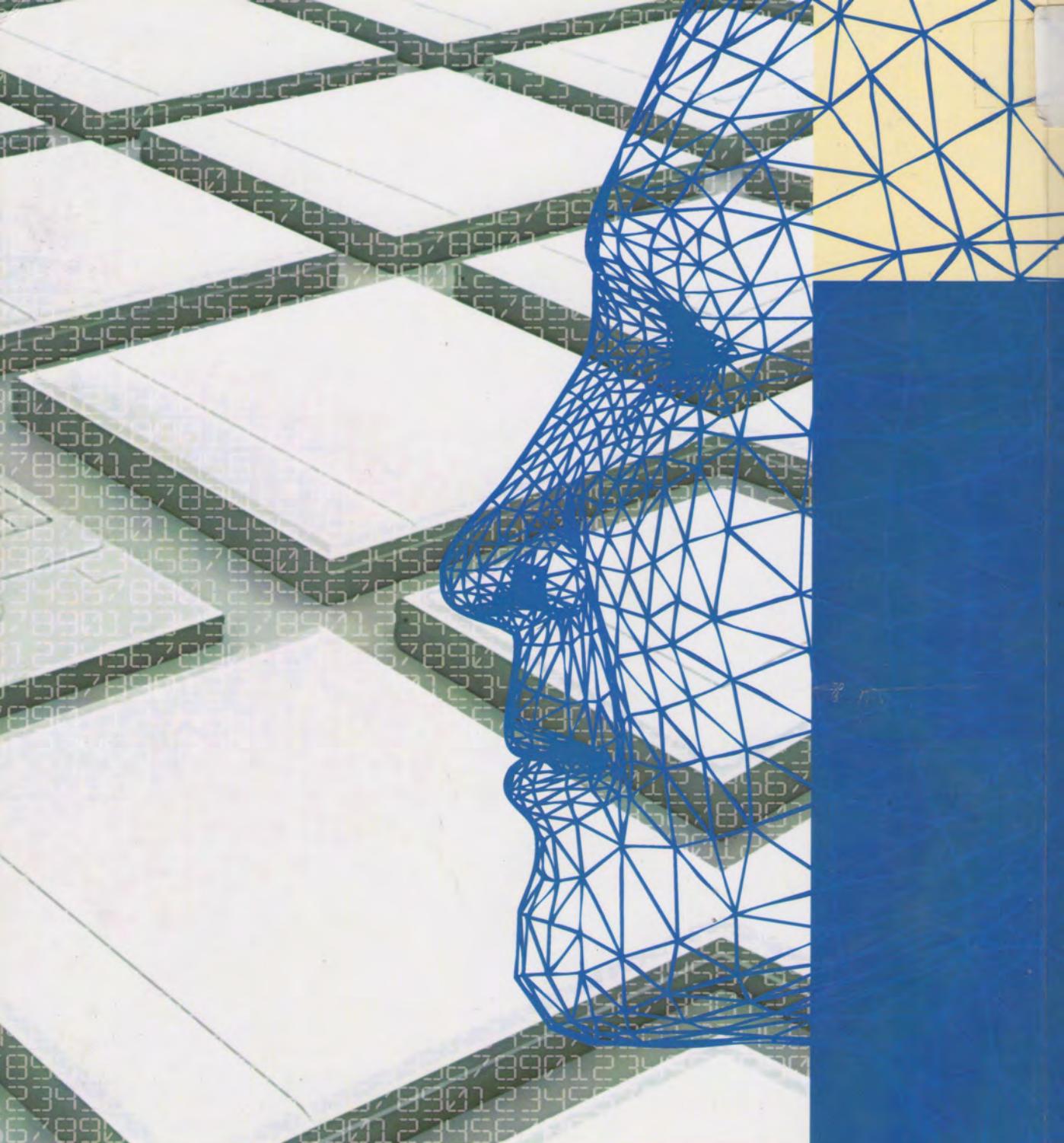


Leo Budin
Marin Golub
Domagoj Jakobović
Leonardo Jelenković

Operacijski sustavi





ISBN 978-953-197-610-7



9 789531 976107

Uporabu ovog sveučilišnog udžbenika odobrio je Senat Sveučilišta u Zagrebu.
(Klasa: 032-01/09-01/88, Ur. broj: 380-04/ 38-10-4 od 15. veljače 2010.)

Intelектualno je vlasništvo, poput svakog drugog vlasništva, neotuđivo, zakonom zaštićeno i mora se poštovati. Nijedan dio ove knjige ne smije se preslikavati niti umnažati na bilo koji način, bez pismenog dopuštenja nakladnika.

CIP zapis dostupan u računalnom katalogu
Nacionalne i sveučilišne knjižnice u Zagrebu
pod brojem 730213.

ISBN 978-953-197-610-7

**Leo Budin
Marin Golub
Domagoj Jakobović
Leonardo Jelenković**

OPERACIJSKI SUSTAVI

1. izdanje

Zagreb, 2010.

© akademik Leo Budin, prof. emer.,
doc. dr. sc. Marin Golub,
doc. dr. sc. Domagoj Jakobović,
doc. dr. sc. Leonardo Jelenković, 2010.

Urednica
Sandra Gračan, dipl. inž.

Recenzenti
prof. dr. sc. Ignac Lovrek
prof. dr. sc. Goran Martinović
prof. dr. sc. Siniša Srbljić

Lektorica
Antonija Vidović, prof.

Crteži, slog i prijelom
Nataša Jocić, dipl. inž.

Dizajn ovitka
Julija Vojković

Nakladnik
ELEMENT d.o.o., Zagreb
www.element.hr
element@element.hr

Tisk
ELEMENT d.o.o., Zagreb

Predgovor

Operacijski je sustav sastavni dio svakog današnjeg računalnog sustava. Pojednostavljeni bi se moglo reći da je operacijski sustav skup programskih proširenja računalnog sklopovlja koji potpomaže izvođenje raznovrsnih operacija¹ potrebnih za izvođenje korisničkih programa.

Moglo bi se, nadalje, reći da operacijski sustav ima dva osnovna zadatka, i to:

- omogućivanje što prikladnije uporabe računala,
- omogućivanje što djelotvornijeg iskorištavanja svih sklopovskih i programske komponenti računalnih sustava.

Svaki se korisnik računala već u prvom susretu s računalom, pri pokušaju pokretanja nekog programa susreće s operacijskim sustavom. U današnje se vrijeme taj susret odvija u obliku dijaloga u kojem se s pomoću *slikovnog korisničkog sučelja* (engl. *Graphical User Interface – GUI*) na interaktivni način (s pomoću tipkovnice ili miša) mogu odabratne neke ponuđene aktivnosti. Obično se takvim početnim dijalogom pokreće neki od pohranjenih *primjenskih* ili *aplikacijskih programa*² koji nam služe za obavljanje nekih pojedinačnih poslova ili nakupine poslova. Primjenske programe pokreću i upotrebljavaju korisnici računalnih sustava i stoga se oni nazivaju i *korisničkim programima* (engl. *User Program*).

Nakon pokretanja nekog od primjenskih programa korisnik sudjeluje u njegovu izvođenju s pomoću sučelja koje je izgrađeno unutar tog programa. Primjerice, ako je pokrenut program za pisanje i obradu tekstova, sučelje tog programa preobličit će računalo u vrlo napredan stroj za pisanje koji omogućuje trajno čuvanje napisanog teksta, kao i njegovo raznovrsno preoblikovanje. Pogled korisnika na računalo odvija se kroz *sučelje programa* koje omogućuje provođenje svih potrebnih aktivnosti za obradu teksta i možemo reći da je računalo pretvoreno u *virtualni pisači stroj*.

S druge strane, programer koji priprema programe pokrenut će kroz sučelje prema operacijskom sustavu najprije program za pripremu teksta programa i nakon toga će kroz sučelje tog programa utipkavati redom instrukcije svojeg programa. Nakon što završi utipkavanje, pokrenut će program za prevođenje svojeg programa u strojni oblik i tom će prilikom ispravljati pogreške koje je načinio prilikom pisanja programa. Nakon uspješnog prevodenja potrebno je još ispitati i ispravnost djelovanja programa, što se čini uslužnim programom za ispitivanje novih programa. Programer, prema tome, pri pripremi novih programa koristi nekoliko programa koji čine tzv. programsku potporu ili *radnu okolinu za razvoj novih programa* (engl. *Development Environment*).

Raznovrsne druge primjene zahtijevaju pripremu odgovarajućih uslužnih programa koji će računalni sustav preobraziti u odgovarajuću radnu okolinu. Osim dviju već spomenutih primjena proširene su i primjene računala za:

¹ Engleski naziv *Operating System* pogrešno je prevoditi kao *operativni sustav* jer se time mijenja smisao naziva. Pridjev *operativni* označava djelotvornost, pa bi se moglo reći da operacijski sustav u jednom trenutku može biti operativan (djeluje) ili neoperativan (ne djeluje).

² Atribut *aplikacijski* dolazi od latinskog *applicare* – primjeniti. U engleskom se jeziku za primjenske programe koristi naziv *Application program*.

- pristup do baza podataka u različitim informacijskim sustavima;
- pripremu teksta;
- razvoj i ispitivanje novih primjenskih programa;
- ostvarenje komunikacije između korisnika;
- obavljanje znanstveno-istraživačkih izračunavanja;
- inženjerska projektiranja u kojima je osobito izražena vizualizacija rezultata;
- nadziranje i upravljanje tehničkih objekata i sustava;
- kućne i osobne potrebe;
- igru i zabavu.

Računala su jedinstvene tehničke naprave upravo zbog toga što se mogu vrlo brzo i prividno jednostavno preobraziti u virtualne radne okoline za raznovrsne primjene. Jedan te isti fizički stroj će se samo jednostavnom zamjenom aktivnog programa pretvoriti iz stroja za pisanje u stroj za crtanje ili u stroj zaigranje neke igre.

Ostvarenje svih tih raznolikih radnih okolina mora podržavati operacijski sustav. Operacijski sustav, s jedne strane, mora omogućivati što djelotvornije *stvaranje* novih programa, a s druge strane, omogućivati što djelotvornije *izvođenje* tih istih programa.

Za pripremanje programa koristi se uobičajeno neki viši programski jezik. Mnoge operacije koje želimo izazvati programom teško je izraziti jednostavnim instrukcijama. Osim toga, programiranje pojedinih operacija koje pokreću pojedine ulazno-izlazne naprave zahtijevalo bi detaljno poznavanje sklopolja. Zbog toga se u okviru operacijskog sustava priprema zbirka funkcija operacijskog sustava koje se mogu pozivati iz primjenskih programa. Ta se zbirka funkcija naziva *sučeljem operacijskog sustava za primjenske programe*. Prema engleskom nazivu za to se sučelje upotrebljava kratica *API* (engl. *Application Programming Interface* – sučelje za programiranje primjenskih programa). To je sučelje praktički nevidljivo velikoj većini korisnika računala, ali čini vrlo važan sastavni dio operacijskog sustava.

Pri prvom susretu s računalom upoznajemo se, u pravilu, samo sa sučeljem operacijskog sustava prema čovjeku. Ako želimo koristiti računalo samo za neke određene jednostavne poslove, ne moramo ni znati mnogo više od toga. Upravo je i prednost prikladnog korisničkog sučelja i jednostavnosti operacija to što čak i povremeni korisnik može jednostavno pokrenuti i upotrebljavati računalo. Za takvo upoznavanje operacijskih sustava dovoljno je dobro proučiti neki dobar priručnik.

Međutim, za naprednije i zahtjevnije korisnike važno je detaljnije poznavanje djelovanja računalnih sustava i operacija s pomoću kojih se može djelovati na ponašanje sustava. Oni ljudi koji pripremaju, održavaju i modificiraju programe, ili žele s postojećim programima postići učinkovitije djelovanje programa, moći će to bolje činiti ako poznaju osnovne zasade na kojima počiva izgradnja današnjih operacijskih sustava. Za takve je čitatelje pripremljen ovaj udžbenik.

Sadržaj udžbenika oblikovan je na temelju dugogodišnjeg nastavnog iskustva autora. Gradivo povezano s operacijskim sustavima predavalо se na Elektrotehničkom fakultetu Sveučilišta u Zagrebu od 1980. godine u okviru predmeta *Sistemski programi i Programski sustavi za rad u realnom vremenu*. Danas se na Fakultetu elektrotehnike i računarstva

na studiju Računarstva predaje predmet *Operacijski sustavi*. Predmet *Operacijski sustavi* predaje se i na studiju matematike Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu, na Fakultetu organizacije i informatike u Varaždinu, te na Sveučilištu u Dubrovniku, a isti predmet predavao se i na studiju matematike Pedagoškog fakulteta Sveučilišta Josip Juraj Strossmayer u Osijeku.

Pokazalo se da pristup izučavanju gradiva povezanog s operacijskim sustavima koje se nudi ovim udžbenikom omogućuje studentima sustavno upoznavanje s načelima izgradnje i djelovanja operacijskih sustava. S obzirom na to da je operacijski sustav nadogradnja računalnog sklopolja, izlaganje obuhvaća i ona osnovna svojstva računalnog sklopolja koja određuju ponašanje sustava i objašnjavaju motive za pojedine načine ostvarenja njihove programske nadgradnje.

U novije vrijeme povećanje moći računalnih sustava ostvaruje se korištenjem paralelnih arhitekturnih rješenja (npr. višestruki procesori, engl. *multicore processors*), a ne više samo povećanjem frekvencije radnog takta procesora. Zbog toga je tim važnije poznавanje načela rada i takvog sklopolja i mogućnosti njegova iskorištenja kroz sučelje operacijskog sustava jer će se jedino tada moći izgraditi kvalitetna programska podrška koja će moći učinkovito iskoristiti raspoložive resurse.

Gdje god je to u tekstu potrebno i moguće, uvode se prikladni modeli, koji s jedne strane pomažu u objašnjavanju funkcija sustava, a s druge strane omogućuju približno izračunavanje nekih kvantitativnih parametara sustava.

Gradivo je podijeljeno u dvanaest poglavlja. U *prvom se poglavljju* objašnjava hijerarhijski pristup u izgradnji i analizi složenih sustava. Objavljaju se načela slojevite izgradnje sustava i prednosti uporabe objektno usmjerene izgradnje sustava. Opisuje se moguća podjela računalnih sustava na tipične slojeve koji motiviraju strukturiranje sljedećih nekoliko poglavlja.

U *drugom se poglavljju* na modelu rudimentarnog računala sastavljenog od procesorsko-spremničkog podsustava obrazlažu osnovna svojstva procesora i uvodi pojam dretve koja određuje osnovnu razinu zrnatosti u razmatranju operacijskih sustava. Obrazlažu se važne uloge programske brojila, registra kazaljke stoga i načini poziva procedura kao osnovnih mehanizama za organizaciju izvođenja programa.

Treće poglavje uvodi u razmatranje ulazno-izlazne naprave. Objavljava načine priključivanja ulazno-izlaznih naprava, programsko obavljanje ulazno-izlaznih operacija, te prekidni rad kao osnovni mehanizam sinkronizacije procesora sa zbivanjima u vanjskim napravama. Razmatraju se motivi za uvođenje pristupnih sklopova koji neposredno pristupaju spremniku, kao i načela izgradnje višeprocesorskih sustava.

U *četvrtom se poglavljju* objavljaju pojmovi poslova i njihove pretvorbe u procese. Objavljaju se načela višezadaćnosti i njezina ostvarenja s pomoću višedretvenih procesa te pojmovi zavisnosti i nezavisnosti i međusobnog isključivanja. Obrazlažu se mogući načini ostvarivanja međusobnog isključivanja u jednoprocesorskim i čvrsto povezanim višeprocesorskim sustavima.

U *petom se poglavljju* uvodi model jednostavne jezgre (mikrojezgre) operacijskog sustava za jednoprocesorski sustav. Opisana je struktura podataka modela jezgre i njezine procedure koje ostvaruju međusobno isključivanje i sinkronizaciju dretvi. Obrazložen je

mehanizam ostvarivanja ulazno-izlaznih operacija, binarnog i općeg semafora, te načini ostvarivanja kašnjenja. Na kraju se objašnjava mogućnost ostvarenja jezgre za višeoperatorski čvrsto povezani sustav.

Sesto poglavlje bavi se osnovnim mehanizmima komunikacije između dretvi. Opisuju se osnovni mehanizmi razmjene poruka između dretvi koje se odvijaju u istom procesu. Opisuje se mogućnost nastajanja potpunog zastoja pri uporabi jednostavnih mehanizama uzajamnog isključivanja. Uvodi se pojam monitora, te obrazlažu načini ostvarenja suradnje dretvi nadzirane monitorskim funkcijama.

U *sedmom se poglavlju* opisuju načini vrednovanja ponašanja sustava. Obrazlažu se mogućnosti ocjene ponašanja sustava u potpuno determinističkim uvjetima, te osnovni model ocjene ponašanja sustava uz nedeterminističko opterećenje. Pritom se ukratko izvode osnovni izrazi za prosječno zadržavanje poslova u sustavu, kao i prosječni broj poslova u sustavu ako događaji dolazaka novih poslova podliježu Poissonovoj razdiobi, a trajanje je poslova podvrgnuto eksponencijalnoj razdiobi. Ovo se poglavlje u prvom čitanju može preskočiti, a neki krugovi čitatelja mogu ga i potpuno zanemariti.

Osmo poglavlje bavi se problemom korištenja radnog spremnika i mogućnostima ostvarenja dovoljno velikog spremničkog prostora za pojedine procese. S obzirom na to da se veliki prostor često ostvaruje na manjem fizičkom spremničkom prostoru, i to tako da se neki trenutačno nepotrebni sadržaji odlažu na vanjski pomoći spremnik, u poglavlju se detaljno opisuju svojstva današnjih vanjskih spremnika. Posebice se ukazuje na vremenske parametre vanjskih spremnika i utvrđuje da oni mogu bitno utjecati na ponašanje sustava. Posebice se obrazlaže ostvarenje virtualnog spremnika stranicenjem.

U *devetom poglavlju* obrađena su osnovna svojstva datotečnog podsustava. Ukazuje se na osnovne mehanizme pohranjivanja datoteka i na osnovne operacije s datotekama. Opisuju se načini smještanja datoteka na diskove, koji umanjuju problem fragmentacije prostora na diskovima.

Sljedeće, *deseto poglavlje*, obrađuje osnovne postavke mrežnih i raspolijeljenih sustava. Ustanavljuje se da je razmjena poruka osnovni način komuniciranja između procesa. Neupoštevanje sklopovske podloge za ostvarenje uzajamnog isključivanja i problemi s ostvarenjem vremenskog uredenja otežavaju problem sinkronizacije. Opisuju se načini i problemi ostvarenja poziva udaljenih procedura i prividnog zajedničkog spremničkog prostora.

U *jedanaestom se poglavlju* obrađuju problemi sigurnosti računalnih sustava, i to posebice raspolijeljenih sustava. Opisani su mogući načini ugrožavanja sigurnosti, kao i metode za borbu protiv pokušavanja neovlaštenog pristupa i korištenja računalnih sredstava. Uvode se i osnovni postupci kriptiranja i protokola koji koriste kombinaciju kriptiranja s javnim i tajnim ključevima te postupci utvrđivanja autentičnosti sudionika u komunikaciji i digitalnog potpisivanja dokumenata.

Na kraju, u *dvanaestom poglavlju*, govori se o višediskovnim zalihosnim spremnicima. Objašnjava se postupak modeliranja zalihosnih sustava uz pomoć Markovljevih lanaca. Navedeni su i opisani osnovni načini zalihosne organizacije diskova.

U udžbeniku se za ilustraciju koriste primjeri iz nekih varijanti operacijskih sustava tipa *UNIX* organiziranih oko mikrojezgri te iz operacijskog sustava *Windows*. Udžbenik je

zamišljen tako da može poslužiti kao podloga za detaljnije izučavanje dokumentacije pojedinih konkretnih operacijskih sustava. Usvojeno gradivo iz ovog udžbenika omogućit će čitatelju da s razumijevanjem upotrebljava operacije koje mu pojedini operacijski sustav nudi te da odabere one mogućnosti operacijskih sustava koje će poboljšati ponašanje i djelotvornost njegovih konkretnih primjenskih programa. Oni čitatelji koji budu pripremali vlastite programe moći će s mnogo više sigurnosti odabirati funkcije koje operacijski sustav nudi kroz svoje sučelje primjenskih programa.

Autori su u pripremi ovog udžbenika posebice vodili računa o tome da je razdoblje napretka mikroelektroničke tehnologije u kojem se brzina rada procesora udvostručavala svakih 18 do 24 mjeseca pri kraju. Prema tome, daljnje povećavanje performansi može se postići jedino uvišestručavanjem računalnog sklopolja (prvenstveno procesora). Zbog toga se u udžbeniku naročita pažnja posvećuje upravo paralelizmu i problemima koji s tim u vezi nastaju. Vjerujemo da ćemo čitateljima udžbenika olakšati razumijevanje novih trendova u izgradnji operacijskih sustava, a nekim i aktivno djelovanje u toj izgradnji.

Nastavnici mogu udžbenik koristiti na različite načine. Način uporabe ovisi o predznanju studenata. Općenito se za proučavanje operacijskih sustava prepostavlja neko osnovno predznanje o arhitekturi računalnih sustava, kao i određeno iskustvo u primjeni računalnih sustava. I ovaj će se udžbenik lakše čitati ako su ti preduvjeti ispunjeni. Međutim, u tekstu se ovog udžbenika, na mjestima gdje je to potrebno, objašnjavaju osnovna svojstva sklopovske podloge na koju se nadograđuju programski ostvareni mehanizmi. Time je omogućeno da se udžbenik čita kao jedna zaokružena cjelina. Jasno je da se za produbljivanje pojedinih znanja treba posegnuti za odgovarajućom literaturom iz područja arhitekture. Isto tako, razmatranje osnovnih svojstava mrežnih i raspodijeljenih operacijskih sustava nije moguće bez poznavanja načela međuračunalne komunikacije. Stoga se u onim poglavljima ovog udžbenika koja se bave tom problematikom navode i osnovne postavke komunikacijskih mehanizama, no čitatelj koji želi detaljnije izučavati računalne mreže i komunikacijske sustave trebat će posegnuti za odgovarajućim udžbenicima iz tog područja. Oni čitatelji koji nisu zainteresirani za kvantitativno vrednovanje ponašanja sustava mogu u potpunosti preskočiti sedmo poglavlje, a i one dijelove pojedinih poglavljja u kojima se procjenjuje ponašanje sustava.

Iako je udžbenik prilagođen uporabi u visokoškolskoj nastavi, on se može koristiti (barem neki njegovi dijelovi) i u nekim specijaliziranim srednjim školama, kao i na tečajevima trajnog obrazovanja, gdje će poslužiti kao koncepcijska podloga za izučavanje određenih konkretnih operacijskih sustava.

Nastanak ovog udžbenika bio je dugotrajan proces tijekom kojeg su mnoge osobe na različite načine pridonijele njegovu osmišljavanju i oblikovanju. Svima njima autori najljepše zahvaljuju. Sve njih nije moguće nabrojiti, ali neke želimo posebno istaknuti svjesni opasnosti da pritom možemo nekoga i zaboraviti. Zahvaljujemo se u prvom redu profesorima Stanku Turku i Urošu Perušku na početnom poticaju za pripremu ovakvog udžbenika još osamdesetih godina prošlog stoljeća prilikom uvođenja smjera Računska tehnika na tadašnjem Elektrotehničkom fakultetu Sveučilišta u Zagrebu. Gradivo je nastajalo postupno i za njegov današnji oblik zasluzni su mnogi koji su, ili kao asistenti ili kao nastavnici, na razne načine sudjelovali u oblikovanju i provođenju nastave, ne samo na Fakultetu elektrotehnike i računarstva (Elektrotehničkom fakultetu) u Zagrebu, već i na drugim visokoškolskim

ustanovama diljem Hrvatske. Doprinos svakog pojedinca teško je kvantificirati pa ih navodimo abecednim redom. To su: Felice Balarin, Davor Cihlar, Sven Gotovac, Tomislav Grčanac, Darko Fisher, Zoran Kalafatić, Dragutin Kermek, Goran Omrčen Čeko, Siniša Srbljić, Vlado Stanisljević, Drago Vuković, Vlado Sruk i Ivan Zoraja. Svima njima najljepše zahvaljujemo. Profesor Dalibor Vrsalović upozorio nas je da bolje obradimo neke teme povezane s raspoređivanjem, i to posebice temu o inverziji prioriteta koja je često potpuno zapostavljena, za što mu posebice zahvaljujemo. Recenzentima profesorima Ignacu Lovreku, Siniši Srbljiću i Goranu Martinoviću zahvalni smo na primjedbama koje su pridonijele znatnom poboljšanju teksta. Profesor Ignac Lovrek posebno nas je upozorio da istaknemo teme koje podupiru današnji trend razvitka operacijskih sustava usmjerenog na obuhvaćanje mogućnosti koju nude današnji višestruki procesori na čemu smo mu posebno zahvalni. Profesor Vlado Mikuličić dobiva našu zahvalu jer je njegovom zaslugom uspješno i brzo proveden postupak recenziranja i odobravanja ove knjige kao udžbenika Sveučilišta u Zagrebu. Konačno, ali ne i najmanje važno, zahvaljujemo na ljubaznosti i susretljivosti u izdavačkoj kući *Element*, i osobite zahvale urednici Sandri Gračan, Nataši Jocić koja je strpljivo i s pažnjom pripremila slogan, crteže i prijelom, lektorici Antoniji Vidović te Juliji Vojković koja je načinila dizajn ovitka.

Zagreb, 1. veljače 2010.

Autori

Sadržaj

1. Uvod	1
1.1. Prvi susret s operacijskim sustavom.....	1
1.1.1. Zadaci operacijskog sustava.....	1
1.1.2. Odvijanje tipičnog posla u računalnom sustavu	2
1.2. Hjерархиjska izgradnja operacijskog sustava.....	4
1.3. Načini izučavanja operacijskih sustava	6
2. Model jednostavnog računala	9
2.1. Von Neumannov model računala i načini njegova ostvarenja	9
2.1.1. Funkcijski model računala	9
2.1.2. Sabirnička građa računala	10
2.1.3. Radni ili središnji spremnik računala	11
2.1.4. Rudimentarno računalo, radni spremnik.....	13
2.1.5. Rudimentarno računalo – procesor.....	15
2.1.6. Brzina rada procesora, priručni spremnik	19
2.1.7. Instrukcijski skup procesora	20
2.2. Instrukcije za poziv potprograma i povratak iz potprograma	24
2.2.1. Načini razmjene podataka između potprograma i programa	28
2.2.2. Instrukcijska dretva	29
2.3. Računalni proces	30
3. Obavljanje ulazno-izlaznih operacija, prekidni rad	33
3.1. Prikљučivanje ulazno-izlaznih naprava	33
3.2. Prenošenje pojedinačnih znakova, prekidni rad procesora	35
3.2.1. Prenošenje znakova radnim čekanjem.....	35
3.2.2. Prekidni način rada procesora	39
3.3. Podsistav za prihvat prekida	42
3.3.1. Najjednostavniji oblik podsustava za prihvatanje više prekida	42
3.3.2. Podsistav za prihvat prekida razvrstanih po prioritetima s najjednostavnijim sklopovljem	44
3.3.3. Sklopovska potpora za ostvarenje višestrukog prekidanja	49
3.3.4. Prekidi generirani unutar procesora, poziv sustavskih potprograma.....	53
3.4. Prenošenje blokova znakova, sklopovi s neposrednim pristupom spremniku	54

3.5. Čvrsto povezani višeprocesorski sustav	56
3.6. Sabirnički sustavi stvarnih računala	59
4. Međusobno isključivanje u višedretvenim sustavima	61
4.1. Programi, procesi i dretve	61
4.2. Višedretveno ostvarenje zadataka, sustav podzadataka	62
4.2.1. Zadaci i podzadaci	62
4.2.2. Model višedretvenosti	64
4.2.3. Sustav dretvi	66
4.2.4. Međusobno isključivanje	68
4.2.5. Cikličke dretve	69
4.3. Ostvarenje međusobnog isključivanja dviju dretvi	69
4.3.1. Prvi pokušaj	70
4.3.2. Drugi pokušaj	73
4.3.3. Treći pokušaj	73
4.3.4. Četvrti pokušaj	74
4.3.5. Peti pokušaj	75
4.3.6. Šesti pokušaj – Dekkerov postupak	76
4.3.7. Petersonov postupak međusobnog isključivanja dviju dretvi	78
4.4. Međusobno isključivanje većeg broja dretvi – Lamportov protokol	80
4.5. Sklopovska potpora međusobnom isključivanju	83
5. Jezgra operacijskog sustava	89
5.1. Radno okruženje za izvođenje dretvi – jednostavni model jezgre	89
5.2. Struktura podataka jednostavnog modela jezgre – stanja dretvi	91
5.2.1. Lista postojećih dretvi, pasivno stanje	93
5.2.2. Aktivno stanje dretve	93
5.2.3. Pripravno stanje dretve, red pripravnih dretvi	93
5.2.4. Blokirana stanja dretvi	97
5.2.5. Prikaz mogućih stanja dretvi	100
5.3. Jezgrine funkcije	102
5.3.1. Ulazak u jezgru i izlazak iz jezgre	102
5.3.2. Funkcije za binarni semafor	103
5.3.3. Funkcije za opći semafor	106
5.3.4. Funkcije za ostvarivanje kašnjenja	109
5.3.5. Funkcije za obavljanje ulazno-izlaznih operacija	110

5.4. Ostvarenje jezgre u čvrsto povezanom višeprocesorskom sustavu	111
5.5. Objektni model jezgre operacijskog sustava	115
6. Međudretvena komunikacija i koncepcija monitora	117
6.1. Problem proizvodača i potrošača	117
6.1.1. Međudretvena komunikacija s pomoću neograničenog spremnika	118
6.1.2. Međudretvena komunikacija s pomoću ograničenog spremnika	121
6.1.3. Međudretvena komunikacija s pomoću reda poruka	123
6.1.4. Sinkronizacija dretvi	126
6.2. Potpuni zastoj	127
6.2.1. Uvjeti za nastajanje potpunog zastoja	128
6.3. Koncepcija monitora	133
6.3.1. Jezgrine funkcije za ostvarivanje monitora	134
6.3.2. Primjeri izgradnje monitora	137
6.3.3. Suvremenije ostvarenje monitora	142
6.4. Inverzija prioriteta	145
6.4.1. Mogući problemi pri sinkronizaciji dretvi	145
6.5. Izgradnja modernih operacijskih sustava	150
7. Analiza vremenskih svojstava računalnog sustava	157
7.1. Uvodna razmatranja	157
7.1.1. Periodni poslovi	161
7.2. Povezanost Poissonove i eksponencijalne razdiobe	163
7.2.1. Poissonova razdioba	163
7.2.2. Eksponencijalna razdioba i njezina veza s Poissonovom	167
7.3. Analiza sustava s Poissonovom razdiobom dolazaka i eksponencijalnom razdiobom trajanja obrade	169
7.4. Osnovni načini dodjeljivanja procesora dretvama	176
7.4.1. Dodjeljivanje po redu prispjeća	176
7.4.2. Kružno dodjeljivanje procesora	178
8. Gospodarenje spremničkim prostorom	187
8.1. Uvodna razmatranja	187
8.2. Osnovna svojstva magnetskih diskova	189
8.2.1. Organizacija zapisivanja sadržaja na disku	190
8.2.2. Vremenska svojstva diskova	193

8.2.3. Disk kao dopunski spremnik radnom spremniku	196
8.2.4. Procesni informacijski blok	198
8.3. Pregled razvitka načina dodjeljivanja radnog spremnika	199
8.3.1. Statičko rasporedavanje radnog spremnika	199
8.3.2. Dinamičko rasporedavanje radnog spremnika	202
8.3.3. Preklopni način uporabe radnog spremnika	209
8.4. Dodjeljivanje spremnika straničenjem.....	210
8.4.1. Sklopovska podloga straničenju.....	210
8.4.2. Opisnik virtualnog adresnog prostora.....	215
8.4.3. Priručni međuspremnik za prevodenje adresa	217
8.4.4. Straničenje na zahtjev	219
8.4.5. Strategije zamjene stranica.....	222
8.4.6. Teorijske strategije zamjene stranica	224
8.4.7. Praktične aproksimacije strategija zamjene stranica	227
8.4.8. Raspodjela okvira u višeprogramskom radu	229
8.4.9. Podjela okvira procesima	230
8.4.10. Radni skup	232
8.5. Zaključne napomene o gospodarenju spremničkim prostorom	234
9. Datotečni podsustav	241
9.1. Uloga datoteka u računalnim sustavima	241
9.2. Struktura datoteka	242
9.3. Smještanje datoteka na disku	245
9.4. Načela ostvarenja datotečnih funkcija	249
9.5. Metode posluživanja zahtjeva za pristup datotekama	252
10. Komunikacija između procesa	257
10.1. Komunikacija između procesa unutar istog računalnog sustava	257
10.1.1. Dijeljeni spremnički prostor	258
10.1.2. Razmjena poruka između procesa	259
10.2. Komunikacija između procesa u raspodijeljenim sustavima	260
10.2.1. Osnove umrežavanja	260
10.2.2. Struktura Interneta	262
10.2.3. Komunikacija između procesa	264
10.3. Međusobno isključivanje u raspodijeljenim sustavima	270
10.3.1. Međusobno isključivanje – osnovni mehanizam ostvarenja funkcija operacijskog sustava.....	270

10.3.2. Vremensko uređenje događaja u raspodijeljenim sustavima	271
10.3.3. Protokoli medusobnog isključivanja u raspodijeljenim sustavima	273
10.3.4. Protokol Ricarta i Agrawala	275
11. Sigurnost računalnih sustava	281
11.1. Uvod	281
11.1.1. Uvodne napomene	281
11.1.2. Vrste napada na sigurnost	283
11.1.3. Sigurnosni zahtjevi	285
11.1.4. Utjecaj pojedinih komponenti računalnih sustava na sigurnost	286
11.2. Osnove kriptografije	288
11.3. Simetrični kriptosustavi	291
11.3.1. Data Encryption Standard (DES)	292
11.3.2. Ustrostručeni DES, 3DES	293
11.3.3. Izbijeljeni DES, DESX	293
11.3.4. Kriptosustav IDEA	294
11.3.5. Napredni kriptosustav AES	296
11.4. Načini kriptiranja	301
11.4.1. Elektronička bilježnica	301
11.4.2. Ulančavanje	302
11.4.3. CFB i OFB načini kriptiranja	303
11.4.4. Brojač	304
11.5. Asimetrični kriptosustavi, sustavi s javnim ključem	304
11.5.1. Neke činjenice i algoritmi iz teorije brojeva	304
11.5.2. Asimetrični kriptosustav RSA	309
11.5.3. Komuniciranje uporabom kriptosustava RSA	310
11.5.4. Dobrota RSA kriptosustava	312
11.6. Sažetak poruke, utvrđivanje bespriječornosti	317
11.6.1. Digitalna omotnica	317
11.6.2. Digitalni pečat	319
11.6.3. Funkcije za izračunavanje sažetka, funkcije sažimanja	320
11.6.4. Važna svojstva funkcija za izračunavanje sažetka poruke	323
11.7. Sigurnosni protokoli	324
11.7.1. Diffie-Hellmanov postupak za razmjenu tajnog ključa	324
11.7.2. Raspodjela ključeva u zatvorenom simetričnom kriptosustavu	327

11.7.3. Raspodjela ključeva u zatvorenom asimetričnom kriptosustavu	333
11.7.4. Autentifikacija u zatvorenim sustavima	336
11.8. Prijava za rad	343
11.8.1. Kriptiranje lozinki.....	343
11.8.2. Otežavanje pogodaanja lozinke	344
11.8.3. Zaštita pritupanja pojedinim sredstvima – autorizacija	344
11.9. Autentifikacijski protokol Kerberos	345
11.9.1. Struktura sustava u kojem djeluje Kerberos protokol	346
11.9.2. Kerberos protokol	347
11.10. Infrastruktura javnih ključeva	350
11.10.1. Digitalni certifikat	350
11.10.2. Provjera certifikata u otvorenoj mreži	353
11.10.3. Infrastruktura javnih ključeva zasnovana na X.509 modelu	356
11.10.4. Problem opozivanja certifikata	359
11.11. Sigurnosna zaštitna stijena.....	360
11.12. Zaključne napomene	362
12. Višediskovni zalihosni spremnici	365
12.1. Osnovna razmatranja.....	365
12.2. Modeliranje zalihosnih sustava	368
12.2.1. Pouzdanost i nepouzdanost sustava	368
12.2.2. Model ponašanja popravljive komponente s konstantnim brzinama kvarenja i popravljanja	375
12.2.3. Modeliranje višekomponentnih sustava	378
12.3. Načini zalihosne organizacije diskova	386
12.3.1. RAID 0 – nezalihosna organizacija	387
12.3.2. RAID 1 – zrcaljena organizacija	387
12.3.3. RAID 2 – organizacija zasnovana na Hammingovim kodovima	388
12.3.4. RAID 3 – paritetna organizacija sitne zrnatosti	389
12.3.5. RAID 4 – paritetna organizacija krupne zrnatosti	389
12.3.6. RAID 5 – paritetna organizacija krupne zrnatosti s raspodijeljenim paritetnim pojasevima	390
12.3.7. RAID 6 – organizacija sa zaštitom od dvostrukog kvara ($P + Q$ zalihost)	391
12.3.8. Višerazinski RAID sustavi	391
Literatura	395
Kazalo pojmova	397

1.

Uvod

1.1. Prvi susret s operacijskim sustavom

1.1.1. Zadaci operacijskog sustava

Računalni sustav sastavljen je od procesora, radnog spremnika, vanjskih spremnika i različitih ulazno-izlaznih naprava. Te komponente čine sklopolje računala. Samo sklopolje računala (mogli bismo reći: "golo računalo") nije nam od velike koristi ako uz njega ne postoji odgovarajuća programska oprema. Različiti primjenski programi, s pomoću kojih korisnici računala obavljaju sebi korisne zadatke, transformiraju računalni sustav u odgovarajući virtualni stroj. Kao potpora svim tim raznovrsnim programima služi skup osnovnih programa koji omogućuju provođenje radnih zahvata na računalu – izvođenje *operacija računala*. Taj se skup programa naziva *operacijski sustav*. Moglo bi se reći da operacijski sustav "nadoknađuje" sva ograničenja i nedostatke sklopolja i stvara privid stroja koji je mnogo prikladniji za korištenje. Korisnik ne mora biti niti svjestan detalja provođenja neke operacije koju je on jednostavno zatražio. Operacijski sustav *skriva* od korisnika mnoge njemu nevažne *detalje* izvođenja neke operacije. Jedan je od zadataka operacijskog sustava dakle, *olakšavanje uporabe računala*.

Drugi važan zadatak operacijskog sustava jest organizacija *djelotvornog iskorištavanja svih dijelova* računala. Naime, unutar računala može se istodobno odvijati više poslova. Primjerice, istodobno dok se procesor "brine" o izvođenju jednog niza instrukcija, pristupni sklop pisača može iz glavnog spremnika prenositi sadržaj na pisač. Operacijski sustav se mora pobrinuti da se procesor prebacuje s izvođenja jednog niza instrukcija na drugi i podržati *višeprogramski rad*. Operacijski sustav mora svakom pojedinom programu omogućiti pristup do potrebnih mu datoteka i svih ostalih sredstava. Sva nužna sredstva operacijski sustav mora dodjeljivati pojedinim programima i oduzimati ih od drugih tako da ona budu što je moguće bolje iskorištena. Osim toga, operacijski sustav mora omogućiti ostvarenje komunikacije između računala ako su ona spojena u mrežu.

Sve operacije koje operacijski sustav omogućuje moraju se na neki način pokrenuti. Neke zahtjeve za pokretanje operacija postavlja čovjek – korisnik računala – a neki zahtjevi dolaze neposredno iz programa. Korisnik može, primjerice, zahtijevati da se neki program pokrene ili da se neka datoteka premjesti s diskete na disk ili obrnuto. Primjeri za zahtjeve koji dolaze iz programa jesu zahtjevi za obavljanje ulaznih i izlaznih operacija. Način postavljanja zahtjeva operacijskom sustavu, kao i izgled povratnih poruka operacijskog sustava, mora biti dogovoren. Utvrđeni način takva komuniciranja zovemo sučeljem. Naziv sučelja koristi se, općenito, za čvrsto dogovoreni način uspostavljanja veze između nekih, inače razdvojenih, cjelina.

Operacije operacijskog sustava, dakle, može pokrenuti:

- čovjek preko korisničkog sučelja ili
- program preko sučelja primjenskog programa.

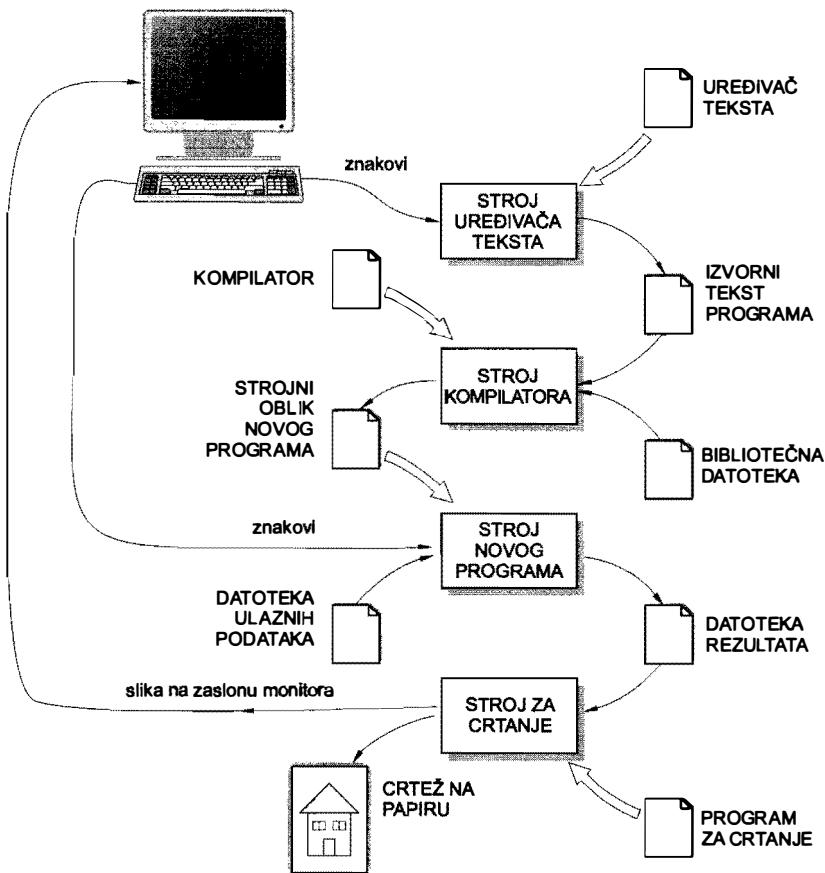
Operacijski sustav preko tih sučelja povratno vraća rezultate zatraženih operacija ili samo informacije o obavljenom zahtjevu. Pojedine operacije operacijskog sustava može pokrenuti čovjek kroz korisničko sučelje prema operacijskom sustavu. Najčešće operacije koje pokreće čovjek odnose se na datoteke: pretražuje se imenik datoteka kako bi se pronašlo željenu datoteku, datoteke se premještaju, kopiraju, preimenuju.

Svoje zahtjeve čovjek postavlja tako da ih upisuje na tipkovnici ili ih odabire na slikovnom sučelju. Već prvi pristup računalu zahtijeva upoznavanje nekoliko osnovnih naredbi operacijskog sustavu. U današnjim operacijskim sustavima prevladava uporaba slikovnog sučelja između čovjeka i stroja. Slikovna sučelja sa sličicama – ikonama – koje simboliziraju pojedine operacije ili objekte olakšavaju čovjeku uporabu računala. Pojedine komande nude se i u pisanom obliku i samo ih treba s pomoću miša odabrati iz ponuđenog izbornika – menija. Isto tako, povratne informacije operacijskog sustava pojavljuju se u vrlo razumljivom slikovnom obliku. Posljednjih se godina izgled slikovnog sučelja pomalo ujednačava i postoje već i neki dogовори koji vode prema standardizaciji slikovnog sučelja. Zaslon monitora tako postaje “prozor”¹ kroz koji se gleda u unutrašnjost računala.

1.1.2. Odvijanje tipičnog posla u računalnom sustavu

Napomenimo da se svi programi i svi podaci u računalu trajno čuvaju u obliku datoteka u vanjskim spremnicima. Stoga je pri upoznavanju računala i njegova operacijskog sustava najvažnije upoznati rad s datotekama. Sve što se u računalu događa svodi se na premještanje, mijenjanje i pohranjivanje datoteka. Svaki program koji se pokreće u računalu dolazi iz neke datoteke. Za obavljanje bilo kojeg posla računalom potrebna nam je skupina odgovarajućih programa koji su pohranjeni u obliku datoteka. Ti programi stvaraju radno okruženje za obavljanje tog posla.

¹ Engleski naziv za prozor – *window* – poslužio je kao naziv operacijskih sustava *Windows*, *Windows 95*, *Windows NT* tvrtke Microsoft, kao i za grafičko korisničko sučelja *X/Windows* za operacijske sustave *UNIX*.



Slika 1.1. Uloga datoteka u odvijanju programa

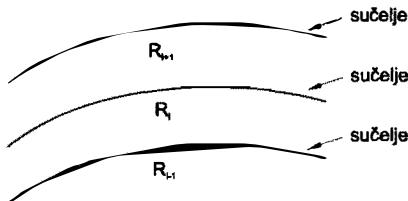
Primjer postupka pripreme jednog programa i njegova korištenja prikazan je na slici 1.1. Datoteke su prikazane kao krugovi, a aktivni programi kao pravokutnici. Neke datoteke koje sadržavaju gotove strojne programe postat će aktiviranjem programi. Na slici je ta pretvorba simbolizirana podebljanom strelicom koja vodi od kruga na pravokutnik. Programi kroz svoje sučelje prihvataju ili pojedinačne znakove s tipkovnice ili podatke pripremljene u ulaznim datotekama i proizvode izlazne datoteke. Posao se odvija u nekoliko koraka:

- Najprije se iz datoteke aktivira program za uređivanje teksta. On prihvata znakove s tipkovnice i pohranjuje ih postupno u datoteku izvornog programa.
- Nakon toga se iz svoje datoteke aktivira kompilator. On prihvata datoteku izvornog teksta kao ulaznu datoteku i uz dodatno korištenje bibliotečne datoteke izgrađuje strojni oblik programa, koji se opet pohranjuje u datoteku.
- Novostvoreni strojni program se aktivira i postaje program koji prihvata ulazne podatke s tipkovnica ili iz unaprijed pripremljene podatkovne datoteke.

- Taj program izračunava rezultate i pohranjuje ih u datoteku rezultata.
- Na kraju, može se aktivirati neki program za crtanje koji će kao ulaznu datoteku prihvati datoteku rezultata i proizvesti crtež na papiru i rastersku sliku na zaslonu. Iz ovog je prikaza vidljivo kako se računalo pretvara iz jednog virtualnog stroja u drugi jednostavnom zamjenom primjenskih programa. Operacijski sustav mora omogućiti da se te pretvorbe obavlaju što jednostavnije.

1.2. Hijerarhijska izgradnjva operacijskog sustava

Već se iz ovog kratkog opisa poželjnog ponašanja računalnog sustava naslućuje da su operacije koje mora omogućiti operacijski sustav razmjerno složene. Znamo da se te operacije moraju provoditi na računalnom sklopolvu koje je u stanju izvoditi samo vrlo jednostavne instrukcije. Stoga pri zasnivanju i ostvarenju operacijskih sustava (a i primjenskih programa) treba sviadati golemi jaz između mogućnosti sklopolva koje manipulira, takoreći, pojedinačnim bitovima pa do složenih operacija koje oživotvoruju naše apstraktne zamisli. Sviadavanje tog jaza pokušava se ostvariti svojevrsnom hijerarhijskom izgradnjom sustava.



Slika 1.2. Razine u hijerarhijskoj izgradnji sustava

Hijerarhijski pristup izgradnje složenih sustava koristi se i u mnogim drugim područjima. Principi stroge izgradnje hijerarhijskog sustava su sljedeći:

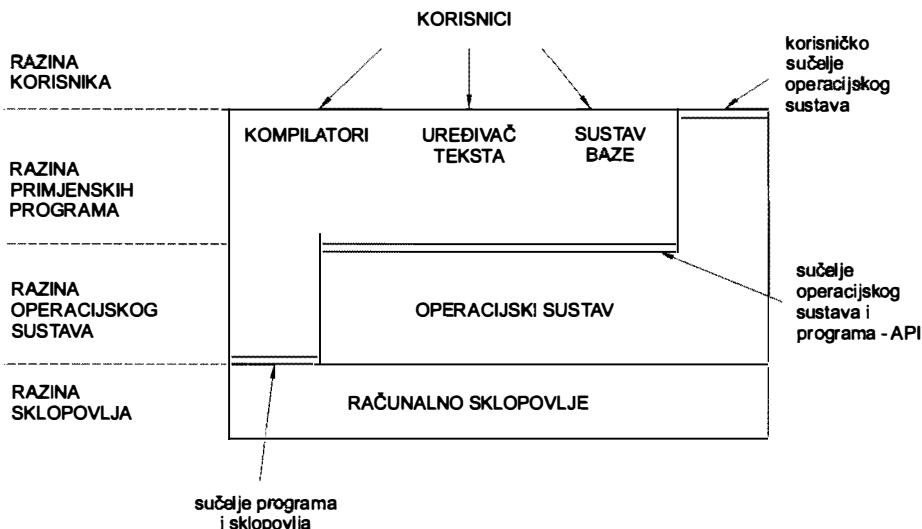
- sustav se izgrađuje po razinama;
- svaka razina sastoji se od objekata i operacija nad tim objektima;
- objekti i operacije neke razine izgrađuju se samo s pomoću objekata i operacija prve neposredne niže razine;
- detalji ostvarenja objekata i operacija pojedine razine skriveni su (a nisu nam ni važni)².

Slika 1.2. simbolizira hijerarhijsku izgradnju. Razina R_i ostvarena je samo s pomoću objekata i operacija razine R_{i-1} , a razina R_{i+1} samo od objekata i operacija iz razine

² Ovdje spominjani objekti i operacije nad njima ne trebaju se do kraja poistovjećivati s objektima u objektno zasnovanom programiranju, iako mnoge sličnosti nisu slučajne.

R_j . Razina R_{j+1} "ne vidi" objekte i operacije iz razine R_{j-1} . To znači da bismo razinu R_{j-1} mogli i zamijeniti a da to razina R_{j+1} i ne primjeti, tj. da ona kroz svoje sučelje "vidi" jednake objekte i operacije u razini R_j . Prema tome, u razini R_j trebalo bi s novom razinom R_{j-1} opet izgraditi jednake objekte i operacije.

U izgradnji operacijskih sustava ne slijede se do kraja principi stroge hijerarhijske izgradnje. Tako se, primjerice, u nekoj razini može koristiti ne samo operacije neposredne niže razine već i dalnjih nižih razina. Takav se pristup može približiti strogoj hijerarhijskoj izgradnji ako se smatra da se neki objekti i operacije prenose nepromijenjeni u više razine. Oni objekti i operacije koji nisu preneseni ostaju za više razine "skriveni", tj. ne mogu se u višim razinama koristiti.



Slika 1.3. Hijerarhijska struktura računalnog sustava

Načelo hijerarhijske izgrađnje može se uočiti već i na slici 1.3. koja predstavlja grubu slojevitu strukturu sustava. Na računalno sklopolje, koje čini osnovnu razinu računalnog sustava, oslonjena je razina operacijskog sustava računala. Iznad razine operacijskog sustava nalazi se razina primjenskih programi koji transformiraju računalni sustav u različite virtualne strojeve. Različiti korisnici sa svoje razine kroz sučelja primjenskih programa gledaju "svoj" virtualni stroj. Ti korisnici moraju poznavati samo neke najosnovnije funkcije operacijskog sustava, a mogu do njih pristupiti kroz sučelje operacijskog sustava predviđeno za korisnike. U današnjim se primjenskim programima nastoji što više međusobno ujednačiti sučelja kako bi "horizontalno" premještanje korisnika bilo što jednostavnije. Što više, dio sučelja operacijskog sustava koji je vidljiv s korisničke razine također je sličan sučeljima programa.

Primjenski su programi, s jedne strane, neposredno oslonjeni na sklopolje (izvođenje programa svodi se na izvođenje niza strojnih instrukcija) i, s druge strane, na funkcije pripremljene unutar operacijskog sustava (sustavske funkcije) koje su dohvatljive kroz

njegovo sučelje prema primjenskim programima – API. U neku bismo ruku mogli isto tako reći da skup strojnih instrukcija pojedinog procesora čini sučelje između programa i računalnog sklopolja. Ne zaboravimo da svi programi, bez obzira na to u kojem su programskom jeziku pripremljeni moraju biti prevedeni u strojni oblik prije nego li započne njihovo izvođenje. *Strojni oblik programa* sastoji se od niza *strojnih instrukcija* koje u sklopolju računala izazivaju neke osnovne operacije. Jasno je da su i funkcije operacijskog sustava također oslonjene na sklopolje i da su ostvarene nizom strojnih instrukcija.

1.3. Načini izučavanja operacijskih sustava

Pri razmatranju operacijskih sustava treba uzeti u obzir da je način i dubina njihova izučavanja određena ciljevima ljudi razvrstanih u pojedine interesne skupine:

- *Obični korisnici* računala, koji žele koristiti računalo samo kao pomagalo u svom svakidašnjem radu ne moraju mnogo znati ni o računalu, ni o njegovu operacijskom sustavu. Oni moraju moći samo pokrenuti svoje primjenske programe i svladati uporabu tih programa. Osnovna obuka o operacijskom sustavu za takve će korisnike biti vrlo kratka i jednostavna i sastojat će se od upoznavanja sučelja i nekoliko komandi za pokretanje osnovnih operacija operacijskog sustava.
- *Napredni korisnici* računala žele svoje primjenske programe izvoditi na što djelotvorniji način, zadovoljiti neka ograničenja nametnutna načinom njihove uporabe i s nekog određenog stanovišta optimirati njihovo izvođenje. Takvi korisnici moraju detaljnije poznavati svojstva računalne opreme na kojoj se izvode njihovi programi, pa prema tome i svojstva i mogućnosti operacijskog sustava. Upravo kroz operacijski sustav korisnici pristupaju do svih sredstava svog računalnog sustava. Oni moraju dovoljno dobro poznavati osnovne mehanizme kojima se ostvaruju pojedine funkcije operacijskog sustava kako bi s razumijevanjem mogli poduzimati odgovarajuće zahvate na svom sustavu.
- *Programeri primjenskih programa* koji će pripremati nove primjenske programe moraju dobro poznavati sve mogućnosti operacijskog sustava, i to posebice skup operacija koje nudi API, kako bi u svojim programima mogli djelotvorno iskoristiti računalni sustav. Operacijski ih sustav oslobađa mnogih mučnih detalja pristupa računalnim sredstvima – ti su detalji razriješeni unutar procedura i funkcija koje se pozivaju kroz API.
- *Specijalisti iz područja računarstva* koji će se baviti zasnivanjem, projektiranjem i održavanjem računalnih sustava (pa eventualno i pregradnjama i dogradnjama operacijskih sustava).

Gradivo koje je obrađeno u ovom udžbeniku može biti zanimljivo svim spomenutim interesnim skupinama. Naime, gradivo pojedinih poglavlja izloženo je tako da se kreće

od jednostavnih obrazloženja općih postavki prema sve složenijem i detaljnijem objašnjanju teme koja se razmatra. Za prvu interesnu skupinu običnih korisnika gradivo je u pravilu malo preopširno, dok će potencijalni specijalisti nakon izučavanja cijelog udžbenika morati potražiti još i dodatnu literaturu. Moglo bi se, dakle, smatrati da je gradivo najbolje prilagođeno dvjema srednjim interesnim skupinama.

Za razmatranje djelovanja računalnog sustava prikladno nam je, barem donekle, slijediti strukturu njegove izgradnje, kako bismo lakše svladali njegovu složenost. Pritom su, u načelu, moguća dva pristupa:

- sustav se može početi razmatrati s "vanske strane", tako da se podje od korisničkog sučelja i u objašnjavanju slijedi redom one funkcije nižih razina koje su nužne za razumijevanje nekog podskupa operacija koje omogućuju ostvarenje razmatrane funkcije;
- sustav se može početi razmatrati od najnižih razina i postupno opisivati sve više i više razine.

Prvi je pristup uobičajen pri brzom upotrebnom upoznavanju operacijskog sustava. Svaki korisnik koji želi koristiti računalo mora naučiti neke osnovne komande koje mu omogućuju pokretanje želenog primjenskog programa i pritom ne mora ni znati kako se pojedine operacije provode. Za takav pristup upoznavanju operacijskog sustava najbolje je koristiti priručnike koji ukratko opisuju osnovne komande operacijskog sustava i uzorke njihova korištenja. U današnjim se računalima iz grafičkog sučelja može pokrenuti uslužni program s uputama (engl. *Help*) koji na zaslonu monitora ispisuje osnovna svojstva pojedinih komandi i načine njihove uporabe. Međutim, u takvom se pristupu vrlo često mora prihvatići neke činjenice bez njihova potpunog razumijevanja, a ostaju nejasni i motivi koji su uzrokovali načine ostvarenja pojedinih funkcija.

Drugi pristup je prikladniji pri izučavanju računalnih sustava u okviru redovitog školovanja kada se bez posebne žurbe može pristupiti sustavnom izučavanju čitavog gradiva. U tom se pristupu mogu postupno objašnjavati pojedine razine sustava upravo onim redoslijedom kako se one i izgrađuju. Pritom se može naglašavati ona osnovna načela izgradnje koja imaju trajniju vrijednost i koja ne ovise o nekom konkretnom programskom rješenju.

Čitatelj koji na takav način pristupi usvajanju gradiva neće moći nakon prvih nekoliko lekcija koristiti računalo, ali će na kraju bolje razumjeti mnoge detalje ponašanja računalnih sustava i moći će se vrlo dobro snaći u raznovrsnim radnim okruženjima. On će naučiti osnovna načela izgradnje složenih programskih sustava i lakše će, i to s razumijevanjem, usvajati novine u izgradnji operacijskih sustava koji su još u razdoblju intenzivnog razvitka.

