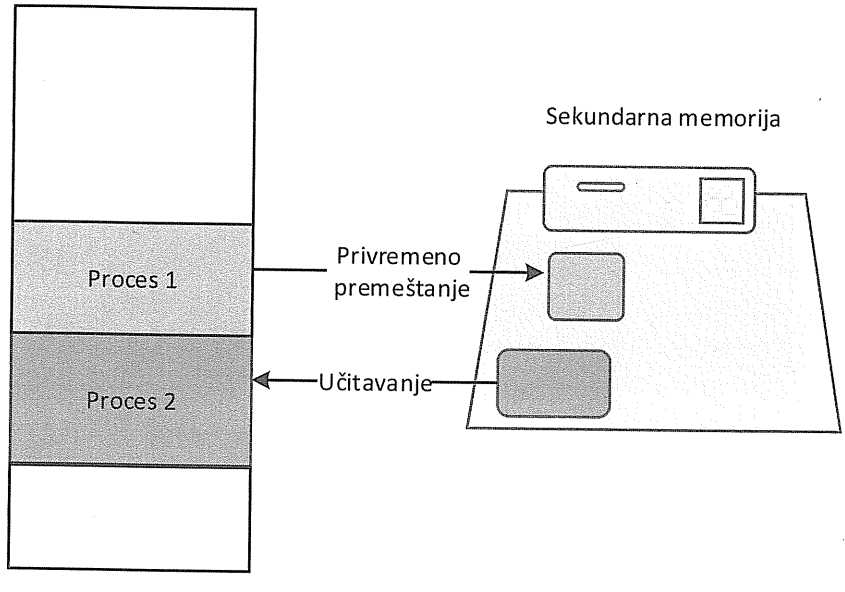
Izvršavanje procesa može biti blokirano, npr. zbog čekanja na ulazno-izlazne operacije. Kod monoprogramiranja u ovakvim situacijama procesor prestaje sa korisnim radom, pa je stepen njegove neiskorišćenosti veliki. Kod savremenih računara, da bi se povećala iskorišćenost procesora, koriste se napredniji pristupi upravljanja procesima zasnovani na multiprogramiranju. Ovakvi pristupi omogućavaju da se više procesa može naći u memoriji i da se prividno izvršavaju istovremeno. Kada se tekući proces zaustavi zbog čekanja na neku ulazno-izlaznu operaciju, procesor može da izvršava instrukcije nekog drugog procesa.

U nekim slučajevima proces troši i do 80% ukupnog vremena izvršavanja zbog čekanja na neke spoljašnje operacije (npr. ulazno-izlazne operacije), dok se samo 20% vremena koristi za pravo izračunavanje. Ako bi procesor istovremeno izvršavao 5 programa, iskorišćenost bi bila 100%. Ovo zahteva da ni u jednom trenutku ne postoje dva ili više procesa koji se nalaze u stanju čekanja. Ako se sa *p* označi procenat vremena koje proces troši na čekanje i ako postoji *n* procesa u memoriji, tada je verovatnoća da se svih *n* procesa nađe u stanju čekanja p^n. Odavde se dobija da procenat vremena u kojem procesor nije besposlen iznosi: $1 - p^n$. Ovaj procenat se naziva iskorišćenost procesora, a broj *n* se naziva stepen multiprogramiranja. Navedena formula za iskorišćenost procesora je aproksimativne prirode, jer u stvarnosti procesi nemaju isto vreme čekanja.

Postoje različite tehnike koje obezbeđuju memoriju potrebnu procesima za izvršavanje u sistemima zasnovanim na multiprogramiranju i sličnim konceptima. Neke od njih biće opisane u nastavku.

5.2.1. Prebacivanje

Za vreme izvršavanja, program (proces) mora biti učitan u memoriju. Međutim, dok proces čeka na neke resurse nije neophodno da se nalazi u memoriji. *Prebacivanje* je jednostavna tehnika upravljanja memorijom (i procesima) koju operativni sistem može koristiti u cilju povećanja iskorišćenosti procesora. Ideja ove tehnike je da se blokirani proces privremeno premesti u sekundarnu memoriju (disk), a da se u memoriju učita drugi proces (slika 5.8.). Na ovaj način se formira red privremeno blokiranih procesa. Procesi iz reda blokiranih procesa, koji su u mogućnosti da nastave sa radom, formiraju red spremnih procesa. Nakon prebacivanja blokiranog procesa u sekundarnu memoriju operativni sistem odlučuje da li će učitati novi program u memoriju, ili će aktivirati neki iz reda spremnih procesa.

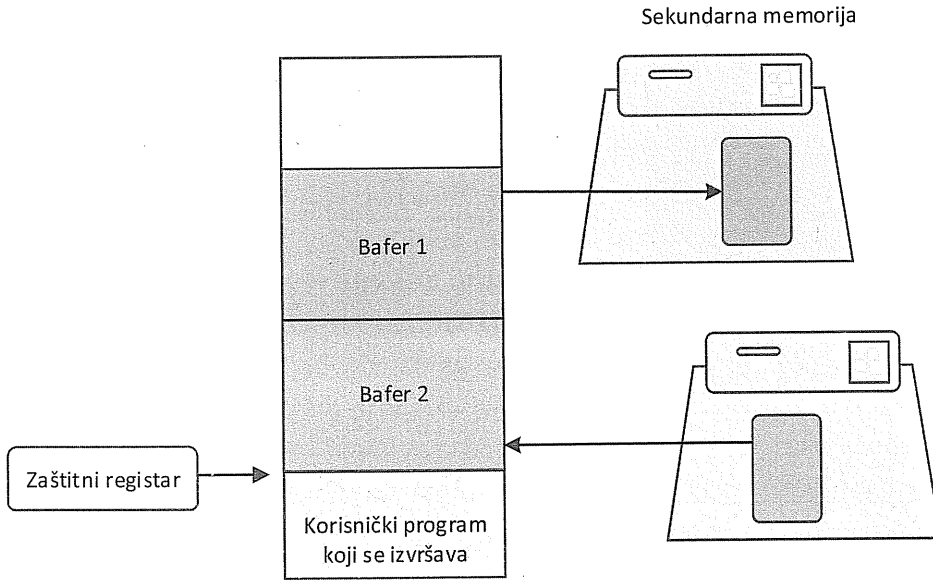


Slika 5.8. Prebacivanje

Efikasnost prebacivanja veoma zavisi od brzine sekundarne memorije, koja bi trebalo da bude i dovoljno velika da prihvati memorijske slike za sve procese. Osim toga, potrebno je omogućiti direktan pristup memorijskim slikama procesa.

Vreme potrebno za prebacivanje procesa nije zanemarljivo malo tako da se teži da vremena trajanja procesorskih aktivnosti procesa budu srazmerno duža od vremena potrebnog za prebacivanje kako bi iskorišćenost procesora bila što bolja. Najviše vremena potrebnog za prebacivanje procesa se troši na prenos, a to vreme je direktno proporcionalno količini podataka koji se prebacuju. Brzina i efikasnost prebacivanja se poboljšava sa unapređenjem performansi jedinica sekundarne memorije.

Napredniji pristup, kada je prebacivanje u pitanju, podrazumeva da se efikasnost sistema poboljša preklapanjem prebacivanja i izvršavanja procesa. Ideja je da se preklopi prebacivanje jednog procesa sa izvršenjem drugog. Za realizaciju ovakvog pristupa potrebno je rezervisati bar dva memorijska regiona koji mogu prihvatiti dve memorijske slike procesa. Ovi regioni se obično nazivaju baferi (slika 5.9.).



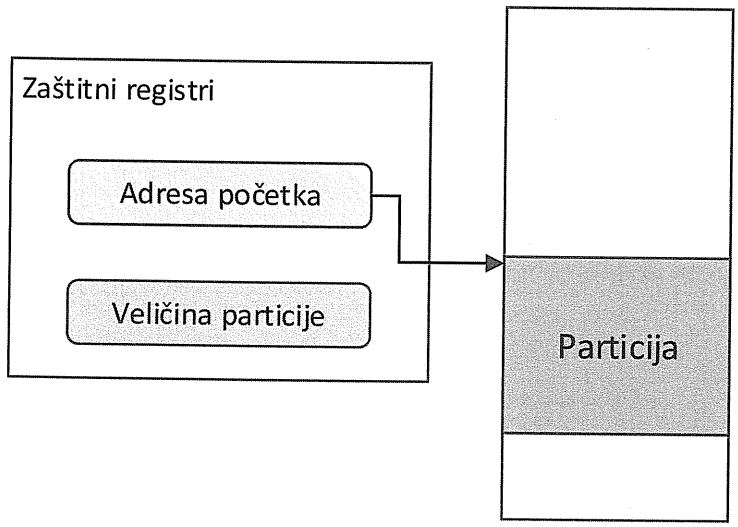
Slika 5.9. Baferi i sekundarna memorija

Kada proces koji se izvršava oslobodi procesor, premešta se u slobodni bafer i pokreće se njegovo pomeranje na sekundarnu memoriju. Naredni proces se pomera iz bafera u oblast memorije predviđene za izvršavanje.

Zatim se u odgovarajući bafer učitava proces koji je sledeći na redu za izvršavanje. Ovakvim pristupom se preklapa izvršavanje jednog procesa sa učitavanjem i upisivanjem drugih procesa u memoriju.

5.2.2. Particije

Tehnika prebacivanja podrazumeva da se, pored operativnog sistema, u memoriji najčešće nalazi samo jedan proces. Porast kapaciteta memorije omogućio je da se umesto prebacivanja koristi efikasnija tehnika koja se naziva *particionisanje memorije*. Osnovna ideja je da se memorija podeli na *n* delova – *particija*, koje predstavljaju neprekidne delove memorije. Particije mogu biti jednake, ali i različitih veličina. U svaku particiju se može smestiti po jedan proces. Stepen multiprogramiranja u ovom slučaju je *n*. Ako su sve particije zauzete, a neki program treba da započne izvršavanje, onda se formira red spremnih procesa. Kada se neka od zauzetih particija oslobodi, učitava se neki od spremnih procesa. Ako je više particija slobodno, postoje različite politike na osnovu kojih se procesu dodeljuje particija.



Slika 5.10. Particije

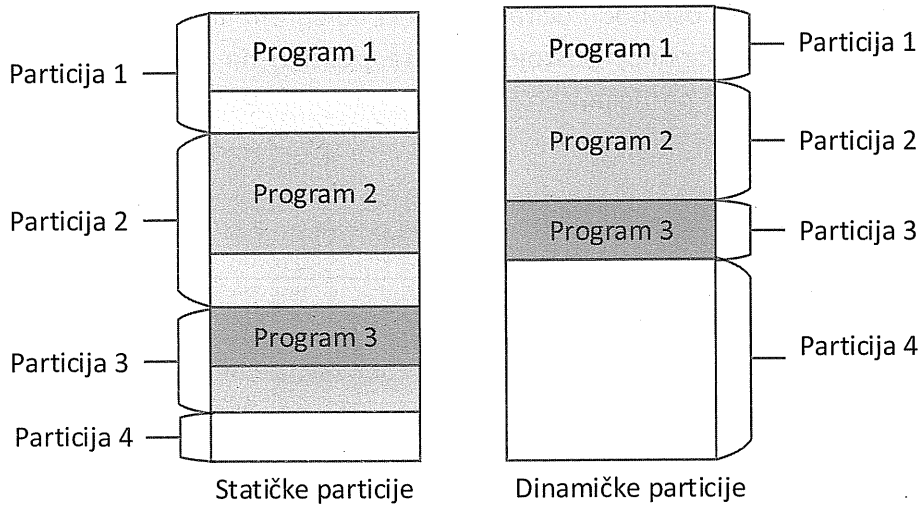
Potrebno je particije zaštititi od pristupa drugih procesa. Ova zaštita se realizuje (najčešće) uz pomoć hardverske podrške koja se sastoji od po dva registra za svaku particiju. U zavisnosti od implementacije, ti registri

mogu sadržati fizičke adrese početka i kraja particije, ali i adresu početka i veličinu particije (slika 5.10.). Tako je svakom procesu ograničen pristup samo na adrese koje odgovaraju njegovoj particiji. Ukoliko proces pokuša da pristupi nekoj nedozvoljenoj adresi, operativni sistem preduzima određene akcije (npr. prekida izvršavanje procesa).

Postoje dva pristupa kada je patricionisanje memorije u pitanju:

1. Statički;
2. Dinamički.

Statički pristup podrazumeva da se veličina particija ne menja u toku rada dok *dinamički* pristup dozvoljava promenu veličine memorijskih oblasti u zavisnosti od potreba procesa i stanja u sistemu (slika 5.11.).



Slika 5.11. Statičke i dinamičke particije

5.2.3. Statičke particije

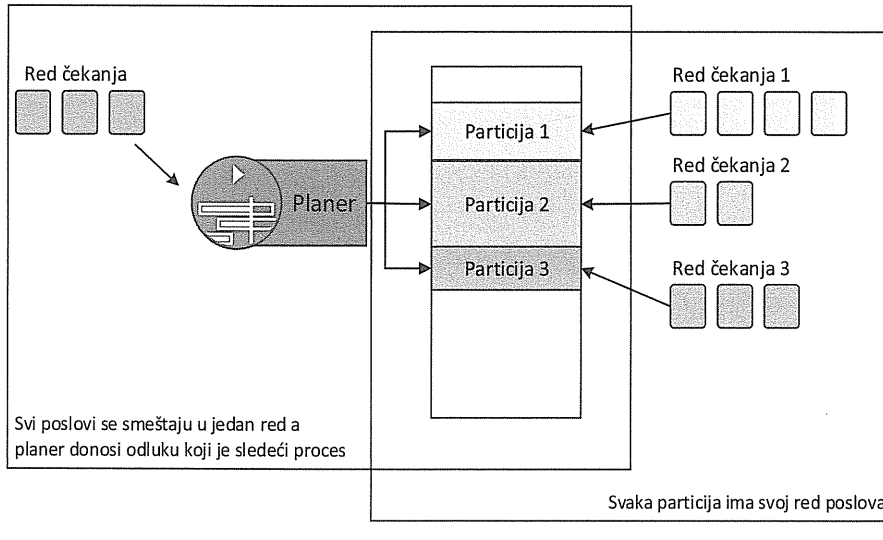
Kod *statičkog particionisanja* radna memorija se statički deli na fiksni broj particija, gde svaka particija može sadržati proces. Broj particija zavisi od različitih faktora, kao što su: prosečna veličina procesa, veličina radne memorije, brzina procesora itd. Veličine particija su fiksne i najčešće nisu

jednake veličine. Particije manje veličine služe za manje i kratke procese, dok se veće particije ostavljaju za zahtevnije procese.

Sistemi koji podržavaju statičke particije, procese koji pristižu u sistem smeštaju u red spremnih poslova. Planeri vode računa o memorijskim potrebama svakog od procesa kao i o raspoloživim particijama kako bi odredili kojim procesima treba dodeliti koju particiju. Kada se procesu dodeli memorijski prostor, on se smešta u odgovarajuću particiju. Tek tada taj proces može konkurisati za dobijanje procesora i ostalih resursa koji su mu potrebni za izvršavanje. Kada proces završi sa svojim radom, on oslobađa particiju u kojoj se izvršavao i onda se ona može dodeliti drugom procesu.

Kada je u pitanju odabir particije u koju bi proces trebalo da bude smešten, postoje dve strategije.

Prva strategija podrazumeva da svaka particija ima svoj red poslova u koji se smeštaju procesi čiji memorijski zahtevi odgovaraju veličini particije.



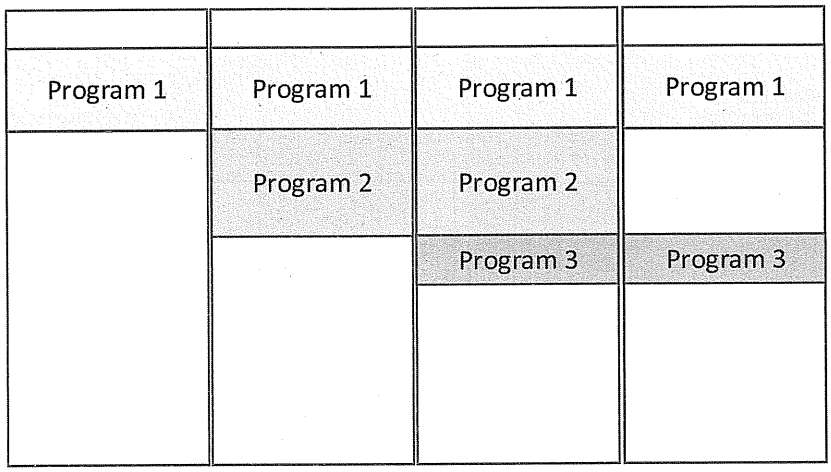
Slika 5.12. Redovi poslova

Drugi pristup je da se svi poslovi smeštaju u jedan red, a da planer donosi odluku koji je sledeći proces koji bi trebalo da se izvrši (slika 5.12.). Tada se čeka da se particija odgovarajuće veličine oslobodi. Jedna od čestih

modifikacija ovog pristupa ima za cilj da spreči da memorijske particije budu prazne ako postoje procesi koji nisu na redu a mogli bi da se u njima izvršavaju. Ovo se postiže tako što se prvi proces u redu, koji čeka da se oslobodi odgovarajuća particija, preskače ukoliko postoji slobodna particija koja odgovara nekom drugom procesu koji se nalazi u redu. Ovakav pristup otvara pitanje da li proces treba da sačeka na oslobađanje njemu najpribližnije particije po veličini ili da se, u slučaju da postoji veća raspoloživa particija, proces smesti u tu particiju. Ovim se smanjuje čekanje, ali se troši više memorije nego što je procesu potrebno i potencijalno usporava proces sa većim memorijskim zahtevima koji se nalazi dalje u redu.

5.2.4. Dinamičke particije

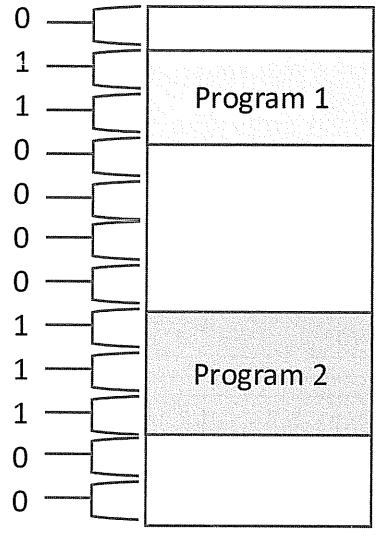
Dinamički pristup kod particionisanja memorije je sličan kao i statički, osim toga što veličine particija nisu fiksirane. Sistemi koji podržavaju dinamičke particije moraju voditi računa o slobodnoj i zauzetoj memoriji.



Slika 5.13. Dinamičke particije

Pri pokretanju računara, u memoriju se učitava operativni sistem. Ostatak memorije (koji je slobodan) obrazuje jednu slobodnu particiju. Kada se pokrene proces koji je prvi predviđen za izvršavanje, slobodna particija se deli na dva dela: jedan deo je onaj koji se dodeljuje procesu, a drugi deo ostaje slobodan za ostale procese koji treba da se učitaju u memoriju.

Vremenom neki procesi završavaju svoje izvršavanje ostavljajući pri tome prazne delove memorije (particije) koje su koristili (slika 5.13.). Kada novi proces dođe na red za izvršavanje najpre se pronalazi dovoljno velika slobodna particija, a zatim se iz nje izdvaja onoliko memorije koliko je procesu potrebno. Nakon završetka izvršavanja proces oslobađa memoriju. Ako pored upravo oslobođene memorije već postoji slobodna particija, oslobođena memorije se spaja sa susednom slobodnom memorijom i obrazuje novu veću slobodnu particiju.



Slika 5.14. Bit-mapa

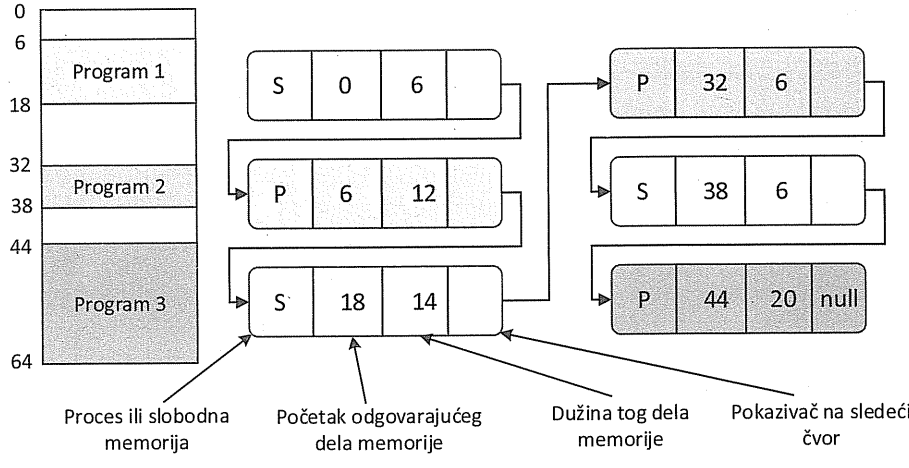
Evidenciju o slobodnim i zauzetim delovima memorije operativni sistem obično vodi korišćenjem bit-mapa ili povezanih listi.

Bit-mape podrazumevaju da se memorija podeli na jednake delove i da se svakom od tih delova pridruži po jedan bit. Ovaj bit ima vrednost 1 ukoliko je odgovarajući deo memorije zauzet, a u suprotnom je vrednost bita jednaka 0 (slika 5.14.). Pri ovakvoj realizaciji, za punjenje procesa u memoriju potrebno je pronaći dovoljno veliki niz nula, odnosno dovoljno slobodne memorije u koju se proces može smestiti. Implementacija bit-

mapa je jednostavna ali je vreme potrebno za pretraživanje i pronalaženje slobodne particije dosta veliko, tako da se češće koriste povezane liste.

Povezana lista (slika 5.15.) sastoji se od čvorova, pri čemu svaki čvor sadrži sledeće podatke:

- Da li je u odgovarajući deo na koji se čvor odnosi smešten proces ili je u pitanju slobodna memorija;
- Početak odgovarajućeg dela memorije;
- Dužina tog dela memorije;
- Pokazivač na sledeći čvor koji čuva informacije o narednom delu memorije.



Slika 5.15. Povezane liste

U nekim implementacijama se liste slobodnih i zauzetih delova memorije implementiraju odvojeno. Time se olakšava pretraživanje slobodnih delova, ali se otežava ažuriranje obe liste.

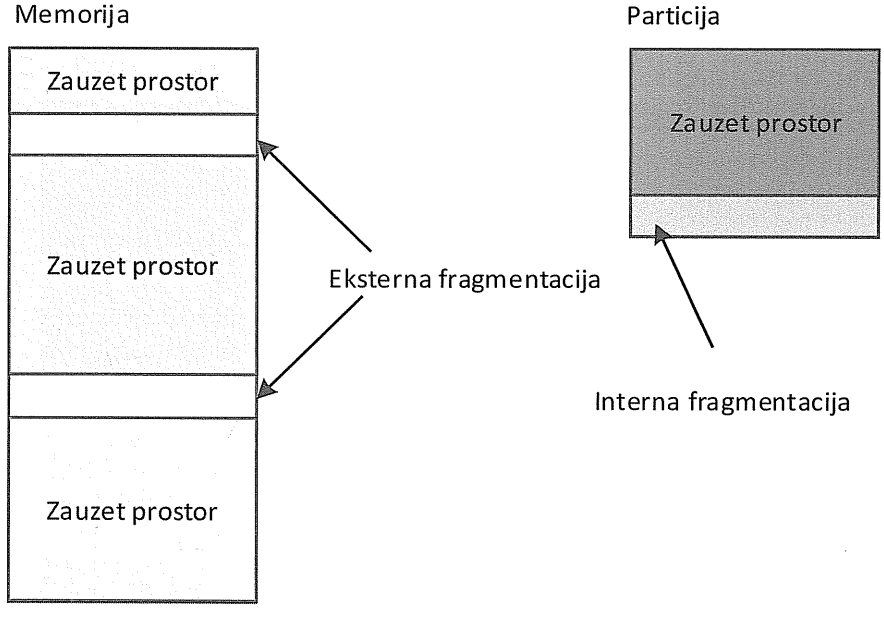
5.2.5. Fragmentacija

Jedna od neželjenih pojava kod sistema sa particijama je *fragmentacija*. Ona nastaje kada u računarskom sistemu postoje delovi memorije koji su slobodni, ali ih sistem ne može iskoristiti. Fragmenti predstavljaju

neiskorišćen slobodan memorijski prostor, iako postoje procesi koji čekaju da se izvrše. Fragmentacija može biti interna i eksterna.

Interna fragmentacija se odnosi na prostor koji ostaje neiskorišćen kada se proces smesti u deo memorije (particiju) veći od njegovih memorijskih potreba. Na primer, ako proces zauzima m memorijskih reči, a učita se u particiju veličine n reči, pri čemu je $n > m$, tada razlika $n - m$ memorijskih reči predstavlja fragment nastao usled interne fragmentacije. Interna fragmentacija se skoro sigurno javlja kod statičkog particionisanja memorije.

Eksterna fragmentacija podrazumeva da u sistemu postoje delovi memorije koji su slobodni ali nisu dovoljno veliki da se u njih smesti neki proces. Iz tog razloga ovi delovi memorije su neupotrebljivi (slika 5.16.).

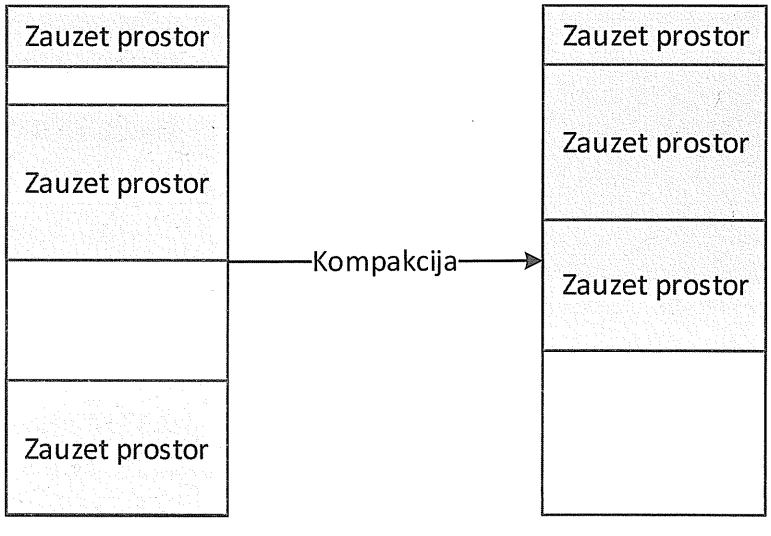


Slika 5.16. Fragmentacija

Kod dinamičkih particija ne javlja se problem interne fragmentacije, ali može nastati eksterna fragmentacija. Naime, procesu se od slobodne particije dodeljuje onoliko memorije koliko mu je potrebno. Ostatak particije formira novu slobodnu particiju i stoji na raspolaganju narednim

procesima. Međutim, može se desiti da je taj ostatak nedovoljno veliki da sadrži proces. Pri tome, u memoriji se može naći veći broj malih slobodnih particija. Eksterna fragmentacija nastaje kada pojedinačno nijedna od tih particija nije dovoljno velika za izvršavanje procesa. Česta je situacija da su sve slobodne memorijske lokacije zajedno dovoljno velike za izvršavanje procesa. Pošto ne formiraju jedinstvenu particiju, ne mogu se iskoristiti.

Jedan od načina da se reši problem eksterne fragmentacije kod dinamičkih particija je *kompakcija*. Kompakcija podrazumeva da operativni sistem zaustavi procese i da se izvrši relokacija procesa tako da se oni sabiju na početak (ili kraj) memorije. Pri tome se slobodna memorija grupiše na drugi kraj memorije i formira jedinstvenu particiju (slika 5.17.). Pošto je ova operacija vremenski dosta zahtevna, sistem će izgubiti na efikasnosti ako se često koristi, tako da se upravljanje memorijom ne bi trebalo zasnivati na kompakciji.



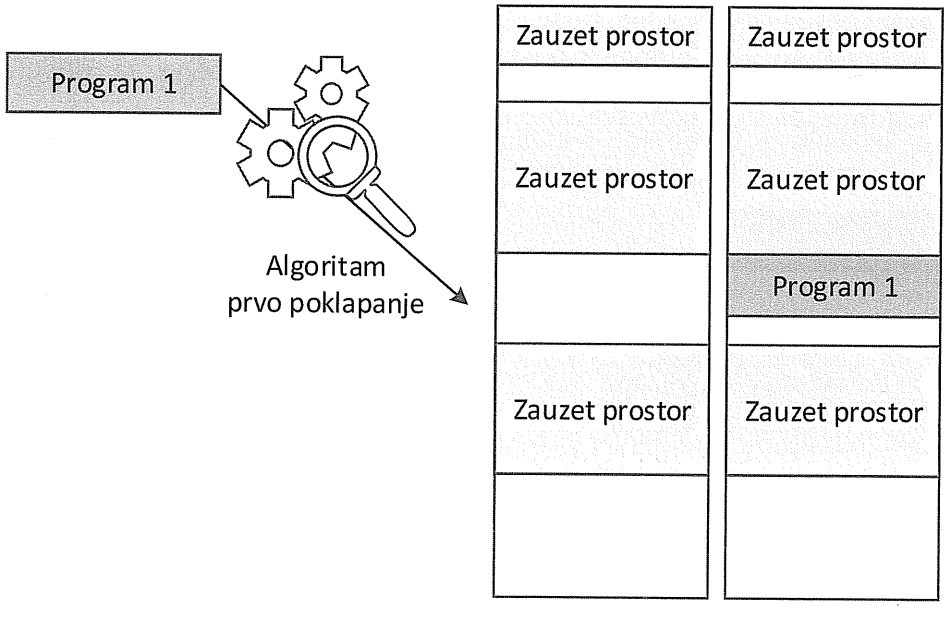
Slika 5.17. Kompakcija memorije

Eksterna fragmentacija se javlja i kod statičkog particionisanja, kada nijedna slobodna particija nije dovoljno velika za izvršavanje procesa.

5.2.6. Algoritmi za dodelu memorije

Postoje različiti načini na koje se procesu od raspoložive memorije može dodeliti deo potreban za njegovo izvršavanje.

Prvo poklapanje (*First-Fit*) je pristup koji se često koristi. On podrazumeva da se proces smesti u prvu slobodnu zonu koja je dovoljno velika da on može da stane u nju (slika 5.18.). Pretraga za slobodnim prostorom kreće od početka memorije. Pošto je prostor u koji se smešta proces veći od njegovih potreba, deo koji je zauzeo se označava kao zauzet, a preostali deo memorije i dalje ostaje na raspolaganju drugim procesima. Vreme potrebno za dodelu memorije procesu je relativno malo, ali je zato loša iskorišćenost memorije zbog česte pojave fragmentacije.

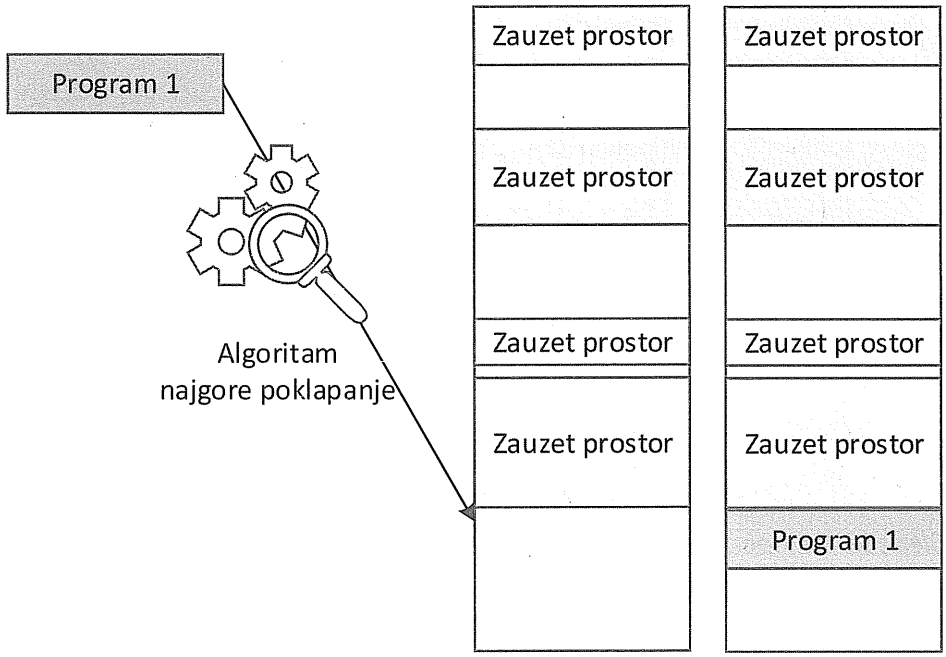


Slika 5.18. Algoritam prvo poklapanje

Sledeće poklapanje (*Next-Fit*) je jedna od modifikacija prethodnog pristupa koja podrazumeva da pretraga za slobodnom memorijom ne kreće svaki put od početka, već da se nastavi od lokacije gde je prethodna pretraga stala. Međutim, pokazalo se da pristup sledećeg poklapanja ima malo lošije performanse od prvog poklapanja.

Slučajno (*Random*) *raspoređivanje* je jedna od mogućih realizacija dodele memorije koja podrazumeva da se od svih slobodnih i dovoljno velikih zona (za smeštanje procesa) u memoriji slučajno izabere ona u koju će se smestiti proces. Ovakvo rešenje je relativno zahtevno, jer je potrebno odrediti delove memorije koji su kandidati za smeštanje procesa, ali i generisati slučajan broj.

U praksi se dosta efikasnim pokazao pristup zasnovan na ideji da se proces smesti u slobodni deo memorije koji je najmanji od svih slobodnih delova koji su veći od potreba procesa. *Najbolje poklapanje* (*Best-Fit*) je uobičajeni naziv za ovakav pristup. On je dosta zahtevan za implementaciju. Naime, potrebno je proći kroz celu memoriju, izračunati veličinu svih slobodnih regiona, i pronaći najmanji koji je veći od memorijskih zahteva koje ima proces. U odnosu na pristup prvog poklapanja, ovaj algoritam zahteva malo više vremena za dodelu memorije, ali su njegovi sveukupni rezultati veoma dobri.

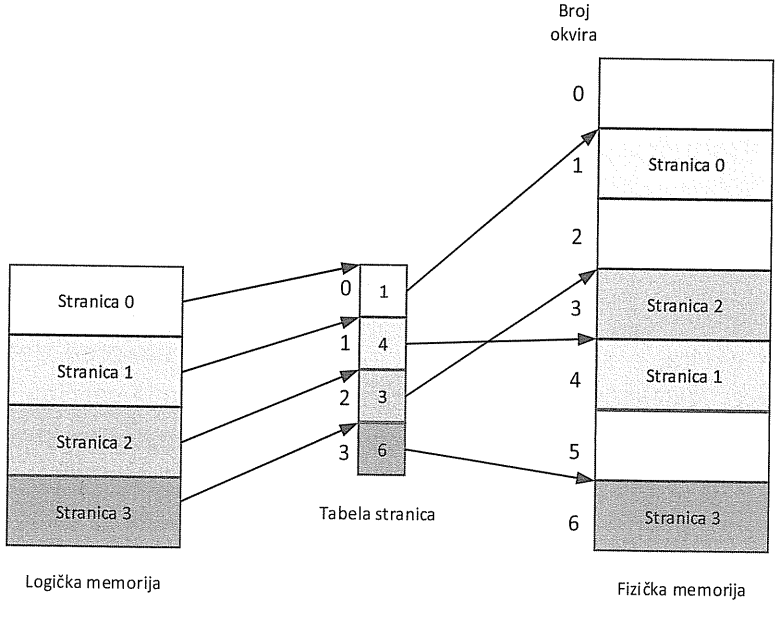


Slika 5.19. Algoritam najgore poklapanje

Najgore poklapanje (*Worst-Fit*) je pristup koji se, suprotno prethodnom, zasniva na ideji da se proces smesti u deo memorije u koji se najgore uklapa, tj. u region sa najvećim kapacitetom (slika 5.19.). Ovakva ideja proistekla je iz pretpostavke da bi ovakvim raspoređivanjem procesa u što veće particije slobodni ostatak mogao da se iskoristi za neki drugi proces.

5.3. STRANIČENJE

Straničenje je jedan od prvih načina upravljanja memorijom koji je dozvoljavao da memorija dodeljena procesu ne bude alocirana u jednom komadu, tj. isključivo na adresama koje se nalaze jedna za drugom. Ovakvim pristupom se omogućava da proces dobije memoriju, bez obzira gde se ona nalazi, ukoliko je slobodna. Osnovna ideja straničenja je da se radna memorija podeli na *okvire* (*frames*) fiksirane veličine, a da se logička memorija potrebna procesu podeli na *stranice* koje su iste veličine kao i okviri. Kada proces treba da se učita u memoriju, pronalaze se slobodni okviri i u njih se smeštaju odgovarajuće stranice.

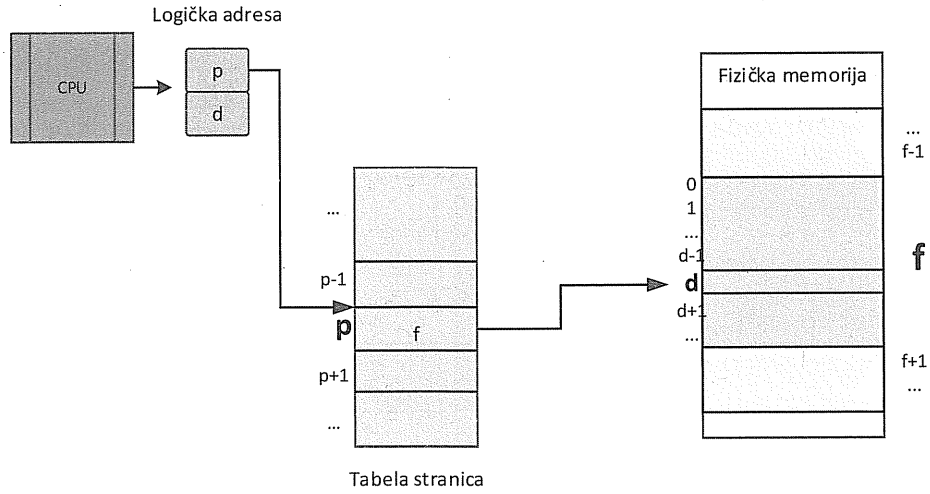


Slika 5.20. Straničenje

Za vreme izvršavanja programa procesor radi sa logičkim adresama. Kod ovakve organizacije *logička adresa* se sastoji iz dva dela:

1. Broja stranice – *p*;
2. Pozicije u okviru stranice – *d*.

Logičke adrese je potrebno prevesti u fizičke, a za to se koristi posebna struktura koja se naziva *tabela stranica* (slika 5.20.). Operativni sistem generiše po jednu tabelu stranica za svaki proces koji se izvršava. Tabela stranica svakog procesa sadrži podatke o tome u kojim okvirima u memoriji su smeštene njegove stranice. Fizička adresa se dobija na osnovu ovih podataka dodavanjem vrednosti pozicije u stranici (*d*) na adresu okvira u kojem se nalazi odgovarajuća stranica (slika 5.21).

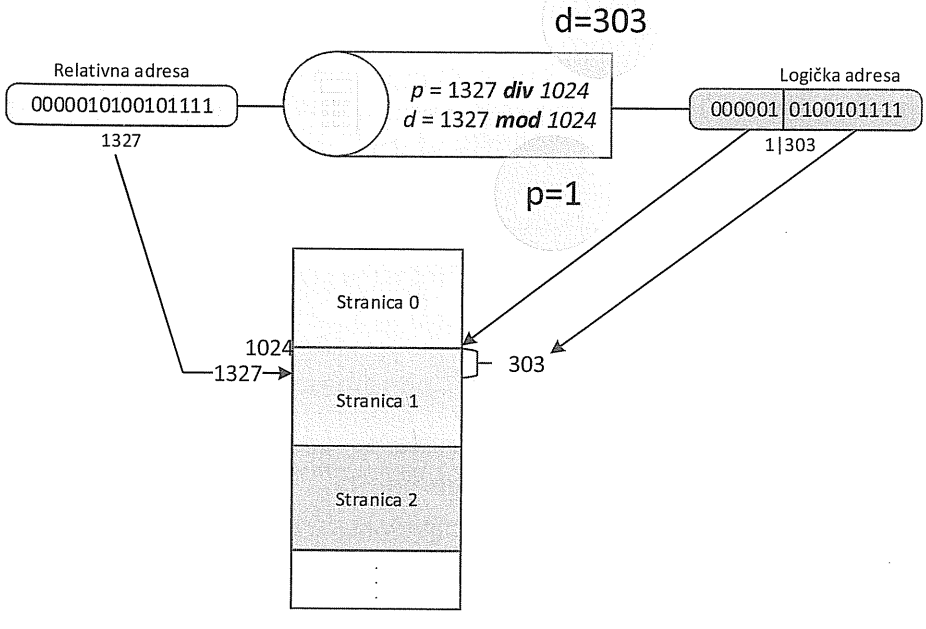


Slika 5.21. Prevođenje logičke u fizičku adresu

Veličina okvira, a samim tim i stranice, obično je unapred definisana hardverski. Najčešće je ta veličina stepen broja 2. Ako je veličina stranice *P*, tada se na osnovu relativne adrese *U* mogu izračunati broj stranice *p* i pozicija u okviru stranice *d* na sledeći način:

- $p = U$ **div** P;
- $d = U$ **mod** P.

Veličina stranice koja je jednaka nekom stepenu broja 2 omogućava da relativne i logičke adrese budu jednake. Ako je veličina stranice jednaka 2^n, tada nižih *n* bitova relativne adrese predstavlja poziciju u okviru stranice, dok viši bitovi relativne adrese odgovaraju rednom broju stranice. Na primer, neka je veličina stranice 1KB = 1024B = 2^{10}B i neka se radi o 16-bitnom adresnom prostoru. Na osnovu navedenih formula, za relativnu adresu 1327 (tj. 1327. bajt po redu) može se izračunati da je *d*=303, a *p*=1, pa je odgovarajuća logička adresa 1|303 (*p*|d). Relativna adresa 1327. bajta se binarno može zapisati 0000010100101111, a isti binarni zapis ima i odgovarajuća logička adresa jer nižih 10 bitova predstavlja binarni zapis broja 303, a viši bitovi odgovaraju broju 1 (slika 5.22.).

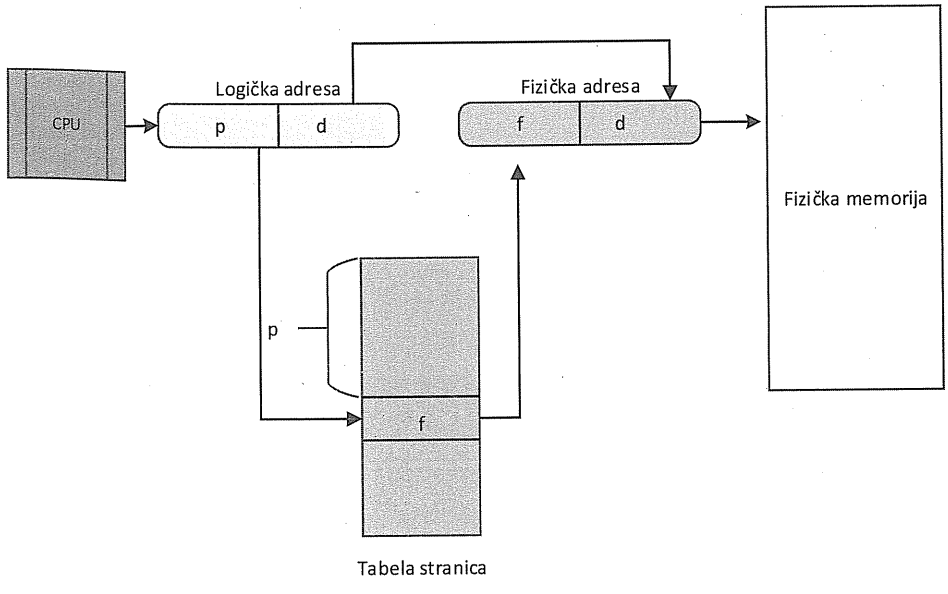


Slika 5.22. Relativne i logičke adrese kada je veličina stanice stepen broja 2

Osim toga, veličina stranice koja je jednaka stepenu broja 2 omogućava da se preslikavanje logičkih u fizičke adrese izvede na veoma jednostavan način. Naime, da bi se od logičke (i relativne) adrese dobila fizička, potrebno je u tabeli stranica pronaći gde se u fizičkoj memoriji nalazi

odgovarajuća stranica i vrednost te adrese postaviti na više bitove relativne adrese (slika 5.23.).

Za određivanje koji proces je sledeći za izvršavanje odgovoran je planer poslova. Kada proces želi da krene sa izvršavanjem, planer izračunava njegovu veličinu izraženu u stranicama i proverava raspoložive memorijske okvire. Za izvršavanje procesa koji zahteva *n* stranica potrebno je da *n* straničnih okvira bude slobodno. Ukoliko postoji dovoljno slobodnih okvira, stranice procesa se pune u dodeljene okvire, a broj okvira se smešta u tabelu stranica tog procesa.



Slika 5.23. Dobijanje fizičke adrese kada je veličina stranice jednaka stepenu 2

Jasno je da kod straničenja nema eksterne fragmentacije, jer svaki slobodni okvir može da bude dodeljen. Međutim, skoro sigurno postoji izvesna interna fragmentacija, jer obično poslednji okvir dodeljen procesu ne bude potpuno popunjen. Još jedna od prednosti straničenja je mogućnost deljenja zajedničkog koda. Ukoliko se kôd ne samomodifikuje u toku izvršavanja, moguće je da više korisnika koristi istu kopiju tog koda u memoriji.

Zaštita memorije vrši se uz pomoć zaštitnih bitova koji se pridružuju svakoj stranici i čuvaju u tabeli stranica. Jednim bitom se označava da li se određena stranica može čitati i modifikovati ili samo čitati. Pošto svaki pristup memoriji podrazumeva čitanje tabele stranica radi određivanja broja memorijskog okvira, tom prilikom se proverava da li se stranica može modifikovati ili je dozvoljeno samo njeno čitanje.

