

RANKO POPOVIĆ

IRINA BRANOVIC

MARKO ŠARAC

## **OPERATIVNI SISTEMI**

---

UNIVERZITET SINGIDUNUM

Beograd, 2011

OPERATIVNI SISTEMI  
prvo izdanje

*Autori:*

Prof. dr Ranko Popović  
Doc. dr Irina Branović  
Marko Šarac

*Recenzenti:*

Prof. dr Dejan Živković  
Prof dr. Boško Nikolić

*Izdavač:*

UNIVERZITET SINGIDUNUM  
Danijelova 32, Beograd  
[www.singidunum.ac.rs](http://www.singidunum.ac.rs)

*Za izdavača:*

Prof. dr Milovan Stanišić

*Tehnička obrada:*

Irina Branović

*Dizajn korica:*

Ime Prezime

*Godina izdanja:*

2011

*Tiraž:*

??? primeraka

*Štampa:*

Mladost Grup  
Loznica

*ISBN:*

# Sadržaj

<b>Predgovor</b> . . . . .	7
<b>I. Operativni sistemi</b>	
<b>Glava 1. Kako radi računar?</b> . . . . .	11
1.1. Osnovne komponente računarskog sistema . . . . .	11
1.2. Arhitektura računarskog sistema . . . . .	11
1.3. Procesor . . . . .	12
1.4. Memorija . . . . .	13
1.5. Ulazno-izlazni uređaji (periferije) . . . . .	16
1.6. Organizacija računara . . . . .	17
<b>Glava 2. Pregled operativnih sistema</b> . . . . .	19
2.1. Operativni sistem iz perspektive korisnika . . . . .	19
2.2. Operativni sistem iz perspektive računara . . . . .	20
2.3. Definicija operativnog sistema . . . . .	20
2.4. Struktura operativnog sistema . . . . .	21
2.5. Kako radi operativni sistem?	22
2.6. Tipovi operativnih sistema . . . . .	26
2.7. Usluge operativnog sistema . . . . .	27
2.8. Sistemske pozivne ruke . . . . .	29
2.9. Struktura operativnog sistema . . . . .	31
2.10. Pokretanje operativnog sistema . . . . .	34
<b>Glava 3. Procesi i niti</b> . . . . .	37
3.1. Šta je proces?	37
3.2. Blok za kontrolu procesa (PCB)	40
3.3. Raspoređivanje procesa . . . . .	42
3.4. Komunikacija između procesa . . . . .	45
3.5. Niti . . . . .	46
<b>Glava 4. Raspoređivanje procesa</b> . . . . .	51
4.1. Kriterijumi za algoritme raspoređivanja . . . . .	51
4.2. Raspoređivanje sa prinudnom suspenzijom procesa . . . . .	52
4.3. Raspoređivanje tipa "prvi došao, prvi uslužen" (FCFS)	52
4.4. Raspoređivanje tipa "prvo najkraći posao" (SJF)	53
4.5. Raspoređivanje po prioritetu . . . . .	53
4.6. Kružno raspoređivanje (RR)	54

4.7. Raspoređivanje redovima u više nivoa . . . . .	55
4.8. Raspoređivanje u multiprocesorskim sistemima . . . . .	55
<b>Glava 5. Sinhronizacija procesa . . . . .</b>	<b>57</b>
5.1. Stanje trke . . . . .	57
5.2. Problem kritične sekcije . . . . .	58
5.3. Zaključavanje . . . . .	60
5.4. Semafori . . . . .	61
5.5. Monitori . . . . .	65
5.6. Klasični problemi sinhronizacije . . . . .	67
5.7. Potpuni zastoj . . . . .	71
<b>Glava 6. Upravljanje memorijom . . . . .</b>	<b>75</b>
6.1. Zahtevi u upravljanju memorijom . . . . .	75
6.2. Podela memorije na particije . . . . .	78
6.3. Straničenje . . . . .	79
6.4. Segmentacija . . . . .	82
6.5. Virtuelna memorija . . . . .	83
6.6. Operativni sistem i upravljanje memorijom . . . . .	90
<b>Glava 7. Sistem datoteka . . . . .</b>	<b>93</b>
7.1. Struktura podataka . . . . .	93
7.2. Osnovne operacije sa datotekama . . . . .	94
7.3. Sistemi za upravljanje datotekama . . . . .	94
7.4. Arhitektura sistemskog softvera za rad sa datotekama . . . . .	95
7.5. Funkcije za upravljanje datotekama . . . . .	96
7.6. Organizacija datoteka . . . . .	97
7.7. Direktorijumi datoteka . . . . .	102
7.8. Deljenje datoteka . . . . .	104
7.9. Zapisi kao blokovi (record blocking) . . . . .	105
7.10. Upravljanje sekundarnim skladištenjem . . . . .	106
7.11. Metode alokacije datoteka . . . . .	108
7.12. Upravljanje slobodnim prostorom . . . . .	110
7.13. Pouzdanost . . . . .	112
<b>Glava 8. Ulazno-izlazni (U/I) sistem . . . . .</b>	<b>113</b>
8.1. Organizacija U/I funkcija . . . . .	113
8.2. Direktan pristup memoriji (DMA) . . . . .	114
8.3. Ciljevi u projektovanju U/I sistema . . . . .	115
8.4. Disk raspoređivanje . . . . .	117
8.5. Algoritmi za raspoređivanje diska . . . . .	119
8.6. Redundantan niz nezavisnih diskova (RAID) . . . . .	122
8.7. Disk keš . . . . .	124
<b>II. Jezgro operativnog sistema Linux</b>	
<b>Glava 9. Uvod u jezgro (kernel) Linuxa . . . . .</b>	<b>127</b>
9.1. Operativni sistem UNIX kao preteča Linuxa . . . . .	127
9.2. Nastanak Linuxa . . . . .	128

9.3. Pregled jezgra operativnih sistema . . . . .	128
9.4. Monolitna jezgra i mikrokerneli . . . . .	130
9.5. Jezgro Linuxa u poređenju sa kernelom klasičnog UNIX-a . . . . .	131
9.6. Verzije jezgra Linuxa . . . . .	132
9.7. Početak rada sa kernelom . . . . .	133
<b>Glava 10. Upravljanje procesima u Linuxu</b> . . . . .	137
10.1. Deskriptor procesa . . . . .	138
10.2. Stanja procesa u Linuxu . . . . .	140
10.3. Kontekst procesa . . . . .	140
10.4. Copy-on-Write . . . . .	141
10.5. Sistemski poziv fork() . . . . .	142
10.6. Niti u Linuxu . . . . .	142
10.7. Završetak procesa . . . . .	144
10.8. Raspoređivanje procesa u Linuxu . . . . .	144
<b>Glava 11. Sistemski pozivi</b> . . . . .	155
11.1. Brojevi sistemskih poziva . . . . .	155
11.2. Upravljač sistemskih poziva . . . . .	156
11.3. Pristup sistemskom pozivu iz korisničkog prostora . . . . .	160
<b>Glava 12. Prekidi i upravljači prekida</b> . . . . .	161
12.1. Upravljači prekida . . . . .	161
12.2. Primena upravljanja prekidima . . . . .	165
<b>Glava 13. Sinhronizacija u kernelu</b> . . . . .	169
13.1. Kritične sekcije i stanje trke . . . . .	169
13.2. Zaključavanje . . . . .	169
13.3. Metodi sinhronizacije u kernelu . . . . .	170
13.4. Spinlok . . . . .	172
13.5. Semafori . . . . .	174
<b>Glava 14. Upravljanje memorijom</b> . . . . .	177
14.1. Stranice . . . . .	177
14.2. Zone . . . . .	178
14.3. Učitavanje stranica . . . . .	179
14.4. Oslobađanje stranica . . . . .	180
14.5. Funkcija kmalloc() . . . . .	180
14.6. Funkcija vmalloc() . . . . .	181
14.7. Statičko alociranje na steku . . . . .	182
14.8. Mapiranje stranica u memoriji sa velikim adresama . . . . .	182
<b>Glava 15. Virtuelni sistem datoteka</b> . . . . .	185
15.1. Interfejs sistema datoteka . . . . .	185
15.2. UNIX sistem datoteka . . . . .	186
15.3. Objekti sistema datoteka i njihove strukture podataka . . . . .	187
15.4. Objekat superblok . . . . .	188
15.5. Objekat inode . . . . .	190
15.6. Objekat dentry . . . . .	192

15.7. Objekat file . . . . .	194
15.8. Strukture podataka pridružene sistemima datoteka . . . . .	196
15.9. Strukture podataka pridružene procesu . . . . .	197

### **III. Primeri operativnih sistema**

<b>Instaliranje i praktična primena virtuelnih mašina</b> . . . . .	201
<b>Operativni sistemi sa komandnom linijom</b> . . . . .	205
DOS - Disk Operating System . . . . .	206
<b>Instaliranje virtuelnog operativnog sistema Windows XP</b> . . . . .	211
<b>Instaliranje virtuelnog operativnog sistema Windows 7</b> . . . . .	219
Opcije za administriranje operativnog sistema Windows 7 . . . . .	225
<b>Instaliranje virtuelnog operativnog sistema Ubuntu 11.04</b> . . . . .	231
<b>Distribucije Linuxa</b> . . . . .	245
Debian . . . . .	246
Backtrack . . . . .	248
<b>Instaliranje operativnog sistema FreeBSD</b> . . . . .	249
<b>Operativni sistemi za servere: Windows Server 2008</b> . . . . .	255
<b>Operativni sistemi za mobilne uređaje</b> . . . . .	257
<b>Dodatak A: Kratak uvod u jezik C</b> . . . . .	259
<b>Dodatak B: Kratak pregled UNIX komandi</b> . . . . .	279
<b>Indeks</b> . . . . .	289
<b>Bibliografija</b> . . . . .	295

## **Predgovor**

Sadržaj ove knjige prilagođen je nastavnom planu i programu istoimenog predmeta na Fakultetu za Informatiku i Računarstvo Univerziteta Singidunum u Beogradu. Autori se nadaju da će knjiga poslužiti i svima onima koji žele da steknu osnovna znanja o operativnim sistemima.

U skladu sa nastavnim planom predmeta, knjiga je podeljena u tri dela. Prvi deo objašnjava osnovne pojmove i principe projektovanja savremenih operativnih sistema. U drugom delu detaljno je opisana implementacija jezgra operativnog sistema Linux, dok treći deo na primerima virtualnih mašina i različitih operativnih sistema ilustruje teorijske principe objašnjene u knjizi.

Čitaocima koji nisu upoznati sa osnovama jezika C namenjen je dodatak u kome će pronaći osnove ovog programskog jezika čije je poznavanje neophodno da bi se upućenost u operativne sisteme mogla nazvati kompletnom.

Zahvaljujemo svima čije su preporuke doprinele da ova knjiga bude bolja, a posebno studentima.

Sve primedbe, komentari i pohvale mogu se uputiti na adresu  
[rpopovic@singidunum.ac.rs](mailto:rpopovic@singidunum.ac.rs).



**Deo I**

## **Operativni sistemi**



## Glava 1

# Kako radi računar?

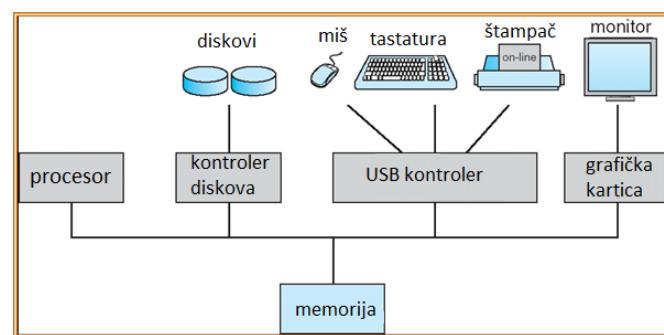
### 1.1. Osnovne komponente računarskog sistema

Računarski sistem može se grubo podeliti u dva dela: hardver i softver.

Hardver, čiji su najvažniji sastavni delovi procesor, memorija, magistrala i ulazno izlazni uređaji, obezbeđuje osnovne resurse za funkcionisanje sistema. Pod softverom se obično podrazumevaju operativni sistem (poznat i kao sistemski softver) i aplikativni softver. Operativni sistem upravlja hardverom i koordinira njegovo deljenje između različitih aplikacija i korisnika. Aplikativni programi (npr. softver za obradu teksta, tabelarna izračunavanja, kompjajleri, Web čitači) rešavaju probleme korisnika. Hardver i softver su logički ekvivalentni; bilo koja operacija koju izvodi softver može se ugraditi direktno u hardver. Takođe, svaka instrukcija koju izvršava hardver može se simulirati u softveru. Presudni faktori za odlučivanje koje će se funkcije implementirati hardverski, a koje softverski jesu cena, brzina, pouzdanost i brzina promene okruženja.

### 1.2. Arhitektura računarskog sistema

Savremen računarski sistem opšte namene sastoji se od jednog ili više procesora, brojnih ulazno-izlaznih uređaja (periferija) povezanih na zajedničku magistralu (bus) koja vodi do memorije (Slika 1.1).



Slika 1.1: Moderan računarski sistem.

### 1.3. Procesor

Za procesor ili CPU (Central Processing Unit) se obično kaže da je mozak sistema, jer je njegova funkcija da izvršava programe koji se nalaze u radnoj memoriji računara. Procesor dohvata instrukcije iz memorije, ispituje ih i zatim redom izvršava. CPU se sastoji iz nekoliko delova. **Kontrolna jedinica** (control unit) dohvata instrukcije iz memorije i određuje kog su tipa. **Aritmetičko-logička jedinica** (ALU, Arithmetic and Logical Unit) izvršava operacije poput sabiranja i logičke konjukcije. Procesor ima i malo memorije koja je veoma brza, a služi da se u njoj čuvaju privremeni rezultati i kontrolne informacije. Tu memoriju čini nekoliko **registara**, od kojih svaki ima svoju namenu. Registri se mogu podeliti u dve grupe: one koji su vidljivi za korisnika, i kontrolno-statusne registre. U prvu grupu spadaju programski brojač i registar instrukcija. Najvažniji registar je **programski brojač** (program counter, PC) koji ukazuje na sledeću instrukciju koja treba da se izvrši. **Registar instrukcija** (instruction register, IR) sadrži instrukciju koja se trenutno izvršava. U grupu registara koji su vidljivi korisnicima spadaju registri podataka koji čuvaju međurezultate izračunavanja, kao i adresni registri u koje se smeštaju adrese i podaci iz radne memorije. Primer adresnog registra je pokazivač steka (stack pointer); detalje o steku potražite u odeljku 1.4.1.

Procesor izvršava svaku instrukciju u nizu malih koraka:

1. Instrukcija se dohvata iz memorije i smešta u registar instrukcija.
2. Programski brojač se menja tako da ukazuje na sledeću instrukciju.
3. Određuje se tip instrukcije koja je upravo dohvaćena.
4. Ako su instrukciji potrebni podaci iz memorije, određuje se gde se oni nalaze.
5. Ako su potrebni podaci, oni se dohvataju iz radne memorije i smeštaju u registre procesora.
6. Instrukcija se izvršava.
7. Rezultati se smeštaju na odgovarajuće mesto.
8. Izvršavanje sledeće instrukcije počinje od tačke 1.

Rani računari imali su samo jedan procesor koji je izvršavao instrukcije jednu po jednu; noviji računari imaju više nezavisnih procesora koji dele zajedničku radnu memoriju. Ovakav pristup zove se **multiprocesiranje** i omogućuje istovremeno izvršavanje nekoliko instrukcija.

Takođe, moderni procesori u izvršavanju instrukcija primenjuju tzv. **pajplajn** (pipeline). U ovom pristupu za svaki od prethodno pomenutih koraka u izvršavanju instrukcija (tj. prenošenje u registar instrukcija, određivanje tipa instrukcije itd.) postoji posebna jedinica koja ga izvršava. Kada se računar pokrene, prva jedinica dohvata prvu instrukciju, druga počinje da je tumači (dekodira), a za to vreme je prva jedinica već zauzeta dohvatanjem sledeće instrukcije. Nešto kasnije, dok je treća jedinica zaposlena ispitivanjem da li početnoj instrukciji trebaju podaci iz memorije, druga jedinica dekoduje drugu instrukciju, a prva jedinica već dohvata treću instrukciju iz

memorije. Pajplajn procesori se mogu uporediti sa proizvodnom trakom; dok provodi na početku linije nisu još ni sastavljeni, oni na kraju su skoro završeni.

## 1.4. Memorija

Programi koje računar izvršava moraju da se nalaze u glavnoj memoriji (RAM ili random access memory). Radna memorija je jedini veliki prostor za smeštanje podataka kome procesor može direktno da pristupa.

Memorije su obično implementirane u poluprovodničkoj DRAM (dynamic RAM) tehnologiji sa nizom memorijskih reči (lokacija), tako da svaka lokacija ima svoju adresu. Ako memorija ima  $n$  lokacija, one će imati adrese od 0 do  $n-1$ . Sve memorijske lokacije sadrže isti broj bitova. Ako lokacija ima  $k$  bitova, ona može da sadrži neku od  $2^k$  mogućih kombinacija podataka. Interakcija sa memorijom obavlja se pomoću niza naredbi `load` i `store`. Naredba `load` prenosi reč iz glavne memorije u interni register procesora, dok naredba `store` prenosi sadržaj iz registra u zadatu lokaciju glavne memorije. Osim eksplicitnih čitanja i pisanja u glavnu memoriju, procesor takođe automatski iz memorije čita naredbe koje treba da se izvrše. Procesor komunicira sa radnom memorijom preko dva registra: adresnog registra memorije (MAR, Memory Address Register) i registra za memorijske podatke (MDR, Memory Data Register).

Idealno bi bilo kada bi se naredbe za izvršavanje i podaci za programe stalno nalazili u glavnoj memoriji, ali to nije moguće iz dva razloga: prvo, glavna memorija je premala da bi u nju stalo sve što je potrebno, i drugo, ona gubi sadržaj čim se isključi napajanje. Zbog toga većina računarskih sistema ima i sekundarne sisteme za čuvanje podataka (secondary storage) koji služe kao "produžetak" glavne memorije. Najčešće se kao uređaj za sekundarnu memoriju koristi magnetni disk, koji čuva i programe i podatke. Većina programa (čitači Weba, kompjajleri, programi za obradu teksta i tabelarna izračunavanja) čuvaju se na disku do trenutka dok se ne učitaju u glavnu memoriju.

U opštem slučaju, međutim, strukture za čuvanje podataka koje smo do sada pomenu (registri, glavna memorija i magnetni disk) čine samo deo sistema za čuvanje podataka. Njemu pripadaju i keš memorija, kompakt diskovi, magnetne trake itd. Osnovni zadatak svih sistema za čuvanje podataka jeste da te podatke čuvaju sve dok oni sledeći put ne zatrebaju. Najvažnije razlike između memorija su u brzini, ceni, kapacitetu i trajnosti podataka. Različiti sistemi za čuvanje podataka u računarskom sistemu mogu se organizovati u hijerarhiju (Slika 1.2). Memorije na višem nivou su skuplje, ali i brže; kako se spuštamo niže, cena po bitu pada, a vreme pristupa raste.



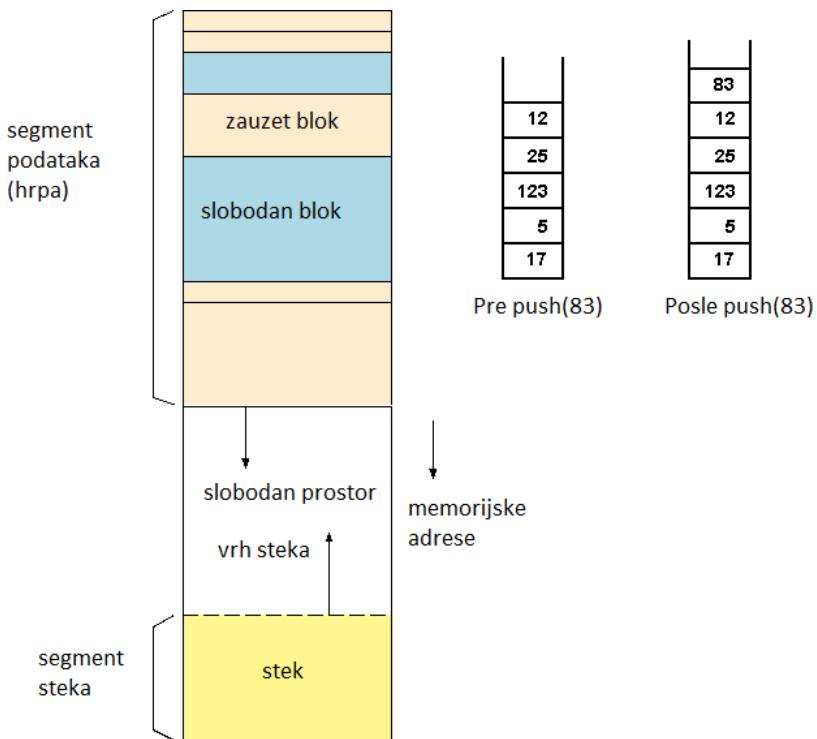
Slika 1.2: Hiperarhija uređaja za čuvanje podataka.

#### 1.4.1. Stek

Stek (stack) je skup sekvencijalnih lokacija u radnoj memoriji koji liči na hrpu papira; na nju se dodaju papiri sa vrha i odatle se i skidaju. U svakom trenutku može se pristupati samo jednoj lokaciji koja se nalazi na vrhu steka; zbog toga se ova struktura zove i LIFO (last-in, first-out).

Stek se implementira tako što se u memoriji odvaja kontinualan skup lokacija u kojima će se čuvati sadržaj steka. Obično je popunjeno samo deo tih lokacija, a ostatak steka se popunjava kako on raste. U radu sa stekom potrebno je znati tri adrese, koje se obično čuvaju u registrima procesora:

- ▷ Pokazivač steka: sadrži adresu vrha steka; njegove vrednosti se menjaju (inkrementiraju/dekrementiraju) naredbama push (stavljanje na stek) i pop (skidanje sa steka).
  - ▷ Osnova steka (stack base): sadrži adresu najniže lokacije u rezervisanom bloku, koja služi i kao prva lokacija za smeštanje elementa u prazan stek. Ako se operacija pop primeni na prazan stek, prijavljuje se greška.
  - ▷ Granica steka (stack limit): sadrži adresu drugog kraja, odnosno vrha rezervisanog bloka. Ako se operacija push primeni na pun stek, prijavljuje se greška.
- Na većini današnjih mašina stek "raste" od viših ka nižim memorijskim adresama, tj. adresa osnove steka ima višu vrednost od adresu vrha steka (Slika 1.3).



Slika 1.3: Radna memorija i stek.

#### 1.4.2. Keš memorije

U svim fazama izvršavanja instrukcija, procesor pristupa memoriji barem jednom, da bi dohvatio instrukciju, a zatim i da bi dobio podatke i(lj) sačuvao rezultate. Pošto procesor radi sa taktom koji je kraći od vremena pristupa memoriji, brzina izvršavanja instrukcije direktno zavisi od vremena pristupa memoriji. Kada bi radna memorija mogla da bude proizvedena u istoj tehnologiji kao procesorski registri, tada bi pomenuta vremena bila približno ista, ali je to nažalost preskupo. Zbog toga se između procesora i radne memorije dodaje malo brze memorije koja se zove **keš memorija** (cache). Keš memorija sadrži kopiju malog dela sadržaja radne memorije; kada procesor zatraži čitanje bajta sadržaja iz memorije, prvo se proverava da li je taj sadržaj već u keš memoriji, i ako jeste, šalje se procesoru. Ako nije, onda se blok bajtova, tj. određen fiksni broj bajtova iz glavne memorije učitava u keš memoriju, a traženi bajt se šalje procesoru. To se radi zato što se najčešće dešava da kada je određeni podatak potreban procesoru, velika je verovatnoća da će mu zatrebat i neki podatak iz neposredne blizine (tzv. princip lokaliteta referenci).

Prilikom projektovanja keš memorije mora se voditi računa o sledećim parametrima:

- ▷ kapacitet keš memorije
- ▷ veličina bloka
- ▷ funkcija za mapiranje
- ▷ algoritam zamene
- ▷ tehnika upisa u radnu memoriju

Čak i vrlo mala keš memorija može primetno da poboljša performanse sistema. Kada je reč o veličini bloka, praktične primene su pokazale sa porastom bloka, broj "pogodaka" (tj. broj podataka koji su upisani u keš a procesor ih je zatražio) na početku raste, ali kada se pređe optimalna veličina bloka, broj "pogodaka" počinje da opada. Verovatnoća da će procesor zatražiti novoupisane podatke je manja od verovatnoće da će zatražiti već korišćene podatke koji moraju da se izbace iz keš memorije da bi se upisali novi blokovi podataka.

Kada se u keš memoriju upiše novi blok, funkcija za mapiranje (preslikavanje) određuje koja će mu lokacija u keš memoriji biti dodeljena; to mora da se obavi tako da se minimizira verovatnoća da će biti zamenjen blok koji će procesoru zatrebati u skorije vreme.

Algoritam zamene u funkciji za mapiranje određuje koji će blok biti zamenjen novim blokom iz memorije kada u keš memoriji više nema mesta za upis novih blokova. Obično se koristi tehnika LRU (least recently used), odnosno zamenjuju se blokovi koji su najduže u keš memoriji a da ih procesor nije zatražio.

Ako procesor promeni sadržaj bloka, on se mora upisati u radnu memoriju pre nego što se zameni drugim blokom. Tehnika upisa diktira da li će se to raditi prilikom svakog ažuriranja blokova, ili samo kada se blok zamenjuje. I u ovom slučaju mora se naći kompromis između efikasnosti i problema sinhronizacije podataka.

## 1.5. Ulazno-izlazni uređaji (periferije)

Već smo pomenuli da računarski sistem čine procesor i brojni kontroleri uređaja koje povezuje zajednička magistrala za podatke. Svaki kontroler je zadužen za različitu vrstu uređaja. Jedan kontroler može da upravlja i sa nekoliko uređaja; takav je na primer, SCSI (small computer systems interface) kontroler. Zadatak kontrolera je da prenese podatke između periferijskih uređaja koje kontroliše i lokalnog bafera. Operativni sistem obično ima **upravljački program** (drayver) za svaki kontroler koji predstavlja uniformni interfejs uređaja ka ostatku operativnog sistema.

Umesto da procesor besposleno čeka dok se ne obavi neka U/I operacija, korišti se tehnika prekida (interrupt). Kada se desi neki događaj, npr. U/I uređaj završi operaciju koju je obavljaо, to se obično signalizira **prekidom** koji stiže od hardvera ili iz softvera. Hardver može da izazove prekid u bilo kom trenutku tako što će poslati signal procesoru, obično preko sistemske magistrale. Softver može da izazove prekid

izvršavanjem specijalne operacije koja se zove **sistemski poziv** (system call ili monitor call).

Kada se procesor "prekine", on prestaje da radi ono što je do tada radio i odmah prelazi na izvršavanje prekidne rutine koja se nalazi na fiksnoj lokaciji u memoriji. Ta lokacija po pravilu sadrži početnu adresu servisne procedure za prekid. Kada se završi izvršavanje servisne procedure, tj. obavi se tzv. obrada prekida, procesor nastavlja rad tamo gde ga je prekinuo.

Da bi počela U/I operacija, upravljački program u odgovarajuće registre kontrolera šalje podatke. Kontroler zatim ispituje sadržaj svojih registara da bi odlučio šta da preduzme (npr. da učita znak sa tastature). Kontroler započinje prenos podataka iz uređaja u svoj bafer; kada se prenos završi, obaveštava upravljački program putem prekida da je završio sa radom. Upravljački program zatim vraća kontrolu operativnom sistemu, kome istovremeno šalje i pokazivač na učitane podatke.

Ovakva vrsta ulaza-izlaza kojim se upravlja pomoću prekida je podesna kada se prenosi mala količina podataka, ali nije pogodna za prenos velike količine podataka, npr. sa diska. Taj problem se rešava direktnim pristupom memoriji (DMA, direct memory access). Nakon postavljanja bafera, pokazivača i brojača za U/I uređaj, kontroler prenosi blokove podataka direktno iz svog bafera u memoriju (ili obratno), bez intervencije procesora. Generiše se samo jedan prekid po bloku, da bi se upravljački program obavestio da je operacija završena, umesto jednog prekida po bajtu koji se generiše pri radu sa sporim uređajima. Dok kontroler uređaja obavlja te operacije, procesor je slobodan za druge poslove.

## 1.6. Organizacija računara

Računarski sistem može da bude organizovan na različite načine, u zavisnosti od broja procesora koje koristi.

### 1.6.1. Jednoprocesorski sistemi

Većina sistema ima samo jedan procesor. Međutim, jednoprocesorski sistemi se međusobno veoma razlikuju, jer mogu biti i džepni i mainframe računari. U svima njima postoji samo jedan glavni, odnosno centralni procesor koji je u stanju da izvršava opšti skup naredbi, uključujući i one koje stižu iz korisničkih procesa. Skoro svi ovakvi sistemi imaju i druge procesore specijalne namene (npr. grafičke procesore, kontrolere za tastaturu i sl.). Ti specijalni procesori izvršavaju ograničen skup naredbi i nisu u stanju da izvršavaju korisničke procese. Ponekad njima upravlja operativni sistem, u smislu da im šalje informacije o novom zadatku ili nadgleda njihov status. Na primer, mikrokontroler diska prima niz zahteva od centralnog procesora i implementira sopstveni algoritam za red podataka i raspoređivanje. Ovakva organizacija oloboda glavnog procesora zadatka raspoređivanja procesa za disk. PC sadrži specijalni

procesor u tastaturi koji konvertuje pritisnute tastere u kôd koji se šalje centralnom procesoru. U suštini, i ako sistem ima više procesora, ali je samo jedan od njih centralni procesor opšte namene, onda se radi o jednoprocесорском систему.

### 1.6.2. Multiprocesorski sistemi

Sistemi sa više procesora, paralelni ili multiprocesorski sistemi (multiprocessor systems) postaju sve zastupljeniji. To su sistemi u kojima dva ili više procesora direktno komuniciraju, dele magistralu, a ponekad i takt, memoriju i periferijske uređaje. Sistemi sa više procesora imaju tri glavne prednosti: brži su, jeftiniji i pouzdaniiji.

Današnji multiprocesorski sistemi postoje u dve varijante. U sistemima koji koriste asimetrično multiprocesiranje (asymmetric multiprocessing) svakom procesoru se dodeljuje poseban zadatak. U tom slučaju glavni (master) procesor kontroliše ceo sistem, dok drugi, podređeni (slave) procesori čekaju uputstva od njega ili im je pak dodeljen specifičan zadatak. Češće se ipak u multiprocesorskim sistemima koristi simetrično multiprocesiranje (SMP) u kome svaki procesor izvršava sve zadatke unutar operativnog sistema. Kod simetričnog multiprocesiranja svi procesori su međusobno jednak, tj. ne postoji master-slave odnos između procesora.

Takođe, u poslednje vreme u dizajnu procesora sve češće se sreće organizacija u kojoj isti procesorski čip sadrži nekoliko jezgara (multicore). To su u suštini multiprocesorski čipovi. Iako se njihova arhitektura razlikuje od običnih jednojezgarnih procesora, oni iz ugla operativnog sistema izgledaju kao nekoliko standardnih procesora.

Najnoviji trend su **blejd serveri** u kojima se u istom kućištu nalazi nekoliko procesora, U/I i mrežnih kartica. Razlika između blejd servera i multiprocesorskih sistema je u tom što se svaki procesor u blejd serveru podiže nezavisno i ima sopstveni operativni sistem. Neki blejd serveri su i multiprocesorski, čime se narušava granica između različitih tipova sistema. U suštini, blejd serveri se sastoje od nekoliko nezavisnih multiprocesorskih sistema.

Još jedan tip sistema sa više procesora je **klaster**. Poput multiprocesorskih sistema, klasterski sistemi sadrže više procesora koji obavljaju obradu. Klasteri se od multiprocesorskih sistema razlikuju po tome što su sastavljeni od nekoliko zasebnih sistema; u tom smislu definicija klastera nije jedinstvena. Ipak, najšire prihvaćena definicija jeste da klasterski sistemi dele prostor za čuvanje podataka i da su blisko povezani putem lokalne mreže (LAN).

## **Glava 2**

# **Pregled operativnih sistema**

Operativni sistem je program koji upravlja hardverom računara, a istovremeno služi i kao osnova za aplikativne programe i kao posrednik između korisnika računara i hardvera. Gledano iz ugla korisnika računara, operativni sistem je interfejs ka računarskom sistemu. Različiti operativni sistemi predstavljaju se različitim interfejsima: neki su projektovani tako da budu udobni za korišćenje, drugi da budu efikasni, a treći pak kombinuju oba pristupa. Gledano iz ugla računarskog sistema, operativni sistem je softver čija je najvažnija uloga da efikasno upravlja sistemskim resursima.

Operativni sistem se može uporediti sa dirigentom orkestra: sam po sebi, on nema nikakvu funkciju, ali obezbeđuje okruženje u kome neko drugi može da radi nešto korisno.

### **2.1. Operativni sistem iz perspektive korisnika**

Iz ugla korisnika, izgled računara se menja u zavisnosti od interfejsa koji mu on predstavlja. Većina korisnika računara sedi ispred PC računara koji se sastoji od monitora, tastature, miša i sistemske jedinice. Takav sistem je projektovan za jednog korisnika koji ima apsolutni monopol nad njegovim resursima. U tom slučaju, operativni sistem je projektovan tako da bude lak za korišćenje, pri čemu je izvesna pažnja poklonjena i performansama sistema, kao i korišćenju resursa.

Neki korisnici sede ispred terminala povezanog sa mejnfrejm računarom; u tom slučaju, istom računaru preko drugih terminala pristupaju i drugi korisnici koji tada dele resurse ili razmenjuju informacije. U takvim slučajevima operativni sistem je projektovan tako da maksimizira korišćenje resursa, odnosno da obezbedi da se svo dostupno procesorsko vreme, memorija i ulazno-izlazni uređaji koriste efikasno, a da pri tom svi korisnici budu približno ravnopravni u njihovom korišćenju.

Postoje i slučajevi kada korisnici koriste umrežene radne stанице povezane sa drugim radnim stanicama ili serverima. Takvi korisnici na raspolaganju imaju sopstvene resurse, ali i deljene resurse kao što su mreža i serveri (npr. serveri za razmenu datoteke, štampanje itd.). U takvim slučajevima operativni sistem je projektovan tako da omogući kompromis između upotrebljivosti na nivou pojedinačnih korisnika i korišćenja resursa.

U poslednje vreme sve češće se koriste razne vrste džepnih računara. Takvi uređaji su pretežno samostalne jedinice namenjene pojedinačnim korisnicima. Neki od njih se umrežavaju, najčešće bežično. Zbog ograničene snage, brzine i interfejsa operativni sistemi za džepne uređaje su projektovani tako da naglasak bude na pojedinačnim korisnicima, ali se vodi računa i o potrošnji baterije.

Neke operativne sisteme uopšte nije briga za to kako će se predstavljati korisnicima. Na primer, računari ugrađeni u kućne uređaje i automobile ponekad imaju numeričke tastature ili lampice koje prikazuju stanje, ali su njihovi operativni sistemi projektovani tako da se izvršavaju bez ikakvog uticaja korisnika.

## 2.2. Operativni sistem iz perspektive računara

Sa tačke gledišta računara, operativni sistem je program koji je najbliži hardveru. U tom smislu, on se može posmatrati i kao upravljač resursima. Računarski sistem ima brojne resurse koji su potrebni za rešavanje problema, na primer procesorsko vreme, prostor u memoriji, prostor za smeštanje datoteka, U/I uređaje itd. Operativni sistem upravlja brojnim i potencijalno suprotstavljenim zahtevima za dodelu resursa i mora da ih dodeli određenim programima i korisnicima tako da sistem u celini radi efikasno i pouzdano. Dodeljivanje resursa je posebno važno kada veći broj korisnika pristupa istom računaru.

Operativni sistem je program kao i svi drugi; ono što ga čini toliko važnijim od svih ostalih jeste njegova namena. To je program koji vodi procesor u korišćenju sistemskih resursa i upravlja vremenom izvršavanja svih drugih programa. Zbog zahteva da budu laki za korišćenje i održavanje, a istovremeno i efikasni, operativni sistemi spadaju u najsloženije programe koji su ikada napravljeni.

Posmatrano iz nešto drugačije perspektive, operativni sistem je i sredstvo za upravljanje različitim U/I uređajima i korisničkim programima. Operativni sistem je zato pravo kontrolni program koji upravlja korisničkim aplikacijama tako da se izbegnu greške i nepravilno korišćenje računara. U tom smislu ima jedno specijalno zaduženje: kontrolu rada ulazno-izlaznih uređaja.

## 2.3. Definicija operativnog sistema

Posle svega što smo rekli, teško je precizno definisati šta je tačno operativni sistem. Takođe, ne postoji opšte prihvaćena definicija šta sve čini operativni sistem. Postoje operativni sistemi koji zauzimaju manje od 1 MB i nemaju čak ni program za obradu teksta, ali isto tako i oni koji zauzimaju gigabajte prostora i nude kompletan grafički interfejs. Zbog toga se najčešće podrazumeva da je operativni sistem program koji se stalno izvršava u računaru (odnosno program koji se stalno nalazi u radnoj me-

moriji računara) i tada se obično zove **kernel** ili **jezgro**, a sve ostalo spada u sistemski odnosno aplikativni softver. Mi ćemo se držati ove definicije.

## 2.4. Struktura operativnog sistema

Operativni sistem obezbeđuje okruženje u kome se izvršavaju programi. Iako se različiti operativni sistemi međusobno razlikuju, ipak svi imaju neke zajedničke karakteristike, o kojima će biti reči u ovom odeljku.

Jedna od najvažnijih osobina modernih operativnih sistema jeste podrška za izvršavanje više programa, tj. **multiprogramiranje** (multiprogramming) ili **multitasking**. U opštem slučaju, isti korisnik ne može sve vreme da zauzima procesor ili U/I uređaje. Paralelnim izvršavanjem poslova povećava se iskorišćenost procesora tako što se poslovi (kôd i podaci) organizuju tako da procesor uvek ima šta da radi. Operativni sistem jednostavno prelazi na izvršavanje drugog posla ako ono što je dotad izvršavao mora da pričeka, na primer zato što se čeka da se oslobođi neki resurs. Kada preuzeti posao mora da sačeka, trenutni posao se zamenjuje novim i tako ukrug. Dogod postoji barem jedan posao koji može da se izvršava, procesor ne čeka neiskorišćen.

Sistemi sa multiprogramiranjem obezbeđuju okruženje u kome se različiti sistemski resursi (npr. procesor, memorija i periferijski uređaji) koriste efikasno, ali ne obezbeđuju interakciju korisnika sa operativnim sistemom. Vremenska raspodela je logičan nastavak multiprogramiranja. U **sistemima sa vremenskom raspodelom** (time sharing), procesor izvršava nekoliko poslova tako što prelazi između njih, ali se prelazi dešavaju toliko često da korisnici mogu da komuniciraju sa svim programima koji se izvršavaju. Operativni sistemi koji podržavaju vremensku raspodelu su interaktivni.

Operativni sistem sa vremenskom raspodelom koristi raspoređivanje zadataka i multiprogramiranje da bi svakom korisniku dodelio mali deo računarskih resursa. Svaki korisnik ima barem jedan zaseban program u memoriji. Program koji se nalazi u memoriji i izvršava se naziva se **proces**. Kada se proces izvršava, on se obično izvršava veoma kratko vreme pre nego što se završi ili zatraži neku U/I operaciju. Ulaz i izlaz mogu biti interaktivni, tj. izlaz se šalje na prikaz korisniku, a ulaz stiže sa tastature, miša ili nekog drugog uređaja kojim upravlja korisnik. Pošto interaktivni ulaz i izlaz obično radi "ljudskom brzinom", zahteva prilično vremena. Zbog toga operativni sistem mora za to vreme procesoru da obezbedi neki drugi posao.

Vremenska raspodela i multiprogramiranje zahtevaju da u memoriji bude istovremeno prisutno nekoliko poslova. Pošto je memorija obično nedovoljno velika da bi primila sve poslove, oni se u početku čuvaju na disku u tzv. redu poslova (job queue). U njemu se nalaze svi poslovi koji čekaju da im se dodeli prostor u radnoj memoriji. Ako je nekoliko poslova spremno za učitavanje u radnu memoriju, a nema mesta za sve, sistem mora da izabere neke od njih. Donošenje takve odluke zove se **rasporedivanje poslova** (job scheduling). I na kraju, paralelno izvršavanje nekoliko poslova

zahmeta da njihov međusobni uticaj bude sveden na najmanju moguću meru, uključujući raspoređivanje procesa, čuvanje podataka na disku i upravljanje memorijom.

Operativni sistem sa vremenskom raspodelom mora da obezbedi prihvatljivo vreme odziva, što se najčešće postiže tehnikom **virtuelne memorije**, koja omogućuje izvršavanje procesa koji se ne nalaze kompletno u memoriji. Najvažnija prednost tehnike virtuelne memorije je to što omogućuje korisnicima da izvršavaju programe koji su veći od postojeće fizičke memorije. Takođe, memorija se na apstraktan način deli na veliki, uniformni prostor za smeštanje podataka, čime se logička memorija koju "vidi" korisnik odvaja od fizičke memorije. Time se programeri oslobođaju brige oko ograničenog kapaciteta memorije.

Sistemi sa vremenskom raspodelom moraju da upravljaju i datotekama i da obezbede mehanizam za zaštitu resursa od neodgovarajućeg korišćenja. Da bi se osiguralo izvršavanje odgovarajućim redom, sistem mora da obezbedi mehanizme za sinhronizaciju poslova i njihovu međusobnu komunikaciju, kao i to da se poslovi ne "zaglave", večno čekajući jedan na drugog.

## 2.5. Kako radi operativni sistem?

Operativni sistemi rade po principu prekida (odeljak 1.5). Ako nema procesa koji bi se izvršavali, U/I uređaja koje bi trebalo opslužiti niti korisnika kojima treba odgovoriti, operativni sistem će strpljivo i neprimetno čekati da se nešto desi. Događaji se najčešće signaliziraju putem **prekida** (interrupt) ili **izuzetka** (exception, trap). Izuzetak se softverski generiše, a prouzrokuje ga greška (npr. deljenje nulom ili pristup nepostojećoj memorijskoj adresi) ili specifični zahtev operativnom sistemu iz korisničkog programa da se dobije neka usluga. Za posebne tipove prekida posebni delovi kôda u operativnom sistemu (tzv. prekidne rutine) određuju koja će akcija biti preduzeta.

### 2.5.1. Dvojni režim rada

Da bi se osiguralo pravilno izvršavanje operativnog sistema, mora se razlikovati izvršavanje kôda operativnog sistema i korisnički definisanog kôda. To je u mnogim računarskim sistemima obezbeđeno hardverski, da bi se znalo u kom režimu rada je procesor.

Potrebno je imati dva različita režima rada: **korisnički režim** (user mode) i **sistemska režim** (kernel mode, system mode, privileged mode). U hardver računara se dodaje bit koji se zove **bit režima** (mode bit) da bi se označio trenutni režim rada: sistemska(0) ili korisnički(1). Pomoću bita režima u stanju smo da razlikujemo zadatku koji se izvršava u okviru operativnog sistema od zadatka koji se izvršava za korisnika. Kada sistem izvršava neku korisničku aplikaciju, nalazi se u korisničkom režimu. Međutim, kada ta korisnička aplikacija zatraži uslugu od operativnog sistema (pomoću

sistemskog poziva), sistem mora da pređe iz korisničkog u sistemski režim da bi udovljiо tom zahtevu.

Kada se sistem podiže, rad započinje u sistemskom režimu. Nakon toga se u memoriju učitava operativni sistem, a zatim sistem prelazi u korisnički režim pokretanjem korisničkih aplikacija. Kad god se desi prekid, hardver prelazi iz korisničkog u sistemski režim (tj. menja status bita režima u 0). To znači da je sistemski režim aktivan kad god operativni sistem preuzeće kontrolu nad računarom, a pre nego što prepusti kontrolu nekoj korisničkoj aplikaciji, kada računar prelazi u korisnički režim (postavljanjem bita režima na 1).

Dvojni režim rada omogućuje zaštitu operativnog sistema od korisnika, kao i zaštitu korisnika jedan od drugog. Ta zaštita ostvaruje se tako što se neke mašinske instrukcije tretiraju kao privilegovane. Hardver omogućuje privilegovanim instrukcijama da se izvršavaju samo u sistemskom režimu. Ako se pokuša sa izvršavanjem privilegovane instrukcije u korisničkom režimu, hardver je neće izvršiti već će je tretirati kao nevažeću i poslati prekid operativnom sistemu da bi on preuzeo kontrolu.

Instrukcija za prelazak u korisnički režim rada je primer privilegovane instrukcije. Među drugim primerima su U/I kontrola, upravljanje tajmerom i prekidima; privilegovanih instrukcija zapravo ima prilično i sa njima ćemo se sretati u nastavku knjige.

Sada možemo da rezimiramo kako izgleda ciklus izvršavanja instrukcije u računarskom sistemu. Početna kontrola je unutar operativnog sistema, gde se instrukcije izvršavaju u sistemskom režimu. Kada se kontrola prenese u korisničku aplikaciju, sistem prelazi u korisnički režim. Na kraju se kontrola vraća u operativni sistem putem prekida, izuzetka, odnosno sistemskog poziva.

Sistemski poziv obično se realizuje u obliku prekida pomoću instrukcije `trap`, iako neki sistemi imaju i posebnu instrukciju `syscall`. Jezgro operativnog sistema (kernel) ispituje prekidnu instrukciju da bi odredio o kom se sistemskom pozivu radi; parametar označava koji tip servisa zahteva korisnički program. Dodatne informacije potrebne za zahtev mogu biti prosleđene u registrima, na steku, ili u memoriji (preko pokazivača na memorijske lokacije prosleđene u registrima). Jezgro proverava da li su parametri ispravni, izvršava zahtev i vraća kontrolu instrukciji koja sledi nakon sistemskog poziva.

Neophodno je obezbediti i kontrolu operativnog sistema nad procesorom. Ne sme da se desi da se korisnički program "zaglavi" u beskonačnoj petlji, da ne šalje sistemske pozive i da nikad ne vrati kontrolu operativnom sistemu. Da bi se postigao taj cilj, koristi se **tajmer**, koji se podešava tako da prekida računar nakon isteka zadatog vremenskog perioda. Taj period može da bude fiksan (npr. 1/60 deo sekunda) ili promenljiv. Kad god otkuca takt, brojač se smanjuje za jedan (dekrementira). Kada brojač odbroji do 0, generiše se prekid.

Pre nego što preda kontrolu korisniku, operativni sistem proverava da li je tajmer podešen za prekid. Ako se tajmer generiše prekid, kontrola se automatski predaje

operativnom sistemu, koji može prema prekidu da se odnosi kao da se radi o fatalnoj greški, ili može programu da dodeli još vremena za izvršavanje. Pri tom je očigledno da instrukcije koje menjaju sadržaj tajmera moraju da budu privilegovane.

### 2.5.2. Upravljanje procesima

Program ne radi ništa ako procesor ne izvršava njegove instrukcije. Kao što smo pomenuli, program koji se izvršava zove se **proces**. Program kome je dodeljeno vreme u sistemu sa vremenskom raspodelom, npr. kompjajler (prevodilac) je primer procesa, isto kao i program za obradu teksta. Sistemski zadatak kao što je slanje izlaza na štampač, takođe može da bude proces (ili deo procesa). Zasad proces možemo posmatrati i kao posao ili program sa vremenskom raspodelom. Procesu su potrebni određeni resursi, uključujući procesorsko vreme, memoriju, datoteke i U/I uređaje, da bi obavio svoj zadatak. Ti resursi se dodeljuju procesoru kada on nastaje ili se alociraju dok se on izvršava. Kada se proces završi, operativni sistem će oslobođiti sve resurse koje je proces koristio.

Treba naglasiti da program sam po sebi nije proces; program je *pasivan entitet*, npr. datoteka na disku, dok je proces za razliku od njega *aktivni entitet*. Proces u jednoj niti ima jedan programski brojač (program counter) koji zadaje sledeću instrukciju koja će biti izvršena (o nitima će biti reči kasnije). Izvršavanje takvog procesa mora biti sekvenčijalno. Procesor izvršava jednu instrukciju procesa za drugom, sve dok se proces ne završi. Štaviše, u istom trenutku može da se izvršava samo jedna instrukcija procesa. Na taj način, iako sa istim programom mogu da budu povezana dva procesa, oni se posmatraju kao dva posebna niza instrukcija za izvršavanje. Višenitni proces (multithreaded process) ima nekoliko programskih brojača, a svaki od njih pokazuje na sledeću instrukciju koja će se izvršavati za datu nit.

Proces je jedinica rada u sistemu. Takav sistem sastoji se od skupa procesa, pri čemu su neki od njih procesi operativnog sistema (oni koji izvršavaju sistemski kôd), a ostatak čine korisnički procesi. Svi ti procesi potencijalno mogu da se izvršavaju konkurentno, tj. istovremeno. U tom smislu, operativni sistem je zadužen za kreiranje i uništavanje korisničkih i sistemskih procesa, suspendovanje i nastavljanje procesa, sinhronizaciju procesa i sl.

### 2.5.3. Upravljanje memorijom

Glavna (radna) memorija je veoma važna za rad modernog operativnog sistema. Radna memorija je (u opštem slučaju jedino) skladište podataka kojima procesor i U/I uređaji mogu brzo da pristupaju. Na primer, da bi procesor obradivao podatke sa diska, ti podaci prvo moraju da se učitaju u radnu memoriju pomoću U/I poziva koje generiše procesor. Na isti način, instrukcije moraju da se nađu u memoriji da bi ih procesor izvršavao.

Da bi se poboljšalo iskorišćenje procesora, računari moraju da drže nekoliko procesa u memoriji, što stvara potrebu za upravljanjem memorijom. Postoje razne metode upravljanja memorijom, a efikasnost svake od njih zavisi od situacije. U pogledu upravljanja memorijom, operativni sistem je zadužen za praćenje ko i koliko koristi delove memorije, odlučivanje koje procese (ili njihove delove) i podatke treba smestiti u memoriju ili izbaciti iz nje, kao i za alociranje i dealociranje prostora u memoriji po potrebi.

#### 2.5.4. Upravljanje prostorom za čuvanje podataka

Operativni sistem kreira apstraktnu jedinicu za čuvanje podataka koja se zove **datoteka** (file). Datoteke su povezane sa fizičkim uređajima kojima se pristupa preko odgovarajućih kontrolera. Upravljanje datotekama je jedna od najvidljivijih komponenti operativnog sistema. Računari mogu da čuvaju informacije na nekoliko različitih vrsta fizičkih medijuma, među kojima su najčešće korišćeni magnetni diskovi, optički diskovi i magnetne trake.

Datoteka je skup povezanih informacija koje definiše njen autor. Datoteke najčešće predstavljaju programe i podatke. Datoteke sa podacima mogu biti tekstualne ili binarne. Pojam datoteke je izuzetno apstraktan, a operativni sistem ga implementira upravljanjem uređajima za čuvanje velike količine podataka. Datoteke se takođe najčešće organizuju u direktorijume da bi se lakše koristile.

Zadaci operativnog sistema u pogledu upravljanja datotekama jesu kreiranje i uništavanje datoteka, kreiranje i uništavanje direktorijuma, podrška za operacije za rad sa datotekama i direktorijumima i sl.

Pošto je radna memorija premala da bi u nju stali svi podaci i programi, i pošto se podaci koji se u njoj nalaze gube kada se isključi napajanje, računarski sistem mora da obezbedi sekundarnu memoriju kao rezervu za glavnu, radnu memoriju. Većina modernih računarskih sistema koristi diskove kao sekundarne medijume za čuvanje programa i podataka. Većina programa čuva se na disku dok se ne učita u memoriju, a zatim se disk koristi i kao izvor i kao odredište podataka. Zbog toga je pravilno upravljanje diskovima od velike važnosti za računarski sistem. U pogledu upravljanja diskovima, operativni sistem je zadužen za upravljanje slobodnim prostorom, alociranje mesta za čuvanje podataka i sl.

Često se koristi i prostor za čuvanje podataka koji je sporiji, veći i jeftiniji od sekundarne memorije. Magnetne trake, CD i DVD diskovi su tipični primer tercijarnih uređaja za čuvanje podataka. Iako tercijarna memorija nije bitna kada je reč o performansama sistema, i ona zahteva upravljanje; neki operativni sistemi preuzimaju i taj zadatak, dok ga drugi prepuštaju korisničkim aplikacijama.

## 2.6. Tipovi operativnih sistema

U zavisnosti od namene, postoje različiti tipovi operativnih sistema čiji je kratak pregled dat u nastavku.

- ▷ **Distribuirani sistemi** (distributed OS): skup fizički razdvojenih, umreženih računarskih sistema koji omogućuju pristup različitim resursima koje sistem održava. Pristup zajedničkim resursima ubrzava rad, poboljšava funkcionalnost, dostupnost podataka i pouzdanost. U distribuiranim sistemima računari ne dele zajedničku memoriju i sistemski sat.
- ▷ **Mrežni operativni sistemi** (network OS): obezbeđuju okruženja u kojima korisnici sa svojih lokalnih mašina mogu pristupati resursima udaljenih sistema na dva načina: procedurom daljinskog prijavljivanja na sistem (remote login) ili razmenom datoteka sa udaljenim sistemom (remote file transfer).
- ▷ **Sistemi za rad u realnom vremenu** (real-time embedded OS): Embedded računari su najčešći tip računara koji se nalazi svuda: od automobila do mikrotalasnih rerni. Obično obavljaju vrlo specifične poslove, a sistemi na kojima se izvršavaju su vrlo ograničeni. Za embedded uređaje razvijaju se specijalni operativni sistemi, vrlo ograničenih mogućnosti, bez korisničkog interfejsa, čiji je glavni zadatak da upravljaju hardverom uređaja. Embedded operativni sistemi gotovo uvek u rade u realnom vremenu, što znači da postavljaju vrlo stroga vremenska ograničenja za rad procesora ili tok podataka. Obrada u realnom vremenu mora da se završi u predviđenom vremenskom roku, jer će sistem u suprotnom zakazati. To je u sasvim drugačije nego u sistemima sa vremenskom raspodelom, u kojima je brz odziv poželjan, ali ne i neophodan.
- ▷ **Multimedijijski sistemi** (multimedia OS): Većina operativnih sistema predviđena je za rad sa tekstualnim datotekama, programima, tabelama i sl. Međutim, u poslednje vreme sve češće se u računarske sisteme ugrađuju multimedijalni podaci koje čine audio i video datoteke. Podaci u njima razlikuju se od "običnih" podataka po tome što se prenose unutar definisanih vremenskih prozora (npr. u slučaju videa, 30 slika u sekundi). Multimedija je opšti pojam za aplikacije koje se danas dosta koriste, npr. MP3 muziku, DVD filmove, video konferencije, animacije, prenos slike uživo preko Weba. Sve češće se multimedija osim na standardnim računarskim sreće i u džepnim uređajima (PDA i "pametnim" mobilnim telefonima) za koje se projektuju specijalni operativni sistemi.
- ▷ **Operativni sistemi za džepne uređaje** (handheld systems): Pod "džepnim" uređajima podrazumevamo lične digitalne pomoćnike (PDA, Personal Digital Assistant) kao što su Palm, PocketPC ili Blackberry uređaji i "pametne" mobilne telefone. Na njima se izvršavaju specijalno projektovani operativni sistemi koji moraju da vode računa o ograničenoj memoriji, ekranu i procesorskoj snazi ovakvih uređaja. Na primer, trenutno vrlo malo džepnih uređaja koristi virtualnu memoriju, pa programeri aplikacija moraju da se dovijaju unutar granica postojeće fizičke memo-

- rije. Ipak, ograničenja u funkcionalnosti džepnih uređaja nadomešćuje njihova pogodnost i prenosivost, pa se oni sve više koriste i razvijaju.
- ▷ **Cloud operativni sistemi** (sistemi “u oblaku”): Radi se o pojednostavljenim operativnim sistemima koji rade kao čitači Weba i obezbeđuju pristup različitim Web aplikacijama. Te aplikacije omogućuju korisnicima da obavljaju prostije zadatke, a da im pri tom nije potreban kompletan operativni sistem. Zbog svoje jednostavnosti, cloud operativni sistem može da se podigne veoma brzo. Ovakvi operativni sistemi projektovani su za netbook računare, mobilne uređaje koji se povezuju sa Internetom i za računare koji se prvenstveno koriste za pristup Internetu. Iz “oblaka” korisnik može lako da podigne glavni operativni sistem. Postoje i tzv. cloud aplikacije koje se umesto na lokalnom disku računara izvršavaju “u oblaku”, odnosno na Internetu; to se zove cloud computing, a da bi se koristilo, potreban je samo računar sa osnovnim operativnim sistemom i čitačem Weba.

## 2.7. Usluge operativnog sistema

Operativni sistem obezbeđuje okruženje za izvršavanje programa u kome programima i njihovim korisnicima nudi određene **usluge** (services). Konkretnе usluge razlikuju se od jednog do drugog operativnog sistema, ali ipak imaju slične osobine. Iz perspektive korisnika, operativni sistem nudi:

- ▷ **Korisnički interfejs:** gotovo svi operativni sistemi imaju korisnički interfejs (user interface, UI). Ovaj interfejs može imati više oblika. Jedan je interfejs komandne linije (command line interface, CLI) u kome korisnici unose tekstualne komande. Drugi je paketni interfejs (batch interface) u kome se komande i direktive za kontrolu tih komandi unose u datoteku koje se zatim izvršavaju. Međutim, najčešće se koristi grafički interfejs (graphical user interface, GUI) u kome postoje prozori kojima se upravlja U/I uređajima, meniji za izbor opcija i tastatura za unos teksta. Neki sistemi nude sve tri vrste interfejsa, ili samo neke od njih. U nekim operativnim sistemima interpreter komandi uključen je u jezgro (kernel). U drugim pak to je specijalan program koji se izvršava kada započne posao ili kada se korisnik prvi put prijavi (u interaktivnim sistemima). U sistemima u kojima postoji više interpretera komandi, oni su poznati i kao ljske (shell). Na primer, u UNIX sistemima korisnik može da bira između *Bourne shell*, *C shell*, *Bourne-Again shell*, *Korn shell* itd. Sve ljske imaju vrlo slične funkcionalnosti pa ih korisnici biraju na osnovu ličnih sklonosti. Tradicionalno, u UNIX sistemima dominira korisnički interfejs iz komandne linije, iako svi oni imaju i neki GUI interfejs (Common Desktop Environment, CDE ili X-Windows česti su u komercijalnim verzijama). Grafički interfejsi su razvijani i u različitim open-source projektima kao što su KDE i GNOME u GNU projektu. Odluka da li će se koristiti grafički ili interfejs iz komandne linije uglavnom zavisi od ličnih sklonosti. Korisnici UNIX-a obično koriste komandni,

a većina korisnika Windowsa grafički korisnički interfejs. Korisnički interfejs je različit u različitim sistemima i obično je odvojen od sâme strukture operativnog sistema. Dizajn korisnog i prijateljskog korisničkog interfejsa stoga nije direktno povezan sa projektovanjem operativnog sistema.

- ▷ **Izvršavanje programa:** Sistem mora da bude u stanju da učita program u memoriјu i da ga izvrši. Program mora da bude u stanju da završi svoje izvršavanje, bilo na uobičajen ili neuobičajen način (kada se desi greška).
- ▷ **U/I operacije:** Program koji se izvršava može da zahteva ulaz/izlaz iz datoteke ili U/I uređaja. Zbog efikasnosti i zaštite, korisnici obično ne mogu da kontrolišu U/I uređaje direktno, već to mora da uradi operativni sistem.
- ▷ **Rad sa sistemom datoteka.** Sistem datoteka je posebno važan. Programi moraju da čitaju i pišu datoteke, da ih kreiraju i brišu, traže i prikazuju informacije o njima. Nekim programima potrebno je da upravljaju dozvolama da bi omogućili ili zabranili pristup datotekama.
- ▷ **Komunikacije.** Česte su situacije u kojima jedan proces treba da razmenjuje informacije sa drugim. Takva komunikacija na primer može da se odvija između procesa koji se izvršavaju u istom računaru ili između procesa koji se izvršavaju na različitim računarskim sistemima u mreži.
- ▷ **Otkrivanje grešaka.** Operativni sistem mora stalno da bude na oprezu od grešaka koje mogu da se dese u procesoru i memorijskom hardveru, U/I uređajima ili korisničkim programima. Za sve vrste grešaka operativni sistem treba da preduzme odgovarajuću akciju da bi obezbedio efikasan i neometan rad.

Iz ugla računarskog sistema, operativni sistem obezbeđuje sledeće usluge:

- ▷ **Alokacija resursa.** Kada postoji više korisnika ili poslova koji se izvršavaju paralelno, za svaki od njih moraju da se odvoje (alociraju) resursi. Operativni sistem upravlja velikim brojem različitih tipova resursa. Za neke (kao što su taktovi procesora, radna memorija i prostor za čuvanje podataka) mora da postoji specijalan kod za alokaciju, dok drugi (kao što su U/I uređaji) imaju mnogo opštijih način zauzimanja i oslobođanja.
- ▷ **Praćenje rada.** Želimo da znamo koji korisnici koriste koje resurse i u kojoj meri. Praćenje zauzetosti resursa može da se iskoristi za statistike o iskorišćenosti sistema, koje nam mogu poslužiti za podešavanje sistema da bi mu se poboljšale performanse.
- ▷ **Zaštita i bezbednost.** Vlasnici informacija koje se čuvaju u višekorisničkom ili umreženom operativnom sistemu mogu da požele da upotrebe te informacije. Kada se nekoliko procesa izvršava paralelno, treba onemogućiti da jedan proces ometa drugi ili da proces ometa operativni sistem. Zaštita podrazumeva kontrolu pristupa svim resursima sistema, a bezbednost zaštitu sistema od spoljašnjih pretnji.

## 2.8. Sistemski pozivi

**Sistemski pozivi** (system calls) obezbeđuju interfejs za usluge koje nudi operativni sistem. Ovi pozivi su u opštem slučaju dostupni kao rutine u jezicima C i C++, iako neki zadaci niskog nivoa (npr. direktni pristup hardveru) ponekad moraju da se pišu u asemblerском jeziku.

Većina programera međutim ne vidi detalje implementacije sistemskih poziva. Projektanti aplikacija prave programe u skladu sa interfejsom za programiranje aplikacija (Application Programming Interface, API). On određuje skup funkcija koje su na raspolaganju programeru aplikacija, uključujući i parametre koji se prosleđuju svim funkcijama i povratne vrednosti koje programer može da očekuje. Tri najčešća korišćena API interfejsa sistemskih poziva su Win32 za Windows sisteme, POSIX za sve varijante UNIX-a, Linuxa i MAC OS-a i Java API za projektovanje programa koji se izvršavaju u Javinoj virtuelnoj mašini.

U pozadini, funkcije koje čine API pozivaju operativni sistem umesto programera aplikacije. Postoji više razloga za to što programeri koriste API umesto da direktno pozivaju operativni sistem. Jedna od prednosti API programiranja jeste prenosivost: programer koji projektuje aplikaciju pomoću API interfejsa očekuje da će ta aplikacija raditi na bilo kom sistemu koji podržava taj API (iako su razlike između sistema često tolike da je to nemoguće). Takođe, stvarni sistemski pozivi često su mnogo detaljniji i sa njima se teže radi nego sa interfejsom.

Sistem za podršku izvršavanju većine programskih jezika (tj. skup funkcija ugrađenih u biblioteke kompjajlera) obezbeđuje **interfejs za sistemske pozive** (system call interface) koji služi kao veza sa sistemskim pozivima koje nudi operativni sistem. Interfejs za sistemske pozive presreće pozive funkcija u API interfejsu i poziva odgovarajući sistemski poziv unutar operativnog sistema. Obično je svakom sistemskom pozivu pridružen broj, a interfejs za sistemske pozive održava tabelu indeksiranu u skladu sa tim brojevima. Interfejs za sistemske pozive zatim obavlja odgovarajući sistemski poziv u jezgru operativnog sistema, a zatim vraća status sistemskog poziva i povratne vrednosti (ako ih ima).

Za prosleđivanje parametara sistemskog poziva operativnom sistemu koriste se tri metode. Najjednostavniji način jeste da se parametri proslede u registrima. Međutim, ponekad se dešava da ima više parametara nego što ima registara. U takvim slučajevima parametri se čuvaju u blokovima, ili u tabeli u memoriji, a adresa bloka se prosleđuje kao parametar u registru. Ovakav pristup koriste operativni sistemi Linux i Solaris. Parametri takođe mogu biti stavljeni na stek operacijom push iz programa, a operativni sistem će ih "skinuti" sa steka operacijom pop. U nekim operativnim sistemima koriste se metode bloka ili steka zato što ne ograničavaju broj niti dužinu parametara koji se prosleđuju.

### 2.8.1. Tipovi sistemskih poziva

Sistemski pozivi mogu se grubo podeliti u pet kategorija:

- ▷ upravljanje procesima
- ▷ rad sa datotekama
- ▷ upravljanje uređajima
- ▷ održavanje informacija
- ▷ komunikacija

Postoje različite kategorije sistemskih poziva (za upravljanje procesima, rad sa datotekama, upravljanje uređajima itd). Radi ilustracije opisaćemo sistemske pozive za upravljanje procesima.

Program koji se izvršava mora da bude u stanju da zaustavi izvršavanje bilo na standardan način (sistemskim pozivom `end`), bilo na nestandardan način (sistemskim pozivom `abort`). Ako se sistemska poziv obavlja u slučaju nestandardnog završetka rada, ili program ima problema i generiše izuzetak (trap), pravi se slika trenutnog stanja memorije (memory dump) i generiše se poruka o grešci. Sadržaj memorije upisuje se u datoteku i ispituje pomoću **debagera** (debugger); to je sistemska program koja programerima olakšava pronalaženje i ispravljanje grešaka u aplikacijama. Proses koji izvršava neki program može da zatraži učitavanje (`load`) ili izvršavanje (`execute`) nekog drugog programa. Ako se kontrola vrati u postojeći program kada se nov program završi, mora da se sačuva memorijска slika postojećeg programa; na taj način smo efikasno napravili mehanizam pomoću koga jedan program može da poziva drugi. Ako oba programa nastave da se izvršavaju konkurentno, napravili smo nov proces u multiprogramiranju. Često postoji poseban sistemska poziv specijalno za tu namenu (`create process` ili `submit job`).

Ako kreiramo nov posao (proces), ili čak i skup procesa, trebalo bi da budemo u stanju da kontrolišemo njihovo izvršavanje. Ta kontrola podrazumeva mogućnost određivanja i resetovanja atributa procesa, uključujući njegov prioritet, maksimalno dozvoljeno vreme za izvršavanje itd. (sistemska poziva `get process attributes` i `set process attributes`). Možda ćemo takođe željeti da okončamo proces koji smo pokrenuli (`terminate process`) ako više nije potreban.

Pošto smo napravili nove procese, možda ćemo čekati dok se ne završe. Možda ćemo željeti da prođe određeno vreme (`wait time`), ili, što je verovatnije, da se desi neki određeni događaj (`wait event`). Prosesi bi trebalo da signaliziraju kada se desi taj događaj (`signal event`).

Postoji i skup sistemskih poziva koristan za debagovanje programa. Mnogi sistemi podržavaju poziv za prikaz sadržaja memorije (`dump`). Sistemska poziv `trace` prikazuje sve izvršene instrukcije, ali ga podržava manji broj sistema.

## 2.9. Struktura operativnog sistema

Složen i veliki softver kao što je moderan operativni sistem mora pažljivo da se projektuje da bi radio pravilno i bio lak za menjanje. Uobičajen pristup ovom problemu je da se taj zadatak izdeli na manje, jasno definisane celine. Već smo ukratko pominjali sastavne delove operativnog sistema; u ovom odeljku pozabavićemo se time kako se one međusobno povezuju u jezgro operativnog sistema.

### 2.9.1. Prosta struktura

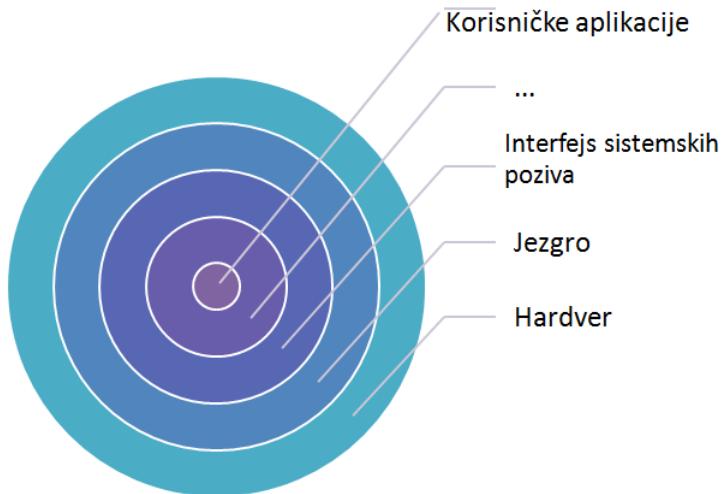
Većina komercijalnih sistema nema dobro definisanu strukturu. Često su oni razvijani tako što su mali, ograničeni sistemi prerasli svoju prvobitnu namenu; primer takvog operativnog sistema je MS-DOS. Kada je nastao, njegovi autori nisu prepostavili da će postati toliko popularan, pa nije pažljivo izdeljen u module. Drugi primer ograničenog strukturiranja je prvobitni UNIX koga je u početku ograničavao hardver. Ovaj operativni sistem sastojao se od dva odvojena dela: jezgra i sistemskih programa. Jezgro je zatim izdeljeno u niz interfejsa i upravljačkih programa za uređaje, koji su se tokom godina razvijali. Na taj u način jezgro je ugrađeno previše funkcija, pa se takva struktura teško implementirala i održavala.

### 2.9.2. Slojevit pristup

Uz odgovarajuću hardversku podršku, operativni sistem može se podeliti u manje delove; takvim pristupom omogućuje mu se da zadrži veću kontrolu nad računaram i aplikacijama koje ga koriste. Sve funkcionalnosti su podeljene u komponente, pri čemu je važno i skrivanje informacija, jer je programerima ostavljena sloboda da implementiraju rutine niskog nivoa onako kako smatraju da treba, pod uslovom da interfejs rutine ostane isti i da ona obavlja ono što treba.

Sistem se može podeliti u module na različite načine. Jedan način je slojevit pristup, u kome se operativni sistem deli u brojne slojeve (layer), osnosno nivoe. Najniži sloj (sloj 0) je hardver, a najviši (sloj N) je korisnički interfejs. Slojevita struktura prikazana je na Slici 2.1.

Sloj operativnog sistema je implementacija apstraktnog objekta koji čine podaci i operacije nad njima. Tipičan sloj operativnog sistema sastoji se od struktura podataka i skupa rutina koje mogu da pozivaju slojevi višeg nivoa.



Slika 2.1: Slojevit operativni sistem.

Najvažnija prednost slojevitog pristupa jeste jednostavnost konstruisanja i debugovanja. Slojevi se prave tako da svaki od njih koristi isključivo funkcije i usluge nivoa ispod. Prvi sloj može da se debaguje nezavisno od ostatka sistema, jer u implementaciji funkcija koristi samo hadrver (za koji se pretpostavlja da je ispravan). Kada se prvi sloj debaguje, može se pretpostaviti da radi ispravno i preći na debagovanje drugog sloja, i tako redom. Svaki sledeći sloj implementira se isključivo korišćenjem operacija prethodnih slojeva. Pri tom, sloj ne mora da zna kako su te operacije implementirane, već samo šta one rade. Na taj način svaki sledeći sloj skriva postojanje određnih struktura podataka, operacija i hardvera.

Najveći problem slojevitog pristupa jeste pravilno definisanje različitih slojeva. Pošto sloj može da koristi samo slojeve nižeg nivoa, pažljivo planiranje je neophodno. Na primer, upravljački program za prostor na disku koji koriste algoritmi za virtuelnu memoriju mora da se nalazi na nižem nivou od rutina za upravljanje memorijom, jer će ga ono koristiti.

Problem kod slojevitog pristupa je i to što je obično manje efikasan od drugih. Na primer, kada korisnički program izvrši U/I operaciju, izvršava se sistemski poziv koji se "hvata" u U/I sloju, koji zatim poziva sloj za upravljanje memorijom, a ovaj zatim zove sloj za CPU raspoređivanje, da bi se na kraju stiglo do hadrvera. U svakom sloju menjaju se parametri, prosleđuju podaci itd. Svaki sloj je dodatno usporavanje sistemskog poziva, a ukupan rezultat je to da poziv traje duže nego u sistemima bez slojeva. Zbog toga se poslednjih godina u projektovanju operativnih sistema teži tome da on ima manje slojeva sa više funkcija; na taj način zadržavaju se prednosti modularnog kôda, a izbegavaju problemi sa definisanjem slojeva i interakcijom između njih.

### 2.9.3. Mikrokerneli

Već smo na primeru operativnog sistema UNIX videli da se sa njegovim razvojem jezgro povećavalo što je otežavalo njegovo održavanje. Sredinom osamdesetih godina razvijen je operativni sistem Mach u kome je jezgro podeljeno u module (mikrokernel). Operativni sistem je strukturiran tako da su iz jezgra uklonjene sve komponente koje nisu presudno bitne za rad; one su zatim implementirane kao sistemske i korisnički programi. Iako nema čvrstog pravila, mikrokernel obično obuhvata samo minimalno upravljanje procesima i memorijom, kao i komuniciranje.

Najvažnija funkcija mikrokernela je da obezbedi komunikaciju između klijentskog programa i različitih servisa koji se takođe izvršavaju u korisničkom prostoru.

Prednost mikrokernela je lako proširivanje operativnog sistema. Svi novi servisi dodaju se u korisnički prostor, pa ne zahtevaju menjanje jezgra. Kada jezgro ne mora da se menja, i izmena ima manje jer je i mikrokernel mali. Takav operativni sistem lakše se prenosi sa jedne hardverske platforme na drugu. Mikrokernel je takođe bezbedniji i pouzdaniji, jer se većina servisa izvršava kao korisnički, a ne sistemske procese.

Nažalost, mikrokerneli nemaju tako dobre performanse zbog opterećenja sistemskim funkcijama.

### 2.9.4. Moduli

Možda najbolja postojeća tehnika projektovanja operativnih sistema podrazumeva korišćenje objektno orijentisanih tehnika za pravljenje modularnog jezgra. U takvom pristupu jezgro ima skup osnovnih komponenata s kojima se tokom podizanja ili rada dinamički povezuju dodatni servisi. Ovakva strategija koristi module koji se dinamički učitavaju i sreće se u modernim implementacijama UNIX-a (kao što su Solaris, Linux i Mac OS).

### 2.9.5. Virtuelne mašine

Slojevit pristup opisan u odeljku 2.9.2 logično je nastavljen uvođenjem **virtuelnih mašina**. Ideja virtuelne mašine jeste da se apstrahuje hardver računara (procesor, memorija, diskovi, mrežne kartice itd.) u nekoliko različitih okruženja za rad, stvarajući na taj način privid da se svako od tih okruženja izvršava u sopstvenom, privatnom računaru.

Korišćenjem raspoređivanja procesa i tehnika virtuelne memorije, o kojima će biti reči kasnije, operativni sistem može da stvori privid da proces ima sopstveni procesor sa sopstvenom (virtuelnom) memorijom. Proses obično ima i dodatne osobine koje ne obezbeđuje sam hardver, kao što su sistemski pozivi i sistem datoteka. Pristup virtuelne mašine ne obezbeđuje nijednu od tih dodatnih funkcionalnosti, već interfejs

koji je identičan postojećem hardveru. Svakom procesu na raspolaganju je (virtuelna) kopija računara na kome se izvršava.

Postoji nekoliko razloga za pravljenje virtuelne mašine, a svi su povezani sa mogućnošću da se deli isti hardver, a da se konkurentno izvršavaju različita okruženja (tj. različiti operativni sistemi). Iako je zamisao virtuelnih mašina korisna, veoma ih je teško implementirati. Najveći problem zapravo je napraviti precizan duplikat fizičke mašine. Prisetimo se da stvarni računar može da radi u dva režima: korisničkom i sistemskom. Softver za virtuelnu mašinu može da radi u sistemskom režimu, pošto je on i operativni sistem. Međutim, sâma virtuelna mašina mora da se izvršava u korisničkom režimu. Međutim, baš kao i fizička mašina, i virtuelna mašina ima dva režima rada. Zbog toga mora da postoji virtuelni korisnički režim i vituelni sistemski režim, koji se izvršavaju u korisničkom režimu stvarne mašine. Operacije koje prouzrokuju prelazak iz korisničkog u sistemski režim na stvarnoj mašini (kao što je sistemski poziv ili pokušaj izvršavanja privilegovane instrukcije) moraju da prouzrokuju prelaz iz virtuelnog korisničkog režima u virtuelni sistemski režim na virtuelnoj mašini.

Taj prelaz može da se ostvari na sledeći način: kada program koji radi u virtuelnom korisničkom okruženju na virtuelnoj mašini obavi sistemski poziv, time pouzrokuje prelazak u monitor virtuelne mašine na stvarnom računaru. Kada se kontrola dodeli monitoru virtuelne mašine, on menja sadržaj registra i brojača instrukcija virtuelne mašine radi simuliranja efekta sistemskog poziva. Zatim vraća kontrolu virtuelnoj mašini, koja je tada u virtuelnom sistemskom režimu.

Najveća razlika u osnosu na normalno izvršavanje operativnog sistema je u brzini. Osim usporenog pristupa U/I uređajima, procesor mora da se multiprogramira između različitih virtuelnih mašina. Prednost virtuelnih mašina s druge strane jeste u tome što su različiti sistemski resursi potpuno zaštićeni: svaka virtuelna mašina je potpuno izolovana od drugih. U trećem delu knjige dati su praktični primeri primene virtuelnih mašina (VMware).

## 2.10. Pokretanje operativnog sistema

Operativni sistem se obično distribuira na disku; za generisanje sistema koristi se specijalan program (SYSGEN) koji čita parametre iz datoteke, od operatera sistema traži informacije o konfiguraciji hardvera, ili pokušava sam da odredi koje su komponente prisutne u sistemu.

Da bi računar počeo da radi, na primer kada se upali ili ugasi i ponovo pokrene, mora da postoji početni program koji će se izvršavati. Taj početni program, koji se zove **bootstrap** program, obično je vrlo jednostavan i čuva se u ROM (read-only memory) ili EEPROM (electrically erasable ROM) memoriji. Poznat je i pod nazivom **firmver** (firmware). Ovaj program inicijalizuje sve aspekte sistema, od registara pro-

cesora, preko kontrolera uređaja do sadržaja memorije. Bootstrap program mora da zna kako da učita operativni sistem tako da on počne da se izvršava. Da bi to postigao, bootstrap program mora da pronađe operativni sistem i da u radnu memoriju učita jezgro (kernel) operativnog sistema. Nakon toga operativni sistem počinje da izvršava prvi proces i čeka da se desi neki događaj. Međutim, kako će hardver znati gde je jezgro operativnog sistema i kako da ga učita? Procedura pokretanja računara učitavanjem jezgra operativnog sistema zove se podizanje (booting). Na većini računara postoji programčić koji se zove bootstrap loader, a služi za lociranje jezgra, njegovo učitavanje u radnu memoriju i pokretanje operativnog sistema. U nekim sistemima, kao što su PC računari, koristi se postupak u dva koraka gde jedan prostiji bootstrap program pronalazi složeniji bootstrap loader na disku, a ovaj zatim učitava jezgro.

Bootstrap program može da obavlja razne zadatke. Jedan od zadataka obično je da pokrene dijagnostiku da bi odredio status maštine. Ako je mašina u redu, program može da nastavi sa koracima za podizanje sistema. Ovaj program takođe inicijalizuje sve aspekte sistema, od procesorskih registara preko kontrolera uređaja do sadržaja radne memorije. Na kraju, pokreće i operativni sistem.

U nekim uređajima, kao što su mobilni telefoni, lični digitalni pomoćnici (PDA) i konzole za igru, čitav operativni sistem nalazi se u ROM memoriji. Čuvanje operativnog sistema u memoriji je pogodno za male operativne sisteme, ali je problem kod ovakvog pristupa to što menjanje kôda bootstrap programa zahteva menjanje hardverskih ROM čipova. U nekim sistemima taj problem se rešava tako što se koristi EEPROM memorija (electrically erasable programmable read-only memory); ova memorija služi samo za čitanje, osim kada joj se eksplisitno ne uputi komanda kojom se omogućuje upis. U opštem slučaju, problem sa ROM memorijom jeste to što je izvršavanje kôda sporije nego izvršavanje kôda u RAM memoriji. Neki sistemi čuvaju operativni sistem u ROM memoriji i kopiraju ga u RAM da bi se brže izvršavao.

U velikim operativnim sistemima (uključujući i većinu operativnih sistema opšte namene kao što su Windows, MAC OS X i UNIX) ili u sistemima koji se često menjaju, bootstrap program se čuva u ROM memoriji, a operativni sistem na disku. U tom slučaju, bootstrap obavlja dijagnostiku i sadrži malo kôda kojim se učitava blok podataka sa fiksne lokacije na disku u memoriju i izvršava se od tog bloka za podizanje (boot block). Program koji se čuva u bloku za podizanje obično sadrži jednostavan kôd koji zna samo adresu na disku i dužinu ostatka bootstrap programa. Na taj način bootstrap program na disku i sâm operativni sistem mogu se lako menjati upisom novih verzija na disk. Disk koji ima particiju za podizanje (a o tome će biti reči u nastavku knjige) zove se **disk za podizanje sistema** ili sistemski disk (boot disk, system disk).



## Glava 3

# Procesi i niti

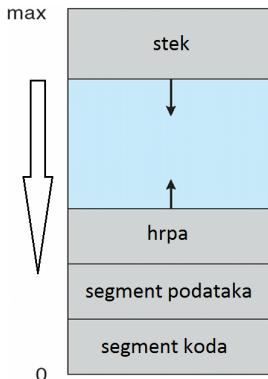
### 3.1. Šta je proces?

Rani računarski sistemi omogućavali su da se u određenom trenutku izvršava samo jedan program. On je imao potpunu kontrolu nad sistemom i pristup svim resursima. Nasuprot tome, današnji računarski sistemi omogućuju da se u memoriji istovremeno nalazi više programa koji se izvršavaju konkurentno. Takav razvoj zahtevaо je čvršću kontrolu i modularnost programa, što je dovelo do pojma procesa. **Proces** (process) je program koji se izvršava, a istovremeno i osnovna jedinica rada u modernim sistemima sa vremenskom raspodelom.

Sistem se dakle sastoji od skupa procesa: postoje procesi operativnog sistema koji izvršavaju sistemski kôd, kao i korisnički procesi koji izvršavaju korisnički kôd. Obe vrste procesa mogu da se izvršavaju konkurentno, pri čemu se procesorsko vreme deli između njih. Dodeljivanjem procesora različitim procesima operativni sistem može da poveća produktivnost računara.

Postavlja se pitanje šta su to aktivnosti procesora. Sistem sa paketnom obradom (batch) izvršava poslove (jobs), dok sistem sa vremenskom raspodelom ima korisničke programe ili zadatke (task). Čak je i u sistemima sa jednim korisnikom kao što je Microsoft Windows, korisnik u stanju da izvršava nekoliko programa istovremeno (npr. program za obradu teksta, čitač Weba i klijent za e-poštu). Čak i kada korisnik može da pokrene samo jedan program, operativni sistem mora da podrži svoje interne aktivnosti, kao što je na primer upravljanje memorijom. Po mnogo čemu sve te aktivnosti su slične, i zovemo ih *procesi*.

Proces je više od kôda; on sadrži i trenutnu aktivnost koju predstavljaju vrednosti programskog brojača i sadržaj registara procesora. Proces u opštem slučaju ima stek na kome se nalaze privremeni podaci (parametri funkcija, povratne adrese i lokalne promenljive), segment kôda (text segment) i segment podataka (data section) koji sadrži globalne promenljive. Proces može da sadrži i dinamičku memoriju ili hrpu (heap), tj. memoriju koja je alocirana tokom izvršavanja procesa. Struktura procesa u memoriji prikazana je na Slici 3.1.



Slika 3.1: Proces u memoriji.

Naglasimo da program sam po sebi nije proces. Program je pasivan entitet, npr. datoteka sa spiskom instrukcija koja se čuva na disku (i često se zove **izvršna datoteka** ili executable file), dok je proces aktivan entitet, sa programskim brojačem koji ukazuje na sledeću instrukciju koja treba da se završi i sa pridruženim skupom resursa. *Program postaje proces kada se njegova izvršna datoteka učita u memoriju.*

Iako sa istim programom mogu biti povezana dva procesa, ti procesi se posmatraju kao nezavisni skupovi instrukcija za izvršavanje. Na primer, nekoliko korisnika može da pokrene različitu kopiju istog programa za e-poštu, ili isti korisnik može da pokrene nekoliko kopija čitača Weba. Svi oni su posebni procesi, pa iako im je segment kôda (odnosno tekstualni deo) isti, podaci, hrpa i stek im se razlikuju. Često se sreće i proces koji tokom izvršavanja obuhvata druge procese; o tome će biti reči kasnije.

### 3.1.1. Stanja procesa

Tokom izvršavanja, proces menja stanja. Stanje procesa delimično je određeno njegovim trenutnim aktivnostima. Svaki proces može da bude u nekom od sledećih stanja:

- ▷ **New:** proces se kreira.
- ▷ **Running:** instrukcije se izvršavaju.
- ▷ **Waiting:** proces čeka da se desi neki događaj (npr. završetak U/I operacije ili dolazak signala).
- ▷ **Ready:** proces čeka da bude dodeljen procesoru.
- ▷ **Terminated:** izvršavanje procesa je završeno.

Nazivi stanja su proizvoljni i menjaju se od sistema do sistema, ali se stanja koja predstavljaju sreću u svim operativnim sistemima. Postoje i operativni sistemi u kojima su stanja procesa još detaljnije razrađena. *Važno je razumeti da u bilo kom trenutku jedan procesor može da izvršava samo jedan proces* (tj. samo jedan od svih procesa

može da bude u stanju *Running*). Međutim, može da postoji mnogo procesa u stanju *Ready* i *Waiting*. Dijagram stanja procesa prikazan je na Slici 3.2.



Slika 3.2: Dijagram stanja procesa.

Stanje *New* odgovara procesu koji je upravo definisan. To znači da je operativni sistem obavio sve neophodne operacije za njegovo kreiranje (npr. dodelio mu je identifikator), ali još nije počeo da ga izvršava, tj. proces još uvek nije u radnoj memoriji već se još uvek nalazi u sekundarnoj memoriji (obično na disku).