U ovom je udžbeniku stoga usvojen ovaj drugi pristup izučavanju operacijskih sustava. Polazi se od razine sklopovlja i postupno se izgrađuju sve više i više razine sustava. Pritom se oblikuju neki pojednostavljeni modeli sklopovskih komponenti koji olakšavaju razumijevanje osnovnih mehanizama, ali su dovoljno bliski današnjim ostvarenjima i prema tome omogućuju poimanje stvarnih mogućnosti sustava.



PITANJA ZA PROVJERU ZNANJA 1

- 1. Što je operacijski sustav?**
- 2. Koji su osnovni zadaci operacijskog sustava?**
- 3. Navesti osnovne dijelove operacijskog sustava.**
- 4. Što je to sučelje?**
- 5. Što je to sučelje primjenskih programa (API)?**

2.

Model jednostavnog računala

2.1. Von Neumannov model računala i načini njegova ostvarenja

2.1.1. Funkcijski model računala

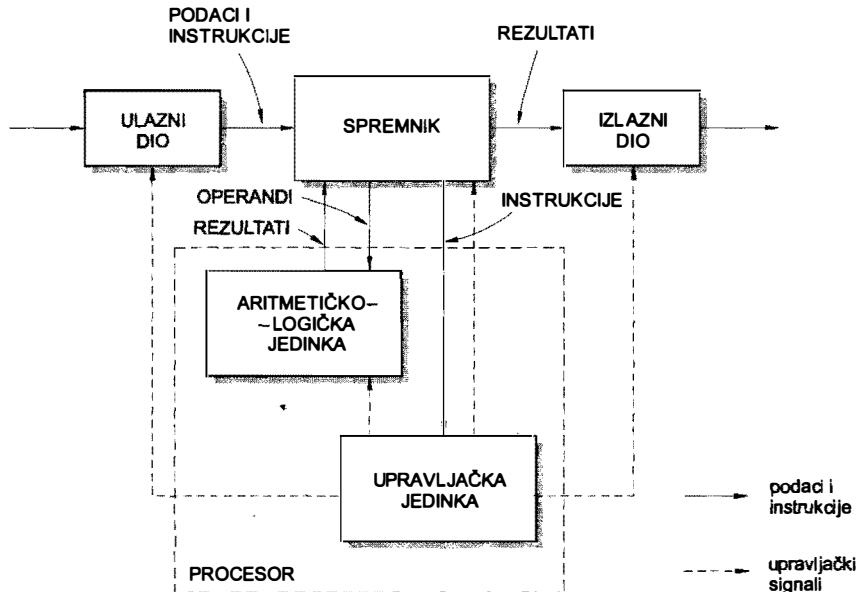
Današnji računalni sustavi zasnivaju se još uvijek pretežito na koncepcijskom modelu koji je još 1945. godine opisao John von Neumann.

Von Neumannov model utvrđuje da svako računalo mora imati sljedeće dijelove:

- *ulazni dio* preko kojeg se u spremnik unose iz okoline podaci i instrukcije programa;
- *izlazni dio* preko kojeg se u okolini prenose rezultati programa;
- *radni ili glavni spremnik* u koji se pohranjuju svi podaci i instrukcije programa une-seni izvana, kao i rezultati djelovanja instrukcija;
- *aritmetičko-logičku jedinku* koja može izvoditi instrukcijama zadane aritmetičke i logičke operacije;
- *upravljačku jedinku* koja dohvaća instrukcije iz spremnika, dekodira ih i na temelju toga upravlja aritmetičko-logičkom jedinkom, te ulaznim i izlaznim dijelovima.

Slika 2.1. ilustrira međusobnu povezanost svih tih dijelova. Na slici su označeni tokovi podataka, instrukcija i upravljačkih signala.

Središnji dio računala je *spremnik*. U njega se slijevaju svi podaci i instrukcije koje se unose u računalo preko *ulaznog dijela*, te svi rezultati operacija iz aritmetičko-logičke jedinke. Preko izlaznog dijela rezultati izračunavanja prenose se u okolinu. Iz spremnika *upravljačka jedinka* dohvaća instrukcije i na temelju njih upravlja preostalim dijelovima računala. Upravljačka jedinka određuje koju će operaciju izvesti *aritmetičko-logička jedinka*. Upravljačka jedinka i aritmetičko-logička jedinka spregnute su današnjim računalima u jednu cjelinu i, dodatno, s jednim skupom registara čine *procesor*.



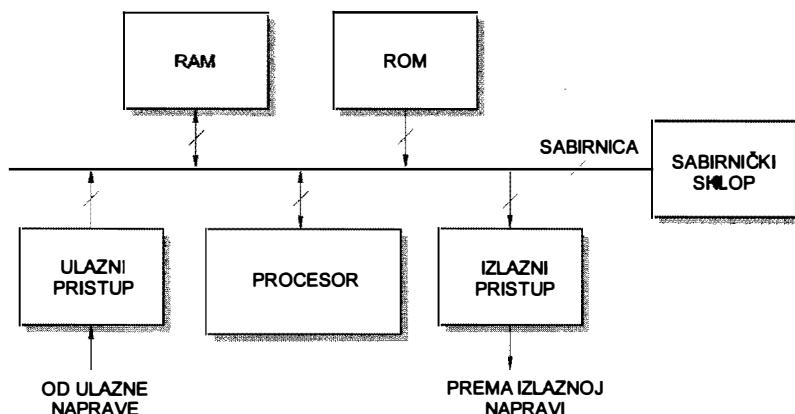
Slika 2.1. Funkcijski Von Neumannov model računala

2.1.2. Sabirnička građa računala

U funkcijском prikazu računala na slici 2.1. vidljivo je kako su pojedini dijelovi računala međusobno povezani. Svaka od crta koja predstavlja tok podataka, instrukcija ili upravljačkih signala sastoji se od većeg broja vodiča preko kojih se prenose električki signali kojima se prenose bitovi. Za prijenos svakog bita potreban nam je stoga jedan vodič. Svaka crta u funkcijском prikazu računala predstavlja prospojni put kojim se istovremeno prenosi potrebni broj bitova. Takvo isprepleteno međusobno povezivanje pojedinih dijelova računala nespretno je i stoga je osmišljen *sabirnički sustav* za njihovo povezivanje. Sabirnica¹ je jedan zajednički snop vodiča na koji su spojeni svi dijelovi računala. Osim vodiča sabirnica ima i svoj sabirnički elektronički sklop, koji pomaže pri ostvarivanju veza.

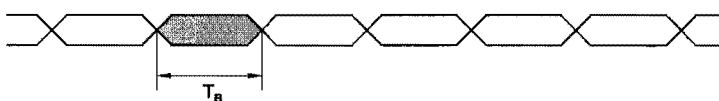
Računalo sa sabirničkim povezivanjem dijelova prikazano je na slici 2.2. Sve potrebne razmijene podataka, instrukcija i upravljačkih signala obavljaju se preko zajedničkih vodiča. Kosa crtica preko crte koja predstavlja sabirnicu označava da je riječ o snopu vodiča. Jasno je da se preko zajedničkih vodiča ne može uspostavljati više istovremenih veza već se mogu obavljati samo pojedinačni prijenosi. Sabirnica se stoga mora naizmjence – s podjelom vremena (engl. *time share*) – koristiti za ostvarenje potrebnih veza između dijelova računala.

¹ U engleskom jeziku je sabirnica dobila naziv *bus*, što je izvorni naziv za autobus koji sabire i razvozi putnike.



Slika 2.2. Sklopolje računala povezano sabirnicom

Vrijeme se na sabirnici dijeli na sabirničke periode ili *sabirničke cikluse*,² kao što je prikazano na slici 2.3. U jednom sabirničkom ciklusu trajanja T_B ostvaruje se jedna veza, tj. prijenos jednog sadržaja. Primjerice, sabirnica koja bi imala ciklus trajanja $T_B = 100$ ns omogućila bi da se u jednoj sekundi obavi 10 milijuna prijenosa sadržaja. Ako bi se u jednom ciklusu prenosi samo po jedan bajt, onda bi se preko sabirnice moglo prenijeti 10 MB u sekundi.



Slika 2.3. Podjela vremena na sabirničke cikluse

2.1.3. Radni ili središnji spremnik računala

Ponovimo još jedanput da je središnji dio svakog računala spremnik. S obzirom na to da u računalnim sustavima postoje raznovrsni spremnici, za spremnik Von Neumannova modela koristi se naziv radni spremnik ili središnji spremnik. *Radni spremnik* dobio je svoj naziv zbog toga što pri izvođenju programa u njemu moraju biti smještene i instrukcije programa i podaci na koje te instrukcije djeluju. Radni spremnik je, isto tako, prolazno odredište i ishodište svih informacija koje kolaju između svih ostalih dijelova računala.

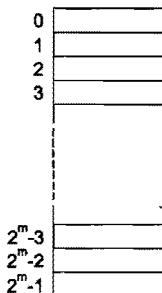
U današnjim su računalima spremnici organizirani tako da se sastoje od bajtova koji se sastoje od osam bitova. Za označavanje bajta služi kratica B. Za bajt se može koristiti i naziv *oktet*³ ili *osmorka* bitova, ali se time uglavnom podrazumijeva bilo kakva nakupina

² Ciklus dolazi od grčkog *kyklos* – krug, kolo, okruglo – a znači neku pojavu koja se redovito ponavlja. Slično značenje ima i naziv perioda, koji ćemo koristiti u opisu rada procesora, ali je prikladnije za sabirničku periodu koristiti naziv ciklus da bi se istaknula razlika prema periodi takt-a procesora.

³ U francuskom se za bajt upotrebljava naziv *octet*.

od osam bitova. Isto tako, za nakupinu od četiri bita koristi se naziv *četvorka* bitova ili *kuartet*⁴. Tako se može reći da jedan bajt sadržava jedan oktet ili dva kvarteta bitova, odnosno jednu osmorku ili dvije četvorke bitova.

Do svakog bajta u spremniku može se neposredno pristupiti. Svaki bajt dobiva svoj redni broj. Taj je broj čvrsto povezan s tim bajtom i s pomoću njega se bajt odabire. Stoga se taj broj naziva *adresom* bajta.



Slika 2.4. Adrese bajtova

Adrese bajtova unutar spremnika izražavaju se brojevima zapisanima u binarnom obliku. Ako se za zapisivanje adrese koristi m bitova, tada se može zapisati 2^m različitih adresa, a adrese se kreću u granicama od 0 do 2^{m-1} . S brojem m određena je veličina tzv. *adresnog prostora*.

PRIMJER 2.1.



Podsjetimo se da se veličina spremnika izražava kao cijelobrojni višekratnik od 2^{10} ili 2^{20} . Naime, pokazuje se da je:

$$2^{10} = 1024 \cong 1000 = 10^3,$$

odnosno:

$$2^{10} \cong 10^3.$$

Kako se za $10^3 = 1000$ koristi kratica k (pisana malim slovom i čita se "kilo"), to je za $2^{10} = 1024$ upotrijebljena kratica K (pisana velikim slovom i čita se "ka"). Tako je, primjerice, uz $m = 16$ moguće adresirati spremnik veličine:

$$2^{16} B = 2^6 \times 2^{10} B = 2^6 \times 1 KB = 64 KB.$$

Sljedeća veća jedinica za veličinu spremnika je MB (čita se: "megabajt"). Pritom je:

$$1 MB = 2^{20} B = 2^{10} \times 2^{10} B = 1024 \times 1 KB = 1024 \times 1024 B = 1048576 B,$$

što je približno jednako jedan milijun, odnosno $1\,000\,000$ ili 10^6 , pa se stoga i koristi prefiks *mega*.

⁴ U engleskom se za kvartet koristi naziv *nibble* – što znači mali zalogaj.

Isto tako definira se i jedinica GB (čita se: "gigabajt") koja iznosi:

$$\begin{aligned}1 \text{ GB} &= 2^{30} \text{ B} = 2^{10} \times 2^{20} \text{ B} = 1024 \times 1 \text{ MB} = 1024 \times 1024 \times 1 \text{ KB} \\&= 1024 \times 1024 \times 1024 \text{ B} = 1\,048\,576 \text{ B} = 1\,073\,741\,824 \text{ B},\end{aligned}$$

što je približno jednako jednoj milijardi, odnosno $1\,000\,000\,000$ ili 10^9 , pa se stoga i koristi prefiks *giga*.

Tako, primjerice, adrese s $m = 32$ bita mogu adresirati adresni prostor veličine:

$$2^{32} \text{ B} = 2^2 \times 2^{30} \text{ B} = 4 \times 1 \text{ GB} = 4 \text{ GB}.$$

2.1.4. Rudimentarno računalo, radni spremnik

Za opis nekih osnovnih svojstava računala možemo razmatrati model računala koji se sastoji samo od *procesora* i *radnog spremnika*. Takvo nam rudimentarno računalo može poslužiti za modeliranje ponašanja računalnog sustava, pri čemu prepostavljamo:

- da je u radnom spremniku smješten strojni oblik programa kao niz instrukcija pohranjenih u uzastopne spremničke lokacije;
- da se u radnom spremniku nalaze i svi ulazni podaci koje će program dohvaćati tijekom izvođenja;
- da će program rezultate koje bude proizvodio tijekom izvođenja pohranjivati u za to predviđene lokacije radnog spremnika.

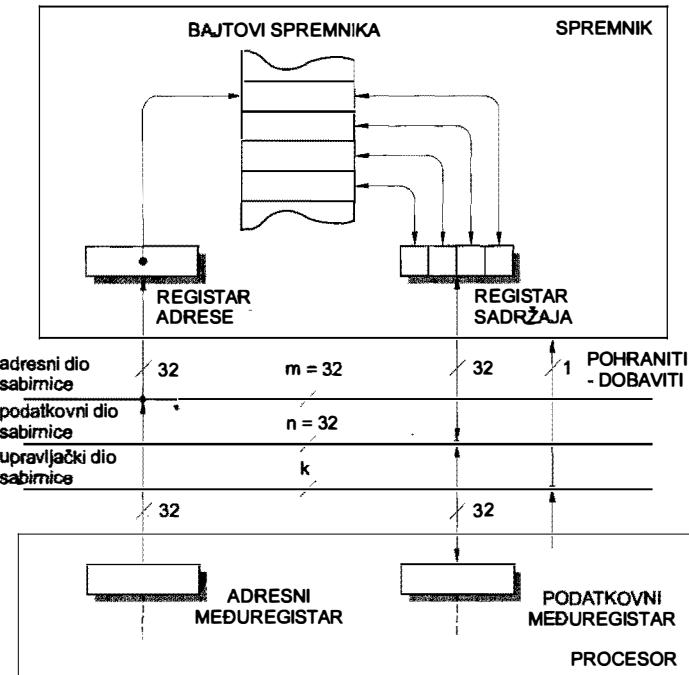
Ovakvo računalo ne bi imalo nikakvu praktičnu vrijednost, jer ne bi imalo veze s okolnim svjetom, ali dobro može modelirati računalni sustav u razdoblju između unošenja ulaznih podataka i prenošenja rezultata izračunavanja u vanjski svijet.

Pogledajmo malo podrobnije kako bi takvo računalo moglo djelovati. Spremnik i procesor u ovakovom rudimentarnom računalu mogli biti neposredno povezani, ali je zbog kasnijeg proširenja ovog modela prikladno razmotriti povezivanje preko sabirnice, kao što prikazuje slika 2.5. Sabirница je ovdje prikazana detaljnije i vidi se da ona ima tri dijela:

- adresni dio sabirnice s m vodiča;
- podatkovni dio s n vodiča;
- upravljački dio s k vodiča.

Na slici je pretpostavljeno da je $m = 32$ i $n = 32$, tj. da koristimo tridesetdvobitovnu spremničku adresu, te da se preko sabirnice istovremeno može prenijeti četiri bajta. Govorimo da je *širina* pristupa do spremnika četiri bajta ili 32 bita.⁵

⁵ Danas se u osobnim računalima najčešće koristi tridesetdvobitovna arhitektura s $m = 32$ i $n = 32$, a u snažnijim računalnim sustavima šezdesetčetverobitovna arhitektura s $m = 64$ i $n = 64$. U šezdesetčetverobitnoj arhitekturi može se adresirati adresni prostor veličine $2^{64} \text{ B} = 2^4 \times 2^{60} \text{ B} \approx 16 \times 10^{18} \text{ B} = 16\text{EB}$ (čita se: "eksabajt"), a širina pristupa omogućuje istovremenih prijenos osam bajtova.



Slika 2.5. Procesor i spremnik povezani na sabirnicu

Procesor je, također, povezan na sabirnicu preko registara. Nazvat ćemo ih međuregistra-
ma prema sabirnici ili, kraće, samo *međuregistrima*. Iz *adresnog međuregistra* postavlja se
adresa na adresni dio sabirnice, a *podatkovni međuregistar* ima "dvosmjerno" djelovanje:
podaci se mogu prenositi iz procesora u spremniku ili iz spremnika u procesor. Adresa
koju procesor postavlja na sabirnicu adresira prvi od četiri bajta koji će biti dobavljeni iz
spremnika (u 64-bitovnoj arhitekturi to je adresa prvog od osam bitova) ili pohranjeni u
spremnik. Pokazalo se praktičnim da se sadržaji koji imaju više od osam bitova (višekratnik
od osam) pohranjuju s *poravnatim adresama* (adrese su cjelobrojni višekratnici
od broja bajtova potrebnih za pohranjivanje sadržaja).

U jednom spremničkom ciklusu može se obaviti jedno pohranjivanje (“pisanje”) u spremnik ili jedno dobavljanje sadržaja (“čitanje”) iz spremnika⁶. U *ciklusu pohranjivanja* događa se sljedeće:

- *procesor* na adresni dio sabirnice postavlja adresu iz svog adresnog međuregistra, signal pohranjivanja na priklučak POHRANITI-DOBAVITI, preko upravljačkog dijela sabirnice, te postavlja podatak iz svog podatkovnog registra na podatkovni dio sabirnice;
 - *spremnik* prihvata adresu u svoj registar adrese, prihvata podatak u registar sadržaja i pohranjuje sadržaj u bajtove spremnika.

⁶ U engleskom se jeziku za operacije premještanja sadržaja koriste nazivi *store* – pohraniti ili *write* – pisati, te *load* – nakrcati (misli se na podatkovni međuregistar, odnosno neki drugi od registara procesora) ili *read* – čitati. U uporabi je i naziv *move* – premještit, uz koji se naznačuje ishodište i odredište premeštanog sadržaja.

U *ciklusu dobavljanja* podatak se iz spremnika u procesor prenosi na sljedeći način:

- *procesor* na adresni dio sabirnice postavlja adresu iz svog adresnog međuregistra i signal dobavljanja na priključak POHRANITI-DOBAVITI, preko upravljačkog dijela sabirnice;
- *spremnik* prihvata sa sabirnice adresu u svoj registar adrese, prebacuje iz bajtova spremnika sadržaje u registar sadržaja, te postavlja sadržaj na podatkovni dio sabirnice;
- *procesor* prihvata u svoj podatkovni registar sadržaj s podatkovnog dijela sabirnice.

Već smo spomenuli da spremnički ciklus ima neko konačno trajanje T_B . To se vrijeme ne može skratiti ispod neke donje granice zbog fizikalnih ograničenja sklopolvlja i vodiča. Kao što smo već spomenuli, uz $T_B = 100$ ns moguće je obaviti 10 milijuna prijenosa. No, ako istovremeno prenosimo četiri bajta, onda se može prenijeti 40 MB u sekundi, a uz širinu prijenosa od osam bajtova brzina prijenosa iznosila bi 80 MB u sekundi.

Ne zaboravimo da su svi dijelovi računala u jednosabirničkom sustavu povezani preko jedne sabirnice. Sve razmjene sadržaja obavljaju se preko te sabirnice, pa se sabirnički ciklusi moraju dijeliti svim sudionicima u "prometu" preko sabirnice, te ona može postati prometno "usko grlo" i ograničiti brzinu rada računala. Vidjet ćemo kako se taj problem donekle ublažava uporabom priručnog spremnika nakon što razmotrimo funkcionalni opis procesora.

2.1.5. Rudimentarno računalo – procesor

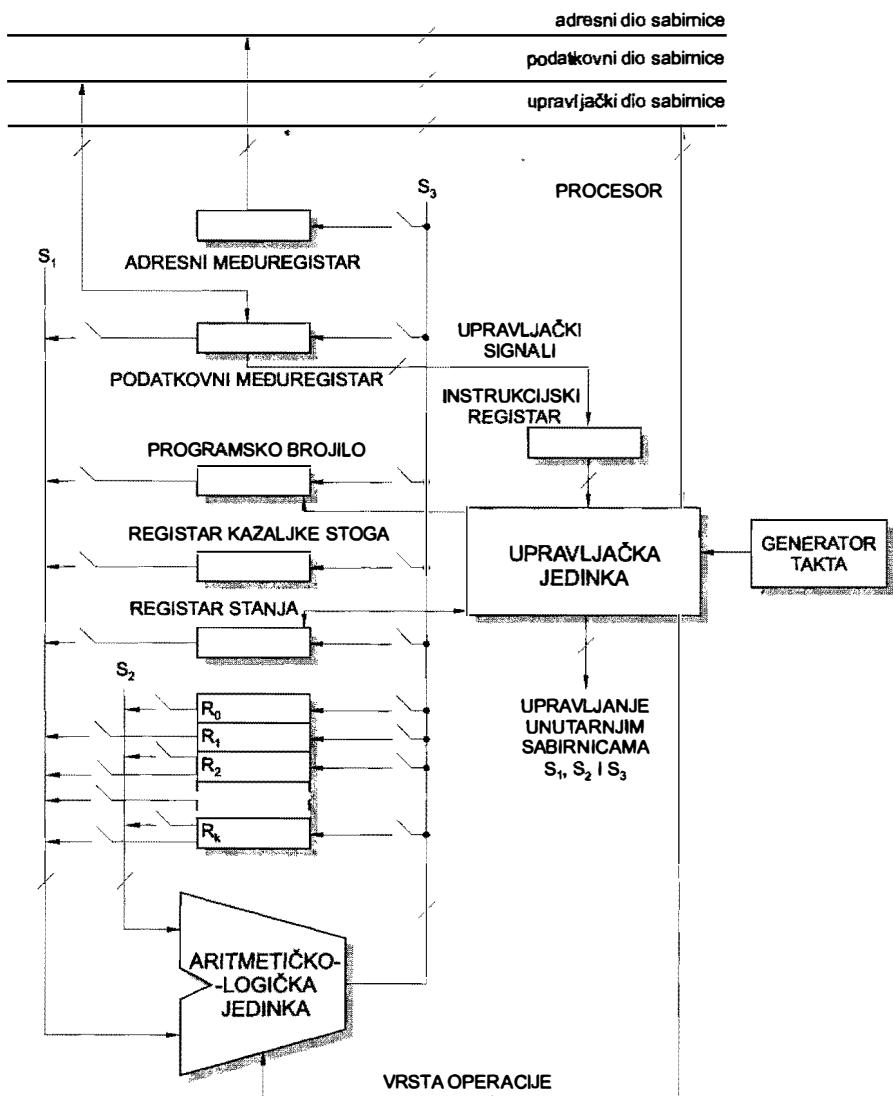
Treba posebno naglasiti da radni spremnik računala (što vrijedi i za ostale spremnike računalnog sustava) služi samo za pohranu nizova bitova. Interpretacija značenja tih nizova bitova događa se izvan spremnika. U rudimentarnom računalu sadržaji spremničkih lokacija zbivat će se u procesoru (interpretacija sadržaja moguća je i u drugim dijelovima računala – primjerice, na zaslonu monitora nizovi bitova pretvaraju se u izgled pojedinog slova). Unutar procesora se sadržaji koji se dobavljaju iz procesora interpretiraju dvojako:

- kao instrukcije strojnog programa računala;
- kao osnovni tipovi podataka.

Procesor adresira instrukcije i podatke na bitno različite načine. Instrukcije se dohvaćaju s adrese na koju pokazuje posebni registar procesora koji nazivamo programskim brojilom. Podaci se adresiraju tijekom izvođenja instrukcije, i to na temelju informacije o smještaju podataka sadržanih u instrukciji. Dio instrukcije koji definira operaciju koja će se izvoditi određuje i način interpretacije niza bitova koji će biti dobavljeni iz spremnika. Prema tome, procesorom je određen tip podataka. Za svaki tip podataka propisani su načini zapisivanja. Na toj strojnoj razini računala dogovoreni su oblici *osnovnih tipova podataka*, možemo reći i *osnovni objekti*. Za njih su na strojnoj razini ostvarene operacije u elektroničkom sklopolvu računala, a u strojnem jeziku postoje i instrukcije koje izazivaju obavljanje tih osnovnih operacija. Svi ostali složeniji tipovi podataka, potrebni za opise

složenijih objekata, moraju se oblikovati od tih osnovnih tipova, a operacije za te objekte izgrađuju se programskim funkcijama.

U Von Neumannovu modelu računala procesor čine aritmetičko-logička jedinka i upravljačka jedinka, te skup registara. Mi ćemo ovdje razmotriti pojednostavljeno djelovanje procesora, i to na vrlo pojednostavljenom modelu koji prikazuje slika 2.6. Stvarni su procesori mnogo složeniji od ovog modela, a i međusobno se prilično razlikuju, ali se njihova djelovanja mogu shvatiti na osnovi razmatranja ovog jednostavnog modela.



Slika 2.6. Pojednostavljeni model procesora

Prepostavit ćemo da su brojevi bitova adrese i podataka jednaki kao što smo prepostavili i na slici 2.5. U tom slučaju su svi registri jednakog duljina i svi prospojni putovi imaju jednak broj vodiča, pa to pojednostavljuje razmatranje.

Osnovna svojstva i ponašanje procesora određeni su *skupom registara* i *skupom instrukcija* koje procesor može obaviti. Registri služe za pohranjivanje svih informacijskih sadržaja koji ulaze i izlaze iz procesora i u njemu se transformiraju. Skup instrukcija određen je izvedbom aritmetičko-logičke i upravljačke jedinke procesora.

Opišimo najprije pojedine registre i njihovu ulogu u radu procesora:

- *Adresni međuregistar* već je opisan u prethodnom odjeljku. On služi za adresiranje spremnika ali i za adresiranje ostalih dijelova računala.
- *Podatkovni međuregistar* je posrednik za razmjenu sadržaja između procesora i ostalih dijelova računala. Svi podaci koji se žele prenijeti iz procesora na sabirnicu prolaze kroz podatkovni međuregistar.
- U *instrucijski registar* prenosi se instrukcija dobavljena iz spremnika. Bitovi instrukcije dovode se na upravljačku jedinku koja iz njih ustanavljuje koju operaciju procesor treba obaviti.
- *Programsko brojilo* (engl. *Program Counter* (PC)) je registar koji sadržava adresu instrukcije koju sljedeću treba obaviti. Njega upravljačka jedinka automatski povećava tako da se instrukcije pohranjene u spremniku izvode jedna iza druge.
- *Registar kazaljke stoga* (engl. *Stack Pointer* (SP)) služi za poseban način adresiranja spremnika. U spremniku se rezervira posebna skupina uzastopnih bajtova i zapiše u registar kazaljke stoga najvišu adresu te skupine. S pomoću tog regista ostvaruje se pohranjivanje podataka na stog.
- Bitovi *registra stanja* (engl. *Status Register*. ili *Condition Code Register* – registar uvjeta, *Flag Register* – registar zastavica) služe za zapisivanje različitih zastavica koje označavaju ispravnost ili neispravnost rezultata i operacija ili neke druge pojave. Vrijednosti tih zastavica jesu uvjeti na temelju kojih upravljački sklop određuje daljnji tijek odvijanja programa.
- Skupina *općih registrova* (imenovanih s R_0 do R_K) služi za pohranjivanje operanada i rezultata, te za pripremanje adresa budućih pristupa do spremnika.

Registri su povezani međusobno i s ostalim dijelovima procesora, posebice s aritmetičko-logičkom jedinkom. U ovom pojednostavljenom modelu procesora predviđa se povezivanje s pomoću *unutarnjih sabirnica* S_1 , S_2 i S_3 . Preklopke na crtama koje povezuju registre na sabirnicu simboliziraju povezanost registra na sabirnicu. Zamišljene preklopke otvara i zatvara upravljački sklop prema potrebi.

Upravljačka jedinka (engl. *Control Unit*) upravlja radom cijelog procesora. Ona dekodira instrukciju dovedenu u instrucijski registar i na temelju toga povezuje potrebne registre na unutarnje sabirnice, određuje aritmetičko-logičkoj jedinki vrstu operacije koju treba obaviti i automatski prelazi s instrukcije na instrukciju.

Upravljačka jedinka, a time i svi ostali dijelovi računala rade u ritmu koji određuje generator takta. *Generator taka* generira impulse koji pobuđuju električne sklopove. U današnjim procesorima frekvencija generatora ritma kreće se u granicama od nekoliko stotina MHz pa do nekoliko GHz. (Podsjetimo se da, primjerice, frekvencija generatora takta od 1 GHz određuje da je perioda pobudnih impulsa $T_S = 1 \text{ ns}$.) Sve pojave unutar procesora sinkronizirane⁷ su s generatorom takta. Generator takta, dakle, određuje brzinu rada procesora.

Konačno, *aritmetičko-logička jedinka* (engl. *Arithmetic-Logic Unit*) obavlja, kao što joj to i ime kaže, aritmetičke i logičke operacije s operandima koji se u našem modelu procesora dovode na njezin ulaz preko sabirnica S_1 i S_2 i rezultat postavlja na sabirnicu S_3 . Upravljačka jedinka koja je dekodirala instrukciju preko upravljačkih vodiča određuje aritmetičko-logičkoj jedinici operaciju koju ona mora obaviti. Isto tako, upravljačka jedinka iz instrukcije ustanavljuje odakle dolaze operandi i gdje treba pohraniti rezultate, pa u skladu s tim povezuje na unutarnje sabirnice pojedine od registara, tj. zatvara odgovarajuće preklopke prema sabirnicama S_1 , S_2 i S_3 . U našem pojednostavljenom modelu jedinka može i "propustiti" bez promjene sadržaj iz jednog od registara povezanog na sabirnicu S_1 do sabirnice S_3 kako bi sadržaji mogli prenositi iz jednog u drugi registar.

Nakon što smo ukratko upoznali sastavne dijelove procesora ustanovimo kako on radi. Prepostavljamo da su u spremniku pohranjene instrukcije programa, i to onim redoslijedom kako ih treba izvoditi. Taj niz instrukcija zovemo *strojnim programom*. U programsko brojilo (PC) mora se postaviti adresa prve instrukcije strojnog programa.

Procesor se može promatrati kao automat koji nakon uključivanja trajno izvodi instrukciju za instrukcijom strojnog programa. Ponašanje sklopova procesora pri izvođenju svake instrukcije može se opisati na sljedeći način⁸:



```

ponavljati {
    dohvati iz spremnika instrukciju na koju pokazuje programsko brojilo;
    dekodirati instrukciju, odrediti operaciju koju treba izvesti;
    povecati sadrzaj programskog brojila tako da pokazuje na sljedeću instrukciju;
    odrediti odakle dolaze operandi i gdje se pohranjuje rezultat;
    operande dovesti na aritmetičko-logičku jedinku, izvesti zadatu operaciju;
    pohraniti rezultat u odredište;
}
dok je (procesor uključen);

```

Procesor je, dakle, automatski izvoditelj programa koji obavlja instrukcije onim redom kojim su one smještene u spremniku. Ako taj redoslijed izvođenja treba narušiti, tj. iz-

⁷ Sinkronizirati dolazi od grčkog *syn* – s i *chronos* – vrijeme i znači: vremenski uskladiti.

⁸ Ovaj opis slijedi pravila pisanja uvedenih u programskom jeziku C. Svaka "instrukcija", zaključena sa znakom točka-zarez označava neku osnovnu aktivnost mikroelektričnog sklopova. Cijeli prolaz kroz petlju obavlja se tijekom izvođenja jedne instrukcije. Na jednak način opisivat ćemo i mnogo složenije aktivnosti operacijskog sustava gdje će nam pojedine "instrukcije" obilježavati izvođenje cijelih programskih odsječaka koji se mogu sastojati od nekoliko stotina ili nekoliko tisuća instrukcija.

vesti neke "skokove" pri izvođenju programa, onda se unutar instrukcije mora prisilno promijeniti sadržaj programskog brojila.

Iz gornjeg je opisa vidljivo da se izvođenje instrukcije može podijeliti u tri faze:

- U prvoj fazi *dohvata instrukcije* (engl. *fetch*) događa se sljedeće:
 - sadržaj programskog brojila prebacuje se u adresni međuregistar;
 - upravljačka jedinka pokreće aktivnost dohvata instrukcije iz spremnika;
 - instrukcija dolazi u podatkovni međuregistar i iz njega u instrukcijski registar.
- U drugoj fazi *dekodiranja instrukcije* (engl. *decode*) upravljačka jedinka:
 - utvrđuje na temelju dijela bitova instrukcije (koje nazivamo *operacijskim kodom*) operaciju koju treba provesti;
 - povećava sadržaj programskog brojila tako da pokazuje na sljedeću instrukciju;
 - šalje upravljačke signale aritmetičko-logičkoj jedinku kako bi ona "znala" koju operaciju treba obaviti;
 - na temelju dijela bitova instrukcije (koje nazivamo *adresnim dijelom instrukcije*) ustanavljuje iz kojih registara dolaze operandi i gdje treba pohraniti rezultat, te "zatvara" odgovarajuće preklopke na unutarnjim sabirnicama;
 - ako adresni dio instrukcije određuje da operand dolazi iz spremnika, onda se adresa operanda prebacuje u adresni međuregistar i pokreće dobavljanje operanda iz spremnika, pa će se taj operand naći u podatkovnom međuspremniku i iz njega dovesti u aritmetičko-logičku jedinku.
- U trećoj se fazi *obavljanja operacije* (engl. *execute*) događa sljedeće:
 - aritmetičko-logička jedinka obavlja zadanu operaciju i pohranjuje rezultat preko sabirnice S_3 u odredište;
 - vrijednosti pojedinih zastavica koje ovise o dobivenom rezultatu pohranjuju se u registar stanja.

2.1.6. Brzina rada procesora, priručni spremnik

Gornji opis ponašanja procesora ukazuje da izvođenje jedne instrukcije traje određeno vrijeme. S obzirom na to da je rad procesora sinkroniziran s generatorom takta, trajanje instrukcije je cijelobrojni višekratnik periode takta. Prema tome, isti procesor može raditi različitim brzinama ako se promijeni frekvencija generatora takta. Za svaki je procesor određena gornja granica te frekvencije, no odabrana frekvencija rada ovisi i o okolnom skloplju računala (osobito o mogućoj brzini prijenosa podataka preko sabirnice).

Često se brzinsko svojstvo procesora iskazuje brojem instrukcija koje može izvesti u sekundi. To ovisi o tome koliko perioda takta procesor troši za izvođenje instrukcija. S obzirom na to da različite instrukcije mogu "potrošiti" različit broj perioda takta uzima se neki prosječni broj perioda odgovarajuće mješavine različitih instrukcija. Primjerice,

ako je perioda takta $T_S = 10$ ns i za izvođenje instrukcija procesor troši 5 perioda, prosječno trajanje jedne instrukcije je 50 ns, tj. u jednoj sekundi može se izvesti 20 milijuna instrukcija. Kaže se da je računalna moć procesora 20 MIPS (milijuna instrukcija po sekundi)⁹.

Međutim, brzina rada računala ne ovisi samo o brzini procesora, već i o ostalom računalnom sklopolju. S obzirom na to da se instrukcije i podaci dobavljaju iz spremnika, instrukcije se ne mogu dobavljati brže nego što to dopušta sabirnica. U jednom sabirničkom ciklusu može se dobaviti samo onoliko bajtova koliko to dopušta širina pristupa. Tako bi sabirnica sa sabirničkim ciklusom od $T_B = 100$ ns dopuštala izvođenje 10 milijuna instrukcija u sekundi, iako procesor ima moć od 20 MIPS. Štoviše, ustanovili smo da neke instrukcije mogu izazvati i višekratni pristup do spremnika. Time se brzina rada i dalje smanjuje. Sabirnica sa svojom ograničenom brzinom prijenosa bitno ograničava mogućnosti procesora.

Graditelji računala nastoje na razne načine ubrzati rad računala. Ovdje nije mjesto razmatranju takvih poboljšanja. Zainteresirani čitatelj upućuje se na literaturu u kojoj su takva arhitektonska poboljšanja temeljito opisana¹⁰. Međutim jedno od poboljšanja toliko je rasprostranjeno da ga se i u ovom pojednostavljenom prikazu ne može preskočiti. Naime, neposredno uz procesor dodaje se jedan manji spremnik (on čak može biti izведен na istom procesorskom čipu), tzv. priručni spremnik (engl. *cache* – spremište, tajno skrovište), koji ima dopuštenu brzinu pristupa sukladnu brzini rada procesora. Instrukcije i podaci koji se nalaze u tom spremniku dostupni su, dakle, u istom vremenu kao i registri procesora (u nekim arhitekturnim rješenjima predviđeni su razdvojeni priručni spremnici za instrukcije i podatke). Tada se procesor u radu može približiti svojim maksimalnim brzinskim mogućnostima. Priručni spremnik može imati samo ograničenu veličinu i posebnim se mehanizmima treba osigurati da se kopija sadržaja onog dijela glavnog spremnika koji se upravo koristi u njega premjesti. Tijekom izvođenja programa razni dijelovi glavnog spremnika premještaju se u priručni spremnik i obrnuto, iz priručnog spremnika u glavni. Sve se to događa "skriveno" (otuda i engleski naziv za priručni spremnik). Strojni program i način adresiranja ne moraju se mijenjati u odnosu na izvođenje bez priručnog spremnika. Jedina vidljiva razlika je u trajanju izvođenja programa. S obzirom na to da je glavnina mehanizama za djelovanje sustava riješena sklopovski, u ovom se udžbeniku time nećemo posebno baviti.

2.1.7. Instrukcijski skup procesora

Izvedba aritmetičko-logičke jedinke i upravljačke jedinke određuje *instrukcijski skup* (engl. *Instruction Set*) nekog procesora. Već smo rekli da upravljačka jedinka mora dekodirati instrukcije i na temelju njihova sadržaja narediti aritmetičko-logičkoj jedinkoj operaciju treba obaviti te povezati potrebne registre na unutarnje sabirnice i postavljati upravljačke signale ostalim dijelovima računala. Aritmetičko-logička jedinka mora moći provesti zadalu operaciju.

⁹ Izvorno je MIPS kratica od engleskog *Milion Instructions per Second*.

¹⁰ Vidi: S. Ribarić, Napredne arhitekture RISC i CISC procesora, Školska knjiga, Zagreb, 1995.

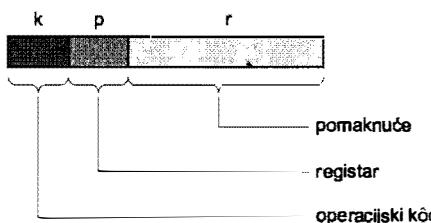
Iz svega što je dosada rečeno o procesoru možemo zaključiti da je on vrlo složena mikroelektronička tvorevina. Procesorski čip sadržava raznolike sklopove: registre, upravljačku jedinku, aritmetičku jedinku, priručni spremnik. Ti su sklopovi, osim toga, međusobno povezani složenim vezama. Procesorski je čip stoga mnogo složeniji od spremničkog čipa, pa ograničenja u njegovoj izgradnji, nametnuta mikroelektroničkom tehnologijom, dolaze više do izražaja. Uz danu površinu pločice silicija i tehnološke mogućnosti može se na pločicu staviti ograničeni broj tranzistora, koji su osnovni građevni elementi svih sklopova procesora. Pokazalo se s vremenom da je djelotvorniji procesor koji ima *više registara i skromniji instrukcijski skup*, od onog koji ima bogatiji instrukcijski skup (što znači veću aritmetičko-logičku i upravljačku jedinku) ali zato manje registara. To je dovelo do toga da se danas proizvode tzv. RISC¹¹ procesori koji imaju reducirani instrukcijski skup, ali zato više registara, pa i priručnog spremničkog prostora. Ostale procesore zovu CISC¹² procesorima (oni su to ime dobili tek kada je smišljen naziv RISC).

Mi ćemo ukratko razmotriti osnovne vrste instrukcije jer nam njihovo razmatranje olakšava razumijevanje nekih pojava povezanih s funkcijama operacijskog sustava. Za naš ćemo model procesora predvidjeti neki zamišljeni oblik instrukcija koje olakšavaju objašnjavanje osnovnih činjenica. Stvarni se procesori međusobno razlikuju oblicima i detaljima izvođenja instrukcija, ali im je djelovanje sukladno našem pojednostavljenom modelu.

Instrukcije procesora mogu se podijeliti u nekoliko podskupina, i to:

- instrukcije za premještanje sadržaja između spremnika i registara procesora¹³;
- instrukcije za obavljanje aritmetičkih i logičkih operacija;
- instrukcije za programske skokove ili grananja;
- instrukcije za posebna upravljačka djelovanja.

Već smo rekli da se instrukcija sastoji od dva dijela: dijela za smještaj operacijskog koda i adresnog dijela instrukcije. Izgledi instrukcije različitih podskupina međusobno se donekle razlikuju.



Slika 2.7. Izgled instrukcije za premještanje sadržaja

Mogući izgled neke *instrukcije za premještanje* sadržaja prikazan je na slici 2.7. Ona se sastoji od tri dijela:

¹¹ RISC je kratica od engleskog *Reduced Instruction Set Computer* – računalo s reduciranim instrukcijskim skupom.

¹² CISC je kratica od engleskog *Complex Instruction Set Computer* – računalo sa složenim instrukcijskim skupom.

¹³ U engleskom se jeziku u opisu tih instrukcija koristi riječ *move* – pomaknuti, s tim da se posebno označava smjer premještanja ili nazivi *load* – nakrcati, napuniti (registrov) i *store* – pohraniti (sadržaj registra u spremnik).

- k bitova operacijskog koda,
- p bitova za izbor registra i
- r bitova za pomaknuće.

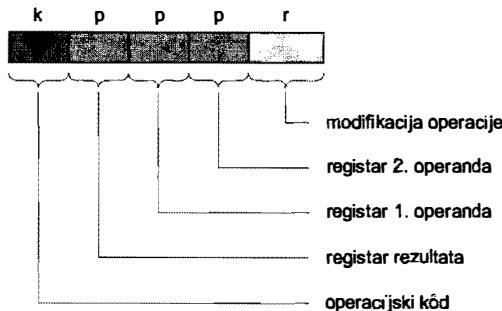
S k bitova operacijskog koda moguće je razlikovati 2^k instrukcija. Od toga su nam potrebne dvije kodne vrijednosti za premeštanje: jedna za punjenje registra iz spremnika (dohvat iz spremnika) i druga za pohranjivanje sadržaja registra u spremnik.

S p bitova za izbor registra može se odabrati jedan od najviše 2^p registara. Tako bi, primjerice, procesor sa 64 registra (od R_0 do R_{63}) trebao 6 bitova za izbor registra.

Preostalih $r = n - k - p$ bitova služi za zapisivanje sadržaja koji možemo nazvati pomaknućem (engl. *displacement, offset*) s pomoću kojeg se određuje adresa u spremniku. Naime, adresiranje spremnika može se obaviti tako da se pomaknuće iz instrukcije pribroji nekoj početnoj baznoj adresi pohranjenoj u nekom "skrivenom" pomoćnom adresnom spremniku. Na taj se način s manje bitova iz instrukcije može dobiti puni broj bitova spremničke adrese.

Mogući izgled troadresne instrukcije za *obavljanje operacije* prikazan je slikom 2.8. Ona ima pet dijelova:

- k bitova operacijskog koda,
- tri puta po p bitova za izbor registara i
- r bitova za možebitnu modifikaciju instrukcije, što povećava broj mogućih operacija.



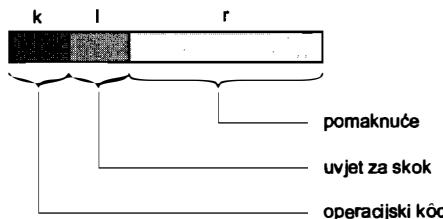
Slika 2.8. Izgled instrukcije za obavljanje operacije

Operacijski kôd određuje vrstu operacije. Upravljačka jedinka nakon dekodiranja operacijskog koda i r modifikacijskih bitova postavlja upravljačke bitove aritmetičko-logičkoj jedinki i na temelju sadržaja triju grupa od po p bitova zatvara preklopke na unutarnjim sabirnicama S_1 , S_2 i S_3 . Posljedica takve instrukcije jest obavljanje zatražene operacije s operandima iz registara operanada i pohranjivanje rezultata u registar rezultata.

Izgled instrukcije za *programske skokove* prikazuje slika 2.9. Operacijskim kodom određuje se vrsta skoka. Dodatnih l bitova može se iskoristiti za definiranje uvjeta za skok. Pomaknuće, na isti način kao kod instrukcija za premeštanje sadržaja, određuje adresu. Ta će adresa biti adresa sljedeće instrukcije koju treba izvesti. Dakle, instrukcijama skoka



narušit će se slijedno izvođenje programa koje inače procesor automatski obavlja. Tim instrukcijama izaziva se "skakanje" ili "grananje" programa, pa otuda dolazi i naziv tim instrukcijama¹⁴.



Slika 2.9. Izgled instrukcije za programske skokove

Instrukcije se mogu podjeliti na:

- instrukcije za bezuvjetne skokove;
- instrukcije za uvjetovane skokove.

Instrukcije za *bezuvjetne skokove* izazivaju skok u programu na adresu koja je određena pomaknućem adresnog dijela instrukcije, i to bez ispitivanja bilo kakvog uvjeta. Procesor izvodi takvu instrukciju na sljedeći način:

```

ponavljati {
    dohvati iz spremnika instrukciju na koju pokazuje programsko brojilo;
    dekodirati instrukciju, odrediti operaciju koju treba izvesti;
    povećati sadržaj programskog brojila tako da pokazuje na sljedeću instrukciju;
    ako je (dekodirana instrukcija skoka)
        iz pomaknuća zapisanog u instrukciji odrediti adresu i smjestiti tu
        adresu u programsko brojilo;
    inače
        obaviti instrukciju na način određen dekodiranim operacijskim kodom;
}
dok je (procesor uključen);

```



U početnom dijelu izvođenja, sve do dekodiranja instrukcije, procesor obavlja jednak posao kao i za sve ostale instrukcije: dohvaća instrukcije i povećava sadržaj programskog brojila tako da pokazuje na sljedeću instrukciju po redu. Ako je upravljačka jedinka dekodirala instrukciju bezuvjetnog skoka, onda će se unutar instrukcije taj sadržaj programskog brojila promijeniti tako da pokazuje na novu adresu. Sljedeća instrukcija koja će se izvesti, kada se procesor vrati na početak izvođenja petlje koja opisuje izvođenje instrukcija, započet će dohvatom instrukcije s te nove adrese.

Unutar instrukcija za *uvjetovane skokove* ispituje se uvjet zadan instrukcijom. Uvjeti koji određuju da li će se skok obaviti ili ne određeni su vrijednošću zastavica (engl. *flag*)

¹⁴ U engleskom se koriste nazivi *jump instructions* – instrukcije skoka i *branch instructions* – instrukcije grana.

zapisanih u pojedinom bitovima *registra stanja* procesora. Zastavice označavaju posebne slučajeve rezultata neke od prethodnih operacija koje je procesor izveo, tako da riječima možemo izreći neke od mogućih instrukcija uvjetovanih skokova ovako:

- “skočiti ako je rezultat bio nula”,
- “skočiti ako je rezultat bio različit od nule”,
- “skočiti ako je rezultat bio manji od nule”,
- “skočiti ako je rezultat bio veći od nule ili jednak nuli”.

Procesor obavlja instrukciju uvjetovanog skoka ovako:



```

ponavljati {
    dohvati iz spremnika instrukciju na koju pokazuje programsko brojilo;
    dekodirati instrukciju, odrediti operaciju koju treba izvesti;
    povećati sadržaj programskog brojila tako da pokazuje na sljedeću instrukciju;
    ako je (dekodirana instrukcija uvjetovanog skoka ^ uvjet ispunjen)
        iz pomaknuća zapisanog u instrukciji odrediti adresu i smjestiti
            tu adresu u programsko brojilo;
    inače
        obaviti instrukciju na način određen dekodiranim operacijskim kodom;
}
dok je (procesor uključen);

```

Za razliku od instrukcije bezuvjetnog skoka ovdje upravljačka jedinka ispituje odgovarajući bit u registru stanja i samo ako je uvjet ispunjen mijenja sadržaj programskog brojila. Kada uvjet nije ispunjen, sadržaj programskog brojila ostat će nepromijenjen i program će se nastaviti izvoditi instrukcijom koja slijedi instrukciju skoka.

Dakle, instrukcije uvjetovanih skokova omogućuju grananje programa i *ostvarivanje programskih petlji*. Petlje koje smo upoznali u našem jeziku zasnivanja programa mogu se prevesti u strojni oblik programa.

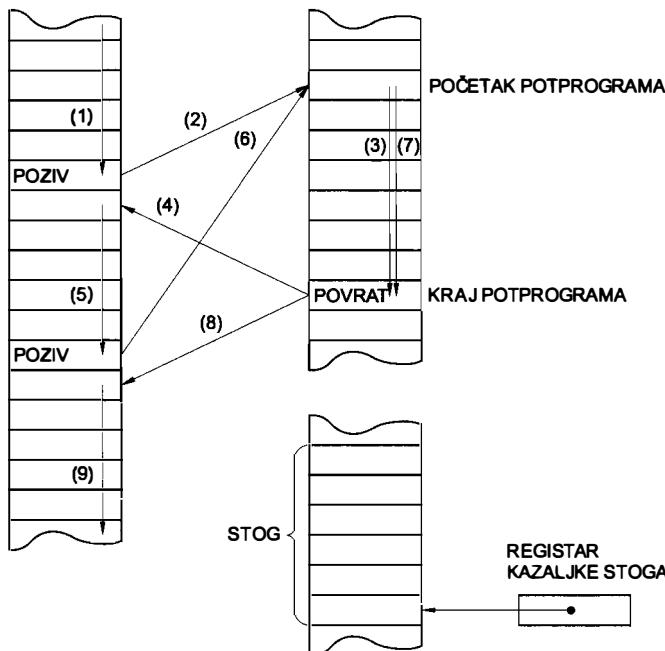
2.2. Instrukcije za poziv potprograma i povratak iz potprograma

Pokazalo se vrlo praktičnim neke programske zadatke koji se često ponavljaju pripremiti kao potprograme, za koje se u višim programskim jezicima koriste nazivi: procedura (*procedure*), funkcija (*function*), rutine (*routine, subroutine*). Mi ćemo za strojni oblik programa koristiti naziv potprogram, s tim da ćemo za neke posebne programske odsječke koristiti posebne nazive koje ćemo uvesti kasnije. Unutar programa potprogrami se “pozivaju” na onim mjestima gdje je to potrebno, oni obavljaju svoj dio posla i nakon toga program mora nastaviti svoje izvođenje instrukcijom koja se nalazi na adresi iza instrukcije poziva potprograma.

Dvije posebne instrukcije skoka služe za ostvarenje potprograma. To su:

- instrukcija za poziv potprograma;
- instrukcija za povratak iz potprograma.

Način odvijanja programa s pozivima potprograma prikazan je na slici 2.10. Spremnik računala prikazan je na slici razlomljen u tri dijela kako bi se jednostavnije prikazao tijek izvođenja programa. U jednom dijelu spremnika neka je pohranjen program. U drugom dijelu spremnika smješten je programski odsječak potprograma. Način odvijanja programa s pozivima potprograma prikazan je na slici 2.10.



Slika 2.10. Tijek odvijanja programa s pozivima potprograma

Treći posebno prikazani dio spremnika rezerviran je za smještanje *stoga*. Ustanovili smo u opisu procesora da u skupu registara mora postojati register kazaljke stoga – SP. Prije izvođenja programa u taj se register mora zapisati početna vrijednost adrese koja pokazuje na najvišu adresu dijela spremnika rezerviranog za stog. Nakon svakog smještanja nekog sadržaja na stog adresa zapisana u registru SP smanjuje se tako da on pokazuje na sljedeću slobodnu adresu. Ako na stog stavljamo ponovno novi sadržaj, stog će “rasti” prema manjim adresama (moguća je i izvedba stoga u kojoj kazaljka stoga početno pokazuje na najmanju adresu prostora predviđenog za stog i da stog raste prema višim adresama). Pri uzimanju sa stoga najprije će se povećati adresa u SP i nakon toga uzeti sadržaj zauzeti podatak zapisan na adresiranoj lokaciji. To se može nastaviti činiti tako dugo dok na stogu ima pohranjenih sadržaja. Prema tome, mehanizam stogovnog adresiranja ostvaren u procesoru omogućuje da se sadržaji dohvataju obrnutim redoslijedom od njihova stavljanja na stog, tj. obrnuto od redoslijeda njihova prispjeća¹⁵.

¹⁵ U engleskom se jeziku za takav način pohranjivanja sadržaja koristi kratica LIFO od “last in first out” – “zadnji unutra,

Na slici 2.10. posebno je prikazan registar kazaljke stoga, koji je inače sastavni dio procesora. Odvijanje programa, koji dva puta poziva jedan potprogram, simbolički je prikazano strelicama s brojevima koji pokazuju redoslijed njihova izvođenja. Program se odvija na sljedeći način:

- (1) izvodi se niz instrukcija programa;
- (2) izvodi se skok na potprogram;
- (3) izvodi se niz instrukcija potprograma;
- (4) skače se natrag u program na instrukciju iza prvog poziva potprograma;
- (5) izvodi se niz instrukcija programa;
- (6) ponovno se skače na potprogram;
- (7) drugi put se izvodi niz instrukcija potprograma;
- (8) skače se natrag u program na instrukciju iza drugog poziva potprograma;
- (9) nastavlja se izvođenje instrukcija programa.

Skokovi iz programa u potprogram jednoznačno su određeni. Naime, adresa početka procedure mora biti poznata i može se zapisati u instrukciju skoka koju u ovom slučaju zovemo "pozivom potprograma" (engl. *procedure call*). Međutim, instrukcija povratnog skoka iz potprograma u program, ili kraće: instrukcija povrata (engl. *return*), morala bi u svom adresnom dijelu imati upisane različite adrese jer se natrag u program skače na različita mjesta. Taj nam problem razrješava stogovno adresiranje s pomoću regista kazaljke stoga procesora, kao što se vidi u opisu instrukcije za poziv potprograma. Procesor izvodi instrukciju poziva potprograma ovako:



```

ponavljati {
    dohvati iz spremnika instrukciju na koju poka uje programsko brojilo;
    dekodirati instrukciju, odrediti operaciju koju treba izvesti;
    povećati sadržaj programskog brojila tako da pokazuje na sljedeću instrukciju;
    ako je (dekodirana instrukcija poziva potprograma) {
        pohraniti sadržaj programskog brojila na stog;
        smanjiti sadržaj регистра SP tako da pokazuje na sljedeće prvo mjesto;
        iz adresnog dijela instrukcije odrediti adresu početka potprograma;
        smjestiti tu adresu u programsko brojilo;
    }
    inače {
        obaviti instrukciju na način određen dekodiranim operacijskim kodom;
    }
    dok je (procesor uključen);
}

```

Dakle, tijekom izvođenja instrukcije poziva procedure sadržaj programskog brojila, koji je već uvećan tako da pokazuje na sljedeću instrukciju, pohranjuje tu adresu na vrh stoga, a

prvi van". Pohranjivanje sadržaja u kojem se oni dohvaćaju po redu prispjeća obilježava se kroz FIFO od "first in first out" – "prvi unutra prvi van".

u programsko se brojilo zapisuje adresa prve instrukcije potprograma. Sljedeća instrukcija koju će nakon toga izvesti procesor bit će upravo ta instrukcija.

Instrukciju povratka iz potprograma procesor obavlja ovako:

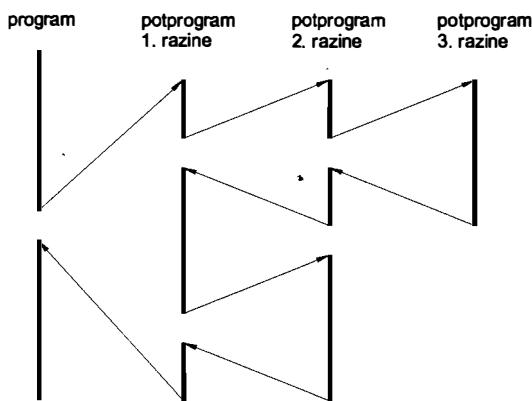
```

ponavljati {
    dohvatiti iz spremnika instrukciju na koju pokazuje programsko brojilo;
    povećati sadržaj programskog brojila tako da pokazuje na sljedeću instrukciju;
    dekodirati instrukciju, odrediti operaciju koju treba izvesti;
    ako je (dekodirana instrukcija povratka iz potprograma) {
        povećati sadržaj registra kazaljke stoga SP;
        premjestiti sadržaj sa vrha stoga na koji pokazuje SP u programsko
        brojilo;
    }
    inače
        obaviti instrukciju na način određen dekodiranim operacijskim kodom;
}
dok je (procesor uključen);

```



Izvođenje te instrukcije jednostavno je: adresa koja se nalazi na vrhu stoga prebacuje se u programsko brojilo i sljedeća instrukcija koju će procesor sada dohvatiti bit će instrukcija smještena iza instrukcije kojom je potprogram pozvan. To, jasno, vrijedi samo onda ako u međuvremenu stog nije mijenjan, odnosno ako su promjene na stogu tako pažljivo obavljene da se na njegovu vrhu u trenutku povratka nalazi prava povratna adresa.



Slika 2.11. Pozivanje potprograma unutar potprograma

Ovaj mehanizam poziva potprograma omogućuje da se unutar jednog potprograma poziva drugi potprogram. Takvi pozivi mogu se proizvoljno ponavljati, što dopušta proizvoljni broj razina ugniježđenih potprograma. Tijek izvođenja takvih ugniježđenih potprograma ilustrira slika 2.11.

2.2.1. Načini razmjene podataka između potprograma i programa

Potprogram obavlja neku pretvorbu ulaznih parametara u izlazne rezultate. Prema tome, pri pozivu potprograma njemu treba na neki način prenijeti ulazne podatke, a potprogram pri završetku treba vratiti rezultate programu koji ga je pozvao (ili drugom potprogramu koji ga je pozvao). U načelu se razmjena podataka može obaviti na dva osnovna načina:

- prenošenjem vrijednosti parametara ili
- prenošenjem adrese parametara.