5.3.1. Hardverska podrška - tabela stranica

U najjednostavnijem slučaju tabela stranica se čuva u skupu registara. Međutim, korišćenje registara za čuvanje tabele stranica je pogodno samo u situacijama kada je tabela relativno male veličine. U situacijama kada je tabela stranica veća, ona se čuva u memoriji, a na njenu lokaciju u memoriji pokazuje *bazni registar tabele stranica – PTBR* (*Page Table Base Register*).

Čuvanje cele tabele stranica u memoriji je neefikasno, jer je za pristup nekoj memorijskoj lokaciji potrebno dva puta pristupiti memoriji, čime se sistem usporava dva puta. Naime, da bi se pristupilo nekoj lokaciji, mora se najpre pristupiti tabeli stranica koja se nalazi u memoriji na lokaciji koja se čuva u baznom registru tabele stranica. U tabeli se pronalazi adresa okvira u koji je smeštena stranica kojoj pripada tražena lokacija. Na osnovu adrese okvira i pozicije u stranici, pristupa se traženoj lokaciji.

5.3.2. Asocijativna memorija

Implementacija efikasne tabele stranica je od vitalnog značaja kod sistema koji podržavaju straničenje. Za te potrebe najčešće se koristi veoma brza *asocijativna memorija* (*Translation Lookaside Buffer* - TLB), koja je u stvari keš memorija koja je posebno dizajnirana za potrebe straničenja. Asocijativna memorija je malo sporija od registara, ali je većeg kapaciteta, pa se deo tabele stranica može smestiti u nju. Ova memorija ima osobinu da čuva podatke u parovima (*ključ, vrednost*), a pretraga se vrši simultano (odjednom) nad svim ključevima. Ključevi su redni brojevi stranica, a vrednosti su odgovarajuće adrese memorijskih okvira (slika 5.24.).

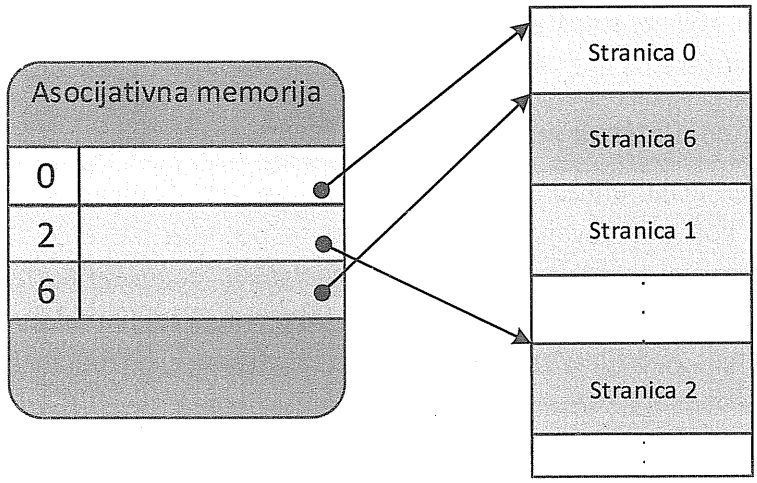
Kada procesor pokuša da pristupi nekoj logičkoj adresi, najpre određuje redni broj stranice koja sadrži tu adresu. Zatim se vrši pretraga asocijativne memorije. Ako je pretraga uspešna, procesor može odmah da pristupi lokaciji u memoriji na koju se logička adresa odnosi. U suprotnom kaže se da je došlo do promašaja, tako da je potrebno da se pretraži deo tabele stranica koji je u memoriji.

Procenat uspešnog pronalaska željenih podataka u asocijativnoj memoriji naziva se *nivo pogotka* (*hit ratio*). Ako je nivo pogotka 80%, a pretraživanje asocijativne memorije traje 30 puta manje u odnosu na vreme pristupa memoriji, tada se vreme nalaženja stranice može izračunati na sledeći način:

$$t_{ef} = 0.80 \cdot 1 + 0.20 \cdot (1 + 30) = 7.$$

$$f = \frac{30}{7} \cong 4.29.$$

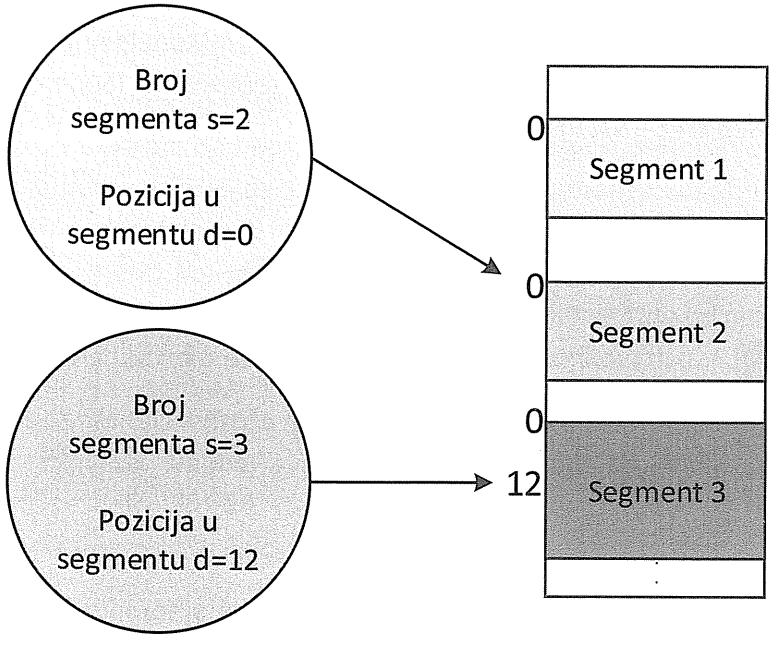
Dakle, nalaženje stranice je, u ovom primeru, 4.29 puta brže kada se koristi asocijativna memorija uz pretpostavku da je nivo pogotka 80% (u realnim situacijama je obično veći) što predstavlja značajno ubrzanje.



Slika 5.24. Asocijativna memorija

5.4. SEGMENTACIJA

Prevedeni korisnički program se može posmatrati kao skup različitih logičkih celina – *segmenata*. Na primer, neke od njih su: glavni program, stek, hip, biblioteke, globalni podaci itd. Osnovna ideja segmentacije je da se svakom segmentu dodeli poseban memorijski prostor. Svaki segment je atomičan, tj. ili će se ceo segment naći u memoriji ili neće uopšte biti učitan u memoriju. Segmenti se mogu naći bilo gde u memoriji, ali jedan segment mora biti u neprekidnom memorijskom bloku. Nije neophodno da segmenti budu iste veličine, već se svakom segmentu dodeljuje tačno onoliko memorije koliko je potrebno.



Slika 5.25. Segmentacija

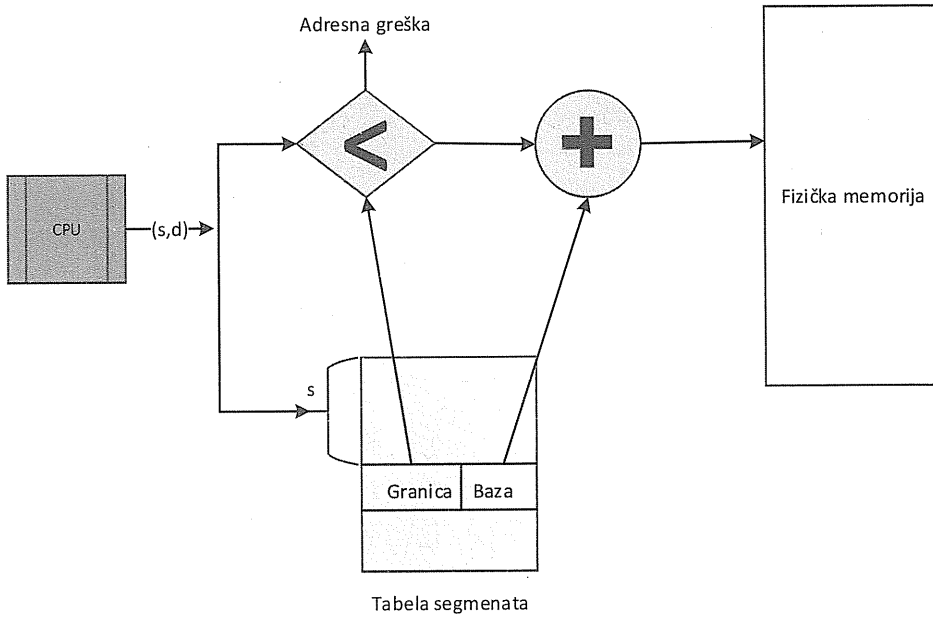
Slično kao kod straničenja, logičke adrese kod segmentacije se sastoje iz dva dela (slika 5.25.):

1. Broja segmenta – *s*;
2. Pozicije u segmentu – d.

Dvodimenzionalne logičke adrese potrebno je preslikati u jednodimenzione fizičke adrese. Za te potrebe uvodi se tabela segmenata.

5.4.1. Tabela segmenata

Svakom procesu se dodeljuje po jedna *tabela segmenata* u kojoj se čuva po jedan red za svaki segment procesa. Taj red sadrži redni broj segmenta, adresu početka segmenta u memoriji i dužinu segmenta. Kao i kod straničenja, tabela segmenata može biti smeštena u registrima, asocijativnoj memoriji ili u radnoj memoriji. *Bazni registar tabele segmenata* (*Segment Table Base Register – STBR*) sadrži adresu lokacije gde je smeštena tabela segmenata. Kako broj segmenata jednog procesa može da varira, potrebno je voditi računa o veličini tabele segmenata. Za to se obično koristi *registar dužine tabele segmenata* (*Segment Table Length Register – STLR*).



Slika 5.26. Preslikavanje logičkih adresa u fizičke

Mapiranje logičkih adresa u fizičke sastoji se od sledećih akcija (slika 5.26.). Najpre se za logičku adresu (*s*, *d*) proverava da li je broj segmenta

validan, tj. da li je s < STLR. Ako je broj segmenta regularan, onda se on dodaje na vrednost registra STBR. Time se dobija memorijska adresa odgovarajućeg reda u tabeli segmenata. Sa te adrese se učitava dužina segmenta i fizička adresa početka segmenta. Prvo se proverava da li je pozicija u segmentu validna, tj. da li je d manje od dužine segmenta. Ako je to tačno, onda se na fizičku adresu početka segmenta dodaje d i dobija se odgovarajuća fizička adresa.

Kao i kod straničenja, preslikavanje logičkih adresa u fizičke zahteva dva obraćanja memoriji, što usporava rad računarskog sistema dva puta. Analogno kao kod straničenja efikasnost upravljanja memorijom ovakvim pristupom se može ubrzati korišćenjem asocijativne memorije.

5.4.2. Karakteristike segmentacije

Prednost segmentacije u odnosu na ostale koncepte se ogleda u zaštiti memorije. S obzirom da segmenti predstavljaju logički definisane delove programa, verovatno je da će se sve stavke segmenta koristiti na isti način. Dakle, očekivano je da neki segmenti sadrže instrukcije, a neki podatke. Segmenti koji sadrže instrukcije mogu se označiti tako da se mogu samo čitati i da se ne mogu modifikovati, čime se štite od neželjenih promena i pristupa ako se kôd ne može samomodifikovati.

Takođe, segmentacija pruža mogućnost deljenja koda i podataka između različitih procesa. Naime, svaki proces ima svoju tabelu segmenata. Segmenti se dele ukoliko stavke u tabelama segmenata dva različita procesa ukazuju na istu fizičku lokaciju.

Dodeljivanje memorije kod segmentacije je slično kao i kod straničenja. Jedina razlika je u tome što su segmenti najčešće različite veličine. Potrebno je pronaći i dodeliti memoriju za sve segmente korisničkog programa. Dodela memorije predstavlja problem dinamičke dodele memorije i može se rešiti korišćenjem sličnih algoritama kao i kada je straničenje u pitanju.

Veličina segmenata može biti različita, pa zato segmentacija podseća na dinamičko particionisanje. Razlika je u tome što se kod segmentacije

proces ne nalazi u jednom memorijskom bloku, već se delovi procesa mogu naći bilo gde u memoriji. Kao i dinamičko particionisanje, segmentacija rešava problem interne fragmentacije, ali ostaje problem eksterne fragmentacije. Međutim, ovde je eksterna fragmentacija manje izražena zato što su segmenti manji od kompletnog programa, pa se lakše mogu umetnuti u uske slobodne delove memorije.

U nekim implementacijama kombinuju se segmentacija i straničenje kako bi se iskoristile prednosti i umanjili nedostaci koje imaju ova dva koncepta. *Segmentacija sa straničenjem* predstavlja pristup koji podrazumeva podelu segmenata na stranice čime se omogućava da segmenti ne moraju da se smeštaju u neprekidne memorijske blokove. Na ovaj način se veliki segmenti lakše smeštaju u memoriju i smanjuje se eksterna fragmentacija.

Pitanja

5.1. Šta je memorija?
5.2. Kako se na osnovu načina na koji se čuvaju podaci mogu podeliti memorije?
5.3. Kako se na osnovu brzine pristupa mogu podeliti memorije?
5.4. Koje su aktuelne karakteristike memorija?
5.5. Šta su registri?
5.6. Koja je osnovna uloga keš memorije?
5.7. Šta se nalazi u primarnoj memoriji?
5.8. Koje su glavne odlike RAM memorije?
5.9. Šta je ROM memorija?
5.10. Šta se nalazi u ROM memoriji?
5.11. Koja je uloga sekundarne memorije?
5.12. Navesti primere sekundarnih memorija i prodiskutovati njihove karakteristike.
5.13. Od čega se sastoji memorija?
5.14. Šta je memorijska reč?
5.15. Šta su simboličke, a šta fizičke adrese?
5.16. Kada se može obaviti povezivanje logičkih i fizičkih adresa?
5.17. Objasniti pojam relativne adrese.
5.18. U čemu se sastoji interakcija procesora i memorije tokom izvršavanja procesa?
5.19. Šta je fizički adresni prostor?
5.20. U kojim situacijama se logički i fizički adresni prostori podudaraju?
5.21. Koja je uloga jedinice za upravljanje memorijom (MMU uređaja)?
5.22. Koji su glavni uslovi koje bi operativni sistemi trebalo da ispune prilikom upravljanja memorijom?
5.23. Objasniti koncept monoprogramiranja.
5.24. Na koji način se obezbeđuje zaštita koda i podataka kod monoprogramiranja.
5.25. Na čemu se zasniva tehnika prebacivanja za upravljanje memorijom?
5.26. Na koji način se može poboljšati efikasnost tehnike prebacivanja?

5.27. Koje su glavne odlike tehnike particionisanja memorije?
5.28. Od čega zavisi stepen multiprogramiranja u sistemima koji podržavaju particionisanje memorije?
5.29. Na koji način se realizuje zaštita particija od neželjenih pristupa?
5.30. Kakvi pristupi postoje kada je particionisanje memorije u pitanju?
5.31. Koje su odlike statičkih particija?
5.32. Od čega zavisi broj statičkih particija u memoriji?
5.33. Kakve strategije postoje kada je u pitanju izbor particije u koju će određeni proces biti smešten?
5.34. Koje su odlike dinamičkog pristupa kod particionisanja memorije?
5.35. Na koji način se obično vodi evidencija o slobodnim i zauzetim delovima memorije kada je dinamički pristup particionisanja memorije u pitanju?
5.36. Šta je fragmentacija u sistemima sa particijama?
5.37. Kakvi tipovi fragmentacije postoje?
5.38. Koji tipovi fragmentacije se javljaju kod statičkih, a koji kod dinamičkih particija?
5.39. Šta je kompakcija?
5.40. Koji načini za dodelu raspoložive memorije procesu postoje?
5.41. Prodiskutovati pristup prvo poklapanje za dodeljivanje memorije i njegovu modifikaciju – sledeće poklapanje.
5.42. Analizirati slučajno (*random*) raspoređivanje.
5.43. Prodiskutovati pristup najbolje poklapanje za smeštanje procesa u memoriju.
5.44. Analizirati pristup najgore poklapanje kada je smeštanje procesa u memoriju u pitanju.
5.45. Šta je glavna novina koju je straničenje donelo kada je upravljanje memorijom u pitanju?
5.46. Od čega se sastoji logička adresa kada je straničenje u pitanju?
5.47. Šta je tabela stranica?
5.48. Na koji način se logička adresa prevodi u fizičku kod straničenja?
5.49. Navesti prednosti koje donosi veličina stranice koja je jednaka stepenu broja 2 kada je straničenje u pitanju.

5.50. Na koji način se pokreće novi proces na sistemima koji podržavaju straničenje?
5.51. Kakva je situacija sa fragmentacijom kada je straničenje u pitanju?
5.52. Kakva je hardverska podrška potrebna da bi se realizovalo straničenje?
5.53. Šta je asocijativna memorija?
5.54. Analizirati prednosti koje donosi korišćenje asocijativne memorije za straničenje.
5.55. Šta se podrazumeva pod pojmom nivo pogotka (*hit ratio*) kada je straničenje u pitanju?
5.56. Koja je osnovna ideja segmentacije?
5.57. Od čega se sastoji logička adresa kada je segmentacija u pitanju?
5.58. Prodiskutovati prednosti i mane segmentacije.
5.59. Prodiskutovati pristupe koji kombinuju koncepte segmenatcije i straničenja.

6. VIRTUELNA MEMORIJA

Virtuelna memorija (*Virtual memory*) predstavlja način upravljanja memorijom koji omogućava da se procesu na raspolaganje stavi memorija koja je drugačije veličine od one koja stvarno (fizički) postoji u sistemu. Pri tome je operativni sistem zadužen da omogući preslikavanje virtuelne memorije u fizičku memoriju. Drugim rečima, virtuelna memorija je memorijski model koji razdvaja memorijski prostor koji je dostupan programeru, odnosno procesu za izvršavanje, od fizičkog memorijskog prostora koji ima radna memorija.

Pojam virtuelne memorije se često, pogrešno, povezuje isključivo sa korišćenjem hard diska u situacijama kada radna memorija nije dovoljno velika da prihvati sve pokrenute procese, odnosno sve delove potrebne da bi se proces izvršavao. Međutim, prve upotrebe koncepta virtuelne memorije podrazumevale su da se, za tadašnje prilike, velika radna memorija podeli na manje delove koje bi onda programeri koristili za smeštanje svojih procesa. Sistemi sa deljenjem vremena (*Time-sharing*) su tipičan primer sistema koji su zasnovani na ovakvoj implementaciji virtuelne memorije.

Na savremenim sistemima, pojam virtuelne memorije se najčešće vezuje za tehniku koja omogućava da se izvršava proces koji nije u potpunosti u radnoj memoriji. Direktna (pozitivna) posledica ovog koncepta je veći stepen multiprogramiranja. Pošto je memorija bolje iskorišćena, može se učitati veći broj procesa. Još jedna prednost ovakvog koncepta je ta da korisnički procesi mogu biti veći od fizičke memorije. Takođe, pošto se u memoriji nalaze najčešće korišćeni delovi programa, smanjuje se broj ulazno-izlaznih operacija za zamenu i učitavanje procesa, čime će se efektivno svaki proces brže izvršavati.

Ovakav pristup nije jednostavno implementirati na zadovoljavajući način pa bi ga trebalo oprezno koristiti. Naime, loša realizacija memorijskog modela može da dovede do ozbiljnog gubitka performansi sistema i

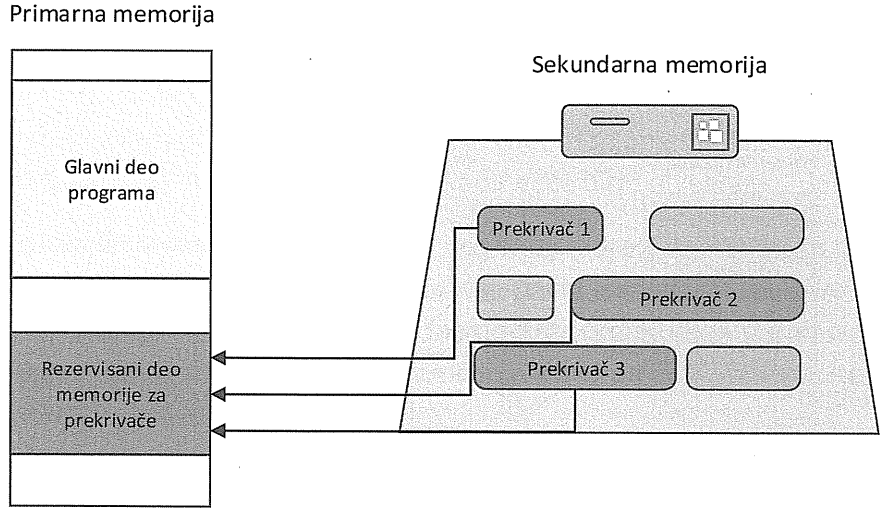
gubitka podataka. U istoriji operativnih sistema je bilo dosta pokušaja implementacije koncepta virtuelne memorije, a najpopularniji i najuspešniji su bili:

- Prekrivači;
- Dinamičko punjenje;
- Straničenje na zahtev.

6.1. Prekrivači

Prekrivači (*Overlays*) predstavljaju jedan od prvih pristupa za upravljanje memorijom, zasnovanih na ideji da se omogući izvršavanje programa koji nisu kompletno učitani u memoriju. Osnovni zadatak je da se identifikuju moduli programa koji su relativno nezavisni. Ovi moduli se nazivaju prekrivači. Oni obično sadrže delove programa koji se koriste u različitim vremenskim trenucima, delove koji se retko koriste, ali i one za koje postoji šansa da ne budu potrebni tokom izvršavanja procesa. Na primer, takvi su delovi za reakciju na greške određenog tipa, delovi za obradu izuzetaka itd.

Prekrivači se odvajaju od glavnog dela programa i smeštaju u sekundarnu memoriju. Prilikom alociranja memorije za glavni program, rezerviše se jedan slobodan deo koji je dovoljno velik za prihvatanje najvećeg prekrivača. Kada je, za dalje izvršavanje programa, potreban deo programa iz nekog prekrivača, taj prekrivač se učitava u rezervisani deo memorijskog prostora (slika 6.1.). Nakon toga, program može da izvršava zahtevane instrukcije. Kasnije, kada se zahteva deo programa iz drugog prekrivača, taj prekrivač će biti učitan u rezervisani deo memorije, tj. prepisaće se njen prethodni sadržaj (prekrivač). Drugim rečima, prekrivači će se po potrebi smenjivati u rezervisanom delu memorije. Naprednije verzije pristupa za upravljanje memorijom zasnovane na prekrivačima omogućavaju prisustvo više prekrivača u memoriji i samim time imaju komplikovaniji način njihovog raspoređivanja u rezervisanom delu memorije.



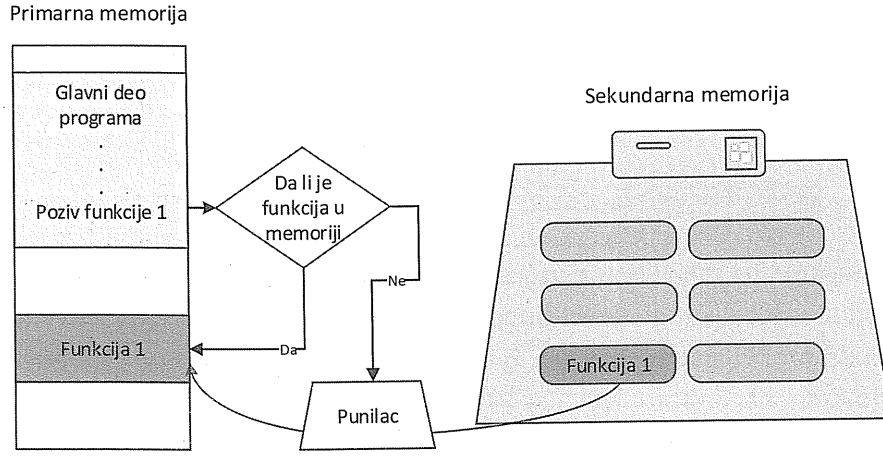
Slika 6.1. Prekrivači

Korišćenje prekrivača i sličnih metoda bilo je popularno osamdesetih i devedesetih godina dvadesetog veka, jer su resursi računarskih sistema bili veoma skupi i ograničeni. Najveći nedostatak ovakvog pristupa je to što operaciju deljenja programa na delove precizno i dovoljno dobro može da uradi jedino programer, odnosno ona se ne može automatizovati. Bilo je pokušaja da se implementiraju automatski prekrivači koje formira sistem na osnovu pretpostavki o izvršavanju programa. Međutim, oni su bili neuspešni, jer je veoma teško predvideti tok izvršavanja programa.

6.2. Dinamičko punjenje

Dinamičko punjenje je zasnovano na sličnim principima na kojima se zasnivaju i prekrivači. Osnovna ideja kod ovog pristupa je da se funkcije i procedure ne smeštaju u memoriju sve do trenutka dok ne budu potrebne, odnosno pozvane. Funkcije i procedure se nalaze na sekundarnoj memoriji, a glavni program se smešta u memoriju i izvršava. U trenutku kada se pozove neka od funkcija ili procedura, sistem prvo proveri da li je ona već u memoriji. Ako to nije slučaj, vrši se njeno prebacivanje sa sekundarnog medijuma. Punilac se poziva da bi obavio

ovo punjenje memorije, ali i ažurirao odgovarajuće tabele kako bi odgovarale novom stanju u sistemu (slika 6.2.). Nakon učitavanja, nova funkcija počinje da se izvršava.



Slika 6.2. Dinamičko punjenje

Iz prethodnog je jasno da je glavna prednost dinamičkog punjenja to što se funkcije koje nisu potrebne za izvršavanje procesa nikada ne pune u memoriju. Ovakav pristup naročito dolazi do izražaja u programima koji imaju dosta koda koji se odnosi na slučajeve koji se retko ili skoro nikada ne događaju.

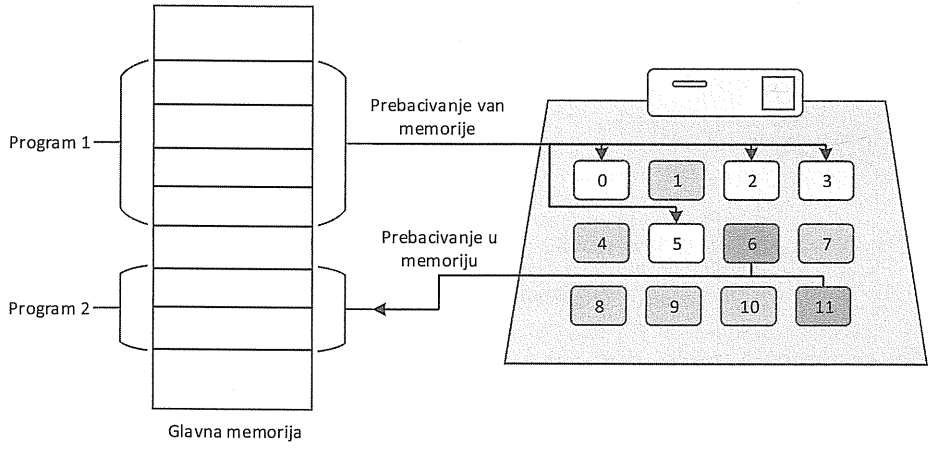
Dinamičko punjenje ne zahteva podršku operativnog sistema tako da ga korisnici mogu relativno jednostavno implementirati. Međutim, efikasnije funkcionisanje sistema iziskuje metode koje će se izvršavati automatski bez kontrole korisnika.

6.3. STRANIČENJE NA ZAHTEV

Najpopularniji način implementacije virtuelne memorije, koji se u praksi pokazao kao najefikasniji, je *straničenje na zahtev* (*Demand Paging*). Slično kao kod straničenja, ovaj pristup deli program na stranice (slika 6.3.). Prilikom pokretanja programa u radnu memoriju se smeštaju samo one stranice koje su neophodne u prvom trenutku. Time se smanjuje

količina zauzete memorije i povećava stepen multiprogramiranja. Kada se neka stranica zatraži pri izvršavanju procesa, prvo se proverava da li je ta stranica već u memoriji, pa ako nije pronalazi se na sekundarnoj memoriji i prebacuje u radnu. Naravno, ovakvim pristupom se postiže da se u radnu memoriju ne učitavaju stranice koje nikada neće biti potrebne.

Kod straničenja na zahtev, veličina logičkog adresnog prostora nije ograničena fizičkom memorijom. Na taj način se obezbeđuje velika virtuelna memorija na sistemima sa manjom fizičkom memorijom, čime se olakšava programiranje jer programer ne mora da brine o kapacitetu raspoložive fizičke memorije.

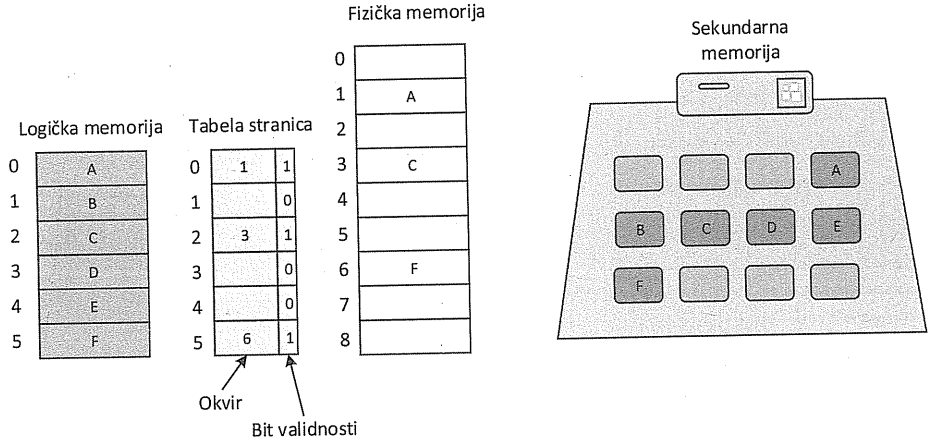


Slika 6.3. Straničenje na zahtev

Straničenje na zahtev se može shvatiti kao kombinacija straničenja sa prebacivanjem (*swapping*). Za razliku od klasičnog straničenja, ova metoda dozvoljava da neke od stranica ne budu u radnoj memoriji tokom izvršavanja procesa, već da se po potrebi učitavaju. Za potrebe odloženog učitavanja koristi se poseban modul koji se naziva *lenji prebacivač* (*lazy swapper*) ili *pejdžer* (*pager*).

Poput klasičnog straničenja, straničenje na zahtev se implementira uz pomoć tabele koja sadrži informacije o stranicama. Međutim, ova *tabela stranica* obično ima i dodatnu kolonu u kojoj se čuva bit validnosti (slika 6.4.). Za stranicu se kaže da je „validna“ ako se nalazi u radnoj memoriji i

tada je odgovarajući bit validnosti postavljen na 1. Ako stranica nije u memoriji, onda je ona „nevalidna“ i odgovarajući bit validnosti ima vrednost 0. Kada proces pokuša da pristupi nekoj stranici, u tabeli stranica se proverava ovaj bit. Ako je stranica već u radnoj memoriji, može joj se odmah pristupiti. U suprotnom dolazi do takozvanog *promašaja stranice* (*page fault*).

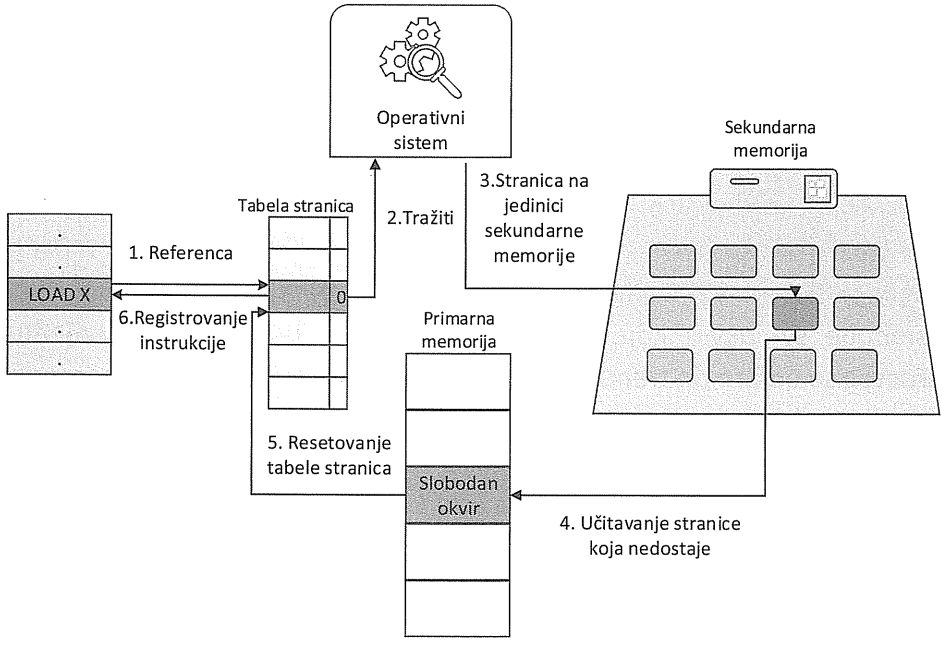


Slika 6.4. Straničenje na zahtev sa bitom validnosti

Do promašaja neće doći ako proces nikada ne pokuša da pristupi stranicama koje nisu u radnoj memoriji, odnosno ako se napravi dobra procena i u radnu memoriju smeste sve potrebne stranice. U tom slučaju proces će se izvršavati na isti način kao i kada su sve stranice u memoriji. U suprotnom, ako proces pokuša da pristupi stranici koja nije u memoriji, javlja se promašaj stranice, a zatim i prekid. Na prekid se reaguje i omogućava nastavak izvršavanja procesa donošenjem potrebne stranice u memoriju (slika 6.5.).

Kada dođe do promašaja stranice, prvo se vrši provera da li tražena stranica uopšte pripada procesu koji ju je zatražio. Ako to nije slučaj, dolazi do prekida programa usled greške pri adresiranju. Ukoliko stranica pripada procesu, onda je potrebno učitati je u radnu memoriju. Učitavanje se izvodi u tri koraka:

1. Pronalazi se slobodan memorijski okvir ukoliko postoji ili oslobađa jedan od zauzetih ukoliko su svi okviri zauzeti.
2. Učitava se stranica iz sekundarne u radnu memoriju.
3. Modifikuje se tabela stranica tako da bit validnosti govori da je stranica u radnoj memoriji.



Slika 6.5. Učitavanje odgovarajuće stranice posle promašaja

Donošenjem potrebne stranice stvaraju se uslovi za nastavak izvršavanja prekinutog procesa na mestu gde je došlo do promašaja stranice. Na ovaj način se omogućava izvršavanje procesa koji nije u potpunosti učitan u radnu memoriju.

Jedan od specijalnih slučajeva straničenja na zahtev je takozvano *čisto straničenje*. Ovakav pristup podrazumeva da proces krene u izvršavanje bez i jedne učitane stranice. Promašaji će se ređati u početku sve dok se ne iskristališe radni skup stranica koje su potrebne za izvršavanje.

6.3.1. Zamena stranice

Jedan od ciljeva straničenja na zahtev jeste povećanje stepena multiprogramiranja, tako da se teži učitavanju što većeg broja procesa u radnu memoriju. Da bi to bilo moguće, potrebno je da se za svaki proces minimizuje broj stranica učitanih u memoriju. Čak i u slučaju optimalnog izbora stranica za procese, radna memorija može postati značajno popunjena nakon određenog vremena rada sistema. Problem nastaje kada je potrebno učitati stranicu potrebnu za izvršavanje nekog od procesa, a pri tome ne postoji slobodan memorijski okvir.

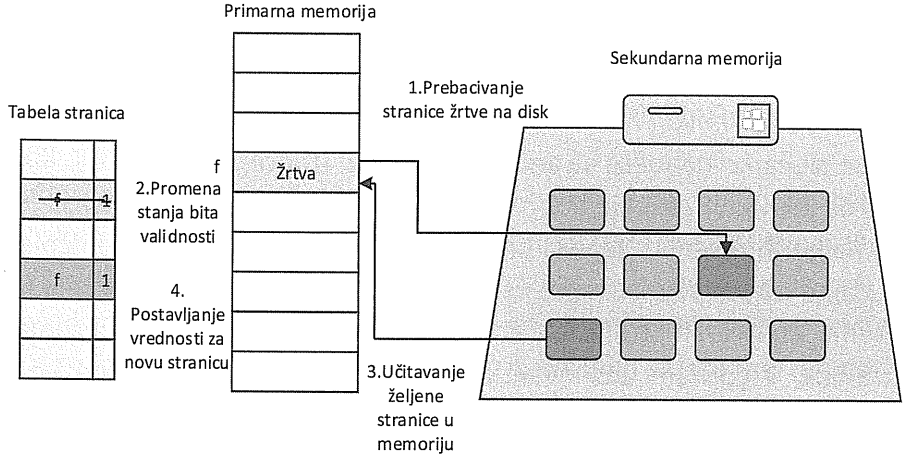
Neka od rešenja ovog problema podrazumevaju da se prekine onaj proces koji traži učitavanje stranice i kasnije ponovo pokrene od početka. Drugi način je da se sve stranice procesa prebace na spoljašnju memoriju i oslobode svi okviri koje je proces zauzimao. Izvršavanje sklonjenog procesa se odlaže do trenutka kada će moći da se učita u radnu memoriju i izvrši u celosti. Ovakvim postupcima se smanjuje efikasnost sistema, ali i stepen multiprogramiranja tako da se retko primenjuju. Ovakva rešenja je pogodno primeniti u sistemima kod kojih se promašaj stranice retko događa.

Rešenje koje se najčešće primenjuje je *zamena stranice*. Kada je memorija popunjena, a potrebno je učitati novu stranicu, ideja je da se po nekom kriterijumu pronađe okvir (žrtva) koji će se isprazniti kako bi se u njega smestila nova stranica. Izabrana stranica se iz okvira prebacuje na sekundarnu memoriju i pri tome se ažurira tabela stranica tako da odgovara trenutnom stanju u sistemu. Na ovaj način se oslobađa okvir za učitavanje nove stranice (slika 6.6.).

Algoritam zamene stranice se može predstaviti na sledeći način:

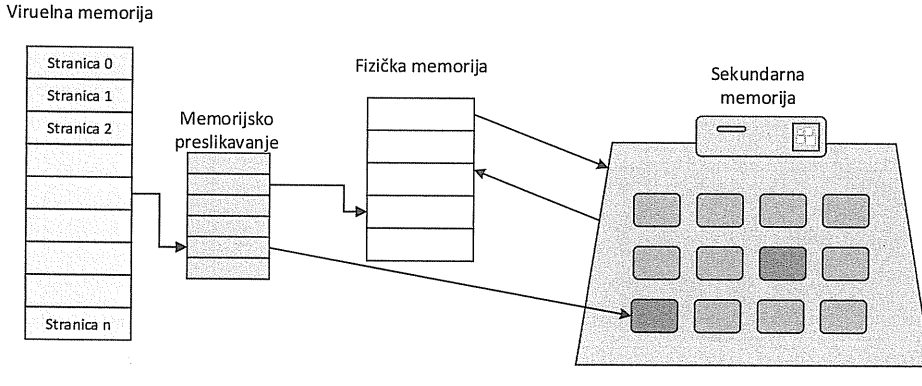
1. Naći lokaciju željene stranice na disku.
2. Naći slobodan okvir.
   a. Ako postoji slobodan okvir, koristiti njega.
   b. Ako ne postoji, upotrebiti algoritam za izbor okvira žrtve.
   c. Upisati žrtvu na disk i saglasno tome izmeniti tabelu stranica.

3. Učitati željenu stranicu u novooslobođeni okvir i ažurirati tabelu stranica.
4. Nastaviti sa izvršavanjem procesa.



Slika 6.6. Zamena stranice

Zamena stranice je veoma bitna za straničenje na zahtev jer ona omogućava njenu realizaciju i efikasnost. Uz pomoć dobro realizovane zamene stranica postiže se razdvajanje logičke memorije od fizičke memorije (slika 6.7.).



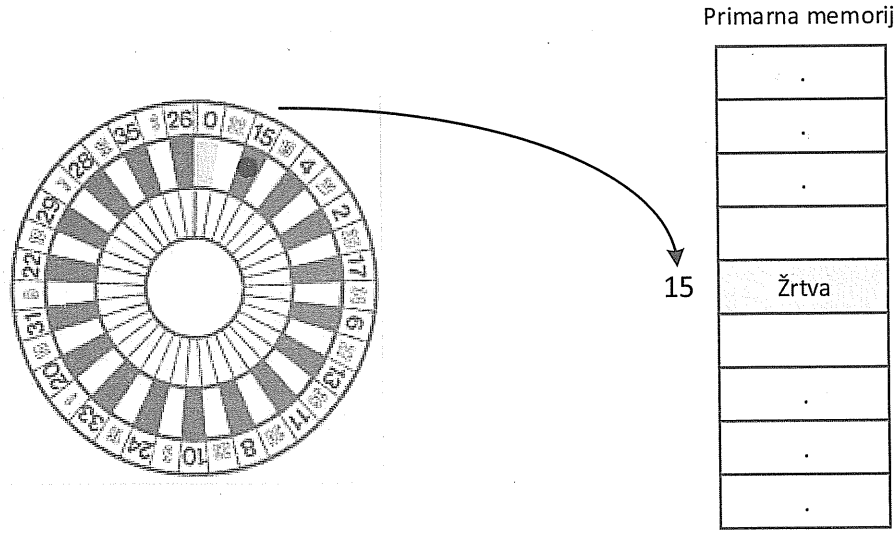
Slika 6.7. Razdvajanje fizičke i logičke memorije

6.4. ALGORITMI ZA IZBACIVANJE STRANICE

Postoje razni algoritmi za izbor stranice koja će biti izbačena iz memorije, a neki od njih biće predstavljeni u nastavku.

6.4.1. Algoritam zasnovan na slučajnom izbacivanju

Algoritam zasnovan na slučajnom izbacivanju stranice se zasniva na ideji da se slučajno odabere stranica koja će biti izbačena (slika 6.8.). Iako se čini da je ovakvo rešenje dobro i pravedno, u praksi se pokazalo kao jedno od najgorih rešenja. Najveći problem kod ovog algoritma je činjenica da slučajno izbacivanje stranice često može da dovede do izbacivanja nekih veoma korišćenih stranica, čime se doprinosi velikom broju promašaja stranica. Algoritam zasnovan na slučajnom izbacivanju stranica nije zanemarljive vremenske složenosti jer implementacija slučajnog izbora stranice nije trivijalna, a ne daje ni dobre rezultate. Zbog toga se ovaj algoritam retko koristi i najčešće služi samo za poređenje sa drugim algoritmima.



Slika 6.8. Algoritam zasnovan na slučajnom izbacivanju

6.4.2. Beladijev optimalni algoritam

Nasuprot prethodnom algoritmu, *Beladijev* (*László Bélády*) *pristup* predstavlja najbolji mogući način za izbor stranice koja će biti izbačena iz memorije. Ovaj pristup se naziva *Beladijev optimalni algoritam* za zamenu stranice.

U trenutku kada nastane promašaj stranice, određeni skup stranica jednog procesa nalazi se u memoriji. Neke od tih stranica mogu biti potrebne već u sledećoj ili u nekoj od skorijih instrukcija. Sa druge strane, neke stranice neće biti referisane u skorije vreme. Osnovna ideja Beladijevog optimalnog algoritma je da se izbaci neka od stranica koja bi najkasnije u budućnosti bila potrebna. Na primer, neka se program sastoji od 5 stranica označenih brojevima od 1 do 5, od kojih se u radnoj memoriji mogu naći najviše 3 stranice. Neka je redosled stranica 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. U tabeli 6.1. se može videti da će, nakon određenog vremena, u memoriju biti učitane stranice 1, 2, 3. U sledećem trenutku program zahteva stranicu 4, pa je potrebno izbaciti neku od učitanih stranica. Pošto se iz datog redosleda može videti da će treća stranica najkasnije biti potrebna, ona se izbacuje iz memorije. Na njeno mesto učitava se stranica broj 4, a isti postupak se nastavlja i za naredne stranice. U tabeli 6.1. se može videti da je ukupno bilo 7 promašaja stranica.

Tabela 6.1. Beladijev pristup

	Vreme											
Veličina	**1**	1	1	1	1	1	1	1	1	**3**	3	3
		2	2	2	2	2	2	2	2	2	**4**	4
			3	**4**	4	4	**5**	5	5	5	5	5
	F	F	F	F			F			F	F	

Najveći problem Beladijevog optimalnog algoritma je taj što ga je skoro nemoguće implementirati. Naime, kada nastane promašaj stranice, operativni sistem nije u mogućnosti da zna koja stranica će biti potrebna u sledećem koraku. Zato je ovaj algoritam više teoretskog značaja i koristi se za upoređivanje sa realnim algoritmima.

6.4.3. FIFO algoritam

FIFO (*First In, First Out*) *algoritam* je jedan od pristupa koji se prirodno nameće i predlaže da se iz memorije izbacuju stranice po redosledu po kojem su učitane. Loša osobina ovog algoritma je činjenica da se u memoriju obično na početku učitavaju stranice koje su važne i često se koriste, a one bi ovakvim pristupom bile relativno brzo izbačene. U tabeli 6.2. je dat primer ponašanja FIFO algoritma za istu sekvencu zahtevanih stranica kao u prethodnom primeru. U ovom slučaju FIFO algoritam je prouzrokovao 9 promašaja stranica.

Tabela 6.2. FIFO algoritam

	Vreme											
Veličina	**1**	1	1	**4**	4	4	**5**	5	5	5	5	5
		2	2	2	**1**	1	1	1	1	**3**	3	3
			3	3	3	**2**	2	2	2	2	**4**	4
	F	F	F	F	F	F	F			F	F	

Kada je implementacija u pitanju, potreban je jedan memorijski registar po stranici u koji bi se upisivao trenutak kada je stranica ušla u memoriju, a onda bi se ta vremena sortirala. Sve to je vremenski, ali i prostorno, veoma zahtevno. Zato se češće za implementaciju FIFO algoritma koristi povezana lista u koju se smeštaju redni brojevi stranica. Prilikom učitavanja stranice njen redni broj se smešta na kraj liste, a stranica za izbacivanje se nalazi na početku liste. Na taj način se rešava problem čuvanja i sortiranja vremena ulaska stranica.

Tabela 6.3. FIFO anomalija

	Vreme											
Veličina	**1**	1	1	1	1	1	**5**	5	5	5	**4**	4
		2	2	2	2	2	2	**1**	1	1	1	**5**
			3	3	3	3	3	3	**2**	2	2	2
				4	4	4	4	4	4	**3**	3	3
	F	F	F	F			F	F	F	F	F	F

U nekim slučajevima, kada se poveća broj okvira za neki proces, može se desiti da se poveća i broj promašaja stranica. Ova pojava se naziva *FIFO* ili *Beladijeva anomalija*. Upravo navedena sekvenca zahtevnih stranica ispoljava ovo ponašanje kada se broj mogućih okvira poveća na četiri (tabela 6.3.). U tom slučaju FIFO algoritam dovodi do 10 promašaja stranica.