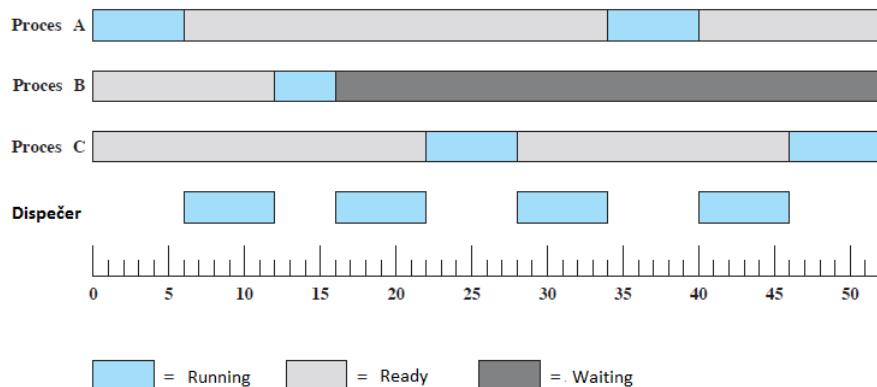
Proces se nalazi u fazi *Terminated* kada se izvrši njegova poslednja instrukcija, kada se izvršavanje prekine zbog fatalne greške ili kada neki drugi proces sa odgovarajućim ovlašćenjem zatraži da se on završi. Proces u stanju *Terminated* više nije kandidat za izvršavanje; operativni sistem još izvesno vreme čuva podatke povezane sa njim da bi pomoćnim programima omogućio da dobiju neophodne informacije, a zatim briše sve podatke o njemu iz sistema.

Dijagram procesa na Slici 3.2 pokazuje da su mogući sledeći prelazi između stanja:

- ▷ *New - Ready*: Procesor je spreman da počne sa izvršavanjem procesa. U većini sistema postoji ograničenje broja procesa koji se izvršavaju, odnosno količine memorije dodeljene procesima, da prevelik broj procesa ne bi ugrozio performanse sistema.
- ▷ *Ready - Running*: Kada dođe trenutak za izbor novog procesa koji će se izvršavati, operativni sistem (odnosno raspoređivač) će izabrati neki od procesa u stanju *Ready*.
- ▷ *Running - Terminated*: Proces koji se trenutno izvršava okončava se ako je izvršena njegova poslednja instrukcija ili ako iz nekog drugog razloga njegovo izvršavanje treba da se završi.
- ▷ *Running - Ready*: Najčešći razlog za ovaj prelaz je to što je proces koji se trenutno izvršava potrošio svo vreme koje mu je dodeljeno za izvršavanje; gotovo svi operativni sistemi sa vremenskom raspodelom podržavaju ovakav pristup. Postoje i

- sistemi u kojima je primenjen prioritet procesa, pa se prelaz između ova dva stanja dešava ako se za izvršavanje prijavi proces sa višim prioritetom.
- ▷ *Running - Waiting:* Proces se šalje u stanje *Waiting* ako zahteva nešto na šta mora da sačeka, npr. uslugu koju operativni sistem ne može trenutno da mu pruži, neki resurs ili U/I operaciju koja mora da se obavi da bi se izvršavanje nastavilo.
  - ▷ *Waiting - Ready:* Ovaj prelaz se dešava kada se desi događaj na koji je proces češkao.

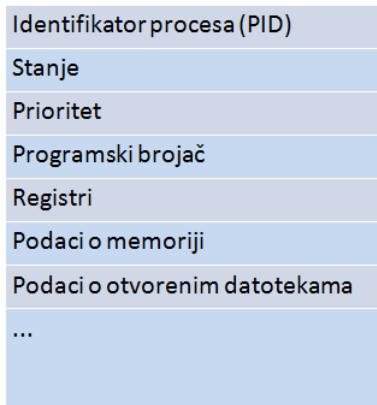
Slika 3.3 ilustruje jedan primer prelaska tri procesa između različitih stanja tokom izvršavanja.



Slika 3.3: Primer prelaska procesa između različitih stanja tokom izvršavanja.

### 3.2. Blok za kontrolu procesa (PCB)

Svaki proces je u operativnom sistemu predstavljen blokom za kontrolu procesa (PCB, Process Control Block), prikazanom na Slici 3.4. **Blok za kontrolu procesa** sadrži sledeće informacije povezane sa procesom:



Slika 3.4: Pojednostavljen blok za kontrolu procesa (PCB).

- ▷ **Identifikator procesa** (process ID, PID): Jedinstveni identifikator koji se pridružuje svakom procesu da bi se on razlikovao od ostalih.
- ▷ **Stanje procesa** (stanje može da bude *New*, *Ready*, *Running*, *Waiting* itd.)
- ▷ **Programski brojač** (brojač sadrži adresu sledeće instrukcije koju proces treba da izvrši)
- ▷ **Sadržaj registara procesora** (Tokom izvršavanja procesa u registrima procesora nalaze se podaci koji se u slučaju prekida moraju sačuvati da bi se izvršavanje procesa kasnije pravilno nastavilo)
- ▷ **Informacije o raspoređivanju** (prioritet procesa, pokazivači na redove za raspoređivanje i svi drugi parametri bitni za raspoređivanje procesa)
- ▷ **Informacije o upravljanju memorijom** (vrednost osnovnih i graničnih registara steka, sadržaj tabela bitnih za upravljanje memorijom, o čemu će biti reči u Poglavlju 6.).
- ▷ **Informacije za praćenje** (utrošeno procesorsko i ukupno vreme, vremenska ograničenja, brojevi korisničkih naloga, brojevi procesa i sl.)
- ▷ **Informacije o U/I statusu** (spisak U/I uređaja dodeljenih procesu, spisak otvorenih datoteka itd.)

Blok za kontrolu procesa (PCB) je najvažnija struktura podataka u operativnom sistemu. Svaki PCB sadrži sve informacije o procesu koje su potrebne operativnom sistemu; ove blokove čitaju i menjaju gotovo svi moduli operativnog sistema, uključujući one za raspoređivanje, alokaciju resursa, obradu prekida itd.

Model procesa koji smo dosad opisivali podrazumeva da je proces program koji se izvršava u jednoj **niti** (thread). Na primer, ako proces izvršava program za obradu teksta, izvršava se jedinstvena nit instrukcija. Ova jedinstvena nit kontrole omogućuje procesu da izvršava samo jedan zadatak u isto vreme. Korisnik ne može istovremeno

da kuca i da pokrene modul za proveru pravopisa u istom procesu. Mnogi moderni operativni sistemi proširili su pojam procesa tako da se on izvršava u više niti čime je omogućeno istovremeno izvršavanje nekoliko zadataka. O tome će biti više reči u nastavku poglavlja.

### 3.3. Raspoređivanje procesa

Cilj multiprogramiranja je da se u svakom trenutku izvršava neki proces, da bi se procesor maksimalno iskoristio. Cilj vremenske raspodele je da procesor prelazi između procesa toliko često da korisnici mogu da komuniciraju sa svakim programom dok se izvršava. Da bi se ostvarili ti ciljevi, **rasporedivač procesa** (process scheduler) bira neki proces u stanju *Ready* (između svih procesa dostupnih za izvršavanje) i šalje ga procesoru na izvršavanje. U jednoprocесорском систему никада се у одређеном trenutku ne izvršava više од једног процеса. Ако има више процеса, сvi остали морају да чекају да се процесор ослободи.

#### 3.3.1. Redovi za raspoređivanje

Kako procesi ulaze u sistem, smeštaju se u **red poslova** (job queue). Procesi koji se nalaze u radnoj memoriji u stanjima *Ready* i *Waiting* čuvaju se u listi koja se zove **red spremnih poslova** (ready queue). Ovaj red se u opštem slučaju čuva kao ulančana lista. Zaglavje reda spremnih poslova sadrži pokazivače na prvi i poslednji PCB u listi. Svaki PCB sadrži polje za pokazivač na sledeći PCB u redu spremnih poslova.

Sistem sadrži i druge redove. Kada se procesu dodeli procesor, on se izvesno vreme izvršava i na kraju završava izvršavanje, desni se prekid ili čeka na određeni događaj, npr. ispunjavanje U/I zahteva. Pretpostavimo da proces zahteva neki deljeni U/I uređaj, kao što je disk. Pošto u sistemu ima mnogo procesa, disk može da bude zauzet U/I zahtevom nekog drugog procesa. U tom slučaju proces mora da čeka da se disk oslobodi. Spisak procesa koji čekaju na određeni U/I uređaj zove se **red uređaja** (device queue). Za svaki uređaj postoji poseban red uređaja.

Nov proces se u početku stavlja u red spremnih poslova gde čeka dok ne bude izabran za izvršavanje. Kada se proces dodeli procesoru i započne izvršavanje, može da se desi neki od sledećih događaja:

- ▷ proces može da zatraži U/I i u tom slučaju se smešta u U/I red
- ▷ proces može da napravi nov potproces i da sačeka da se on završi
- ▷ proces može prinudno da bude uklonjen iz procesora, usled prekida, i da bude враћен u red spremnih poslova.

U prva dva slučaja proces na kraju prelazi iz stanja *Waiting* u stanje *Ready* i vraća se u red spremnih poslova. Proses nastavlja taj ciklus dok se ne završi; tada se uklanja iz svih redova, a PCB i resursi koje je zauzimao se oslobađaju.

### 3.3.2. Rasporedivači

Proces se tokom svog životnog veka kreće kroz različite redove za raspoređivanje. Operativni sistem mora na neki način da bira procese iz tih redova. Biranje procesa za izvršavanje obavlja se pomoću **rasporedivača** (scheduler).

Često se kreira više procesa nego što trenutno može da se izvršava. Ti procesi se smeštaju u uređaj za čuvanje podataka (obično je to disk) radi kasnijeg izvršavanja. **Dugoročni rasporedivač** (long-term scheduler ili job scheduler) bira procese sa diska i učitava ih u memoriju. **Kratkoročni rasporedivač** (short-term scheduler) bira između procesa koji su u stanju *Ready* i nekom od njih dodeljuje procesor.

Najvažnija razlika između ova dva rasporedivača jeste u frekvenciji izvršavanja. Kratkoročni rasporedivač mora često da bira nov proces za procesor. Proces se ponkad izvršava samo nekoliko milisekundi pre nego što biva prinuđen da sačeka na ispunjenje U/I zahteva. Često se kratkoročni rasporedivač pokreće na svakih sto milisekundi, i zbog toga mora da bude brz. Dugoročni rasporedivač se izvršava mnogo ređe; između kreiranja dva procesa može ponekad da prođe i nekoliko minuta. Dugoročni rasporedivač upravlja brojem procesa u memoriji. Pošto prosečna brzina kreiranja mora biti jednaka prosečnoj brzini procesa koji napuštaju sistem, dugoročni rasporedivač treba da se poziva samo kada neki proces napusti sistem. Zbog dužeg intervala između izvršavanja, dugoročni rasporedivač ima na raspolaganju više vremena da odluci koji će proces izabrati za izvršavanje.

Veoma je važno da dugoročni rasporedivač pažljivo bira procese. Postoje procesi koji troše više vremena na U/I operacije, kao i procesi koji ih ređe generišu i većinu vremena provode u izračunavanjima. Važno je da dugoročni rasporedivač izabere dobru kombinaciju ove dve vrste procesa.

### 3.3.3. Promena konteksta

Prekidi prouzrokuju prestanak rada procesora na trenutnom zadatku i prelazak u prekidnu rutinu jezgra. Takve operacije često se dešavaju u sistemima opšte namene. Kada se desi prekid, sistem mora da sačuva **kontekst** procesa koji se izvršava u procesoru da bi mogao da ga povrati kada se završi obrada prekida, što u suštini znači da procesor treba da suspenduje izvršavanje tekućeg procesa i da ga kasnije nastavi. Kontekst je zapravo PCB procesa; on obuhvata vrednosti procesorskih registora, stanje procesa i informacije o upravljanju memorijom. U opštem slučaju, čuva se stanje trenutnog procesa, a zatim se ono rekonstruiše da bi se nastavio rad. Ovaj postupak se zove **promena konteksta** (context switch). Kada se promeni kontekst, jezgro operativnog sistema čuva kontekst starog procesa u bloku za obradu procesa (PCB) i učitava kontekst novog procesa koji treba da se izvrši. Vreme utrošeno na promenu konteksta je čist gubitak, jer sistem u tom periodu ne radi ništa korisno. Tipično vreme promene konteksta je nekoliko milisekundi, a zavisi od brzine memorije, broja registara čija vrednost mora da se kopira i podržanih instrukcija.

### 3.3.4. Kreiranje procesa

Većina operativnih sistema procese razlikuje na osnovu celobrojnog identifikatora (PID). U opštem slučaju, novom procesu će biti potrebni određeni resursi za izvršavanje zadatka. Kada operativni sistem iz bilo kog razloga odluči da kreira nov proces, postupiće po sledećoj šemi:

- ▷ Novom procesoru dodeliće jedinstveni identifikator (PID)
- ▷ Alociraće procesor u memoriji za nov proces
- ▷ Inicijalizovaće PCB novog procesa
- ▷ Staviće nov proces u odgovarajući red (npr. red procesa spremnih za izvršavanje)
- ▷ Napraviće ili ažurirati sve potrebne strukture podataka

Postojeći proces može da kreira nekoliko novih procesa pomoću odgovarajućeg sistemskog poziva. Proces koji pravi nove procese zove se *proces roditelj*, a novi proces zove se potproces ili *proces dete*. Postupak kreiranja novih procesa može se nastaviti tako da se kao rezultat dobije stablo procesa.

Kada proces roditelj napravi potproces, proces dete će resurse moći da dobije direktno od operativnog sistema, ili da bude ograničen da koristi resurse koji su već dodeljeni procesu roditelju. Ograničavanje procesa deteta da koristi resurse procesa roditelja onemogućava situacije u kojima bi neki proces mogao da preoptereti sistem kreiranjem prevelikog broja potprocesa. Kada je reč o izvršavanju procesa, postoje dve mogućnosti: proces roditelj može da nastavi da se izvršava konkurentno s procesima decom, a može i da sačeka dok se neki od procesa dece ne završi. U pogledu adresnog prostora takođe postoje dve mogućnosti: segment kôda procesa deteta može da bude kopija procesa roditelja, a može i da izvršava neki drugi kôd.

Kao primer, proučićemo kako se kreiraju procesi deca u UNIX sistemima. Proses dete se pravi pomoću sistemskog poziva `fork()`. Nov proces sadrži kopiju adresnog prostora procesa roditelja, što olakšava komunikaciju između procesa roditelja i njegovih potprocesa. Oba procesa (roditelj i dete) nastavljaju da se izvršavaju od instrukcije iza sistemskog poziva `fork()`, uz jednu razliku: vrednost koju poziv `fork()` vraća nakon izvršavanja za proces dete je 0, dok je povratna istog sistemskog poziva u procesu roditelju PID procesa deteta.

Nakon sistemskog poziva `fork()`, jedan od dva procesa koristi sistemski poziv `exec()` da bi zamenio memorijski prostor dodeljen procesu novim programom. Sistemski poziv `exec()` učitava binarnu datoteku u memoriju (pri tom uništavajući sliku programa koji je pozvao `exec()` u memoriji) i počinje da je izvršava. Na taj način dva procesa su u stanju da u početku komuniciraju, a zatim podu svaki svojim putem. Proses roditelj nakon toga može da kreira nove procese decu, a ako nema šta da radi dok se proces dete izvršava, može da iskoristi sistemski poziv `wait()` da bi izašao iz reda spremnih poslova dok se izvršavanje procesa deteta ne završi.

### 3.3.5. Završavanje procesa

Proces se završava kada se izvrši njegova poslednja naredba, nakon čega on zahteva od operativnog sistema da ga obriše sistemskim pozivom `exit()`. Tada proces obično vraća procesu roditelju neku celobrojnu vrednost pomoću sistemskog poziva `wait()`. Operativni sistem oslobađa sve resurse koje je proces koristio (memoriju, datoteke, U/I bafere i sl.)

Proces može da zahteva da se neki drugi proces okonča pomoću odgovarajućeg sistemskog poziva. Takav poziv obično može da uputi samo roditelj procesa koji treba da se završi, jer bi u suprotnom korisnici mogli namerno da prekidaju procese drugih korisnika sistema. Proces roditelj mora da zna koji su procesi njegova deca; to je i razlog zbog koga sistemski poziv za kreiranje procesa deteta vraća njegov PID procesu roditelju.

U UNIX sistemima poziv za okončanje izvršavanja je `exit()`. Dešava se da proces roditelj čeka da se izvršavanje procesa deteta završi pomoću poziva `wait()`. Kada se to desi, poziv `wait()` vraća PID okončanog procesa deteta. Ako se u međuvremenu završi proces roditelj, njegovoj deci se kao roditelj dodeljuje proces `init`.

## 3.4. Komunikacija između procesa

Najčešće se koriste dva modela komunikacije između procesa: prosleđivanje poruka i deljenje memorije. U **modelu prosleđivanja poruka** (message passing), procesi razmenjuju poruke, direktno ili indirektno, preko zajedničkog poštanskog sandučeta. Pre nego što komunikacija započne, mora da se uspostavi veza; svaki proces ima ime, koje se prevodi u identifikator (PID) po kome ga operativni sistem prati. To prevođenje obavljuju sistemski pozivi `get hostid` i `get processid`. Identifikatori se zatim prosleđuju opštim pozivima `open` i `close` za sistem datoteka. Proces sa druge strane obično mora da dâ dozvolu za uspostavljanje komunikacije pozivom `accept communication`. Većina procesa koji prihvataju komuniciranje su *demoni* (daemons), tj. sistemski programi napravljeni za tu specijalnu namenu. Oni izvršavaju poziv `wait for connection` i "bude se" kada se uspostavi veza. Izvor komunikacije, poznat i kao *klijent*, i demon koji prihvata komunikaciju, poznat kao *server*, nakon toga razmenjuju poruke pomoću poziva `read message` i `write message`. Poziv `close connection` završava komunikaciju.

U modelu **deljene memorije** (shared memory), procesi koriste sistemske pozive `shared memory create` i `shared memory attach` da bi napravili zajedničku memoriju i dobili pristup delovima memorije koje koriste drugi procesi. Prijetimo se da u opštem slučaju operativni sistem sprečava jedan proces da pristupi memoriji dodeljenoj drugom procesu. Model deljene memorije zahteva da se dva procesa (ili više njih) slože da se to ograničenje ukloni. Nakon toga oni mogu da razmenjuju informacije čitanjem i upisivanjem u deljenu (zajedničku) oblast memorije. Oblik i lokaciju po-

dataka određuju sâmi procesi, tj. to nije pod kontrolom operativnog sistema. Procesi su takođe odgovorni za to da u istu oblast memorije ne upisuju istovremeno.

Oba pomenuta modela često se sreću u operativnim sistemima, a u većini sistema implementirana su oba. Razmenjivanje poruka je pogodno za manju količinu podataka, pošto nema opasnosti od konflikta. Takođe, lakše se implementira nego deljena memorija. S druge strane, deljena memorija omogućuje brže komuniciranje, ali zahteva da se obrati posebna pažnja na zaštitu i sinhronizaciju procesa koji je koriste.

### 3.5. Niti

Dosad smo razmatrali model u kome se svaki proces izvršava u jedinstvenoj niti. Većina modernih operativnih sistema podržava i izvršavanje istog procesa u više niti (multithreading). Pre nego što je uveden pojam niti, za istu svrhu primenjivana je tehnika kreiranja procesa-dece opisana u prethodnim odeljcima. Međutim, kreiranje procesa zahteva dosta vremena i resursa; rad sa nitima je neuporedivo efikasniji.

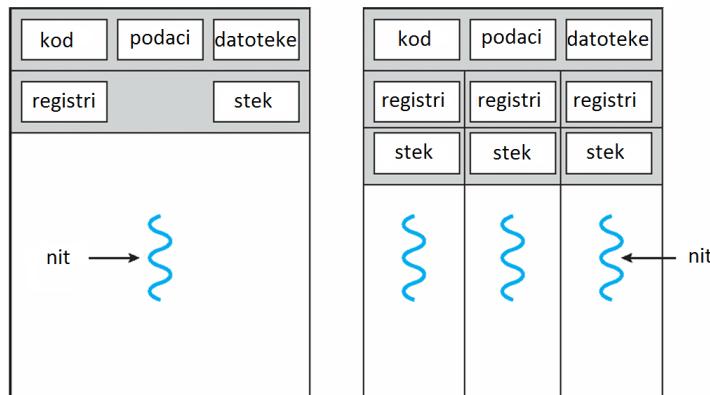
Niti se ponekad zovu i **“laci” procesi** (lightweight processes), za razliku od standardnih procesa koji se nazivaju **“teški” procesi** (heavyweight processes). Nit je mehanizam kojim se obezbeđuje nezavisno izvršavanje jednog zadatka unutar procesa. Više niti se u multiprocesorskom sistemu može izvršavati paralelno, dok se u jedno-procesorskom sistemu može stvoriti privid njihovog istovremenog izvršavanja. Veći broj niti doprinosi većoj interaktivnosti procesa. Na primer, programi za unos teksta omogućuju da se određeni dokument snima na disk dok se istovremeno kuca na tastaturi, jer se sastoji od posebnih niti za snimanje na disk i učitavanje znakova sa tastature.

Nit (thread) je osnovna jedinica korišćenja procesora. Višenitni proces sadrži nekoliko tokova kontrole (tj. niti) unutar istog adresnog prostora. Sve niti istog procesa sa njim dele stanje i resurse, nalaze se u istom adresnom prostoru i pristupaju istim podacima. Svaka nit unutar procesa međutim ima sopstveni identifikator, stanje (*Ready, Running, Waiting* itd.), programski brojač, sadržaj registara i stek. Slika 3.5 ilustruje razliku između standardnog procesa koji se izvršava u jednoj niti i višenitnog procesa.

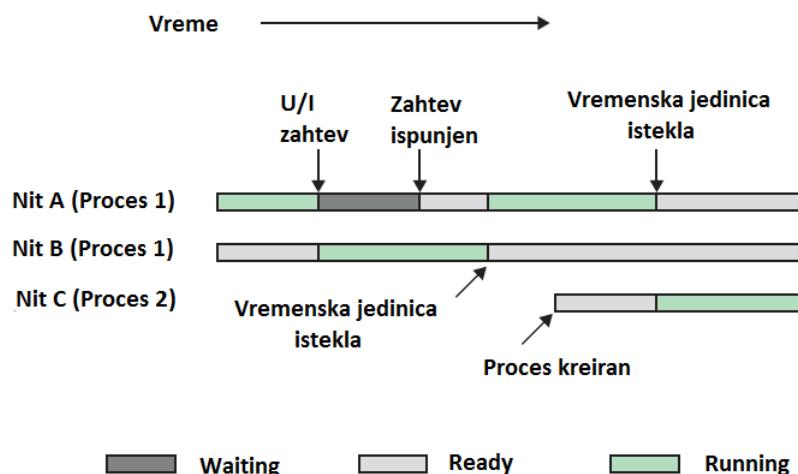
I sâma jezgra modernih operativnih sistema podržavaju višenitni rad. U jezgru je aktivno nekoliko niti, od kojih svaka obavlja specifičan zadatak (npr. obradu prekida ili upravljanje uređajima). Prednosti korišćenja niti nad kreiranjem novih procesa su višestruke:

- ▷ Nova nit se kreira i uništava mnogo brže nego nov proces, što čini programe interaktivnijim
- ▷ Niti omogućuju efikasnije korišćenje resursa
- ▷ Prelazak između niti istog procesa je mnogo brži nego prelazak između različitih procesa

- ▷ Niti omogućuju efikasniju komunikaciju između programa koji se izvršavaju, jer niti unutar istog procesa dele memoriju i datoteke pa mogu da komuniciraju bez posredstva jezgra operativnog sistema.



Slika 3.5: Proces koji se izvršava u jednoj niti nasuprot višenitnom procesu.



Slika 3.6: Primer višenitnog rada na jednom procesoru.

Kao što smo već pomenuli, u sistemu sa jednim procesorom multiprogramiranje omogućuje preplitanje niti u više procesa. U primeru sa Slike 3.6 u procesu se prepliće tri niti u dva procesa. Izvršavanje prelazi sa jedne niti u drugu kada se nit koja se trenutno izvršava blokira ili istekne vremenska jedinica (kvantum vremena) dodeljen procesoru. U ovom primeru, nit C počinje da se izvršava kada istekne vremenski kvantum niti A, iako je u tom trenutku i nit B spremna za izvršavanje. Izbor između niti B i niti C je odluka rasporedivača, o čemu ćemo saznati u sledećem poglavlju knjige.

Pošto niti procesa dele isti adresni prostor i druge resurse, npr. otvorene datoteke, ako jedna nit izmeni resurs, to će uticati na sve druge niti istog procesa. Zbog toga je neophodno sinhronizovati aktivnosti različitih niti da ne bi ometale jedna drugu ili narušavale strukturu podataka. Na primer, ako jedna nit pokuša da doda element u dvostruko ulančanu listu, taj element može da bude izgubljen ili struktura liste može da bude narušena. Problemi sinhronizacije niti u suštini su isti kao problemi sinhronizacije procesa i o njima će biti reči u nastavku knjige.

### 3.5.1. Korisničke i sistemske niti

Operativni sistem može da podržava niti na nivou korisnika ili u jezgru. Korisničkim nitima upravlja se bez podrške jezgra, dok nitima jezgra direktno upravlja operativni sistem. Skoro svi operativni sistemi podržavaju niti jezgra (kernel threads).

U slučaju korisničkih niti, posao upravljanja nitima obavlja aplikacija, a jezgro uopšte ne zna za postojanje niti. Svaka aplikacija može da bude programirana tako da bude višenitna uz pomoć odgovarajuće biblioteke. **Biblioteka niti** (thread library) je interfejs (API) za kreiranje i uništavanje niti, razmenu podataka i poruka između njih, raspoređivanje izvršavanja niti, čuvanje i rekonstruisanje konteksta niti.

Aplikacija standardno počinje izvršavanje u jednoj niti; aplikacija i njena nit dodeljene su jedinstvenom procesu kojim upravlja jezgro operativnog sistema. U svakom trenutku tokom izvršavanja aplikacija može da kreira novu nit koja će se izvršavati u istom procesu, što se postiže pomoću odgovarajuće funkcije u biblioteci niti. Biblioteka pravi novu strukturu podataka za nit, a zatim prepušta kontrolu nekoj od niti procesa koja se nalazi u stanju *Ready*, pridržavajući se utvrđenog algoritma raspoređivanja niti.

Jezgro operativnog sistema uopšte ne zna za te aktivnosti, i nastavlja da radi sa višenitnim procesom na standardan način, raspoređujući ga u redove u zavisnosti od stanja. Na Slici 3.7 koja ilustruje način rada korisničkih i sistemske niti, slučaj (b) prikazuje "čiste" niti jezgra. Jezgro održava informacije o kontekstu procesa u celini, kao i informacije o kontekstu pojedinačnih niti unutar procesa. Raspoređivanje unutar jezgra obavlja se na nivou niti, čime se prevazilaze dva važna ograničenja pristupa sa Slike 3.7 (a) kada se koriste samo korisničke niti. Prvo, jezgro može istovremeno da raspoređuje više niti istog procesa na nekoliko procesora. Drugo, ako je jedna nit u procesu blokirana, jezgro može da pošalje drugu nit istog procesa na izvršavanje. Takođe, sâme rutine jezgra mogu da budu višenitne. S druge strane, korišćenje niti jezgra zahteva prenos kontrole sa jedne niti na drugu unutar istog procesa, a to zahteva promenu režima rada jezgra. Neki operativni sistemi koriste kombinovan pristup, prikazan na Slici 3.7 (c), koji primenjuje i korisničke i niti jezgra. U takvom pristupu, niti se kreiraju u korisničkom prostoru, gde se obavlja i najveći deo posla sinhronizacije i raspoređivanja niti u aplikaciji. Više korisničkih niti iste aplikacije mapiraju se na (manji ili jednak) broj korisničkih niti. Programer može da podešava broj korisničkih

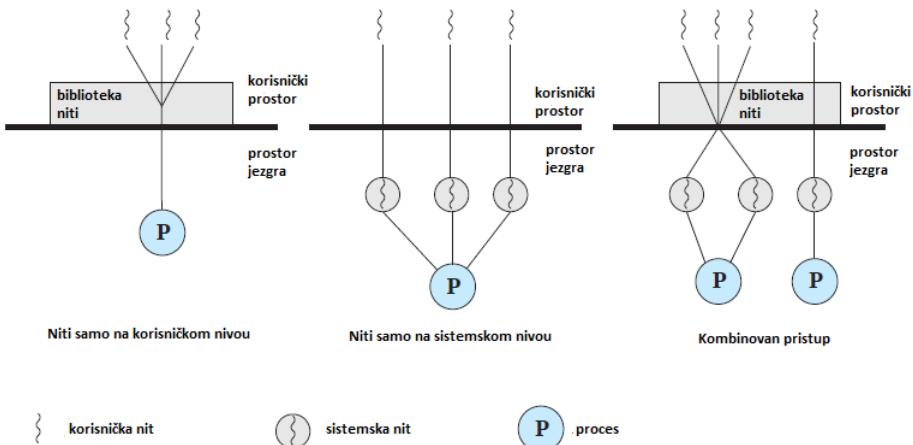
niti za određenu aplikaciju i mašinu da bi postigao najbolje rezultate. Ovakav pristup je primenjen u operativnom sistemu Solaris i omogućuje da se više niti unutar iste aplikacije paralelno izvršava na više procesora (Slika 3.7).

Dakle, u zaključku, nabrajamo prednosti korišćenja korisničkih niti:

- ▷ Menjanje niti ne zahteva privilegije izvršavanja u sistemskom režimu jer se sve strukture podataka za upravljanje nitima nalaze u adresnom prostoru jednog procesa.
- ▷ Raspoređivanje se može implementirati za svaku aplikaciju posebno. U zavisnosti od toga o kom programu se radi, u nekim slučajevima biće pogodnije kružno raspoređivanje, a u drugim princip prioriteta (Poglavlje 4).
- ▷ Korisničke niti mogu da se izvršavaju na bilo kom operativnom sistemu i ne zahtevaju nikakve izmene jezgra.

Ipak, korisničke niti imaju i dva velika nedostatka:

- ▷ Mnogi sistemski pozivi su blokirajući; ako korisnička nit obavi sistemski poziv, biće blokirane sve niti procesa.
- ▷ Višenitna aplikacija ne može da iskoristi prednosti multiprocesiranja. Jezgro deljuje jednom procesu samo jedan procesor, što znači da se unutar procesa može izvršavati samo jedna nit.



Slika 3.7: Korisničke i sistemske niti.



## Glava 4

# Raspoređivanje procesa

U jednoprocesorskim sistemima, procesor u određenom trenutku može da izvršava samo jedan proces; svi drugi procesi moraju da čekaju dok ne dođu na red. U multiprocesorskim sistemima, više procesa može da se izvršava istovremeno (parallelno). Da procesor ne bi bio besposlen ako proces koji trenutno izvršava čeka na neki događaj, uvedena je tehnika multiprogramiranja u kojoj se procesor naizmenično dodeljuje aktivnim procesima. U multiprogramiranju, proces se izvršava dok ne istekne vreme koje mu je dodeljeno za izvršavanje, ili dok ne dođe u situaciju da mora da čeka na neki događaj.

Dodela procesora po nekom algoritmu je jedna od najvažnijih funkcija operativnog sistema. U ovom poglavlju proučićemo najčešće korištene tehnike za raspoređivanje procesa, tj. algoritme za dodelu procesa procesoru.

### 4.1. Kriterijumi za algoritme raspoređivanja

Različiti algoritmi za raspoređivanje procesa imaju različite osobine i o tome treba voditi računa prilikom izbora. Postoji više kriterijuma po kojima se oni mogu porebiti, a najvažniji su sledeći:

- ▷ **Iskorišćenje procesora** (CPU utilization): Procesor treba da bude što je moguće više iskorišćen; u realnim okolnostima, opterećenje se kreće između 40 i 90 procenata.
- ▷ **Propusnost sistema** (throughput): Podrazumeva broj procesa koji se izvršavaju u jedinici vremena. Za dugačke procese, može biti jedan proces na sat, a za brze operacije i po deset procesa u sekundu.
- ▷ **Ukupno vreme potrebno za izvršavanje procesa** (turnaround time): Sa tačke gledišta određenog procesa, važno je koliko ukupno traje njegovo izvršavanje. Ukupno vreme izvršavanja procesa je zbir perioda provedenih u čekanju da proces uđe u radnu memoriju, čekanja u redu spremnih poslova, izvršavanja u procesoru i obavljanja U/I operacija.
- ▷ **Ukupno vreme u redu čekanja** (waiting time): Algoritam za raspoređivanje procesa ne utiče na vreme u kome proces obavlja U/I operacije, već samo na vreme

- koje proces provodi u redu čekanja. Vreme čekanja je zbir vremenskih perioda provedenih u redu čekanja na procesor.
- ▷ **Vreme odziva** (response time): U interaktivnom sistemu ukupno vreme izvršavanja nije najbolji kriterijum. Često proces može da prikaže korisniku neke rezultate relativno brzo, a da zatim nastavi rad. Zbog toga vreme od slanja zahteva do prvog odziva može da bude korisna mera.

#### 4.2. Raspoređivanje sa prinudnom suspenzijom procesa

Kad god procesor postane besposlen, operativni sistem mora da izabere neki od procesa iz reda spremnih procesa i da ga pošalje na izvršavanje. Taj posao obavlja **kratkoročni rasporedivač** (short-term scheduler). Red spremnih poslova u opštem slučaju ne mora da bude realizovan kao FIFO (first-in, first-out) red; on može da bude i stablo ili neuređena ulančana lista. Međutim, bez obzira kako je implementiran, svaki red spremnih poslova kao elemente ima blokove za kontrolu procesa (PCB).

Stanja u kojima se aktivira kratkoročni rasporedivač su sledeća:

1. Prelaz procesa iz stanja *Running* u stanje *Waiting* (na primer, zato što je proces zatražio U/I operaciju ili čeka da se izvrši proces-dete)
2. Prelaz iz stanja *Running* u stanje *Ready* (na primer, kada se desi prekid)
3. Prelaz iz stanja *Waiting* u stanje *Ready* (na primer, zato što je obavljena U/I operacija)
4. Prelaz u stanje *Terminated* (kada se izvršavanje procesa završi)

Kada se raspoređivanje dešava samo u situacijama 1 i 4, kažemo da se radi o **raspoređivanju bez prinudne suspenzije procesa** (nonpreemptive ili cooperative scheduling). Kod ovakvog tipa raspoređivanja, kada se proces dodeli procesoru na izvršavanje, on zadržava procesor sve dok se ne završi njegovo izvršavanje ili dok ne pređe u stanje *Waiting* (tj. desi se situacija 1 ili 4).

Kod **raspoređivanja sa prinudnom suspenzijom procesa** (preemptive scheduling), rasporedivač u bilo kom trenutku može da prekine izvršavanje tekućeg procesa i da dodeli procesoru na izvršavanje neki drugi proces koji čeka u stanju *Ready*.

#### 4.3. Raspoređivanje tipa “prvi došao, prvi uslužen” (FCFS)

Algoritam tipa “prvi došao, prvi uslužen” (FCFS, First Come First Served) je najprostiji algoritam za raspoređivanje, u kome se procesi dodeljuju procesoru onim redom kojim pristižu u red čekanja. Red čekanja je standardni FIFO (First In, First Out) red u kome se PCB procesa koji je upravo stigao u red čekanja stavlja na kraj reda (liste), a procesor se dodeljuje procesu koji se nalazi na početku liste.

Ovaj algoritam se vrlo lako implementira, ali je srednje vreme čekanja u redu vrlo dugačko. Ono zavisi i od trajanja pojedinačnih procesa, ali i od trenutka njihovog

ulaska u sistem. Kod algoritma FCFS moguća je i pojava konvoj efekta, kada svi procesi u redu čekaju da se završi jedan proces koji dugo traje. Ovaj algoritam obično se implementira bez prinudnog suspendovanja procesa jer ne poštuje prioritete procesa već samo vreme dolaska u red čekanja. To znači da se proces koji je dodeljen procesoru izvršava do kraja, ili do trenutka dok ne pređe u stanje *Waiting* zbog čekanja na U/I operaciju. Zbog toga je FCFS raspoređivanje izrazito nepovoljno za sisteme sa vremenskom raspodelom (time-sharing) u kojima je važno da se svakom korisniku dodeljuje procesor u regularnim intervalima.

#### 4.4. Raspoređivanje tipa “prvo najkraći posao” (SJF)

U algoritmu tipa “prvo najkraći posao” (SJF, Shortest Job First) za sve procese u redu čekanja procenjuje se vreme potrebno za izvršavanje, a procesor se dodeljuje onom procesu kome treba najmanje vremena za izvršavanje. Na one procese iz reda koji imaju ista procenjena vremena izvršavanja primenjuje se algoritam FCFS.

Osnovni nedostatak ovog algoritma je to što sistem ne može unapred da zna koliko će procesi trajati, tj. ne može da proceni ukupno trajanje procesa. Procena koliko će proces trajati zasniva se na činjenici da se proces sastoji od više ciklusa korišćenja procesora (CPU burst) koji se prekidaju U/I operacijama. Trajanje sledećeg ciklusa korišćenja procesora procenjuje se na osnovu trajanja prethodnih ciklusa korišćenja procesora.

SJF je optimalno raspoređivanje ako je kriterijum srednje vreme čekanja, zato što daje minimalno srednje vreme čekanja za dati skup procesa.

Razlikuju se dve vrste SJF algoritma, u zavisnosti od toga da li su implementirani sa prinudnim suspendovanjem procesa ili bez njega. Ove dve varijante se ponašaju različito kada se nov proces pojavi u redu čekanja dok se tekući proces izvršava. Varijanta SJF bez prinudnog suspendovanja uvek će završiti tekući proces, bez obzira da li se u redu pojavio nov proces. U varijanti SJF sa prinudnim suspendovanjem, ukoliko je vreme potrebno za izvršenje novog procesa kraće od vremena potrebnog za završetak aktivnosti tekućeg procesa, procesor će biti dodeljen novom procesu. SJF bez prinudnog suspendovanja naziva se raspoređivanje po najmanjem preostalom vremenu (Shortest Remaining Time, SRT).

#### 4.5. Raspoređivanje po prioritetu

U prioritetnim algoritmima svakom procesu se dodeljuje prioritet, a nakon toga algoritam dodeljuje procesor onom procesu čiji je prioritet najveći. Prioritet je celobrojna vrednost, pri čemu je usvojeno da manji broj znači veći prioritet (tj. vrednost 0 predstavlja maksimalan prioritet). Ako su procesi istog prioriteta, onda se odluka donosi po principu FCFS.

Raspoređivanje tipa "prvi došao, prvi uslužen" (FCFS) je specijalan slučaj prioritetnog algoritma u kome je prioritet obrnuto proporcionalan trajanju sledećeg procesorskog ciklusa procesa. To znači da duži procesi imaju manji prioritet i obrnuto.

Prioritetni algoritmi mogu biti implementirani sa prinudnim suspendovanjem procesa ili bez njega. U slučaju implementacije sa suspendovanjem, izvršavanje tekućeg procesa se prekida ako se pojavi proces višeg prioriteta (a provera se obavlja kad god se pojavi nov proces). Ako je algoritam implementiran bez suspendovanja, nov proces će biti postavljen na početak reda.

Najveći problem prioritetnih algoritama je mogućnost neograničenog blokiranja ili "izgladnjivanja" (starvation). U prioritetnom algoritmu može se desiti da neki procesi niskog prioriteta večno čekaju, tj. da im nikad ne bude dodeljen procesor. To se dešava u opterećenom sistemu u kome se stalno pojavljuju novi procesi visokog prioriteta. Rešenje za problem neograničenog blokiranja je "starenje" procesa (aging); to je tehnika kojom se postupno povećava prioritet procesa koji dugo čekaju u sistemu.

#### 4.6. Kružno raspoređivanje (RR)

Kružni algoritam (Round Robin, RR) je posebno projektovan za sisteme sa vremenskom raspodelom. Ovaj algoritam podseća na FCFS, ali se za prelaz između procesa koristi prinudno suspendovanje (preemption). Prvo se zadaje vremenska jedinica (time quantum ili time slice) koja ima vrednost između 10 i 100 milisekundi. Red spremnih poslova implementira se kao kružni red; kratkoročni raspoređivač prolazi kroz red spremnih poslova i redom dodeljuje procesor svakom od njih u trajanju od jedne vremenske jedinice.

Red spremnih poslova u ovom slučaju je FIFO red u kome se novi procesi dodaju na kraj reda; kratkoročni raspoređivač bira prvi proces iz reda spremnih procesa, podešava tajmer tako da generiše prekid nakon isteka vremenske jedinice i šalje proces na izvršavanje.

Ako izvršavanje procesa traje manje od vremenske jedinice, sâm proces će oslobiti procesor i raspoređivač će izabratи sledeći proces iz reda. U suprotnom, tajmer će generisati prekid, procesor će promeniti kontekst, a proces će biti poslat na kraj reda spremnih poslova. Raspoređivač će nakon toga dodeliti procesor prvom procesu iz reda.

Kružno raspoređivanje ima najbolje vreme odziva (response time), a veće srednje vreme završetka procesa (turnaround time) u odnosu na algoritam SJF. Takođe, performanse ovog algoritma veoma zavise od toga kolika je vremenska jedinica. Ako je ona veoma dugačka, onda se kružni algoritam svodi na FCFS, a ako je jako kratka, algoritam se zove i deljenje procesora (processor sharing). U praktičnim primenama vremenska jedinica se definiše tako da približno 80% procesorskih operacija (CPU burst) bude kraće od nje.

#### 4.7. Raspoređivanje redovima u više nivoa

Ova klasa algoritama za raspoređivanje razvijena je za slučajeve kada se procesi mogu lako klasifikovati u grupe. Na primer, često se procesi dele u interaktivne (foreground) i pozadinske (background, batch). Ove dve klase procesa imaju različite zahteve u pogledu vremena odziva, pa im odgovaraju različiti algoritmi za raspoređivanje.

U raspoređivanju pomoću redova u više nivoa (multilevel queue scheduling), red spremnih poslova deli se u nekoliko zasebnih redova, pri čemu se procesi trajno pridružuju nekom od njih prema različitim kriterijumima (zauzetoj memoriji, prioritetu i sl.). Na svaki od redova primenjuje se poseban algoritam za raspoređivanje. Na primer, sistem sa interaktivnim i pozadinskim poslovima imao bi dva reda u koje bi se procesi delili po tipu; red interaktivnih poslova bio bi raspoređivan kružnim algoritmom, a red pozadinskih poslova FCFS algoritmom.

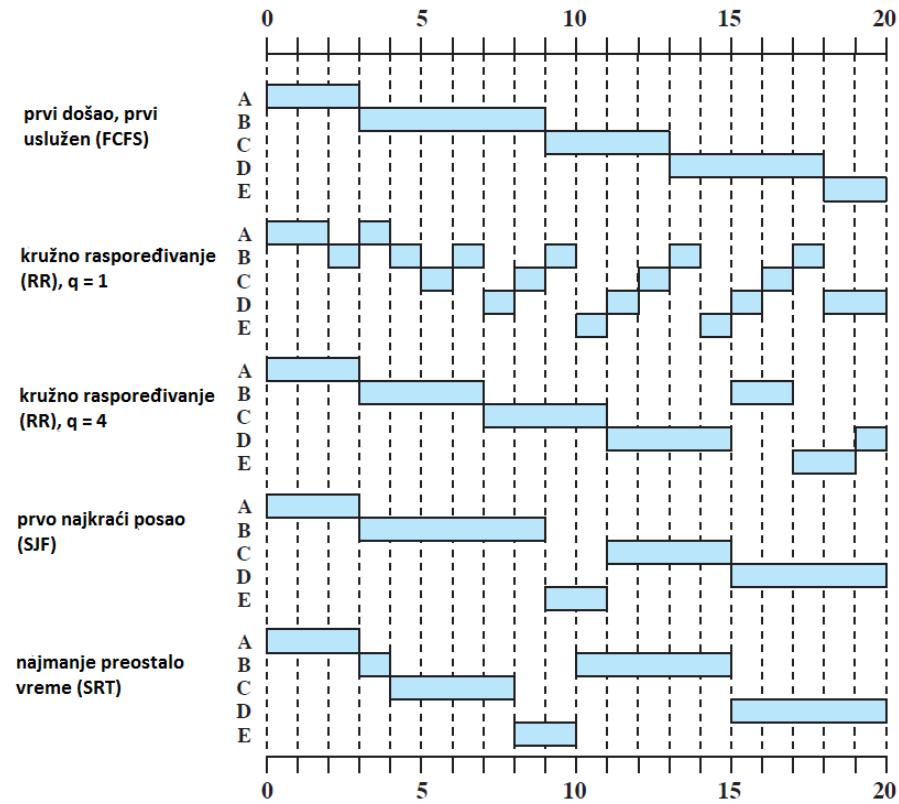
Ovakav način raspoređivanja zahteva i dodatno raspoređivanje između redova, koje se po pravilu realizuje kao prioritetno raspoređivanje sa prinudnom suspenzijom procesa. Na primer, red interaktivnih poslova mogao bi da ima absolutni prioritet nad redom pozadinskih poslova.

#### 4.8. Raspoređivanje u multiprocesorskim sistemima

Dosad smo razmatrali samo raspoređivanje u sistemima sa jednim procesorom. Kada je na raspolaganju više procesora, pojavljuje se i mogućnost deljenja opterećenja (load sharing), ali se usložnjava i problem raspoređivanja procesa. Kao i u slučaju raspoređivanja u jednoprocesorskim sistemima, postoji više mogućnosti, a koji algoritam je najbolji često zavisi od okolnosti.

Jedan od mogućih pristupa problemu rasporedovanja u multiprocesorskim sistemima je da se ceo posao donošenja odluka, U/I obrade i drugih sistemskih aktivnosti dodeli glavnom (master) procesoru. Drugi, podređeni (slave) procesori izvršavaju samo korisnički kôd. To je tzv. asimetrično multiprocesiranje (asymmetric multiprocessing) i ono je prilično jednostavno jer samo jedan procesor pristupa sistemskim strukturama podataka, te nema potrebe za razmenom podataka.

Drugi pristup je simetrično multiprocesiranje (symmetric multiprocessing), u kome svaki procesor vodi računa o raspoređivanju. Svi procesi mogu da se nalaze u zajedničkom redu spemnih poslova, a može da postoji i poseban red za svaki procesor. Međutim, ako svi procesori pokušavaju da pristupaju istoj strukturi podataka i ažuriraju je, raspoređivač mora da bude pažljivo napravljen.



Slika 4.1: Grafički dijagram različitih metoda raspoređivanja procesa.

## **Glava 5**

# **Sinhronizacija procesa**

U jednoprocesorskom sistemu sa multiprogramiranjem, procesi se prepliću u vremenu da bi se stvorio privid istovremenog izvršavanja. Iako to zapravo nije stvarno paralelno izvršavanje, i mada se gubi vreme na menjanje procesa koji se izvršavaju, multiprogramiranjem se povećava efikasnost rada. U sistemu sa više procesora, osim što je izvršavanje procesa moguće preplitati, oni se mogu i preklapati tokom izvršavanja. Na prvi pogled, čini se da su preplitanje i preklapanje potpuno različiti tipovi izvršavanja i da zbog toga prouzrokuju drugačije probleme. Međutim, obe tehnike se mogu posmatrati kao primeri konkurentne obrade, što znači da su problemi isti. U slučaju sistema sa jednim procesorom, problemi proističu iz toga što se relativna brzina izvršavanja procesa ne može predvideti jer zavisi od aktivnosti drugih procesa, od načina na koji operativni sistem radi sa prekidima i od politike raspoređivanja procesa. Kako budemo objašnjavali probleme koji proističu iz istovremenog izvršavanja procesa u sistemima sa jednim procesorom, postaće jasno da se isti problemi pojavljuju i u multiprocesorskim sistemima.

U prethodnim poglavljima razvili smo model sistema u kome se izvršava nekoliko procesa ili niti koji mogu i da dele podatke i resurse. Posledica mogućnosti konkurentnog pristupa zajedničkim podacima može da bude nekonzistentnost tih podataka. Na primer, zamislimo slučaj u kome dva procesa žele da sačuvaju neku vrednost u istoj memorijskoj lokaciji. Operativni sistem mora da obezbedi mehanizme kojima će sprečiti konkurentan pristup procesa istoj lokaciji. To je problem sinhronizacije procesa.

### **5.1. Stanje trke**

Stanje trke (race condition) se dešava kada nekoliko procesa ili niti upisuju i čitaju podatke na taj način da konačan rezultat zavisi od redosleda izvršavanja instrukcija u različitim procesima. Objasnićemo ovu pojavu na primerima.

Prepostavimo da dva procesa, P1 i P2, dele globalnu promenljivu a. U nekom trenutku izvršavanja, proces P1 ažurira vrednost promenljive a na 1, a isto tako u nekom trenutku svog izvršavanja proces P2 ažurira a na 2. To znači da su ova dva

procesa "u trci" za upis u promenljivu a. U ovom primeru, vrednost promenljive a odrediće proces koji ju je poslednji ažurirao.

Sada zamislimo dva procesa, P1 i P2 koji dele globalne promenljive b i c sa početnim vrednostima b=1 i c=2. U nekom trenutku izvršavanja proces P1 izvršava naredbu  $b=b+c$ , a isto tako u nekom trenutku izvršavanja proces P2 izvršava naredbu  $c=b+c$ . Uočite da dva procesa ažuriraju različite promenljive. Međutim, konačne vrednosti promenljivih zavisiće od redosleda u kome se izvršavaju naredbe dodele. Ako P1 prvi izvrši naredbu dodelje, konačne vrednosti promenljivih biće b=3 i c=5. Ako P2 prvi izvrši dodelu, konačne vrednosti promenljivih biće b=4 i c=3.

Situacije kao što su ove upravo opisane često se dešavaju u operativnim sistemima kada njegovi različiti delovi rade sa resursima. Zbog toga je važno proučiti tehnike sinhronizacije procesa.

## 5.2. Problem kritične sekcije

Zamislimo sistem koji se sastoji od više procesa. Svaki od tih procesa ima deo kôda koji se zove **kritična sekcija** (critical section) u kome procesi menjaju zajedničke promenljive, ažuriraju tabelu, upisuju u datoteku i sl. Kada jedan proces izvršava kritičnu sekciju kôda, nijednom drugom procesu ne bi smelo da bude dozvoljeno da izvršava istu kritičnu sekciju, odnosno ne sme se dozvoliti da dva procesa istovremeno izvršavaju kritičnu sekciju. Problem kritične sekcije je kako projektovati protokol koji će procesi koristiti za saradnju. Svaki proces mora da zatraži dozvolu da uđe u svoju kritičnu sekciju. Deo kôda u kome se implementira taj zahtev zove se ulazna sekcija (entry section). Iza kritične sekcije nalazi se izlazna sekcija (exit section), a ostatak je preostala sekcija (remainder section). Opšta struktura tipičnog procesa izgleda ovako:

```
do {
    ulazna sekcija
    kritična sekcija
    izlazna sekcija
    preostala sekcija
} while (true);
```

Rešenja za problem kritične sekcije moraju da zadovolje tri važna zahteva:

1. Međusobno isključivanje: Ako jedan proces izvršava svoju kritičnu sekciju, onda nijedan drugi proces ne sme da izvršava svoju kritičnu sekciju.
2. Ako nijedan proces ne izvršava kritičnu sekciju, a neki procesi žele da uđu u nju, onda samo oni procesi koji ne izvršavaju svoje preostale sekcije mogu da učestvuju u odlučivanju koji će biti sledeći proces koji će ući u kritičnu sekciju, i ta odluka ne sme da se odlaže beskonačno.

3. Procesi kojima se dozvoljava da uđu u kritične sekcije mogu da uđu u njih ograničen broj puta nakon što drugi proces podnese zahtev za ulazak u kritičnu sekciju, a pre nego što se taj zahtev odobri.

Prepostavićemo da se svaki proces izvršava nekom brzinom većom od nule, ali ne možemo prepostaviti ništa u vezi relativne brzine svih procesa koji se izvršavaju.

U određenom trenutku u sistemu može da bude aktivno mnogo procesa u jezgru. Usled toga kôd jezgra operativnog sistema može da se nađe u stanju trke. Kao primer, zamislimo strukturu jezgra koja održava listu svih otvorenih datoteka u sistemu. Ta lista mora da se ažurira kada se nova datoteka otvoriti ili zatvori, i to dodavanjem elemenata u listu ili uklanjanjem elemenata iz nje. Ako bi se desilo da dva procesa istovremeno otvaraju datoteku, posebna ažuriranja liste prouzrokovala bi stanje trke. Zbog toga projektanti operativnog sistema moraju da obezbede da se to ne desi.

Obratite pažnju da se u operativnim sistemima koji ne primenjuju prinudnu suspenziju procesa (non preemptive) stanje trke ne može da se pojavi, jer se svaki proces izvršava do kraja, kada dobrovoljno predaje kontrolu procesoru. Stanje trke aktuelno je samo u operativnim sistemima sa prinudnom suspenzijom procesa (preemptive), koji se moraju pažljivo projektovati. Jezgra sa prinudnom suspenzijom procesa omogućuju programiranje u realnom vremenu i odziv im je mnogo kraći jer procesi jezgra po pravilu ne mogu da se izvršavaju tokom dugo da bi drugi procesi dugo čekali. Na primer, Windows XP koristi jezgro bez prinudne suspenzije, dok Solaris i Linux (od verzije jezgra 2.6) primenjuju prinudnu suspenziju procesa.

### 5.2.1. Petersonovo rešenje problema kritične sekcije

Ovo klasično rešenje je dobro za razumevanje složenosti problema i ograničeno je na dva procesa koji naizmenično izvršavaju svoje kritične sekcije i preostale sekcije. Rešenje zahteva korišćenje dve promenljive:

```
int red;
bool fleg[2];
```

Promenljiva `red` označava čiji je red da uđe u kritičnu sekciju, odnosno `red == i` znači da proces  $P_i$  ulazi u svoju kritičnu sekciju. Niz `fleg` koristi se za označavanje da je proces spreman da uđe u kritičnu sekciju, odnosno ako je `fleg[i]==true`, to znači da je proces  $P_i$  spreman da uđe u kritičnu sekciju. Petersonovo rešenje se može predstaviti sledećim pseudokodom:

```
do {
    fleg[i] = true;
    red = j;
    while(fleg[i] && red == j);
        kritična sekcija
    fleg[i] = false;
        preostala sekcija
} while(true);
```

Da bi ušao u kritičnu sekciju, proces  $P_i$  prvo postavlja  $fleg[i]$  na true, a zatim postavlja red na  $j$ , čime stavlja do znanja drugim procesima da mogu da zahtevaju ulazak u kritičnu sekciju. Ako dva procesa  $P_i$  i  $P_j$  pokušaju da uđu u kritičnu sekciju približno u isto vreme, onda će promenljiva  $red$  biti postavljena na  $i$  i na  $j$  približno istovremeno. Samo jedna od te dve dodele će potrajati, dok će druga biti odmah prepisana. Konačna vrednost promenljive  $red$  odrediće kojem će od ta dva procesa biti dozvoljeno da uđe u kritičnu sekciju.

Proces  $P_i$  može da uđe u svoju kritičnu sekciju samo ako je  $fleg[j]==false$  ili je  $red==i$ . Takođe, ako dva procesa istovremeno izvršavaju svoje kritične sekcije, onda je  $fleg[i]==fleg[j]==true$ . Iz prethodnog sledi da oba procesa nisu mogla uspešno da izvrše while naredbu približno istovremeno jer vrednost promenljive  $red$  može da bude ili  $i$ , ili  $j$ . Dakle, jedan od procesa (na primer  $P_j$ ) je uspešno izvršio naredbu while, a  $P_i$  je morao da izvrši barem još jednu naredbu ( $red=j$ ). Pošto su u tom trenutku  $fleg[j]==true$  i  $red==j$ , ovaj uslov će važiti sve dok je  $P_j$  u svojoj kritičnoj sekciji. To znači da važi međusobno isključenje.

Razmotrimo sada druga dva uslova koja mora da zadovolji rešenje za problem kritične sekcije. Proces  $P_i$  može da se spreči da uđe u kritičnu sekciju samo ako je "zaglavljen" u while petlji sa uslovima  $fleg[j]==true$  i  $red==j$ . Ako  $P_j$  nije spremjan da uđe u kritičnu sekciju, onda je  $fleg[j]==false$ , pa  $P_i$  ulazi u kritičnu sekciju. Ako je  $P_j$  postavio  $fleg[j]$  na true i izvršava se u svojoj while petlji, onda je  $red==i$  ili  $red==j$ . Ako je  $red==i$ , onda će  $P_i$  ući u kritičnu sekciju. Međutim, kada  $P_j$  izade iz svoje kritične sekcije, postaviće  $fleg[j]$  na false, što će procesu  $P_i$  omogućiti da uđe u kritičnu sekciju. Ako  $P_j$  resetuje  $fleg[j]$  na true, mora i da postavi  $red$  na  $i$ . Tako, pošto  $P_i$  ne menja vrednost promenljive  $red$  dok izvršava naredbu while,  $P_i$  će ući u kritičnu sekciju nakon najviše jednog prolaska  $P_j$  kroz kritičnu sekciju.

### 5.3. Zaključavanje

Bilo koje rešenje problema kritične sekcije zahteva jednostavnu alatku koja se zove **brava** (lock). Stanje trke se sprečava tako što se zahteva da kritične sekcije budu zaštićene bravom. Proces mora da dobije bravu pre ulaska u kritičnu sekciju da bi je zaključao, a oslobođa bravu kada izade iz nje.

```
do {
    zaključavanje brave
        kritična sekcija
    otključavanje brave
        preostala sekcija
} while(true);
```

Postoje različita rešenja za problem kritične sekcije koja koriste zaključavanje; neka su softverska, a neka hardverska. U nastavku ćemo proučiti neka od njih.

Mnogi moderni računarski sistemi imaju specijalne hardverske instrukcije koje omogućuju korisnicima da promene jednu reč ili da zamene dve reči na nedeljiv (atomic) način, odnosno u jednom nedeljivom koraku. Te specijalne instrukcije mogu se koristiti za rešavanje problema kritične sekcijske na relativno jednostavan način. Kao primer, razmotrimo instrukciju `swap()` koja nedeljivo zamenjuje sadržaj dve reči:

```
do {
    ključ = true;
    while(ključ == true)
        swap(brava, ključ);
        kriticna sekcija
    brava = false;
    preostala sekcija
} while(true);
```

Ako računar podržava instrukciju `swap()`, onda se međusobno isključenje procesa može postići na sledeći način. Deklariše se globalna logička promenljiva `brava` koja se inicijalizuje na `false`. Osim toga, svaki proces ima lokalnu logičku promenljivu `ključ`. Treba ipak obratiti pažnju da ovo rešenje zadovoljava uslov međusobnog isključenja, ali ne zadovoljava uslov ograničenog čekanja na ulazak u kritičnu sekciju.

## 5.4. Semafori

Hardverska rešenja problema kritične sekcijske kao što su nedeljive instrukcije programerima su prilično teška za korišćenje. Da bi se taj problem prevazišao, koristi se tehniku **semafora** (semaphore).

Semafor `S` je celobrojna promenljiva kojoj se, osim inicijalizacije, pristupa samo preko dve nedeljive (atomic) operacije: `wait()` i `signal()`. Operacija `wait()` se definiše kao:

```
wait(S) {
    while S <= 0; //ne radi ništa
    S--;
}
```

a operacija `signal()` kao:

```
signal(S) {
    S++;
}
```

Svaka promena celobrojne vrednosti semafora mora da se obavi nedeljivo, odnosno kada jedan proces menja vrednost semafora, nijedan drugi proces ne sme istovremeno da je menja. Takođe, testiranje celobrojne vrednosti semafora u operaciji `wait()` i njegovo eventualno menjanje (`S--`) moraju da se obave nedeljivo.

Operativni sistemi razlikuju **brojačke semafore** (counting semaphores) i **binarne semafore** (binary semaphore). Vrednost brojačkog semafora može da bude proizvoljna pozitivna vrednost, dok vrednost binarnog semafora može da bude 0 ili 1. U nekim sistemima binarni semafori se zovu **mutexi** (mutex), jer omogućuju međusobno isključivanje (**mutual exclusion**).

Binarni semafori mogu se iskoristiti za rešavanje problema kritične sekcije za više procesa. Pretpostavimo da n procesa deli semafor, `mutex`, inicijalizovan na 1; rešenje problema je sledeće:

```
do{
    wait(mutex);
    kritična sekcija
    signal(mutex);
    preostala sekcija;
} while(true);
```

Semafore možemo da iskoristimo i za rešavanje drugih problema sinhronizacije. Na primer, pretpostavimo da proces P1 izvršava naredbu S1, a proces P2 naredbu S2, i da je zahtev da se naredba S2 izvrši tek pošto se izvrši naredba S1. Problem se može rešiti tako što će P1 i P2 deliti semafor `synch` koji će biti inicijalizovan na nula. U proces P1 će biti dodata naredba:

```
S1;
signal(synch);
a u proces P2:
wait(synch);
S2;
```

Pošto je `synch` inicijalizovan na 0, P2 će izvršiti naredbu S2 tek nakon što P1 pozove `signal(synch)`, a to će se desiti tek pošto se izvrši naredba S1.

Najvažniji nedostatak implementacije semafora koju smo upravo opisali jeste to što zahteva "uposleno čekanje" (busy waiting). Dok je proces u svojoj kritičnoj sekciji, drugi procesi koji pokušavaju da uđu u kritičnu sekciju moraju da se "vrte" u svojim ulaznim sekcijama. Očigledno je da to može prouzrokovati problem u stvarnim multiprogramskim sistemima u kojima se procesor deli između brojnih procesa. Zaposleno čekanje troši procesorske cikluse koje bi neki drugi procesi mogli efikasno da iskoriste. Vrsta semafora koju smo upravo opisali zove se **spinlok** jer se proces "vrti" (spins) dok čeka da dobije bravu (lock). Spinlok semafori su korisni kada se očekuje da će zaključavanje kritične sekcije trajati kratko i često se primenjuju u multiprocesorskim sistemima u kojima jedna nit može da se "vrti" na jednom procesoru, dok druga nit izvršava kritičnu sekciju na drugom procesoru.

Da bi se prevazišla potreba za "uposlenim čekanjem", može se promeniti definicija semaforskih operacija `wait()` i `signal()`. Kada proces izvrši operaciju `wait()` i ustanovi da vrednost semafora nije pozitivna, mora da čeka. Međutim, umesto da "uposleno čeka", može sam sebe da blokira i da se premesti u red čekanja povezan

sa semaforom, pri čemu stanje procesa postaje *Waiting*. Kontrola se zatim prepušta raspoređivaču procesa, koji bira sledeći proces koji će biti izvršen.

Proces koji je blokiran u čekanju na semafor S treba da se vrati na izvršavanje kada neki drugi proces obavi operaciju `signal()`. Proces se “budi” operacijom `wakeup()` koja menja stanje procesa iz *Waiting* u *Ready*. Nakon toga proces se premešta u red spremnih poslova. Pri tom procesor može ali i ne mora da prekine izvršavanje tekućeg procesa, u zavisnosti od toga kako je implementirano raspoređivanje procesa.

Da bismo implementirali semafore po ovoj definiciji, definisamo ih kao C strukturu (Dodatak A):

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

Svaki semafor ima celobrojnu vrednost `value` i listu procesa `list`. Kada proces mora da čeka na semafor, on se dodaje u listu procesa. Operacija `signal()` uklanja jedan proces iz liste procesa koji čekaju na semafor i “budi” ga.

Semaferska operacija `wait()` sada se može definisati kao:

```
wait(semaphore *s) {
    s->value--;
    if(s->value < 0) {
        dodaj ovaj proces u s->list;
        block();
    }
}
```

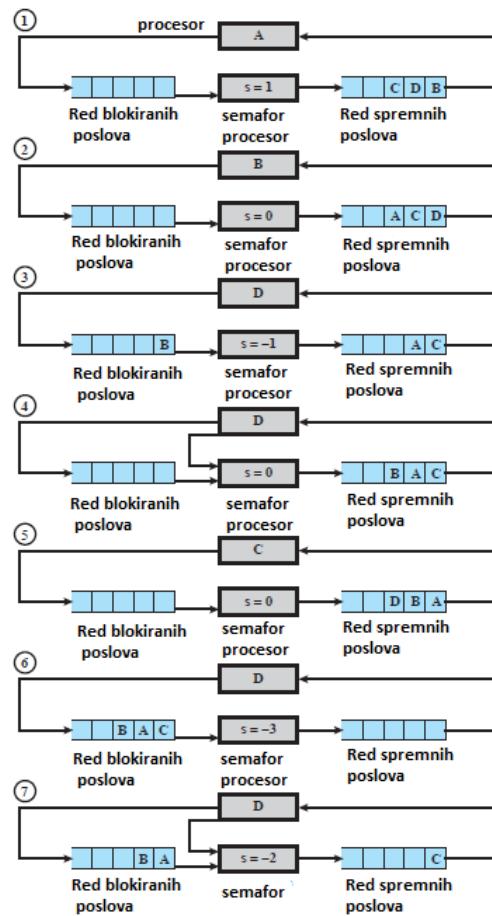
Operacija `signal()` definiše se kao:

```
signal(semaphore *s) {
    s->value++;
    if(s->value <=0) {
        ukloni proces iz s->list;
        wakeup(proces);
    }
}
```

Operacija `block()` suspenduje proces koji je poziva, a operacija `wakeup(p)` nastavlja izvršavanje blokiranog procesa p. Ove dve funkcije operativni sistem obezbeđuje u vidu sistemskih poziva.

Postavlja se pitanje po kom redosledu se procesi uklanjuju iz reda čekanja na semafor. Najpravednija je FIFO (first-in-first-out) politika, u kojoj se proces koji je najduže bio blokiran prvi oslobađa iz reda. Semafori koji primenjuju takvu politiku zovu se **jaki semafori** (strong semaphore), za razliku od **slabih semafora** (weak semaphores) koji ne zadaju redosled kojim se procesi uklanjuju iz reda. Slika 5.1 ilustruje

način rada jakog semafora. Procesi A, B i C zavise od rezultata procesa D. U početku se izvršava proces A, procesi B, C i D su u stanju *Ready*, a semafor ima vrednost 1, što znači da je dostupan jedan rezultat semafora D. Kada proces A izvrši operaciju `wait()` na semaforu `s`, on se dekrementira tako da mu je vrednost 0, pa proces A može da nastavi da se izvršava, da bi na kraju ponovo stigao u red spremnih procesa. Zatim se izvršava proces B koji izdaje naredbu `wait()` i blokira se, dozvoljavajući izvršavanje procesa D. Kada D izda nov rezultat, izvršava operaciju `signal()` koja omogućuje procesu B da se premesti u red spremnih poslova. D ponovo stiže u red spremnih poslova i proces C počinje da se izvršava, ali se blokira kada obavi operaciju `wait()`. Slično, procesi A i B čekaju na semafor, dozvoljavajući procesu D da nastavi izvršavanje. Kada D izda rezultat, obavlja operaciju `signal()` koja prevodi proces C u red spremnih poslova.



Slika 5.1: Primer mehanizma semafora.

## 5.5. Monitori

Iako su semafori efikasan i koristan način za sinhronizaciju procesa, njihovo neispravno korišćenje može da prouzrokuje greške u vremenu izvršavanja koje se teško otkrivaju jer se dešavaju samo u slučajevima tačno određene sekvene izvršavanja instrukcija. Ponovo ćemo se vratiti na problem kritične sekcije, u kome svi procesi dele semafor `mutex` inicijalizovan na 1. Svaki proces mora da izvrši operaciju `wait(mutex)` pre nego što uđe u kritičnu sekciju, odnosno `signal(mutex)` kada izade iz nje. Ako se taj sled pozivanja ne poštuje, može se desiti da se dva procesa istovremeno nađu u kritičnoj sekciji. Na primer, pretpostavimo da proces izmeni redosled pozivanja operacija `wait()` i `signal()` za semafor `mutex()`, na sledeći način:

```
signal(mutex);
    kritična sekcija
wait(mutex);
```

U ovom slučaju nekoliko procesa može da se nađe istovremeno u kritičnoj sekciji, čime je narušen princip međusobnog isključenja. Ova greška se može otkriti jedino ako je nekoliko procesa istovremeno aktivno u kritičnoj sekciji, ali takva situacija po nekad se ne može lako izazvati.

Zatim, pretpostavimo da proces zameni `signal(mutex)` sa `wait(mutex)`, odnosno:

```
wait(mutex);
    kritična sekcija
wait(mutex);
```

U ovom slučaju, desiće se potpuni zastoj, odnosno nijedan proces neće moći da uđe u kritičnu sekciju.

Treće, pretpostavimo da proces "zaboravi" da pozove `wait(mutex)` ili `signal(mutex)`, ili obe funkcije. U tom slučaju može biti narušen princip međusobnog isključenja, a može se pojaviti i potpuni zastoj.

Prethodni primjeri ilustruju da je veoma lako pogrešno upotrebiti semafore, što prouzrokuje greške u sinhronizaciji. Zbog toga su u višim programskim jezicima razvijene konstrukcije poput **monitora** čija je sintaksa:

```
monitor ime_monitora {
    //deklaracije deljenih promenljivih
    procedura P1(...) { ...}
    procedura P2(...) {....}
    .
    .
    .
    procedura Pn(...) {...}
    inicijalizacioni kod(...) {....}
}
```

Konstrukcija monitora obezbeđuje da unutar monitora u određenom trenutku bude aktivan samo jedan proces. Procedura definisana unutar monitora može da pristupi samo promenljivama koje su deklarisane lokalno, unutar monitora, kao i svojim parametrima. Da bi se rešio problem sinhronizacije, monitoru se moraju dodati definicije promenljivih tipa uslova:

```
uslov x, y;
```

Jedine operacije koje se mogu primenjivati na uslovne promenljive su `wait()` i `signal()`. Operacija `x.wait()` znači da će proces koji je poziva biti suspendovan dok neki drugi proces ne pozove `x.signal()`. Operacija `signal()` nastavlja izvršavanje samo jednog procesa. Ako nijedan proces nije suspendovan, onda pozivanje operacije `signal()` nema nikakvog efekta, odnosno stanje uslovne promenljive `x` je isto kao da operacija nikada nije bila izvršena. Primetite razliku u odnosu na operaciju `signal()` za semafore, koja uvek utiče na stanje semafora.

Prepostavimo sada da je operaciju `x.signal()` pozvao proces P1, i da postoji suspendovan proces P2 povezan sa uslovom `x`. Ako se suspendovanom procesu P2 dozvoli da nastavi izvršavanje, proces P1 koji šalje signal mora da čeka. U suprotnom bi procesi P1 i P2 bili istovremeno aktivni unutar monitora. Međutim, oba procesa nastavljaju da se izvršavaju, pri čemu postoje dve mogućnosti:

1. *Signaliziraj i čekaj.* P1 ili čeka dok P2 izade iz monitora ili čeka neki drugi uslov.
2. *Signaliziraj i nastavi.* P2 čeka dok P1 ne izade iz monitora ili čeka drugi uslov.

### 5.5.1. Implementacija monitora pomoću semafora

Za svaki monitor uvodi se semafor `mutex` i inicijalizuje na 1. Proces mora da izvrši `wait(mutex)` pre nego što uđe u monitor, a kada izade iz njega, mora da izvrši `signal(mutex)`.

Pošto proces koji signalizira mora da čeka da proces koji se nastavlja završi izvršavanje ili se blokira, uvodi se dodatni semafor `next` i inicijalizuje na 0; njega će koristiti procesi koji signaliziraju da bi sami sebe suspendovali. Celobrojna promenljiva `next_count` broji suspendovane procese za `next`. Svaka spoljašnja procedura P se zamenuju sa:

```
wait(mutex);
telo P
if(next_count > 0)
    signal(next);
else
    signal(mutex);
```

Međusobno isključenje unutar monitora je obezbedeno.

Sada ćemo opisati kako se implementiraju uslovne promenljive. Za svaki uslov `x` uvođe se semafor `x_sem` i celobrojna promenljiva `x_count` i inicijalizuju se na 0. Operacija `x.wait()` se implementira kao:

```

x_count++;
if(next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;

```

Operacija `x.signal()` se implementira kao:

```

if(x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}

```

## 5.6. Klasični problemi sinhronizacije

U ovom odeljku predstavićemo nekoliko problema koji se koriste za proveru gotovo svih novih algoritama za sinhronizaciju. U rešenjima za opisane probleme korištćemo semafore za sinhronizaciju.

### 5.6.1. Problem proizvođača i potrošača

Jedan od najčešćih problema u konkurentnoj obradi je problem proizvođača i potrošača (producer/consumer). Ovaj problem se javlja kada ima nekoliko proizvođača koji smeštaju neku vrstu podataka (zapise, znakove) u bafer. Sa druge strane postoji jedan potrošač koji podatke čita iz bafera redom, jedan po jedan. Sistem mora da obezbedi preklapanje operacija sa baferom, odnosno u istom trenutku baferu može da pristupi ili proizvođač, ili potrošač.

Za početak, prepostavimo da je bafer neograničen i da se sastoji od linearнog niza elemenata. Funkcije proizvođača i potrošača možemo da definišemo na sledeći način:

proizvođač:

```

while(true) {
    //proizvodi stavku v
    b[in]=v;
    in++;
}

```

potrošač:

```
while(true) {
    while(in < out) ne radi ništa;
    w = b[out];
    out++;
    //troši stavku w
}
```

Proizvođač generiše podatke i smešta ih u bafer b svojim tempom. Kad god doda stavku, inkrementira se indeks in bafera. Potrošač slično tome čita podatke iz bafera, ali mora da proveri da bafer slučajno nije prazan (in > out).

Implementiraćemo ovaj sistem pomoću binarnih semafora. Semafor `mutex` se koristi za međusobno isključivanje (tj. onemogućavanje istovremenog pristupa baferu), semafor `empty` primorava potrošača da stane kada je bafer prazan, a semafor `full` označava da je bafer pun.

proizvođač:

```
do {
    produce();
    wait(empty);
    wait(mutex);
    ...
    signal(mutex);
    signal(full);
} while(true);
```

potrošač:

```
do {
    wait(full);
    wait(mutex);
    ...
    signal(mutex);
    signal(empty);
    consume();
} while(true);
```

### 5.6.2. Problem čitača i pisača

Podaci se mogu deliti između nekoliko konkurentnih procesa. Neki od tih procesa će ih samo čitati, a neki će ih i čitati i menjati (upisivati). Te dve vrste procesa zvaćemo čitači i pisači. Ako dva čitača istovremeno pristupaju istom podatku, očigledno je da se neće desiti ništa; međutim, ako pisač i neki drugi proces (čitač ili pisač) istovremeno pristupaju podatku, može nastati problem.

Da bi se problem izbegao, zahteva se da pisači imaju isključiv pristup deljenim podacima. Postoji nekoliko varijacija na temu problema čitača i pisača; ovde ćemo

predstaviti rešenje za problem u kome se zahteva da nijedan čitač ne čeka osim ako neki pisač nije već dobio dozvolu da koristi deljene podatke.

Za čitače se uvode semafori `mutex` i `write` koji se inicijalizuju na 1, kao i celobrojna promenljiva `readcount` koja se inicijalizuje na 0. Semafor `write` je zajednički za čitače i pisače, a semafor `mutex` se koristi da bi obezbedio međusobno isključenje prilikom ažuriranja promenljive `readcount`. Promenljiva `readcount` prati koliko ima procesa koji trenutno čitaju objekat. Semafor `write` služi za međusobno isključenje pisača, a koriste ga i prvi ili poslednji čitač koji ulaze ili izlaze iz kritične sekcije. Ovaj semafor ne koriste čitači koji ulaze ili izlaze iz kritične sekcije dok se drugi čitači nalaze u kritičnim sekcijama.

pisac:

```
do {
    wait(write);
    pisanje
    signal(write);
} while(true);
```

čitač:

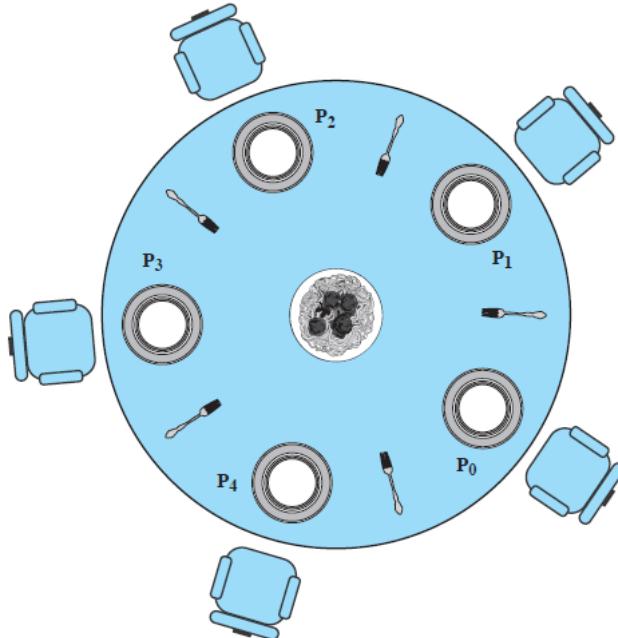
```
do {
    wait(mutex);
    readcount++;
    if(readcount == 1)
        wait(write);
    signal(mutex);
    citanje
    wait(mutex);
    readcount--;
    if(readcount == 0)
        signal(write);
    signal(mutex);
} while(true);
```

Primetite da ako je pisač u kritičnoj sekciji, a n čitača čeka, onda je jedan čitač u redu za `write`, a n-1 čitača su u redu za `mutex`. Takođe, kada pisač pozove `signal(write)`, može se nastaviti izvršavanje čitača koji čekaju, ili jednog pisača koji čeka. Ovaj izbor obavlja raspoređivač procesa.

### 5.6.3. Problem filozofa za večerom

Zamislimo pet filozofa koji život provode razmišljajući i jedući. Oni sede za okruglim stolom sa pet stolica, po jedna za svakog filozofa (Slika 5.2). Na sredini stola nalazi se činija sa špagetama, a na stolu ima pet viljuški. Kada filozof misli, on ne komunicira sa svojim kolegama. S vremena na vreme filozof ogladni i pokušava da

uhvati jednu od dve viljuške koje su mu najbliže. U svakom trenutku filozof može da podigne samo jednu viljušku. Kada gladan filozof ima obe viljuške istovremeno, on jede bez ispuštanja viljušaka. Kada završi sa jelom, filozof spušta viljuške i počinje ponovo da razmišlja.



Slika 5.2: Filozofi za večerom.

Predstavićemo rešenje za problem sinhronizacije filozofa koji večeraju pomoću semafora.

U rešenju sa semaforima svaka viljuška se predstavlja posebnim semaforom. Filozof pokušava da uzme viljušku tako što poziva operaciju `wait()` za viljušku; kada spušta viljušku, poziva operaciju `signal()` za odgovarajuće semafore. Prema tome, deljeni podaci su semafori `viljuška[0]`, ..., `viljuška[4]` koji se inicijalizuju na 1. Struktura rešenja je sledeća:

```

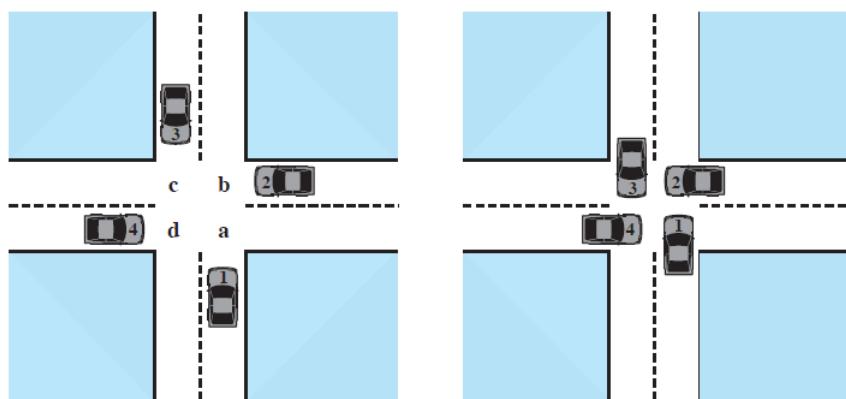
do {
    wait(viljuška[i]);
    wait(viljuška[(i+1) % 5]);
    jede();
    signal(viljuška[i]);
    signal(viljuška[(i+1) % 5]);
    misli();
} while(true);
    
```

Iako ovo rešenje garantuje da dva susedna filozofa neće jesti istovremeno, može da prouzrokuje potpuni zastoj. Pretpostavimo da svih pet filozofa ogladne u istom trenutku i da se svaki uhvati za levu viljušku. Tada će svi semafori imati vrednost 0. Ako svaki filozof pokuša da uzme desnu viljušku, to će biti odlagano unedogled. Problem potpunog zastoja rešava se primenom monitora.

### 5.7. Potpuni zastoj

Potpuni zastoj (deadlock) dešava se kada se skup procesa takmiči za sistemske resurse ili međusobno komunicira. Procesi mogu da se nađu u potpunom zastaju (ili mrtvoj petlji) kada svakog od njih blokira neki događaj (npr. čekaju da se oslobođe neki sistemski resurs), a taj događaj zavisi od drugih blokiranih procesa u potpunom zastaju. Zastoj je potpun zato što nijedan od tih događaja ne može nikako da se desi. Za razliku od drugih problema upravljanja konkurentnim procesima, u opštem slučaju ne postoji efikasno rešenje za izbegavanje potpunog zastoja.

U svim slučajevima potpunog zastoja postoji nekoliko procesa čiji su zahtevi u sukobu. Uobičajen primer kojim se ilustruje potpuni zastoj je primer raskrsnice. Na Slici 5.3 je prikazan primer potpunog zastoja u saobraćaju, kada četiri automobila pristigu u raskrsnicu približno u isto vreme. Četiri kvadranta raskrsnice predstavljaju resurse nad kojima je potrebno ostvariti kontrolu. Praktično pravilo koje se primenjuje u raskrsnicama je da prednost ima vozilo koje se nalazi sa desne strane, ali ovo pravilo važi samo ako na raskrsnici ima maksimalno tri vozila. Međutim, ako četiri vozila pristignu u raskrsnicu približno u isto vreme, nijedno neće ući u raskrsnicu zato što ima vozilo sa desne strane, čime će se prouzrokovati potpuni zastoj.



Slika 5.3: Potpuni zastoj (deadlock) u saobraćaju.

U uobičajenom režimu korišćenja resursa, proces može da bude u nekoj od tri faze:

- ▷ Zahtev (request): proces zahteva resurs, a ako ne može da ga dobije, čeka da se on oslobodi.
- ▷ Korišćenje (use): proces je dobio resurs i koristi ga.
- ▷ Oslobađanje (release): proces oslobada resurs pošto je završio sa korišćenjem.

Zamislimo situaciju sistema koji ima tri CD uređaja, koje su zauzela tri procesa. Ako bilo koji od tih procesa pokuša da dobije drugi CD uređaj, sva tri procesa će biti u stanju potpunog zastoja, jer svi čekaju da se ispunii uslov oslobađanja uređaja, koji može da ispunii samo neki od preostalih procesa koji čekaju. Potpuni zastoj se često javlja u višenitnim aplikacijama jer se više niti takmiči za iste resurse; zbog toga se višenitne aplikacije moraju pažljivo programirati.

Potpuni zastoj se može definisati kao trajno blokiranje skupa procesa koji se takmiče za sistemske resurse. Zastoj nastupa ako su istovremeno ispunjeni sledeći uslovi:

1. međusobno isključenje (mutual exclusion): barem jedan resurs mora da bude u stanju u kome se ne može deliti, odnosno samo jedan proces može da koristi određeni resurs. Ako drugi proces zatraži isti resurs, dodela mora biti odložena sve dok se resurs ne oslobodi.
2. dobrovoljno oslobađanje resursa (no preemption): resurs može biti oslobođen samo ako ga dobrovoljno oslobodi proces kome je dodeljen.
3. zadržavanje resursa tokom čekanja (hold and wait): proces mora da ima barem jedan resurs dok čeka da mu se dodeli neki drugi resurs koji je trenutno dodeljen nekom drugom procesu
4. kružno čekanje (circular wait): postoji skup procesa P1, P2, ..., Pn takvih da proces P1 čeka na resurs koji drži proces P2, proces P2 čeka na resurs koji drži P3, i tako redom do procesa Pn koji čeka na resurs koji drži proces P0.

### 5.7.1. Sprečavanje potpunog zastoja

Da bi se sprečio potpuni zastoj, dovoljno je da samo jedan od prethodno pomenuta četiri uslova ne važi. Razmotrićemo ih pojedinačno.

1. Međusobno isključenje: ovo je uslov koji se obično ne može izbeći. Ako je zahtev za resursima isključiv, onda on mora biti podržan u operativnom sistemu. Neki resursi, npr. datoteke, omogućuju neisključiv pristup tokom čitanja, ali isključiv tokom upisivanja. I u takvim slučajevima može se desiti potpuni zastoj ako više procesa traži dozvolu za upis u datoteku.
2. Zadržavanje resursa tokom čekanja: ovaj uslov može da se spreči tako što se zahteva da proces sve potrebne resurse zatraži u istom trenutku, tako da se drži blokiran sve do momenta kada svi resursi mogu da mu se dodele istovremeno. Ovakav pristup nije efikasan iz dva razloga. Prvo, može se desiti da proces dugo čeka da se oslobole svi potrebni resursi, kada bi možda mogao da obavi deo po-

- sla sa delom dostupnih resursa. Drugo, resursi dodeljeni procesu mogli bi dugo da ostanu neiskorišćeni, a za to vreme su uskraćeni drugim procesima. Takođe, ponekad proces ne može da zna unapred koji će mu resursi biti potrebni.
3. Zadržavanje resursa tokom čekanja: ovaj uslov se može sprečiti na više načina. Prvo, ako se procesu koji drži određene resurse uskrati mogućnost za daljim zahtevima, taj proces će morati da oslobodi resurse koje drži i, ako je potrebno, da ih ponovo zahteva. Druga mogućnost je da operativni sistem prinudno suspenduje proces koji drži resurse koje zahteva drugi proces; ovaav pristup deluje samo ako procesi koji su u potpunom zastoji nisu istog prioriteta.
  4. Kružno čekanje: ovaj uslov se može izbeći ako se definiše linearan redosled tipova resursa. Ako neki proces drži resurse tipa R, onda ubuduće može da zahteva samo resurse koji su iza tipa R u redosledu. Iako rešava problem, ovaj pristup usporava procese i nepotrebno uskraćuje resurse.

### 5.7.2. Otkrivanje i oporavak od zastoja

Operativni sistemi mogu biti projektovani tako da predviđaju mogućnost pojave potpunog zastoja i sprečavaju ga. To zahteva da se u trenutku zahteva za resursom predvidi da li taj zahtev potencijalno može da izazove potpuni zastoj, odnosno da se predvidi kakvi će biti budući zahtevi za resursima.

Neki operativni sistemi (kao što su Windows i UNIX) uopšte ne razmatraju mogućnost da dođe do potpunog zastoja. Drugi pak ne sprečavaju pojavu zastoja, ali koriste algoritme koji ga otkrivaju. Tada su na raspolaganju su dve mogućnosti. Prva je da se jedan ili nekoliko procesa napuste da bi se sprečilo kružno čekanje. Druga je da se određeni resursi nasilno preuzmu od procesa u potpunom zastoju. Detaljan opis algoritama za otkrivanje i oporavak od zastoja prevazilazi okvire ovog kursa, pa se čitaoci upućuju na nekoliko odličnih knjiga u odeljku literature.