Pri pozivu s *prenošenjem vrijednosti parametara* (engl. *call – by – value*) potprogramu se prenose vrijednosti parametara s kojima on mora obaviti programirane operacije. Takav način poziva prikidan je onda kada broj parametara nije prevelik. Postoje razne mogućnosti ostvarenja takvog pozivā, no mi se ovdje nećemo time posebno baviti. Spomenimo samo da je najjednostavnije prenošenje parametara s pomoću registara procesora. Prije poziva potprograma program treba u neke registre procesora smjestiti ulazne vrijednosti (parametre) potprograma, a instrukcije potprograma moraju te vrijednosti preuzeti iz tih registara. Rezultate koje je proizveo, potprogram opet može smjestiti u registre procesora, te će nakon povratka iz potprograma program u registrima pronaći tražene rezultate.

Umjesto registara procesora može se za prenošenje parametara koristiti i *stog*. Pritom program mora prije poziva potprograma na stog staviti parametre (to mogu biti ili stvarne vrijednosti parametara ili adrese parametara). Nakon poziva potprograma na vrhu stoga nalazi se povratna adresa, a ispod nje parametri. Prema tome, potprogram mora najprije sa stoga skinuti i negdje pohraniti tu povratnu adresu. Nakon toga on sa stoga može preuzeti sve parametre i obaviti svoj posao, te rezultate staviti na stog. Na kraju on mora na stog vratiti sačuvanu povratnu adresu. Program će sa stoga, nakon obavljene instrukcije povratka iz potprograma, moći preuzeti vraćene rezultate.

Pri pozivu s *prenošenjem adrese parametara* (engl. *call – by – reference*) potprogramu se ne prenose vrijednosti parametara nego adrese spremničkih lokacija u kojima su smještene vrijednosti parametara. Takav poziv nužno je koristiti onda kada se od potprograma zahtijeva transformiranje velike količine podataka. Prije poziva potprograma, u registre procesora pohranjuju se početne adrese i veličine strukture podataka koju želimo prenijeti potprogramu, kao i adrese strukture u koju će potprogram smještati rezultate. Posebno treba naglasiti da poziv s prenošenjem adresa može biti ostvaren samo onda kada sve instrukcije potprograma mogu dohvati isti adresni prostor kao i instrukcije programa. U jednostavnom modelu računala to je uvijek moguće. Međutim, vidjet ćemo ubrzo da u malo složenijem modelu adresni prostor više nije jedinstven i da poziv potprograma s prenošenjem adresa nije moguće provesti.

U različitim programskim jezicima mogu se uočiti znatne razlike u ostvarivanju mehanizama poziva potprograma. U nekim se programskim jezicima, ili bolje rečeno, *jezičnim procesorima*¹⁶ potprogrami definiraju na različite načine. Tako, primjerice, programski

¹⁶ *Jezični procesor* (engleski: *language processor*) naziv je za objedinjenu sklopovsko-programsku tvorevinu koja podržava postupak prevodenja programa napisanog u višem programskom jeziku u strojni oblik (engleski *compile-time support*) i zatim izvođenje tog strojnog programa (engl. *run-time support*).

jezik Pascal razlikuje dva oblika potprograma: procedure i funkcije. Funkcije se od procedura razlikuju po tome što vraćaju jednu vrijednost i mogu se smatrati nekom vrstom operatorka, dok procedure djeluju na varijable deklarirane u popisu parametara. U programskim jezicima C i C++ svi su potprogrami istog oblika i nazivaju se funkcijama. U deklaraciji funkcija mora se označiti koji tip vrijednosti će funkcije vratiti, pri čemu se može posebno (deklaracijom void) označiti da funkcija ne mora vratiti nikakvu vrijednost i tada je ona slična proceduri iz programskog jezika Pascal. Mi u dalnjem tekstu nećemo činiti takve razlike i koristit ćemo naziv potprogram.

Nazive *procedura* i *poziv procedure* koristit ćemo kasnije kada budemo govorili o posebnom načinu razmjene podataka i poruka. Sa stanovišta operacijskih sustava bilo bi prikladno ujednačiti načine i modele poziva procedure i prijenosa parametara.

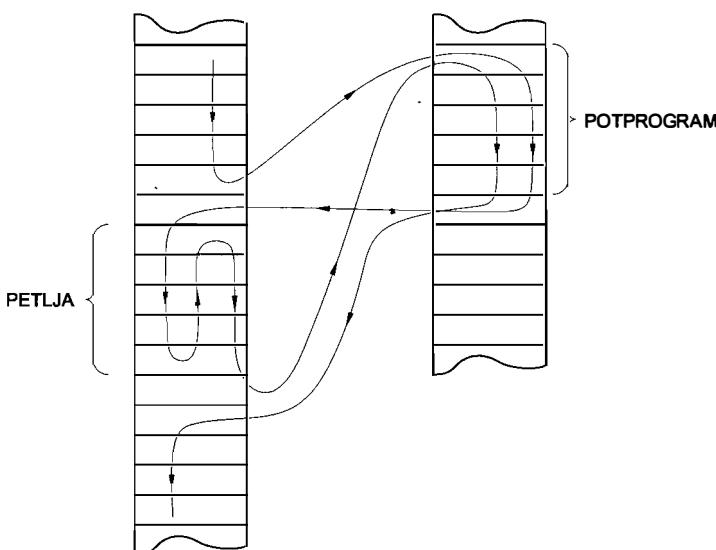
2.2.2. Instrukcijska dretva

Na osnovi svega što je u ovom poglavlju rečeno, možemo utvrditi da se naš jednostavni model računala sastoji od:

- spremnika i
- procesora.

Možemo zamisliti da je spremnik podijeljen na tri dijela:

- dio u kojem je smješten program (kao niz strojnih instrukcija);
- dio rezerviran za stog;
- dio za smještanje podataka.



Slika 2.12. Instrukcijska dretva

U našem jednostavnom modelu – modelu rudimentarnog računala – procesor djeluje na

cjelovitom adresnom prostoru. Pretpostavljamo da adresni dio instrukcija omogućuje adresiranje tog cjelovitog adresnog prostora, što znači da instrukcije mogu dohvatiti podatke smještene bilo gdje u tom prostoru i da programsko brojilo može dohvatiti instrukcije smještene bilo gdje u tom prostoru.

Procesor se sastoji od skupa registara, upravljačke i aritmetičko-logičke jedinke. On se ponaša kao automat koji izvodi jednu instrukciju iza druge. Skup registara procesora može se podijeliti na:

- registre za privremeno pohranjivanje ulaznih podataka, međurezultata i rezultata, te adresa;
- programsko brojilo;
- registar stanja;
- registar kazaljke stoga. .

To je okruženje u kojem se može izvoditi program.

Mi, jasno, moramo pretpostaviti da je prije toga strojni program na neki način pohranjen u spremnik i da su, isto tako, početni ili ulazni podaci također pohranjeni u spremnik. Ako nakon toga na neki način u programsko brojilo upišemo adresu prve instrukcije programa, procesor će automatski početi izvoditi jednu instrukciju za drugom – time će započeti izvođenje tog programa. Može se reći da procesor “provlači” kroz niz instrukcija *instrucijsku dretvu* ili, kraće, samo *dretvu* (engl. *thread*).

2.3. Računalni proces

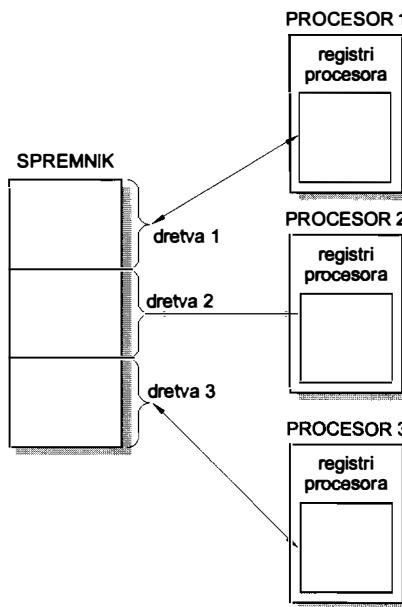
Program koji je zapisan na papiru ili pohranjen u strojnom obliku u spremniku računala statička je tvorevina – niz instrukcija koji izvoditelj programa, tj. procesor mora razumjeti i znati obaviti. Međutim, kada se program počinje izvoditi, njemu se mogu pripisati neka vremenska svojstva:

- trenutak početka izvođenja programa;
- trenutak završetka izvođenja programa;
- trajanje izvođenja programa;
- zaustavljanje izvođenja programa i sl.

Time izvođenje programa dobiva obilježje procesa. S obzirom na to da se proces odvija u računalu govorimo o *računalnom procesu* ili, ako je jasno da je riječ o računalnom procesu, onda možemo kraće reći samo *proces*.

Vidjet ćemo kasnije da pojам *računalnog procesa* obuhvaća ne samo instrukcije programa tijekom njihova izvođenja, već i sve druge pojave koje su povezane s izvođenjem tog programa (strukture podataka i datoteke koje on koristi, ostvarenje ulaznih i izlaznih operacija i sl.). Prije započinjanja nekog procesa u računalu se moraju stvoriti uvjeti za njegovo izvođenje. Osnovni je zadatak operacijskog sustava stvaranje takvih uvjeta za odvijanje programa u kojima se izvođenje može opisati ovakvim modelom koji se zasniva na opisu rudimentarnog računala.

Prema tome, mi u stvarnim računalnim sustavima možemo ponašanje programa promatrati kao proces. Unutar procesa mora postojati barem jedna instruksijska dretva. Operacijski sustav mora uspostaviti sve uvjete za odvijanje procesa. Između ostalog, on mora osigurati nesmetano izvođenje dretve, tj. osigurati spremnički prostor i procesor u skladu s modelom našeg rudimentarnog računala. Brzina izvođenja dretve određena je brzinom procesora. Katkada će se izvođenje dretve morati na tren zaustaviti i kasnije nastaviti.



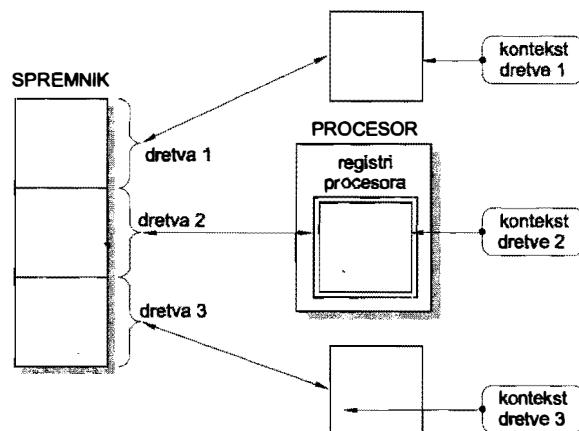
Slika 2.13. Višedretveni rad na višeprocesorskom računalu

U računalima s više procesora (u višeprocesorskim računalima) proces bismo mogli podijeliti na *više dretvi*. Spremnik bi tada trebalo podijeliti tako da svaka dretva dobije svoj dio adresnog prostora iz kojeg pripadni procesor dohvata instrukcije i podatke. Svaki procesor "provlačio" bi svoju dretvu kroz instrukcije i tako neki zajednički posao može biti brže obavljen (sl. 2.13.). Kažemo da takva računala podržavaju *višedretveni rad* (engl. *multithreading*).

U današnje vrijeme gotovo svi (višenamjenski) računalni sustavi podržavaju višedretveni rad, što zbog jednostavnosti izvedbe primjenskih programa, što zbog povećanja učinkovitosti iskorištenja svih dostupnih resursa sustava (npr. iskorištenje svih procesorskih jedinki višestrukog procesora na čipu).

Štoviše, višedretveni se rad može provesti i u jednoprocesorskom računalu, i to tako da taj jedan jedini procesor naizmjence "provlači" jednu od više dretvi. U tom se slučaju ne može govoriti o ubrzavanju odvijanja procesa, ali se može postići da procesor izvodi jednu od dretvi za vrijeme dok druga mora zbog nekog razloga čekati. S obzirom na to da svaka dretva za izvođenje treba "svoj procesor", u jednoprocesorskom sustavu mora se, pri prebacivanju izvođenja s jedne dretve na drugu, osigurati da svaka dretva radi sa svojim skupom registara. To se može postići tako da se sadržaj registara procesora one dretve čije se izvođenje želi prekinuti pohrani na neko rezervirano mjesto u spremniku, a

u registre procesora smjesti sadržaje koji pripadaju dretvi čije izvođenje upravo treba započeti (sl. 2.14.). Ako se takva promjena sadržaja registara obavi ispravno, onda se stvara privid da svaka dretva posjeduje "vlastiti" procesor. Sadržaj registara procesora zovemo *kontekstom dretve*, a promjenu sadržaja registara *promjenom konteksta* (engl. *context switching*).



Slika 2.14. Višedretveni rad na jednoprocesorskom računalu

S obzirom na to da je promjena konteksta temeljni mehanizam na kojem počiva ostvarenje svih funkcija operacijskih sustava, njome ćemo se u sljedećem poglavlju posebice baviti.



PITANJA ZA PROVJERU ZNANJA 2

1. Čime su određena svojstva i ponašanje procesora?
2. Navesti osnovni skup registara procesora.
3. Što je to sabirnički ciklus?
4. U pseudokodu napisati što procesor trajno radi.
5. Što je kontekst dretve?
6. Što se zbiva pri izvođenju instrukcije za poziv potprograma?
7. Definirati osnovne pojmove: program, proces, dretvu.
8. Kako je moguć višeprogramska rad na jednoprocesorskom računalu?

3.

Obavljanje ulazno-izlaznih operacija, prekidni rad

3.1. Priključivanje ulazno-izlaznih naprava

U model rudimentarnog računala iz prethodnog poglavlja treba dodati ulazne i izlazne dijelove kako bi se on nadopunio do Von Neumannova modela.

Uobičajene ulazne naprave svakog sobnog računala jesu:

- tipkovnica za unošenje pojedinačnih znakova u računalo i
- miš za postavljanje neke točke na zaslonu monitora i unošenje njezinih koordinata u računalo.

Najčešće izlazne naprave su:

- monitor i
- pisač.

Sve ulazne i izlazne naprave nazivamo zbirnim imenom *ulazno-izlazne naprave (UI-naprave)* ili *periferijske naprave*.

U skupinu tih naprava možemo ubrojiti i vanjske spremnike, koji su i ulazne i izlazne naprave. To su:

- magnetski diskovi;
- magnetske diskete;
- optički diskovi.

Osim ovih osnovnih naprava današnja računala mogu se opremiti raznovrsnim ulazno-izlaznim napravama i uređajima. Tako se na računalo mogu priključiti:

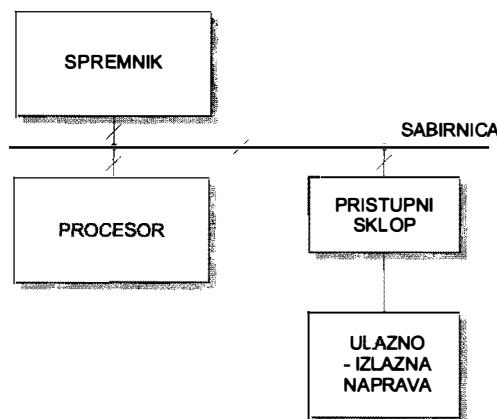
- naprave za čitanje znakova i slika;
- naprave za crtanje i izradu slika;
- videokamera;
- videorekorder;
- kompaktni disk (CD);

- mikrofon;
- zvučnici;
- muzička klavijatura.

Vidljivo je da su ulazno-izlazne naprave po svojim fizikalnim načelima konstrukcije i djelovanja vrlo šarolike. O tim načelima ovisi i brzina rada pojedinih naprava, pa su i one međusobno vrlo različite. Odvijanje posla u napravama u načelu je neovisno o brzini odvijanja programa u procesoru i stoga je potrebno posebnu brigu posvetiti vremenskom uskladivanju – sinkronizaciji – rada procesora i ulazno-izlaznih naprava.

Ipak je i u toj šarolikosti uvedena sustavnost u načine priključivanja ulazno-izlaznih naprava. U prvom redu svaka ulazno-izlazna naprava ima svoj vlastiti *upravljački elektronički sklop* koji se brine o svim detaljima ponašanja naprave. Naprava je spojena na *pristupni sklop računala* koji je s jedne strane prilagođen toj napravi, a s druge strane prilagođava se protokolima sabirničkog sustava. Osim sklopova za prenošenje sadržaja pristupni sklop mora omogućiti i ostvarenje sinkronizacije.

Slika 3.1. ilustrira način povezivanja neke naprave na sabirnicu računala.



Slika 3.1. Način povezivanja ulazno-izlazne naprave

Po načinu rada, naprave i njihove pristupne sklopove možemo podijeliti na dvije skupine:

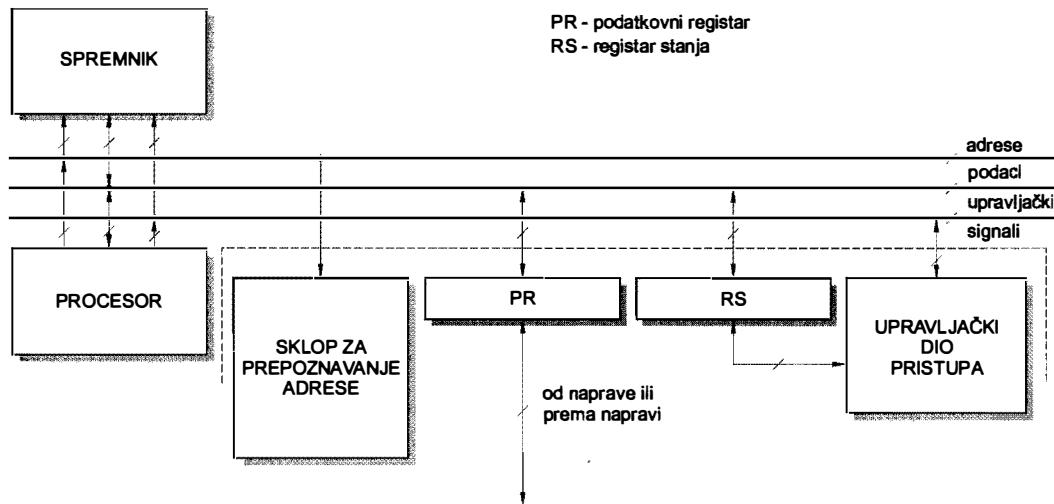
- naprave kod kojih se prenose *pojedinačni znakovi* (bajtovi) i
- naprave kod kojih se prenose *blokovi znakova* (niz uzastopnih bajtova).

U dalnjem ćemo se tekstu pozabaviti samo onim detaljima sklopolja koji su nužni za razumijevanje mehanizama sinkronizacije. Najprije ćemo razmotriti načela prijenosa pojedinačnih znakova, a zatim opisati i načela prijenosa blokova znakova s neposrednim pristupom spremniku. S neposrednim pristupom spremniku može se povezati i opis osnovnih načela višeprocesorskog rada, te je taj opis dodan na kraju ovog poglavlja.

3.2. Prenošenje pojedinačnih znakova, prekidni rad procesora

3.2.1. Prenošenje znakova radnim čekanjem

Na slici 3.2. prikazan je detaljniji izgled pristupnog sklopa za prijenos pojedinačnih znakova. Izgled pristupnog sklopa sa strane sabirnice podsjeća nas na izgled spremnika. Podatkovni registar pristupnog sklopa možemo promatrati kao jedan bajt spremnika. Na slici je prikazan i drugi registar – registar stanja – u čije se bitove mogu pohraniti informacije o stanju pristupnog sklopa, kao i upravljački bitovi koji mogu modificirati ponašanje pristupa. Procesor može te registre adresirati jednakom nacinom, tako da ih može i adresirati pojedine bajtove spremnika. Sklop za prepoznavanje adrese određuje koji je od registara adresiran, a posebni upravljački signal određuje da li se adresirani registar čita ili piše.



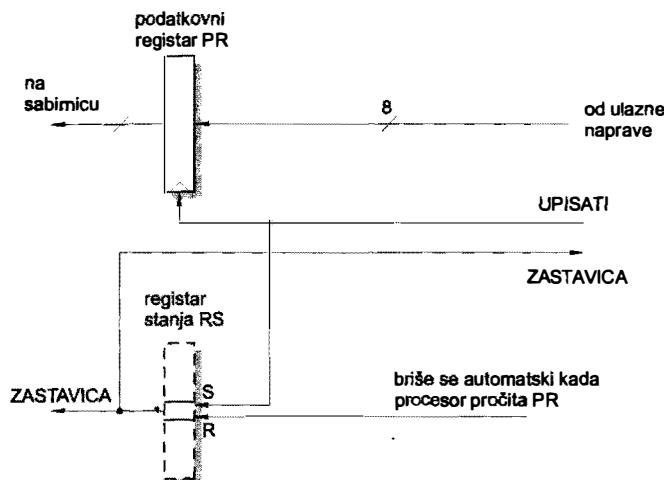
Slika 3.2. Pristupni sklop prijenosa pojedinačnog znaka

Pri obavljanju izlazne operacije, tj. pri prenošenju sadržaja nekog bajta spremnika prema izlaznoj napravi procesor mora:

- jednom instrukcijom dobiti taj sadržaj iz spremnika u svoj registar i
- drugom instrukcijom prenjeti taj znak u podatkovni registar pristupnog sklopa.

Pristupni sklop i upravljačko sklopolje naprave moraju osigurati da se taj sadržaj podatkovnog registra prenese dalje prema izlaznoj napravi. Pri prenošenju uzastopnih znakova procesor ne bi smio staviti sljedeći znak u podatkovni registar pristupnog sklopa ako prethodni znak još nije "potrošen". U registru stanja pristupnog sklopa posebni bit – izlazna zastavica – ostaje postavljen tako dugo dok znak ne bude prihvачen od strane izlazne naprave.

Na sličan način obavlja se i ulazna operacija, tj. prenošenje jednog znaka iz ulazne naprave u radni spremnik. Promotrit ćemo to detaljnije s pomoću slike 3.3.



Slika 3.3. Ulaz pojedinačnog znaka

Od ulazne naprave dolazi osam vodiča preko kojih se dovodi osam bitova jednog bajta. Sadržaj koji je tim vodičima doveden podatkovnog registra PR upisat će se u registar tek kada ulazna naprava (zapravo, upravljački sklop naprave) posebnim signalom UPISATI to naredi. Istodobno s upisom novog sadržaja u podatkovni registar postavlja se i posebni bit označen kao ZASTAVICA, što je simbolizirano dovodenjem signala UPISATI do oznaka S uz taj bit. Vrijednost bita ZASTAVICA, pretvorena u električki signal vraća se natrag prema ulaznoj napravi. Upravljački sklop naprave mora biti načinjen tako da signal UPISATI postavlja samo onda kada je ZASTAVICA spuštena. Tim mehanizmom omogućena je sinkronizacija sa strane ulazne naprave: ona će pohraniti novi znak u podatkovni registar samo kada je ZASTAVICA spuštena i automatski pri tom upisu ona će se automatski podići.

Sa strane procesora sinkronizacija se obavlja programski. Procesor će to obaviti izvođenjem kratkog programskog odsječka u kojem on uzastopce čita registar stanja RS pristupnog sklopa, i to tako dugo dok se ispitivanjem bita ZASTAVICA ne ustanovi da je on postavljen. Nakon što to ustanovi, treba pročitati podatkovni registar PR. Prenošenjem sadržaja registra PR u jedan od registara procesora (i nakon toga, ako je to potrebno, u spremnik) pristupni sklop postaje pripravan za prihvatanje novog znaka. Upravljački dio pristupnog sklopa stoga pri čitanju registra PR automatski spušta bit ZASTAVICA u nulu, što je simbolizirano oznakom R uz bit ZASTAVICA.

Za ilustraciju mehanizma međusobnog isključivanja poslužit će podskup mnemoničkih instrukcija arhitekture ARM procesora. Prepostavimo za procesor sljedeće instrukcije:

	<pre> LDR r0,[r1] ; prenijeti u r0 sadržaj s adresu koja se nalazi u r1 STR r0,[r1] ; prenijeti sadržaj r0 na adresu koja se nalazi u r1 ADR r1,SIMBIME ; smjestiti u r1 adresu čije je simboličko ime SIMBIME CMP r0,#0 ; usporedba r0 s nulom BEQ PETLJA ; skočiti na PETLJA ako prethodna usporedba daje jednakost BAL PETLJA ; skočiti uvijek </pre>
--	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Opisani prihvati jednog znaka od ulazne naprave sažeto se može opisati sljedećim programskim odsječkom:

```

ADR r1,RS ; 
ADR r2,PR ; 
PETLJA LDR r0,[r1] ; čitati RS
    CMP r0,#0 ; (ostali bitovi su 0)
    BEQ PETLJA ;
    LDR r0,[r2] ; čitati PR

```

Isti programski odsječak napisan u pseudokodu glasi:

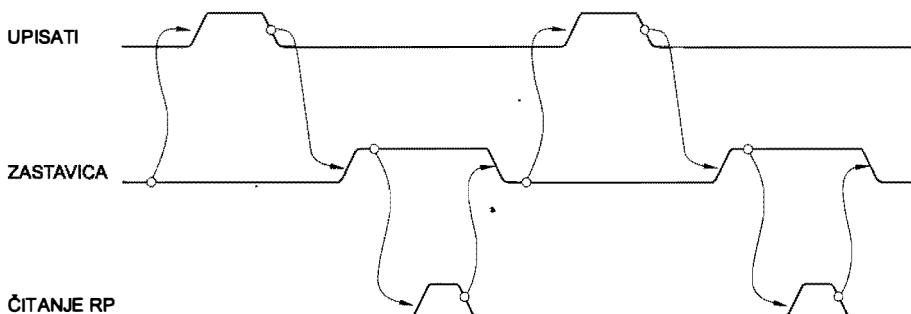
```

pročitati registar RS;
dok je (ZASTAVICA = 0) {
    pročitati registar stanja RS;
}
pročitati registar PR; //time se automatski briše ZASTAVICA

```

Primijetimo da se taj odsječak sastoji samo od nekoliko strojnih instrukcija.

Vodići UPISATI i ZASTAVICA “vidljivi su” na izvodima pristupnog sklopa i na njima su električki signali. Kada bismo, dakle, na njih priključili osciloskop i gledali što se događa na njima pri uzastopnom unošenju niza znakova, uočili bismo promjene napona ilustrirane s prva dva reda slike 3.4.



Slika 3.4. Protokol "dvožičnog rukovanja"

Na slici je dodan i treći redak koji predviđa pojavu čitanja registra podataka. Male pomoćne strelice na slici označavaju uvjetovanost ili uzročnost pojedinih zbivanja. Tako kružić na niskoj razini signala ZASTAVICA označava da ulazna naprava smije generirati signal UPISATI. Kružić na padajućem bridu signala UPISATI uzrokuje podizanje signala ZASTAVICA, a podignuta ZASTAVICA je preduvjet za čitanje registra PR. U trećem je redu slike simbolički predstavljeno čitanje registra PR, kako bi se pokazalo da završetak čitanja uzrokuje spuštanje signala ZASTAVICA.

Pristupni sklop takvim svojim ponašanjem određuje protokol komuniciranja između ulazne naprave i računala. S obzirom na to da se signali na dva vodiča ("žice") naizmjence podižu i spuštaju taj je protokol nazvan dvožičnim rukovanjem (engl. *two-wire handshaking*). Na sličan način može se obavljati i prijenos pojedinačnih znakova iz računala u izlaznu napravu.

Velik nedostatak ovakvog načina prijenosa znakova jest programsko ispitivanje bita ZASTAVICA. Procesor izvodi instrukcije male *petlje čekalice* za vrijeme čekanja na *događaj* (engl. *event*) podizanja signala ZASTAVICA. Taj je događaj potpuno nezavisan o ponašanju procesora. Njegova je pojava vremenski neodređena. Procesor, prema tome, troši svoje vrijeme izvodeći bespotrebne instrukcije i troši svoje, pa i sabirničke cikluse, koji bi možda mogli biti iskorišteni na drugi način.

PRIMJER 3.1.



Pretpostavimo, primjerice, da računalo treba prihvati znakove koji pristižu brzinom od 1000 znakova u sekundi, tj. u razmaku od 1 ms i da procesor ima brzinu od 10 MIPS. U tom slučaju procesor bi između dolazaka dvaju znakova mogao izvesti 10000 instrukcija. Pretpostavimo nadalje da procesor trajno izvodi sljedeći programski odsječak kojim čita nadolazeće znakove:



```

ADR r1,RS      ; 
ADR r2,PR      ; 
ADR r3,POREDAK ; 
PETLJA LDR r0,[r1]   ; } malo
                  ; } petlja
                  ; } čekalica
                  CMP r0,#0      ; } 
                  BEQ PETLJA    ; } 
LDR r0,[r2]      ; } 
STR r0,[r3]      ; } 
INC r3          ; } 
BAL PETLJA      ; } velika
                  ; } petlja

```

U pseudokodu navedeni programski odsječak glasi:



```

ponavljati {
    čitati registar RS;
    dok je (ZASTAVICA = 0) {
        čitati registar RS;
    }
    čitati znak iz podatkovnog registra PR;
    pohraniti pročitani znak u spremnik;
}
dok je (procesor uključen);

```

Taj se odsječak programma sastoji od dva dijela: dijela koji se sastoji od petlje čekalice u kojoj procesor samo troši vrijeme i dijela u kojem se znak prenosi iz podatkovnog registra pristupnog sklopa u jedan bajt spremnika. Koristan posao obavlja samo ovaj drugi dio odsječka od svega desetak strojnih instrukcija. To znači da samo 0.1% vremena procesor radi koristan posao, a ostatak od 99.9% utrošen je za čekanje.

3.2.2. Prekidni način rada procesora

Potpuno je razumljivo da radno čekanje na asinkroni događaj dolaska znaka nije prikladno rješenje. Za vrijeme čekanja na dolazak znakova procesor bi mogao obavljati neku drugu radnu dretvu, a na poseban način trebalo bi ga upozoriti da je novi znak pristigao u podatkovni registar pristupnog sklopa i prekinuti izvođenje te dretve.

Ta je zamisao ostvarena mehanizmom prekidnog načina rada procesora (engl. *interrupt mechanism*). Za ostvarenje prekidnog rada potrebno je u naš pojednostavljeni model računala dodati još neka svojstva kako pristupnog sklopa, tako i procesora.

Model pristupnog sklopa mora biti modifiran tako da je podizanje bita ZASTAVICA u registru stanja popraćeno generiranjem električkog signala koji se preko posebnog vodiča dovodi do procesora. Taj signal "upozorava" procesor da je novi znak pristigao.

Ponašanje procesora pri izvođenju svake instrukcije treba nadopuniti tako da on na kraju izvođenja svake instrukcije ispituje je li se pojavio prekidni signal. Detalji prihvata tog prekidnog signala u različitim arhitekturama izvode se na različite načine. Mi ćemo ovdje taj mehanizam opisati na pojednostavljeni način kako bismo s razumijevanjem mogli izučiti svojstva jezgre operacijskog sustava i osnovne pojave u računalnim sustavima.

Pojava prekidnog signala prebacuje procesor u tzv. *sustavski način rada*. S obzirom na to da će se na taj način pozivati potprogrami koji čine jezgru operacijskog sustava taj se način rada procesora naziva još i *jezgrenim načinom rada* (engl. *system mode* – sustavski način, *kernel mode* – jezgreni način). Dretva koja je pojavom prekidnog signala prekinuta najčešće izvodi neki korisnički posao i zbog toga kažemo da se ona obavlja u *korisničkom načinu rada* (engl. *user mode*).

U našem modelu procesor pri prelasku u jezgreni način djeluje tako da:

- privremeno onemogući daljnje prekidanje;
- djeluje na adresni dio sabirnice tako da adresira odvojeni dio spremnika (ime je spremnik podijeljen na dva dijela: korisnički dio i sustavski dio);
- aktivira drugi sustavski registar kazaljke stoga, koji se uz već spomenuti korisnički registar kazaljke stoga nalazi u procesoru (ime se omogućuje ostvarenje posebnog sustavskog stoga, koji nema vezu s korisničkim stogom, taj se stog mora nalaziti u jezgrenom adresnom prostoru);
- pohranjuje trenutačni sadržaj programskog brojila na sustavski stog;

- u programsko brojilo stavlja adresu na kojoj počinje potprogram za obradu prekida (ta se adresa nalazi u jezgrenom adresnom prostoru).

Opis djelovanja procesora pri izvođenju instrukcija koji smo uveli u prethodnom poglavlju treba, prema tome, nadopuniti, pa on izgleda ovako:



```

ponavljati {
    dohvati iz spremnika instrukciju na koju pokazuje programsko brojilo;
    dekodirati instrukciju, odrediti operaciju koju treba izvesti;
    povećati sadržaj programskog brojila tako da pokazuje na sljedeću instrukciju;
    odrediti odakle dolaze operandi i kuda se pohranjuje rezultat;
    operande dovesti na aritmetičko-logičku jedinku, izvesti zadani operaciju;
    pohraniti rezultat u odredište;
    ako je (prekidni signal postavljen) {
        onemogućiti daljnje prekidanje;
        prebaciti adresiranje u sustavski adresni prostor i aktivirati sustavsku
        kazaljku stoga;
        pohraniti programsko brojilo na sustavski stog;
        staviti u programsko brojilo adresu potprograma za obradu prekida;
    }
}
dok je (procesor uključen);

```

Postavlja se pitanje odakle se dobavlja adresa koju treba staviti u programsko brojilo. Možemo zamisliti da je ta adresa pohranjena u jedan od "skrivenih registara" unutar procesora pri proizvodnji procesora. To znači da na tu adresu treba smjestiti prvu instrukciju programskog odsječka u kojem će se obaviti posluživanje prekida.

Prekidni signal, prema tome, djeluje kao sklopovski izazvani poziv potprograma. Za razliku od uobičajenog potprograma ovaj je poziv došao u neodređenom trenutku. Procesor u svojim registrima ima pohranjene sadržaje koji čine trenutačni kontekst dretve koja se upravo izvodi. Taj će kontekst trebati sačuvati kako bismo prekinutu dretvu kasnije mogli nesmetano nastaviti. Taj je kontekst najjednostavnije pohraniti na sustavski stog. Na tom se stogu već nalazi adresa programskog brojila (jer to automatski obavlja procesor pri pojavi prekidnog signala). Prema tome, prvi posao potprograma za obradu prekida mora biti pohranjivanje sadržaja svih registara procesora na sustavski stog (u te se registre treba ubrojiti i korisnički registar kazaljke stoga). U instrucijskom skupu procesora postoje posebne instrukcije za pohranjivanje registara na stog. Mi ćemo tu operaciju u dalnjem tekstu nazivati pohraniti kontekst.

Nakon što je potprogram za posluživanje prekida (kažemo i: obradu prekida) obavio svoj posao, primjerice, prihvati ili ispis znaka, treba ponovno pokrenuti prekinutu dretvu korisničkog programa. To se vraćanje obavlja tako da se najprije sa sustavskog stoga posebnim instrukcijama vraća sadržaje registara (oni se vraćaju obrnutim redoslijedom od onoga kojim su na stog stavljeni). Tu ćemo operaciju u dalnjem tekstu nazivati obnoviti kontekst. Nakon te operacije na stogu ostaje samo sadržaj programskog brojila.

Prije vraćanja tog sadržaja u procesor mora se procesoru omogućiti prekidanje. U skupu instrukcija procesora postoji posebna instrukcija koju ćemo nazvati omogućiti prekidanje. Ta instrukcija djeluje tako da će prekidanje biti omogućeno tek nakon što se posebnom instrukcijom u procesor vrati sadržaj programskog brojila. Novi prekid će, dakle, biti prihvачen tek kada se procesor potpuno vrati na izvođenje prekinute dretve.

Instrukcija koja vraća sadržaj programskog brojila sa stoga mora osim toga prebaciti adresiranje u korisnički adresni prostor i aktivirati korisnički register kazaljke stoga. Nazvat ćemo tu instrukciju vratiti se iz prekidnog načina rada¹.

Na kraju, možemo ustanoviti da će procesor u opisanom načinu rada izvoditi dvije dretve:

- jednu dretvu koja se izvodi u korisničkom načinu rada;
- drugu dretvu koja se izvodi u jezgrenom načinu rada procesora.

Ako se na taj način obavlja ulazna operacija unošenje niza znakova, te će se dvije dretve naizmjence izvoditi. Svaka pojava prekida izaziva pohranjivanje konteksta korisničke dretve i vraćanje tog konteksta u procesor na kraju posluživanja prekida. Prema tome, procesor će uz pojavu svakog prekida morati obaviti i dio neproduktivnog kućanskog posla (engl. *housekeeping, overhead work*). Međutim, ne zaboravimo što smo time postigli – na taj način možemo izbjegći dugotrajno radno čekanje.

PRIMJER 3.2.

U primjeru 3.1. ustanovili smo da procesor računalne snage 10 MIPS pri prihvaćanju znakova koji pristižu brzinom od 1000 znakova u sekundi, radeći u režimu radnog čekanja, utroši 99,9% svog vremena na čekanje. Kolika je iskoristenost procesora u prekidnom načinu prenošenja znakova?

U prekidnom načinu rada procesor će za prihvati i pohranjivanje jednog znaka opet trebati desetak instrukcija. Međutim, pohranjivanje konteksta i njegova obnova moglo bi zahtijevati po stotinjak instrukcija. Tih 200 instrukcija po svakom prenesenom znaku jedine su neproduktivne instrukcije (jer nema više petlje čekalice). U jednoj sekundi prihvata se 1000 znakova, te je od 10^7 instrukcija njih 2×10^5 neproduktivnih. To znači da se na kućanske poslove utroši 2% vremena i da se ostatak od 98% vremena može korisno upotrijebiti za obavljanje nekog korisnog posla.

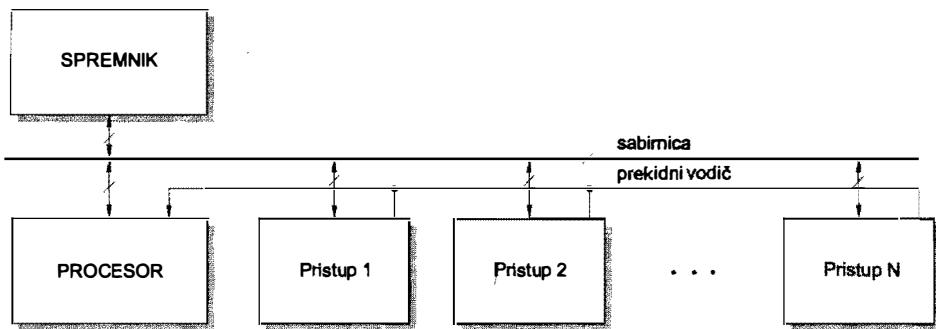


¹ Uz instrukciju *omogućiti prekidanje* (engl. *enable interrupt*) u instrukcijskom skupu nalazi se i instrukcija *onemogućiti prekidanje* (engl. *disable interrupt*). Tom se drugom instrukcijom može odgoditi djelovanje prekidnog signala u nekom od sjeću programa za koji je važno neprekidivo izvođenje. Te dvije instrukcije, zajedno s instrukcijom *povratak iz prekidnog rada* (engl. *return from interrupt*) ostvarene su u različitim arhitekturama na različite načine. Mi ih u našem jednostavnom modelu razmatramo pojednostavljeno, kako bismo lakše objasnili samo načela prekidnog načina rada. Napomenimo da djelovanje instrukcije omogućivanja prekidanja mora biti odgođeno, kako bi se procesor potpuno vratio u stanje prije pojave prekida prije nego li prihvati novi prekidni signal.

3.3. Podsustav za prihvat prekida

3.3.1. Najjednostavniji oblik podsustava za prihvatanje više prekida

Računalo obično ima više ulazno-izlaznih naprava te se u zasnivanju sustava mora računati s više mogućih izvora prekidnih signala. Prepostavimo da je svaka ulazna, odnosno izlazna naprava priključena preko svog pristupnog sklopa koji može generirati prekidni signal. Model procesora opisan u prethodnoj točki ima samo jedan prekidni ulaz. Prepostavljamo da prekidne signale svih pristupa možemo spojiti zajedno i time postići da se prekid pojavi na ulazu procesora kada bilo koji pristup (barem jedan) generira prekidni signal. Takav je računalni sustav predstavljen slikom 3.5.



Slika 3.5. Više prekidnih signala spojenih na jedan prekidni ulaz procesora

Pri pojavi prekidnog signala na ulazu procesora procesor će reagirati na isti način kao i u prethodnom opisu, tj. pri pojavi prekida zabranit će se daljnje prekidanje, procesor će prijeći u sustavski način rada, programsko brojilo stavlja se na stog i u programsko brojilo upisuje se adresa prve instrukcije odsječka programa za posluživanje prekida.

U tom odsječku programa treba ustanoviti koji je od pristupnih sklopova izazvao prekid uzastopnim čitanjem registara stanja pristupnih sklopova. Taj bi potprogram izgledao ovako:



```

(pojavio se prekidni signal, onemogućeno je prekidanje i programsko brojilo se
nalazi na sustavskom stogu)
{
    pohraniti kontekst na sustavski stog;
    pročitati registar stanja pristupa RS_N;
    ako je (ZASTAVICA_N = 1)
        skočiti na obradu prekida_N;
    pročitati registar stanja pristupa RS_N-1;
}
  
```

```

ako je (ZASTAVICA_N-1 = 1)
    skočiti na obradu prekida_N-1;
    :
pročitati registar stanja pristupa RS_1;
ako je (ZASTAVICA_1 = 1)
    skočiti na obradu prekida_1;
obnoviti kontekst sa sustavskog stoga;
omogućiti prekidanje;
vratiti se iz prekidnog načina;
}

```

Ovaj programski odsječak, tzv. ispitni lanac, ispituje redom zastavice u registrima stanja pojedinih pristupnih sklopova. Ako se ustanovi da je neka zastavica postavljena, dogodit će se skok na dio programa za obradu tog prekida. Završetak ovog odsječka možda je malo neobičan. Naime, ako se pri uzastopnom ispitivanju ustanovi da ni jedan od ispitanih $N-1$ pristupa nije postavio svoju zastavicu, onda bi se i bez posebnog ispitivanja zastavice u registru stanja pristupa 1 moglo zaključiti da je taj pristup uzrokovao prekid. Ovdje se, međutim, ispituje i taj pristup, što znači da bi se mogao otkriti i neki lažni prekid koji je nastao, primjerice, nekom smetnjom na sabirnici računala. Ako ispitivanje i te zadnje zastavice pokaže da ni jedan pristup nije postavio zastavicu, u ispitnom se lancu ustanovljuje da je prekid lažan i obnavlja se kontekst prekinute dretve, omogućuje prekidanje i na kraju.

Odsječak programa za obradu prekida pojedinog pristupa mora imati sljedeći oblik:

```

(obrađa prekida pristupa I)
{
    pročitati podatkovni registar PR_I
    (ili upisati podatak u podatkovni registar PR_I);
    poništiti zastavicu u registru stanja RS_I;
    obaviti sve dodatne aktivnosti;
    :
    obnoviti kontekst sa sustavskog stoga;
    omogućiti prekidanje;
    vratiti se iz prekidnog načina;
}

```



Nakon obavljanja svih aktivnosti oko obrade prekida procesor će se vratiti na izvođenje dretve koja je bila prekinuta. Odsječak, naime, završava obnovom konteksta sa sustavskog stoga i vraćanjem iz prekidnog načina rada u korisnički način.

Ovakvo rješenje podsustava za prihvatanje prekida ima niz nedostataka. Najveći od njih je da je tijekom cijele obrade prekida zabranjeno svako drugo prekidanje. To znači da pojava nekog prekida čija obrada zahtijeva trenutačno reagiranje ne može biti prihvaćena, ako traje obrada nekog drugog manje važnog prekida koji se pojavio ranije i pritom može biti i dugotrajan.

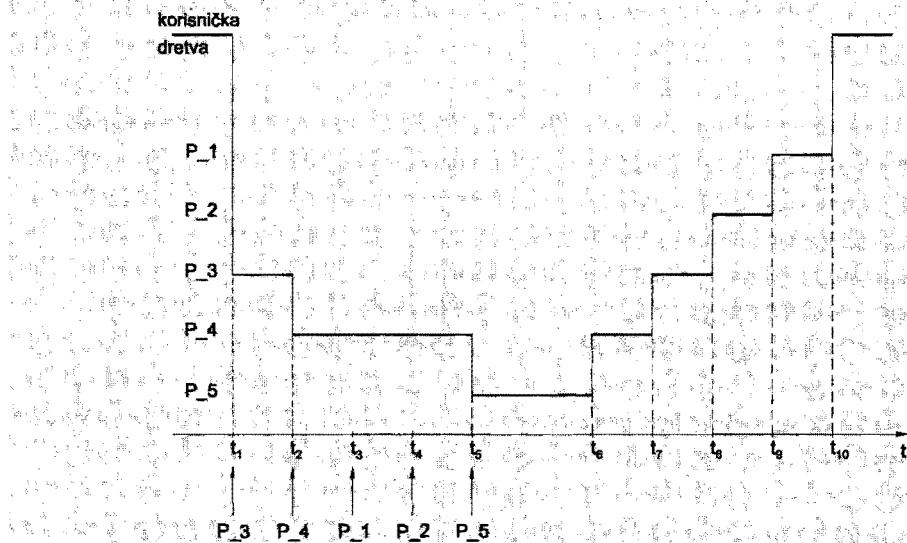
3.3.2. Podsustav za prihvat prekida razvrstanih po prioritetima s najjednostavnijim sklopovljem

Gore spomenuti nedostatak najjednostavnijeg sklopovskog rješenja podsustava za prihvat prekida mogao bi se otkloniti odgovarajućim programskim rješenjem. Prepostaviti ćemo da su pristupni sklopovi razvrstani po *važnosti* ili *prioritetu* (engl. *priority*). Prioritet možemo označiti prirodnim brojevima tako da veći broj ima značenje većeg prioriteta. Prethodno opisani ispitni lanac započinje pregledavanje pristupa s najvećim brojem, pa će njime otkriti najprije zahtjev za prekid višeg prioriteta, ako se istodobno pojavi dva ili više zahtjeva za prekid. Međutim, tim redoslijedom ispitivanja ne postiže se uvažavanje prioriteta ako se pojavi prekid višeg prioriteta nakon što je zahtjev nižeg prioriteta već prihvaćen.

PRIMJER 3.3.



Prepostavimo da računalni sustav ima pet pristupnih sklopova i da su oni razvrstani u pet razina prioriteta. Mi bismo željeli obradu prekida obaviti tako da pojava prekida višeg prioriteta može prekinuti obradu prekida nižeg prioriteta. Uz prepostavku da se u trenucima t_1, t_2, t_3, t_4, t_5 pojave redom prekidi na pristupima P_3, P_4, P_1, P_2, P_5 , željeli bismo obraditi te prekide na način prikazan slikom 3.6.



Slika 3.6. Željeni način ponašanja prekidnog podsustava

Tijekom izvođenja korisničke dretve prihvata se prekid P_3 i nakon što započne njegova obrada prihvata se prekid P_4 , koji će prekinuti obradu prekida P_3 i započetiće njegova obrada. Pojava prekida niže razine P_1 i P_2 ne bi smjerala prekinuti obradu prekida P_4 , ali pojava prekida P_5 mora se prihvati. Sve prekinute i odgođene obrade treba nastaviti, i to u skladu sa zadanim prioritetima.

Uz sklopolje prema slici 3.5., program za prihvat i obradu prekida mogao bi se zasnovati na sljedećim pretpostavkama:

- neka se program sastoji od neprekidivih dijelova koji obavljaju kućanske poslove i prekidivih dijelova u kojima se obavlja sama obrada prekida;
- svi prekidi, bez obzira na prioritet, prihvataju se onda kada se program nalazi u prekidivom dijelu;
- ako je novoprispjeli prekid nižeg prioriteta, on se stavlja u listu čekanja, a ako ima viši prioritet, on se počinje obrađivati;
- kada završi obrada nekog prekida nastavlja se obrada sljedećih (onih koji su bili prekinuti ili onih koji su na listi čekanja);
- ako ni jedan prekid više ne čeka na obradu, nastavlja se izvođenje korisničkog programa.

Program koji bi mogao obavljati opisani posao mora imati prikladnu strukturu podataka u koju se mogu pohraniti svi važni podaci uključujući i kontekst dretvi koje se izvode. Uz pretpostavku da imamo N pristupa u sustavu mora postojati mogućnost pohranjivanja konteksta za ukupno $N+1$ dretvi: njih N za obradu svakog prekida i jednu koja izvodi korisnički program. Korisnička dretva ima najniži prioritet i zbog jednostavnosti razmatranja možemo taj prioritet označiti brojem 0.

U sustavskom (jezgrinu) dijelu spremničkog prostora uz sustavski stog, smjestit ćemo i potrebnu strukturu podataka. Ona se sastoji od:

- varijable T_P u koju stavljamo broj tekućeg prioriteta dretve koja se upravo izvodi;
- poretka zapisa $KON[N]$ u koji se može pohranjivati cijeli kontekst pojedine dretve i dodatno još i varijabla T_P ;
- poretka $K_Z[N]$ u koji se pohranjuju kopije zastavica iz pristupnih sklopova (ako je $K_Z[I] = 0$, pristup I ne čeka na obradu prekida, a ako je $K_Z[I] = 1$, pristup I čeka na obradu prekida).

Pretpostavimo da je uspostavljeno radno stanje u kojem procesor izvodi korisničku dretvu i varijabla tekućeg prioriteta T_P sadrži vrijednost 0. Obradu prekida na željeni način moglo bi se obaviti na sljedeći način:

(pojavio se prekidni signal, onemogućeno je prekidanje i programsko brojilo se nalazi na sustavskom stogu)

```
{
    pohraniti kontekst na sustavski stog;
    ispitnim lancem ustanoviti uzrok prekida, tj. indeks pristupa I;
    ako je ( $1 \leq I \leq N$ , tj. prekid nije lažan){
        postaviti oznaku čekanja  $K\_Z[I] = 1$ ;
        poništiti zastavicu u registru stanja pristupa I;
        dok je ((postoji  $K\_Z[J] \neq 0$ )  $\wedge$  ( $J > T\_P$ )){
            odabrati najveći J;
             $K\_Z[J] = 0$ ;
            pohraniti kontekst sa sustavskog stoga i  $T\_P$  u zapis  $KON[J]$ ;
        }
    }
}
```



```

        T_P = J;
        omogućiti prekidanje;
        pozvati potprogram za obradu prekida J;
        onemogućiti prekidanje;
        vratiti na sustavski stog i u varijablu T_P sadržaj iz KON[J];
    }
}

obnoviti kontekst sa sustavskog stoga;
omogućiti prekidanje;
vratiti se u način rada prekinute dretve i obnoviti programsko brojilo sa
sustavskog stoga;
}
}

```

U prvi mah ovaj program izgleda malo nerazumljivo. Zbog toga ćemo ga raščlaniti. Ulazak u program nastaje sklopovskim prekidom.

Pogledajmo najprije kako se program odvija kada se dogodi samo jedan prekid. U trenutku prekida pohranjuje se kontekst dretve korisničkog programa, čiji je prioritet jednak 0, na sustavski stog. Kada se otkrije uzrok prekida, tj. indeks I, postavlja se oznaka $K_Z[I] = 1$ i poništava zastavica u registru stanja pristupa I. Nakon toga se ustanovljuje da postoji uvjet za izvođenje instrukcija petlje dok je, ustanovljuje se da je $J == I$, te se pohranjuje kontekst sa sustavskog stoga i vrijednost T_P u zapis $KON[I]$. Nakon toga se dopušta prekidanje i poziva potprogram za obradu prekida. Tim pozivom zapravo započinje stvarna obrada prekida. Uočimo da se taj potprogram odvija u sličnom režimu kao i korisnička dretva i da se može prekidati. S obzirom na to da smo pretpostavili pojavu samo jednog prekida, ovaj će potprogram nesmetano obaviti svoj posao i vratit će se nakon obavljenja posla na instrukciju iza njegova poziva, a to je instrukcija zabrane prekidanja. Time opet započinje dio kućanskih poslova. Iz zapisu $KON[I]$ vraća se na sustavski stog kontekst osnovnog programa, a u varijablu T_P zapisuje se ponovno vrijednost 0. Kako nema postavljenih dalnjih oznaka čekanja $K_Z[I]$, petlja dok je nema uvjeta za daljnje izvođenje pa je završen i ogrankak instrukcije ako je, te se program nastavlja vraćanjem konteksta u procesor i nastavlja se izvođenje osnovne korisničke dretve.

Međutim, kada se istodobno pojavljuje više prekida ili se pojavi neki novi prekid kada se prethodni obraduje, tada će se taj prekid prihvati kada obrada stigne do prekidivog dijela. Drugim riječima, potprogram za obradu prekida može biti prekinut. Taj će prekid izazvati skok na početak našeg odsječka za prihvat prekida i pohraniti će se kontekst dretve koja upravo jedan prekid obraduje. Uočimo da se ovaj naš programski odsječak ponaša kao rekurentni program – unutar programa poziva se taj isti program. Novi prekid može izazvati dvojaki učinak. Ako mu je prioritet veći od onog koji se upravo obraduje, njegova će dretva biti pokrenuta, a ako mu je prioritet niži od tekućeg, bit će njegova pojava samo zabilježena u oznaci čekanja i nastavit će se izvoditi dretva koja je bila prekinuta.

Instrukcijom vratiti se u način rada prekinute dretve i obnoviti brojilo sa sustavskog stoga vraćamo se ili u prekinutu korisničku dretvu ili u prekinutu dretvu koja je obavljala obradu nekog prekida.

**PRIMJER 3.4.**

Gore opisani postupak možemo najlakše ra umjeti razmatranjem stanja pojedinih varijabli u nekim karakterističnim trenucima odvijanja obrade prekida iz primjera 3.3. Slika 3.7. ilustrira promjene stanja u strukturi podataka tijekom odvijanja cijelog posla.

vrijeme	T_P	stog	KON[1]	KON[2]	KON[3]	KON[4]	KON[5]	K_Z
$t < t_1$	0	-	-	-	-	-	-	00000
$t = t_1$	0	reg[0]	-	-	0,reg[0]	-	-	00100
	3	-	-	-	-	-	-	00000
$t = t_2$	3	reg[3]	-	-	0,reg[0]	-	-	00010
	4	-	-	-	-	3,reg[3]	-	00000
$t = t_3$	4	reg[4]	-	-	0,reg[0]	3,reg[3]	-	10000
	4	-	-	-	0,reg[0]	3,reg[3]	-	10000
$t = t_4$	4	reg[4]	-	-	0,reg[0]	3,reg[3]	-	11000
	4	-	-	-	0,reg[0]	3,reg[3]	-	11000
$t = t_5$	4	reg[4]	-	-	0,reg[0]	3,reg[3]	-	11001
	5	-	-	-	0,reg[0]	3,reg[3]	4,reg[4]	11000
$t = t_6$	4	reg[4]	-	-	0,reg[0]	3,reg[3]	-	11000
	4	-	-	-	0,reg[0]	3,reg[3]	-	11000
$t = t_7$	3	reg[3]	-	-	0,reg[0]	-	-	11000
	3	-	-	-	0,reg[0]	-	-	11000
$t = t_8$	0	reg[0]	-	-	-	-	-	10000
	2	-	-	-	0,reg[0]	-	-	10000
$t = t_9$	0	reg[0]	-	-	-	-	-	10000
	1	-	-	-	0,reg[0]	-	-	00000
$t = t_{10}$	0	reg[0]	-	-	-	-	-	00000
	0	-	-	-	-	-	-	00000

Slika 3.7. Promjene sadržaja u strukturi podataka tijekom obrade prekida

Svakom trenutku vremena u kojem se pojavljuje neki novi prekid (to su trenuci od t_1 do t_5) pridružena su dva stanja:

- prvo od njih određeno je sadržajem varijabli neposredno nakon prekida (varijabla T_P sadrži prioritet dretve koja je upravo prekinuta; na sustavskom stogu se nalazi kontekst dretve koja je prekinuta, što je označeno kao reg[T_P], a u vektoru kopija zastavica K_Z označen je jedinicom novoprdošli prekid);

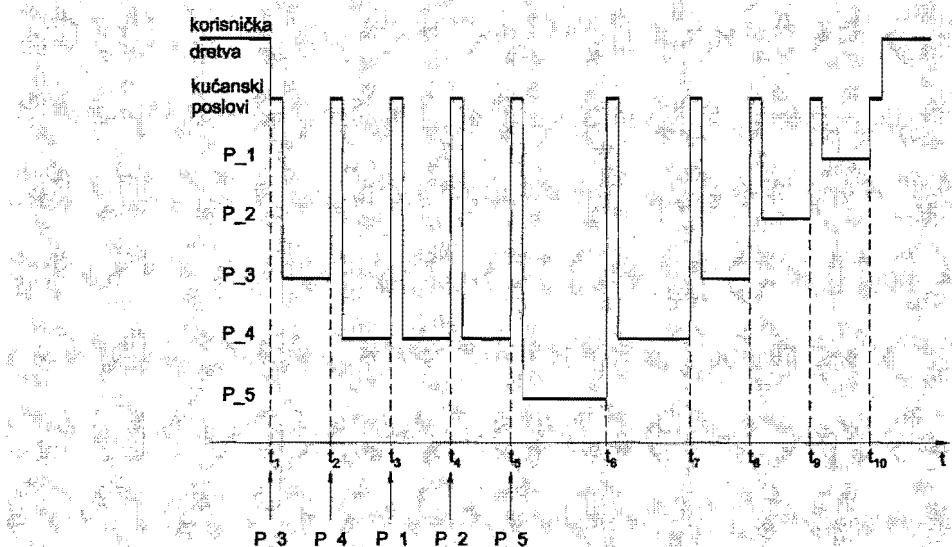
- drugo stanje opisuje sadržaj nakon donošenja odluke o tome koja će od dretvi biti pokrenuta, tj. nakon što procesor ponovno počinje raditi u prekidivom načinu rada (pritom se briše jedinica u vektoru K_Z ako je pokrenuta obrada nekog od prekida koji je čekao na obradu, u varijabli T_P zapisuje novi tekući prioritet ako kreće obrada novoprispjelog prekida, a u zapis $KON[T_P]$ prepisuje se kontekst sa sustavskog stoga, te stari prioritet).

Uočimo da se kontekst koji se nalazio na sustavskom stogu u prvom od ta dva stanja vraća natrag u procesor ako je novopridošli prekid nižeg prioriteta te se prekinuta dretva nastavlja.