Ipak, ovaj primer je veštački napravljen. Verovatnoća da u praksi dođe do FIFO anomalije je veoma mala. U praksi se radi sa nekoliko hiljada stranica, pa je gotovo nemoguće da se dogodi primer FIFO anomalije.

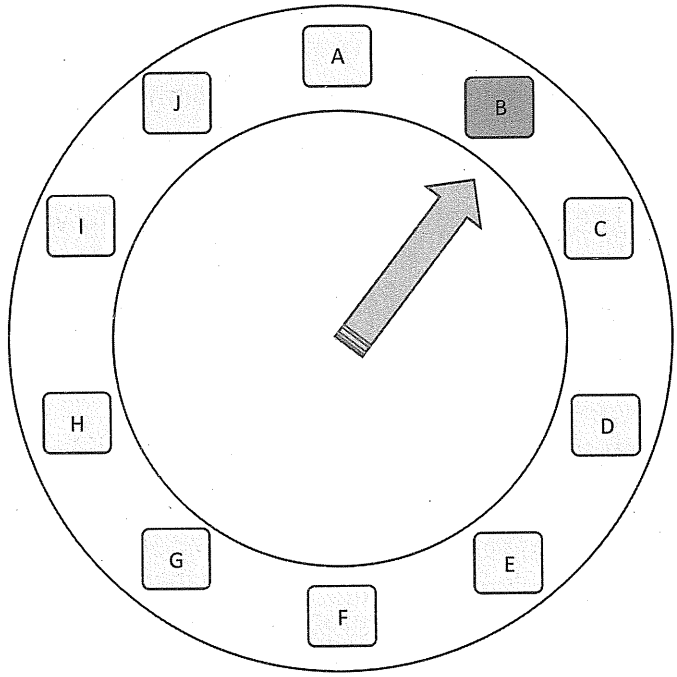
6.4.4. Algoritam druge šanse

FIFO algoritam se na jednostavan način može modifikovati tako da se izbegne izbacivanje veoma bitnih stranica koje se prve učitavaju. *Algoritam druge šanse* (*Second Chance Algorithm*) je uobičajeni naziv za modifikovani algoritam. Modifikacija podrazumeva postojanje jednog bita po stranici koji se naziva *bit referisanosti* (*R*).

Bit referisanosti se postavlja na 1 kada se stranica učitava u memoriju i kada se pristupa stranici. Potraga za stranicom za izbacivanje kreće od prve učitane stranice. Ako je njen bit referisanosti 1, onda se on postavlja na 0, a vreme učitavanja stranice se resetuje na tekuće vreme (ili se redni broj stranice premešta na kraj povezane liste), tako da se stranica tretira kao poslednja učitana. Drugim rečima, stranici se daje „druga šansa". Ukoliko se ta stranica često koristi, njen bit referisanosti će se vrlo brzo opet postaviti na 1. Kada pretraga naiđe na stranicu sa bitom referisanosti jednakim 0, to znači da ta stranica nije bila referisana od trenutka kada joj je data druga šansa, pa se izbacuje iz memorije. Problem pri korišćenju ovog algoritma se javlja u slučaju kada su svi bitovi referisanosti postavljeni na 1. Tada će algoritam posetiti svaku stranicu i postaviti njen bit referisanosti na 0. Na kraju, pretraga za stranicom žrtvom će se vratiti na početak i tada će prva posećena stranica biti izbačena.

6.4.5. Algoritam sata

Algoritam sata (*Clock Algorithm*) je modifikacija algoritma druge šanse koja se pokazala bolje u praksi. Naime, kod algoritma druge šanse se stranica često premešta na kraj povezane liste stranica. Pri tome, nije izvesno da će to premeštanje poboljšati efikasnost izvršavanja procesa i smanjiti nastanak promašaja stranica.



Slika 6.9. Algoritam sata

Prilikom primene algoritma sata svi memorijski okviri jednog procesa se posmatraju kao povezana kružna lista (slika 6.9.). Za njegovu implementaciju se koristi jedan pokazivač, koji u početku pokazuje na prvi memorijski okvir. I ovde se svakoj stranici pridružuje bit referisanosti čija vrednost ima isto značenje kao kod algoritma druge šanse. Kada su svi okviri zauzeti, pretraga za stranicom koju treba izbaciti kreće od memorijskog okvira na koji pokazuje pokazivač. Ako je bit referisanosti te stranice 1, on se postavlja na 0, a pokazivač se pomera za jedno mesto. Kada se naiđe na stranicu čiji je bit referisanosti 0, stranica se izbacuje iz

memorije, a na njeno mesto se učitava zahtevana stranica. Na kraju, pokazivač se pomera za jedno mesto unapred, odakle će krenuti naredna potraga za stranicom koja treba da bude žrtvovana. Kod ovog algoritma se, takođe, javlja problem kada su svi bitovi referisanosti postavljeni na 1.

6.4.6. LRU algoritam

LRU (*Least Recently Used*) *algoritam* se zasniva na ideji da se iz memorije izbaci stranica koja je korišćena najdalje u prošlosti u odnosu na ostale stranice. U ovom pristupu pokušava se predviđanje dalje sekvence zahtevanih stranica na osnovu prethodnog izvršavanja programa. Predviđanje se zasniva na pretpostavci da su one stranice koje su bile skoro korišćene veoma važne za proces, pa će biti potrebne i u bliskoj budućnosti. Suprotno njima, pretpostavlja se da stranice, koje nisu zahtevane u skorije vreme, verovatno neće biti potrebne ni u bliskoj budućnosti.

Implementacija ovakvog algoritma zahteva registar koji bi čuvao informacije o vremenu korišćenja stranice. Ta vremena je potrebno sortirati pri svakom izbacivanju stranice. Zbog toga je LRU algoritam veoma zahtevan za implementaciju. Primer izvršavanja LRU algoritma dat je u tabeli 6.4.

Tabela 6.4. LRU

	Vreme											
Veličina	**1**	1	1	**4**	4	4	**5**	5	5	**3**	3	3
		2	2	2	**1**	1	1	1	1	1	**4**	4
			3	3	3	**2**	2	2	2	2	2	**5**
	F	F	F	F	F	F	F			F	F	F

Najveća mana ovog algoritma odnosi se na činjenicu da se nikada ne može predvideti koja će sledeća stranica procesu biti potrebna, pa se može napraviti primer na kojem se LRU algoritam ne ponaša na željeni način. Na primer, neka u sekvenci zahtevanih stranica postoji ciklus dužine n koji se stalno ponavlja. Ako je broj okvira dodeljenih procesu $n - 1$, tada će u

svakom koraku doći do promašaja stranice. Međutim, i ovaj primer je veštački napravljen i u stvarnosti malo verovatan.

6.4.7. NRU algoritam

NRU (*Not Recently Used*) *algoritam* predstavlja aproksimaciju LRU algoritma. NRU algoritam predlaže izbacivanje stranice koja nije skoro korišćena, pri čemu ta stranica ne mora nužno biti najduže nekorišćena. Zbog toga je implementacija NRU algoritma jednostavnija u odnosu na LRU, jer od hardvera zahteva postojanje jednog bita po stranici.

Bit referisanosti se postavlja na jedinicu onda kada se ta stranica koristi. Posle izvesnog vremena, može se dogoditi da su svi bitovi postavljeni na jedinicu, odnosno da se sve stranice koriste bar jednom. U tom slučaju sve stranice su ravnopravne pa je potrebno uvesti dodatne modifikacije algoritma koje će sprečiti nastanak ovakvih situacija. Jedna od njih se sastoji u periodičnom resetovanju svih bitova referisanosti na nulu. Za ovu modifikaciju najvažnije je odrediti vremensku konstantu t nakon koje će se resetovati bitovi. Posle vremena $t/2$, nakon resetovanja, neke stranice bi trebalo da imaju postavljene bitove referisanosti na 1, a neke na 0, pa su tada te stranice kandidati za izbacivanje. Odabrati konstantu t nije jednostavno jer ona može biti prekratka ili predugačka, što može dovesti do neželjenog ponašanja sistema. U praksi, obično za svaki sistem i svaki program treba eksperimentisati sa ovom konstantom dok se ne dođe do odgovarajuće vrednosti.

Ipak, posle resetovanja nastaje problem, a to je da su svi bitovi referisanosti postavljeni na 0. U tom trenutku se ne zna koju stranicu treba izbaciti jer su sve ravnopravne.

6.4.8. Modifikacija NRU algoritma

NRU algoritam se može modifikovati tako da daje bolje rezultate. Modifikacija podrazumeva da se uvede još jedan bit koji se naziva *bit modifikacije* (M). Ovaj bit je jednak nuli kada se stranica učita, a postavlja se na 1 ako se sadržaj stranice menja. Sa ovako definisanim bitom

modifikacije i prethodno definisanim bitom referisanosti moguće su sledeće njihove vrednosti (RM):

1. Stranica koja nije referisana ni modifikovana – 00;
2. Stranica koja nije referisana a modifikovana je – 01;
3. Stranica koja je referisana a nije modifikovana – 10;
4. Stranica koja je referisana i modifikovana – 11.

Ovim redom su stranice označene kao poželjne za izbacivanje.

Na prvi pogled, čini se da kombinacija 01 nije moguća jer je referisanje širi pojam od modifikovanja. Drugim rečima, čini se da bi stranica, koja je modifikovana istovremeno trebalo da bude i referisana, jer se stranica prvo referiše, pa se onda modifikuje. Međutim, kada istekne vremenska konstanta *t,* resetuju se samo bitovi referisanosti, a bitovi modifikacije ostaju nepromenjeni, pa samim tim i sve modifikovane stranice imaju ovu kombinaciju. Na ovaj način, stvara se nova klasa stranica za koje se smatra da su poželjnije od stranica kojima nije ni pristupano, a nisu ni modifikovane.

6.4.9. LFU i MFU algoritmi

Osnovna ideja *LFU* (*Least Frequently Used*) *algoritma* je da se iz memorije izbaci najmanje često korišćena stranica, odnosno stranica koja od svih stranica u memoriji ima najmanju frekvenciju korišćenja. Motivacija za takav postupak leži u težnji da se stranice koje se često koriste drže u memoriji, a one koje se ne koriste tako često drže u sekundarnoj memoriji. Implementacija ovog algoritma je hardverski zahtevnija od prethodno pomenutih jer treba voditi računa o frekvenciji pojavljivanja, odnosno stalno ažurirati i sortirati podatke. Takođe, ostaje otvoreno pitanje u kolikom vremenskom periodu treba meriti referisanje na stranice.

Velika mana ovakvog pristupa leži u lošem tretmanu stranica koje su tek ušle u sistem, ali su veoma potrebne i dosta će se koristiti u budućnosti. Frekvencija korišćenja takvih stranica je mala, pa će one biti kandidati za izbacivanje. Drugi problem LFU algoritma se javlja u slučajevima kada se

pojedine stranice koriste često u određenom vremenskom periodu, a kasnije neće biti potrebne. Nakon završetka rada sa takvim stranicama, njihova frekvencija je velika. Zato će one ostati u memoriji još neko vreme, iako u budućnosti neće biti potrebne procesu.

Za razliku od LFU algoritma, *MFU* (*Most Frequently Used*) algoritam podrazumeva da se iz memorije izbacuju stranice sa najvećom frekvencijom korišćenja. On se zasniva na ideji da će u budućnosti biti potrebnije stranice koje su tek ušle u sistem i imaju manju frekvenciju korišćenja. U nekim situacijama ovaj algoritam daje odlične rezultate. Međutim, MFU algoritam se susreće sa različitim problemima. Jedan od njih nastaje kada program tokom celog svog izvršavanja često koristi neku stranicu, pa se tada ta stranica često izbacuje.

Pitanja

6.1. Objasniti osnovnu ideju koncepta virtuelne memorije.
6.2. Kakve su bile prve upotrebe koncepta virtuelne memorije?
6.3. Za šta se obično vezuje pojam virtuelne memorije na savremenim računarskim sistemima.
6.4. Koje su najpoznatije implementacije virtuelne memorije?
6.5. Objasniti koncept prekrivača.
6.6. Šta je glavni nedostatak prekrivača?
6.7. Objasniti koncept dinamičkog punjenja.
6.8. U čemu je razlika između prekrivača i dinamičkog punjenja?
6.9. U kojim situacijama do izražaja dolaze prednosti koje donosi dinamičko punjenje?
6.10. Na kojem konceptu se zasniva straničenje na zahtev?
6.11. Šta je potrebno za implementaciju straničenja na zahtev?
6.12. U čemu je glavna razlika između straničenja i straničenja na zahtev?
6.13. Šta je bit validnosti i čemu služi?
6.14. Šta se podrazumeva pod pojmom promašaj stranice?
6.15. Koji se koraci preduzimaju kada dođe do promašaja stranice?
6.16. Šta se podrazumeva pod pojmom čisto straničenje?
6.17. Opisati algoritam za zamenu stranice.
6.18. Koje su mane slučajnog izbacivanja stranice iz memorije?
6.19. Koji je najveći nedostatak Beladijevog pristupa za izbor stranice za izbacivanje?
6.20. Šta je potrebno za implementaciju FIFO algoritma za izbacivanje stranica?
6.21. Šta je FIFO anomalija?
6.22. Konstruisati primer FIFO anomalije.
6.23. Objasniti algoritam druge šanse.
6.24. Šta je najveći problem kada je algoritam druge šanse u pitanju?
6.25. Objasniti kako funkcioniše algoritam sata.

6.26. Šta je potrebno za implementaciju algoritma sata?
6.27. Na kojoj ideji se zasniva LRU algoritam?
6.28. Prodiskutovati složenost LRU algoritma.
6.29. Šta je potrebno za implementaciju LRU algoritma?
6.30. Koja je najveća mana LRU algoritma?
6.31. Šta je NRU algoritam?
6.32. Koja je glavna prednost NRU algoritma u odnosu na LRU algoritam?
6.33. Na koji način se NRU algoritam može modifikovati tako da daje bolje rezultate?
6.34. Na kojoj ideji se zasniva LFU algoritam?
6.35. Prodiskutovati složenost LFU algoritma.
6.36. Šta je potrebno za implementaciju LFU algoritma?
6.37. Koja je najveća mana LFU algoritma?
6.38. Prodiskutovati MFU algoritam za izbacivanje stranice.

7. FAJL SISTEM

Trajno čuvanje podataka je mogućnost koja se podrazumeva i bez koje se ne mogu zamisliti savremeni računarski sistemi. Radna (primarna) memorija nije pogodna za skladištenje podataka jer njena veličina nije dovoljno velika. Naime, jasno je da npr. baza podataka (u nekoj firmi), koja sadrži ogromnu količinu podataka, ne bi mogla da se smesti u radnu memoriju. Takođe, problem predstavlja i činjenica da se sadržaj radne memorije dodeljen procesu gubi po prestanku rada sistema, ali i samog procesa. Iako bi se, u teoriji, proces mogao držati aktivnim sve vreme, u praksi se mora voditi računa i o neželjenim događajima kao što su gubitak napajanja, iznenadni prestanak izvršavanja operativnog sistema usled kritične greške (softverske ili hardverske), prestanak izvršavanja samog procesa usled greške itd. Zbog toga se za potrebe skladištenja podataka koriste sekundarne (spoljašnje) memorije.

Da bi se prevazišli prethodno navedeni problemi, ali i pružila što veća udobnost korisnicima, rešenje za trajno čuvanje podataka treba da zadovolji sledeće uslove:

1. Sačuvani podaci mogu biti trajno zapisani, tj. ne gube se po prestanku izvršavanja procesa koji ih koristi ili po prestanku rada sistema.
2. Velika količina podataka može biti sačuvana.
3. Podaci su nezavisni od procesa, tj. više procesa im može pristupiti, a ne samo onaj koji ih je kreirao.

7.1. INTERFEJS FAJL SISTEMA

Kao što se koncept procesora i njegov rad skriva, a korisniku se prikazuje samo apstrakcija u vidu procesa koji se izvršava, tako je i *fajl* (*file*) apstrakcija informacije zapisane na sekundarnoj memoriji. Ovu

apstrakciju pruža *fajl sistem* (*File system*) i to tako da korisnik „vidi" samo određeni interfejs pogodan za korišćenje, ali ne i detalje implementacije.

Centralni pojam u radu sa podacima jeste *fajl* (datoteka) koji predstavlja imenovanu kolekciju informacija, zapisanu u memoriji. Sa korisničke strane, to je najmanja struktura informacije koja se može upisati u memoriju. Fajl može sadržati podatke, ali i programe. Kada su u pitanju podaci, oni mogu biti zapisani u obliku teksta, ali i u drugim formama (često se ovi drugi nazivaju binarnim fajlovima), koje program za koji su namenjeni ume da pročita. Dakle, sadržaj fajla je u opštem slučaju niz bajtova, a interpretacija tog niza zavisi od namene, preciznije programa koji ima zadatak da ga pročita.

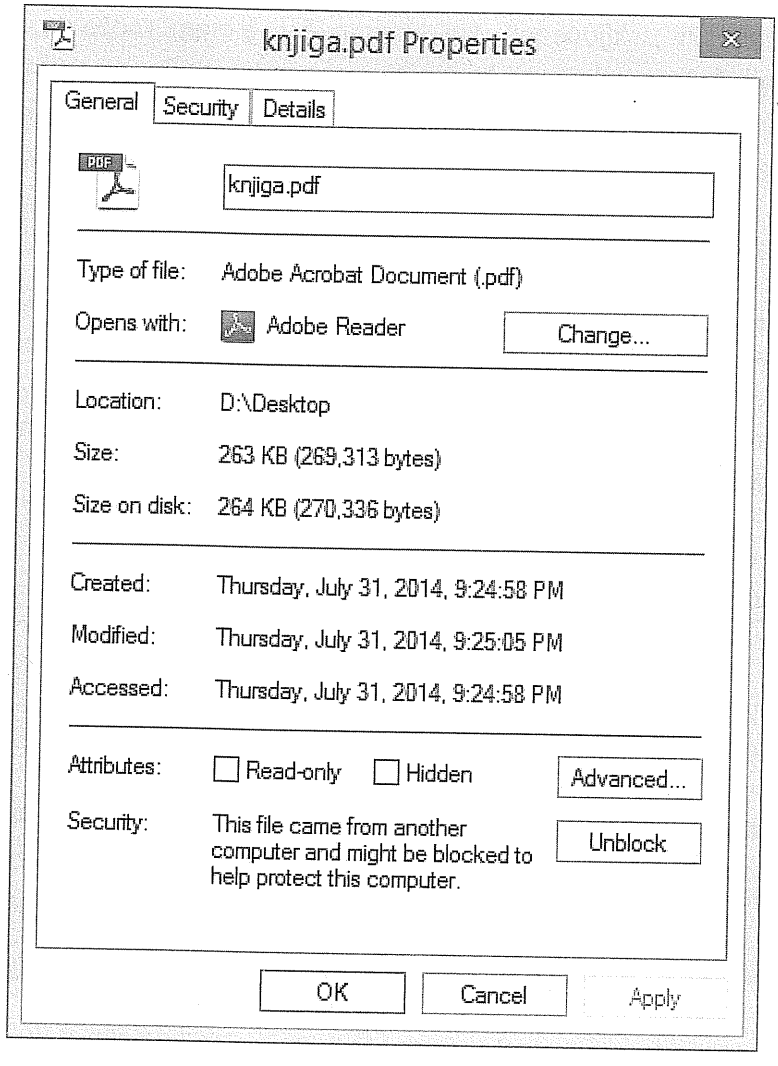
7.1.1. Atributi fajlova

Fajl sistem korisnicima daje mogućnost imenovanja fajla. Štaviše, svaki fajl mora imati ime koje postaje njegova referenca. Na taj način on prestaje da bude vezan za proces koji ga je kreirao i za sistem gde je nastao. Na primer, fajlovi se mogu kreirati na fleš memoriji i sa nje ih može pročitati neki drugi proces, na nekom drugom sistemu.

Skup atributa (informacija) koje fajl sistem čuva o fajlu razlikuje se od sistema do sistema, ali su obično podržani sledeći:

- Ime;
- Lokacija;
- Veličina;
- Vreme kreiranja, poslednje modifikacije i poslednjeg pristupa;
- Identifikator vlasnika fajla;
- Prava pristupa.

Neki od atributa su obavezni, kao što su ime fajla, ili njegova lokacija, koja pruža informaciju gde je fajl smešten, na kom uređaju i na kojoj lokaciji u strukturi fajl sistema tog uređaja. Drugi, kao što su vremena kreiranja ili modifikacije, nisu obavezni, tako da od konkretnog sistema zavisi da li će biti podržani. Na slici 7.1. prikazani su atributi fajla operativnog sistema Windows.



Slika 7.1. Atributi fajla – operativni sistem Windows

Informacije o vremenu kreiranja, modifikacije i poslednjeg pristupa mogu biti korisne u različitim situacijama. Na primer, ako je fajl koji sadrži izvorni kôd programa modifikovan nakon nastanka izvršnog fajla, to je znak da se program treba prevesti ponovo, kako bi se promene u izvornom kodu primenile.

Među podržanim atributima obično se nalazi i skup raznih *flegova* (zastavica). Fleg je obično jedan bit, i njegova vrednost se interpretira kao

„uključeno“ ili „isključeno“. Na primer, fajl sistem može postojanje određenih fajlova sakriti od korisnika, a za to se obično koristi fleg koji određuje da li je fajl sakriven ili ne. Zbog visoke važnosti, sistemski fajlovi se mogu obeležiti uključivanjem posebnog flega kojim bi se operativnom sistemu signaliziralo da prikaže specijalno upozorenje pri pokušaju modifikacije ili brisanja ovako označenih fajlova.

Na osnovu atributa koji definišu prava pristupa, operativni sistem određuje ko može da pristupi fajlu i modifikuje ga, a ko ne. Moguće su razne šeme, tako da se nekim korisnicima odredi samo pravo čitanja, drugima i čitanja i pisanja, ali ne izvršavanja itd. U tabeli 7.1. mogu se videti neki od uobičajenih atributa fajlova.

Tabela 7.1. Pregled atributa fajlova

Atribut	Značenje
Kreator	Korisnik koji je kreirao fajl
Vlasnik	Aktuelni vlasnik fajla
Vreme kreiranja	Tačan datum i vreme kreiranja fajla
Vreme poslednjeg pristupa	Datum i vreme poslednjeg pristupa fajlu
Vreme poslednje promene	Datum i vreme poslednje promene fajla
Trenutna veličina	Broj bajtova fajla
Šifra	Lozinka za pristup fajlu
Sistemski fleg	0 ako je fajl običan a 1 ako je sistemski
ASCII fleg	0 ako je ASCII fajl a 1 ako je binarni
Fleg direktnog pristupa	0 ako je sekvencijalni a 1 ako je direktan pristup
Fleg za čitanje	0 ako se može i pisati a 1 ako je samo za čitanje

7.1.2. Operacije nad fajlovima

Kroz sistemske pozive, fajl sistem može pružiti razne funkcije. Većina fajl sistema podržava sledeće operacije:

- *Kreiranje* – upisivanje novog fajla;
- *Brisanje* – uklanjanje postojećeg fajla;

- *Čitanje* – čitanje informacije iz fajla;
- *Pisanje* – ažuriranje fajla, bilo menjanjem već postojećih podataka, ili dodavanjem novih;
- *Repozicioniranje* – pomeranje pokazivača na određeni deo fajla kako bi se isti ažurirao;
- *Skraćivanje* – brisanje krajnjeg dela fajla.

Druge operacije se mogu izvesti kombinovanjem ovih osnovnih. Na primer, fajl se može iskopirati kreiranjem novog fajla i kopiranjem sadržaja starog fajla.

Pre korišćenja fajla, proces ga mora „otvoriti". Za ovo se koristi poseban sistemski poziv koji služi da se fajl označi kao otvoren, inicijalizuju potrebne strukture, učitaju atributi i adrese na kojima se čuvaju podaci. Kada više nije potreban, fajl se zatvara korišćenjem sistemskog poziva za tu namenu. Fajl se zatvaranjem briše iz tabele otvorenih i oslobađaju se strukture koje više nisu potrebne.

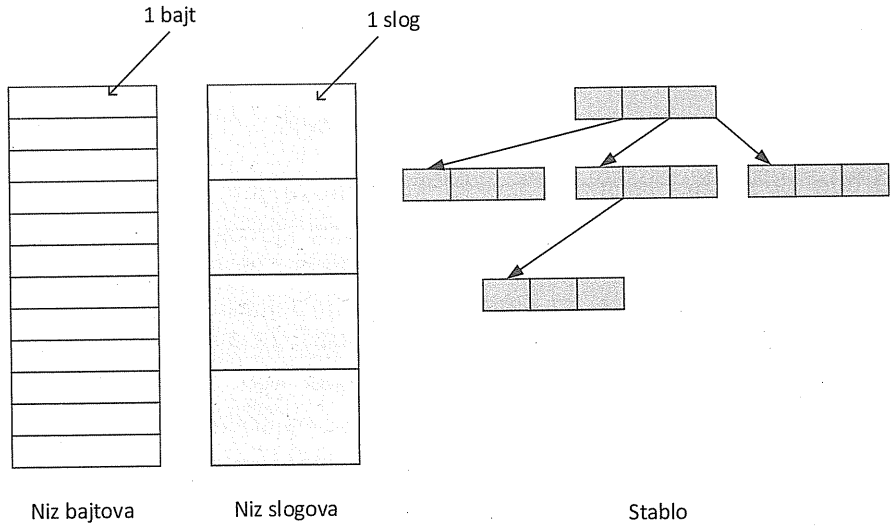
Procesima je potrebno obezbediti mogućnost da čitaju, upisuju i menjaju atribute fajlova. Na primer, da bi program sortirao fajlove po veličini, on mora biti u stanju da pročita taj atribut. S druge strane, program može imati potrebu da neke fajlove označi kao skrivene, pa mu se mora pružiti mogućnost da modifikuje atribute.

7.1.3. Struktura fajlova

Operativni sistemi fajlove mogu struktuirati na više načina (slika 7.2.). Niz bajtova je jedna od čestih organizacija. Sva kontrola nad sadržajem fajla prepuštena je programu koji njime manipuliše. Operativni sistem vidi samo niz bajtova i ne razume unutrašnju strukturu. Ovakav pristup je veoma fleksibilan.

Drugi pristup podrazumeva da se fajlovi podele u slogove fiksirane veličine sa tačno određenom strukturom. Na ovaj način fajl postaje sekvenca (niz) slogova. Čitanje i pisanje se obavlja na nivou sloga.

Treći pristup je da se fajl implementira kao stablo (drvo). Ovo ubrzava pretragu podataka, ali je implementacija komplikovanija.



Slika 7.2. Strukture zapisa fajla

Na modernim sistemima dominantno je strukturiranje fajla kao niza bajtova.

7.1.4. Tipovi fajlova

Pri dizajniranju operativnog sistema, postavlja se pitanje da li će sistem biti napravljen tako da prepoznaje i podržava tipove fajlova. Ako sistem prepoznaje određeni tip, onda će znati kako da rukuje određenim fajlom, tj. na koji način da interpretira sadržane informacije. Na primer, ako bi se fajl koji sadrži video informacije otvorio u tekst editoru, dobio bi se besmislen tekst, jer editor čita informacije iz fajla i interpretira ih kao tekst (npr. čitaju se ASCII karakteri bajt po bajt), a nisu bile namenjene za takvu vrstu upotrebe. Ovo je moguće, jer je fajl niz bajtova. Kako će se oni interpretirati i da li će se interpretirati na predviđeni način zavisi od načina na koji ih korisnik koristi.

Česta tehnika jeste da se pored imena, na kraj fajla doda tačka i za njom kratka *ekstenzija* (nastavak) koja govori o tipu. Ova informacija je korisna

i korisniku i operativnom sistemu, jer se na osnovu nje zna koje operacije mogu da se izvrše nad fajlom i na koji način se podaci čitaju. Operativni sistem može na osnovu ekstenzije da definiše podrazumevani program koji će se koristiti za otvaranje određenog tipa fajla. Na primer, može se definisati da tekst editor otvara sve fajlove sa ekstenzijom .txt, a da program za pregledanje slika otvara fajlove sa ekstenzijama: .jpg i .png. Ekstenzija je samo nagoveštaj koja ne garantuje tip fajla. Takođe, nije obavezno da se uopšte sadrži u imenu fajla, ali olakšava rad ako je prisutna i tačno navedena. U tabeli 7.2. mogu se videti neke od standardnih ekstenzija fajlova.

Tabela 7.2. Standardne ekstenzije fajlova

Ekstenzija	Vrsta fajla
.html , .htm, .xml, .tex	Etiketirani dokumenti
.txt	Tekstualni dokument
.jpg, .png, .tif,	Slika
.c, .cpp, .cs, .java, .asm	Kôd programa
.exe, .jar, .com, .bin	Izvršni fajlovi
.o	Objektni fajlovi
.lib, .a, .so, .dll	Biblioteke za programe
.mpg, .avi, .flv, .mpeg, .mov	Video – Multimedija
.mp3	Muzika
.pdf, .ps,	Dokumenti koji se mogu pregledati nezavisno od sistema
.zip, .rar, .tar	Komprimovane arhive
.doc, .docx, .rtf	Formati jezičkih procesora
.bat, .sh	Paketni (batch) fajlovi

U nekim sistemima vrsta fajla se određuje na osnovu niza bitova koji se nalazi na početku fajla i koji se naziva *magični broj*. To je još jedan od načina na koji se pruža uvid u tip fajla. Kao i kada je u pitanju ekstenzija, operativni sistem ne može zahtevati da fajl sadrži magični broj, jer je on mogao nastati na sistemu koji ih ne podržava i ne koristi. U tabeli 7.3. mogu se videti magični brojevi za neke od standardnih tipova fajlova.

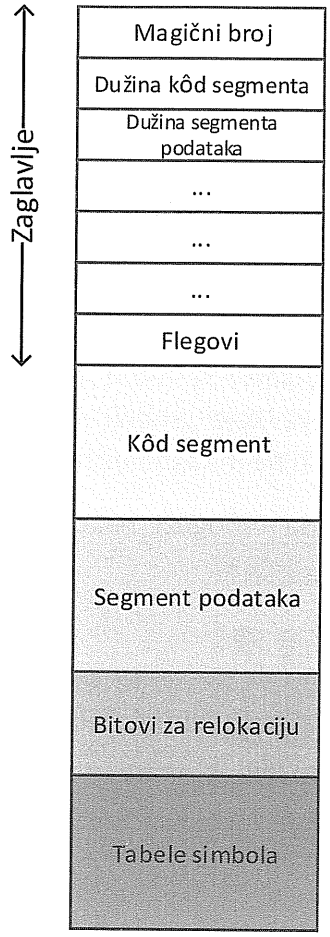
Tabela 7.3. Magični brojevi (Heksadekadni zapis)

Tip fajla	Česta ekstenzija	Magični broj
Izvršni fajl (Microsoft DOS, Windows, OS/2)	.exe	4D 5A
Izvršni fajl (UNIX)		7F 45 4C 46
Arhivirani fajl (ZIP, JAR, ODF, OOXML)	.zip, .jar, .odt, .ods	50 4B 03 04 ILI 50 4B 07 08
Arhivirani fajl (RAR)	.rar	52 61 72 21 1A 07 00
TAR (POSIX)	.tar	75 73 74 61 72
JPEG File Interchange Format	.jpg	FF D8 FF E0
Bit-mapa	.bmp	42 4D
GIF format slike	.gif	53 49 4D 50 4C 45
PNG format	.png	89 50 4E 47
Windows-ov meta fajl	.wmf	D7 CD C6 9A
MPEG-4 video fajl	.mp4	00 00 00 NN 66 74 79 70 33 67 70 35
Video fajl AVI formata	.avi	52 49 46 46
Muzički mp3 fajl	.mp3	49 44 33
Postscript format	.[e]ps	25 21
Xfig format	.fig	23 46 49 47
TIFF format (Motorola – big endian)	.tif	4D 00 2A
TIFF format (Intel – little endian)	.tif	49 49 2A 00
Fotošop fajl	.psd	38 42 50 53
Adobe PDF dokument	.pdf	25 50 44 46
Microsoft WORD dokumet	.doc	D0 CF 11 E0 A1 B1 1A E1

Operativni sistemi obično podržavaju nekoliko tipova fajlova. Skoro na svim sistemima postoje *regularni fajlovi* koji čuvaju informacije korisnika.

Direktorijumi (folderi) su sistemski fajlovi koji odražavaju strukturu fajl sistema. Na nekim sistemima postoje i takozvani *specijalni fajlovi*. To su interfejsi drajvera uređaja koje operativni sistem vidi kao obične fajlove, radi lakšeg rukovanja. Time se postiže da program manipuliše uređajem koristeći iste sistemske pozive kao da se radi o fajlu. Takvi su *karakter specijalni fajlovi*, koji modeluju serijske uređaje kao što su štampač,

terminal, mreža itd. i *blok specijalni fajlovi*, koji pružaju mogućnost čitanja ili upisa bloka određene veličine.



Slika 7.3. Sastavni delovi izvršnog fajla

Regularni fajl može biti ASCII ili binarni. *ASCII fajl* se sastoji od niza znakova (karaktera). Oni se ne moraju posebno interpretirati, jer univerzalnost ASCII formata omogućava da se mogu obrađivati u većini editora i da ih bez modifikacije mogu koristiti različiti programi. Svi ostali se nazivaju binarni. *Binarni fajlovi* se ne mogu interpretirati bajt po bajt kao ASCII i imaju strukturu poznatu programu koji njima rukuje.

Tipičan binarni izvršni fajl sastoji se od zaglavlja, kôd segmenta, segmenta podataka, bitova za relokaciju i tabele simbola koja se koristi pri debagovanju tj. otklanjanju grešaka (slika 7.3.). Magični broj je prvi podatak u zaglavlju i on govori operativnom sistemu da se radi o izvršnom fajlu. Zatim slede podaci o dužini raznih delova fajla, i na kraju određeni flegovi. Nakon zaglavlja slede podaci samog programa. U slučaju izvršnog fajla, operativni sistem poznaje njegovu unutrašnju strukturu, jer je upravo on i odgovoran za njegovo pokretanje.

7.2. Direktorijumi

Direktorijum (*directory*, *folder*) se može pojednostavljeno zamisliti kao tabela sa spiskom fajlova koje „sadrži". Organizacija direktorijuma i fajlova se može realizovati na mnogo načina.

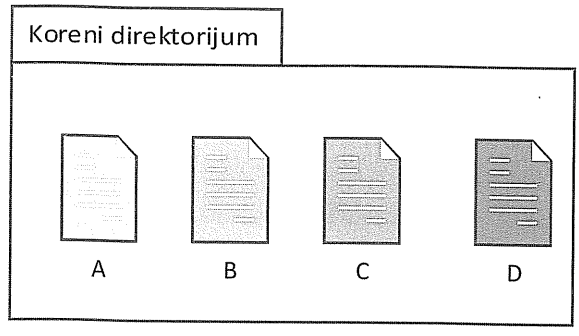
Fajl sistem bi trebalo da omogući nekoliko osnovnih operacija nad direktorijumom:

- Kreiranje;
- Brisanje;
- Izlistavanje svih fajlova u direktorijumu;
- Dodavanje novog fajla;
- Brisanje fajla.

7.2.1. Organizacija direktorijuma – jedan nivo

Najjednostavniji pristup organizacije direktorijuma podrazumeva da postoji jedan koreni (*root*) direktorijum u kojem se nalaze svi fajlovi (slika 7.4). Na prvim računarima sa malo fajlova ovo je bilo prihvatljivo, pogotovo što su ti sistemi bili jednokorisnički.

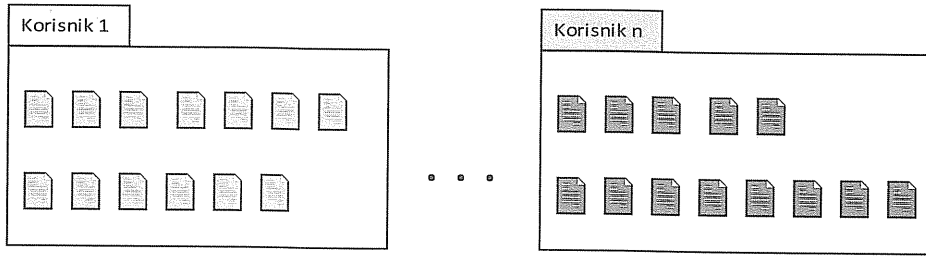
Iako jednostavan za implementaciju, ovakav pristup je vrlo ograničen jer dva fajla ne mogu imati isto ime. Svaki korisnik, ili aplikacija, bi morao pre kreiranja fajla da proveri da li fajl sa tim imenom već postoji i da, ako je to slučaj, odabere drugo ime.



Slika 7.4. Organizacija sa jednim korenim direktorijumom

7.2.2. Organizacija direktorijuma – dva nivoa

Organizacija na dva nivoa predstavlja unapređenje koje podrazumeva da se svakom korisniku dodeli poseban direktorijum. Ovakva struktura ima dva nivoa, prvi direktorijum je koreni, a unutar njega su direktorijumi za svakog korisnika (slika 7.5.).



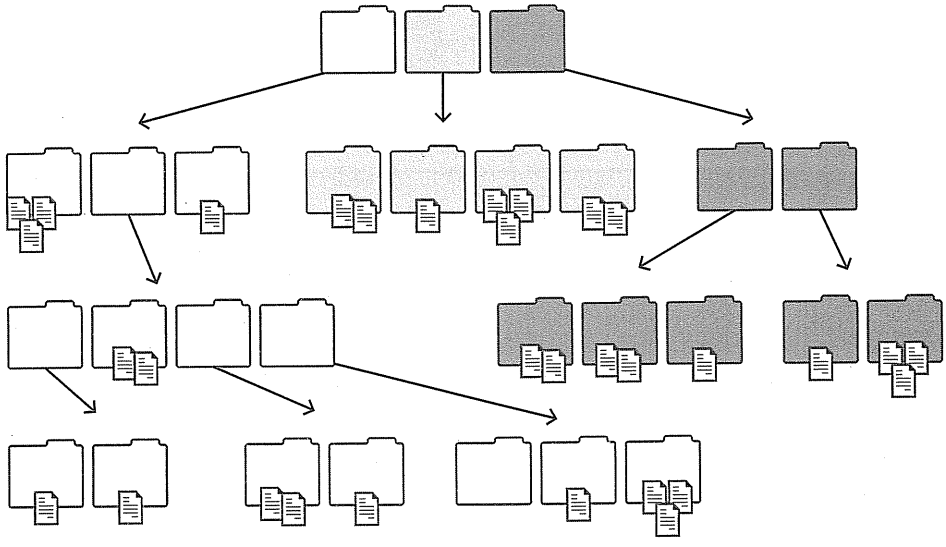
Slika 7.5. Organizacija direktorijuma u dva nivoa

Iako je ovo bolje rešenje od prethodnog rešenja sa jednim nivoom, neki problemi su „nasleđeni". Naime, jedan korisnik može imati veliki broj fajlova, i traženje novog imena za svaki novi je neprihvatljivo. Sa druge strane, svaki korisnik je izolovan ako mu se ne dozvoli da pristupa direktorijumima drugih korisnika.

7.2.3. Organizacija direktorijuma u strukturu stabla

Prirodno unapređenje, generalizacija prethodne ideje, predstavlja pristup organizacije direktorijuma u strukturu stabla (drveta), tako da svaki direktorijum može sadržati fajlove ali i druge direktorijume, bez ograničenja dubine (slika 7.6.). Pri tome, svaki fajl ima jedinstvenu

putanju, od korenog direktorijuma do samog fajla i mora imati jedinstveno ime samo u poslednjem direktorijumu kojem pripada. Većina modernih fajl sistema je organizovana na ovakav način.



Slika 7.6. Organizacija direktorijuma strukturom stabla

Važno je primetiti da je direktorijum u stvari fajl koji ima strukturu definisanu fajl sistemom. Na primer, svaki fajl može imati jedan bit na početku koji govori operativnom sistemu da li je to direktorijum ili obični fajl. U slučaju da je u pitanju direktorijum, operativni sistem će biti u mogućnosti da pročita njegov sadržaj, tj. listu fajlova koje sadrži.

7.2.4. Apsolutne i relativne putanje

U mnogim operativnim sistemima procesi imaju *radni* (trenutni) *direktorijum* u kojem se izvršavaju. Tu se, uglavnom, nalaze fajlovi potrebni za izvršavanje procesa. Radni direktorijum se može promeniti i za to je obično obezbeđen sistemski poziv koji kao parametar prima putanju do novog. Ako je potrebno pristupiti fajlu koji se ne nalazi u radnom direktorijumu, jedna opcija bila bi da se radni promeni u onaj gde se traženi fajl nalazi. Druga mogućnost je da se navede *apsolutna putanja* do fajla, tj. putanja koja počinje korenim direktorijumom sistema i završava se imenom fajla, navodeći sve poddirektorijume koji čine

putanju. Da bi korisnik „došao“ do željenog fajla, on mora otvoriti redom sve direktorijume od korenog do krajnjeg u kom se nalazi fajl. Apsolutna putanja je, dakle, niz imena direktorijuma koji se moraju proći da bi se došlo do određenog fajla, počevši od korenog direktorijuma, i koji se završava imenom traženog fajla.

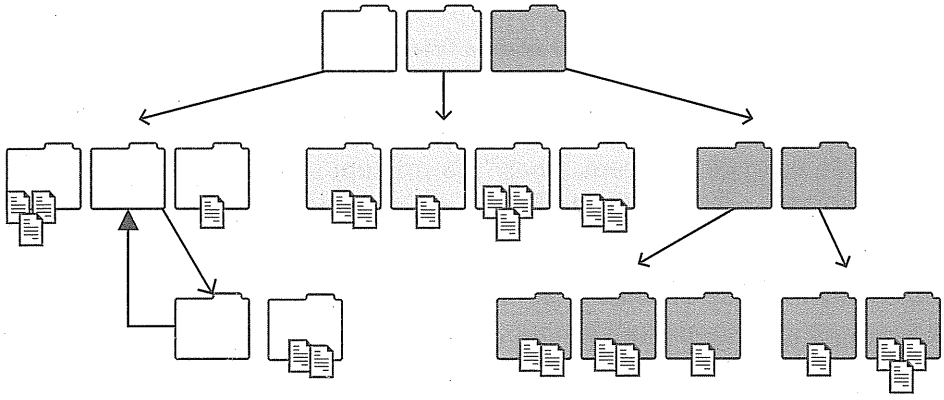
Na primer, neka je dat program koji od slike u boji pravi crno-belu i neka on ulazne slike očekuje u direktorijumu „ulaz“, a izlazne snima u „izlaz“ koji se nalaze u radnom direktorijumu. U ovom slučaju radni direktorijum je upravo direktorijum u kojem je pomenuti program pokrenut. Postavlja se pitanje kako će program „znati“ da pristupi ulaznim slikama kada ne zna apsolutnu putanju do direktorijuma „ulaz“. Direktorijum „ulaz“ je u njegovom radnom direktorijumu, tako da je rešenje problema jednostavno jer se apsolutna putanja može dobiti nastavljanjem putanja do radnog direktorijuma i imena direktorijuma „ulaz“. Da ovu operaciju nastavljanja imena putanja korisnik ne bi morao uvek da ponavlja, uvedene su *relativne putanje*. Relativna putanja je deo apsolutne putanje koja počinje radnim direktorijumom. Ako se ona navede, operativni sistem sam uradi nastavljanje do apsolutne kako bi našao tražene fajlove. Relativne putanje ovde bi bile samo „ulaz“ i „izlaz“.

U prethodnom primeru, dovoljno je bilo navesti da se ulazne slike nalaze u direktorijumu „ulaz“ i operativni sistem bi znao da je reč o relativnoj putanji. Neka prethodni program sada očekuje slike u direktorijumu „ulaz“ koji se nalazi u roditeljskom direktorijumu u odnosu na njegov radni. Postavlja se pitanje kako da se pronađe putanja do roditeljskog direktorijuma. Korisnik bi mogao iz putanje do radnog direktorijuma da izvede putanju do roditeljskog, ali da bi se taj proces olakšao, uvedena su dva specijalna direktorijuma koja se implicitno nalaze u svakom. Njihova imena su „.“ i „..“ (tačka i tačka-tačka). Tačka je referenca za sam direktorijum, dok je tačka-tačka referenca za roditeljski. Tražena relativna putanja u ovom primeru bi tada bila „../ulaz“. Trebalo bi primetiti da bi korektne putanje bile:

- „./../ulaz“ ;

- „././././././././././././.././././././././ulaz/././././././“.

Ovo možda izgleda besmisleno, ali treba obratiti pažnju, kako bi se izbegle beskonačne petlje kada se konstruišu programi koji pretražuju fajl sistem. Uvođenjem linkova, struktura direktorijuma postaje graf koji može imati cikluse (slika 7.7.).



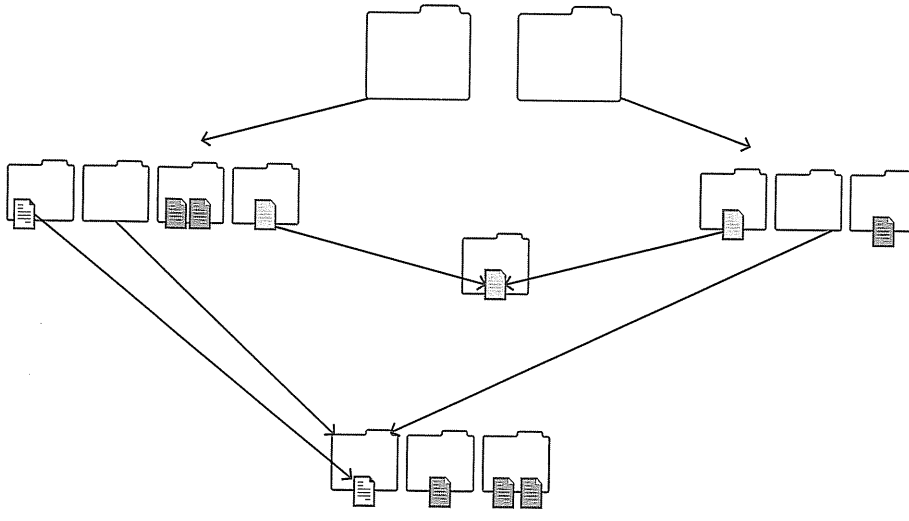
Slika 7.7. Struktura direktorijuma

7.2.5. Deljenje fajlova, aciklični graf

U situacijama kada više korisnika treba da deli fajlove, npr. ako rade na istom projektu, zgodno je da fajlovi budu u direktorijumima svakog od korisnika. Očigledno rešenje je da se fajlovi kopiraju kod svakog od korisnika, ali cilj je da se, ako neko izmeni fajl, te promene vide i kod drugih korisnika, što se jednostavnim kopiranjem ne bi postiglo.