## **Glava 6**

# **Upravljanje memorijom**

U sistemima koji rade sa jednim programom, glavna memorija se deli na dva dela: jedan deo je za jezgro (kernel) operativnog sistema, a drugi za program koji se trenutno izvršava. U sistemima koji rade sa više programa korisnički deo memorije mora se dalje deliti i prilagoditi radu sa više procesa. Zadatak operativnog sistema je da upravlja memorijom. Efikasno upravljanje memorijom veoma je važno za sisteme koji rade sa više programa (tzv. multiprogramski sistemi). Ako se samo nekoliko procesa nalazi u memoriji, onda će najveći deo vremena svi procesi čekati na U/I operacije i procesor će biti besposlen. Dakle, memorija treba da se alocira, tj. dodeli da bi se korisno procesorsko vreme dodelilo spremnim procesima.

Upravljanje memorijom je jedan od najvažnijih i najsloženijih zadataka operativnog sistema. Ono posmatra memoriju kao resurs koji treba da se dodeli i da se deli između više aktivnih procesa. Da bi se efikasno koristili procesor i U/I resursi, poželjno je u glavnoj memoriji održavati što više procesa. Pored toga, poželjno je osloboditi programere ograničenja vezanih za veličinu memorije. Osnovne tehnike upravljanja memorijom su straničenje i segmentacija. Kod straničenja, svaki proces se deli na relativno male stranice fiksne dužine. Segmentacija omogućava upotrebu delova procesa različitih veličina. Moguće je kombinovati segmentaciju i straničenje u prostoj šemi za upravljanje memorijom.

### **6.1. Zahtevi u upravljanju memorijom**

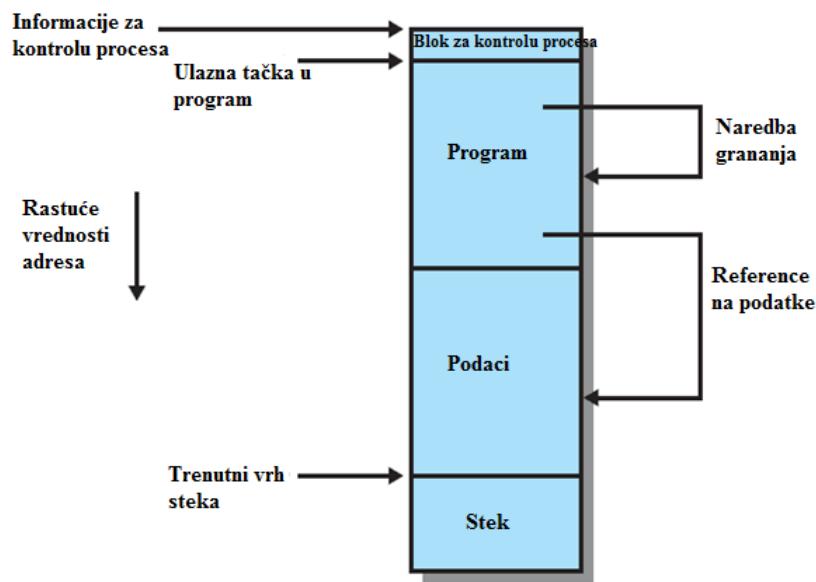
Upravljanje memorijom treba da zadovolji sledeće zahteve:

1. relokacija
2. zaštita
3. deljenje
4. logička organizacija
5. fizička organizacija

U nastavku ćemo detaljnije objasniti svaki od pomenutih zahteva.

### 6.1.1. Relokacija

U sistemu sa više programa raspoloživa glavna memorija se obično deli između više procesa. Programer ne može da zna unapred koji drugi programi će biti u glavnoj memoriji tokom izvršavanja njegovog programa. Takođe, programeri bi trebalo da imaju mogućnost da zamenjuju aktivne procese u glavnoj memoriji, tj. da obezbeđuju procese spremne za izvršavanje da bi se maksimiziralo iskorišćenje procesora. Program koji se prebacuje na disk treba da se premesti u neko drugo područje u memoriji. Procesor mora da se bavi memorijskim referencama unutar programa. Procesor i operativni sistem treba da budu sposobni da prevedu memorijske reference iz kôda programa u stvarne fizičke adrese memorije, tj. tekuće lokacije programa u glavnoj memoriji. Prvi korak u kreiranju aktivnog procesa je učitavanje programa u glavnu memoriju i kreiranje "slike" procesa (process image).



Slika 6.1: Adresni zahtevi procesa.

Aplikacija se sastoји од određenog broja kompajliranih i asembleriranih modula u obliku objektnog kôda. Oni su povezani da bi se razrešila međusobna referenciranja modula. U isto vreme se razrešavaju reference prema biblioteci rutina. Same bibliotečke rutine mogu biti ugrađene u program ili referencirane kao deljeni kôd koji mora da obezbedi operativni sistem tokom izvršavanja (run time).

### 6.1.2. Zaštita

Svaki proces trebalo bi da bude zaštićen od slučajnih ili namernih neželjenih uticaja drugih procesa. Drugi procesi ne bi smeli bez dozvole da referenciraju memoriske lokacije koje tekući proces čita ili upisuje. Korisnički proces ne može da pristupi bilo kom delu operativnog sistema, ni programu ni podacima. Uslovi zaštite memorije prvenstveno se ostvaruju hardverski, a ne softverski (preko operativnog sistema).

### 6.1.3. Deljenje

Svaki zaštitni mehanizam mora da bude prilagodljiv i da dozvoli da više procesa pristupi istom delu glavne memorije. Ako više procesa izvršava isti program, treba dozvoliti svakom procesu da pristupi istoj kopiji programa, a ne da ima sopstvenu kopiju. Procesi koji sarađuju na istom zadatku mogu istovremeno da pristupaju strukturama podataka. Sistem za upravljanje memorijom mora da omogući kontrolisani pristup deljenim oblastima memorije bez ugrožavanja osnovne zaštite.

### 6.1.4. Logička organizacija

Glavna memorija u računarskom sistemu je organizovana kao linearni ili jednodimenzionalni adresni prostor sastavljen od nizova bajtova ili reči. Slično je organizovana i sekundarna memorija na fizičkom nivou. Većina programa je organizovana u modulima, od kojih se neki ne mogu menjati (samo se čitaju ili izvršavaju), a neki sadrže podatke koji mogu da se menjaju. Ako operativni sistem i računarski hardver mogu efikasno da rade sa korisničkim programima i podacima u obliku neke vrste modula, to ima svojih prednosti:

- ▷ moduli mogu biti pisani i kompajlirani nezavisno
- ▷ za različite module mogu se definisati različiti stepeni zaštite
- ▷ moduli mogu biti deljeni između procesa

Pomenute zahteve zadovoljava jedna od tehnika upravljanja memorijom koja se zove segmentacija.

### 6.1.5. Fizička organizacija

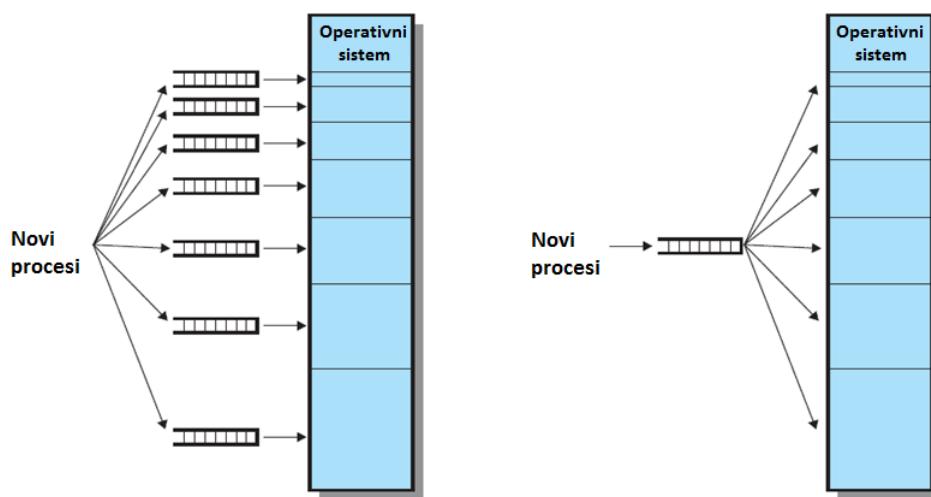
Računarska memorija je organizovana u najmanje dva nivoa, koje zovemo glavna memorija i sekundarna memorija. Glavna memorija obezbeđuje brz pristup po relativno visokoj ceni, ali ne obezbeđuje trajno skladištenje podataka. Sekundarna memorija je sporija i jeftinija od glavne memorije i po pravilu obezbeđuje trajno skladištenje podataka. U šemi dva nivoa, glavni sistemski problem je organizacija toka informacija između glavne i sekundarne memorije, što je osnova upravljanja memorijom.

## 6.2. Podela memorije na particije

Osnovna operacija upravljanja memorijom je učitavanje procesa u glavnu memoriju da bi ih procesor izvršio. U skoro svim modernim sistemima koji rade sa više programa ovo podrazumeva naprednu šemu upravljanja memorijom poznatu kao virtuelna memorija. Virtuelna memorija je zasnovana na tehnikama segmentacije i straničenja. S druge strane, postoji i tehnika upravljanja memorijom koja ne koristi virtuelnu memoriju: to je podela memorije na particije.

### 6.2.1. Podela na particije fiksne veličine

U većini šema za upravljanje memorijom, možemo da prepostavimo da operativni sistem zauzima određeni fiksni deo glavne memorije, a da je ostatak raspoređan drugim procesima. Najjednostavnija tehnika za upravljanje neiskorišćenom memorijom je da se ona podeli na oblasti (particije) sa fiksnim granicama. Međutim, upotreba fiksnih particija jednake veličine prouzrokuje teškoće. Na primer, program može da bude suviše velik da bi stao u particiju. U tom slučaju, programer mora da projektuje program koji koristi preklapanja (overlay), tako da se u određenom trenutku samo deo programa nalazi u glavnoj memoriji. Takođe, u slučaju particija fiksne veličine glavna memorija se ne koristi efikasno jer svaki program, bez obzira koliko mali, zauzima celu particiju. To je tzv. **unutrašnja fragmentacija**. Zbog toga se particije fiksne veličine skoro i ne koriste u modernim operativnim sistemima, već se koriste isključivo particije nejednakih veličina.



Slika 6.2: Dodela memorije kod podele na particije.

### 6.2.2. Dinamička podela na particije

Teškoće sa deljenjem na fiksne particije se prevazilaze dinamičkim deljenjem na particije. Kod dinamičkog deljenja memorije, particije su promenljive dužine i broja. Kada se proces učita u glavnu memoriju, njemu se dodeljuje tačno onoliko memorije koliko zahteva. Međutim, primena ovog metoda dovodi do situacije u kojoj postoji mnogo malih praznina, tj. neiskorišćenih delova u memoriji. Kako vreme prolazi, memorija postaje sve više fragmentirana, pa se smanjuje njeno iskorišćenje. Ova pojava se zove **spoljašnja fragmentacija** i suprotna je unutrašnjoj fragmentaciji. Tehnika za prevazilaženje spoljašnje fragmentacije je sažimanje: s vremena na vreme, operativni sistem pomera procese da budu susedni da bi se sva slobodna memorija grupisala u jedan blok. Sažimanje memorije zahteva vreme, pa projektanti operativnih sistema moraju da odluče na koji način da pridruže procese memoriji. Postoje tri algoritma smeštanja procesa u glavnu memoriju, u kojima su glavni parametri slobodni blokovi veličine jednake ili veće od procesa koji se smešta u memoriju.

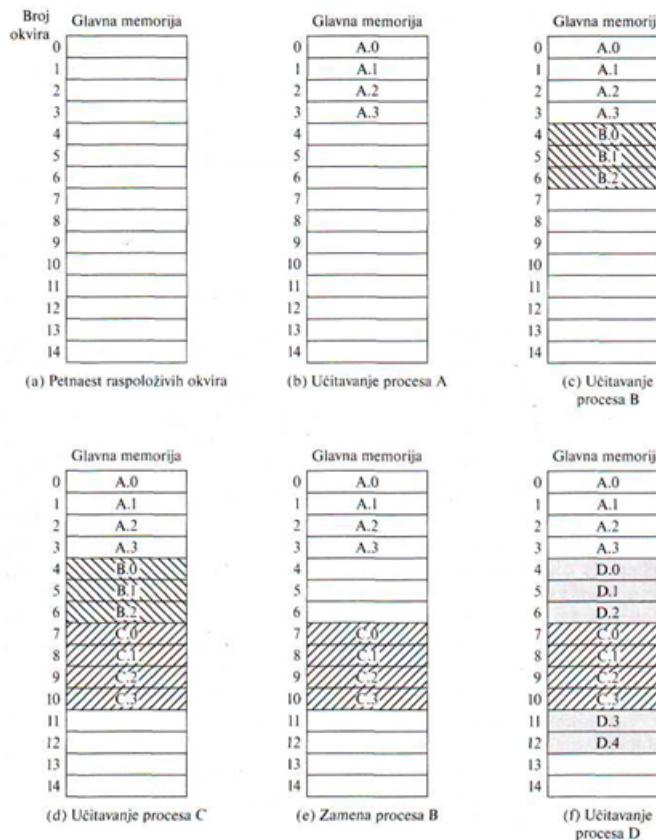
- ▷ Best-fit: odabira blok koji je po veličini najbliži zahtevu.
- ▷ First-fit: pregleda memoriju od početka i dodeljuje prvi slobodan blok koji je dovoljno velik da primi proces.
- ▷ Next-fit: nastavlja da pregleda memoriju od lokacije koja je poslednja dodeljena i odlučuje se za blok koji je dovoljno velik.

## 6.3. Straničenje

Deljenje na particije fiksne i promenljive veličine je neefikasno u pogledu iskoristišenja memorije; prva tehnika za rezultat ima unutrašnju, a druga spoljašnju fragmentaciju. Pretpostavimo, međutim, da je glavna memorija izdeljena na jednake delove fiksne veličine koji su relativno mali i da se svaki proces takođe deli na male jednake delove te iste fiksne veličine. Tada delovi procesa, poznati kao **stranice** (pages), mogu da se dodele raspoloživim delovima memorije, poznatim kao **okviri stranica** (page frame). Pokazaćemo da se neiskorišćen prostor u memoriji koji je posledica unutrašnje fragmentacije za svaki proces sastoji samo od dela poslednje stranice procesa, tj. da tehnika straničenja ne prouzrokuje spoljašnju fragmentaciju.

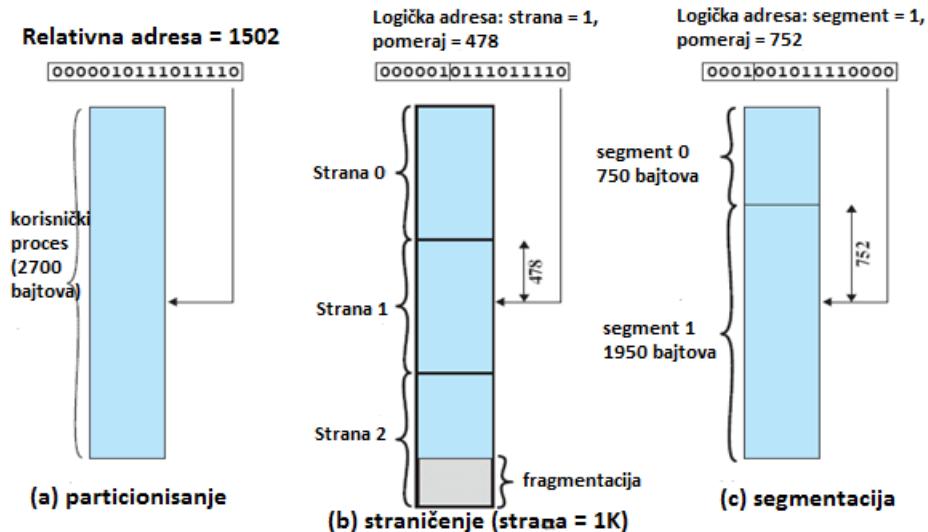
Na Slici 6.3 ilustrovano je korišćenje stranica i okvira. U datom trenutku, neki od okvira u memoriji su u upotrebi, a neki su slobodni. Operativni sistem održava listu slobodnih okvira. Proces A koji se čuva na disku sastoji se od četiri stranice. Kada dođe vreme da se učita taj proces, operativni sistem nalazi četiri slobodna okvira i učitava četiri stranice procesa A u ta četiri okvira (Slika 6.3 b). Zatim se učitavaju proces B, koji se sastoji od tri stranice i proces C, koji se sastoji od četiri stranice. Nakon toga se proces B suspenduje i zamjenjuje (swap out) iz glavne memorije. Kasnije, svi procesi u glavnoj memoriji su blokirani, a operativni sistem treba da učita novi proces D, koji se sastoji od pet stranica. Pretpostavimo da ne postoji dovoljno neupotrebljenih sused-

nih okvira koji bi sadržali proces. To ne sprečava operativni sistem da učita stranicu D. U tom slučaju se koristi pojam logičke adrese. **Logička adresa** je referenca na memorijsku lokaciju nezavisno od tekuće dodele podataka memoriji. Operativni sistem održava tabelu stranica za svaki proces. Tabela stranica pokazuje lokaciju okvira za svaku stranicu procesa.



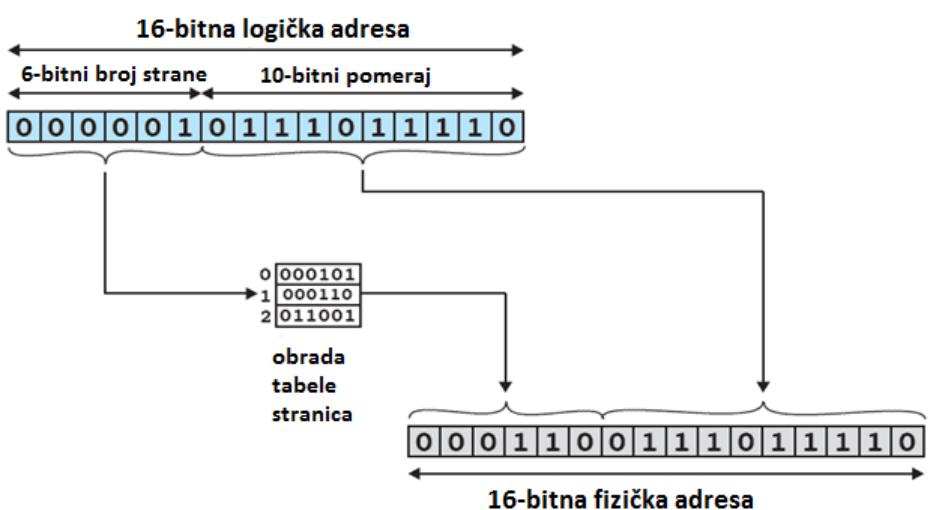
Slika 6.3: Dodela stranica procesa slobodnim okvirima.

Unutar programa, svaka logička adresa se sastoji od broja stranice (page number) i pomeraja (offset) unutar stranice. Podsetimo se da je u slučaju proste particije logička adresa zapravo položaj reči u odnosu na početak programa; procesor prevodi logičku adresu u fizičku adresu.



Slika 6.4: Logička adresa.

Kod straničenja, preslikavanje logičkih u fizičke adrese (apsolutne adrese, tj. stvarne lokacije u glavnoj memoriji) obavlja hardver procesora. Procesor mora da zna kako da pristupi tabeli stranica tekućeg procesa. Polazeći od logičke adrese (broja stranice i pomeraja unutar nje) procesor koristi tabelu stranica da bi ustanovio fizičku adresu (tj. broj okvira i pomeraj unutar njega).

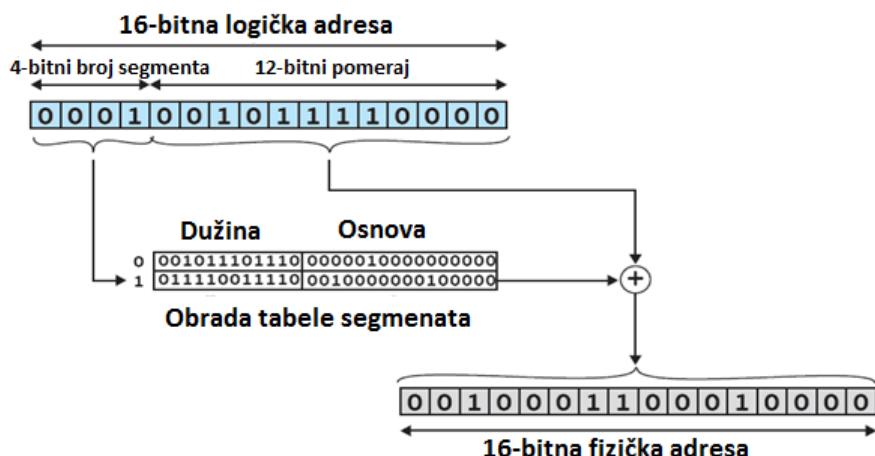


Slika 6.5: Dobijanje fizičke adrese preslikavanjem logičke adrese kod straničenja.

Straničenje je slično podeli na fiksne particije. Razlika je što su kod straničenja particije male, program može zauzimati više od jedne particije, i one ne moraju da budu susedne. Dakle, kod jednostavnog straničenja, glavna memorija je podeljena na mnogo malih okvira iste veličine. Svaki proces je podeljen na stranice veličine okvira; manji procesi zahtevaju nekoliko stranica, a veći više stranica. Kada se proces učita, sve njegove stranice se učitavaju u raspoložive okvire i postavlja se tabela stranica. Taj pristup rešava mnoge probleme koji se javljaju u podeli na particije.

#### 6.4. Segmentacija

Korisnički program i njemu pridruženi podaci mogu da se podele na određen broj segmenata. Segmenti svih programa ne moraju da budu iste veličine, mada postoji maksimalna dužina segmenta. Kao kod straničenja, logička adresa koja koristi segmentaciju sastoji se od dva dela, u ovom slučaju od broja segmenta i pomeraja unutar tog segmenta.



Slika 6.6: Dobijanje fizičke adrese preslikavanjem logičke adrese kod segmentacije.

Zbog upotrebe segmenata različitih veličina, segmentacija je slična dinamičkoj podeli na particije. U odsustvu šeme za preklapanje ili upotrebe virtuelne memorije, zahteva se da se svi segmenti programa učitaju u memoriju radi izvršavanja. Razlika u poređenju sa dinamičkom podelom na particije je u tome što kod segmentacije program može da zauzme više od jedne particije i te particije ne moraju da budu susedne. Segmentacija rešava problem unutrašnje fragmentacije, ali, kao i kod dinamičke podeli na particije, ostaje problem spoljašnje fragmentacije. Međutim, zbog toga što se proces deli na izvestan broj manjih delova, spoljašnja fragmentacija je po pravilu manja. Za razliku od straničenja koje je nevidljivo za programera, segmentacija je obično vidljiva i pogodna za organizaciju programa i podataka. Programer

ili kompjajler obično pridružuju programe i podatke različitim segmentima. U cilju modularnog programiranja program ili podaci mogu dalje da se podele na više manjih segmenata. Loša strana ove tehnike je to što programer mora znati ograničenja maksimalne veličine segmenata. Sledeća pogodnost korišćenja segmenata različitih veličina je to što ne postoji prosta veza između logičkih i fizičkih adresa. Slično straničenju, tehnika jednostavne segmentacije koristi tabelu segmenata za svaki proces i listu slobodnih blokova u glavnoj memoriji. Svaki upis u tabelu segmenata treba da dà početnu adresu odgovarajućih segmenata u glavnoj memoriji. Upis takođe treba da obezbedi dužinu segmenta da se ne bi koristile pogrešne adrese. Kada proces uđe u stanje izvršavanja, adresa iz tabele segmenata se učitava u specijalan registar hardvera za upravljanje memorijom.

## 6.5. **Virtuelna memorija**

Pošto je upravljanje memorijom složena aktivnost procesorskog hardvera i softvera operativnog sistema, prvo ćemo prikazati hardverski aspekt virtuelne memorije, definijući straničenje, segmentaciju i njihovu kombinaciju. Zatim ćemo definisati dizajn virtuelne memorije kao karakteristiku operativnog sistema.

Kod tehnike virtuelne memorije sve adresne reference su logičke reference koje se tokom izvršavanja prevode u realne adrese. To omogućuje lociranje procesa bilo gde u glavnoj memoriji i menjanje njegove lokacije tokom vremena. Virtuelna memorija dozvoljava procesu da bude podeljen na manje delove. Tokom izvršavanja ne zahteva se da delovi budu susedni u glavnoj memoriji, niti da svi budu u glavnoj memoriji. Dve osnovne tehnike virtuelne memorije su straničenje i segmentacija. Kod jednostavnijih šema upravljanja memorijom moguća je i njihova kombinacija. Virtuelna memorija zahteva i hardversku i softversku podršku. Hardversku podršku obezbeđuje procesor, a podrazumeva dinamičko prevođenje virtuelnih adresa u fizičke i generisanje prekida kada referencirana stranica ili segment nisu u memoriji. Prekid zatim poziva softver za upravljanje memorijom operativnog sistema.

### 6.5.1. **Hardver i kontrolne strukture**

U tehnikama straničenja i segmentacije, sve memorijske reference unutar procesa su logičke adrese koje se dinamički prevode u fizičke adrese tokom izvršavanja. To znači da proces može biti zamenjen tako da u različitim trenucima izvršavanja zauzima različite delove glavne memorije.

Proces može da se podeli na veći broj delova (stranica ili segmenata) koji ne moraju da budu susedni u glavnoj memoriji. Primena takve tehnike podrazumeva kombinaciju dinamičkog prevođenja adresa tokom izvršavanja i korišćenja tabela stranica ili segmenata. Nije neophodno da sve stranice ili svi segmenti procesa budu u glavnoj memoriji tokom izvršavanja procesa. Deo procesa koji je u određenom trenutku u

glavnoj memoriji definiše se kao rezidentan deo procesa. Da bi se poboljšalo iskorišćenje sistema, više procesa treba da bude u glavnoj memoriji, pri čemu proces može biti i veći od glavne memorije. Pošto se proces izvršava u glavnoj memoriji, ta memorija se označava kao realna memorija. Programer ili korisnik sistema vide mnogo veću memoriju koja je alocirana sa diska, a zove se virtuelna memorija.

### 6.5.2. Lokalitet i virtuelna memorija

Virtuelna memorija zasnovana na straničenju ili straničenju sa segmentacijom postala je nezaobilazna komponenta savremenih operativnih sistema. U tehnikama virtualizacije memorije pojavljuje se problem poznat kao **thrashing**. Sistem troši mnogo više vremena na prebacivanje delova procesa iz glavne memorije u sekundarnu nego na stvarno izvršavanje instrukcija u procesu. Rešenje ovog problema zasniva se na predviđanju koji delovi procesa će biti korišćeni u bliskoj budućnosti. Ako se primeni i princip lokaliteta koji kaže da program i podaci referencirani u procesu teže da se klasterizuju, moguće "je pametno" predvideti koji delovi procesa će se koristiti u bliskoj budućnosti, čime se izbegava thrashing.

### 6.5.3. Straničenje

Termin virtuelna memorija se obično povezuje sa sistemom koji koristi straničenje. Straničenje je u virtuelizaciji memorije prva koristila kompanija Atlas Computer pre nego što je ušlo u komercijalnu upotrebu. Kod običnog straničenja, svaki proces ima svoju tabelu stranica. Kada su sve stranice procesa učitane u glavnu memoriju, kreira se tabela stranica za taj proces i učitava u glavnu memoriju. Svaki upis u tabelu stranica sadrži broj okvira odgovarajuće stranice u glavnoj memoriji. Tabela stranica je potrebna i u tehnici virtuelne memorije zasnovanoj na straničenju. Obično se svakom procesu pridružuje jedinstvena tabela stranica. Međutim, u ovom slučaju, tabela stranica upisa postaje složenija. Pošto samo neke stranice procesa mogu biti u glavnoj memoriji, potreban je dodatni bit u svakom upisu tabele stranica koji će označavati da li je odgovarajuća stranica prisutna u glavnoj memoriji ili nije (bit P). Ako bit pokazuje da je stranica u memoriji, onda upis uključuje i broj okvira te stranice. Tabela upisa stranica uključuje i bit M koji definiše da li je sadržaj stranice promenjen od poslednjeg učitavanja u glavnu memoriju. Ako nema promena, onda nije neophodno ponovo upisivati stranicu kada dođe vreme da se ona zameni iz okvira u kome se trenutno nalazi. Postoje i drugi kontrolni bitovi za zaštitu ili deljenje na nivou stranica (Slika 6.7).

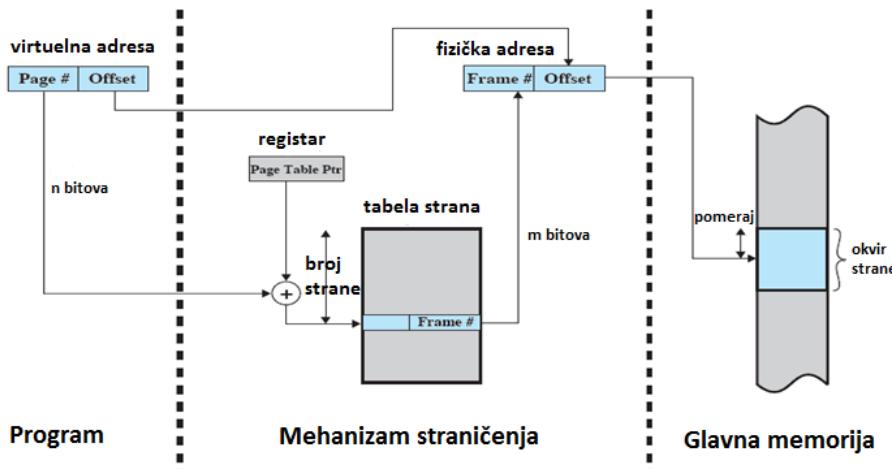


Slika 6.7: Straničenje i tabela upisa stranica.

#### 6.5.4. Struktura tabele stranica

Osnovni mehanizam za učitavanje reči iz memorije uključuje prevođenje virtuelnih (logičkih) adresa, sastavljenih od broja stranice i pomeraja, u fizičku adresu koja se sastoji od broja okvira i pomeraja, uz pomoć tabele stranica.

Pošto je tabela stranica promenljive veličine, zavisno od veličine procesa, ne možemo očekivati da se ona čuva u registrima procesora; umesto toga, ona mora da bude u glavnoj memoriji da bi joj se lakše pristupilo. Slika 6.8 predstavlja hardversku implementaciju prevođenja adresa. Kada određeni proces počne da se izvršava, u registru se nalazi početna adresa tabele strana za taj proces. Broj strane virtuelne adrese se koristi kao indeks te tabele koji definiše odgovarajući broj okvira. Taj podatak se kombinuje sa delom virtuelne adrese koji definiše pomeraj da bi se dobila željena fizička (realna) adresa. Broj polja stranica obično je mnogo veći od broja polja okvira ( $n > m$ ). Količina memorije posvećena tabelama stranica ponekad može biti neprihvatljivo velika. Da bi se prevazišao taj problem, većina tehniku virtuelne memorije čuva tabele stranica u virtuelnoj memoriji umesto u fizičkoj memoriji. Neki procesori koriste šemu sa dva nivoa za organizaciju velikih tabel stranica, u kojoj postoji direktorijum stranica koji definiše tabelu upisa stranica (page entry).



Slika 6.8: Prevodenje adresa u sistemu straničenja.

### 6.5.5. Invertovana tabela stranica

Postoje i invertovane tabele stranica u kojima se deo virtuelne adrese sa brojem stranice mapira u heš (hash) vrednost pomoću proste funkcije heširanja. Heš vrednost je pokazivač na invertovanu tabelu stranica koja sadrži tabelu upisa stranica. Uglavnom se koristi po jedan upis u invertovanu tabelu stranica za svaki okvir stranice u realnoj memoriji, umesto jedan po virtuelnoj stranici. Na taj način za tabele se zahteva fiksni deo realne memorije bez obzira na broj procesa ili podržanih virtuelnih stranica. Pošto se nekoliko virtuelnih adresa može mapirati u istu heš tabelu upisa, koristi se tehnika ulančavanja. Tabela stranica se zove invertovana zato što umesto po broju virtuelne stranice indeksira upise u tabelu stranica po broju okvira.

### 6.5.6. Translation Lookaside Buffer (TLB)

Svaka referenca na virtuelnu stranicu može da prouzrokuje dva pristupa fizičkoj memoriji: jedan da bi se dobio odgovarajući upis u tabelu stranica i drugi da bi se dobio željeni podatak. Da bi se to izbeglo, većina tehnika virtuelne memorije koristi specijalnu i veoma brzu keš memoriju za upise u tabelu stranica, koja se zove translation lookaside buffer (TLB). TLB radi na isti način kao i memorijski keš i sadrži poslednje upise u tabelu stranica. Za datu virtuelnu adresu, procesor prvo ispituje da li je željeni upis u tabelu stranica u kešu (TLB pogodak); ako jeste, broj okvira se čita iz tabele i izračunava se fizička adresa. Ako se ne pronađe traženi upis (TLB prošašaj), procesor koristi broj strane za indeksiranje tabele stranica procesa i ispituje odgovarajući upis u tabelu stranica. Ako je bit prisustva postavljen, onda je stranica u glavnoj memoriji, pa procesor može da dobije broj okvira iz tabele upisa stranica da bi izračunao fizičku adresu. Procesor takođe ažurira TLB kako bi se uključio ovaj novi upis u tabelu stranica. Ako bit prisustva nije postavljen, onda željena stranica nije u glavnoj memoriji i generiše se greška u pristupu memoriji, tj. pogrešna stranica (page fault). Hardver signalizira operativnom sistemu da učita potrebnu stranicu i ažurira tabelu stranica. Ova tehnika se često označava kao asocijativno mapiranje i suprotna je direktnom mapiranju ili indeksiranju koje smo ranije objasnili.

### 6.5.7. Veličina stranice

Važno pitanje pri projektovanju hardvera je veličina stranice, jer zavisi od više faktora. Što je stranica manja, manja je i unutrašnja fragmentacija. Da bi se optimizovalo korišćenje glavne memorije, treba smanjiti unutrašnju fragmentaciju. Što je stranica manja, veći je broj stranica po jednom procesu; što je veći broj stranica po procesu, veća je i tabela stranica. Za veće programe u okruženju koji radi sa više programa, to može da znači da neki delovi tabele stranica aktivnog procesa moraju da budu u virtuelnoj memoriji, a ne u glavnoj. To može prouzrokovati dvostruku pogrešnu stranicu (page fault) za jednu referencu u memoriji: prvi put da bi se učitao potreban deo ta-

bele stranica u memoriju i drugi put da bi se učitala stranica procesa. Takođe, fizičke karakteristike većine sekundarnih memorijskih uređaja koji su rotacioni diktiraju da veće stranice efikasnije prenose blokove podataka. Veličina stranice utiče i na brzinu pojave pogrešne stranice. Posmatrajmo princip lokaliteta stranica: ako je stranica veoma mala, u glavnoj memoriji će moći da se iskoristi relativno veliki broj stranica po jednom procesu. Nakon određenog vremena sve stranice u memoriji sadržaće delove procesa koji su blizu skorašnjih referenci, pa će brzina pojave pogrešnih stranica biti mala. Kako se veličina stranice povećava, svaka stranica pojedinačno sadržavaće lokacije koje su dalje od svih poslednjih referenci. Time efekat principa lokaliteta slabi, a brzina pojave pogrešnih stranica raste. Ova brzina opada sa približavanjem veličine stranice veličini kompletног procesa, a jednaka je nuli kada stranica obuhvata ceo proces.

Brzina pojave pogrešne stranice zavisi i od broja okvira dodeljenih procesu. Za fiksnu veličinu stranica, brzina pojave pogrešnih stranica opada sa porastom broja stranica koje se održavaju u glavnoj memoriji. Politika operativnog sistema u pogledu količine memorije dodeljene svakom procesu utiče na odluku o veličini stranice prilikom projektovanja hardvera. I na kraju, problem veličine stranice je povezan sa veličinom glavne fizičke memorije i veličine programa. Sa porastom kapaciteta glavne memorije raste i adresni prostor koji aplikacije koriste; to je očigledno kod PC računara i radnih stanica, u kojima softverske aplikacije postaju veoma složene.

#### 6.5.8. Segmentacija

Segmentacija dozvoljava programeru da vidi memoriju kao skup više adresnih prostora, odnosno segmenata. Veličina segmenata je različita i određuje se dinamički. Memorijske reference se sastoje od broja segmenta i pomeraja. Ovakva organizacija ima nekoliko prednosti u odnosu na nesegmentirani adresni prostor iz ugla programera:

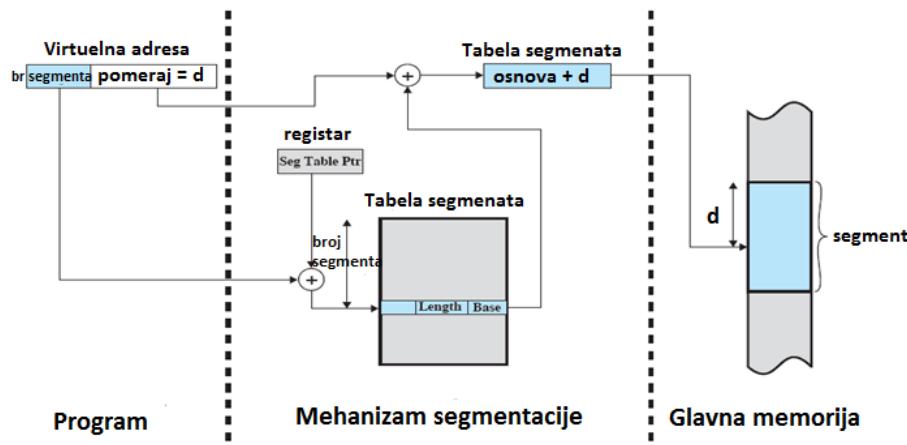
1. Rad sa rastućim strukturama podataka je prostiji: strukturama podataka se mogu dodeliti sopstveni segmenti, a operativni sistem će proširiti ili smanjiti segment po potrebi.
2. Korišćenjem više segmenata, dozvoljava se programima da budu zamenjeni i ponovo kompajlirani nezavisno, bez ponovnog povezivanja i učitavanja svih programa.
3. Deljenje između procesa: programer može da smesti koristan program ili korišnu tabelu podataka u segment koji drugi proces može da referencira.
4. Efikasna zaštita: pošto segment može da bude konstruisan tako da sadrži dobro definisan skup programa i podataka, programer ili administrator sistema može da dodeli ili oduzme privilegovane pristupe na uobičajen način.



Slika 6.9: Tabela segmenata.

Prijetimo se da kod proste segmentacije svaki proces ima sopstvenu tabelu segmenata. Kada su svi segmenti učitani u glavnu memoriju, kreira se tabela segmenata za taj proces koja se učitava u glavnu memoriju.

Svaki upis u tabelu segmenata sadrži početnu adresu odgovarajućeg segmenta u glavnoj memoriji, kao i dužinu segmenta. Isti je slučaj i kod virtualne memorije zasnovane na segmentaciji, ali tabela upisa segmenata postaje složenija. Pošto samo neki od segmenata procesa mogu biti u glavnoj memoriji, potreban je dodatni bit (P) za svaku tabelu upisa segmenata koji pokazuje da li je odgovarajući segment prisutan u glavnoj memoriji ili ne. Ako bit pokazuje da je segment u memoriji, upis uključuje i početnu adresu i dužinu tog segmenta. Drugi kontrolni bit u tabeli upisa segmenata je bit promene (M) koji pokazuje da li je sadržaj segmenta promenjen od trenutka poslednjeg učitavanja u memoriju. Mogu biti prisutni i drugi kontrolni bitovi.



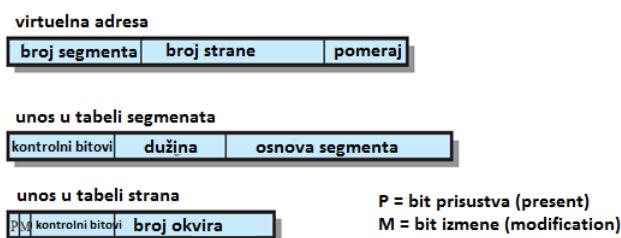
Slika 6.10: Prevođenje adresa u sistemu sa segmentacijom.

Osnovni mehanizam za čitanje reči iz memorije uključuje prevodenje virtualne, odnosno logičke adrese sastavljene od broja segmenta i pomeraja u fizičku adresu uz pomoć tabele segmenata. Pošto je tabela segmenata promenljive dužine, zavisno od veličine procesa, ne možemo očekivati da se čuva u registrima procesa, već mora da bude u glavnoj memoriji. Kada započne izvršavanje određenog procesa, registar

sadrži početnu adresu tabele segmenata za proces. Broj segmenta virtuelne adrese se koristi za indeksiranje te tabele i pronalaženje odgovarajuće adrese u glavnoj memoriji za početak segmenta. Da bi se izračunala željena adresa, dodaje se i deo sa pomerajem.

#### 6.5.9. Kombinacija straničenja i segmentacije

I straničenje i segmentacija imaju svoje dobre osobine. Straničenje je nevidljivo za programera, uklanja spoljašnju fragmentaciju i na taj način povećava efikasnost glavne memorije. Pored toga, pošto su delovi procesa koji se pomeraju u memoriji fiksne veličine, moguće je razviti napredne algoritme upravljanja memorijom koji koriste ponašanje programa. Segmentacija koja je vidljiva za programera omogućuje mu da rukuje rastućim strukturama podataka, modularnošću i podrškom deljenju i zaštiti. Da bi se objedinile prednosti oba pristupa, neki sistemi su opremljeni posebno prilagođenim procesorskim hardverom i softverom operativnog sistema. U kombinovanom sistemu straničenja i segmentacije, adresni prostor korisnika je podeljen na segmente, pri čemu programer zna koliko ih ima. Svaki segment je podeljen na stranice fiksne veličine, koja odgovara dužini okvira u glavnoj memoriji. Ako je veličina segmenta manja od veličine stranice, segment zauzima samo jednu stranicu. S tačke gledišta programera, logička adresa se i dalje sastoji od broja segmenta i pomeraja u segmentu. S tačke gledišta sistema, pomeraj u segmentu se vidi kao broj stranice i pomeraj u stranici za stranicu sa zadatim segmentom.



Slika 6.11: Kombinacija segmentacije i straničenja.

#### 6.5.10. Zaštita i deljenje

Sâma segmentacija primenjuje zaštitu i deljenje jer svaka tabela upisa segmenata sadrži dužinu i osnovnu adresu, pa program ne može da pristupi adresama van tog prostora. Da bi se ostvarilo deljenje, segment može da bude referenciran u tabelama segmenata nekoliko procesa. Isti mehanizam je moguće primeniti i za straničenje, ali se teže implementira jer struktura stranica programa i podataka nije vidljiva za programera. Napredniji mehanizam je korišćenje strukture zaštitnih prstenova. Svaki

prsten sa nižim brojem (ili unutrašnji prsten) ima veće privilegije od prstena sa većim brojem (spoljašnjeg prstena). Prsten broj 0 obično je rezervisan za funkcije jezgra operativnog sistema, a prsten najvećeg broja za korisničke aplikacije. Program može da pristupi samo podacima koji se nalaze u istom prstenu ili prstenu sa manjom privilegijom, odnosno da pozove servis koji je u istom prstenu ili prstenu sa većom privilegijom.

## 6.6. Operativni sistem i upravljanje memorijom

Tokom projektovanja dela operativnog sistema koji se bavi upravljanjem memorijom treba doneti tri važne odluke:

- ▷ da li će se koristiti virtuelna memorija,
- ▷ da li će se koristiti straničenje, segmentacija ili njihova kombinacija
- ▷ koji algoritmi će se koristiti za različite aspekte upravljanja memorijom

### 6.6.1. Politika učitavanja stranice u glavnu memoriju

Stranice procesa treba da se učitaju na zahtev, a može se iskoristiti i predstraničenje učitavanjem većeg broja susednih stranica odjednom.

### 6.6.2. Politika smeštanja stranica

Određuje gde u fizičkoj memoriji smestiti delove procesa. Kod sistema koji koriste isključivo segmentaciju, dolazećem segmentu mora da se nađe slobodan prostor odgovarajuće veličine.

### 6.6.3. Politika zamene stranica

Kada je memorija puna, mora da se doneše odluka koja stranica (ili stranice) mora da bude zamenjena. Ovaj deo tehnika upravljanja memorijom je verovatno onaj koji se najviše proučava. Kada su svi okviri u glavnoj memoriji zauzeti, a neophodno je učitati novu stranicu da bi se razrešila pogrešna stranica (page fault), politika zamene određuje koja stranica u memoriji treba da bude zamenjena. Cilj je pronaći stranicu koja neće biti referencirana u skorijoj budućnosti. Taj postupak se ne može primeniti jer zahteva da operativni sistem primeni predviđanje, ali daje optimalni algoritam koji služi za procenu kvaliteta drugih algoritama. Zbog principa lokaliteta, velika je verovatnoća da će stranice koje su poslednje referencirane biti referencirane u skorašnjoj budućnosti. Većina politika pokušava da predvidi buduće ponašanje na osnovu analize prethodnog ponašanja. Što je efikasnija i neprednija politika zamene, to su veći i hardverski i softverski troškovi primene tog rešenja. Postoje i određena ograničenja, na primer pojedini okviri u glavnoj memoriji mogu biti zaključani, pa se stranica koja

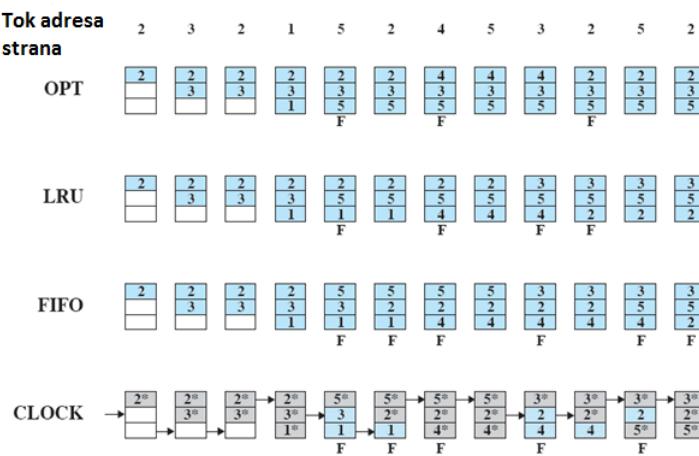
je privremeno smeštena u takvom okviru ne može zameniti. Osnovni algoritmi izbora stranica koje treba da se zamene su:

- ▷ optimalni (OPT)
- ▷ najmanje korišćena stranica u poslednje vreme (Least recently used – LRU)
- ▷ prva stranica učitana, prva zamenjena (First in - first out – FIFO)
- ▷ algoritam sata (CLOCK)

Poređenje algoritama za zamenu stranica je prikazano na Slici 6.12. Optimalni algoritam (OPT) se zasniva na tome da se za zamenu izabere stranica za koju je procenjeno vreme sledećeg referenciranja najduže. To podrazumeva da operativni sistem treba da "zna" buduće događaje, pa algoritam služi samo za poređenje. Politika zamene LRU koja koristi najmanje korišćenu (referenciranu) stranicu u poslednje vreme na osnovu principa lokaliteta treba da pronađe stranicu koja ne bi trebalo da bude referencirana u skoroj budućnosti. Ovo je slično optimalnom algoritmu. Problem kod ovog algoritma je to što se teško primenjuje i prouzrokuje velike dodatne troškove.

Algoritam FIFO (prva stranica učitana, prva zamenjena) posmatra okvire stranica dodeljene procesu kao kružni bafer. Stranice se uklanjuju po kružnom principu (round-robin). Ovaj algoritam zamene je najprostiji za primenu, a zamenjuje stranicu koja je najduže u memoriji. Loša osobina mu je to što će ta stranica možda potrebna u sledećem referenciranju.

Najprostija forma algoritma sata (CLOCK) zahteva dodatni bit koji se zove bit korišćenja. Kada se stranica po prvi put učita u okvir u memoriji, bit korišćenja za taj okvir se postavi na 1. Kada se stranica referencira, bit korišćenja se postavlja na 1. Kada treba da se zameni stranica, operativni sistem pregleda bafer da bi našao okvir sa bitom korišćenja postavljenim na 0. Tokom pretraživanja zbog zamene svaki bit korišćenja postavljen na 1 menja se u 0.



Slika 6.12: Poređenje algoritama zamene stranica.

#### **6.6.4. Upravljanje rezidentnim skupom**

Straničenjem virtuelne memorije nije neophodno i nije moguće učitati sve stranice jednog procesa u glavnu memoriju i pripremiti ih za izvršenje. Operativni sistem treba da odluci koliko stranica da učita u glavnu memoriju, odnosno koliko memorije da dodeli određenom procesu. Kod savremenih operativnih sistema postoji:

- ▷ Fiksna alokacija: dodeljuje procesu fiksan broj stranica unutar kojih se izvršava. Kada se pojavi pogrešna stranica, jedna od stranica procesa treba da se zameni.
- ▷ Promenljiva alokacija: dozvoljava da se procesu tokom vremena dodeljuje promenljiv broj okvira. Ako je broj okvira premali, pogrešne stranice će se brzo pojavljivati, a ako je alokacija prevelika, u glavnoj memoriji postojaće višak programa. Operativni sistem održava listu slobodnih okvira. Slobodan okvir se dodaje rezidentnom skupu procesa kada se pogrešna strana pojavi (to je tzv. lokalna zamena). Ako nema slobodnih okvira, zamenjuje se neki okvir drugog procesa (tzv. globalna zamena).

#### **6.6.5. Politika čišćenja**

Politika čišćenja je suprotna od politike učitavanja stranice. Izmenjene stranice procesa mogu biti upisane tokom zamene, ili se može iskoristiti politika pretčišćenja koja klasterizuje izlaznih aktivnosti upisujući odjednom veći broj stranica. Često se koristi pristup baferovanja stranica. Zamenjena stranica se smešta u dve liste: izmjenjenu i neizmjenjenu. Stranice u izmjenjenoj listi se periodično upisuju u blokovima.

#### **6.6.6. Politika kontrole**

Kontrola učitavanja bavi se određivanjem broja stranica koje će se nalaziti u glavnoj memoriji u bilo kom trenutku. Operativni sistem kontroliše nivo multiprogramiranja: suviše procesa može dovesti do thrashinga (odeljak 6.5.2). Ako stepen multiprogramiranja treba da se smanji, nekoliko tekućih rezidentnih procesa mora biti suspendovano, što se može utvrditi prema sledećim kriterijumima:

- ▷ proces sa najmanjim prioritetom,
- ▷ proces koji nema svoje stranice u glavnoj memoriji,
- ▷ poslednji proces koji je aktiviran,
- ▷ proces sa najmanjim rezidentnim skupom,
- ▷ najveći proces, itd.

## Glava 7

# Sistem datoteka

S tačke gledišta korisnika, jedan od najinteresantnijih delova operativnog sistema je sistem datoteka (file system). On obezbeđuje apstrakcije resursa koji su obično pri-druženi uređajima za sekundarno skladištenje. Sistem datoteka dozvoljava korisniku da kreira kolekcije podataka zvane **datoteke** (files) sa željenim osobinama kao što su:

- ▷ dugoročno postojanje: datoteke se skladište na disku ili nekom drugom sekundarnom skladištu i ne gube se nakon odjavljivanja korisnika ili isključivanja računara,
- ▷ mogućnost deljenja između procesa: datoteke imaju imena i mogu im se dodeliti dozvole pristupa koje kontrolišu deljenje
- ▷ struktura: zavisno od sistema, datoteka može da ima internu strukturu koja je pogodna za određene aplikacije.

Pored toga datoteka može biti organizovana hijerarhijski ili preko složenijih struktura kako bi se odrazila veza između datoteka. Svaki sistem datoteka obezbeđuje i skup funkcija koje se mogu primenjivati na datoteke. Tipične operacije uključuju:

- ▷ *Create*: definiše se nova datoteka i smešta u strukturu datoteka.
- ▷ *Delete*: datoteka se uklanja iz strukture datoteka i uništava.
- ▷ *Open*: proces deklariše da će otvoriti postojeću datoteku, a sistem dozvoljava procesu da izvrši funkcije nad datotekom.
- ▷ *Close*: datoteka se zatvara za dati proces, tako da on ne može više da izvršava funkcije nad datotekom sve dok je proces ponovo ne otvorи.
- ▷ *Read*: proces čita sve podatke iz datoteke (ili deo podataka).
- ▷ *Write*: proces ažurira datoteku tako što dodaje nove podatke koji joj povećavaju veličinu ili tako što menja vrednosti postojećih podataka u datoteci.

### 7.1. Struktura podataka

- ▷ *Polje* (field): osnovni element podataka. Jedno polje sadrži jednu vrednost, kao što je prezime studenta, datum ili očitana vrednost senzora. Polje je definisano svojom dužinom i tipom podatka (npr. ASCII string ili decimal).
- ▷ *Zapis* (record): skup povezanih polja koji aplikacija tretira kao jedinicu.
- ▷ *Datoteka* (file): skup sličnih zapisa. Korisnici i aplikacije datoteku mogu da referenciraju po imenu. Datoteke imaju imena i mogu se kreirati i brisati. Ograničenja

- pristupa obično se primenjuju na nivou datoteka. Kod nekih složenijih sistema moguća je kontrola na nivou zapisa, čak i polja.
- ▷ *Baza podataka* (database): baza podataka je skup povezanih podataka. Veza između podataka je eksplicitna; baza podataka projektovana je tako da je može koristiti veći broj aplikacija. Baza podataka se sastoji od jednog ili više tipova datoteka. Sistem za upravljanje bazama podataka obično je odvojen od operativnog sistema.

## 7.2. Osnovne operacije sa datotekama

- ▷ *Retrieve\_All*: čitanje svih zapisa datoteke.
- ▷ *Retrieve\_One*: čitanje samo jednog zapisa.
- ▷ *Retrieve\_Next*: čitanje zapisa koji je sledeći u nekom logičkom nizu u odnosu na poslednji dobijeni zapis.
- ▷ *Retrieve\_Previous*: čitanje zapisa koji prethodi poslednjem dobijenom zapisu.
- ▷ *Insert\_One*: umetanje novog zapisa u datoteku.
- ▷ *Delete\_One*: brisanje postojećeg zapisa.
- ▷ *Update\_One*: čitanje zapisa, ažuriranje jednog ili više polja, i ponovni upis ažuriranog zapisa u datoteku.
- ▷ *Retrieve\_Few*: čitanje više zapisa.

## 7.3. Sistemi za upravljanje datotekama

Sistem upravljanja datotekama je skup sistemskog softvera koji korisnicima i aplikacijama obezbeđuje usluge pri korišćenju datoteka. Jedini način da korisnik ili aplikacija pristupe datoteci je kroz sistem upravljanja datotekama. Ciljevi ovog sistema su sledeći:

- ▷ izaći u susret potrebama upravljanja podacima i zahtevima korisnika, što uključuje skladištenje podataka i sposobnost izvršenja ranije pomenutih operacija,
- ▷ garantovati da su podaci u datotekama ispravni,
- ▷ optimizovati karakteristike, sa sistemske tačke gledišta, u odnosu na celokupnu propusnost i sa tačke gledišta korisnika u odnosu na vreme odziva,
- ▷ obezbediti U/I podršku za veliki broj različitih tipova uređaja za skladištenje,
- ▷ minimizirati ili otkloniti potencijalne mogućnosti za gubljenje ili uništavanje podataka,
- ▷ obezbediti standardizovan skup U/I interfejsnih rutina za korišćenje procesa,
- ▷ obezbediti U/I podršku za više korisnika, za slučaj višekorisničkih sistema.

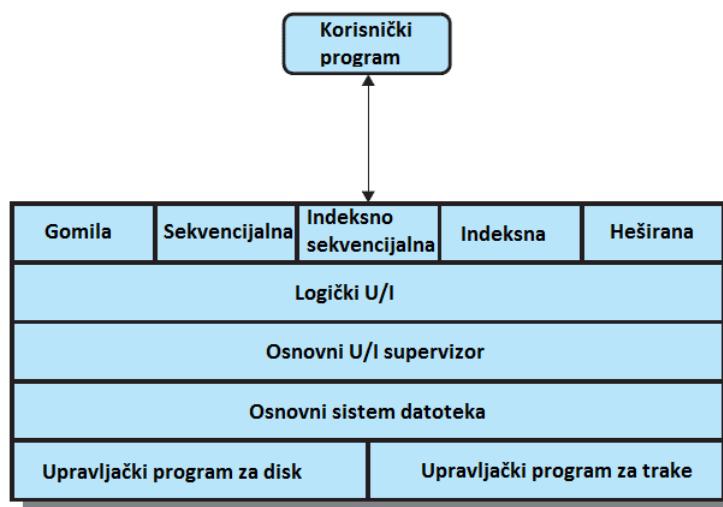
Za interaktivni sistem opšte namene definisamo minimalan skup zahteva za svakog korisnika da bi mogao da:

1. kreira, briše, čita, upisuje i menja datoteke

2. ima kontrolisan pristup datotekama drugih korisnika
3. kontroliše koji tipovi pristupa su dozvoljeni korisnicima datoteka
4. restrukturira datoteke u oblik koji odgovara problemu
5. premešta podatke između datoteka
6. sigurno uskladišti i povrati datoteke u slučaju oštećenja
7. pristupi datotekama koristeći simbolička imena.

#### 7.4. Arhitektura sistemskog softvera za rad sa datotekama

Tipična softverska organizacija data je na Slici 8.1.



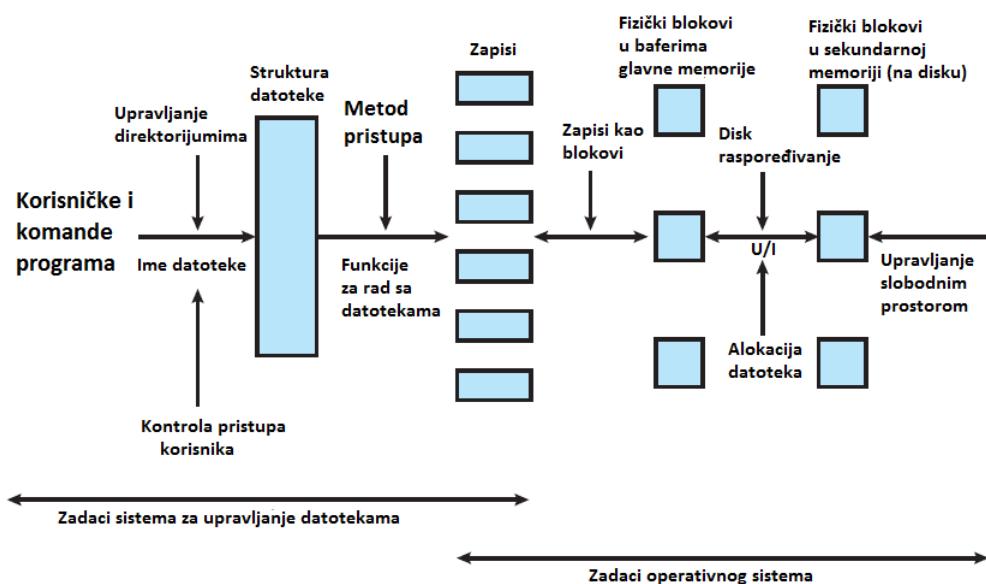
Slika 8.1: Sistemski softver za rad sa datotekama

Na najnižem nivou, upravljački programi (drajveri) uređaja komuniciraju direktno sa periferijskim uređajima ili njihovim kontrolerima ili kanalima. Upravljački program uređaja je odgovoran za pokretanje U/I operacija na uređaju i obradu U/I zahteva. Za operacije sa datotekama tipični uređaji su diskovi. Upravljački programi uređaja su obično sastavni deo operativnih sistema. Sledeći nivo je označen kao osnovni sistem datoteka, ili fizički U/I nivo. To je osnovni interfejs ka spoljašnjem okruženju računarskog sistema zadužen za smeštanje blokova na uređaj za sekundarno skladištenje i baferovanje tih blokova u glavnoj memoriji. Ovaj nivo se ne bavi sadržajem podataka ili struktura uključenih datoteka i deo je operativnog sistema.

Osnovni U/I supervizor je odgovoran za sve početne i završne U/I operacije sa datotekama. Na ovom nivou održavaju se kontrolne strukture koje se bave ulazom i izlazom uređaja, raspoređivanjem i statusom datoteka. Osnovni U/I supervizor bira

uredaj na kome će se izvršiti U/I operacije sa datotekom. On se takođe bavi raspoređivanjem i pristupom disku da bi se optimizovale performanse. Deo je operativnog sistema. U/I baferi se pridružuju, a sekundarna memorija se alocira. Logički U/I omogućuje korisnicima i aplikacijama da pristupe zapisima. Dakle, gde god se osnovni sistem datoteka bavi blokovima podataka, logički U/I modul se bavi zapisima datoteka. Nivo sistema datoteka najbliži korisniku često se označava kao metod pristupa. On obezbeđuje standardni interfejs između aplikacija i sistema datoteka s jedne strane, i uređaja koji čuvaju te podatke s druge strane.

### 7.5. Funkcije za upravljanje datotekama



Slika 8.2: Elementi upravljanja datotekama.

Korisnici i aplikacioni programi komuniciraju sa sistemom datoteka tako što šalju komande za kreiranje i brisanje datoteka ili izvršavanje operacija sa njima. Sistem datoteka pre svega treba da prepozna i locira izabranu datoteku. To zahteva korišćenje neke vrste direktorijuma koji služi za opis lokacije svih datoteka i njihovih atributa. Samo ovlašćeni korisnici imaju pristup određenim datotekama na određene načine. Osnovne operacije koje korisnik ili aplikacija mogu da izvrše sa datotekom izvršavaju se na nivou zapisa. Korisnici vide datoteku kao organizovanu sekvencijalnu strukturu zapisa (zapis korisnika su uskladišteni po abecednom redosledu od poslednjeg imena). Kada se korisnici i aplikacije bave zapisima, U/I je dat na osnovu blokova. Da bi se podržao U/I rad sa blokovima datoteke, potrebno je više funkcija. Neophodno

je upravljanje sekundarnom memorijom, što podrazumeva alociranje datoteka i slobodnih blokova. Disk raspoređivanje i alociranje datoteka se posmatraju kroz optimizaciju performansi.

## 7.6. Organizacija datoteka

Organizaciju datoteka definišemo preko logičkog strukturiranja zapisa, kao što je određeno načinom na koji im se pristupa. Fizička organizacija datoteke na uređaju za sekundarno skladištenje zavisi od strategije rada sa blokovima i strategije alokacije datoteke. U izboru načina organizovanja datoteka potrebno je voditi računa o više kriterijuma:

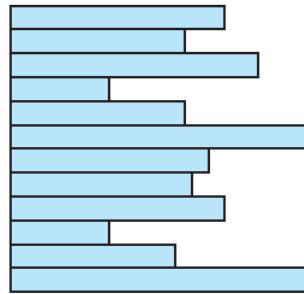
- ▷ kratko vreme pristupa
- ▷ lako ažuriranje
- ▷ ekonomičnost skladištenja
- ▷ lako održavanje
- ▷ pouzdanost

Relativan prioritet ovih kriterijuma zavisi od aplikacija koje koriste datoteku. Ovi kriterijumi su često međusobno u suprotnosti. Nabroјemo nekoliko alternativnih načina organizovanja datoteka koji se najčešće primenjuju:

- ▷ gomila podataka (pile)
- ▷ sekvencijalna datoteka
- ▷ indeksno sekvencijalna datoteka
- ▷ indeksna datoteka
- ▷ direktna (heširana) datoteka

### 7.6.1. Gomilanje podataka

Najprostiji oblik organizovanja datoteka je gomilanje podataka (pile). Podaci su skupljeni po redosledu kako stižu. Svaki zapis se sastoji od jedne grupe podataka. Svrha gomilanja je da prosto prikupi gomilu podataka i sačuva ih. Zapisi mogu da imaju različita polja, ili slična polja u različitom redosledu. Na taj način svako polje je samoopisujuće, uključujući i ime polja kao i vrednost. Dužina svakog polja mora biti implicitno označena delimiterima, eksplicitno uključena kao potpolje, ili standardna (default) za takav tip polja (Slika 8.3).

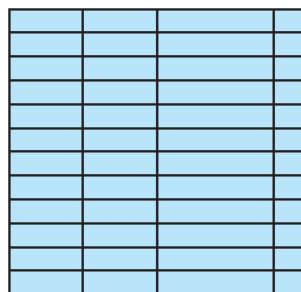


Slika 8.3: Gomilanje podataka (pile).

Pošto ne postoji struktura za “nagomilanu” datoteku, zapisu se pristupa nakon detaljnog pretraživanja. Ako želimo da nađemo zapis koji sadrži određeno polje sa određenom vrednošću, neophodno je ispitati sve zapise “na gomili” dok se ne pronađe željeni zapis, ili dok se ne pretraži cela datoteka. Gomilanje se koristi kod datoteka kada su podaci prikupljeni i uskladišteni pre obrade ili kada ih nije lako organizovati. Ovaj tip datoteka dobro koristi prostor kada su veličina i struktura uskladištenih podataka promenljivi i pogodan je kada su pretraživanja detaljna i laka za ažuriranje. Međutim, ovakav tip datoteka ne odgovara mnogim aplikacijama.

#### 7.6.2. Sekvencijalna datoteka

Najčešće primenjivan oblik strukture datoteke je sekvencijalna datoteka (Slika 8.4). Kod ovog tipa datoteke za zapise se koristi fiksni format. Svi zapisi su iste dužine, saставljeni od istog broja polja fiksne dužine. Pošto su dužina i pozicija svakog polja poznati, samo vrednosti polja treba da budu uskladištene; ime polja i dužina svakog polja su atributi strukture datoteke. Jedno posebno polje, obično prvo u svakom zapisu, označeno je kao **polje sa ključem** (key field).



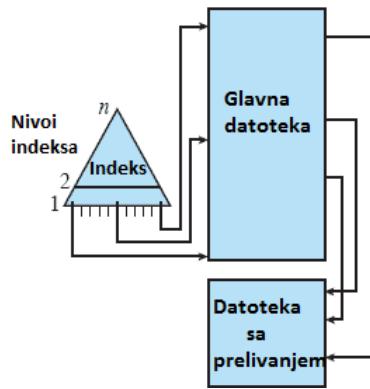
Slika 8.4: Sekvencijalna datoteka.

Polje sa ključem jedinstveno identificuje zapis; vrednosti ključeva za različite zapise su uvek različiti. Zapisi su uskladišteni kao sekvence ključeva: po abecedi za tekstualni ključ, odnosno po numeričkom redosledu za numerički ključ.

Sekvencijalne datoteke se obično koriste u paketnim aplikacijama i u opštem slučaju su optimalne za takve aplikacije ako one uključuju obradu svih zapisa. Sekvencijalno organizovanje datoteka se primenjuje za skladištenje na trakama i diskovima. Za interaktivne aplikacije koje uključuju upite i(lj) ažuriranje pojedinačnih zapisa, sekvencijalna datoteka daje slabe performanse. Pristup zahteva sekvencijalno pretraživanje datoteke na osnovu ključa. Ako cela datoteka ili njen veći deo može da se učita u glavnu memoriju odjednom, moguće su efikasnije tehnike pretraživanja. Međutim, da bi se pristupilo zapisu u velikoj sekvencijalnoj datoteci, obrada postaje obimna što prouzrokuje veliko kašnjenje. Pojavljuju se i problemi sa dodavanjem novog sadržaja datoteci. Sekvencijalna datoteka je po pravilu uskladištena po jednostavnom sekvencijalnom redosledu zapisa unutar blokova, što znači da se fizička organizacija datoteke na traci ili disku direktno poklapa sa njenom logičkom organizacijom. U ovom slučaju, uobičajeni postupak je da se novi zapisi smeste u odvojenu "nagomilanu" (pile) datoteku, nazvanu log datoteka, transakcionalna datoteka ili dnevnik. Periodično paketno (batch) ažuriranje se izvršava tako što spaja log datoteku sa glavnom datotekom da bi se kreirala nova datoteka po ispravnom sledu ključa. Alternativa je fizičko organizovanje sekvencijalne datoteke po principu ulančane liste. Jedan ili više zapisa se skladište u svaki fizički blok. Svaki blok na disku sadrži pokazivač na sledeći blok. Umetanje novih zapisa podrazumeva rad sa pokazivačima, ali ne zahteva da novi zapisi zauzmu određenu poziciju fizičkog bloka. Na taj način se ostvaruju dodatne pogodnosti u odnosu na vreme obrade i opšte troškove.

### 7.6.3. Indeksno sekvencijalna datoteka

Popularan pristup kojim se prevazilaze nedostaci sekvencijalnih datoteka jeste korišćenje indeksno sekvencijalnih datoteka. Indeksno sekvencijalna datoteka održava rad sa ključem kao sekvencijalna datoteka: zapisi su organizovani u sekvence na osnovu polja sa ključem. Dodatak su indeks datoteke za podršku slučajnom (random) pristupu i datoteke sa "prelivanjem" (overflow). Indeks ubrzava pronalaženje okruženja u blizini željenog zapisa. Datoteka sa prelivanjem je slična log datoteci koja se koristi sa sekvencijalnim datotekama, ali je integrisana. To znači da na zapis koji ne može da stane u datoteku "sa prelivanjem" pokazuje pokazivač poslednjeg zapisa u toj datoteci (Slika 8.5).



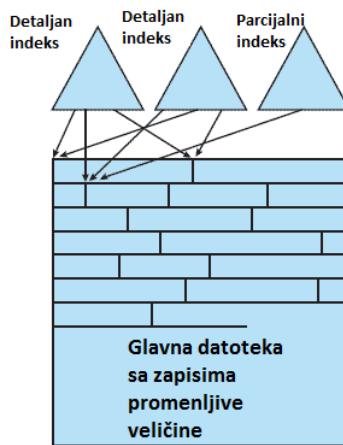
Slika 8.5: Indeksno sekvencijalna datoteka.

U najprostijoj indeksno sekvencijalnoj strukturi koristi se jedan nivo indeksiranja. Indeks je u ovom slučaju jedna sekvencijalna datoteka. Svaki zapis u indeksnoj datoteci sastoji se iz dva polja: polja sa ključem (key field), koje je isto kao polje sa ključem u glavnoj datoteci i pokazivač na glavnu datoteku. Da bi se pronašlo određeno polje, pretražuje se indeks da bi se našla najveća vrednost ključa koja je jednaka željenoj vrednosti ključa ili joj prethodi. Pretraživanje se nastavlja u glavnoj datoteci na lokaciji na koju pokazuje pokazivač. Dodavanje u datoteku se obavlja na sledeći način: svaki zapis u glavnoj datoteci sadrži jedno dodatno polje koje nije vidljivo za aplikaciju, a koje je pokazivač na datoteku sa prelivanjem. Kada novi zapis treba da se doda u datoteku, on se dodaje datoteci sa prelivanjem. Zapis u glavnoj datoteci koji istog trenutka prethodi novom zapisu u logičkoj sekvenci se ažurira tako da sadrži pokazivač na novi zapis u datoteci sa prelivanjem. Ako je prethodni zapis sâm u datoteci sa prelivanjem, onda se pokazivač na taj zapis ažurira. Kao i kod sekvencijalne datoteke, indeksna sekvencijalna datoteka se povremeno spaja sa datotekom sa prelivanjem u paketnom režimu.

Indeksno sekvencijalna datoteka primetno smanjuje vreme potrebno za pristup jednom zapisu, bez žrtvovanja sekvencijalne prirode datoteke. Da bi se sekvencijalno obradila cela datoteka, zapisi u glavnoj datoteci se obrađuju redom sve dok se nađe na pokazivač na datoteku sa prelivanjem; zatim se pristup nastavlja u datoteci sa prelivanjem dok se ne nađe na null pokazivač, kada se pristup nastavlja na mestu gde se stalo u glavnoj datoteci. Da bi se povećala efikasnost u pristupu može se koristiti indeksiranje u više nivoa. Na taj način se najniži nivo indeksne datoteke posmatra kao sekvencijalna datoteka, a najviši nivo indeksne datoteke se kreira za tu datoteku.

#### 7.6.4. Indeksna datoteka

Indeksna sekvencijalna datoteka zadržava jedno ograničenje sekvencijalne datoteke: efikasnost obrade je ograničena zbog toga što se zasniva na jednom polju datoteke. Kada je neophodno pretražiti zapis na osnovu nekog drugog atributa, a ne na osnovu polja sa ključem, nijedan oblik sekvencijalnih datoteka nije odgovarajući. U nekim aplikacijama ta funkcija je poželjna.



Slika 8.6: Indeksna datoteka.

Da bi se postigla ova prilagodljivost, potrebna je struktura koja koristi višestruke indekse, po jedan za svaki tip polja koje može biti subjekat pretraživanja. U opštim indeksnim datotekama, pojam sekvencijalnosti i jedan ključ se odbacuju. Zapisima se pristupa isključivo preko indeksa. Usled toga nema ograničenja smeštanja zapisa sve dok pokazivač na najmanje jedan indeks referencira taj zapis. Šta više, mogu da se iskoriste zapisi promenljive dužine. Koriste se dva tipa indeksa: detaljni (exhaustive) i parcijalni (partial). Detaljni indeks sadrži jedan upis za svaki zapis u glavnoj datoteci. Sâm indeks je organizovan kao sekvencijalna datoteka zbog lakog pretraživanja. Parcijalni indeks sadrži upise za zapise za koje postoji polje interesovanja. Ako se koriste zapisi promenljive dužine, neki zapisi neće sadržavati sva polja. Kada se novi zapis dodaje glavnoj datoteci svi indeksi datoteka moraju da budu ažurirani. Indeksne datoteke se najčešće koriste u aplikacijama u kojima su vremenske linije informacija kritične, a podaci se retko detaljno pretražuju (npr. rezervacije avionskih karata i kontrolni sistemi inventara).

#### 7.6.5. Direktne (heširane) datoteke

Ove datoteke koriste mogućnost direktnog pristupa bilo kom bloku poznate adrese na disku. Kao i sa sekvencijalnim i indeksno sekvencijalnim datotekama, za svaki za-

pis je neophodno polje sa ključem. Međutim, ovde nema sekvencijalnog redosleda. Direktne datoteke koriste heširanje vrednosti ključa u kombinaciji sa datotekama sa prelivanjem. Često se koriste u primenama gde se zahteva brzo pretraživanje, gde se koriste se zapisi fiksne dužine ili tamo gde se uvek pristupa samo jednom zapisu. Primeri su direktorijumi, tabele sa cenama, rasporedi i tabele sa imenima.

## 7.7. Direktorijumi datoteka

Direktorijumi datoteka su pridruženi svakom sistemu za upravljanje datotekama i skupu datoteka. Oni sadrže informacije o datotekama, uključujući attribute, lokacije i vlasništvo. Oprativni sistem upravlja većim delom ovih informacija, posebno onima koje su povezane sa skladištenjem. Sâm direktorijum je datoteka kojoj mogu da pristupe različite rutine za upravljanje datotekama. Mada su neke informacije u direktorijumima na raspolaganju korisnicima i aplikacijama, one se dobijaju indirektno, od sistemskih rutina. S tačke gledišta korisnika, direktorijum obezbeđuje mapiranje između imena datoteka, poznatih korisnicima i aplikacijama, i samih datoteka. Svaki upis, tj. ulaz u datoteku uključuje ime datoteke. Gotovo svi sistemi rade sa različitim tipovima datoteka i različitim organizacijama datoteka, pa su sve ove informacije na raspolaganju. Važna kategorija informacija o svakoj datoteci je skladištenje, uključujući lokaciju i veličinu. U deljenim sistemima važno je obezbediti informacije za kontrolu pristupa datoteci. Po pravilu je jedan korisnik vlasnik datoteke i može garantovati odredene privilegije pristupa drugim korisnicima. Na kraju, potrebne su korisne informacije za upravljanje tekućim korišćenjem datoteke, kao i zapisi istorije njenog korišćenja.

### 7.7.1. Struktura direktorijuma

Načini na koji se informacije o datotekama skladište veoma se razlikuju u različitim sistemima. Neke informacije se mogu uskladištiti u zaglavju zapisa pridruženog datoteci; to smanjuje količinu memorije potrebne za direktorijum, čineći ga lakšim za održavanje i ubrzava pristup u odnosu na direktorijum u glavnoj memoriji. Najprostiji oblik strukture za direktorijum je lista upisa; postoji po jedan upis za svaku datoteku. Ova struktura se može predstaviti jednom sekvencijalnom datotekom sa imenom datoteke koje služi kao ključ. Ovo nije pogodno kada više korisnika dele sistem, čak ni za jednog korisnika koji ima više datoteka. Da bi definisali zahteve za strukturu datoteka, objasnićemo operacije koje se mogu izvršiti na direktorijumu:

- ▷ *Pretraživanje*: kada korisnik ili aplikacija referencira datoteku, direktorijum mora da se pretraži da bi se našao upis koji odgovara toj datoteci.
- ▷ *Kreiranje datoteke*: kada se nova datoteka kreira, direktorijumu mora da se doda jedan upis.

- ▷ *Brisanje datoteke:* kada se datoteka obriše, upis mora da se ukloni iz direktorijuma.
- ▷ *Listanje direktorijuma:* ako se traži ceo direktorijum ili njegov deo, daje se spisak datoteka čiji je vlasnik korisnik koji je izdao zahtev zajedno sa određenim atributima za svaku datoteku (tip, kontrolne informacije pristupa, informacije o korišćenju).
- ▷ *Ažuriranje direktorijuma:* pošto se neki atributi datoteka skladište u direktorijumu, promena jednog atributa zahteva promenu u odgovarajućem upisu u direktorijumu.

Prosta lista nije pogodna za podršku ovim operacijama, pa se problem rešava šemama sa dva nivoa. Postoji jedan direktorijum za svakog korisnika i glavni (master) direktorijum. Master direktorijum ima po jedan upis za svaki korisnički direktorijum, a sadrži adresu i kontrolne informacije pristupa. Svaki korisnički direktorijum je lista datoteka tog korisnika. Zbog toga imena moraju da budu jedinstvena samo unutar skupa datoteka jednog korisnika, a sistem datoteka može lako da primeni ograničenja pristupa direktorijumima. Međutim, korisniku se ne obezbeđuje pomoć u strukturiranju skupa datoteka. Najprilagodljiviji pristup koji je skoro univerzalno prihvaćen je hijerarhijski pristup ili pristup strukture stabla. I tu postoji master direktorijum koji sadrži veći broj korisničkih direktorijuma. Svaki od ovih korisničkih direktorijuma može da ima poddirektorijume i datoteke kao upise. Svaki direktorijum je uskladišten kao sekvensijalna datoteka. Kada direktorijumi imaju veliki broj upisa, ovakva organizacija može da prouzrokuje nepotrebno duga vremena pretraživanja. U tom slučaju primenjuje se heš struktura.

### 7.7.2. Dodeljivanje imena

Korisnicima treba da se omogući da referenciraju datoteke simboličkim imenima. Svaka datoteka u sistemu mora da ima jedinstveno ime da bi reference na nju bile nedvosmislene. S druge strane, ne može se tražiti od korisnika da obezbede jedinstvena imena, naročito ne u deljenim sistemima. Korišćenjem direktorijuma strukture stabla minimiziraju se teškoće u dodeli jedinstvenih imena. Bilo koja datoteka u sistemu može da bude locirana praćenjem putanje od korena (root) ili master direktorijuma prema donjim granama dok se ne stigne do datoteke. Niz imena direktorijuma zajedno sa samim imenom datoteke definišu ime putanje (pathname) za datu datoteku. Dakle, datoteke mogu da imaju ista imena, ali imena putanja moraju da im budu različita da bi bile jedinstveno definisane u sistemu. Za interaktivne korisnike ili procese obično se definiše radni direktorijum (home directory). Datoteke se zatim referenciraju relativno u odnosu na taj radni direktorijum.

## 7.8. Deljenje datoteka

U višekorisničkom sistemu uvek postoje zahtevi za dozvolu deljenja datoteka između više korisnika. Pozabavilićemo se sa dva problema: pravom pristupa i upravljanjem istovremenim pristupom.

### 7.8.1. Prava pristupa

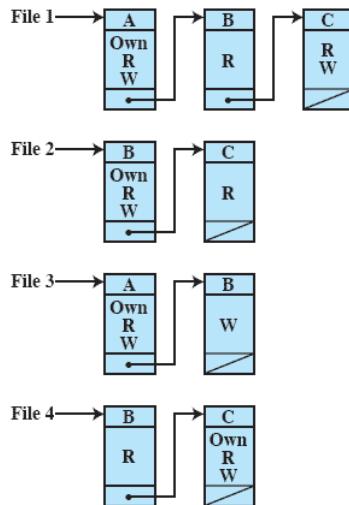
Sistem datoteka treba da obezbedi prilagodljiv način za deljenje datoteka između korisnika kojim se može kontrolisati pristup određenim datotekama. Korisnicima ili grupama korisnika se mogu dodeliti određena prava pristupa datoteci. Sledeća lista definije prava pristupa koja se mogu dodeliti određenom korisniku za određenu datoteku:

- ▷ *Bez prava pristupa*: korisnik ne samo da nema pravo da pristupi datoteci nego je ni ne vidi, odnosno ne može da pročita korisnički direktorijum koji uključuje ovu datoteku.
- ▷ *Znanje*: korisnik može da utvrdi da datoteka postoji i ko je njen vlasnik. Korisnik zatim može da traži od vlasnika dodatna prava pristupa.
- ▷ *Izvršavanje*: korisnik može da učita i izvrši program, ali ga ne može kopirati.
- ▷ *Čitanje*: korisnik može da čita sadržaj datoteke, kao i da je kopira i izvrši.
- ▷ *Dodavanje*: korisnik može da doda podatke datoteci, najčešće na kraju, ali ne može da menja ili briše postojeći sadržaj te datoteke.
- ▷ *Ažuriranje*: korisnik može da menja, briše i dodaje podatke datoteci.
- ▷ *Promena zaštite*: korisnik može da menja dodeljena prava pristupa drugim korisnicima. Ova prava obično ima samo vlasnik datoteke. U nekim sistemima vlasnik ima pravo da proširi prava na druge korisnike.
- ▷ *Brisanje*: korisnik ima pravo da izbriše datoteku iz sistema datoteka. Jedan korisnik se može označiti kao vlasnik datoteke, što je obično osoba koja je kreirala datoteku.

Vlasnik ima sva prava pristupa sa prethodne liste i može da dodeli drugim korisnicima ta prava. Pristup se može obezrediti različitim klasama korisnika:

- ▷ *Određeni korisnik*: individualni korisnici označeni korisničkim identitetom ID.
- ▷ *Grupe korisnika*: skup korisnika koji nisu definisani kao samostalni korisnici. Sistem mora na neki način da vodi računa o članstvu u grupama korisnika.
- ▷ *Svi korisnici*: svi korisnici imaju pristup sistemu. Ovo su javne datoteke.

Veza između korisnika i datoteka definiše se matricom pristupa ili kontrolnom listom pristupa kao na sledećoj slici.



Slika 8.7: Kontrolna lista pristupa

### 7.8.2. Istovremeni pristup

Kada se dodeli pristup jednom ili više korisnika, operativni sistem ili sistem za upravljanje datotekama moraju da vode računa o istovremenom pristupu datotekama zaključavajući ili celu datoteku ili pojedine zapise tokom ažuriranja. Ovo je problem čitača/pisača (odeljak 5.6.2). Prilikom projektovanja deljenog pristupa datotekama mora se voditi računa o međusobnom isključivanju i mogućnosti potpunog zastoja.

## 7.9. Zapis i blokovi (record blocking)

Kao što smo ranije objasnili, zapisi su logičke jedinice pristupa strukturiranim datotekama, a blokovi su U/I jedinice za sekundarno skladištenje. Za izvršavanje, U/I zapisi moraju se organizovati kao blokovi. Razmotrićemo nekoliko problema koji se pojavljuju.

Da li blokovi treba da budu fiksne ili promenljive dužine? U većini sistema blokovi su fiksne dužine, što može da uprosti U/I, alokaciju bafera u glavnoj memoriji i organizaciju blokova u sekundarnom skladištenju.

Kolika treba da bude relativna veličina bloka u odnosu na srednju vrednost veličine zapisa? Što je veći blok, to je više zapisa koji prolaze jednu U/I operaciju. Ako se datoteka obrađuje ili pretražuje sekvencijalno, to je prednost zato što se broj U/I operacija smanjuje ako se koriste veći blokovi i na taj način se ubrzava obrada. Ako se zapisima pristupa slučajno i ne posmatra se određeni lokalitet referenci, onda veći

blokovi prouzrokuju nepotreban prenos zapisa koji se ne koriste. Međutim, kombinovanjem brzine pojave sekvencijalnih operacija sa potencijalom lokaliteta referenci, vreme U/I prenosa opada sa korišćenjem većih blokova. Veći blokovi zahtevaju veće U/I bafere, otežavajući upravljanje baferima. Kada se određuje veličina bloka, mogu da se iskoriste sledeća tri metoda rada sa blokovima:

1. Rad sa fiksним blokovima: koriste se zapisi fiksne dužine sa integralnim brojem zapisa uskladištenim u bloku. Na kraju svakog bloka može se pojaviti neiskorišćen prostor (unutrašnja fragmentacija).
2. Rad sa spregnutim blokovima promenljive dužine: koriste se zapisi promenljive dužine koji se pakuju u blokove u kojima nema neiskorišćenog prostora. Jedan zapis može da bude spregnut u dva bloka; tada je nastavak zapisa definisan pokazivačem na drugi blok.
3. Rad sa nespregnutim blokovima promenljive dužine: koriste se zapisi promenljive dužine, ali se zapisi ne nalaze u različitim blokovima. U većini blokova postoji prostor koji je neiskorišćen zbog nemogućnosti da se iskoristi ostatak bloka ako je sledeći zapis veći od preostalog neiskorišćenog prostora.

Rad sa fiksnim blokovima je zajednički režim za sekvencijalne datoteke sa zapisima fiksne dužine. Rad sa spregnutim blokovima promenljive dužine je efikasan za skladištenje i nije ograničen veličinom zapisa. Ova tehnika je teška za primenu jer zapisi koji se nalaze na dva bloka zahtevaju dve U/I operacije pa se datoteke teško ažuriraju, bez obzira na to kako su organizovane. Rad sa nespregnutim blokovima promenljive dužine kao rezultat daje neiskorišćen prostor i zapise ograničene veličine u odnosu na veličinu bloka. Tehnike koje povezuju zapise i blokove mogu da sarađuju sa hardverom virtuelne memorije (ako on postoji). U okruženju virtuelne memorije poželjno je da stranica bude osnovna jedinica prenosa. Pošto su stranice su u opštem slučaju sasvim male, nepraktično ih je tretirati kao blok za rad sa nespregnutim blokovima. Neki sistemi kombinuju više stranica za kreiranje većeg bloka za U/I svrhe.