Svakom trenutku koji obilježava završetak neke obrade prekida (to su trenuci od t_6 do t_{10}) također su pridružena dva stanja:

- prvo od njih određeno je sadržajem varijabli nakon što se po povratku iz potprograma za obradu prekida J vrati na sustavski stog i u varijablu T_P sadržaj iz zapis $KON[J]$;
- drugo stanje opisuje sadržaj pojedinih varijabli nakon donošenja odluke o tome koja će se dretva nastaviti (pri čemu se kontekst vraćen iz $KON[J]$ vraća u procesor ili natrag u poredak KON u zapis s indeksom one dreve koja je odabrana za pokretanje).

Iz prethodnog je opisa vidljivo da se u ovom rješenju s minimalnom sklopovskom potporom mora obaviti mnogo programskog neproduktivnog kućanskog posla. Taj se kućanski posao obavlja u neprekidivom načinu rada i on se obavlja uz svaki dogadjaj: dolazak prekida ili završetak obrade prekida. Željeno ponašanje sustava koje je prikazano slikom 3.7. u ovom ostvarenju izgleda kao na slici 3.8.

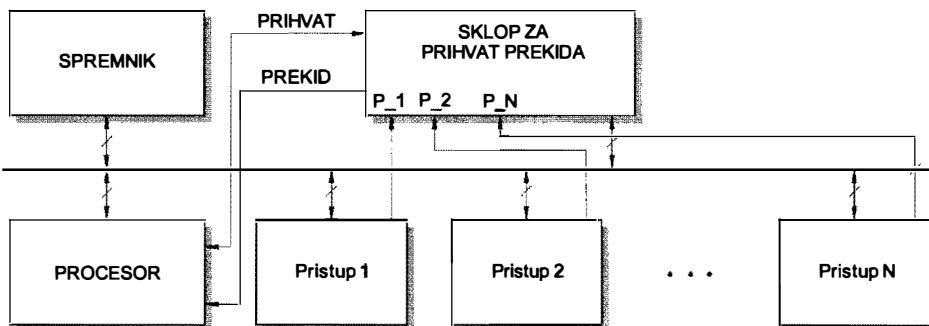


Slika 3.8. Ostvarenje prekldnog rada najjednostavnijim sklopovljem

3.3.3. Sklopovska potpora za ostvarenje višestrukog prekidanja

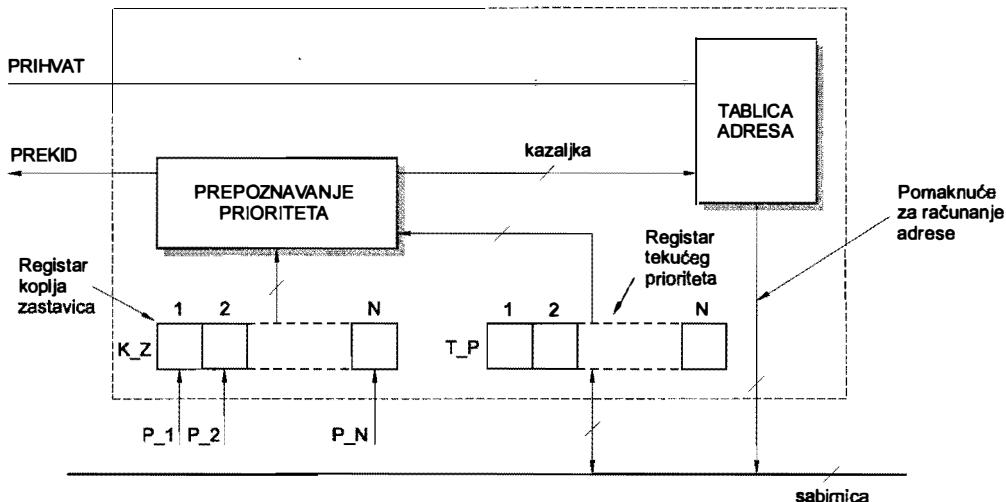
Analizom primjera 3.4. može se zaključiti da se uporabom prikladnog sklopolja može zamjetno poboljšati podsustav za prihvat prekida. U različitim arhitekturama susreću se različita rješenja toga problema. Mi ćemo ovdje opisati samo osnovne zamisli rješenja i pokazati jednostavni model sklopovske potpore za to rješenje.

Osnovna zamisao tog rješenja svodi se na to da se prekidni vodiči od pojedinih pristupnih sklopova ne dovode neposredno na procesor već na ulaze posebnog sklopa za prihvat prekida, kako to prikazuje slika 3.9. Taj bi sklop trebao prema procesoru propustiti samo onaj prekidni signal koji ima veći prioritet od dretve koju procesor upravo izvodi.



Slika 3.9. Sustav sa sklopom za prihvat prekida

Do procesora dolazi, prema tome, samo jedan prekidni signal, i to samo onda kada je to potrebno. Ponašanje procesora pri pojavi prekida također bi trebalo modificirati. U prvom redu, procesor bi trebao u trenutku kada prihvata prekid (tj. nalazi se u prekidivom načinu rada i završava izvođenje neke instrukcije) postaviti signal PRIHVAT koji će djelovati na sklop za prihvat prekida.



Slika 3.10. Detaljniji izgled sklopa za prihvat prekida

Željeno ponašanje sklopa za prihvat prekida može se opisati na temelju njegova detaljnijeg izgleda prikazanog slikom 3.10.

Sklop sadrži dva registra:

- registar kopija zastavica K_Z u čije se bitove zapisuje jedinica kada pripadni pristupni sklop zatraži prekid;
- registar T_P u koji se zapisuje prioritet dretve koju procesor upravo izvodi (tekući prioritet može se označiti jedinicom u odgovarajućem bitu registra kada su svi bitovi jednak nuli, izvodi se korisnička dretva).

Odabrani nazivi tih dvaju registara podsjećaju nas na varijable s istim ulogama u prethodnom rješenju. Sadržaji tih dvaju registara dovode se na sklop za prepoznavanje prioriteta. Taj će sklop propustiti prekid prema procesoru samo onda kada je prioritet zahtjeva veći od onog zabilježenog u registru T_P. Istodobno s propuštanjem prekidnog signala prema procesoru mora se obrisati bit u registru K_Z te generirati pripadnu vrijednost kazaljke usmjerenu na tablicu adresa. Ta tablica sadrži adrese (ili pomaknuća s pomoću kojih se može jednostavno odrediti adresa) na kojima započinje odsječak programa za obradu prekida koji je propušten prema procesoru. Taj se sadržaj iz tablice pod utjecajem signala PRIHVAT prenosi na sabirnicu odakle ga procesor može prihvati i neposredno oblikovati adresu na koju treba program izazvati skok. Na taj se način izbjegava postupno ispitivanje uzroka prekida.

U jednom rješenju koje bi bilo što je više moguće usmjereno na sklopovlje moglo bi se zamisliti da procesor pri pojavi prekida na svom ulazu na kraju izvođenja tekuće instrukcije obavi sljedeće aktivnosti:



```

    ako je (prekidni signal postavljen) {
        zabraniti daljnje prekidanje;
        prebaciti adresiranje u sustavski adresni prostor i aktivirati sustavsku
        kazaljku stoga;
        pohraniti programsko brojilo i sve ostale registre na sustavski stog;
        postaviti signal PRIHVAT,
        sa sabirnice preuzeti sadržaj i iz njega odrediti adresu odsječka za obradu
        prekida;
        staviti tu adresu u programsko brojilo;
    }

```

Ovakav bi procesor automatski pohranjivao na stog ne samo programsko brojilo nego cijeli kontekst prekinute dretve (to i ne mora biti uvijek najpraktičnije, ali ovdje nam služi samo kao ilustracija maksimalno mogućeg sklopovskog rješenja).

Za ovakav bi se procesor moglo zamisliti i posebnu instrukciju vratiti se u prekinuti dretvu koja bi morala obaviti sljedeće:



```

    obnoviti kontekst sa sustavskog stoga;
    omogućiti prekidanje;
    vratiti se u način rada prekinute dretve i obnoviti programsko brojilo sa sustavskog
    stoga;

```

Uz takav bi se procesor podsustav za obradu prekida sastojao od posebnih odsječaka za obradu pojedinih prekida. Početne adrese tih odsječaka morale bi biti pohranjene u tablicu adresa sklopa za prihvat prekida. Obrada prekida s indeksom I mogla bi izgledati ovako:

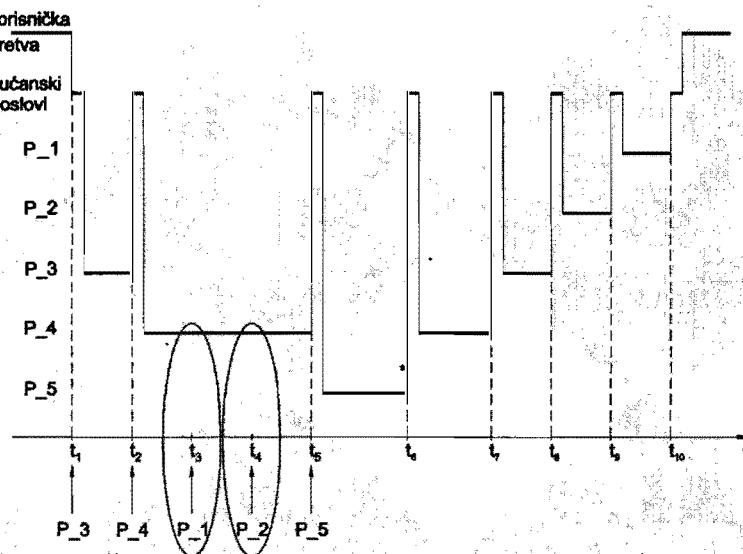
dohvatiti registar T_P i staviti ga na sustavski stog;
 zapisati u $T_P[I] = 1$;
 omogućiti prekidanje;
 obaviti obradu prekida I;
 onemogućiti prekidanje;
 obnoviti sa sustavskog stoga registar T_P ;
 vratiti se u prekinutu dretvu;



Pritom čak ne bi trebalo ni koristiti nikakvu dodatnu strukturu podataka jer se sve što je potrebno pohranjuje na sustavskom stogu.

PRIMJER 3.5.

Zamišljeni način prihvata i obrade prekida prema slici 3.7. ovaj će sustav ostvariti na način ilustriran slikom 3.11.



Slika 3.11. Ostvarenje prekidnog rada s pomoću sklopa za prihvat prekida

U usporedbi sa slikom 3.8. primjećujemo da u trenucima t_3 i t_4 , kada nailaze prekidi P_1 i P_2 , procesor uopće ne biva prekidan, a u ostalim trenucima trajanje neprekidivih dijelova mnogo je kraće, što je slikom teško jasno izraziti.

Pogledajmo što se događa sa sadržajem sustavskog stoga i registrom maski tijekom odvijanja prihvata i obrade prekida. Sadržaj na sustavskom stogu možemo pratiti prema slici 3.12.

vrijeme	K_Z	T_P	sustavski stog
$t < t_1$	00000	00000	
$t = t_1$	00100	00000	
	00000	00100	00000,reg[0]
$t = t_2$	00010	00100	00000,reg[0]
	00000	00010	00100,reg[3]; 00000,reg[0]
$t = t_3$	10000	00010	00100,reg[3]; 00000,reg[0] NEMA
	10000	00010	00100,reg[3]; 00000,reg[0] PREKIDA!
$t = t_4$	11000	00010	00100,reg[3]; 00000,reg[0] NEMA
	11000	00010	00100,reg[3]; 00000,reg[0] PREKIDA!
$t = t_5$	11001	00010	00100,reg[3]; 00000,reg[0]
	11000	00001	00010,reg[4]; 00100,reg[3]; 00000,reg[0]
$t = t_6$	11000	00010	00100,reg[3]; 00000,reg[0]
	11000	00010	00100,reg[3]; 00000,reg[0]
$t = t_7$	11000	00100	00000,reg[0]
	11000	00100	00000,reg[0]
$t = t_8$	11000	00000	
	10000	01000	00000,reg[0]
$t = t_9$	10000	00000	
	00000	10000	00000,reg[0]
$t = t_{10}$	00000	00000	

Slika 3.12. Promjena sadržaja u registrima sklopa za prihvat prekida i na sustavskom stogu tijekom prihvata i obrade prekida

Iz slike 3.12. vidljivo je da se u trenucima t_3 i t_4 , kada nailaze prekidi P_1 i P_2, događa promjena samo u registru kopija zastavica K_Z. Prekidi uopće ne prodire do procesora i on uopće ne "primjećuje" da se nešto događa. Registr tekućeg prioriteta T_P i sadržaj stoga uopće se ne mijenja.

3.3.4. Prekidi generirani unutar procesora, poziv sustavskih potprograma

Jednostavni sklop za prihvat prekida prema slici 3.10. može nam poslužiti pri objašnjavanju mehanizma prihvata tzv. unutarnjih prekida procesora. U raznim dijelovima procesora mogu se pri izvođenju dretvi pojaviti neka nenormalna stanja (pokušaj dijeljenja s nulom, adresiranje nepostojeće lokacije u adresnom prostoru, dekodiranje nepostojećeg operacijskog koda i sl.). U takvim slučajevima treba izvođenje dretve prekinuti i po mogućnosti dojaviti uzrok njezina prekidanja. Onaj dio procesora u kojem se ustanovi takva nenormalnost može generirati prekidni signal, koji se može dovesti do jednog od bitova registra K_Z. Položaj tog bita u registru određuje i prioritet takvog prekida, a pripadna adresa u vektoru adresa izazvat će skok na odgovarajući odsječak za obradu nastalog prekida. Pritom smo prepostavili da je sklop za prihvat prekida postao sastavni dio procesora. Unutarnji i vanjski prekidi mogu se, prema tome, promatrati na potpuno jednak način.

Instrukcijski skup procesora mora sadržati barem jednu instrukciju koja omogućuje namjerno izazivanje prekida iz programa. Takav se prekid stoga naziva *programskim prekidom* (engl. *software interrupt*). Djelovanje je te instrukcije takvo da procesor prevodi u sustavski način rada. S obzirom na to da to izazivanje prekida namjerno i potpuno predviđeno, kontekst (sadržan u registrima procesora) može imati potpuno određeni sadržaj. Tako se u jedan od registara procesora može pohraniti kôd željene funkcije koju želimo pokrenuti tim programskim prekidom. U odsječku programa čija se početna adresa mora nalaziti u vektoru adresa iz tog se koda može odrediti daljnji skok na neki pododsječak koji će obaviti odabranu funkciju. Nadalje, registri procesora mogu poslužiti za prenošenje parametara iz programa u tu funkciju. Povratak iz obrade takvog prekida može vratiti rezultate obrade traženog prekida. Prema tome, mehanizam programskog prekida može se iskoristiti za svojevrsno pozivanje neke vrste potprograma koji se izvode u sustavskom ili jezgrenom načinu rada procesora. Stoga se za takve instrukcije koristi i naziv *poziv sustava* (engl. *system call*) ili *poziv jezgre*. Preciznije rečeno, te instrukcije pozivaju jednu od funkcija koje želimo obaviti. Mi ćemo, od sada nadalje, takve pozive nazivati pozivima *jezgrinih funkcija*. Kada budemo opisivali programske odsječke, njihovu ćemo deklaraciju započinjati ključnom riječju j-funkcija.

Svaka arhitektura procesora rješava mehanizam prekidanja na svoj način. Sklop za prihvat prekida pritom nije posebni sklop, već čini sastavni dio procesora. Svi prekidi (vanjski i unutarnji) objedinjeni su u jedan podsustav i jasno razvrstani u prioritetne razine. Mi se time nećemo posebno baviti, a zainteresirani se čitatelj upućuje na odgovarajuću literaturu ili dokumentaciju proizvođača. Spomenimo samo da su posebni prekidni ulazi predviđeni za pokretanje procesora nakon njegova uključivanja, kao i prekidni ulaz koji izaziva prekid kada napon napajanja počinje opadati, tako da se omogući što urednije obustavljanje dretve koja se u taj čas izvodi.

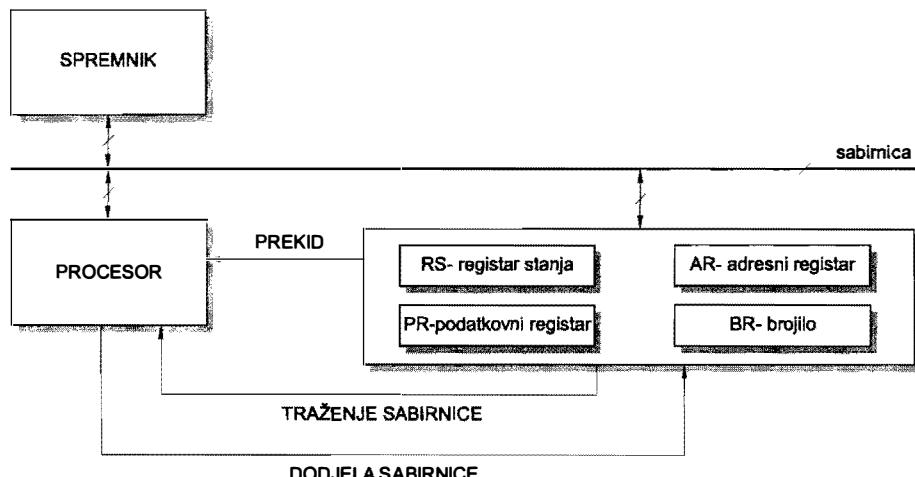
3.4. Prenošenje blokova znakova, skloovi s neposrednim pristupom spremniku

Mnoge su aktivnosti računala povezane s prenošenjem većeg broja znakova – blokova znakova. Na takav se način obavlja: prenošenje sadržaja iz radnog spremnika na disk i obrnuto, prenošenje sadržaja iz radnog spremnika u videospremnik pristupnog sklopa monitora, prenošenje sadržaja prema komunikacijskim pristupnim sklopovima. U prethodnoj smo točki ustanovili da prenošenje pojedinačnih znakova bloka u prekidnom načinu rada izaziva mnogo kućanskog posla. Procesor je prekidan pri prijenosu svakog znaka, a nakon svakog prekida prenosi se samo po jedan znak. Pritom, procesor obavlja vrlo jednostavan posao koji se sastoji od izvođenja samo nekoliko instrukcija.

Taj bi se cijeli posao mogao prepustiti pristupnom sklopu. Jednostavni model pristupnog sklopa, koji sadrži podatkovni registar i registar stanja, trebao bi se nadopuniti s još dva registra:

- adresnim registrom AR u kojem će se nalaziti adresa u spremniku na koju (ili s koje) će se obaviti prijenos pojedinog znaka;
- brojilom BR u kojem se nalazi broj znakova koje još treba prenijeti.

Ta dva dodatna registra trebaju imati svoje adrese tako da ih se može programski dohvati. Skica takvog pristupnog sklopa prikazana je na slici 3.13.



Slika 3.13. Jednostavni model sklopa za neposredni pristup spremniku

Operaciju prenošenje bloka znakova programski se započinje tako da procesor zapiše u adresni registar AR početnu adresu bloka u spremniku i u registar BR broj znakova koji se

želi prenijeti. Nakon toga se upisom odgovarajućeg sadržaja u registar stanja RS (kojeg možemo smatrati i upravljačkim registrom) pokreće prijenos bloka.

Pristupni sklop sam mora moći izvoditi jednostavni program za prenošenje bloka znakova. Pritom on mora dobiti određena prava nad sabirnicom. Naime, inače procesor potpuno upravlja sabirnicom: on postavlja adrese i upravljačke signale na vodiće sabirnice. Pristupni sklop bi se postavljanjem svojih adresa i upravljačkih signala "sukobio" s procesorom. Zbog toga je osmišljen mehanizam prepuštanja sabirnice, po kojem pristupni sklop od procesora traži pristup na sabirnicu. Procesor na signal traženja sabirnice djeluje tako da što je prije moguće odustane od jednog sabirničkog ciklusa (time što izolira svoje izvode od sabirnice) i dojavi da je to načinio signalom dodjele sabirnice prema pristupnom sklopu (engl. *bus request* – zahtjev sabirnice, *bus grant* – dodjela sabirnice).

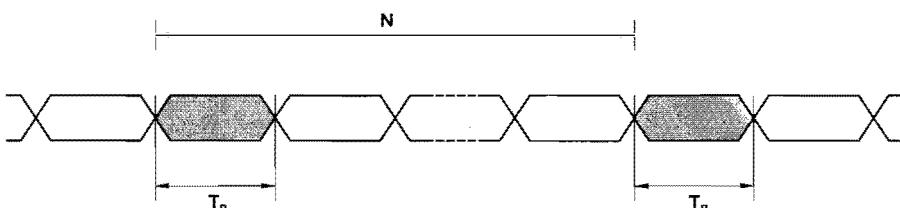
Unutar pristupnog sklopa obavlja se sljedeći mali programski odsječak:

```
dok je (BR > 0) {
    zatražiti sabirnicu;
    čekati na dodjelu sabirnice;
    postaviti na adresni dio sabirnice sadržaj registra AR;
    prenijeti na tu adresu sadržaj podatkovnog PR (ili obrnute);
    AR = AR + 1;
    BR = BR - 1;
}
postaviti signal PREKID;
```



Prekid procesora dogodit će se tek nakon što se prenese cijeli blok. Obrada toga prekida sastoji se od ponovne inicijalizacije registara AR i BR ako se želi nastaviti s prijenosom sljedećeg bloka ili obustavljanjem prijenosa.

Odvijanje prijenosa cijelog bloka između dvaju prekida obavlja se bez sudjelovanja procesora. Otuda i dolazi naziv ovakvog načina prijenosa za koji se koristi i akronim DMA (engl. *Direct Memory Access* – neposredni pristup spremniku). Sa stajališta izvođenja programa, jedina je vidljiva posljedica prijenosa prepuštanje jednog sabirničkog ciklusa pristupnom sklopu po prenesenom znaku (sl. 3.14.).



Slika 3.14. Prepuštanje spremničkog ciklusa pristupu

PRIMJER 3.6.

U primjeru 3.1. ustanovili smo da procesor računalne snage 10 MIPS pri prenošenju znakova koji nailaze brzinom od 1000 znakova u sekundi troši 99.9% svog vremena na neproduktivno čekanje. U primjeru 3.2. pokazali smo da bi taj procesor u danim uvjetima mogao biti korisno aktiviran u 98% vremena.

Ovdje zbog jednostavnosti razmatranja pretpostavimo da sabirnički ciklus traje koliko i jedna instrukcija, tj. da je $T_B = 100$ ns i da je za izvođenje svake instrukcije potreban jedan spremnički ciklus. Vremenski razmak između dva znaka, uz brzinu prijenosa od 1000 znakova u sekundi jednak je 1 ms. To znači da se između dva uzastopna dolaska znakova izvede $N = 10000$ sabirničkih ciklusa od kojih je za prijenos potreban samo jedan, kao što to ilustrira slika 3.14. Prema tome, procesor je za 99.99% svog vremena iskoristiv za druge poslove (pritom smo zanemarili utjecaj obrade prekida na kraju prijenosa bloka).

Pogledajmo to i na drugi način. Ako pretpostavimo da vrijede podaci iz primjera 3.2., tj. da trošimo 200 instrukcija po svakom prenesenom znaku, onda bi se u prekidnom načinu rada s prenošenjem pojedinačnih znakova maksimalno moglo prihvati 5 $\times 10^4$ znakova u sekundi. Sklop s neposrednim pristupom spremniku mogao bi dobiti svaki drugi spremnički ciklus, što znači da bi se moglo prihvati znakove koji nailaze brzinom od 5×10^6 znakova u sekundi, a procesor bi još u vijek 50% svog vremena obavljao koristan posao.

Pristupni skloovi s neposrednim pristupom spremniku sa svojim su prekidnim dijelom uključeni u prekidni podsustav računala, a u prijenosu blokova podataka međusobno konkuriraju za dodjelu pojedinačnih sabirničkih ciklusa. Ovdje opet treba spomenuti da se detalji ostvarenja opisanih mehanizama u različitim arhitekturama međusobno mogu pričinjeno razlikovati.

3.5. Čvrsto povezani višeprcesorski sustav

Načela ostvarenja pristupnih skloova s neposrednim pristupom mogu se razmijerno jednostavno nadopuniti time da se automat koji izvodi "program" prenosa znakova nadomesti procesorom i lokalnim spremnikom. Takav bi pristupni sklop mogao u svom lokalnom spremniku:

- pohraniti program kojim bi se pristupni sklop mogao prilagoditi raznovrsnim ulazno-izlaznim napravama;
- osigurati prostor za smještanje cijelih blokova znakova (bajtova), tj. smjestiti međusobno spremnik u koji bi se znakovi na putu od ulaznih napravama ili prema njima mogli pohraniti.

Takav pristupni sklop postaje ulazno-izlazno računalo čiji procesor sa svojim instrukcijskim skupom i vlastitim prekidnim sustavom može na sebe preuzeti sve poslovanje s

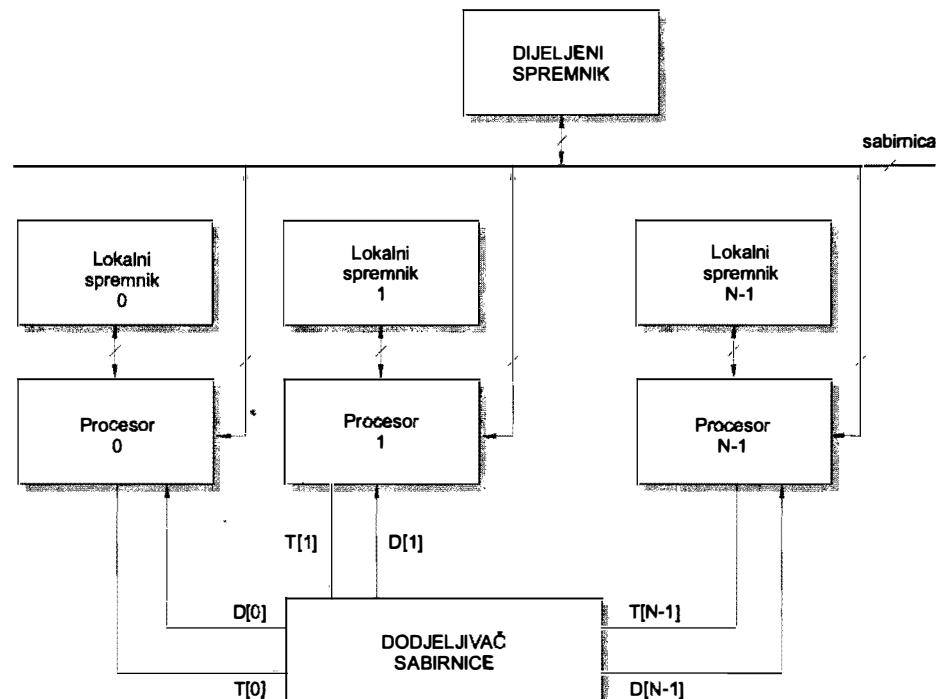
ulazno-izlaznim napravama. Procesor u pristupnom sklopu može, pod utjecajem programa koji se nalaze u njihovu lokalnom spremniku, na sebe preuzeti obradu prekida, prethodno obraditi sve ulazne podatke (ako je riječ o ulaznoj operaciji), pripremiti u svom lokalnom spremniku sređeni blok podataka i zatim ih neposredno prenijeti u radni spremnik glavnog procesora. Jednako tako, pristupni sklop može preuzeti cijeli blok izlaznih podataka iz glavnog radnog spremnika u svoj lokalni spremnik i prenositi ih zatim u neku izlaznu napravu.

Današnji pristupni skloovi koji podržavaju prijenos blokova znakova izvedeni su na tim načelima. Oni su s jedne strane prilagođeni sabirnici računala, preko koje neposredno pristupaju glavnom spremniku, a s druge strane razmjenjuju podatke s ulazno-izlaznim napravama, pa čak i upravljaju radom tih naprava. Pri razmjeni znakova s glavnim spremnikom za prenošenje svakog znaka moraju dobiti jedan spremnički ciklus². Glavni ili centralni procesor mora prepustati pristupnom sklopu pojedinačne cikluse.

Ovakvo gledanje na pristupne sklopove koji uspostavljaju neposrednu vezu s glavnim spremnikom može nam poslužiti i za razumijevanje osnovnih načela ostvarenja višeprocesorskih sustava. Mi se ovdje nećemo upuštati u različite mogućnosti izgradnje višeprocesorskih sustava, već ćemo se ograničiti samo na opisivanje jednog jednostavnog modela koji će nam poslužiti za razjašnjenje osnovnih načela rada. Većina današnjih višeprocesorskih i sličnih sustava (npr. višestruki procesor na čipu) ipak se može prikazati navedenim modelom.

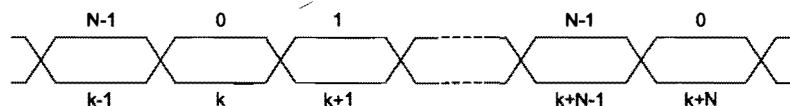
Slika 3.15. prikazuje model takvog sustava. On se sastoji od N procesora od kojih svaki ima svoj lokalni spremnik u koji samo on može pristupati. Svaki procesor, osim toga, može pristupiti do jednog zajedničkog ili *dijeljenog spremnika* (engl. *shared memory*), i to preko zajedničke sabirnice. U jednom sabirničkom ciklusu na sabirnicu, pa prema tome i do spremnika, može pristupiti samo jedan od procesora. Ovdje su svi procesori ravnopravni i stoga se ne može jednomu od njih prepustiti dodjeljivanje sabirnice. U takvom sustavu mora postojati posebni sklopovski *dodjeljivač sabirnice* (engl. *bus arbiter*). Procesor I koji želi pristup do zajedničkog dijeljenog spremnika postavlja signal traženja sabirnice $T[I]$ dodjeljivaču sabirnice. Dodjeljivač sabirnice na početku svakog spremničkog ciklusa odlučuje kojem će od procesora dodijeliti sabirnicu. Ta se odluka prema tome mora donijeti u vremenu koje je mnogo kraće od trajanja jednog spremničkog ciklusa, tj. u vremenu koje se mjeri nanosekundama. Dodjeljivač, prema tome, mora biti vrlo brzi elektronički sklop kojim će se ostvariti pravedna dodjela sabirnice. U jednom sabirničkom ciklusu dodjeljivač će samo jednom procesoru dodijeliti sabirnicu, tj. postaviti mu signal $D[I]$. Ako neki procesor postavi svoj zahtjev za dodjelu sabirnice, on će u svom izvođenju zastati dok mu se sabirница ne dodijeli. Najpravednije je sabirnicu dodjeljivati redom svim procesorima (pod uvjetom da su ju tražili). Ako je u jednom sabirničkom ciklusu sabirnicu dobio procesor I , onda bi sljedeći ciklus trebao dobiti prvi procesor s indeksom J iz niza $(I + 1) \bmod N, (I + 2) \bmod N, \dots, (I + N) \bmod N$ za koji je postavljen zahtjev $T[I]$.

² Preko sabirnica s većim brojem podatkovnih vodiča može se istodobno prenositi i više znakova (primjerice, s 32 bita može se prenijeti 4 bajta), čime se ubrzava prijenos bloka.



Slika 3.15. Sabirnički povezani višeprocesorski sustav

Prema tome, ako svih N procesora intenzivno pristupa do dijeljenog spremnika, onda će im pristup biti dodjeljivan redom, a nakon procesora $N - 1$ pravo pristupa dobit će opet procesor s indeksom 0, kako je to označeno na slici 3.16.



Slika 3.16. Dodjeljivanje sabirničkih ciklusa procesorima kada svi traže pristup

Najbolja iskorištenost ovakvog sustava postigla bi se onda kada bi procesori svaki N -ti ciklus pristupali dijeljenom spremniku, a preostalih $N - 1$ ciklusa svaki u svoj lokalni spremnik. Takovo idealno ponašanje nije praktički ostvarivo, ali mu se možemo približiti ako procesor u prosjeku $1/N$ dio svog vremena ili manje od toga pristupa do dijeljenog spremnika.

Procesori u takvom sustavu mogu biti svi jednaki i tada sustav nazivamo *homogenim* višeprocesorskim sustavom. U homogenim sustavima pojedine dretve mogu se izvoditi na bilo kojem procesoru. Ako sustav nije homogen, tj. ako se sastoji od raznovrsnih procesora, onda su dretve unaprijed pridijeljene pojedinim procesorima i obavljaju posebne specijalizirane poslove.

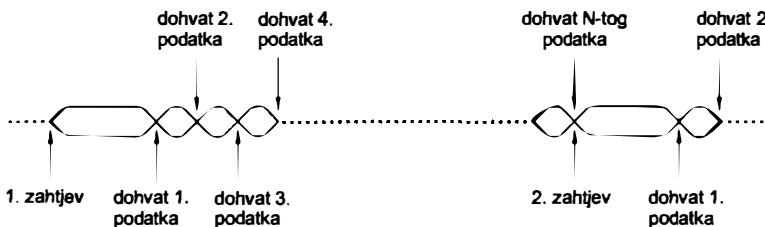
Posebno, treba naglasiti da u jednom sabirničkom ciklusu procesor može ili iz spremnika dohvatiti neki podatak ili u spremnik pohraniti neki podatak. Drugim riječima, procesor u jednom sabirničkom ciklusu može samo pročitati vrijednost neke varijable ili samo pisati vrijednost u neku varijablu smještenu u dijeljeni spremnik. Vidjet ćemo da ta činjenica otežava ostvarenje nekih mehanizama te se takvo ponašanje mora modificirati kao što ćemo vidjeti u sljedećem poglavlju.

Na kraju, ustanovimo da procesori sa svojim lokalnim spremnicima podsjećaju na samostalna računala, koja međusobno komuniciraju preko dijeljenog spremnika. Dijeljeni spremnik, zajednička sabirnica i dodjeljivač sabirnice omogućuju da tih N "računala" međusobno komunicira. Povezanost tako dobivenog sustava omogućuje jednostavnu razmjenu podataka preko dijeljenog spremnika, i to brzinom koja je određena brzinom sabirnice. Pokazat će se kasnije da ta povezanost pojednostavljuje ostvarenje nekih osnovnih mehanizama operacijskih sustava. Ovakvi višeprocesorski sustavi dobili su stoga naziv *čvrsto povezani višeprocesorski sustavi*. Sustavi u kojima nije ostvarena takva veza nazivaju se *raspodjeljenim sustavima* a katkada i *labavo povezanim* sustavima (engl. *tightly coupled* – čvrsto povezani i *loosely coupled* – labavo povezani).

3.6. Sabirnički sustavi stvarnih računala

Iako i stvarna računala načelno podržavaju prikazane načine rada sabirnicom, većinom to ipak rade na nešto drukčiji način. Frekvencija rada procesora obično je mnogostruko veća od brzine ostalih komponenti računalnog sustava. Rijetke komponente, poput spremnika i grafičkog podsustava rade usporedivim frekvencijama, tj. "samo" nekoliko puta nižim. Zbog toga se osim jedne "brze" sabirnice u sustave uvodi i barem jedna sporija sabirnica, koja je s brzom povezana preko prikladnih premosnika (engl. *bridge*). Na sporiju (ili više njih) spojene su "sporije" komponente sustava, kao ulazno-izlazni podsustavi (njihovi međusklopovi).

Međutim, ni takva podjela nije dovoljno učinkovita za iskorištenje mogućnosti procesora koji je i dalje višestruko brži od ostalih najbržih komponenti. Zato se intenzivno koriste priručni spremnici procesora (opširnije opisani u poglavlju 8.5.).



Slika 3.17. Prenošenje podataka sabirnicom u blokovima

Također, sabirnicom se najčešće ne prenose pojedinačni znakovi ili riječi (koje širina sabirnica podržava), već blokovi podataka.

Primjerice, procesor jednim zahtjevom prema spremniku može zahtijevati dohvati ili pohranu bloka podataka. Spremnik, koji je sporiji, može paralelno s postavljanjem prve tražene riječi bloka na sabirnicu pripremati i drugu (i možda i treću, četvrtu). U konačnici u kraćem se vremenskom razdoblju preko sabirnice na ovaj način prenosi više podataka. Slika 3.17. prikazuje jedan takav prijenos podataka.



PITANJA ZA PROVJERU ZNANJA 3

1. Skicirati način spajanja UI naprave na sabirnicu.
2. Što je radno čekanje?
3. Skicirati signale dvožičnog rukovanja.
4. Što se zbiva kada se dogodi prekid?
5. Kako treba nodopuniti ponašanje procesora da on omogućuje prekidni rad bez sklopa za prihvat prekida?
6. Pojasniti instrukcije "pohraniti kontekst" i "vratiti se iz prekidnog načina"?
7. Što treba načiniti na početku svakog potprograma za obradu prekida?
8. Zašto se programsko brojilo tretira zasebno prilikom pohrane konteksta?
9. Što se zbiva kada obrada nekog prekida završi?
10. Koje strukture podataka treba sadržavati operacijski sustav koji omogućuje prihvat prekida različitih prioriteta?
11. Opisati sklop za prihvat prekida.
12. Kako treba nodopuniti ponašanje procesora da on omogućuje prekidni rad sa sklopopom za prihvat prekida?
13. Navesti koje sve radnje mogu generirati prekide unutar procesora.
14. U kojem će se slučaju dogoditi "poziv jezgre", odnosno "ulazak u jezgru" i što se tada poziva?
15. Navesti osnovne registre pristupnog sklopa za neposredni pristup spremniku (DMA).
16. U pseudokodu napisati programski odsječak koji obavlja sklop za neposredni pristup spremniku.
17. Opisati čvrsto povezani višeprocesorski sustav.

4.

Medusobno isključivanje u višedretvenim sistemima

4.1. Programi, procesi i dretve

Računalo obavlja neki korisni *zadatak* (engl. *task*) tako da izvodi programe pripremljene u višem programskom jeziku. Programi određuju sve aktivnosti koje računalni sustav treba obaviti, način unošenja ulaznih podataka, način prikaza rezultata, te način njihova trajnijeg pohranjivanja.

Kada se program preveden u strojni oblik pokrene, on dobiva neke vremenske atribute, kao što su: trenutak početka, trajanje izvođenja, trenutak završetka. Tijekom svog izvođenja program može zastati i biti ponovno pokrenut. Prema tome, program koji se izvodi djeluje tako da na uredni način utječe na promjene stanja računalnog sustava. Na kraju 2. poglavljia ustanovili smo da se zbog toga program naziva računalnim procesom ili kraće samo *procesom*.

S obzirom na to da pojedini procesi u raznim fazama svojeg odvijanja raznoliko troše pojedine dijelove računalnog sustava, pokazalo se razumnim posao organizirati tako da se u istom vremenskom razdoblju izvodi više zadataka. Dakle, više procesa može istodobno napredovati ako se odvijaju u različitim dijelovima računalnog sustava. Primjerice, ako jedan proces čeka na završetak ulazne operacije kako bi mogao nastaviti neko izračunavanje, za to vrijeme drugi proces može izvoditi svoje instrukcije. Pojedine, jasno razlučive dijelove računala možemo nazvati računalnim sredstvima ili samo *sredstvima* (engl. *resources*). Kada se govori o sredstvima, onda to nisu samo pojedini sklopovski dijelovi već i neki programski dijelovi, kao i neki podaci. Procesi se mogu odvijati i tako da se neko sredstvo naizmjence dodjeljuje pojedinom od procesa, pa oni samo prividno istodobno napreduju.

Takav *višezadačni* ili *višeprogramski rad* (engl. *multitasking, multiprogramming*) omogućuje s jedne strane bolje iskorištenje svih sredstava računalnog sustava, a s druge strane olakšava i organizaciju poslova koji se obavljaju računalnim sustavom.

Odvijanje procesa obavlja se izvođenjem njegova niza instrukcija, tj. njegove dretve. U svakom procesu mora, prema tome, biti prepoznatljiva *barem jedna dretva*. Proses se

obavlja tako da procesor izvodi tu dretvu. Pokazalo se, međutim, praktičnim pojedine zadatke dijeliti na podzadatke kako bi se s jedne strane lakše svladala složenost zasnivanja i izgradnje programskih sustava, a s druge strane omogućilo bolje iskorištenje računalnih sredstava sustava, i to *unutar jednog procesa*. Stoga su u suvremene operacijske sustave uvedeni mehanizmi koji podržavaju izvođenje procesa s više dretvi pa govorimo o višedretvenom načinu rada ili *višedretvenosti* (engl. *multithreading*). Dakle, suvremeni su operacijski sustavi višezadačni i višedretveni.

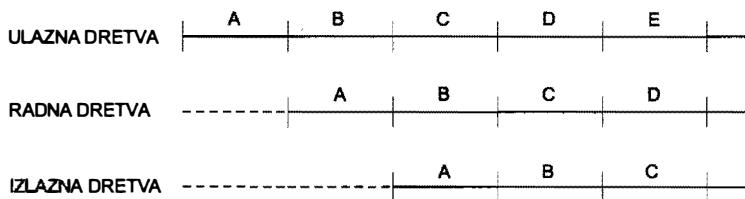
Dretve koje djeluju unutar jednog procesa dijele sva sredstva koja je operacijski sustav stvao na raspolaganje tom procesu. Posebice se to odnosi na adresni prostor procesa. Prema tome, sve dretve istog procesa mogu neposredno pristupiti do svih adresa adresnog prostora svog procesa. Međutim, dva procesa ne mogu jedan drugom neposredno adresirati varijable. Ako je potrebno obaviti razmjenu podataka između dva procesa, onda se to mora činiti posredno s pomoću mehanizama koje osigurava operacijski sustav.

Pogledajmo ponovno završni odjeljak drugoga poglavlja, gdje smo ustanovili osnovna načela ostvarenja višedretvenog rada. Nadalje, u trećem smo poglavlju ustanovili da prebacivanje izvođenja s jedne dretve ne drugu zahtjeva promjenu konteksta u procesoru i upoznali smo osnovnu sklopovsku podlogu za djelotvorno i pouzdano ostvarenje promjene konteksta – prekidni način rada procesora. Iz tih razmatranja već proizlazi kako je važno neke kućanske poslove obaviti tako da se neki podaci dohvaćaju pojedinačno, nesmetano od drugih. U procesorima je to riješeno mehanizmom onemogućivanja prekida koji se automatski događa u trenutku pojave sklopovskog prekida ili izvedbom instrukcije sustavskog poziva (programskim prekidom).

4.2. Višedretveno ostvarenje zadataka, sustav podzadataka

4.2.1. Zadaci i podzadaci

Višedretvena organizacija programskih zadataka smislena je samo onda ako se pri osmišljavanju njihova ostvarenja prepoznaju jasno razlučivi podzadatci. Većina bi se programa, primjerice, mogla podijeliti na tri podzadataka: podzadatak za čitanje ulaznih podataka, podzadatak za potrebna izračunavanja (obradu) i podzadatak za obavljanje izlaznih operacija. Razumljivo je da se ti podzadaci unutar jednog zadatka moraju obavljati navedenim redoslijedom. Međutim, ako se taj programski zadatak ponavlja, onda se pojedini podzadaci mogu obavljati i istodobno. Posao bi mogle obavljati tri dretve. Prva, ulazna dretva dobavlja ulazne podatke, predaje ih drugoj radnoj dretvi i zatim prelazi na ponovno dobavljanje sljedećih ulaznih podataka. Druga, radna dretva preuzima podatke od ulazne dretve, obavi potrebnu obradu, predaje rezultate izlaznoj dretvi i vraća se ponovno na preuzimanje ulaznih podataka. Treća, izlazna dretva preuzima rezultate od radne dretve, obavlja izlaznu operaciju i vraća se na preuzimanje sljedećih rezultata. Slika 4.1. prikazuje moguće ponašanje ovakvoga trodretvenog zadatka u kojem se postiže privid cjevodvodne obrade podataka.



Slika 4.1. Cjevodno ponašanje trodretvenog zadatka

Uzastopni podaci koji su označeni s A, B, C, ... obrađuju se uzastopno pojedinim dretvama. Načelo cjevodnog rada koristi se i u procesorima, gdje se preklapaju pojedine faze izvođenja uzastopnih instrukcija.

Međutim, u procesorima se može namjestiti ponašanje sklopovlja tako da pojedine faze traju jednako i da se izvođenje svake faze zbiva u različitim sklopovskim dijelovima. U opisanom višedretvenom načinu obavljanja nekog zadatka takvo ponašanje najčešće se ne može očekivati.

Naime, prava istodobnost mogla bi se postići kada bismo na raspolažanju imali tri procesora. Tako bi ulaznu operaciju trebao obavljati posebni ulazni procesor, a izlaznu operaciju posebni izlazni procesor. Tomu bismo se mogli djelomice približiti kada bi ulaznu operaciju prepustili jednom, a izlaznu operaciju drugom pristupnom sklopu s neposrednim pristupom spremniku. Centralni procesor koji bi u prvom redu obavljao radnu dretvu morao bi pritom biti djelomice angažiran i za izvođenje ulazne i izlazne dretve.

Nadalje, ne može se očekivati da će pojedine faze višedretvenog ostvarenja opisanog zadataka trajati jednako. Zbog toga je potrebno predviđjeti mehanizme *sinkronizacije* dretvi. Ulagana dretva mora na neki način dojaviti radnoj dretvi da je dobavila ulazne podatke. Radna dretva u tom trenutku može biti zauzeta pa će podaci morati biti privremeno pohranjeni. Ako je radna dretva već obavila prethodnu obradu, onda je ona morala čekati na dolazak novih podataka i njihovim se dolaskom pokreće. Jednako tako potrebno će biti uskladiti rad radne i izlazne dretve.

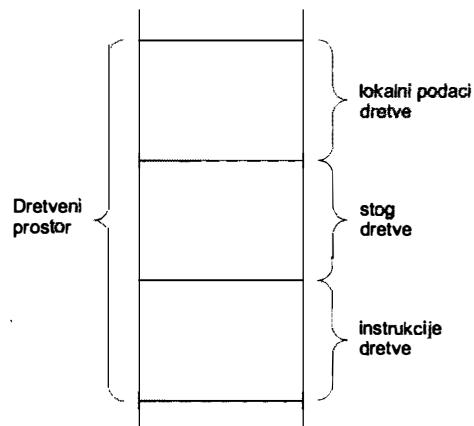
Uz problem sinkronizacije važno je postići i mogućnost da dretve neke svoje aktivnosti obavljaju jedna po jedna. Već spomenuto svojstvo onemogućivanja prekida procesora osigurava da dretva koja se u tom režimu izvodi ne može biti ni od čega prekinuta. Ta će se dretva izvoditi nesmetano tako dugo dok "sama ne odluči" dopuštanje prekidanja. Sličan mehanizam, tzv. *međusobno isključivanje* (engl. *mutual exclusion*) dretvi potrebno je ugraditi u radno okruženje za izvođenje dretvi kako bi se mogla provesti pouzdana organizacija višedretvenih zadataka. Na tu ćemo se temu vratiti nakon što opišemo model višedretvenog zadatka.

4.2.2. Model višedretvenosti

Proces koji raspolaže svim sredstvima potrebnim za svoje izvođenje između ostalog raspolaže svojim adresnim prostorom. Taj adresni prostor stoji na raspolaganju svim dretvama procesa.

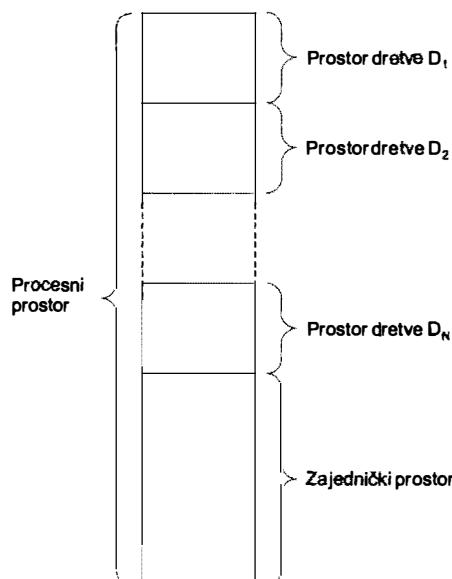
Svaka dretva zauzima svoj dio korisničkog spremničkog prostora. Slika 4.2. prikazuje kako je taj prostor podijeljen na tri dijela:

- dio u koji su smještene instrukcije dretve (strojni program koji određuje način izvođenja dretve);
- dio u kojem je smješten stog dretve;
- dio u kojem su smješteni lokalni podaci dretve.



Slika 4.2. Dretveni dio spremničkog prostora

U procesnom adresnom prostoru postoji još i jedan zajednički dio koji mogu dohvaćati sve dretve. U tom dijelu prostora možemo smjestiti zajedničke (globalne) varijable svih dretvi. Cijeli procesni adresni prostor prikazuje slika 4.3. (Znamo da u računalnom sustavu osim toga mora postajati još i dio sustavskog spremnika u kojem se uvjek može pohraniti kontekst svih dretvi i u kojem su smješteni i svi drugi sustavski podaci i potprogrami.)



Slika 4.3. Podjela procesnog spremničkog prostora

U razmatranju višedretvenosti treba pretpostaviti da je u sustavu osigurana pravilna promjena konteksta dretvi i da se dretve mogu nesmetano odvijati, prekidati i nastavljati. Razmatranje ponašanja dretvi možemo olakšati uvodeći neke pretpostavke za koje ćemo kasnije naći opravdanja, a i pokazati načine njihova oživotvorenja.

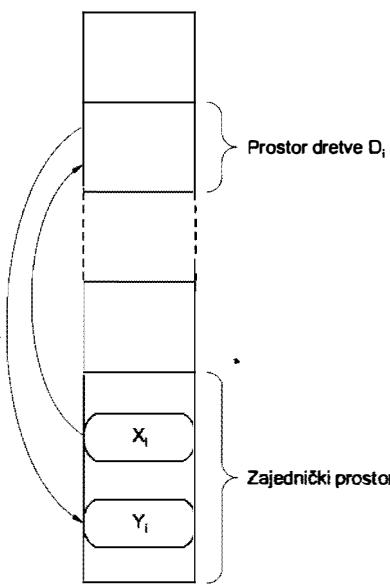
U prvom redu, pretpostaviti ćemo da neka dretva D_i u trenutku kada započinje svoje izvođenje "čita" početne podatke iz jednog podskupa zajedničkog dijela spremnika – iz svoje *domene* X_i . Na kraju svog izvođenja dretva će pisati svoje rezultate opet u jedan podskup zajedničkog dijela spremnika – u svoju *kodomenu* Y_i . Možemo zamisliti da u trenutku kada započinje dretva "prepisuje" ulazne podatke iz X_i u svoje lokalne varijable. Nakon što obavi svu potrebnu obradu koristeći samo svoj dio adresnog prostora dretva u trenutku kada završava svoj posao "piše" rezultate u svoju kodomenu. Po tome je slična potprogramu kojem se prenose vrijednosti parametara.

Izvođenjem dretve obavlja se, prema tome, preslikavanje iz domene X_i u kodomenu Y_i (sl. 4.4.). Možemo reći i da dretva obavlja funkciju:

$$f_i : X_i \rightarrow Y_i$$

ili da je:

$$Y_i = f_i(X_i).$$



Slika 4.4. Dretva čita ulazne podatke iz svoje domene X_i i zapisuje svoje rezultate u svoju kodomenu Y_i

Dva podzadataka ne moraju imati nikakve međusobne veze. Naime, ako dva zadatka nemaju zajedničkih lokacija u svojim domenama i kodomenama, oni jedan drugomu uopće nisu potrebni i mogli bi se izvoditi čak i na različitim računalima. Za takve zadatke kažemo da

su *međusobno nezavisni*. Međusobno nezavisni podzadaci mogu se izvoditi proizvoljnim redoslijedom. Međutim, ako jedan podzadatak čita svoje ulazne podatke iz lokacija u koje neki drugi podzadatak piše svoje rezultate, onda je razumljivo da će konačni rezultat nakon izvođenja cijelog zadatka ovisiti i o redoslijedu izvođenja njima pripadnih dretvi.

Malo podrobnija analiza pokazuje da dva nezavisna podzadataka mogu čak i imati nekih zajedničkih spremničkih lokacija. Naime, ako postoje neke lokacije iz kojih podzadaci samo čitaju svoje ulazne podatke, onda se njihovi sadržaji ni izvođenjem jednog ali ni drugog podzadataka neće mijenjati te će ova podzadataka pronaći u tim lokacijama jednake sadržaje bez obzira na redoslijed izvođenja njihovih dretvi.

Drugim riječima, uvjet nezavisnosti dvaju podzadataka, odnosno njihovih dretvi D_i i D_j , može se izraziti tako da presjeci domene jednoga i kodomene drugoga i presjeci njihovih kodomena budu prazni skupovi, ali se može dopustiti da presjek njihovih domena ne bude prazan. Prema tome, uvjet nezavisnosti može se izraziti kao:

$$(X_i \cap Y_j) \cup (X_j \cap Y_i) \cup (Y_i \cap Y_j) = \emptyset.$$

Složeni programski zadatak može se sastojati od više podzadataka. Pri zasnivanju takva sustava mora se za svaki par zadataka utvrditi zavisnost ili nezavisnost. Ako su zadaci zavisni, onda se mora ustanoviti redoslijed njihova izvođenja. Sustav podzadataka koji će se pri izvođenju pretvoriti u sustav dretvi zajedničkim će djelovanjem obaviti zadani zadatak. Mi kažemo da će se suradnjom dretvi obaviti cjeloviti proces rješavanja zadataka.

4.2.3. Sustav dretvi

Sustav podzadataka zvat ćemo u dalnjem tekstu sustavom dretvi. Sustav dretvi najjednostavnije je predočiti usmjerenim grafom, kao što to prikazuje slika 4.5.

Čvorovi grafa prikazuju pojedine dretve, a strelice u grafu predstavljaju redoslijed izvođenja dretvi. Ako sve dretve u sustavu smatramo elementima skupa dretvi \underline{D} , tj. ako je:

$$\underline{D} = \{D_1, D_2, \dots, D_N\},$$

onda je takav usmjereni graf vizualizacija relacije parcijalnog uređenja na Kartezijevu produktu $\underline{D} \times \underline{D}$. Relacija parcijalnog uređenja, koju možemo izraziti simbolom $<$ i nazvati "treba se dogoditi prije", za pojedine parove skupa utvrđuje redoslijed izvođenja. Tako za graf prema slici 4.5. možemo utvrditi da vrijedi:

$$\begin{aligned} D_1 &< D_2, & D_1 &< D_3, & D_1 &< D_4, \\ D_2 &< D_5, \\ D_3 &< D_5, & D_3 &< D_6, \\ D_4 &< D_6, \\ D_5 &< D_7, \\ D_6 &< D_7. \end{aligned}$$

Relacija parcijalnog uređenja je tranzitivna, tj. vrijedi:

$$(D_i < D_j) \wedge (D_j < D_k) \implies D_i < D_k.$$

Prema tome, u sustavu dretvi prema slici 4.5. utvrđena je relacija "treba se dogoditi prije" za sve sljedeće parove zadataka:

$$\begin{aligned} D_1 &< D_j \quad j \in \{2, 3, 4, 5, 6, 7\}, \\ D_2 &< D_j \quad j \in \{5, 7\}, \\ D_3 &< D_j \quad j \in \{5, 6, 7\}, \\ D_4 &< D_j \quad j \in \{6, 7\}, \\ D_5 &< D_j \quad j \in \{7\}, \\ D_6 &< D_j \quad j \in \{7\}. \end{aligned}$$

D_1 je početna dretva, a D_7 završna dretva. Dretve koje se nalaze na istom putu od početne do završne dretve moraju se izvoditi propisanim redoslijedom, te su dretve međusobno zavisne.

Dretve koje se ne nalaze na istom putu moći će se izvoditi proizvoljnim redoslijedom i moraju biti međusobno nezavisne. U primjeru sustava dretvi sa slike 4.5. međusobno nezavisni moraju biti sljedeći parovi dretvi:

$$\begin{aligned} &(D_2, D_3), (D_2, D_4), (D_2, D_6), \\ &(D_3, D_4), \\ &(D_4, D_5), \\ &(D_5, D_6). \end{aligned}$$

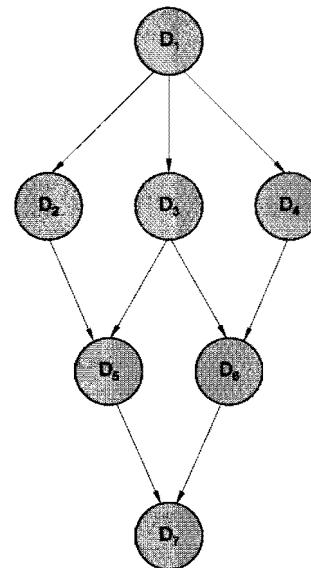
Za sve te parove trebalo bi utvrditi uvjet nezavisnosti, tj. ustanoviti je li ispunjen uvjet:

$$(X_i \cap Y_j) \cup (X_j \cap Y_i) \cup (Y_i \cap Y_j) = \emptyset.$$

Već smo ustanovili da se međusobno nezavisne dretve mogu izvoditi proizvoljnim redoslijedom pa prema tome i istodobno. Tako u primjeru prema slici 4.5., nakon što završi dretva D_1 , dretve D_2 , D_3 i D_4 možemo izvoditi proizvoljnim redoslijedom. Ako se proces izvodi u višeprocesorskom sustavu, onda se te dretve mogu izvoditi i istodobno i na taj način se može ubrzati izvođenje cijelog procesa.

Treba naglasiti da je ovakvo raščlanjivanje zadataka na podzadatke koji su razmješteni na što je moguće više *paralelnih putova* osnovni preduvjet za dobro iskorištenje višeprocesorskih sustava.

Kažemo da se dretve koje se nalaze na paralelnim putovima mogu izvoditi paralelno. Pritom one mogu biti izvođene:



Slika 4.5. Primjena sustava dretvi

- istodobno ako za njihovo izvođenje imamo na raspolaganju više fizičkih procesora ili
- prividno istodobno ako se one izvode na istom procesoru.

Paralelizam u izvođenju nekih složenijih poslova poželjan je i na razini iznad dretvi, tj. na razini procesa. Pritom se može pretpostaviti da će dretve jednoga procesa biti nezavisne s većinom dretvi nekoga drugog procesa te je vrlo vjerojatno da će se uvijek neke od dretvi moći izvoditi.

Međutim, kada je riječ o dretvama istog procesa, dakle dretvama koje surađuju na rješavanju jednog zadatka, može se očekivati mnogo veći broj međusobno zavisnih parova dretvi. Pri zasnivanju nekog programa poželjno je prepoznati što više parallelizma u višedretvenom ostvarenju nekog zadatka. Prvotno osmišljeno parcijalno uređenje sustav dretvi treba pomnivo analizirati sa stajališta zavisnosti zadataka i izostaviti suviše grane u grafu ako se naknadno utvrди da su žbog nezavisnosti parova podzadataka one nepotrebne.