Problem se može rešiti uvođenjem *linkova* (veza). Link je niska unutar fajla koja sadrži putanju do nekog drugog fajla. Rešenje za deljenje fajlova korišćenjem linkova podrazumeva da samo jedan korisnik poseduje originalni fajl, a svi drugi veze ka njemu. Tako svi korisnici, osim jednog, sadrže samo reference na fajl i on fizički postoji samo u jednom direktorijumu. Kada korisnik otvara vezu, operativni sistem prepozna da fajl sadrži link i ne otvara njegov sadržaj, već prati putanju do originalnog fajla. Za ovakvu uređenost fajl sistema kažemo da je realizovana preko *acikličnog grafa* (slika 7.8.).

Brisanje linkova ne utiče na originalni fajl. Brisanje fajla, međutim, čini da sve veze postanu nevalidne, jer pokazuju na fajl koji više ne postoji.



Slika 7.8. Aciklični graf

Uvođenjem linkova, isti fajl može imati više apsolutnih putanja (putanja do „stvarnog" fajla i putanje do linkova), što u nekim situacijama može predstavljati problem. Međutim, obično operativni sistemi pružaju mogućnost da se onemogući praćenje veza, pa se na taj način one mogu zanemariti.

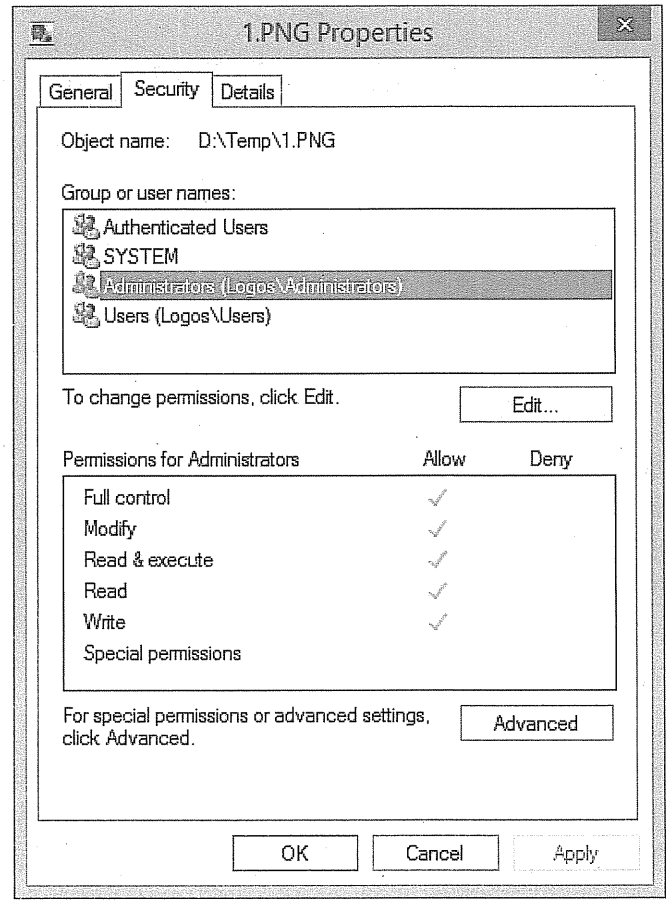
7.3. Prava pristupa

Fajlove treba zaštititi od neovlašćenog pristupa. *Prava pristupa* (*access permissions*) predstavljaju prava koja korisnik može da ostvari nad fajlom. Zaštita podataka na većini sistema zasniva se na kontrolisanom pristupu podacima koji se ostvaruje kroz prava pristupa. Kada su u pitanju jednokorisnički sistemi, mehanizam zaštite se može realizovati veoma jednostavno, jer se ceo sistem može zaključati šifrom čime je sigurnost obezbeđena. Kod višekorisničkih sistema, stvar je malo komplikovanija, jer najčešće nije cilj izolovati korisnike time što bi svako imao pristup

samo svojim fajlovima, već treba omogućiti fleksibilniji mehanizam koji bi određenim korisnicima odobravao pristup, a drugima ne.

Nad fajlovima i direktorijumima najčešće se definišu sledeća tri tipa pristupnih dozvola:

- Dozvola za čitanje;
- Dozvola za pisanje;
- Dozvola za izvršavanje.



Slika 7.9. Prava pristupa – Operativni sistem Windows

Važno je napomenuti da ove dozvole često nemaju isto značenje kada se primenjuju na fajlove i direktorijume. Naime, *dozvola za čitanje* u slučaju

fajla podrazumeva da korisnik može da pristupa podacima fajla i da ih čita dok kod direktorijuma ova dozvola omogućava izlistavanje stavki koje on sadrži. Takođe, *dozvola za pisanje* kada su fajlovi u pitanju dozvoljava dodavanje novih podataka, a kod direktorijuma se odnosi na dodavanje i brisanje fajlova i poddirektorijuma koji se u njemu nalaze. *Dozvola za izvršavanje* omogućava korisniku da izvrši fajl (ako taj fajl predstavlja program), dok je kod direktorijuma neophodna za druge dozvole odnosno, u zavisnosti od sistema, bez nje se ne mogu primeniti neke dozvole.

Problem pristupa se može rešiti tako što će se za svaki fajl održavati lista svih korisnika sa njihovim dozvolama. Time se postiže najveća fleksibilnost, jer se za svakog ponaosob mogu definisati prava. Međutim, korisnika može biti puno, i održavati njihovu listu za svaki fajl nije pogodno.

```
lrwxrwxrwx  1 root   root        16 феб  3 15:34 lampp -> /opt/lampp/xampp
drwxr-xr-x 14 root   root     12288 феб  4 23:00 lib
-rwx------  1 root   root   3361003 јун 10  2013 manager-linux-x64.run
drwxr-xr-x  3 root   root      4096 феб  3 15:34 mysql
drwxr-xr-x  2 root   root      4096 феб  3 15:34 pear
drwxr-xr-x  3 root   root      4096 феб  3 15:34 php
drwxr-xr-x  9 root   root      4096 феб  4 23:00 phpmyadmin
drwxr-xr-x  3 root   root      4096 феб  3 15:34 proftpd
-rw-r--r--  1 root   root       905 феб  3 15:37 properties.ini
-rw-r--r--  1 root   root     73365 нов 19 14:14 RELEASENOTES
drwxr-xr-x  2 root   root      4096 феб  3 15:34 sbin
drwxr-xr-x 45 root   root      4096 феб  3 15:34 share
drwxrwxrwx  2 daemon daemon    4096 феб  4 22:59 temp
-rwx------  1 root   root   7968007 феб  3 15:35 uninstall
-rw-------  1 root   root    257971 феб  3 15:35 uninstall.dat
drwxr-xr-x  6 root   root      4096 мај  2 13:50 var
-rwxr-xr-x  1 root   root     15201 јул 22  2013 xampp
```

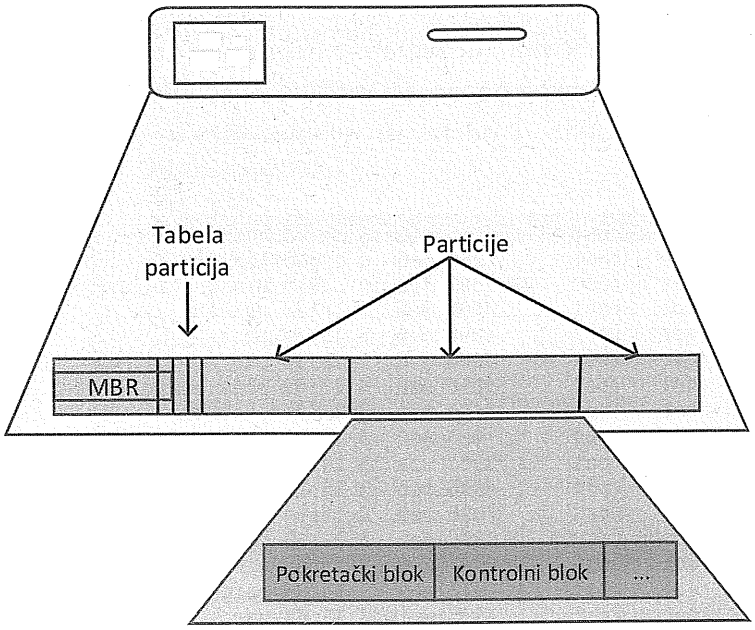
Slika 7.10. Prava pristupa – Operativni sistem Linux

Drugi pristup podrazumeva da se svi korisnici podele u grupe na osnovu zadataka koje izvršavaju. Iz ugla fajla, korisnici se mogu svrstati u tri grupe: vlasnik, članovi grupe i ostali. *Vlasnik* je korisnik koji je kreirao fajl i koji ima prava da drugima određuje prava pristupa. *Članovi grupe* su korisnici koji pripadaju grupi u kojoj je vlasnik fajla, a *ostali* su svi ostali korisnici. Grupe omogućavaju definisanje posebnih pristupnih dozvola za različite korisnike. Uobičajeno je da vlasnik fajla ima sva prava, da grupe

imaju restriktivnija prava, a ostali još manje privilegija kada su operacije nad fajlovima u pitanju. Ovako je postignut kompromis, gde je kompleksnost implementacije daleko manja, a fleksibilnost je u velikoj meri zadržana.

Na slikama 7.9. i 7.10. prikazana su prava pristupa kod operativnih sistema Windows i Linux.

7.4. STRUKTURA I IMPLEMENTACIJA FAJL SISTEMA



Slika 7.11. Struktura fajl sistema

Fajl sistemi su smešteni na neki vid sekundarne memorije, koja se obično naziva disk, bilo da je reč o magnetnom disku ili nekom drugom uređaju. Većina diskova se može podeliti na *particije*, celine koje su logički predstavljene kao posebni diskovi i koje omogućavaju da se na svakoj koristi različit fajl sistem. Prvi sektor diska se naziva *MBR* (*Master Boot Record*) i podaci koji su tu zapisani (MBR program) su prvi koji se učitavaju i izvršavaju pri pokretanju računara. MBR sadrži i tabelu particija u kojoj

je zapisan početak i kraj svake od njih. Jedna particija je u tabeli označena kao aktivna, i prvo što MBR program izvršava jeste nalaženje aktivne particije i učitavanje njenog prvog bloka, koji se naziva *pokretački* (*Boot*) *blok*. Nakon toga, izvršava se pokretački blok aktivne particije što za rezultat ima učitavanje i pokretanje operativnog sistema. Posle pokretačkog bloka, sledi kontrolni blok koji sadrži informacije vezane za particiju kao što su ukupan broj blokova, veličina blokova, broj slobodnih blokova itd. (slika 7.11.). Osim MBR, koristi se i GPT (GUID Partition Table) pristup za podelu hard diskova na particije, koji donosi mnoge prednosti i koji ubrzano zamenjuje MBR. Međutim, MBR pristup je najkompatibilniji i još uvek je neophodan u nekim slučajevima.

7.4.1. Implementacija fajlova

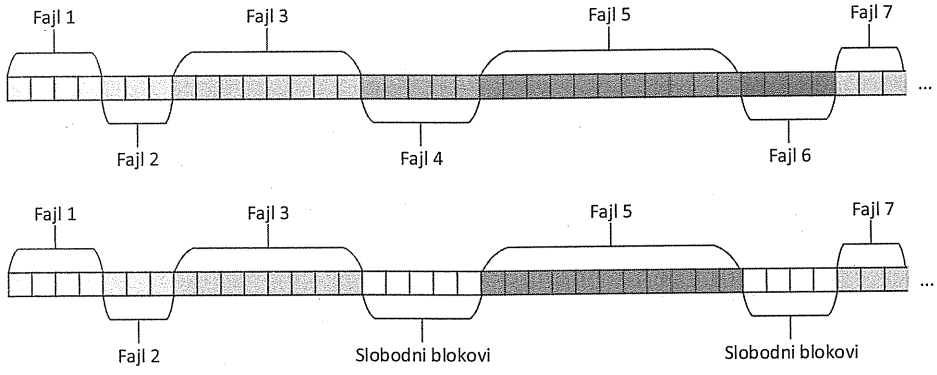
Svaka memorija ima definisanu najmanju jedinicu koja može biti alocirana i adresirana. Takve jedinice zovu se *blokovi*. Na primer, magnetni diskovi obično koriste blokove veličine 512 bajta ili 4 kilobajta. Svaki fajl koji se kreira predstavljen je kao skup blokova koji su mu dodeljeni. Ključni problem, pri implementaciji fajlova, predstavlja način na koji se alociraju slobodni blokovi i kako se čuva informacija koji blokovi pripadaju kom fajlu.

7.4.1.1. Neprekidna alokacija

Neprekidna alokacija je najjednostavniji pristup pri kojem se svaki fajl čuva kao neprekidni niz blokova. Na primer, ako je fajl veličine n blokova i počinje na lokaciji b, onda on zauzima blokove $b, b+1, b+2, ..., b+n-1$. Sledeći fajl se smešta odmah iza poslednjeg bloka prethodnog fajla. Ovo ubrzava sekvencijalno čitanje sa uređaja kao što su magnetni diskovi, jer zahteva minimalno pomeranje glave za čitanje. Štaviše, direktan pristup delu fajla je takođe olakšan, jer ako fajl počinje blokom b, pristup i-tom bloku fajla je pristup bloku $b+i$ u sistemu. Direktorijum za svaki fajl čuva samo početni blok i broj blokova koje fajl zauzima.

Međutim, ovakva organizacija ima ozbiljne probleme pri brisanju fajlova, jer dolazi do fragmentacije. Kako je memorijski prostor alociran neprekidno, ako se fajl obriše, na njegovom mestu ostaje „rupa“ tj.

slobodni prostor (slika 7.12.). Te rupe se popunjavaju novim fajlovima, ali se može desiti da sistem ima dovoljno slobodnog prostora a da ne postoji nijedan neprekidni niz slobodnih blokova dovoljan za smeštanje novog fajla. Analogno, kao i kada je u pitanju radna memorija, ovaj problem se naziva eksterna fragmentacija.



Slika 7.12. Neprekidna alokacija

Drugi problem predstavlja interna fragmentacija. Naime, pošto fajl mora zauzeti ceo broj blokova, poslednji blok je često neiskorišćen do kraja. Na primer, ako su blokovi veličine 512 bajta, a smešta se fajl veličine 1100 bajtova, on će zauzeti 3 bloka, i poslednji će sadržati samo 1100 – 2*512 = 76 bajtova korisnih informacija, a ostalih 436 će ostati neiskorišćeno. Veći problem leži u činjenici da nije moguće pretpostaviti koliko će fajl da raste u budućnosti a samim tim i alocirati dovoljno slobodnog prostora na kraju fajla. Čak i da je ta informacija poznata, prealokacija je neefikasna jer se alocira prostor koji dugo može ostati neiskorišćen čime se problem interne fragmentacije samo uvećava.

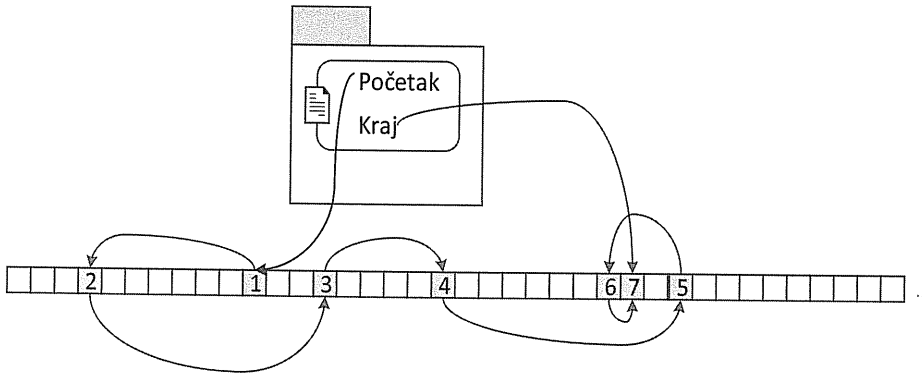
Ovaj pristup je lako implementirati i čitanje je veoma efikasno, ali fragmentacija predstavlja ozbiljan problem koji, uglavnom, onemogućava njegovu praktičnu upotrebu.

7.4.1.2. Povezane liste

Problemi neprekidne alokacije se rešavaju preko povezanih listi. Svaki fajl je predstavljen kao povezana lista blokova. Blok, osim informacije, sadrži

i pokazivač na sledeći blok, tj. polje sa njegovom adresom (slika 7.13.). Direktorijum za svaki fajl sadrži samo adresu prvog bloka, a poslednji blok fajla, kao pokazivač sadrži neku nevalidnu vrednost (*null*), koja označava da je on poslednji.

Novi fajl se kreira tako što se u direktorijum doda novi pokazivač na prvi blok tog fajla. On na početku može sadržati nevalidnu vrednost, kako bi se označilo da je fajl prazan. Upis u fajl bi se izvršio tako što bi sistem našao neki slobodni blok i njegovu adresu upisao u pokazivač direktorijuma. Sledeći bi bio nadovezan na kraj liste itd. Prednost je to što se novi blok može nadovezati bilo gde da je fizički smešten. Tako blokovi jednog fajla mogu biti razbacani svuda po disku. Fajl se čita blok po blok, uvek čitajući adresu narednog bloka iz pokazivača trenutnog, prolazeći kroz listu. Ovako se u potpunosti rešava problem eksterne fragmentacije. Takođe, fajl može rasti sve dok ima slobodnih blokova, tako da veličina fajla ne mora biti deklarisana na početku.



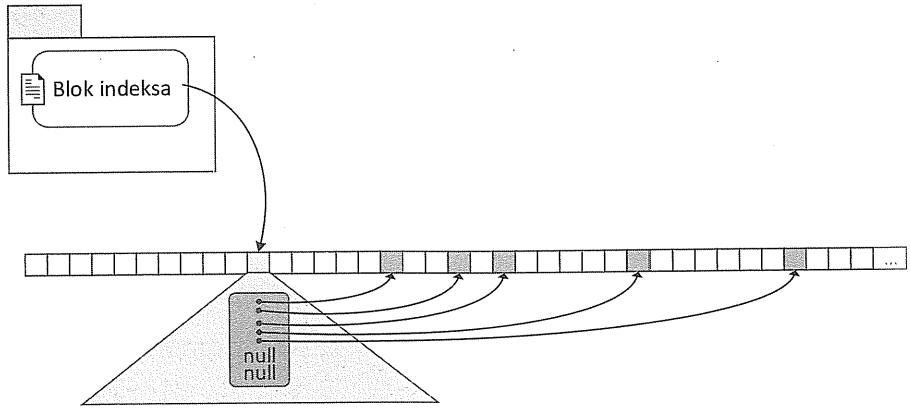
Slika 7.13. Povezane liste

Međutim, postoje i mane kod ovog pristupa. Povezana lista nije efikasna kada je potreban direktan pristup delu fajla. Da bi se došlo do i-tog bloka, mora se krenuti od početnog i pratiti svaki pokazivač, jedan po jedan, kroz svaki blok, sve dok se ne dostigne i-ti. Magnetni diskovi bi pri ovakvoj organizaciji bili neefikasni i zbog činjenice da je, u opštem slučaju,

potrebno pomerati glavu za svaki blok. Novije vrste sekundarne memorije, kao što je SSD disk, nemaju ovakav problem, jer je vreme pristupa bilo kojem delu memorije u bilo kom trenutku isto. Drugi problem je što se, osim same informacije, u svakom bloku mora smestiti i pokazivač, što dodatno zauzima prostor. Ovi problemi se, donekle, mogu rešiti grupisanjem blokova (*clustering*) i alociranjem grupa, umesto pojedinih blokova. Na primer, fajl sistem može definisati grupu kao četiri bloka i odrediti nju kao najmanju jedinicu prostora koja se alocira. Svaka grupa bi se mogla shvatiti kao neprekidna alokacija četiri bloka. Ovo je kompromis kojim se povećava interna fragmentacija.

Pouzdanost je veliki problem kod povezanih lista. Može se zamisliti da greška pri upisu, prestanak napajanja, hardverska greška itd. može prouzrokovati da jedan pokazivač sadrži pogrešnu vrednost. Ovakva greška bi uzrokovala da se fajl nadovezuje na slobodan prostor, ili na neki drugi fajl, narušavajući konzistentnost celog fajl sistema.

7.4.1.3. Indeksirana alokacija



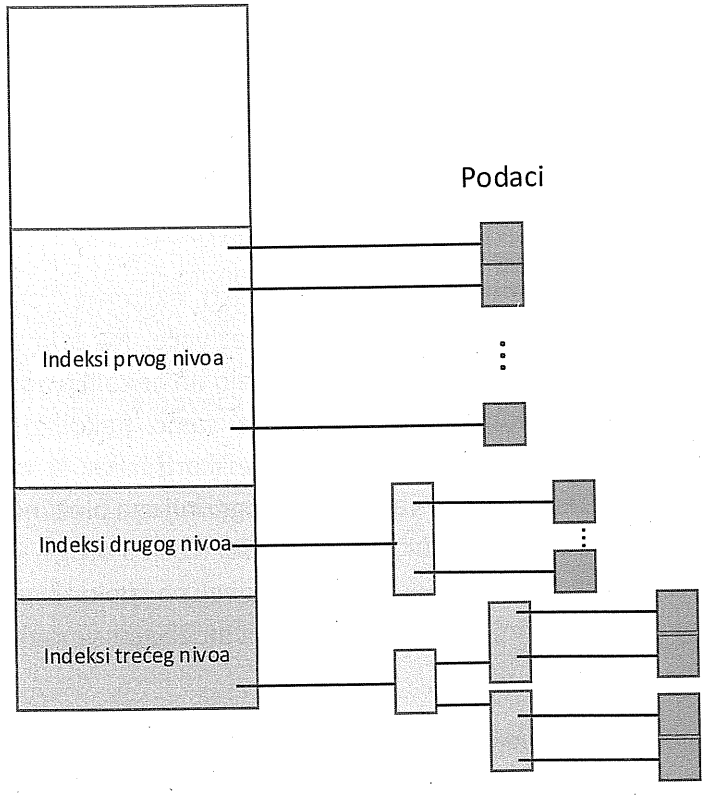
Slika 7.14. Indeksirana alokacija

Iako rešavaju važne probleme, povezane liste uvode novi problem jer ne pružaju efikasan direktan pristup, pošto su pokazivači sadržani u samim blokovima. Ovaj problem se rešava uz pomoć *indeksirane alokacije*, koja sve pokazivače smešta na jedno mesto, koje se naziva blok indeksa. Svaki fajl sadrži blok indeksa, koji sadrži niz adresa blokova koji čine dati fajl

(slika 7.14.). Direktorijum za svaki fajl čuva adresu bloka indeksa. Na ovaj način je obezbeđen efikasan direktan pristup. Naime, da bi se pristupilo i-tom bloku, potrebno je pročitati i-tu adresu u nizu adresa bloka indeksa. Kada se fajl kreira, sve adrese se inicijalizuju na nevalidnu (*null*) vrednost. Kako fajl raste, sistem dodaje blok po blok i njihove adrese se upisuju redom u blok indeksa.

Kod ovakvog pristupa, kao i kod povezanih listi, eksterne fragmentacije nema, ali se zato više prostora troši na pokazivače. Na primer, ako fajl alocira samo nekoliko blokova, da bi se sačuvalo tih nekoliko adresa, mora se alocirati ceo blok indeksa. U njemu će samo nekoliko polja biti popunjeno adresama, a ostala će ostati neiskorišćena. Iz tog razloga veoma je važno odrediti koja je veličina najpogodnija za blok indeksa. Ako se izabere preveliki blok, prostor se neefikasno troši. S druge strane, ako je premali, neće biti dovoljno mesta da se sačuvaju svi pokazivači u slučaju velikih fajlova. Za rešavanje ovakvih problema može se primeniti nekoliko mehanizama:

- *Povezana lista blokova indeksa* – da bi se dozvolilo da fajlovi imaju više blokova nego što postoji mesta za adrese u bloku indeksa, sami blokovi indeksa se nižu u povezane liste, tako što poslednja adresa predstavlja pokazivač na sledeći blok indeksa. Pri tome, mali fajlovi i dalje imaju samo jedan blok indeksa.
- *Indeksi sa više nivoa* – indeks sa dva nivoa bi bio indeks koji ne pokazuje direktno na blokove fajla, već sadrži adrese blokova indeksa koji sadrže adrese blokova fajla. Da bi se pristupilo bloku, sistem u prvom nivou nalazi adresu bloka indeksa i tek u njemu nalazi adresu željenog bloka. Ovaj pristup se može proširiti u indekse višeg nivoa.
- *Hibridni pristup* – koristi se povezana lista blokova indeksa. Određen broj blokova indeksa na početku su direktno indeksirani, što znači da adrese koje oni sadrže pokazuju na blokove fajla. Dalje u listi, određen broj blokova indeksa su indeksi sa dva nivoa, čije adrese pokazuju na blokove indeksa. Sledeći u listi su blokovi indeksa trećeg nivoa itd. (slika 7.15.).



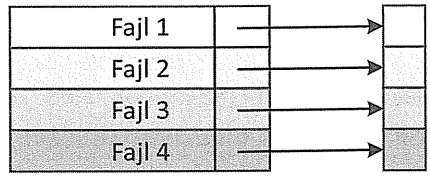
Slika 7.15. Hibridni pristup

7.4.2. Implementacija direktorijuma

Da bi operativni sistem mogao da otvori fajl koji se nalazi na određenoj putanji koju je zadao korisnik, u direktorijumu se mora naći unos koji opisuje fajl i sadrži informacije o blokovima koji mu pripadaju. Štaviše, operativni sistem mora u trenutnom direktorijumu naći unos za svaki naredni iz putanje. Na primer, neka je data putanja „/user1/doc/text". Operativni sistem prvo traži unos za direktorijum „user1" u korenom direktorijumu. Kada ga nađe, otvara ga i u njemu traži informacije o direktorijumu „doc". Najzad, u direktorijumu „doc" se pronalaze informacije o fajlu „text".

7.4.2.1. Lista

Najjednostavnije je direktorijum predstaviti kao listu imena fajlova i pokazivača ka strukturama koje sadrže atribute i informacije o alociranim blokovima (slika 7.16.).



Slika 7.16. Struktura direktorijuma zadata listom fajlova

Pri kreiranju novog fajla, direktorijum, tj. lista imena se pretražuje kako bi se utvrdilo da ne postoji fajl sa istim imenom. Nakon toga se unos za novi fajl dodaje na kraj liste. Pri brisanju se prvo pronalazi fajl sa datim imenom i onda se oslobađaju alocirani blokovi. Posle toga će unos u listi fajlova biti nevažeći pa je potrebno obeležiti ga kao takvog. Za to se može koristiti jedan bit u strukturi koja čuva atribute (npr. 1 predstavlja da fajl postoji, 0 da je obrisan). Bolja ideja je da se koriste povezane liste za svaki unos u direktorijumu. Na taj način se omogućava jednostavnije i efikasnije brisanje unosa.

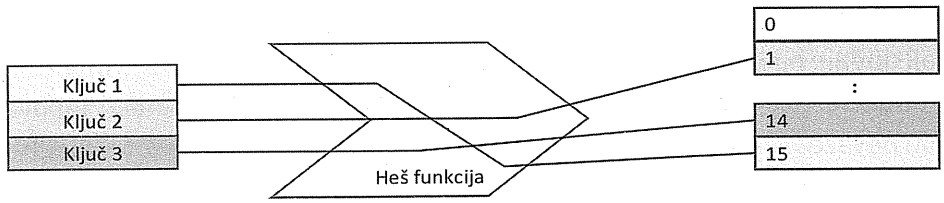
Mana ovog pristupa je to što pronalaženje fajla nije efikasno. S obzirom na to da je pronalaženje fajla sa zadatim imenom osnovna i najčešća operacija, ovo predstavlja ozbiljan problem. Svaki put kada se fajl kreira, briše ili otvara, direktorijum se mora pretražiti. Ovaj problem se može rešiti održavanjem sortirane liste prema imenima fajlova. Tada se za pretraživanje koristi binarna pretraga, što je daleko efikasnije. Međutim, to komplikuje dodavanje novih elemenata. Druga mogućnost je korišćenje balansiranog stabla pretrage.

7.4.2.2. Heš tabela

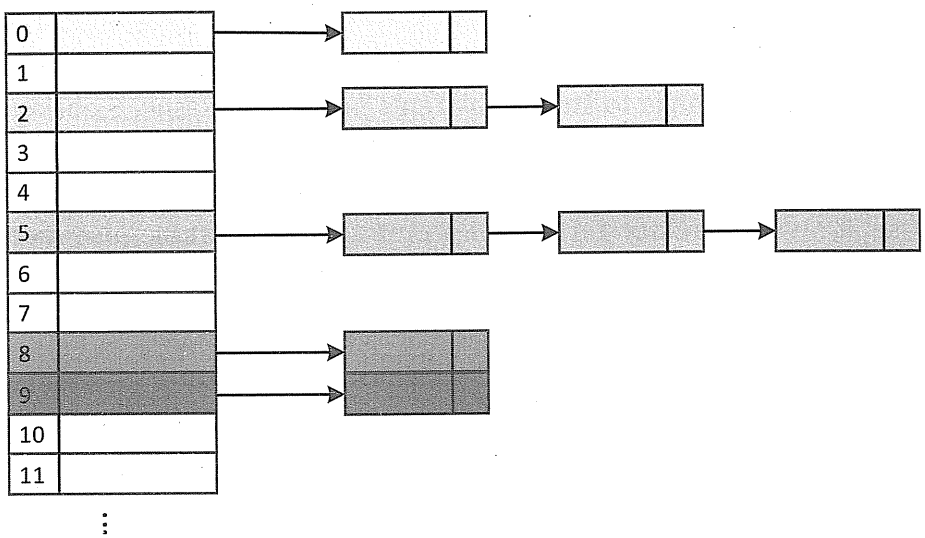
Za direktorijum se može koristiti i *heš tabela* (*Hash table*). Ona funkcioniše na sledeći način. Tabela sadrži imena fajlova koja su poređana u indeksirani niz. Kada je zadato ime koje treba pronaći, koje se naziva ključ, ono se dostavlja heš funkciji koja računa redni broj iz tabele, tj. indeks. Na

osnovu indeksa, ime fajla se nalazi direktno, bez pretraživanja, što je vrlo efikasno (slika 7.17.).

Najjednostavnija heš funkcija se može implementirati kao ostatak pri deljenju nekim brojem. Neka je taj broj, na primer, 256. Od imena fajla bi se mogao dobiti broj tako što bi se sabrale sve ASCII vrednosti koje predstavljaju karaktere imena. Heš funkcija bi taj broj podelila sa 256 i ostatak bi vratila kao indeks. Problem sa heš tabelama predstavlja situacija gde dva različita imena daju iste ključeve, tj. ostatke. Ovo se naziva kolizija. Drugi problem je fiksna dužina tabele. U prethodnom primeru, u tabeli bi imalo mesta samo za 256 imena fajlova, jer toliko ostataka postoji pri deljenju sa 256.



Slika 7.17. Heš funkcija



Slika 7.18. Heš funkcija – ulančavanje

Oba problema se mogu rešiti ulančavanjem. Neka svaki unos u heš tabeli predstavlja povezanu listu (slika 7.18.). Tada se novi unos nadovezuje na stare sa istim indeksom. U najgorem slučaju, svaki ključ bi davao isti indeks, pa bi svi bili nadovezani u jednu listu, koja bi morala da se pretražuje element po element. Međutim, heš funkcije u praksi su napravljene tako da je za dva različita ključa verovatnoća kolizije *1/veličina_tabele*. To znači da je verovatnoća da se za neki indeks stvori duga lista, dok su za druge unosi prazni, jako mala.

7.4.3. Upravljanje slobodnim prostorom

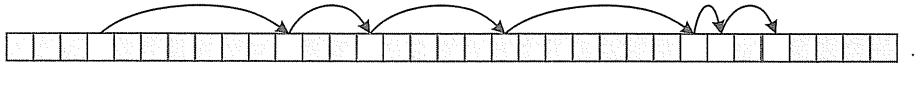
Kada se fajl obriše, potrebno je prostor osloboditi i upotrebiti ga za upis novih podataka. Da bi to bilo moguće, potrebno je da operativni sistem vodi evidenciju o slobodnom prostoru, tačnije da ima „spisak" slobodnih blokova. Kada se kreira fajl, iz spiska se uzima potreban broj blokova koji se alociraju, a njihove adrese se brišu sa spiska. Kada se fajl briše, adrese blokova koje je zauzimao se dopisuju na spisak.

Jedan način je da se spisak implementira kao bit-mapa. Svaki blok se predstavlja nulom ili jedinicom, u zavisnosti od toga da li je zauzet ili slobodan. Na primer, ako su blokovi 3, 5, 7, 11, 12, 13, 14, 18, 20, 21 i 22 slobodni, bit-mapa bi imala sledeći izgled:

0010101000111100010111000000000 ...

Ovo je vrlo jednostavno i efikasno rešenje. Međutim, ako je disk veliki, problem je u tome što onda i bit-mapa takođe mora biti velika.

Drugo rešenje se zasniva na ideji da se koristi povezana lista u kojoj bi se nizali svi slobodni blokovi (slika 7.19.). Ovde direktan pristup nije potreban, jer je sistemu potreban bilo koji slobodan blok da bi ga dodelio fajlu, i prvi u listi može poslužiti.



Slika 7.19. Povezana lista slobodnih blokova

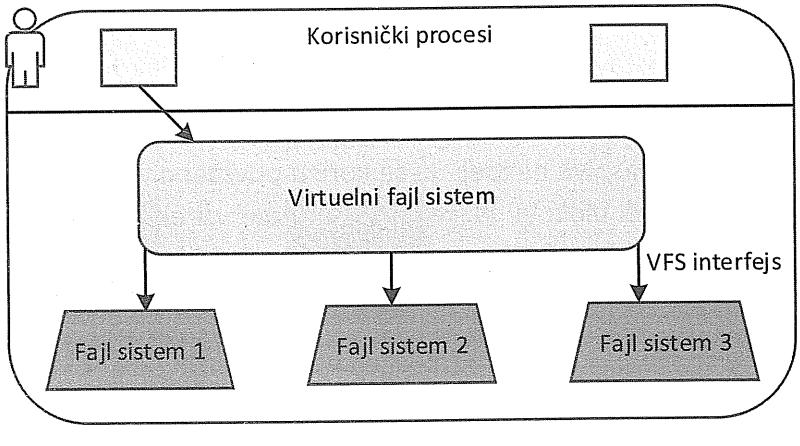
Rešenje se dalje može unaprediti zahvaljujući činjenici da se, uglavnom, više susednih blokova alocira ili oslobađa odjednom. Imajući to na umu, u svakom čvoru liste može se čuvati adresa prvog slobodnog bloka, ali i broj koji označava koliko ima slobodnih blokova do njega.

7.4.4. Performanse

Ma koji algoritam da se koristi, performanse sistema se mogu poboljšati korišćenjem keš memorije. Većina disk kontrolera sadrži memoriju određenog kapaciteta koja se koristi u tu svrhu. Neki sistemi koriste deo glavne memorije kao keš, gde drže skoro korišćene blokove za slučaj da oni ponovo budu potrebni.

7.4.5. Virtuelni fajl sistem

Raznolikost fajl sistema je velika i interfejsi prema korisnicima nisu uvek isti. Na primer, funkcija za čitanje fajlova se različito zove, prima drugačiji skup argumenata, i sl. Da bi se obezbedilo lakše rukovanje, potrebno je uvesti mehanizam koji omogućava da se fajlovi koriste na uniforman način, bez obzira na fajl sistem na kojem se nalaze. Ovo se rešava uz pomoć koncepta *virtuelnog fajl sistema* (*VFS*), koji apstrahuje karakteristike specifičnih fajl sistema, a korisniku pruža uniformni pogled na fajlove (slika 7.20.).



Slika 7.20. Virtuelni fajl sistem

Virtuelni fajl sistem se implementira tako da sa jedne strane pruža interfejs koji sadrži uobičajene sistemske pozive koji se koriste pri radu sa fajlovima, poput onih za otvaranje, pisanje, čitanje itd. Ovim se korisniku omogućava da sa fajlovima rukuje na standardni način. Sa druge strane, VFS za svaki fajl sistem koristi funkcije specifične za isti kako bi izvršio željene operacije.

7.4.6. Sigurnost

Pad sistema može dovesti do nekonzistentnosti podataka. Na primer, kreiranje novog fajla je operacija koja zahteva više promena nad fajl sistemom, kao što su alociranje novih blokova, ažuriranje strukture koja čuva informacije o njima, menjanje strukture direktorijuma i sl. Ove promene mogu biti prekinute padom sistema, što za posledicu obično ima nekonzistentnost podataka. Pored pada sistema, iste probleme može izazvati i greška u implementaciji, kontroleru, pa čak i aplikaciji.

Da bi otkrio grešku, fajl sistem mora skenirati i proveriti sve metapodatke (podaci koje fajl sistem interno koristi a koji opisuju druge podatke npr. vreme modifikacije fajla, veličina itd.). Neke greške je moguće popraviti. Na primer, ako sistem koristi povezanu listu blokova za alokaciju fajla, onda se ceo fajl može povratiti čitanjem blokova, jer je u svakom upisan pokazivač na sledeći. Tako se oštećena struktura direktorijuma može vratiti. Međutim, gubitak direktorijuma pri indeksiranoj alokaciji može biti nepovratan, jer blokovi nemaju nikakve informacije jedni o drugima.

Da bi se obezbedila konzistentnost, često su u upotrebi fajl sistemi koji koriste *dnevnike* (*journaling file system*). Ideja je da se svi metapodaci prvo upišu u dnevnik. Skup operacija koje su potrebne da bi se neki zadatak obavio, npr. kreiranje fajla, naziva se transakcija. Tek kada su planirane operacije upisane u dnevnik, izvršavaju se stvarne promene nad podacima. Kada se jedna operacija završi, pokazivač u dnevniku se uveća tako da pokazuje na sledeću i tako sve dok se ne izvrši kompletan skup, nakon čega se on uklanja iz dnevnika. Ako se dogodi pad sistema, dnevnik će sadržati sve planirane nezavršene operacije koje treba izvršiti nakon ponovnog podizanja sistema.

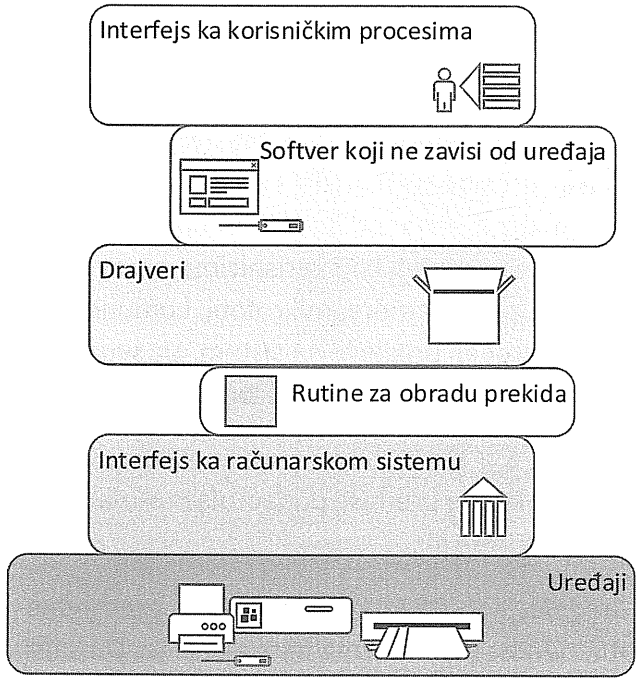
Pitanja

7.1. Iz kojih razloga radna memorija nije pogodna za trajno čuvanje podataka?
7.2. Koje uslove bi trebalo da zadovolji rešenje za trajno čuvanje podataka?
7.3. Šta je fajl, a šta fajl sistem?
7.4. Koje atribute fajlova obično podržavaju savremeni operativni sistemi?
7.5. Na koje načine operativni sistemi mogu struktuirati fajlove?
7.6. Koje tipove fajlova obično podržavaju operativni sistemi?
7.7. Koja je uloga flegova (zastavica)?
7.8. Koje operacije nad fajlovima operativni sistemi obično podržavaju?
7.9. Objasniti način na koji se fajlovi koriste.
7.10. Šta se događa po zatvaranju fajla?
7.11. Koje su prednosti sistema koji imaju mogućnost da razlikuju tipove fajlova?
7.12. Koja je uloga ekstenzije u imenu fajla?
7.13. Čemu služi magični broj u fajlu?
7.14. Kakve tipove fajlova obično podržavaju operativni sistemi?
7.15. Šta je direktorijum?
7.16. Koje operacije nad direktorijumima bi fajl sistem trebalo da omogući?
7.17. Kakve organizacije direktorijuma postoje?
7.18. Prodiskutovati organizaciju direktorijuma na dva nivoa.
7.19. Koje su karakteristike organizacije direktorijuma u strukturu stabla (drveta)?
7.20. Šta su relativne, a šta apsolutne putanje?
7.21. Na koji način se omogućava deljenja fajlova?
7.22. Na koji način se fajlovi mogu zaštititi od neželjenih pristupa?
7.23. Koji tipovi pristupnih dozvola se najčešće definišu nad fajlovima i direktorijumima?
7.24. Prodiskutovati značenje pristupnih dozvola u situacijama kada se primenjuju na fajlove i direktorijume.

7.25. Analizirati strukturu fajl sistema.
7.26. Na koji način se mogu alocirati slobodni blokovi memorije i čuvati informacije o blokovima fajlova?
7.27. Koje su odlike neprekidne alokacije za čuvanje fajlova?
7.28. Kako se uz pomoć povezanih listi mogu čuvati fajlovi?
7.29. Koje su prednosti indeksirane alokacije za čuvanje fajlova?
7.30. Kako se mogu implementirati direktorijumi?
7.31. Objasniti način predstavljanja direktorijuma uz pomoć listi.
7.32. Šta je heš tabela?
7.33. Kako se direktorijumi mogu predstaviti uz pomoć heš tabela?
7.34. Na koji način se manipuliše sa slobodnim prostorom?
7.35. Koje su odlike virtuelnih fajl sistema?
7.36. Navesti primer nekonzistentnosti podataka prouzrokovanih padom sistema.
7.37. Na koji način se može obezbediti konzistentnost podataka u fajl sistemima?

8. UPRAVLJANJE ULAZNO-IZLAZNIM UREĐAJIMA

Jedan od osnovnih zadataka operativnog sistema je da upravlja ulazno-izlaznim uređajima i da pri tome pruži što jednostavniji, ali i koliko je moguće univerzalniji interfejs između uređaja i sistema. Sa druge strane, trebalo bi od korisnika prikriti detalje upravljanja uređajima i obezbediti što pogodniji interfejs za njihovo efikasno korišćenje. Ispunjavanje ovih zahteva predstavlja izazov, uzimajući u obzir činjenicu da postoji čitav spektar različitih uređaja i da se njihov broj i raznolikost svakim danom uvećava. Takođe, ulazno-izlazni uređaji predstavljaju usko grlo kada su performanse u pitanju. Zbog toga je veoma važno da operativni sistem što bolje upravlja ovim uređajima kako bi se postigle optimalne performanse.



Slika 8.1. Nivoi upravljanja ulazno-izlaznim uređajima

Komunikacija sa ulazno-izlaznim uređajima odvija se u posebnom (sistemskom) režimu operativnog sistema za koji je odgovorno jezgro (*kernel*). Sistemski pozivi su jedini način kojim korisnik ima mogućnost da pristupi hardveru. Iz tog razloga, često se događa da se veliki deo softvera u jezgru operativnog sistema odnosi na upravljanje uređajima.

Upravljanje uređajima ima složenu hijerarhiju (slika 8.1). Na najnižem nivou nalaze se uređaji, odnosno hardver, zatim sledi interfejs kojim se on vezuje za računarski sistem. U nekim slučajevima se koriste rutine za obradu prekida koje se nalaze na sledećem nivou a onda slede drajveri koji omogućavaju korišćenje uređaja. Na sledećem (logičkom) nivou se nalazi softver koji ne zavisi od uređaja i na kraju interfejs ka korisničkim procesima uz pomoć kojeg se obezbeđuju usluge.

8.1. HARDVERSKE KOMPONENTE

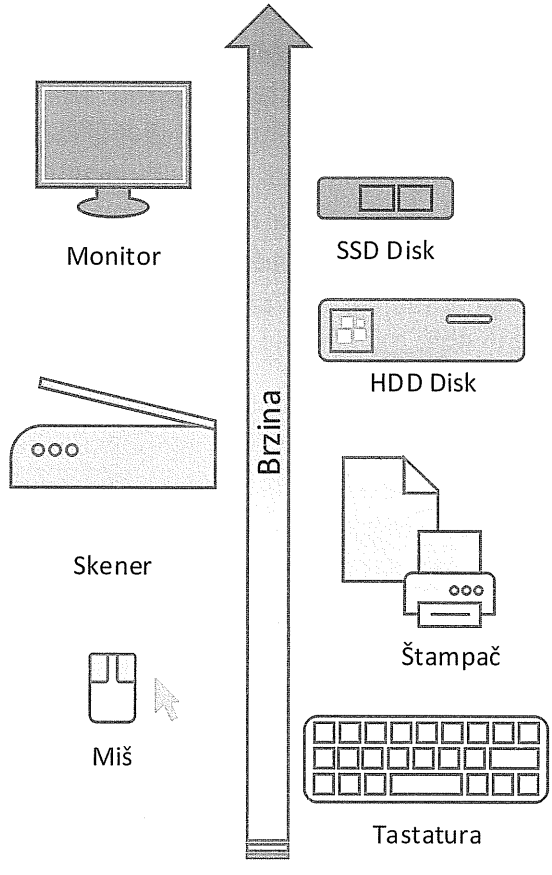
Postoje razni tipovi uređaja koji se mogu koristiti u okviru računarskih sistema. Diskovi i optički uređaji su jedan primer. Oni služe kao sekundarna memorija, odnosno, omogućavaju skladištenje i trajno čuvanje podataka, a često se koriste i za smeštanje podataka koji su potrebni aktivnim procesima prilikom multiprogramiranja. Tastature, miševi, monitori itd. su uređaji koji korisnicima omogućavaju interakciju sa procesima. Tu su i video kamere, mikrofoni, komunikacioni kablovi koji omogućavaju da se uređaji priključe na sistem. Uz svu raznolikost ulazno-izlaznih uređaja, prisutna je i razlika u brzinama pristupa i rada ovih uređaja (slika 8.2.).