## 7.10. Upravljanje sekundarnim skladištenjem

U uređajima za sekundarno skladištenje datoteka se sastoje od skupa blokova. Operativni sistem ili sistem za upravljanje datotekama je odgovoran za alociranje (odelju) blokova datotekama. Razmotrićemo dva problema u upravljanju. Prvo, prostor sekundarnog skladištenja mora da bude dodeljen datotekama, i drugo, neophodno je voditi računa o prostoru koji se može iskoristiti za alokaciju. Ova dva zadatka su povezana.

Kada se kreira nova datoteka, prostor se dodeljuje datoteci kao jedna jedinica ili više susednih jedinica koje nazivamo **porcije** (portion). Porcija može da bude jedan blok, ali i celu datoteku. Primer ovakve strukture je tabela alokacije datoteka (file allocation table, FAT) koja je korišćena u DOS-u i nekim drugim operativnim sistemima.

### 7.10.1. Prealokacija i dinamička alokacija

Prealokacija zahteva da maksimalna veličina datoteke bude deklarisana u trenutku zahteva za njeno kreiranje. Međutim, za mnoge aplikacije veoma je teško (ako ne i nemoguće) proceniti koliko je procena maksimalne veličine datoteke pouzdana. U takvim slučajevima korisnici i programeri aplikacija težiće da precene veličinu datoteke da se ne bi desilo da ponestane prostora. U odnosu na alokaciju u sekundarnom skladištu, to je gubljenje vremena, pa se koristi dinamička alokacija kojom se datoteci dodeljuju potrebne porcije prostora.

#### Veličina porcije

Pri izboru veličine porcije treba uzeti u obzir efikasnost na nivou datoteke i efikasnost na nivou celog sistema. Postoji četiri parametra koje treba imati u vidu:

- ▷ Susedan prostor poboljšava karakteristike, posebno za operacije pretraživanja i rada sa transakcijama.
- ▷ Veliki broj malih porcija povećava tabele potrebne za upravljanje alokacijom.
- ▷ Porcije fiksne veličine (npr. blokovi) mogu uprostiti realokaciju prostora.
- ▷ Porcije promenljive veličine ili male fiksne veličine minimiziraju prostor izgubljen i neiskorišćen zbog prekomerne alokacije.

Ovi zahtevi se mogu razmatrati zajedno, a rezultat je da postoje dve glavne alternative.

1. Promenljive, velike susedne porcije: obezbeđuju bolje karakteristike. Promenljiva veličina umanjuje gubljenje prostora, a tabele alokacije datoteka su male. Međutim, prostor se teško ponovo koristi.
2. Blokovi: Male, fiksne porcije omogućuju veću prilagodljivost. One zahtevaju veće tabele ili složenije strukture za njihovu alokaciju. Susedstvo se odbacuje kao primarni cilj; blokovi se alociraju onoliko koliko je potrebno.

Obe opcije su kompatibilne sa prealokacijom ili dinamičkom alokacijom. U slučaju promenljive, velike susedne porcije, datoteci se unapred dodeljuju susedne grupe blokova, pa nema potrebe za tabelom alokacije datoteka: potreban je samo pokazivač na prvi blok i broj alociranih blokova. U slučaju blokova, sve potrebne porcije alociraju se odjednom, što znači da tabela alokacije za datoteku ostaje fiksne veličine. Sa porcijama promenljive veličine povezana je fragmentacija slobodnog prostora.

Moguće su sledeće alternativne strategije:

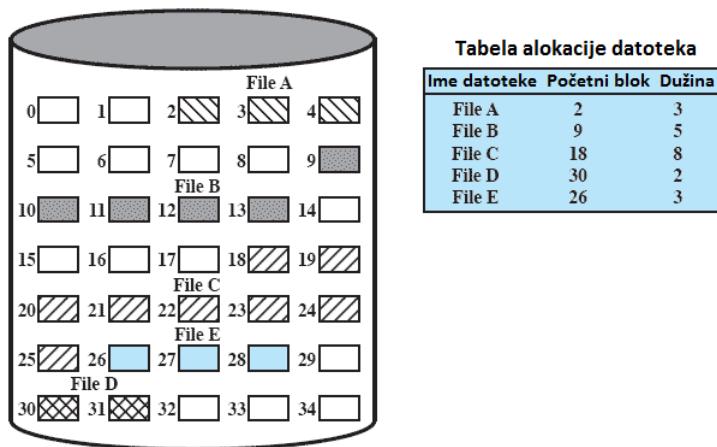
- ▷ Prvi koji odgovara (first fit): izabere se prva nekorišćena susedna grupa blokova dovoljne veličine sa liste slobodnih blokova.
- ▷ Najbolje odgovara (best fit): izabere se najmanje korišćena grupa dovoljne veličine.
- ▷ Najbliže odgovara (closest fit): izabere se nekorišćena grupa dovoljne veličine koja je najbliža prethodnoj alokaciji za datoteku da bi se povećao efekat lokaliteta.

Na modeliranje alternativnih strategija utiče veliki broj međusobno zavisnih faktora:

tipovi datoteka, uzorak pristupa datoteci, stepen multiprogramiranja, keširanje diska, raspoređivanje diska itd.

### 7.11. Metode alokacije datoteka

Najčešće se koriste tri metoda za alokaciju datoteka: susedna, ulančana i indeksna. Kod susedne alokacije, u trenutku kreiranja datoteci se dodeljuje skup susednih blokova. Na taj način definišemo strategiju prealokacije koristeći porcije, tj. delove promenljive veličine. Tabeli alokacije datoteka potrebni su samo jedan upis za svaku datoteku, pokazivač na početni blok i dužina datoteke. Susedna alokacija je najbolja ako se posmatra pojedinačna sekvenčnalna datoteka. U istom trenutku može da se pročita više blokova da bi se poboljšale U/I performanse za sekvenčnalnu obradu. Takođe, veoma je lako dobiti jedan blok.

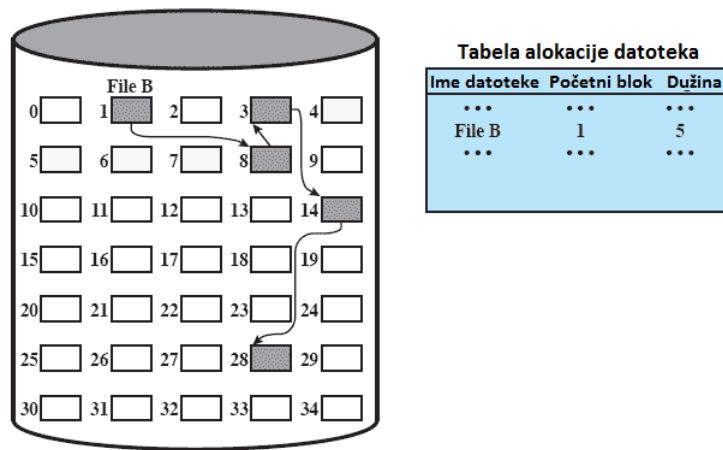


Slika 8.8: Susedna alokacija.

Kod susedne alokacije može da se pojavi spoljašnja fragmentacija, što otežava pronalaženje susednih blokova prostora dovoljne veličine. S vremena na vreme potrebno je primeniti algoritam sažimanja da bi se oslobođio dodatni prostor na disku (Slika 8.8). Takođe, u slučaju preallociranja, neophodno je deklarisati veličinu datoteke u vreme kreiranja, sa problemima koje smo naveli ranije. Kao suprotnost susednoj alokaciji definiše se ulančana alokacija (Slika 8.9). To je alokacija na osnovu pojedinačnog bloka. Svaki blok sadrži pokazivač na sledeći blok u lancu. Alokacionoj tabeli datoteka i u tom slučaju potrebni su samo jedan upis za svaku datoteku, početni blok za pokazivač i dužina datoteke.

Mada je prealokacija moguća, mnogo je jednostavnije alocirati blokove kada su potrebni. Izbor blokova je u tom slučaju veoma lak: bilo koji slobodan blok može

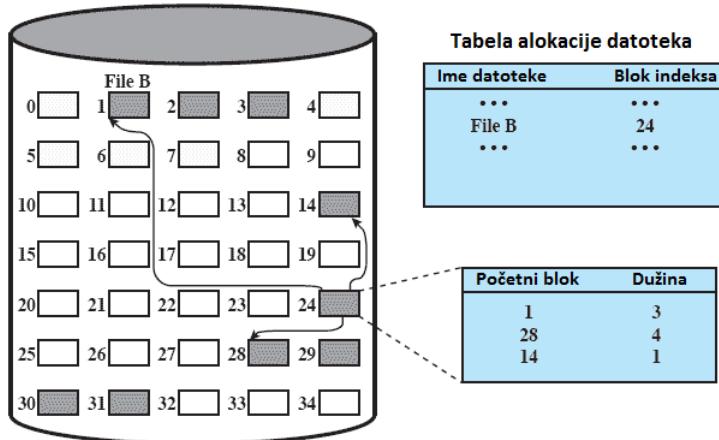
da se doda lancu. Ne postoji spoljašnja fragmentacija zato što je u svakom trenutku potreban samo jedan blok. Ovaj tip fizičke organizacije je najbolji za sekvencialne datoteke. Da bi se izabrao pojedinačni blok datoteke, mora se proći kroz lanac do željenog bloka.



Slika 8.9: Ulančana alokacija.

Jedna od posledica ulančavanja je to da se ne može primeniti princip lokaliteta. Ako je potrebno učitati nekoliko blokova datoteke u istom trenutku, kao kod sekvencialne obrade, potreban je niz pristupa različitim delovima diska. Da bi prevazišli taj problem, neki sistemi periodično konsoliduju datoteke. Indeksna alokacija rešava mnoge probleme susedne i ulančane alokacije. U ovom slučaju tabela alokacije datoteka sadrži odvojene indekse jednog nivoa za svaku datoteku; indeks ima jedan upis za svaku porciju dodeljenu datoteci. Indeksi datoteka obično nisu fizički usklađeni kao deo tabele alokacija datoteka. Umesto toga, indeksi jedne datoteke se drže u odvojenom bloku na koji pokazuje upis za datoteku u tabeli alokacije datoteka. Alokacija može biti zasnovana na blokovima fiksne veličine (Slika 8.10) ili porcijama promenljive veličine.

Alokacija pomoću blokova otklanja spoljašnju fragmentaciju, dok alokacija sa porcijama promenljive veličine pobožava lokalitet. U oba slučaja, s vremena na vreme mora se izvršiti konsolidacija datoteke. Ona smanjuje veličinu indeksa u slučaju porcija promenljive veličine, ali ne u slučaju alokacije blokova. Indeksna alokacija podržava i sekvencialni i direktni pristup datotekama, što je čini najpopularnijim oblikom alokacije datoteka.



Slika 8.10: Indeksna alokacija

## 7.12. Upravljanje slobodnim prostorom

Prostorom koji trenutno nije dodeljen nijednoj datoteci upravlja operativni sistem. Da bi prethodno opisane tehnike alokacije mogle da se primene, neophodno je da znamo koji su blokovi na raspolaganju disku. Uz tabelu alokacije datoteka potrebna nam je i tabela alokacije diska. Primjenjuje se više tehnika.

### 7.12.1. Tabele sa bitovima

Ovaj metod koristi vektor koji sadrži po jedan bit za svaki blok na disku. Svaki upis vrednosti 0 odgovara slobodnom bloku, a svaki upis vrednosti 1 odgovara bloku koji se koristi. Kod tabele sa bitovima veoma je lako pronaći susedne grupe slobodnih blokova. Iako su ove tabele veoma male, mogu znatno da opterete sistem. Količina memorije (u bajtovima) koja je potrebna za blok bitmappe je:

veličina diska u bajtovima / 8 x veličina bloka sistema datoteka

Tako za disk veličine 16GB sa blokovima veličine 512b tabela bitova zauzima približno 4 MB. Ako 4MB glavne memorije može da se "potroši" na tabelu bitova, onda tabela bitova može da se pretraži bez pristupa disku. Međutim, čak i sa današnjim memorijama, kapacitet od 4MB je preveliki deo glavne memorije da bi se posvetio jednoj funkciji. Rešenje je da se tabela bitova smesti na disk. Tabela veličine 4MB zahteva oko 8000 blokova na disku. Nije efikasno pretraživati toliki prostor na disku kad god je blok potreban. Čak i ako je tabela bitova u glavnoj memoriji, detaljno pretraživanje tabele može da uspori sistem datoteka na neprihvatljiv stepen, posebno kada je disk skoro pun i postoji samo nekoliko preostalih blokova. Većina sistema da-

toteka koja koristi tabele bitova održava nezavisne strukture podataka sa sadržajima podopsega tabele bitova.

### 7.12.2. Ulančane slobodne porcije

Slobodne porcije mogu da budu ulančane pomoću pokazivača i vrednosti dužine u svakoj slobodnoj porciji. Ovaj metod ima zanemarljive opšte troškove zato što više nije potrebna tabela alokacije diska, već samo pokazivač na početak lanca i dužina prve porcije. Ovaj metod je podesan za sve metode alokacije datoteka. Ako je alokacija u određenom trenutku jedan blok, treba samo izabrati slobodan blok sa vrha lanca i podesiti prvi pokazivač ili vrednost dužine. Ako je alokacija sa porcijama promenljive dužine, može se koristiti algoritam prvog podešavanja. Primena ovog metoda prouzrokuje izvesne probleme. Nakon određenog vremena disk postaje potpuno fragmentiran, sa mnogo porcija veličine jednog bloka. Takođe treba obratiti pažnju da prilikom alociranja bloka, pre nego što se podaci upišu u blok njega treba pročitati da bi se dobio pokazivač na nov prvi slobodan blok. Kada u određenom trenutku veoma veliki broj pojedinačnih blokova treba da bude alociran za rad sa datotekama, to primetno usporava kreiranje datoteka. Slično, brisanje veoma fragmentovanih datoteka zahteva mnogo vremena.

### 7.12.3. Indeksiranje

U pristupu indeksiranja slobodan prostor se posmatra kao datoteka, a tabela indeksa se koristi na sličan način kao kod alokacije datoteka. Zbog efikasnosti, indeks treba da bude definisan na osnovu porcija promenljive dužine umesto na osnovu blokova. Postoji jedan upis (stavka) u tabeli za svaku slobodnu porciju na disku. Ovaj pristup obezbeđuje efikasnu podršku za sve metode alokacije datoteka.

### 7.12.4. Lista slobodnih blokova

Svakom bloku se pridružuje sekvencijalni broj; lista brojeva svih slobodnih blokova se održava na rezervisanom delu diska. Zavisno od veličine diska, potrebna su 24 ili 32 bita za skladištenje jednog bloka brojeva, tako da je veličina liste slobodnih blokova 24 ili 32 puta veća od veličine odgovarajuće tabele bitova. Zbog toga se lista slobodnih blokova mora čuvati na disku umesto u glavnoj memoriji. Ovaj metod ipak daje zadovoljavajuće rezultate.

### **7.13. Pouzdanost**

Sistem održava kopiju tabele alokacije diska u glavnoj memoriji zbog efikasnijeg rada. Da bi se sprečile greške, tabela alokacije diska koja se nalazi na disku se zaključava sve dok se alokacija ne završi, da drugi korisnici ne bi pristupali tabeli. Često alociranje malih porcija bitno utiče na performanse sistema. Da bi se smanjili opšti troškovi, može se iskoristiti šema alokacije paketnog skladištenja. U ovom slučaju dobija se paket slobodnih delova diska za alokaciju. Odgovarajući delovi na disku koji su iskorišćeni su označeni. Alokacija može da se obavi u glavnoj memoriji. Kada je paket oslobođen, tabela alokacije diska se ažurira na disku i spremna je za nov paket. Ako sistem "zakaže", delovi diska koji su označeni kao iskorišćeni na neki način moraju da se "očiste" pre nego što se ponovo alociraju; tehnike čišćenja zavise od određenih karakteristika sistema datoteka.

## Glava 8

# Ulagno-izlazni (U/I) sistem

Najšarolikiji aspekt projektovanja operativnog sistema verovatno je U/I sistem. Pošto postoji veliki broj uređaja i aplikacija za U/I uređaje, veoma je teško razviti opšte, dosledno rešenje. Spoljašnji uređaji koji definišu U/I kao deo računarskog sistema mogu se grubo podeliti u tri kategorije:

- ▷ *Čitljivi*: služe za komunikaciju sa korisnicima računara (npr. štampači i terminali)
- ▷ *Mašinski čitljivi*: pogodni za komunikaciju sa elektronskom opremom (npr. diskovi, senzori, kontroleri, aktuatori).
- ▷ *Komunikacije*: pogodni za komunikaciju sa udaljenim uređajima (npr. digitalne linije, modemi).

Između ove tri klase uređaja postoje značajne razlike:

- ▷ Brzina podataka: brzina prenosa podataka može da se razlikuje za nekoliko redova veličine.
- ▷ Aplikacija: disk koji se koristi za skladištenje datoteka zahteva softver za upravljanje datotekama. Disk koji se koristi za skladištenje stranica u virtuelnoj memoriji zahteva specijalan hardver i softver. Terminali koje koriste administratori sistema ili obični korisnici imaju različite nivoe privilegija i prioriteta u operativnom sistemu.
- ▷ Složenost kontrola: različiti uređaji zahtevaju kontrolne interfejsne razlike složenosti.
- ▷ Jedinica prenosa: podaci mogu da se prenose kao nizovi bajtova ili znakova (terminal) ili kao veći blokovi (disk).
- ▷ Predstavljanje podataka: različiti uređaji koriste različite šeme kodiranja podataka, uključujući i razlike u kodiranju znakova i paritetu.
- ▷ Greške: priroda grešaka, način na koji se greške prijavljuju, njihove posledice i opseg odgovornosti veoma se razlikuju od uređaja do uređaja.

## 8.1. Organizacija U/I funkcija

Postoje tri tehnike za izvršavanje ulaza/izlaza:

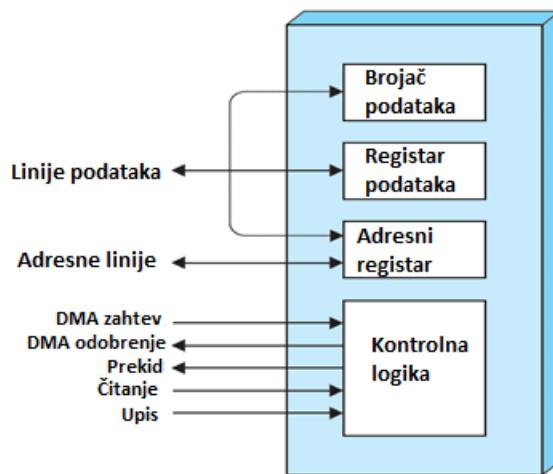
- ▷ Programirani U/I: proces izdaje komandu U/I modulu, i ne radeći ništa čeka da se operacija završi pre nego što nastavi posao.

- ▷ Prekidni U/I: proces izdaje U/I komandu i nastavlja da izvršava određene instrukcije; kada U/I modul završi svoj posao, prekida proces. Ako određene instrukcije nisu u istom procesu, nije neophodno da proces čeka na završavanje U/I operacije. U suprotnom, proces se suspenduje u toku prekida i izvršava se drugi posao.
- ▷ Direktan pristup memoriji (Direct Memory Access - DMA): DMA modul upravlja razmenom podataka između glavne memorije i U/I uređaja. Proces šalje zahtev za prenos bloka podataka DMA modulu i prekida se tek nakon prenosa celog bloka.

## 8.2. Direktan pristup memoriji (DMA)

Slika 7.1 prikazuje logiku direktnog pristupa memoriji. DMA tehnika omogućuje izbegavanje procesora i preuzimanje kontrole nad sistemom da bi se podaci prenosiли u memoriju i iz nje preko sistemske magistrale. DMA tehnika radi na sledeći način: kada procesor želi da učita ili upiše blok podataka, on izdaje komandu DMA modulu šaljući mu sledeće informacije:

- ▷ da li treba koristiti kontrolnu liniju za čitanje (read) ili upis (write) između procesora i DMA modula
- ▷ adresu U/I uređaja koji se koristi, uz komunikaciju preko linije podataka
- ▷ početnu lokaciju u memoriji iz koje se čita ili u koju se upisuje, uz komunikaciju preko linije podatake i čuvanje DMA modula u adresnom registru
- ▷ broj reči koji se upisuje ili čita, uz komunikaciju preko linije podatake i čuvanje DMA modula u brojaču podataka



Slika 7.1: Tipičan DMA blok dijagram

Pošto je dodelio U/I operaciju DMA modulu, procesor nastavlja da radi drugi posao. DMA modul prenosi čitav blok podataka, reč po reč, direktno u memoriju ili iz nje, bez prolaska kroz procesor. Kada se prenos završi, DMA modul šalje prekidni signal procesoru, što znači da je procesor uključen samo na početku i na kraju prenosa podataka.

### 8.3. Ciljevi u projektovanju U/I sistema

U dizajnu U/I sistema stalno su prisutna dva cilja: efikasnost i opštost. Efikasnost je važna zato što su U/I operacije često "usko grlo" u računarskom sistemu. Većina U/I uređaja je izuzetno spora u poređenju sa glavnom memorijom i procesorom. Jedan način na koji se rešava taj problema je multiprogramiranje, koje dozvoljava nekim procesima da čekaju na U/I operacije dok se drugi procesi izvršavaju.

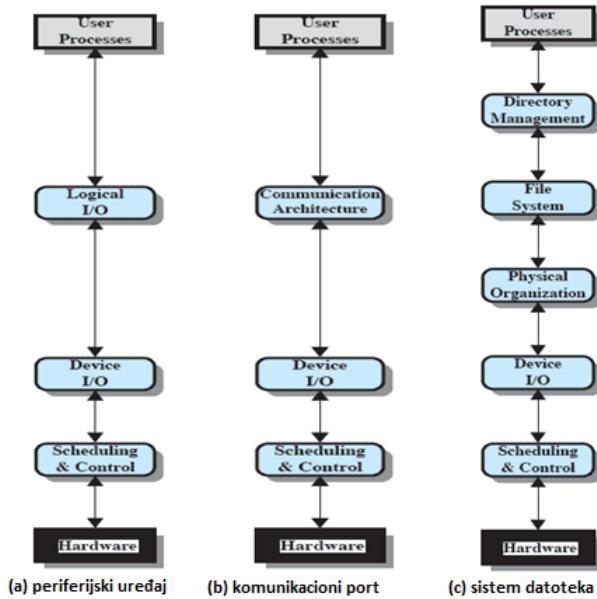
#### 8.3.1. Logička struktura U/I funkcije

Hijerarhijska priroda modernih operativnih sistema podrazumeva da funkcije operativnih sistema treba da budu klasifikovane prema složenosti, karakterističnim vremenima izvršavanja i nivoima apstrakcije. Dakle, operativni sistemi treba da budu organizovani u slojevima, pri čemu svaki sloj izvršava povezani skup funkcija. Svaki sledeći sloj nalazi se iznad nižeg sloja i prikriva detalje realizacije svojih funkcija, a obezbeđuje servise sledećim višim slojevima. Slojevi bi trebalo da budu definisani tako da promena u jednom sloju ne utiče na druge slojeve, jer se tako složeni problem deli na više potproblema kojima se može upravljati.

Na Slici 7.2 prikazana je organizacija U/I sistema koja koristi filozofiju slojeva. Detalji organizacije zavise od tipa uređaja i aplikacija. Predstavljene su tri najvažnije logičke strukture. U delu a) prikazan je lokalni periferni uređaj koji komunicira na prost način, npr. razmenjivanjem niza bajtova ili zapisa. Za takav uređaj uključeni su sledeći slojevi:

- ▷ Logički U/I (logical I/O): ovaj modul se bavi uređajem kao logičkim resursom, tj. upravlja opštim U/I funkcijama u odnosu na korisničke procese, dozvoljavajući im da se bave identifikacijom uređaja i komandama kao što su open, close, read, write, ...
- ▷ Uredaj U/I (device I/O): zahtevane operacije i podaci (znakovi i zapisi u baferima) moraju se konvertovati u odgovarajuće nizove U/I instrukcija, komandi sa kanalima i redosledne kontrolere. Za poboljšanje iskorišćenja mogu da se upotrebe tehnike rada sa baferima.
- ▷ Rasporedivanje i kontrola (scheduling and control): na ovom nivou obavlja se stvarno smeštanje u redove i rasporedivanje U/I operacija, kao i kontrola operacija. Takođe, na ovom nivou rukuje se prekidima, definiše se U/I status i daje kao

izveštaj. Ovo je sloj softvera koji zapravo komunicira sa U/I modulom, odnosno sa hardverom.



Slika 7.2: Model U/I organizacije

Za komunikacione uređaje, U/I struktura (Slika 7.2 b) liči na prethodno opisanu. Najvažnija razlika je u tome da je logički U/I modul zamjenjen komunikacionom arhitekturom koja se sastoji iz više slojeva (primer TCP/IP). Slika 7.2 c prikazuje reprezentativnu strukturu za U/I upravljanje na uređajima za sekundarno skladištenje koji podržavaju sistem datoteka. Vidimo tri dodatna sloja:

- ▷ Upravljanje direktorijumom: u ovom sloju simbolička imena datoteka se konvertuju u identifikatore koji referenciraju datoteku direktno ili indirektno kroz deksiptor datoteke ili tabelu indeksa. Ovaj sloj se bavi i korisničkim operacijama koje utiču na direktorijume, kao što su add, delete, reorganize, ...
- ▷ Sistem datoteka: ovaj sloj se bavi logičkom strukturu datoteka i operacijama koje određuje korisnik kao što su open, close, read, write, ... Na ovom nivou se definišu i prava pristupa.
- ▷ Fizička organizacija: slično virtuelnim memorijskim adresama koje se konvertuju u fizičke adrese glavne memorije, uzimajući u obzir strukture segmentacije i stranjenja, logičke reference na datoteke i zapise se konvertuju u fizičke adrese sekundarnog skladišta, uzimajući u obzir strukture fizičkih staza i sektora uređaja za sekundarno skladištenje. U ovom sloju se obavlja i alokacija prostora sekundarnog skladišta i upravlje se glavnim baferima za skladištenje.

### 8.3.2. U/I baferi

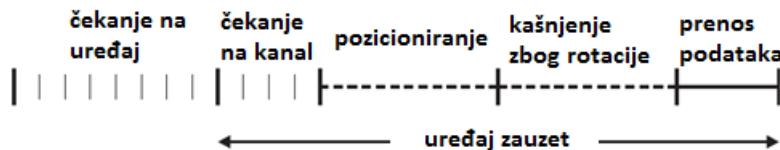
Da bi se izbegli opšti troškovi i neefikasnost pri U/I operacijama, pogodno je ulazni prenos izvršiti unapred, pre postavljanja zahteva, a izlazni prenos izvesno vreme posle zahteva. Ova tehnika je poznata kao **baferovanje** (buffering). Da bi se razumelo baferovanje, važno je razlikovati dva tipa U/I uređaja: one koji su orijentisani na rad sa blokovima i one koji su orijentisani na rad sa tokovima podataka (stream). Blok orijentisani uređaji skladište informacije u blokove koji su obično fiksne veličine, a u određenom trenutku prenosi se jedan blok. U opštem slučaju, moguće je referencirati podatke na osnovu broja bloka u kome se nalaze. Primeri blok orijentisanih uređaja su diskovi i trake. Uredaji orijentisani na rad sa tokovima prenose podatke u računar i iz njega kao nizove bajtova, tj. bez strukture bloka. Terminali, štampači, komunikacioni portovi, miševi i mnogi drugi uređaji koji ne služe za sekundarno skladištenje podataka su orijentisani ka tokovima. Baferovanje je tehnika uklanjanja „špiceva“ U/I zahteva. Baferovanje ne dozvoljava da U/I uređaj opslužuje proces beskonačno dugo, odnosno duže od srednjeg vremena servisiranja zahteva U/I uređaja. Čak i kada ima više bafera, svi baferi mogu biti puni, pa sistem mora da čeka na obradu podataka. U multiprogramskom okruženju u kome postoje različite U/I aktivnosti i servisne aktivnosti procesa, baferovanje je tehnika koji može da poveća efikasnost operativnog sistema i karakteristike pojedinačnih procesa.

## 8.4. Disk raspoređivanje

Brzina diskova je veoma ograničena u poređenju sa brzinama rada procesora i glavne memorije. Na primer, u poređenju sa glavnom memorijom, brzina diska je manja za četiri reda veličine, a očekuje se da će razlika u brzinama biti još veća u skorijoj budućnosti. Zato su karakteristike podsistema skladištenja podataka na disku veoma važne i stalno se istražuje kako da se poboljšaju.

### 8.4.1. Kašnjenje diskova

Stvarni detalji U/I operacija diska zavise od računarskog sistema, operativnog sistema i prirode U/I kanala i kontrolera diska. Opšti vremenski dijagram U/I disk prenosa prikazan je na Slici 7.3.



Slika 7.3: Vremenski prikaz U/I disk prenosa

Kada disk radi, rotira konstantnom brzinom. Kada se podaci čitaju ili upisuju, glava diska mora biti postavljena iznad željene staze i na početak željenog sektora na toj stazi. Vreme potrebno da se glava pozicionira iznad staze zove se **vreme pretraživanja** (seek time). Kada se staza izabere, kontroler diska čeka da odgovarači sektor rotira do linije gde se nalazi glava diska. Vreme potrebno da početak sektora stigne do glave diska zove se **kašnjenje zbog rotacije** (rotational delay ili rotational latency). Zbir vremena pretraživanja i kašnjenja zbog rotacije diska zove se **vreme pristupa** (access time). To je vreme potrebno da se dođe u poziciju čitanja ili upisa podataka, kada se sektor pomera ispod glave diska. Ovo je deo operacije upisa ili čitanja sa prenosom podataka, za koju se definiše i vreme potrebno za prenos podataka (transfer time). Osim vremena pristupa i vremena prenosa, postoji više kašnjenja zbog čekanja u redu na U/I operacije diska. Kada proces izda U/I zahtev, on prvo da čeka u redu dok uređaj ne postane dostupan, i tek tada se uređaj dodeljuje procesu. Ako uređaj deli jedan ili više U/I kanala sa drugim diskovima, onda postoji i dodatno čekanje da se oslobođi traženi kanal. Kada se kanal oslobođi, obavlja se pretraživanje i počinje pristup disku.

### Vreme pretraživanja

Vreme za koje se ruka diska pomeri do željene staze je vreme pretraživanja. Ono se sastoji od dve ključne komponente: početno startup vreme i vreme koje je potrebno da glava diska dođe do određene staze kada se potvrdi identifikacija staze. Kod savremenih diskova (veličine 3,5 inča – 8,9 cm) stalno se smanjuje rastojanje koje ruka diska treba da pređe. Tipično srednje vreme pretraživanja diska je kraće od 10 ms.

### Kašnjenje usled rotacije

Disk rotira brzinom koja je u opsegu od 3600 obrtaja u minuti (rpm) do 15000 rpm. Srednja vrednost kašnjenja usled rotacije je približno 2 ms.

### Vreme prenosa

Vreme prenosa ka disku ili sa diska zavisi od brzine rotacije diska na sledeći način:  
 $T = b/(rN)$

gde je T vreme prenosa, b broj bajtova koji treba da se prenesu, N broj bajtova na stazi i r brzina rotacije (broj obrtaja u sekundu). Prema tome, ukupno srednje vreme pristupa može da se predstavi kao:

$$Ta = Ts + 1/(2r) + b/(rN)$$

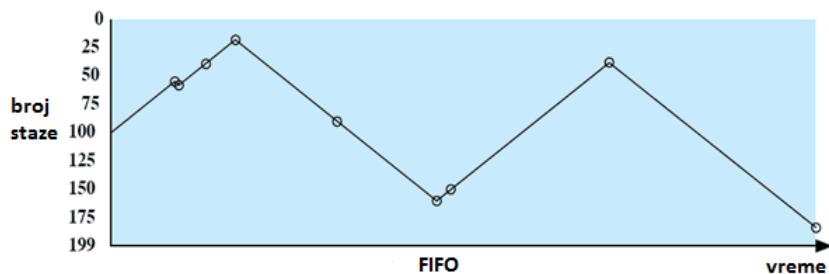
gde je Ts srednja vrednost vremena pretraživanja.

## 8.5. Algoritmi za raspoređivanje diska

Očigledno je da redosled po kome se čita sadržaj sektora sa diska ima ogroman uticaj na U/I karakteristike. Razlog zbog koga postoje razlike u karakteristikama je vreme pretraživanja. Ako zahtevi za pristup sektorima uključuju biranje staza na slučajan način, onda će karakteristike U/I disk sistema biti veoma slabe. Da bi se rešio taj problem, potrebno je smanjiti srednje vreme traženja odgovarajućih staza. Posmatrajmo tipičnu situaciju u multiprogramskom okruženju, u kome operativni sistem održava red sa zahtevima za svaki U/I uređaj. Za jedan disk postojaće veći broj U/I zahteva (čitanje i upis) od raznih procesa iz reda. Ako izaberemo jedan uzorak iz reda na slučajan način, onda možemo da očekujemo da će se staze koje će biti posećene pojavljivati na slučajan način, dajući slabe performanse. Slučajno raspoređivanje je korisno za poređenje sa ostalim tehnikama.

### 8.5.1. FIFO raspoređivanje

Prvi stigao, prvi servisiran (first in, first out – FIFO) je najprostiji oblik raspoređivanja koji obrađuje zahtev iz reda na sekvenčalan način. Prednost ove strategije je to što se ista pažnja poklanja svakom zahtevu prema redosledu pristizanja.



Slika 7.4: FIFO rasporedivanje.

Ako postoji samo nekoliko procesa koji zahtevaju pristup i ako ima mnogo zahteva koji traže klasterizovane sektore datoteke, karakteristike FIFO raspoređivanja će biti dobre. Međutim, ako se mnogo procesa takmiči da pristupi disku na slučajan način, moraju se koristiti druge, naprednije tehnike raspoređivanja.

Slika 7.4 prikazuje FIFO disk rasporedivanje. Vertikalna osa odgovara stazama na disku. Horizontalna osa odgovara vremenu, što je ekvivalentno broju predenih staza. Pretpostavljeno je da je početna lokacija glave diska na stazi 100, da je broj staza 200 i da red zahteva diska ima slučajno raspoređene zahteve. Zahtevi za stazama po redosledu primljenom od disk rasporedioca su: 55, 58, 39, 18, 90, 160, 150, 38 i 184.

### 8.5.2. Rasporedivanje po prioritetu

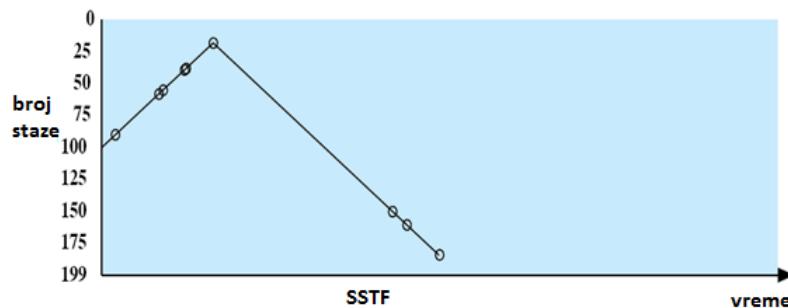
Kod ove vrste rasporedivanja kontrola rasporedivanja je van dometa softvera za upravljanje diskom. Ovaj pristup se ne koristi za optimizaciju iskorišćenja diska, već za druge ciljeve unutar operativnog sistema. Ovo rasporedivanje daje veći prioritet kratkim paketnim poslovima i interaktivnim poslovima, a postiže loše rezultate sa bazama podataka.

### 8.5.3. LIFO rasporedivanje

Ako se u sistemima za obradu transakcija uređaj dodeljuje poslednjem korisniku, kao rezultat može da se dobije malo ili nikakvo pomeranje ruke diska kroz sekvenčalnu datoteku. Uz pomoć dobrih osobina principa lokaliteta može da se poboljša propusnost i smanji dužina reda. Taj princip se koristi u rasporedivanju tipa "poslednji stigao, prvi uslužen" (Last In First Out – LIFO). Što duže posao aktivno koristi sistem datoteka, obraduje se brže. Međutim, ako je radno opterećenje diska visoko, može se pojavit „gladovanje“ (tj. da se zahtevi koji nikako neće stići na red). FIFO i LIFO rasporedivanja zasnivaju se isključivo na atributima reda sa zahtevima. Ako rasporedivač zna tekuću poziciju staze, onda se može primeniti rasporedivanje zasnovano na zahtevanom uzorku.

### 8.5.4. SSTF rasporedivanje

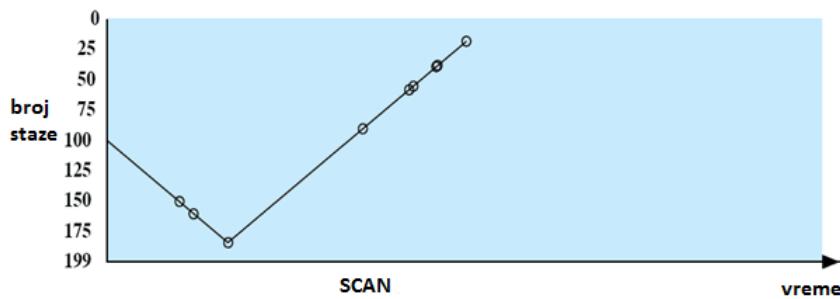
SSTF algoritam (najkraće vreme servisiranja prvo, Shortest Service Time First – SSTF) bira zahtev U/I diska koji zahteva najmanje pomeranje ruke diska od tekuće pozicije. Uvek se bira zahtev sa minimalnim vremenom pretraživanja. To ne garantuje da će srednje vreme pretraživanja većeg broja pomeranja ruke diska biti minimalno. Algoritam ima bolje karakteristike od FIFO algoritma. Slika 7.5 prikazuje performanse SSTF rasporedivanja na istom primeru kao i FIFO.



Slika 7.5: SSTF

### 8.5.5. SCAN raspoređivanje

Uz izuzetak FIFO raspoređivanja, u svim dosad opisanim algoritmima može da se desi da neki zahtevi ostanu neizvršeni sve dok se ceo red ne isprazni. Uvek mogu da stignu novi zahtevi koji će biti izabrani pre postojećeg zahteva. Jednostavno rešenje koje sprečava pojavu "gladovanja" (starvation) je SCAN algoritam, poznat i kao algoritam lifta. Ruka diska se pomera samo u jednom smeru zadovoljavajući sve zahteve na tom putu, sve dok ne stigne do poslednje staze u tom smeru ili dok više nema zahteva u tom smeru. Nakon toga se smer menja i skeniranje se nastavlja u suprotnom smeru uz rešavanje svih zahteva na tom putu redom (Slika 7.6).

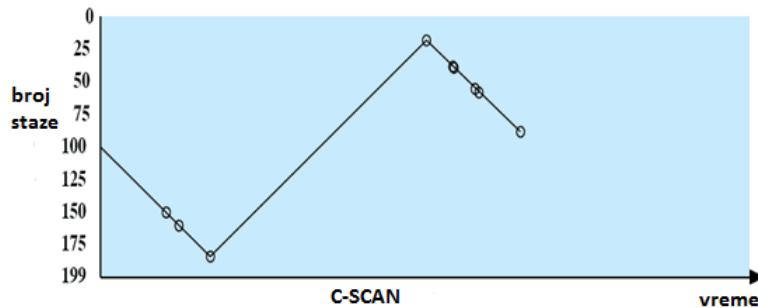


Slika 7.6: SCAN

Kada se redovi dinamički menjaju, SCAN raspoređivanje je slično SSTF raspoređivanju sve dok uzorci zahteva ne postanu neuobičajeni. SCAN, kao i SSTF i LIFO, ne koristi efekat lokaliteta. SCAN algoritam daje prednost poslovima čiji su zahtevi na stazama blizu unutrašnje i spoljašnje strane diska i nedavno pristiglim poslovima. Prvi problem se rešava C-SCAN algoritmom, a drugi algoritmom N-step(SCAN).

### 8.5.6. C-SCAN raspoređivanje

Kružni SCAN algoritam ograničava skeniranje samo u jednom smeru. Kada se poseti poslednja potrebna staza u jednom smeru, ruka diska se vraća na suprotan kraj diska i skeniranje počinje ispočetka. Time se smanjuje maksimalno kašnjenje (Slika 7.7).



Slika 7.7: C-SCAN

#### 8.5.7. N-step-SCAN i FSCAN raspoređivanje

N-step-SCAN algoritam deli zahteve za diskom koji su u redu na podredove dužine N. Podredovi se obrađuju jedan po jedan primenom SCAN algoritma. Kada se red obradi, novi zahtevi se mogu dodati nekom drugom redu. Ako ima više od N zahteva na kraju skeniranja, onda se svi oni obrađuju u sledećem skeniranju. Sa velikim brojem N ovaj algoritam ima karakteristike kao SCAN, a za N=1, postaje FIFO.

FSCAN je algoritam koji koristi dva podreda. Kada skeniranje počne, svi zahtevi su u jednom podredu, a drugi je prazan. Svi novi zahtevi se stavljaju u drugi podred, čime se odlaže njihovo izvršavanje sve dok se svi stari zahtevi ne obrade.

### 8.6. Redundantan niz nezavisnih diskova (RAID)

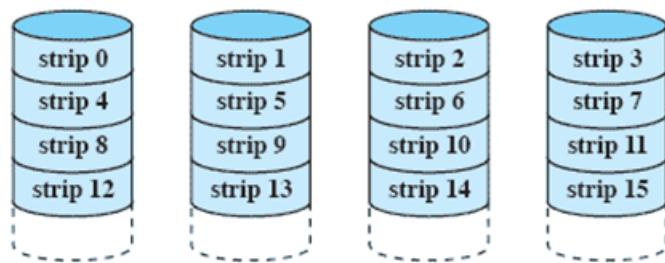
Kao što smo ranije pomenuli, karakteristike uređaja za sekundarno skladištenje se mnogo sporije poboljšavaju od brzine procesora i glavne memorije. Jedan od načina poboljšanja karakteristika skladištenja na disku je primena niza diskova koji rade nezavisno i paralelno. Kada u sistemu ima više diskova, odvojeni U/I zahtevi mogu da se izvršavaju paralelno samo ako se podaci nalaze na više diskova. Jedan U/I zahtev može da se izvrši paralelno samo ako je blok podataka kome se pristupa distribuiran na više diskova. Standardizovana šema za rad sa više diskova poznata je kao RAID (Redundant Array of Independent Disks). Ova šema se sastoji od sedam nivoa (od 0 do 6) koji ne predstavljaju hijerarhijske veze, već arhitekture različitog dizajna koje dele tri zajedničke karakteristitke:

1. RAID je skup fizičkih diskova koje operativni sistem vidi kao jedan logički disk
  2. Podaci su distribuirani na nizu fizičkih diskova.
  3. Redundantni (dodatni) diskovi se koriste za skladištenje informacija o parnosti, koje garantuju da se u slučaju greške podaci mogu povratiti.
- RAID 0 i 1 ne podržavaju treću karakteristiku. U tehnici RAID, diskovi velikog kapaciteta zamenjuju se s nekoliko diskova manjeg kapaciteta na kojima su distribuirani

podaci da bi se omogućio istovremeni pristup podacima sa više diskova. Na taj način se poboljšavaju U/I performanse. RAID šema ubrzava rad, ali povećava i verovatnoću pojave greške. Zato se koriste informacije o parnosti (parity) koje omogućavaju "oporavak" (tj. ponovno nalaženje) podataka izgubljenih zbog greške na disku. Prikazaćemo detaljnije nivoe 0, 1 i 5 koji se najčešće primenjuju.

### 8.6.1. RAID 0 (bez redundantnosti)

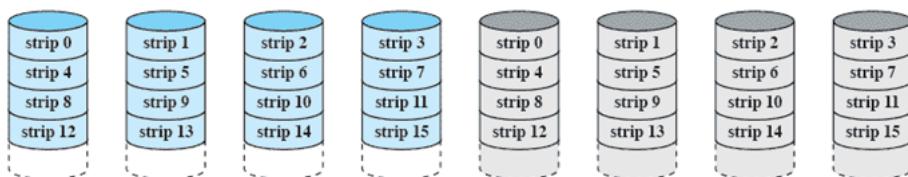
Ovaj nivo jednostavno raspodeljuje podatke na niz diskova (Slika 7.8). Sistem radi sa logičkim diskovima koji su podeljeni na "trake" (strip) koje mogu biti fizički blokovi, sektori ili neke druge jedinice. Trake su mapirane na kružni način (round robin) na susedne fizičke diskove u RAID nizu. Prednost ove šeme je paralelno izvršavanje U/I zahteva na trakama.



Slika 7.8: RAID 0

### 8.6.2. RAID 1 (preslikavanje)

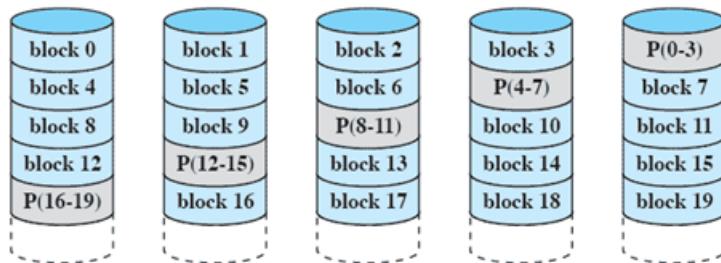
U tehnici RAID 1 (Slika 7.9) svaka logička traka je preslikana na dva posebna fizička diska. Za svaki disk u nizu postoji disk "u ogledalu" (mirror) koji sadrži identične podatke. Ako se na jednom disku pojavi greška, podaci se mogu dobiti sa drugog diska.



Slika 7.9: RAID 1

### 8.6.3. RAID 5 (distribuirana parnost na nivou blokova)

RAID 5 koristi tehniku nezavisnog pristupa, u kojoj svaki disk radi nezavisno tako da odvojeni U/I zahtevi mogu da se izvršavaju paralelno (Slika 7.10). I ovde se koristi podela podataka u trake. U odgovarajućim trakama na svim diskovima čuvaju se bitovi parnosti, čime se izbegava potencijalno U/I "usko grlo" jednog paritetnog diska. Tipična alokacija je kružna (round robin) šema.



Slika 7.10: RAID 5

Preostale RAID šeme su: RAID 2 (redundantnost uz Hammingov kod), RAID 3 (isprepletena parnost na nivou bitova), RAID 4 (parnost na nivou blokova) i RAID 6 (dvojna redundantnost).

## 8.7. Disk keš

Termin keš memorija se obično koristi za memoriju koja je manja i brža od glavne memorije, a smeštena je između glavne memorije i procesora. Takva keš memorija smanjuje srednje vreme pristupa memoriji korišćenjem principa lokaliteta. Isti princip se može primeniti na disk memoriju. Disk keš je bafer u glavnoj memoriji za sektore sa diska. Keš sadrži kopije određenih sektora na disku. Kada se pošalje jedan U/I zahtev za određeni sektor, prvo se proverava da li je sektor u disk kešu. Ako jeste, zahtev se odatle zadovoljava, a ako nije zahtevani sektor se učitava sa diska u disk keš. Zbog principa lokaliteta referenci, ako je blok podataka učitan u keš da bi zadovoljio jedan U/I zahtev, vrlo je verovatno da će uskoro biti ponovo referenciran.

**Deo II**

## **Jezgro operativnog sistema Linux**



## Glava 9

# Uvod u jezgro (kernel) Linuxa

### 9.1. Operativni sistem UNIX kao preteča Linuxa

Nakon četiri decenije korišćenja, operativni sistem UNIX je još uvek jedan od naj-snažnijih i najelegantnijih operativnih sistema. Kreirali su ga 1969. godine Denis Riči i Ken Tompson. UNIX je nastao iz Multicsa, projekta višekorisničkog operativnog sistema u Bellovim laboratorijama. Nakon što se projekat Multics neuspešno završio, članovi istraživačkog centra ostali su bez posla na razvoju interaktivnog operativnog sistema. U letu 1969. godine programeri Belovih laboratorijski skicirali su sistem datoteka koji će kasnije postati deo UNIX-a. Tokom testiranja tog dizajna, Ken Tompson je primenio nov sistem na PDP-7. Godine 1973. operativni sistem je ponovo napisan u jeziku C, što je otvorilo put budućoj prenosivosti sistema. Prva verzija UNIX-a koja je bila u širokoj upotrebi van Bellovih laboratorijskih bilo je šesto izdanje, poznato pod nazivom V6. Nakon toga i druge kompanije počele su da instaliraju UNIX na novim mašinama. Pojavilo se nekoliko njegovih varijanti: 1977 Unix System III, a 1982 kompanija AT&T je razvila System V Release 4 (SVR4). Jednostavnost dizajna povezana sa činjenicom da je distribuiran sa izvornim (source) kodom dovela je do brzog razvoja ovog operativnog sistema. Najpoznatije varijante su stigle sa Univerziteta u Berkliju, pod nazivom Berkeley Software Distributions (BSD). U tim verzijama UNIX-a dodati su virtuelna memorija, straničenje na zahtev i protokoli TCP/IP. Poslednje varijante su Darwin, Dragonfly BSD, FreeBSD, NetBSD i OpenBSD sistemi. Poznate kompanije su razvijale sopstvene komercijalne verzije UNIX-a sa višestrukim radnim stanicama i serverima kao što su Digitalov Tru64, Hewlett Packardov HP-UX, IBM-ov AIX, Sequentov DYNIX/ptx, SGI-jev IRIX i Sunov Solaris. Kao dobre osobine UNIX-a pokazale su se mali broj (nekoliko stotina) sistemskih poziva, pregledan dizajn, sistem datoteka. Ove osobine uprošćuju rad sa podacima i uređajima svodeći ga na skup prostih sistemskih poziva: `open()`, `read()`, `write()`, `ioctl()` i `close()`. Uz to, jezgro UNIX-a i sa njim povezani sistemski servisi napisani su u jeziku C, što omogućuje prenosivost. UNIX veoma brzo kreira procese i ima jedinstven sistemski poziv `fork()`. Konačno, UNIX obezbeđuje prostu ali robusnu komunikaciju između procesa. UNIX je danas moderan operativni sistem koji podržava multitasking, višenitni rad, virtuelnu memoriju, straničenje na zahtev, deljene biblioteke sa učitavanjem na zahtev i TCP/IP.

umrežavanje. Postoje varijante koje rade sa stotinama procesora, kao i one naznjene malim, "embedded" uređajima.

## 9.2. Nastanak Linuxa

Linux je razvio student Univerziteta u Helsinkiju Linus Torvalds 1991. godine kao operativni sistem za računare koji koriste Intelove mikroprocesore 80386. Linus je koristio Minix, jeftin Unix kreiran kao sredstvo za učenje, ali zbog licenci koje su usporavale distribuciju izmena u izvornom kôdu, odlučio se da napiše sopstveni operativni sistem. Prvo je napisao prost emulator za terminale koji je koristio za povezivanje sa većim Unix sistemom na fakultetu, a zatim ga je razvijao i poboljšavao. Godine 1991. postavio je prvu verziju Linuxa na Internet. Nakon toga Linux je postao kolaborativni projekat u čijem razvoju učestvuju mnogi programeri. Linux radi na sledećim procesorima: AMD x86-64, ARM, Compaq Alpha, CRIS, DEC VAX, H8/300, Hitachi SuperH, HP PA-RISC, IBM S/390, Intel IA-64, MIPS, Motorola 68000, PowerPC, SPARC, UltraSPARC i v850. Pojavile su se nove kompanije specijalizovane za Linux kao što su MontaVista i Red Hat, a velike kompanije kao što su IBM i Novell obezbeđuju Linux rešenja za embedded sisteme, desktop sisteme i servere.

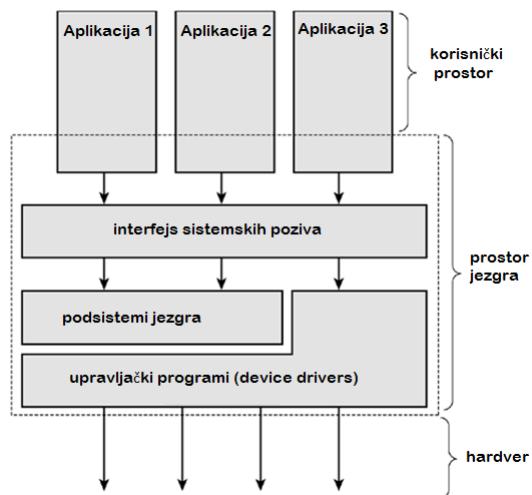
Linux je klon UNIX-a, ali nije UNIX. Mada je pozajmio mnoge ideje od UNIX-a i primenjuje njegov interfejs za programiranje API (kao što je definisano standardom POSIX), njegov kôd on nije direktna kopija izvornog kôda UNIX-a kao u drugim UNIX sistemima. Jedna od najinteresantnijih karakteristika Linuxa jeste to da nije komercijalni proizvod, već projekat razvijen saradnjom preko Interneta. Iako Linus Torvalds ostaje kreator Linuxa i održava kernel, razvoj nastavlja povezana grupa programera. Linuxov kernel je slobodan softver, tj. softver otvorenog koda sa licencem GNU General Public License (GPL, verzija 2.0). Korisnici su slobodni da preuzimaju izvorni kôd Linuxa sa Interneta i da načine bilo kakve promene. Jedina obaveza koju imaju je da ako distribuiraju svoje promene, moraju korisnicima da daju ista prava koja sami koriste, uključujući i raspolaganje izvornim kodom.

Osnovni delovi operativnog sistema Linux su kernel, C biblioteka i prevodilac, niz alata i osnovne sistemske funkcije (kao što su proces prijavljivanja i interpreter komandi). U neke Linux sisteme uključeno je desktop okruženje X Window System sa svim karakteristikama (GNOME, KDE). Postoji ogroman broj slobodnih komercijalnih aplikacija za Linux. Kada se kaže Linux najčešće se misli na kernel, odnosno jezgro samog operativnog sistema.

## 9.3. Pregled jezgra operativnih sistema

Operativni sistem se posmatra kao skup delova sistema odgovornog za osnovno korišćenje i administriranje računara. Pod tim se podrazumevaju jezgro (kernel),

upravljački programi uređaja (drivers), softver za podizanje sistema (boot loader), interpreter komandi (command shell), korisnički interfejsi za rad sa datotekama i sistemske usluge. Termin sistem upućuje na operativni sistem i sve aplikacije koje rade u njemu. Jezgro ili kernel je softver koji obezbeđuje osnovne servise za sve delove sistema, upravlja hardverom i raspoređuje sistemske resurse. Kernel se ponekad naziva unutrašnjost operativnog sistema. Tipične komponente kornela su prekidne rutine, tj. upravljači prekida (interrupt handlers) koje obrađuju zahteve za prekidom, rasporedioca za deljenje vremena procesora između više procesa (scheduler), sistem za upravljanje memorijom, tj. adresnim prostorom procesa i sistemski servisi kao što su umrežavanje i komunikacija između procesa. Kernel uključuje i zaštitu memorijskog prostora i pun pristup hardveru. Stanje sistema i memorijski prostor zajedno se zovu kernel prostor. Korisničke aplikacije se izvršavaju u korisničkom prostoru. Aplikacije koje se izvršavaju u sistemu komuniciraju sa kernelom preko sistemskih poziva. Neki bibliotečki pozivi imaju širu funkciju od sâmog sistemskog poziva; tada je poziv u kernelu samo jedan korak u funkciji (takva je npr. C funkcija printf() koja obezbeđuje formatizovanje i baferovanje podataka, a samo eventualno pozive write() za ispis podataka na konzolu). Nasuprot tome, neki bibliotečki pozivi imaju vezu sa kernelom "jedan-na-jedan" (npr. bibliotečka funkcija open() svodi se na čist sistemski poziv open()). Druge bibliotečke funkcije, kao na primer strcpy(), uopšte i ne koriste kernel. Kada neka aplikacija izvršava sistemski poziv, kaže se da kernel radi u ime aplikacije, odnosno da aplikacija izvršava sistemski poziv u kernel prostoru i da kernel radi u kontekstu procesa. Dakle, osnovni način na koji aplikacije rade je sistemski poziv u kernelu.



Slika 9.1: Veza između aplikacija, kornela i hardvera.

Kernel upravlja i sistemskim hardverom. Skoro sve arhitekture, uključujući i sisteme koje Linux podržava, obezbeđuju rad sa prekidima (interrupt). Kada hardver želi da komunicira sa sistemom, on generiše prekid koji asinhrono prekida kernel. Prekidi se identifikuju po broju. Kernel koristi broj da bi izvršio specifičnu prekidnu rutinu procesa (interrupt handler) i odgovorio na njega. Na primer, kada se pritisne taster na tastaturi, kontroler tastature generiše prekid da bi stavio do znanja sistemu da postoji novi podatak u baferu tastature. Kernel beleži koji je broj generisanog prekida i izvršava odgovarajuću prekidnu rutinu. Ona sa svoje strane obraduje podatak sa tastature i oslobađa kontroler tastature za obradu novih podataka. Da bi se obezbedila sinhronizacija, kernel može i da onemogući prekide. U mnogim operativnim sistemima, uključujući i Linux, prekidne rutine ne rade u kontekstu procesa, već u specijalnom prekidnom kontekstu koji nije pridružen nijednom procesu. Ovaj specijalni kontekst postoji isključivo da bi prekidna rutina brzo odgovorila na prekid. U Linuxu se može generalizovati da svaki procesor u datom trenutku radi nešto od sledećeg:

- U prostoru kernela, u kontekstu procesa, izvršava određeni proces
- U prostoru kernela, u kontekstu prekida, rukuje prekidom bez pridruživanja procesu.
- U korisničkom prostoru izvršava korisnički kôd u procesu.

#### 9.4. Monolitna jezgra i mikrokerneli

Rad jezgra operativnog sistema može da se prikaže preko dva glavna pojma projektovanja: monolitni kernel i mikrokerneli. Monolitna jezgra su se koristila do 1980. godine, a bila su implementirana kao veliki procesi koji se kompletno izvršavaju u jedinstvenom adresnom prostoru. Zbog toga su se takvi kerneli obično čuvali na disku u obliku statičkog binarnog kôda. Svi servisi kornela izvršavali su se u velikom objedinjenom adresnom prostoru kernela. Komunikacija unutar kernela u slučaju monolitnih jezgara je trivijalna zato što rade u sistemskom režimu, u istom adresnom prostoru: kernel može da pozove funkcije direktno, kao što radi aplikacija u korisničkom prostoru. Zagovornici ovog modela ističu njegovu jednostavnost i dobre karakteristike, koje su činile osnovu prvobitnih UNIX sistema.

S druge strane, mikrokerneli nisu implementirani kao prosti, veliki procesi, već je funkcionalnost kernela spuštena na nivo odvojenih procesa, koji se obično zovu serveri. Kod ovakvog načina organizacije jezgra, isključivo serveri zahtevaju rad u privilegovanom režimu, dok ostatak jezgra radi u korisničkom prostoru. Svi serveri se održavaju odvojeno i rade u različitim adresnim prostorima. Zbog toga nije moguće direktno pozvati funkciju kao u monolitnim kernelima. Komunikacija u mikrokerneima se izvodi preko prosleđivanja poruka (message passing): u sistem je ugrađen mehanizam komunikacije između procesa (interprocess communication, IPC) tako da

različiti serveri komuniciraju i traže usluge jedni od drugih slanjem poruka. Odvajanje različitih servera sprečava greške koje mogu nastati u jednom serveru kao rezultat grešaka nekog drugog servera. Takođe, modularnost sistema dozvoljava zamenu jednog servera drugim. Međutim, pošto je mehanizam komunikacije zahtevniji od običnih funkcijskih poziva, kao i zbog neophodne promene konteksta (context switch) iz prostora kernela u korisnički prostor i obrnuto, prosleđivanje poruka prouzrokuje kašnjenje i manju propusnost u poređenju sa monolitnim kernelima sa prostim pozivima funkcija.

U svim praktično implementiranim mikrokernelima većina servera (ili čak svi) su izmešteni u prostor jezgra, jer se tako ublažavaju negativni efekti čestih promena konteksta i dozvoljavaju direktni pozivi funkcija. Jezgra operativnih sistema Windows NT i Mach (na kome je zasnovan Mac OS X) su primeri mikrokernela. Poslednje verzije Windowsa NT i Mac OS X-a ne koriste mikrokernel servere u korisničkom prostoru, čime je potisнутa najvažnija ideja mikrokernela. U Linuxu je monolitni kernel koji se izvršava u jednom adresnom prostoru u potpunosti u sistemskom režimu, ali je Linux istovremeno pozajmio dosta toga i od mikrokernela: koristi modularni dizajn sa priručnim suspendovanjem procesa (preemption), podržava niti jezgra i sposoban je da dinamički učita posebne module kôda jezgra u kernel. Istovremeno, Linux nema sledeće osobine mikrokernel dizajna: svi procesi rade u sistemskom režimu sa direktnim pozivom funkcija i ne koristi prosleđivanje poruka kao metod komunikacije između procesa. Linux je modularan i sam raspoređuje procese.

## 9.5. Jezgro Linuxa u poređenju sa kernelom klasičnog UNIX-a

Zahvaljujući zajedničkom poreklu i istom interfejsu za programiranje (API), moderni UNIX kerneli dele zajedničke osobine. Uz nekoliko izuzetaka, kernel UNIX-a je tipično monolitan i statički. On postoji kao samoizvršavajuća slika koji radi u jednom adresnom prostoru. UNIX obično zahteva jedinicu za upravljanje memorijom (memory management unit, MMU) sa podrškom za straničenje; to je hardver koji omogućava sistemu da primeni zaštitu memorije i obezbedi jedinstveni adresni prostor svakom procesu.

U nastavu je predstavljena analiza karakteristika koje se razlikuju kod Linuxovog kernela i drugih UNIX varijanti:

- Linux podržava dinamičko učitavanje modula kernela. Iako je kernel Linuxa monolitan, on može dinamički da učita i vrati kôd kernela na zahtev.
- Linux podržava simetrične multiprocesore (Symmetrical Multiprocessing, SMP). Mada mnoge komercijalne varijante UNIX-a sada podržavaju SMP, to nije slučaj u tradicionalnim UNIX implementacijama.
- Linux kernel podržava raspoređivanje sa priručnim suspendovanjem (preemptive scheduling). Za razliku od tradicionalnih UNIX varijanti, Linux kernel je sposoban

ban da prekine izvršenje zadatka čak i onda kada se on izvršava u kernelu. Od drugih komercijalnih Unix implementacija, Solaris i IRIX imaju kernel sa prinudnom suspenzijom procesa, ali većina tradicionalnih UNIX kernela nije takva.

- Linux ne razlikuje niti i standardne procese. Iz ugla kernela svi procesi su isti; neki samo dele resurse.
- Linux obezbeđuje objektno orijentisan model uređaja sa klasama uređaja, događajima i sistemom datoteka uređaja korisničkog prostora (sysfs).
- Iz Linuxa su izostavljene neke loše projektovane UNIX karakteristike, npr. tokovi (stream) i loši standardi.

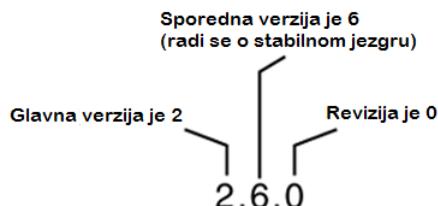
## 9.6. Verzije jezgra Linuxa

Linux kerneli mogu biti stabilni i razvojni. Stabilni kerneli su izdanja pogodna za široku upotrebu. Nove stabilne verzije kernela izdaju se najčešće samo da bi se obezbedile ispravke grešaka i novi upravljački programi (drajveri). S druge strane, razvojni kerneli trpe značajne promene.

Verzije kernela označavaju se sa tri broja razdvojena tačkama. Prvi broj je glavno izdanje (major release), drugi broj je sporedna verzija (minor release), a treći je revizija (revision). Sporedna verzija određuje da li je kernel u stabilnoj ili razvojnoj fazi: paran broj označava stabilnu verziju, a neparan razvojnu verziju jezgra. Na primer, verzija kernela 2.6.0 označava stabilan kernel. Prva dva broja definišu "seriju kernela", u ovom slučaju to je serija kernela 2.6.

Pojavljuvale su se sledeće stabilne verzije kernela: [1.0.9] [1.2.13] [2.0.40] [2.2.26] [2.4.18] [2.4.20] [2.4.28] [2.6.10] [2.6.11]

Razvojna verzija serije 1.3 stabilizovana je na 2.0, a 2.5 na 2.6. Poslednja stabilna verzija otvorenog koda Linuxovog kernela je 3.0. Treća iteracija nazvana 3.0.0-rc1 pojavila se 15 godina nakon verzije 2.0.



Slika 9.2: Oznake verzija Linuxovog jezgra.

## 9.7. Početak rada sa kernelom

Izvorni kôd Linuxa se može dobiti u obliku standardne distribucije izvornog kôda (tarball) i dodatnih "zakrpa" (patch) sa zvaničnog sajta za Linuxov kernel, <http://www.kernel.org> (Slika 9.3).

### 9.7.1. Instaliranje izvornog kôda kernela

Izvorni kôd kernela se distribuira ili u GNU formatu zip (gzip) ili češće u formatu bzip2. Kôd kernela u formatu bzip2 nalazi se u datoteci `linux-x.y.z.tar.bz2`, gde su `x.y.z` verzije izvornog kôda. Ako je izvorni kôd komprimovan u bzip2 arhivu, u komandnoj liniji treba otkucati

```
$ tar xvjf linux-x.y.z.tar.bz2  
Ako je izvorni kod komprimovan u zip arhivu, treba otkucati  
$ tar xvzf linux-x.y.z.tar.gz  
Rezultat raspakivanja biće direktorijum linux-x.y.z.
```

### 9.7.2. Korišćenje zakrpa

Distribucija izvornog kôda i novih verzija ostvaruje se zakrpama (patch). Da bi se primenile dodatne zakrpe (incremental patch) unutar stabla izvornog kernela, treba otkucati

```
$ patch p1 < ./patch-x.y.z
```

### 9.7.3. Stablo izvornog kôda kernela

Stablo izvornog kôda podeljeno je na direktorijume i poddirektorijume koji su naborjani i opisani počev od korenskog (root) direktorijuma u sledećoj Tabeli 9.1.



Slika 9.3: Web strana za preuzimanje jezgra Linuxa.

Tabela 9.1: Stablo izvornog kôda jezgra Linuxa

Direktorijum	Opis
arch	Izvor za specifične arhitekture
crypto	Interfejs za šifrovanje (CryptoAPI)
documentation	Dokumentacija za kernel
drivers	Upravljački programi uređaja
fs	Virtuelni i pojedinačni sistemi datoteka
include	Zaglavlja kernela
init	Podizanje i inicijalizacija kernela
ipc	Kôd za komunikaciju između procesa
kernel	Podsistemi jezgra, kao što je rasporedivač
lib	Pomoćne rutine
mm	Podsistemi upravljanja memorijom i virtuelnom memorijom
net	Mrežni podsistem
scripts	Skriptovi za kompajliranje (build) kernela

Tabela 9.1: Stablo izvornog kôda jezgra Linuxa	
security	Modul bezbednosti Linuxa
sound	Podsistem za zvuk
usr	Rani kôd korisničkog prostora (nazvan initramfs)

#### 9.7.4. Kompajliranje i podešavanje kernela

Kernel nudi više alata za konfigurisanje. Najprostije je u komandnoj liniji uneti komandu:

```
$ make config
```

Ova naredba prikazuje sve opcije i od korisnika traži da interaktivno bira yes, no ili module. Pošto to dugo traje, može da se pokrene i komanda:

```
$ make menuconfig
```

ili za X11 grafiku:

```
$ make xconfig
```

za gtk+ grafiku:

```
$ make gconfig
```

Komanda:

```
$ make defconfig
```

podešava standardnu konfiguraciju za određenu arhitekturu. Konfiguracione opcije nalaze se u datoteci .config korena stabla izvornog kôda kernela. Veoma često se ta datoteka direktno menja, a nakon toga konfiguracija se ažurira naredbom:

```
$ make oldconfig
```

Ovo bi trebalo da se uradi pre kompajliranja (building) kernela. Nakon podešavanja konfiguracije jezgra, ono treba da se kompajlira naredbom:

```
$ make
```

#### 9.7.5. Instalacija kernela

Nakog kompajliranja, kernel treba da se instalira. Način instaliranja zavisi od arhitekture sistema i modula za podizanje (boot loader), pa u odgovarajućoj dokumentaciji treba potražiti uputstva gde treba kopirati sliku kernela i kako je treba podesiti za podizanje. Na primer, za x86 arhitekturu treba kopirati arch/i386/boot/bzImage u /boot, dodeliti mu ime vmlinuz-version i promeniti sadržaj datoteke /boot/grub/grub.conf gde treba navesti novi kernel. U sistemu koji koristi modul za podizanje LILO treba promeniti datoteku /etc/lilo.conf. Instaliranje modula kernela je automatsko i nezavisno od arhitekture, a postiže se komandom

```
$ make modules_install
```

pri čemu se svi instalirani kompajlirani moduli smeštaju u direktorijum /lib.

Treba imati u vidu da se kernel prilično razlikuje od standardnih aplikacija korišćkog prostora:

- nema pristup C biblioteci
- napisan je u GNU C-u
- nema zaštitu memorije kao korisnički prostor
- ne može lako da radi sa promenljivama u pokretnom zarezu (floating point)
- ima mali stek fiksne veličine
- pošto koristi asimetrične prekide, prinudno preuzimanje procesa i podržava SMP, sinhronizacija i konkurentnost su najvažniji za kernel
- važno je da kernel bude prenosiv, tj. da radi na raznim platformama.

## Glava 10

# Upravljanje procesima u Linuxu

Proces je, uz datoteku, jedna od osnovnih apstrakcija u operativnom sistemu. Proces je program (objektni kôd uskladišten na nekom medijumu) u izvršenju. Međutim, procesi se ne mogu svesti samo na izvršavanje programskog kôda (koji se u UNIX-u često zove i sekcija kôda). Oni uključuju i skup resursa kao što su otvorene datoteke i signali, interni podaci jezgra, stanje procesa, adresni prostor, jedna ili više niti u izvršenju i sekcija podataka u kome se nalaze globalne promenljive. Procesi su zapravo sve ono što je rezultat izvršavanja programskog kôda. Niti izvršenja, ili samo niti, su objekti aktivnosti unutar procesa. Svaka nit uključuje jedinstveni programski brojač, stek procesa i skup registara procesora. Kernel raspoređuje pojedinačne niti, a ne procese. U tradicionalnim UNIX sistemima, svaki proces se sastoji od jedne niti. U modernim sistemima, međutim, programi sa više niti su uobičajeni.

Linux nitima prilazi na specifičan način, jer ne pravi razliku između niti i procesa. U Linuxu je nit samo specijalna vrsta procesa. U modernim operativnim sistemima, procesi obezbeđuju dve vrste virtualizacije procesora i memorije. Virtuelni procesor "zavarava" proces da ima monopol nad sistemom, iako je procesor zapravo podeđen između velikog broja drugih procesa. Postupak "zavaravanja" spada u kategoriju raspoređivanja procesa. Virtuelna memorija dozvoljava dodelu memorije procesu. Treba obratiti pažnju da niti dele apstraktну virtuelnu memoriju, ali svaka nit ima sopstveni, "virtuelizovan" procesor. Sâm program nije proces; proces je aktivan program sa povezanim resursima. Dva ili više procesa mogu da postoje i da izvršavaju isti program. U stvari, dva ili više procesa mogu da postoje i da dele različite resurse, kao što su otvorene datoteke ili adresni prostor.

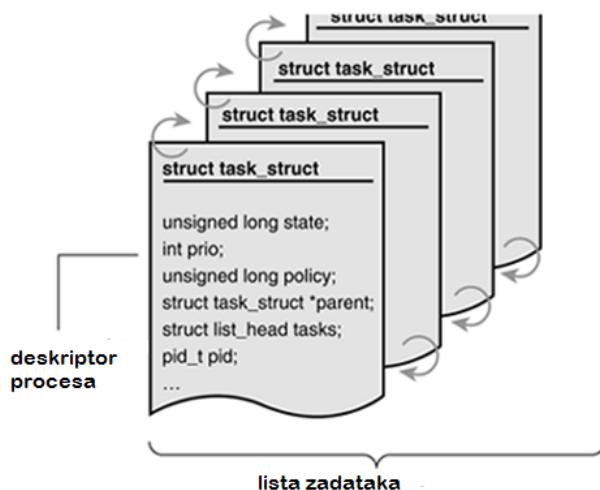
Ime koje se u Linuxu koristi za proces je **zadatak** (task). U Linuxu se proces pravi sistemskim pozivom `fork()` kojim se kreira novi proces tako što se duplira postojeći. Proses koji poziva `fork()` je proces roditelj, dok je kreiran proces dete. Proses roditelj nastavlja da se izvršava, a proces dete počinje izvršavanje na istom mestu gde se sistemski poziv vraća. Sistemski poziv `fork()` se vraća iz kernela dva puta: jednom u proces roditelj i ponovo u nov proces dete.

Za kreiranje novog adresnog prostora i učitavanje novog programa u njega koristi se porodica funkcijskih programa `exec*`(). U jezgru Linuxa, sistemski poziv `fork()` je implementiran preko sistemskog poziva `clone()`, o čemu će biti više reči u sle-

dećem odeljku. Program se završava sistemskim pozivom `exit()`. Ova funkcija završava proces i oslobađa sve njegove resurse. Proces roditelj može da sazna status završenog procesa deteta preko sistemskog poziva `wait4()`, koji omogućuje procesu da čeka završetak određenog procesa. Kada se proces završi, šalje se u specijalno "zombi" stanje koje se koristi za predstavljanje završenih procesa sve dok proces roditelj ne pozove `wait()` ili `waitpid()`. Linux sistemi preko C biblioteke obično obezbeđuju funkcije `wait()`, `waitpid()`, `wait3()` i `wait4()`. Sve ove funkcije vraćaju status završenog procesa sa nešto drugačijom semantikom.

### 10.1. Deskriptor procesa

Kernel uvezuje procese u kružnu dvostruko ulančanu listu zadataka (task). Elementi u listi zadataka su deskriptori procesa tipa C strukture `task_struct`, definisane u datoteci `<linux/sched.h>`. Deskriptor procesa sadrži sve informacije o određenom procesu (to je zapravo blok za kontrolu procesa, PCB). Struktura `task_struct` je relativno velika struktura podataka: zauzima približno 1,7 KB za 32-bitne mašine, a sadrži sve informacije koje jezgro treba da ima o procesu: otvorene datoteke, adresni prostor procesa, stanja procesa itd. (Slika 10.1)



Slika 10.1: Deskriptor procesa i lista zadataka.

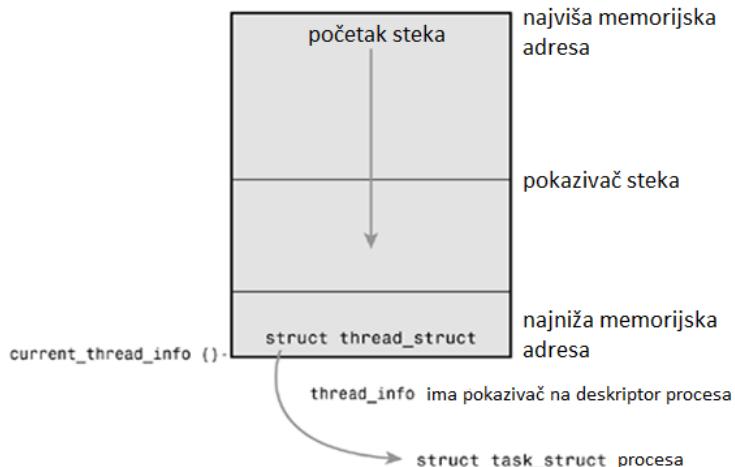
Struktura zadataka (`thread_info`) definisana je za arhitekture x86 u zaglavljiju `<asm/thread_info.h>`:

```

struct thread_info {
    struct task_struct *task;
    struct exec_domain *exec_domain;
    unsigned long flags;
    unsigned long status;
    __u32 cpu;
    __s32 preempt_count;
    mm_segment_t addr_limit;
    struct restart_block restart_block;
    unsigned long previous_esp;
    __u8 supervisor_stack[0];
};

```

Svaka struktura zadatka `thread_info` alocirana je na kraju steka zadatka. Element strukture `*task` je pokazivač na stvarnu strukturu `task_struct`, što je prikazano na Slici 10.2.



Slika 10.2: Deskriptor i stek zadatka.

### 10.1.1. Skladištenje deskriptora procesa

Sistem identificuje proces pomoću jedinstvenog identifikacionog broja (PID). PID je brojna vrednost predstavljena tipom `pid_t` (što je tipično `int`). Kernel čuva ovu vrednost kao `pid` unutar svakog deskriptora procesa (strukture `task_struct`). Administrator operativnog sistema može ovu vrednost da poveća izmenom datoteke `/proc/sys/kernel/pid_max`. Unutar kernela zadaci se obično referenciraju direktno pokazivačem na njihovu strukturu `task_struct`. Kernel kôd koji se bavi procesima radi direktno sa strukturu `task_struct`. Zbog toga je veoma korisna mogućnost

da se brzo pronađe deskriptor procesa zadatka koji se izvršava, što omogućuje makro `current`. Ovaj makro se posebno implementira za svaku arhitekturu.

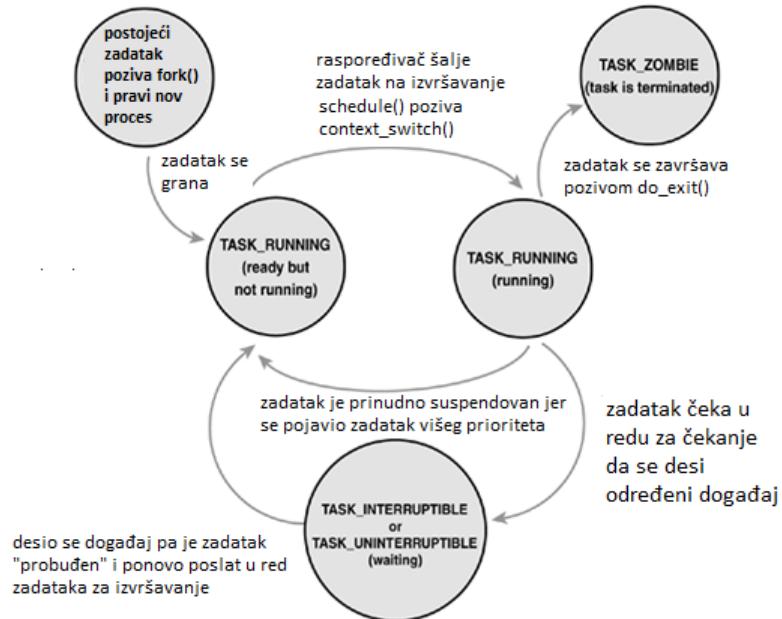
## 10.2. Stanja procesa u Linuxu

Polje deskriptora opisuje trenutno stanje procesa. Svaki proces u sistemu je u jednom od pet poznatih stanja, pa je stanje predstavljeno jednim od pet flegova:

- `TASK_RUNNING`: proces se može izvršiti; proces se ili trenutno izvršava ili je u redu procesa spremnih za izvršenje. Ovo je jedino moguće stanje za proces koji se izvršava u korisničkom prostoru; može se takođe primeniti i za proces koji se aktivno izvršava u prostoru jezgra.
- `TASK_INTERRUPTIBLE`: proces "spava" (blokiran je), čekajući da se ispuní neki uslov. Kada je uslov ispunjen, kernel postavlja stanje procesa na `TASK_RUNNING`. Proces se takođe "budi" prerano i postaje kandidat za izvršavanje ako primi signal.
- `TASK_UNINTERRUPTIBLE`: ovo stanje je identično stanju `TASK_INTERRUPTIBLE`, osim što se proces ne "budi" i ne postaje kandidat za izvršavanje kada primi signal. Ovo stanje se koristi u situacijama kada proces mora da čeka bez prekida ili kada se događaj očekuje veoma brzo.
- `TASK_ZOMBIE`: zadatak je završen, ali njegov proces roditelj još nije izdao sistemski poziv `wait4()`. Deskriptor procesa mora da se sačuva za slučaj da proces roditelj želi da mu pristupi. Kada proces roditelj pozove `wait4()`, deskriptor procesa se dealocira.
- `TASK_STOPPED`: Izvršavanje procesa se zaustavlja; zadatak se ne izvršava niti se bira za izvršavanje. Zadatak je u ovom stanju ako primi neki od signala `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, ili `SIGTTOU` (ili ako primi bilo kakav signal tokom debagovanja).

## 10.3. Kontekst procesa

Tokom izvršavanja programskog kôda u procesu, kôd se čita iz datoteke koja se izvršava unutar adresnog prostora programa. Normalno izvršavanje programa se ostvaruje u korisničkom prostoru. Kada program izvršava sistemski poziv ili generiše izuzetak, on ulazi u prostor jezgra i definiše se kontekst procesa. Nakon napuštanja kernele, proces nastavlja izvršenje u korisničkom prostoru sve dok proces višeg prioriteta ne postane kandidat za izvršavanje. Tada se poziva rasporedivač procesa (scheduler) koji šalje proces višeg prioriteta na izvršavanje.



Slika 10.3: Dijagram tokova stanja procesa u Linuxu.

#### 10.4. Copy-on-Write

Tradicionalno, kada se izvršava sistemski poziv `fork()`, svi resursi koje poseduje proces roditelj se dupliraju, a kopija resursa se predaje procesu detetu. Ovaj postupak je neefikasan. Zbog toga je u Linuxu sistemski poziv `fork()` implementiran kroz korišćenje copy-on-write stranica. Copy-on-write (ili COW) je tehnika odlaganja kopiranja podataka. Umesto dupliranja adresnog prostora procesa, proces roditelj i proces dete mogu da dele istu kopiju adresnog prostora. Podaci se, međutim, obeležavaju na takav način da ako se upisuju, pravi se duplikat i tada svaki proces prima jedinstvenu kopiju podataka. Dakle, duplikati resursa prave se samo u slučaju upisivanja podataka. Sve dok se podaci samo učitavaju, oni se dele, tj. ne pravi se posebna kopija za proces dete. Ova tehnika odlaže kopiranje svake strane u adresni prostor sve dok se ne zatraži upisivanje. U slučaju da se nikada ne upisuje, na primer, ako se `exec()` pozove odmah nakon sistemskog poziva `fork()`, nema nikakve potrebe za kopiranjem. Jedini opšti trošak koji se pojavljuje izvršenjem sistemskog poziva `fork()` je dupliranje tabele strana procesa roditelja i kreiranje jedinstvenog deskriptora za proces dete. Ovo je važna optimizacija u Linuxu u poređenju sa UNIX filozofijom koja ne podržava brzo izvršavanje procesa.

## 10.5. Sistemski poziv fork()

Linux implementira `fork()` preko sistemskog poziva `clone()`. Bibliotečki pozivi `fork()`, `vfork()` i `_clone()` pozivaju sistemski poziv `clone()` sa neophodnim flegovima. Sistemski poziv `clone()` poziva `do_fork()`, koji je definisan u datoteci `kernel/fork.c`. Ova funkcija poziva `copy_process()` i tada počinje izvršavanje procesa. Sistemski poziv `vfork()` ima isti efekat kao `fork()`, osim što se ne kopira tabela upisa strana procesa roditelja. Umesto toga, proces dete se izvršava kao samostalna nit u adresnom prostoru roditelja, a roditelj je blokiran sve dok proces dete ne pozove `exec()` ili se ne završi.

## 10.6. Niti u Linuxu

Niti su popularna programska apstrakcija koja obezbeđuje više tokova izvršavanja unutar istog programa u deljenom memorijskom adresnom prostoru. Niti mogu da dele i otvorene datoteke, kao i druge resurse. U sistemima sa jednim procesorom niti dozvoljavaju konkurentno programiranje, dok u multiprocesorskim sistemima omogućuju pravi paralelizam.

Linux ima jedinstvenu implementaciju niti: kernel uopšte ne poznaje pojам niti, već ih implementira kao standardne procese. Linux kernel ne obezbeđuje nikakve specijalne semantike raspoređivanja niti strukture podataka za predstavljanje niti. Umesto toga, nit je samo proces koji deli određene resurse sa drugim procesima. Svaka nit ima jedinstvenu strukturu `task_struct`. Ovaj pristup se razlikuje od pristupa u drugim operativnim sistemima kao što su Microsoftov Windows ili Sunov Solaris, u kojima kernel eksplicitno podržava rad sa nitima (koje se ponekad zovu i laki procesi).

U Linuxu, niti se kreiraju kao standardni zadaci, sa izuzetkom da sistemski poziv `clone()` postavlja flegove prema specifičnim resursima koji se dele:

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

Prehodni kôd daje isti rezultat kao i običan `fork()`, osim što su adresni prostor, resursi sistema datoteka, deskriptori datoteka i rukovaoci signalima deljeni. Suprotno, običan `fork()` može da se implementira kao:

```
clone(SIGCHLD, 0);
```

Sistemski poziv `vfork()` je implementiran kao:

```
clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0);
```

Sledeća tabela prikazuje flegove i odgovarajuće opise sistemskog poziva `clone()`, koji su definisani u zaglavju `<linux/sched.h>`.

Tabela 10.1: Flegovi funkcije <code>clone()</code>	
Fleg	Značenje
<code>CLONE_FILES</code>	Roditelj i dete dele otvorene datoteke.
<code>CLONE_FS</code>	Roditelj i dete dele informacije o sistemu datoteka.
<code>CLONE_IDLETASK</code>	Postavlja PID na 0 (koristi se samo za zadatke koji se ne izvršavaju).
<code>CLONE_NEWNS</code>	Kreiranje novog prostora imena za proces dete.
<code>CLONE_PARENT</code>	Dete treba da ima istog roditelja kao njegov roditelj.
<code>CLONE_PTRACE</code>	Nastavlja praćenje deteta.
<code>CLONE_SETTID</code>	Upisuje TID u korisnički prostor.
<code>CLONE_SETTLS</code>	Pravi novi TLS (thread-local storage) za dete.
<code>CLONE_SIGHAND</code>	Roditelj i dete dele rukovaće signalima i blokirane signale.
<code>CLONE_SYSVSEM</code>	Roditelj i dete dele System V SEM_UNDO semantiku.
<code>CLONE_THREAD</code>	Roditelj i dete su u istoj grupi niti.
<code>CLONE_VFORK</code>	Korišćen je vfork() i roditelj će spavati dok ga dete ne probudi.
<code>CLONE_UNTRACED</code>	Ne dozvoliti da proces praćenja primora CLONE_PTRACE za dete.
<code>CLONE_STOP</code>	Proces treba da započne u stanju TASK_STOPPED.
<code>CLONE_SETTLS</code>	Pravi nov TLS za dete.
<code>CLONE_CHILD_CLEARTID</code>	Briše TID u detetu.
<code>CLONE_CHILD_SETTID</code>	Postavlja TID u detetu.
<code>CLONE_PARENT_SETTID</code>	Postavlja TID u roditelju.
<code>CLONE_VM</code>	Roditelj i dete dele adresni prostor.

### 10.6.1. Niti jezgra

Veoma često kernel izvršava neke operacije u pozadini, koristeći niti jezgra. To su standardni procesi koji postoje samo u prostoru kernela. Bitna razlika između niti jezgra i običnih procesa je u tome što niti jezgra nemaju adresni prostor (tj. njihov pokazivač ima vrednost NULL). One rade samo u prostoru kernela i ne menjaju kon-

tekst u korisnički prostor. Niti jezgra se mogu raspoređivati i prinudno suspendovati kao svaki običan proces.