Iz opisanog načina odvijanja posla može se zaključiti da je potrebno izgraditi mehanizam koji će omogućiti *pokretanje, zaustavljanje* dretvi, te načine kojima će jedna dretva obavijestiti drugu da je ona završila svoje izvođenje, a koje smo nazvali *sinkronizacijskim mehanizmima*.

Osim toga, zavisne dretve moraju na neki način i razmjenjivati podatke. S obzirom na to da programi obavljaju vrlo raznovrsne poslove, poželjno je imati na raspolaganju nekoliko načina *razmjene podataka*. Jedan od najjednostavnijih načina postiže se tako da se jedna dretva neposredno piše u domenu druge dretve. Druga mogućnost razmjene podataka ostvaruje se s pomoću *poruka* koje moraju imati dogovoren oblik, a razmjenjuju se u posebnom, rezerviranom dijelu zajedničkog spremnika.

4.2.4. Međusobno isključivanje

Posebno je važno imati na raspolaganju i već spomenuti mehanizam međusobnog isključivanja koji osigurava da se neka sredstva računalnog sustava koriste pojedinačno. Dijelovi dretvi koji koriste neko zajedničko sredstvo nazivaju se *kritičnim odsječcima* (engl. *critical section*). Dretve, koje su inače nezavisne, smiju prolaziti kroz svoj kritični odsječak samo pojedinačno. Kada se dretve izvode u preostalom dijelu, možemo reći u svom nekritičnom odsječku, mogu se izvoditi proizvoljnom brzinom i redoslijedom.

Neprekidivi dijelovi obrade prekida zapravo su kritični odsječci. Dakle, ako se dretve izvode na jednoprocесорском sustavu, onda dretva može osigurati svoj isključivi rad tako da onemogući prekidanje. Nakon onemogućivanja prekidanja nijedan prekid neće moći proći do procesora. Kada dretva obavi svoj kritični odsječak, ona može programski omogućiti prekid i time prelazi u svoj nekritični odsječak.

Međutim, u višeprocesorskom sustavu zabrana prekidanja jednom procesoru ne rješava problem. S obzirom na to da je mehanizam međusobnog isključivanja podloga za ostvarenje mnogih drugih mehanizama njime ćemo se podrobnije pozabaviti. Pri razmatranju

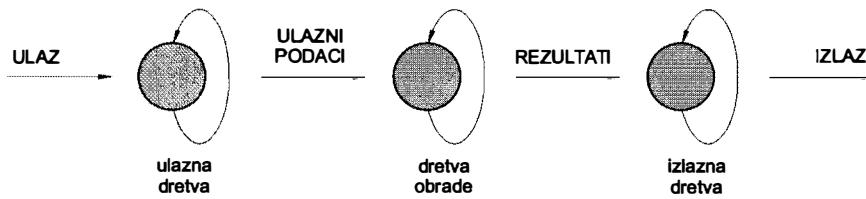
ostvarenja međusobnog isključivanja prepostaviti ćemo da se dretve u svom izvođenju ponavljaju, tj. da se izvode ciklički.

4.2.5. Cikličke dretve

Mnoge se operacije u računalnim sustavima ponavljaju. To se posebice odnosi na funkcije koje obavlja operacijski sustav. Stoga bi se cjevovodno ponašanje računalnog sustava opisano slikom 4.1. moglo postići tako da se:

- ulazna dretva uzastopce ponavlja prihvatajući u svakom prolazu kroz petlju ponavljanja novu skupinu podataka i prenoseći ih dretvi obrade;
- radna dretva prihvata podatke koje joj predaje ulazna petlja, obraduje ih te predaje rezultate izlaznoj dretvi i vraća se na početak, gdje čeka nove podatke;
- izlazna dretva prihvata rezultate radne dretve, prosljeđuje ih u izlaznu napravu i vraća se na svoj početak, gdje čeka nove rezultate.

Sve se te dretve trajno ponavljaju i možemo ih nazvati i *cikličkim dretvama*. U slikovnom prikazu to ćemo ponavljanje istaknuti granom koja uz čvor čini petlju, kao što je to ilustrirano slikom 4.6.



Slika 4.6. Cikličke dretve u ostvarenju cjevovodne obrade

Napomenimo da se i nekoliko dretvi povezanih u podsustav pa i cijeli procesi mogu ponašati ciklički kada obavljaju poslove koji se u računalnom sustavu ponavljaju.

4.3. Ostvarenje međusobnog isključivanja dviju dretvi

Prepostavimo, zbog pojednostavljenja razmatranja, da se višedretveni proces odvija na višeprocesorskom sustavu kakav je prikazan na slici 3.15. Prepostaviti ćemo nadalje da se svaka dretva obavlja u jednom procesoru te da je adresni prostor dretve D_i smješten u lokalni spremnik uz procesor i . Sve dretve mogu pristupiti do dijeljenog spremnika u kojem su smještene zajedničke varijable. Iz opisa rada takvog čvrsto povezanog sustava (vidi odjeljak 3.5.) slijedi da u uzastopnim spremničkim ciklusima do spremnika mogu pristupati različiti procesori i pritom obaviti ili samo jedno čitanje ili samo jedno pisanje.

Prepostavit ćemo, isto tako, da se sve dretve periodično ponavljaju i da se sastoje od dva dijela: jednog kritičnog odsječka i jednog nekritičnog odsječka. Svaka od N dretvi, dakle, neka obavlja sljedeću petlju:



```
dok je (1) {
    kritični odsječak;
    nekritični odsječak;
}
```

Sve su dretve međusobno nezavisne i mogu se odvijati istodobno uz jedini uvjet da se samo jedna od njih smije naći u kritičnom odsječku.

Postavlja se pitanje kako ostvariti postupak međusobnog isključivanja ako dodatno zahtijevamo još i ispunjenje sljedećih uvjeta:

- želimo da mehanizam međusobnog isključivanja djeluje i u uvjetima kada su brzine izvođenja dretvi proizvoljne;
- kada neka od dretvi zastane u svom nekritičnom dijelu, ona ne smije sprječiti ulazak druge dretve u svoj kritični odsječak;
- izbor jedne od dretvi koja smije ući u kritični odsječak treba obaviti u konačnom vremenu.

Zbog pojednostavljenja razmatranja analizirat ćemo sustav sa samo dvije dretve. Naime, ako se pokaže da neki pokušaj nije dobar za sustav s dvije dretve, onda sigurno ne vrijedi za više dretvi, a tek ako se pokaže da je neki pokušaj rješenja uspješan za dvije dretve, onda ga ima smisla razmatrati i za veći broj dretvi.

Prema tome, mi ćemo promatrati dvije dretve D_0 i D_1 i kada se u opisima postupaka pojavljuju indeksi i i j , onda znamo da je $j = 1 - i$.

4.3.1. Prvi pokušaj

Prva pomisao za ostvarenje međusobnog isključivanja svodi se na uvođenje jedne varijable, nazovimo je ZASTAVICA, koja je smještena u zajedničkom procesnom prostoru pristupačnom svim dretvama. Vrijednosti u toj varijabli mogu imati sljedeća značenja:

- ZASTAVICA == 0 označava slobodan ulaz u kritični odsječak;
 ZASTAVICA == 1 označava zabranjen ulaz u kritični odsječak (jer se jedna dretva već nalazi u kritičnom odsječku).

Svaka od dvije dretve (jer trenutačno promatramo samo dvije dretve) mogla bi se u naivnom rješenju napisati ovako:



```
dok je (1) {
    čitati varijablu ZASTAVICA;
    dok je (ZASTAVICA != 0) {
        čitati varijablu ZASTAVICA;
    }
    ZASTAVICA = 1;
    kritični odsječak;
    ZASTAVICA = 0;
    nekritični osječak;
}
```

Pri razmatranju ovoga programskog rješenja a i svih ostalih koje ćemo razmatrati, moramo zamisliti kako izgleda strojni oblik programa koji smo zapisali.

Prepostavit ćemo da procesor ima u instrukcijskom skupu instrukcije opisane u prethodnom poglavlju. Opisani programski odsječak u strojnom obliku izgleda ovako:

```
ADR r1,ZASTAVICA ;
PETLJA LDR r0,[r1]      ; čitati ZASTAVICA
    CMP r0,#1           ;
    BEQ PETLJA          ;
    STR #1,[r1]          ; ZASTAVICA = 1
    prva instrukcija kritičnog odsječka;
    :
    zadnja instrukcija kritičnog odsječka;
    STR #0,[r1]          ; ZASTAVICA = 0
    prva instrukcija nekritičnog odsječka;
    :
    zadnja instrukcija nekritičnog odsječka;
    BAL PETLJA          ;
```



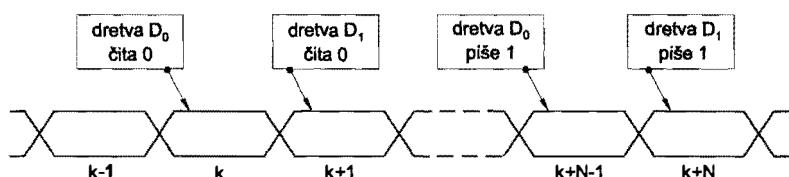
Izvođenje instrukcije "pročitati varijablu ZASTAVICA" moramo zamisliti kao prenošenje vrijednosti iz lokacije čiju smo adresu simbolički označili imenom ZASTAVICA u neki podatkovni registar procesora i što se zbiva tijekom jednog sabirničkog ciklusa. Izvođenje instrukcije pridruživanja vrijednosti varijabli, primjerice "ZASTAVICA = 0", svodi se na pohranjivanje nekog podatkovnog регистра procesora u pripadnu lokaciju spremnika. To se pohranjivanje također zbiva tijekom jednog sabirničkog ciklusa. Imajmo to na umu pri analizi svih daljnjih algoritama.

Vratimo se sada na analizu gornjeg programa. Čitanje varijable ZASTAVICA obavlja se jednom instrukcijom, i to u jednom sabirničkom ciklusu. Ispitivanje vrijednosti te varijable obavlja se u procesoru, praktički opet jednom instrukcijom. Ako je dobavljena vrijednost bila jednaka nuli, druga se dretva ne nalazi u kritičnom odsječku. U tom slučaju dretva će se nastaviti izvoditi tako da će najprije pohraniti (opet jednom instrukcijom i

trošenjem jednog sabirničkog ciklusa) u varijablu ZASTAVICA vrijednost 1 i zatim uči u kritični odsječak. Na kraju kritičnog odsječka dretva će pohraniti u varijablu ZASTAVICA vrijednost 0 i nastaviti izvođenje u nekritičnom dijelu. Nakon završetka kritičnog dijela ova ciklička dretva ponovno se vraća na svoj početak i pokušava ponovno uči u kritični odsječak.

Ako se jedna dretva, primjerice D_i , nalazi u svom kritičnom odsječku, druga dretva D_j koja želi uči u svoj kritični odsječak naći će u varijabli ZASTAVICA vrijednost 1 i počet će izvoditi petlju čekalicu. Ona će se vrtjeti u toj petlji tako dugo dok suparnička dretva D_i ne zapiše u varijablu ZASTAVICA vrijednost 0. Kada se to dogodi, dretva D_j zapisat će u varijablu vrijednost 1 i uči u svoj kritični odsječak. Uočimo da je procesor j u kojem se izvodi dretva D_j zauzeti cijelo vrijeme u režimu radnog čekanja sličnom onom kod programskog ispitivanja zastavice u pristupnom sklopu pri obavljanju neke ulazno-izlazne operacije. Dodatno, taj procesor ne troši samo svoje vrijeme već troši i sabirničke cikluse jer neprestano čita varijablu ZASTAVICA.

Međutim, najgore od svega je to da predloženi pokušaj rješenja uopće ne obavlja korektno međusobno isključivanje. Može se, naime, dogoditi da obje dretve istovremeno žele uči u kritični odsječak i da one obje pročitaju istu vrijednost varijable ZASTAVICA, tj. nađu u njoj nulu. To se može dogoditi ako su dretve na različitim procesorima te u uzastopnim sabirničkim ciklusima pristupaju toj varijabli. Međutim, i u jednoprocесorskim se sustavima to može dogoditi ako se prvu dretvu koja je upravo pročitala varijablu ZASTAVICA prekine prekidom nakon te instrukcije. Pri povratku iz prekida možemo se vratiti u drugu dretvu koja tada može doći do dijela ispitivanja varijable ZASTAVICA te i ona čita nulu. Obje će ustanoviti da je pročitana vrijednost jednaka 0, obje će zapisati varijablu pohraniti vrijednost 1 (ništa ne smeta ako se dva puta u neku varijablu pohranjuje ista vrijednost) i obje će se dretve naći u kritičnim odsječcima. Osnovni je problem u tome da dretva koja prva pročita vrijednost 0 u jednom sabirničkom ciklusu ne stigne pohraniti vrijednost 1 jer već sljedeći sabirnički ciklus dobiva drugi procesor.



Slika 4.7. Mogući slijed događaja u kojem obje dretve D_0 i D_1 mogu istovremeno uči u kritični odsječak

Prema tome, ovaj prvi pokušaj nije dobar i treba potražiti neko drugo rješenje.

4.3.2. Drugi pokušaj

Pokušaj s jednom varijablom **ZASTAVICA** nije bio uspješan. Pokušajmo osmisliti drugo rješenje u kojem bi opet jedna varijabla poslužila za donošenje odluke o izboru dretve koja smije ući u kritični odsječak. Neka se ta varijabla naziva **PRAVO**. Ta varijabla neka poprima vrijednosti indeksa dretvi. Ako je **PRAVO == I**, onda dretva D_i smije ući u kritični odsječak. Dretva D_i mogla bi izgledati ovako:

```
dok je (1) {
    čitati varijablu PRAVO;
    dok je (PRAVO != I) {
        čitati varijablu PRAVO;
    }
    kritični odsječak;
    PRAVO = J;
    nekritisni osječak;
}
```



Jednostavna analiza ovog pokušaja ukazuje da ovdje nije uvažen zahtjev da dretva koja zastane u svom nekritičnom dijelu ne smije spriječiti ulazak druge u svoj kritični odsječak. Naime, ako je u nekom trenutku varijabla **PRAVO** imala vrijednost **I** i dretva D_i je ušla i nakon toga izašla iz svog kritičnog odsječka, ona je pohranila u varijablu **PRAVO** vrijednost **J**. Neka nakon toga ta dretva stane u svom nekritičnom odsječku. Dretva D_j moći će jedanput ući u svoj kritični odsječak i zapisati nakon toga vrijednost **I** u varijablu **PRAVO**. S obzirom na to da dretva D_i stoji u svom nekritičnom odsječku ona neće to svoje **PRAVO** iskoristiti i nikad neće vratiti pravo ulaska dretvi D_j .

Ovaj pokušaj rješenja omogućuje, dakle, da se dretve odvijaju u svojim kritičnim odsječcima samo naizmjence. Jedna od dretvi ne može dva puta uzastopce ući u kritični odsječak.

4.3.3. Treći pokušaj

U prva dva pokušaja upotrebljavali smo samo jednu varijablu koju su obje dretve i čitale i u nju pisale. Time su zapravo dvije dretve koje su inače međusobno nezavisne postale zavisne. Znamo da bi u takvim slučajevima trebalo odrediti redoslijed izvođenja dretvi ako se želi postići određenost u ponašanju takvog sustava.

To navodi na pomisao da se umjesto jedne varijable **ZASTAVICA** uvedu dvije varijable, **ZASTAVICA[0]** i **ZASTAVICA[1]**. Dretva D_i mijenja vrijednost samo svoje zastavice **ZASTAVICA[I]**, a zastavicu **ZASTAVICA[J]** suparničke dretve samo čita. Vrijednosti zastavica imaju sljedeća značenja:

`ZASTAVICA[I] == 0` označava da se dretva D_i nalazi u nekriticnom odsječku;
`ZASTAVICA[I] == 1` označava da se dretva D_i nalazi u kriticnom odsječku.

Prema tome, dretva koja želi ući u kriticni odsječak učinit će to samo onda kada se suparnička dretva nalazi u svom nekriticnom odsječku.

Dretva D_i mogla bi izgledati ovako (kao i do sada prepostavljamo da je jednako tako ostvarena i dretva D_j):



```

dok je (1) {
    čitati varijablu ZASTAVICA[J];
    dok je (ZASTAVICA[J] != 0) {
        čitati varijablu ZASTAVICA[J];
    }
    ZASTAVICA[I] = 1;
    kriticni odsječak;
    ZASTAVICA[I] = 0;
    nekriticni odsječak;
}

```

Ako su početne vrijednosti obje zastavice jednake 0, prva dretva koja pokuša ući naći će zastavicu suparničke dretve jednaku nuli, postavit će svoju zastavicu i ući u kriticni odsječak. Ako sada druga dretva pokuša ući u svoj kriticni odsječak, ona će biti zaustavljena u petlji čekalici tako dugo dok prva dretva ne spusti svoju zastavicu.

Međutim, i ovdje se pojavljuje isti problem kao i kod prvog pokušaja. Naime, obje dretve mogu istovremeno pokušati ući u kriticne odsječke. One pritom mogu obje pročitati u dva uzastopna sabirnička ciklusa vrijednosti 0, svaka iz zastavice suparničke dretve, i na taj način mogu obje ući u kriticne odsječke pa ni ovaj pokušaj ne vodi do ispravnog rješenja.

4.3.4. Četvrti pokušaj

Moglo bi se pomisliti da se problem iz prethodnog pokušaja može zaobići tako da dretva prije ispitivanja suparničke zastavice podigne svoju zastavicu i time naznači da želi ući u kriticni odsječak. Na taj način modificirana dretva D_i mogla bi izgledati ovako:



```

dok je (1) {
    ZASTAVICA[I] = 1;
    čitati varijablu ZASTAVICA[J];
    dok je (ZASTAVICA[J] != 0) {
        čitati varijablu ZASTAVICA[J];
    }
    kriticni odsječak;
    ZASTAVICA[I] = 0;
    nekriticni osječak;
}

```



To, međutim, opet nije korektno rješenje. Naime, ako opet obje dretve zaželete istovremeno ući u kritične odsječke, one mogu u dva uzastopna sabirnička ciklusa zapisati vrijednosti 1 u svoje zastavice. Nakon toga će, dakle, u zastavicama biti pohranjene vrijednosti:

```
ZASTAVICA[I] == 1
ZASTAVICA[J] == 1.
```

I jedna i druga dretva nakon toga ulaze svaka u svoju petlju čekalicu i u njoj ostaju trajno. Nikada više ni jedna dretva neće moći spustiti svoju zastavicu. Ovakvu ćemo pojavu tzv. potpunog zastoja (engl. *dead lock* – mrtvo zaglavljene) susresti i detaljnije opisati kasnije. Prema tome, ovo sasvim sigurno nije uspješan pokušaj.

4.3.5. Peti pokušaj

Nastanak potpunog zastoja moglo bi se možda izbjegići modifikacijom dretvi iz četvrtog pokušaja. Dretva koja je podignula svoju zastavicu i nakon toga utvrdila da je zastavica suprotstavljenje dretve također podignula zastavicu trebala bi svoju zastavicu privremeno spustiti i u tom stanju sačekati spuštanje zastavice druge dretve. Ovako popravljena dretva D_i mogla bi izgledati ovako:

```
dok je (1) {
    ZASTAVICA[I] = 1;
    čitati varijablu ZASTAVICA[J];
    dok je (ZASTAVICA[J] != 0) {
        ZASTAVICA[I] = 0;
        čitati varijablu ZASTAVICA[J];
        dok je (ZASTAVICA[J] != 0) {
            čitati varijablu ZASTAVICA[J];
        }
        ZASTAVICA[I] = 1;
        čitati varijablu ZASTAVICA[J];
    }
    kritični odsječak;
    ZASTAVICA[I] = 0;
    nekritični osječak;
}
```



Analiza ovog načina ostvarenja kritičnog odsječka pokazuje da i ono nije korektno. Naime, dva bi procesora svoje dretve mogli izvoditi istodobno tako da:

- obje dretve u dva uzastopna sabirnička ciklusa podignu svoje zastavice;
- obje dretve pročitaju u dva uzastopna ciklusa vrijednosti zastavice suprotstavljenje dretve;

- obje dretve ustanove da je suprotstavljena zastavica podignuta i u dva uzastopna sabirnička ciklusa spuste svoje zastavice;
- nakon toga obje dretve u dva uzastopna sabirnička ciklusa ustanove da su zastavice spuštene, prođu kroz manju petlju čekalicu i podignu svoje zastavice;
- pri ispitivanju uvjeta veće petlje čekalice obje dretve ustanove da su zastavice podignute i opet ih spuštaju.

Na taj bi način dretve opet trajno ostale u tim petljama, iako bi se zastavice intenzivno podizale i spuštale. Razlika u odnosu na prethodni pokušaj je u tome što se više instrukcija mora dogadati istodobno i vjerojatnost nastanka potpunog zastoja je manja.¹ Kada bi se neki od procesora malo usporio, ovdje bi se moglo dogoditi da jedna od dretvi uspije "pobjeći" iz sinkronizma i uspije ući u kritični odsječak. Stoga bi jedno od mogućih rješenja moglo biti uvođenje nekih dodatnih kašnjenja u jednu od dretvi.

Međutim, pitanje je da li u općem slučaju to uvijek možemo načiniti. Mi smo zbog toga i postavili dodatni zahtjev da brzina pojedinih dretvi može biti proizvoljna.

Svi ovi neuspjeli pokušaji potvrđuju nam da ostvarenje međusobnog isključivanja i nije tako jednostavan problem, čak i za samo dvije dretve. Postavlja se pitanje je li on uopće rješiv uz dane početne pretpostavke.

4.3.6. Šesti pokušaj – Dekkerov postupak

Može se pokazati da se problem u danom radnom okruženju ipak može riješiti, i to čak na više načina. Ovdje ćemo se zadržati na rješenju koje je ponudio nizozemski matematičar T. Dekker, a opisao ga je E. W. Dijkstra. Rješenje se zasniva na kombinacijama zamisli iz drugog i petog pokušaja. Naime, u drugom je pokušaju varijabla PRAVO određivala da se dvije dretve mogu odvijati samo naizmjence. Tamo nam je to smetalo jer je jedna dretva koja je zastala u svom nekritičnom odsječku sprečavala drugu dretvu da dva puta uzastopce ona uđe u svoj kritični odsječak. Međutim, ako mi tu varijablu ispitujemo onda i samo onda kada obje dretve konkuriraju za ulazak u kritični odsječak, tada ona može odrediti redoslijed ulaska i pomoći u rješenju problema međusobnog isključivanja. Takvo rješenje vodi na algoritam za dretvu D_i koji izgleda ovako:

```
dok je (i) {
    ZASTAVICA[i] = 1;
    citati varijablu ZASTAVICA[j];
    dok je (ZASTAVICA[j] != 0) {
        citati varijablu PRAVO;
        ako je (PRAVO != i) {
            ZASTAVICA[i] = 0;
        }
    }
}
```

¹ Ovakav potpuni zastoj kod kojeg se stalno nešto događa, a ipak nema napretka dretvi naziva se u engleskom *live lock* – živo zaglavljenje.

```

dok je (PRAVO != I) {
    čitati varijablu PRAVO;
}
ZASTAVICA[I] = 1;
}
čitati varijablu ZASTAVICA[J];
}
kritični odsječak;
PRAVO = J;
ZASTAVICA[I] = 0;
nekritični osječak;
}

```

Mnemonički oblik strojnog programskog odsječka za dretvu D_0 :

```

ADR r1,ZASTAVICA_0;
ADR r2,ZASTAVICA_1;
ADR r3,PRAVO      ;
→ PETLJA  STR #1,[r1]      ; ZASTAVICA_0 = 1
→ ČEKA_1   LDR r0,[r2]      ; čitati ZASTAVICA_1
              CMP r0,#0          ; ako je ZASTAVICA_1 jednaka 0
              BEQ KR_OD          ; skočiti na kritični odsječak
              LDR r0,[r3]          ; inače čitati PRAVO
              CMP r0,#0          ; ako je PRAVO jednako 0
              BEQ ČEKA_1          ; skočiti na ČEKA_1
              STR #0,[r1]          ; inače ZASTAVICA_0 = 0
→ ČEKA_2   LDR r0,[r3]      ; čitati PRAVO
              CMP r0,#1          ; dok je PRAVO jednako 1
              BEQ ČEKA_2          ; čekati u petlji ČEKA_2
              STR #1,[r1]          ; ZASTAVICA_1 = 1
              BAL ČEKA_1          ;
→ KR_OD   prva instrukcija kritičnog odsječka;
              . . .
zadnja instrukcija kritičnog odsječka;
STR #0,[r1]      ; ZASTAVICA = 0
STR #1,[r3]      ; PRAVO = 1
prva instrukcija nekritičnog odsječka;
              . . .
zadnja instrukcija nekritičnog odsječka;
BAL PETLJA

```



Uočimo da se u ovom pokušaju varijabla PRAVO ispituje samo onda kada obje dretve istovremeno podignu svoje zastavice. Ako neka dretva želi ući u kritični odsječak kada je druga izvan kritičnog odsječka, onda se varijabla PRAVO uopće ne ispituje. Time se izbjegava uočeni nedostatak drugog pokušaja da se dretve mogu izvoditi samo naizmjence. Bez obzira na to, na izlasku iz kritičnog odsječka varijabla PRAVO svaki se put obnavlja tako da se pravo ulaska u kritični odsječak prepusti suprotstavljenoj dretvi, ako ona želi ući u svoj kritični odsječak.

Dekkerovo je rješenje korektno. Ono niti dopušta istodobno ulazjenje dvjema dretvama u kritične odsječke, niti može izazvati potpuni zastoj. U to se možemo uvjeriti tako da razmotrimo sve moguće scenarije ponašanja dviju dretvi, a postoji mogućnost i formalnog dokaza korektnosti rješenja u što se mi ovdje nećemo upuštati.

4.3.7. Petersonov postupak međusobnog isključivanja dviju dretvi

Petersonov je postupak malo pojednostavljenje Dekkerova algoritma. Varijablu PRAVO preimenovat ćemo u NE_PRAVO. Svaka dretva koja želi ući u kritični odsječak postavlja svoju zastavicu i nakon toga zapisuje u varijablu NE_PRAVO svoj indeks. Ako obje dretve istovremeno žele ući u kritični odsječak, onda će varijabla NE_PRAVO poprimiti vrijednost indeksa one dretve čiji je procesor zadnji dobio pravo pristupa na sabirnicu. Vrijednost koja je prije toga bila upisana u tu varijablu izgubljena je. Nakon toga dretva čeka u petlji tako dugo dok suprotstavljena petlja ne spusti svoju zastavicu.

Dretva D_i obavlja se na sljedeći način:



```
dok je (1) {
    ZASTAVICA[I] = 1;
    NE_PRAVO = I;
    čitati varijablu ZASTAVICA[J];
    čitati varijablu NE_PRAVO;
    dok je ((NE_PRAVO == I) && (ZASTAVICA[J] == 1)) {
        čitati varijablu ZASTAVICA[J];
        čitati varijablu NE_PRAVO;
    }
    kritični odsječak;
    ZASTAVICA[I] = 0;
    nekritični osječak;
}
```

Mnemonički oblik strojnog programskog odsječka za dretvu D_0 :



```

        ADR r1,ZASTAVICA_0;
        ADR r2,ZASTAVICA_1;
        ADR r3,NE_PRAVO

    PETLJA STR #1,[r1]      ; ZASTAVICA_0 = 1
                      STR #0,[r3]      ; NE_PRAVO = 0
    ČEKAJ   LDR r0,[r2]      ; čitati ZASTAVICA_1
                      CMP r0,#0      ; ako je ZASTAVICA_1 jednaka 0
                      BEQ KR_OD       ; skočiti na kritični odsječak
                      LDR r0,[r3]      ; inače čitati NE_PRAVO
                      CMP r0,#0      ; ako je NE_PRAVO jednako 0
                      BEQ ČEKAJ       ; dretva 1 ima pravo
    KR_OD    prva instrukcija kritičnog odsječka;
                      zadnja instrukcija kritičnog odsječka;
                      STR #0,[r1]      ; ZASTAVICA_0 = 0
                      prva instrukcija nekritičnog odsječka;
                      zadnja instrukcija nekritičnog odsječka;
BAL PETLJA          ;

```

Imajmo na umu da razmatramo ponašanje samo dviju dretvi s indeksima 0 i 1 i da je $J = I - 1$. Početne vrijednosti svih varijabli neka sve budu jednake 0.

Ako samo jedna od dretvi, recimo dretva 0, želi ući u kritični odsječak, ona će postaviti $NE_PRAVO = 0$ i $ZASTAVICA[0] = 1$. Nakon toga će ustanoviti da uvjet za ostanak u petlji nije ispunjen, jer je $ZASTAVICA[1]$ jednaka 0, i ući će odmah u kritični odsječak.

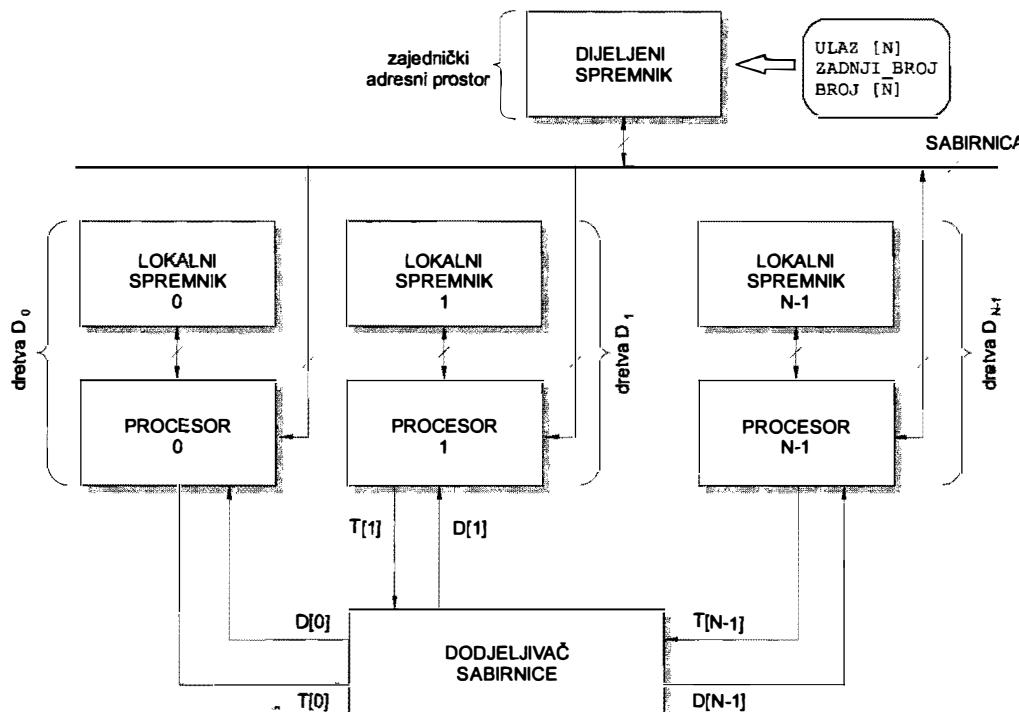
Ako obje dretve istovremeno postave svoje zastavice, onda će u petlji čekalici ostati ona dretva koja je zadnja uspjela upisati svoj indeks u varijablu NE_PRAVO (čijem je procesoru dodjeljivač sabirnice drugom redu dodijelio sabirnicu za pisanje u varijablu $PRAVO$). Suprotstavljenja petlja naći će da njezin uvjet za ponavljanje petlje nije ispunjen i ući će u svoj kritični odsječak. Tek kada ta suprotstavljenja dretva spusti svoju zastavicu, uvjet za čekanje prestaje biti ispunjen i dretva ulazi u svoj kritični odsječak.

4.4. Međusobno isključivanje većeg broja dretvi – Lamportov protokol

Postupci po Dekkeru i Petersonu ukazuju da bi imalo smisla potražiti rješenje koje bi vrijedilo i za više dretvi. U svakom slučaju vidljivo je da se uvjet za ulazak u kritični odsječak mora sastojati od ispitivanja dviju varijabli.

Prvo takvo rješenje predložio je L. Lamport. On je to rješenje prozvao pekarskim algoritmom². Naime, zamisao postupka može se objasniti na primjeru prodavaonice u kojoj jedan prodavač poslužuju kupce (Lamport je u svom opisu odabrao da to bude prodavaonica kruha koja se nalazi uz pekarnicu). Kako se kupci ne bi gurali oko pulta iza kojeg stoji prodavač na ulazu u prodavaonicu dijele se brojevi. Svi kupci vide broj kupca koji se upravo poslužuje pa pultu bez dodatnog guranja pristupa kupac sa sljedećim brojem.

Neka se za ulazak u kritični odsječak natječe N dretvi s indeksima $0, 1, 2, \dots, N-1$. U strukturi podataka svaka će dretva dobiti svoju varijablu u koju će se zapisati broj dobiven "pri ulasku u trgovinu". Te varijable neka su svrstane u poredak BROJ[N]. Osim toga,



Slika 4.8. Dretve i struktura podataka u višeprocesorskom sustavu

² U engleskom jeziku taj postupak nazivaju *Lamport's bakery algorithm*.

svaka od dretvi dobiva još jednu nadzornu varijablu koja označava da se dretva upravo nalazi u fazi dodjele broja i da stoga treba pričekati s ispitivanjem dodijeljenog broja. Te dodatne nadzorne varijable svih dretvi može se svrstati u poredak $ULAZ[N]$.

Ne zaboravimo u kakvom se radnom okruženju izvode naše dretve:

- svaka dretva ima svoj procesor i svoj lokalni spremnik;
- svaki od procesora može preko zajedničke sabirnice pristupati do zajedničkog dijeljenog spremnika u kojem smještamo zajedničke varijable;
- pristup do zajedničkog spremnika odobrava dodjeljivač sabirnice, i to tako da dodjeljuje po jedan sabirnički ciklus u kojem se može obaviti ili jedno čitanje ili jedno pisanje kako je to prikazano na sl. 4.8.

U zajednički spremnik smješteni su poredci $BROJ[N]$ i $ULAZ[N]$. U globalnom spremniku neka se nalazi jedna varijabla $ZADNJI_BROJ$, koja se na početku inicijalizira s 0.

Dretva D_i kada želi ući u kritični odsječak dolazi do svog broja na sljedeći način:

```
ULAZ[I] = 1;
čitati ZADNJI_BROJ;
BROJ[I] = ZADNJI_BROJ + 1;
ZADNJI_BROJ = BROJ[I];
ULAZ[I] = 0;
```



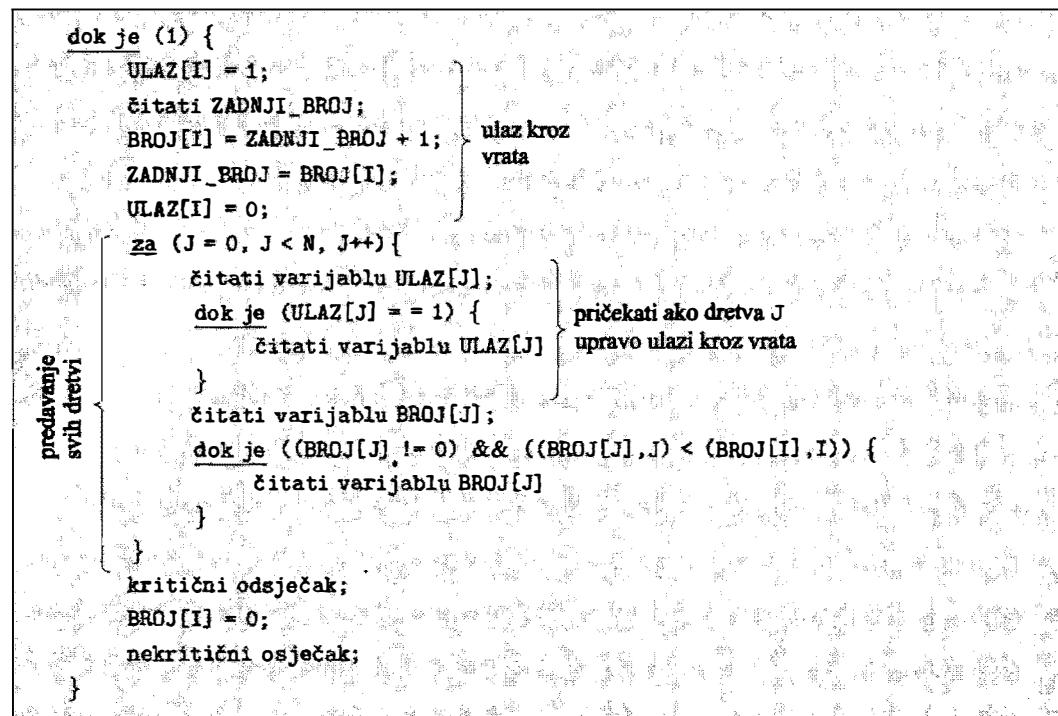
Znamo da u takvom postupku dodjele brojeva mogu nastati problemi. Naime, ako dvije ili više dretvi istovremeno želi ući u kritični odsječak, sve one mogu u uzastopnim sabirničkim ciklusima pročitati istu staru vrijednost zadnjeg dodijeljenog broja prije nego li se taj broj uveća. Ta se pojava u radnom okruženju u kojem se dretve izvode ne može nikako izbjegći.

Moguće rješenje proizlazi iz osnovne zamisli Dekkerova postupka. Tamo se postupalo tako da se od dvije dretve koje su postavile svoje zastavice, i time najavile želju za ulazak u kritični odsječak, odabrala ona na koju je pokazivala varijabla PRAVO. Varijabla PRAVO tamo je mijenjala svoju vrijednost. Ovdje se može dogovoriti da vrijednost indeksa određuje pravo ulaska onda kada dretve dobiju jednakе brojeve. To znači da je početnim dodjeljivanjem indeksa dretvama unaprijed utvrđen redoslijed ulaska u kritični odsječak za slučajeve kada dvije ili više dretvi žele istovremeno ući u kritični odsječak.

Zbog toga ćemo uvesti usporedbu parova prirodnih brojeva (a, b) i (b, c) koja je definirana ovako:

$$(a, b) < (c, d) \text{ je istinito kada je} \\ a < c \text{ ili} \\ (a = c) \wedge (b < d).$$

Na temelju opisanih zamisli može se napisati kako izgleda program dretva D_i :



Nakon što je dretva “ušla kroz vrata” i dodijeljen joj je njezin BROJ [I], ona započinje pregledavanje svih ostalih dretvi.

U petlji koja pregledava redom sve ostale dretve (u gornjem se programu zbog jednostavnijeg izgleda programa pretpostavlja da dretva pregledava i svoje vlastite varijable, ali se pritom na njima ne može zadržati jer odmah napušta petljje čekalice) može zastati na dva mesta:

- dretva D_i zastaje samo na kratko ispitujući varijablu ULAZ [J] ako dretva upravo traži dodjelu broja;
- dretva D_i zastaje dok dretva D_j koja ima BROJ [J] manji od BROJ [I] ne izađe iz kritičnog odsječka i zapiše BROJ [J]=0 (ili su ti brojevi jednaki ali je $J < I$).

Prvi kratki zastanak potreban je da se izbjegne čitanje varijable BROJ [J] u razdoblju kada se on upravo mijenja.

U drugoj petlji čekalici dretva će ostati tako dugo dok dretva čiji je broj bio manji od BROJ [I] ne vrati u varijablu BROJ [J] vrijednost 0. Ako je neka druga dretva D_k koju smo već pregledali ($K < J$) dobila u međuvremenu svoj broj BROJ [K], on je sigurno veći od BROJ [I] i prema tome ne smeta što ga više nećemo gledati.

Isto tako, dok dretva D_i čeka u petlji čekalici da dretva D_j spusti svoj BROJ [J] u nulu, neka druga dretva D_k koju još nismo pregledali ($K > J$) može dobiti svoj BROJ [K]. Taj je broj sigurno veći od BROJ [I] pa kod pregledavanja varijabli te dretve neće doći do zadržavanja dretve D_i .

Dokazivanje korektnosti Lamportova postupka može se i formalno provesti. Mi se u to nećemo upuštati. Zadovoljit ćemo se konstatacijom da nije moguće konstruirati scenarij odvijanja dretvi koji bi izazvao neku neželjenu pojavu.

4.5. Sklopovska potpora međusobnom isključivanju

Pri razmatranju prekidnog načina rada procesora ustanovili smo da odgovarajući sklopovski mehanizmi mogu u velikoj mjeri olakšati ostvarenja nekih mehanizama. Takvo se svojstvo procesora da posebnom instrukcijom onemogući prekidanje upotrebljava za ostvarivanje međusobnog isključivanja. Naime, ako procesor izvodi neku dretvu i ona izvede instrukciju onemogućivanja prekida, onda nije moguće tu dretvu prekinuti. Ona će se izvoditi nesmetano dok sama posebnom instrukcijom ponovno ne omogući prekidanje. Podsetimo se da se u jednoprocесорском sustavu može izvoditi više dretvi uz pretpostavku da smo na neki način osigurali pravilnu zamjenu njihova konteksta. Obrane prekida koje smo razmatrali u prethodnom poglavlju obavljaju se tako da su međusobno isključene. Mehanizmi za izazivanje sklopovskih i programskih prekida stoga su povezani s automatskim onemogućivanjem prekida.

Prema tome, u jednoprocесорском bi sustavu svaka dretva D_i prije ulaska u kritični odsječak morala izvesti instrukciju onemogućivanja prekida, a izlazak iz kritičnog odsječka obavlja se instrukcijom omogućivanja prekida koja bi izgledala ovako:

```
dok je (1) {
    onemogući prekidanje;
    kritični odsječak;
    omogući prekidanje;
    nekritični odsječak;
}
```

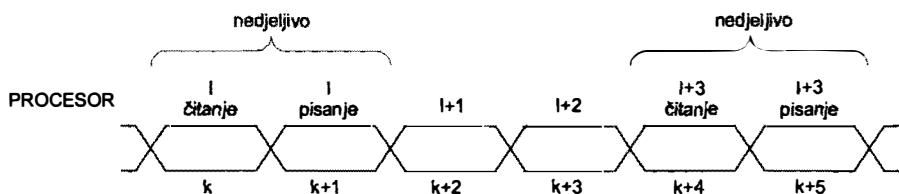


Ona dretva koja prva uspije izvesti instrukciju onemogućivanja prekidanja uči će u kritični odsječak. Tek nakon što ona izade iz kritičnog odsječka pruža se prilika da neki prekid uopće aktivira drugu dretvu, koja će onda moći uči u svoj kritični odsječak.

Međutim, u višeprocесорском sustavu samo onemogućivanje prekidanja ništa ne pomaže. Podsetimo se na naš model višeprocесorskog sustava opisan u odjeljku 3.5. Dretva koja želi uči u svoj kritični odsječak može onemogućiti prekidanje svoga procesora, ali ne može zabraniti prekidanje ostalim procesorima.

Sklopovska potpora ostvarenju međusobnog isključivanja izvedena je na drugi način. U opisanom modelu pretpostavlja se da proces pri izvođenju uobičajenih instrukcija čitanja ili pisanja drži svoj signal traženja $T[I]$ postavljenim do isteka sabirničkog ciklusa koji mu je dodijeljen. Sljedeći sabirnički ciklus može dobiti drugi procesor. Skup instrukcija

procesora treba nadopuniti barem jednom instrukcijom kojoj se dodjeljuju *dva uzastopna nedjeljiva sabirnička ciklusa*. U našem modelu višeprocesorskog sustava možemo zamisliti da procesor drži svoj signal postavljenim tijekom dva uzastopna ciklusa, a dodjeljivač sabirnice neće dodjeliti sabirnicu drugom procesoru dok se taj signal traženja ne poništi.



Slika 4.9. Dodjeljivanje dvaju nedjeljivih sabirničkih ciklusa

U prvom od ta dva ciklusa obavlja se čitanje sadržaja iz adresirane lokacije, a u drugom od ta dva ciklusa pisanje u tu istu lokaciju³. Između te dvije akcije procesor može obaviti samo jednostavne operacije. Tako se u različitim arhitekturama susreću instrukcije koje tijekom izvođenja obavljaju jednu od sljedećih aktivnosti:

- u prvom ciklusu dobave sadržaj adresirane lokacije i smjeste ga u jedan od registara procesora, a u drugom ciklusu pohranjuju u tu lokaciju vrijednost 1 (takve instrukcije imaju naziv “*ispitati i postaviti*”, engleski: “*test and set*”, s akronimom TAS);
- u prvom ciklusu dobave sadržaj adresirane lokacije i smjeste ga u jedan od registara procesora, a u drugom ciklusu pohranjuju u tu lokaciju vrijednost koja je prije toga bila pohranjena u tom registru (takve instrukcije imaju naziv “*zamijeniti*”, engleski: “*swap*”);
- u prvom ciklusu dobave sadržaj adresirane lokacije i smjeste ga u jedan od registara procesora, a u drugom ciklusu pohranjuju na tu lokaciju taj sadržaj uvećan za jedan (engleski: “*fetch-and-add*”).

Mi ćemo u našem jednostavnom modelu, zbog pojednostavljenja razmatranja, pretpostaviti postojanje instrukcije *ispitati i postaviti*. Ona nam, naime, omogućuje ostvarenje međusobnog isključivanja zasnovano na zamisli prvog pokušaja iz odjeljka 4.3.

Varijabla ZASTAVICA smještena je u zajedničkom procesnom prostoru pristupačnom svim dretvama, a vrijednosti u toj varijabli imaju sljedeća značenja:

- ZASTAVICA == 0 označava slobodan ulaz u kritični odsječak;
 ZASTAVICA == 1 označava zabranjen ulaz u kritični odsječak
 (jer se jedna dretva već nalazi u kritičnom odsječku).

Svaka od dretvi (može ih biti proizvoljno mnogo) koja konkurira za ulazak u kritični odsječak nadziran varijablom ZASTAVICA izgleda ovako jednostavno:

³ Ta se dva ciklusa često nazivaju jednim udvojenim ciklусom čitanja i pisanja. Prema tome procesor kada dobije pristup sabirnici može je koristiti trojako: za jedno čitanje (*read cycle*), za jedno pisanje (*write cycle*), za nedjeljivo čitanje i pisanje (*read-modify-write cycle*).

```

dok je (i) {
    "ispitati i postaviti" varijablu ZASTAVICA;
    dok je (ZASTAVICA != 0) {
        "ispitati i postaviti" varijablu ZASTAVICA;
    }
    kritični odsječak;
    ZASTAVICA = 0;
    nekritisni odsječak;
}

```

Ako je varijabla ZASTAVICA početno inicijalizirana s vrijednošću 0, onda će dretva koja prva uspije izvesti instrukciju **ispitati i postaviti** dobaviti vrijednost 0, a istodobno (točnije rečeno: prije nego bilo koja druga dretva uspije pročitati tu istu varijablu) u nju će pohraniti vrijednost 1. Prema tome, prije ulaska u kritični odsječak nije potrebno posebnom instrukcijom zapisivati vrijednost 1. Nakon izlaska iz kritičnog odsječka dretva će u normalnom ciklusu pisanja pohraniti u varijablu ZASTAVICA vrijednost 0.

Sve dretve koje su nakon prve dretve pokušale ući u kritični odsječak ostat će u svojim petljama čekalicama jer će iz varijable ZASTAVICA dobavljati vrijednost 1 (instrukcije "ispitati i postaviti", kada pročitaju vrijednost 1 iz varijable ZASTAVICA, ponovno u nju pišu vrijednost 1, tj. vrijednost varijable time se ne mijenja). Ona dretva koja prva dohvati vrijednost 0 kada se ona pojavi u varijabli ZASTAVICA, izaći će iz petlje čekalice i ući u svoj kritični odsječak. Vrijednost 0 bit će samo kratkotrajno (do prvog izvođenja instrukcije "ispitati i postaviti") pohranjena u varijabli ZASTAVICA.

Prepostavimo da u našem modelu procesora postoji instrukcija:

```
SWP r0,r1,[r2]
```

koja u dva nedjeljiva uzastopna sabirnička ciklusa:

- prenese sadržaj s adrese na koju pokazuje registar r2 u registar r1;
- prenese sadržaj registra r0 na istu adresu.

Mnemonički oblik strojnog programa izgledat će ovako:

```

ADR r2,ZASTAVICA      ;
PETLJA LDR r0,#1        ; u registar r0 zapisati 1
    ← ČEKAJ SWP r0,r1,[r2]   ; r1 = ZASTAVICA, ZASTAVICA = r0
                    CMP r1,#1      ; ako je r1 == 1
                    BEQ ČEKAJ      ; čekati
    prva instrukcija kritičnog odsječka;
    . . .
    zadnja instrukcija kritičnog odsječka;
STR #0,[r2]              ; ZASTAVICA = 0
    prva instrukcija nekritisnog odsječka;
    . . .
    zadnja instrukcija nekritisnog odsječka;
BAL PETLJA               ;

```

Ovakva su rješenja vrlo jednostavna i pouzdana, ali ipak imaju dva nedostatka:

- unaprijed se ne može odrediti redoslijed ulaska dretvi u kritični odsječak;
- dretve koje žele ući u kritični odsječak izvode radno čekanje (time one beskorisno troše i vrijeme svojih procesora i sabirničke cikluse).

Ovaj drugi nedostatak mnogo je ozbiljniji. Dugotrajni kritični odsječak koji izvodi jedan procesor uzrokovat će trošenje vremena svih ostalih procesora čije dretve također žele ući u kritični odsječak. Stoga je razumno posao u računalnom sustavu organizirati na taj način tako da se radno čekanje svede na minimum. Bilo bi razumno one dretve koje čekaju na ulazak u kritični odsječak ukloniti iz procesora i procesor prepustiti nekoj drugoj dretvi. Tek kada dretva bude mogla ući u svoj kritični odsječak treba je ponovno pokrenuti.

Već smo kod opisa obrade prekida ustanovali kako se to može činiti. Na određenim mjestima u spremniku treba osigurati mjesto za pohranjivanje konteksta dretvi i treba osigurati ispravnu i pravovremenu zamjenu konteksta dretvi. Struktura podataka u koju se pohranjuju konteksti činit će strukturu podatka jezgre operacijskog sustava.

Dretva koja želi ući u kritični odsječak neće moći jednostavnom instrukcijom “ispitati i postaviti” ustavoviti smije li ući u kritični odsječak, već će morati pozvati odgovarajuću jezgrinu funkciju koja će dretvi ili dopustiti ulazak u kritični odsječak ili dretvu izbaciti iz procesora (pohraniti njezin kontekst) te pokrenuti neku drugu dretvu.

Isto tako, dretva koja izlazi iz kritičnog odsječka neće neposredno zapisivati nulu u varijablu zastavica, već će pozivati jezgrinu funkciju koja će ili promijeniti vrijednost zastavice ili pokrenuti sljedeću dretvu koja je čekala na ulazak u kritični odsječak.

Na taj će nam način sklopovski detalji povezani s međusobnim isključivanjem na razini programiranja dretvi biti skriveni – međusobno isključivanje obavit će za nas jezgra. Međutim, jezgrine funkcije ne bi se mogle ostvariti bez pouzdanih mehanizama za podržavanje međusobnog isključivanja. U sljedećem poglavlju pozabavit ćemo se mogućim načinima ostvarenja jezgre.



PITANJA ZA PROVJERUZNANJA 4

1. Koje računalne resurse dijele dretve istog procesa?
2. Što je zajedničko dretvama različitih procesa?
3. Kako je podijeljen spremnički prostor procesa, a kako dretveni spremnički podprostor?
4. Navesti uvjet nezavisnosti zadataka.
5. Navesti uvjete koje mora zadovoljavati algoritam međusobnog isključivanja dretvi.
6. Za zadani algoritam međusobnog isključivanja ustanoviti je li ispravan. Obrazložiti odgovor.

Dretva I {

```
dok je (1) {
    dok je (ZASTAVICA[J] != 0),
        ZASTAVICA[I] = 1;
        kritični odsječak;
        ZASTAVICA[I] = 0;
        nekriticni odsječak;
}
```

7. Čemu služi Dekkerov, a čemu Lamportov algoritam? Koje strukture podataka koriste?
8. Navesti Dekkerov, odnosno Lamportov algoritam.
9. Usporediti Petersonov i Dekkerov algoritam.
10. Navesti najjednostavniji način međusobnog isključivanja više dretvi na jednoprocesorskom računalu?
11. Navesti nedjeljive instrukcije procesora koje služe kao sklopovska potpora međusobnom isključivanju.
12. U pseudokodu riješiti problem međusobnog isključivanja više dretvi uz pomoć nedjeljivih instrukcija TAS, SWAP i FATCH_AND_ADD. Koja je prednost tih rješenja u odnosu na Lamportov algoritam međusobnog isključivanja?
13. Koji je najveći zajednički nedostatak algoritmima međusobnog isključivanja (Dekkerov, Petersonov, Lamportov te algoritmima ostvarenima uz pomoć sklopovske potpore)?

5.

Jezgra operacijskog sustava

5.1. Radno okruženje za izvođenje dretvi – jednostavni model jezgre

U prethodnom smo poglavlju ustanovili da se svaki programski zadatak tijekom njegova izvođenja promatra kao računalni proces, čiji se podzadaci tijekom izvođenja nazivaju dretvama.

Prije pokretanja nekog procesa operacijski sustav će osigurati cijelo okruženje za njegovo izvođenje. Vidjet ćemo kasnije kako operacijski sustav procesu dodjeljuje potrebni adresni prostor spremnika i kako s njim povezuje potrebne datoteke i ulazno-izlazne naprave. Za sada ćemo pretpostaviti da je procesu stvoren potrebnii adresni prostor i da je on podijeljen na dretvene potprostore s jednim dijelom zajedničkim za sve dretve, onako kako smo to opisali u odjeljku 4.2.

Podsjetimo se da dretva obavlja neku transformaciju ulaznih podataka koje ona u trenutku početka čita iz svoje domene. Ona na svom završetku piše rezultate u svoju kodomenu. Dretva se može pokrenuti tako da se u programsko brojilo procesora pohrani adresa prve instrukcije dretve. Nakon pokretanja dretva će se obaviti do svoga kraja. Ona može biti tijekom svog izvođenja prekidana, ali to neće smetati njezinu izvođenju ako se osigura pravilna promjena konteksta.

U prethodnom smo poglavlju ustanovili na kakve se poteškoće može naići kada se dopusti neposredno međusobno djelovanje dretvi. Zbog toga je razumno zahtijevati da dretve suraduju ili se natječu za neka sredstva na organizirani način. One će to raditi tako da pozivaju odgovarajuće funkcije jezgre.

Jezgra na neki način objedinjuje sve mehanizme koje smo u prethodnim poglavljima opisali. Moglo bi se reći da se jezgra sastoji od skupine funkcija koje se pozivaju bilo sklopovskim bilo programskim prekidima. Dretve koje žele komunicirati s drugim dretvama pozivaju odgovarajuće jezgrine funkcije koje za njih obavljaju zadani posao. Isto tako, kada neka dretva želi obaviti ulaznu ili izlaznu operaciju, ona to obavlja preko jez-

gre. Kada dretva obavi svoj posao, ona to "javlja" jezgri i jezgra će obustaviti njezino izvođenje.

Kako bismo što razumljivije objasnili funkcije jezgre, razmotrit ćemo jednostavni model jezgre zasnovan na sljedećim pretpostavkama:

- u adresnom prostoru procesa smješteni su svi dretveni prostori;
- čitav adresni prostor procesa dohvatljiv je svim dretvama;
- izvođenje dretvi obavlja se u jednoprocesorskom sustavu;
- dretve odgovarajućim programskim prekidom mogu pozivati jezgrine funkcije na način opisan na kraju 3. poglavlja;
- u sustavu postoje i ulazno-izlazne naprave koje ovdje uzimamo u razmatranje kako bismo jezgom obuhvatili i obradu sklopovskih prekida.

Prepostavit ćemo, nadalje, da u sustavu djeluje sklopovski sat koji se može namjestiti tako da izaziva periodne prekide s periodom T_q . Ti će nam prekidi poslužiti za zaustavljanje izvođenja nekih dretvi koje predugo traju, kao i za generiranje zadanih vremenskih intervala. Razumno je smatrati da je perioda otkucaja sata¹ reda veličine milisekunde ili desetak milisekundi (jedna, deset, sto milisekundi).

Prema tome, u sustavu ćemo imati tri vrste prekida:

- sklopovske prekide od ulazno-izlaznih naprava;
- periodne sklopovske prekide od sata;
- programske prekide koje izazivaju dretve.

Svi ti prekidi poslužit će nam za pozivanje jezgrinih funkcija. Prisjetimo se da prekidom procesor ulazi u sustavski način rada s onemogućenim dalnjim prekidanjem. U sustavskom načinu rada on adresira svoj sustavski adresni prostor i upotrebljava svoj sustavski registar kazaljke stoga.

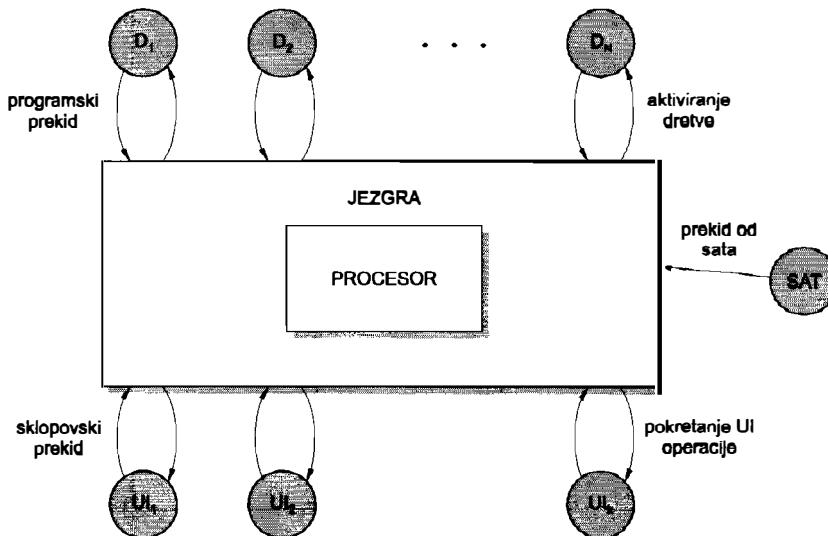
U sustavskom adresnom prostoru nalazit će se sve potrebne strukture podataka i programski odsječci koji će nam ostvariti jezgrine funkcije. Nakon što se obavi potrebna "obrada prekida", tj. izvede prekidom pozvana jezgrina funkcija, bit će pokrenuta neka dretva koju na neki način odabire jezgra. Unutar jezgre se, dakle, moraju donositi odluke koje utječe na daljnje odvijanje dretvi. Te se odluke donose u skladu s nekim pravilima i na temelju odgovarajućih podataka o dretvama koje se čuvaju u strukturi podataka jezgre.