Na osnovu načina na koji se prenose podaci ulazno-izlazni uređaji se mogu podeliti na:

- *Blok uređaje*, koji podatke obrađuju u prethodno definisanom formatu, odnosno blokovima fiksne dužine. Dužina blokova može varirati od uređaja do uređaja a mogu se i adresirati. Svi transferi podataka se obavljaju prebacivanjem odgovarajućeg broja blokova koji su međusobno nezavisni. Svakom bloku se može

direktno pristupiti. Diskovi, DVD-ROM i USB memorije su tipični predstavnici blok uređaja.

- *Karakter uređaje*, koji manipulišu podacima koji su u formi niza (struje, toka) karaktera. Uređaj ovog tipa nemaju mogućnost direktnog pristupanja memoriji i nisu adresibilni. Ovoj grupi uređaja pripadaju štampači, mrežni interfejsi, miševi itd.

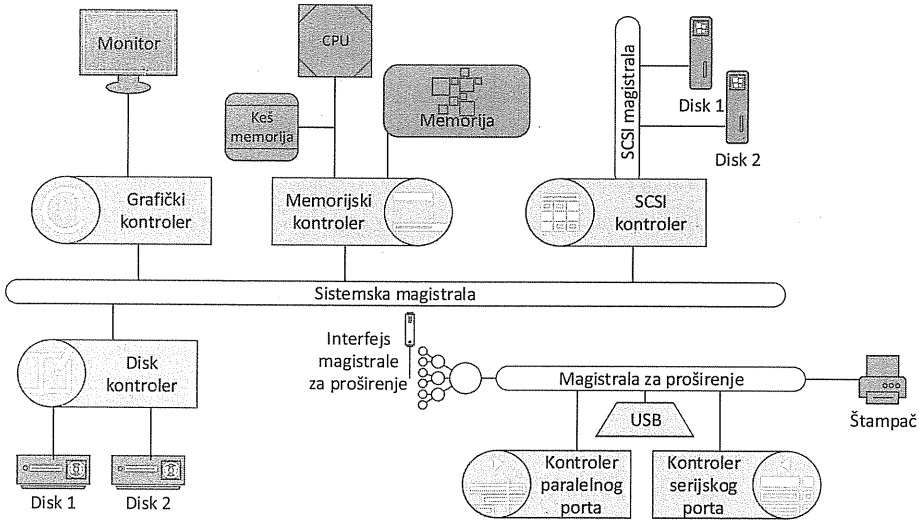


Slika 8.2. Brzina pristupa ulazno-izlaznim uređajima

8.2. INTERFEJS UREĐAJA

Uređaji se na različite načine mogu povezati na računarski sistem. Način povezivanja često određuje brzina uređaja, tako da oni sporiji obično imaju jednostavnije vezivanje dok brži zahtevaju složeniji pristup.

Uređaji se povezuju na sistem uz pomoć *priključaka*. Ova veza ne mora da bude isključivo izvedena uz pomoć kabla, već može biti i bežična. Komunikacija se dalje odvija preko magistrala koje povezuju priključke sa ostalim delovima sistema (slika 8.3). *Magistrala* je skup linija prenosa koje povezuju uređaje i pri tome podržavaju jasno definisana pravila, tj. protokole za komunikaciju između njih. U okviru računarskog sistema obično postoji više magistrala, od kojih su najpoznatije *sistemska magistrala*, koja vezuje procesor sa brzim ulazno-izlaznim uređajima i *magistrala za proširenje* koja sadrži serijske i paralelne priključke namenjene za sporije uređaje poput tastature, miša, štampača itd.



Slika 8.3. Povezivanje uređaja na računarski sistem

Kontroleri su deo hardvera koji mogu opsluživati portove, magistrale ili uređaje. Na primer, *kontroler serijskog porta* je čip koji se nalazi u računarskom sistemu i ima zadatak da kontroliše signale na žicama serijskog porta. Sa druge strane, *SCSI* (*Small Computer System Interface*)

kontroler je adapter koji najčešće ima svoj procesor i memoriju a funkcija mu je da formira novu magistralu na koju se mogu povezivati uređaji.

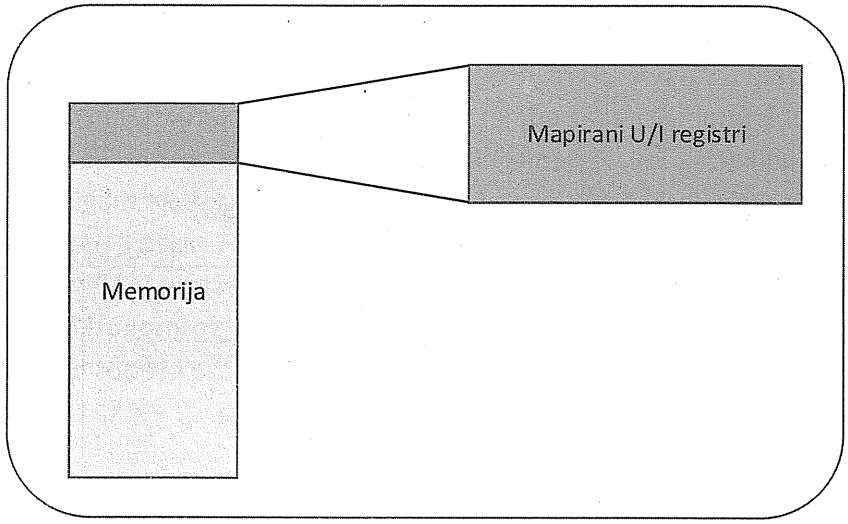
Kontroleri obično imaju registre koji su najčešće memorijski čipovi i koriste se za prenos podataka i za komunikaciju sa ostatkom sistema. Uobičajeno je da kontroleri imaju sledeće registre:

- *Registar primljenih podataka*, u koji se smeštaju podaci preuzeti sa uređaja.
- *Registar poslatih podataka,* u koji se smeštaju podaci koje treba uskladištiti na uređaj.
- *Statusni registar*, u koji se smeštaju podaci u vidu kodova koji govore o trenutnom statusu uređaja. Ovi podaci su namenjeni procesoru i pomoću njih se signalizira da li su podaci učitani, da li je izvršena zadata komanda, da li je došlo do greške itd.
- *Kontrolni registar*, u koji procesor upisuje komande koje treba izvršiti ili informacije u vezi sa načinom rada uređaja.

Procesor komunicira sa uređajem korišćenjem registara. Jedan od načina za komunikaciju je uvođenje posebnih instrukcija za upis ili čitanje iz registara. Takvim instrukcijama aktiviraju se posebne linije magistrale, da bi se pravila razlika između pisanja u memoriju i pisanja na registar kontrolera. Podaci potom putuju magistralom i smeštaju se u registre, ili se prenos vrši u obrnutom smeru. Dakle, registri se mogu koristiti za prenos podataka od procesora ka uređaju i obrnuto, za prenos instrukcija od procesora ka uređaju i za prenos informacija o statusu uređaja procesoru.

Drugi način za komunikaciju podrazumeva da kontroleri podržavaju memorijski mapirane ulazno-izlazne operacije. U tom slučaju je registrima moguće pristupati direktno putem memorijske adrese, kao da su sastavni deo glavne memorije, tako da nema potrebe za uvođenjem posebnih instrukcija. Ovakav pristup zahteva da procesor ima mogućnost da adresira registre kao što adresira ostale delove memorije (slika 8.4). Postoje sistemi koji podržavaju oba pristupa, gde je moguće adresirati registre ali i koristiti posebne instrukcije (npr. grafički procesor GPU).

Memorijsko mapiranje registara ima svojih prednosti, ali i mana. Na ovaj način izbegava se upotreba asemblera koja bi bila obavezna pri radu sa registrima korišćenjem posebnih instrukcija. U slučaju memorijskog mapiranja operativni sistem vidi registre kao obične promenijive pa ih je moguće koristiti u programskim jezicima navođenjem njihovih adresa. Međutim, ova mogućnost može predstavljati problem. Naime, jedna od čestih grešaka pri programiranju jeste upis podataka na slučajni, greškom izabrani deo memorije. Pošto se registru uređaja pristupa kao i svakom drugom delu memorije, njegov sadržaj se lako može greškom promeniti. Ovakve greške bi se izbegle ako bi se pri programiranju koristile posebne instrukcije koje bi jasno određivale da li se podaci upisuju u memoriju ili u registar uređaja. Takođe, zaštita registara od neželjenog pristupa se može realizovati isključivanjem adresa ovih registara iz prostora koji se može adresirati.

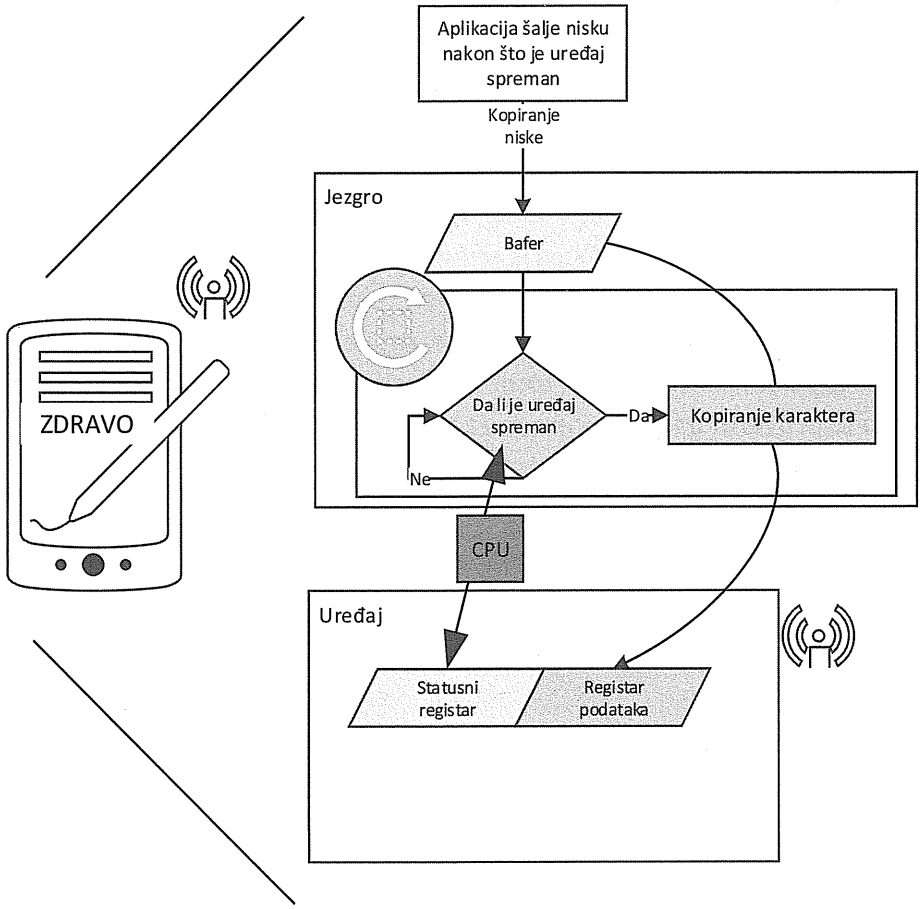


Slika 8.4. Memorijsko mapiranje registra U/I uređaja

8.2.1. Tehnika prozivanja

Tehnika prozivanja (*Polling*) predstavlja najjednostavniji pristup za upravljanje ulazno-izlaznim uređajima koji podrazumeva da procesor odradi sav posao. Ideja je da se uređaj, konkretno njegov statusni registar,

sve vreme proverava. Pri tome uređaj postavljanjem određene vrednosti u registar signalizira sistemu da je pročitao ili pripremio podatak.



Slika 8.5. Tehnika prozivanja

Ova ideja se može prikazati na jednostavnom primeru. Neka je dat uređaj za bežično slanje podataka (npr. *Bluetooth* modul) i aplikacija koja želi da pošalje niz karaktera *„ZDRAVO“* (slika 8.5.). Aplikacija prvo koristi sistemski poziv kako bi jezgru operativnog sistema prosledila zahtev za uređajem. Ako uređaj nije spreman, poziv će vratiti grešku ili će se blokirati sve dok uređaj ne postane spreman. Kada se uređaj dodeli aplikaciji, upućuje se sistemski poziv sa zahtevom da mu se prosledi data niska karaktera. Operativni sistem prvo kopira nisku u bafer jezgra, radi

lakšeg daljeg pristupa, i proverava da li je uređaj spreman da primi podatke. Ako uređaj nije spreman, čeka se. Kada uređaj postane spreman, operativni sistem kopira prvi karakter u registar podataka uređaja. Čim se prvi karakter kopira, operativni sistem proverava da li je uređaj spreman za novi. Ova provera se vrši tako što se čita vrednost statusnog registra, u koji uređaj upisuje dogovorene statusne kodove. Sve dok uređaj ne pošalje prvi karakter (u ovom primeru se podrazumeva da se prenos vrši karakter po karakter), u statusnom registru će se nalaziti kôd koji označava zauzeće. Operativni sistem, odnosno procesor, sve vreme će proveravati sadržaj statusnog registra i čim se iz njega pročita da je uređaj spreman, poslaće mu se drugi karakter i tako u petlji, sve dok se ne prosledi cela niska. Pri tome, procesor neprekidno „proziva" uređaj kako bi proverio da li je spreman za novi karakter, odnosno aktivno čeka.

Ova tehnika je jednostavna, ali je neefikasna, jer se procesorsko vreme troši na učestalo proveravanje statusnog registra. U ovom primeru, procesor bi prosledio karakter, a onda bi većinu vremena provodio proveravajući statusni registar u iščekivanju da će iz njega pročitati kôd koji označava da je moguće proslediti sledeći karakter.

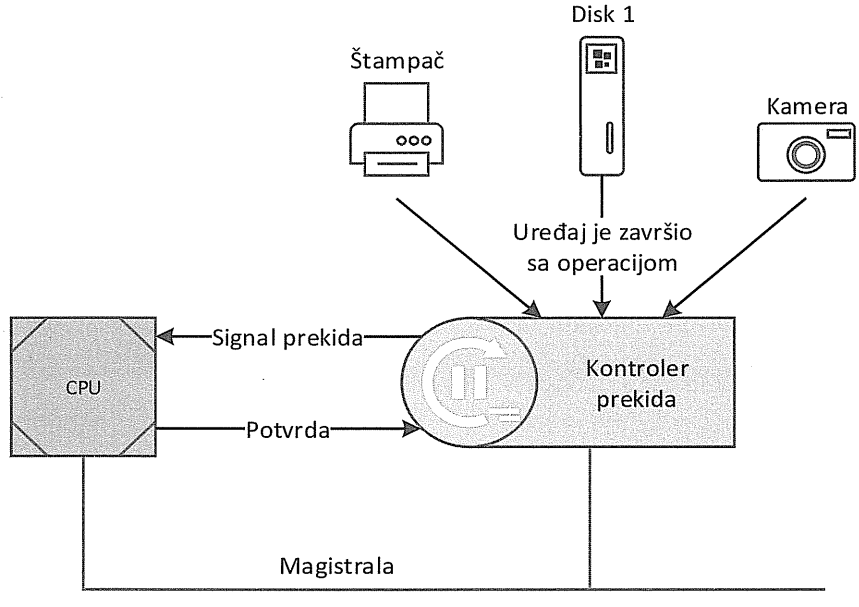
8.2.2. Prekidi

Upravljanje ulazno-izlaznim operacijama korišćenjem *prekida* se zasniva na hardverskom mehanizmu koji daje mogućnost uređajima da signaliziraju stanje, dajući tako vremena procesoru da obavlja drugi posao sve dok uređaj ne pošalje signal da je spreman (slika 8.6.).

Za realizaciju ovakvog pristupa potrebno je da hardverski deo računarskog sistema sadrži posebnu liniju za prekide. Princip funkcionisanja sistema zasnovanog na prekidima se može ilustrovati na sledeći način (slika 8.7.):

1. Proces inicira neku operaciju na uređaju obraćajući se odgovarajućem drajveru.
2. Drajver preko kontrolera prosleđuje operaciju uređaju.

3. Kada uređaj završi sa operacijom njegov kontroler procesoru šalje signal preko linije za slanje prekida.
4. Procesor prekida izvršavanje procesa koji se trenutno obrađuje a dispečer prebacuje njegov kontekst, kako bi se obradio prekid.
5. Prekid se obrađuje uz pomoć rutine za obradu prekida.
6. Nakon obrade prekida prekinuti proces može da nastavi izvršavanje ukoliko mu to planer dozvoli.

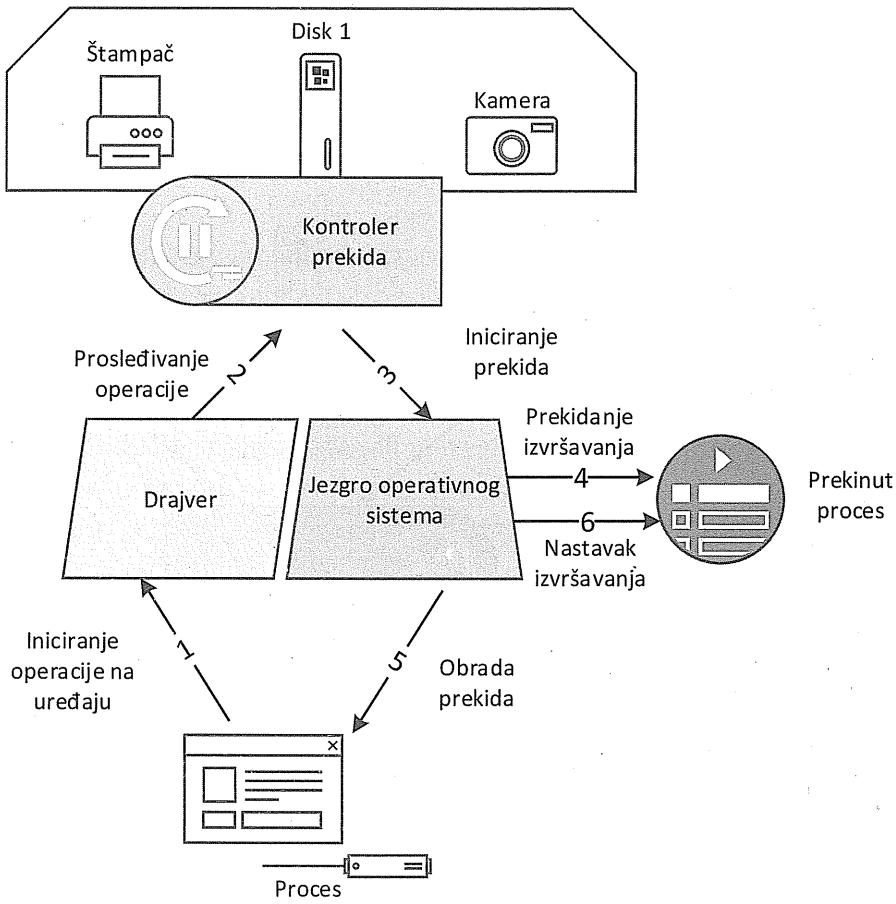


Slika 8.6. Kontroler prekida

Kada je prethodni primer u pitanju, to bi značilo da procesor šalje karakter uređaju i prelazi na izvršavanje nekog drugog zadatka. Kada uređaj prosledi karakter i bude spreman da primi novi, on izaziva prekid. Tada procesor prekida posao koji trenutno radi, prosleđuje novi karakter uređaju i nastavlja prekinuti posao i tako u petlji sve dok se ne pošalju svi karakteri.

U praksi, opisanom mehanizmu se obično dodaje još nekoliko funkcija. Potrebno je da operativni sistem bude u mogućnosti da odloži prekid ako se trenutno izvršava neki veoma bitan proces. Zatim je potrebno efikasno detektovati koji uređaj je prosledio prekid, umesto da se ispituju svi. I

najzad, potrebno je prekidima dodeliti prioritete, kako bi sistem mogao da odreaguje sa različitim vremenima odziva u zavisnosti od hitnosti obrade signaliziranog prekida.



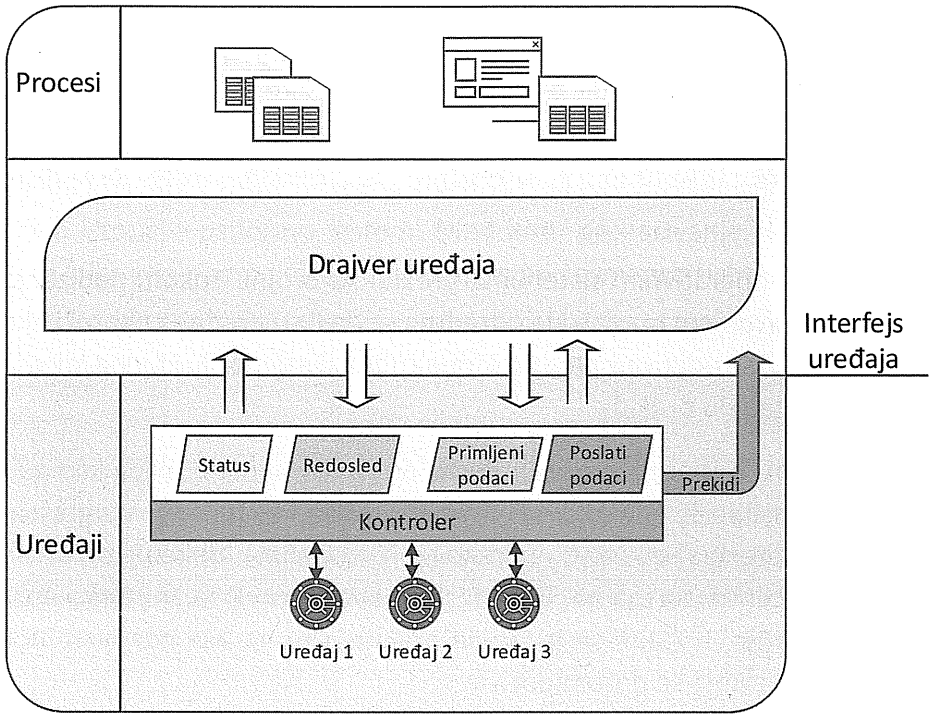
Slika 8.7. Princip funkcionisanja sistema zasnovanog na prekidima

Sve ove funkcije realizuje hardverski uređaj koji se naziva *prioritetni prekidni kontroler PIC* (*Priority Interrupt Controller*) (slika 8.8.). Većina procesora umesto jedne ima dve posebne linije za signale prekida:

- *Nemaskirajuća* (*non-masking*) linija;
- *Maskirajuća* (*masking*) linija.

Signal prekida koji je poslat po nemaskirajućoj liniji uvek može da prekine izvršenje tekućeg procesa. Nemaskirajuća linija se koristi za slanje signala prekida kao što su kritične hardverske greške, npr. greške u memoriji.

Maskirajuća linija se može isključiti (zamaskirati) tokom izvršavanja važnih delova procesa (kritičnih sekcija) koji ne smeju biti prekinuti. Ona se koristi za slanje signala prekida koji potiču od uobičajenih operacija, kao što je zahtev za nekom ulazno-izlaznom operacijom, i standardnih grešaka koje se javljaju na ulazno-izlaznim uređajima.



Slika 8.8. Prekidi

Relativno mali broj uređaja može da pošalje signal prekida, i za svaki od njih postoji posebna rutina za obradu koja se nalazi u memoriji. Mehanizam prekida prihvata adresu, broj koji predstavlja poziciju u *tabeli prekida* (*interrupt vector*), koja čuva adrese svih rutina za obradu prekida u memoriji. Kada dođe do prekida, glavna funkcija tabele je da se ne moraju pretraživati svi mogući izvori kako bi se utvrdilo koji od njih

zahteva operaciju, već da se iz tabele odmah pročita adresa rutine koju treba izvršiti. U praksi, sistemi obično imaju više uređaja koji mogu izazvati prekid nego što postoji mesta u tabeli prekida tako da se taj problem rešava ulančavanjem. Svi izvori se dele u skupove, i za svaki skup se pravi lista koju čine adrese odgovarajućih rutina za obradu prekida. Kada se prekid izazove, rutine iz liste se pretražuju jedna po jedna, dok se ne nađe odgovarajuća. Na ovaj način se postiže kompromis između efikasnosti i veličine tabele.

Sistem prekida obično primenjuje prekidne prioritetne nivoe, a to je mehanizam koji omogućava da se odloži opsluživanje prekida niskih prioriteta, dok se opslužuju prekidi visokih prioriteta (bez maskiranja). Takođe, kada se prekid višeg prioriteta desi, prekida se opsluživanje prekida nižeg prioriteta, i opslužuje se prekid višeg prioriteta (*preemption*).

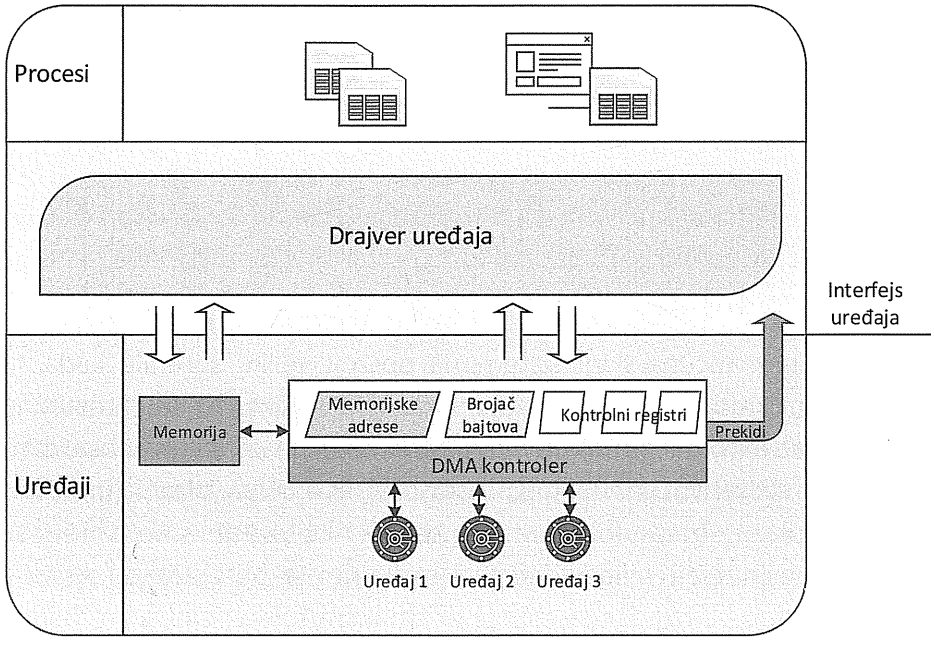
U modernim operativnim sistemima prekidi su brojni. Tokom podizanja sistema, ispitivanjem magistrale određuje se da li su uređaji prisutni i ako jesu, postavlja se odgovarajuća prekidna rutina u memoriju i njena adresa se upisuje u tabelu prekida.

Za vreme ulazno-izlaznih ciklusa, uređaj postavlja signal prekida kad god zahteva servisiranje, tj. kada je komanda gotova ili kad se javila greška. Mehanizam prekida se koristi i za rukovanje izuzecima, deljenje nulom ili pristup zaštićenim adresama. U svim slučajevima, prekida se izvršavanje tekućeg procesa, a zatim se iz tabele prekida čita adresa odgovarajuće servisne rutine koja se potom poziva.

8.2.3. Direktan memorijski pristup – DMA

Prekidi su veoma efikasan pristup ako se ne dešavaju previše često. U prethodnom primeru, sistem sa prekidima bi izvršavao prekid za svaki karakter, što je prilično neefikasno, jer se za svaki prekid vrši prebacivanje konteksta na šta se troši određeno vreme. *Direktan memorijski pristup* (*Direct Memory Access – DMA*) je jedan od načina kojim se rešavaju problemi ovakvog tipa. To je hibridni pristup tehnike prozivanja i prekida.

Za njegovu implementaciju potreban je hardverski uređaj koji se naziva DMA kontroler (slika 8.9.). DMA kontroleri imaju pristup sistemskoj magistrali nezavisno od procesora. Oni imaju nekoliko registara, najčešće registar memorijskih adresa, registar brojača bajtova i kontrolne registre, kojima procesor može da pristupa. U kontrolne registre smeštaju se informacije o ulazno-izlaznom uređaju koji se koristi, smeru transfera podataka, jedinici prenosa, broju jedinica koje treba preneti itd. Na osnovu prethodnog primera može se zaključiti da je suština DMA kontrolera da se on koristi kako bi prosleđivao uređaju karakter po karakter, bez opterećivanja glavnog procesora. DMA u stvari vrši aktivno čekanje (*polling*) na uređaju, dok glavni procesor ostaje slobodan za druge poslove. Ovim se prekidi ne dešavaju kod svakog karaktera, već DMA pristupa memoriji samo kada se bafer isprazni. Trebalo bi primetiti da u trenucima kada DMA pristupa memoriji, glavni procesor je blokiran da uradi isto, što usporava sistem. Međutim, prebacivanje posla na DMA kontroler generalno poboljšava ukupne performanse sistema.



Slika 8.9. Direktan memorijski pristup

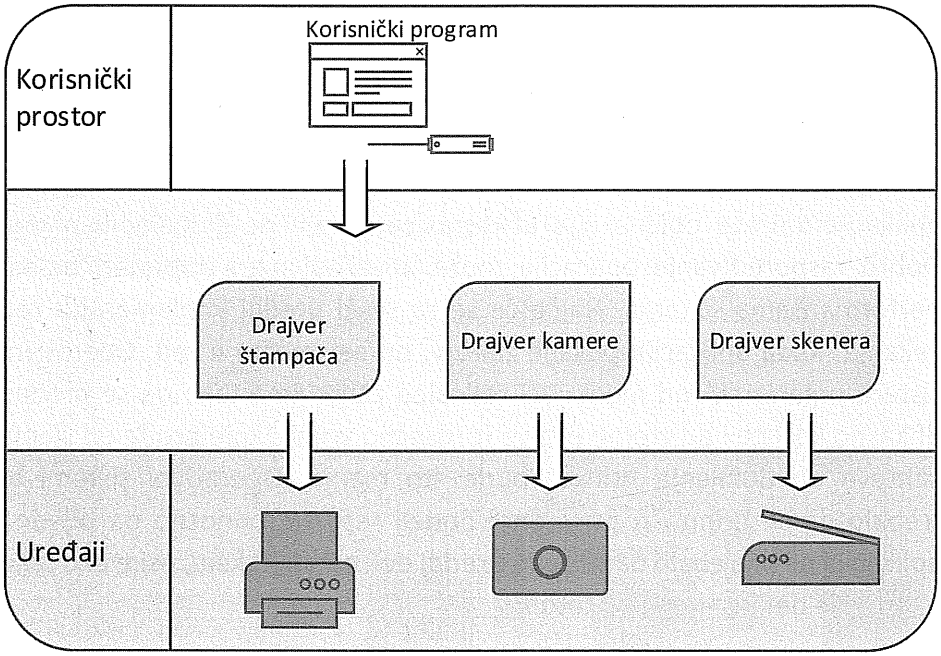
8.3. Drajveri

Veoma je važno da interfejs ulazno-izlaznog podsistema prema aplikacijama, odnosno korisnicima, bude što uniformniji. To podrazumeva da, na primer, ako aplikacija ima potrebu da odštampa podatke, ona ne mora da ima informaciju o tome kom tipu štampača se obraća, niti kako on tačno radi. Uređaji se grupišu u klase i za svaku klasu definiše se skup funkcija kojima se upravlja konkretnim uređajem. Taj skup čini interfejs čija je glavna uloga da razlike i detalje o pojedinačnim uređajima sakrije od aplikacije. To bi značilo da svaka aplikacija koja želi da koristi štampač koristi isti interfejs koji je standardizovan i ne menja se, bez obzira na model, generaciju štampača itd. Detalji upravljanja uređajima su enkapsulirani u posebne module koji se nazivaju drajveri i koji se pišu za svaki uređaj posebno. Oni imaju zadatak da prikriju razlike između uređaja koji pripadaju istoj klasi i pruže što uniformnije interfejse (slika 8.10.). Drajveri se mogu pisati tako da funkcionišu u privilegovanom režimu rada, gde direktno pristupaju potrebnom hardveru, ali i u korisničkom režimu. Prvi pristup je dobar zbog direktnog pristupa i veće slobode koja se pruža drajveru, ali nosi i veću odgovornost, jer loše napisan drajver može prouzrokovati pad sistema. Sa druge strane, drajver koji radi u korisničkom režimu mora koristiti sistemske pozive koji su mu pruženi, što degradira performanse (nekad je i nemoguće napisati drajver kao korisničku aplikaciju), ali zato ne postoji mogućnost pada sistema usled greške u drajveru.

U savremenim računarskim sistemima postoji veliko šarenilo kada su uređaji u pitanju, tako da postoje i razne klasifikacije koje rezultuju različitim pristupom. Uređaji se mogu razlikovati po smeru prenosa, tako da postoje isključivo ulazni (miš, tastatura), isključivo izlazni (monitor, štampač) i ulazno-izlazni (disk, mrežna karta). Ako se osim smera prenosa podataka posmatra i veličina jedinice sa kojom se barata onda postoje blok i karakter uređaji. Dalje, ako se posmatra način pristupa uređajima oni se mogu podeliti na one sa sekvencijalnim pristupom (modem) i one sa direktnim pristupom (fleš memorija). Deljivi uređaji su oni koje može

koristiti veći broj procesa u istom trenutku, dok su nedeljivi oni koje može koristiti tačno jedan proces u jednom trenutku. Jedna od najvažnijih podela ulazno-izlaznih uređaja se može izvesti na osnovu načina upotrebe:

- Uređaji za čuvanje podataka (disk, fleš memorija);
- Uređaji za prenos podataka (mrežna karta, modem);
- Uređaji za interakciju sa korisnicima (miš, tastatura, monitor).



Slika 8.10. Drajveri

Uređaji se na osnovu ovakvih zajedničkih karakteristika grupišu u već pomenute klase koje imaju isti interfejs prema aplikacijama. Bilo bi poželjno da i interfejsi za različite uređaje budu što sličniji. Na taj način se korisnici, ali i operativni sistem, lišavaju razmatranja karakteristika uređaja koje žele da koriste. Dovoljno je da se navede koji uređaj će se koristiti, koju operaciju treba izvršiti i koje podatke treba uzeti ili preneti, odnosno gde treba smestiti podatke koji se uzimaju sa uređaja.

8.4. SOFTVER ZA UPRAVLJANJE KOJI NE ZAVISI OD UREĐAJA

Kada drajveri premoste različite osobine uređaja koji čine računarski sistem i stvore udoban interfejs, na scenu stupa softver koji treba da omogući što efikasnije korišćenje tih uređaja. Pri tome, ovaj sloj softvera se projektuje tako da ne zavisi od uređaja koji se nalaze u sistemu i da prema svima ima što uniformniji pristup. Na ovom nivou se teži da se što manje razlikuje pristup kada su različite klase uređaja u pitanju.

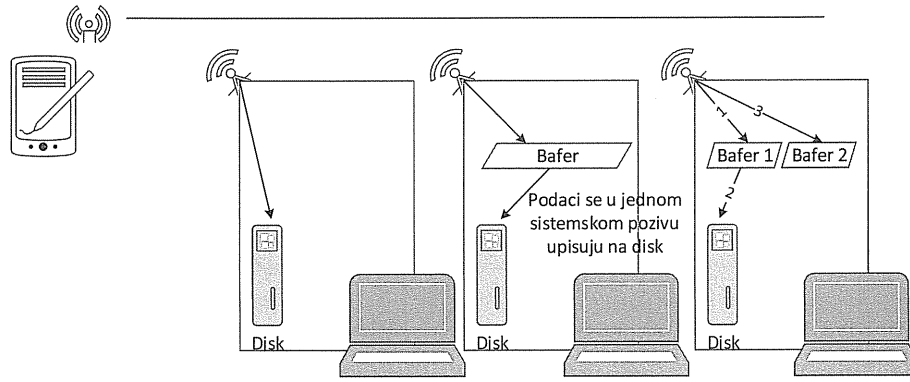
8.4.1. Planiranje ulazno-izlaznih operacija

Vremensko planiranje ulazno-izlaznih operacija podrazumeva određivanje redosleda kojim će se ove operacije izvršavati tako da sistem funkcioniše što efikasnije. Izvršavati operacije onim redosledom kojim ih aplikacije zatraže obično nije efikasno za sistem na globalnom nivou. Dobro raspoređivanje operacija može u velikoj meri doprineti boljim performansama sistema. Najčešće se za svaki uređaj implementira red čekanja. Kada aplikacija pošalje zahtev, on se smešta u red. Operativni sistem organizuje red menjajući redosled zahtevima tako da se uređaji efikasno koriste i da vreme odziva (prosečno vreme koje prođe od slanja zahteva do dobijanja usluge) bude što manje. Operativni sistemi bi trebalo da se brinu i o pravednoj dodeli resursa, odnosno da nijedna aplikacija ne bi trebalo da okupira uređaj dok ostale čekaju, osim ako tako nešto nije definisano prioritetom.

8.4.2. Baferovanje

Pod *baferom* (*data buffer*) se podrazumeva deo memorije u koji se privremeno smeštaju podaci koje treba preneti od uređaja ka aplikaciji (ili drugom uređaju) ili u obrnutom smeru. *Baferovanje* je tehnika koja se zasniva na korišćenju bafera i može se koristiti iz više razloga. Jedna od situacija u kojima se efikasno primenjuje baferovanje je kada uređaji koji „sarađuju“ imaju različite brzine. Na primer, neka su data dva uređaja, *Bluetooth* modul za primanje podataka bežičnim putem i disk računara, koji je daleko brži (slika 8.11.). Neka je zadatak da se podaci koji se primaju uz pomoć uređaja upišu na disk. Bez baferovanja, za svaki karakter koji se

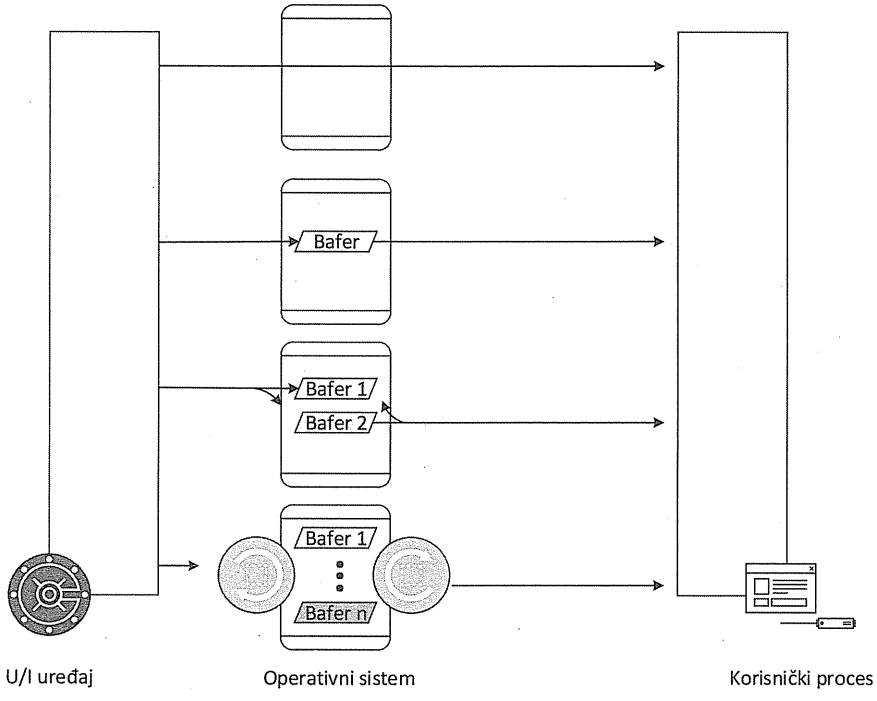
dobije događao bi se prekid kojim bi modul javljao da karakter treba upisati, što ne bi bilo efikasno. Korišćenjem bafera omogućava se privremeno upisivanje podataka u memoriju sve do određene granice, tj. do popunjavanja bafera. Jednom kada se bafer napuni, izvršava se prekid, podaci se jednim sistemskim pozivom upisuju na disk i bafer se puni dalje. Unapređenje bi bilo korišćenje dva bafera, takozvano *dvostruko baferovanje*. Za upis podataka na disk potrebno je neko vreme, a nove podatke koji pristižu treba za to vreme negde smeštati, i za to se koristi drugi bafer. Kada se on napuni, podaci se smeštaju ponovo u prvi i tako naizmenično. Dvostruko baferovanje je jedna od tehnika koja se koristi i pri keširanju diskova, dva bafera na relaciji memorija-disk, za podatke koji se upisuju, i dva bafera na relaciji disk-memorija, za čitanje, čime se premošćava razlika u brzini između memorije i diska. Ova ideja se generalizuje prstenom bafera. Određeni broj bafera se redom puni i po istom redosledu prazni (slika 8.12.).



Slika 8.11. Baferovanje

Baferi su vrlo korisni u mrežnim okruženjima, gde se velike poruke pre slanja dele na manje pakete koji se pojedinačno šalju preko mreže. Paketi dolaze i primaju se u bafer po redosledu koji nije nužno isti kao pri slanju. Neki paketi mogu doći drugim mrežnim putem, a neki se mogu izgubiti i poslati ponovo, što izaziva kašnjenje a time i poremećen redosled kod primaoca. Paketi su, međutim, obeleženi rednim brojevima, tako da se,

nakon što se svi privremeno prime u bafer, od njih može rekonstruisati početna poruka.



Slika 8.12. Bafer, dvostruki bafer i prsten bafera

Održavanje semantike kopiranja je još jedno polje gde je vrlo pogodno koristiti bafer. Ako neka aplikacija u memoriji čuva određene podatke koje želi da upiše na disk, ona izdaje sistemski poziv za upis podataka. Međutim, do nekonzistentnosti podataka može doći ako pre nego što se svi podaci uspešno upišu na disk, aplikacija izvrši promene u memoriji, odnosno promeni izvor sa kog se podaci čitaju. Tako nešto je moguće kod asinhronih sistemskih poziva, gde bi aplikacija zadala komandu da se podaci upišu, i dobila nazad kontrolu pre kompletnog upisa. U takvom slučaju, moglo bi se očekivati da podaci na disku ne bi bili identični onim u memoriji u trenutku poziva funkcije za upis na disk, i kaže se da je semantika kopiranja narušena. Da bi se to izbeglo, operativni sistem u

svom jezgru definiše bafer u koji, pri sistemskom pozivu za upis, kopira sve podatke koje mu aplikacija prosleđuje, pa tek onda predaje kontrolu nazad. Time poziv i dalje ostaje asinhroni, jer se ne čeka da se podaci upišu na disk, ali su isti sigurni od promene u baferu jezgra.

8.4.3. Obrada grešaka

Kod ulazno-izlaznih operacija greške su česte i podrazumeva se da će do njih doći, pa su operativni sistemi dužni da ih obrade na najbolji mogući način. Neke greške su specifične za uređaj i u tim situacijama drajveri preuzimaju odgovornost za njihovu obradu. Međutim, obrada grešaka bi trebalo da bude implementirana na što univerzalniji način.

Greške mogu nastati kada se od ulazno-izlaznih uređaja zahteva nešto što nije moguće uraditi. Tipičan primer je pokušaj pisanja na isključivo ulazne uređaje (tastatura, miš, skener) ili čitanja sa izlaznih uređaja (štampač, monitor, projektor). Takođe, pogrešno adresiranje bafera ili uređaja koji ne postoji u sistemu spadaju u greške ovakvog tipa. Njihova obrada podrazumeva da se procesu pošalje odgovarajući kôd kojim se označava tip greške.

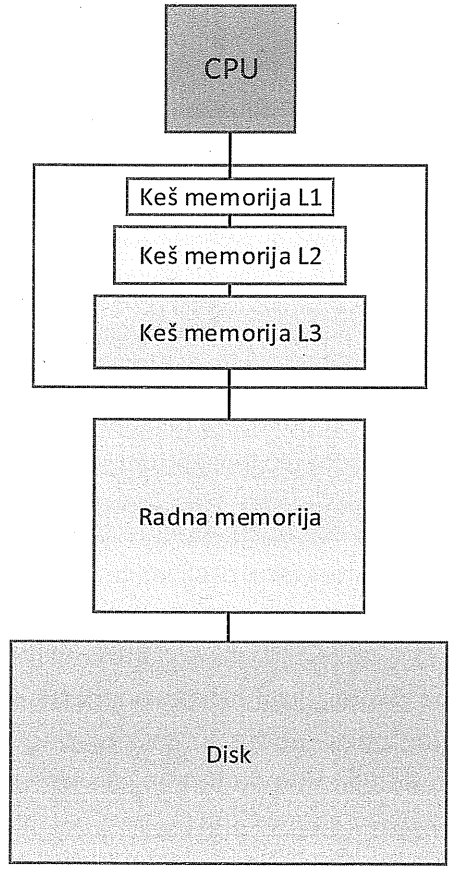
Drugačije greške su ulazno-izlazne greške poput pokušaja čitanja sa uređaja koji nije priključen ili pokušaja pisanja na oštećeni deo diska. Kod ovakvih grešaka konsultuje se drajver radi pronalaženja rešenja, a ako on nije u mogućnosti da problem reši, isti se prosleđuje sistemu za obradu grešaka.

Reakcije operativnog sistema na greške mogu biti različite i zavise od okruženja i prirode greške. Često se u odluku uključuju i sami korisnici tako što im se kroz dijalog ponudi da odaberu koja će biti reakcija. Automatske reakcije mogu podrazumevati ponovne pokušaje da se operacija obavi, npr. kod čitanja podataka, ili samo obaveštenje da je do greške došlo.