## 10.7. Završetak procesa

Kada se proces završi, kernel oslobađa resurse koje koristi proces i obaveštava proces roditelj o završetku. Uništavanje procesa obično se dešava kada proces pozove `exit()`, eksplicitno kada je spreman za završetak ili implicitno na povratku iz glavnog potprograma bilo kog programa. Proces se može završiti i prinudno, ako primi signal ili izuzetak koji ne može da obradi ili ignoriše. Bez obzira kako se proces završava, posao završetka obavlja funkcija `do_exit()`, definisana u datoteci `kernel/exit.c`.

### 10.7.1. Uklanjanje deskriptora procesa

Nakon izvršenja funkcije `do_exit()`, deskriptor završenog procesa još uvek postoji, ali je proces u "zombi" stanju i ne može da se izvršava. To dozvoljava sistemu da dobije informacije od procesa deteta nakon njegovog završetka. Porodica funkcija `wait()` implementirana je preko jednog (vrlo složenog) sistemskog poziva `wait4()`. Standardno ponašanje je suspenzija izvršavanja pozvanog zadatka sve dok postoji barem jedan njegov proces dete, kada funkcija vraća PID procesa deteta koji se završava.

## 10.8. Raspoređivanje procesa u Linuxu

Raspoređivač procesa (scheduler) je modul kernela koji bira sledeći proces za izvršavanje. Raspoređivač deli konačne resurse procesorskog vremena između procesa u sistemu spremnih za izvršavanje. Raspoređivač je osnova multitasking operativnog sistema kao što je Linux. Odlukom koji proces može da počne izvršavanje, raspoređivač direktno utiče na najbolje iskorišćenje sistema i stvara privid da se više procesa izvršava istovremeno.

Multitasking operativni sistem istovremeno održava izvršavanje više od jednog procesa. Na jednoprocесorskoj mašini to stvara privid da se više procesa izvršava konkurentno. Na multiprocesorskoj mašini to omogućava da se procesi stvarno izvršavaju konkurentno, tj. paralelno na različitim procesorima. Postoji i veliki broj procesa koji se "izvršavaju" u pozadini, odnosno kernel ih blokira dok se ne desi neki događaj (ulaz sa tastature, podaci sa mreže itd.). Savremeni Linux sistemi mogu da imaju i po stotinak procesa u memoriji, ali samo jedan može biti u stanju izvršavanja.

Kao i sve UNIX varijante, i Linux primenjuje raspoređivanje sa prinudnom suspenzijom procesa (preemptive multitasking). U takvom načinu raspoređivanja, raspoređivač odlučuje kada proces prestaje da se izvršava, a novi proces nastavlja izvršavanje. Procesorsko vreme tokom koga se proces izvršava pre nego što se suspenduje

je unapred definisano i zove se **vremenski odsečak** (timeslice). Upravljanje vremenim odsečcima omogućuje raspoređivaču globalno raspoređivanje u sistemu, a procese sprečava da preuzmu monopol nad procesorom.

Većina operativnih sistema projektovanih u poslednjoj deceniji podržava raspoređivanje sa prinudnom suspenzijom procesa (UNIX ga podržava od početka). U Linuxu je počev od razvojne serije kernela 2.6 definisan novi raspoređivač, nazvan O(1) koji ćemo u nastavku detaljno proučiti.

### 10.8.1. U/I-ograničeni i procesorski ograničeni procesi

Kada procesi troše najviše vremena na U/I zahteve, kaže se da su U/I ograničeni. Postoje i procesi koji provode dosta vremena izvršavajući kôd; za njih kažemo da su procesorski ograničeni. Ova klasifikacija nije međusobno isključiva. Politika raspoređivanja u sistemu mora da pokuša da zadovolji dva suprotstavljenia cilja: kratko vreme odziva procesa (malo kašnjenje) i maksimalno iskorišćenje sistema (visoka propuštnost). Da bi se zadovoljili ovi uslovi, raspoređivači često koriste složene algoritme da bi na osnovu zadatih kriterijuma odredili procese koji se izvršavaju. Politika raspoređivanja kod Linuxa optimizuje odziv procesa, tj. teži da ostvari minimalno kašnjenje favorizovanjem U/I ograničenih procesa u odnosu na procesorski ograničene procese.

### 10.8.2. Prioritet procesa

Najprostiji tip algoritma raspoređivanja je raspoređivanje po prioritetu. Procesi se rangiraju na osnovu dodeljenog prioriteta i procesorskog vremena koje im je potrebno za izvršenje. Procesi sa većim prioritetom se prvo izvršavaju. Kada procesi imaju isti prioritet, koristi se kružno raspoređivanje (round-robin). U nekim sistemima procesi sa većim prioritetom dobijaju duži vremenski odsečak procesora. Linux koristi dinamičko raspoređivanje zasnovano na prioritetu. Linux kernel primenjuje dva odvojena opsega prioriteta. Prvi je vrednost `nice`, broj između -20 i +19 sa podrazumevanom (default) vrednošću 0. Veća vrednost `nice` odgovara nižem prioritetu (jer to znači da proces ima dobar (`nice`) odnos prema drugim procesima u sistemu). Proses sa `nice` vrednošću od -20 dobija maksimalni odsečak vremena. Drugi opseg je prioritet u realnom vremenu. Vrednosti prioriteta mogu da se podešavaju, ali je podrazumevani opseg od 0 do 99. Svi procesi u realnom vremenu imaju veći prioritet od običnih procesa. Linux primenjuje prioritete u realnom vremenu prema POSIX standardima.

Linuxov raspoređivač je definisan u datoteci `kernel/sched.c`. U nastavku je dat deo kôda kernela v3.0 koji se odnosi na vrednosti `nice`:

```

1397/*
1398 * Nice levels are multiplicative, with a gentle 10% change for every
1399 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
1400 * nice 1, it will get ~10% less CPU time than another CPU-bound task
1401 * that remained on nice 0.
1402 *
1403 * The "10% effect" is relative and cumulative: from _any_ nice level,
1404 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
1405 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
1406 * If a task goes up by ~10% and another task goes down by ~10% then
1407 * the relative distance between them is ~25%.)
1408 */
1409 static const int prio_to_weight[40] = {
1410 /* -20 */ 88761, 71755, 56483, 46273, 36291,
1411 /* -15 */ 29154, 23254, 18705, 14949, 11916,
1412 /* -10 */ 9548, 7620, 6100, 4904, 3906,
1413 /* -5 */ 3121, 2501, 1991, 1586, 1277,
1414 /* 0 */ 1024, 820, 655, 526, 423,
1415 /* 5 */ 335, 272, 215, 172, 137,
1416 /* 10 */ 110, 87, 70, 56, 45,
1417 /* 15 */ 36, 29, 23, 18, 15,
1418};

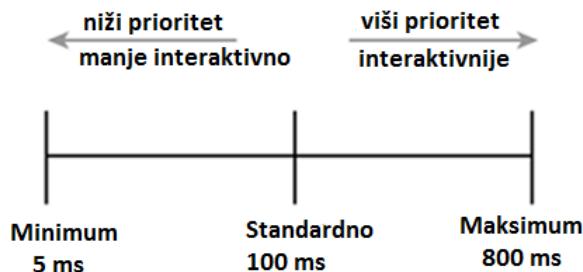
```

### 10.8.3. Vremenski odsečak

Vremenski odsečak (timeslice) je numerička vrednost koja definiše koliko dugo zadatak može da se izvršava pre nego što bude prinudno suspendovan. Vremenski odsečak se ponekad naziva i kvantum ili procesorski odsečak. Određivanje podrazumevanog trajanja vremenskog odsečka nije trivijalan zadatak. Velika vrednost daje slabe interaktivne performanse sistema, pa nije pogodna za aplikacije koje se konkurentno izvršavaju. Mala vrednost prouzrokuje gubljenje mnogo procesorskog vremena na opšte troškove prebacivanja rada sa jednog procesa na drugi. Ponovo se pojavljuju suprotstavljeni ciljevi U/I-ograničenih procesa i procesorski ograničenih procesa: za razliku od U/I-ograničenih procesa koji ne zahtevaju duže vremenske odsečke, procesorski ograničeni procesi traže duže odsečke. Linux koristi činjenicu da procesi sa najvećim prioritetom uvek započinju izvršavanje prvi. Rasporedivač povećava prioritet interaktivnih zadataka omogućavajući im da se češće izvršavaju. Linuxov rasporedivač definiše relativno veliki podrazumevani vremenski odsečak (kvantum), odnosno dinamički određuje vrednost kvantuma procesa na osnovu prioriteta. Primena dinamičkih kvantuma i prioriteta omogućuje robusno raspoređivanje (Slika 10.4).

Treba obratiti pažnju da proces ne mora da iskoristi ceo svoj kvantum odjednom. Kada "istekne" kvantum, proces dalje nije kandidat za izvršavanje sve dok svi ostali procesi ne "potroše" svoje kvantume. U tom trenutku kvantumi svih procesa se ponovo izračunavaju.

Linux primenjuje prinudnu suspenziju procesa (process preemption). Kada proces uđe u stanje TASK\_RUNNING, kernel proverava da li je njegov prioritet veći od prioriteta procesa koji se u tom trenutku izvršava. Ako jeste, poziva se raspoređivač da prisilno suspenduje, odnosno prekine izvršavanje procesa koji se izvršava u tom trenutku i počne izvršavanje novog procesa. Takođe, kada kvantum procesa dostigne vrednost 0, proces se prekida i raspoređivač se poziva da izabere novi proces.



Slika 10.4: Izračunavanje vremenskog odsečka (kvantuma) procesa.

Raspoređivač je projektovan tako da ostvari sledeće specifične ciljeve:

- Potpuna primena O(1) raspoređivanja. Svaki algoritam u novom raspoređivaču se izvršava za konstantno vreme bez obzira koliki je broj procesa koji se izvršavaju.
- Svaki procesor ima svoj sopstveni red čekanja (run queue).
- Obezbeđivanje dobrih karakteristika interaktivnosti. Čak i za vreme značajnih opterećenja, sistem treba da reaguje i trenutno rasporedi interaktivne zadatke.
- Obezbeđivanje pravednog odnosa prema procesima u odnosu na kvantum.
- Optimizacija za slučaj jednog ili dva procesa koji se izvršavaju na multiprocesoru.

#### 10.8.4. Redovi za čekanje

Osnovna struktura podataka kod raspoređivača je red za čekanje, definisana u datoteci `kernel/sched.c` kao struktura `runqueue`. Red za čekanje je lista procesa koji se mogu izvršiti na datom procesoru. Za svaki procesor definiše se po jedan red. U nastavku je dat kôd strukture sa komentarima koji opisuju svako njen polje:

```
struct runqueue {
    spinlock_t lock; /* spin lock that protects this runqueue */
    unsigned long nr_running; /* number of runnable tasks */
    unsigned long nr_switches; /* context switch count */
    unsigned long expired_timestamp; /* time of last array swap */
    unsigned long nr_uninterruptible; /* uninterruptible tasks */
    unsigned long long timestamp_last_tick; /* last scheduler tick */
    struct task_struct *curr; /* currently running task */
    struct task_struct *idle; /* this processor's idle task */
```

```

struct mm_struct *prev_mm; /* mm_struct of last ran task */
struct prio_array *active; /* active priority array */
struct prio_array *expired; /* the expired priority array */
struct prio_array arrays[2]; /* the actual priority arrays */
struct task_struct *migration_thread; /* migration thread */
struct list_head migration_queue; /* migration queue*/
atomic_t nr_iowait; /* number of tasks waiting on I/O */
};


```

S obzirom da su redovi za čekanje jezgro strukture podataka u raspoređivaču, za dobijanje reda za čekanje za dati procesor ili proces koristi se grupa makroa. Makro `cpu_rq(processor)` vraća pokazivač na red za čekanje pridružen datom procesoru; makro `this_rq()` vraća red za čekanje tekućeg procesora, a makro `task_rq(task)` vraća pokazivač na red za čekanje u kome dati zadatak čeka. Pre nego što se počne sa radom sa redom za čekanje, on mora biti zaključan. Pošto je svaki red za čekanje jedinstven za određeni procesor, retko se dešava da procesor želi da zaključa redove za čekanje različitih procesora. Zaključavanje reda za čekanje zabranjuje bilo kakvu promenu u njemu sve dok se čitaju ili upisuju članovi datog reda. Najprostiji scenario zaključavanja reda za čekanje je kada želimo da zaključamo red na kome određeni zadatak radi. U tom slučaju koriste se funkcije `task_rq_lock()` i `task_rq_unlock()`:

```

struct runqueue *rq;
unsigned long flags;
rq = task_rq_lock(task, &flags);
/* manipulate the task's runqueue, rq */
task_rq_unlock(rq, &flags);


```

Alternativno, metod `this_rq_lock()` zaključava tekući red za čekanje, a metod `rq_unlock()` ga otključava:

```

struct runqueue *rq;
rq = this_rq_lock();
/* manipulate this process's current runqueue, rq */
rq_unlock(rq);


```

Da bi se izbegao potpuni zastoj (deadlock), kôd koji želi da zaključa više redova za čekanje treba to uvek da čini po istom redosledu, tj. po rastućem nizu adresa redova za čekanje. Na primer:

```

/* to lock ... */
if (rq1 == rq2)
    spinlock(&rq1->lock);
else {
    if (rq1 < rq2) {
        spin_lock(&rq1->lock);
        spin_lock(&rq2->lock);
    } else {
        spin_lock(&rq2->lock);
    }
}


```

```

        spin_lock(&rq1->lock);
    }
}

/* manipulate both runqueues ... */
/* to unlock ... */
spin_unlock(&rq1->lock);
if (rq1 != rq2)
    spin_unlock(&rq2->lock);

```

Ovi koraci se izvode automatski preko funkcija `double_rq_lock()` i `double_rq_unlock()`. Prethodni koraci postaju:

```

double_rq_lock(rq1, rq2);
/* manipulate both runqueues ... */
double_rq_unlock(rq1, rq2);

```

#### 10.8.5. Nizovi procesa sa prioritetima

Svaki red za čekanje sadrži dva niza procesa raspoređenih po prioritetu: aktivni niz i niz sa procesima čiji je kvantum istekao. Nizovi su definisani u datoteci `kernel/sched.c` kao struktura `prio_array`. Ovi nizovi su strukture podataka složenosti algoritma raspoređivanja  $O(1)$ .

```

struct prio_array {
    int nr_active; /* number of tasks in the queues */
    unsigned long bitmap[BITMAP_SIZE]; /* priority bitmap */
    struct list_head queue[MAX_PRIO]; /* priority queues */
};

```

`MAX_PRIO` je broj nivoa prioriteta u sistemu (standardna vrednost je 140). Za svaki prioritet postoji po jedna struktura `struct list_head`. `BITMAP_SIZE` je broj članova niza tipa `unsigned long` koji definišu bitmapu prioriteta sa po jednim bitom za svaki važeći nivo prioriteta. Pošto ima 140 prioriteta, a reči su 32-bitne, sledi da niz `bitmap` ima 5 elemenata (ukupno 160 bitova). Svaki niz procesa sa prioritetima takođe sadrži niz redova nazvan `queue` definisanih strukturom `struct list_head` (po jedan red za svaki prioritet). Unutar datog prioriteta, zadaci se raspoređuju po kružnom (round robin) algoritmu. Broj zadataka koji se mogu izvršiti u nizu procesa sa prioritetom je definisan celobrojnom promenljivom `nr_active`.

#### 10.8.6. Ponovno izračunavanje vremenskih kvantuma

Raspoređivač u Linuxu od verzije kernela 2.6 održava dva niza procesa sa prioritetima za svaki procesor: niz aktivnih procesa i niz procesa sa isteklim vremenskim kvantumom. Ponovno izračunavanje vremenskih kvantuma svodi se na preusmeravanje dva pokazivača na ove nizove, kao što je definisano u funkciji `schedule()`:

```

struct prio_array *array = rq->active;
if (!array->nr_active) {
    rq->active = rq->expired;
    rq->expired = array;
}

```

### 10.8.7. Funkcija schedule()

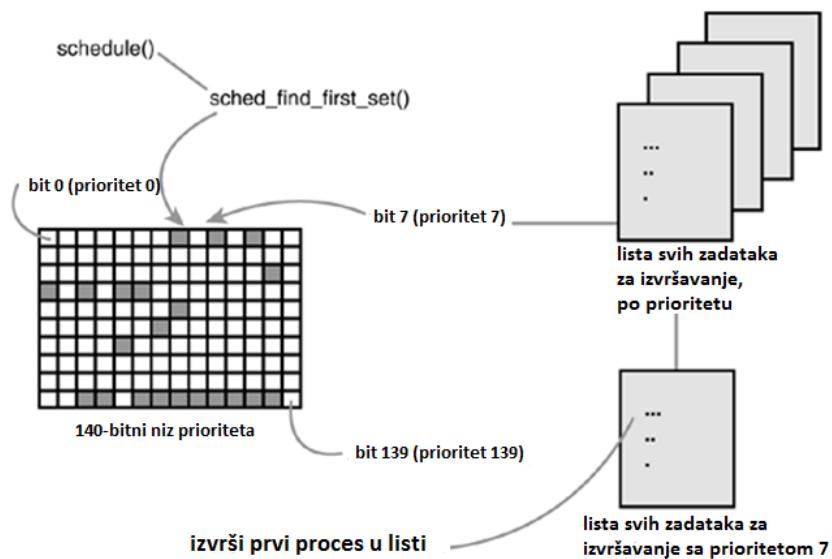
Ovu funkciju eksplisitno poziva kôd kernela da bi odredio sledeći zadatak za izvršavanje, i ona se izvršava nezavisno za svaki procesor. Sledeći kôd određuje zadatak sa najvećim prioritetom:

```

struct task_struct *prev, *next;
struct list_head *queue;
struct prio_array *array;
int idx;
prev = current;
array = rq->active;
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, struct task_struct, run_list);

```

Prvo se pretražuje niz sa aktivnim procesima sa prioritetom, da bi se pronašao prvi postavljen bit koji odgovara zadatku sa najvećim prioritetom koji se može izvršiti. Rasporedivač zatim bira prvi zadatak sa liste sa tim prioritetom (Slika 10.5).



Slika 10.5: Linuxov algoritam raspoređivanja.

Postupak je veoma jednostavan i brz. Broj procesa u sistemu ne utiče na vreme izvršenja ovog kôda, koje je konstantno.

#### 10.8.8. Određivanje prioriteta i vremenskog kvantuma

Procesi imaju početni prioritet koji se zove `nice` vrednost. Opseg vrednosti prioriteta je od -20 (najveći) do +19 (najmanji) sa podrazumevanom (default) vrednošću 0. Ova vrednost se čuva u članu `static_prio` strukture procesora `task_struct` i zove se statički prioritet. Dinamički prioritet `prio` izračunava se kao funkcija statičkog prioriteta i interaktivnosti zadatka, a dobija se kao rezultat metoda `effective_prio()`. Metod počinje sa `nice` vrednošću zadatka i dodaje ili oduzima prioritet u opsegu od -5 do +5 u zavisnosti od interaktivnosti zadatka. Raspoređivač na osnovu heuristike određuje interaktivnost procesa. Ako zadatak provodi najviše vremena stanju "spavanja", onda je on U/I ograničen, a ako troši više vremena na izvršavanje, onda sigurno nije interaktivan. Da bi primenio ovu heuristiku, Linux vodi računa o trajanju "spavanja" procesa i smešta to trajanje u član `sleep_avg` strukture `task_struct`. Opseg vrednosti trajanja je od 0 do `MAX_SLEEP_AVG`, čija je podrazumevana vrednost 10 milisekundi. Član `sleep_avg` se inkrementira dok zadatak "spava", sve dok se ne dostigne vrednost `MAX_SLEEP_AVG`. Sa svakim pokretanjem zadatka, `sleep_avg` se smanjuje dok ne dostigne vrednost nula. Zadatak koji provodi dosta vremena spavajući, ali i kontinualno trošeći svoj vremenski kvantum, neće biti nagrađen velikim dodatkom na vrednost prioriteta, jer osim što se nagrađuju interaktivni zadaci, kažnjavaju se procesorski ograničeni zadaci. Na taj način obezbeđuje se brzi odziv. Nov interaktivni proces brzo dobija veliku vrednost za `sleep_avg`. Pošto se dodavanje ili oduzimanje vrednosti prioriteta primenjuje na početnu `nice` vrednost, korisnik može da utiče na odluke raspoređivanja sistema tako što će promeniti `nice` vrednosti procesa.

Vremenski kvantum je lakše izračunati od `nice` vrednosti jer se zasniva na statičkom prioritetu. Kada se proces prvi put kreira, novi procesi deca i proces roditelj dele preostalo vreme kvantuma roditelja. Na ovaj način se sprečava da korišćenjem sistemskog poziva `fork()` procesi deca dobijaju neograničene vremenske kvantume. Funkcija `task_timeslice()` vraća nove kvantume za dati zadatak. Maksimalna vrednost kvantuma dodeljuje se zadatku s najvećim prioritetom (čija je `nice` vrednost -20) i iznosi 800 milisekundi. Zadatak sa najmanjim prioritetom (čija je `nice` vrednost +19) dobija minimalnu vrednost kvantuma `MIN_TIMESLICE` od 5 milisekundi. Zadaci sa standardnim prioritetom (čija je `nice` vrednost 0) dobijaju kvantum od 100 milisekundi.

#### 10.8.9. Balanser opterećenja

Kako se definiše globalna politika raspoređivanja u multiprocesorskim sistemima? Šta se dešava ako u jednom redu za čekanje jednog procesora ima sedam procesa, a

u drugom redu za čekanje samo jedan proces? Rešenje je tzv. **balanser opterećenja** (load balancer) koji pokušava da ravnomerno rasporedi procese u redove čekanja svih procesora. Balanser opterećenja je definisan u datoteci `kernel/sched.c` kao funkcija `load_balance()` koja ima dva dela. Kada je tekući red za čekanje prazan, poziva se funkcija `schedule()`. Ova funkcija se poziva i preko tajmera, na svaki milisekund kada sistem ništa ne radi, u suprotnom na svakih 200 milisekundi. U jednoprocesorskim sistemima `load_balance()` se nikada ne poziva. Balanser opterećenja se poziva sa tekućim redom za čekanje procesora koji je zaključan i sa onemogućenim prekidima da bi se redovi za čekanje zaštitili od konkurentnog pristupa. Kada `schedule()` poziva `load_balance()`, posao funkcije `schedule()` je lak jer je tekući red za čekanje prazan, pa je nalaženje bilo kog procesa koji će se dodati u red za čekanje jednostavno. Kada se balanser opterećenja pozove preko tajmera, njegov zadatak je da razreši loš balans između redova za čekanje.

Funkcija `load_balance()` i sa njom povezani metodi su definisani kroz sledeće korake:

1. `load_balance()` poziva `find_busiest_queue()` da bi se odredio red za čekanje koji ima najveći broj procesa (tj. najuposleniji red). Ako ne postoji red za čekanje koji ima barem za četvrtinu više procesa od tekućeg, funkcija `find_busiest_queue()` vraća NULL funkciji `load_balance()`. Ako postoji, najuposleniji red za čekanje se vraća.

2. `load_balance()` odlučuje iz kog niza prioriteta u najuposlenijem redu za čekanje funkcija želi da uzme zadatak. Niz sa isteklim vremenima je pogodniji zato što se zadaci koji su u njemu nisu izvršavali prilično dugo, pa nisu u kešu procesora. Ako je red procesa sa isteklim kvantumom prazan, jedini izbor je aktivan red. Keš memorija procesora je lokalno integrisana (na čipu) i nudi brzi pristup preko sistemske memorije. Ako se zadatak izvršava u procesoru, a podaci pridruženi zadatku dovedu u lokalni keš procesora, oni se posmatraju kao "vrući" (hot). Ako ovi podaci nisu u lokalnom kešu procesora, onda se za ovaj zadatak keš posmatra kao "hladan" (cold).

3. `load_balance()` nalazi listu zadataka sa najvećim prioritetom (najmanjom `nice` vrednošću), zato što je osim zadataka sa visokim prioritetom važno da na red za izvršavanje dođu i zadaci sa malim prioritetom (da se ne bi desio problem "izgladnjivanja").

4. svaki zadatak sa datim prioritetom se analizira da bi se našao zadatak koji se ne izvršava, koji je sprečen da se kreće prema unapred određenom procesoru i koji nije u kešu. Ako zadatak zadovoljava ove kriterijume, poziva se funkcija `pull_task()` da bi prebacila zadatak iz najuposlenijeg reda u tekući red za čekanje

5. Prethodni koraci se ponavljaju sve dok redovi za čekanje ne postanu balansirani. Kada se na kraju postignu balansirani redovi, tekući red se otključava, a funkcija `load_balance()` završava.

U nastavku je prikazan kôd za funkciju `load_balance()`:

```

static int load_balance(int this_cpu, runqueue_t *this_rq,
    struct sched_domain *sd, enum idle_type idle) {
    struct sched_group *group;
    runqueue_t *busiest;
    unsigned long imbalance;
    int nr_moved;
    spin_lock(&this_rq->lock);
    group = find_busiest_group(sd, this_cpu, &imbalance, idle);
    if (!group)
        goto out_balanced;
    busiest = find_busiest_queue(group);
    if (!busiest)
        goto out_balanced;
    nr_moved = 0;
    if (busiest->nr_running > 1) {
        double_lock_balance(this_rq, busiest);
        nr_moved = move_tasks(this_rq, this_cpu,
            busiest, imbalance, sd, idle);
        spin_unlock(&busiest->lock);
    }
    spin_unlock(&this_rq->lock);
    if (!nr_moved) {
        sd->nr_balance_failed++;
        if (unlikely(sd->nr_balance_failed > sd->cache_nice_tries+2)) {
            int wake = 0;
            spin_lock(&busiest->lock);
            if (!busiest->active_balance) {
                busiest->active_balance = 1;
                busiest->push_cpu = this_cpu; wake = 1;
            }
            spin_unlock(&busiest->lock);
            if (wake)
                wake_up_process(busiest->migration_thread);
            sd->nr_balance_failed = sd->cache_nice_tries;
        }
    } else sd->nr_balance_failed = 0;
    sd->balance_interval = sd->min_interval;
    return nr_moved;
out_balanced:
    spin_unlock(&this_rq->lock);
    if (sd->balance_interval < sd->max_interval)
        sd->balance_interval *= 2;
    return 0;
}

```

### **10.8.10. Raspoređivanje u realnom vremenu**

Linux obezbeđuje dva načina raspoređivanja u realnom vremenu, SCHED\_FIFO i SCHED\_RR, a za ostale slučajeve koristi se raspoređivanje SCHED\_NORMAL. Funkcija SCHED\_FIFO implementira FIFO algoritam raspoređivanja bez vremenskih kvantuma. Zadatak koji se može izvršiti funkcija SCHED\_FIFO uvek raspoređuje pre svakog zadataka koji raspoređuje SCHED\_NORMAL. Kada zadatak SCHED\_FIFO postane kandidat za izvršenje, on nastavlja da se izvršava dok se ne blokira ili mu se eksplisitno dodeli procesor; on nema dodeljen kvantum i može da se izvršava beskonačno. Samo zadatak većeg prioriteta SCHED\_FIFO ili SCHED\_RR može da suspenduje izvršavanje zadataka SCHED\_FIFO. Nekoliko SCHED\_FIFO zadataka sa istim prioritetom raspoređuju se kružnim (round robin) algoritmom, ali ponovo dobijaju procesor tek kada sami eksplisitno to odluče. Sve dok se zadatak SCHED\_FIFO može izvršavati, svi zadaci sa nižim prioritetom čekaju da se on završi.

## Glava 11

# Sistemski pozivi

Kernel obezbeđuje skup interfejsa pomoću kojih proces koji se izvršava u korisničkom prostoru može da komunicira sa sistemom. Ti interfejsi omogućuju aplikacijama da pristupe hardveru i drugim resursima operativnog sistema. Linux ima mnogo manje sistemskih poziva u poređenju sa drugim operativnim sistemima (oko 250 sistemskih poziva za arhitekturu x86). Pri tom, svakoj arhitekturi je dozvoljeno da definiše sopstvene sistemske pozive. S tačke gledišta programera, sistemski pozivi nisu bitni; programer vidi samo interfejs za programiranje (API). S druge strane, kernel radi samo sa sistemskim pozivima, tj. ne zanima ga kako sistemske pozive koriste bibliotečki pozivi i aplikacije.

Sistemskim pozivima (često zvanim syscalls u Linuxu) obično se pristupa preko funkcijskih poziva. Pozivi mogu da imaju jedan ili više argumenata (ulaza) i da kao rezultat daju nekoliko sporednih efekata (npr. upis u datoteku ili kopiranje podataka). Povratna vrednost sistemskih poziva je tipa `long` i može da predstavlja uspešan zavrsetak ili grešku. Negativna vrednost obično označava grešku, a nula uspešan zavrsetak. Na primer, sistemski poziv `getpid()` vraća ceo broj koji predstavlja PID tekućeg procesa. Implementacija ovog sistemskog poziva u kernelu je veoma jednostavna:

```
asmlinkage long sys_getpid(void) {
    return current->tgid;
}
```

Obratite pažnju na modifikator `asmlinkage` u definiciji funkcije. To je bit koji kaže kompjajleru da argumente funkcije traži isključivo na steku i to je zahtevani modifikator za sve sistemske pozive. Sistemski poziv `getpid()` je definisan u kernelu kao `sys_getpid()`.

### 11.1. Brojevi sistemskih poziva

Svakom sistemskom pozivu u Linuxu se pridružuje jedinstven broj `syscall`. Procesi ne referenciraju sistemski poziv po imenu, već po broju koji se ne može menjati. Kernel održava listu svih registrovanih sistemskih poziva u tabeli sistemskih poziva, koja se nalazi u datoteci `sys_call_table`. Tabela sistemskih poziva je zavisna od

arhitekture i obično definisana u datoteci `entry.S`, koja se za arhitekturu x86 nalazi u direktorijumu `arch/i386/kernel/`.

## 11.2. Upravljač sistemskih poziva

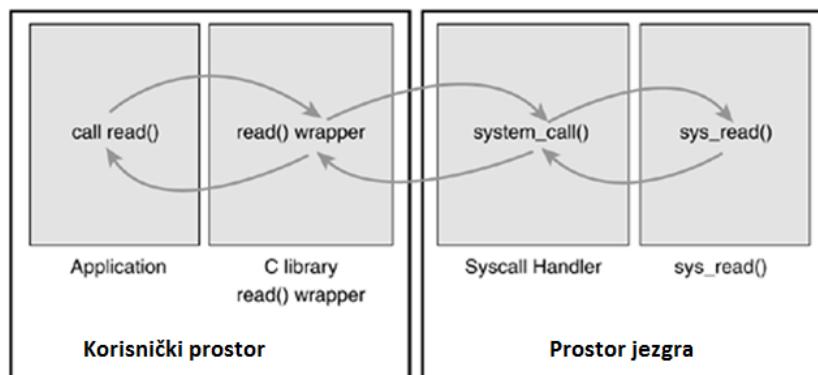
Mehanizam koji signalizira kernelu da aplikacije korisničkog prostora žele da izvrše sistemski poziv je softverski prekid (interrupt). Nakon prekida, ulazi se u sistemski režim i poziva se upravljač sistemskih poziva (system call handler). Softverski prekid za arhitekturu x86 je instrukcija `int $0x80`; njenim izvršavanjem prelazi se u sistemski režim i izvršava se vektor izuzetka (exception vector 128), tj. upravljač sistemskih poziva. Funkcija `system_call()` zavisi od arhitekture i obično je implementirana u asembleru u datoteci `entry.S`. Bez obzira kako se poziva upravljač sistemskih poziva, korisnički režim generiše izuzetak (trap) da bi se ušlo u sistemski režim (Slika 11.1).

### 11.2.1. Označavanje odgovarajućeg sistemskog poziva

Kernelu mora da se prosledi broj sistemskog poziva. U arhitekturi x86, broj sistemskog poziva se prosleđuje preko registra `eax`. Upravljač sistemskih poziva čita vrednost iz registra `eax`. Funkcija `system_call()` proverava ispravnost broja sistemskog poziva upoređujući ga sa vrednošću `NR_syscalls`. Ako je broj sistemskog poziva veći ili jednak vrednosti `NR_syscalls`, funkcija vraća `ENOSYS`. Poziv se izvodi sledećom naredbom:

```
call *sys_call_table(,%eax,4)
```

Pošto je svaki element u tabeli sistemskih poziva 32-bitni (4 bajta), kernel može broj sistemskog poziva sa četiri da bi izračunao lokaciju poziva u tabeli sistemskih poziva.



Slika 11.1: Poziv upravljača sistemskih poziva i izvršenje sistemskog poziva.

### 11.2.2. Implementacija sistemskih poziva

Dodavanje novih sistemskih poziva Linuxu je relativno lako, ali se ne preporučuje. Posao projektovanja i primene sistemskog poziva je veoma težak, dok je sama registracija u kernelu veoma jednostavna. Prvi korak u implementaciji sistemskog poziva je definisanje njegove namene; sistemski poziv može da se koristi samo za jednu funkciju. Sistemski poziv treba da ima jasan i jednostavan interfejs sa što manjim brojem argumenata. Semantika i ponašanje sistemskog poziva su važni i ne smeju se menjati zato što će za njih biti programirane aplikacije. Sistemski poziv treba da bude projektovan tako da bude opšti, robustan, prenosiv i nezavisан od arhitekture.

### 11.2.3. Provera parametara sistemskog poziva

Sistemski pozivi moraju pažljivo da provere sve svoje parametre zbog bezbednosti i stabilnosti kernela. Jedna od najvažnijih provera je ispravnost pokazivača koji stižu od korisnika. Kernel nudi dva metoda za proveravanje parametara i kopiranje u korisnički prostor. Za upis u korisnički prostor koristi se metod `copy_to_user()` koji ima tri argumenta. Prvi je odredišna memorijska adresa u adresnom prostoru procesa. Drugi je izvorni pokazivač u prostoru kernela. Treći argument je veličina podataka koji se kopiraju u bajtovima. Za čitanje iz korisničkog prostora koristi se method `copy_from_user()`. Funkcija iz drugog parametra u prvi parametar učitava broj bajtova određen trećim parametrom. Obe funkcije u slučaju greške vraćaju broj bajtova koje ne mogu da kopiraju, a u slučaju uspeha vraćaju nulu. Kao primer proučićemo sistemski poziv `silly_copy()` koji koristi oba metoda.

```
/* * silly_copy - utterly worthless syscall
 * that copies the len bytes from * 'src' to 'dst',
 * using the kernel as an intermediary in the copy
 * for no good reason. But it makes for a good example! */
asmlinkage long sys_silly_copy(unsigned long *src,
    unsigned long *dst, unsigned long len) {
    unsigned long buf;
    /* fail if the kernel wordsize and user wordsize do not match */
    if (len != sizeof(buf)) return -EINVAL;
    /* copy src, which is in the user's address space, into buf */
    if (copy_from_user(&buf, src, len)) return -EFAULT;
    /* copy buf into dst, which is in the user's address space */
    if (copy_to_user(dst, &buf, len)) return -EFAULT;
    /* return amount of data copied */
    return len;
}
```

Veoma je važna provera dozvola (ovlašćenja). Sistem proverava pristupe određenim resursima. Poziv `capable()` sa važećim flegovima vraća broj različit od nule ako su korisniku dodeljena ovlašćenja, odnosno nulu ako nisu. Na primer, `capable(CAP_SYS_NICE)`

proverava da li korisnik koji poziva funkciju ima dozvolu da promeni nice vrednost drugih procesa. Superkorisnik (superuser) standardno ima sve dozvole, a običan (non-root) korisnik ih nema. Evo još jednog primera kao ilustracije korišćenja dozvola:

```
asmlinkage long sys_am_i_popular (void) {

    /* check whether the user possesses the CAP_SYS_NICE capability */
    if (!capable(CAP_SYS_NICE))
        return EPERM;
    /* return zero for success */
    return 0;
}
```

Sledi deo kôda iz zaglavlja <linux/capability.h> za listu svih mogućnosti i odgovarajućih prava.

```
166 /**
167 ** Linux-specific capabilities
168 */
169
170 /* Without VFS support for capabilities:
171 * Transfer any capability in your permitted set to any pid,
172 * remove any capability in your permitted set from any pid
173 * With VFS support for capabilities (neither of above, but)
174 * Add any capability from current's capability bounding set
175 * to the current process' inheritable set
176 * Allow taking bits out of capability bounding set
177 * Allow modification of the securebits for a process
178 */
179
180 #define CAP_SETPCAP 8
181
182 /* Allow modification of S_IMMUTABLE and S_APPEND file attributes */
183
184 #define CAP_LINUX_IMMUTABLE 9
185
186 /* Allows binding to TCP/UDP sockets below 1024 */
187 /* Allows binding to ATM VCIs below 32 */
188
189 #define CAP_NET_BIND_SERVICE 10
```

#### **11.2.4. Završni korak u povezivanju novog sistemskog poziva sa postojećim kôdom**

Pošto je sistemski poziv napisan, on se veoma lako zvanično registruje:

- ▷ Prvo, treba dodati jedan upis na kraj tabele sistemskih poziva. To treba da se uradi za svaku arhitekturu koja podržava sistemski poziv. Pozicija u tabeli poziva počinje od nule.
- ▷ Za svaku podržanu arhitekturu broj sistemskog poziva treba da se definiše u zaglavlju `<asm/unistd.h>`.
- ▷ Sistemski poziv treba da bude kompajliran u sliku kernela (za razliku od kompajliranja modula). Sistemski poziv treba smestiti u odgovarajuću datoteku u poddirektorijumu `kernel/`, kao što je `sys.c`.

Sada ove korake da možemo da izvršimo za izmišljeni poziv `foo()`. Prvo, treba dodati `sys_foo()` u tabelu sistemskih poziva. Za većinu arhitektura tabela se nalazi u datoteci `entry.S` i izgleda ovako:

```
ENTRY(sys_call_table)
.long sys_restart_syscall /* 0 */
.long sys_exit
.long sys_fork
.long sys_read
.long sys_write
.long sys_open /* 5 */
...
.long sys_mq_unlink
.long sys_mq_timedsend
.long sys_mq_timedreceive /* 280 */
.long sys_mq_notify
.long sys_mq_getsetattr
```

Nov sistemski poziv se dodaje na kraj ove liste:

```
.long sys_foo /* 283 */
```

Broj sistemskog poziva se razlikuje za različite arhitekture. Zatim se broj sistemskog poziva dodaje u zaglavljje `<asm/unistd.h>`:

```
/* * This file contains the system call numbers. */
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5 ...
#define __NR_mq_unlink 278
#define __NR_mq_timedsend 279
#define __NR_mq_timedreceive 280
#define __NR_mq_notify 281
#define __NR_mq_getsetattr 282
```

sa poslednjim redom koji smo dodali:

```
#define __NR_foo 283
```

Na kraju se dodaje implementacija sistemskog poziva `foo()`. Sistemski poziv mora da bude kompajliran u slici kernela u svim konfiguracijama; kôd se dodaje u `kernel/sys.c`.

```
#include <asm/thread_info.h>
/* * sys_foo everyone's favorite system call.
 *
 * Returns the size of the per-process kernel stack. */
asmlinkage long sys_foo(void) {
    return THREAD_SIZE;
}
```

Kernel treba da se podigne i nakon toga korisnik može da upotrebi sistemski poziv `foo()`.

### **11.3. Pristup sistemskom pozivu iz korisničkog prostora**

C biblioteka obezbeđuje podršku za sistemske pozive. Korisničke aplikacije mogu da pročitaju prototipove funkcija iz standardnih zaglavila i povežu se sa C bibliotekom da bi koristile nove sistemske pozive (ili bibliotečke rutine koje koriste sistemske pozive). Ako se sistemski poziv samo napiše, nije sigurno da ga glibc podržava. Paket glibc sadrži standardne biblioteke koje koriste različiti programi u sistemu, na primer standardnu C biblioteku i matematičku biblioteku, bez kojih Linux sistem ne funkcioniše. Ovaj paket sadrži i lokalnu podršku za različite nacionalne jezike. Srećom, Linux obezbeđuje skup makroa za pristup sistemskim pozivima bez eksplisitne bibliotečke podrške.

## Glava 12

# Prekidi i upravljači prekida

Prekidi (interrupts) omogućuju hardveru da komunicira sa procesorom. Prekid proizvodi električni signal koji generiše hardverski uređaj, a usmeren je ka ulaznim pinovima kontrolera prekida, koji sa svoje strane šalje signal procesoru. Procesor registruje signal i prekida sa tekućim poslom da bi upravljao prekidom. Procesor može da obavesti operativni sistem da je došlo do prekida da bi ga operativni sistem obudio na odgovarajući način. Različitim uređajima se pridružuju jedinstvene celobrojne vrednosti za svaki prekid. Te vrednosti se često zovu *linije zahteva za prekidom* (interrupt request, IRQ). Kada se govori o prekidima, važno je razlikovati prekide i izuzetke (exceptions). Za razliku od prekida, izuzeci se pojavljuju sinhrono sa procesorskim taktom, pa se zato zovu i sinhroni prekidi. Izuzetke proizvodi procesor dok izvršava instrukcije, i to kao odziv na grešku u programiranju (npr. deljenje nulom) ili kao reakciju na neuobičajene uslove koje mora da razreši kernel (npr. pogrešne strane u algoritmu straničenja, page fault).

### 12.1. Upravljači prekida

Upravljač prekida (interrupt handler) je specijalna servisna rutina za obradu prekida (interrupt service routine, ISR). Svakom uređaju koji generiše prekide pridružen je jedan upravljač prekida, koji je deo upravljačkog programa (driver), tj. kernel koda koji upravlja radom uređaja. U Linuxu su upravljači prekida obične C funkcije. Upravljač prekida treba da se izvršava brzo i da uradi složen posao; ta dva cilja su međusobno suprotstavljenja. Zato je obrada prekida podeljena u dva dela. Upravljač prekida je prvi deo; on se poziva odmah nakon prijema prekida i izvršava samo vremenski kritične poslove, kao što su potvrda prijema prekida ili resetovanje hardvera. Posao koji se može izvršiti kasnije se odlaže za drugi deo.

Proučimo primer mrežne kartice. Kada mrežna kartica primi dolazne pakete sa mreže, ona treba da obavesti kernel veoma brzo, da optimizuje propusnost i kašnjenje i izbegne isticanje rokova. Zbog toga ona veoma brzo generiše prekid. Kernel odgovara izvršavajući registrovani prekid mrežne kartice. Prekid se izvršava uz potvrdu hardvera, kopiraju se novi mrežni paketi u glavnu memoriju, a mrežna kartica se pri-

prema za još paketa. Ovi poslovi su važni, vremenski kritični i obavljaju se hardverski. Ostatak obrade i upravljanja paketima izvršava se kasnije, u tzv. donjoj polovini.

### 12.1.1. Registracija upravljača prekida

Upravljački program uređaja (drayver) je odgovoran za upravljač prekida i registruje jedan prekid. Upravljački programi uređaja registruju upravljač prekida i uključuju liniju prekida za upravljanje preko sledeće funkcije:

```
/* request_irq: allocate a given interrupt line */
int request_irq(unsigned int irq,
                 irqreturn_t (*handler)(int, void *, struct pt_regs *),
                 unsigned long irqflags, const char *devname, void *dev_id)
```

Prvi parametar, `irq`, određuje broj prekida. Parametar `handler` je funkcijski pokazivač na aktuelni upravljač prekida koji obrađuje ovaj prekid. Ova funkcija se poziva kad god operativni sistem primi prekid. Treći parametar, `irqflags`, može biti ili nula ili bit maske jednog ili više flegova:

- `SA_INTERRUPT` određuje da je dati upravljač prekida brz.
- `SA_SAMPLE_RANDOM` određuje da li prekid koji generiše uređaj deterministički ili ne.
- `SA_SHIRQ` određuje da linija prekida može biti deljena između više upravljača prekida.

Četvrti parametar `devname` je ASCII tekstualna prezentacija uređaja sa datim prekidom. Peti parametar `dev_id` koristi se prvenstveno za deljene linije prekida. Bez ovog parametra bilo bi nemoguće da kernel zna koji upravljač da ukloni sa određene linije prekida.

Funkcija `request_irq()` vraća nulu ako se uspešno izvrši, a vrednost različitu od nule ako postoji greška. U upravljačkom programu, zahtevanje prekidne linije i instaliranje upravljača obavlja funkcija `request_irq()` na sledeći način:

```
if (request_irq(irqn, my_interrupt, SA_SHIRQ, "my_device", dev)) {
    printk(KERN_ERR "my_device: cannot register IRQ %d\n", irqn);
    return -EIO;
}
```

U ovom primeru `irqn` je zahtevana linija prekida, `my_interrupt` je upravljač, linija može biti deljena, ime uređaja je "`my_device`" i prosleđuje se `dev` za `dev_id`.

Kada se drayver oslobođi, treba da se oslobođi registracija upravljača prekida i potencijalno onemogući linija prekida. To se radi pozivom:

```
void free_irq(unsigned int irq, void *dev_id)
    koji se izvodi iz konteksta procesa.
```

### 12.1.2. Pisanje kôda upravljača prekida

Tipična deklaracija upravljača prekida je:

```
static irqreturn_t intr_handler(int irq, void *dev_id,
                               struct pt_regs *regs)
```

Treba obratiti pažnju da ova deklaracija odgovara prototipu argumenta `handler` funkcije `request_irq()`. Prvi parametar `irq` je numerička vrednost linije prekida upravljača. Drugi parametar `dev_id` je generički pokazivač na isti `dev_id` u funkciji `request_irq()` kada je upravljač prekida bio registrovan. Ako je ova vrednost jedinstvena (što se preporučuje zbog podrške deljenju), može da se koristi da bi se razlikovali uređaji koji potencijalno mogu da koriste isti upravljač prekida. Poslednji parametar `regs` sadrži pokazivač na strukturu koja sadrži stanje procesorskih registara pre usluživanja prekida. Funkcija vraća vrednost tipa `irqreturn_t`. Upravljač prekida je označen kao `static` zato što se nikad ne poziva direktno iz druge datoteke.

### 12.1.3. Primer upravljača prekida

Posmatrajmo realan upravljač prekida sata RTC (real-time clock) koji se može naći u datoteci `drivers/char/rtc.c`. Sistemski sat (RTC) se nalazi u mnogim mašinama, uključujući PC. To je uređaj odvojen od sistemskog tajmera, a koristi se za postavljanje sistemskog sata, alarma itd. U većini arhitektura sistemski sat se postava upisivanjem odgovarajućeg vremena u određeni registar. Svaki alarm ili periodični tajmer implementira se preko prekida. Kada se učita upravljački program za RTC, poziva se funkcija `rtc_init()` da ga inicijalizuje. Jedan od zadataka funkcije `rtc_init()` je i registracija upravljača prekida:

```
/* register rtc_interrupt on RTC_IRQ */
if (request_irq(RTC_IRQ, rtc_interrupt,
                 SA_INTERRUPT, "rtc", NULL)) {
    printk(KERN_ERR "rtc: cannot register IRQ %d\n", RTC_IRQ);
    return -EIO;
}
```

Treba obratiti pažnju da je se linija prekida čuva u parametru `RTC_IRQ`. Za PC arhitekturu, RTC se uvek nalazi na prekidnoj liniji IRQ 8. Drugi parametar je upravljač prekida `rtc_interrupt` koji radi sa svim prekidima koji su omogućeni, zahvaljujući flegu `SA_INTERRUPT`. Četvrti parametar određuje ime upravljačkog programa: "rtc". Pošto ovaj uređaj ne može da deli liniju prekida, a upravljač ne koristi neku posebnu vrednost, za `dev_id` se prosleđuje vrednost NULL. Sledi kôd za upravljač:

```

/*
 * A very tiny interrupt handler. It runs with SA_INTERRUPT set,
 * but there is a possibility of conflicting with the set_rtc_mmss()
 * call (the rtc irq and the timer irq can easily run at the same
 * time in two different CPUs). So we need to serialize
 * accesses to the chip with the rtc_lock spinlock that each
 * architecture should implement in the timer code.
 * (See ./arch/XXXX/kernel/time.c for the set_rtc_mmss() function.) */
static irqreturn_t rtc_interrupt(int irq, void *dev_id,
    struct pt_regs *regs) {
/*
 * Can be an alarm interrupt, update complete interrupt,
 * or a periodic interrupt. We store the status in the
 * low byte and the number of interrupts received since
 * the last read in the remainder of rtc_irq_data. */
spin_lock (&rtc_lock);
rtc_irq_data += 0x100;
rtc_irq_data &= ~0xff;
rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);
if (rtc_status & RTC_TIMER_ON)
    mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);
spin_unlock (&rtc_lock);
/*
 * Now do the rest of the actions
 */
spin_lock(&rtc_task_lock);
if (rtc_callback)
    rtc_callback->func(rtc_callback->private_data);
spin_unlock(&rtc_task_lock);
wake_up_interruptible(&rtc_wait);
kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);
return IRQ_HANDLED;
}

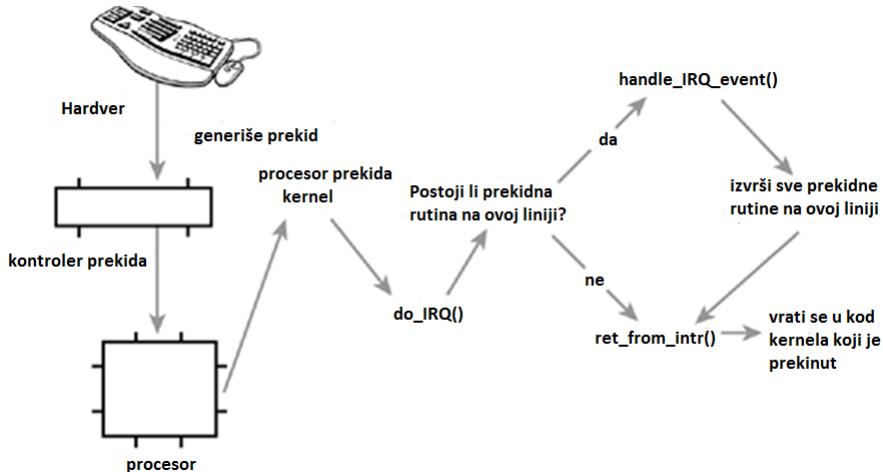
```

Ova funkcija se poziva kad god mašina primi RTC prekid. Funkcija vraća IRQ\_HANDLED za ispravan završetak.

Kada se izvršava donja polovina prekidne rutine, kernel je u kontekstu prekida. Podsetimo se da je kontekst procesa režim kornela. Kontekst prekida nije pridružen procesu. Upravljač prekida zaustavlja izvršavanje drugog kôda (čak i drugog upravljača prekida na drugoj liniji). Zbog asinhronne prirode prekida, važno je da svi upravljači prekida budu brzi i jednostavni.

## 12.2. Primena upravljanja prekidima

Primena sistema upravljanja prekidima zavisi od arhitekture sistema, tj. od procesora, tipa kontrolera prekida i same mašine. Na Slici 12.1 prikazan je dijagram putanje prekida od hardvera do kernela.



Slika 12.1: Putanja prekida od hardvera do kernela.

Uredaj izdaje prekid slanjem električnog signala kontroleru prekida preko magistrale. Ako linija prekida nije maskirana, kontroler prekida šalje prekid procesoru. Kada se prekidi omoguće u procesoru, procesor odmah zaustavlja tekući posao, one-moguće rad sistema prekida, "skače" na unapred određenu lokaciju u memoriji i izvršava kôd koji se tu nalazi. Unapred definisana lokacija u memoriji je ulazna tačka za upravljač prekida i postavlja je kernel. Kernel zna IRQ broj dolazećeg prekida; početna ulazna tačka u prekidnoj rutini samo sačuva IRQ broj i tekuće vrednosti registara (koje pripadaju prekinutom zadatku) na steku. Zatim kernel poziva funkciju do\_IRQ(). Iako je najveći deo kôda upravljača prekida napisan u jeziku C, ipak je zavisran od arhitekture. Funkcija do\_IRQ() se deklariše na sledeći način:

```
unsigned int do_IRQ(struct pt_regs regs)
```

Pošto poziv funkcije u jeziku C standardno smešta argumente funkcije na vrh steka, struktura pt\_regs sadrži početne vrednosti registara koje su prethodno sačuvane u asemblerскоj ulaznoj rutini. Pošto je i vrednost prekida takođe sačuvana, do\_IRQ() može da je iskoristi. Kôd za arhitekturu x86 je:

```
int irq = regs.orig_eax & 0xff;
```

Nakon izračunavanja linije prekida, funkcija do\_IRQ() potvrđuje prijem prekida i onemogućuje pristizanje novih prekida na liniji. Na PC mašini, ove operacije izvodi funkcija mask\_and\_ack\_8259A(), koju poziva do\_IRQ(). Zatim do\_IRQ() omogućuje da se odgovarajući upravljač prekida registruje na liniji, proverava da li je omogućen

i da li se već ne izvršava. Ako je sve u redu, poziva se funkcija `handle_IRQ_event()` koja pokreće instalirane upravljače prekida za liniju. Za arhitekturu x86 funkcija `handle_IRQ_event()` izgleda ovako:

```
asmlinkage int handle_IRQ_event(unsigned int irq, struct pt_regs *regs,
    struct irqaction *action) {
    int status = 1;
    int retval = 0;
    if (!(action->flags & SA_INTERRUPT))
        local_irq_enable();
    do {
        status |= action->flags;
        retval |= action->handler(irq, action->dev_id, regs);
        action = action->next;
    } while (action);
    if (status & SA_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);
    local_irq_disable();
    return retval;
}
```

Prvo, pošto je procesor onemogućio prekide, oni se ponovo omogućuju osim ako `SA_INTERRUPT` nije bio određen tokom registracije upravljača. Podsetimo se da `SA_INTERRUPT` određuje da li upravljač prekida treba da se pokrene sa onemogućenim prekidima. Zatim se svaki potencijalni upravljač prekida izvršava u petlji. Ako se prekidna linija ne deli, petlja se završava nakon prve iteracije; u suprotnom, izvršavaju se svi upravljači. Nakon toga, poziva se funkcija `add_interrupt_randomness()` ako je vrednost `SA_SAMPLE_RANDOM` bila određena tokom registracije. Ova funkcija koristi vreme prekida da bi generisala entropiju za generator slučajnih brojeva. Na kraju, prekidi se opet onemogućavaju (`do_IRQ()` i dalje očekuje da bude isključena) i funkcija se završava. Funkcija `do_IRQ()` vraća se na početnu ulaznu tačku, a zatim skače na funkciju `ret_from_intr()`. Rutina `ret_from_intr()` je napisana u asembleru, kao i početni ulazni kôd. Ova rutina proverava da li je predviđeno preraspoređivanje; ako jeste, i ako se kontrola iz kernela vraća u korisnički prostor (kada prekid prekine izvršenje korisničkog procesa), poziva se funkcija `schedule()`. Ako se kernel vraća u kernel prostor (prekid prekida sâm kernel), funkcija `schedule()` se poziva samo ako je vrednost `preempt_count` nula (inače nije bezbedno prinudno suspendovati kernel). Nakon završetka funkcije `schedule()`, ili ako nema nezavršenog posla, vraćaju se početne vrednosti registara i kernel nastavlja proces koji je bio prekinut. Za arhitekturu x86, početne asemblerске rutine su locirane u datoteci `arch/i386/kernel/entry.S`, dok se C metodi nalaze u `arch/i386/kernel/irq.c`. Slično je kod drugih arhitektura.

### 12.2.1. Kontrola prekida

Jezgro Linuxa implementira familiju interfejsa za rad sa prekidima na mašini. Ovi interfejsi omogućuju korisniku da onemogući sistem prekida za tekući procesor ili da maskira liniju prekida za celu mašinu. Sve pomenute rutine su zavisne od arhitekture i nalaze se u zaglavljima `<asm/system.h>` i `<asm/irq.h>`. Razlozi za kontrolu sistema prekida generalno se svode na potrebu da se obezbedi sinhronizacija. Onemogućavanje prekida garantuje da upravljač prekida neće prinudno suspendovati tekući kôd. Šta više, onemogućavanje prekida istovremeno onemogućava i prinudno suspendovanje kernela (kernel preemption). Međutim, to nije zaštita od konkurentnog pristupa drugih procesora. Pošto Linux podržava multiprocesorski rad, kôd kernela treba na neki način da bude "zaključan" da bi se sprečio konkurentni pristup deljenim podacima drugih procesora. Zaključavanje se često dešava u kombinaciji sa onemogućavanjem lokalnih prekida.

Da bi se prekidi onemogućili lokalno za tekući procesor (i samo za njega), a kasnije ponovo omogućili, treba uraditi sledeće:

```
local_irq_disable();
/* interrupts are disabled .. */
local_irq_enable();
```

Često se samo specifične linije prekida onemogućavaju za ceo sistem; to se zove *maskiranje linije prekida*. Na primer, korisnik možda želi da onemogući prekide za uređaj pre nego što utvrdi njegovo stanje. Linux obezbeđuje četiri interfejsa za ovaj zadatok:

```
void disable_irq(unsigned int irq);
void disable_irq_nosync(unsigned int irq);
void enable_irq(unsigned int irq);
void synchronize_irq(unsigned int irq);
```

Upravljački programi za nove uređaje ne koriste ove interfejse. U modernim računarima, skoro sve linije prekida mogu biti deljene.

### 12.2.2. Status sistema prekida

Makro `irqs_disabled()`, definisan u zaglavljumu `<asm/system.h>`, vraća vrednost različitu od nule kada je sistem prekida lokalnog procesora onemogućen. Ako to nije slučaj, makro vraća nulu. Dva makroa definisana u zaglavljumu `<asm/hardirq.h>` obezbeđuju interfejs za proveru tekućeg konteksta kernela:

```
in_interrupt()
in_irq()
```

Prvi makro vraća vrednost različitu od nule ako je kernel u kontekstu prekida, a nulu kada je kernel u kontekstu procesa. Makro `in_irq()` vraća vrednost različitu od nule samo kada kernel izvršava upravljač prekida.

### **12.2.3. Donja polovina upravljača i odložen posao**

Zbog različitih ograničenja, upravljači prekida mogu da obave samo prvu polovinu obrade prekida. Ta ograničenja su sledeća:

- Upravljači prekida rade asinhrono i prekidaju drugi potencijalno važan kôd. Zbog toga moraju da budu veoma brzi.
- Upravljači prekida rade sa tekućim nivoom prekida koji je u najboljem slučaju one-mogućen (ako vrednost SA\_INTERRUPT nije postavljena), a u najgorem slučaju rade sa svim prekidima koji su onemogućeni na tekućem procesoru (ako je vrednost SA\_INTERRUPT postavljena).
  - Upravljači prekida su često vremenski ograničeni zato što rade sa hardverom.
  - Upravljači prekida ne rade u kontekstu procesa, pa se stoga ne mogu blokirati.

Očigledno je da su upravljači prekida samo deo kompletног rešenja upravljanja hardverskim prekidima. Upravljanje prekidima je podeljeno na dva dela, ili polovine. Prvi deo (gornju polovinu) asinhrono izvršava kernel kao trenutni odziv na hardverski prekid. Zadatak donje polovine je da odloži i izvrši bilo koji posao koji nije vremenski ograničen. Verzija Linuxovog kernela 2.6 ima tri mehanizma rada u donjoj polovini: softirqs, tasklets i work queues (pri čemu su prva dva veoma slična). Mehanizam work queue zasniva se na nitima jezgra i najviše se koristi, zatim se koristi mehanizam tasklets i na kraju softirqs.

## Glava 13

# Sinhronizacija u kernelu

U aplikacijama koje rade sa deljenom memorijom mora se voditi računa o tome da deljeni resursi budu zaštićeni od konkurentnog pristupa. U tome ni kernel nije izuzetak. Deljeni resursi zahtevaju zaštitu od konkurentnog pristupa jer ako više niti pristupa podacima u istom trenutku, one mogu da promene postojeće rezultate ili da pristupe podacima koji su u nekonzistentnom stanju. Termin *nit izvršenja* predstavlja bilo koju instancu kôda koji se izvršava (npr. zadatak u kernelu, upravljač prekida, mehanizme donje polovine upravljanja prekidima ili kernel niti). U verziji kernela 2.0 uvedena je podrška za simetrično multiprocesiranje, tj. mogućnost da kôd kernela može istovremeno da radi sa dva ili više procesora. Od verzije 2.6 kernel je moguće prinudno suspendovati, što znači da rasporedjivač u kernelu može da prekine izvršavanje kôda kernela i dodeli procesoru drugi zadatak.

### 13.1. Kritične sekcije i stanje trke

Kôd u kome se pristupa deljenim podacima i radi sa njima zove se kritična sekcija. Da bi se izbegao konkurentan pristup kritičnim sekcijama, programer mora da obezbedi da se kôd izvršava bez prekida, tj. treba da se izvrši bez prekida kao da je cela kritična sekcija jedna nedeljiva instrukcija. Bila bi greška kada bi se dve niti istovremeno izvršavale u istoj kritičnoj sekciji; takav slučaj nazivamo stanje trke (race condition). Izbegavanje konkurentnosti i stanja trke naziva se sinhronizacija.

### 13.2. Zaključavanje

U određenom trenutku samo jedna nit može da radi sa strukturama podataka. Dakle, pristup tim strukturama podataka treba zaključati (lock) dok je druga nit izvršenja u označenoj sekciji. Mehanizam zaključavanja radi kao brava na vratima. Ako se zamisli soba kao kritična sekcija, u sobi može da bude prisutna samo jedna nit izvršenja u određenom trenutku. Kada nit uđe u sobu, ona zaključa vrata za sobom. Kada nit završi rad sa deljenim podacima, ona napušta sobu i otključava vrata. Ako druga nit stigne do zaključanih vrata, mora da čeka da nit u sobi napusti sobu i otključa

vrata. Niti drže brave (locks); brave štite podatke, odnosno zaključavaju podatke, a ne kôd. Zaključavanje štiti i redove od stanja trke.

U Linuxu su implementirani različiti mehanizmi zaključavanja. Oni se razlikuju po tome kako se zadaci ponašaju kada se primeni zaključavanje: neki zadaci su u stanju „uposlenog čekanja“ (busy wait), tj. vrte se u beskonačnoj petlji čekajući da se sekcija otključa, a drugi su u stanju „spavanja“ (sleep). Konkurentnost u kojoj se dve stvari ne dešavaju u isto vreme, ali se međusobno prepliću zove se pseudo konkurentnost. Na mašini sa simetričnim multiprocesiranjem, dva procesa mogu zaista da se izvršavaju u kritičnoj sekciji u istom trenutku. To je stvarna konkurentnost. Obe vrste konkurentnosti kao rezultat daju iste stanje trke i zahtevaju istu vrstu zaštite.

### 13.3. Metodi sinhronizacije u kernelu

Nedeljive (atomic) operacije sadrže instrukcije koje se izvršavaju bez prekida. Kernel obezbeđuje dva skupa interfejsa za nedeljive operacije, jedan koji radi sa celim brojevima i drugi koji radi sa pojedinačnim bitovima.

#### 13.3.1. Nedeljive operacije sa celim brojevima

Metodi nedeljivog celog broja rade sa specijalnim podacima tipa `atomic_t`. Ovaj tip se koristi umesto funkcija koje rade direktno sa tipom `int` iz C jezika, i to iz sledećih razloga: rad sa nedeljivim funkcijama koje prihvataju samo tip `atomic_t` omogućuje da se nedeljive operacije koriste isključivo sa ovim specijalnim tipovima. Takođe, omogućuje da se tipovi podataka ne prosleđuju ni jednoj drugoj funkciji koja nije nedeljiva. Korišćenje tipa `atomic_t` omogućava da kompajler ne optimizuje pristup vrednosti; važno je da nedeljive operacije prime odgovarajuću memorijsku adresu, a ne neku prethodnu. Konačno, korišćenje tipa `atomic_t` može da sakrije sve razlike u njegovoj primeni zavisne od arhitekture. Deklaracije koje koriste nedeljive celobrojne operacije definisane su u zaglavljumu `<asm/atomic.h>`. Kada korisnik piše kôd za kernel, treba da omogući da su ove operacije ispravno implementirane na svim arhitekturama. Definisanje tipa `atomic_t` je uobičajen način, a može da se postavi i početna vrednost:

```
atomic_t v; /* define v */
atomic_t u = ATOMIC_INIT(0);
/* define u and initialize it to zero */
```

Sve operacije su jednostavne:

```
atomic_set(&v, 4); /* v = 4 (atomically) */
atomic_add(2, &v); /* v = v + 2 = 6 (atomically) */
atomic_inc(&v); /* v = v + 1 = 7 (atomically) */
```

Ako je potrebno da se izvrši konverzija tipa `atomic_t` u tip `int`, treba upotrebiti funkciju `atomic_read()`:

```
printf("%d\n", atomic_read(&v)); /* will print "7" */
Potpun spisak standardnih nedeljivih operacija sa celim brojevima dat je u zaglavju <asm/atomic.h>, a u nastavku prikazujemo samo deo funkcija:
```

Nedeljive celobrojne operacije	Opis
ATOMIC_INIT(int i)	U deklaraciji, inicijalizacija atomic_t na i
int atomic_read atomic_t *v)	Nedeljivo čitanje celobrojne vrednosti v
void atomic_set(atomic_t *v, int i)	Nedeljivo postavljanje v na i
void atomic_add(int i, atomic_t *v)	Nedeljivo sabiranje i sa v
void atomic_sub(int i, atomic_t *v)	Nedeljivo oduzimanje i od v
void atomic_inc(atomic_t *v)	Nedeljivo inkrementiranje v

Nedeljive operacije su obično implementirane kao inline funkcije. Kada određena funkcija nasleđuje osobinu nedeljivosti, ona je samo makro. Korišćenje nedeljivih operacija umesto složenih mehanizama zaključavanja (locking) je uobičajeno.

### 13.3.2. Nedeljive operacije sa bitovima

Nedeljive operacije za rad sa bitovima nalaze se u zaglavljumu <asm/bitops.h>. Funkcije koje rade sa bitovima koriste generičke memorijske adrese, a argumenti su im pokazivači i bitovi brojeva. Bit nula je najmanje značajan bit adrese. Pošto funkcije rade sa generičkim pokazivačima, ne postoji ekvivalent nedeljivom celobrojnog tipu atomic\_t, već programer može da radi sa pokazivačem na bilo koje podatke. Sledi primer:

```
unsigned long word = 0;
set_bit(0, &word); /* bit zero is now set (atomically) */
set_bit(1, &word); /* bit one is now set (atomically) */
printf("%ul\n", word); /* will print "3" */
clear_bit(1, &word); /* bit one is now unset (atomically) */
change_bit(0, &word); /* bit zero is flipped; */
/* atomically sets bit zero and returns the previous value (zero) */
if (test_and_set_bit(0, &word)) {
    /* never true */
}
/* the following is legal;
 * you can mix atomic bit instructions with normal C */
word = 7;
```

U produžetku je dat listing nekoliko standardnih nedeljivih operacija sa bitovima:

Nedeljiva operacija sa bitovima	Opis
<code>void set_bit(int nr, void *addr)</code>	Nedeljivo postavljanje nr-tog bita počev od addr
<code>void clear_bit(int nr, void *addr)</code>	Nedeljivo brisanje nr-tog bita počev od addr

Postoje i verzije funkcija koje rade sa bitovima ali nisu nedeljive. One su istog oblika kao nedeljive funkcije, ali počinju dvostrukom podvučenom linijom. Ako je kôd koji se piše u kernelu bezbedan od uslova trke, mogu se koristiti verzije koje nisu nedeljive jer mogu biti brže (zavisno od arhitekture).

### 13.4. Spinlok

Spinlok je brava koju može da drži samo jedna nit izvršenja. Ako nit izvršenja pokuša da zaključa podatke koji su već zauzeti, ona se uposleno vrti u petlji čekajući da se podaci otključaju. Ako podaci nisu zaključani, nit može odmah da im pristupi, zaključa ih i nastavi sa radom. Na taj način sprečava se da nekoliko niti izvršenja istovremeno uđe u kritičnu sekciju. Treba obratiti pažnju da ista "brava" može da se iskoristi na više lokacija, pa svi pristupi datoj strukturi podataka, na primer, mogu biti zaštićeni i sinhronizovani. Ako se ponovo prisetimo analogije vrata i ključa, spinlok je vratar koji sedi ipred sobe, čekajući da osoba u sobi izade i preda ključ. Ako stigete do vrata i nema nikog unutra, vratar će vam dati ključ da uđete u sobu. Ako ima nekog u sobi, moraćete da čekate ključ ispred sobe, a vratar će s vremenom na vreme proveravati da li ima nekog u sobi. Kada je soba prazna, dobićete ključ i moći ćete da uđete unutra. Zahvaljujući ključu (spinlok), samo jedna osoba (nit izvršenja) u jednom trenutku može biti u sobi (kritičnoj sekciji). Činjenica da zauzet spin lock prouzrokuje da se niti vrte dok čekaju da se ključ osloboodi je veoma važna, jer se gubi procesorsko vreme. Laki spinlok koji drži jedna nit treba da traje što kraće. Alternativa je da se tekuće niti blokiraju ("spavaju") sve dok se ne osloboodi brava, kada se bude. Tada procesor može da izvrši drugi kôd. Ovo unosi dodatne opšte troškove, tj. dve promene konteksta za prebacivanje i vraćanje blokiranih niti, što je sigurno mnogo zahtevnije od primene spin locka. Spinlok treba da traje kraće od promene konteksta niti. U nastavku će biti objašnjeni semafori koji obezbeđuju bravu tako da nit koja čeka prelazi u stanje spavanja dok čeka da se brava osloboodi.

Spinlok je zavisан од arhitekture и implementiran у аSEMBLERU (`<asm/spinlock.h>`). Korisni interfejsi су definisani у заглављу `<linux/spinlock.h>`. Osnovni начин коришћења spinloka је:

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
spin_lock(&mr_lock); /* critical region */
spin_unlock(&mr_lock);
```

Bravu u određenom trenutku može držati samo jedna nit izvršenja. Sledi, samo jedna nit izvršenja može biti u jednom trenutku u kritičnoj sekciji. Ovo zahteva zaštitu niti od konkurentnog pristupa u multiprocesorskim mašinama. U jednoprocесорским mašinama brave se ne koriste. Spinlok se može koristiti u upravljačima prekida, u kojima se semafori ne mogu koristiti zato što su blokirani, tj. uspavani. Ako se brava koristi u upravljaču prekida, pre zaključavanja moraju se onemogućiti lokalni prekidi (zahtevi za prekidom u tekućem procesoru), jer bi u suprotnom bilo moguće da upravljač prekida suspenduje kôd kernela dok je brava aktivna i pokuša da zahteva zaključavanje. Upravljač prekida se vrti u petlji, čekajući da zaključavanje postane moguće. Međutim, nit koja čuva bravu se ne izvršava dok upravljač prekida ne obavi svoj zadatak. Ovo je primer dvostrukog zahteva u slučaju potpunog zastoja (deadlock). Obratite pažnju na to da prekidi treba da se onemoguče samo na tekućem procesoru. Ako se prekid pojavi na drugom procesoru, a vrti se u petlji sa istom bravom, to neće sprečiti nit koja čuva bravu (u drugom procesoru) da otključa kritičnu sekciju. Kernel obezbeđuje interfejs pogodan za onemogućavanje prekida i zahteva zaključavanje na sledeći način:

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;
spin_lock_irqsave(&mr_lock, flags);
/* critical region ... */
spin_unlock_irqrestore(&mr_lock, flags);
```

Rutina `spin_lock_irqsave()` čuva trenutno stanje prekida, onemogućavajući ih lokalno, a zatim dobija bravu. Obrnuto, funkcija `spin_unlock_irqrestore()` otključava bravu i vraća prekide u njihovo prethodno stanje. Na taj način, ako su prekidi u početku onemogućeni, kôd ih neće greškom omogućiti, već će ih održavati onemogućenim. Obratite pažnju da se promenljiva `flags` naizgled prosleđuje po vrednosti, zato što su rutine za zaključavanje delimično implementirane kao makroi. Kako veličina i složenost kernela rastu, sve je teže održavati prekide stalno onemogućenim na putanjama kôda u kernelu. Zbog toga se korišćenje funkcije `spin_lock_irq()` ne preporučuje.

#### 13.4.1. Spinlok metodi

Metod `spin_lock_init()` može da se koristi za inicijalizaciju dinamički kreiranih brave (tip `spinlock_t` kome se može pristupiti samo preko pokazivača). Metod `spin_trylock()` pokušava da dobije bravu (spinlok). Ako brava već postoji, funkcija odmah vraća nulu umesto da se vrti i čeka da se brava oslobodi. Ako zaključavanje uspe, funkcija `spin_trylock()` vraća vrednost različitu od nule. Slično, metoda

`spin_is_locked()` vraća vrednost različitu od nule ako se ostvari zaključavanje, a u suprotnom vraća nulu. Početni deo liste standardnih spinlok metoda je:

```
spin_lock() /*Acquires lock */
spin_lock_irq() /* Disables local interrupts and acquires lock */
```

### 13.5. Semafori

Semafori u Linuxu su "uspavane brave" (sleeping locks): kada zadatak pokuša da dobije semafor koji drugi zadatak već drži, semafor smešta zadatak u red za čekanje i ostavlja ga da "spava". Procesor je tada slobodan da izvršava drugi kôd. Kada proces koji ga drži oslobodi semafor, jedan od zadataka u redu za čekanje se budi da bi dobio semafor. Dakle, više procesa može istovremeno da drži semafor, ako je to potrebno. Ako primenimo analogiju vrata i ključa, kada osoba stigne do vrata, ona može da uzme ključ i uđe u sobu. Kada druga osoba dođe do vrata, a ključ nije na raspolaganju, umesto da se "vrti", osoba upiše svoje ime u listu čekanja. Kada osoba koja se nalazi u sobi napusti sobu, soba proverava listu, i ako ima upisanih imena "budi" prvu osobu sa liste i dozvoljava joj da uđe u sobu. Na ovaj način ključ (semafor) obezbeđuje da je u određenom trenutku samo jedna osoba (nit izvršenja) u sobi (kritičnoj sekciji). Semafori omogućuju bolje iskorišćenje procesora od spinloka zato što se procesorsko vreme ne troši na uposleno čekanje u petlji, ali s druge strane imaju veće opšte troškove. Semafori dozvoljavaju da proizvoljan broj zadataka istovremeno drži bravu. Broj takvih procesa se zadaje tokom deklarisanja i zove se broj korišćenja ili samo *broj*. Ako je *broj* 1, semafor se naziva binarni semafor ili muteks (pošto dozvoljava međusobno isključenje, mutual exclusion). Ako je *broj* veći od 1, semafor se zove brojački semafor (counting semaphore) i dozvoljava da bravu u određenom trenutku drži najviše zadati *broj* zadataka. Brojački semafori ne koriste međusobno isključenje zato što dozvoljavaju da više niti izvršenja istovremeno bude u kritičnoj sekciji. Semafor podržava dve nedeljive operacije: `down()` i `up()`. Metod `down()` se koristi za dobijanje semafora pri čemu se *broj* umanjuje za jedan. Ako je *broj* nula ili veći od nule, omogućava se zaključavanje i zadatak može da uđe u kritičnu sekciju. Ako je *broj* negativan, zadatak se smešta u red za čekanje, a procesor se pomera na drugi posao. Metod `up()` se koristi za oslobođenje semafora nakon završetka korišćenja kritične sekcije i povećava vrednost *broja*.

#### 13.5.1. Kreiranje i inicijalizacija semafora

Primena semafora zavisi od arhitekture, a definisana je u zaglavljumu `<asm/semaphore.h>` kao struktura `struct semaphore`. Statički deklarisani semafori kreiraju se na sledeći način:

```
static DECLARE_SEMAPHORE_GENERIC(name, count)
```

gde je name ime promenljive, a count je broj semafora. Za kreiranje muteks promenljive koristi se naredba:

```
static DECLARE_MUTEX(name);
```

gde je name ime promenljive datog semafora. Najčešće se semafori kreiraju dinamički, kao deo veće strukture:

```
sema_init(sem, count);
```

gde je sem pokazivač, a count je broj korišćenja semafora. Slično, da bi se inicijalizovao dinamički kreiran muteks može se koristiti:

```
init_MUTEX(sem);
```

### 13.5.2. Korišćenje semafora

Funkcija `down_interruptible()` pokušava da dobije semafor; ako ne uspe, "spava" u stanju `TASK_INTERRUPTIBLE`. Ovo stanje procesa ne dozvoljava da signal probudi zadatak. Ako zadatak primi signal dok čeka na semafor, budi se, a funkcija `down_interruptible()` vraća `EINTR`. Slično, funkcija `down()` smešta zadatak u stanje `TASK_UNINTERRUPTIBLE` ako "spava". Ovo u nekim slučajevima nije željeno ponašanje jer proces koji čeka na semafor ne odgovara na signale. Zbog toga se funkcija `down_interruptible()` češće koristi nego funkcija `down()`. Funkcija `down_trylock()` se može upotrebiti za pokušaj dobijanja semafora sa blokiranjem. Ako je semafor zauzet, funkcija odmah vraća broj različit od nule, u suprotnom vraća nulu i proces uspešno držite bravu. Da bi se oslobođio dati semafor, poziva se funkcija `up()`. Posmatrajmo primer:

```
/* define and declare a semaphore, named mr_sem,
 * with a count of one */
static DECLARE_MUTEX(mr_sem);
/* attempt to acquire the semaphore ... */
if (down_interruptible(&mr_sem)) {
    /* signal received, semaphore not acquired ... */
}
/* critical region ... */
/* release the given semaphore */
up(&mr_sem);
```

U nastavku su opisane metode koje se koriste u radu sa semaforima:

Metod	Opis
<code>sema_init(struct semaphore *, int)</code>	Inicijalizuje dinamički kreiran semafor za dati broj semafora
<code>init_MUTEX(struct semaphore *)</code>	Inicijalizuje dinamički kreiran semafor sa count=1
<code>init_MUTEX_LOCKED(struct semaphore *)</code>	Inicijalizuje dinamički kreiran semafor sa count=0 (inicijalno je zaključan)

### 13.5.3. Poređenje spinloka i semafora

Spinlok se može iskoristiti u kontekstu prekida, dok se semafor može iskoristiti kada je zadatak blokiran i "spava". Poređenje je dano u sledećoj tabeli:

Zahtevi	Preporučuje se
Malo troškovi zaključavanja	spinlok
Kratko vreme držanja brave	spinlok
Dugo vreme držanja brave	semafor
Potreba za zaključavanjem iz konteksta prekida	spinlok
Potreba za "spavanjem" dok se drži brava	semafor

### 13.5.4. Barijere

Kada se sinhronizuje više procesora ili hardverski uređaji, ponekad se zahteva da se iz memorije učitava i u memoriju upisuje po redosledu koji je određen programskim kôdom. Međutim, i kompjajler i procesor mogu da promene čitanje i upis da bi poboljšali performanse. Svi procesori koji menjaju redosled čitanja ili upisa podržavaju mašinske instrukcije kojima se to postiže. Takođe, kompjajleru je moguće dati instrukcije da ne menja redosled; te instrukcije se zovu **barijere**. Intelovi procesori iz familije x86 nikada ne menjaju redosled upisa, dok drugi procesori to rade.

## Glava 14

# Upravljanje memorijom

Ovo poglavlje opisuje metode koje se koriste za upravljanje memorijom unutar Linuxovog kernela.

### 14.1. Stranice

Fizičke stranice su osnovne jedinice upravljanja memorijom u kernelu. Iako je najmanja jedinica adresiranja procesora obično reč (ili bajt), jedinica za upravljanje memorijom (memory management unit, MMU) koja prevodi virtualne adrese u fizičke po pravilu radi sa stranicama. MMU upravlja tabelom stranica u sistemu. Različite arhitekture definišu različite veličine stranice, a mnoge podržavaju i više veličina stranice. Po pravilu, 32-bitne arhitekture koriste stranice veličine 4KB, a većina 64-bitnih arhitektura ima stranice veličine 8KB. Na osnovu toga se izračunava da je u mašini koja radi sa stranicama veličine 4KB i veličinom fizičke memorije od 1GB, fizička memorija podeljena u 262 144 različite stranice. Kernel svaku fizičku stranicu u sistemu modelira struktrom struct page definisanom u zaglavju <linux/mm.h>:

```
struct page {
    page_flags_t flags;
    atomic_t _count;
    atomic_t _mapcount;
    unsigned long private;
    struct address_space *mapping;
    pgoff_t index;
    struct list_head lru;
    void *virtual;
};
```

Polje flags čuva status stranice, npr. da li je strana “dirty” ili je zaključana u memoriji. Na raspolaganju su istovremeno najmanje 32 različite vrednosti za promenljivu flags, definisane u zaglavju <linux/page-flags.h>. Polje \_count sadrži broj referenci na stranicu. Polje virtual je adresa strane u virtualnoj memoriji. Važno je razumeti da je struktura stranice pridružena fizičkim stranama, a ne virtualnim. Cilj je da strukture podataka opišu fizičku memoriju, a ne podatke koji se nalaze u njima.

Kernel koristi ovu strukturu za praćenje svih stranica u sistemu. Kernel treba da zna da li je stranica slobodna (nije alocirana, tj. dodeljena), a ako nije, treba da zna ko je njen vlasnik (korisnik, dinamički alocirani kernel podaci, statički kernel kod, keš strana itd). Prepostavimo da struktura stranice troši 40 bajtova memorije, fizička stranica u sistemu 4KB i da sistem ima 128MB fizičke memorije. U tom slučaju sve strukture stranica u sistemu troše oko 1MB memorije, što nisu veliki troškovi.

## 14.2. Zone

Zbog hardverskih ograničenja, kernel ne može prema svim stranicama da se odnosi na isti način. Neke stranice zbog fizičkih adresa u memoriji ne mogu da se koriste za određene zadatke. Zbog toga kernel deli stranice po sličnim osobinama u različite zone. Linux radi sa hardverom sa sledećim adresiranjem u memoriji:

- ▷ Neki hardverski uređaji su sposobni da direktno pristupe memoriji (uz pomoć DMA) sa određene memorijske adrese.
- ▷ Neke arhitekture su sposobne da fizički adresiraju veći adresni prostor memorije nego što mogu virtualne adrese. Zbog toga u njima postoji memorija koja nije trajno mapirana u adresni prostor kernela.

Pomenuta ograničenja diktiraju postojanje tri memorijske zone u Linuxu:

- ▷ `ZONE_DMA`: sadrži stranice koje su sposobne za DMA.
- ▷ `ZONE_NORMAL`: sadrži normalne, standardno mapirane stranice.
- ▷ `ZONE_HIGHMEM`: sadrži visoku memoriju, tj. stranice koje nisu trajno mapirane u adresni prostor kernela.

Zone su definisane u zaglavlju `<linux/mmzone.h>` u strukturi `struct zone`:

```
struct zone {
    spinlock_t lock;
    unsigned long free_pages;
    unsigned long pages_min;
    unsigned long pages_low;
    unsigned long pages_high;
    unsigned long protection[MAX_NR_ZONES];
    spinlock_t lru_lock;
    struct list_head active_list;
    struct list_head inactive_list;
    unsigned long nr_scan_active;
    unsigned long nr_scan_inactive;
    unsigned long nr_active;
    unsigned long nr_inactive;
    int all_unreclaimable;
    unsigned long pages_scanned;
    int temp_priority;
    int prev_priority;
```

```

    struct free_area free_area[MAX_ORDER];
    wait_queue_head_t *wait_table;
    unsigned long wait_table_size;
    unsigned long wait_table_bits;
    struct per_cpu_pageset pageset[NR_CPUS];
    struct pglist_data *zone_pgdat;
    struct page *zone_mem_map;
    unsigned long zone_start_pfn;
    char *name;
    unsigned long spanned_pages;
    unsigned long present_pages;
};


```

Struktura je velika, ali pošto postoje samo tri zone u sistemu, ima samo tri strukture. Polje `lock` je spinlok koji štiti strukturu od konkurentnog pristupa. Treba obratiti pažnju da se štiti samo struktura, a ne sve stranice u zoni. Polje `free_pages` je broj slobodnih stranica u zoni. Kernel pokušava da održi najmanje `pages_min` stranica slobodnim (kroz zamenu, tj. swapping), ako je to moguće. Polje `name` je string završen nulom koji predstavlja ime zone. Kernel inicijalizuje ovu vrednost tokom podizanja sistema (boot) u datoteci `mm/page_alloc.c` i daje imena zonama "DMA", "Normal" i "HighMem".

### 14.3. Učitavanje stranica

Kernel koristi mehanizam niskog nivoa kada zahteva memoriju, zajedno sa neko-liko interfejsa za pristup memoriji. Svi interfejsi alociranja memorije sa granularnošću strane deklarisani su u zaglavljumu `<linux/gfp.h>`. Glavna funkcija je:

```
struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)
```

Ova funkcija alocira  $2^{\text{order}}$  neprekidnih fizičkih stranica (pri čemu je `order >> 1`) i vraća pokazivač na prvu strukturu stranice `page`; ako se desi greška vraća `NULL`. Stranica može da se konvertuje u logičku adresu pomoću funkcije:

```
void * page_address(struct page *page)
```

Ova funkcija vraća pokazivač na logičku adresu na kojoj se trenutno nalazi fizička stranica. Ako tekuća struktura `page` nije potrebna, može se pozvati:

```
unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int order)
```

Ova funkcija radi isto kao i `alloc_pages()`, osim što direktno vraća logičku adresu prve zahtevane stranice. Pošto su stranice susedne, druge stranice prosto slede prvu.

Ako je potrebna samo jedna stranica, koriste se dve funkcije:

```
struct page * alloc_page(unsigned int gfp_mask)
```

```
unsigned long __get_free_page(unsigned int gfp_mask)
```

## 14.4. Oslobođanje stranica

Familija funkcija koja omogućuje oslobođanje alociranih stranica kada više nisu potrebne je:

```
void __free_pages(struct page *page, unsigned int order)
void free_pages(unsigned long addr, unsigned int order)
void free_page(unsigned long addr)
```

Prilikom oslobođanja treba biti pažljiv jer oslobođate samo stranice koje ste vi aločirali. Prosleđivanje pogrešne strukture stranice (ili adresu) ili neispravan redosled mogu da prouzrokuju oštećenje podataka. Pogledajmo primer kada želimo da aločiramo osam stranica:

```
unsigned long page;
page = __get_free_pages(GFP_KERNEL, 3);
if (!page) {
    /* nema dovoljno memorije:
     * morate obraditi ovu gresku! */
    return ENOMEM;
}
/* 'page' je sada adresa
 * prve od osam uzastopnih stranica ... */
free_pages(page, 3);
/* * stranice su sada oslobođene
 * i vise ne bi trebalo
 * pristupati adresi koja se
 * cuva u 'page' */
```

Parametar GFP\_KERNEL je primer flega gfp\_mask. Pomenute funkcije su korisne kada su potrebni delovi memorije veličine jedne stranice, posebno ako je potrebna tačno jedna ili dve stranice. Za opštije alokacije veličine bajta kernel obezbeđuje funkciju kmalloc().