Jezgra bi se mogla predstaviti slikom 5.1. Jezgra se može smatrati prvom hijerarhijski izgrađenom razinom iznad procesora. Ta razina sastoji se od strukture podataka jezgre i jezgrinih funkcija. Dretve D_1, D_2, \dots, D_N simbolizirane su krugovima. Strelice usmjerene od dretvi prema jezgri simboliziraju pozive jezgrinih funkcija ostvarene programskim

¹ Svako računalo opremljeno je sklopovskim satom (napajanj posebnom trajnjicom baterijom) koji nakon sinkronizacije s astronomskim vremenom služi za čitanje vremena. Taj sat sklopovski broji impulse preciznog generatora takta (to može biti i generator takt-a procesora) i u svojim registrima trajno sadrži podatke o vremenu s preciznošću do milisekunde. Iz tog se sata mogu dobivati periodički impulsi koji izazivaju prekide. Mi ćemo te prekide nazvati *otkucajima sata*.



prekidima. Strelice usmjerene od jezgre prema dretvi ukazuju na to da dretve mogu biti pokrenute samo djelovanjem jezgre.



Slika 5.1. Slikovni prikaz jezgre

Ulagano-izlazne naprave (ili kraće: *UI* naprave) simbolizirane su također krugovima označenim s UI_1, UI_2, \dots, UI_K . Svaka se ulagano-izlazna operacija pokreće iz jezgre operacijskog sustava. Naprava će se javiti prekidom kada pokrenuta operacija bude završena. Prekid je simboliziran strelicom usmjerrenom od naprave prema jezgri, a pokretanje *UI* operacije simbolizirano je strelicom usmjerrenom od jezgre prema napravi.

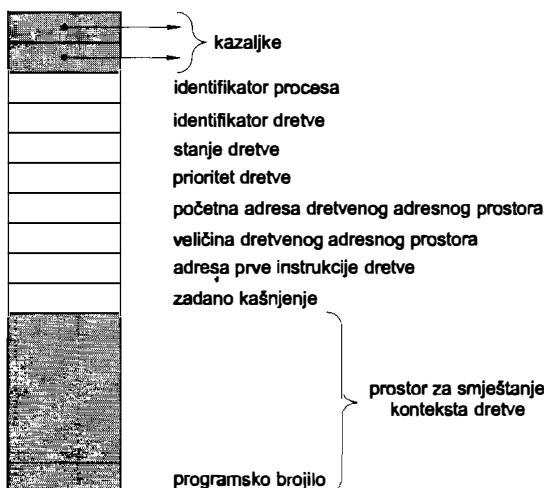
Osim dretvi i *UI* naprave, na slici je prikazan i sat koji prekidom također poziva jezgrinu funkciju. Prema satu nije usmjerena strelica jer on trajno radi i automatski se pokreće prilikom pokretanja sustava.

Poziv jezgrine funkcije – ulazak u jezgru – zbiwa se, dakle, prekidom. Izlazak iz jezgre svodi se na pokretanje jedne od dretvi, pri čemu se procesor vraća u korisnički način rada.

5.2. Struktura podataka jednostavnog modela jezgre – stanja dretvi

Rekli smo da se jezgra sastoji od svoje strukture podataka i funkcija smještenih u sustavski dio spremnika. U strukturi podataka jezgre moraju se pohraniti sve informacije o dretvama potrebne za donošenje odluka o njihovu izvođenju. U ovom ćemo odjeljku opisati strukturu podataka jednostavnog modela jezgre i opisati ulogu pojedinih komponenti te strukture, a u sljedećem ćemo odjeljku razmotriti pojedine jezgrine funkcije.

Za svaku se dretvu moraju u jezgri pohraniti svi važni podaci. Najprikladnije je te podatke smjestiti u jedan zapis koji možemo nazvati *opisnikom* ili *deskriptorom dretve*. Uz podatke važne za opis stanja dretve u opisniku su predviđena i mjesta za smještanje kazaljki koje će nam omogućiti da ga lako premještamo iz jedne dinamičke strukture (liste) u drugu.



Slika 5.2. Sadržaj opisnika dretve

Na slici 5.2. prikazan je sadržaj opisnika u našem modelu jezgre. Uz mesta za kazaljke (koja su u opisniku simbolizirana izlaznom strelicom), u opisniku su predviđena mesta za smještanje sljedećih parametara:

- **Identifikator_procesa** označava kojem procesu pripada dretva (prirodni broj).
- **Identifikator_dretve** omogućuje međusobno razlikovanje dretvi (prirodni broj).
- **Stanje_dretve** je parametar koji označava u kojem se stanju dretva trenutačno nalazi. (Vidjet ćemo da dretva u našem modelu može biti pasivna, aktivna, blokirana i pripravna za izvođenje.)
- **Prioritet_dretve** je parametar koji određuje prednost dretvi pri dodjeljivanju procesora (prioritet ćemo izraziti prirodnim brojem, i to tako da veći broj označava veći prioritet).
- **Početna_adresa_dretvenog_adresnog_prostora** i **Veličina_prostora** opisuju smještanje dretvenog adresnog prostora unutar procesnog adresnog prostora.
- **Adresa_prve_instrukcije** sadrži adresu na kojoj je smještena adresa prve instrukcije dretve.
- **Zadano_kašnjenje** je parametar koji će odrediti odgađanje izvođenja dretve za zadani broj perioda otkucaja sata.
- **Prostor_za_smještanje_konteksta** poslužit će nam za smještanje konteksta dretve onda kada dretva bude prekinuta u svom izvođenju.

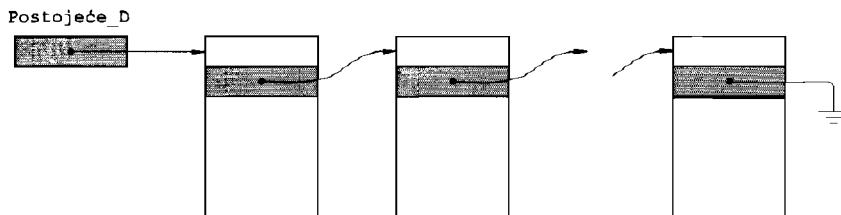
U strukturi podataka jezgre postoje ili se mogu kreirati *zaglavljiva liste* u koje će se svrstavati opisnici dretvi. Pojedine jezgrine funkcije premještati će, prema potrebi, opisnike



dretvi iz jedne liste u drugu. Smještanje opisnika dretve u pojedinu od lista odredit će njezino trenutačno stanje. Opisat ćemo ukratko liste u strukturi podataka jezgre i s njima povezana stanja dretvi.

5.2.1. Lista postojećih dretvi, pasivno stanje

Sve dretve koje su smještene u adresnom prostoru procesa prikladno je smjestiti u jednu listu tako da se, kada je to potrebno, njihovi opisnici mogu svi redom pregledati. Na slici 5.3. prikazana je takva lista čije je zaglavje nazvano *Postojeće_D*, gdje sufiks *_D* nadomešta riječ dretva.



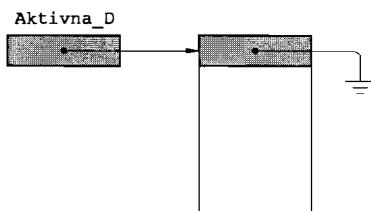
Slika 5.3. Lista postojećih dretvi

Povezivanje u tu listu obavljeno je drugom po redu kazaljkom opisnika jer će nam prva po redu kazaljka svakog opisnika poslužiti za povezivanje u druge liste. Na taj način uvijek možemo obići sve opisnike u listi postojećih dretvi bez obzira na to u kojoj se on listi trenutačno nalazi.

Za dretvu čiji se opisnik nalazi samo u listi *Postojeće_D*, odnosno ne nalazi se ni u jednoj drugoj listi, kažemo da se nalazi u *pasivnom stanju*.

5.2.2. Aktivno stanje dretve

U jednoprocesorskom sustavu samo jedna od dretvi može biti *aktivna*. To je ona dretva čije instrukcije procesor upravo izvodi. Opisnik te dretve stavit će jezgra u posebnu listu, čije ćemo zaglavje označiti s *Aktivna_D*. U toj se listi (možemo je nazvati i redom) smije nalaziti najviše jedan opisnik dretve. Lista je prikazana na slici 5.4.



Slika 5.4. Lista aktivne dretve

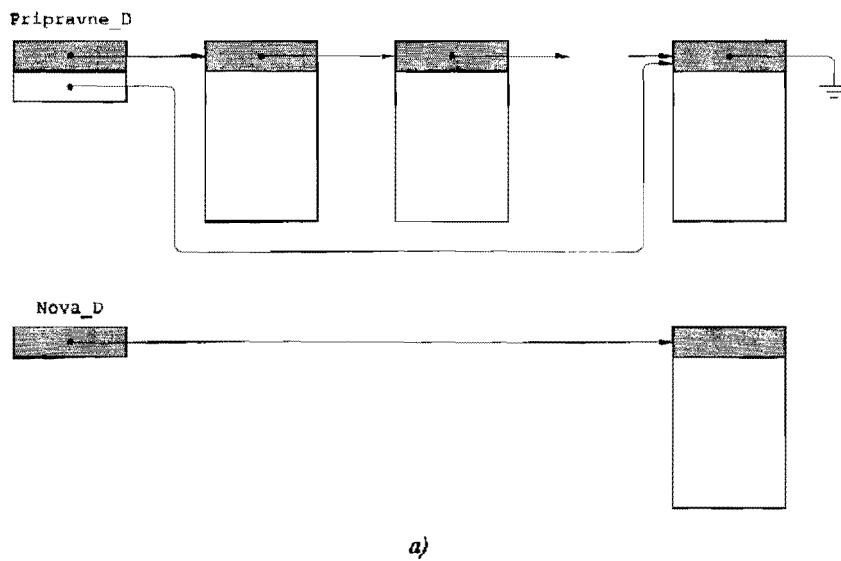
Pri svakom pohranjivanju konteksta sadržaji registara procesora smještati će se na za to predviđeno mjesto unutar opisnika koji se nalazi u toj listi. Nakon toga taj će se opisnik premjestiti u neku drugu listu, a u listu **Aktivna_D** premjestiti će se opisnik koji upravo treba biti aktiviran. Iz tog će se opisnika napuniti registri procesora kontekstom te nove aktivne dretve.

5.2.3. Pripravno stanje dretve, red pripravnih dretvi

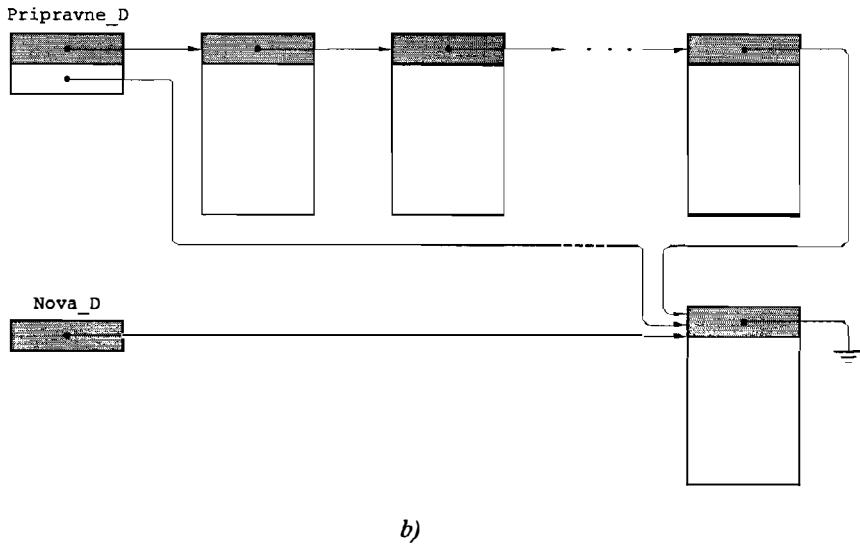
U jednoprocесорском sustavu samo jedna dretva može u jednom trenutku biti aktivna. Sve ostale dretve koje bi se također mogle izvoditi moraju čekati na dodjelu procesora. Tako, primjerice, u sustavu dretvi prema slici 4.5. nakon završetka dretve D_1 tri dretve mogu započeti izvođenje. Samo jedna od njih može biti pripuštena u procesor i postat će aktivna, dok ostale dvije moraju čekati. Postavlja se pitanje na koji se način odabire dretva koja će biti pripuštena u procesor, odnosno na koji se način *pridjeljuje* ili *raspoređuje* procesor za obavljanje pojedinih dretvi².

U najjednostavnijem obliku mogu se dretve koje čekaju na dodjelu procesora pripuštati onim redoslijedom kojim su one postajale pripravne za izvođenje. U strukturi podataka jezgre pritom treba s pomoću liste oblikovati red pripravnih dretvi. Takav je red prikazan na slici 5.5.

Opisnik nove dretve koja postaje pripravna za izvođenje dolazi na kraj liste, a dretva čiji se opisnik nalazi na početku liste bit će prva pripuštena u procesor. Na slici 5.5. je, usput,

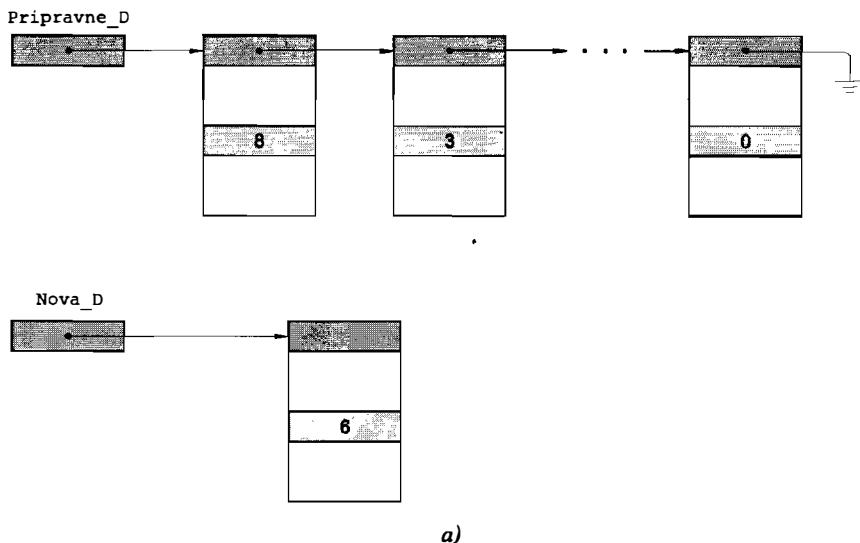


² U engleskom se jeziku za problem raspoređivanja procesora (a i ostalih sredstava računala) upotrebljava naziv *scheduling*.

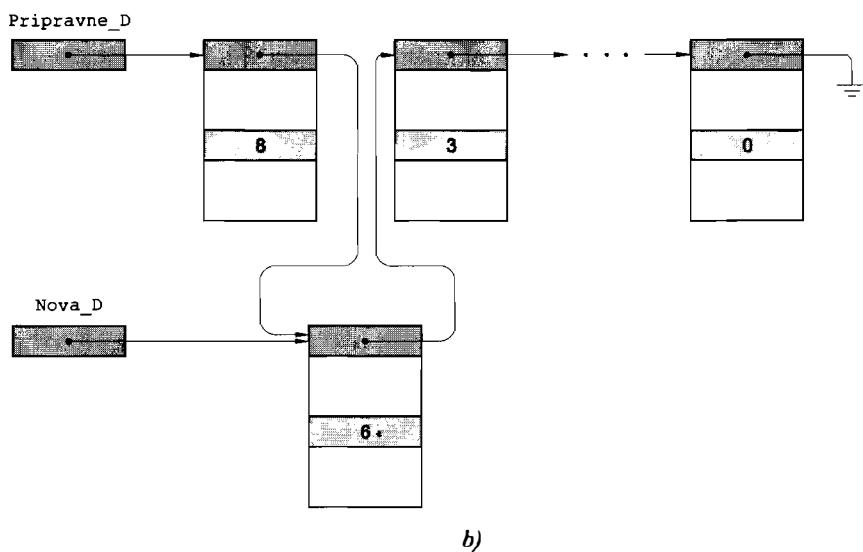
**Slika 5.5.** Red pripravnih dretvi po redu prispijeća:

a) prije umetanja opisnika nove dretve, b) nakon umetanja opisnika nove dretve

ilustrirana činjenica da će umetanje novog opisnika na kraj liste biti obavljeno sa složenošću $O(1)$, tj. trajanje umetanja neće ovisiti o duljini reda ako se zaglavlje *Pripravne_D* sačini od dvije kazaljke: jedne koja pokazuje na početak reda i druge koja pokazuje na kraj reda. Ovako oblikovani red osigurava dodjelu procesora dretvama *po redu prispijeća*³.



³ U engleskom se jeziku za takav način posluživanja dretvi koristi kratica *FCFS* od *First-Come, First-Served* – prvi prispio, prvi poslužen. Lista koja podržava disciplinu posluživanja po redu prispijeća dobila je naziv *queue* – red, koji se također obilježava kraticom *FIFO* od *First-In, First-Out* – prvi unutra, prvi van. Podsjetimo se da se lista u koju se elementi stavljaju i uzimaju s istog kraja naziva stogom (engl. *stack*) i da se obilježava kraticom *LIFO* od *Last-In, First-Out* – zadnji unutra, zadnji van.



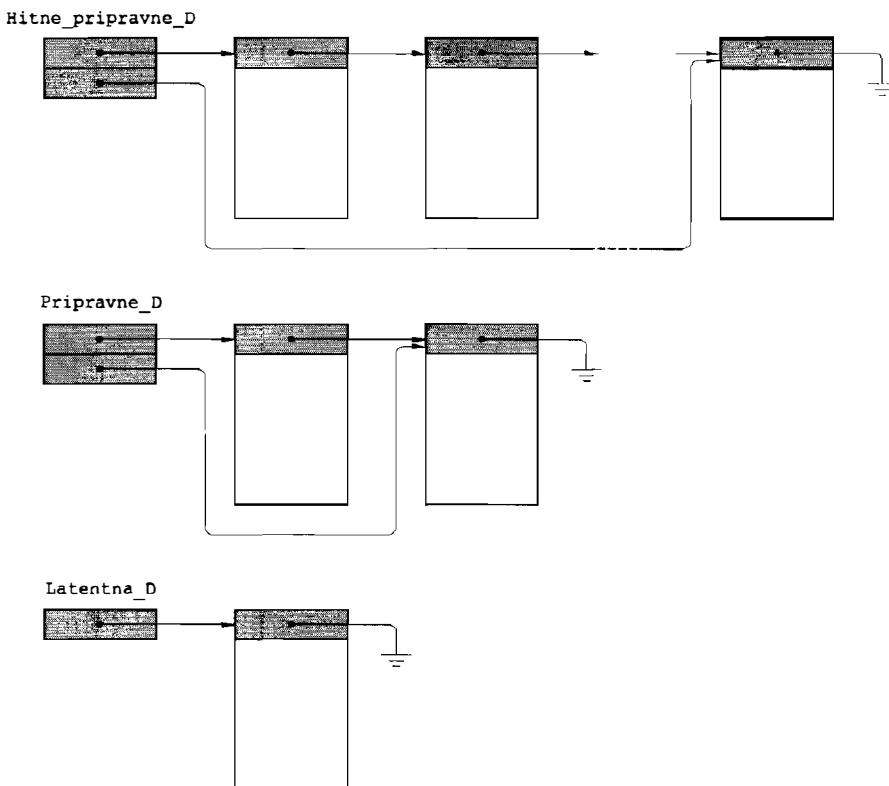
Slika 5.6. Red pripravnih dretvi organiziran po prioritetima:

a) prije umetanja opisnika nove dretve, b) nakon umetanja opisnika nove dretve

Drugi se mogući način odabira dretvi može zasnovati na uvažavanju prioritetima pojedinih dretvi. Pri opisu opisnika dretve ustanovili smo da se dretvama mogu pripisati prioriteti. Opisnike pripravnih dretvi treba svrstatи u listu u skladu s njihovim prioritetom. Slika 5.6. prikazuje red *Pripravne_D* organiziran na taj način. Iz slike se može zaključiti da umetanje opisnika u takav red zahtijeva pregledavanje cijelog reda i složenosti je $O(N)$ ⁴. S obzirom na to da broj razina prioriteta nije prevelik, obično se postupa tako da se za svaku razinu prioriteta predvidi posebni red koji se oblikuje po redu prispijeća. Novi opisnik tada se jednostavno, sa složenošću $O(1)$, stavlja na kraj reda one razine kojoj pripada. Dretve se za izvođenje odabiru tako da se najprije uzimaju dretve iz reda najvišeg prioriteta. Tek kada se taj red isprazni, počinje pražnjenje reda sljedeće razine. Općenito se redovi nižih razina počinju prazniti tek kada su sve dretve iz viših razina obavljene.

Postavlja se pitanje što procesor radi kada ni jedna dretva nije pripravna za izvođenje. To je najprikladnije riješiti jednom latentnom dretvom koja ne radi ništa korisno (ali ni ništa štetno) i samo troši vrijeme procesora. Ona se aktivira onda kada u redu pripravnih dretvi nema niti jedne druge dretve. Tako bismo u jednostavnom sustavu s dvije razine prioriteta imali trorazinski red pripravnih dretvi, kao što to prikazuje slika 5.7. Najprije će se obavljati dretve stavljene iz reda *Hitne_pripravne_D*. Tek kada se taj red potpuno isprazni, počinju se izvoditi dretve čiji su opisnici smješteni u red *Pripravne_D*. Ako u međuvremenu postane pripravna neka nova hitna dretva, ona će opet biti izvedena prije ostalih dretvi iz reda normalnih dretvi. Kada se ova reda isprazne, tj. kada nema niti jedne pripravne dretve, izvodit će se latentna dretva. Možemo reći da je procesor za vrijeme izvođenja latentne dretve prazan.

⁴ Možemo reći da lista pripravnih procesa mora ostati sortirana nakon umetanja novog opisnika. Ovdje nas to posebno ne zanima, ali zainteresirani čitatelj može se podsjetiti da se takva lista može oblikovati kao uravnoteženo binarno stablo te da su operacije uzmimanja i stavljanja u takvu listu složenosti $O(\log N)$.



Slika 5.7. Red pripravnih dretvi s dvije razine prioriteta i latentnom dretvom

Vratit ćemo se na problem rasporedjivanja procesora u poglavljju 7. Već smo rekli da se dretva aktivira tako da se njezin opisnik premjesti iz reda **Pripravne_D** u red **Aktivna_D** i nakon toga se iz tog opisnika prenosi kontekst dretve u registre procesora.

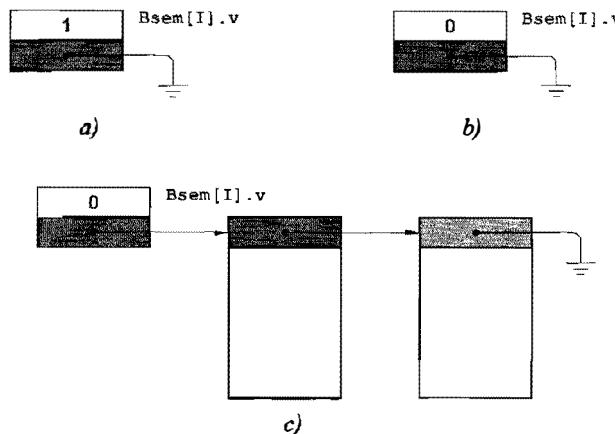
5.2.4. Blokirana stanja dretvi

Dretve tijekom svog izvođenja mogu biti blokirane čekajući na ispunjenje nekog uvjeta za njihovo daljnje napredovanje. U našem modelu jezgre pretpostaviti ćemo da dretve mogu biti blokirane na četiri načina:

- čekanjem na binarnom semaforu;
- čekanjem ne općem semaforu;
- čekanjem na istek zadanog intervala kašnjenja;
- čekanjem na završetak ulazno-izlazne operacije.

Binarni semafor je mehanizam koji omogućuje ostvarenje međusobnog isključivanja dretvi. Binarni semafor sastoji se od jedne varijable, imenujmo je **Bsem[I].v**, koja igra ulogu zastavice i pridružene joj kazaljke koja pokazuje na red opisnika dretvi koje

nisu uspjele proći semafor. Vrijednost semafora $Bsem[I].v == 1$ označava da je semafor prolazan, a vrijednost $Bsem[I].v == 0$ označava da je semafor neprolazan. U sustavu možemo imati i više semafora, što je simbolizirano indeksom I. Dretva koja nađe na prolazni semafor bit će propuštena i jezgra će staviti vrijednost 0 u $Bsem[I].v$. Pretpostavlja se da će dretva nakon nekog vremena javiti da izlazi iz kritičnog odsječka, čime se stvaraju uvjeti za postavljanje vrijednosti semafora u 1. Ako jedna ili više dretvi pokušaju proći uz neprolazni semafor, one će biti zaustavljane i njihovi će opisnici biti prebačeni u red pridružen tom semaforu. Dretve će iz tog reda biti deblokirane jedna po jedna svaki put kada prethodna dretva javi da izlazi iz kritičnog odsječka. Moguća stanja binarnog semafora prikazuje slika 5.8. Kada je semafor prolazan, $Bsem[I].v == 1$ i ni jedan opisnik se ne nalazi u redu semafora. Slika 5.8.b) prikazuje stanje semafora kada je jedna dretva prošla semafor, ali niti jedna druga nije pokušala proći kraj njega. Konačno, slika 5.8.c) ilustrira semafor koji je prošla jedna dretva, a druge dvije su pokušale proći i čekaju u redu za prolaz. Uobičajeno se redovi uz semafore stvaraju po redu prispijeća dretvi, čak i onda kada su procesi razvrstani u razine prioriteta koji se uvažavaju pri raspoređivanju procesora.



Slika 5.8. Moguća stanja binarnog semafora:

- a) semafor je prolazan,
- b) semafor je neprolazan (jedna dretva je prošla semafor),
- c) semafor je neprolazan (jedna dretva je prošla i dvije čekaju na prolaz)

Opći semafor razlikuje se od binarnog semafora po tome što njegova vrijednost $Osem[J].v$ može poprimiti vrijednost cijelog broja, a ne samo vrijednosti 0 ili 1. Kada dretva pokušava proći opći semafor, jezgra smanji njegovu vrijednost za jedan i ako je nakon toga $Osem[J].v \geq 0$, dopušta dretvi prolaz. Ako je $Osem[J].v < 0$, onda se dretva blokira u redu pridruženome tom općem semaforu. Neka druga dretva koja zahtijeva postavljanje općeg semafora uzrokovat će povećanje vrijednosti $Osem[J].v$. Time se može ispuniti uvjet za pokretanje dretve koja je bila blokirana. Vidjet ćemo pri opisu funkcija da se opći semafor može ostvariti na nekoliko načina.

Treći razlog blokiranja neke dretve može biti željeno *odgađanje izvođenja* za zadani interval vremena. Taj interval može biti cjelobrojni višekratnik periode otkucaja sata T_q ,

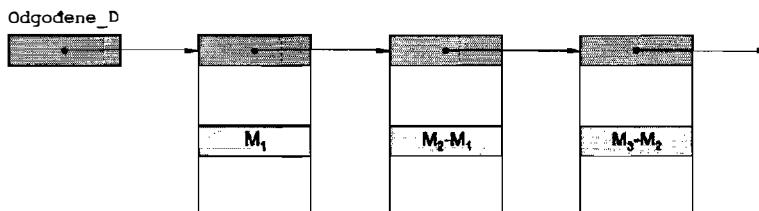
odnosno

$$\Delta T = M \times T_q.$$

Opisnik dretve koju se želi odgoditi stavlja se u red `Odgođene_D` i u njegovu lokaciju `Zadano_kašnjenje` zapisuje broj M . Pri svakom prekidu od sata vrijednost pohranjena u opisniku smanjuje se za jedan, i kada dosegne nulu, dretva se iz reda odgođenih dretvi prebacuje među pripravne dretve. Ako više dretvi treba odgodeno izvoditi, njih se u red odgođenih dretvi može smjestiti sortirano. Ako dretve želimo odgoditi tako da je:

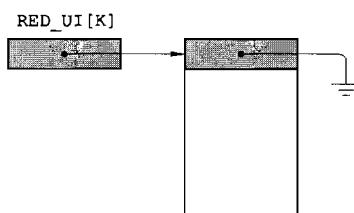
$$M_1 < M_2 < M_3 < \dots,$$

onda se može oblikovati red odgođenih dretvi u skladu sa slikom 5.9. U prvi opisnik upisuje se apsolutna vrijednost zadatog kašnjenja, a u ostale opisnike samo dodatno odgađanje u odnosu na prethodnu dretvu. Time se postiže to da se smanjenje u lokaciji `Zadano_kašnjenje` obavlja samo u prvom opisniku. Kada ta vrijednost u prvom opisniku padne na nulu, opisnik se uklanja iz reda i počinje smanjivanje u drugom opisniku.



Slika 5.9. Red odgođenih dretvi

Konačno, četvrti razlog blokiranja u našem modelu dretve jest čekanje na završetak ulazno-izlazne operacije. Dretva može obavljati ulazno-izlaznu operaciju samo preko jezgre. Prepostavit ćemo da svaku ulazno-izlaznu napravu dretve koriste pojedinačno. To znači da bismo svakoj ulazno-izlaznoj napravi trebali pridružiti jedan binarni semafor koji dretva najprije mora proći. Nakon toga ona će pozivati jezgrinu funkciju s pomoću koje će obavljati ulazno-izlazne operacije. Prepostavimo da dretva koja je zatražila obavljanje operacija mora pričekati završetak operacije. Njegovu će se opisnik nakon početka operacije smjestiti u `RED_UI [K]`, kao što prikazuje slika 5.10. S obzirom na to da smo prepostavili da se ulazno-izlazne naprave koriste pojedinačno, u tom se redu može naći samo jedan opisnik. On će bit premješten u red pripravnih dretvi onda kada se dogodi sklopovalski prekid od naprave K.



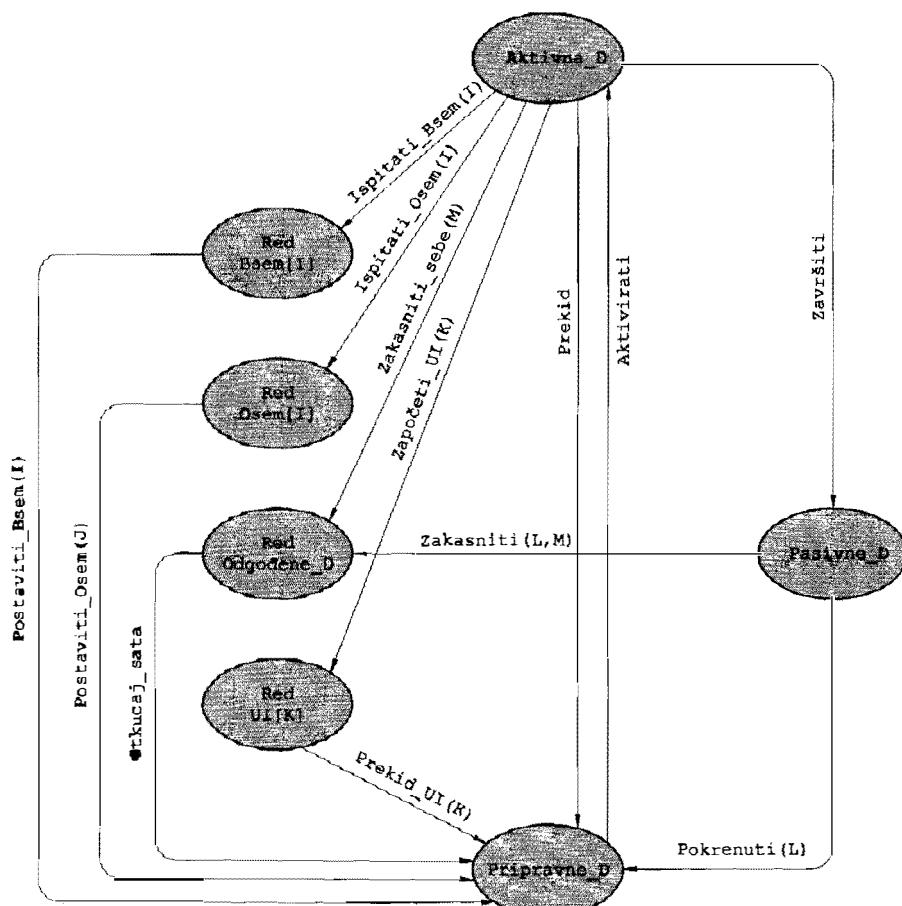
Slika 5.10. Red čekanja na završetak ulazno-izlazne operacije



Ako sve dretve iz sustava dretvi intenzivno obavljaju ulazno-izlazne operacije, onda se može dogoditi da su one u nekom vremenskom razdoblju sve blokirane. U računalnom se sustavu ništa ne događa (osim izvođenja latentne dretve), on čeka da se dogodi neki prekid koji će "oživjeti" barem jednu dretvu.

5.2.5. Prikaz mogućih stanja dretvi

Tijekom svojeg izvođenja pojedina se dretva može nalaziti u jednom od gore opisanih stanja. Za svako od stanja postoje pripadne liste, osim za pasivno stanje. Dretva prelazi iz jednog stanja u drugo pod utjecajem jezgrinih funkcija koje ćemo detaljnije razmotriti u sljedećem odjeljku. Slika 5.11. prikazuje sva moguća stanja i prijelaze između njih u obliku grafa. Stanja su predviđena čvorovima koji nose jednake nazine kao i zaglavljaju lista u kojima se nalaze opisnici dretvi kada se nalaze u tom stanju. Iznimka je jedino stanje



Slika 5.11. Graf mogućih prijelaza dretvi iz stanja u stanje

koje smo nazvali **Pasivne_D**. Naime, listu s istim imenom nismo predvidjeli jer se dretva može smatrati pasivnom onda kada postoji samo u listi **Postojeće_D**, a ne nalazi se ni u jednoj drugoj listi.

Trenutačno stanje dretve može se zapisati i u opisnik dretve (u kojoj je za to predviđeno mjesto), tako da se pri pregledavanju liste **Postojeće_D** može ustanoviti u kojem se stanju dretva nalazi.

Utvrdimo još jedanput da se pojedina dretva u jednom trenutku može naći samo u jednom od stanja. Mogući prijelazi između stanja označeni su na slici 5.11. strelicama. Uz strelice su napisana imena jezgrinih funkcija koje pod određenim uvjetima mogu uzrokovati pojedine od prijelaza, kao i prekidi pod čijim se utjecajem mogu dogoditi pojedini prelazi.

Na slici je vidljivo da dretve prije aktiviranja moraju proći kroz pripravno stanje. Prilikom aktiviranja opisnik prve po redu dretve iz reda **Pripravne_D** prebacuje se u red **Aktivna_D**. Iz tog se deskriptora tada puni kontekst dretve u procesor i počinje njezino izvođenje. Iz aktivnog stanja dretva može izaći na nekoliko načina:

- ako je dretva uspješno obavila svoj posao, ona će pozvati jezgrinu funkciju **Završiti** i prijeći u pasivno stanje;
- ako dretva želi ispitati binarni ili opći semafor, ona će pozvati jezgrine funkcije **Ispitati_Bsem(I)**, odnosno **Ispitati_Osem(J)** i, ako su semafori neprolazni, bit će blokirana;
- ako dretva sama sebe želi zakasniti, pozvat će jezgrinu funkciju **Zakasniti_sebe(M)** i njezin će opisnik biti prebačen u red **Odgodenе_D**;
- ako dretva želi obaviti ulazno-izlaznu operaciju preko naprave **UI[K]**, ona će biti blokirana i njezin će opisnik biti premješten u pripadni red.

Osim toga, dretva može biti "izbačena" iz aktivnog stanja onda kada se dogodi bilo koji sklopoški prekid. Prekid, u pravilu, nema veze s trenutačno aktivnom dretvom. Toj se dretvi mora oduzeti procesor kako bi se mogla obaviti jezgrena aktivnost koja je posljedica toga prekida. Kontekst aktivne dretve premješta se zbog toga iz procesora u opisnik koji se nalazi u redu **Aktivna_D** i taj se opisnik vraća u red pripravnih dretvi jer će se ta dretva moći normalno nastaviti. Nakon što se unutar jezgre obradi prekid (jezgrinom funkcijom koja će tim prekidom biti pozvana) može se dogoditi aktiviranje te iste dretve, ali isto tako i neke druge dretve.

Dretva koja se nalazi u blokiranim stanju sama sebi ne može nikako pomoći. Ona može biti izbavljena iz blokiranih stanja na sljedeće načine:

- ako je dretva bila blokirana na binarnom ili općem semaforu, deblokirat će je neka druga dretva pozivajući funkcije **Postaviti_Bsem[I]**, odnosno **Postaviti_Osem[J]**;
- ako se dretva nalazi u redu **Odgodenе_D**, deblokirat će je funkcija koja obrađuje prekid od sata, nazovimo je **Otkucaj_sata**;
- ako se dretva nalazi u redu naprave **UI[K]**, deblokirat će je funkcija **Prekid_UI(K)** koja obrađuje prekid od te naprave.

Spomenimo još funkciju Pokrenuti(L) koja dretvu s identifikatorom L iz pasivnog stanja prevodi u pripravno stanje te funkciju Zakasniti(L,M) koja dretvu iz pasivnog stanja premješta u red Odgođene_D i na taj način obavlja odgođeno pokretanje dretve L.

Pogledat ćemo u sljedećem odjeljku kako se mogu ostvariti gore opisane jezgrine funkcije.

5.3. Jezgrine funkcije

5.3.1. Ulazak u jezgru i izlazak iz jezgre

Poziv jezgrine funkcije – ulazak u jezgru – zbiva se pod utjecajem sklopovskog ili programskega prekida. U jednoprocесorskom sustavu time je osigurano međusobno isključeno obavljanje jezgrinih procedura. U 3. smo se poglavljju detaljno pozabavili problemima obrade prekida. Stoga ovdje možemo samo kratko ponoviti da ćemo pretpostavljati kako se pojmom sklopovskog prekida ili programski izazvanog prekida događa sljedeće:

- onemogućuje se prekidanje;
- programsko brojilo smješta se na sustavski stog;
- prekidni podsustav premješta kontekst iz registara procesora na sustavski stog;
- prekidni podsustav poziva odgovarajuću jezgrinu funkciju.

Jezgrina funkcija koja je prekidom pozvana morala bi na svome početku najprije premjestiti kontekst sa sustavskog stoga u opisnik dretve koja je u trenutku prekida bila aktivna. Taj se opisnik nalazi u redu Aktivna_D pa ćemo u opisima jezgrinih funkcija taj čin kratko opisati instrukcijom:

 pohraniti kontekst u opisnik Aktivna_D;

U praktičnim izvedbama jezgre može se pohranjivanje konteksta skratiti tako da se pri obradi prekida najprije kazaljka sustavskog stoga usmjeri na opisnik aktivne dretve i time izbjegne kasnije premještanje konteksta.

Na početku poglavљa rekli smo da se izlazak iz jezgre svodi na aktiviranje jedne od dretvi. To će, osim u nekim iznimnim slučajevima, biti dretva koja se nalazi na prvom mjestu u redu Pripravne_D. Postupak aktiviranja možemo opisati sljedećim slijedom instrukcija:

 premjestiti prvi opisnik iz reda Pripravne_D u red Aktivna_D;
obnoviti kontekst iz opisnika Aktivna_D;
omogućiti prekidanje;
vratiti se iz prekidnog načina;

Instrukcija vratiti se iz prekidnog načina vraća u procesor sadržaj programskog brojila i prevodi procesor iz sustavskog u korisnički način rada.

Gornji ćemo slijed instrukcija u opisu funkcija uvijek kada je to moguće skraćeno opisati jednom instrukcijom:

```
aktivirati prvu dretvu iz reda Pripravne_D;
```



Jezgrine funkcije opisivat ćemo u obliku njihove deklaracije s tim da ćemo opis započeti s ključnom riječi j-funkcija.

5.3.2. Funkcije za binarni semafor

Binarni semafor može nam poslužiti za ostvarenje međusobnog isključivanja⁵. Binarnih semafora u strukturi podataka jezgre može biti više pa binarne semafore razlikujemo po njihovu indeksu. Osnovnu ulogu binarnih semafora i željeni način njihova rada razmotrili smo u prethodnom odjeljku. Uz binarni semafor potrebne su nam dvije funkcije:

- Ispitati_Bsem(I), koju dretva poziva kada želi ući u kritični odsječak i
- Postaviti_Bsem(I), koju dretva poziva kada izlazi iz kritičnog odsječka.

Zaštita kritičnog odsječka ovim funkcijama mogla bi izgledati ovako:

```
Is_itati_Bsem(I);
kritični_odsječak;
Postaviti_Bsem(I);
nekritični_odsječak;
```



Iz opisa djelovanja binarnog semafora slijedi da bi funkcija ispitivanja binarnog semafora mogla izgledati ovako:

```
j-funkcija Ispitati_Bsem(I) {
    pohraniti kontekst u opisnik Aktivna_D;
    ako je (Bsem[I].v == 1) {
        Bsem[I].v = 0;
        obnoviti kontekst iz opisnika Aktivna_D;
        omogućiti prekidanje;
        vratiti se iz prekidnog načina;
    }
    inače {
        premjestiti opisnik iz reda Aktivna_D u red Bsem[I];
        aktivirati prvu dretvu iz reda Pripravne_D;
    }
}
```



⁵ U konkretnim ostvarenjima jezri za mehanizam binarnog semafora upotrebljava se kratica **mutex** (od engleskog *mutual exclusion*). Ovdje zadržavamo naziv binarni semafor zbog toga što se njegovim poopćenjem on može dovesti u vezu s općim semaforom, ili drugičje rečeno: binarni semafor može se smatrati specijalnim slučajem općeg semafora.

Ova je funkcija osmišljena tako da dretva koja naiđe na prolazni semafor nastavlja svoje izvođenje. Kontekst te dretve koji je pri pozivu bio pohranjen u opisnik `Aktivna_D` vraća se neposredno u procesor. Jedina posljedica poziva jezgrine funkcije bit će promjena vrijednosti semafora. Ako je dretva naišla na neprolazni semafor njezin će opisnik bit premješten u red tog semafora, tj. dretva postaje blokirana, a aktivirat će se prva po redu dretva pripravna za izvođenje.

Druga funkcija koja djeluje na binarni semafor jest funkcija `Postaviti_Bsem(I)`. Ta bi funkcija mogla izgledati ovako:

```
j-funkcija Postaviti_Bsem(I) {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u red Pripravne_D;
    ako je (red Bsem[I] neprazan) {
        premjestiti prvi opisnik iz reda Bsem[I] u red Pripravne_D;
    }
    inače {
        Bsem[I].v = 1;
    }
    aktivirati prvu dretvu iz reda Pripravne_D;
}
```

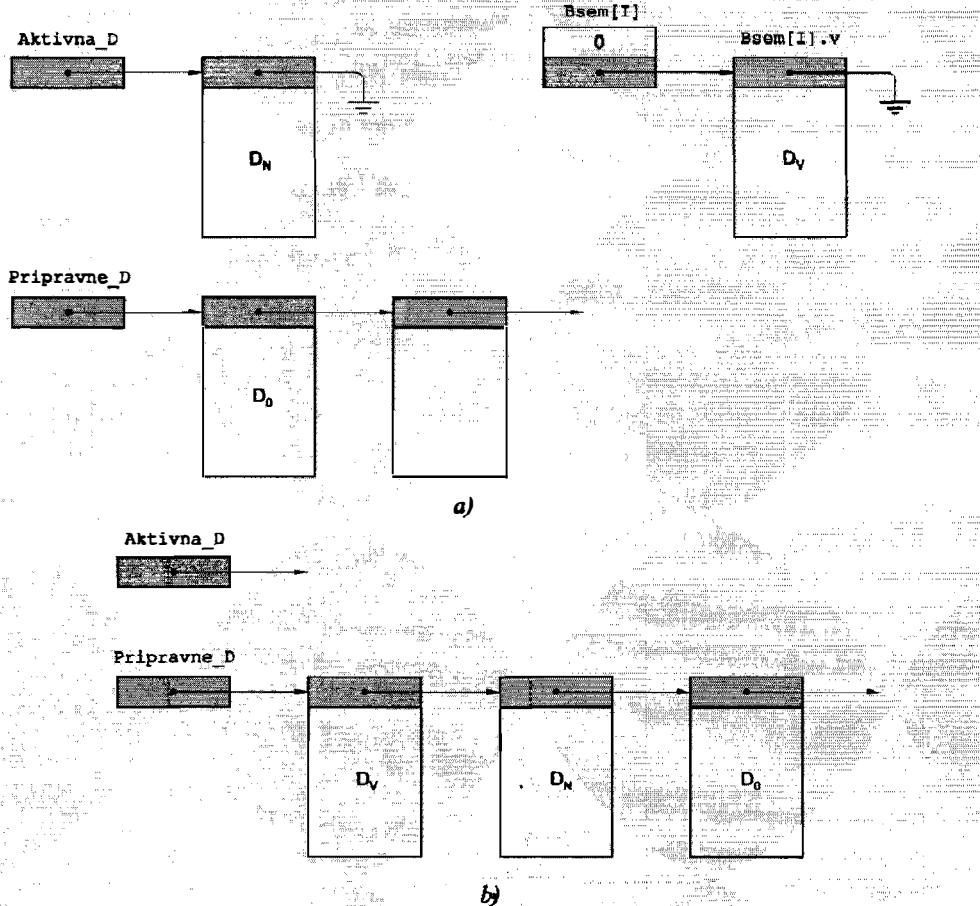
U prvi mah može izgledati nelogično da se dretva koja hoće samo nešto omogućiti (osloboditi prolaz kroz kritični odsječak nekoj drugoj dretvi) ne pušta neposredno natrag u procesor, već se vraća u red pripravnih dretvi. Na kraju funkcije aktivira se dretva čiji se opisnik nalazi na prvom mjestu u tom redu. To može biti ista dretva koja je i pozvala funkciju. Sljedeći primjer ukazuje na to da ovakav izgled funkcije ima svog opravdanja.

PRIMJER 5.1.

Zamislimo dva scenarija uporabe binarnog semafora. Po prvom scenaru dretva prolazi kroz binarni semafor, obavlja svoj kritični odsječak i nakon toga postavlja binarni semafor natrag u prolazno stanje i u tom razdoblju ni jedna druga dretva ne pokušava proći pokraj istog semafora. Ako je red pripravnih dretvi organiziran po prioritetskom načelu, tada će funkcija `Postaviti_Bsem(I)` morati staviti opisnik dretve koja ju je pozvala na početak reda⁶. U tom će se slučaju jedan te isti opisnik premještati najprije iz reda `Aktivna_D` u red `Pripravne_D` i zatim natrag. Međutim, u višedretvenom sustavu to je vjerojatan i drukčiji scenarij. Zamislimo da neka dretva D_V višeg prioriteta pokušava proći neki binarni semafor `Bsem[J]` i na njemu se blokira. Nakon toga biva aktivirana neka dretva nižeg prioriteta D_N i uspješno prođe binarni semafor `Bsem[I]`. Dok se dretva D_N nalazi u kritičnom odsječku zaštićenom `Bsem[I]`, ona može biti prekinuta i vraćena među pripravne dretve. Za to vrijeme dretva D_V može uspjeti obaviti kritični

⁶ Dretva je bila aktivirana jer se nalazila na prvom mjestu, a do trenutka kada ona poziva jezgru ni jedna druga dretva nije mogla biti stavljena u red pripravnih dretvi!

odsječak zaštićen binarnim semaforom $Bsem[J]$ (koji je oslobođila neka treće dretva) i može pokušati proći neprolazni semafor $Bsem[I]$. Na tom će se semaforu dretva D_V blokirati i sada opet dretva D_N uspijeva ući u procesor. U tom trenutku u strukturi podataka jezgre zatječemo stanje prema slici 5.12.a).



Slika 5.12. Dio strukture podataka jezgre
a) prije poziva funkcije Postaviti_Bsem(I),
b) prije izvođenja instrukcije "aktivirati prvu dretvu iz reda Pripravne_D"

Ako sada dretva D_N pozove funkciju $Postaviti_Bsem(I)$, onda će se u redu $Pripravne_D$ u trenutku prije nego li funkcija izvede instrukciju "aktivirati prvu dretvu iz reda $Pripravne_D$ " naći opisnici dretvi D_V i D_N poredani prema slici 5.12.b). Nakon izlaska iz jezgre stoga će postati aktivna dretva D_V , kako i treba biti s obzirom na to da ima najveći prioritet u redu pripravnih dretvi.

Opisani problem kada dretva višeg prioriteta bude blokirana zbog sredstva koje je pretodno zauzela dretva nižeg prioriteta nazivamo problemom inverzije prioriteta. Problem je detaljnije opisan u poglavljju 6.4.

5.3.3. Funkcije za opći semafor

Opći semafor može poslužiti za sinkronizaciju dretvi te za brojenje događaja i sredstava. Moguće su različite izvedbe općeg semafora.

Najprije ćemo opisati način izvedbe općeg semafora kojim se ostvaruje mehanizam semafora u skladu s izvornim opisom E.W. Dijkstre⁷. Takav ćemo opći semafor označiti s Os. Pri opisivanju strukture podataka jezgre ustavili smo da se opći semafor razlikuje od binarnog semafora po tome što će njegova vrijednost biti cijeli broj. Pri ispitivanju semafora najprije mu se zatečena vrijednost smanjuje za jedan i zatim se dretva koja je zatražila njegovo ispitivanje blokira ako je vrijednost postala manja od nule. Pri kreiranju općeg semafora mora postojati mogućnost zadavanja početne vrijednosti semafora (koja može biti i manja od nule!). Najjednostavnije je razmatrati djelovanje takvog semafora u slučaju kada ga ispituje samo jedna dretva. Takav bismo semafor mogli nazvati privatnim semaforom pridružene mu dretve.

Semafor koji slijedi koncipiran je upravo u tom okruženju, pridijeljen je samo jednoj dretvi koja ga ispituje. Ispitivanje ovog semafora od strane više dretvi neće se zato ni razmatrati jer on za to nije namijenjen.

Funkcija ispitivanja takvog semafora mogla bi izgledati ovako:



```
j-funkcija Ispitati_Os(J) {
    pohraniti kontekst u opisnik Aktivna_D;
    Os[J].v = Os[J].v - 1;
    ako je (Os[J].v >= 0) {
        obnoviti kontekst iz opisnika Aktivna_D;
        omogućiti prekidanje;
        vratiti se iz prekidnog načina;
    }
    inače {
        premjestiti opisnik iz reda Aktivna_D u red Os[J];
        aktivirati prvu dretvu iz reda Pripravne_D;
    }
}
```

Komplementarna funkcija postavljanja ovakvog općeg semafora mogla bi izgledati ovako:



```
j-funkcija Postaviti_Os(J) {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u red Pripravne_D;
    Os[J].v = Os[J].v + 1;
```

⁷ Dijkstra, E.W., *Cooperating Sequential Processes* u knjizi *Programming Languages*, Academic Press, London 1968., (rad je prvočno objavljen u obliku izvještaja Tehničkog sveučilišta u Eindhovenu, Nizozemska, 1965. godine).



```

    ako je (Os[J].v == 0) ∧ (red Os[J] neprazan) {
        premjestiti opisnik iz reda Os[J] u red Pripravne_D;
    }
    aktivirati prvu dretvu iz reda Pripravne_D;
}

```

Ovako izvedeni semafor mogao bi poslužiti kao sinkronizacijski mehanizam u sustavu dretvi. Podsjetimo se da se sustav dretvi sastoji od međusobno nezavisnih i međusobno zavisnih dretvi. Za zavisne dretve mora biti određen redoslijed izvođenja. Tako smo u poglavlju 4. na primjeru sustava dretvi prikazanom na slici 4.5. ustanovili da dretve D_2 , D_3 i D_4 smiju započeti svoje izvođenje tek kada završi dretva D_1 . Dretva D_5 smije započeti tak kada završe dretve D_2 i D_3 koje joj neposredno prethode, a dretva D_6 smije započeti tek kada završe njezine neposredne prethodnice D_3 i D_4 . Konačno, dretva D_7 smije započeti tek kada svoje izvođenje završe dretve D_5 i D_6 . Ovakvo bi se izvođenje moglo postići tako da se svakoj dretvi D_i pridruži njezin "privatni" semafor $Os[I]$ i da mu se pripiše početna vrijednost $Os[I].v$ jednaka $(1 - Np[I])$, gdje je $Np[I]$ broj neposrednih prethodnika dretve D_i . Programe dretvi treba nadopuniti tako da svaka dretva prije svoje prve instrukcije poziva funkciju kojom ispituje svoj semafor, a nakon svoga završetka poziva funkcije kojima postavlja semafore svih svojih neposrednih sljedbenica.

PRIMJER 5.2.



Pogledajmo kako bi trebalo nadopuniti programe dretvi kako bi se ostvario sustav prikazan na slici 4.5. Iz slike se može ustanoviti broj neposrednih prethodnika i odrediti početne vrijednosti semafora prema sljedećoj tablici:

Tablica 5.1. Početne vrijednosti općeg semafora

I	1	2	3	4	5	6	7
Np[I]	0	1	1	1	2	2	2
Os[I].v	1	0	0	0	1	-1	-1

Ako s $D(I)$ obilježimo programske odsječak dretve D_I , onda ćemo nadopunjene dretve D'_I dobiti na sljedeći način:

D'_1	Ispitati_Os(1); $D(1)$; Postaviti_Os(2); Postaviti_Os(3); Postaviti_Os(4);
D'_2	Ispitati_Os(2); $D(2)$; Postaviti_Os(5);
D'_3	Ispitati_Os(3); $D(3)$; Postaviti_Os(5); Postaviti_Os(6);
D'_4	Ispitati_Os(4); $D(4)$; Postaviti_Os(6);
D'_5	Ispitati_Os(5); $D(5)$; Postaviti_Os(7);
D'_6	Ispitati_Os(6); $D(6)$; Postaviti_Os(7);
D'_7	Ispitati_Os(7); $D(7)$;

Nakon što smo tako nadopunili programske odsječke dretvi, one se mogu sve proglašiti pripravnima i može ih se u proizvoljnem redoslijedu smjestiti u red pripravnih dretvi.

Uvjerite se da će opisani mehanizam općeg semafora osigurati izvođenje sustava propisanim redoslijedom.

Od istih dretvi možemo izgraditi drugi sustav uspostavljanjem drukčijih veza između dretvi. Programski ćemo te promjene ostvariti samo drukčijim pozivima funkcija za postavljanje semafora te promjenom početnih vrijednosti semafora.

Ovakvo ostvarenje općeg semafora možemo nazvati *sinkronizacijskim semaforom*.

U suvremenim operacijskim sustavima uobičajeno je malo drukčije ostvarenje općeg semafora. U tom ostvarenju vrijednost semafora ne može postati negativna. Takav semafor ne mora biti privatni, tj. bilo koja dretva može ispitivati semafor, a jasno i postavljati semafor. Vrijednost semafora smanjuje se samo onda ako smanjena vrijednost neće postati negativna.

Funkcija ispitivanja takvog semafora izgledala bi ovako:



```
j-funkcija Ispitati_Osem(J) {
    pohraniti kontekst u opisnik Aktivna_D;
    ako je (Osem[J].v >= 1) {
        Osem[J].v = Osem[J].v - 1;
        obnoviti kontekst iz opisnika Aktivna_D;
        omogućiti prekidanje;
        v atiti se iz prekidnog načina;
    }
    inače {
        premjestiti opisnik iz reda Aktivna_D u red Osem[J];
        aktivirati prvu dretvu iz reda Pripravne_D;
    }
}
```

U ovoj se izvedbi semafora njegova vrijednost smanjuje samo onda ako je veća ili jednaka jedan. Prema tome, ona ne može postati negativna. U slučaju kada neka dretva pokuša proći semafor čija je vrijednost jednaka nuli, ona će se na semaforu blokirati.

Funkcija postavljanja semafora ima sljedeći oblik:



```
j-funkcija Postaviti_Osem(J) {
    pohraniti kontekst u opisnik Aktivna_D;
    premještiti opisnik iz reda Aktivna_D u red Pripravne_D;
    ako je (red Osem[J] neprazan) {
        premještiti prvi opisnik iz reda Osem[J] u red Pripravne_D;
    }
    inače {
        Osem[J].v = Osem[J].v + 1;
    }
    aktivirati prvu dretvu iz reda Pripravne_D;
}
```

Ovaj se semafor može nazvati i brojačkim semaforom. Naime, ako mu se pripiše neka početna vrijednost $N > 0$, on će dopustiti da N dretvi ispita njegovu vrijednost i da se tek sljedeća blokira. Isto tako, jedna ciklička dretva mogla bi obaviti svojih N ciklusa i nakon toga se blokirati. Prema tome, ovakav je semafor prikladan za brojenje / kontrolu / trošenje ograničenog broja sredstava računalnog sustava ili za brojenje nekih događaja u sustavu. Stoga je i dobio naziv *brojački semafor*, a nazivaju ga i *brojilom događaja* (engl. *counting semaphore* ili *event counter*).

U nedostatku binarnog semafora i opći se semafor može iskoristiti za međusobno isključivanje ako mu je početna vrijednost postavljena na jedan te je osigurano *naizmjenično pozivanje* funkcija *Ispitati_Osem* i *Postaviti_Osem*, počevši s navedenim redoslijedom.

5.3.4. Funkcije za ostvarivanje kašnjenja

Pri opisivanju strukture podataka naše jezgre ustanovili smo na koji se način oblikuje red odgodenih dretvi. Rekli smo da dretva može zatražiti da sama bude odgodena za M perioda otkucaja sata. U tom slučaju ona će pozvati jezgrinu funkciju *Zakasniti_sebe(M)*. Ta funkcija može izgledati ovako:

```
j-funkcija Zakasniti_sebe(M) {
    pohraniti kontekst u opisnik Aktivna_D;
    uvrstiti opisnik iz reda Aktivna_D u red Odgodene_D;
    aktivirati prvu dretvu iz reda Pripravne_D;
}
```



Aktivna dretva može zatražiti i odgodeno izvođenje neke druge dretve s indeksom L i u tom slučaju ona bi mogla pozvati ovaku funkciju:

```
j-funkcija Zakasniti(L,M) {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u red Pripravne_D;
    pronaći opisnik dretve L u listi Postojeće_D;
    ako je (dretva L nije pasivna) {
        dojaviti pogrešku;
    }
    inače {
        uvrstiti opisnik dretve L u red Odgodene_D;
    }
    aktivirati prvu dretvu iz reda Pripravne_D;
}
```



Funkcija smije odgođeno pokrenuti neku dretvu samo ako je ona pasivna. Zbog toga se pokušaj pokretanja dretve koja nije bila pasivna mora smatrati pogreškom, pa se to posebno dojavljuje.