8.4.4. Keširanje

Keš (*cache*) memorija je brza memorija koja čuva kopiju podataka. Ona je obično malog kapaciteta, ali veće brzine od radne (primarne, glavne)

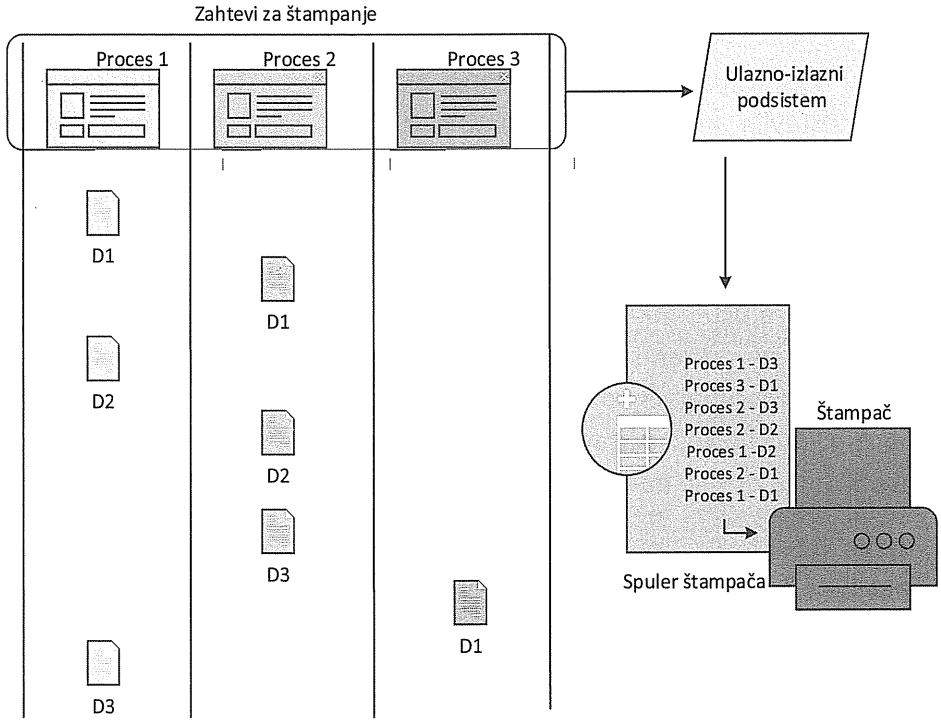
memorije. *Keširanje* je tehnika pri kojoj se deo podataka kopira u keš memoriju, i kada se zatraže podaci (ponovo), prvo se proverava da li se oni nalaze u kešu. Ako se nalaze, štedi se vreme koje bi se trošilo na pristup radnoj, sporijoj memoriji, i podaci se čitaju iz keš memorije. Ako se traženi podaci ne nalaze u keš memoriji, onda se oni ipak moraju pročitati iz radne memorije. Na primer, moderni procesori sadrže nekoliko nivoa keš memorije. Svaka je brža od prethodne, ali i manjeg kapaciteta. Prvo se podatak traži u najmanjoj, najbržoj memoriji. Ako nije tamo, prelazi se na sledeći nivo i tako redom (slika 8.13.). Keširanje je bitna tehnika koja značajno doprinosi performansama sistema.



Slika 8.13. Keš memorija

Važno je napomenuti da su keširanje i baferovanje dve različite tehnike, ali se u pojedinim slučajevima ista memorija može koristiti u obe svrhe. Ako se podaci pročitaju sa diska i kopiraju u bafer u memoriji, radi se o baferovanju. Međutim, aplikacija može nadalje iste podatke čitati direktno iz memorije, bez trošenja vremena na pristup daleko sporijem disku, pa je time efektivno izvršeno i keširanje.

8.4.5. Spuler



Slika 8.14. Spuler štampača

Pod nedeljivim uređajima podrazumevaju se uređaji koje u jednom trenutku može koristiti samo jedan proces i pri tome ih mora iskoristiti do kraja. *Spuler* (*SPOOL – Simultaneous Peripheral Operations On-Line*) je u stvari bafer u kojem se čuvaju podaci koje treba proslediti uređaju. Zgodan primer za demonstraciju rada spulera je štampač (slika 8.14.). Operativni sistem obezbeđuje da više aplikacija može zadati zahtev štampaču i proslediti podatke za štampanje u isto vreme, ali da se ti

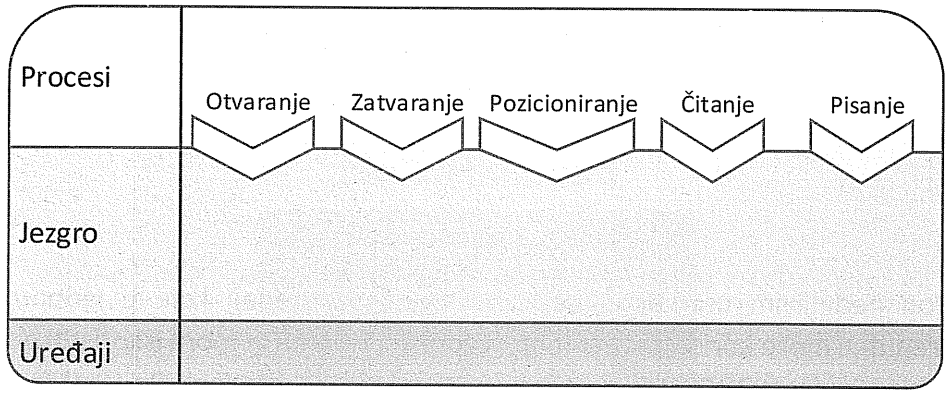
podaci štampaju redom. Podaci iz svake aplikacije se kopiraju u posebne fajlove. Kada su podaci jedne aplikacije odštampani, spuler preusmerava ulazni tok podataka tako da on postaje sledeći fajl iz reda čekanja.

Korišćenje spulera doprinosi poboljšanju efikasnosti sistema. Na ovaj način se omogućava da proces podnese svoj zahtev, kopira podatke u spuler i nastavi sa daljim radom, bez čekanja da se operacija izvrši.

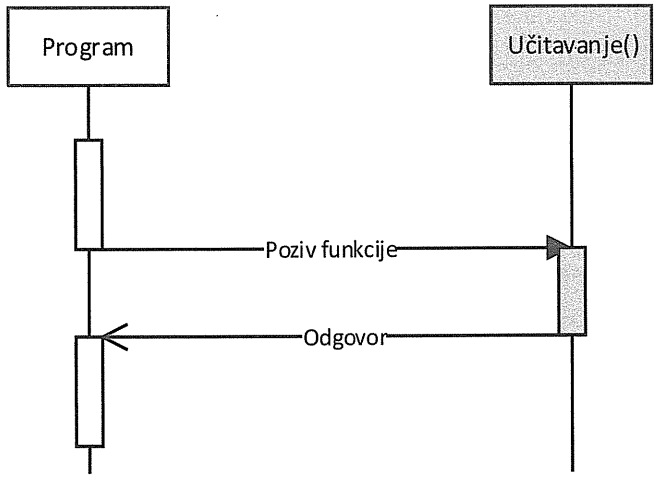
8.5. INTERFEJS KA KORISNIČKIM PROCESIMA

Na vrhu hijerarhije upravljanja ulazno-izlaznim uređajima se nalazi sloj koji predstavlja interfejs prema aplikativnom softveru, odnosno ka procesima koji koriste uređaje. Sistemski pozivi su mehanizam kojim se korisničkim procesima obezbeđuje korišćenje uređaja.

Sistemski pozivi mogu biti *blokirajući* i *neblokirajući*. Kada aplikacija pozove blokirajuću funkciju, dalje izvršavanje aplikacije je suspendovano i tek kada se sistemski poziv završi, aplikacija nastavlja dalje sa izvršavanjem, primajući povratnu vrednost sistemskog poziva. Primer bi bio poziv funkcije koja učitava karakter sa ulaza, i pritom čeka, blokirajući aplikaciju, sve dok se na ulazu ne pojavi karakter (slika 8.16.).

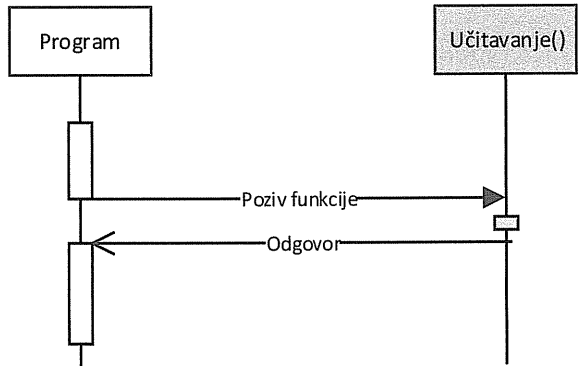


Slika 8.15. Interfejs ka procesima



Slika 8.16. Blokirajući sistemski poziv

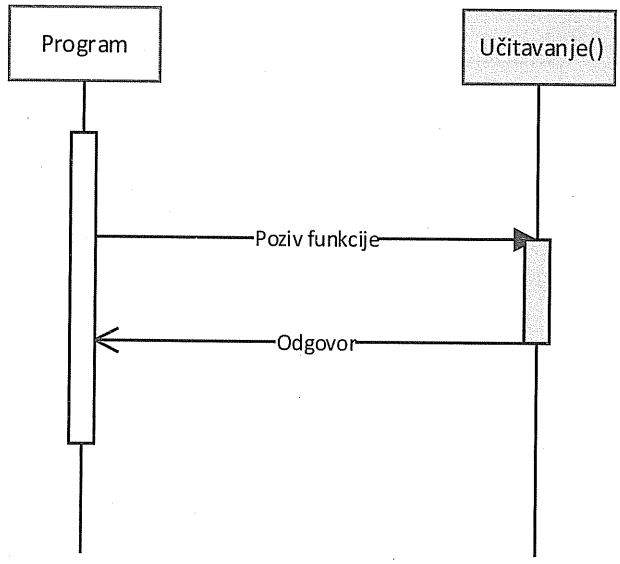
Neblokirajući sistemski poziv bi se mogao opisati na primeru funkcije koja učitava karakter sa ulaza, kao u prethodnom primeru, ali ako na ulazu ne postoji jedan, funkcija se svejedno završava (vraća kôd greške ili slično), i ne čeka da se na ulazu pojavi karakter da bi vratila vrednost (slika 8.17.).



Slika 8.17. Neblokirajući sistemski poziv

Alternativa neblokirajućim sistemskim pozivima jesu asinhroni pozivi. Asinhroni pozivi su neblokirajući u smislu da dozvoljavaju dalje izvršavanje aplikacije nakon što se pozovu, ali pritom ne vraćaju nužno

vrednost odmah nakon poziva, već se izvršavaju paralelno sa aplikacijom. Primer bi bila funkcija koja učitava karakter sa ulaza i čeka dok se jedan ne pojavi, ali dopušta aplikaciji da nastavi dalji rad paralelno. Kada se karakter pojavi, funkcija vraća vrednost aplikaciji (slika 8.18.).

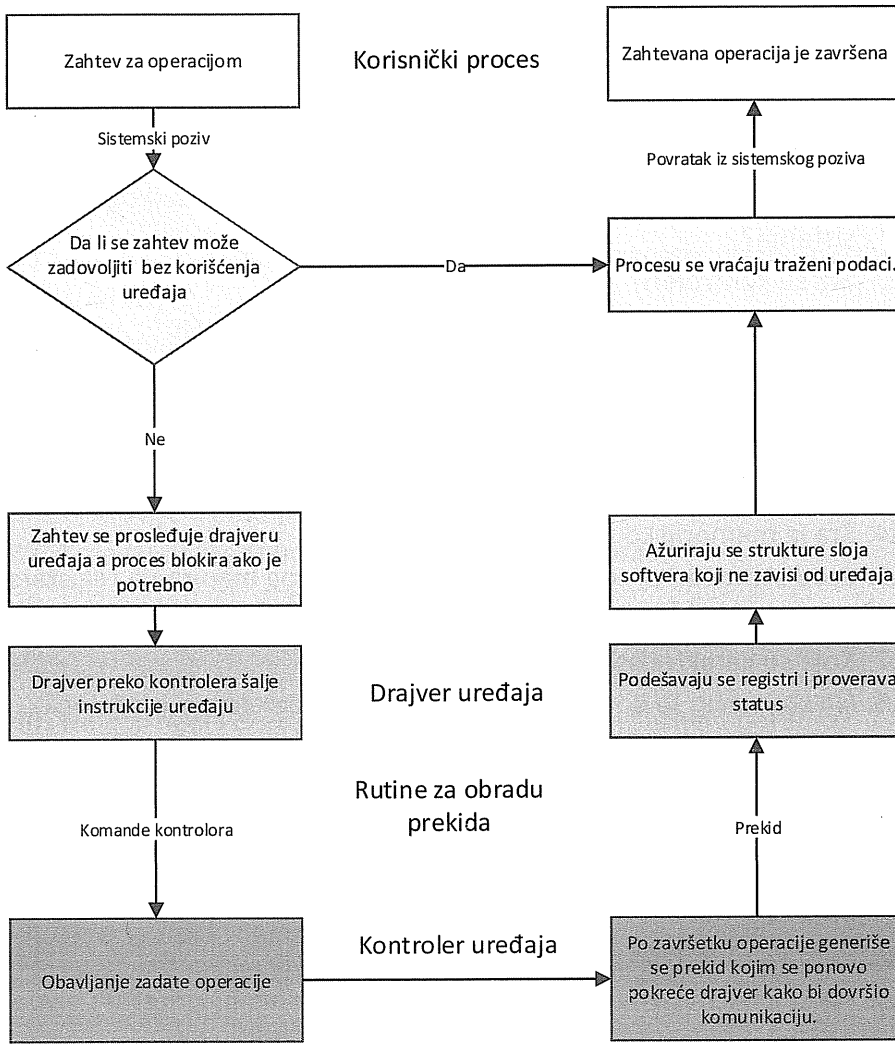


Slika 8.18. Asinhroni sistemski poziv

Imajući u vidu arhitekturu sistema za upravljanje ulazno-izlaznim uređajima, primer komunikacije korisničkog procesa sa hardverom se može prikazati na sledeći način (slika 8.19.):

1. Proces sistemskim pozivom saopštava svoj zahtev za operacijom na određenom uređaju.
2. Sloj nezavisan od hardvera prima ovaj zahtev i proverava da li može da zadovolji zahtev bez korišćenja uređaja (npr. traženi podatak je već u keš memoriji) pa ako je to ispunjeno, vraća rezultat procesu.
3. U suprotnom, zahtev se prosleđuje drajveru uređaja a proces blokira ako od tražene operacije zavisi nastavak njegovog izvršavanja.

4. Drajver preko kontrolera šalje instrukcije uređaju i obavlja zadatu operaciju.
5. Po završetku operacije generiše se prekid kojim se ponovo pokreće drajver kako bi dovršio komunikaciju.
6. Podešavaju se registri i proverava status.
7. Ažuriraju se strukture sloja softvera koji ne zavisi od uređaja.
8. Procesu se vraćaju traženi podaci.



Slika 8.19. Upravljanje ulazno-izlaznim uređajima

Pitanja

8.1. Koji su glavni zahtevi koje operativni sistemi treba da zadovolje kako bi upravljanje ulazno-izlaznim uređajima bilo što bolje?
8.2. Na koji način korisnik može da pristupi hardveru?
8.3. Kakvu hijerarhiju ima upravljanje uređajima?
8.4. Prodiskutovati o karakteristikama savremenih hardverskih komponenti računarskog sistema.
8.5. Kako se na osnovu načina na koji se prenose podaci mogu podeliti ulazno-izlazni uređaji?
8.6. Na koji način se uređaji mogu povezati na računarski sistem?
8.7. Šta su magistrale i koja je njihova uloga u računarskim sistemima?
8.8. Koje vrste magistrala obično postoje u okviru računarskog sistema?
8.9. Šta su kontroleri?
8.10. Koje registre obično imaju kontroleri i čemu oni služe?
8.11. Na koji način procesor komunicira sa ulazno-izlaznim uređajima?
8.12. Koji su osnovni zahtevi kada je upravljanje ulazno-izlaznim uređajima u pitanju?
8.13. U kojem režimu se obavljaju ulazno-izlazne operacije?
8.14. Šta je memorijsko mapiranje registara?
8.15. Prodiskutovati prednosti i mane memorijskog mapiranja registara.
8.16. Objasniti način na koji funkcioniše tehnika prozivanja.
8.17. Koje su karakteristike tehnike prozivanja?
8.18. Na koji način se korišćenjem prekida upravlja ulazno-izlaznim operacijama?
8.19. Čemu služi prioritetni prekidni kontroler (PIC)?
8.20. Kakve linije za slanje signala prekida obično postoje?
8.21. U kakvim situacijama se koristi maskirajuća linija za slanje signala prekida?
8.22. Čemu služi tabela prekida?
8.23. Na koji način funkcioniše mehanizam prekidnih prioritetnih nivoa?
8.24. Analizirati prekide na savremenim računarskim sistemima.

8.25. Objasniti principe na kojima funkcioniše direktan memorijski pristup – DMA.
8.26. Od čega se sastoje DMA kontroleri?
8.27. Da li procesor može da pristupa memoriji u trenucima dok DMA kontroler to radi?
8.28. Šta je osnovna uloga drajvera?
8.29. Prodiskutovati režime rada u kojima se mogu izvršavati drajveri.
8.30. Na koje načine se mogu klasifikovati uređaji u savremenim računarskim sistemima?
8.31. Kako se na osnovu načina upotrebe mogu podeliti ulazno-izlazni uređaji?
8.32. Koji su glavni zadaci softvera za upravljanje koji ne zavisi od uređaja?
8.33. Šta se podrazumeva pod vremenskim planiranjem ulazno-izlaznih operacija?
8.34. O čemu bi trebalo da brinu operativni sistemi prilikom planiranja ulazno-izlaznih operacija?
8.35. Šta je bafer?
8.36. Šta se podrazumeva pod baferovanjem kada su ulazno-izlazne operacije u pitanju?
8.37. Koje su odlike tehnike dvostrukog baferovanja?
8.38. Prodiskutovati prednosti koje donosi koncept zasnovan na kružnom baferu?
8.39. Navesti situacije u kojima je pogodno koristiti bafer.
8.40. Na koji način dolazi do grešaka kod ulazno-izlaznih operacija?
8.41. Na koji način se obrađuju greške pri ulazno-izlaznim operacijama?
8.42. Kakve mogu biti reakcije operativnog sistema na greške i od čega one zavise?
8.43. Koje su glavne odlike tehnike keširanja?
8.44. Gde se fizički nalazi keš memorija?
8.45. Objasniti razlike između keširanja i baferovanja.
8.46. Šta je spuler?
8.47. Koje prednosti donosi korišćenje spulera?
8.48. Navesti primer upotrebe spulera.

8.49. Koje vrste sistemskih poziva postoje?
8.50. Navesti primer blokirajućeg sistemskog poziva.
8.51. Navesti primer neblokirajućeg sistemskog poziva.
8.52. Šta su asinhroni sistemski pozivi?
8.53. Opisati način komunikacije korisničkog procesa sa hardverom.

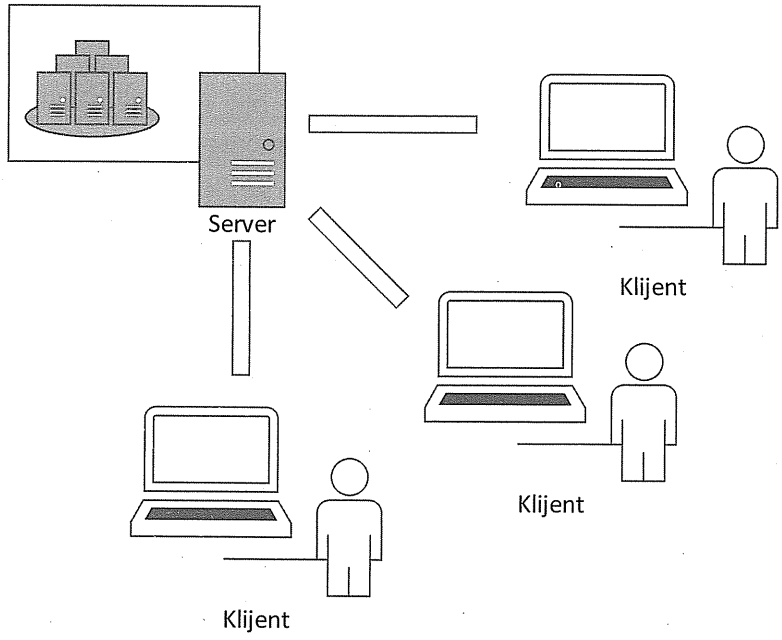
9. RAČUNARSKE MREŽE

Modul operativnog sistema koji ima zadatak da upravlja računarskim mrežama, u odnosu na ostale, poslednji je uključen u osnovne module. Razlog za to je kasnija ekspanzija računarskih mreža. Tokom prve dve decenije postojanja, računarski sistemi su bili centralizovani, obično izdvojeni u posebne sobe. Koncept računarskog centra, kao sobe sa računarom kome korisnici donose podatke za izračunavanje danas je gotovo nepoznat. Taj model je zamenjen drugim, gde puno računara obavlja posao zajedno, a podaci se dele preko mreže, ili jedan moćan računar obavlja posao, a rezultatima se pristupa preko mreže. Ovakvi modeli se primenjuju i za poslovne ili naučne potrebe. Običan korisnik, danas, provodi dosta vremena na internetu, pre svega vebu, što takođe omogućavaju računarske mreže. Iz tog razloga se teško može zamisliti operativni sistem koji ih ne podržava. Štaviše, postoje i operativni sistemi čija je jedina uloga rad sa veb aplikacijama (*Chrome OS*), pošto mnogo korisnika najveći deo vremena na računaru provodi na vebu.

Prema Tanenbamumu [10], *računarske mreže* predstavljaju skup nezavisnih računara i drugih uređaja (štampači, skeneri, eksterni hard diskovi, ruteri itd.) povezanih jedinstvenom tehnologijom koja omogućuje razmenu podataka i deljenje resursa. Uređaji na mreži se često nazivaju čvorovima. Čvorovi mogu biti povezani na različite načine: bakarnim, optičkim i koaksijalnim kablom, bežičnom vezom, radio i mikro talasima, infracrvenim zracima itd.

Potreba za masovnim umrežavanjem potekla je prvenstveno od firmi koje su na taj način želele da dele informacije i hardverske resurse. Ideja je bila da se na moćnim računarima, koji su se nazivali serveri, uskladište podaci, a da im korisnici pristupaju sa slabijih, a time i jeftinijih, računara tako što će izdavati zahteve na koje serveri odgovaraju. Ovakva organizacija sistema se naziva *klijent-server arhitektura* i danas je u širokoj upotrebi (slika 9.1.). Pored deljenja resursa, firme su imale potrebu i za efikasnom komunikacijom. Računarska mreža omogućava da se na jednostavan

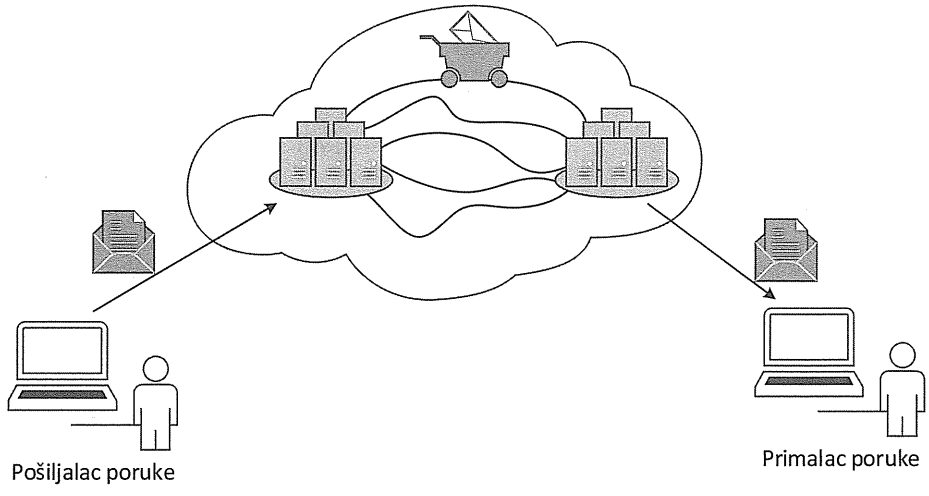
način razmeni elektronska pošta i obave telefonski i video razgovori, što značajno smanjuje troškove. Poslednjih godina elektronsko poslovanje doživljava ekspanziju, gde firme preko mreže nude svoje usluge i proizvode korisnicima.



Slika 9.1. Klijent-server arhitektura računarske mreže

Potrebe običnih korisnika, koji računare koriste kod kuće, bile su nešto drugačije. U početku su ljudi želeli da na brz i jednostavan način dođu do informacija. Ekspanzija mreža prouzrokovala je pad tiraža štampanih medija i rast broja korisnika koji elektronskim putem pristupaju informacijama. Javila se potreba i za razmenom „znanja" pa su tako nastale elektronske enciklopedije. Komunikacija i razmena poruka u realnom vremenu su takođe pogodnosti koje su doprinele ekspanziji računarskih mreža kada su obični korisnici u pitanju (slika 9.2.). Na primer, veliku popularnost doživeo je program *Talk* pomoću kojeg su korisnici mogli da razmenjuju poruke. Vremenom se razvio veliki broj programa ovakvog tipa što je na kraju dovelo do pojave društvenih mreža. Pored komunikacije, društvene mreže predstavljaju vid zabave, u šta još spada razmena audio i video materijala, televizijskih i radio programa itd. U

poslednje vreme raste popularnost elektronske trgovine. Sa povećanjem bezbednosti mreža, raste i broj korisnika koji elektronskim putem plaćaju račune i upravljaju svojim bankovnim računima.



Slika 9.2. Razmena poruka

9.1. KLASIFIKACIJA MREŽA

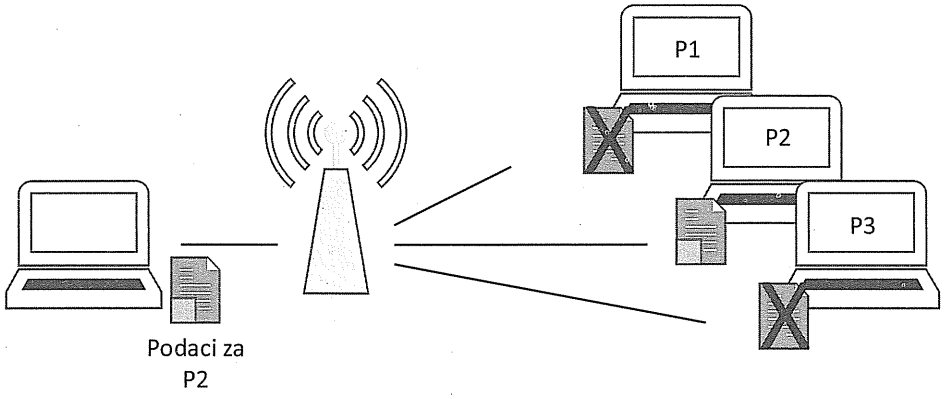
Zvanično prihvaćena podela mreža ne postoji. Ipak, sve mreže se mogu podeliti na osnovu više kriterijuma od kojih su najvažnija dva: tehnologije prenosa i veličine.

9.1.1. Klasifikacija na osnovu tehnologije prenosa

Na osnovu tehnologije prenosa mreže se mogu podeliti na *emisione mreže* (*broadcast networks*) i *mreže „od tačke do tačke"* (*point-to-point networks*).

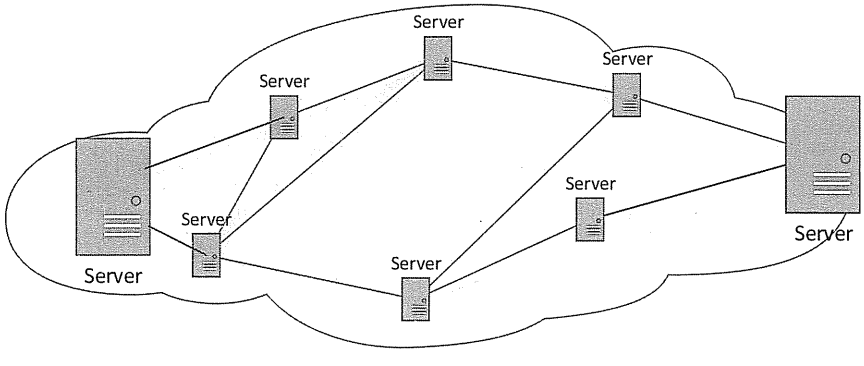
Kod *emisionih mreža* pretpostavka je da su svi računari koji učestvuju u komunikaciji povezani na jedinstveni komunikacijski kanal. Ovaj kanal služi za razmenu poruka koje može da šalje bilo koji računar, a te poruke

primaju svi ostali računari. Svaka poruka ima adresno polje u kome se navodi primalac. Kada poruka stigne do računara, računar najpre proverava adresno polje pa ukoliko je poruka njemu namenjena, pristupa se obradi te poruke, inače se poruka zanemaruje. Bežična mreža je tipičan primer emisione mreže (slika 9.3.).



Slika 9.3. Emisiona mreže

Mreže od tačke do tačke se zasnivaju na vezama između parova računara (slika 9.4.). Ukoliko ne postoji direktna veza između računara pošiljaoca i računara primaoca, tokom svog puta poruka prolazi kroz jedan ili više čvorova – tačaka. Obično postoji više različitih mogućnosti za putanju poruke, pa je pronalaženje optimalne jedan od problema matematičke optimizacije.



Slika 9.4. Mreže „od tačke do tačke"

9.1.2. Klasifikacija na osnovu veličine

Na osnovu veličine mreže se dele na: lične, lokalne, gradske i regionalne mreže. Povezivanjem dve ili više mreža dobija se *kombinovana mreža* (*internetwork*). *Internet* je primer najveće kombinovane mreže.

Lična mreža (*Personal Area Network – PAN*) je računarska mreža koja služi za komunikaciju između računara i drugih uređaja. Neki od uređaja koji se mogu naći u PAN-u su računari, štampači, skeneri, telefoni itd. Domet lične mreže je obično do 10 metara. Na primer, povezivanjem prenosivog računara i mobilnog telefona obrazuje se PAN. Često se to povezivanje ostvaruje bežičnim putem i tada se ova mreža naziva *bežični PAN* (*Wireless PAN – WPAN*).

Lokalna mreža (*Local Area Network – LAN*) je mreža koja povezuje računare i uređaje na ograničenom području poput kuće, škole, zgrade itd. Ova mreža je pogodna za deljenje resursa kao što su dokumenti, štampači, a koristi se i za igre. Mnoge LAN mreže povezuju računare sa radnim stanicama (*workstations*). Radne stanice su posebni računari sa boljim performansama koji obično izvršavaju vremenski i prostorno zahtevne procese. Oni se često koriste za čuvanje velikih količina podataka.

LAN mreže su ograničenih dimenzija što olakšava njihovo projektovanje, povezivanje i obično omogućava veoma brz prenos podataka.

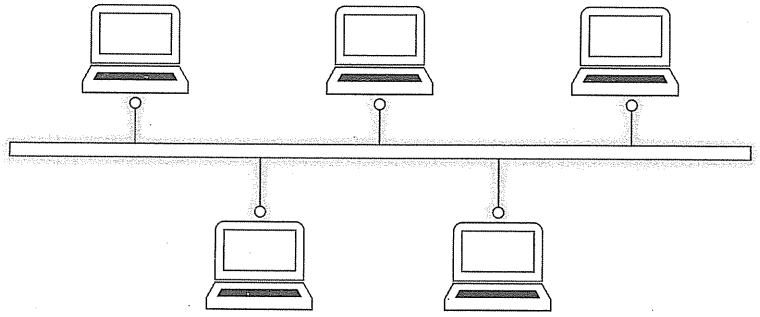
Postoje različite vrste LAN mreža koje se generalno mogu razlikovati po sledećim karakteristikama:

- *Topologija* – predstavlja prostornu uređenost mreže.
- *Protokoli* – određuju pravila i specifikacije kodiranja za razmenu podataka. Protokoli još određuju da li mreže koriste arhitekturu „od korisnika do korisnika“ (*peer-to-peer* – P2P) ili klijent-server arhitekturu. Kod P2P arhitekture mrežu čine čvorovi istog ranga (odgovornosti, sposobnosti), što se razlikuje od klijent-server arhitekture gde neki računari obezbeđuju usluge drugim.

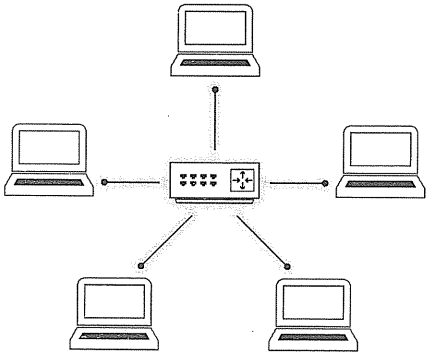
- *Tehnologija za povezivanje uređaja* može biti: bakarni, koaksijalni i optički kabl, ili bežična tehnologija. Ako se koristi bežična tehnologija, tada se radi o bežičnom LAN-u (*Wireless LAN* – WLAN).

Topologija može biti *fizička* i *logička*. *Fizička topologija* odnosi se na raspored uređaja na mreži i može biti:

- *Bus topologija* – podrazumeva da su svi uređaji povezani na jedinstveni centralni kabl (Slika 9.5.). Ova topologija je jeftina, jednostavna za održavanje i često se koristi za male LAN mreže. Na primer, Eternet (*Ethernet*) koristi ovu topologiju.



Slika 9.5. Bus topologija

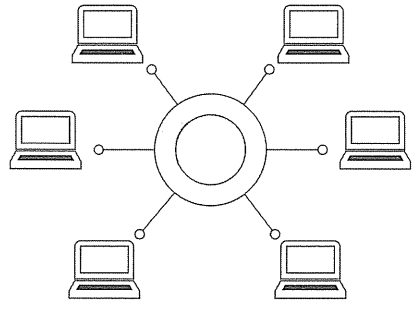


Slika 9.6. Topologija zvezde

- *Topologija zvezde* – svi uređaji su povezani na centralni uređaj (*hub*) koji ima više utičnica (portova). Slika 9.6. ilustruje mrežu

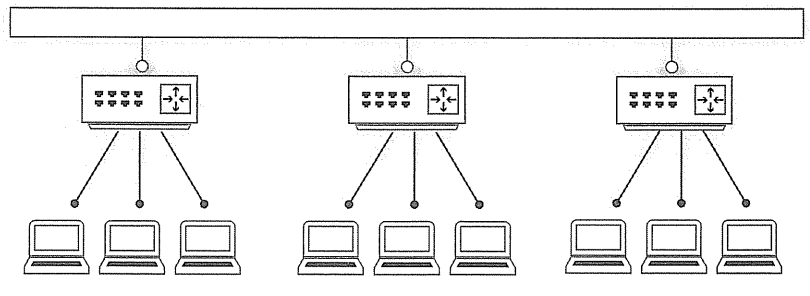
koja odgovara topologiji zvezde. Kada poruka stigne na jednu utičnicu, ona se prosleđuje kroz sve ostale utičnice. Mreža sa topologijom zvezde se relativno jednostavno implementira i održava. Prednost je i to što gubitak jednog računara ne utiče na ostatak mreže. Međutim, otkazivanje centralnog uređaja onesposobljava kompletnu mrežu, a može se javiti problem uskog grla jer sve poruke prolaze kroz centralni uređaj.

- *Topologija prstena* – svaki uređaj je povezan sa susedna dva uređaja formirajući tako zatvorenu kružnicu (slika 9.7.). Poruke putuju od čvora do čvora, a svaki čvor čita samo poruke koje su njemu namenjene. Mreže sa ovom topologijom nije jednostavno implementirati, i relativno su skupe za održavanje. Međutim, one imaju veći protok i mogu pokriti veće udaljenosti.



Slika 9.7. Topologija prstena

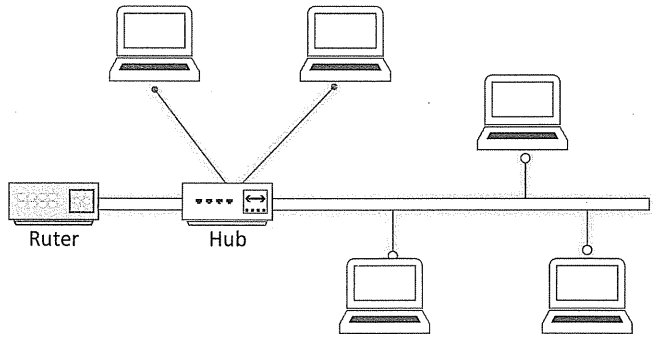
- *Topologija drveta* – kombinuje karakteristike bus topologije i topologije zvezde. Sastoji se od grupe čvorova organizovanih u topologiji zvezde koji su povezani na centralni kabl (slika 9.8.).



Slika 9.8. Topologija drveta

Logička topologija se često naziva i topologija signala. Ovom topologijom se definiše kako će podaci putovati kroz mrežu, ne uzimajući u obzir fizičku povezanost uređaja na mreži. Postoje dve vrste logičke topologije:

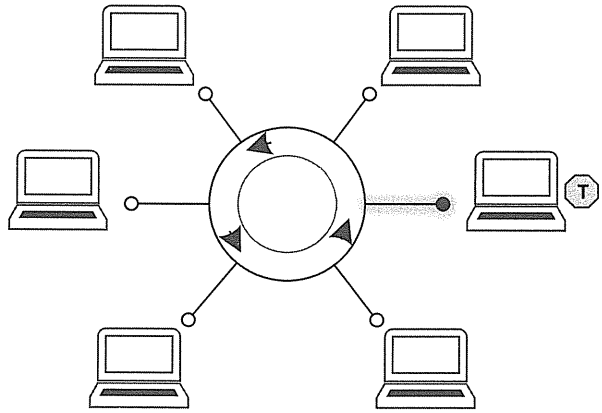
- *Topologija deljenog medija* – podrazumeva da svi čvorovi dele sve fizičke medije unutar mreže (slika 9.9.). Prednost ove topologije je u tome što čvorovi imaju neograničeni pristup svim medijima. Međutim, problem koji se često javlja je kolizija poruka. Kolizija nastaje kada dva računara u isto vreme pokušaju da emituju poruke. Postoje posebni protokoli koji služe za izbegavanje kolizije. Eternet (*Ethernet*) je primer mreže sa topologijom deljenog medija. Da bi se izbegla kolizija, svaki računar na Eternetu osluškuje mrežu i emituje poruke samo kada je mreža slobodna. Ukoliko se desi da dva računara emituju poruke u isto vreme, svaki od njih čeka određeno vreme pre nego što opet pokuša sa emitovanjem.



Slika 9.9. Topologija deljenog medija

- *Topologija žetona* (*token*) – koristi žeton pristupa (ili samo žeton) kojim određuje koji računar ima pravo da emituje poruke (slika 9.10.). Žeton putuje kroz mrežu, i ukoliko računar želi da emituje poruke, najpre mora da dohvati žeton sa mreže. Zajedno sa porukama računar šalje i žeton odredišnom računaru, koji u odgovoru taj žeton vraća nazad. Nakon emitovanja poruka, računar oslobađa žeton tako da može da ga koristi neki drugi

računar. Mreže sa topologijom žetona nemaju problem sa kolizijom poruka, jer na osnovu žetona u jednom trenutku samo jedan računar može da emituje poruke. Problem kod ovih mreža je nastanak kašnjenja, jer svaki računar mora da čeka da dobije žeton. Najčešće se mreže sa topologijom žetona implementiraju kao mreže sa fizičkom topologijom prstena. Tada žeton putuje u krug kroz sve računare. Primer takve mreže je IBM-ova *Token Ring* mreža.

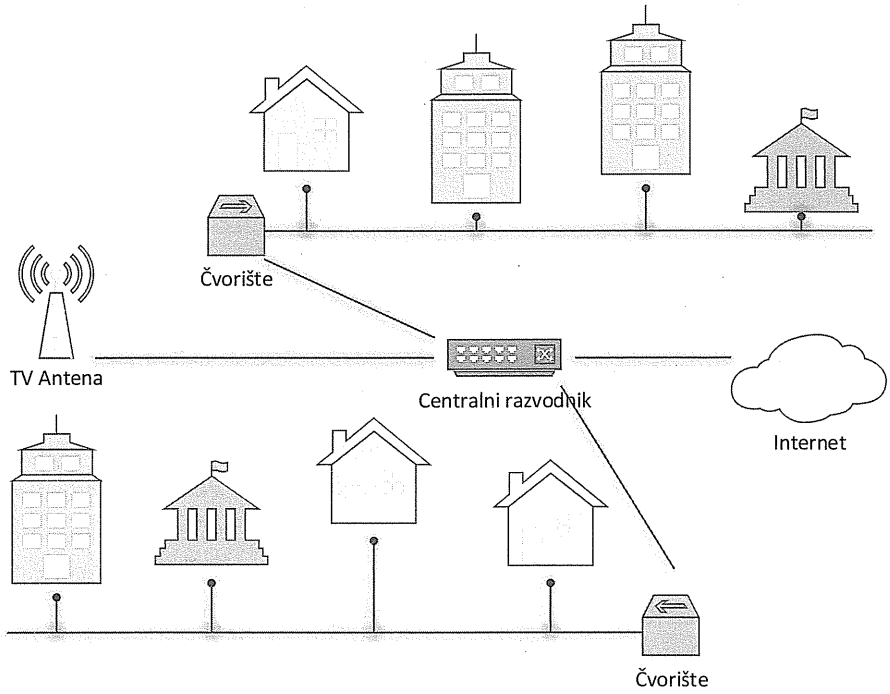


Slika 9.10. Topologija tokena

Gradska mreža (*Metropolitan Area Network – MAN*) je velika mreža koja pokriva područje veličine jednog grada ili opštine. MAN obično povezuje više lokalnih mreža pomoću optičkog kabla čime ostvaruje veoma brzu i efikasnu mrežu. MAN mreža takođe obezbeđuje i vezu ka regionalnim mrežama i internetu. Na primer, jedna kompanija može imati kancelarije na više adresa u jednom gradu. Ukoliko je potrebno omogućiti komunikaciju između računara u tim kancelarijama, potrebno je implementirati gradsku mrežu. Još jedan primer gradske mreže je mreža kablovske televizije (slika 9.11.). Ova mreža postoji u mnogim gradovima i predstavlja sistem kablova koji povezuje korisnike do centralnog razvodnika odakle se emituje televizijski i internet signal.

Regionalna mreža (*Wide Area Network – WAN*) – povezuje velike geografske oblasti kao što su države, regioni ili kontinenti. Primer WAN

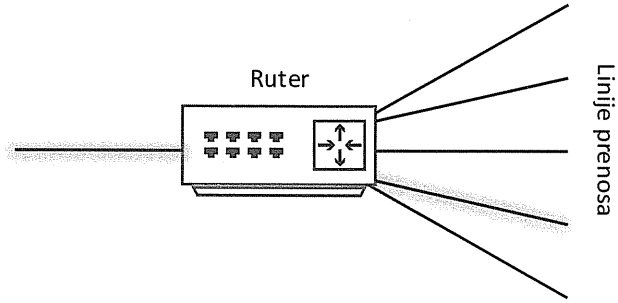
mreže je sistem rezervacija karata u vazdušnom saobraćaju, gde se terminali mogu naći širom države. Na terminalima se obavlja sama rezervacija, a oni putem WAN mreže pristupaju podacima iz centralnog računara. Regionalna mreža povezuje manje mreže u jednu celinu, a deo mreže koji obavlja to povezivanje naziva se komunikaciona podmreža, ili samo podmreža. Uloga podmreže je da omogući razmenu poruka (podataka) između dva računara koji su povezani udaljenim LAN (ili MAN) mrežama. Podmreža je izgrađena od dva osnovna elementa: prenosnih linija i rutera (mrežnih usmerivača). *Prenosne linije* služe za prenos bitova između računara i to su najčešće bakarni, koaksijalni i optički kablovi, ali i bežične veze. *Ruteri* spajaju tri ili više linija prenosa i njihov zadatak je da prispele podatke usmere dalje nekom od prenosnih linija (slika 9.12.).



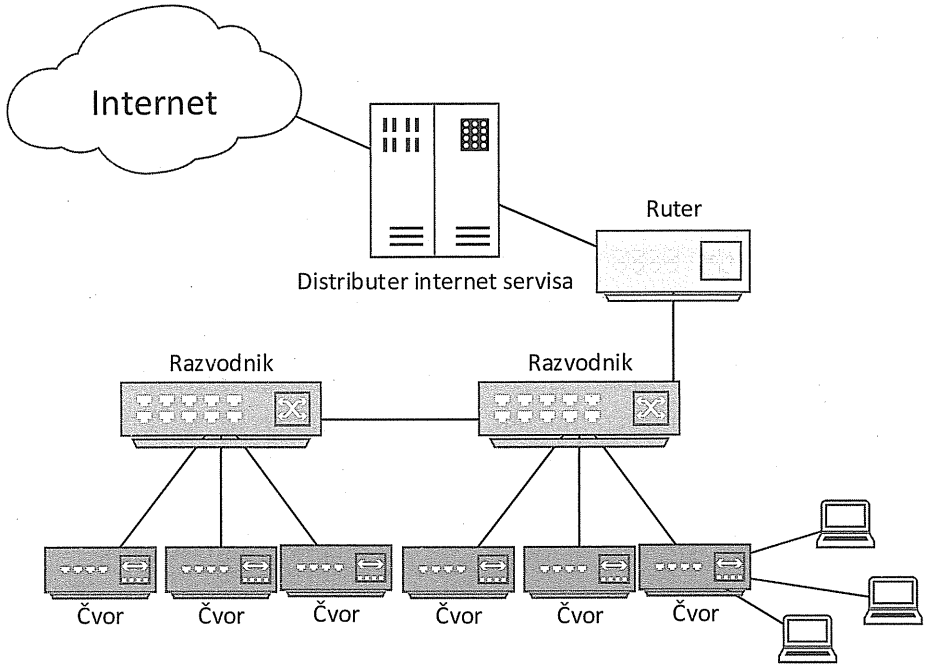
Slika 9.11. Gradska mreža – Kablovska televizija

Kombinovana mreža (*internetwork* ili *internet)* – predstavlja skup međusobno povezanih jasno razgraničenih mreža. Ova mreža povezuje manje mreže (LAN, MAN, WAN) kreirajući mrežu globalnih razmera.

Implementacija kombinovane mreže je veoma kompleksan zadatak zbog velikog broja tehnologija koje se koriste za implementaciju manjih mreža. Ipak, ruteri (*routers*), mostovi (*bridges*) i mrežni prolazi (*gateways*) su uređaji koji to omogućavaju. *Mostovi* su uređaji koji povezuju dve LAN mreže koje koriste isti protokol (npr. Eternet). *Mrežni prolazi* su uređaji koji služe da premoste različitosti u mrežnim tehnologijama. U slučaju servera koji sadrži veb stranice, mrežni prolaz je računar koji usmerava saobraćaj sa servera na spoljnu mrežu, čime omogućuje dostupnost tih stranica. Često mrežni prolaz ima ulogu *servera posrednika* (*proxy server*). U kućnoj varijanti mrežni prolaz predstavlja dobavljač internet usluga (*Internet Servise Provider* – ISP). Engleski termini *internetwork* i *internet* su nastali od fraze *interconnected network* i odnose se na kombinovanu mrežu u opštem smislu. Internet je jedan konkretan primer kombinovane mreže (slika 9.13.) , ali je naziv koji je prihvaćen na globalnom nivou i prema *Pravopisu srpskog jezika* piše se malim početnim slovom.