## 14.5. Funkcija kmalloc()

Funkcija kmalloc() radi slično rutini iz korisničkog prostora malloc(), sa izuzetkom dodatnih parametara. Funkcija kmalloc() je jednostavan interfejs za dobijanje kernel memorije veličine bajtova, a deklarisana je u zaglavlju <linux/slab.h>:

```
void * kmalloc(size_t size, int flags)
```

Funkcija vraća pokazivač na oblast u memoriji koja je najmanje veličine size bajtova i koja je fizički susedna. Ako postoji greška, funkcija vraća NULL. Alokacija memorije kernelu polazi za rukom sve dok ima dovoljno slobodne memorije; ako je više nema, na osnovu povratne vrednosti funkcije kmalloc() koja je u tom slučaju NULL treba reagovati na grešku na odgovarajući način.

Metod kfree() je deklarisana u zaglavlju <linux/slab.h> na sledeći način:

```
void kfree(const void *ptr)
```

Ovaj metod oslobađa blok memorije prethodno alocirane funkcijom `kmalloc()`.

## 14.6. Funkcija `vmalloc()`

Funkcija `vmalloc()` radi na sličan način kao i `kmalloc()`, osim što alocira memoriju koja je samo virtualno susedna (nije neophodno da je fizički susedna). Strane koje vraća `malloc()` su susedne unutar virtualnog adresnog prostora procesora, ali nema garancije da su one susedne i u fizičkoj memoriji. Funkcija `kmalloc()` garantuje da su strane i fizički i virtualno susedne. Memorija koja se pojavljuje u kernelu je logički susedna. Uprkos činjenici da se fizički susedna memorija zahteva samo u određenim slučajevima, kôd kernela uglavnom koristi funkciju `kmalloc()` a ne `vmalloc()` za dobijanje memorije. Funkcija `vmalloc()` se koristi samo kada je to neophodno, tj. za dobijanje veoma velikih regionalnih memorija. Na primer, kada se moduli dinamički učitavaju u kernel, oni se učitavaju u memoriju kreiranu pomoću funkcije `vmalloc()`. Ova funkcija postiže slabije rezultate pri radu sa TLB-om. TLB (translation lookaside buffer) je hardverski keš koji koriste mnoge arhitekture da bi keširale mapiranje virtualnih adresa u fizičke adrese. Pošto se većina pristupa memoriji obavlja preko virtualnog adresiranja, time se poboljšavaju karakteristike sistema. Funkcija `vmalloc()` je deklarisana u zagлављу `<linux/vmalloc.h>` i definisana u `mm/vmalloc.c`. Koristi se isto kao C funkcija `malloc()` iz korisničkog prostora:

```
void * vmalloc(unsigned long size)
```

Funkcija vraća pokazivač na najmanje `size` bajtova virtualnog susednog prostora u memoriji. U slučaju greške, funkcija vraća NULL. Funkcija može da „spava“ i tada ne može biti pozvana iz konteksta prekida ili iz drugih situacija u kojima blokiranje nije dozvoljeno. Za oslobadanje memorije dobijene preko funkcije `vmalloc()` koristi se funkcija `vfree()`:

```
void vfree(void *addr)
```

Ova funkcija oslobađa blok memorije koji počinje na adresi `addr`. Funkcija takođe može da „spava“ i tada ne može biti pozvana iz konteksta prekida. Korišćenje funkcije je veoma jednostavno:

```
char *buf;
buf = vmalloc(16 * PAGE_SIZE);
/* get 16 pages */
if (!buf) /* error! failed to allocate memory */
/* * buf now points to at least a 16*PAGE_SIZE bytes
 * of virtually contiguous block of memory */
```

Nakon završetka korišćenja, memoriju treba osloboditi pozivom funkcije `vfree(buf)`.

### 14.7. Statičko alociranje na steku

U korisničkom prostoru moguće je alocirati prostor na steku zato što je veličina prostora koji treba da bude alociran poznata unapred. Stek kernela je mali i fiksne veličine. Kada se svakom procesu dodeli mali stek fiksne veličine, potrošnja memorije se minimizira, a kernel se ne opterećuje kôdom koji upravlja radom steka. Veličina steka kernela po procesu zavisi od arhitekture i opcija definisanih u vreme kompjuiranja operativnog sistema. Pre verzije 2.6 kernela, veličina kernel steka je bila dve stranice po procesu (obično 8KB za 32-bitnu arhitekturu, odnosno 16KB za 64-bitnu arhitekturu zato što su veličine strana 4KB i 8KB, redom). Počev od verzije kernela 2.6, stek se postavlja na jednu stranu, i to iz dva razloga. Prvo, procesi troše malo strana. Drugi, važniji razlog je to što sa povećanjem vremena rada postaje veoma teško pronaći dve fizički susedne nealocirane strane. Fizička memorija postaje fragmentirana, pa alokacija usamljenog novog procesa virtuelne mašine postaje veoma skupa. Tada se uvode stekovi prekida koji obezbeđuju stek za upravljače prekida jednog procesora; takvi stekovi više ne dele kernel stek prekinutog procesa već imaju sopstvene stekove.

### 14.8. Mapiranje stranica u memoriji sa velikim adresama

Stranice u memoriji na višim adresama ne moraju da budu stalno mapirane u adresni prostor kernela. To znači da stranice dobijene preko funkcije `alloc_pages()` sa flegom `__GFP_HIGHMEM` možda nemaju logičku adresu. U arhitekturi x86 cela fizička memorija iznad 896MB je visoka memorija i nije stalno niti automatski mapirana u adresni prostor kernela, uprkos tome što su x86 procesori sposobni da fizički adresiraju vrednosti adresa iznad 4GB fizičke memorije. Nakon alociranja, stranica mora da bude mapirana u logički adresni prostor kernela. U arhitekturi x86, stranice u visokoj memoriji su mapirane negde između 3 i 4GB.

Za mapiranje strukture stranice u adresni prostor kernela koristi se funkcija  
`void *kmap(struct page *page)`

Ova funkcija radi ili u visokoj ili niskoj memoriji. Ako struktura strane pripada strani u niskoj memoriji, funkcija vraća virtualnu adresu strane. Ako se strana nalazi u visokoj memoriji, mapiranje je trajno i funkcija vraća adresu. Funkcija može da "spava"; tada `kmap()` radi samo u kontekstu procesa. Pošto je broj trajnih mapiranja ograničen, visoka memorija ne treba da bude mapirana kada više nije potrebna. To se postiže preko funkcije

`void kunmap(struct page *page)`

Kada treba obaviti mapiranje, ali tekući kontekst nije u mogućnosti da "spava", kernel obezbeđuje privremeno mapiranje (koje se takođe zove i nedeljivo mapiranje). Radi se o skupu rezervnih mapiranja za privremene svrhe. Kernel može nedeljivo da mapira stranicu u visokoj memoriji u neko rezervisano mapiranje. Privremeno mapiranje dakle može da se koristi u slučajevima kada nema "spavanja", npr. za upravljače

prekida, zato što mapiranje ne može da bude blokirano. Privremeno mapiranje postiže se funkcijom

```
void *kmap_atomic(struct page *page, enum km_type type)
```

Parametar type je nabranje koje opisuje svrhu privremenog mapiranja i definisano je u zaglavlju `<asm/kmap_types.h>`. Ova funkcija ne blokira, pa može da se koristi u kontekstu prekida i za druge slučajeve gde nema ponovnog raspoređivanja. Ona onemogućuje i prinudnu suspenziju kernela, što je važno zato što je mapiranje jedinstveno za svaki procesor. Ponovno raspoređivanje moglo bi da promeni redosled izvršavanja zadataka u različitim procesorima. Mapiranje se poništava na sledeći način:

```
void kunmap_atomic(void *kvaddr, enum km_type type)
```

#### 14.8.1. Koji alokacioni metod iskoristiti?

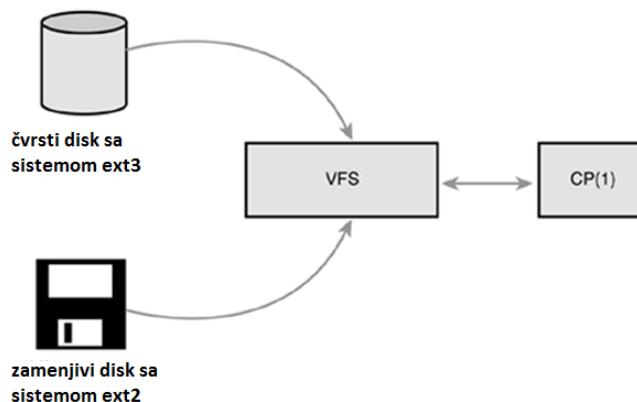
Ako su potrebne neprekidne fizičke stranice, treba upotrebiti neki alokator stranica niskog nivoa ili `kmalloc()`. To je standardan način alociranja memorije unutar kernela. Ako treba alocirati stranice iz više memorije, upotrebite `alloc_pages()`. Funkcija `alloc_pages()` vraća strukturu stranice, a ne pokazivač na logičku adresu. Pošto visoka memorija možda nije mapirana, jedini način da joj se pristupi može biti preko odgovarajuće strukture stranice. Da biste dobili stvarni pokazivač, koristite `kmap()` za mapiranje visoke memorije u logički adresni prostor kernela. Ako nisu potrebne fizičke susedne stranice, već samo virtualno susedne, treba upotrebiti `vmalloc()`, mada uvek treba imati na umu slabe karakteristike funkcije `vmalloc()` u poređenju sa funkcijom `kmalloc()`. Funkcija `vmalloc()` alocira kao u korisničkom prostoru, mapirajući velike delove fizičke memorije u susedni logički adresni prostor.



## Glava 15

# Virtuelni sistem datoteka

Virtuelni sistem datoteka (Virtual File System, VFS) je podsistem kernela koji implementira interfejsa sistema datoteka za programe korisničkog prostora. Svi sistemi datoteka se oslanjaju na VFS da bi im se dozvolilo ne samo da postoje, nego i da sarađuju. To omogućuje programima da koriste standardne UNIX sistemske pozive `read()` i `write()` za različite sisteme datoteka na različitim medijumima, kao što je prikazano na slici 15.1.



Slika 15.1: Virtuelni sistem datoteka: Korišćenje cp(1) karakteristika za premeštanje podataka sa diska sa sistemom ext3 na prenosivi disk sa sistemom ext2 (dva različita sistema datoteka, dva različita medijuma i jedan VFS).

### 15.1. Interfejs sistema datoteka

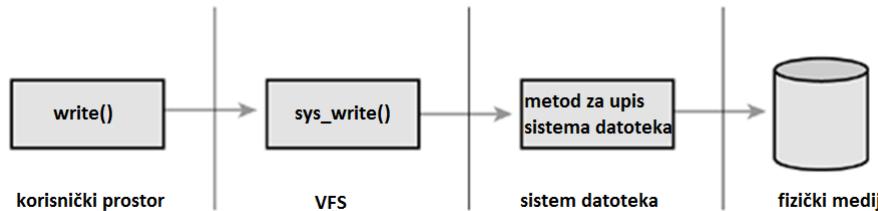
Virtuelni sistem datoteka je “lepak” koji omogućuje sistemskim pozivima kao što su `open()`, `read()` i `write()` da rade bez obzira na sistem datoteka koji se koristi ili na fizički medijum. Moderni operativni sistemi apstrahuju pristup sistemima datoteka preko virtuelnog interfejsa koji omogućuje saradnju i uopšten pristup.

Opšti interfejs za bilo koji tip sistema datoteka je ostvarljiv samo zato što sâm kernel primenjuje sloj apstrakcije oko svog interfejsa sistema datoteka niskog nivoa. Taj

sloj apstrakcije omogućuje Linuxu podršku za različite sisteme datoteka, čak i ako se oni veoma razlikuju u podržanim karakteristikama ili ponašanju. Kao primer, razmotrimo izvršenje programa u korisničkom prostoru:

```
write(f, &buf, len);
```

Ova naredba upisuje `len` bajtova na koje pokazuje pokazivač `buf` u tekuću poziciju u datoteci predstavljenoj deskriptorom `f`. Ovim sistemskim pozivom prvo rukuje generički sistemski poziv `sys_write()` koji određuje aktuelni metod upisivanja u datoteku u sistemu datoteka u kome se nalazi `f`. Generički sistemski poziv za upis tada poziva ovaj metod da upiše podatke na medijum, što je deo implementacije sistema datoteka. Na Slici 15.2 prikazan je dijagram toka od korisničkog poziva `write()` preko podataka do fizičkog medijuma. Sa jedne strane sistemskog poziva je generički VFS interfejs, koji obezbeđuje vezu ka korisničkom prostoru; sa druge strane sistemskog poziva je deo sistema datoteka koji se bavi implementacijom detalja.



Slika 15.2: Tok podataka od korisničkog prostora iz koga se izdaje sistemski poziv `write()`, kroz generički poziv virtuelnog sistema datoteka (VFS), preko specifičnog metoda `write()` sistema datoteka do fizičkog medijuma.

## 15.2. UNIX sistem datoteka

UNIX obezbeđuje četiri osnovne apstrakcije u odnosu na sistem datoteka: datoteke, direktorijske upise, inode i tačke montiranja. Sistemi datoteka sadrže datoteke, direktorijske upise i pridružene kontrolne informacije. Tipične operacije u sistemu su kreiranje, brisanje i montiranje datoteka i direktorijskih stabala. U UNIX-u, sistemi datoteka su montirani u određenim tačkama globalne hijerarhije, poznatim kao prostor imena. To omogućuje da se svi montirani sistemi datoteka prikažu kao upisi u prostom stablu. Datoteka je uređeni niz, tj. string bajtova. Prvi bajt označava početak datoteke, a poslednji bajt označava njen kraj. Svaku datoteku je pridruženo ime za identifikaciju i sistema i korisnika. Tipične operacije sa datotekom su čitanje (read), upis (write), kreiranje (create) i brisanje (delete). Datoteke su organizovane u direktorijske upise, koji mogu da sadrže poddirektorijske upise. Direktorijski mogu biti ugnježđeni čime se formiraju putanje. Svaka komponenta putanje zove se unos direktorijskog imena (directory entry). Primer putanja je `/home/korisnik/ime_datoteke;`

u ovom slučaju korenski (root) direktorijum je `/`, direktorijumi `home` i `korisnik` i datoteka `ime_datoteke` su unosi direktorijuma nazvani **dentry** (skraćenica od directory entry). UNIX direktorijumi su zapravo standardne datoteke koje sadrže listu datoteka u tom direktorijumu i sa njima se mogu obavljati iste operacije se kao sa datotekama. Informacije o datoteci kao što su dozvole pristupa, veličina, vlasništvo, vreme kreiranja itd. nazivaju se metapodacima (metadata) i smeštaju se u posebne strukture zvane **inode** (skraćenica od index node). Sve te informacije zajedno sa kontrolnim informacijama sistema datoteka nalaze se u superbloku. To je struktura podataka koja sadrži informacije o sistemu datoteka kao celini.

Linux podržava brojne sisteme datoteka (preko 50 u zvaničnoj verziji kernela), od izvornih kao što su ext2 i ext3, do mrežnih sistema datoteka (kao što su NFS i Coda). VFS sloj obezbeđuje sistemima datoteka radno okruženje za primenu i interfejs za rad sa standardnim sistemskim pozivima.

### 15.3. Objekti sistema datoteka i njihove strukture podataka

VFS je objektno orijentisan. Familija struktura podataka predstavlja zajednički model datoteke. Ove strukture podataka su slične objektima. Pošto je kernel programiran u jeziku C, bez pogodnosti jezika koji su objektno orijentisani, strukture podataka su predstavljene kao C strukture. Strukture sadrže i podatke i pokazivače na funkcije implementirane u sistemu datoteka. Četiri osnovna tipa VFS objekata su:

- ▷ superblok (određena montirana datoteka sistema)
- ▷ inode (predstavlja određenu datoteku)
- ▷ dentry (unos u direktorijum, tj. prosta komponenta putanje)
- ▷ otvorena datoteka pridružena procesu

Pošto VFS tretira direktorijume kao standardne datoteke, ne postoji objekat direktorijum. Svaki objekat nalazi se unutar nekog od prethodno navedenih primarnih objekata. Najvažniji primarni objekti za koje kernel poziva određene metode su:

- ▷ super\_operations (sadrži metode koje kernel može da pozove u specifičnom sistemu datoteka, npr. `read_inode()` i `sync_fs()`)
- ▷ inode\_operations (sadrži metode koje kernel može da pozove u specifičnoj datoteci, npr. `create()` i `link()`)
- ▷ dentry\_operations (sadrži metode koje kernel može da pozove za određen dentry, npr. `d_compare()` i `d_delete()`)
- ▷ file (sadrži metode koje proces može da pozove za otvorenu datoteku, npr. `read()` i `write()`).

Primarni objekti su implementirani kao strukture pokazivača na funkcije koje rade sa roditeljskim objektom. Za mnoge metode objekti mogu da naslede generičku funkciju ako je njena osnovna funkcionalnost dovoljna. U suprotnom, specifična instanca određenog sistema datoteka dopunjuje pokazivače njihovim metodama specifičnim

za taj sistem datoteka. Objekti povezani sa strukturama nisu standardni objekti, po-put onih u jezicima C++ ili Java. Ove strukture predstavljaju određene instance objekata, njihovih pridruženih podataka i metoda koji rade sa njima.

Svaki registrovani sistem datoteka je predstavljen strukturom `file_system_type`. Ovaj objekat opisuje sistem datoteka i njegove mogućnosti. Svaka tačka montiranja je predstavljena strukturom `vfsmount`. Strukture koje opisuju sistem datoteka i dатотеке pridružene svakom procesu su: `file_struct`, `fs_struct` i `namespace`.

#### 15.4. Objekat superblok

Svaki sistem datoteka implementira objekat superblok i u njemu čuva informacije koje ga opisuju. Ovaj objekat obično odgovara superbloku ili kontrolnom bloku sistema datoteka koji je smešten u specijalnom sektoru na disku. Objekat superblok je predstavljen strukturom `super_block` i definisan u zaglavlju `<linux/fs.h>`. Sledi njegov kôd:

```
struct super_block {
    struct list_head s_list; /* list of all superblocks */
    dev_t s_dev; /* identifier */
    unsigned long s_blocksizе; /* block size in bytes */
    unsigned long s_old_blocksizе; /* old block size in bytes */
    unsigned char s_blocksizе_bits; /* block size in bits */
    unsigned char s_dirt; /* dirty flag */
    unsigned long long s_maxbytes; /* max file size */
    struct file_system_type s_type; /* filesystem type */
    struct super_operations s_op; /* superblock methods */
    struct dquot_operations *dq_op; /* quota methods */
    struct quotactl_ops *s_qcop; /* quota control methods */
    struct export_operations *s_export_op; /* export methods */
    unsigned long s_flags; /* mount flags */
    unsigned long s_magic; /* filesystem's magic number */
    struct dentry *s_root; /* directory mount point */
    struct rw_semaphore s_umount; /* unmount semaphore */
    struct semaphore s_lock; /* superblock semaphore */
    int s_count; /* superblock ref count */
    int s_syncing; /* filesystem syncing flag */
    int s_need_sync_fs; /* not-yet-synced flag */
    atomic_t s_active; /* active reference count */
    void *s_security; /* security module */
    struct list_head s_dirty; /* list of dirty inodes */
    struct list_head s_io; /* list of writebacks */
    struct hlist_head s_anon; /* anonymous dentries */
    struct list_head s_files; /* list of assigned files */
    struct block_device *s_bdev; /* associated block device */
```

```

    struct list_head s_instances; /* instances of this fs */
    struct quota_info s_dquot; /* quota-specific options */
    char s_id[32]; /* text name */
    void *s_fs_info; /* filesystem-specific info */
    struct semaphore s_vfs_rename_sem; /* rename semaphore */
};

Kôd za kreiranje, upravljanje i uništavanje superblokova nalazi se u datoteci fs/super.c. Objekat superblok se kreira i inicijalizuje pomoću funkcije alloc_super().

```

#### 15.4.1. Operacije sa superblokom

Najvažniji član u objektu superblok je `s_op`; to je zapravo tabela operacija sa superblokom, predstavljena strukturom `super_operations` i definisana u zaglavlju `<linux/fs.h>`. Sledi njen kôd:

```

struct super_operations {
    struct inode *(*alloc_inode) (struct super_block *sb);
    void (*destroy_inode) (struct inode *);
    void (*read_inode) (struct inode *);
    void (*dirty_inode) (struct inode *);
    void (*write_inode) (struct inode *, int);
    void (*put_inode) (struct inode *);
    void (*drop_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*sync_fs) (struct super_block *, int);
    void (*write_super_lockfs) (struct super_block *);
    void (*unlockfs) (struct super_block *);
    int (*statfs) (struct super_block *, struct statfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);
    int (*show_options) (struct seq_file *, struct vfsmount *);
};

Svaki član ove strukture je pokazivač na funkciju koja radi sa superblok objektom. Operacije sa superblokom su operacije niskog nivoa i izvršavaju se u sistemu datoteka i njegovim inodovima. U nastavku je nabrojano nekoliko operacija sa superblokom koje rade sa strukturom super_operations:

```

- ▷ `struct inode * alloc_inode(struct super_block *sb)`: funkcija koja kreira i inicijalizuje novi inode objekat pod datim superblokom.
- ▷ `void destroy_inode(struct inode *inode)`: funkcija koja dealocira dati inode.
- ▷ `void read_inode(struct inode *inode)`: funkcija koja čita datoteku inode određenu sa `inode->i_ino` sa diska i popunjava ostatak strukture inode.

▷ `void dirty_inode(struct inode *inode)`: funkcija koju poziva VFS kada je inode „prljav“ (modifikovan).

Sve pomenute funkcije se pozivaju u virtuelnom sistemu datoteka u kontekstu procesa i mogu se po potrebi blokirati.

### 15.5. Objekat inode

Objekat inode predstavlja sve informacije koje su kernelu potrebne za rad sa datotekom ili direktorijumom. Za UNIX-olike sisteme datoteka, informacija se dobija čitanjem inoda sa diska. Objekat inode je predstavljen strukturom `inode` i definisan u zaglavlju `<linux/fs.h>`:

```
struct inode {
    struct hlist_node i_hash; /* hash list */
    struct list_head i_list; /* list of inodes */
    struct list_head i_dentry; /* list of dentries */
    unsigned long i_ino; /* inode number */
    atomic_t i_count; /* reference counter */
    umode_t i_mode; /* access permissions */
    unsigned int i_nlink; /* number of hard links */
    uid_t i_uid; /* user id of owner */
    gid_t i_gid; /* group id of owner */
    kdev_t i_rdev; /* real device node */
    loff_t i_size; /* file size in bytes */
    struct timespec i_atime; /* last access time */
    struct timespec i_mtime; /* last modify time */
    struct timespec i_ctime; /* last change time */
    unsigned int i_blkbits; /* block size in bits */
    unsigned long i_blksize; /* block size in bytes */
    unsigned long i_version; /* version number */
    unsigned long i_blocks; /* file size in blocks */
    unsigned short i_bytes; /* bytes consumed */
    spinlock_t i_lock; /* spinlock */
    struct rw_semaphore i_alloc_sem; /* nests inside of i_sem */
    struct semaphore i_sem; /* inode semaphore */
    struct inode_operations *i_op; /* inode ops table */
    struct file_operations *i_fop; /* default inode ops */
    struct super_block *i_sb; /* associated superblock */
    struct file_lock *i_flock; /* file lock list */
    struct address_space *i_mapping; /* associated mapping */
    struct address_space i_data; /* mapping for device */
    struct dquot *i_dquot[MAXQUOTAS]; /* disk quotas for inode */
    struct list_head i_devices; /* list of block devices */
    struct pipe_inode_info *i_pipe; /* pipe information */
    struct block_device *i_bdev; /* block device driver */
```

```

unsigned long i_dnotify_mask; /* directory notify mask */
struct dnotify_struct *i_dnotify; /* dnotify */
unsigned long i_state; /* state flags */
unsigned long dirtied_when; /* first dirtying time */
unsigned int i_flags; /* filesystem flags */
unsigned char i_sock; /* is this a socket? */
atomic_t i_writecount; /* count of writers */
void *i_security; /* security module */
__u32 i_generation; /* inode version number */
union {
    void *generic_ip; /* filesystem-specific info */
} u;
};


```

### 15.5.1. Operacije sa inodom

Kao u slučaju operacija sa superblokom, član `inode_operations` je veoma važan. On opisuje implementirane funkcije sistema datoteka koje VFS može da pozove za inode. Inode operacije se pozivaju na sledeći način:

```
i->i_op->truncate(i)
```

gde je `i` referenca na određeni inode. U ovom slučaju za dati inode poziva se operacija `truncate()` definisana za sistem datoteka u kome postoji `i`. Struktura `inode_operations` je definisana u zaglavlju `<linux/fs.h>`:

```

struct inode_operations {
    int (*create) (struct inode *, struct dentry *,int);
    struct dentry * (*lookup) (struct inode *, struct dentry *);
    int (*link) (struct dentry *, struct inode *, struct dentry *);
    int (*unlink) (struct inode *, struct dentry *);
    int (*symlink) (struct inode *, struct dentry *, const char *);
    int (*mkdir) (struct inode *, struct dentry *, int);
    int (*rmdir) (struct inode *, struct dentry *);
    int (*mknod) (struct inode *, struct dentry *, int, dev_t);
    int (*rename) (struct inode *, struct dentry *,
                  struct inode *, struct dentry *);
    int (*readlink) (struct dentry *, char *, int);
    int (*follow_link) (struct dentry *, struct nameidata *);
    int (*put_link) (struct dentry *, struct nameidata *);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int);
    int (*setattr) (struct dentry *, struct iattr *);
    int (*getattr) (struct vfsmount *, struct dentry *, struct kstat *);
    int (*setxattr) (struct dentry *, const char *, const void *,
                    size_t, int);
    ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
};


```

```
    int (*removexattr) (struct dentry *, const char *);  
};
```

Virtuelni sistem datoteka između ostalih nudi i sledeće funkcije za inode:

- ▷ `int create(struct inode *dir, struct dentry *dentry, int mode)`: VFS poziva ovu funkciju iz sistemskih poziva `create()` i `open()` za kreiranje novog inoda pridruženog objektu dentry u zadatom početnom režimu.
- ▷ `struct dentry * lookup(struct inode *dir, struct dentry *dentry)`: funkcija koja pretražuje direktorijum za jedan inode prema imenu datoteke zadatom u dentry.
- ▷ `int link(struct dentry *old_dentry, struct inode *dir, struct dentry *dentry)`: funkcija koja se poziva sistemskim pozivom `link()` za kreiranje hard linka do datoteke `old_dentry` u direktorijumu `dir` sa novim imenom datoteke `dentry`.

## 15.6. Objekat dentry

Već smo saznali da virtuelni sistem datoteka tretira direktorijume kao datoteke. U putanji `/bin/vi`, `bin` i `vi` su datoteke; `bin` je specijalna datoteka direktorijuma, a `vi` je standardna datoteka. Objekat inode predstavlja obe ove komponente. VFS koristi koncept stavki direktorijuma (directory entry, dentry). To su određene komponente putanje kao i sama datoteka. Koristeći prethodni primer, `/`, `bin` i `vi` su dentry objekti. Objekti dentry su predstavljeni strukturom `dentry` i definisani u zaglavljku `<linux/dcache.h>`. Sledi struktura sa komentarima koji opisuju svaki član:

```
struct dentry {  
    atomic_t d_count; /* usage count */  
    unsigned long d_vfs_flags; /* dentry cache flags */  
    spinlock_t d_lock; /* per-dentry lock */  
    struct inode *d_inode; /* associated inode */  
    struct list_head d_lru; /* unused list */  
    struct list_head d_child; /* list of dentries within */  
    struct list_head d_subdirs; /* subdirectories */  
    struct list_head d_alias; /* list of alias inodes */  
    unsigned long d_time; /* revalidate time */  
    struct dentry_operations *d_op; /* dentry operations table */  
    struct super_block *d_sb; /* superblock of file */  
    unsigned int d_flags; /* dentry flags */  
    int d_mounted; /* is this a mount point? */  
    void *d_fsdmeta; /* filesystem-specific data */  
    struct rcu_head d_rcu; /* RCU locking */  
    struct dcookie_struct *d_cookie; /* cookie */  
    struct dentry *d_parent; /* dentry object of parent */  
    struct qstr d_name; /* dentry name */
```

```

struct hlist_node d_hash; /* list of hash table entries */
struct hlist_head *d_bucket; /* hash bucket */
unsigned char d_iname[DNAME_INLINE_LEN_MIN]; /* short name */
};

```

Ispravan objekat dentry može biti u jednom od sledećih stanja: korišćen, neiskorišćen ili negativan. Objekat dentry može biti i oslobođen, tj. nalaziti se kao objekat u kešu, kada ne postoji validna referenca na objekat dentry u bilo kom virtuelnom sistemu datoteka niti bilo gde u kôdu sistema datoteka.

Kernel kešira dentry objekat u dentry keš (dcache). Dentry keš se sastoji od tri dela:

- ▷ Liste "korišćenih" dentrija povezanih sa pridruženim inodom preko polja `i_dentry` objekta inode. Pošto dati inode može da ima više linkova, može biti više dentry objekata, pa se koristi lista.
- ▷ Dvostruko ulančane liste najmanje korišćenih (least recently used, LRU) i negativnih dentry objekata. Lista je sa umetanjem i sortirana po vremenu, tj. upisi na početku liste su noviji. Kada kernel mora da ukloni upise da bi oslobodio memoriju, upisi se uklanjaju sa liste sa kraja, jer se tu nalaze najstariji upisi za koje je najmanje verovatno da će se koristiti u bliskoj budućnosti
- ▷ Heš tabele u kojoj se funkcije heširanja koriste za brzo rešavanje date putanje u pridruženom dentry objektu. Tabele su predstavljene nizom `dentry_hashtable`. Svaki elemenat je pokazivač na listu upisa (dentries) koji heširaju istu vrednost. Veličina ovog niza zavisi od količine fizičke RAM memorije u sistemu.

### 15.6.1. Operacije sa objektom dentry

Struktura `dentry_operations` zadaje metode koje VFS poziva za stavke direkto-rijuma (dentry) u datom sistemu datoteka. Struktura `dentry_operations` je definisana u zaglavljju `<linux/dcache.h>`:

```

struct dentry_operations {
    int (*d_revalidate) (struct dentry *, int);
    int (*d_hash) (struct dentry *, struct qstr *);
    int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
    int (*d_delete) (struct dentry *);
    void (*d_release) (struct dentry *);
    void (*d_iput) (struct dentry *, struct inode *);
};

```

Metodi za rad sa objektom dentry su sledeći:

- ▷ `int d_revalidate(struct dentry *dentry, int flags)`: određuje da li je dati dentry objekat ispravan.
- ▷ `int d_hash(struct dentry *dentry, struct qstr *name)`: kreira heš vrednost za dati dentry.

- ▷ `int d_compare(struct dentry *dentry, struct qstr *name1, struct qstr *name2):` VFS poziva ovu funkciju da bi uporedio datoteke name1 i name2.

## 15.7. Objekat file

Objekat file se koristi za predstavljanje datoteke koju je otvorio proces. Procesi rade direktno sa objektom file, a ne sa superblokom, inodom ili dentrijima. Objekat file je predstava otvorene datoteke u memoriji. Objekat (ali ne fizička datoteka) se kreira kao odziv na sistemski poziv `open()`, a uništava kao rezultat sistemskog poziva `close()`. Ovi sistemski pozivi su definisani u tabeli operacija sa datotekama. Pošto više procesa mogu da otvore datoteku i rade sa njom u isto vreme, može da postoji više objekata file za istu datoteku. Objekat file predstavlja pogled procesa na otvorenu datoteku. Objekat pokazuje na dentry (koji sa svoje strane pokazuje na inode) koji stvarno predstavlja otvorenu datoteku. Objekti inode i dentry su jedinstveni. Objekat file je predstavljen strukturom `file` i definisan u zaglavlju `<linux/fs.h>`. Sledi kôd strukture sa odgovarajućim komentarima:

```
struct file {
    struct list_head f_list; /* list of file objects */
    struct dentry *f_dentry; /* associated dentry object */
    struct vfsmount *f_vfsmnt; /* associated mounted fs */
    struct file_operations *f_op; /* file operations table */
    atomic_t f_count; /* file object's usage count */
    unsigned int f_flags; /* flags specified on open */
    mode_t f_mode; /* file access mode */
    loff_t f_pos; /* file offset (file pointer) */
    struct fown_struct f_owner; /* owner data for signals */
    unsigned int f_uid; /* user's UID */
    unsigned int f_gid; /* user's GID */
    int f_error; /* error code */
    struct file_ra_state f_ra; /* read-ahead state */
    unsigned long f_version; /* version number */
    void *f_security; /* security module */
    void *private_data; /* tty driver hook */
    struct list_head f_ep_links; /* list of eventpoll links */
    spinlock_t f_ep_lock; /* eventpoll lock */
    struct address_space *f_mapping; /* page cache mapping */
};
```

Slično objektu dentry, objekat file u stvari ne odgovara bilo kom podatku na disku. Zbog toga ne postoji flag u objektu koji određuje da li je objekat izmenjen i da li treba da bude upisan na disk. Objekat file pokazuje na pridružen objekat dentry preko po-

kazivača f\_dentry. Objekat dentry pokazuje na pridruženi inode, na osnovu koga se može zaključiti da li je datoteka izmenjena (dirty).

### 15.7.1. Operacije sa objektima file

Kao i u slučaju ostalih VFS objekata, veoma je važna tabela operacija sa objektima file. Odgovarajući metodi se nalaze u strukturi `file_operations`, definisanoj u zaglavlju `<linux/fs.h>`:

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char *, size_t, loff_t);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    ssize_t (*aio_write) (struct kiocb *, const char *, size_t, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int,
                  unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int);
    int (*aio_fsync) (struct kiocb *, int);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *,
                     unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *,
                      unsigned long, loff_t *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t,
                        read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int,
                        size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)
    (struct file *, unsigned long, unsigned long,
     unsigned long, unsigned long);
    int (*check_flags) (int flags);
    int (*dir_notify) (struct file *filp, unsigned long arg);
    int (*flock) (struct file *filp, int cmd,
                  struct file_lock *fl);
};
```

Sistemi datoteka mogu da primene jedinstvene funkcije za svaku od ovih operacija, ali i da koriste generički metod ako on postoji. U nastavku je dat spisak funkcija koje se mogu primeniti na objekat file:

- ▷ `loff_t llseek(struct file *file, loff_t offset, int origin)`: ažurira pokazivač na datoteku za dati pomeraj (offset). Poziva se preko sistemskog poziva `llseek()`.
- ▷ `ssize_t read(struct file *file, char *buf, size_t count, loff_t *offset)`: učitava count bajtova iz date datoteke na poziciji offset u buf. Pokazivač na datoteku se zatim ažurira. Ovu funkciju poziva sistemski poziv `read()`.
- ▷ `ssize_t aio_read(struct kiocb *iocb, char *buf, size_t count, loff_t offset)`: počinje asinhrono čitanje count bajtova u datoteku buf opisanu u iocb. Poziva je sistemski poziv `aio_read()`.

### 15.8. Strukture podataka pridružene sistemima datoteka

Uz osnovne VFS objekte, kernel za upravljanje podacima koristi druge standardne strukture podataka. Prvi objekat se koristi za opisivanje specifičnih varijanti sistema datoteka, kao što su ext3 ili XFS. Druga struktura podataka se koristi za opisivanje montirane instance sistema datoteka. Pošto Linux podržava veliki broj sistema datoteka, neophodna je specijalna struktura u kernelu za opis karakteristika i ponašanja svakog sistema datoteka.

```
struct file_system_type {
    const char *name; /* filesystem's name */
    struct subsystem subsys; /* sysfs subsystem object */
    int fs_flags; /* filesystem type flags */
    /* the following is used to read the superblock off the disk */
    struct super_block *(*get_sb) (struct file_system_type *,
        int, char *, void *);
    /* the following is used to terminate access to the superblock */
    void (*kill_sb) (struct super_block *);
    struct module *owner; /* module owning the filesystem */
    struct file_system_type *next; /* next file_system_type in list */
    struct list_head fs_supers; /* list of superblock objects */
};
```

Struktura vfsmount je definisana u zaglavlju <linux/mount.h> sledećim kôdom:

```
struct vfsmount {
    struct list_head mnt_hash; /* hash table list */
    struct vfsmount *mnt_parent; /* parent filesystem */
    struct dentry *mnt_mountpoint; /* dentry of this mount point */
    struct dentry *mnt_root; /* dentry of root of this fs */
```

```

struct super_block *mnt_sb; /* superblock of this filesystem */
struct list_head mnt_mounts; /* list of children */
struct list_head mnt_child; /* list of children */
atomic_t mnt_count; /* usage count */
int mnt_flags; /* mount flags */
char *mnt_devname; /* device file name */
struct list_head mnt_list; /* list of descriptors */
struct list_head mnt_fslink; /* fs-specific expiry list */
struct namespace *mnt_namespace /* associated namespace */
};


```

## 15.9. Strukture podataka pridružene procesu

Svaki proces u sistemu ima svoju sopstvenu listu otvorenih datoteka, korenski direktorijum sistema datoteka, tekući radni direktorijum, tačke montiranja itd. VFS sloj i procese u sistemu povezuju tri strukture podataka: `files_struct`, `fs_struct` i `namespace`. Struktura `files_struct` je definisana u zaglavljumu `<linux/file.h>`. Sve informacije procesa su definisane sledećim kôdom:

```

struct files_struct {
    atomic_t count; /* structure's usage count */
    spinlock_t file_lock; /* lock protecting this structure */
    int max_fds; /* maximum number of file objects */
    int max_fdset; /* maximum number of file descriptors */
    int next_fd; /* next file descriptor number */
    struct file **fd; /* array of all file objects */
    fd_set *close_on_exec; /* file descriptors to close on exec() */
    fd_set *open_fds; /* pointer to open file descriptors */
    fd_set close_on_exec_init; /* initial files to close on exec() */
    fd_set open_fds_init; /* initial set of file descriptors */
    struct file *fd_array[NR_OPEN_DEFAULT];
    /* default array of file objects */
};


```

Niz `fd` pokazuje na listu otvorenih fajl objekata. Standardno je to niz `fd_array`. Druga struktura je `fs_struct` i sadrži informacije o sistemu datoteka; na nju pokazuje polje `fs` u deskriptoru procesa. Struktura je definisana u zaglavljumu `<linux/fs_struct.h>` i prikazana ovde sa odgovarajućim komentarima:

```

struct fs_struct {
    atomic_t count; /* structure usage count */
    rwlock_t lock; /* lock protecting structure */
    int umask; /* default file permissions*/
    struct dentry *root; /* dentry of the root directory */
    struct dentry *pwd; /* dentry of the current directory */


```

```
struct dentry *altroot; /* dentry of the alternative root */
struct vfsmount *rootmnt; /* mount object of the root directory */
struct vfsmount *pwdmnt; /* mount object of the current directory */
struct vfsmount *altrootmnt;
/* mount object of the alternative root */
};
```

Ova struktura drži tekući radni direktorijum (pwd) i korenski (root) direktorijum tekućeg procesa. Treća struktura je `namespace`, definisana u zaglavljumu `<linux/ns.h>` na koju pokazuje polje `namespace` u deskriptoru procesa. Ova struktura omogućuje da svaki proces ima jedinstven pogled na montirani sistem datoteka. Sledi kôd strukture sa odgovarajućim komentarima:

```
struct namespace {
    atomic_t count; /* structure usage count */
    struct vfsmount *root; /* mount object of root directory */
    struct list_head list; /* list of mount points */
    struct rw_semaphore sem; /* semaphore protecting the namespace */
};
```

Član strukture `list` određuje dvostruko ulančanu listu montiranog sistema datoteka koji čini prostor imena (`namespace`). Ova struktura podataka je povezana sa svakim deskriptorom procesa. Svi procesi dele isti prostor imena, tj. vide istu hijerarhiju sistema datoteka u istoj tabeli montiranja.

**Deo III**

## **Primeri operativnih sistema**



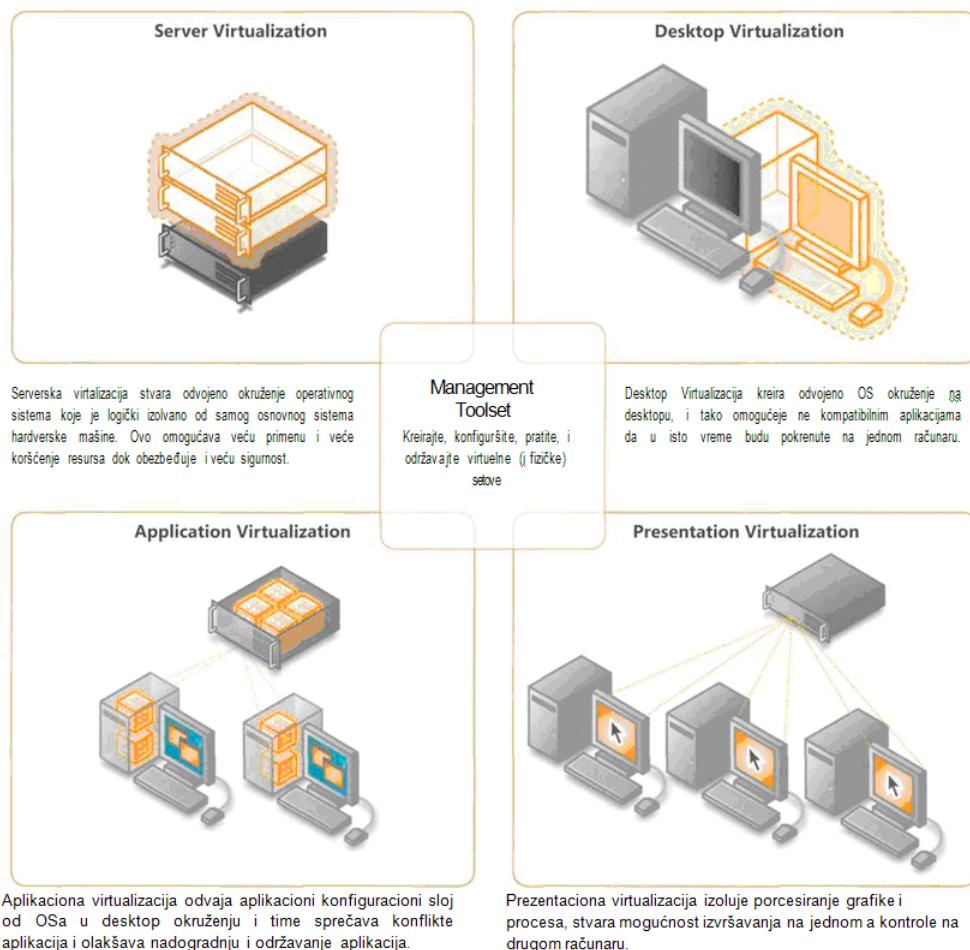
## **Instaliranje i praktična primena virtuelnih mašina**

Virtuelizacija u računarskim naukama predstavlja kreiranje virtuelne verzije hardverske platforme, operativnog sistema, mrežnih resursa itd. Virtuelizacija može biti hardverska, softverska, mrežna, aplikativna i virtuelizacija skladištenja podataka:

- ▷ virtuelizacija skladištenja podataka (storage virtualization): povezivanje više fizičkih uređaja za skladištenje u jedan virtuelni),
- ▷ mrežna virtuelizacija (network virtualization): omogućuje razdvajanje dostupnih propusnih opsega u zasebne kanale da bi mogli da se dodeljuju pojedinim resursima),
- ▷ serverska virtuelizacija (server virtualization): često se zove i hardverska virtuelizacija. Prikriva fizičke karakteristike servera, virtuelni serveri dele resurse fizičkog servera),
- ▷ desktop virtuelizacija (desktop virtualization): radne stanice su tzv. „tanki klijenti“ (thin clients) jer nemaju svoj operativni sistem već se on u potpunosti izvršava na serveru),
- ▷ aplikativna virtuelizacija (application virtualization): poslovne aplikacije se pokreću virtuelno sa radnih stanicu ali zapravo troše resurse servera).

Slika 1 ilustruje primere primene Microsoft Hyper-V virtuelizacije. Uz pomoć virtuelizacije moguće je mnogo poboljšati efikasnost iskorišćenja računarskih resursa u odnosu na mašinu bez virtuelizacije (približno kao u mainframe okruženju). Istovremeno je moguće smanjiti broj fizičkih servera za 8 do 30 puta. Usled toga znatno se smanjuje i utrošak električne energije u računarskom centru (koja se, zapravo, pretvara u toplotu) i fizički prostor potreban za instalaciju. Napredne opcije omogućavaju da servis (kako ga vide krajnji korisnici) bude mnogo raspoloživiji od hardvera na kome je instaliran. Neki servisi bez prekida rade gotovo hiljadu dana, s tim što su u međuvremenu više desetina puta prebačeni sa hosta na host. Stavljanje novog virtuelnog servera na raspolaganje IT odeljenju ili korisnicima je postupak koji traje nekoliko minuta (podrazumeva kloniranje, kreiranje na osnovu šablonu), u poređenju sa tri do četiri nedelje koliko je potrebno za nabavku i instalaciju fizičkog servera. Arhitektura rezervnog računskog centra (koji se koristi u slučaju zakazivanja) može biti vrlo kompaktna i ekonomična, tj. potpuno virtuelizovana. Štaviše, ova infrastruktura može da se deli sa drugim korisnicima, pa je jeftinija nego sopstvena. Uprošćava

se i instalacija i administracija operativnih sistema i aplikacija koje oni podržavaju (iskustvo pokazuje da se potrebni ljudski i vremenski resursi smanjuju na trećinu). Distribucija, testiranje i procena kvaliteta novog softvera koji se isporučuje u obliku virtuelnih mašina je znatno ubrzana i olakšana. Na Web lokaciji [vmware.com](http://vmware.com) može se naći preko 300 takvih mašina, a isti pristup su prihvatali i ostali isporučiocci softvera.

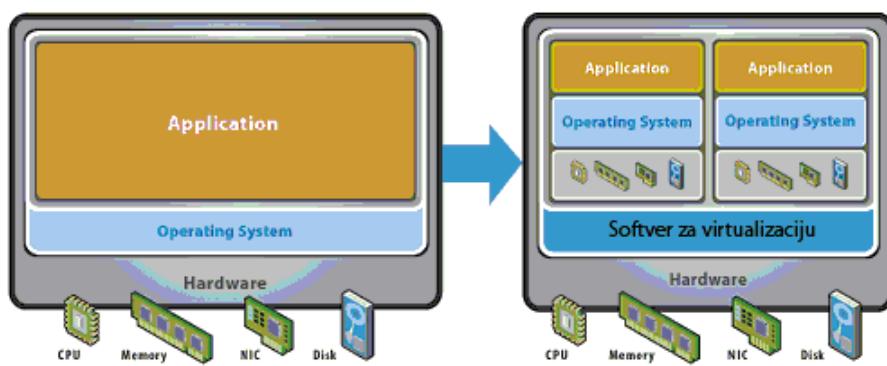


Slika 1: Mogućnosti primene tehnologije Microsoft Hyper-V

U SAD, procenat virtuelizovanog dela infrastrukture računskih centara je vrlo visok, preko 40%. IT analitičari procenjuju da za dve do tri godine neće postojati fizički server bez instaliranog softvera za virtuelizaciju, i da će to promeniti koncepciju operativnog sistema opšte namene i modele licenciranja softvera.

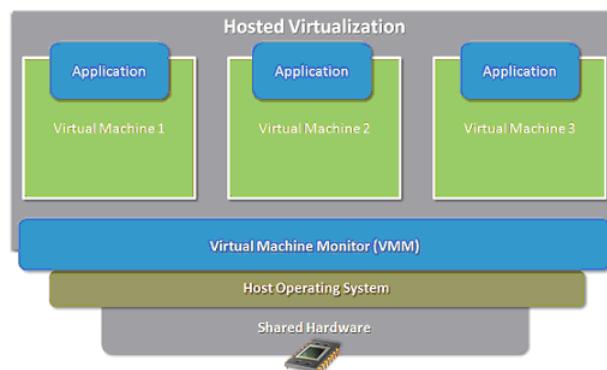
## Hardverska podrška

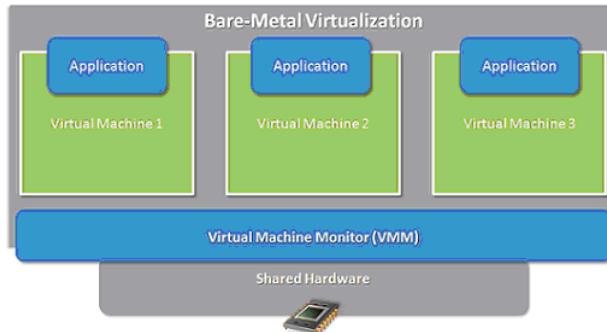
Savremena virtualizacija dozvoljava pokretanje više različitih operativnih sistema na jednom računaru. Svaka virtuelna mašina je nezavisan operativni sistem sa sopstvenim hardverom, ali svi operativni sistemi dele resurse zajedničkog (fizičkog) hardvera. Kontrolu hardvera sprovodi softver nazvan virtuelna mašina (VM), koji kontroliše pristupe procesoru, memoriji, ulazno-izlaznim uređajima, diskovima i mrežnom hardveru.



Slika 2: Prikaz prebacivanja fizičkog računara u virtuelni.

Da bi se na računaru kreirale virtuelne particije, sistemu se dodaje tanki softverski sloj nazvan Virtual Machine Monitor (VMM). Ovaj sloj je odgovoran za upravljanje hardverskim resursima i razrešavanje zahteva operativnog sistema, odnosno aplikacija koje su pod njim pokrenute.





Slika 3: Dva najvažnija modela virtuelizacije: virtuelizacija zasnovana na host operativnom sistemu (gore) i direktna virtuelizacija na sistemu softvera za virtuelizaciju (dole).

VMM je virtuelni skup procesora, memorije, diska i mreže za svakog korisnika. Osnovne funkcije koje ovaj softver obavlja su emulacija hardvera prema operativnim sistemima korisnika, izolovanje rada pojedinačnih virtuelnih mašina, alokacija resursa potrebnih svakoj virtuelnoj mašini, uz održavanje balansa među zahtevima. U 2010. godini 23% aplikacija radilo je na virtuelnim mašinama, a prognoza je da će do 2012. godine ta brojka porasti na 48%. Tržištem virtuelnih mašina dominira proizvođač VMware. Pretpostavka je da na tržištu trenutno postoji 10,8 miliona virtuelnih mašina, a da je tržišni udeo firme VMware 84%, Microsofta 11%, Citrixa 4%, ostalih proizvođača 1%. Najpoznatiji programi za virtuelizaciju su:

- ▷ VMware Player, Server, Workstation, ESX i ESXi;
- ▷ Microsoft Hyper-V;
- ▷ Oracle (Sun) VirtualBox;
- ▷ Citrix XenServer
- ▷ Parallels Desktop

## **Operativni sistemi sa komandnom linijom**

Operativne sisteme sa komandnom linijom (CLI, Command Line Interface) predstavljamo pomoću virtuelnog računara zasnovanog na operativnom sistemu MS-DOS. Ovaj operativni sistem proizvod je kompanije Microsoft, razvijen je 1981. godine, i u gotovo neizmenjenom obliku zadržao se do kraja devedesetih godina prošlog veka. U osnovi, radi se o operativnom sistemu koji radi sa diskovima i isključivo sa tekstualnim interfejsom (tastaturom).

Komandna linija je linija u kojoj se kucaju komande (naredbe). Komandni prompt pokazuje da se nalazite u komandnoj liniji. Prompt može biti slovo za logičku oznaku disk jedinice, praćeno obrnutom kosom crtom (backslash), npr. C:\ ili A:\ i nazivom direktorijuma (na primer, C:\dos). Slovo pokazuje koja je disk jedinica aktivna. MS-DOS pretražuje aktivnu disk jedinicu da bi pronašao informaciju koja mu je potrebna za izvršavanje komande.

Osim komandne linije, za rad sa većinom MS-DOS komandi može se koristiti i MS-DOS Shell. To je prozor koji omogućuje vizuelni rad sa MS-DOS-om i prikazuje disk jedinice, direktorijume, datoteke i programe koji su na raspolaganju. Komande u MS-DOS Shelli navedene su u tzv. menijima; nazivi ovih menija smešteni su duž vrha ekrana. U MS-DOS Shelli komande se biraju u meniju, uz pomoć tastature ili miša. Treba napomenuti da se ne mogu sve MS-DOS komande koristiti iz MS-DOS Shella; neke komande moraju se kucati u komandnoj liniji.

Sve verzije sa nazivom PC-DOS Microsoft je razvio za IBM. Početne i kranje verzije DOS-a su:

- ▷ PC-DOS 1.0 - avgust, 1981 - s prvim IBM PC računarom,
- ▷ MS-DOS 1.25 - januar, 1982 - za prve kompatibilne IBM računare,
- ▷ MS-DOS 6.22 - jun, 1994 - poslednja samostalna verzija,
- ▷ PC-DOS 7.0 - april, 1995,
- ▷ Windows 95/DOS 7.0 - avgust, 1995 - prva nesamostalna verzija,
- ▷ Windows 95 OSR2/DOS 7.1 - avgust, 1997 - dodata podrška za sistem datoteka FAT32



Slika 4: Izgled ekrana sa komandnom linijom u operativnom sistemu Windows 7.

### DOS - Disk Operating System

DOS je skraćeni naziv za Disk Operating System. To ime ne znači da je njegova uloga vezana isključivo za rad sa diskom. Ime je ostalo kao istorijsko nasleđe.

U nastavku je dat pregled najčešće korišćenih komandi DOS-a. Treba znati da DOS ne razlikuje mala i velika slova (tj. nije case sensitive).

Komanda	Opis	Primer
<b>TREE</b>	Komanda za pregled svih direktorijuma	C:\tree C: Argument C: je oznaka diska čiji spisak direktorijuma sa svim poddirektorijumima se traži.
<b>MKDIR [MD]</b>	Komanda za kreiranje poddirektorijuma na disku. MD je skraćeni oblik komande.	md C:\prvi\drugi\ Argument C: je oznaka diska na kome se kreira poddirektorijum.
<b>CHDIR [CD]</b>	Komanda za promenu aktivnog direktorijuma. CD je skraćeni oblik komande koji se češće koristi.	cd c:\prvi\drugi Argument C: je oznaka diska na koji se komanda odnosi.

Komanda	Opis	Primer
<b>RMDIR [RD]</b>	Komanda za brisanje (isključivo praznih) poddirektorijuma. RD je skraćeni oblik komande koji se češće koristi.	rd \prvi\drugi rd primer U prvom primeru biće izbrisani direktorijum drugi koji se nalazi u direktorijumu prvi, a u drugom primeru biće izbrisani direktorijum primer.
<b>PATH</b>	Komanda za zadavanje putanje na kojoj DOS traži korisničke programe, spoljašnje komande i batch datoteke ukoliko se oni ne nalaze u aktivnom poddirektorijumu.	path C:\; C:\prvi; A:\; C:\prvi\drugi\drugi1 U ovom primeru DOS prvo traži komandu ili program u osnovnom direktorijumu diska C; ako je tu ne nađe traži je u poddirektorijumu prvi, zatim u osnovnom direktorijumu na disketu i na kraju na disku u poddirektorijumu drugi.
<b>DATE</b>	Komanda za postavljanje datuma. Argumenti su mm (mesec), dd (dan u mesecu) i yy (dvocifreni završetak godine).	date Current date is Tue 1-01-1980 Enter new date (mm-dd-yy):
<b>TIME</b>	Komanda za postavljanje vremena. Argumenti su sati, minuti, sekunde i stotinke.	time Current time is Tue 00:00:36,93 Enter new time:
<b>PROMPT</b>	Komanda za promenu odzivnog znaka (prompt) DOS-a.	prompt \$t\$d\$n\$g Argumenti navedeni posle reči prompt predstavljaju niz proizvoljnih znakova. Pre svakog specijalnog znaka obavezno se dodaje \$. Oznake i značenja su: t - trenutno vreme; d - važeći datum; n - aktivna disk jedinica; p - aktivni poddirektorijum; v - verzija DOS-a; g - matematički znak za veće itd.

Komanda	Opis	Primer
<b>VER</b>	Komanda za ispisivanje verzije DOS-a.	ver MS-DOS Version 6.20
<b>CLS</b>	Komanda za brisanje sadržine ekrana.	
<b>DIR</b>	Komanda za prikaz sadržaja diska, uključujući i dodatne informacije o datotekama poput veličine, datuma kreiranja i sl.	dir/p Argument /p omogućava prikaz sadržaja po stranama.
<b>COPY</b>	Komanda za kopiranje sa diska na disk, i u okviru istog diska.	copy text.bak text.txt Kopiranje u okviru istog diska u kom se datoteka text.bak kopira u datoteku text.txt u istom direktorijumu. copy a:.* c: Sve datoteke sa disketne jedinice a kopiraju se na disk c.
<b>RENAME [REN]</b>	Komanda koja menja ime datoteke.	ren text.bak text1.bak Naziv datoteke TEXT se menja u TEXT1
<b>DELETE [DEL]</b>	Komanda za brisanje datoteka.	del text.bak U ovom primeru datoteka TEXT će biti obrisana. Dozvoljena je upotreba džoker znakova. del *.bak Biće obrisane sve datoteke sa ekstenzionom BAK.

**DŽOKER ZNACI:** Džoker je znak koji može da zameni jedan ili više znakova. Džoker znaci se koriste uglavnom pri navođenju imena datoteka, ali samo kada se te datoteke nalaze kao argumenti unutar nekih komandi. Znak pitanja ? zamenjuje jedan znak. Drugi džoker znak je zvezdica \* i ona zamenjuje proizvoljan broj znakova.

\*.exe – sve datoteke tipa EXE

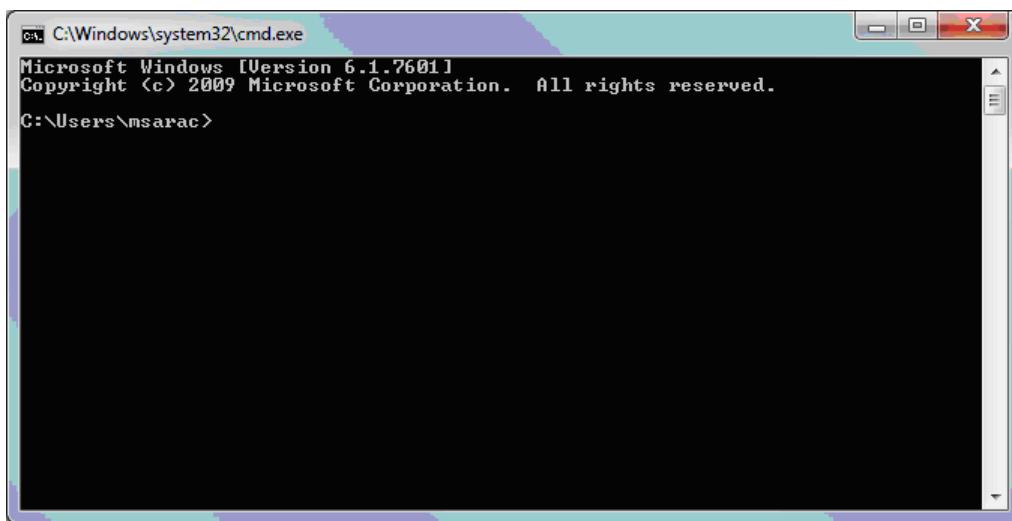
\*.bat – sve datoteke tipa BAT

\*.\* - sve datoteke

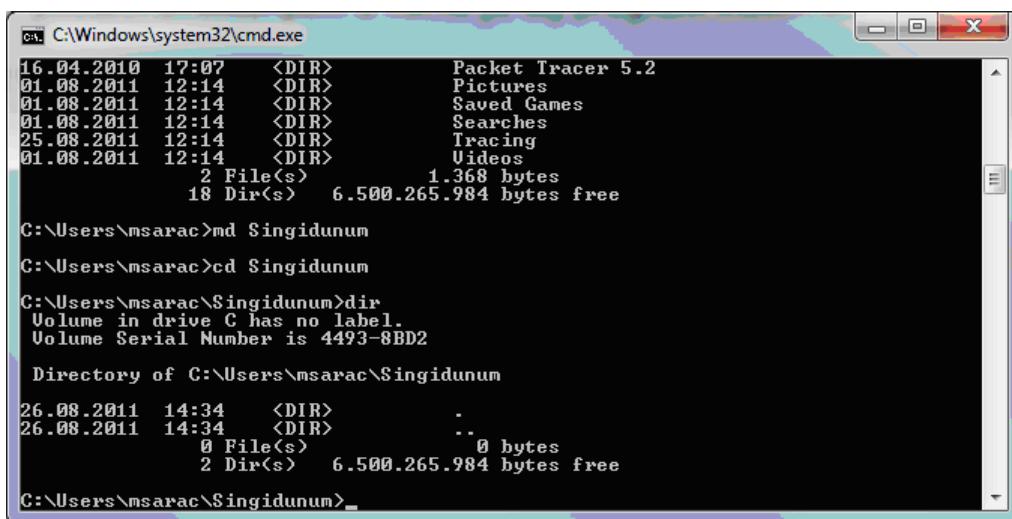
???.com - datoteke tipa COM čiji naziv ima 1, 2, ili 3 znaka

**Primeri DOS komandi u CLI interfejsu Microsoft Windowsa 7**

Da bi pokrenuli interfejs iz komandne linije (CLI), u polje za pretraživanje operativnog sistema Windows 7 unećemo komandu CMD i pritisnuti taster Enter. Pojaviće se ekran sa slike:



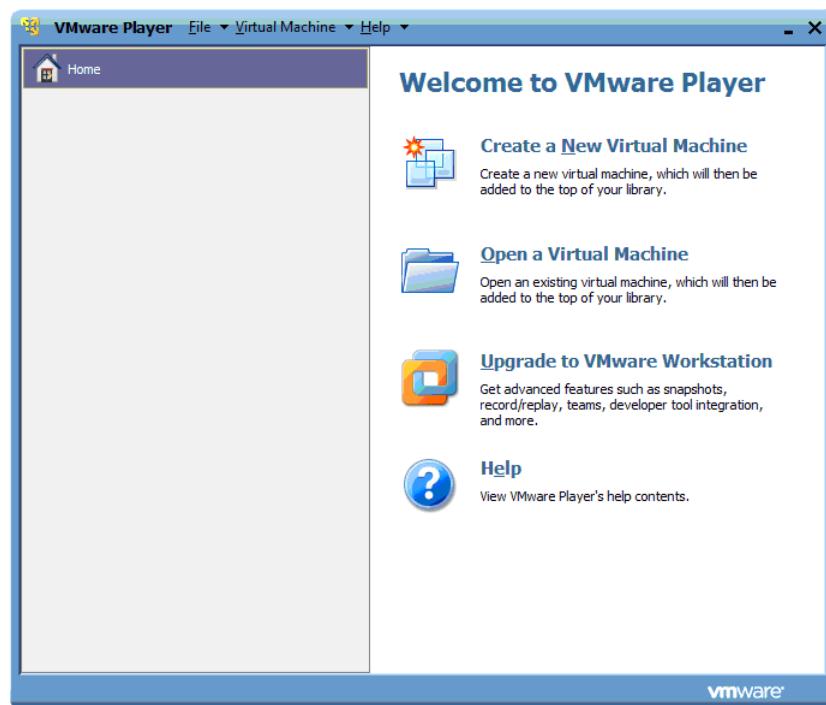
Dalje korišćenje nastavljamo samo pomoću tastature i komandi koje smo već opisali u prethodnom tekstu.



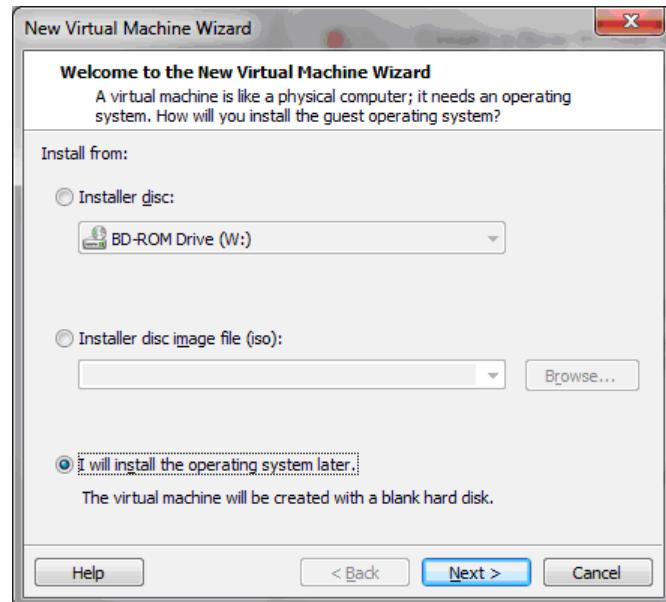
Komanda `dir` omogućiće nam prikaz sadržaja svih direktorijuma i datoteka u sklopu direktorijuma u kome se trenutno nalazimo. Za potrebe primera kreirali smo direktorijum sa imenom Singidunum pomoću komande `md Singidunum`. Nakon toga smo trenutno aktivni direktorijum promenili u direktorijum `Singidunum` pomoću komande `cd Singidunum`. Zatim smo prikazali sadržaj direktorijuma Singidunum korišćenjem komande `dir` i videli da se radi o praznom direktorijumu koji je upravo kreiran. Neke naredbe i sistemske komande se izvršavaju isključivo pomoću komandnog interfejsa.

## Instaliranje virtuelnog operativnog sistema Windows XP

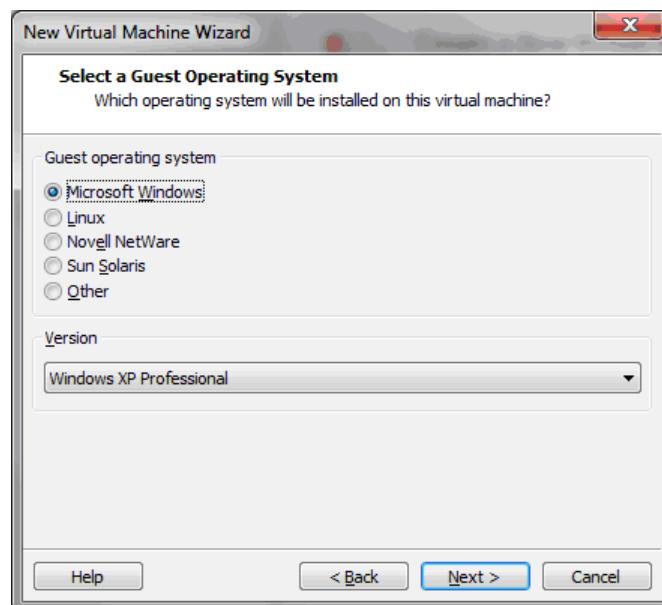
Opisaćemo korake potrebne za instalaciju virtuelnog računara zasnovanog na operativnom sistemu Windows XP.



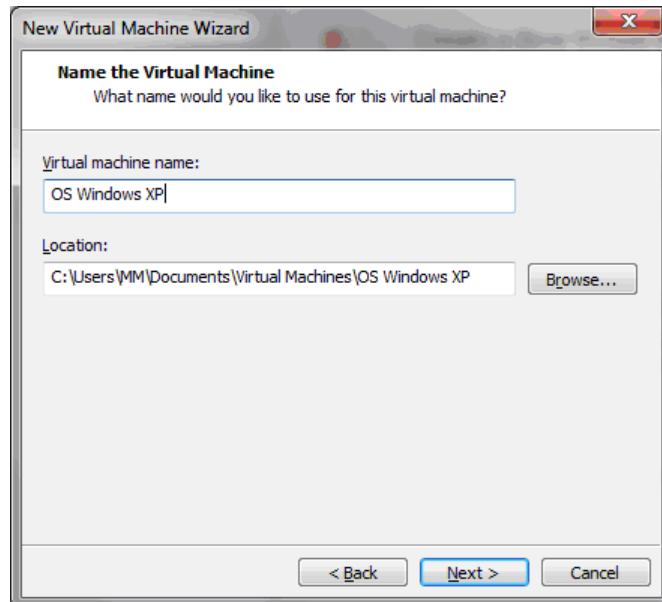
Slika 5: Osnovni meni programa VMware Player.



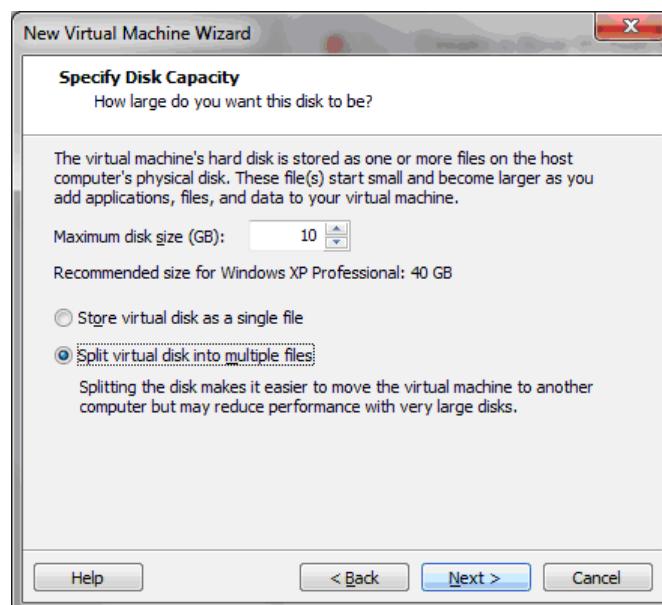
Slika 6: Meni koji se dobija izborom opcije Create a New Virtual Machine.



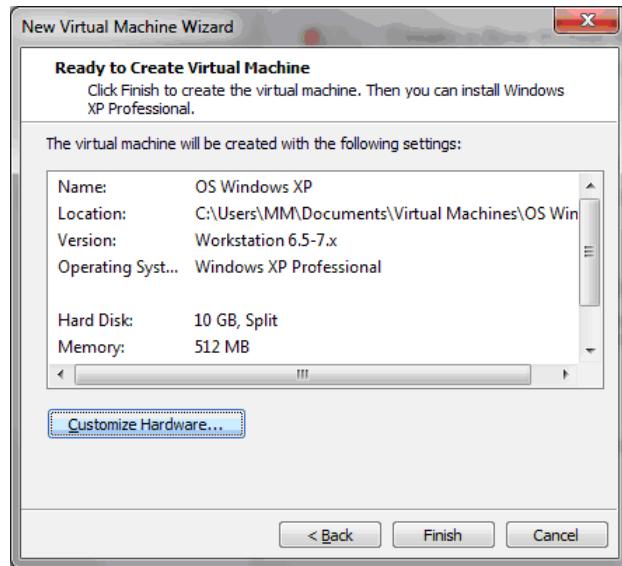
Slika 7: Meni za izbor "gostujućeg" operativnog sistema.



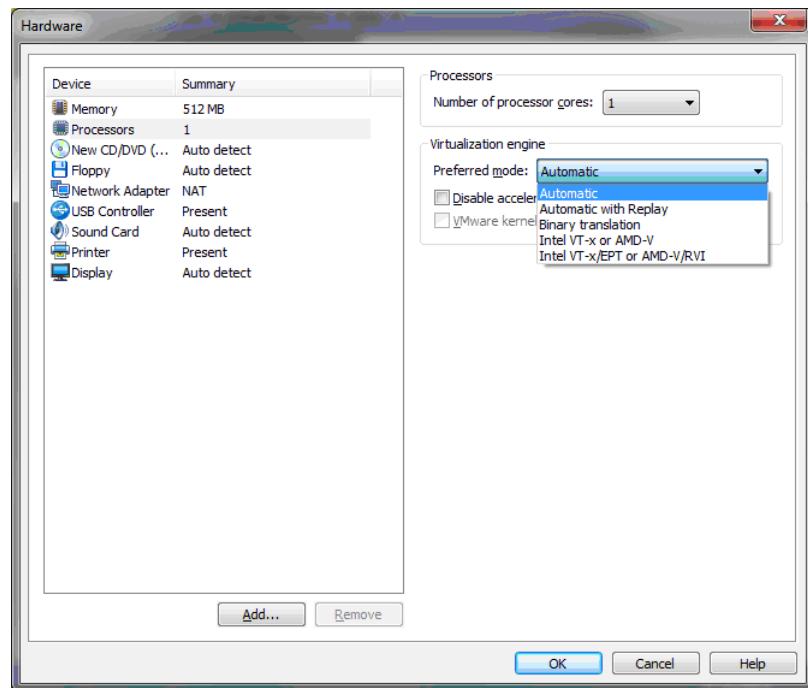
Slika 8: Osnovni podaci o virtuelnoj mašini.



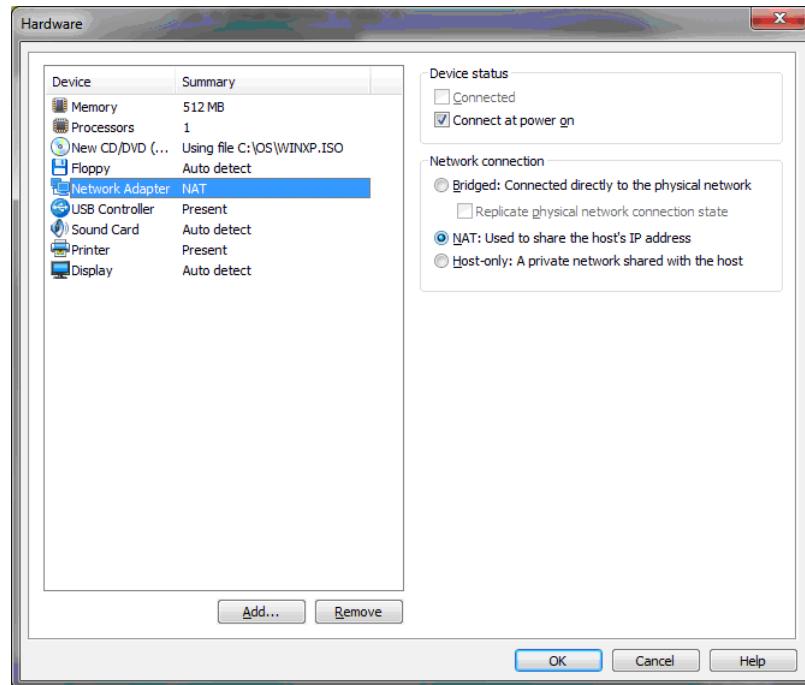
Slika 9: Izbor veličine diska i načina skladištenja.



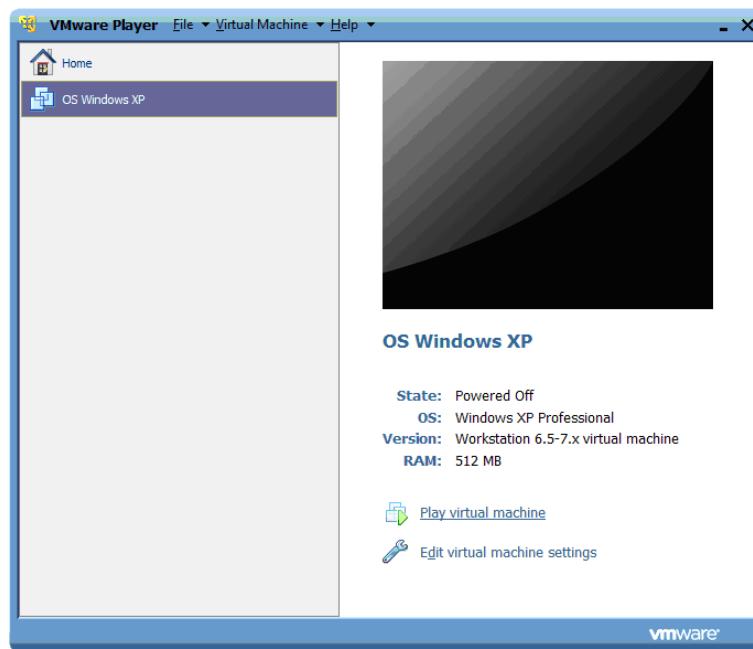
Slika 10: Prikaz izabranih opcija.



Slika 11: Podešavanje procesora i tipa virtuelizacije.

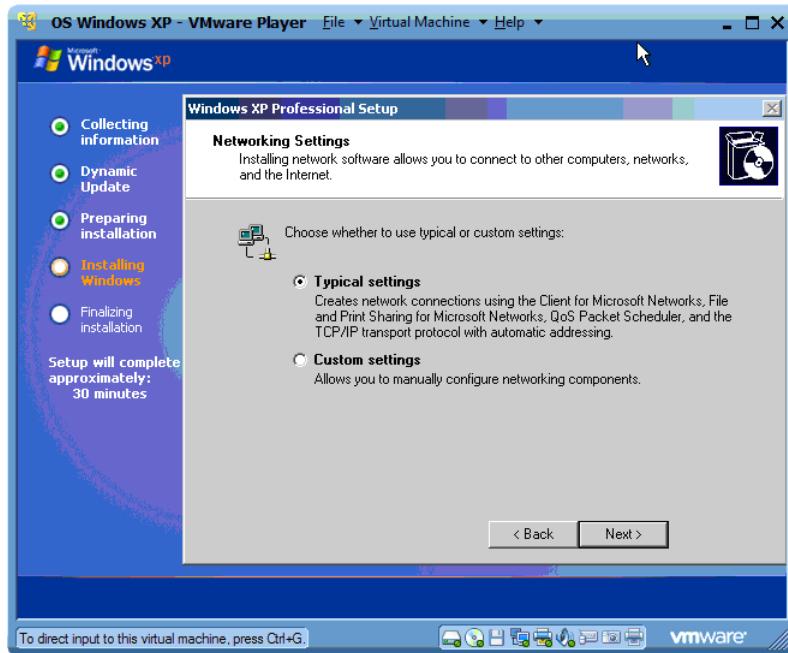


Slika 12: Podešavanje mrežne kartice i tipa povezivanja.

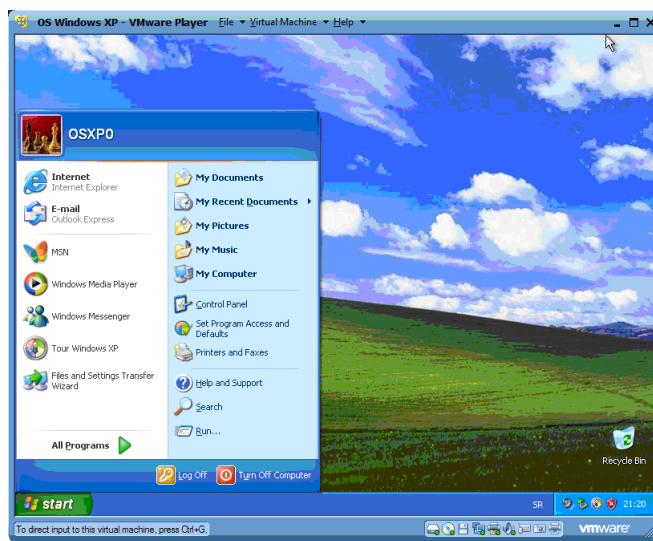


Slika 13: Ekran za pokretanje virtuelne mašine.

Nakon toga sledi standardna procedura instaliranja operativnog sistema Windows XP sa odgovarajućim podešavanjima. Dalja instalacija teče automatizovano, a zavisno od hardvera računara traje od 10 do 30 minuta.



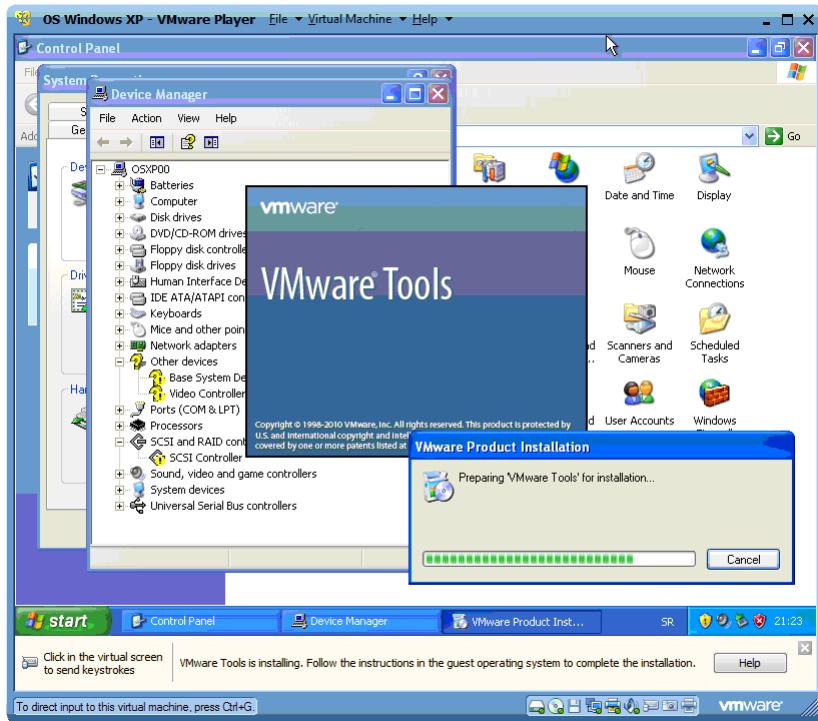
Slika 14: Izbor dodatnih mrežnih podešavanja, izbor radne grupe ili domena.



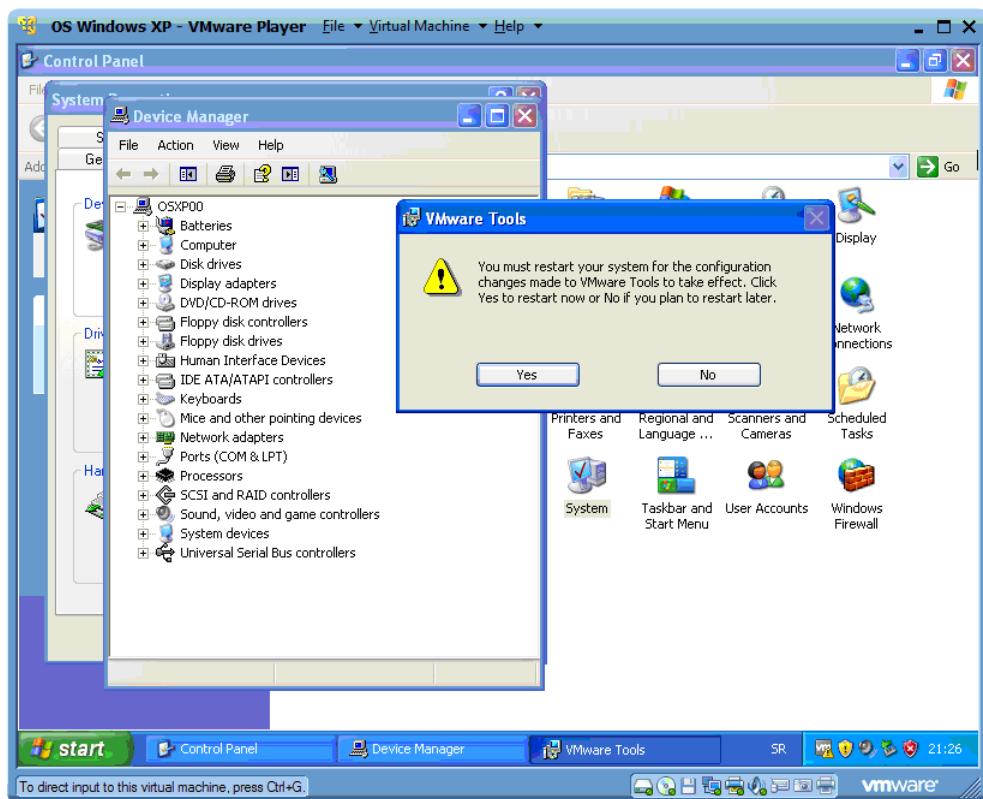
Slika 15: Ekran nakon prvog prijavljivanja.

Kao i kod stvarnog računara, i ovde je potrebno instalirati upravljačke programe (drajvere) da bi računar radio brže i stabilnije. VM drajveri nude i neke dodatne mogućnosti za sam VM računar. Do instalacije drajvera stiže se preko Virtual Machine i poslednje opcije, Install VMware Tools. Nakon toga pokreće se instalacija koja je pričinio automatizovana, sa svega nekoliko ekrana koji nude mogućnost podešavanja.

Dok virtuelni računar radi, treba koristiti opcije samog VMware Playera. Izaberite opciju Virtual Machine u gornjem delu ekrana, a zatim Virtual Machine Settings. Videćete da ima dosta opcija koje ne mogu da se promene dok VM radi. Kada se završi rad virtuelne mašine, sve opcije ponovo postaju dostupne. Opisali smo postupak instalacije VM računara pod operativnim sistemom Windows XP. Windows XP se daje ponaša kao da je instaliran na fizičkom računaru sa svojstvima koja smo dodelili ovom virtuelnom računaru. Zvanične, besplatne i gotove virtuelne računare možete preuzeti sa adresi <http://www.vmware.com/appliances/directory/>



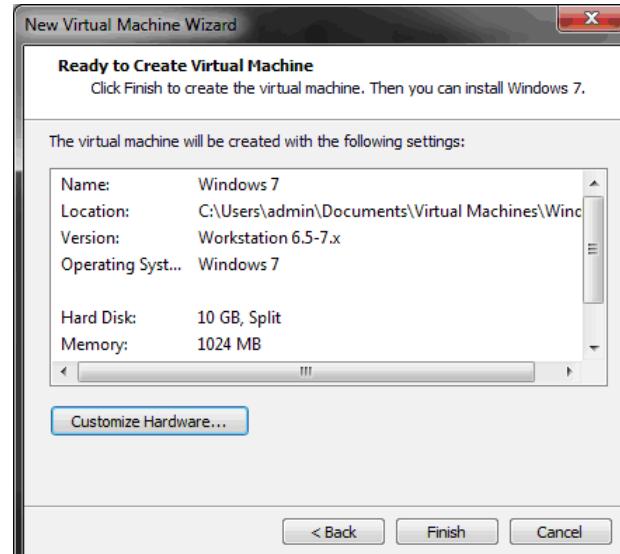
Slika 16: Instalacija VMware drajvera protiče kao i instalacija svakog drugog programa.



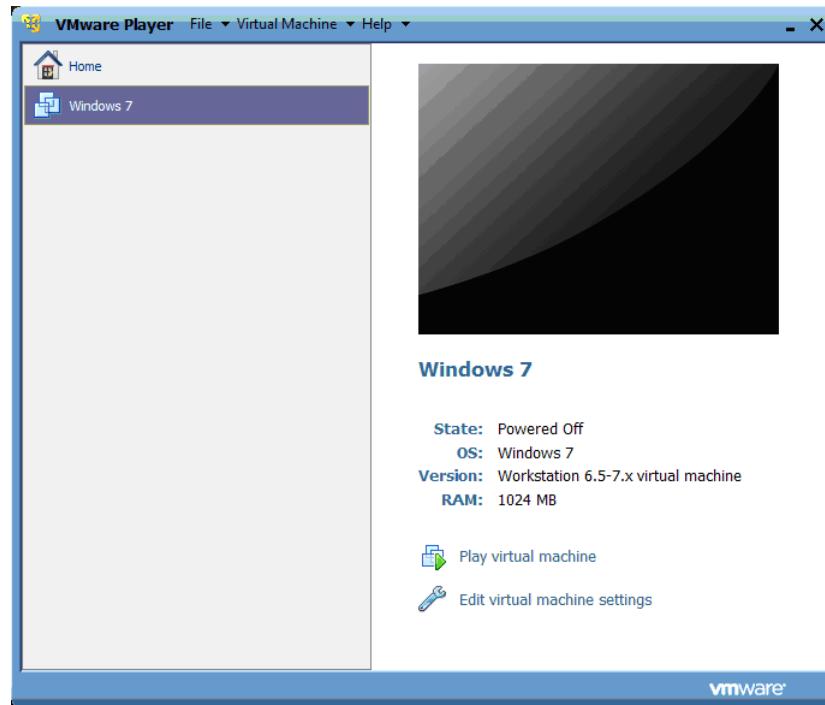
Slika 17: Nakon instaliranih drajvera sledi restart računara i nastavak rada.

## Instaliranje virtuelnog operativnog sistema Windows 7

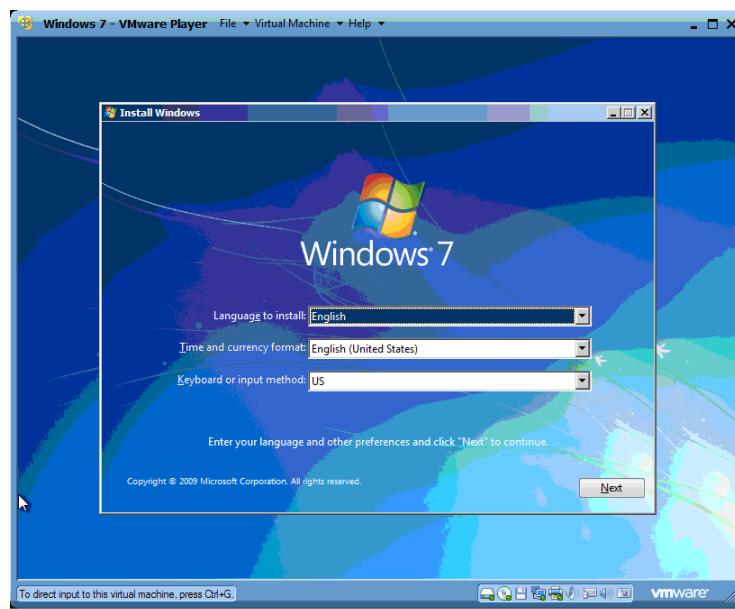
Prvih nekoliko koraka instalacije virtuelnog računara zasnovanog na operativnom sistemu Windows 7 liće na instalaciju računara zasnovanog na operativnom sistemu Windows XP, pa iz tog razloga preskačemo uvodne ekrane i nastavljamo od dela koji se razlikuje. Osnovne razlike u podešavanju virtuelne mašine odnose se na količinu prostora na disku koja je potrebna operativnom sistemu Windows 7, kao i na količinu radne memorije koja u ovom slučaju mora biti najmanje 1024 MB. Ostala podešavanja u većini slučajeva mogu ostati ista kao i kod podešavanja virtuelne mašine za instalaciju operativnog sistema Windows XP.



Slika 18: Izgled ekrana pred sam završetak instaliranja (opcija Finish).

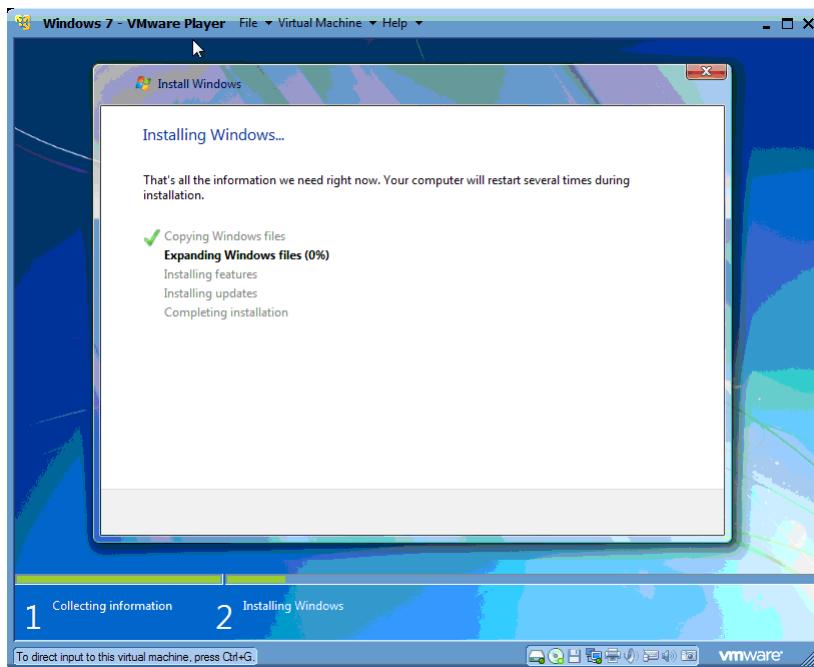


Slika 19: Ekran za pokretanje virtuelne mašine.



Slika 20: Početak instalacije ili popravke operativnog sistema Windows 7.

Sledeći ekran predstavlja opis licence MICROSOFT SOFTWARE LICENCE TERMS. U ovom ekranu treba potvrditi opciju „I accept the license terms“, što znači da ste saglasni sa pravilima i time nastavljate instalaciju. Na sledećem ekranu treba izabrati tip instalacije. Ako postoji prethodna verzija Windowsa, može se odabratи opcija “Upgrade” koja će sačuvati postojeće datoteke na disku. Druga mogućnost je “Custom (advanced)”, čime se opredeljujete za potpuno novu instalaciju.

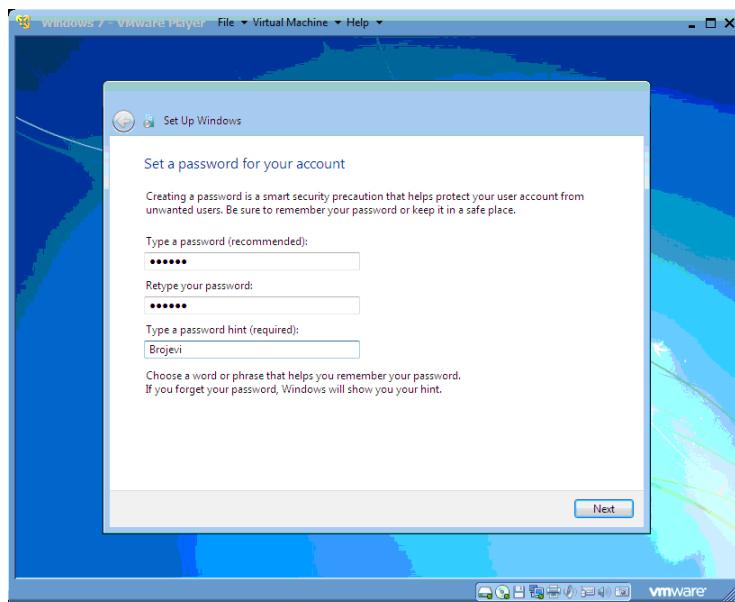


Slika 21: Kopiranje instalacionih datoteka i instalacija.

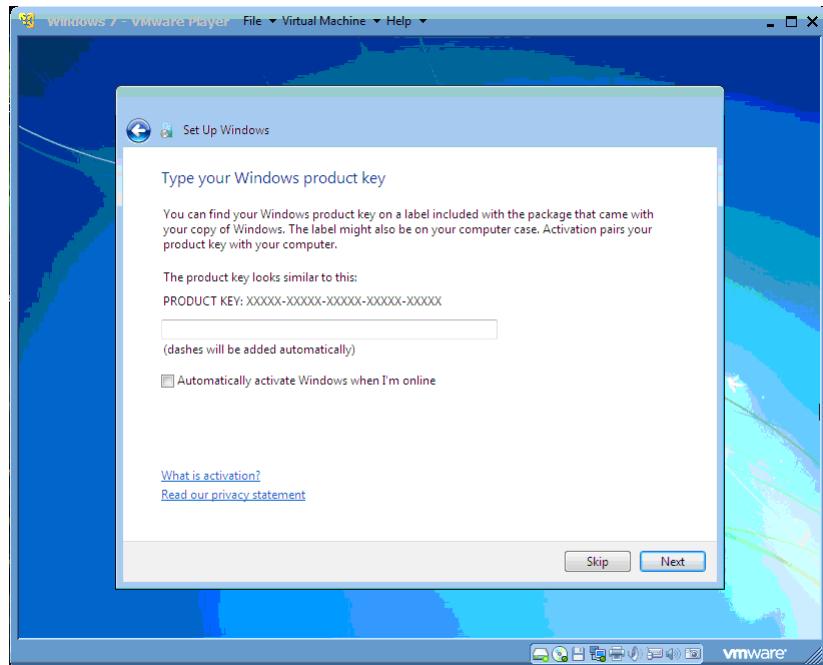
Sledeće podešavanje je izbor diska na koji želite da instalirate operativni sistem. Možete da birate disk, kreiranje particije ili (ako postoji RAID kontroler) da instalirate drajver da bi disk postao vidljiv za Windows. Formatiranje sistema datoteka je NTFS (New Technology File System); prilikom formatiranja kreira se zasebna particija od 100MB pod nazivom „System Reserved“ na kojoj se nalaze sistemske datoteke za slučaj pada sistema ili nekog oštećenja.



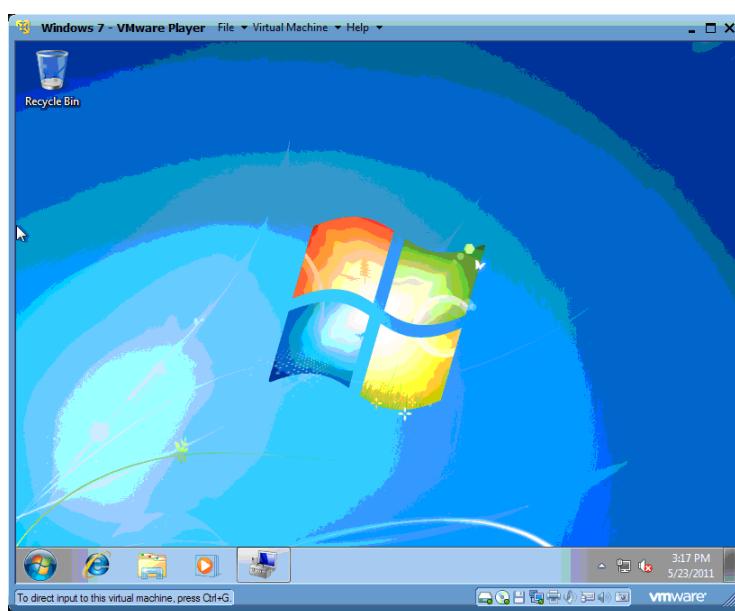
Slika 22: Prvi meni koji će nas dočekati je kreiranje novog naloga (korisnika) sa administratorskim privilegijama i dodeljivanje imena računaru.



Slika 23: Kreiranje administratorske lozinke i podsetnika.



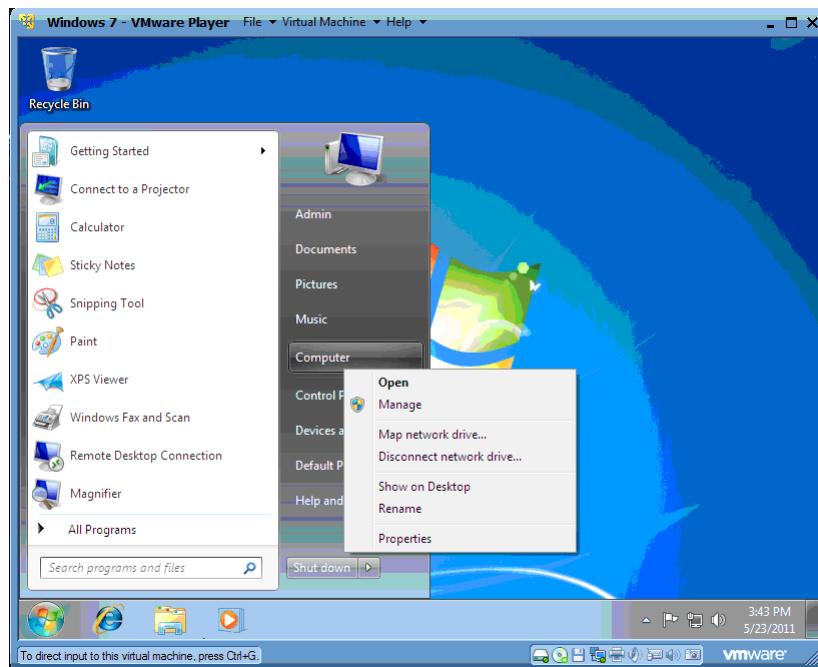
Slika 24: Unošenje ključa; prednost Windowsa 7 u odnosu na Windows XP je što postoji period od 30 dana korišćenja pre aktivacije.



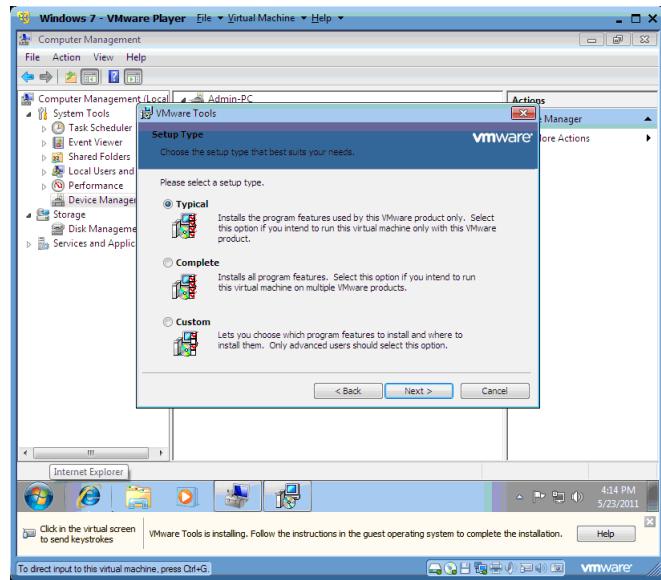
Slika 25: Ekran nakon prvog prijavljivanja.

Prednost Windowsa 7 u odnosu na ranije verzije je što sa Interneta preuzima najnovije verzije drajvera, ali je ipak potrebno instalirati neke drajvere koje Windows nema. VM drajveri nude i dodatne mogućnosti za sam VM računar. Da bi se instalirali drajveri, otvara se Virtual Machine i bira se poslednja opcija Install VMware Tools. Nakon toga počinje instalacija koja je prilično automatizovana, sa svega nekoliko ekrana koji omogućuju podešavanja.

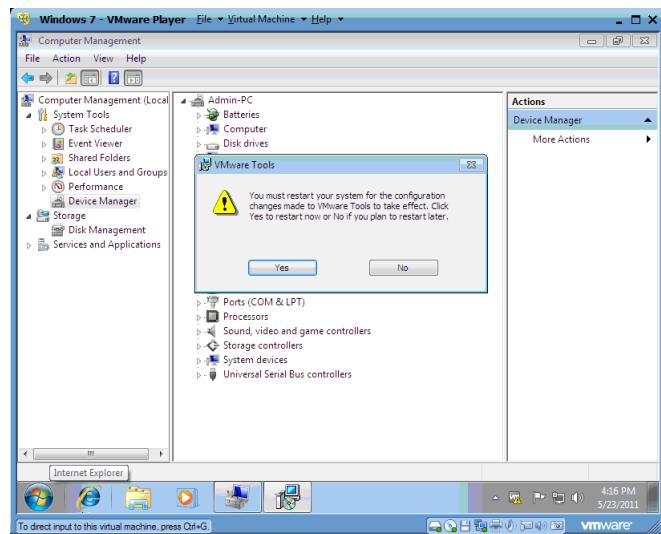
Dok virtualni računar radi korisno je isprobati opcije samog VMware Playera. Izaberite opciju Virtual Machine u gornjem delu ekrana, a zatim Virtual Machine Settings. Postoji mnogo opcija koje se ne mogu menjati dok VM radi; kada se završi, sve opcije ponovo postaju dostupne. Sam operativni sistem Windows 7 ponaša se kao da je instaliran na fizičkom računaru sa svojstvima dodeljenim virtuelnom računaru. Zvanične, besplatne gotove virtuelne računare možete preuzeti sa adrese <http://www.vmware.com/appliances/directory/>



Slika 26: Desnim pritiskom miša na Computer preko opcije Manage stižemo do podataka o računaru.



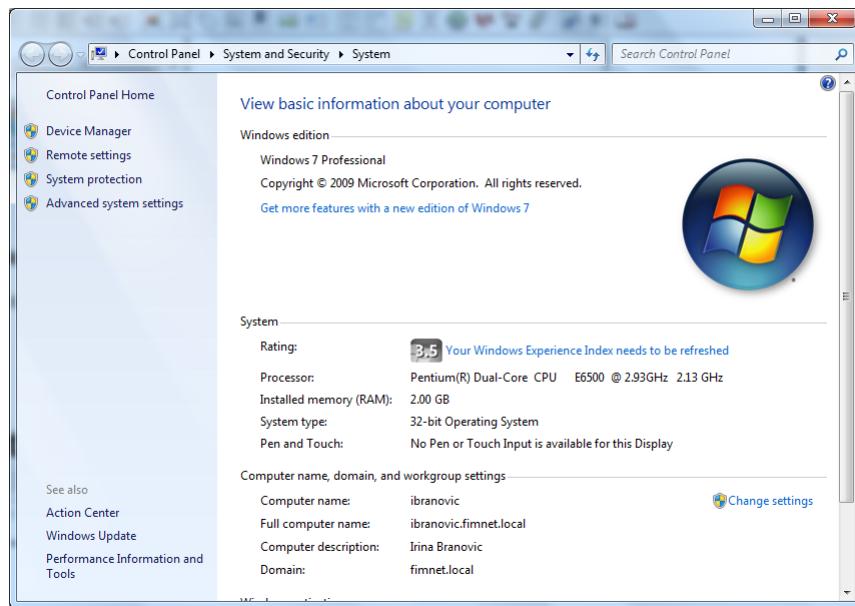
Slika 27: Instalacija VMware drajvera protiče kao i instalacija svakog drugog drajvera/programa.



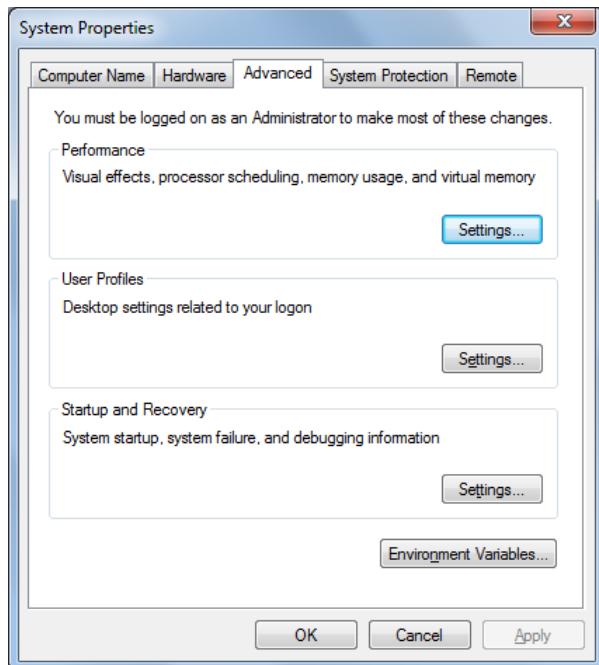
Slika 28: Nakon instaliranih drajvera, treba restartovati računar.

## Opcije za administriranje operativnog sistema Windows 7

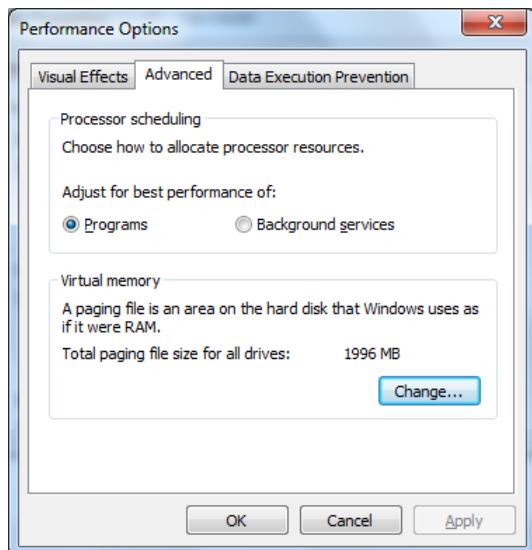
Biranjem opcije Start i desnim pritiskom opcije Computer, zatim Properties, pojavljuje se ekran sa slikom:



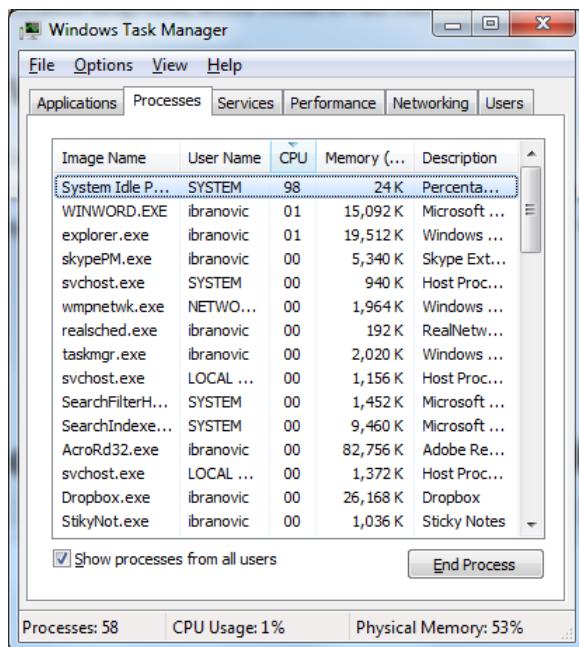
Biranjem opcije Advanced System Settings pojavljuje se dijalog System Properties sa osnovnim informacijama o sistemu:



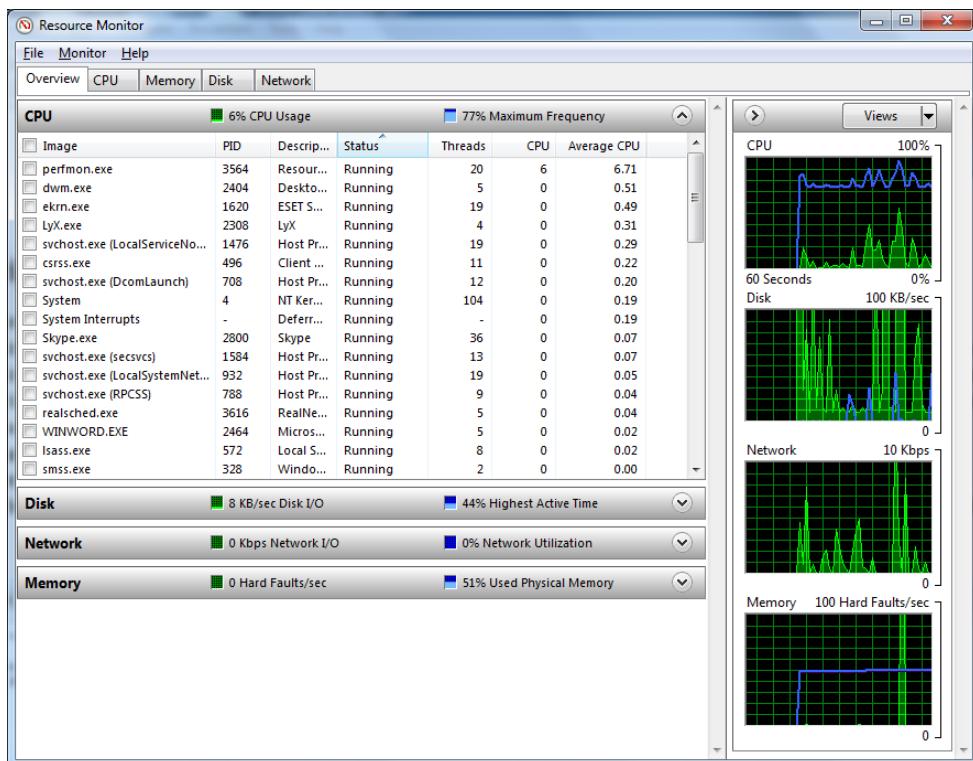
Biranjem kartice Advanced, Performance i pritiskom na dugme Settings pojavljuje se dijalog u kome je moguće menjati veličinu virtuelne memorije.



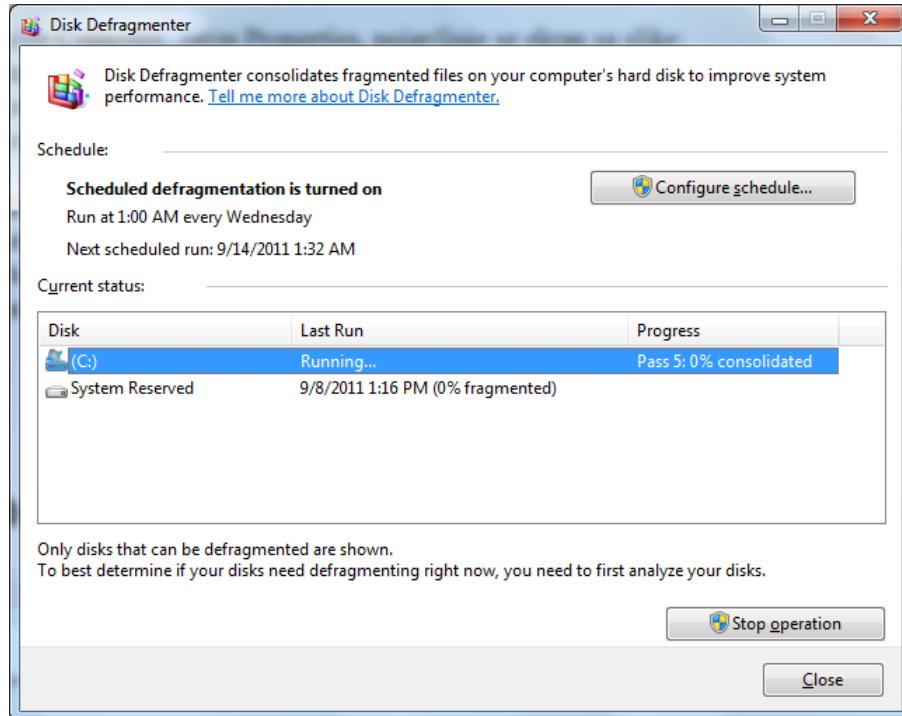
Pritiskom na tastere Ctrl+Alt+Delete pojavljuje se Task Manager u kome se mogu videti svi procesi u sistemu, količina iskorišćene memorije i opterećenost procesora. U ovom prozoru procesi se mogu prinudno zaustavljati pritiskom na dugme End Process u donjem desnom uglu dijaloga. Takođe, procesi se mogu sortirati na osnovu korišćenja procesora i memorije.



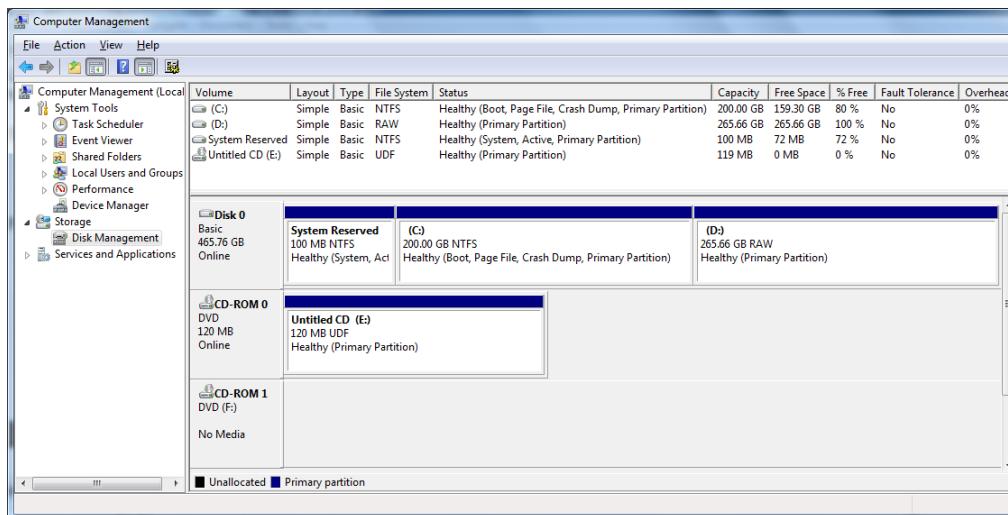
Kartica Performance u Task Manageru prikazuje vremenski dijagram opterećenosti procesora i korišćenja RAM memorije. Detaljniji prikaz svih procesa, niti, diskova, memorije i mreže u sistemu može se dobiti izborom opcije Start > All Programs > Accessories > System Tools > Resource Monitor:



Izborom opcije Start > All Programs > Accessories > System Tools > Disk Defragmenter pokreće se alatka za defragmentiranje diska:



Start > Control Panel > System and Security > Administrative Tools > Computer Management > Disk Management otvara dijalog za prikaz sistema datoteka i particija diskova:

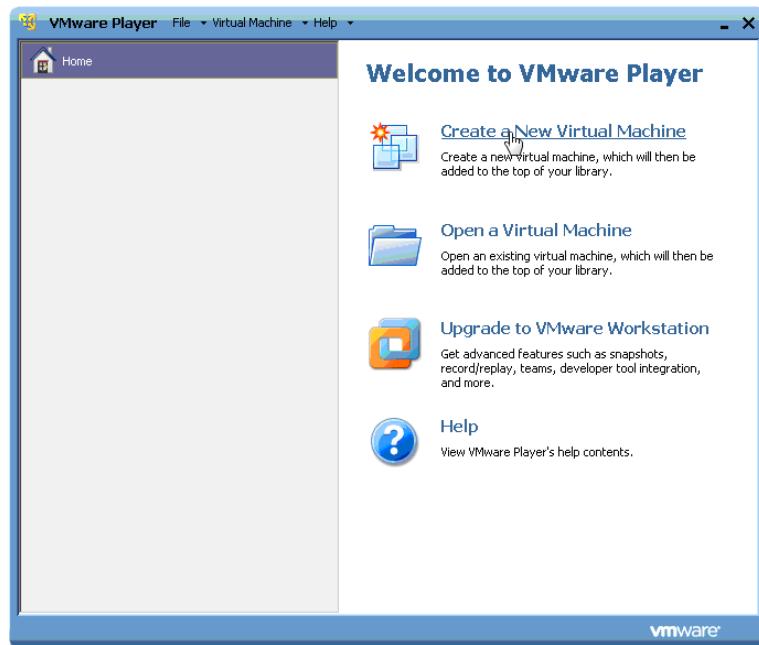


U ovom prozoru vidimo da je disk C: formatiran u sistemu NTFS, kao i da postoji mala primarna particija D: koja nije formatirana (RAW).

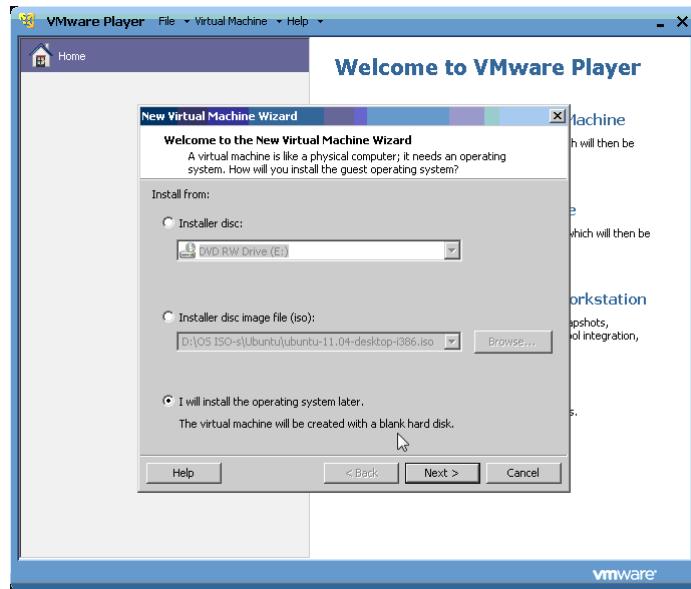
Čitaocima se preporučuje da za vežbu pokušaju da pronađu odgovarajuće opcije za upravljanje virtuelnom memorijom, procesima i diskovima u svim ostalim operativnim sistemima opisanim u ovom dodatku.

# Instaliranje virtuelnog operativnog sistema Ubuntu 11.04

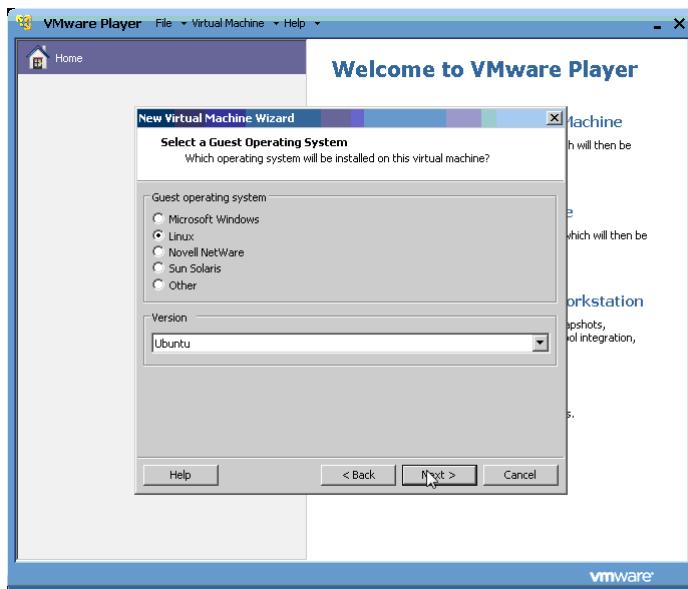
Nakon pokretanja VMWare Playera prikazuje se osnovni ekran programa.



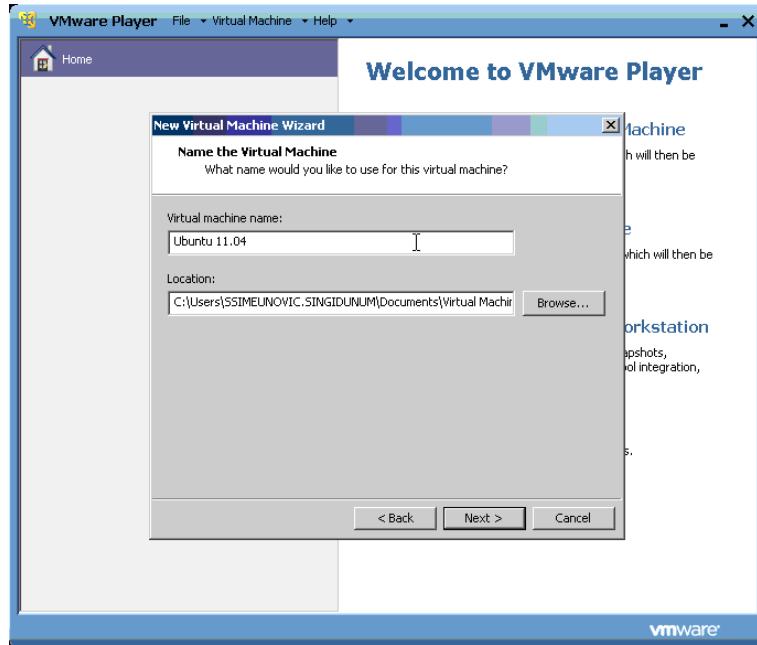
Slika 29: Da bismo kreirali novu virtuelnu mašinu, treba odabratи opciju "Create a New Virtual Machine".



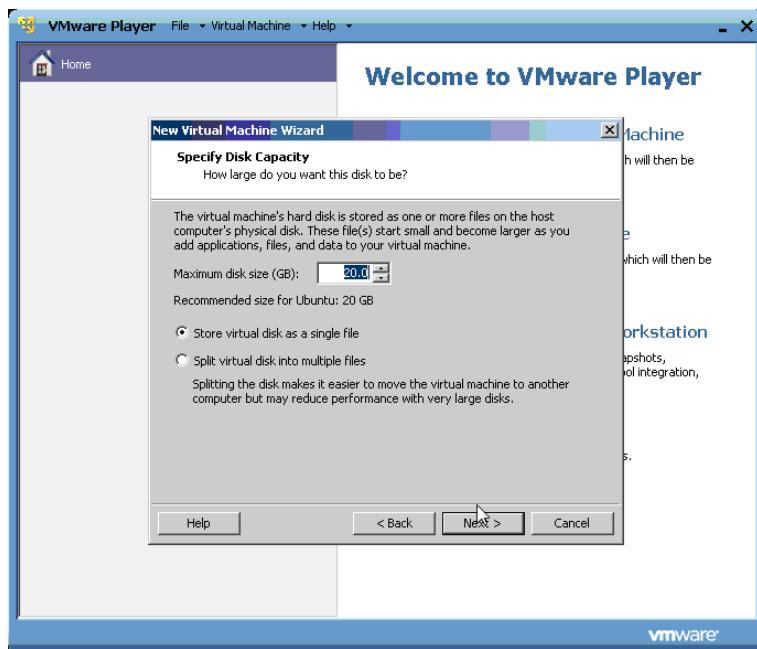
Slika 30: Pojaviće se prozor za izbor instalacionog diska sistema. Pošto ne želimo da odmah instaliramo sistem, odabraćemo opciju I will install the operating system later i pritisnuti dugme Next.



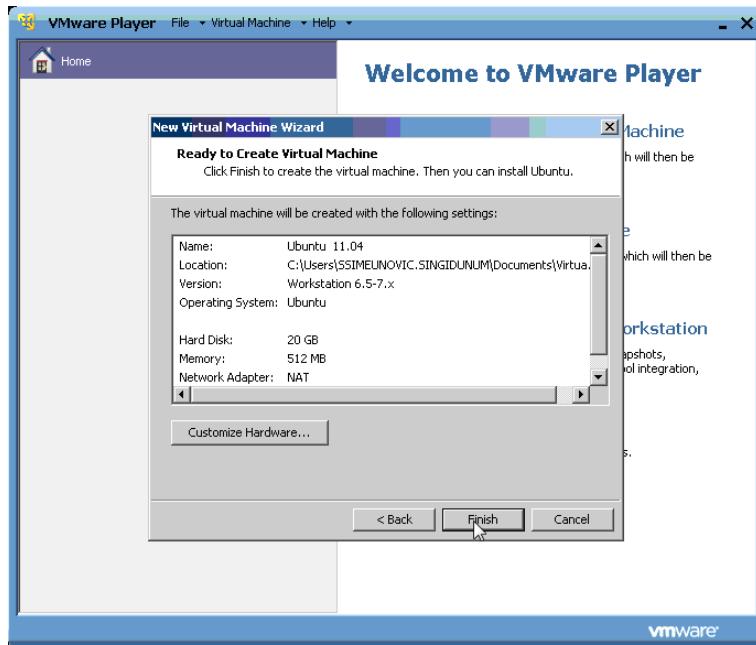
Slika 31: Prozor za izbor "gostujućeg" operativnog sistema. Nakon izbora opcija treba pritisnuti dugme Next.



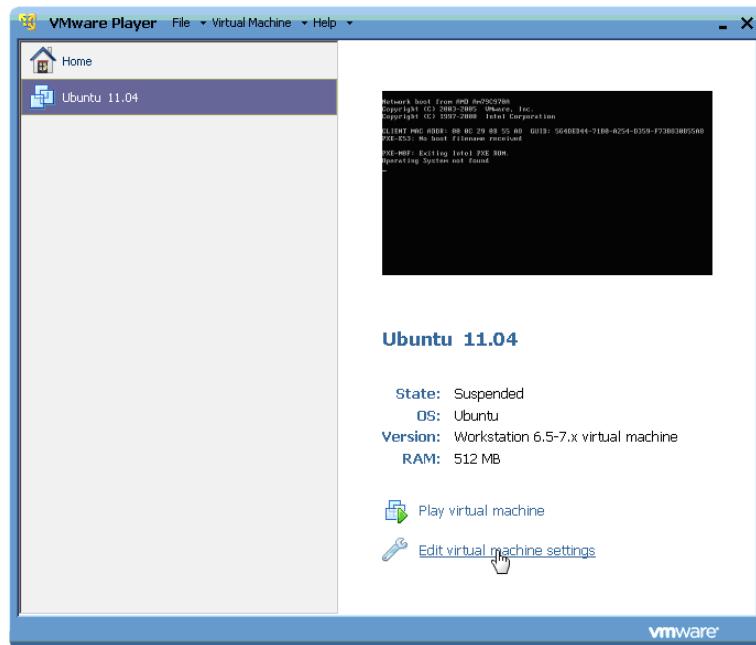
Slika 32: Prozor za unos naziva i izbor lokacije za virtuelnu mašinu.



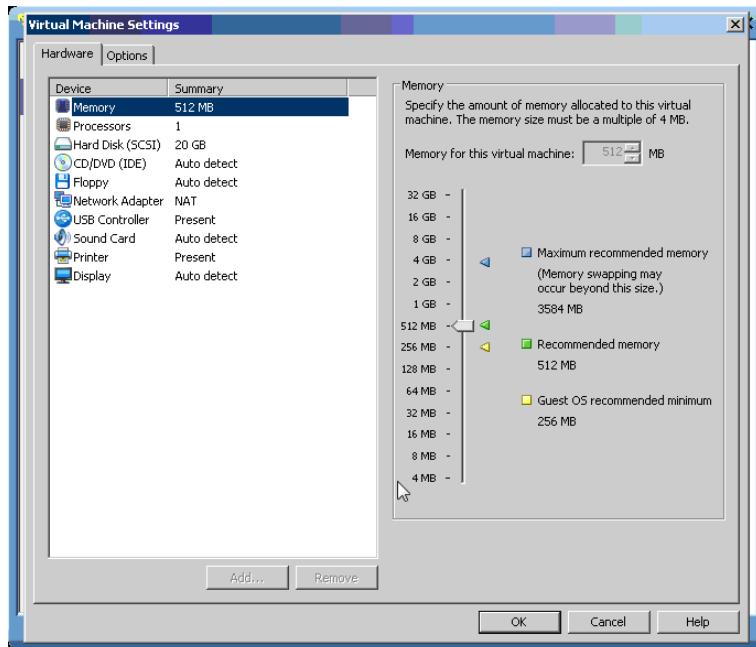
Slika 33: Izbor veličine i tipa virtuelnog diska.



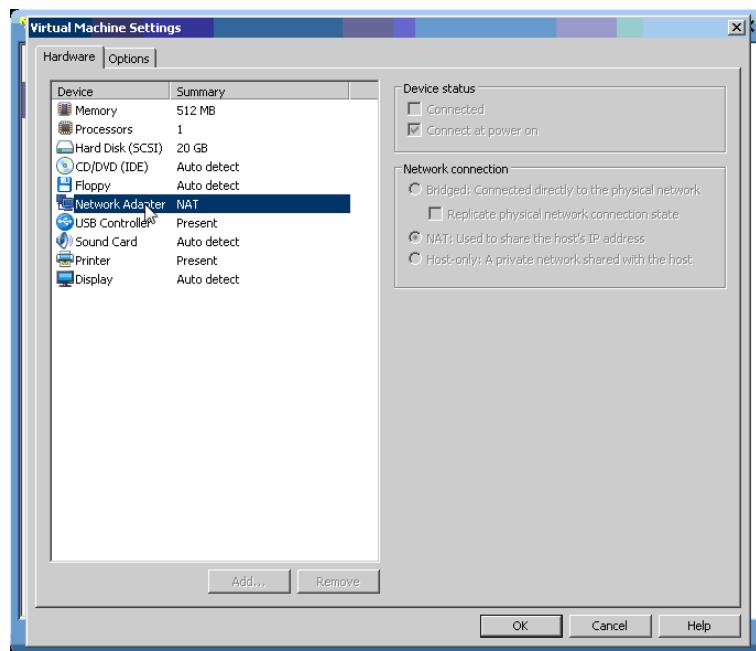
Slika 34: Prikaz izabranih opcija.



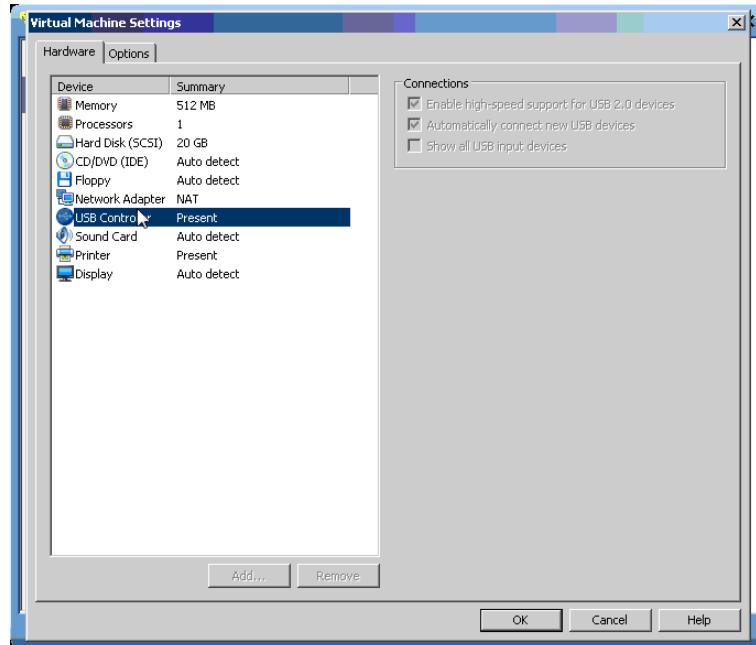
Slika 35: Da bismo dodatno podesili virtuelnu mašinu, treba izabrati opciju Edit virtual machine settings.



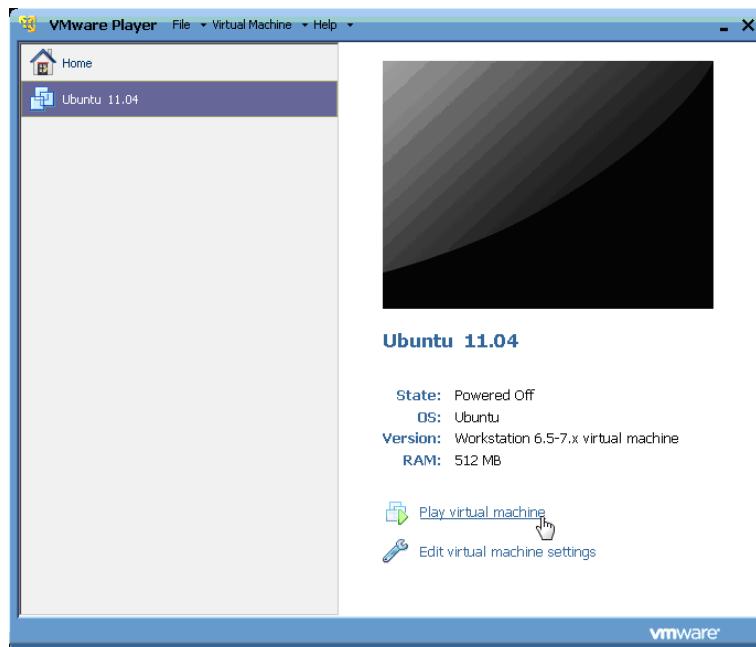
Slika 36: Prozor za detaljnija podešavanja virtuelne mašine; prikazano je podešavanje RAM memorije.



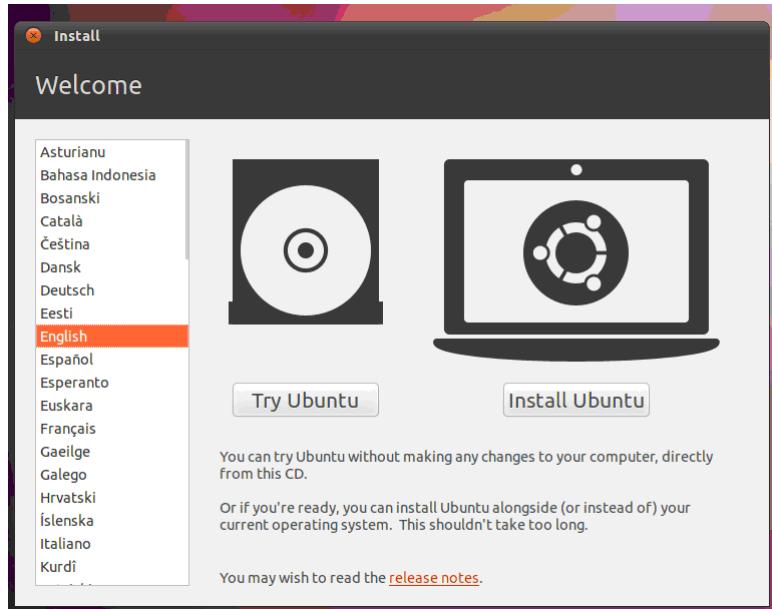
Slika 37: Podešavanje mrežne kartice i tipa povezivanja.



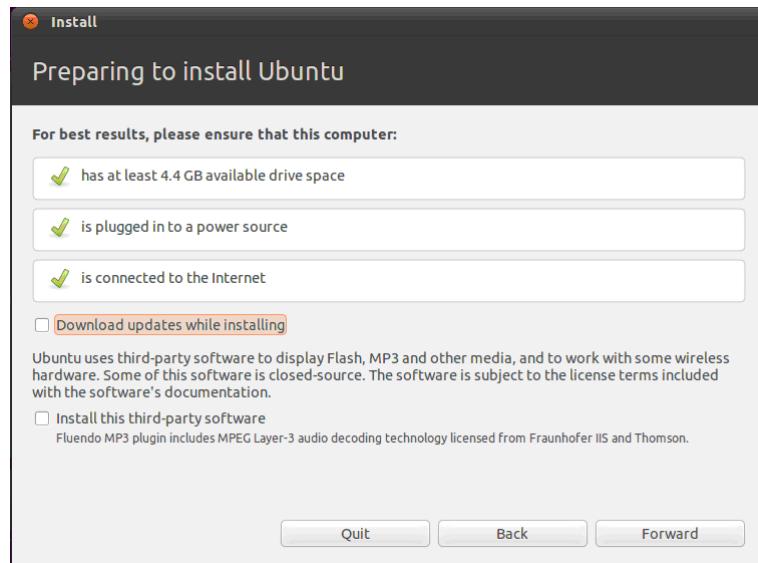
Slika 38: Podešavanje USB veze između HOST i VM računara.



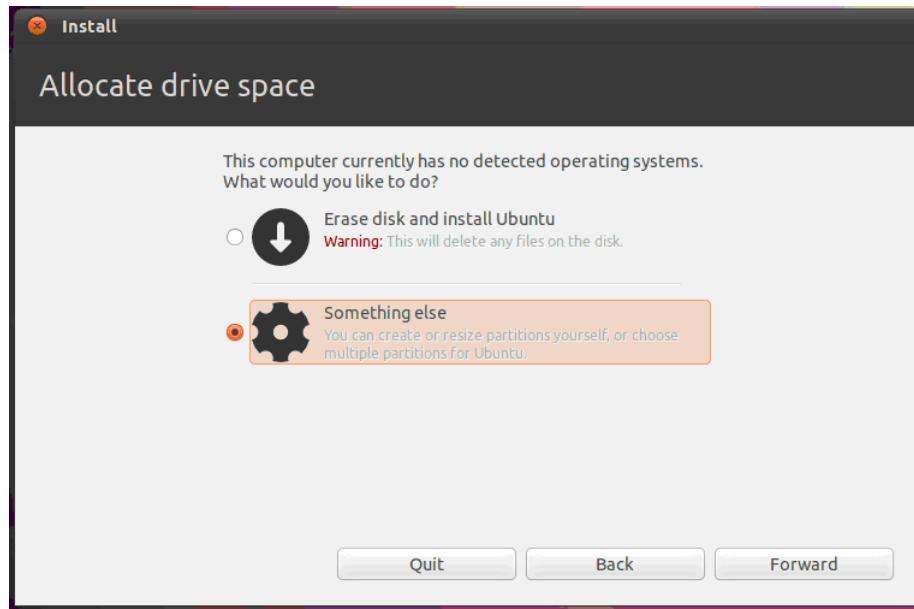
Slika 39: Izgled korisničkog interfejsa za virtuelni računar. Da bi se pokrenuo sistem, treba izabrati opciju Play virtual machine.



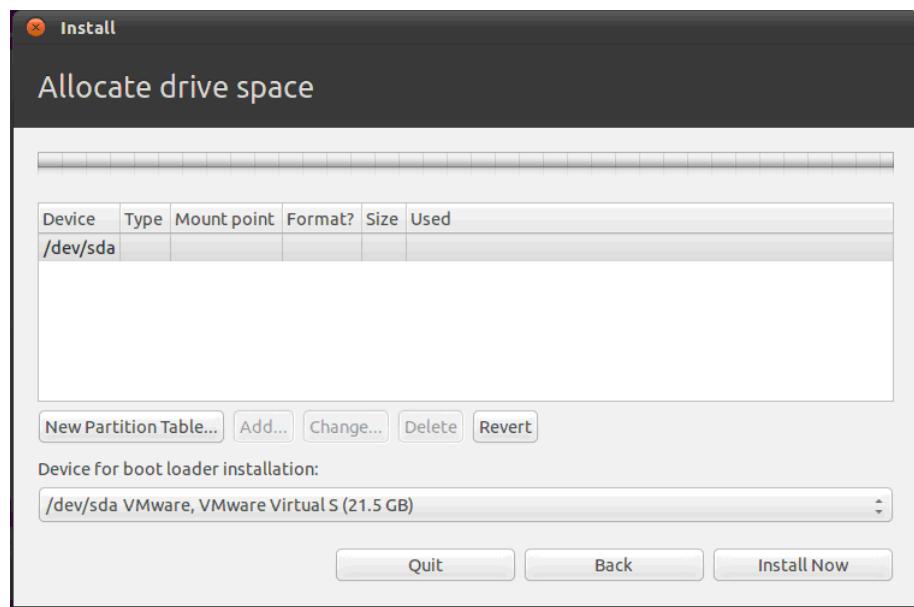
Slika 40: Instalacioni interfejs. Za instalaciju treba izabrati opciju Install Ubuntu.



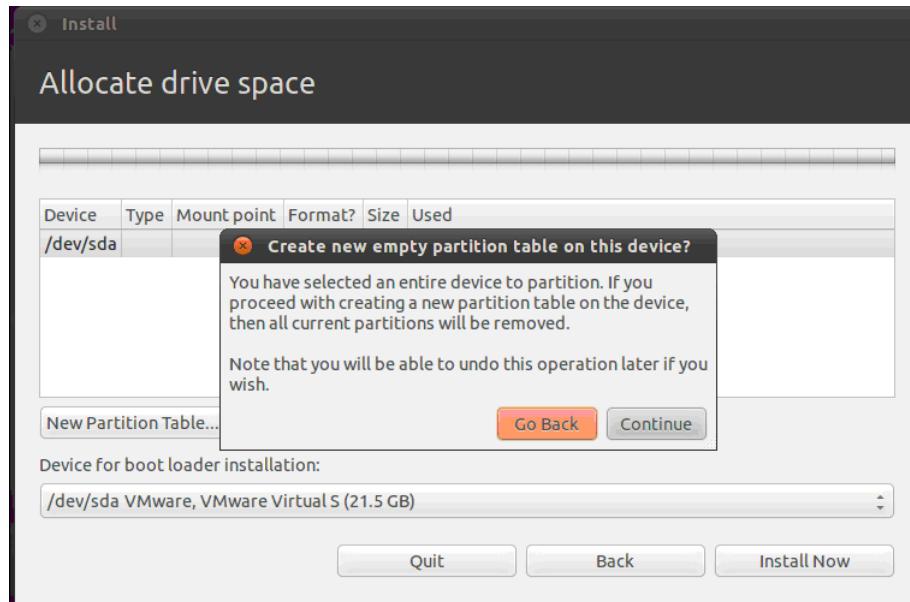
Slika 41: Prozor sa informacijama o zahtevima pre instaliranja, opcijama za ažuriranje u toku instalacije i instalacijom određenih dodataka (npr. MP3 kodeka).



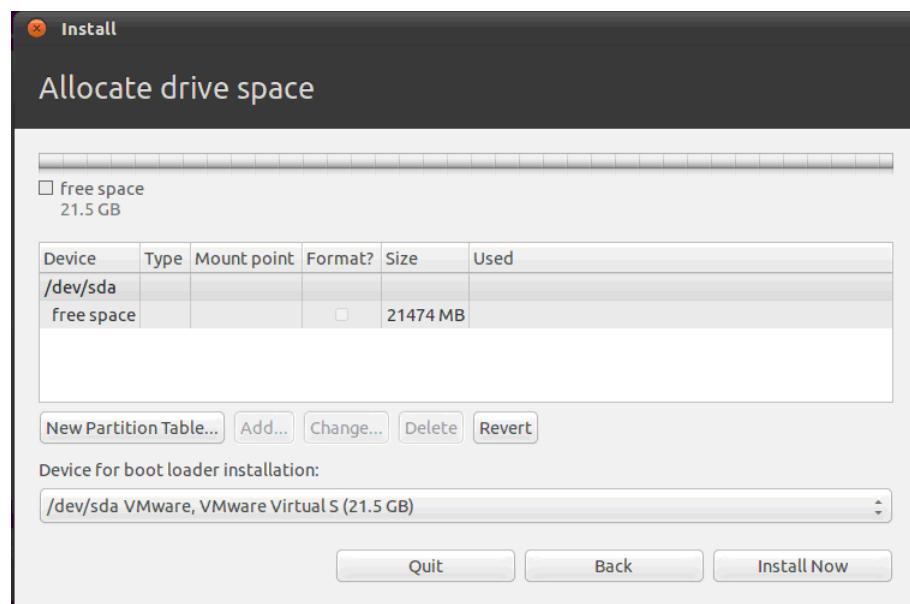
Slika 42: Prozor za alociranje prostora na disku.



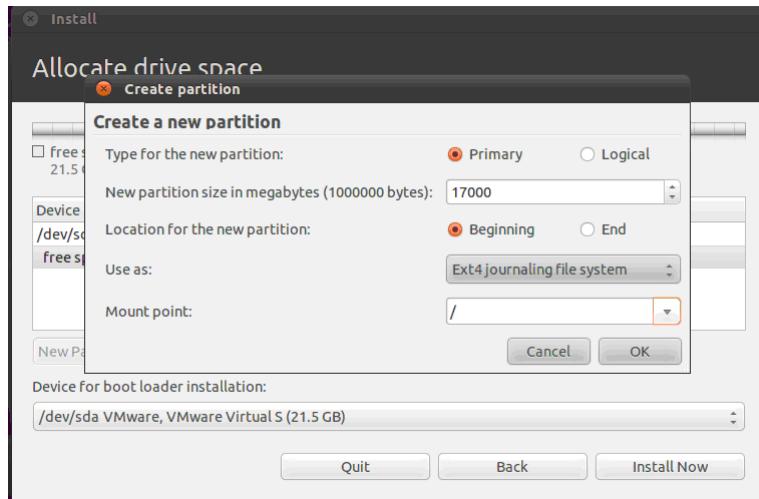
Slika 43: Prozor za podelu diska na particije.



Slika 44: Nakon izbora opcije New Partition Table pojaviće se prozor upozorenja.  
Treba izabrati opciju Continue.

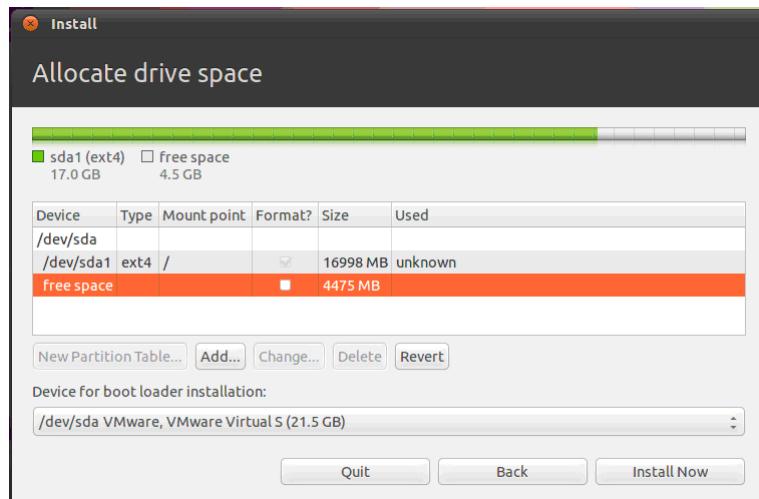


Slika 45: Prozor za particionisanje sa virtuelnim diskom.



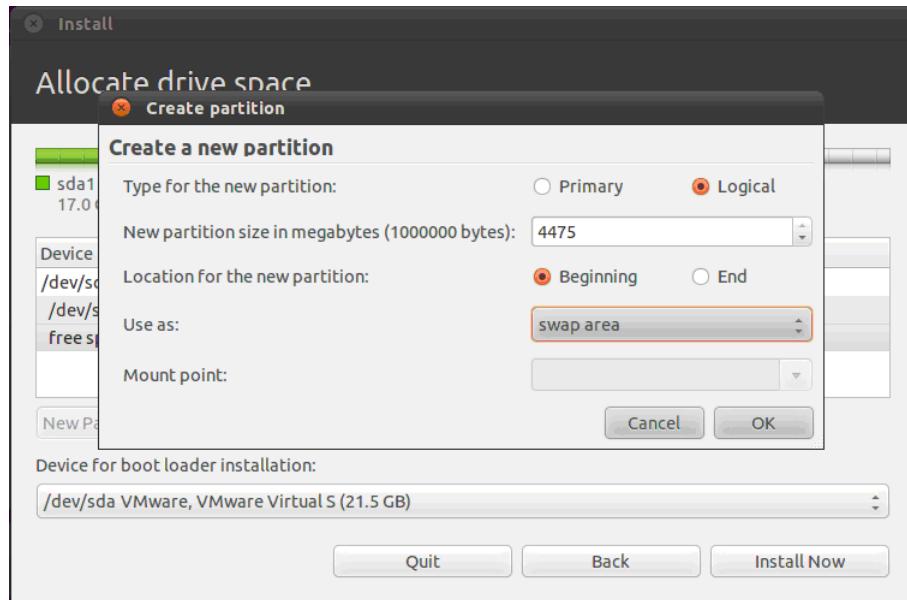
Slika 46: Dodavanje nove primarne particije. Potrebno je odabrati veličinu, tip i tačku gde će se montirati nove particije. Preporučljivo je da tip particije bude ext4.

Pod opcijom "Mount point" treba odabratи "/"..

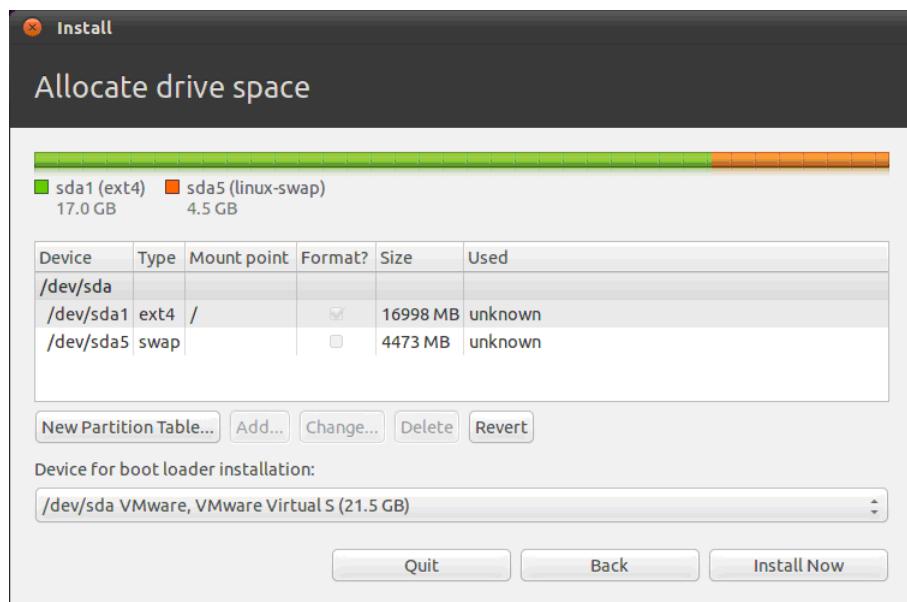


Slika 47: Nakon potvrde dobijamo prikaz virtuelnog diska i njegovih particija.

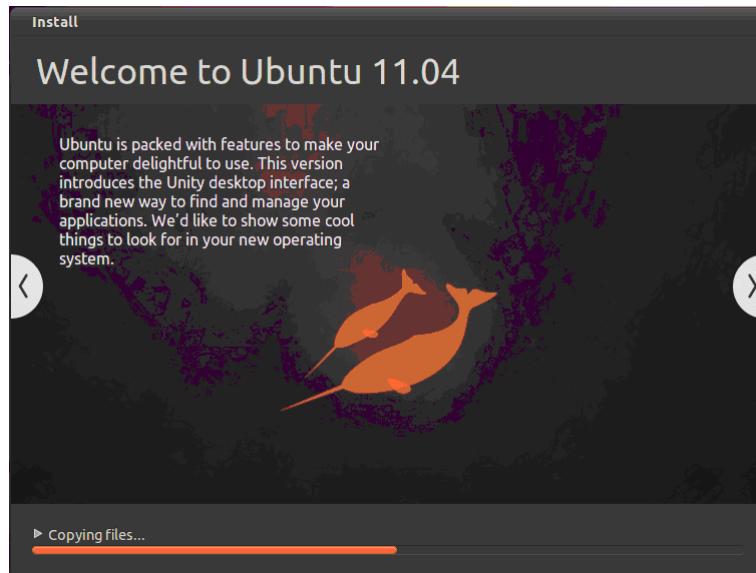
Treba još samo napraviti swap particiju, a zatim nastaviti sa procesom instalacije. Pritiskom na free space i biranjem opcije Add pokrećemo proceduru za kreiranje nove particije. Može da postoji i više od dve particije.



Slika 48: SWAP particija treba da bude bar 1.5 puta veća od kapaciteta RAM memorije računara. U praksi se često koristi odnos RAM memorije i SWAP particije 1:2.



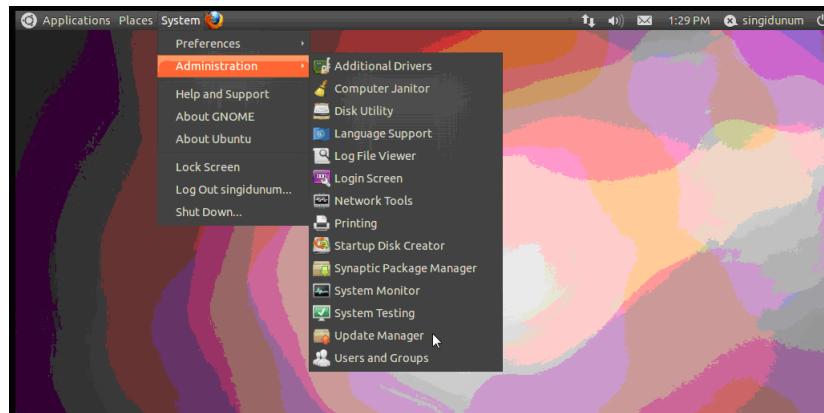
Slika 49: Nakon što se kreiraju primarna i swap particija, proces instalacije može da se nastavi.



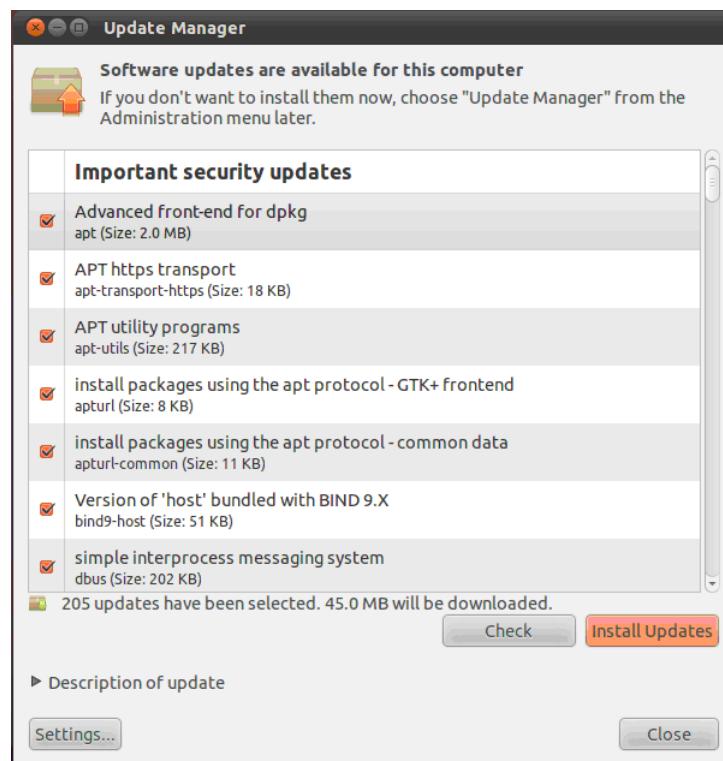
Slika 50: Završni korak instalacije. Nakon uspešne instalacije potrebno je izvaditi instalacioni CD i pritisnuti Enter da bi se računar restartovao.



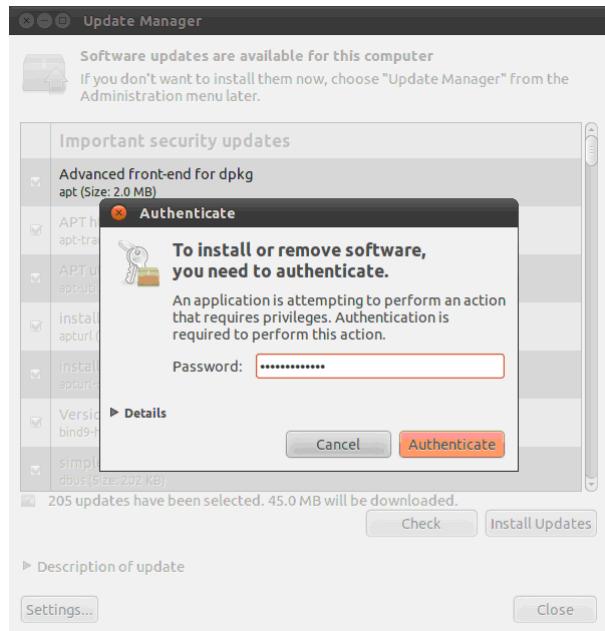
Slika 51: Po što se računar ponovo pokrenuo, potrebno je prijaviti se na sistem.



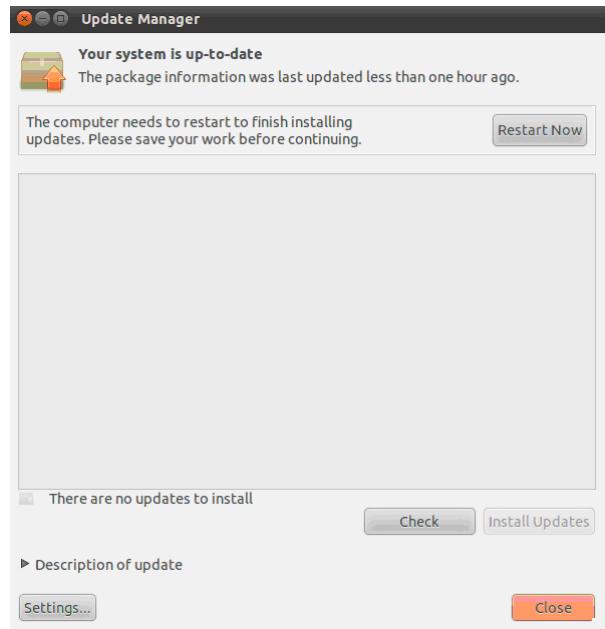
Slika 52: Kada se sistem pokrene, preporučljivo ga je ažurirati.



Slika 53: Ako ima dostupnih zakrpa, sistem će nas obavestiti.



Slika 54: Pre nego što počne sa instalacijom, sistem traži lozinku.

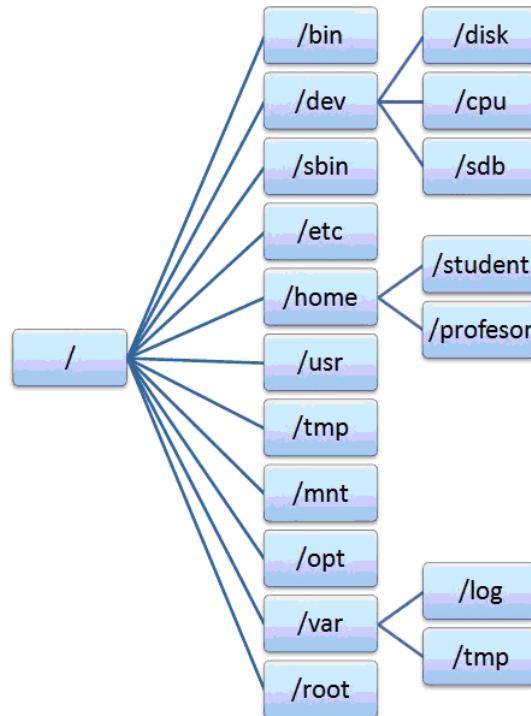


Slika 55: Nakon završenog ažuriranja potrebno je restartovati računar.

## Distribucije Linuxa

Linux kernel podržava multitasking sa prinudnom suspenzijom procesa (kako u sistemskom, tako i u korisničkom režimu), niti, virtuelnu memoriju, deljene biblioteke, upravljanje memorijom, Internet protokol itd. Linux je monolitno jezgro, što znači da se drajveri za uređaje, kao i dodaci za kernel nalaze i izvršavaju u prostoru kornela (tzv. nultom prstenu u većini arhitektura), sa izuzetkom nekoliko dodataka koji se pokreću u korisničkom prostoru. Time je omogućen potpun pristup hardveru. Grafički sistem, za razliku od MS Windowsa, nije deo kornela. Deljenje zadataka (procesa ili niti) omogućuje pravilno korišćenje prekida i bolju podršku za simetrično multiprocesiranje (SMP). Kraći je odziv, što Linux čini podesnjim za rad sa aplikacijama u realnom vremenu. Linux je većim delom pisan u nadograđenoj i donekle izmenjenoj verziji programskog jezika C, koju podržava kompjajler GCC (GNU Compiler Collection). Zbog podržanih nadogradnji jezika C, GCC je dugo bio jedini kompjajler sposoban da ispravno interpretira Linux jezgro. Linux koristi i druge programske jezike, npr. Perl, Python i jezike za shell scripting. Neki drajveri mogu biti napisani i u jezicima C++ i Fortran, ali to nije preporučljivo. Struktura sistema datoteka u Linuxu je takva da počinje od korena (root) i grana se na poddirektorijume. U Linuxu je sve datoteka (npr. uređaji kao što su disk, USB fleš memorije itd.). Datoteke u Linuxu se dele na sledeće tipove:

- ▷ direktorijume (datoteke koje sadrže spisak drugih datoteka)
- ▷ specijalne datoteke (mehanizmi korišćeni za ulaz i izlaz)
- ▷ linkove (sistem koji omogućuje da pojedine datoteke budu vidljive u nekoliko direktorijuma, slično prečicama u Windowsu)
- ▷ (domain) sockets - specijalan tip datoteke, sličan TCP/IP soketima (krajnjim tačkama komunikacije), koji omogućuje zaštitu internim umreženim procesima помоћu kontrole pristupa sistemu datoteka.
- ▷ named pipes –slično soketima, prave put preko koga komuniciraju procesi, ali ne koriste mrežnu logiku soketa.



Slika 56: Linuxov sistem datoteka.

Devedesetih godina prošlog veka, ext2 je bio standardni sistem datoteka. Kada bi slučajno nestalo struje, korisnik bi često gubio podatke sa particij ext2. Zbog toga su dva projekta počela sa razvojem zamene za sistem ext2; bili su to ext3 i ReiserFs. Cilj projekta ext3 bio je da napravi brz sistem datoteka, i ukoliko bi došlo do pada sistema, bilo bi dovoljno da korisnik restartuje sistem, a ne i da proverava njegov integritet. Nedostatak sistema datoteka ext3 je to što ne može da povrati jednom obrisane datoteke. GNU/Linux podržava sledeće sisteme datoteka: ext2, ext3, ext4, ReiserFs, Journaled File System (JFS), XFS, NTFS.

## Debian

Debian je distribucija Linuxa sastavljena od softverskih paketa koji se distribuiju kao besplatni i otvorenog koda pod GNU General Public licencom, kao i drugim open source licencama. To je jedna je od najuticajnijih distribucija koje koriste GNU OS alate i Linux kernel. Poznat je po strogoj privrženosti filozofiji otvorenog softvera, primeni saradnje u razvoju softvera i procesima testiranja. Popularan je i u server i u desktop varijanti. Projekat Debian je nezavisna decentralizovana organi-

zacija koju ne podržava nijedna kompanija, kao što je to slučaj sa nekim distribucijama GNU/Linuxa (Ubuntu, OpenSUSE, Fedora, Mandriva...). Debian je osnova za mnoge druge distribucije, npr. Ubuntu, Dreamlinux, Knoppix, BackTrack itd. Sadrži preko 25.000 softverskih paketa za čak 12 procesorskih arhitektura. Standardna instalacija Debiana koristi grafičko okruženje GNOME, a nudi veliki broj programa korisnih za osnovne poslove na računaru. Pored okruženja GNOME, postoje gotove ISO slike na diskovima sa instaliranim grafičkim okruženjem KDE, zbirkom aplikacija, Xfce i LXDE. Pošto se Debian paketi ne ažuriraju između zvaničnih izdanja, postoje korisnici koji se odlučuju za verzije koje nisu stabilne ili se testiraju. Međutim, takve distribucije mogu prouzrokovati probleme u radu, čak i pad sistema. Za svaki Debian paket postoji osoba koja ga održava, prati izdanja autora i obezbeđuje usklađenost paketa sa ostatkom distribucije, kao i sa Debian politikom. Postoji dobro razvijen sistem za prijavljivanje, praćenje i ispravke grešaka. Trenutno aktuelne verzije Debian distribucije su:

- ▷ stabilna (Lenny). Zamrznuta zbog testiranja i ispravke grešaka. Ažurira se samo u slučaju većih bezbednosnih ispravki.
- ▷ test (Squeeze). Sadrži više paketa od stabilne, koji nisu dovoljno testirani da bi ušli u stabilnu distribuciju. Neprekidno se ažurira dok ne uđe u zamrznuto stanje.
- ▷ nestabilna (Sid). Sadrži pakete koji su trenutno u fazi razvoja. Ova verzija je predviđena za Debian programere koji učestvuju u projektu i potrebne su im najnovije biblioteke.

Instalacija softverskih paketa u distribuciji Debian obavlja se pomoću paketa:

- ▷ dpkg: uslužni program komandne linije za instaliranje, uklanjanje i obezbeđivanje informacija o lokalnim .deb paketima. GDebi proširuje funkcionalnost paketa dpkg mogućnošću instaliranja iz mrežnog skladišta i može se koristiti kako iz komandne linije, tako i u grafičkom interfejsu.
- ▷ apt: instaliranje paketa iz skladišta na mreži apt proširuje funkcionalnost dpkg mogućnostima pretraživanja, preuzimanja i instaliranja paketa iz mrežnog skladišta zajedno sa njihovim bibliotekama, bilo iz binarnih datoteka, bilo kompajliranjem izvornog koda. Služi i za nadogradnju paketa, kao i cele distribucije operativnog sistema.

Debian Live je verzija Debiana koje se može direktno pokrenuti sa prenosivih medijuma (CD/DVD ili USB fleš memorija) ili preko mreže, bez instaliranja na disk. Na taj način korisnik može da isproba Debian pre instalacije ili da ga koristi kao disk za podizanje sistema. Postoji gotova ISO slika sa Debian Live CD-om za spašavanje sistema u nekoliko verzija (GNOME, KDE, Xfce ...). Za instaliranje na disk koristi se Debian Installer koji se nalazi na CD-u.

Debian nema posebnih hardverskih zahteva osim onih uobičajenih za Linux kernel i GNU alate. Linux, a samim tim i Debian, podržava više procesora, tj. simetrično multiprocesiranje. Hardverski zahtevi su različiti zavisno od stepena instalacije, a ra-

stu sa povećanjem broja instaliranih komponenti. Bez grafičkog okruženja zahteva 64MB memorije (preporučuje se 256MB) i 1GB prostora na disku. Sa grafičkim okruženjem je potrebno 64MB memorije (preporučuje se 512MB) i 5GB prostora na disku. Preporučeni minimum za frekvenciju procesora je 1GHz (za desktop sisteme).

## **Backtrack**

Backtrack je najcenjenija i najkorišćenija Linux distribucija kada je reč o zaštiti informacionih sistema. Zasniva se na distribuciji Ubuntu, koristi grafički interfejs KDE i ima brojne bezbednosne i forenzičke alate. Kao i u distribuciji Ubuntu, skup alata je moguće ažurirati i proširivati koristeći mrežna skladišta (online repository). Aktuelna verzija Backtrack 4 R2 je besplatna. Backtrack je moguće instalirati i koristiti kao primarni operativni sistem ili ga pokretati sa LiveDVD-a ili USB diska. Prvo izdanje Backtracka pojavilo se 2007, a tokom godina se nametnulo kao standard za testiranje bezbednosti i digitalnu forenziku. Backtrack 4 nudi brojne aplikacije, od kojih su najinteresantnije:

- ▷ Metasploit Framework: okruženje za testiranje bezbednosti operativnih sistema. Otvorenog je koda i sadrži najveću bazu testiranih propusta na svetu.
- ▷ Wireshark: popularan softver za analizu mrežnih protokola. Najvažnija funkcija ovog softvera je "hvatanje" paketa u mrežnom interfejsu i njihov detaljan prikaz radi analize.
- ▷ Nmap ili Network Mapper: program koji se koristi za pronalaženje hostova i servisa na računarskoj mreži.
- ▷ Hydra: alat za provjeru lozinki.
- ▷ Hashcat: jedan od najbržih alata za razbijanje heševa.
- ▷ Maltego: aplikacija za prikupljanje informacija i forenziku.
- ▷ Foremost: alat za rekonstruisanje obrisanih datoteka (file carving).
- ▷ MacChanger: alatka za promenu MAC adrese.
- ▷ S.E.T.: baza alata za socijalni inženjering. Može da klonira Web lokacije.

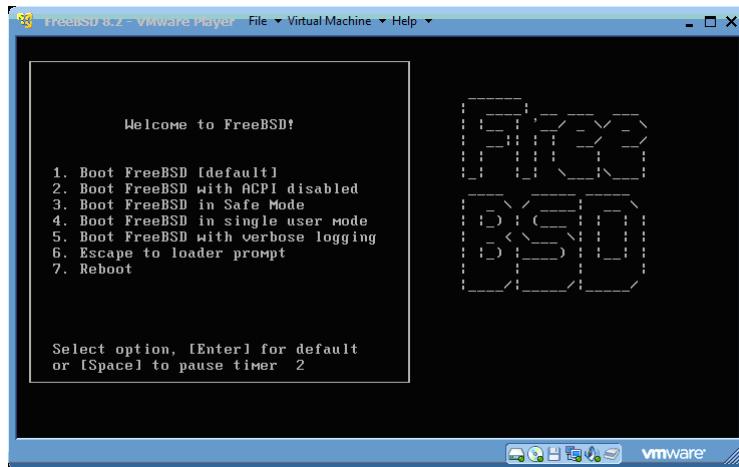
## **Instaliranje operativnog sistema FreeBSD**

FreeBSD je slobodan operativni sistem iz grupe operativnih sistema Unix razvijen na Univerzitetu Berkli 1993 godine. Osnovni delovi, tj. slojevi ovog operativnog sistema su:

- ▷ Kernel
- ▷ Drajveri za periferne uređaje
- ▷ Skup korisničkih aplikacija i biblioteka

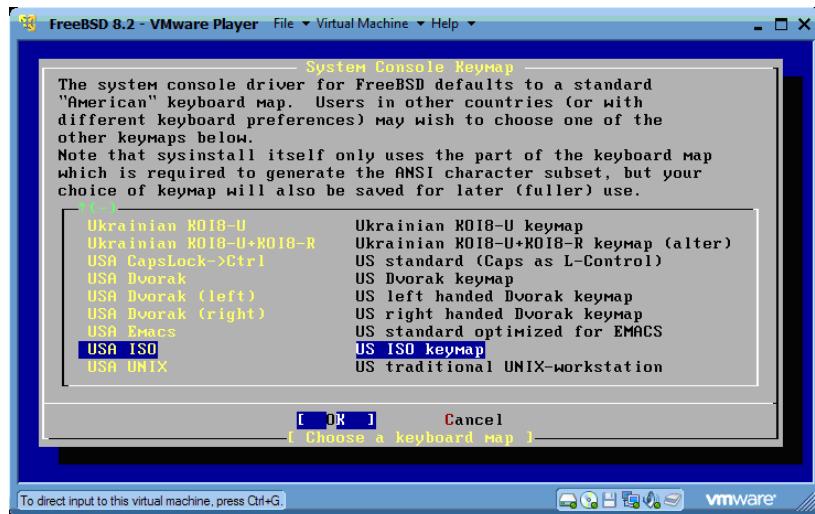
FreeBSD je prvenstveno orijentisan ka okruženjima koja zahtevaju stabilnost, brzinu i sigurnost (npr. računski centri, klaster sistemi, složeni informacioni sistemi, simulacije ekperimenata itd). Sam sistem se može koristiti kao desktop okruženje u svakodnevnoj primeni. Instaliranje aplikacija obavlja se preko FreeBSD portova, koji maksimalno olakšavaju instalaciju aplikacija prisutnih u port sistemu. FreeBSD port sistem je skup javnih servera na kojima se nalaze aplikacije u paketima unapred podešenim za instaliranje. Prednost sistema FreeBSD nad Linuxom je to što aplikacije koje su razvijane za Linux rade i na FreeBSD-u, ali ne i obrnuto. FreeBSD omogućuje prilagođavanje radnog okruženja, a nudi brojne kernel module u osnovi sistema. Ugrađena podrška za skup mrežnih protokola TCP/IP, IPv6, SCTP, IPsec omogućuje lako integrisanje FreeBSD sistema u bilo koje okruženje bez instalacije dodatnih aplikacija. Opsluživanje velikog broja korisnika kroz servisne aplikacije kao sto su HTTP ili FTP server veoma se lako realizuje instalacijom samih servisa iz već pomenutih FreeBSD portova, dok se podešavanje obavlja brzo u konzoli operativnog sistema. Sistem za čuvanje podataka je preuzet od ranijih verzija UNIX-a i zove se UFS2 (Unix File System). Struktura direktorijuma je vrlo slična Linuxovoj. Bezbednost u okviru sistema implementirana je pomoću liste kontrola pristupa (ACL). FreeBSD (verzija 8.2) se sa adrese [www.freebsd.org](http://www.freebsd.org) može preuzeti u nekoliko formata namenjenim različitim arhitekturama procesora.

### Instaliranje operativnog sistema FreeBSD



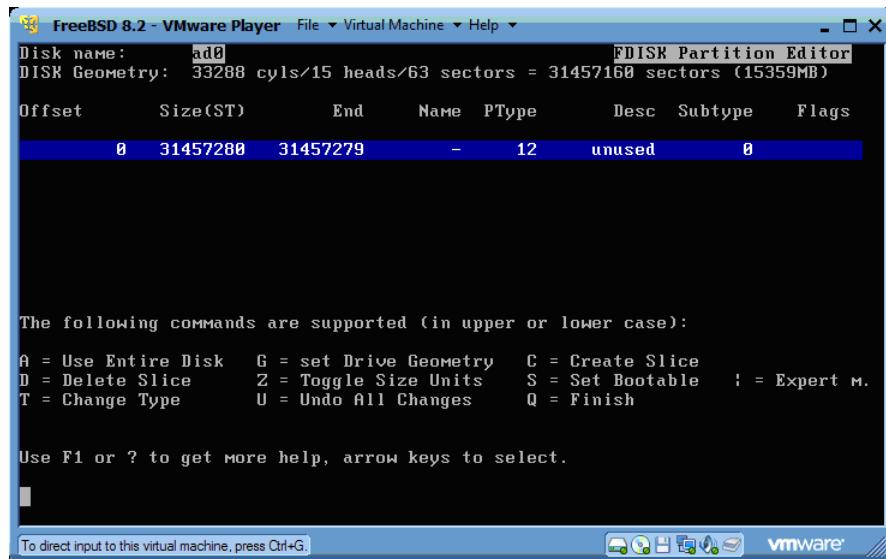
Slika 57: Nakon podizanja sistema pojavljuje se meni sa sedam opcija od kojih pet omogućuju instaliranje operativnog sistema na različite načine. Izborom prve stavke menija instalacija započinje sa standardnim podešavanjima.

U sledećem koraku se bira željena zemlja/region u prikazanoj listi. Nakon izbora zemlje bira se raspored tastera na tastaturi. Izborom opcije Standard u listi, instalacija sistema se nastavlja sa standardnim parametrima. Dodatne opcije ukazuju da se sistem može instalirati na različite načine.



Slika 58: U ovom koraku može se promeniti raspored tastature, instalirati dokumentacija i učitati već pripremljeni parametri.

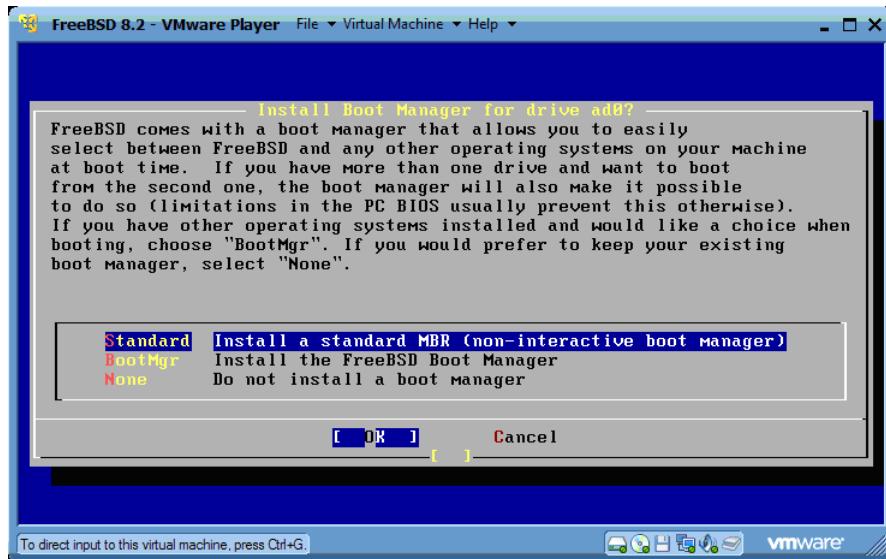
Pravljenje sistemske particije, swap i dodatnih particija obavlja se pomoću alata FDISK za partitionisanje diska. Unošenjem parametra „C“ može se napraviti particija određene veličine, ali u ovom slučaju ceo disk će biti iskorišćen kao sistemska particija (izabrana je opcija „A“).



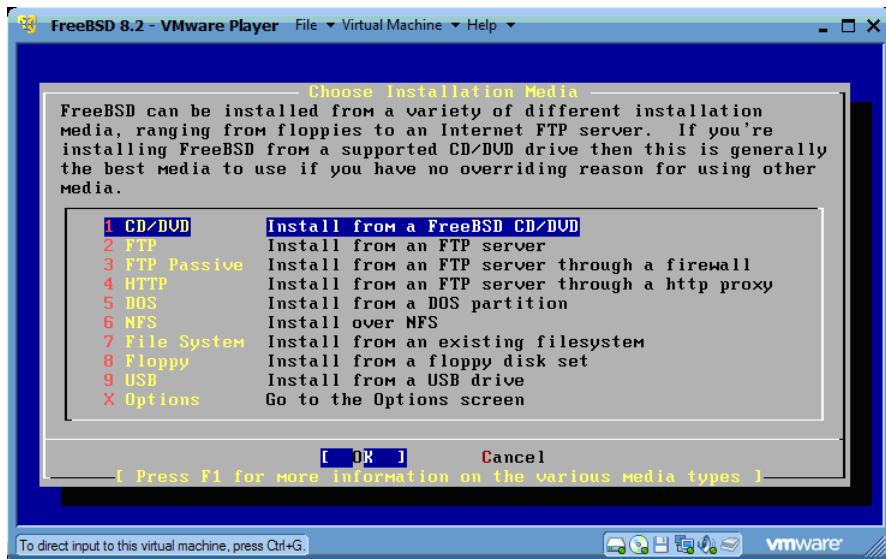
Slika 59: Pravljenje particija na disku.

Zatim se instalira upravljač podizanjem operativnog sistema (boot manager), izborom jedne od sledeće tri opcije:

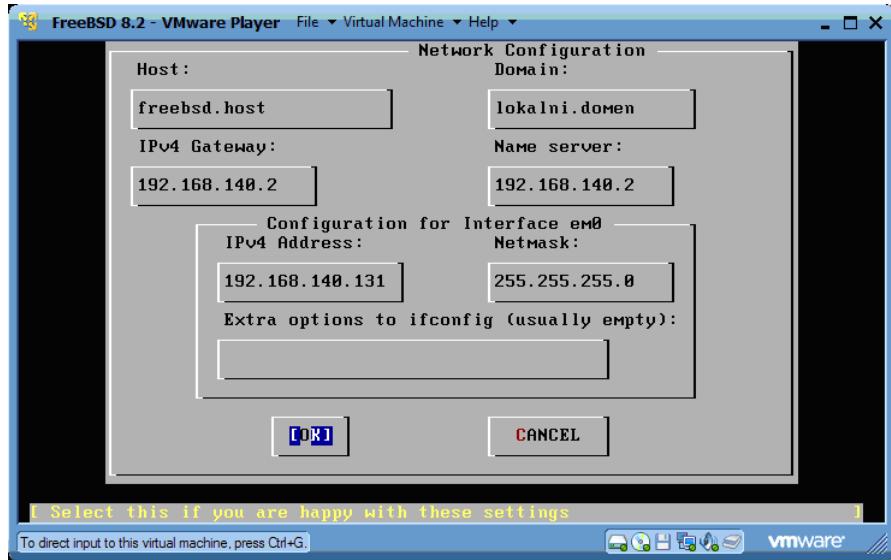
1. Standard – upravljač će biti statičan
2. BootMgr – ako u računaru postoji još jedan operativni sistem, ova opcija omogućuje izbor sistema pri pokretanju računara.
3. None – upravljač neće biti instaliran



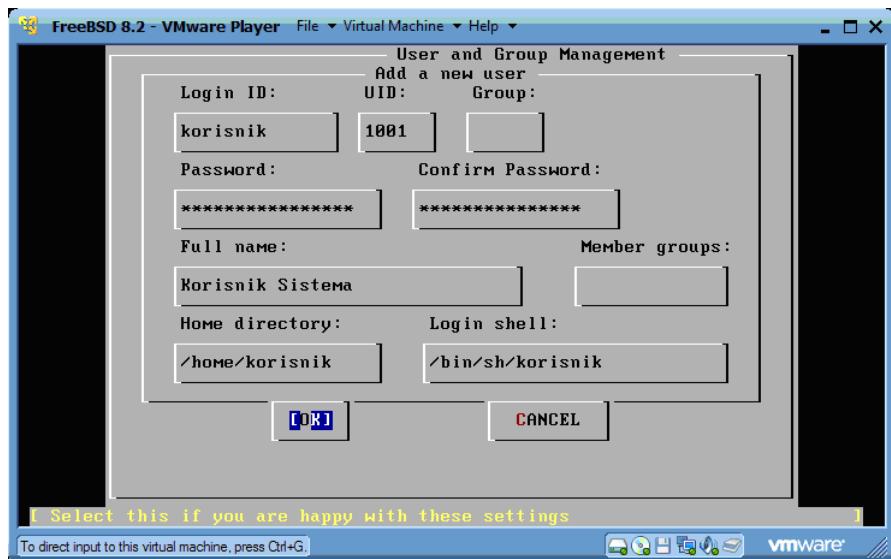
Slika 60: Instaliranje upravljača podizanjem sistema.



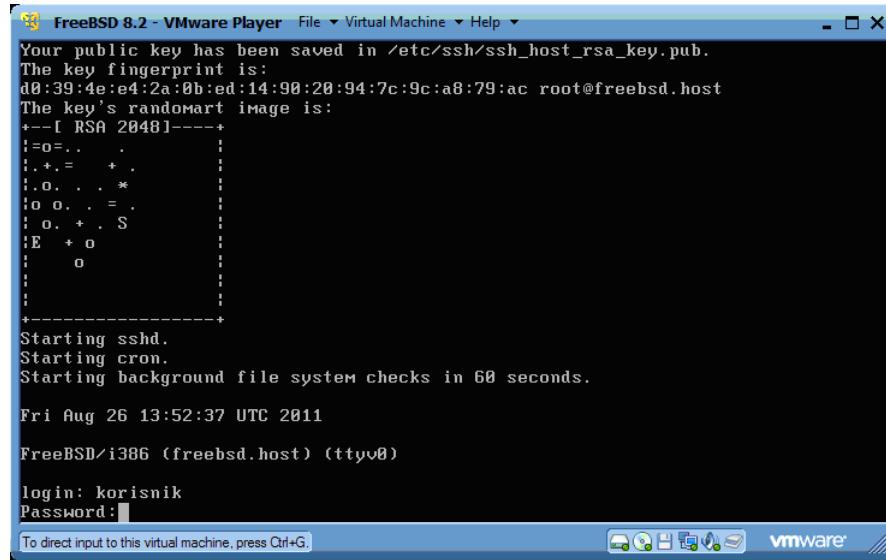
Slika 61: U listi ponuđenih opcija treba odabratи lokaciju sa koje će sistem biti instaliran (CD/DVD).



Slika 62: U prikazanom obrascu mogu se podešiti mrežni parametri, ali je to moguće i posle instaliranja.



Slika 63: Popunjavanje obrasca sa informacijama o korisniku kome će biti dozvoljen pristup sistemu.

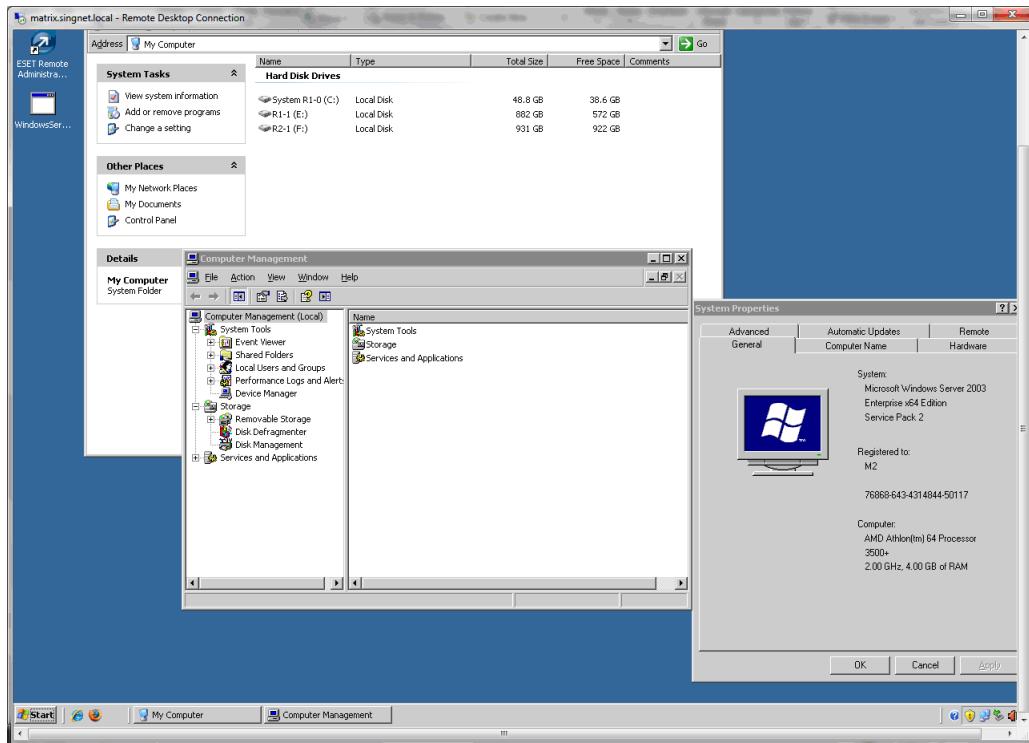


Slika 64: Nakon završetka instalacije i restartovanja računara, može se pristupiti sistemu.

## **Operativni sistemi za servere: Windows Server 2008**

Windows Server je Microsoftov serverski operativni sistem. Ovaj Microsoftov izvod ima samo izdanja optimizovana za ulogu servera u mrežnom okruženju, a to su Windows Server 2003 Standard Edition, Windows Server 2003 Enterprise Edition, Windows Server 2003 Datacenter Edition i Windows Server 2003 Web Edition. Sve verzije ovog operativnog sistema podržavaju deljenje datoteka i štampača, rade kao serveri aplikacija i serveri e-pošte, pružaju usluge autentifikacije korisnika, deluju kao sertifikaciono telo za sertifikate X.509, uravnotežuju mrežno opterećenje (Network Load Balancing - NLB) itd. Poslednja verzija Servera 2003 je Web Edition. Osnovna ideja s kojom je započeta realizacija ove verzije je želja Microsofta da novom verzijom Web servera, takozvanim IIS-om, nadmaši svoje glavne konkurente: Apache i Sun. Sa tim ciljem nekoliko funkcija osnovne verzije operativnog sistema je isključeno, a osiromašena verzija je ponuđena proizvođačima hardvera za prodaju zajedno sa serverima. Ta verzija može da pročita samo 2GB memorije (Windows 2000 je podržavao 4GB), a onemogućene su i sledeće funkcije:

- ▷ Ne radi kao kontroler domena (ali se može povezati na postojeći domen),
- ▷ Ne povezuje sisteme sa udaljenih lokacija putem „Terminal Services“ (ali podržava Remote Desktop Connection),
- ▷ Nije moguće deljenje veze sa Internetom (Internet Connection Sharing) niti mrežno premošćavanje (Net Bridging),
- ▷ Ne radi kao server za dinamičko dodeljivanje IP adresa (DHCP) i server za faks.



Slika 65: Izgled korisničkog interfejsa Windows Servera 2003 preko daljinske veze.

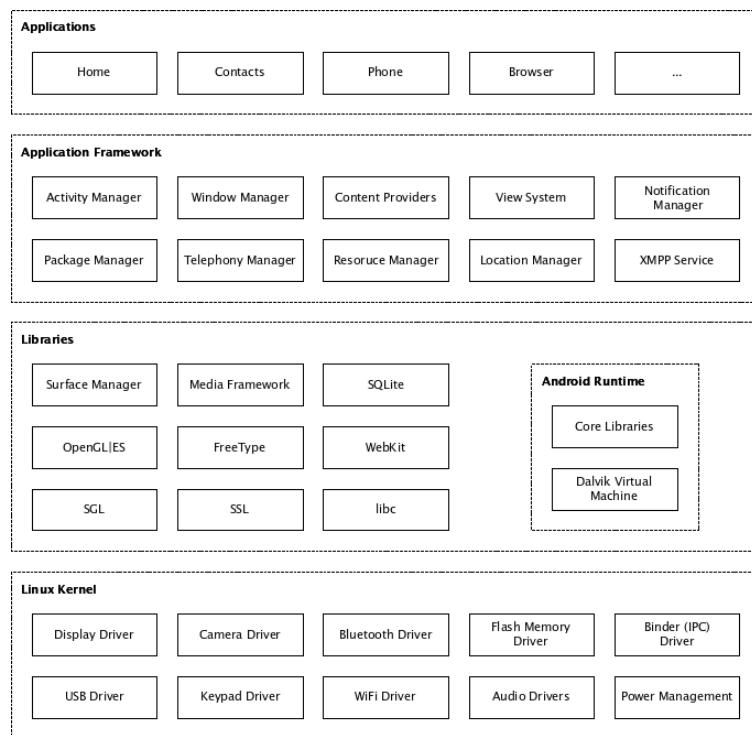
S pojavom operativnog sistema Windows NT 4, Microsoft je predstavio skuplju verziju Servera, pod nazivom NT 4 Server Enterprise koja je imala novu mogućnost klasterizacije i širi memorijski model. Windows Server 2003 Enterprise Edition i dalje podržava klasterizaciju (tada sa najviše četiri računara po klasteru). Pored toga, verzija Enterprise Edition omogućuje pokretanje servera iz "mrežnog skladišnog prostora" (Storage Area Network - SAN), „živo“ instaliranje (hot-install) memorije kao u verziji Datacenter, i ugradnju najviše četiri procesora. Windows Server 2003 sadrži jedan važan alat za administratore, a to je servis koji omogućuje upravljanje sistemom na način koji je bio moguć samo u „mainframe“ računarima pre mnogo godina, nазван Windows Resource Manager. Windows Resource Manager omogućuje raspodelu resursa na način koji odgovara korisniku, npr. Može se zadati koliko će memorije ili procesorske snage moći da koristi SQL Server. Ovaj koristan alat dobija se samo u verziji Datacenter. Osim toga, Datacenter omogućuje klasterizaciju osam računara, kao i instaliranje memorije „na živo“ što smo već pomenuli za verziju Enterprise. Treba samo da otvorite kućište, stavite nov memorijski modul, sačekate par sekundi i sistem automatski prepoznaće novu memoriju, bez potrebe za restartovanjem računara.

Aktuelna verzija je Windows Server 2008 R2.

# Operativni sistemi za mobilne uređaje

Najpoznatiji operativni sistemi mobilnih uređaja su: Android, Symbian, Windows Mobile 7, iOS i Samsung Bada.

Android je operativni sistem otvorenog koda namenjen mobilnim telefonima i tablet računarima. Razvio ga je konzorcijum Open Handset Alliance u saradnji sa kompanijama Google, HTC, Sony, Dell, Intel, Texas Instruments i NVidia. Operativni sistem Android je zasnovan na Linuxovom kernelu. Sadrži module koji povezuju softverske komponente ili korisnike sa aplikacijama, bibliotekama i interfejsima za programiranje (API) napisanim u programskom jeziku Java (Slika 66).



Slika 66: Osnovne komponente operativnog sistema Android.

Android koristi virtuelnu mašinu koja je projektovana tako da optimizuje memorijske i hardverske resurse u mobilnom okruženju. Za pisanje Android aplikacije prvenstveno se koristi razvojno okruženje za Javu. Trenutno na tržištu softvera (Android Market) postoji više od 250 000 aplikacija za telefone sa operativnim sistemom Android. Primena operativnog sistema Android počinje sa verzijom 1.0 koja se pojavila 2008. godine. Tokom poslednje tri godine objavljeno je više verzija operativnog sistema Android, koje su se uglavnom sastojale od ispravki grešaka iz prethodnih verzija i novih funkcija. Aktuelna verzija je 3.2.

## Dodatak A: Kratak uvod u jezik C

Jezik C je izmišljen za potrebe projektovanja operativnog sistema koji bi mogao ponovo da se kompajlira (prevodi) za različite hardverske platforme (tj. za različite procesore). Pošto su operativni sistemi uglavnom programirani u jeziku C, ovaj jezik je prvi izbor kada je reč o programiranju bilo kakve aplikacije koja treba efikasno da komunicira sa operativnim sistemom.

Većina korisnika računara koji ne programiraju misle da je jezik C samo jedan od mnogobrojnih programskih jezika. To je međutim potpuno pogrešno: jezik C je zapravo osnova celokupnog računarstva. UNIX, Microsoft Windows, kancelarijske aplikacije, čitači Weba i još mnogo štošta drugog napisani su u jeziku C. Devedeset devet procenata vremena koje provodite ispred računara verovatno je potrošeno u nekoj aplikaciji programiranoj u jeziku C. Približno 70% open-source softvera je napisano u jeziku C, a preostalih 30% u jezicima čiji su kompajleri ili interpretatori napisani u tom jeziku.

Takođe, za jezik C ne postoji zamena. Pošto on svoju namenu ispunjava gotovo savršeno, nikada se neće ni pojaviti potreba da se zameni nekim drugim jezikom. *Drugi jezici možda su pogodniji za druge namene, ali C svoju svrhu ispunjava savršeno.* Sigurno je da će još veoma dugo svi budući operativni sistemi biti pisani u jeziku C. Iz tog razloga poznavanje operativnih sistema ne može biti kompletno sve dok se ne savlada C programiranje.

### Najjednostavniji C program

Počećemo od jednostavnog C programa kome ćemo dodavati osnovne elemente.

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char* argv) {
    printf("Zdravo!\n");
    return 3;
}
```