Deblokiranje dretvi koje se nalaze u redu Odgođene_D obaviti će funkcija koja se poziva prekidom od sata (ili drukčije rečeno: koja obrađuje prekid od sata). Ona to može obaviti na sljedeći način:



```

j-funkcija Otkucaj_sata {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u red Pripravne_D;
    ako je (red Odgođene_D neprazan) {
        umanjiti u prvom opisniku reda Odgođene_D sadržaj lokacije
        Zadano_kašnjenje za jedan;
        ako je (rezultat smanjenja jednak nuli) {
            premjestiti prvi opisnik iz reda Odgođene_D u red Pripravne_D
            (a i sve sljedeće opisnike ako u lokaciji
            Zadano_kašnjenje imaju zapisanu vrijednost nula);
        }
    }
    aktivirati prvu dretvu iz reda Pripravne_D;
}

```

Ako više od jedne dretve treba pokrenuti odgođeno u isto vrijeme, onda se pri uvrštenju njihovih opisnika u red Odgođene_D samo u prvom može zapisati zadano kašnjenje, a drugima se pohranjuje razlika kašnjenja jednaka nuli, kao što smo ustanovili pri opisu strukture podataka jezgre.

5.3.5. Funkcije za obavljanje ulazno-izlaznih operacija

Dretva koja želi obaviti neku ulaznu ili izlaznu operaciju ne može to obaviti neposredno. Ona mora pozvati odgovarajuću jezgrinu funkciju. U pozivu te funkcije ona će specificirati vrstu operacije i adrese izlaznih podataka, odnosno adrese na koje treba smjestiti ulazne podatke. Prema tome, poziv jezgrine funkcije kojim započinje neka ulazno-izlazna operacija rezultirat će sljedećom aktivnošću jezgre:



```

j-funkcija Započeti_UI(K) {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u red UI[K];
    pokrenuti ulazno-izlaznu operaciju na napravi K;
    aktivirati prvu dretvu iz reda Pripravne_D;
}

```

Mi u našem jednostavnom modelu jezgre prepostavljamo da dretva koja je pokrenula ulazno-izlaznu operaciju mora čekati dok se ta operacija ne obavi. Zbog toga opisnik dretve biva blokiran u redu UI [K]. Instrukcija pokrenuti ulazno-izlaznu operaciju na napravi K može izazvati vrlo složenu operaciju prenošenja podataka pozivajući i posebne potprograme koji upravljaju specificiranim napravom. Na taj način mogu biti pokrenute i posebne dretve koje obavljaju posao prenošenja podataka. U svakom slučaju mi očekujemo da će naprava javiti prekidom da je zatraženi posao obavljen.

Prekid od ulazno-izlazne naprave pokrenut će jezgrinu funkciju koju smo nazvali Prekid_UI(K). Ta funkcija obavlja sljedeći posao:

```
j-funkcija Prekid_UI(K) {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u red Pripravne_D;
    premjestiti opisnik iz reda UI[K] u red Pripravne_D;
    aktivirati prvu dretvu iz reda Pripravne_D;
}
```



Funkcija najprije pohranjuje kontekst dretve koja se izvodila u procesoru (i nema neposredne veze s tim prekidom) i njezin opisnik premješta u red Pripravne_D. Nakon toga deblokira dretvu time što premješta opisnik dretve iz reda UI[K] također u red Pripravne_D te završava aktiviranjem dretve koja je prva na redu za izvođenje.

5.4. Ostvarenje jezgre u čvrsto povezanom višeprocesorskom sustavu

U dosadašnjem smo opisu jezgre prepostavljali da se dretve odvijaju u jednoprocesorskom sustavu i da se jezgrine funkcije, zbog načina njihova pozivanja sklopovskim ili programskim prekidom, odvijaju međusobno isključeno. Znamo da u višeprocesorskom sustavu zabrana prekidanja u jednom procesoru ne osigurava međusobno isključivanje.

Sada ćemo prepostaviti da se sustav dretvi odvija u čvrsto povezanom višeprocesorskom sustavu koji se može izvesti iz onog predstavljenog slikom 3.15. i koji se ponaša u skladu s opisom iz odjeljka 3.5. Podsetimo se da tamo svi procesori mogu adresirati svaki svoj lokalni spremnik i jedan zajednički dijeljeni spremnik.

Pri razmatranju spremničkog prostora sustava dretvi koje se sve izvode unutar jednog procesa mi smo do sada prepostavljali da sve dretve mogu dohvaćati cijeli procesni spremnički prostor.

U višeprocesorskom sustavu možemo prepostaviti razne načine uporabe spremničkog prostora, pa možemo:

- zanemariti postojanje lokalnih spremnika i pretpostaviti da se svi dretveni prostori i zajednički prostor nalaze u dijeljenom spremniku;
- pretpostaviti da su samo instrukcije dretvi smještene u lokalne spremnike, a da se stogovi dretvi i njihovi lokalni podaci i zajednički prostor nalaze u dijeljenom spremniku ili
- pretpostaviti da se cijeli dretveni prostori nalaze u pojedinim lokalnim spremnicima i da se u dijeljenom spremniku nalazi samo zajednički prostor svih dretvi.

S obzirom na te raznovrsne mogućnosti podjele prostora može se posao u višeprocesorskom sustavu organizirati tako da se sve dretve mogu izvoditi u bilo kojem od procesora (tada govorimo o *homogenom* višeprocesorskem sustavu) ili da se pojedine dretve mogu izvoditi samo u određenom procesoru (tada govorimo o *nehomogenom* višeprocesorskem sustavu). U homogenom sustavu može se, dakle, izvođenje dretve prepustiti bilo kojem od procesora, što olakšava njihovo raspoređivanje. U nehomogenom sustavu mora postojati mogućnost pridjeljivanja dretvi pojedinom od procesora. Za pojedine vrste poslova može se naći prednosti ili jednog ili drugog načina raspodjela dretvi. Ovdje nećemo detaljnije razmatrati taj problem.

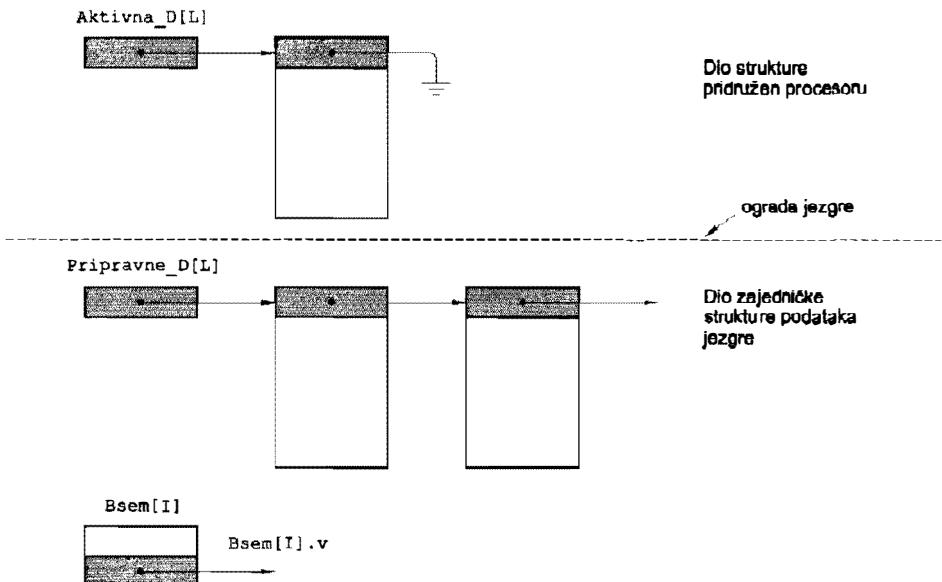
Međutim, struktura podataka jezgre, bez obzira na način uporabe procesorâ, mora se nalaziti u dijeljenom spremniku. Ona mora biti dohvatljiva svim dretvama bez obzira na to u kojim se procesorima one izvode. Do te strukture jezgrine funkcije moraju pristupati međusobno isključeno. U 4. smo poglavljju naučili da zabrana prekidanja pojedinoga procesora nije dovoljna za osiguranje međusobnog isključivanja, ali i da instrukcije tipa "ispitati i postaviti" rješavaju taj problem.

Uz strukturu podataka jezgre stoga ćemo uvesti još jednu varijablu – nazovimo je **DGRA_DA_JEZGRE** – koja će nam poslužiti kao zastavica koju na svom početku treba ispitivati svaka jezgrina funkcija.

Mi ćemo, samo kao ilustraciju mogućnosti ostvarenja jezgre u višeprocesorskem sustavu razmotriti par funkcija za binarni semafor. Na slici 5.13. prikazan je dio strukture podataka jezgre koji će nam olakšati objašnjenje djelovanja funkcija za binarni semafor. Pretpostavit ćemo da imamo nehomogeni sustav i da se svaka dretva izvodi na svom procesoru. Svaki od procesora imat će svoj:

- red **Aktivna_D[L]** i
- red pripravnih dretvi **Pripravne_D[L]**.

Kada neka dretva iz procesora L poziva jezgru, zabrana prekidanja djeluje samo na taj procesor. Jedina aktivnost koja se može obaviti bez opasnosti tim procesorom samostalno je pohranjivanje konteksta u opisnik koji se nalazi u redu **Aktivna_D[L]**. Svaki daljnji zahvat mora se zaštititi ispitivanjem zastavice **DGRADA_JEZGRE**. U red pripravnih dretvi za procesor L, iako je on pridružen samo tom procesoru, ne smije se pristupati nezaštićeno. Naime, u taj red mogu stavljati opisnike i neke jezgrene funkcije koje su pozivane iz drugih procesora.



Slika 5.13. Dio strukture podataka jezgre višeprocesorskog sustava

Razjasnit ćemo to na primjerima funkcija za binarni semafor. Osim indeksa binarnog semafora I koji želimo ispitati, u pozivu funkcije pojavljuje se i indeks procesora L odakle je poziv upućen. Iako se taj indeks nalazi i u opisniku dretve, ovdje je on posebno istaknut zbog jednostavnijeg objašnjavanja osnovne zamisli djelovanja funkcije. Funkcija ispitivanja semafora mogla bi izgledati ovako:

```
j-funkcija Ispitati_Bsem(I,L) {
    pohraniti kontekst u opisnik Aktivna_D[L];
    "ispitati i postaviti" varijablu OGRADA_JEZGRE;
    dok je (OGRADA_JEZGRE != 0) {
        "ispitati i postaviti" varijablu OGRADA_JEZGRE;
    }
    ako je (Bsem[I].v == 1) {
        Bsem[I].v = 0;
    }
    inače {
        premjestiti opisnik iz reda Aktivna_D[L] u red Bsem[I];
        premjestiti prvi opisnik iz reda Pripravne_D[L]
        u red Aktivna_D[L];
    }
    OGRADA_JEZGRE = 0;
    obnoviti kontekst iz opisnika Aktivna_D[L] u procesoru L;
    omogući prekidanje u procesoru L;
    vratiti se iz prekidnog načina rada u procesoru L;
}
```

Napomenimo da prolaz kroz ogradije jezgre može izazvati radno čekanje procesora L (za takvo se radno čekanje u ovom kontekstu koristi engleski termin *spinlock*). To se zbiava samo onda ako neki drugi procesor izvodi jezgrinu proceduru, i to samo za vrijeme trajanja te procedure. Zbog toga jezgrine funkcije ne smiju biti dugotrajne (to vrijedi i za jezgrine funkcije jednoprocесorskog sustava). Iz opisa gornje funkcije vidljivo je da se iz ogradije jezgre izlazi tako da se u zastavicu OGRADA_JEZGRE pohranjuje vrijednost 0 čim je to moguće. Ako je binarni semafor prolazan, onda će funkcija samo zapisati u Bsem[I].v vrijednost nula i odmah spustiti zastavicu jer vraćanje konteksta iz opisnika koji se nalazi u redu Aktivna_D[I] više nije u kritičnom odsječku. Međutim, ako je semafor neprolazan, onda se još u kritičnom odsječku mora premjestiti opisnik dretve koja je pozvala funkciju i bit će blokirana u red Bsem[I], i nakon toga premjestiti prvi po redu opisnik iz reda Pripravne_D[I] u red Aktivna_D[I], i tek nakon toga se izlazi iz ogradije jezgre. Uočimo da blokirana dretva na binarnom semaforu ne uzrokuje radno čekanje. Procesor L obavlja neku drugu dretvu (ako takva dretva postoji!).

Nakon što je dretva blokirana, neka druga dretva koja je prije toga prošla kroz semafor može iz drugog procesora M pozvati funkciju za postavljanje tog semafora. Ta bi funkcija mogla izgledati ovako:

```
j-funkcija Postaviti_Bsem(I,M) {
    pohraniti kontekst u opisnik Aktivna_D[M];
    "ispitati i postaviti" varijablu OGRADA_JEZGRE;
    dok je (OGRADA_JEZGRE != 0) {
        "ispitati i postaviti" varijablu OGRADA_JEZGRE;
    }
    premjestiti opisnik iz reda Aktivna_D[M] u red Pripravne_D[M];
    ako je (red Bsem[I] neprazan) {
        premjestiti prvi opisnik iz reda Bsem[I] u red Pripravne_D[L];
    }
    inace {
        Bsem[I].v = 1;
    }
    premjestiti prvi opisnik iz reda Pripravne_D[M] u red Aktivna_D[M];
    OGRADA_JEZGRE = 0;
    obnoviti kontekst iz opisnika Aktivna_D[M] u procesoru M;
    omogućiti prekidanje u procesoru M;
    vratiti se iz prekidanog načina rada u procesoru M;
}
```

Iz ovog je opisa vidljivo da funkcija koja je pozvana iz procesora M djeluje na red pripravnih dretvi procesora L zbog toga što deblokira dretvu koja pripada procesoru L. Zbog toga manipuliranje s redovima pripravnih procesa treba obaviti u kritičnom odsječku – unutar ogradije jezgre.

Na sličan način moraju biti izgrađene i sve druge jezgrine funkcije višeprocesorskih sustava.

Iako je u ovom primjeru pretpostavljen nehomogeni sustav, i u homogenim je sustavima uobičajeno da se redovi pripravnih dretvi organiziraju po procesorima. Razlog je u učinkovitom iskoristenju priručnog spremnika. Naime, moguće je da se dijelovi dretve (i instrukcije i podaci) zadrže u priručnom spremniku procesora između dvaju uzastopnih aktiviranja te iste dretve pa se time smanjuje komunikacija s glavnim (i sporijim) spremnikom (u ovom kontekstu ova se pojava spominje kao "hot cache"). Naravno, jezgra operacijskog sustava mora voditi računa o pravednoj raspodjeli računalnog vremena među svim pripravnim dretvama te će ih možda povremeno morati i premještati s jednog procesora na drugi.

5.5. Objektni model jezgre operacijskog sustava

Suvremeni pristup izgradnji složenih programskih sustava zasniva se na objektnom pristupu. Objekt se sastoji od strukture podataka i od funkcija koje djeluju na te podatke. Odaziv objekta na pojedine funkcije ovisi o trenutačnom unutarnjem stanju objekta. Podsetimo se što smo u odjeljku 1.2. prvog poglavlja rekli o načelu hijerarhijske izgradnje operacijskog sustava. Jezgra operacijskog sustava jedna je od razina hijerarhijske izgradnje. Opisana struktura podataka jezgre i načini ostvarenja pojedinih jezgrinih funkcija mogu biti skrivene višim razinama programa koji će te funkcije koristiti.

U našem dosadašnjem opisu jezgre susretali smo različite objekte ili još bolje rečeno *klase objekata* za koje su definirane neke operacije. Spominjali smo primjerice klase procesa, dretvi, binarnih i općih semafora. Do pojedinih objekata određene klase možemo doći preko kazaljki ili simboličkih identifikatora – imena. Tako se svi objekti dretvi mogu dohvatiti preko zaglavlja `Postojeće_D`. Svaka pojedina dretva jedan je konkretni objekt klase dretvi. Dretve kao objekti jedne klase mogu koristiti objekte drugih klasa pri svom obavljanju. Tako dretve mogu upotrebljavati objekte iz klase binarnih semafora za međusobno isključivanje i objekte iz klase općih semafora za sinkronizaciju ili brojenje nekih događaja.

Objekti jezgre ostvaruju se unutar sustavskog adresnog prostora. Oni tamo skriveno izgraduju svoje strukture podataka i u tim strukturama pohranjuju svoja stanja. Izvan jezgre poznati su nam samo načini pokretanja pojedinih operacija nad tim objektima. Osim osnovnih operacija kojima se obavljaju osnovne funkcije objekata (koje smo mi za naš model jezgre u ovom poglavlju izučavali) u stvarnim operacijskim sustavima postoje operacije za:

- kreiranje (stvaranje) pojedinačnih objekata pojedine klase;
- dovođenje objekata u početno stanje (inicijalizaciju objekata);
- pregledavanje stanja objekata;
- uništavanje objekata.

Objekti jezgre operacijskog sustava skrivaju sklopovske detalje odvijanja dretvi u računaru i omogućuju nam da na ujednačeni način gledamo na različite sklopovske mehanizme.

Mi smo u prethodnim poglavljima vidjeli da bi čak i u vrlo pojednostavljenim modelima sklopolja programiranje na strojnoj razini bilo vrlo mučno i potpuno nedjelotvorno. Pokazat će se da objekti jezgre omogućuju izgradnju dalnjih razina operacijskog sustava koje će ostvarivati prikladna sučelja za pripremu korisničkih programa.

Opširnije ćemo se na izgradnju suvremenih operacijskih sustava osvrnuti na kraju sljedećega poglavlja, nakon što razmotrimo još neke načine ostvarivanja sustavskih funkcija.



PITANJA ZA PROVJERU ZNANJA 5

1. Što predstavlja pojam "ulazak u jezgru" i kada se zbiva?
2. Na što se svodi "izlazak iz jezgre"?
3. Navesti izvore prekida u jednostavnom modelu jezgre.
4. Od čega se sastoji jezgra operacijskog sustava?
5. Navesti sadržaj opisnika dretve.
6. Navesti strukture podataka jezgre.
7. Koja su blokirana stanja dretvi?
8. Skicirati graf mogućih stanja dretvi.
9. Što obavlja instrukcija "aktivirati prvu dretvu iz reda Pripravne_D"?
10. Što obavlja instrukcija "vratiti se iz prekidnog načina"?
11. Čemu služe jezgrini mehanizmi binarni i opći semafor (BSEM i OS)?
12. Koje strukture podataka koriste BSEM, OS i OSEM?
13. U pseudokodu napisati jezgrine funkcije Čekaj_BSEM, Postavi_BSEM, Čekaj_OS, Postavi_OS, Čekaj_OSEM i Postavi_OSEM.
14. Opisati način umetanja opisnika dretve u listu Zakašnjele_D. Koja se vrijednost upisuje u polje Zadano_kašnjenje u opisniku dretve?
15. Koje vrste prekida uzrokuju jezgrine funkcije Započeti_UI i Prekid_UI u jednostavnom modelu jezgre?
16. Može li se prekinuti dretva koja obavlja neku jezgrinu funkciju?
17. Na koji se način jezgrine funkcije obavljaju međusobno isključivo na jednoprocesorskom računalu, a kako na višeprocesorskom računalu?

6.

Međudretvena komunikacija i konцепција monitora

6.1. Problem proizvođača i potrošača

U višedretvenom sustavu dretve mogu surađivati na zajedničkom poslu tako da razmjenjuju podatke preko zajedničkih lokacija u dijeljenom spremniku. Sinkronizacijski mehanizmi opisani u prethodnome poglavlju omogućuju postizanje ispravnog redoslijeda pristupa međusobno zavisnih dretvi takvim podacima. S obzirom na to da već i najmanja pogreška pri osmišljavanju i ostvarivanju takvih sustava može izazvati velike neželjene posljedice, pokazalo se korisnim već i u same mehanizme razmjene podataka između dretvi ugraditi sinkronizacijske mehanizme.

U višedretvenom sustavu koji smo razmatrali u odjeljku 4.2. ustanovili smo da bi se mnogi programski zadaci koje provodimo u računalu mogli prikazati svojevrsnom cjevovodnom organizacijom posla u kojoj cikličke dretve jedna drugoj predaju dijelove podataka na daljnju obradu (vidi sliku 4.6.). Model predaje podataka od strane jedne dretve drugoj naziva se u literaturi problemom proizvođača i potrošača (engl. *Producer/Consumer Problem*).

Proizvođač je ciklička dretva u kojoj se generiraju podaci, ti se podaci uobičaju u obliku *poruke* i šalju potrošaču. *Potrošač* je također ciklička dretva koja čeka na poruke, prihvata poruku i zatim podatke prispjele u poruci potroši. Mi ćemo skraćeno govoriti da proizvođač proizvodi, a potrošač troši poruke.

Cikličku dretvu proizvođača možemo dakle opisati ovako:

```
dok je (1) {  
    proizvesti poruku;  
    poslati poruku;  
}
```



a cikličku dretvu potrošača ovako:

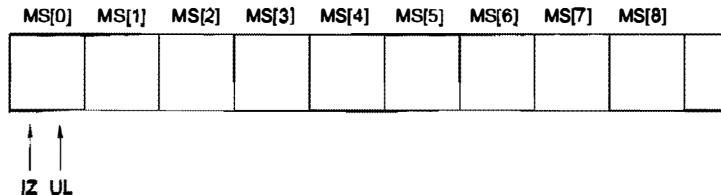


dok je (1) {
 čekati na poruku;
 prihvatići poruku;
 potrošiti poruku;
 }

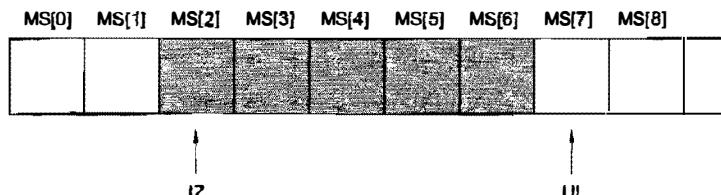
U općem slučaju treba pretpostaviti da proizvođač i potrošač obavljaju svoj posao proizvoljnom brzinom. To znači da mogu postojati vremenski intervali kada će proizvođač proizvoditi mnogo poruka, koje potrošač neće moći odmah nakon njihova nailaska potrošiti. Takve poruke treba privremeno pohraniti kako bi ih proizvođač u vremenskom intervalu smanjene proizvodnje poruka mogao potrošiti. Moramo, naime, pretpostaviti da će u dužem vremenskom razdoblju prosječna proizvodnja poruka biti takva da ih potrošač sve može potrošiti. Za pohranjivanje poruka poslužit će nam međuspremnik u čije se pretince poruke mogu pohraniti. Taj bismo međuspremnik mogli smjestiti u spremnički prostor procesa kojem pripadaju obje naše dretve¹.

6.1.1. Međudretvena komunikacija s pomoću neograničenog spremnika

Međuspremnik možemo zamisliti kao poredak pretinaca u koji pristaju oblikovane poruke (u najjednostavnijem slučaju pretinac može biti veličine jednog bajta ako se poruke sastoje od pojedinačnih znakova). Pretinice ćemo imenovati kao MS[1] kao što je to ilustrirano slikom 6.1.

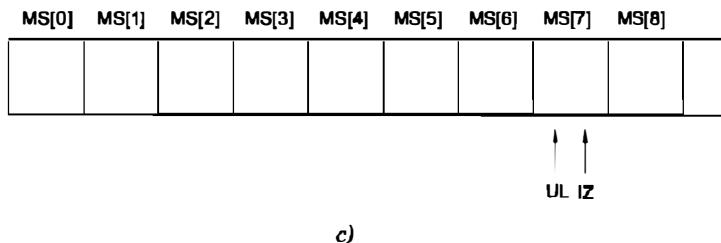


a)



b)

¹ Meduspremnik se u engleskom jeziku zbog svog svojstva ublažavanja trenutačnog preopterećenja porukama naziva *buffer* — ublaživač udaraca. Ovdje ćemo pretpostaviti da je meduspremnik smješten u spremnički prostor procesa kojem obje dretve pripadaju. U 10. poglavljiju razmatrat ćemo slučaj kada dretve ne pripadaju istom procesu.



Slika 6.1. Neograničeni spremnik

Prepostaviti ćemo da broj pretinaca nije ograničen. Uvest ćemo dva indeksa (ili: kazaljke) za dohvaćanje pretinaca, i to:

- indeks UL koji pokazuje na prazni pretinac u koji potrošač treba smjestiti sljedeću poruku i
- indeks IZ koji pokazuje na pretinac iz kojeg proizvođač upravo treba preuzeti poruku.

Na početku obje kazaljke pokazuju na pretinac $MS[0]$, tj. $UL == 0$ i $IZ == 0$. Slika 6.1.a) prikazuje početno stanje praznog međuspremnika.

Proizvođač uvijek pronalazi mesta u međuspremniku i može se ponašati ovako:

```
dok je (1) {
    proizvesti poruku P;
    MS[UL] = P;
    UL = UL + 1;
}
```



Međutim, potrošač može trošiti poruku tek nakon što neka poruka bude stavljena u spremnik. Drugim riječima, on može preuzeti poruku iz međuspremnika u svoj lokalni spremnik R tek kada ustanovi da je ispunjen uvjet $UL > IZ$. Prema tome, potrošač bi trebao izgledati ovako:

```
dok je (1) {
    pričekati da bude ispunjen uvjet UL > IZ;
    R = MS[IZ];
    IZ = IZ + 1;
    potrošiti poruku R;
}
```



Proizvođač može nesmetano, proizvoljnom brzinom, stavljati poruke u međuspremnik jer broj pretinaca nije ograničen. Potrošač će svojom vlastitom brzinom uzimati iz međuspremnika poruke onda kada ih bude. Na slici 6.1.b) prikazano je stanje međuspremnika nakon što je proizvođač u njega stavio sedam poruka, a potrošač potrošio dvije poruke.

U međuspremniku se nalazi još pet nepotrošenih poruka. Proizvođač će svoju sljedeću poruku staviti u pretinac MS [7] jer je UL == 7, a potrošač mora preuzeti sljedeću poruku iz pretinca MS [2] jer je IZ == 2.

Ako sada proizvođač zastane i jedno vrijeme ne proizvodi poruke, onda će potrošač uspjeti potrošiti sve poruke iz međuspremnika i pomicat će svoju kazaljku tako da će ona poprimiti vrijednost IZ == 7. Kada se to dogodi, u međuspremniku više neće biti poruka i međuspremnik će doći u stanje prikazano slikom 6.1.c). Kazaljke UL i IZ nisu jednake nuli, ali je njihova razlika jednaka nuli. Vidimo da je broj poruka u spremniku uvek jednak razlici UL - IZ. Ta se razlika povećava za jedan pri stavljanju svake poruke u spremnik i smanjuje za jedan pri svakom uzimanju poruke iz međuspremnika.

Pokazuje se da bi za brojenje poruka u spremniku mogao poslužiti brojački semafor opisan u odjeljku 5.3. Pri pokretanju višedretvenog sustava trebalo bi dakle uz međuspremnik MS te uz kazaljke UL i IZ postavljene u početnu vrijednost 0, uvesti i opći semafor, nazovimo ga Osem[1], s početnom vrijednošću Osem[1].v == 0. Proizvođač mora postavljati taj semafor te on sada izgleda ovako:

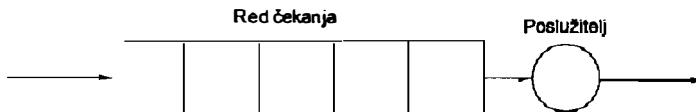
```
dok je (1) {
    proizvesti poruku P;
    MS[UL] = P;
    UL = UL + 1;
    Postaviti_Osem(1);
}
```

Proizvođač ispituje semafor i može se napisati u ovom obliku:

```
dok je (1) {
    Ispitati_Osem(1);
    R = MS[IZ];
    IZ = IZ + 1;
    potrošiti poruku R;
}
```

Možemo reći da se ispred potrošača oblikuje red poruka. Proizvođač stavlja novu poruku u red, a potrošač uzima poruku iz reda i potroši je. Za njezino trošenje potrošaču je potrebno određeno vrijeme i zbog toga se može dogoditi da se u redu poruka povremeno nađe veći broj poruka, koje čekaju kako bi bile potrošene. Ovakav *red čekanja* zajedno s potrošačem, koji možemo nazvati *poslužiteljem*, čini najjednostavniji model za analizu ponašanja sustava². Takav model možemo prikazati na način predložen slikom 6.2. Simbol proizvođača možemo i izostaviti, tako da ostaje samo strelica koja simbolizira ulazak poruka. Poslužitelj je ciklička dretva potrošača koja redom uzima poruke iz reda čekanja (međuspremnika) i, nakon što ih obradi, šalje na izlaz rezultate obrade.

² Analiza sustava koji se sastoji od redova (engl. queue) i poslužitelja (engl. server) može se obavljati s pomoću postupaka teorije redova (engl. queuing theory), čije elemente spomenuti u 7. poglavljju.

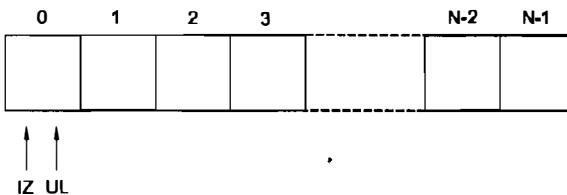


Slika 6.2. Poslužitelj s redom poruka

6.1.2. Međudretvena komunikacija s pomoću ograničenog spremnika

Ostvarenje komunikacije s pomoću neograničenog spremnika bilo bi vrlo nepraktično jer bi s vremenom kazaljke UL i IZ poprimale sve veće vrijednosti i za međuspremnik bi trebalo rezervirati veliki dio procesnog spremničkog prostora. Međutim, veliki dio tog prostora uvijek bi bio prazan. Red poruka koji će biti pohranjivan u tom se međuspremniku "pomicće u desno" tijekom vremena, ali "lijeko" od njega ostaju prazna mjesta. Ako smo sigurni da se u redu neće nikada naći više od N poruka, onda možemo međuspremnik ograničiti na N pretinaca i povećanje kazaljki izvesti po modulu N , tj. načiniti ciklički spremnik, kao što pokazuje slika 6.3. Kada kazaljke UL ili IZ dosegnu desni kraj međuspremnika, njihovo povećanje mora ih vratiti u vrijednost 0. Pomicanje kazaljki ćemo, dakle, zapisati kao:

$$\begin{aligned} \text{UL} &= (\text{UL} + 1) \bmod N; \\ \text{IZ} &= (\text{IZ} + 1) \bmod N; \end{aligned}$$



Slika 6.3. Ograničeni međuspremnik

Međutim, ako se ne može sa sigurnošću utvrditi da se ograničeni spremnik neće prepuniti, onda bi se potrošač prije stavljanja poruke u red morao uvjeriti da u međuspremniku ima praznih pretinaca. Jednako kao što potrošač mora čekati da se u praznom redu pojavi poruka, tako proizvođač mora čekati da se u punom spremniku pojavi jedan pretinac (tj. da potrošač potroši jednu poruku). To nas navodi na pomisao da posebnim brojačkim semaforom brojimo prazne pretince. Prema tome, uz semafor $\text{Osem}[1]$ koji broji poruke uvest ćemo semafor $\text{Osem}[2]$, koji će brojati prazne pretince s početnom vrijednošću $\text{Osem}[2] . v == N$. Ostale početne vrijednosti jednake su onima kod komunikacije s pomoću neograničenog spremnika.

Potrošač bi se, dakle, mogao opisati ovako:



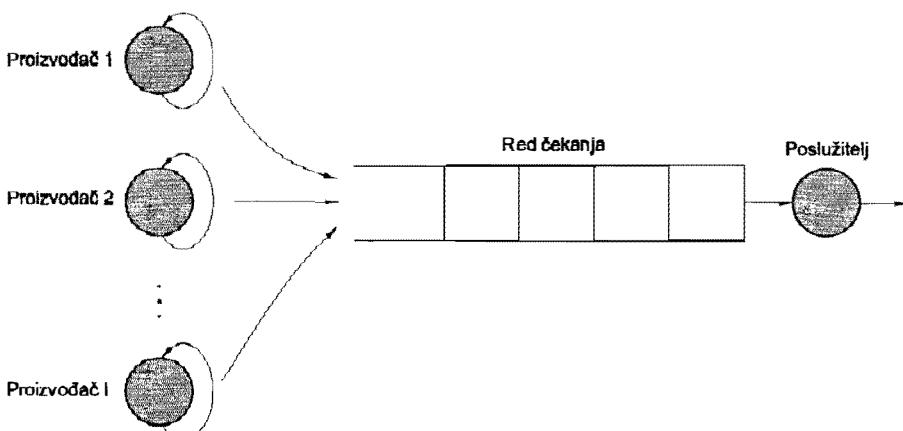
```
dok je (i) {
    proizvesti poruku P;
    Ispitati_Osem(2);
    MS[JL] = P;
    UL = (UL + 1) mod N;
    Postaviti_Osem(1)
}
```

Potrošač se mora modificirati tako da svaki put kada preuzme poruku iz međuspremnika postavlja `Osem[2]` i izgleda ovako:



```
dok je (1) {
    Ispitati_Osem(1);
    R = MS[IZ];
    IZ = (IZ + 1) mod N;
    Postaviti_Osem(2);
    potrošiti poruku R;
}
```

U prvi nam se trenutak može činiti da proizvođač i potrošač ne bi smjeli istodobno pristupati međuspremniku i da bi se taj pristup morao smatrati kritičnim odsječkom. Međutim, ovdje to nije tako jer proizvođač upotrebljava samo kazaljku `UL`, a potrošač samo kazaljku `IZ`. Prema tome, oni djeluju samo na svoje lokalne varijable. Opcim je semaforima osigurano da će oni djelovati na te svoje varijable samo onda kada je to smisleno.



Slika 6.4. Sustav s više proizvođača

Međutim, ako primjerice jednom potrošaču poruke šalje više proizvođača, kao što to prikazuje slika 6.4., onda svi oni moraju dohvaćati kazaljku UL. Za sve proizvođače pristup do kazaljke UL postaje kritični odaječak i treba ga zaštiti dodatnim binarnim semaforom. Svi proizvođači bi se morali nadopuniti tako da izgledaju ovako:

```
dok je (1) {
    proizvesti poruku P;
    Ispitati_Osem(2);
    Ispitati_Bsem(1);
    MS[UL] = P;
    UL = (UL + 1) mod N;
    Postaviti_Bsem(1);
    Postaviti_Usem(1)
}
```

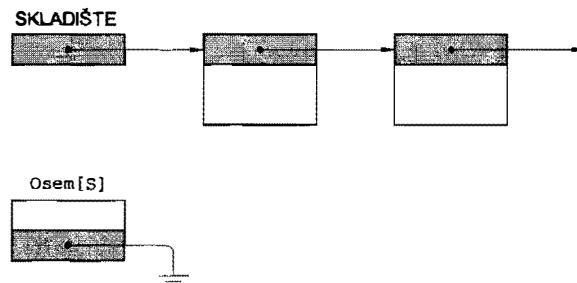


Potrošač bi ostao nepromijenjen. Kada bismo u sustav dodali i više potrošača, onda bi i za njih trebalo uvesti dodatni binarni semafor.

Ovdje već zamjećujemo neke nedostatke uvedenih jezgrinih funkcija. Dretva koja se bude odvijala u skladu s gornjim opisom morat će pozvati ukupno četiri jezgrine funkcije. Svaki taj poziv izaziva dvije promjene konteksta (jednu kod pozivanja funkcije i drugu prilikom ponovnog aktiviranja dretve). Vidjet ćemo da uporaba funkcija može imati i ozbiljnije posljedice. Ovdje zaključujemo da bi bilo dobro kada bi se s jednim pozivom jezgre moglo obaviti ispitivanje više uvjeta.

6.1.3. Međudretvena komunikacija s pomoću reda poruka

Zamislimo da u nekom višedretvenom sustavu imamo više parova proizvođača i potrošača. Za svaki taj par trebali bismo rezervirati međuspremnik s dovoljnim brojem pretinaca kako bismo smanjili mogućnost blokiranja proizvođača u razdobljima kada on intenzivno proizvodi poruke. Međutim, mala je vjerojatnost da će u istom vremenskom razdoblju baš svi proizvođači jednako intenzivno raditi. Zbog toga će u svako doba u svim tim redovima biti mnogo praznih pretinaca. To nas navodi na pomisao da redove organiziramo dinamički. Sve prazne pretince smjestit ćemo u jedno zajedničko skladište pretinaca (prepostaviti ćemo zbog pojednostavljenja opisa da su svi pretinci jednake veličine). U tom skladištu može biti mnogo manje pretinaca od zbroja svih pretinaca u statički pridijeljenim međuspremnicima. Skladište pretinaca može se organizirati kao stog (sasvim je svejedno kojim se redom prazni pretinci uzimaju iz skladišta i vraćaju u skladište!) sa zaglavljem SKLADIŠTE i jednim općim semaforom Osem[S] u koji početno treba zapisati broj pretinaca u skladištu, recimo N, kao što to prikazuje slika 6.5.

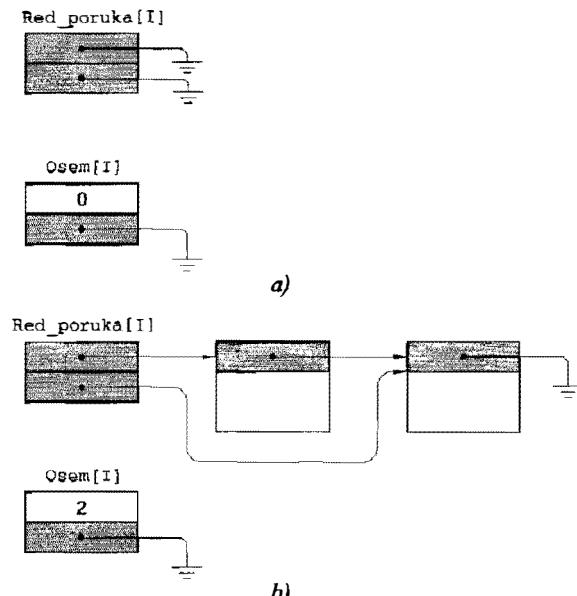


Slika 6.5. Skladište pretinaca

Za svaki par proizvođač-potrošač treba predvidjeti zaglavljje Red_poruka[I] u koji bi poruke trebalo stavljati redom prispijeća. Na slici 6.6.a) prikazano je zaglavljje praznog reda. Svakom redu je pridružen opći semafor Osem[I] koji će nam služiti za brojenje poruka u redu. U početnom stanju red je prazan i stoga je početna vrijednost Osem[I].v == 0.

Proizvođač I mora najprije dobaviti prazni pretinac iz skladišta, staviti u njega poruku i uvrstiti pretinac u red poruka. On se pri dobavljanju pretinca eventualno može blokirati na Osem[S] ako je u skladištu ponestalo pretinaca. Pri stavljanju poruke u red poruka proizvođač će povećati brojilo poruka u redu, tj. postavljati Osem[I].

Potrošač I čeka na semaforu Osem[I] ako u redu nema poruka. Ako u redu poruka ima poruka (slika 6.6.b)) prikazuje red s dvije poruke), on će preuzeti poruku iz prvog po redu pretinca i vratiti ispraznjeni pretinac u skladište. Pritom on mora povećati brojilo praznih pretinaca u skladištu, tj. postavljati semafor Osem[S].



Slika 6.6. a) Prazni red poruka, b) Red s dvije poruke

Uočimo da u ovom rješenju pristup do reda poruka i do skladišta pretinaca, čak i za samo jedan par proizvođača-potrošača, moramo obaviti kao kritični odsječak. Naime, kada u redu poruka postaje neke poruke, onda proizvođač djeluje samo na kazaljku zaglavljiva koja pokazuje na kraj reda, a potrošač na kazaljku zaglavljiva koja pokazuje na početak reda (te kazaljke podsjećaju na kazaljke UL i IZ u rješenju s ograničenim ili neograničenim međuspremnikom) i one bi se mogli smatrati privatnim kazaljkama proizvođača, odnosno potrošača. Međutim, kada se u prazni red stavlja prva poruka, proizvođač će morati djelovati na obje kazaljke, a kada potrošač iz reda uzima zadnju poruku, on će također djelovati na obje kazaljke. Zbog toga je pristup do reda poruka kritični odsječak.

Uzimanje praznog pretinca iz skladišta pretinaca koje obavlja proizvođač i vraćanje pretinaca koje obavlja potrošač također je kritični odsječak, jer obojica pritom mijenjaju kazaljku SKLADIŠTE.

Pristup do skladišta pretinaca i do reda poruka bilo bi razumno objediniti u jedan kritični odsječak i zaštiti ga binarnim semaforom, nazovimo ga Bsem[K].

Temeljem opisanih zamisli, proizvođač I mogao bi se napisati u ovom obliku:

```
dok je (I) {
    proizvesti poruku P;
    Ispitati_Osem(S);
    Ispitati_Bsem(K);
    dobaviti pretinac sa stoga SKLADIŠTE;
    staviti P u pretinac i uvrstiti ga u Red_poruka[I];
    Postaviti_Bsem[K];
    Postaviti_Osem(I);
}
```



Potrošač I mogao bi izgledati ovako:

```
dok je (I) {
    Ispitati_Osem(I);
    Ispitati_Bsem(K);
    preuzeti poruku R iz prvog pretinca u Red_poruka[I];
    vratiti ispraznjeni pretinac na stog SKLADIŠTE;
    Postaviti_Bsem[K];
    Postaviti_Osem(S);
    potrošiti poruku R;
}
```



6.1.4. Sinkronizacija dretvi

Problem proizvođača-potrošača zanimljiv je i stoga što se na njega mogu svesti ili iz njega izvesti i mehanizmi sinkronizacije. Mi smo problem sinkronizacije obradili već u poglavljiju 5. i u primjeru 5.2. pokazali kako se općeniti sustav dretvi koji se mora izvesti u skladu s utvrđenim parcijalnim uređenjem može jednostavno obaviti modificiranim općim semaforom čije vrijednosti mogu biti i negativne.

Pogledat ćemo ovdje što možemo postići modifikacijom komunikacijske ostvarene s pomoću ograničenog međuspremnika koji ima samo jedan pretinac. U tom slučaju nam kazaljke UL i IZ nisu potrebne jer bi pokazivale uvijek na isti pretinac. Nazovimo taj pretinac MS. Početna vrijednost općeg semafora koji broji prispjele poruke je $Osem[1]$. $v == 0$, a početna vrijednost semafora koji broji prazne pretince treba biti jednaka $Osem[2]$. $v == 1$.

Proizvođač bi mogao izgledati ovako:



```
dok je (1) {
    proizvesti poruku P;
    Ispitati_Osem(2);
    MS = P;
    Postaviti_Osem(1);
}
```

a potrošač se mora modificirati tako da izgleda ovako:



```
dok je (1) {
    Ispitati_Osem(1);
    R = MS;
    Postaviti_Osem(2);
    potrošiti poruku R;
}
```

Uočimo da će se ove dvije dretve obavljati naizmjence. Početno može krenuti proizvođač i, nakon što on stavi poruku u međuspremnik, može krenuti potrošač. Proizvođač mora pričekati dok potrošač ne isprazni spremnik i tek tada može u spremnik staviti svoju poruku. U načelu se isti učinak može postići i binarnim semaforom.

Ako nam nije važna poruka već samo želimo da se dvije dretve naizmjence izvode, onda se i taj jedan pretinac međuspremnika može ukloniti. Dvije cikličke dretve D_i i D_j možemo nadovezati i ciklički ih nadovezano izvoditi ako ih napišemo u sljedećem obliku:

- dretva D_i

```
dok je (1) {
    Ispitati_Osem(I);
    nešto raditi;
    Postaviti_Osem(J);
}
```

- dretva D_j

```
dok je (1) {
    Ispitati_Osem(J);
    nešto raditi;
    Postaviti_Osem(I);
}
```

Prepostavimo da su početne vrijednosti semafora $Osem[I].v == 1$ i $Osem[J].v == 0$ krenut će najprije dretva D_i , a zatim naizmjence D_j i D_i .

6.2. Potpuni zastoj

Kao što smo u prethodnim odjeljcima vidjeli, s pomoću osnovnih funkcija jezgre mogu se obaviti raznoliki mehanizmi komuniciranja između dretvi. Međutim, već smo uočili da te funkcije imaju i određene nedostatke.

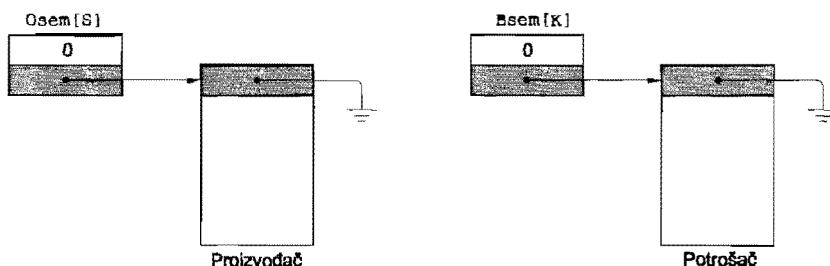
U prvom redu, funkcije obavljaju samo pojedinačna ispitivanja jednostavnih uvjeta i zbog toga se moraju, već i u sasvim jednostavnim primjenama, pretjerano pozivati. Ne zaboravimo da svako pozivanje jezgrine funkcije unosi mnogo kućanskog posla. Bilo bi stoga prikladno raspolagati s mehanizmima koji omogućuju ispitivanje i nekih složenijih uvjeta.

Drugo, nepažljivom uporabom funkcija mogu se izazvati neželjene posljedice. Pogledajmo, primjerice, što bi se moglo dogoditi pri komuniciranju između proizvođača i potrošača s pomoću reda poruka koji smo opisali u odjeljku 6.1. Ako se nepažnjom promijeni redoslijed ispitivanja binarnog i općeg semafora u dretvi proizvođača, ona bi izgledala ovako:

```
dok je (1) {
    proizvesti poruku P;
    Ispitati_Bsem(K);
    Ispitati_Osem(S);
    dobaviti pretinac sa stoga SKLADIŠTE;
    staviti P u pretinac i uvrstiti ga u Red_poruka[I];
    Postaviti_Bsem[K];
    Postaviti_Osem(I);
}
```



Dretva potrošača neka ostane nepromijenjena i neka postoji samo jedan par proizvođač-potrošač. Neka u nekom razdoblju proizvođač intenzivno proizvodi poruke, a potrošač malo zastane u njihovu trošenju. Tada se može dogoditi da u jednome trenutku ponestane praznih pretinaca u skladištu. Ako tada proizvođač pokuša poslati sljedeću poruku, on će najprije proći kroz binarni semafor $Bsem[K]$ i zatim ostati blokiran na općem semaforu $Osem[S]$ koji broji prazne pretinice. Ako u tome trenutku kreće u daljnje izvođenje potrošač, on će najprije ispitivanjem semafora $Osem[I]$ ustanoviti da u redu poruka ima pretinaca s porukama i zatim će pokušati proći binarni semafor $Bsem[K]$. Taj je semafor, međutim, neprolazan i dretva potrošača će se na njemu blokirati. Sada su i proizvođač i potrošač blokirani. U strukturi podataka jezgre opisnik dretve proizvođača smješten je u red uz opći semafor, a opisnik dretve potrošača u red uz binarni semafor $Bsem[K]$, kao što prikazuje slika 6.7.



Slika 6.7. Uzajamno blokiranje dretvi proizvođača i potrošača

Dretve ostaju ovako blokirane zauvijek. Mi kažemo da je nastao potpuni zastoj dretvi³. Dretve se ne mogu nikako deblokirati. Kada bi jedna od dretvi uspjela krenuti, ona bi deblokirala drugu.

Uočimo da je riječ o maloj pogrešci pri pisanju programa. Kada je redoslijed ispitivanja binarnog i općeg semafora onakav kao u izvornom obliku, potpuni se zastoj neće moći dogoditi.

6.2.1. Uvjeti za nastajanje potpunog zastoja

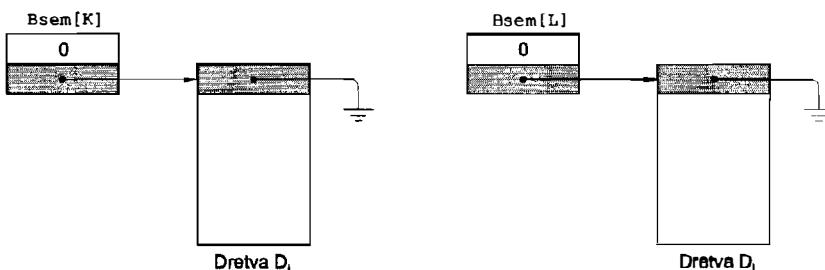
Potpuni zastoj može se dogoditi kada se najmanje dvije dretve nadmeću za najmanje dva sredstva. Prepostavimo da dvije dretve D_i i D_j za svoje izvođenje trebaju dva sredstva koja se moraju upotrebljavati međusobno isključeno. Pojedinačnu uporabu dva sredstva može se ostvariti kritičnim odsječcima zaštićenim binarnim semaforima $Bsem[K]$ i $Bsem[L]$. Prepostavimo da dretve imaju sljedeći izgled:

³ Prvi put smo taj fenomen ustanovili pri pokušavanju rješavanja problema međusobnog isključivanja u 4. poglavlju i tamo smo rekli da se on u engleskom jeziku naziva *dead lock*.



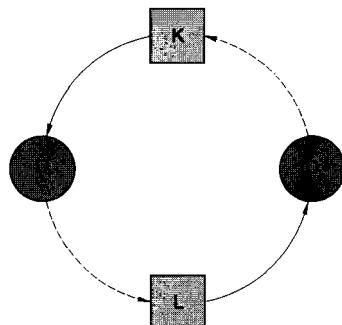
<ul style="list-style-type: none"> • dretva D_i <li style="text-align: center;">: (1) Ispitati_Bsem(K); <li style="text-align: center;">: (3) Ispitati_Bsem(L); <li style="text-align: center;">: <li style="text-align: center;">: Postaviti_Bsem(K); <li style="text-align: center;">: Postaviti_Bsem(L); <li style="text-align: center;">: 	<ul style="list-style-type: none"> • dretva D_j <li style="text-align: center;">: (2) Ispitati_Bsem(L); <li style="text-align: center;">: (4) Ispitati_Bsem(K); <li style="text-align: center;">: <li style="text-align: center;">: Postaviti_Bsem(K); <li style="text-align: center;">: Postaviti_Bsem(L); <li style="text-align: center;">:
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Tijekom istodobnog izvođenja tih dviju dretvi može se dogoditi da se pozivi jezgrinih funkcija kojima se ispituju binarni semafori obavi redoslijedom koji pokazuju brojevi u zagradama ispred pojedinog poziva. U tom će slučaju dretva D_i uspješno proći semafor $Bsem[K]$, a dretva D_j semafor $Bsem[L]$, ali će se nakon toga one obje uzajamno blokirati na tim semaforima. U strukturi podataka jezgre opisnik će se dretve D_i naći u redu semafora $Bsem[L]$, a opisnik dretve D_j u redu semafora $Bsem[K]$, kao što prikazuje slika 6.8.



Slika 6.8. Uzajamno blokiranje dviju dretvi na dvama semafora

Slikovni prikaz potpunog zastoja prikazan je na slici 6.9. Dretve su prikazane kao krugovi, a sredstva koja one traže kvadratima. Strelica od kvadrata prema krugu označava da je sredstvo dodijeljeno dretvi, a crtkana strelica od kruga prema kvadratu označava da



Slika 6.9. Slikovni prikaz petlje potpunog zastoja

dretva zahtjeva sredstvo. Na slici je vidljivo kako se dretve D_i i D_j nalaze u petlji potpunog zastaja.

U ovom bi se jednostavnom slučaju potpuni zastoj mogao izbjegći, kada bi obje dretve tražile prolaz kroz semafore jednakim redoslijedom. Prema tome, zamjenom redoslijeda dviju poziva funkcija za ispitivanje binarnih semafora ovdje bi problem bio riješen.

Međutim, taj se jednostavni uvjet ne može uvijek ispuniti. Za analizu fenomena potpunog zastaja prikidan je primjer s pet dretvi i pet sredstava kojeg je smislio E.W. Dijkstra i nazvao ga problemom pet filozofa. Cikličke dretve opisuju ponašanje filozofa koji naizmjence misle i jedu. Prema tome, ponašanje filozofa može se opisati na sljedeći način:

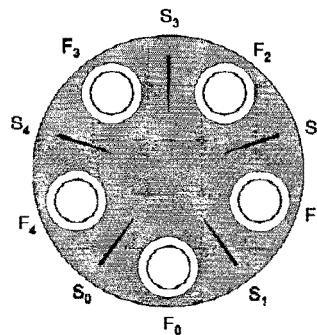
dok je (1) {

 misliti;

 jesti;

}

Filozofi jedu kinesku hranu, i to u skladu s vrlo strogim protokolom. Svaki od njih ima svoje mjesto za okruglim stolom. Na svakom mjestu stoji jedan tanjur i između njih po jedan štapić za jelo. Za jelo su potrebna dva štapića. Problem je u tome da na stolu ima ukupno samo pet štapića i da ih filozofi moraju dijeliti. Prema tome, štapići su ta oskudna sredstva koja se moraju upotrebljavati međusobno isključeno. Raspored tanjura i štapića na stolu prikazan je slikom 6.10.



Slika 6.10. Raspored sjedenja filozofa za stolom

Protokol koji svi filozofi uvažavaju pri jelu može se opisati na sljedeći način:

dok je (1) {

 misliti;

 prići k stolu;

 uzeti lijevi štapić;

 uzeti desni štapić;

 jesti;

 spustiti lijevi štapić;

 spustiti desni štapić;

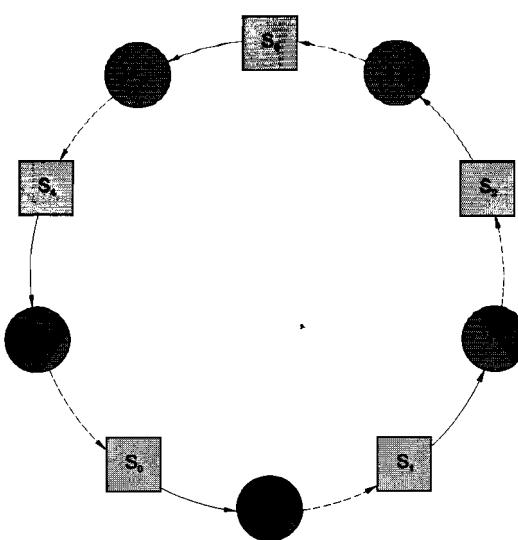
}

Uporaba štapića kritični je odsječak koji možemo ostvariti međusobnim isključivanjem s pomoću binarnog semafora. Ako filozofe obrojimo kao na slici 6.10., onda lijevi štapić filozofa F_i možemo zaštititi semaforom $Bsem[I]$, a desni štapić semaforom $Bsem[(I+1) \bmod 5]$ pa bismo ponašanje filozofa koji ima indeks I preciznije mogli opisati na sljedeći način:

```
dok je (1) {
    misliti;
    Ispitati_Bsem[I];
    Ispitati_Bsem[(I+1) mod 5];
    jesti;
    Postaviti_Bsem[I];
    Postaviti_Bsem[(I+1) mod 5];
}
```



Moglo bi se dogoditi da svi filozofi u istom trenutku ogladne, svi priđu k stolu, svi uzmu lijevi štapić (tj. prođu kraj semafora $Bsem[I]$), ali zatim više ne mogu dohvatići desni štapić (tj. blokiraju se na semaforu $Bsem[(I+1) \bmod 5]$) i tako, stojeći sa štapićem u lijevoj ruci, umru od gladi. Slikovni prikaz petljе stvorenoga potpunog zastoja prikazuje slika 6.11.



Slika 6.11. Slikovni prikaz potpunog zastoja u problemu pet filozofa

Ovdje propisani redoslijed uzimanja sredstava ne pomaže jer je lijevi štapić jednog filozofa ujedno i desni štapić njegova lijevog susjeda za stolom. Ovaj nam primjer nagoviješta da problem potpunog zastoja nije sasvim bezazlen i da ga treba ozbiljno shvaćati, te da pri zasnivanju sustava dretvi treba obratiti pozornost na zaštitu od potpunog zastoja.

Jedno od mogućih rješenja izbjegavanja potpunog zastoja proizlazi iz činjenice da potpunog zastoja neće biti ako se za stol priupusti najviše četiri filozofa. Naime, tada po načelu golubinjaka⁴ jedan od filozofa može dobiti dva štapića i potpunog zastoja neće biti. To bi se načelo moglo ugraditi u rješenje problema pet filozofa tako da se uvede jedan opći semafor koji će k stolu pripuštati najviše četiri filozofa. Nazovemo li taj semafor s `Osem[Stol]` i pripisemo li mu početnu vrijednost `Osem[Stol] . v == 4`, sljedeći opis dretve filozofa F_i rješava nas potpunog zastoja:



```
dok je (1) {
    misliti;
    Ispitati_Osem[Stol];
    Ispitati_Bsem[I];
    Ispitati_Bsem[(I+1) mod 5];
    jesti;
    Postaviti_Bsem[I];
    Postaviti_Bsem[(I+1) mod 5];
    Postaviti_Osem[Stol];
}
```

Postavlja se pitanje mogu li se postaviti neka opća načela za svladavanje problema potpunog zastoja. Mi se ovdje nećemo time opširnije baviti već ćemo ih samo kratko dotaknuti.

Iz prethodnih se primjera može shvatiti da za nastajanje potpunog zastoja postoje tri sljedeća nužna uvjeta:

- neko sredstvo u istome trenutku može upotrebljavati samo jedna od dretvi (međusobno isključivo);
- dretvi se sredstvo ne može oduzeti – ona ga otpušta sama kada ga više ne treba;
- dretva drži dodijeljeno joj sredstvo dok čeka na dodjelu dodatnog sredstva.