Slika 9.12. Ruter i linije prenosa



Slika 9.13. Internet

9.2. Istorija interneta

Sa razvojem računara pedesetih godina dvadesetog veka javila se potreba za njihovom međusobnom komunikacijom. Krajem pedesetih u Sjedinjenim Američkim Državama osnovana je agencija ARPA (*Advanced Research Projects Agency*) sa primarnim ciljem da razvija informacione tehnologije otporne na nuklearne napade. Godine 1967. predstavnici ARPA-e i stručnjaci iz privatnog sektora sastali su se sa predstavnicima Ministarstva odbrane (*Department of Defense*) Sjedinjenih Američkih Država. Tema sastanka bila je diskusija o protokolima za razmenu podataka između računara. Tako je 1969. godine nastala mreža *ARPANET* koja je koristila NCP protokol (*Network Control Protocol*).

ARPANET je trebalo da omogući razmenu podataka između vojnih centara, državnih institucija i univerziteta u SAD-u. Ova mreža je na

početku povezivala 4 lokacije (Univerzitet Kalifornija u Los Anđelesu, Univerzitet Kalifornija u Santa Barbari, Centar za istraživanja Stenford i Univerzitet Juta), ali se taj broj brzo uvećao. Ubrzo se javljaju i druge manje mreže čiji su tvorci želeli da se povežu sa ARPANET-om. Međutim, istraživanja su pokazala da protokoli ARPANET-a nisu bili pogodni za komunikaciju između različitih mreža pa je ARPA u međuvremenu nastavila istraživanja sa ciljem da napravi pogodnije protokole. Tako je nastao skup protokola *TCP/IP* (*Transmission Control Protocol – TCP, Internet Protocol – IP*).

TCP/IP je nastao 1978. godine. Prva implementacija interneta javila se 1980. kada je ARPA krenula sa menjanjem protokola ARPANET-a na TCP/IP. Uporedo sa tim, u cilju širenja novog protokola, ARPA je potpisala ugovor sa kompanijom BBN (*Bolt, Beranek, Newman*) da implementira TCP/IP protokol za svoj *Berkeley UNIX* operativni sistem (drugi naziv za ovaj operativni sistem je BSD – *Berkeley Software Distribution*). Slični ugovori potpisani su i sa ostalim vodećim kompanijama (kao što su IBM i HP) koje su implementirale TCP/IP na svojim platformama. Prva verzija BSD-a sa TCP/IP protokolom objavljena je 1983. godine. Iste godine ARPA je završila zamenu protokola na svim mašinama u ARPANET-u i objavila da svi uređaji koji žele da se priključe na ARPANET moraju koristiti TCP/IP.

Tokom 1980-tih nove i postojeće mreže (naročito LAN mreže) povezivane su sa ARPANET-om. Da bi se omogućila komunikacija između mašina, dodeljivani su im određeni brojevi koji su predstavljali njihove adrese unutar mreže tzv. IP adrese (videti deo 9.5.1). Sa porastom mreže, pronalaženje konkretne mašine u njoj nije bilo jednostavno zbog velikog broja adresa. Zato se uporedo sa adresama (brojevima) mašine označavaju i nazivima, a napravljen je i DNS sistem (*Domain Name System*) koji je zadužen za konverziju naziva računara u njegovu adresu. Od tada, DNS se razvio u generalizovanu distribuiranu bazu za čuvanje raznih podataka vezanih za nazive mašina.

Povezivanje sa ARPANET-om nije bilo jednostavno. Da bi se neka mreža povezala sa ARPANET-om morala je da ima odobrenje od Ministarstva

odbrane SAD-a, a mnoge to nisu imale. Organizacija NFS (*National Science Foundation*) je uvidela koliko je uticaj ARPANET imao na naučna istraživanja pa je odlučila da napravi glavnu (*backbone*) mrežu koja će povezati 6 super-računarskih centara: San Dijego, Boulder, Šampanj, Pitsburg, Itaka i Prinston. Pored svakog super-računara nalazio se jedan manji računar. Ti manju računari su koristili TCP/IP, bili su međusobno povezani i formirali su podmrežu. Pored šest super-računarskih centara, NFS je u tu mrežu uključio i mnoge regionalne mreže. Kompletna mreža, uključujući glavnu i regionalne, nazvana je NSFNET i ona je bila povezana sa ARPANET-om.

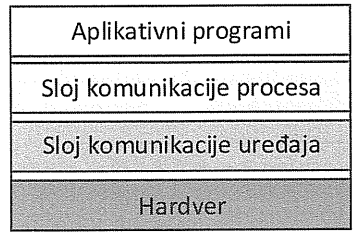
Tokom 1990-tih mnoge druge države i regioni su formirali slične mreže kao što su ARPANET i NSFNET. Neke od njih u Evropi su EuropaNET i EBONE. Povezivanjem tih mreža nastao je internet koji se brzo razvijao od samog početka. Pravu ekspanziju internet je doživeo pojavom veba (*World Wide Web*) ranih devedesetih.

9.3. Arhitektura mreže

U prethodnom delu bilo je reči o fizičkim karakteristikama računarskih mreža na osnovu kojih se može ustanoviti postojanje različitih načina i tehnologija za implementaciju mreže. Od mreža se zahteva da obezbede brzu, efikasnu i robusnu vezu između velikog broja uređaja. Potrebno je da mreže poseduju određenu fleksibilnost kako bi se obezbedila mogućnost tehnološkog unapređenja i fizičkog proširenja. Takođe, mora se voditi računa o tome da će mreže održavati ljudi različitih nivoa stručnosti.

Dizajniranje mreže, koja zadovoljava sve ove uslove, nije jednostavan zadatak. Jedno od rešenja je apstrakcija mreže u više slojeva u cilju smanjenja kompleksnosti zadatka (slika 9.14.). Osnovna ideja ovog pristupa je da se usluge koje nudi mrežni hardver organizuju kao jedan (najniži) sloj i da se na njega dalje nadovežu viši slojevi, gde svaki viši sloj predstavlja viši stepen apstrakcije. Najniži sloj se sastoji od funkcionalnosti koje nudi mrežni hardver. Sloj iznad njega koristi te

funkcionalnosti i sloju iznad obezbeđuje mogućnost konekcije između dva uređaja. Na osnovu tih mogućnosti sledeći sloj kreira servis koji ostvaruje međuprocesnu komunikaciju. Na samom vrhu nalazi se aplikacija koja se na osnovu sloja ispod povezuje sa procesom na udaljenom računaru. Važno je napomenuti da broj slojeva nije fiksiran i zavisi od konkretne implementacije mreže. Takođe, apstrakcija mreže u više slojeva nije jedino moguće rešenje problema implementacije mreže, ali jeste najčešće korišćeno. Neki od razloga česte primene ovog pristupa su modularnost i učaurivanje (enkapsulacija) koji upravo proističu iz apstrakcije mreže u više slojeva. Modularnost omogućuje menjanje (poboljšanje) ili čak zamenu pojedinačnog sloja bez uticaja na ostale slojeve. Ukoliko su potrebne izmene u mreži, jednostavnije je to uraditi na jednom sloju. Enkapsulacija se odnosi na činjenicu da su detalji implementacije sloja skriveni unutar njega, tj. nisu vidljivi izvan sloja.

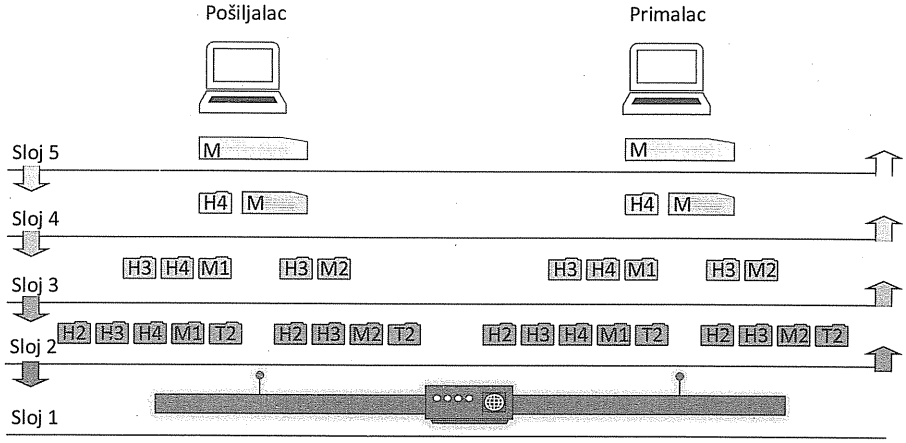


Slika 9.14. Primer slojeva

Svaki sloj definiše dva interfejsa. Prvi je *servisni interfejs* i on obuhvata sve operacije koje taj sloj nudi sloju iznad sebe. Drugi interfejs definiše komunikaciju između dva ista sloja na različitim mašinama i naziva se *protokol sloja*. Jedan sloj može imati više protokola koji definišu različite načine komunikacije. *Arhitekturu mreže* čini skup slojeva i protokola. *Stek protokola* je niz koji se sastoji od po jednog protokola po sloju počevši od najvišeg pa do najnižeg sloja.

Komunikacija između dva procesa (aplikacije) na umreženim računarima počinje tako što prvi proces prosleđuje poruku najvišem nivou mreže (slika 9.15.). Najviši nivo obrađuje poruku (npr. deli je na manje poruke, kompresuje) i prosleđuje je nivou ispod sebe, koji nastavlja ovaj proces sve do najnižeg nivoa. Najniži nivo je taj koji fizički prenosi poruku do

drugog računara. Svaki nivo na dospelu poruku dopisuje zaglavlje koje sadrži potrebne podatke (npr. redni broj poruke, vreme slanja, ime protokola koji se koristi itd.). Kada poruka stigne na određenu destinaciju, ona prolazi kroz sve slojeve mreže, ali u obrnutom redosledu. Svaki sloj čita zaglavlje i koristi informacije koje su u njemu zapisane, nakon čega ga odstranjuje i sloju iznad prosleđuje podatke. Kada poruka stigne do procesa na vrhu, kome je poslata, sva zaglavlja su odstranjena i proces dobija originalnu poruku.



Slika 9.15. Proces slanja poruke

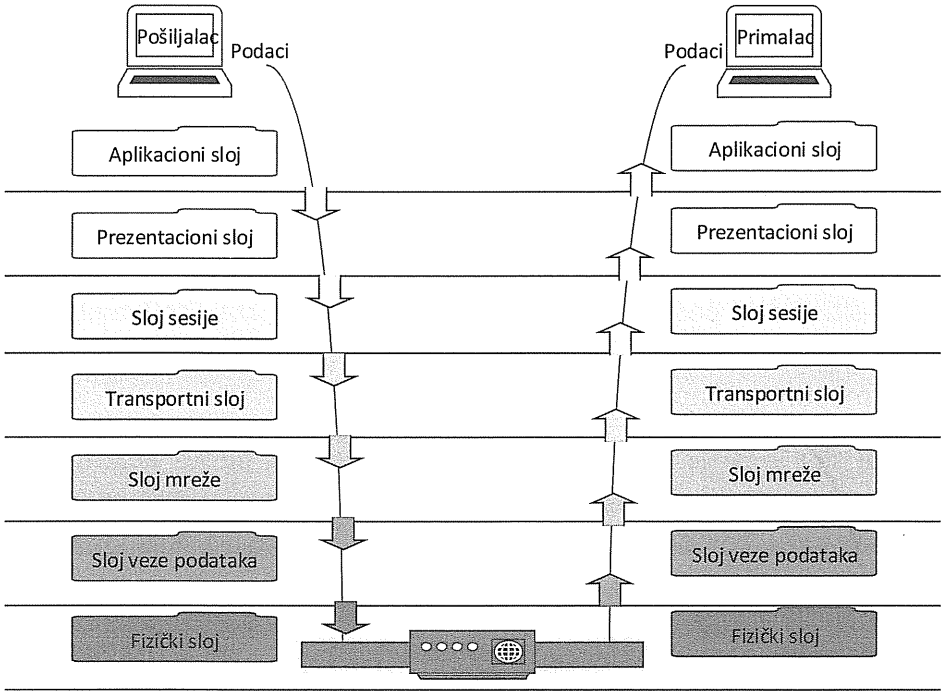
9.4. Referentni model ISO OSI

Prva standardizovana arhitektura mreže predložena je od strane ISO organizacije (*International Organization for Standardization*) 1983. godine. Ova arhitektura je nazvana *OSI* (*Open Systems Interconnection*) i sastoji se od sedam slojeva (slika 9.16.).

Na samom vrhu se nalazi *sloj aplikacija*. On obuhvata različite protokole među kojima je i HTTP (*Hyper Text Transfer Protocol*). HTTP je osnovni protokol za komunikaciju na vebu (*World Wide Web* – WWW). Prisutni su još i sledeći protokoli: DNS, FTP, SMTP, POP3.

Sloj prezentacije je zadužen da obezbedi nezavisnost od načina predstavljanja podataka. Na primer, na ovom sloju se odlučuje o tome koliko će se bitova koristiti za predstavljanje celih brojeva (16, 32 ili 64), ili kako se formatira video tok (*video stream*).

Ispod sloja prezentacije nalazi se *sloj sesije* koji je zadužen za uspostavljanje sesija. Sesija je niz interakcija između dva učesnika u komunikaciji tokom jedne konekcije. Ovaj sloj obezbeđuje različite servise, kao što su kontrola dijaloga i sinhronizacija. U kontroli dijaloga odlučuje se koji je učesnik u komunikaciji na redu. Sinhronizacija se koristi u slučaju pada mreže tokom razmene velikih količina podataka. Kada se mreža oporavi od pada, sinhronizacijom se uspostavlja konzistentno stanje u komunikaciji. Tada se komunikacija nastavlja gde je stala i ne mora da počne od samog početka.



Slika 9.16. Referentni model ISO OSI

Transportni sloj realizuje međuprocesni kanal i upravlja komunikacijom između dva procesa na umreženim mašinama između kojih se može naći različit broj čvorova. Niži slojevi komuniciraju sa direktnim susedima u mreži i implementirani su u svim čvorovima mreže. Podaci koji se razmenjuju na nivou transportnog sloja nazivaju se poruke (*messages*). Na ovom sloju poruke se dele na manje pakete (*packets*). Paketi su posebno formatirane celine koje se sastoje iz dva dela. Jedan deo su korisnički podaci, dok drugi deo predstavljaju kontrolni podaci. Paketi se u određenom redosledu prosleđuju nižim slojevima, a oni na osnovu kontrolnog dela prenose paket do željene destinacije. Transportni sloj obezbeđuje da svi paketi stignu u odgovarajućem redosledu od pošiljaoca do primaoca poruke.

Sloj mreže je zadužen za usmeravanje (rutiranje) paketa između čvorova u mreži. Problem mrežnog sloja je nastanak „uskih grla" koji se javlja kada se u podmreži nađe veliki broj paketa. Zajedno sa višim slojevima, sloj mreže je odgovoran i za upravljanje zagušenjem u mreži.

Sledeći sloj je sloj *veze podataka*. On prihvata pakete od višeg sloja, a zatim te pakete organizuje u okvire podataka (*data frames*) koje prosleđuje fizičkom sloju. Veličina okvira podataka može biti od nekoliko stotina do nekoliko hiljada bajtova. Osnovni problem ovog sloja je kako usaglasiti brzog pošiljaoca i sporog primaoca poruka. Kod emisionih mreža javlja se problem kontrole deljenog kanala. Ovaj sloj se najčešće implementira u adapterima mreže (*network adapters*).

Najniži sloj je *fizički sloj* koji je zadužen za prenos bitova kroz mrežu. Problemi ovog sloja su elektronske prirode, kao npr. kako uspostaviti početnu vezu, koji signali će se koristiti za predstavljanje nule i jedinice, koliko ti signali traju, kako se završava komunikacija itd.

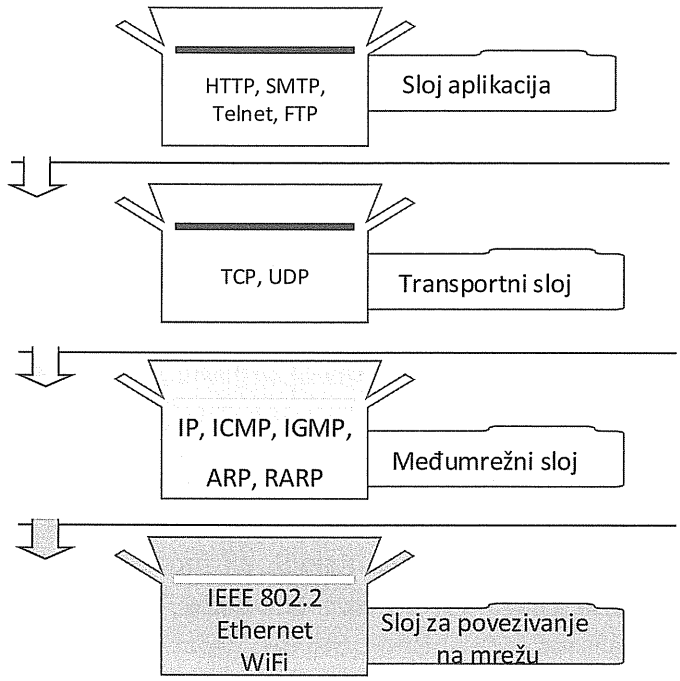
9.5. Referentni model TCP/IP

Za razliku od OSI modela koji je predložen od strane ISO organizacije kao standard, referentni model *TCP/IP* je imao drugačiji redosled u svom

razvoju – najpre su nastali protokoli, a zatim je izvršena standardizacija. TCP/IP je nastajao u više iteracija i istorija njegovog razvoja prepliće se sa istorijom interneta. Zapravo, ovaj model je omogućio nastanak interneta.

TCP/IP je dobio ime po svoja dva najvažnija protokola: *TCP* (*Transmission Control Protocol*) i *IP* (*Internet Protocol*). Osnovni cilj modela bio je da omogući komunikaciju unutar kombinovane mreže (*internetwork* ili *internet*). Svaka mreža ima svoj komunikacioni interfejs. TCP/IP predstavlja apstrakciju komunikacionih mehanizama svih mreža tako što sakriva njihove specifičnosti i obezbeđuje jedinstveni interfejs nezavisan od fizičke mreže.

TCP/IP model se sastoji od 4 sloja (slika 9.17.): sloj aplikacija, transportni sloj, međumrežni sloj i sloj za povezivanje na mrežu.



Slika 9.17. Prikaz referentnog modela TCP/IP

Kod TCP/IP modela najviši sloj je *sloj aplikacija*, dok slojevi prezentacije i sesije ne postoje. Same aplikacije su zadužene za predstavljanje podataka

i upravljanje sesijama. Sloj aplikacija sadrži razne protokole višeg nivoa, kao što su: HTTP, FTP, TELNET, SMTP, POP3, DNS.

Srž *transportnog sloja* kod TCP/IP modela su protokoli TCP i UDP. *TCP* (*Transmission Control Protocol*) je pouzdan protokol, tj. garantuje prenos podataka bez greške. Podatke dobijene iz sloja aplikacija TCP deli na pakete određene veličine koje prosleđuje međumrežnom sloju. Za svaki paket se očekuje potvrda o pristizanju. Ukoliko potvrda ne stigne u određenom periodu, ponavlja se slanje paketa. Pored toga, TCP podržava kompresiju podataka, kontrolu zagušenja i kontrolu toka. Zbog svojih osobina TCP je najčešće korišćeni protokol transportnog sloja. *UDP protokol* (*User Datagram Protocol*) je daleko jednostavniji, ali zato ne garantuje pouzdan prenos. Obično se UDP koristi u situacijama kada brzina prenosa podataka ima prednost u odnosu na tačnost (određenog broja) bitova, kao npr. kod video prenosa.

Međumrežni sloj je zadužen za prenos paketa na željenu destinaciju, pri čemu paketi putuju nezavisno. Zbog toga se može desiti da paketi ne stignu u redosledu kojim su poslati. Viši slojevi su zaduženi za pravilno raspoređivanje pristiglih paketa. Osnovni protokol ovog sloja je IP (*Internet Protocol*) koji je odgovoran za rutiranje paketa. IP je protokol bez uspostavljanja konekcije, nije pouzdan, ne podržava kontrolu toka i oporavak od greške. Ukoliko je to potrebno, ove funkcionalnosti moraju biti pokrivene u višim slojevima. Ostali protokoli ovog sloja su: ICMP, IGMP, ARP, RARP.

Najniži sloj TCP/IP modela je *sloj za povezivanje na mrežu*. Zapravo, to i nije pravi sloj, već je to skup interfejsa postojećih mreža. TCP/IP model ne definiše nikakav protokol na ovom sloju, ali može koristiti gotovo sve poznate interfejse mreža kao što su: IEEE 802.2, Ethernet, WiFi, Token Ring mreža itd.

9.5.1. IP adresiranje i rutiranje

Kombinovana mreža (*internetwork* ili *internet*) predstavlja kolekciju međusobno povezanih različitih mreža. *IP protokol* je zaslužan za

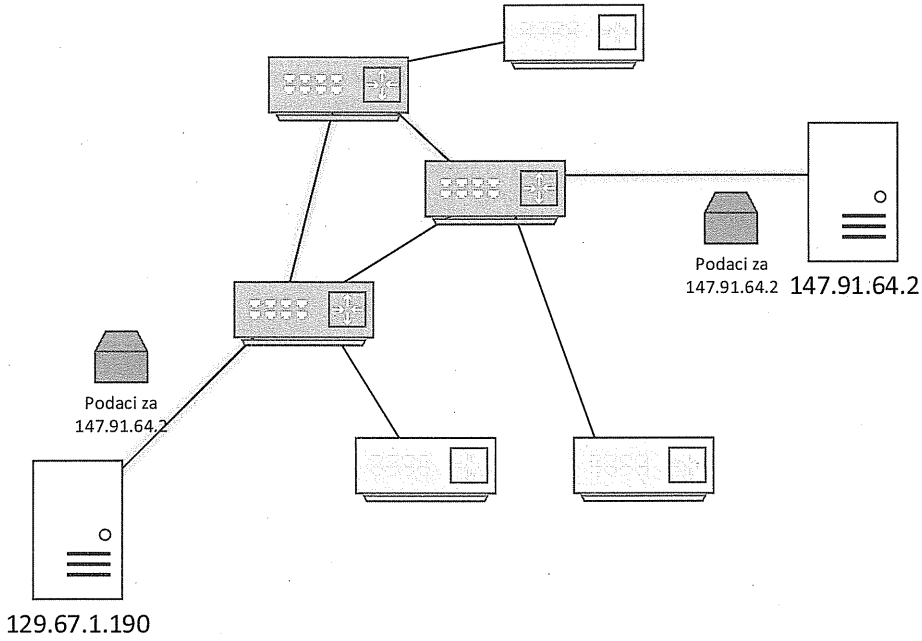
sakrivanje osobine fizičke mreže čime kreira prividno jedinstvenu mrežu. Podaci koji se razmenjuju IP protokolom nazivaju se IP datagrami. *Datagrami* su poruke specifičnog formata i sastoje se od dva osnovna dela – zaglavlja i tela datagrama. Zaglavlje datagrama sadrži podatke kao što su:

- Verzija IP protokola – 4 bita, moguće vrednosti su IPv4 i IPv6;
- Veličina zaglavlja u 32-bitnim rečima – 4 bita;
- Ukupna veličina datagrama u bajtovima – 2 bajta, pa je najveća veličina datagrama 65535 bajtova;
- Protokol transportnog sloja – 1 bajt, a neke od mogućih vrednosti su TCP ili UDP;
- IP adresa pošiljaoca datagrama;
- IP adresa primaoca datagrama itd.

IP adresa služi za identifikovanje (lociranje) svakog računara na mreži. Da bi računar bio dostupan na mreži, mora imati svoju IP adresu. Mašine koje su povezane na više različitih mreža imaju po jednu IP adresu za svaku mrežu. Prema tome, ruteri imaju više IP adresa jer su povezani na više mreža. Da bi ruter mogao da pošalje poruku odgovarajućoj mreži, on mora da zna na osnovu IP adrese (tj. njenog mrežnog dela) kroz koju „žicu" treba dalje proslediti poruku. Svaki ruter u sebi sadrži tabelu rutiranja (*routing table*) koja mu obezbeđuje ove podatke. Prilikom slanja poruka na veće udaljenosti, one najčešće prolaze kroz veći broj rutera (slika 9.18.). Tom prilikom svaki ruter određuje dalji put poruke na osnovi njene IP adrese i svoje tabele rutiranja. Proces određivanja putanje poruke od pošiljaoca do primaoca naziva se IP rutiranje.

Postoje različite vrste IP adresa od kojih su najznačajnije: *IPv4* i *IPv6*. IPv4 adresa je 32-bitni neoznačen ceo broj. Dužina IPv6 adrese je 16 bajtova, a u nastavku ove sekcije adresiranje i rutiranje će biti predstavljeno sa IPv4 adresama. Zapis IP adrese se sastoji od 4 decimalna cela broja koji su razdvojeni tačkama. Svaki od tih brojeva predstavlja jedan bajt 32-bitne adrese što znači da može imati vrednost od 0 do 255. Primer jedne validne IP adrese je 147.91.66.10. Ovaj zapis se koristi radi lakše čitljivosti, dok

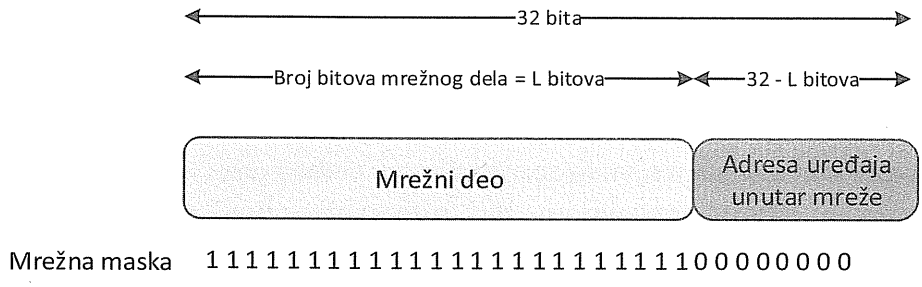
računari koriste bitovski zapis. Primer još čitljivijeg zapisa je *myhost.example.com*. Iza svakog ovakvog zapisa nalazi se IP adresa, a DNS (*Domain Name System*) serveri su zaduženi za njihovu međusobnu konverziju.



Slika 9.18. IP rutiranje

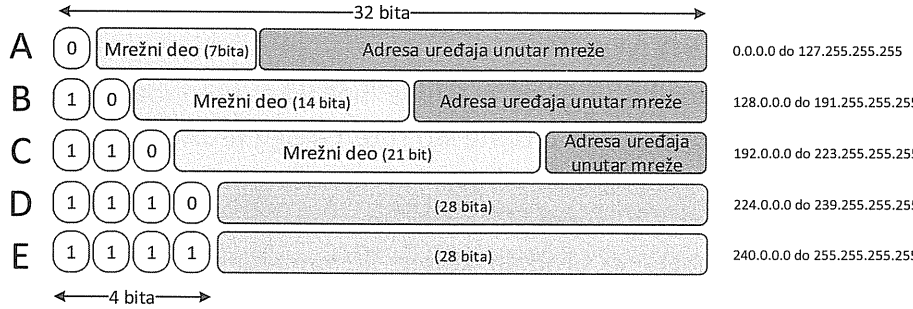
IP (IPv4) adresa se sastoji od dva dela. Prvi je mrežni deo i određuje adresu čitave mreže. Drugi deo se odnosi na adresu uređaja unutar mreže. To znači da sve mašine koje su povezane na jednu mrežu imaju isti mrežni deo. Posebne RIR (*Regional Internet Registries*) organizacije se bave dodeljivanjem mrežne adrese mrežama u različitim delovima sveta. Broj bitova mrežnog dela u IP adresi varira i ne može se dobiti iz same adrese, pa su protokoli rutiranja zaduženi za prenos ove informacije. Često se prilikom zapisa IP adrese ističe veličina mrežnog dela tako što se na zapis adrese nadoveže „/L“, gde je L broj bitova mrežnog dela. Na primer, zapis 147.91.66.10/16 označava da prvih 16 bitova navedene adrese predstavlja njen mrežni deo. Izdvajanje mrežnog dela iz IP adrese omogućuje mrežna maska (*subnet mask*). To je 32-bitni broj kod koga L

viših (prvih) bitova ima vrednost 1, dok ostali bitovi imaju vrednost 0. Kada se izvrši bitovska konjunkcija adrese i mrežne maske dobija se mrežni deo IP adrese (slika 9.19.).



Slika 9.19. Mrežna maska

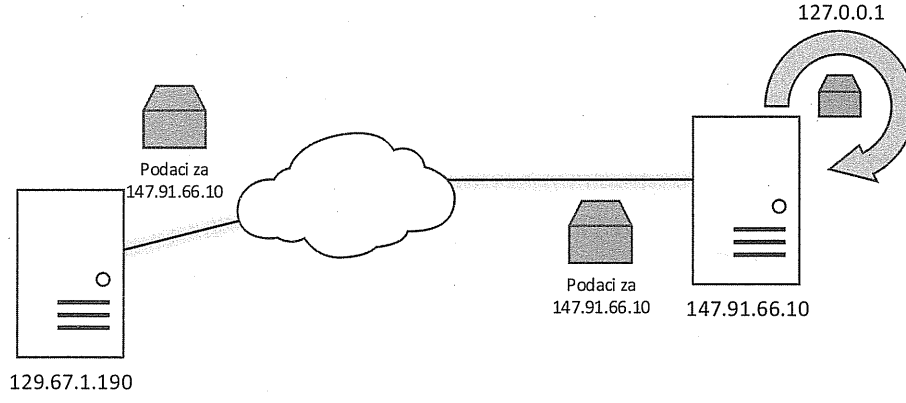
Postoji pet klasa IP adresa i one su prikazane na Slici 9.20. Klase A, B i C mogu imati respektivno 128 mreža sa (oko) 16 miliona uređaja, 16 384 mreža sa 65 536 uređaja, i (oko) 2 miliona mreža sa 256 uređaja. Klasa D je rezervisana za višesmerno slanje datagrama (*multicasting*), dok su adrese iz klase E rezervisane za istraživanja u budućnosti.



Slika 9.20. Klase IP adresa

Neke vrednosti mrežnog dela i dela uređaja IP adrese imaju specijalno značenje. Ukoliko deo IP adrese uređaja čine samo nule, onda adresa referiše na tekući uređaj. Sve nule u mrežnoj adresi označavaju tekuću mrežu. Adresa koja se sastoji od svih jedinica (255.255.255.255) referiše na sve mašine lokalne mreže. Ukoliko mrežni deo adrese sadrži adresu neke udaljene mreže, a ostatak adrese se sastoji samo od jedinica, onda ta adresa označava sve uređaje udaljene mreže. Na primer, adresa

147.91.255.255 je adresa koja referiše na sve uređaje povezane na mreži 147.91. Adresa 127.0.0.1 je specijalna adresa rezervisana za upotrebu na svakom računaru (slika 9.21.). Ona služi da označi lokalni računar. Kada računar šalje poruke na adresu 127.0.0.1, poruke ne dospevaju do lokalne mreže. One se automatski preusmeravaju na isti (lokalni) računar od strane njegovog mrežnog adaptera (mrežne kartice).



Slika 9.21. IP adresiranje

Hijerarhijska organizacija IP adresa ima nekoliko prednosti i mana. Osnovna prednost se ogleda u tome što ruteri mogu prosleđivati poruke samo na osnovu mrežnog dela adrese. Na taj način rutiranje je značajno olakšano. Kada poruka stigne do određene mreže, na osnovu adrese uređaja biva dostavljena uređaju kojem je namenjena. Mana ovakve organizacije je u tome što IP adrese uređaja zavise od mreže na kojoj se nalazi uređaj. Problem može nastati kod bežičnih mreža kada pokretni uređaji menjaju mrežu na koju su povezani a potrebno je da zadrže svoju IP adresu. Druga mana se odnosi na veliki broj neiskorišćenih IP adresa. Česta je pojava da je na mrežu povezano značajno manje uređaja nego što može, što nije poželjno jer se internet razvija velikom brzinom i odavno je bilo jasno da će u budućnosti biti problema sa manjkom IP adresa.

Da bi se IP adrese bolje iskoristile, uvodi se koncept IP podmreže (*IP subnet*). Ukoliko je jednoj mreži dodeljeno više IP adresa nego što joj je potrebno, skup tih adresa se može podeliti na više podskupova. Mreža se može organizovati kao skup više podmreža (*subnetworks*), pri čemu se svakoj dodeljuje jedan podskup adresa. Ideja je da se deo adrese uređaja podeli na dva dela: podmrežnu adresu i adresu uređaja. Sada je IP adresa oblika *<mreža><podmreža><uređaj>*. Podmrežni deo i deo uređaja čine lokalni deo IP adrese. Podela u podmreže se obavlja lokalno a broj bitova za određivanje podmreže može biti proizvoljan. Uređaj u mreži podeljenoj u podmreže je svestan te podele, dok uređaj u udaljenoj mreži podeljenu mrežu i dalje vidi kao jedinstvenu, a lokalni deo IP adrese kao adresu uređaja unutar mreže.

Podela IP adresa na podskupove se obavlja na sledeći način. Neka je mrežna adresa 147.91. Tada su sve adrese te mreže (njih 65 536) oblika:

10010011. 01011011.xxxxxxxx.yyyyyyyy/16

Ukoliko je potrebno podeliti mrežu na dve podmreže, adrese se mogu podeliti u dva skupa od po 32 768 adresa na sledeći način:

10010011. 01011011.0xxxxxxx.yyyyyyyy/17,
10010011. 01011011.1xxxxxxx.yyyyyyyy/17.

Ukoliko je potrebno dalje deljenje mreže, postupak se ponavlja. Na primer, mrežu treba podeliti na tri podmreže. To se može uraditi na sledeći način:

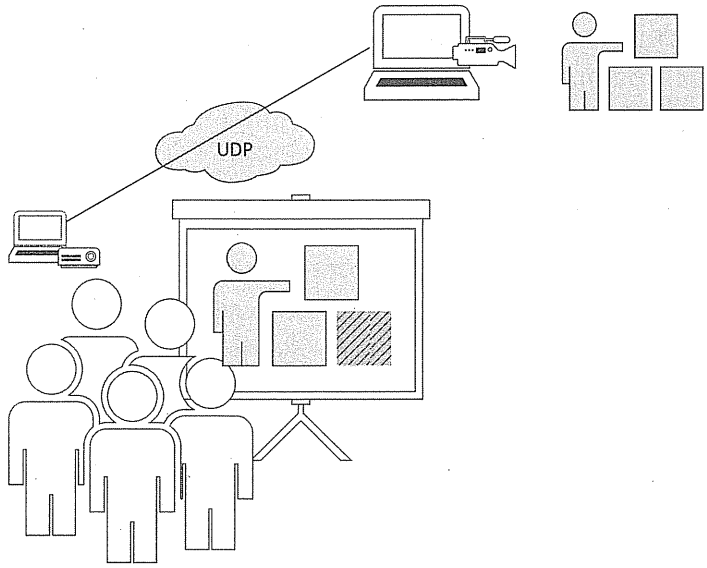
10010011. 01011011.0xxxxxxx.yyyyyyyy/17,
10010011. 01011011.10xxxxxx.yyyyyyyy/18,
10010011. 01011011.11xxxxxx.yyyyyyyy/18.

Prva podmreža ima 32 768 adresa, dok ostale dve podmreže imaju upola manje adresa. Kada poruka stigne do glavnog rutera mreže potrebno je dalje poruku proslediti određenoj podmreži. Da bi to ruter mogao da uradi, potrebno je samo da zna mrežne maske svih podmreža.

9.5.2. UDP protokol

UDP je protokol transportnog sloja u TCP/IP modelu dizajniran za korišćenje sa IP protokolom. Ovaj protokol omogućava programima da razmenjuju poruke minimalnim mehanizmom, bez uspostavljanja

konekcije. To znači da program pošiljalac pre slanja poruka ne proverava da li je program primalac dostupan i spreman za prihvatanje poruka. Zbog toga UDP ne garantuje isporuku poruka, pravilan redosled prilikom pristizanja poruka, a ne poseduje ni zaštitu od dupliranja poruka. Da bi postigao bolje performanse, UDP dozvoljava da pojedine poruke budu odbačene (bez zahteva za ponovnim slanjem) i da poruke mogu pristizati u proizvoljnom redosledu.



Slika 9.22. Primena UDP protokola

UDP poruka se sastoji od zaglavlja i korisničkih podataka. Zaglavlje UDP poruke ima 4 polja od po 2 bajta, i to su:

- Port pošiljaoca;
- Port primaoca;
- Dužina UDP poruke;
- Kontrolna suma (*checksum*).

UDP portovi omogućavaju da različite aplikacije na jednom računaru koriste (istu) mrežu. Svakoj aplikaciji se dodeljuje poseban port – broj na osnovu koga se može identifikovati aplikacija na računaru. Da bi jedna

aplikacija poslala poruku drugoj na udaljenom računaru, ona mora tu poruku da pošalje na adresu udaljenog računara sa odgovarajućim portom aplikacije. Portovi su 16-bitni brojevi, pa mogu imati vrednost od 0 do 65535. Vrednosti od 0 do 1023 su rezervisane za privilegovane servise. Na primer, SMTP protokol koristi port 25, HTTP koristi port 80, a POP3 koristi port 110. Dužina UDP poruke odnosi se na broj bajtova zajedno sa zaglavljem, dok se kontrolna suma koristi kao vid zaštite od neželjenih promena u poruci nastalih tokom transporta. Kontrolna suma je rezultat posebnog algoritma koji radi sa 16-bitnim rečima. Ulaz za algoritam se sastoji od bitova kompletne UDP poruke i bitova dela IP zaglavlja. Ako broj bitova nije deljiv sa 16, onda se UDP dopunjuje nulama. Promena bitova tokom transporta izaziva i promenu kontrolne sume na osnovu čega primalac može da odredi da li je primio konzistentne podatke ili ne.

UDP je transakciono orijentisan, tj. pogodan za jednostavne protokole tipa zahtev-odgovor. Primeri takvih protokola su DNS i NTP (*Network Time Protocol*). Zbog svoje jednostavnosti, UDP se koristi kod aplikacija u realnom vremenu gde je brzina bitnija od ispravnosti pristiglih podataka. Neke od njih su aplikacije za video konferencije (slika 9.22.), video igre, IPTV (*IP Television*) i VoIP (*Voice over IP*) sistemi itd.

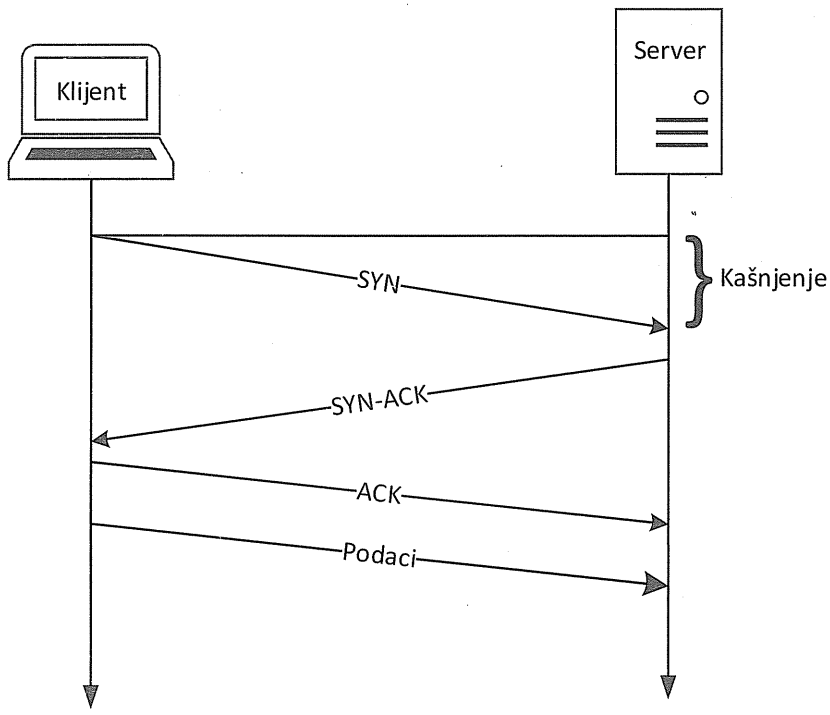
9.5.3. TCP protokol

TCP je jedan od najvažnijih protokola transportnog sloja u TCP/IP modelu, koji se koristi u kombinaciji sa IP protokolom. Kao i UDP, TCP koristi portove da bi omogućio korišćenje mreže većem broju aplikacija na jednom računaru. Za razliku od UPD protokola, ovaj protokol pruža aplikacijama više pogodnosti. TCP je pouzdan protokol, što znači da omogućuje uspostavljanje konekcije između procesa i garantuje da će podaci biti preneti bez greške na određenu destinaciju.

Da bi se omogućila pouzdanost, zaglavlje TCP paketa se u odnosu na UDP proširuje dodatnim poljima i kontrolnim bitovima. Kontrolni bitovi sadrže kontrolne signale koji su potrebni za uspostavljanje pouzdane komunikacije. Postoji ukupno 6 kontrolnih signala (i za svaki od njih po

jedan bit), a najčešće korišćeni su SYN (*Synchronization*) i ACK (*Acknowledgment*) signali. Signal SYN se koristi za početak komunikacije, dok signal ACK služi da potvrdi da je paket uspešno primljen.

Komunikacija dva procesa počinje uspostavljanjem konekcije. Konekcija se uspostavlja međusobnom razmenom praznih paketa koji sadrže SYN i ACK kontrolne signale. Na primer, neka je potrebno da klijent pošalje podatke serveru (slika 9.23.). Klijent prvo šalje prazan paket sa postavljenim SYN signalom. Server potvrđuje da je primio paket ACK signalom i šalje SYN signal. Ove signale nije neophodno slati u odvojenim paketima, zato server šalje jedan prazan paket sa postavljenim SYN i ACK signalima. Kada klijent primi ovaj paket, on odgovara praznim paketom sa postavljenim ACK signalom, nakon čega počinje da šalje podatke. Pošto je za uspostavljanje konekcije potrebno razmeniti 3 paketa, ovaj proces se još naziva i trostruko rukovanje (*Three-Way Handshake*).

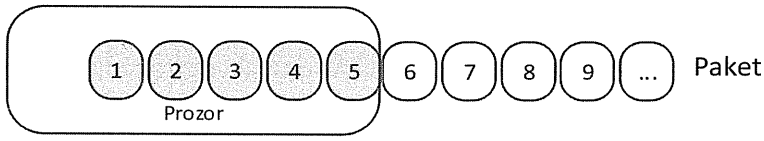


Slika 9.23. TCP protokol

Podatke pristigle sa aplikativnog sloja TCP deli u pakete određene veličine i svakom pridružuje redni broj. Na osnovu rednih brojeva primalac može da organizuje pakete i regeneriše poruku, ali i da utvrdi da li neki paket nedostaje. Od primaoca se očekuje da pošalje potvrdu (ACK) o poslednjem uspešno primljenom paketu. Tom prilikom primalac šalje i broj bajtova koji u tom trenutku može da primi. Ako potvrda (ACK) ne stigne u određenom vremenskom periodu, paket se ponovo šalje.

Da bi se bolje iskoristio protok mreže, koristi se poseban tzv. *mehanizam prozora*. Zagušenje se može izbeći ako bi se paketi slali jedan po jedan, pri čemu bi se svaki sledeći paket slao tek po primljenoj potvrdi o uspešnom transferu prethodnog paketa. Ovakav način slanja ne koristi najbolje protok mreže. Mehanizam prozora dozvoljava da se šalje veći broj paketa istovremeno čime se povećava propusnost mreže, odnosno količina podataka koji se mogu preneti mrežom u jedinici vremena. Mehanizam prozora radi na nivou bajtova, ali će radi lakšeg razumevanja biti predstavljen na nivou paketa. Osnovna karakteristika prozora je veličina i ona predstavlja broj paketa koje pošiljalac može da šalje bez čekanja potvrde (slika 9.24.).

Na primer, neka je veličina prozora 5. Pošiljalac šalje prvih 5 paketa i za svaki od njih počinje da meri vreme čekanja na odgovor. Neka su svi paketi osim trećeg stigli na odredište. Primalac može slanjem ACK2 signala da potvrdi da je drugi paket poslednji regularno stigao. Pošiljalac „pomera" prozor za dva mesta udesno. Kada istekne vreme čekanja, ponovo se šalje treći paket. Sada pretpostavimo da je paket stigao na odredište. Pošto je peti paket poslednji regularan, primalac odgovara sa ACK5. Na osnovu toga pošiljalac zna da je prvih 5 paketa ispravno poslao, pa pomera prozor udesno na odgovarajuću poziciju i nastavlja sa slanjem.



Slika 9.24. Mehanizam prozora

9.6. Poređenje i kritika OSI i TCP/IP modela

Tokom godina oba modela su ostavila trag na razvoj mreža i mrežne industrije. U njihovim implementacijama u realnim mrežama pokazalo se da i OSI i TCP/IP model imaju nedostatke koje je važno spomenuti. TCP/IP je najčešće korišćeni standard u realnim aplikacijama, dok OSI modela gotovo da nema. Ipak, OSI model se koristi u teoriji kao paradigma apstrakcije mreža i za bolje razumevanje mrežne komunikacije.

Modeli OSI i TCP/IP imaju dosta toga zajedničkog. Oba modela dele organizaciju po slojevima. Sloj aplikacije je zajednički za oba modela. U praksi, ovaj sloj sadrži različite servise u zavisnosti od modela. Na primer, u aplikativnom sloju TCP/IP ima dodatne servise koji obavljaju posao slojeva prezentacije i sesije, jer on ne poseduje te slojeve. Takođe, zajednički su i transportni i mrežni sloj. Oba modela podrazumevaju da se paketi navode kroz mrežu. To znači da se pojedini paketi mogu prenositi različitim putanjama do iste destinacije.