Snimite ovaj program u datoteci `zdravo.c`. Sada treba kompajlirati taj program. Kompajliranje (prevođenje) je postupak kojim se C kôd pretvara u asemblerske naredbe. Asemblerske naredbe su programski kôd koji procesor (80?86, SPARC, PowerPC ili

neki drugi) može direktno da razume. Dobijena izvršna binarna datoteka je veoma brza jer je izvršava sâm procesor, tj. čip na matičnoj ploči koji bajt po bajt dohvata reč *Zdravo!* iz memorije i izvršava svaku naredbu. Radni takt procesora (u mega-hercima) približno odgovara broju miliona naredbi u sekundi (*Mega Instructions Per Second, MIPS*) koje je procesor u stanju da izvrši. Interpretirani jezici (kao što su komandni skriptovi ili Java) mnogo su sporiji jer je procesor nije u stanju da razume takav kôd.

Izvršimo sada naredbu:

```
gcc -Wall -o zdravo zdravo.c
```

Opcija `-o zdravo` kaže GNU C kompjajleru `gcc` (koji se u drugim operativnim sistemima zove `cc`) da napravi binarnu datoteku `zdravo` umesto binarne datoteke standardnog naziva `a.out` (koja bi se inače dobila kada se ne bi navela opcija `-o` i ime datoteke). Opcija `-Wall` znači da želimo da vidimo sva upozorenja (all warnings) tokom prevodenja. Ova opcija nije neophodna, ali je korisna za uočavanje i ispravljanje grešaka u programima.

Zatim, pokrenimo program komandom

```
./zdravo
```

U jeziku C, sav kôd nalazi se unutar funkcija. Prva funkcija koju operativni sistem poziva jeste funkcija `main`.

Otkucajte `echo $?` da biste videli povratni kôd programa. Videćete da je on 3, odnosno povratna vrednost iz funkcije `main` (vrednost iza naredbe `return`).

Treba primetiti i znakove navoda sa obe strane reči (stringa) koji se prikazuje na ekranu. U jeziku C, stringovi se navode unutar navodnika. Unutar string konstante (ovde "*Zdravo!*"), specijalna sekvenca `\n` označava znak za novi red. U C programu primetićete i mnoštvo tačka-zapeta; svaka naredba u tom jeziku završava se znakom `;` koji se mora navesti.

Sada probajmo sledeći program:

```
#include<stdlib.h>
#include <stdio.h>
int main(int argc, char* argv) {
    printf("broj %d, broj %d\n", 1+2, 10);
    exit(3);
}
```

Komanda `printf` služi za prikaz rezultata na ekranu. Istovremeno je ova funkcija i deo standardne biblioteke funkcija jezika C. Drugim rečima, određeno je da bilo koja

implementacija jezika C mora da ima funkciju `printf` koja treba da se ponaša na predviđen način.

`%d` zadaje da ispis treba da bude u formatu decimalnog broja. Broj kojim se zamenjuje navodi se kao prvi argument funkcije `printf` nakon string konstante (a to je u ovom slučaju `1+2`). Sledeći `%d` biće zamenjen drugim argumentom (tj. brojem 10). `%d` zove se *specifikator formata*; on u osnovi konvertuje ceo broj u decimalni prikaz.

### Promenljive i tipovi

U jeziku C ne možete da koristite promenljivu bilo kad i bilo gde, ni bez dodeljene početne vrednosti, već kompjleru morate izričito da kažete pre svakog bloka kôda (naredbi koje se nalaze unutar para vitičastih zagrada) koje ćete promenljive koristiti unutar njega. To se postiže *deklarisanjem* promenljivih.

```
#include<stdlib.h>
#include <stdio.h>
int main(int argc, char* argv) {
    int x;
    int y;
    x = 10;
    y = 2;
    printf("broj %d, broj %d\n", 1+y, x);
    exit(3);
}
```

Naredba `int x` je deklaracija promenljive. Ona programu kaže da treba da rezerviše prostor za jednu celobrojnu (integer) promenljivu kojoj će se kasnije pristupati preko imena `x`. `int` je tip promenljive. `x = 10` je dodela vrednosti 10 promenljivoj `x`. Postoje posebni tipovi za različite klase brojeva sa kojima se radi, kao i pridruženi specifikatori formata za njihov ispis.

Primetićete da se `%f` koristi za brojeve u pokretnom zarezu obične i dvostrukе tačnosti (tipovi `float` i `double`). To je zato što se tip `float` uvek prvo konvertuje u tip `double` pre operacija ovakvog tipa.

### Funkcije

```
#include<stdlib.h>
#include <stdio.h>
int main(int argc, char* argv) {
    char a;
    short b;
    int c;
    long d;
    float e;
    double f;
    long double g;
```

```

a = 'A';
b = 10;
c = 1000000;
d = 1000000;
e = 3.14159;
f = 10e300;
g = 10e300;
printf("%c, %hd, %d, %ld, %f, %f, %Lf\n", a, b, c, d, e, f, g);
exit(3);
}
void pomnozi_i_prikazi (int x, int y) {
    printf("%d * %d = %d\n", x, y, x * y);
}
int main(int argc, char* argv) {
    pomnozi_i_prikazi(30, 5);
    pomnozi_i_prikazi(12, 3);
    exit(3);
}

```

Ovde vidimo funkciju različitu od main koja se poziva iz funkcije main. Funkcija je prvo *deklarisana* naredbom

```
void pomnozi_i_prikazi (int x, int y)
```

Ova deklaracija navodi povratni tip funkcije (u ovom slučaju void, jer funkcija ne vraća ništa), ime funkcije (pomnozi\_i\_prikazi), a zatim i *argumente* koji će joj biti prosleđeni. Brojevima koji će biti prosledeni funkciji dodeljuju se imena, x i y, a pre prosleđivanja funkciji oni se konvertuju u tip x i y (u ovom slučaju, int i int). Stvarni C kôd koji čini telo funkcije stavlja se unutar vitičastih zagrada { i }.

Drugim rečima, gornji kôd je ekvivalentan sa:

```
#include<stdlib.h>
#include <stdio.h>
void pomnozi_i_prikazi () {
    int x;
    int y;
    x = <prvi_prosledjeni_broj>
    y = <drugi_prosledjeni_broj>
    printf("%d * %d = %d\n", x, y, x * y);
}
```

### Naredbe for, while, if i switch

Lako je prepoznati format naredbi for, while i if bez dodatnih objašnjenja; C kôd se stavlja u blokove naredbi unutar vitičastih zagrada. Kao u većini programskih jezika, kada želimo da dodamo 1 nekoj promenljivoj (tj. da je inkrementiramo), moramo da napišemo  $x = x + 1$ . U jeziku C skraćenica za inkrementiranje je  $x++$ , a za dekrementiranje (smanjivanje celobrojne promenljive za 1) je  $x--$ .

Petlja for sadrži tri naredbe između zagrada (...); prva naredba sadrži inicijalizaciju, druga je poređenje, a treća naredba koja se izvršava nakon svakog izvršavanja bloka naredbe. Blok naredbi nakon ključne reči for ponavlja se sve dok rezultat poređenja ne postane netačan.

```
#include<stdlib.h>
#include <stdio.h>
int main (int argc, char* argv) {
    int x;
    x = 10;
    if(x == 10) {
        printf("x je tacno 10\n");
        x++;
    } else if (x == 20) {
        printf("x je jednako 20\n");
    } else {
        printf("x nije ni 10 ni 20\n");
    }
    if(x > 10) {
        printf("x je vece od 10\n");
    }
    while(x > 0) {
        printf("x je %d\n", x);
        x = x - 1;
    }
    for(x = 0; x < 10; x++) {
        printf("x je %d\n", x);
    }
    switch(x) {
        case 9:
            printf("x je devet\n");
            break;
        case 10:
            printf("x je deset\n");
            break;
        case 11:
            printf("x je 11\n");
            break;
        default:
            printf("sta li je x?\n");
            break;
    }
    return 0;
}
```

Naredba switch u zavisnosti od vrednosti argumenta koji se nalazi iza reči switch odlučuje na koju će granu case skočiti. U ovom slučaju to će očigledno biti printf ("x je 10\n"); zato što je promenljiva x imala vrednost 10 kada je završena prethodna

for petlja. Naredbe `break` znače da petlja `switch` treba da se završi i da se izvršavanje nastavlja iza nje.

Primetite da je operator poređenja u jeziku C ==, a ne =. Simbol = označava do-delu vrednosti promenljivoj, dok je == operator poređenja.

### Stringovi, nizovi i alokacija memorije

Niz brojeva u jeziku C se definiše kao

```
int y[10];
```

Sledeći program ilustruje korišćenje nizova.

```
#include<stdlib.h>
#include <stdio.h>
int main(int argc, char* argv) {
    int x;
    char y[11];
    for(x = 0; x < 10; x++) {
        y[x] = x * 2;
    }
    for(x = 0; x < 10; x++) {
        printf("element %d je %d\n", x, y[x]);
    }
    return 0;
}
```

Ako niz sadrži znakove (tip `char`), onda se on zove *string*:

```
#include<stdlib.h>
#include <stdio.h>
int main(int argc, char* argv) {
    int x;
    char y[11];
    for(x = 0; x < 10; x++) {
        y[x] = 65 + x * 2;
    }
    for(x = 0; x < 10; x++) {
        printf("element %d je %d\n", x, y[x]);
    }
    y[10] = 0;
    printf("string je %s\n", y);
    return 0;
}
```

Obratite pažnju da string mora da se završava nulom (tj. mora da bude *null terminated*). To znači da poslednji znak u nizu znakova (stringu) mora da bude 0. Naredba `y[10] = 0` postavlja jedanaest elementa niza na nula. To istovremeno znači i da string mora da bude za jedan `char` duži nego što očekujete.

Takođe, obratite pažnju da je prvi element niza `y[0]`, a ne `y[1]` kao u nekim drugim programskim jezicima.

U prethodnom primeru, red `y[11]` rezerviše 11 bajtova za string. Šta bi se desilo kada bi nam zatrebao string od 100 000 bajtova? C dozvoljava da se od jezgra operativnog sistema (kernela) zatraži memorija. To se zove *alociranje memorije*. Svaki netrivialni program za sebe će alocirati memoriju i nema drugog načina da se za program obezbede veći blokovi memorije koje bi koristio. Pogledajmo sledeći primer:

```
#include<stdlib.h>
#include <stdio.h>
int main(int argc, char* argv) {
    int x;
    char* y;
    y = malloc(11);
    printf("%ld\n", y);
    for(x = 0; x < 10; x++) {
        y[x] = 65 + x * 2;
    }
    y[10] = 0;
    printf("string je %s\n", y);
    free(y);
    return 0;
}
```

Naredbom se deklariše promenljiva (broj) koja se zove `y`, a *pokazuje* na neku lokaciju u memoriji. Zvezdica `*` u ovom slučaju označava pokazivač (pointer). Na primer, ako imamo mašinu sa 256 megabajta memorije + swap memorija, onda bi promenljiva `y` imala otprilike isti opseg. Numerička vrednost promenljive `y` štampa se naredbom `printf("%ld\n", y)`, ali ta vrednost programera ne zanima.

Kada program završi sa korišćenjem memorije, mora da je vrati operativnom sistemu pomoću naredbe `free`. Programi koji ne oslobođaju svu alociranu memoriju izazivaju tzv. curenje memorije (*memory leak*).

Alociranje memorije često zahteva prethodni proračun da bi se odredila količina potrebne memorije. U prethodnom primeru alocirali smo prostor za 11 znakova. Pošto svaki znak (tip `char`) zauzima jedan bajt, to nije nikakav problem. Međutim, šta bi se desilo kada bi alocirali prostor za 11 celih brojeva (tip `int`)? Na PC računaru, jedan ceo broj zauzima 4 bajta (32 bita). Za određivanje veličine nekog tipa koristi se ključna reč `sizeof`.

```
#include<stdlib.h>
#include <stdio.h>
int main(int argc, char* argv) {
    int a;
    int b;
    int c;
    int d;
    int f;
    int g;
    a = sizeof(char);
```

```

b = sizeof(short);
c = sizeof(int);
d = sizeof(long);
e = sizeof(float);
f = sizeof(double);
f = sizeof(long double);
printf("%d, %d, %d, %d, %d, %d, %d\n", a, b, c, d, e, f, g);
return 0;
}

```

Ovaj program prikazaće veličine svih tipova u bajtovima. Sada možemo lako da alociramo nizove drugih tipova (a ne samo znakova).

```

#include<stdlib.h>
#include <stdio.h>
int main(int argc, char* argv) {
    int x;
    int* y;
    y = malloc(10 * sizeof(int));
    printf("%ld\n", y);
    for(x = 0; x < 10; x++) {
        y[x] = 65 + x * 2;
    }
    for(x = 0; x < 10; x++) {
        printf("%d\n", y[x]);
    }
    free(y);
    return 0;
}

```

Na mnogim mašinama tip `int` zauzima 32 bita (4 bajta), ali to nikada ne treba uzeti za sigurno. *Uvek koristite ključnu reč `sizeof` za alociranje memorije.*

## Operacije sa stringovima

C programi verovatno češće rade sa stringovima nego sa bilo kojim drugim podacima. Evo programa koji rečenicu deli na reči:

```

#include<stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv) {
    int duzina_reci;
    int i;
    int duzina_recenice;
    char p[256];
    char q;
    strcpy(p, "cao, moje ime je Nikola");
    duzina_recenice = strlen(p);
    duzina_reci = 0;

```

```

for(i = 0; i <= duzina_recenice; i++) {
    if(p[i] == ' ' || i == duzina_recenice) {
        q = malloc(duzina_reci + 1);
        if(q == 0) {
            perror("malloc je zakazao!");
            abort();
        }
        strncpy(q, p + i - duzina_reci, duzina_reci);
        q[duzina_reci] = 0;
        printf("rec: %s\n", q);
        free(q);
        duzina_reci = 0;
    } else {
        duzina_reci++;
    }
}
return 0;
}

```

Ovde uvodimo još tri funkcije standardne C biblioteke. `strcpy` je skraćenica za `stringcopy`. Ova funkcija kopira bajtove sa jednog mesta na drugo redom, sve dok ne stigne do bajta nula (tj. do kraja stringa). Naredba `strcpy(p, "cao, moje ime je nikola")` kopira tekst u niz znakova `p`, koji se zove i odredište za kopiranje.

`strlen` je skraćenica za `stringlength` (dužinu niza). Ova funkcija određuje dužinu niza, što je ovde zapravo broj znakova uvećan za jedan (nulu na kraju znakovnog niza).

Da bismo se kretali kroz rečenicu, potrebna nam je `for` petlja. Promenljiva `i` prati trenutni položaj unutar rečenice.

Naredba kaže da kada najđemo na znak '' , to znači da smo stigli do kraja reči. Znamo i da je kraj rečenice takođe granica za reč, čak i ako na kraju rečenice nema razmaka (a po pravilu ga nema). Znak || znači ili. Sada možemo da alociramo memoriju za tekuću reč i da kopiramo tu reč u memoriju. Za to će nam poslužiti funkcija `strncpy`. Ona kopira string, ali samo do granice od `duzina_reci` znakova (poslednji argument). Poput funkcije `strcpy`, prvi argument je odredište kopije, a drugi argument je izvor odakle se kopira.

Da bismo izračunali položaj početka poslednje reči, koristimo izraz

`p + i - duzina_reci`

To znači da dodajemo `i` na memorijsku lokaciju `p`, a zatim se vraćamo unazad za `duzina_reci` da bismo postavili funkciju `strncpy` na odgovarajući položaj.

Na kraju, string dopunjujemo krajnjom nulom u naredbi. Zatim možemo da odštampamo `q`, oslobodimo korišćenu memoriju naredbom `free` i da nastavimo sa sledećom reči.

## Strukture

U nekim programerskim zadacima dešava se da je potrebno objediniti nekoliko promenljivih različitih tipova da bi im se lakše pristupalo. Tipičan primer je adresa, koja se sastoji od kućnog broja, ulice, poštanskog broja, grada i države. Jezik C podržava pojam strukture koja objedinjuje nekoliko promenljivih različitih tipova: `int`, `char`, `float`, nizove, čak i druge strukture. Promenljive unutar strukture zovu se *članovi strukture*.

Za definisanje strukture koristi se ključna reč `struct`:

```
struct ime_strukture{
    clanovi_strukture
};
```

Evo primera koji definiše strukturu adresu:

```
struct adresa{
    unsigned int kucni_broj;
    char ulica[50];
    int post_broj;
    char grad[50];
    char drzava[50];
};
```

Da bi se kreirala promenljiva tipa strukture, prvi način je da se nakon deklaracije strukture navedu nazivi promenljivih tipa te strukture:

```
struct ime_strukture {
    clan_strukture;
    ...
} instanca_1,instanca_2 instanca_n;
```

Drugi način je da se promenljive tipa strukture deklarišu odvojeno od deklaracije strukture:

```
struct ime_strukture instanca_1,instanca_2 instanca_n;
```

Za pristup članovima strukture koristi se operator tačka (`.`) između imena strukture i podatka člana strukture:

```
ime_strukture.clan_strukture
```

Na primer, da bi se pristupilo imenu ulice u strukturi adresa:

```
struct adresa adr;
adr.drzava = "Srbija";
```

## Rad sa datotekama

U većini programskega jezika, rad sa datotekama (fajlovima) podrazumeva tri koraka: *otvaranje* datoteke, *čitanje* iz datoteke ili *upis* u nju, i na kraju *zatvaranje*. Da

biste operativnom sistemu saopštili da ste spremni za rad sa datotekom, koristite naredbu `fopen`:

```
#include<stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv) {
    int c;
    FILE *f;
    f = fopen("mojtest.c","r");
    if(f ==0){
        perror("fopen");
        return 1;
    }
    for(;;) {
        c = fgetc(f);
        if(c == -1)
            break;
        printf("%c", c);
    }
    fclose(f);
    return 0;
}
```

Ovde je predstavljen nov tip: `FILE *`. Radi se o promenljivoj za rad sa datotekama koja mora biti inicijalizovana naredbom `fopen` pre nego što se upotrebi. Funkcija `fopen` ima dva argumenta: prvi je ime datoteke, a drugi je string koji objašnjava kako želimo da otvorimo datoteku (u ovom slučaju, "r" znači čitanje od početka datoteke, dok bi "w" značilo upis).

Ako je povratna vrednost funkcije `fopen` nula, to znači da otvaranje datoteke nije uspelo. Funkcija `perror` tada prikazuje tekstualnu poruku sa greškom (na primer, `No such file or directory`). Veoma je važno da se povratne vrednosti svih bibliotečkih poziva proveravaju na ovakav način. Takve provere činiće otprilike trećinu svakog C programa.

Komanda `fgetc` čita znak iz datoteke. Bajt po bajt iz datoteke čita se redom sve dok se ne stigne do kraja datoteke, kada funkcija `fgetc` vraća -1. Naredba `break` kaže da u tom slučaju treba trenutno prekinuti petlju `for` i nastaviti izvršavanje u redu `fclose(f)`. Naredbe `break` mogu da se pojave i unutar petlje `while`.

Primetićete da je naredba `for` prazna. To je dozvoljeno u jeziku C i označava beskonačnu petlju (iz koje se obično izlazi naredbom `break` kada je ispunjen neki uslov).

### **Čitanje parametara iz komandne linije u C programima**

Dosad ste se sigurno zapitali čemu služe (`int argc, char* argv`) u funkciji `main`. To su argumenti koji se funkciji `main` prosleđuju iz komandne linije. `argc` je

ukupan broj argumenata (argument count), a `argv` (argument value) je niz stringova za svaki argument. Njihovo prikazivanje je lako:

```
#include<stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv) {
    int i;
    for(i = 0; i < argc; i++) {
        printf("argument %d je %s\n", i, argv[i]);
    }
    return 0;
}
```

### Naredbe `#include` i prototipovi

Na početku svakog programa postojaće jedna ili više naredbi `#include`. Ove naredbe kažu kompjajleru da treba da učita drugi C program. U izvornom jeziku C nema mnogo načina za zaštitu od grešaka; na primer, funkcija `strcpy` mogla bi da se koristi sa jednim, tri ili četiri argumenta i C program bi se i dalje kompjajlirao. Međutim, prouzrokovao bi haos u internoj memoriji i krah programa. Drugi C programi zovu se *datoteke zaglavljia* (header files). Oni sadrže opise kako bi funkcije trebalo da se koriste. Svaka funkcija koju poželite da upotrebite nalazi se u nekom zaglavljiju, a njen opis se zove *prototip funkcije*.

Prototip funkcije se piše na isti način kao i sama funkcija, ali bez tela, tj. kôda. Na primer, prototip funkcije `brojanje_reci` bio bi jednostavno

```
void brojanje_reci(char* ime_fajla);
```

Tačka-zapeta na kraju je neophodna da bi se prototip funkcije razlikovao od sâme funkcije.

Nakon što se definiše prototip funkcije, `gcc` će se snažno usprotiviti svakom pokušaju da se ona upotrebi na način različit od predviđenog (npr. da joj se prosledi manje ili više argumenata ili da joj se proslede argumenti drugačijeg tipa).

Primetićete da smo prilikom korišćenja funkcija za stringove dodali zaglavljje `string.h`. Ako pokušamo da uklonimo red `#include <string.h>` i da ponovo kompjajliramo programe, pojaviće se sledeća greška:

```
mojtest.c:21 warning: implicit declaration of function 'strncpy'
```

koja objašnjava u čemu je problem.

Prototipovi funkcija daju jasnú definiciju načina korišćenja svake funkcije. U opisima Linux i UNIX komandi (tzv. man stranama, Dodatak B) prvo će biti naveden prototip funkcije da bi bilo jasno koji argumenti joj se prosleđuju i kog tipa treba da budu.

## C komentari

C komentar označava se sa `/* komentar */` i može da se proteže u više redova. Sve što se nalazi između znakova `/*` i `*/` kompjajler ignoriše. Trebalo bi da postoji komentar za svaku funkciju, kao i za sve naredbe koje nisu očigledne. Važi pravilo da je program kome je potrebno mnogo komentara loše napisan. Takođe, nikada ne treba komentarisati nešto očigledno, niti objašnjavati *šta* kôd radi umesto *kako* to radi. Ne preporučuje se “crtkanje” između funkcija, pa je bolje pisati ovako:

```
/* vraca -1 ako se desi greska, prihvata pozitivan ceo broj */
int sqr(int x) {
...
nego ovako:
/**********SQR *****/
* x - argument koji se kvadrira *
* povratna vrednost =
* -1 ako se desi greska *
* kvadrat broja x (ako je OK) *
/*********/
int sqr(int x) {
...
```

U jeziku C++ dozvoljen je i komentar `//`, pri čemu se ignoriše sve od dvostrukе kose crte do kraja reda. Ovakvu vrstu komentara prihvata i gcc, ali ga ne bi trebalo koristiti osim ako zaista ne programirate u jeziku C++. Često ćete videti da programeri stavljuju blokove kôda u komentar tako što ih okružuju direktivama `#if 0 ... #endif`; efekat je isti kao kada se stavi običan komentar, ali ovakva notacija omogućuje stavljanje komentara unutar komentara. Na primer:

```
int x = 10;
#if 0
    printf("debug: x je %d\n", x); /* ispis debug informacija */
#endif
y = x + 10;
...
```

## C makroi `#define` i `#if`

Sve što počinje znakom `#` zapravo nije C, već tzv. *preprocesorska direktiva*. C program se prvo propušta kroz preprocesor koji uklanja sve viškove (poput komentara, naredbi `#include` i svega ostalog što počinje znakom `#`). C program može da se učini mnogo čitljivijim ako se umesto konstantnih vrednosti definišu *makroi*. Na primer,

```
#define START_BUFFER_SIZE 256;
```

definiše tekst `START_BUFFER_SIZE` kao konstantu 256. Nakon toga, gde god se u programu pojavi `START_BUFFER_SIZE`, kompjajler će videti broj 256. To je mnogo elegantniji stil programiranja jer kada bismo, na primer, poželeli da promenimo vrednost te konstante iz 256 u neki drugi broj, to bismo mogli da uradimo samo na jednom mestu. `START_BUFFER_SIZE` je takođe jasnije, što čini program razumljivijim.

Kad god vam je potrebna neka konstanta (kao što je 256), treba da je zamenite makroom koji se definiše na početku programa.

Postojanje makroa možete da proverite pomoću direktiva `#ifdef` i `#ifndef`. Direktive `#` zapravo su same za sebe mali programski jezik.

## C biblioteke

Već smo pominjali standardnu biblioteku jezika C. Jezik C sam po sebi ne nudi gotovo ništa; sve što je korisno postoji kao spoljašnja funkcija. Spoljašnje funkcije grupisane su u biblioteke. Standardna C biblioteka pod Linuxom nalazi se u datoteci `/lib/libc.so.6`.

Mnoge funkcije biblioteke imaju odgovarajuće man strane, ali za neke uopšte ne postoji dokumentacija pa će morati da čitate komentare u zaglavlјima. Funkcije je bolje ne koristiti osim ako niste sigurni da se radi o *standardnim* funkcijama (u smislu da su podržane i na drugim sistemima).

Prilično je lako napraviti sopstvenu biblioteku. Recimo da imamo dve datoteke sa nekoliko funkcija koje želimo da kompajliramo u biblioteku. Te dve datoteke su `simple_math_sqrt.c`

```
#include <stdlib.h>
#include <stdio.h>
static int aps_greska(int a, int b) {
    if(a > b)
        return a - b;
    return b - a;
}
int simple_math_isqrt(int x) {
    int rezultat;
    if(x < 0) {

        fprintf(stderr, "simple_math_sqrt: vadijenje korena negativnog broja\n");
        abort();
    }
    rezultat = 2;
    while(aps_greska(rezultat * rezultat, x) > 1) {
        rezultat = (x / rezultat + rezultat) / 2;
    }
    return rezultat;
}
isimple_math_pow.c

#include <stdlib.h>
#include <stdio.h>
int simple_math_ipow(int x, int y) {
    int rezultat;
    if(x == 1 || y == 0)
        return 1;
```

```

if(x == 0 && y < 0) {

    fprintf(stderr, "simple_math_ipow: podizanje nule na negativni stepen\n");
    abort();
}
if(y < 0)
    return 0;
rezultat = 1;
while(y > 0) {
    rezultat = rezultat * x;
    y--;
}
return rezultat;
}

```

Biblioteku bismo želeli da nazovemo `simple_math`. Dobra je praksa da se sve funkcije biblioteke nazovu sa `simple_math_????`. Funkcija `aps_greska` neće se koristiti izvan datoteke `simple_math_sqrt.c` pa smo ispred nje stavili ključnu reč `static`, što znači da je reč o *lokalnoj* funkciji.

Kôd možemo da kompajliramo na sledeći način:

```

gcc -Wall -c simple_math_sqrt.c
gcc -Wall -c simple_math_pow.c

```

Opcija `-c` znači da želimo samo da kompajliramo, tj. ne želimo da dobijemo izvršnu datoteku. Biće generisane datoteke `simple_math_sqrt.o` i `simple_math_pow.o`; to su tzv. *objektne datoteke*.

Sada treba da *arhiviramo* objektne datoteke u biblioteku, što ćemo postići pomoću naredbe ar:

```

ar libsimple_math.a simple_math_sqrt.o simple_math_pow.o
simple_math_pow.o
ranlib libsimple_math.a

```

Komanda `ranlib` indeksira arhivu.

Biblioteka sada može da se upotrebi. Napravićemo datoteku `mojtest.c`:

```

#include <stdlib.h>
#include <stdio.h>
int main(int argc, char* argv) {
    printf("%d\n", simple_math_ipow(4, 3));
    printf("%d\n", simple_math_isqrt(50));
    return 0;
}

```

koju ćemo kompajlirati i pokrenuti pomoću komandi

```
gcc -Wall -c mojtest.c
gcc -o mojtest mojtest.o -L. -lsimple_math
```

Prva naredba kompajlira datoteku `mojtest.c` u `mojtest.o`, a druga naredba se zove *povezivanje programa* (linking). Povezivanjem se `mojtest.o` i biblioteke objedinjuju u jedinstvenu izvršnu datoteku. Opcija `L.` znači da sve biblioteke treba tražiti u tekućem direktorijumu (obično se pregledaju samo direktorijumi `/lib` i `/usr/lib`). Opcija `-lsimple_math` znači da treba napraviti biblioteku `libsimple_math.a` (`lib` i `a` se automatski dodaju). Ova operacija se zove *statičko povezivanje* jer se dešava pre izvršavanja programa, a umeće sve objektne datoteke u izvršnu datoteku.

Uzgred, često se dešava da se sa istim programom statički povezuju brojne datoteke. Tada je redosled povezivanja važan: biblioteka sa najmanje zavisnosti trebalo bi da bude poslednja, jer će se u suprotnom pojavljivati greške u referenciranju simbola.

Možemo i da napravimo zaglavje `simple_math.h` za korišćenje biblioteke:

```
/* racuna kvadratni koren celog broja, odustaje u slučaju greske */
int simple_math_isqrt(int x);
/* racuna stepen celog broja, odustaje u slučaju greske */
int simple_math_ipow(int x, int y);
Na početak datoteke mojtest.c dodaćemo #include "simple_math.h":
#include <stdlib.h>
#include <stdio.h>
#include "simple_math.h"
```

Ovim dodatkom rešićemo se upozoravajućih poruka *implicit declaration of function*. Pošto se zaglavje `simple_math.h` nalazi u istom direktorijumu kao i datoteka `mojtest.c`, ne navodimo zaglavje unutar znakova `<>` već koristimo navodnike.

## C projekti - Makefile

Šta se radi kada malo izmenimo samo jednu datoteku od mnogih, od kojih se sastoji C program (a to se često dešava u programiranju)? Mogli bismo da napišemo skript za kompajliranje i povezivanje, ali bi se pomoću njega sve ponovo kompajliralo, a ne samo izmenjena datoteka. Ono što nam zapravo treba jeste datoteka koja ponovo kompajlira samo objektne datoteke čiji se izvorni kôd promenio; za to služu uslužna alatka `make`. To je program koji kompajlira u skladu sa uputstvima iz datoteke `Makefile` u tekućem direktorijumu. Datoteke `Makefile` sadrže spiskove pravila i zavisnosti koji opisuju kako se dobija program.

Unutar datoteke `Makefile` treba napisati pravila šta od čega zavisi, tako da ih protumači komanda `make`, zajedno sa skript komandama neophodnim za postizanje svakog međucilja.

Prva zavisnost u našem postupku kompajliranja jeste to što `mojtest` zavisi od obe biblioteke, `libsimple_math.a` i objektne datoteke `mojtest.o`. U stilu komande make napisaćemo red u datoteci `Makefile` koji izgleda ovako:

```
mojtest: libsimple_math.a mojtest.o
```

Što znači da datoteke `libsimple_math.a` i `mojtest.o` moraju da postoje i da se ažuriraju pre datoteke `mojtest`. Datoteka `mojtest` se zove ciljna datoteka komande `make`. Ispod ovog reda, moramo da navedemo kako će biti napravljena datoteka `mojtest`:

```
gcc -Wall -o $@ mojtest.o -L. -lsimple_math
```

`$@` je ime sâme ciljne datoteke, što se zamenuje sa `mojtest`. *Prazan prostor ispred komande gcc je tabulator, a ne znakovi za razmak.*

Sledeća zavisnost je to što `libsimple_math.a` zavisi od datoteka `simple_math_sqrt.o` i `simple_math_pow.o`. To je još jedna zavisnost koju treba dodati, zajedno sa skriptom za kompajliranje ciljne datoteke. Kompletno pravilo za `Makefile` je:

```
libsimple_math.a: simple_math_sqrt.o simple_math_pow.o
    rm -f $@
    ar rc $@ simple_math_sqrt.o simple_math_pow.o
    ranlib $@
```

*Ponovo ne zaboravite da se leva marga sastoji od jednog tabulatora, a ne od razmaka!*

Poslednja zavisnost je to što datoteke `simple_math_sqrt.o` i `simple_math_pow.o` zavise od datoteka `simple_math_sqrt.c` i `simple_math_pow.c`. To zahteva dva make pravila za ciljnu datoteku, ali make ima i skraćeni način za navođenje takvih pravila u slučaju većeg broja C datoteka sa izvornim kodom:

```
.c.o:
    gcc -Wall --c -o $*.o $<
```

Što znači da bilo koja potrebna `.o` datoteka može da se dobije od `.c` datoteke sličnog imena pomoću komande `gcc -Wall -c -o $*.o $<`, gde je `$*.o` ime objektne datoteke, a `$<` je ime datoteke od koje `$*.o` zavisi, redom.

Pravila u Makefile mogu da se stavlja proizvoljnim redosledom, pa je najbolje prvo početi od najočiglednijih zbog razumljivosti. Postoji još jedno pravilo koje se stavlja pre svih:

```
all: libsimple_math.a mojtest
```

Ciljna datoteka all: je pravilo koje make pokušava da zadovolji ako se komanda make navede bez argumenata u komandnoj liniji. To znači da će libsimple\_math.a i mojtest biti poslednje dve datoteke koje će biti napravljene, odnosno one se nalaze na vrhu liste zavisnosti.

Makefile takođe ima sopstveni oblik promenljivih okruženja; primetili ste da smo tekst simple\_math koristili u tri pravila. Zbog toga ima smisla definisati makro, za slučaj da nam zatreba da promenimo ime biblioteke.

Naša konačna datoteka Makefile izgleda ovako:

```
# komentari pocinju znakom taraba #
# Makefile za pravljenje libsimple_math.a i programa mojtest
OBJS = simple_math_sqrt.o simple_math_pow.o
LIBNAME = simple_math
CFLAGS = -Wall
all: lib$(LIBNAME).a mojtest
mojtest: lib$(LIBNAME).a mojtest.o
        gcc $(CFLAGS) -o $@ mojtest.o -L. -l$(LIBNAME)
lib$(LIBNAME).a: $(OBJS)
        rm -f $@
        ar rc $@ $(OBJS)
        ranlib $@
.c.o:
        gcc $(CFLAGS) -c -o $*.o $<
clean:
        rm -f *.o *.a mojtest
```

Sada je dovoljno samo da u tekućem direktorijumu otkucamo

```
make
```

i sve datoteke će biti "napravljene".

Primetićete da smo dodali još jednu nepovezanu ciljnu datoteku clean:. Ciljne datoteke mogu se direktno pokrenuti u komandnoj liniji:

```
make clean
```

ova komanda uklanja sve napravljene datoteke.  
Datoteke Makefile osim pravljenja C programa imaju i mnoge druge namene (za sve što treba da se napravi iz izvornih datoteka može da se u pomoć pozove Makefile).



## Dodatak B: Kratak pregled UNIX komandi

Prilikom rada u terminalu UNIX-olikih sistema, treba znati sledeće:

- ▷ UNIX sistemi razlikuju mala i velika slova (tj. oni su case-sensitive). Ako se u komandi promeni samo jedno slovo iz malog u veliko ili obrnuto, to je sasvim drugačija komanda. Isto važi i za datoteke, direktorijume i sintaksu većine podržanih programskih jezika.
- ▷ Neke komande imaju opcije, tzv. flegove. Ista komanda može da se koristi u više varijanti (npr. `ls -l` ili `ls -a`; u ovom slučaju `ls` je komanda za prikaz sadržaja tekućeg direktorijuma, `fleg -l` kaže da treba ga prikaže u dužem formatu, a `fleg -a` traži da se prikažu sve datoteke, čak i one skrivene). Flegovi se mogu kombinovati (tj. `ls -al` je isto što i `ls -a -l`). Između naziva komande i flegova stavljaj se razmak.
- ▷ Većina komandi prihvata opcije `-h` i `--help` koje prikazuju kratak pregled načina njenog korišćenja. Ove opcije su korisne kada poznajete komandu a želite da se podsetite opcija; za komande koje ne poznajete počnite od `man` strana koje ih detaljno opisuju.
- ▷ U istom redu može se zadati nekoliko komandi; u tom slučaju, one se razdvajaju tačkomzapetom, (npr. `ls ; pwd` prikazuje datoteke u tekućem direktorijumu, a nakon toga prikazuje koji je tekući direktorijum).
- ▷ Komande nisu prijateljske, tj. često se dešava da se nakon izvršenja komande u terminalu ne prikaže ništa. To je znak da je komanda uspešno izvršena; sistem će prikazati poruku samo ako se tokom izvršavanja komande desilo nešto nepredviđeno.
- ▷ U opštem slučaju, svaka komanda (npr. `ls`, `cd` itd.) se može posmatrati kao poseban program koji operativni sistem izvršava. Na primer, ako u terminalu otkucate `ls /etc`, program `ls` biće pokrenut u direktorijumu `/etc`. Svaki program zahteva ulaz (u ovom slučaju, to je sadržaj direktorijuma) i proizvodi izlaz (tj. prikazuje rezultate). Često se dešava da programi treba da se pokrenu jedan za drugim. Na primer, mogli bismo da otvorimo datoteku i da nakon toga pokrenemo proverimo pravopis u njoj. Radeći to, mi zapravo povezujemo dva programa, odnosno izlaz prvog programa (otvaranja datoteke) šaljemo kao ulaz drugom programu (proveri pravopisa). Postupak kojim se izlaz jednog programa prosleđuje kao ulaz drugom

- programu zove se nadovezivanje komandi (piping). Može se nadovezati proizvoljan broj komandi. Nadovezivanje se označava simbolom | (pipe). Na primer, komanda `ls | spell | sort` bi napravila spisak datoteka u tekućem direktorijumu, proverila pravopis spiska i na kraju sortirala pogrešno napisane reči koje bi bile prikazane na ekranu (pravilno napisane reči ne bi bile prikazane, jer je ulaz za komandu `sort` izlaz komande `spell`).
- ▷ Džoker znakovi: ? zamenjuje jedan znak ili broj, \* zamenjuje nula ili više znakova i(li) brojeva. Možete ih koristiti na bilo kom mestu; na primer, \*p će pronaći sve reči koje se završavaju na, p\* sve one koje se počinju na p, a tr?ka će pronaći reč traka.
  - ▷ Datoteke čiji nazivi počinju tačkom su skrivene, najčešće zato da bi se izbeglo njihovo nehotično menjanje; možete ih prikazati komandom `ls -a`.
  - ▷ Ako komanda kao argument očekuje naziv datoteke ili direktorijuma, oni joj se mogu proslediti na dva načina. Ako se komanda izvršava u istom direktorijumu u kome se nalazi i datoteka (tj. datoteka se nalazi u tekućem direktorijumu), onda se njen naziv može uneti direktno (npr. `cp moja_datoteka nova_datoteka`). U suprotnom, može se uneti kompletna putanja do datoteke, npr. `cp /home/korisnik/moja_datoteka /home/korisnik/nova_datoteka`. Administratori često koriste i označavanje `./moja_datoteka`, gde je `./` skraćenica za tekući direktorijum. Nazivi datoteka koji počinju kosom crtom / zovu se absolutne putanje, i odnose se na putanju počev od korenskog (root) direktorijuma čija je oznaka /. Nazivi koji ne počinju kosom crtom zovu se relativne putanje i odnose se na putanju u odnosu na tekući direktorijum, odnosno lokaciju u sistemu datoteka gde se trenutno nalazite.
  - ▷ Kada u terminalu otkucate komandu, ona mora da se učita sa diska iz jednog od nekoliko uobičajenih direktorijuma gde se čuvaju komande. Komanda se raspoređuje u stablu direktorijuma u skladu sa tipom, a ne u skladu sa tim kom softverskom paketu pripada. Tako na primer izvršna datoteka programa za obradu teksta može da se nalazi u direktorijumu gde se čuvaju izvršne datoteke, a odgovarajuće datoteke sa fontovima čuvaju se tamo gde su smešteni fontovi drugih programa. Kada u terminalu otkucate komandu sa putanjom, npr. `/bin/cp`, sistem će pokušati da izvrši komandu `cp` iz direktorijuma `/bin`. Ako otkucate samo `cp`, pokušaće da pronađe komandu `cp` u svim poddirektorijumima promenljive okruženja PATH. Da biste videli koja vam je putanja, otkucajte `echo $PATH`. Postoje i komande za prilagođavanje okruženja pomoću kojih možete (trajno ili privremeno) da menjate vrednost promenljive PATH.
  - ▷ Često je potrebno preusmeriti izlaz komande u datoteku; za to se koristi simbol > (na primer, komanda `ls /usr/local/bin > lokalni.programi.txt` preusmerava listu sadržaja direktorijuma `/usr/local/bin` u datoteku `lokalni.programi.txt` (na ekranu se neće prikazati ništa). Ako želite da dopišete rezultat na kraj po-

stojeće datoteke, upotrebite simbol >> (npr. `ls/usr/bin >> svi.programi.txt`). Komande možete da nadovezujete i preusmeravate u isto vreme (npr. `ls | wc -l >> infodir` broji koliko stavki ima u tekućem direktorijumu i upisuje taj broj u datoteku `infodir`).

### Osnovne komande

Komanda	Opis
<code>apropos komanda</code>	Pronalazi odgovarajuće man strane za komandu
<code>cat datoteka</code>	Prikazuje sadržaj datoteke na ekranu
<code>cat datoteka1 datoteka2</code>	Prikazuje datoteku1 i datoteku2
<code>cd</code>	Vraća se u home direktorijum sa bilo kog mesta
<code>cd ..</code>	Pomera se za jedan nivo iznad u stablu direktorijuma
<code>cd /etc</code>	Pomera se u direktorijum /etc u odnosu na korenski (root) direktorijum
<code>cd ~/poddirektorijum</code>	Znak ~ je korisna prečica za home direktorijum
<code>cd Projekti</code>	Pomera se u direktorijum Projekti relativno u odnosu na tekući direktorijum
<code>exit (Ctrl + D)</code>	Završava trenutni proces (obično terminal)
<code>less datoteka</code>	Prikazuje sadržaj datoteke ekran po ekran
<code>logout</code>	Završava UNIX sesiju
<code>ls</code>	Prikazuje spisak datoteka i direktorijuma
<code>ls /</code>	Prikazuje spisak datoteka u korenskom (root) direktorijumu
<code>ls /direktorijum</code>	Prikazuje sadržaj direktorijuma
<code>ls -a</code>	Prikazuje sve datoteke i direktorijume, uključujući i skrivene
<code>ls -l</code>	Prikazuje sadržaj direktorijuma uz dodatne informacije
<code>man komanda</code>	Prikazuje man (manual) strane koje opisuju komandu
<code>more datoteka</code>	Prikazuje sadržaj datoteke ekran po ekran
<code>passwd</code>	Menja lozinku
<code>pwd</code>	Prikazuje putanju i naziv direktorijuma u kome se nalazite
<code>su -username</code>	Prijava vas kao korisnika username bez potrebe da se odjavljujete

**UNIX direktorijumi i njihov sadržaj**

<i>Direktorijum</i>	<i>Sadržaj</i>
/bin	Osnovni programi i komande koje koriste svi korisnici sistema
/boot	Datoteke koje koristi bootloader
/dev	Uredaji (CD-ROM, USB i sl.) i specijalne datoteke
/etc	Datoteke za konfiguraciju sistema i globalna podešavanja
/home	Matični (home) direktorijumi korisnika
/lib	Osnovne deljene biblioteke i moduli jezgra operativnog sistema
/mnt	Tačka gde se montiraju (mount) privremeni sistemi datoteka
/opt	Direktorijum za dopunske softverske pakete
/proc	Jezgro (kernel) i informacije o procesima
/root	Matični (home) direktorijum administratora sistema (root korisnika)
/sbin	Osnovne komande i programi za podizanje sistema
/tmp	Privremene datoteke
/usr/bin	Komande i programi koji nisu toliko značajni za sistem kao oni koji se nalaze u direktorijumu /bin ali se instaliraju zajedno sa operativnim sistemom
/usr/include	Standardna zaglavlja (include file) za C programe
/usr/lib	Biblioteke za programiranje i za instalirane pakete
/usr/local	Većina datoteka i podataka koji su prilagođeni u sistemu
/usr/local/bin	Lokalno programirani ili instalirani paketi
/usr/local/man	Man strane za lokalne pakete
/usr/src	Izvorni (source) kôd standardnih programa
/usr/share	Deljene datoteke (nezavisne od sistema)
/var	Promenljivi podaci: dnevnični sistema (log), privremeni podaci iz programa, poruke e-pošte korisnika i sl.
/var/account	Dnevnični prijavljivanja na sistem
/var/lock	Zaključane datoteke koje su napravili različiti programi
/var/log	Datoteke i direktorijumi sa dnevnicima (log)

## Korišćenje datoteka i direktorijuma

<i>Komanda</i>	<i>Opis</i>
<code>cp postojeća_datoteka nova_datoteka</code>	Kopira postojeću datoteku u novu datoteku
<code>cp -i postojeća_datoteka stara_datoteka</code>	Kopira postojeću datoteku u staru datoteku, ali pritom pita za dozvolu da je prepiše
<code>cp -r /Projekti /shared/Projekti</code>	Kopira direktorijum /Projekti u novo ime /shared/Projekti uz naznaku da je reč o rekurzivnom kopiranju (-r)
<code>find -name izgubljeno -print</code>	Pronalazi datoteku u tekućem direktorijumu ili poddirektorijumima koji se zovu izgubljeno
<code>mkdir Nov_direktorijum</code>	Pravi nov direktorijum koji se zove Nov_direktorijum
<code>mv postojeća_datoteka nova_datoteka</code>	Preimenuje postojeću_datoteku u nova_datoteka
<code>mv -i stara_datoteka nova_datoteka</code>	Preimenuje stara_datoteku u novu_datoteku i zahteva od sistema da traži dozvolu pre nego što prepiše (uništi) stare datoteke
<code>rm -i *</code>	Interaktivno brisanje sadržaja tekućeg direktorijuma, uz traženje dozvole pre brisanja.
<code>rm -i nepotrebna_dat</code>	Interaktivno briše datoteku nepotrebna_dat
<code>rm -ir dan*</code>	Interaktivno brisanje svih datoteka i direktorijuma koji počinju sa dan u tekućem direktorijumu, kao i svih datoteka i direktorijuma u poddirektorijumima koji počinju sa dan
<code>rmdir Direktorijum</code>	Briše prazan direktorijum Direktorijum
<code>touch nova_datoteka</code>	Pravi praznu datoteku nova_datoteka
<code>which komanda</code>	Pronalazi kompletну putanju do komande komanda. Korisno za pronalaženje da li se neka komanda uopšte nalazi na putanji, odnosno ako ih ima više, za saznavanje koja će se od njih izvršiti.

### Upravljanje vlasništvom i dozvolama

Komanda	Opis
chgrp	Menja pridruženost datoteka ili direktorijuma određenoj grupi
chgrp grupa datoteka	Menja pridruženost datoteke datoteka grupi grupa
chgrp -R grupa direktorijum	Rekurzivno menja pridruženost direktorijuma i svih njegovih poddirektorijuma grupi grupa
chmod	Menja dozvole za datoteku
chmod a-w datoteka	Uklanja dozvolu upisa (write, w) u datoteku za sve (all, a)
chmod g+w datoteka	Dodaje dozvolu upisa u datoteku za tekuću grupu
chmod -R go-rwx *	Povlači sve dozvole od svih osim od korisnika za sve datoteke u tekućem direktorijumu, njegovim poddirektorijumima i sadržajem
chmod u=rwx, g=rx, o=r datoteka	Postavlja dozvole za datoteku tako da korisnik može da je čita, upisuje i izvršava, grupa da čita i izvršava (execute, x), a drugi (others, o) samo da je čitaju (read, r)
chmod ugo=*	Povlači sve dozvole za sve u tekućem direktorijumu od svih
chown	Menja vlasništvo nad datotekom ili direktorijumom
chown -R korisnik Direktorijum	Rekurzivno menja vlasništvo Direktorijuma i njegovog kompletног sadržaja i dodeljuje ga korisniku
chown korisnik datoteka	Menja vlasništvo nad datotekom i dodeljuje ga korisniku

## Rad sa datotekama

Komanda	Opis
cmp stara_datoteka nova_datoteka	Poredi staru_datoteku sa novom_datotekom
crypt	Šifruje ili dešifruje datoteku zaštićenu lozinkom
diff -b stara_datoteka nova_datoteka	Pronalazi razlike (uz zanemarivanje praznih redova) između stare_datoteke i nove_datoteke
diff Stari_direktorijum Nov_direktorijum	Pronalazi razlike između Starog_direktorijuma i Novog_direktorijuma
diff -i nova_datoteka stara_datoteka	Pronalazi razlike između nove_datoteke i stare_datoteke uz zanemarivanje malih i velikih sloka
grep izraz datoteka	Pronalazi izraz u datoteci i prikazuje redove koji sadrže izraz
grep -c izraz datoteka	Broji koliko puta se izraz pojavljuje u datoteci
grep -v izraz datoteka	Pronalazi sve redove u datoteci koji ne sadrže izraz
head datoteka	Prikazuje prvih deset redova datoteke
sort datoteka	Sortira datoteku
sort -n datoteka	Sortira datoteku numerički
tail datoteka	Prikazuje poslednjih deset redova datoteke
wc -b datoteka	Broji koliko datoteka ima bajtova
wc datoteka	Broji redove, reči i bajtove u datoteci
wc -l datoteka	Broji redove u datoteci
wc -w datoteka	Broji reči u datoteci
ispell datoteka	Interaktivno proverava pravopis u datoteci

### Prikazivanje informacija o sistemu

<i>Komanda</i>	<i>Opis</i>
<code>df</code>	Prikazuje instalirane diskove, lokaciju montiranih sistema datoteka, količinu iskorišćenog i slobodnog prostora
<code>df /usr/local/src</code>	Pronalazi gde je montiran <code>/usr/local/src</code> i koliko na njemu ima mesta
<code>du</code>	Prikazuje informacije o korišćenju diska (disk usage) u tekućem direktorijumu i njegovim poddirektorijumima
<code>du /home</code>	Pronalazi informacije o korišćenju diska u direktorijumu <code>/home</code>
<code>du -k</code>	Prikazuje informacije o korišćenju diska u kilobajtima
<code>file /usr/bin/pico</code>	Prikazuje kog je tipa datoteka <code>/usr/bin/pico</code>
<code>finger</code>	Prikazuje ko je još prijavljen na sistem
<code>id</code>	Prikazuje numeričku vrednost korisničkog imena prijavljenog korisnika
<code>id korisnik</code>	Prikazuje pripadnost korisnika grupama
<code>quota</code>	Prikazuje da li je korisnik premašio dodeljenu kvotu
<code>quota -v</code>	Prikazuje trenutna podešavanja kvote i njenu iskorišćenost za prijavljenog korisnika
<code>uname</code>	Prikazuje podatke o vrsti UNIX sistema koji se izvršava
<code>uname -a</code>	Prikazuje sve informacije o sistemu, uključujući tip sistema, ime računara, verziju i hardver
<code>w</code>	Prikazuje informacije o drugim korisnicima sistema i njihovim aktivnostima
<code>who</code>	Prikazuje informacije o drugim korisnicima sistema
<code>whoami</code>	Prikazuje sa kojim korisničkim nalogom ste trenutno prijavljeni u sistemu

bc	Kalkulator
cal	Prikazuje kalendar za tekući mesec

### Upravljanje procesima

Komanda	Opis
Ctrl + Z	Suspenduje posao, program ili proces koji se trenutno izvršava
kill %ftp	“Ubija” proces po imenu
kill 16217	“Ubija” proces sa brojem 16127
kill -9 16217	“Ubija” proces sa brojem 16127; flag -9 omogućuje i ukidanje procesa koje obična komanda kill ne “ubija”
nice	Pokreće posao “nežnije”, tj. sporije i sa manjim uticajem na sistem i druge korisnike
ps	Prikazuje trenutne procese prijavljenog korisnika
ps -a	Prikazuje sve procese svih korisnika
ps -f	Prikazuje stablo procesa (forest), tj. procese i njihove međusobne veze
ps -x	Prikazuje sistemske procese (demone)
top	Nadgleda opterećenje sistema i procese u realnom vremenu

### Rad sa kodiranim i(li) komprimovanim datotekama

Komanda	Opis
compress -c datoteka.tar > datoteka.tar.Z	Komprimuje datoteku datoteka.tar sa istim imenom uz dodatak ekstenzije Z uz istovremeno zadržavanje originalne datoteke
compress datoteka.tar	Komprimuje datoteku.tar uz njenu zamenu arhivom kojoj se dodaje nastavak .Z
gunzip arhiva.tar.gz	Raspakuje datoteku arhiva.tar.gz
gzip arhiva.tar	Komprimuje arhivu.tar. Spakovana datoteka zameniće originalnu i imaće dodatak .gz
gzip -d	Dekomprimuje (raspakuje) datoteku

<code>tar -cf nova_datoteka.tar Direktorijum</code>	Pravi novu tar arhivu koja sadrži sve datoteke i direktorijume iz Direktorijuma
<code>tar -v</code>	Opisuje šta se dešava tokom arhiviranja (v, verbose)
<code>tar -xf arhiva.tar</code>	Raspakuje sadržaj arhive.tar
<code>uncompress -c arhiva.tar.Z &gt; arhiva.tar</code>	Dekomprimuje (raspakuje) sadržaj arhive.tar.Z uz zadržavanje originalne arhive
<code>unzip zipovano</code>	Raspakuje zipovano bez posebnog nastavka imena
<code>zip zipovano datoteka</code>	Pravi novu datoteku koja će se zvati zipovano od datoteke

# Indeks

- adresa
  - fizička, 81, 88
  - logička, 80, 83, 88
- alociranje memorije, 265
- alokacija
  - indeksna, 110
  - pomoću blokova, 109
  - susedna, 108
  - ulančana, 109
- apsolutna putanja, 280
- aritmetičko-logička jedinica procesora, 12
- asimetrično multiprocesiranje, 18
- atomic\_t, tip, 170
- baferovanje, 117
- balanser opterećenja, 152
- barijere, 176
- baza podataka, 94
- bit režima, 22
- blejd server, 18
- blok za kontrolu procesa (PCB), 40
- blokovi
  - fiksni, 106
  - nespregnuti, 106
  - spregnuti, 106
- bootstrap program, 34
- brava, 60, 173
- copy-on-write, 141
- curenje memorije, 265
- džepni računari, 20
- datoteka, 25
- datoteke, 93
  - deljenje, 104
  - direktne, 102
  - direktorijumi, 102
  - dodeljivanje imena, 103
- funkcije za upravljanje, 96
- indeksne, 101
- indeksno sekvencijalne, 99
- istovremenih pristupa, 105
- kontrolna lista pristupa, 104
- metode alokacije, 108
- organizacija, 97
- osnovne operacije, 94
- prava pristupa, 104
- prealokacija, 107
- sa gomilanjem podataka, 97
- sa prelivanjem, 99
- sekvensijalne, 98
- sistemi upravljanja, 94
- tabela alokacije, 106
- debager, 30
- deljenje opterećenja procesora, 55
- direktorijumi, 102
  - radni, 103
  - struktura, 102
- diskovi
  - algoritmi za raspoređivanje, 119
  - kašnjenje, 117
  - kašnjenje zbog rotacije, 118
  - keš, 124
  - RAID, 122
  - raspoređivanje, 117
  - vreme prenosa, 118
  - vreme pretraživanja, 118
  - vreme pristupa, 118
- DMA, 17, 114
- drajver, 16
- firmver, 34
- fragmentacija
  - spoljašnja, 79
  - unutrašnja, 78, 106

- funkcija
  - spin\_lock(), 174
  - spin\_trylock(), 173
- Funkcije
  - load\_balance(), 152
- funcije
  - down\_interruptible(), 175
  - down\_trylock(), 175
  - free\_irq(), 162
  - kfree(), 180
  - kmalloc(), 180
  - kmap(), 182
  - kunmap(), 182
  - request\_irq(), 162
  - schedule(), 150
  - task\_rq\_lock(), 148
  - task\_rq\_unlock(), 148
  - this\_rq\_lock(), 148
  - vfree(), 181
  - vmalloc(), 181
- glibc, 160
- hardver, 11
- hrpa, 37
- identifikator procesa (PID), 41
- indeks
  - detaljan, 101
  - parcijalan, 101
- interfejs
  - grafički, 27
  - komandna linija, 27
- interfejs za programiranje aplikacija (API),
  - 29
- interpreter komandi, 129
- izvršna datoteka, 38
- jednoprocесорски sistemi, 17
- jezik C, 259
- kernel, 21, 129
  - instaliranje, 135
  - instaliranje izvornog kôda, 133
  - podešavanje, 135
  - razvojna verzija, 132
  - stabilna verzija, 132
- stablo izvornog kôda, 133
- verzije, 132
- klaster, 18
- kontrolna jedinica procesora, 12
- korisnički interfejs, 27
- kritična sekcija, 58, 169, 172
  - monitori, 65
  - Petersonovo rešenje, 59
  - zaključavanje, 60
- kvantum, 146
  - dinamički, 146
- Linus Torvalds, 128
- Linux
  - i UNIX, 131
  - verzije kernela, 132
- lokalitet referenci, 15
- LRU, 16
- magistrala, 11
- magnetni disk, 13
- Makefile, 274
- makro, 271
- međusobno isključivanje, 58
  - muteks, 62
- mejnfrejm, 19
- memorija, 24
  - adresni registar (MAR), 13
  - DRAM, 13
  - fizička organizacija, 77
  - glavna, 13
  - hijerarhija, 13
  - hrpa, 37
  - keš, 15
  - logička organizacija, 77
  - niska, 182
  - podela na particije, 78
  - reči, 13
  - registar podataka (MDR), 13
  - relokacija, 76
  - segmentacija, 82
  - sekundarna, 13, 25
  - statičko alociranje na steku, 182
  - straničenje, 79
  - tercijarna, 25
  - upravljanje, 75

- upravljanje u Linuxu, 177
- virtuelna, 83
- visoka, 182
- mikrokernel, 33, 130
- MIPS, 260
- MMU, 177
- monitori, 65
  - implementacija pomoću semafora, 66
- multicore procesor, 18
- multiprocesiranje, 12
  - asimetrično, 55
  - simetrično, 55
- multiprocesorski sistemi, 18
- multiprogramiranje, 21
- multitasking, 21
- nadovezivanje komandi (piping), 280
- nedeljive operacije
  - sa bitovima, 171
  - sa celim brojevima, 170
- nedeljivo izvršavanje instrukcija, 61
- nice vrednost, 145, 151
- nit, 41, 46
  - biblioteke, 48
  - jezgra, 48, 143
  - korisnička, 48
  - u Linuxu, 142
- objektne datoteke, 273
- operativni sistem
  - jezgro, 21
  - podizanje, 35
  - slojevi, 31
  - usluge, 27
- operativni sistemi
  - džepni, 26
  - distribuirani, 26
  - Linux, 128
  - mrežni, 26
  - multimedijijski, 26
  - u oblaku, 27
  - UNIX, 127
  - za rad u realnom vremenu, 26
- pajplajn, 12
- paralelni sistemi, 18
- particije
- fiksne veličine, 78
- pokazivač, 265
- pokazivač steka, 12
- polje, 93
- polje sa ključem, 99, 100
- porciјe, 106, 107
  - blokovi, 107
  - ulančane slobodne, 111
- POSIX, 128
- potpuni zastoj, 71, 148, 173
  - otkrivanje, 73
  - sprečavanje, 72
  - uslovi, 72
- povezivanje programa, 274
- prealokacija, 108
- prekid, 16, 22, 130, 161
  - kontrola(), 167
  - linije zahteva (IRQ), 161
  - obrada, 17
- preprocesorska direktiva, 271
- privilegovane instrukcije, 23
- proces, 21, 24, 37
  - blok za kontrolu, 40
  - deljenje, 77
  - deskriptor, 138
  - dete, 44
  - identifikator, 41
  - identifikator(PID), 139
  - interaktivni, 55
  - komunikacija deljenjem memorije, 45
  - komunikacija prosleđivanjem poruka, 45, 130
  - kontekst, 43, 140
  - kreiranje, 44
  - laki (nit), 46
  - pozadinski, 55
  - prelazi, 39
  - prinudna suspenzija, 59
  - raspoređivanje, 51
  - rezidentan, 84
  - roditelj, 44, 138
  - sinhronizacija, 57
  - slika u memoriji, 76
  - stanja, 38
  - stanja u Linuxu, 140
  - teški, 46

- višenitni, 24
- završetak, 144
- procesa
  - dete, 138
- procesi
  - i niti, 137
  - nizovi sa prioritetima, 149
  - rasporedjivanje po prioritetu, 145
  - rasporedjivanje u Linuxu, 144
  - red za čekanje, 147
  - U/I ograničeni, 145
- procesor, 12
  - deljenje, 54
- programski brojač, 12, 24, 37, 41
- prototip funkcije, 270
- radne stanice, 19
- RAID, 122
- RAM, 13
- rasporedivač, 42, 43
  - dugoročni, 43
  - kratkoročni, 43, 52
- rasporedjivanje
  - bez prinudne suspenzije procesa (non-preemptive), 52
  - C-SCAN, 121
  - FCFS, 52
  - FIFO, 119
  - FSCAN, 122
  - kružno (RR), 54
  - LIFO, 120
  - N-step-SCAN, 122
  - po prioritetu, 53, 120
  - redovima u više nivoa, 55
  - sa prinudnom suspenzijom procesa, 145
  - sa prinudnom suspenzijom procesa (preemptive), 52
  - SCAN, 121
  - SJF, 53
  - SRT, 53
  - SSTF, 120
    - u realnom vremenu, 154
  - rasporedjivanje poslova, 21
  - režim rada
    - korisnički, 22
  - sistemski, 22
  - red poslova, 21, 42
  - red spremnih poslova, 42
  - red uređaja, 42
  - register instrukcija, 12
  - registri, 12
  - relativna putanja, 280
  - ROM, 34
  - SCSI, 16
  - segmentacija, 82, 87
    - zaštita i deljenje, 89
  - sekcija
    - izlazna, 58
    - kritična, 58
    - preostala, 58
    - ulazna, 58
  - semafori, 61
    - binarni, 62
    - brojački, 62
    - jaki, 63
    - kreiranje i inicijalizacija, 174
    - muteks, 62
    - operacija signal(), 61
    - operacija wait(), 61
    - spinlok, 62
      - u Linuxu, 174
  - server, 19
  - simetrično multiprocesiranje, 18
  - sinhronizacija
    - klasični problemi, 67
    - problem čitača i pisača, 68
    - problem filozofa za večerom, 69
    - problem proizvodača i potrošača, 67
    - u kernelu Linuxa, 169
  - sistem
    - propusnost, 51
    - vreme odziva, 52
  - sistem datoteka, 93
  - sistem sa vremenskom raspodelom, 21
  - sistemska disk, 35
  - sistemska poziv, 17, 29
    - exec(), 44
    - fork(), 44
    - interfejs, 29
    - wait(), 45

- sistemski pozivi, 155
  - brojevi u Linuxu, 155
  - do\_exit(), 144
  - fork(), 142
  - implementacija, 157
  - označavanje, 156
  - provera parametara, 157
  - upravljač, 156
  - wait(), 144
- spinlock\_t, tip, 173
- spinlok, 63, 172
  - poređenje sa semaforima u Linuxu, 176
- stanje trke, 57, 169
- statičko povezivanje, 274
- stek, 14
- straničenje, 79, 84
  - i segmentacija, 89
  - i zone u Linuxu, 178
- stranice, 79, 177
  - algoritmi zamene, 91
  - okviri, 79
  - oslobađanje, 180
  - pogrešne (page fault), 86
  - politika čišćenja, 92
  - politika smeštanja, 90
  - politika učitavanja, 90
  - politika zamene, 90
  - pomeraj, 80
  - učitavanje, 179
  - veličina, 86
- string, 264
- struktura podataka, 93
- tabela bitova, 110
- tabela segmenata, 83
- tabela stranica, 80, 84
  - invertovana, 86
  - struktura, 85
- tajmer, 23
- thrashing, 84
- TLB, 86
- U/I baferi, 117
- U/I sistem, 113
  - logička struktura, 115
  - organizacija, 113
- U/I uređaji
  - blok orijentisani, 117
  - orientisani na rad sa tokovima, 117
- upravljač prekida, 129, 161
- upravljački program, 16
- virtuelna mašina, 33
- virtuelna memorija, 22, 83
  - i princip lokaliteta, 84
- vremenski odsečak, 145, 146
- zadatak, 137
  - struktura, 139
- zaključavanje, 169
- zakrpa, 133
- zapis, 93
  - kao blokovi, 105
  - sekvensijalna struktura, 96



## Bibliografija

- [1] A. Silberschatz, P.B. Galvin, G. Gagne, *Operating System Concepts, Seventh Edition*, John Wiley and Sons, 2005.
- [2] W. Stallings, *Operating Systems: Internals and Design Principles, Fifth Edition*, Prentice Hall, 2005.
- [3] A.S. Tanenbaum, A.S. Woodhull, *Operating Systems: Design and Implementation, Third Edition*, Prentice Hall, 2006.
- [4] W.S. Davis, T.M. Rajkumar, *Operating Systems: A systematic view, Sixth Edition*, Addison-Wesley, 2004.
- [5] R. Love, *Linux Kernel Development, Second Edition*, Novell Press, 2005.
- [6] J. Bacon, T. Harris, *Operating Systems: Concurrent and distributed software design*, Addison-Wesley, 2003.
- [7] B. Đorđević, D. Pleskonjić, N. Maček, *Operativni sistemi: teorija, praksa i rešeni zadaci*, Mikro knjiga, 2005.
- [8] G. Coulouris, J. Dollimore, T. Kindberg, *Distributed Systems: Concepts and design, Fourth Edition*, Addison-Wesley, 2005.
- [9] P. Sheer, *Linux: Rute User's Tutorial and Exposition*, Prentice Hall, 2001.
- [10] D. Ray, E. Ray, *UNIX and Linux: Visual Quickstart Guide, Fourth Edition*, Peachpit Press, 2009.