Pored navedenih sličnosti, ovi modeli se veoma razlikuju. Prvo što se može uočiti je različit broj slojeva. Sedam slojeva u OSI modelu su više politička nego tehnička odluka. Osnovni razlog ovakve OSI arhitekture je IBM-ova SNA arhitektura (*System Network Architecture*) koja je imala 7 slojeva, jer se 1970-tih smatralo da bi IBM mogao da bude vodeći akter u industriji mreža. Zato je OSI dizajniran tako da se može primeniti na SNA mrežama. TCP/IP objedinjuje sloj prezentacije i sesije sa slojem aplikacije, kao i fizički sloj sa slojem veze podataka.

Značajne razlike postoje u dizajnu modela, što je posledica toka njihovog razvoja. Razvoj TCP/IP modela povezan je sa nastankom interneta. Kasnih 1970-tih nastajali su protokoli u cilju prevazilaženja tehničkih različitosti mreža koje su se povezivale na ARPANET. Kada su nastali protokoli, model TCP/IP se pojavio kao njihova specifikacija. Zbog ovakvog razvoja TCP/IP veoma dobro funkcioniše sa postojećim protokolima. Negativna posledica ovakvog razvoja je nedovoljno razgraničenje servisa, interfejsa i protokola, pa TCP/IP nije pogodan za uvođenje novih tehnologija i

protokola. Sa druge strane, OSI model je imao obrnut razvojni proces. Prvo je dizajniran model, pa su onda nastajali protokoli. Zato OSI model razgraničava koncepte servisa, interfejsa i protokola, pa se za njega kaže da je generički model nezavisan od protokola. Mana ovakvog razvoja OSI modela je što njegovi protokoli ne funkcionišu dobro. Loša je i raspodela funkcionalnosti po slojevima. Neki slojevi kao što su sloj prezentacije i sesije su gotovo prazni, dok su mrežni i sloj veze podataka prepunjeni. Velika mana OSI modela je što se implementacija ključnih funkcionalnosti ne vrši na posebnim slojevima. Neke funkcionalnosti, kao što je kontrola toka, ponavljaju se na svim slojevima, što značajno usložnjava ovaj model.

Sa stanovišta korisnika (aplikacije), veliki nedostatak OSI modela je što na transportnom sloju podržava samo komunikaciju sa uspostavljanjem konekcije, dok na mrežnom sloju podržava komunikaciju sa konekcijom i bez nje. Kod TCP/IP modela je obrnut slučaj – podržava komunikaciju bez konekcije na mrežnom sloju, dok na transportnom podržava oba načina komunikacije. To je velika prednost TCP/IP modela zato što aplikacije mogu da biraju način komunikacije (aplikacije koriste usluge transportnog sloja).

Navedene osobine dovele su do velike popularnosti TCP/IP modela, dok OSI model nikada nije zaživeo. Problem OSI modela nije samo kompleksnost i neefikasnost koju je ispoljio u prvim implementacijama. Trenutak pojavljivanja je bio loš. Tada su protokoli TCP/IP modela već bili u upotrebi, i niko nije imao potrebu da ih menja. TCP/IP se pojavio uz Berkeley UNIX operativni sistem i bio je besplatan. Mnogi su na TCP/IP gledali kao na deo UNIX operativnog sistema koji je 1980-tih bio veoma rasprostranjen. Iako se OSI popravljao tokom vremena, ostala je slika o njemu kao neefikasnom modelu koji se koristi u teorijskim razmatranjima.

9.7. Bezbednost računarskih mreža

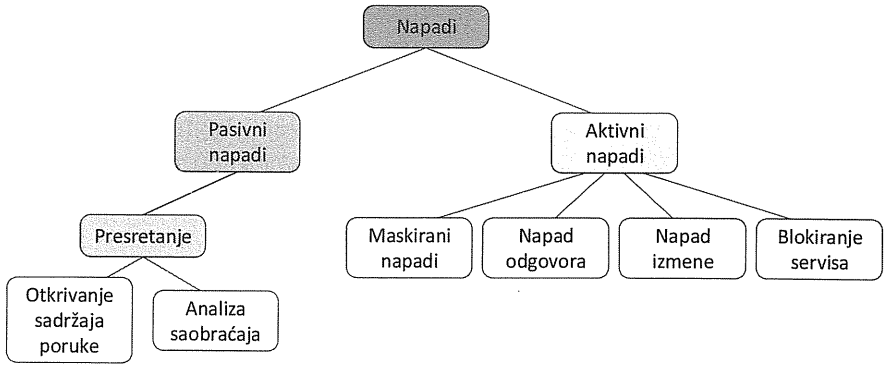
Osnovna ideja računarskih mreža je da omoguće pristup podacima na udaljenim lokacijama bez fizičkog prisustva. Svakim danom raste količina ličnih podataka na internetu i gotovo svaka moderna kompanija koristi

pogodnosti računarskih mreža u poslovne svrhe. Za korisnike mreža sa lošim namerama svi ti podaci predstavljaju potencijalnu metu za napad i zloupotrebu.

Kada je koncept računarskih mreža bio relativno nov, bezbednost mreža se nije mnogo razmatrala. Čak se smatralo da bezbednost mreža nije neophodna. Takvo razmišljanje je bilo opravdano jer i da je postojao neko ko bi želeo da zloupotrebi mrežu, nije imao tehnologiju i znanje da to uradi. Danas je situacija drugačija. Postoji mnogo ljudi koji su stekli određen nivo znanja o mrežama a mogli bi da ga zloupotrebe. Skoro svakog dana u medijima se može čuti neka vest o zloupotrebama na internetu, kao što su krađa identiteta ili poverljivih finansijskih podataka. Tačno je da su oni najčešća meta napada, ali nisu samo podaci na mreži ugroženi. Vrlo često se dešavaju napadi „sabotaže" koji imaju za cilj da onemoguće pojedine servise. Zbog svega navedenog, bezbednost na mrežama je danas primarna tema u tehnološkoj industriji sa ciljem da se smanji, ako ne i potpuno ukloni mogućnost napada na mreži. Da bi se računarske mreže zaštitile od neželjenih napada, neophodno je različite vrste napada razumeti i proučiti.

9.7.1. Teorijska podela napada

Napadi se grubo mogu podeliti na *pasivne* i *aktivne* (slika 9.25.).



Slika 9.25. Podela napada

Kod *pasivnih napada* napadač presreće poruke i pretražuje ih sa ciljem da sazna nešto više o mreži, učesnicima u komunikaciji ili samoj komunikaciji. Termin „pasivni" označava da napadač ne vrši nikakvu izmenu podataka. Zbog toga je veoma teško prepoznati ovu vrstu napada. Iako pasivni napadi deluju bezopasno, potencijalna šteta može biti velika, jer se podaci dobijeni iz pasivnih napada koriste za aktivne napade. Postoje dva osnovna tipa pasivnih napada, a to su otkrivanje sadržaja poruka i analiza saobraćaja.

Napad otkrivanja sadržaja poruka (*Release of message content*) je prilično jednostavan. Elektronska pošta ili neki drugi vidovi komunikacije često sadrže osetljive podatke. Ovi napadi se sastoje od praćenja nezaštićene komunikacije ili dešifrovanja slabo šifrovanih poruka (dobijenih nedovoljno kompleksnim algoritmom šifrovanja) da bi se došlo do poverljivih podataka kao što su podaci za autentifikaciju. Više o autentifikaciji se može videti u delu 9.7.4. Osnovnu zaštitu od ovakvih napada pruža kriptografija.

Kod *analize saobraćaja* (*Traffic analysis*) sadržaj poruke nije bitan. Na osnovu frekventnosti poruka napadač pokušava da sazna nešto više o sistemu. Na primer, ako napadač primeti da je razmena poruka sa određenim računarom u mreži učestalija, on može zaključiti da određeni računar ima važnu ulogu u mreži (npr. ulogu servera). Sledeće akcije napadača mogu biti usmerene ka izvršavanju aktivnog napada sa ciljem da se navedeni računar onesposobi.

Osnovna karakteristika *aktivnih napada* je pokušaj nedozvoljene promene sistema koji se napada. To može biti izmena ili uništavanje podataka koji se prenose kroz mrežu, izmena ili uništavanje podataka koji se čuvaju na određenom računaru, neovlašćeno kreiranje novih podataka, obaranje pojedinih servisa sistema ili čitavog sistema itd. Ovi napadi se lakše prepoznaju zbog izmena na sistemima i štete koju prouzrokuju. Aktivni napadi se mogu podeliti u 4 podvrste:

- Maskirani napadi;
- Napad odgovora;

- Napad izmene;
- Blokiranje servisa.

Maskirani napad (*Masquerade attack*) se odnosi na situaciju u kojoj se napadač predstavlja kao neko drugi sa ciljem da dobije privilegovan status unutar sistema. Maskirani napadi se mogu realizovati na različite načine. Krađa autentifikacionih podataka nekog legalnog korisnika je uobičajeni slučaj. Drugi način je pronalaženje bezbednosnih slabosti ili izbegavanje autentifikacionog postupka nekog sistema.

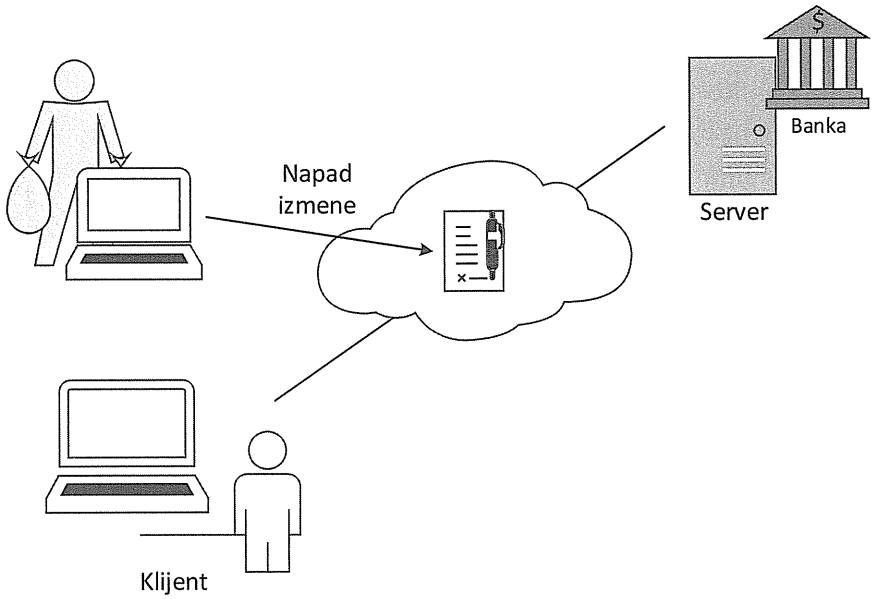
Napad odgovora (*Replay message attack*) podrazumeva da napadač poseduje kopije šifrovanih poruka upućenih nekom računaru. Na primer, kopije se mogu dobiti prisluškivanjem i presretanjem komunikacije dva računara na mreži. Napadač kasnije može poslati navedenom računaru presretnute poruke očekujući da će dobiti odgovor. Na taj način napadač pokušava da dođe do zaštićenih podataka.

Napad izmene (*Alteration message attack*) se dešava kada napadač presretne i izmeni poruku (slika 9.26.). Na primer, napadač može u zaglavlju izmeniti adresu primaoca da bi je preusmerio. Takođe, može izmeniti sadržaj poruke da bi ostvario ličnu korist.

Blokiranje servisa (*Denial-of-Service attack – DoS*) služi da onemogući normalno funkcionisanje sistema. Najčešće, meta napada su veb servisi i serveri za elektronsku poštu, ali to mogu biti bilo koji uređaji povezani na mrežu. Ideja ovih napada je da se servis optereti velikim brojem zahteva. To dovodi do usporenja u funkcionisanju servisa, a često i do pada servisa. Najčešći primer DoS napada je preopterećenje mreže beskorisnim podacima od strane napadača. Isto tako, veliki broj (regularnih) zahteva za prijavljivanje može izazvati blokadu servisa.

Popularna vrsta DoS napada u prošlosti bio je tzv. *ping smrti* (*Ping-of-Death – PoD*). Ping je posebna funkcionalnost operativnih sistema koja se obično koristi za testiranje ispravnosti računarskih mreža. Ping je program koji šalje pakete na određenu adresu, a kao odgovor očekuje iste te pakete. U zavisnosti od pristiglih paketa, program će ispisati statističke

podatke o eventualnim gubicima podataka, vremenu potrebnom za slanje i primanje paketa, i sl. Ping smrti se realizovao slanjem ping zahteva većeg od 65536 bajtova, što je najveći dozvoljen broj bajtova u paketu IP protokola. Zbog toga se vrši fragmentacija podataka u dva paketa. Kada paketi stignu na odredište, oni se spajaju u početnu poruku. Ta količina podataka može da izazove prepunjenost bafera i pad sistema. PoD je bio moguć zbog greške u operativnim sistemima, koja je kasnije ispravljena.



Slika 9.26. Napad izmene

DoS napad može biti iniciran sa jednog računara. Međutim, zaštitni softveri (npr. zaštitni zid – *firewall*) omogućuju da se pojedinim računarima zabrani mogućnost komunikacije i pristup resursima na sistemu, što predstavlja zaštitu od ovakve vrste napada. Tako je nastao distribuirani DoS – DDoS. Za ovaj napad je potreban veći broj računara sa kojih se u unapred dogovoreno vreme pokreće DoS napad. DDoS je značajno efektivniji napad u odnosu na DoS.

9.7.2. Praktični napadi

Realni primeri napada su veoma raznovrsni i nije jednostavno razvrstati ih prema navedenim kategorijama. Neki od njih se mogu rasporediti u više kategorija. Zbog ovoga biće navedeni najznačajniji primeri realnih napada bez njihove stroge klasifikacije.

- *Skeniranje portova* (*Port scanning*) je napad koji nije u potpunosti pasivan jer zahteva određene akcije napadača, ali se ne može ni svrstati u aktivne. Ovaj napad je moguće realizovati na sistemima koji koriste TCP ili UDP portove. Skeniranjem portova napadač pokušava da pronađe portove preko kojih bi mogao da izvrši aktivne napade. Napadač ispituje redom sve portove tako što pokušava da uspostavi TCP/IP konekciju na tom portu. Ukoliko se na nekom portu uspostavi konekcija, može se pokušati sa pronalaženjem dodatnih slabosti procesa koji koristi taj port.

- *Prepunjenost bafera* (*Buffer overflow*) je napad koji koristi nedostatke programa (bagove) i pokušava da u neki od bafera sačuva količinu podataka veću od njegovog kapaciteta. Baferi služe za čuvanje ograničene količine podataka, pa će se višak podataka upisati u susednu memoriju. To može da zaustavi rad programa, ali i da dovede do složenijih neželjenih posledica ako taj višak podataka sadrži neke instrukcije. Neka je, na primer, napadač generisao (napisao) deo programa, koji je zatim preveo u mašinski kôd. Napadač može pokušati da izvrši taj kôd tako što će ulazni bafer napuniti do kraja regularnim podacima, a zatim na njih nadovezati bajtove mašinskog koda. Ovaj napad se može sprečiti jednostavnom proverom da li je veličina ulaznih podataka manja od kapaciteta bafera.

  Ovakvi nedostaci otkriveni su 2000. godine kod programa Microsoft Outlook i Outlook Express. Napad se mogao izvršiti običnim slanjem poruke elektronske pošte. Greška u delu

programa koji je bio zadužen za čitanje zaglavlja poruke, omogućila je napadačima da proslede i izvrše bilo koje instrukcije. Zaštita nije bila moguća kao kod poruka sa virusom, gde je dovoljno ne otvarati poruku. Microsoft je ubrzo napravio i objavio zakrpu (*patch*) da bi popravio navedene programe.

- *Računarski virus* je deo programa koji je umetnut u regularni program. Virus se sam umnožava (kopiranjem) i umeće u druge programe na računaru i tom prilikom može se sam menjati (mutirati). Računar se može zaraziti na razne načine: skidanjem zaraženih podataka sa interneta, pomoću poruke elektronske pošte, preko zaraženih USB diskova i drugih medija itd. Virusi ne „inficiraju" samo programe. Oni se mogu umetnuti i u neke dokumente koji imaju izvršni sadržaj. Takvi su Microsoft Office dokumenti. U određenim delovima oni sadrže makroe koji se izvršavaju automatski prilikom otvaranja dokumenta.

Virusi mogu imati razne destruktivne osobine. Oni mogu brisati dokumente na sistemu, zauzimati prostor na disku, memoriji ili procesorsko vreme, mogu snimati sve unose sa namerom da dođu do osetljivih podataka, mogu ispisivati na ekranu poruke raznog sadržaja, ili samo ukazivati na postojanje slabosti u sistemu.

Personalni računari su najčešća meta virusa. Višekorisnički operativni sistemi, kao što je UNIX, nisu pogodni za viruse jer ti operativni sistemi zabranjuju izmenu izvršnih fajlova. Čak 99% virusa su „namenjeni" Windows operativnim sistemima.

Od svih napada virusi nanose najveće štete koja se na godišnjem nivou meri milijardama dolara. U martu 1999. godine pokrenut je virus „Melissa" koji je bio umetnut u Microsoft Word dokument. Zaraženi dokument se širio putem elektronske pošte. Kada bi korisnik otvorio pristigli dokument, automatski bi se poslala ista poruka na 50 prvih adresa iz njegove adresne liste. Virus se veoma brzo širio pa su mnoge kompanije bile primorane da ugase svoje

servise za elektronsku poštu. Procenjuje se da je ovaj virus naneo štetu kompanijama u vrednosti 1.2 milijarde dolara. Sličan virus nazvan „Love Bug“ se javio 2000. godine. Ovaj virus se širio elektronskim porukama sa naslovom „I love you“ i procenjuje se da je napravio štetu od 10 milijardi dolara.

- *Trojanski konj*, popularno trojanac, je program koji je maskiran tako da izgleda kao regularna aplikacija, ali u sebi sadrži destruktivne instrukcije. Za razliku od virusa, trojanci se ne umnožavaju sami. Trojanci se na internetu često predstavljaju kao koristan softver. Drugi često korišćeni način za širenje trojanaca je elektronska pošta.

  Nakon skidanja trojanca sa interneta, korisnik ga pokreće ne sumnjajući da je to destruktivni softver. Kada se trojanac pokrene, rezultat može biti različit. Neki manje opasni trojanci mogu dodavati veliki broj prečica i ikonica na radnu površinu. Oni opasniji mogu brisati (menjati) korisničke podatke, zauzimati resurse računara ili dovesti do pada sistema. Često se trojanci koriste i za krađu autentifikacionih podataka tako što su napravljeni da izgledaju kao forma za prijavljivanje. Od korisnika se zahteva da se „prijavi“ da bi mogao da koristi pogodnosti softvera koji je skinuo. Ako se korisnik prevari i unese rezultate, napad je uspešno izvršen. Trojanci se neretko koriste kao inicijatori virusa. U tom slučaju, oni sa sobom nose virus i njihovim pokretanjem virus se ubacuje u sistem.

- *Računarski crvi* (*computer worm*) su mali delovi softvera koji za širenje koriste bezbednosne propuste u mreži. Slično kao i virusi, crvi se umnožavaju tako što sami sebe kopiraju. Za razliku od virusa, koji menjaju programe ili fajlove da bi se širili, crvi su samostalni i ne zahtevaju akciju korisnika za širenje. Crvi mogu da prouzrokuju istu štetu kao i virusi. Ipak, oni najčešće ne oštećuju podatke, već samo zauzimaju memoriju ili opterećuju mrežu

smanjujući njen protok. Zauzimanje resursa ima za cilj da se onemogući normalan rad pojedinih računara ili čitavih mreža.

Prvog računarskog crva napravio je student (*Robert Morris*) sa univerziteta Cornell 1988. godine. Prema tvrdnjama autora, on nije imao destruktivne namere, već je napravio ovaj program u eksperimentalne svrhe da bi izmerio veličinu interneta. Ovaj crv je pokrenut sa MIT-a (*Massachusetts Institute of Technology*) i zbog greške koju je imao širio se mnogo većom brzinom od predviđene. To je izazvalo pad mnogih računarskih sistema širom SAD-a, uključujući računare na univerzitetima, u vojsci, istraživačkim centrima itd. Kada je shvatio šta se dešava, Morris je poslao anonimnu poruku kroz mrežu sa instrukcijama kako da se crv uništi. Međutim, već tada je bilo kasno. Cena uklanjanja virusa sa jednog računara kretala se u opsegu od 200 do 53000 dolara.

- *„Čovek u sredini“* (*Man in the middle*) je napad u kome napadač prisluškuje komunikaciju između dva korisnika koji u komunikaciji koriste asimetrične algoritme šifrovanja (videti deo 9.7.3.). Ovi algoritmi koriste dva ključa – javni ključ koji služi za šifrovanje poruka i tajni ključ koji služi za dešifrovanje poruka. Svaki učesnik u komunikaciji zna samo svoj tajni ključ, dok su javni ključevi svima poznati. Pošiljalac šifruje poruku koristeći javni ključ primaoca i na nju nadovezuje svoj javni ključ. Primalac dešifruje poruku svojim tajnim ključem, a odgovor šifruje javnim ključem pošiljaoca koji je primio zajedno sa porukom. U opštem slučaju, napadač presreće poruku koja sadrži javni ključ pošiljaoca, zamenjuje javni ključ svojim javnim ključem i prosleđuje poruku primaocu. Zatim, napadač presreće odgovor pošiljaoca koji može da dešifruje jer je poruka šifrovana njegovim javnim ključem.

9.7.3. Odbrana od napada na mreži

Tokom proteklih godina sa razvojem napada unutar računarskih mreža razvijala se i odbrana od njih. Postoje različite tehnike i softveri koji

sprečavaju napade. Pojedinačno, ove tehnike ne mogu obezbediti potpunu sigurnost sistema. Tek kada se one iskombinuju u nekom sistemu, onda se može pretpostaviti da je taj sistem bezbedan. Na žalost, potpuna bezbednost se ne može garantovati. Proizvođači softverskih i hardverskih alata za odbranu su u stalnom nadmetanju sa kreativnim napadačima.

Antivirus je zaštitni softver koji štiti računare od malicioznog softvera (*malicious software*). Drugi naziv za maliciozni softver je malver (*malware*) i on obuhvata viruse, trojance, crve i slične softvere. Da bi bio efikasan, antivirus je stalno aktivan u sistemu kao pozadinski proces i on sprečava, detektuje i uklanja napade malicioznog softvera. Antivirus koristi različite tehnike prepoznavanja malvera, a najčešće korišćene su detektovanje potpisa i heuristička detekcija.

Detektovanja potpisa je tehnika koja se obično koristi za prepoznavanje virusa. *Potpis virusa* je heš vrednost (videti deo 9.7.4.) njegovog koda. Antivirus koristi heš mapu koja sadrži potpise virusa. Za vreme svog rada, antivirus skenira dokumente u računaru i pokušava da pronađe delove koda. Kada naiđe na kôd, on računa heš vrednost tog koda i upoređuje sa potpisima u heš mapi. Ako se dogodi poklapanje, antivirus može preduzeti različite akcije, kao što su brisanje dokumenta, pokušaj popravke dokumenta ili smeštanje dokumenta u karantin. Pošto se zasniva na bazi poznatih virusa, ovaj metod nije efikasan kod novih virusa. Da bi bili efikasniji, antivirusi preporučuju često ažuriranje heš mape. Takođe, proizvođači antivirusnog softvera podstiču korisnike da dele (*upload*) nove viruse kako bi se oni proučili i njihovi potpisi dodali u heš mapu.

Heuristička detekcija virusa se zasniva na praćenju svih programa i pokušaju da se prepoznaju sumnjiva ponašanja. Ova tehnika ne koristi heš mapu i zastupljena je kod naprednijih antivirusa da bi prepoznali nove viruse ili varijacije postojećih virusa. Na primer, antivirus registruje pokušaj menjanja izvršnog fajla i o tome obaveštava korisnika koji

odlučuje o daljim akcijama. Moderni komercijalni antivirusi kombinuju ove tehnike.

Iako predstavljaju veoma efikasnu zaštitu, antivirusi imaju nekoliko mana. Prva od njih je što ne obezbeđuju potpunu zaštitu od malvera. Napadi novih malvera za koje antivirusi nemaju podatke popularno se nazivaju „napadi od nultog dana" (*zero-day attacks*). Ovi napadi mogu biti veoma uspešni u periodu od pokretanja novog virusa pa sve do njegovog registrovanja i proučavanja njegovog potpisa. Druga mana antivirusa je ta što oni zauzimaju resurse računara, što može uticati na njegove performanse.

Zaštitni zid (*Firewall*) je metod ili uređaj koji reguliše nivo poverenja između dve ili više mreža. To može biti softver ili hardver, mada se moderni zaštitni zidovi implementiraju kao posebne mašine koje kombinuju softver sa hardverom. Zaštitni zid štiti lokalnu mrežu od pristupa neautorizovanih korisnika sa interneta, ali može da se koristi za regulisanje saobraćaja između dve mreže iste kompanije. Sve poruke koje pristižu u lokalnu mrežu ili je napuštaju ili prolaze kroz zaštitni zid. One poruke koje ne zadovoljavaju navedeni sigurnosni kriterijum bivaju odbačene. Zaštitni zidovi ne mogu da pruže zaštitu od virusa, ali u određenim okolnostima mogu da spreče viruse da šalju podatke sa zaraženih računara.

Postoje različite vrste zaštitnih zidova:

- *Zaštitni zid za filtriranje paketa* (*Packet filtering firewall*) – ovaj zaštitni zid, u zavisnosti od konfiguracije, prihvata ili odbija pakete na osnovu informacija iz zaglavlja paketa. Filtriranje paketa se može obavljati na mrežnom i transportnom sloju i to na osnovu IP adrese pošiljaoca i/ili primaoca, portova pošiljaoca i/ili primaoca, protokola itd. Filtriranje paketa je efektivna metoda zaštite od neautorizovanog pristupa sistemu, ali je veoma zahtevna za konfigurisanje.

- *Kružni prolaz* (*Circular-level gateway*) – ova vrsta zaštitnog zida obezbeđuje sigurnost TCP konekcije. Kružni prolaz nadgleda proces uspostavljanja TCP konekcija i proverava validnost tih konekcija na osnovu zadatih pravila. Pravila za definisanje validnih sesija mogu biti vezana za IP adrese primaoca i/ili pošiljaoca, njihove portove, doba dana, protokol, korisničko ime, lozinku itd. Kada ustanovi validnost konekcije, kružni prolaz otvara sesiju i dozvoljava razmenu podataka. Kada se otvori sesija, dalje provere se ne izvršavaju. Podaci koji prolaze kroz kružni prolaz ka internetu deluju kao da potiču od samog kružnog prolaza. Zbog toga udaljeni računar ne može da odredi interne privatne IP adrese računara (npr. neke organizacije). Ova tehnika se naziva prevođenje mrežnih adresa (*Network Address Translation* – NAT) i podrazumeva pridruživanje privatnih IP adresa više uređaja jednoj javnoj IP adresi.

- *Aplikacioni prolaz* (*Application-level gateway*) – ovaj zaštitni zid čita sadržaj paketa i na osnovu njega odlučuje da li će ga propustiti ili odbaciti. Aplikacioni prolaz skenira pakete u potrazi za malverom i ukoliko ga nađe, paket se odbacuje. Zbog toga, aplikacioni prolaz pruža najveći stepen bezbednosti. Aplikacioni prolaz se implementira kao „server zastupnik“, tj. proksi server (*proxy server*). To je posebna mašina koja se nalazi između klijentskog i servisnog računara. Kada klijent želi da se poveže sa servisnim računarom, on se zapravo povezuje sa proksi serverom koji zastupa servisni računar. Čitava komunikacija se odvija preko proksi servisa. Servisni računar ostaje zaštićen jer klijent ne može da odredi njegovu adresu. Proksi server zahteva veliku količinu resursa (memorije, procesora), a javlja se i problem rutiranja paketa.

U praksi, mnogi zaštitni zidovi kombinuju navedene metode. Zaštitni zidovi se često ugrađuju u rutere i u operativne sisteme. Windows i Mac

operativni sistemi imaju implementiran zaštitni zid. Takođe, postoje dodatni softverski alati koji nude razne slične usluge, a proizvođači ovih softvera su kompanije koje se bave proizvodnjom antivirus alata.

Kriptografija ima veliku ulogu u bezbednosti računarskih mreža. Reč kriptografija potiče iz grčkog jezika i znači tajno pisanje. Ona omogućuje bezbednu razmenu poruka i sprečava krađu poverljivih podataka, čak i u slučajevima kada je napadač u mogućnosti da presretne te poruke. Osnovna ideja je da pošiljalac transformiše poruku u neki nečitljivi format i pošalje primaocu, koji jedini može da izvrši inverznu transformaciju i dođe do originalne poruke. Ukoliko napadač presretne poruku, on nije u mogućnosti da otkrije njen sadržaj. Pored toga što omogućuje poverljivost podataka, kriptografija obezbeđuje i ostale kritične bezbednosne zahteve kao što su autentifikacija i integritet podataka.

Slede definicije osnovnih pojmova kriptografije. Poruka koja se transformiše naziva se *otvoreni tekst*, a proces transformacije se naziva *šifrovanje*. Rezultat šifrovanja se naziva *šifrat*. Šifrovanje je parametrizovana funkcija (*E*) koja preslikava skup otvorenih tekstova u skup šifrata. Parametar šifrovanja se naziva *ključ* (*k*) i svaki ključ određuje tačno jednu funkciju (odnosno algoritam) E_k . Inverzan proces od šifrovanja je *dešifrovanje* (*D*) i važi $D = E^{-1}$. Ako je sa *m* označen otvoren tekst, a sa *c* njegov šifrat, onda važi: $E_k(m) = c$ i $D_k(c) = m$. Kada napadač dođe u posed šifrata, on pokušava da otkrije sadržaj poruke i ovaj proces se naziva *kriptoanaliza*. Nauka koja se bavi kriptografijom i kriptoanalizom naziva se *kriptologija*.

Prema vrsti ključa, algoritmi šifrovanja se mogu podeliti u dve grupe: simetrični algoritmi i asimetrični algoritmi šifrovanja.

Simetrični algoritmi šifrovanja podrazumevaju postojanje jedinstvenog ključa koji se koristi i za šifrovanje i za dešifrovanje. Neophodno je da taj ključ znaju samo učesnici u komunikaciji, pa se zato ovi algoritmi nazivaju još i algoritmi sa tajnim ključem. Problem ovih algoritama jeste razmena ključeva. Da bi razmenili ključ, učesnici u komunikaciji se moraju lično sastati ili to obaviti razmenu preko neke druge stranke od poverenja.

Prvi široko rasprostranjeni simetrični algoritam bio je *DES* (*Data Encryption Standard*). Razvijen je početkom sedamdesetih godina u IBM-u. Nakon konsultovanja sa organizacijom NSA (*National Security Agency*) 1976. godine, nacionalni biro za standarde (*National Bureau of Standards* – NBS) u SAD prihvatio je DES, što je prouzrokovalo brzo širenje i prihvatanje ovog standarda. DES koristi ključ dužine 56 bitova i izvršava šifrovanje 64-bitnih vrednosti. Šifrovanje se obavlja izvršavanjem serije od 16 rundi (transformacija) nad 64-bitnim blokom koje se zasnivaju na operacijama supstitucije i permutacije. Zbog toga ovaj algoritam pripada klasi tzv. blokovskih algoritama šifrovanja. Danas DES nije siguran algoritam zato što dužina ključa od 56 bitova obezbeđuje 2^{56} različitih vrednosti. Već devedesetih godina razvijeni su dovoljno moćni računari koji su mogli da otkriju ključ grubom silom (isprobavanje svih mogućih vrednosti ključa). Zato je DES bio zamenjen trostrukim *DES-om* (*3DES*). Algoritam 3DES je podrazumevao da se šifrovanje vrši primenom DES algoritma 3 puta (dva puta šifrovanje i jednom dešifrovanje) sa 2 ili 3 različita ključa. 3DES je danas u širokoj upotrebi.

Januara 1997. godine, Nacionalni institut za standarde i tehnologiju (*National Institute of Standard and Technology* – NIST, nekadašnji NBS) raspisao je konkurs za novi algoritam šifrovanja *AES* (*Advanced Encryption Standard*) koji treba da zameni DES. Bilo je odlučeno da ceo proces selekcije bude otvoren za javnost, jer se smatralo da će to dovesti do izbora najboljeg algoritma. Na tom konkursu je pobedio algoritam Rijndael čiji su autori Belgijanci Daemen (*Joan Daemen*) i Rijmen (*Vincent Rijmen*). Algoritam Rijndael je simetrični i blokovski algoritam šifrovanja koji koristi ključ dužine 128, 192 ili 256 bitova. Radi nad blokovima od 128 bitova i primenjuje seriju od 10, 12 ili 14 rundi, u zavisnosti od dužine ključa. Procena je bila da algoritam može da zadovolji bezbednosne zahteve u narednih 30-tak godina. Algoritam je bio lak za softversku i hardversku implementaciju, ali i u ograničenim okruženjima kao što su pametne kartice. Zbog svega navedenog, NIST je 2000. godine proglasio Rijndael za novi standardni algoritam šifrovanja – AES.

Pored navedenih blokovskih simetričnih algoritama, veoma popularan je simetrični protočni algoritam *RC4* (*Rivest Cipher 4*) koga je pronašao Ron Rivest 1987. godine. Za dati ulaz (ključ) ovaj algoritam generiše pseudo slučajan niz bitova proizvoljne dužine. Vrednosti bitova u nizu su iste za istu vrednost ključa. Šifrovanje se vrši tako što se bitovi poruke XOR-uju sa bitovima ključa. U odnosu na blokovske šifre, prednost ovog algoritma je brzina. Prednost blokovskih algoritama je što oni mogu koristiti isti ključ više puta. U određenim (teoretskim) okolnostima, kada je dostupno više šifrata dobijenih istim ključem, moguće je izvršiti kriptoanalizu.

Nedostatak simetričnih algoritama je taj što učesnici u komunikaciji moraju da koriste isti ključ. Pre razmene poruka oni moraju najpre da na neki pouzdan način usaglase ključeve, što nije uvek jednostavno uraditi. Asimetrični algoritmi rešavaju ovaj problem.

Asimetrični algoritmi šifrovanja se drugačije nazivaju algoritmi sa javnim ključem i koriste dva ključa. Jedan ključ se koristi za šifrovanje i to je *javni ključ*, a drugi se koristi za dešifrovanje i to je *tajni ključ*. Korisnik koji želi da učestvuje u komunikaciji bira tajni ključ i on je poznat samo njemu. Na osnovu tajnog ključa on računa javni ključ koji objavljuje. Računanje javnog ključa na osnovu tajnog ključa je relativno jednostavno. Obrnuti proces je praktično nemoguć (u doglednom vremenu), jer se zasniva na rešavanju matematičkih problema vezanih za faktorizaciju velikih celih brojeva, diskretni logaritam i eliptičke krive. Do sada nije pronađen efikasan način za rešavanje navedenih matematičkih problema. Zato javni ključ može biti objavljen bez opasnosti da potencijalni napadač na osnovu njega može doći do tajnog ključa.

Nakon što su svi učesnici u komunikaciji odredili svoje ključeve, komunikacija može da počne. Ako neki korisnik želi da pošalje poruku drugom korisniku, on tu poruku šifruje javnim ključem korisnika primaoca. Jedino primalac može da dešifruje tu poruku, jer samo on zna odgovarajući ključ za dešifrovanje.

Prvi asimetrični algoritam šifrovanja objavljen je 1976. godine a njegovi autori su Difi (*Whitfield Diffie*) i Helman (*Martin Hellmann*). Ovaj

algoritam se zasniva na rešavanju problema diskretnog logaritma i koristi se za razmenu ključeva kod simetričnih algoritama (DES, AES). Drugi popularan asimetrični algoritam je *RSA* koji je objavljen 1977. godine, a njegovi autori su Rivest (*Ronald RIvest*), Samir (*Adi Samir*) i Adleman (*Leonard Adleman*). Algoritam se zasniva na faktorizaciji celog broja koji je proizvod dva velika prosta broja. Osim za razmenu ključeva, RSA se koristi i za šifrovanje dužih poruka.

I simetrični i asimetrični algoritmi imaju prednosti i nedostatke. Prednost simetričnih algoritama je ta što su brži i ne zahtevaju mnogo resursa za izvršavanje. Za svaki par u komunikaciji postoji po jedan ključ, pa ako dođe do otkrivanja jednog ključa, ostala komunikacija time neće biti poremećena. Mane ovih algoritama se odnose na prevelik broj ključeva, kao i problem razmene tih ključeva. Takođe, nije moguće odrediti poreklo i autentičnost poruke, jer i pošiljalac i primalac koriste isti ključ. Sa druge strane, asimetrični algoritmi rešavaju problem razmene ključeva. Oni omogućuju autentifikaciju učesnika tako da se zna autor svake poruke. Time je sprečeno odricanje od poruke, tako da ako je određeni korisnik generisao neku poruku, to ne može negirati. Takođe, asimetrični algoritmi omogućuju detekciju izmene poruka tokom transporta. Nedostatak ovih algoritama je (relativna) zahtevnost za izračunavanje, pa su oni sporiji u odnosu na simetrične algoritme. Kod asimetričnih algoritama javlja se problem sa utvrđivanjem vlasništva javnog ključa. Da bi se postigle najbolje performanse, u realnosti se koristi kombinacija ova dva pristupa.

9.7.4. Heš algoritmi i autentifikacija

Heš algoritmi su posebni kriptografski algoritmi koji se koriste kod autentifikacije. Za razliku od algoritama šifrovanja, heš algoritmi su jednosmerni. Heš algoritam za ulaz prihvata poruke različite dužine, a rezultat je vrednost fiksne dužine. Ovi algoritmi su napravljeni tako da za određeni ulaz relativno jednostavno izračunavaju heš vrednost. Osobina koju treba da poseduju heš algoritmi je otpornost na koliziju. To znači da je verovatnoća da dve različite poruke imaju istu heš vrednost jako mala. Druga važna osobina jeste da je za datu heš vrednost teško naći poruku koja joj odgovara. Jedan od poznatih heš algoritama je *MD5* (*Message*

Digest 5) koji je osmislio Ronald Rivest 1992. godine. Rezultat MD5 algoritma je 128-bitna vrednost. Drugi popularan heš algoritam je *SHA* (*Secure Hash Algorithm*). Postoje 4 verzije ovog algoritma: SHA-0, SHA-1, SHA-2 i SHA-3. Prva verzija SHA algoritma nastala je 1993. godine i osmišljena je od strane NSA (*National Security Agency*). SHA-0 algoritam generiše vrednost dužine 160 bitova. Poslednja verzija SHA algoritma objavljena je 2012. godine.

Autentifikacija je proces identifikovanja učesnika u komunikaciji. Asimetrični algoritmi šifrovanja omogućuju autentifikaciju na jednostavan način. Kada korisnik želi da naglasi da je autor poruke, prvo poruku šifruje svojim tajnim ključem, pa zatim javnim ključem primaoca. Kada poruka stigne na odredište, primalac je najpre dešifruje svojim privatnim ključem, pa zatim javnim ključem pošiljaoca. Primalac je sada siguran ko mu je poslao poruku, jer samo jedan pošiljalac ima navedeni par tajnog i javnog ključa.

Kod ovakve metode za autentifikaciju javlja se problem sa brzinom kod šifrovanja velikih poruka. Asimetrični algoritmi su generalno zahtevniji, a ovakav pristup koristi njihovu dvostruku primenu. Zbog toga se koriste heš algoritmi i postupak je sledeći. Najpre se izračuna heš vrednost poruke, pa se zatim ta vrednost šifruje privatnim ključem. Šifrovana heš vrednost se naziva digitalni potpis (*Digital Signature*). Osnovna poruka se proširuje digitalnim potpisom, šifruje se nekim algoritmom šifrovanja (taj algoritam ne mora nužno biti asimetrični) i šalje na određenu destinaciju. Kada poruka stigne, primalac je dešifruje, pronalazi digitalni potpis koji dešifruje javnim ključem pošiljaoca. Na taj način primalac dobija poslatu heš vrednost poruke. On, takođe, izračunava heš vrednost poruke koju je primio. Ako se ove dve heš vrednosti podudaraju, primalac može da bude siguran u to ko mu je poslao poruku.

Osim za autentifikaciju, heš algoritmi se koriste za proveru izmene poruke tokom transporta. Jedna poruka ima uvek istu heš vrednost. Primalac izračunava heš vrednost primljene poruke i upoređuje je sa vrednošću iz

digitalnog potpisa. Ako se radi o različitim vrednostima, može se pretpostaviti da je neko promenio poruku tokom transporta.

Kod autentifikacije se javlja problem utvrđivanja vlasništva javnog ključa. Može se desiti da napadač izabere par ključeva i da se predstavi kao neko sasvim drugi. Ovaj problem se rešava digitalnim sertifikatima (*Digital Certificates*). *Digitalni sertifikati* su „dokumenti" koji povezuju korisnika i javni ključ. Da bi bili legitimni, digitalni sertifikati moraju biti potpisani od strane neke stranke od poverenja. Stranke od poverenja su posebne agencije koje verifikuju digitalne sertifikate i postoji čitav sistem ovih agencija.

Pitanja

9.1. Iz kojih razloga je modul koji je zadužen za upravljanje mrežama poslednji priključen osnovnim modulima operativnih sistema?
9.2. Šta su računarske mreže?
9.3. Da li su internet i veb računarske mreže?
9.4. Odakle je potekla potreba za umrežavanjem?
9.5. Koji su najvažniji kriterijumi za klasifikaciju mreža?
9.6. Kako se mreže mogu podeliti na osnovu tehnologije prenosa?
9.7. Na koji način se računarske mreže mogu klasifikovati na osnovu veličine?
9.8. Po kojim karakteristikama se mogu razlikovati LAN mreže?
9.9. Kakva može biti topologija LAN mreže?
9.10. Kakva može biti fizička topologija LAN mreže?
9.11. Na šta se odnosi logička topologija mreže?
9.12. Koje su karakteristike gradske mreže (MAN)?
9.13. Navesti primere regionalne mreže (WAN).
9.14. Šta je podmreža?
9.15. Na kakvim elementima je izgrađena podmreža?
9.16. Koja je uloga rutera u računarskim mrežama?
9.17. Šta je kombinovana mreža i od čega se sastoji?
9.18. Uz pomoć kojih hardverskih komponenti se implementira kombinovana mreža?
9.19. Koji su najvažniji trenuci u istoriji razvoja interneta?
9.20. Koje su najvažnije stvari o kojima treba voditi računa pri projektovanju računarskih mreža?
9.21. Koje su prednosti slojevite arhitekture računarskih mreža?
9.22. Koji su slojevi referentnog modela ISO-OSI?
9.23. Koji slojevi čine TCP/IP referentni model?
9.24. Navesti najpoznatije protokole sloja aplikacija TCP/IP referentnog modela.
9.25. Koji su najvažniji protokoli transportnog sloja TCP/IP referentnog modela?
9.26. Koje su odlike TCP protokola?

9.27. Koje su odlike UDP protokola?
9.28. Koja je uloga međumrežnog sloja kada je TCP/IP referentni model u pitanju?
9.29. Koja je uloga IP protokola?
9.30. Šta je IP adresa i čemu ona služi?
9.31. Koje vrste IP adresa postoje?
9.32. Od čega se sastoji IP (IPv4) adresa?
9.33. Koliko postoji klasa IP adresa?
9.34. Prodiskutovati prednosti i mane hijerarhijske organizacije IP adresa.
9.35. Koji se razlozi zbog kojih se uvodi koncept IP podmreže?
9.36. Od čega se sastoji UDP poruka?
9.37. Na koji način se odvija komunikacija dva procesa korišćenjem TCP protokola?
9.38. Na koji način funkcioniše i koja je uloga mehanizma prozora?
9.39. Koje su ključne razlike ISO-OSI i TCP/IP referentnih modela?
9.40. Koje su osnovne zamerke na referentni model ISO-OSI?
9.41. Koje su zamerke na referentni model TCP/IP?
9.42. Koji su osnovni bezbednosni problemi kada su savremene računarske mreže u pitanju?
9.43. Kako se mogu podeliti napadi u okviru mreže?
9.44. Koje vrste aktivnih napada postoje?
9.45. Kakve vrste pasivnih napada postoje?
9.46. Analizirati napad skeniranjem portova.
9.47. Kako se izvodi napad prepunjenost bafera?
9.48. Šta su računarski virusi?
9.49. Šta su trojanci?
9.50. Na koji način se šire crvi kroz mrežu?
9.51. Analizirati napad „čovek u sredini“.
9.52. Na koji način se organizuje odbrana od napada na mreži?
9.53. Šta je antivirus?
9.54. Objasniti na koji način funkcioniše tehnika detektovanja potpisa.
9.55. Na čemu se zasniva heuristička detekcija virusa?
9.56. Koja je uloga zaštitnog zida?

9.57. Koje vrste zaštitnih zidova postoje?
9.58. Kakva je uloga kriptografije u zaštiti računarskih mreža?
9.59. Prodiskutovati DES algoritam.
9.60. Na čemu se zasniva RSA algoritam?
9.61. Prodiskutovati prednosti i mane simetričnih i asimetričnih algoritama za šifrovanje.
9.62. Koje su odlike heš algoritama?
9.63. Šta je autentifikacija?
9.64. Šta je digitalni sertifikat?

LITERATURA

[1] A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts, 9th ed., Wiley Global Education, 2012.

[2] A. S. Tanenbaum, Modern Operating Systems, 3rd ed., Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[3] R. A. Finkel, An Operating Systems Vade Mecum, 2nd ed., Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[4] W. Stallings, Operating Systems - Internals and Design Principles, 7th ed., Pitman, 2011.

[5] E. W. Dijkstra, Cooperating sequential processes, *Programming Languages: NATO Advanced Study Institute*, F. Genuys, Ed., Academic Press, 1968, pp. 43-112.

[6] G. L. Peterson, Myths about the mutual exclusion problem, *Information Processing Letters,* vol. 12, no. 3, pp. 115-116, 1981.

[7] L. Lamport, A New Solution of Dijkstra's Concurrent Programming Problem, *Commun. ACM,* vol. 17, no. 8, pp. 453-455, 1974.

[8] E. W. Dijkstra, Een algorithme ter voorkoming van de dodelijke omarming, http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF.

[9] E. G. Coffman, M. Elphick and A. Shoshani, System Deadlocks, *ACM Comput. Surv.,* vol. 3, no. 2, pp. 67-78, 1971.

[10] A. Tanenbaum, Computer Networks, 4th ur., Prentice Hall Professional Technical Reference, 2002.