Pokazuje se da bi se potpuni zastoj mogao izbjegići ako se otkloni bilo koji od ta tri uvjeta. Otklanjanje prvog uvjeta nije smisленo jer je upravo on i iznjedrio problem – kada bi sredstva mogla biti istovremeno raspoređena na više dretvi, problem potpunog zastoja ne bi ni postojao. Prema tome, pri otklanjanju potpunog zastoja moralo bi se djelovati tako da se na neki način izbjegnu druga dva nužna uvjeta.

Drugi uvjet također nije jednostavno otkloniti. Naime, oduzimanje nekog sredstva koje dretva već upotrebljava može biti povezano s mnogo dodatnog posla. Podsetimo se na to da dretvama oduzimamo procesor kada se pojavi prekid i da pri oduzimanju (i kasnije ponovnoj dodjeli) procesora moramo obavljati promjenu konteksta. Slično bismo i s oduzimanjem svakog drugog sredstva također morali obavljati možda još složenije pohranjivanje i restauriranje svojevrsnog konteksta dretve na tom sredstvu.

Prema tome, kao jedno od osnovnih načela otklanjanja opasnosti od nastajanja potpunog zastoja ostaje nam pokušaj otklanjanja zadnjeg nužnog uvjeta njegova nastajanja. To se

⁴ Načelo golubinjaka primjenjuje se u kombinatorici i u svom jednostavnom obliku glasi ovako: "Ako se $n + 1$ objekt (golub) smješta u n pretinaca (odjeljka golubinjaka), onda se barem u jednom pretincu nalaze dva objekta (goluba)."

načelo svodi na to da se dretvi, uvijek kada je to moguće, sredstva ne dodjeljuju pojedinačno nego sva u isti mah. Dretvu bi pritom trebalo osmisliti tako da ona prije nego što zatraži neko dodatno sredstvo sama dragovoljno otpusti sva sredstva koja je do tada držala. Nakon toga ona mora zatražiti istodobno sva sredstva koja su joj potrebna za daljnje napredovanje.

Dosada opisane jezgrine funkcije ne omogućuju nam oživotvorene ove zamisli. Jezgru, prema tome, treba proširiti. Način takvog proširenja jezgre razmotrit ćemo u sljedećem odjeljku.

6.3. Koncepcija monitora

Iz razmatranja u prethodnom odjeljku proizlazi da su semafori vrlo korisni mehanizmi. Oni omogućuju pouzdano međusobno isključivanje kao i brojenje događaja. Međutim, oni se moraju upotrebljavati vrlo pomnivo, a neke od stvarnih problema s njima ne možemo ni riješiti.

Problemi koji nastaju uporabom tih mehanizama proizlaze zbog dvaju njihovih svojstava:

- svaki semafor ispituje samo jedan jednostavni uvjet;
- ispitivanje nekog semafora povezano je sa zauzećem sredstva koje semafor štiti. Tako nije moguće provesti ispitivanje više semafora i tek nakon toga obaviti rezervaciju sredstava tek ako se ustanovi da su sva tražena sredstva slobodna.

Te je probleme analizirao C.A.R. Hoare i predložio⁵ načine njegova svladavanja. On je ustanovio da bi bilo prikladno mehanizme suradnje dretvi (tada se govorilo o suradnji procesa) objediniti u prikladne nakupine funkcija za razrješavanje nekih cjelovitih problema. Cijelu bi takvu nakupinu trebalo nadzirati objedinjenim nadzornim programom koji je on nazvao *monitorom*.

Na opisanom primjeru pet filozofa mogle bi se napraviti monitorske funkcije *Uzeti_štapiće* i *Vratiti_štapiće* te ih koristiti u kodu dretvi prema:

```
drétvu Filozof(i) {
    dok je (1) {
        misliti;
        Uzeti_štapiće(i);
        jesti;
        Vratiti_štapiće(i);
    }
}
```



Monitor se, slično kao i jezgra, sastoji od svoje strukture podataka i funkcija koje djeluju nad tom strukturon. Izvođenje funkcija monitora ili *monitorskih funkcija* mora biti pojedinačno, tj. pozivanje funkcija je ulazak u kritični odsječak.

⁵ Hoare, C.A.R., *Monitors: An Operating Systems Concept*, Communications of the ACM, vol. 17, Oct 1974, 549–557.

Dakle, sve funkcije jednog monitora na svom početku trebale bi ispitivati binarni semafor. Međutim, za ostvarenje međusobnog isključivanja monitorskih funkcija potrebna nam je posebna vrsta semafora koji mi možemo nazvati monitorskim semaforom i označiti s **Monitor [M]**. Sve funkcije jednog monitora moraju ispitati taj semafor, a nakon svog završetka one moraju postavljati taj semafor. Kada neka dretva pozove monitorsku funkciju, ona će ili proći monitorski semafor, i mi tada kažemo da je ušla u monitor, ili će biti svrstana u red na monitorskom semaforu i za nju kažemo da čeka na ulazak u monitor. Sljedeća dretva može ući u monitor tek kada ga prethodna dretva napusti.

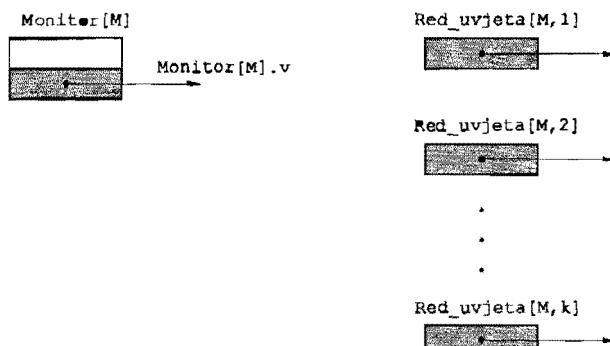
Unutar monitora dretve ispituju neke uvjete za daljnje napredovanje. Ako je uvjet koji dretva unutar monitora ispiće ispunjen, dretva napušta monitor pozivajući jezgrinu funkciju za postavljanje monitorskog semafora i time ujedno omogućuje drugoj dretvi ulazak u monitor.

Ako uvjet za nastavak izvođenja dretve koji dretva ispiće nije ispunjen, dretva se unutar monitora blokira u redu u kojem čeka na ispunjenje tog uvjeta. Blokiranje jedne dretve unutar monitora mora biti popraćeno s propuštanjem sljedeće dretve u monitor (iako blokirana dretva prividno ostaje unutar monitora).

Dretva koja je ušla u monitor može djelovati na ispunjenje nekog uvjeta tako da time deblokira dretvu koja je unutar monitora čekala na ispunjenje tog uvjeta. Zbog toga bi se moglo dogoditi da se unutar monitora nađu dvije dretve: jedna koja je upravo deblokirana i druga koja je tu deblokadu izazvala. Monitorske funkcije moraju se stoga pripremiti tako da se deblokiranje obavlja uvijek na njihovu kraju, kada bi one i tako izašle iz monitora.

6.3.1. Jezgrine funkcije za ostvarivanje monitora

Ostvarenje monitora mora biti potpomognuto prikladnim jezgrinim funkcijama. U strukturi podataka našeg modela jezgre predviđej ćemo za svaki monitor koji želimo ostvariti posebno zaglavlje reda čekanja **Monitor [M]**, a uz njega potreban broj redova čekanja na ispunjenje nekog od uvjeta. Takvih redova čekanja unutar jednog monitora može biti i



Slika 6.12. Struktura podataka jezgre koja pripada jednom monitoru

više pa ćemo ih označavati dodatnim indeksom. Tako ćemo za uvjet K monitora M upotrebljavati identifikator Red_uvjeta[M,K]. Struktura podataka jednog monitora unutar jezgre predstavljena je slikom 6.12.

U našem će nam modelu jezgre za ostvarenje monitora poslužiti četiri jezgrine funkcije:

- Ući_u_monitor(M);
- Izači_iz_monitora(M);
- Uvrstiti_u_red_uvjeta(M,K);
- Oslobođiti_iz_reda_uvjeta(M,K).

Već smo rekli da je jezgrina funkcija Ući_u_monitor(M) po djelovanju jednaka funkciji za ispitivanje binarnog semafora. Jedina razlika između njih u tome je što je Monitor(M) povezan s redovima uvjeta, što će biti vidljivo u opisu funkcija za uvrštenje i uzimanje iz redova uvjeta. Ulazak u monitor obavlja se, dakle, sljedećom jezgrinom funkcijom:

```
j-funkcija Ući_u_monitor(M) {
    pohraniti kontekst u opisnik Aktivna_D;
    ako je (Monitor[M].v = 1) {
        Monitor[M].v = 0;
        obnoviti kontekst iz opisnika Aktivna_D;
        omogućiti prekidanje;
        vratiti se iz prekidnog načina;
    }
    inače {
        premjestiti opisnik iz reda Aktivna_D u red Monitor[M];
        aktivirati prvu dretvu iz reda Pripravne_D;
    }
}
```

Izlazak iz monitora obavlja se pozivom sljedeće jezgrine procedure:

```
j-funkcija Izači_iz_monitora(M) {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u red Pripravne_D;
    ako je (red Monitor[M] neprazan) {
        premjestiti prvi opisnik iz reda Monitor[M] u red Pripravne_D;
    }
    inače {
        Monitor[M].v = 1;
    }
    aktivirati prvu dretvu iz reda Pripravne_D;
}
```

Jezgrina funkcija za blokiranje dretvi unutar monitora pozivat će se unutar neke monitorske funkcije samo onda kada dretvu treba blokirati. Uočimo da se ispitivanje tog uvjeta

obavlja u korisničkom načinu rada te da uvjet može biti i složen, a ne samo jednostavan kao što je to kod binarnog semafora. Funkcija za blokiranje mogla bi izgledati ovako:



```
j-funkcija Uvrstiti_u_red_uvjeta(M,K) {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u Red_uvjeta[M,K];
    ako je (red Monitor[M] neprazan) {
        premjestiti prvi opisnik iz reda Monior[M] u red Pripravne_D;
    }
    inače {
        Monitor[M].v = 1;
    }
    aktivirati prvu dretvu iz reda Pripravne_D;
}
```

Jezgrina funkcija za deblokiranje dretve koja je bila uvrštena u neki od redova uvjeta djelovat će tako da i dretva koja se deblokira i dretva koja je uzrokovala deblokiranje napuštaju monitor. Prema tome, monitorske funkcije morat će biti napisane tako da se deblokiranje obavlja na njihovu kraju kada bi one i tako izašle iz monitora. Svaka će monitorska funkcija morati završiti ili pozivom jezgrine funkcije za izlazak iz monitora, ili pozivom jezgrine funkcije za deblokiranje neke dretve, ili pozivom jezgrine funkcije za blokiranje pozivajuće dretve.

Jezgrina bi funkcija za oslobođanje iz reda uvjeta, prema tome, mogla izgledati ovako:



```
j-funkcija Oslobiti_iz_reda_uvjeta(M,K) {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u red Pripravne_D;
    ako je (Red_uvjeta[M,K] neprazan) {
        premjestiti prvi opisnik iz reda Red_uvjeta[M,K] u red Pripravne_D;
    }
    ako je (red Monitor[M] neprazan) {
        premjestiti prvi opisnik iz reda Monior[M] u red Pripravne_D;
    }
    inače {
        Monitor[M].v = 1;
    }
    aktivirati prvu dretvu iz reda Pripravne_D;
}
```

Naglasimo još jedanput da ovako pripremljena jezgrina funkcija zahtijeva da završetak monitorske funkcije izgleda ovako:



```

    ako je (uvjet za oslobođenje iz reda Red_uvjeta[M,K] ispunjen) {
        Oslobiti_iz_reda_uvjeta(M,K);
    }
    inače {
        Izati_iz_monitora(M);
    }
}

```

6.3.2. Primjeri izgradnje monitora

Prethodna dva para jezgrinih funkcija omogućuju nam stvaranje vlastitih sinkronizacijskih i komunikacijskih mehanizama čija svojstva možemo prilagoditi konkretnim potrebama. Pritom se mogu izbjegići i neželjene posljedice koje mogu nastati uporabom osnovnih jezgrinih funkcija (npr. semafora).

U suvremenim se operacijskim sustavima ta dva para jezgrinih funkcija (nadopunjenih s još nekoliko dodatnih funkcija) mogu smatrati najprikladnjijim oruđem za ostvarenje višedretvenih sustava. U sljedećim ćemo primjerima pogledati kako se s pomoću njih mogu ostvariti mehanizmi za rješavanje problema za koje smo do sada upotrebljavali prvočno opisane jezgrine funkcije.

Izgradnjom monitora zapravo izgrađujemo svoj objekt koji je dostupan samo preko poziva monitorskih funkcija koje ćemo sami napisati. Pritom će te monitorske funkcije upotrebljavati jezgrine funkcije predviđene za izgradnju monitora. Time monitor indirektno upotrebljava i dio strukture podataka koja će se za njega posebno izgraditi u sustavskom adresnom prostoru. Unutar monitora postoji njegova struktura podataka koja je izgrađena u korisničkom adresnom prostoru, ali je dostupna samo kroz pozive monitorskih funkcija. Monitori dolaze do svog punog izražaja upravo u objektno orientiranom načinu izgradnje programskih sustava.

Mi ćemo u sljedećim primjerima deklaracije monitorskih funkcija započinjati ključnom riječju **m-funkcija**, kako bismo ih razlikovali od običnih i od jezgrinih funkcija.

PRIMJER 6.1.

Pogledajmo kako se može izgraditi monitor koji će oponašati binarni semafor. Za njegovo je ostvarenje potrebno deklarirati jedan monitor, nazovimo ga **Monitor[1]** i jedan red uvjeta pridružen tom monitoru, tj. **Red_uvjeta[1,1]**, kao što to prikazuje slika 6.13.



Slika 6.13. Struktura podataka u jezgri za ostvarenje binarnog semafora monitorom

Za ostvarenje monitora trebat će nam još dvije varijable koje će se koristiti unutar monitora. One će nam poslužiti za opisivanje stanja unutar monitora.



Na osnovi stanja tih varijabli odreditićemo uvjete za uvrštanje dretvi u red čekanja ili oslobođanje dretvi iz reda čekanja.

Deklariratićemo varijablu Sem koja će poprimati dvije vrijednosti: vrijednost 1 označavat će da je semafor prolazan, a vrijednost 0 da je semafor neprolazan (početno neka semafor bude prolazan). Druga varijabla Broj_čekača jest cijelobrojna varijabla koja pokazuje koliko dretvi čeka na prolaz kroz semafor (početno je Broj_čekača = 0).

Trebat će nam dvije monitorske funkcije: Čekati(Sem) i Oslobođiti_prolaz(Sem). Prva od njih može izgledati ovako:



```
m-funkcija Čekati(Sem) {
    Ući_u_monitor(1);
    ako je (Sem == 1) {
        Sem = 0;
        Izaci_iz_monitora(1);
    }
    inače {
        Broj_čekača++;
        Uvrstiti_u_red_uvjeta(1,1);
    }
}
```

Funkcija za postavljanje semafora izgledat će ovako:



```
m-funkcija Oslobođiti_prolaz(Sem) {
    Ući_u_monitor(1);
    ako je ((Sem == 0) && (Broj_čekača > 0)) {
        Broj_čekača--;
        Oslobođiti_iz_reda_uvjeta(1,1);
    }
    inače {
        Sem = 1;
        Izaci_iz_monitora(1,1);
    }
}
```

Ovako pripremljene monitorske funkcije mogu se upotrijebiti za međusobno isključivanje. Pri pisanju programa neku bismo cikličku dretvu koja se s drugim takvim dretvama natjeće za ulazak u kritični odsječak mogli napisati ovako:



```
dok je (1) {
    Čekati(Sem);
    kritični odsječak;
    Oslobođiti_prolaz(Sem);
    nekritični odsječak;
}
```

PRIMJER 6.2.

Preko monitora rješavat ćemo problem komunikacije između jednog proizvodača i jednog potrošača s pomoću ograničenog međuspremnika. U tu svrhu deklarirat ćemo Monitor [2] i dva reda uvjeta. U Red_uvjeta[2,1] svrstat ćemo opisnik dretve proizvodača kada u ograničenom međuspremniku ne bude praznih mesta, a u Red_uvjeta[2,2] opisnik dretve potrošača kada u međuspremniku ne bude niti jedne poruke. Unutar monitora deklarirat ćemo još i sljedeće varijable:

- poredak MS[N] s pretincima od MS[0] do MS[N-1];
- kazaljke UL i IZ koje imaju istu ulogu kao i u primjeru komunikacije s pomoću ograničenog spremnika ostvarenog općim semaforima;
- varijabla Broj_mjesta koja pokazuje koliko ima praznih pretinaca u međuspremniku i koja se postavlja u početnu vrijednost jednaku N;
- dvije varijable Potrošač_čeka i Proizvodač_čeka koje poprimaju vrijednost 0 ako potrošač, odnosno proizvodač ne čeka ili vrijednost 1 ako proizvodač, odnosno potrošač čeka u svom redu čekanja.

Uočimo da nam nije potrebno dodatno brojilo koje bi brojilo poruke u međuspremniku. Naime, ako je Broj_mjesta < N, onda znamo da u međuspremniku ima poruka i da ih potrošač smije trošiti.

Uz pomoć uvedenih struktura podataka može se pripremiti monitorska funkcija za slanje poruka u sljedećem obliku:

```
m-funkcija Poslati_poruku(P) {
    Uči_u_monitor(2);
    dok je (Broj_mjesta == 0) {
        Proizvodač_čeka = 1;
        Uvrstiti_u_Red_uvjeta(2,1);
        Uči_u_monitor(2);
    }
    Broj_mjesta--;
    MS[UL] = P;
    UL = (UL + 1) mod N;
    ako je (Potrošač_čeka == 1) {
        Potrošač_čeka = 0;
        Oslobođiti_iz_Reda_uvjeta(2,2);
    }
    inače {
        Izaći_iz_monitora(2);
    }
}
```



Monitorska funkcija za prihvatanje poruka može imati sljedeći oblik:

```
m-funkcija Prihvati_t_poruku(R) {
    Uči_u_monitor(2);
    dok je (Broj_mjesta == N) {
        Potrošač_čeka = 1;
        Uvrstiti_u_red_uvjeta(2,2);
        Uči_u_monitor(2);
    }
    R = MS[IZ];
    IZ = (IZ+1) mod N;
    Broj_mjesta++;
    ako je (Proizvođač_čeka == 1) {
        Proizvođač_čeka = 0;
        Osloboditi_iz_reda_uvjeta(2,1);
    }
    inače {
        Izaći_iz_monitora(2);
    }
}
```

Ove monitorske funkcije izgledaju složenije od proizvođača i potrošača koje smo ostvarili s pomoću dva opća semafora, ali na ovaj način koncipirane funkcije ukazuju nam na mogućnost izgradnje raznovrsnih mehanizama suradnje dr tvi.

Uočimo način na koji je u obje funkcije riješen problem blokiranja i deblokiranja dretvi. Kada uvjet za nastavljanje dretve nije ispunjen, njezin se opisnik stavlja u odgovarajući red uvjeta. Kada se nakon toga, izvođenjem suprotne dretve, stvori uvjet za njezino deblokiranje, tada dretva ponovno ulazi u monitor i za svaku sigurnost ponovno ispituje uvjet za nastavak. U ovom jednostavnom slučaju kada postoji samo jedan proizvođač i potrošač to i nije potrebno. Međutim, ako se problem želi proširiti tako da postoji više proizvođača odnosno potrošača, onda ponovno ispitivanje uvjeta nakon deblokiranja postaje nužnost.

Cikličkog proizvođača napisat ćemo sada vrlo jednostavno. On izgleda ovako:

```
dok je (i) {
    proizvesti poruku P;
    Poslati_poruku(P);
}
```

Potrošač se, također, pojednostavljuje u sljedeći oblik:

```
dok je (i) {
    Prihvati_t_poruku(R);
    potrošiti poruku R;
}
```

PRIMJER 6.3.

Pokušajmo riješiti problem pet filozofa s pomoću monitora. Deklarirat ćemo monitor **Monitor[3]** i za svakog od filozofa I po jedan red uvjeta **Red_uvjeta[3,I]**. Osim toga, deklarirat ćemo i dva porekta **Broj_štapića[S]** i **Filozof_čeka[S]**.

Broj_štapića[I] sadržava broj štapića koje stoe uz tanjur filozofa I. Filozof ne zima štapiće pojedinačno već samo onda kada vidi oba štapića na stolu. Time će se izbjegći potpuni zastoj. Kada filozof I uzme svoje štapiće, on smanjuje broj štapića svom lijevom susjedu, tj. **Broj_štapića[(I+4) mod 5]** i desnom susjedu, tj. **Broj_štapića [(I+1) mod 5]**.

Filozof_čeka [I] == 1 označava da filozof I čeka na dodjelu štapića.

Monitorska funkcija za uzimanje štapića mogla bi izgledati ovako:

```
m-funkcija Uzeti_štapiće(I) {
    Uči_u_monitor(3);
    dok je (Broj_štapića[I] < 2) {
        Filozof_čeka[I] = 1;
        Uvrstiti_u_red_uvjeta(3,I);
        Uči_u_monitor(3);
    }
    Broj_štapića [(I+4) mod 5] --;
    Broj_štapića [(I+1) mod 5] --;
    Izaći_iz_monitora(3);
}
```

Kada filozof I ne vidi na stolu par štapića, on ne uzima niti jedan već ulazi u red čekanja. Time neće spriječiti drugoga da uzme svoje štapiće.

Monitorska funkcija za spuštanje štapića, nakon što poveća broj raspoloživih štapića svojim susjedima, treba pogledati čekaju li oni u svojim redovima i, ako sada imaju dovoljno štapića, osloboditi ih iz redova čekanja. Posebnost ove funkcije u tome je što ona može djelovati na dva reda čekanja. Uočimo da će ona morati ponovno uči u monitor nakon što debloblira dr tvu lijevog susjeda filozofa jer je pozivom jezgrine funkcije **Osloboditi_iz_Redu_uvjeta (3,J)** iz njega izbačena.

```
m-funkcija Spustiti_štapiće(K) {
    Uči_u_monitor(3);
    Broj_štapića [(K+4) mod 5]++;
    Broj_štapića [(K+1) mod 5]++;
    J = (K+4) mod 5;
    ako je ((Broj_štapića[J] == 2) && (Filozof_čeka[J] == 1)) {
        Filozof_čeka[J] = 0;
        Osloboditi_iz_Reduvjeta(3,J);
        Uči_u_monitor(3);
    }
    J = (K+1) mod 5;
```



```

ako je ((Broj_štapića[J] == 2) && (Filozof_čeka[J] == 1)) {
    Filozof_čeka[J] = 0;
    Osloboditi_iz_Reda_uvjeta(3,J);
}
inače {
    Izaći_iz_monitora(3);
}
}
}

```

Uporabom tih monitorskih funkcija ponašanje filozofa I može se opisati na sljedeći način:



```

dok je (1) {
    misliti;
    Uzeti_štapiće(I);
    jesti;
    Spustiti_štapiće(I);
}
}

```

U ovom rješenju neće se pojaviti potpuni zastoj, ali i ono ima svoje nedostatke. Naime, ako filozofi $((I+1) \bmod 5)$ i $((I+4) \bmod 5)$ naizmjence prilaze stolu i uzimaju štapiće tako da jedan uvijek uzme svoje štapiće prije nego ih drugi spusti, tada filozof I može stojeći za stolom izgladnjeti gledajući kako mu susjedi jedu. Takav fenomen, kada neke dretve bivaju odgadane u izvođenju, možemo nazvati izgladnjivanjem (engl. *starvation*).

6.3.3. Suvremenije ostvarenje monitora

U okviru sinkro izacijskih funkcija današnjih operacijskih sustava upotrebljava se inačica izvornog modela opisanog u prethodnom odjeljku. Osnovna je razlika u načinu oslobođanja blokiranih dretvi. Pokušavajući povećati učinkovitost monitorskih funkcija, jezgrina funkcija koja oslobođa dretvu iz reda uvjeta i nakon poziva ostaje u monitoru, dok oslobođena dretva prije nastavka svog rada treba ponovno ući u monitor.

U najkraćem, funkciju kojom se dretva blokira unutar monitora (nazovimo je sada Čekati_u_redu) mogli bismo opisati kao kombinaciju jezgrinih funkcija Uvrstiti_u_red_uvjeta te Ući_u_monitor. Takva bi funkcija izgledala ovako:



```

funkcija Čekati_u_redu(R, M) {
    Uvrstiti_u_red_uvjeta(R, M);
    Ući_u_monitor(M);
}
}

```

Primijetimo da funkcija `Čekati_u_redu` nije jezgrina funkcija, već je ostvarena pozivom dviju jezgrinih funkcija. Jezgrinu funkciju koja oslobađa blokirano dretvu u monitoru (nazovimo je sada `Propustiti_iz_reda`) mogli bismo ukratko opisati kao kombinaciju jezgrinih funkcija `Osloboditi_iz_reda_uvjeta` i `Uči_u_monitor`. Ova funkcija jest jezgrina funkcija koja zapravo oslobađa jednu blokirano dretvu iz reda uvjeta. Pritom dretva koja je pozvala tu funkciju ostaje unutar monitora.

Kako bismo razlikovali ovaj monitor od onog opisanog u odjeljku 6.3.1. nazvat ćemo funkcije ovog modificiranog monitora drugim imenima:

- `Zaključati_monitor(M)` [umjesto `Uči_u_monitor(M)`],
- `Otključati_monitor(M)` [umjesto `Izači_iz_monitora(M)`],
- `Zaustaviti_i_otključati_monitor(R, M)` [umjesto `Uvrstiti_u_red_uvjeta(R, M)`],
- `Propustiti_iz_reda(R)`,
- `Propustiti_sve_iz_reda(R)`.

Prve tri navedene jezgrine funkcije u potpunosti su jednake onima iz prethodno opisanog monitora:

```
j-funkcija Zaključati_monitor(M) {
    pohraniti kontekst u opisnik Aktivna_D;
    ako je (Monitor[M].v == 1) {
        Monitor[M].v = 0;
        obnoviti kontekst iz opisnika Aktivna_D;
        omo učiti prekidanje;
        vratiti se iz prekidnog načina;
    }
    inače {
        premjestiti opisnik iz reda Aktivna_D u red Monitor[M];
        tivirati prvu dretvu iz reda Pripravne_D;
    }
}

j-funkcija Otključati_monitor(M) {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u red Pripravne_D;
    ako je (red Monitor[M] neprazan)
        premjestiti prvi opisnik iz reda Monitor[M] u red Pripravne_D;
    inače
        Monitor[M].v = 1;
        aktivirati prvu dretvu iz reda Pripravne_D;
}
```

```

j-funkcija Zaustaviti_i_otključati_monitor(R, M) {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u red Red_čekanja[R];
    ako je (red Monitor[M] neprazan)
        premjestiti prvi opisnik iz reda Monitor[M] u red Pripravne_D;
    inače
        Monitor[m].v = 1;
    aktivirati prvu dretvu iz reda Pripravne_D;
}

```

Funkcija Čekati_u_redu nije jezgrina funkcija, već se sastoji od poziva dviju jezgrinih funkcija:



```

funkcija Čekati_u_redu(R, M) {
    Zaustaviti_i_otključati_monitor(R, M);
    Zaključati_monitor(M);
}

```

Jezgrine funkcije za oslobođanje iz reda uvjeta bitno se razlikuju kod ovog monitora jer dretva koja je pozvala jezgrinu funkciju i nakon povratka ostaje unutar monitora:



```

j-funkcija Propustiti_iz_reda(R) {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u red Pripravne_D;
    ako je (red Red_čekanja[R] neprazan)
        premjestiti prvi opisnik iz reda Red_čekanja[R] u red Pripravne_D;
    aktivirati prvu dretvu iz reda Pripravne_D;
}

```

Također, pokazala se potreba da se jednom jezgrinom funkcijom mogu osloboditi sve dretve iz reda te je dodana i funkcija:



```

j-funkcija Propustiti_sve_iz_reda(R) {
    pohraniti kontekst u opisnik Aktivna_D;
    premjestiti opisnik iz reda Aktivna_D u red Pripravne_D;
    ako je (red Red_čekanja[R] neprazan)
        premjestiti sve opisnike iz reda Red_čekanja[R] u red Pripravne_D;
    aktivirati prvu dretvu iz reda Pripravne_D;
}

```

Jezgrina funkcija Zaustaviti_i_otključati_monitor ne koristi se izravno već samo preko funkcije Čekati_u_redu.



PRIMJER 6.4.

Prikažimo korištenje ovih monitorskih funkcija za ostvarenje sinkronizacijskog mehanizma poznatog pod nazivom *barijera*. Kod ove sinkronizacije svaka dretva prilikom dolaska do specificiranog dijela – barijere – zaustavlja svoje dalje napredovanje dok i sve ostale dretve ne dodu do istog dijela. Tek tada sve dretve mogu nastaviti dalje (prijeći preko barijere). Monitorska funkcija za ostvarenje barijere za N dretvi može se izgraditi na ovaj način:

```
m-funkcija Barijera(m, red) {
    Zaključati_monitor(m);
    br++;
    ako je (br < N) {
        Čekati_u_redu(red, m);
    }
    inače {
        br = 0;
        Propustiti_sve_iz_reda(red);
    }
    Otključati_monitor(m);
}
```

Prema prikazanom rješenju, sve dretve osim prve blokirat će se u redu *red*. Tek će zadnja (N -ta) dretva propustiti sve blokirane dretve preko barijere.



6.4. Inverzija prioriteta

6.4.1. Mogući problemi pri sinkronizaciji dretvi

Pri rješavanju sinkronizacijskih problema mogu se pojaviti razni problemi. Osnovnu provjeru ispravnosti sinkronizacije potrebno je obaviti simuliranjem osnovnih scenarija izvođenja sustava. Iako ovakva provjera neće nužno pokazati sve probleme, neki će se sinkronizacijski problemi uvidjeti već i ovakvom jednostavnom provjerom. Najbolje bi bilo provesti formalnu provjeru ispravnosti pojedinih rješenja. Međutim, postupci formalne verifikacije vrlo su zahtjevni i obavljaju se najčešće samo za sustave u kojima se traži visoka pouzdanost.

Pri opisu pojedinih sinkronizacijskih mehanizama ukratko su opisani mogući problemi na koje treba obratiti pozornost. To su:

- problem potpunog zastoja,
- problem radnog čekanja i
- problem izgladnjivanja.

Potpuni zastoj nastaje međusobnim blokiranjem i opisan je u odjeljku 6.2. Potpuni zastoj može se manifestirati i beskonačnim ponavljanjem petlji. Jedna dretva ili više njih mogu zbog pogreške pri programiranju ponavljati izvođenje jedne petlje. Posebice je to opasno unutar kritičnog odsječka.

Problem izgladnjivanja nastaje kad jedna dretva duže vrijeme ostaje blokirana zbog načina sinkronizacije i okolnosti kod kojih ostale dretve naizmjence oduzimaju sredstva koja trebaju blokiranoj dretvi. Problem izgladnjivanja prikazali smo u primjeru 6.3. na problemu pet filozofa. Za rješavanje problema općenito je potrebno uvesti dodatnu strukturu podataka koja bilježi statistiku korištenja sredstva te na osnovu nje uvodi dodatne provjere u uvjetima napredovanja dretvi.

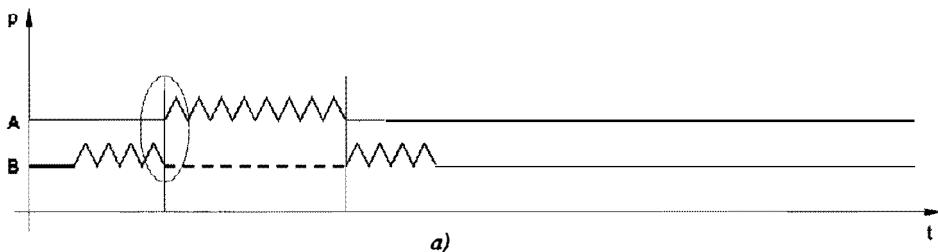
Svako korištenje dijeljenih sredstava koja se mogu promijeniti u nekoj dretvi treba zaštитiti mehanizmima međusobnog isključivanja. Također, sve variable korištene za sinkronizaciju kao i početne vrijednosti semafora trebaju biti definirane.

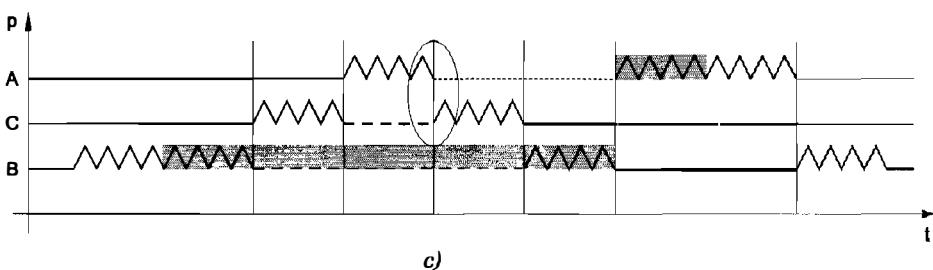
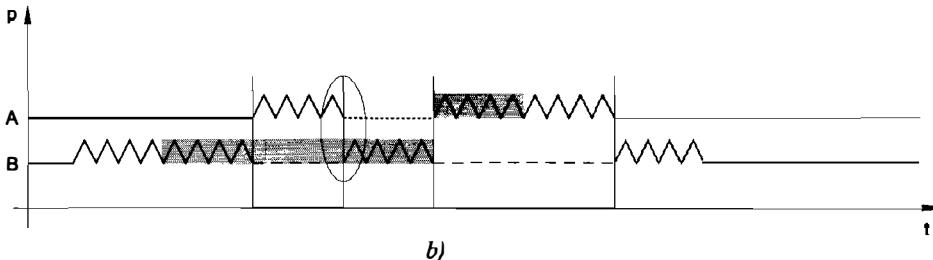
Do sada nije podrobniјe opisan problem inverzije prioriteta. Ukratko smo se na taj problem osvrnuli u primjeru 5.1. u kojem je opisan mogući scenarij zamjene prioriteta pri pozivanju više binarnih semafora. Međutim, problem nije zanemariv i potrebno ga je pomnjiće razmotriti.

Problem inverzije prioriteta

U svim sustavima, a posebice u ugrađenim i sustavima za rad u stvarnom vremenu, dretve se međusobno razlikuju po važnosti, odnosno prioritetu. Pri dodjeli procesora dretva većeg prioriteta ima prednost pred onom manjeg prioriteta. Iako je takva ili slična situacija i u ostalim sustavima, odnosno ostalim operacijskim sustavima koji nisu namijenjeni ugrađenoj uporabi, kod njih prioritet određuje koliko će pojedina dretva dobiti procesorskog vremena, a ne kada će ta dretva postati aktivna. U operacijskim sustavima za rad u stvarnom vremenu jasno je definirano da, kada dretva višeg prioriteta postaje spremna za izvođenje, ona istiskuje dretvu nižeg prioriteta koja se trenutačno izvodi. Međutim, i u takvim se sustavima događaju slučajevi kada dretva nižeg prioriteta blokira izvođenje dretve višeg prioriteta, odnosno dolazi do problema inverzije prioriteta.

Problem inverzije prioriteta nastaje kada dretva višeg prioriteta za nastavak rada treba sredstvo koje je zauzela druga dretva nižeg prioriteta. Na slici 6.14. prikazana su tri slučaja koja se mogu pojaviti u višedretvenom sustavu.




Legenda:

- dretva je neaktivna
- - - - dretva je spremna i čeka na dodjelu procesa (red pripravnih)
- dretva je u redu čekanja (čeka na sredstvo)
- ~~~~~ dretva je aktivna - izvršava se
- dretva ima sredstvo

Slika 6.14. Problem inverzije prioriteta

- dretva višeg prioriteta (A) zauzima procesor (normalna situacija),
- dretva višeg prioriteta (A) je blokirana, a dretva nižeg prioriteta (B) nastavlja s radom,
- dretva nižeg prioriteta (C) dodatno odgađa prioritetniju dretvu

Slika 6.14.a) prikazuje uobičajeno ponašanje kada u sustavu imamo dve dretve različitog prioriteta. Aktiviranjem prioritetnije dretve, dretva manjeg prioriteta se istisne, tj. makne s procesora. Na slici, čim dretva A postane spremna ona istisne dretvu B, koja ima manji prioritet.

Slika 6.14.b) prikazuje sličnu situaciju, ali dretva A i B tijekom rada koriste zajedničko sredstvo, i to međusobno isključivo (zaštićeno npr. binarnim semaforom). Dretva manjeg prioriteta B za vrijeme svog rada (dok je dretva A bila neaktivna) zauzela je sredstvo. Kasnije se dretva A aktivirala te odmah istisnula dretvu B. Međutim, u jednome trenutku dretvi A za nastavak rada treba sredstvo koje dretva B još nije otpustila (binarni semafor je neprolazan). Dretva A se blokira te dretva B ponovno nastavlja s radom. Problem koji je nastao naziva se problem inverzije prioriteta jer dretva manjeg prioriteta radi, dok dretva većeg prioriteta čeka zbog nje. Međutim, čim dretva B oslobodi zauzeto sredstvo,

dretva A se odblokira i odmah zauzima sredstvo te nastavlja s radom (npr. dretva B pozove `PostaviBSem`).

Slika 6.14.c) prikazuje situaciju kad u sustavu osim dretve A i B kao na prethodnoj slici imamo i treću dretvu C prioriteta većeg od dretve B, ali manjeg od dretve A. Kao što se vidi iz slike, ovakva dretva može dodatno odgoditi izvođenje dretve A. U sustavima sa više dretvi vrlo je teško procijeniti koliko se dretva A može odgoditi. Zato je problem inverzije prioriteta vrlo opasan za sustave za rad u stvarnom vremenu!

Metode koje se najčešće koriste u slučajevima problema inverzije prioriteta ne rješavaju sam problem nego ublažavaju njegove posljedice. To rade tako da se dretvi koja je zauzela sredstva potrebna prioritetnoj dretvi (korištenjem sinkronizacijskog mehanizma) omogući što brži rad do oslobođanja dotičnih sredstava. Dva najpoznatija takva protokola jesu protokol nasljeđivanja prioriteta (engl. *priority inheritance protocol*) i protokol stropnog prioriteta (engl. *priority ceiling protocol*). Važna pretpostavka za oba protokola je da dretve nakon što sredstvo zauzmu isto će i oslobođiti nakon nekog vremena.

Kod *protokola nasljeđivanja prioriteta* dretvi nižeg prioriteta, koja je zauzela određeno sredstvo, podiže se prioritet na razinu blokirane dretve. To se zbiva u samoj funkciji sinkronizacije koju poziva prioritetnija dretva (u pozivu koji blokira dretvu višeg prioriteta). Povećanje prioriteta treba zapamtiti u odgovarajućoj podatkovnoj strukturi da bi se kasnije omogućilo vraćanje prioriteta na prijašnju razinu. Smanjivanje prioriteta, odnosno vraćanje na razinu koju je dretva imala prije nasljeđivanja treba obaviti u funkcijama sinkronizacije, i to u funkcijama za oslobođanje sredstva.

Za *protokol stropnog prioriteta* postoje dvije inačice: originalni protokol stropnog prioriteta i pojednostavljeni protokol stropnog prioriteta. Kod obje inačice protokola svakom se sredstvu pridjeljuje stropni prioritet, tj. prioritet najznačajnijeg zadatka koji može zauzeti dotično sredstvo.

Originalni, *složeniji* protokol ima za cilj i izbjegavanje nastajanja potpunog zastoja te višestrukog blokiranja zadataka. Osnovna je ideja protokola da kada jedan zadatak pri ulasku u kritični odsječak zaustavi izvođenje kritičnog odsječka nekog drugog zadataka, tada to obavlja s prioritetom većim od stropnog prioriteta svih zaustavljenih kritičnih odsječaka. Ako kritični odsječak u koji zadatak želi ući nema dovoljno velik prioritet, ulazak mu se privremeno zabranjuje dok se taj uvjet ne ispuni, tj. dok se prije ne izvrše svi kritični odsječci s većim stropnim prioritetom. Definicija protokola koja slijedi preuzeta je iz [Rajkumar, 1991]⁶.

Zadatku *J* koji ima najveći prioritet među zadacima pripravnim za izvođenje dodjeljuje se procesor. Neka *S** označava binarni semafor najvećeg prioriteta koji je zaključan od strane nekog drugog zadataka.

- Prije nego li zadatak *J* uđe u kritični odsječak mora zaključati semafor *S* koji zaštićuje pristup zajedničkim podacima. Zadatak *J* bit će blokiran na zaključavanju semafora *S* ako prioritet zadataka *J* nije veći od stropnog prioriteta semafora *S**. U

⁶ R. Rajkumar, "Synchronization in real-time systems: A Priority Inheritance Approach", Kluwer Academic Publishers, 1991.



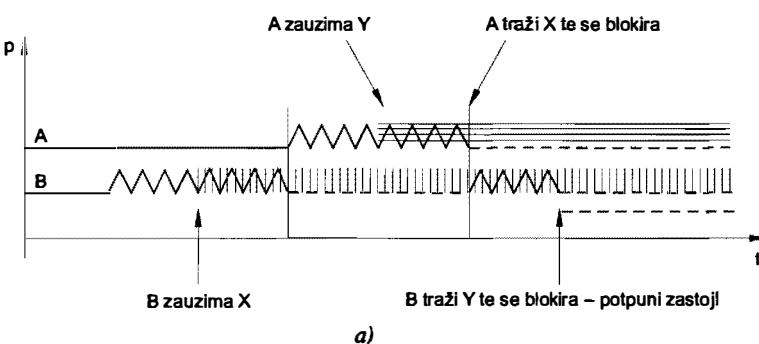
ovom slučaju kaže se da je J blokiran na semaforu S^* i da je blokiran zadatkom koji je zaključao taj semafor. U protivnom, ako je prioritet od J veći, tada mu se dopušta zaključavanje semafora S i ulazak u kritični odsječak. Kada J završi sa svojim kritičnim odsječkom i oslobodi semafor S , zadatak najvećeg prioriteta koji je čekao na taj semafor, ako postoji, bit će aktiviran.

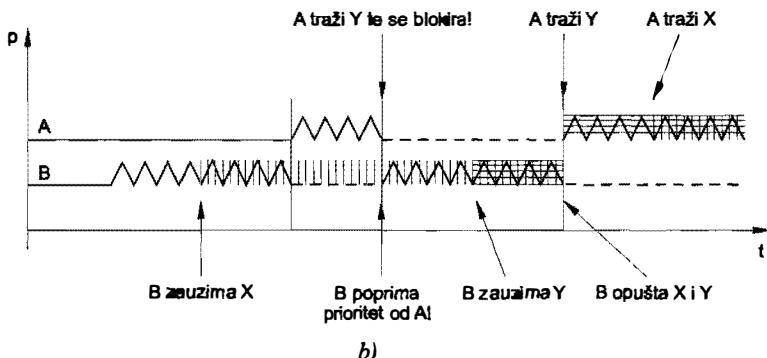
Treba primijetiti da ako je semafor S već zaključan, tada je to sa zadatkom koji ima osnovni ili naslijedeni prioritet jednak ili veći od prioriteta zadatka J koji će u tom slučaju biti blokiran na semaforu S .

2. Zadatak J koristi pridijeljeni mu prioritet osim ako se nalazi u kritičnom odsječku kojim blokira prioritetnije zadatke. Ako blokira prioritetnije zadatke, tada on poprima prioritet $p(J_H)$ koji je jednak prioritetu blokiranog zadatka s najvećim prioritetom. Kada J izlazi iz kritičnog odsječka, vraća mu se prioritet koji je imao prije ulaska u kritični odsječak. Nasljeđivanje prioriteta je tranzitivno. Operacije nasljeđivanja prioriteta i povratka prioriteta nedjeljive su operacije.
3. Zadatak J izvan kritičnog odsječka može blokirati zadatak J_L ako je prioritet zadatka J veći od prioriteta, naslijedenog ili zadanog, kojim se izvodi zadatak J_L .

Analizom ovog protokola može se ustanoviti sličnost s protokolom nasljeđivanja prioriteta. I kod ovog se protokola prioritet blokiranog zadatka nasljeđuje u trenutku blokiranja. Osnovna je razlika što se u nekim slučajevima zadatku neće dopustiti zaključavanje semafora iako je on slobodan, a zbog mogućeg blokiranja zadataka većeg prioriteta nad drugim semaforima. Na prvi pogled to blokiranje izgleda nepotrebno, ali ono zapravo pridonosi rješavanju problema potpunog zastoja, barem osnovnih oblika potpunog zastoja. U slučajevima gdje je problem potpunog zastoja značajan i često se pojavljuje, ovaj je protokol dobar odabir.

Primjena navedenog protokola u sinkronizacijskim funkcijama zahtijeva posebnu pozornost. Za razliku od protokola nasljeđivanja prioriteta i ako je sredstvo slobodno, treba provjeriti smije li zadatak zauzeti sredstvo. Primjer pojavljivanja potpunog zastoja kao i njegova izbjegavanja prikazan je na slici 6.15. .





Slika 6.15.
Primjer potpunog zastoja i njegovo izbjegavanje stropnim protokolom
a) bez korištenja stropnog protokola,
b) uz korištenje stropnog protokola

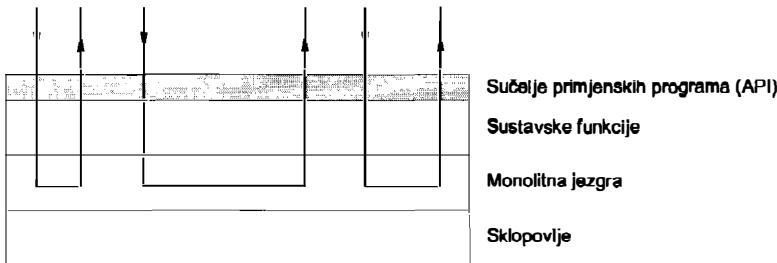
Kod *pojednostavljenog* protokola stropnog prioriteta dretvi se odmah pri zauzimanju sredstva podigne prioritet na unaprijed određenu stropnu vrijednost. Na taj se način za vrijeme korištenja nekog sredstva dretvama povećava prioritet da bi one što prije završile s nješovim korištenjem i osloboidle ga za dretve višeg prioriteta. Protokol je jednostavniji za ostvarenje od prethodnog, ali je po pitanju učinkovitosti lošiji. Dretva nižeg prioriteta začećem određenog sredstva dobiva veći prioritet i istiskuje prioritetnije dretve čak i onda kada od dretvi višeg prioriteta ne postoji trenutačna potreba za sredstvom.

6.5. Izgradnja modernih operacijskih sustava

Operacijski sustavi zasnovani na mikrojezgri

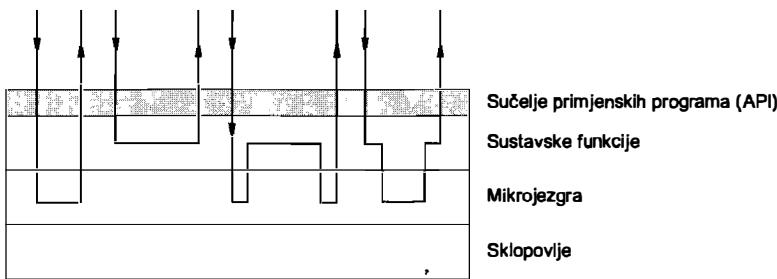
Prvotni operacijski sustavi bili su zasnovani na sustavskim funkcijama⁷ koje su gotovo isključivo bile ostvarivane kao jezgrene funkcije. Takve jezgre nazivamo *monolitnim jezgrama*. Podsetimo se da se jezgrene funkcije odvijaju u sustavskom načinu rada procesora, tj. sa zabranjenim prekidanjem. Svaki prelazak iz korisničkog načina rada u sustavski i obrnuto zahtijeva promjenu konteksta, što bitno usporava odvijanje poslova u računalnom sustavu. Osim toga, neke dugotrajnije sustavske funkcije uzrokuju da prihvati nekih novih prekida (koji mogu vrlo visokog prioriteta) bivaju odgađani na dulje vrijeme. Način pozivanja sustavskih funkcija u takvom okruženju prikazan je slikom 6.16. Svi pozivi sustavskih funkcija kroz API sučelje završavaju u monolitnoj jezgri.

⁷ Sustavske su funkcije one funkcije operacijskog sustava koje su dostupne kroz njegovo sučelje prema primjenskim programima (API).



Slika 6.16. Operacijski sustav s monolitnom jezgrom

Pokazalo se da je koncept monitora pogodan za ostvarenje određenih funkcionalnosti (usluga, sučelja) operacijskih sustava. Tu je zamisao obrazložio C. A. Hoarea u svom prvom objavljenom radu koji spominje monitore. Većina današnjih operacijskih sustava zasnovana je na tzv. mikrojezgri (engl. *microkernel*). U takvoj su mikrojezgri ostvarene samo osnovne jezgrine funkcije. Sve ostale sustavske funkcije ostvaruju se uporabom monitora. Jezgrine su funkcije tada malobrojne i njihovo izvođenje kratko traje. Većina je funkcija operacijskog sustava koje se pozivaju kroz API tako velikim dijelom vremena izvodi izvan jezgre, u prekidivom načinu rada. Time je odziv na neke hitne događaje mnogo brži. Osim toga, nekada operaciju sustavskih funkcija mogu obaviti samo u korisničkom načinu rada procesora. Ilustraciju takvog funkcionalnog rješenja operacijskog sustava prikazuje slika 6.17.



Slika 6.17. Operacijski sustav s mikrojezgrom

PRIMJER 6.5.

Kao primjer ostvarenja sustavskih funkcija slijedi primjer ostvarenja sustava za razmjenu poruka između dretvi istog procesa. Upravljanje i sinkronizacija dretvi ostvarena je uz pomoć monitora.

```
struktura Red_poruka {
    Lista<Kazaljka> poruke;
    Red_uvjeta red_uvjeta;
    Broj br_dretvi;
};
```



```

Monitor m;
Kazaljka<Red_poruka> RED[MAX_REDLOVA];

funkcija Stvoriti_red_poruka(id_reda) {
    id_reda = -1;
    Zatkjučati_monitor(m);
    za i = 1 doMAX_REDLOVA radi {
        ako (red poruka RED[i] nije stvoren) {
            stvori_zaglavje_reda_poruka(RED[i]);
            id_reda = i;
            izadi_iz_petlje;
        }
    }
    Otključati_monitor(m);
}

funkcija Obrisati_red_poruka(id_reda) {
    Zatkjučati_monitor(m);
    oslobođi red poruka RED[id_reda];
    Otključati_monitor(m);
}

funkcija Poslati_poruku(id_reda, poruka) {
    Zatkjučati_monitor(m);
    umetni_poruku_na_kraj_reda_poruka(poruka, RED[id_reda]);
    ako je (RED[id_reda].br_dretvi > 0) {
        Propustiti_iz_reda(RED[id_reda].red_uvjeta);
    }
    Otključati_monitor(m);
}

funkcija Primiti_poruku(id_reda, poruka) {
    Zatkjučati_monitor(m);
    dok je (red poruka RED[id_reda] prazan) {
        RED[id_reda].br_dretvi++;
        Čekati_u_redu(RED[id_reda].red_uvjeta, m);
        RED[id_reda].br_dretvi--;
    }
    poruka = izvadi_prvu_poruku_iz_reda_poruka(RED[id_reda]);
    Otključati_monitor(m);
}

```

Normizacija API funkcija

Kao što je već višekratno rečeno, sve funkcije operacijskog sustava dostupne su korisničkim programima kroz sučelje programskih programa – API. Zbog lakše uporabe i zbog prenosivosti programskih ostvarenja vrlo je važno sučelja normirati. Danas su prepoznatljiva dva takva sučelja. Za otvorene sustave (zasnovane na operacijskom sustavu *Linux* ili nekoj od inačica operacijskog sustava *UNIX*) pretežito se koristi *POSIX* sučelje dok tvrtka *Microsoft* za operacijske sustave *Windows* nudi svoje sučelje *Win32* ali i mogućnost uporabe sučelja *POSIX*, kao što je to ilustrirano slikom 6.18.

Sustavske funkcije
Mikrojezgra
Sklopoljje

Slika 6.18. Operacijski sustav s mogućnošću odabira sučelja programskih programa

Napomenimo da je *POSIX* (od engl. *Portable Operating System Interface* – prenosivo sučelje operacijskog sustava, uz dodatak slova X koje potječe od *UNIX*) zajednički naziv za skupinu normi kojima se definira API. Norme je razradila udruga IEEE i poznate su kao IEEE 1003 norme. Kasnije su od međunarodne organizacije za normiranje (International Standard Organization) prihvачene kao međunarodne norme ISO/ IEC 9945. Norme su nastale iz projekta koji je započeo oko 1985. *POSIX* norme implementirane su u gotovo svim operacijskim sustavima koji vuku korijene iz operacijskog sustava *UNIX*. Dio normi povezan sa stvaranjem dretvi i mehanizmima za njihovu sinkronizaciju definiran je 1995. godine. Podrška za rad s dretvama i sličnim sinkronizacijskim mehanizmima postojala je i prije, u okviru zasebnih inačica *UNIX* sustava, ali sučelje je za svaki takav sustav bilo različito. Definiranjem *POSIX* normi i njegovom implementacijom, povećala se prenosivost programa pisanih za *UNIX* na razini izvornih kodova.

Korisnik koji se odluči za jedno od sučelja morat će detaljno izučiti pripadne funkcije. One su podijeljene u funkcionalne cjeline tako da se u načelu ne mora proučiti cjelokupna opširna biblioteka funkcija već samo njezini dijelovi potrebni za ostvarenje korisnikova konkretnog rješenja. Ovdje ćemo u dvama primjerima ilustrirati taj pristup i povezati funkcije iz *Win32* i *POSIX* sučelja s modelima koje smo koncepcijски razmotrili.

PRIMJER 6.6.

U *Win32* sučelju za rad sa semaforima stoje na raspolaganju funkcije *WaitForSingleObject*, *WaitForMultipleObjects* te *ReleaseSemaphore*. Prve dvije koriste se ne samo za semafore već i za druge oblike sinkronizacije. Kada se koriste za čekanje na semaforu (parametri su semafori), tada neće blokirati dretvu koja ih poziva ako pri smanjivanju vrijednost semafora ne poprimi negativnu vrijednost. Funkcija *ReleaseSemaphore* oslobađa blokirane dretve ili povećava vrijednost semafora za jedan ili više. Primjer koji slijedi prikazuje korištenje funkcija iz *Win32* sučelja za ostvarenje dviju dretvi koje naizmjence ispisuju *Ping* i *Pong*, a međusobno se sinkroniziraju semaforom.





```

#include <windows.h>
#include <stdio.h>
HANDLE hDretva[2], sem_ping, sem_pong;

DWORD WINAPI Ping(LPVOID param) {
    while(1) {
        WaitForSingleObject(sem_ping, 0);
        printf("Ping\n");
        ReleaseSemaphore(sem_pong, 1, NULL);
        Sleep(1000);
    }
    return 0;
}

DWORD WINAPI Pong(LPVOID param) {
    while(1) {
        WaitForSingleObject(sem_pong, 0);
        printf("Pong\n");
        ReleaseSemaphore(sem_ping, 1, NULL);
        Sleep(1000);
    }
    return 0;
}

void main() {
    sem_ping = CreateSemaphore(NULL, 1, 1, NULL);
    sem_pong = CreateSemaphore(NULL, 0, 1, NULL);
    hDretva[0] = CreateThread(NULL, 0, Ping, NULL, 0, NULL);
    hDretva[1] = CreateThread(NULL, 0, Pong, NULL, 0, NULL);
    WaitForMultipleObjects(2, hDretva, TRUE, INFINITE);
    CloseHandle(sem_ping);
    CloseHandle(sem_pong);
}

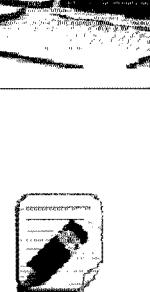
```

Osim samih funkcija sinkronizacije navedeni primjer prikazuje i sučelja za dobavljanje semafora, stvaranje dretvi, ostvarivanje kašnjenja (funkcija Sleep) i drugih.

PRIMJER 6.7.



Ostvarivanje kašnjenja dretvi koje smo u našem modelu jezgre u poglavљу 5. potaknuli jezgrinom funkcijom `Zakasniti`, sebe može se najjednostavnije u *POSIX* sučelju obaviti funkcijama `sleep` i `usleep` i funkcijom `Sleep` iz *Win32* sučelja. Funkcije primaju vremena izražena u sekundama ili manjim jedinicama. Sučelje `clock_nanosleep` prema *POSIX* standardu potencijalno nudi i razlučivost u nanosekundama (stvarna razlučivost ovisi o implementaciji sustava i razlučivosti njegova sata).

**PRIMJER 6.8.**

Popis funkcija koje se koriste za ostvarenje monitora u odjeljku 6.3.3. kao i njihova ekvivalentna API i Win32 prikazana su u tablici 6.1.

Tablica 6.1. Popis funkcija za ostvarenje monitora

Naziv funkcije	POSIX	Win32
Zaključati_monitor	pthread_mutex_lock	EnterCriticalSection
Otključati_monitor	pthread_mutex_unlock	LeaveCriticalSection
Čekati_u_reda	pthread_cond_wait	SleepConditionVariableCS
Propustiti_iz_reda	pthread_cond_signal	WakeConditionVariable
Propustiti_sve_iz_reda	pthread_cond_broadcast	WakeAllConditionVariable

Monitorske funkcije Win32 sučelja u ovom su se obliku pojavile tek od inačice operacijskog sustava naziva *Microsoft Vista* objavljenog 2007. godine. Ove monitorske funkcije omogućuju ostvarenje monitora u skladu s opisom u odjeljku 6.3.3. Podsjetimo se da je potencijalna prednost ovog monitora u tome što dretva koja djeluje na oslobođenje druge dretve blokirane u monitoru i dalje ostaje u monitoru. Pokazalo se, naime, da vrlo često dretva u monitoru osim što oslobada jednu dretvu treba još nešto obaviti unutar monitora (primjerice, osloboditi i neke druge dretve iz drugih redova). U takvom se slučaju smanjuje broj poziva jezgrinih funkcija. Dretva izlazi iz monitora samo pozivom funkcije *Otključati_monitor()*.



PITANJA ZA PROVJERUZNANJA 6

1. Sinkronizirati proizvodača i potrošača korištenjem brojačkog semafora.
2. Sinkronizirati više proizvodača i više potrošača uz pomoć binarnih i brojačkih semafora.
3. Sinkronizirati rad dviju dretvi tako da se one obavljaju naizmjenično.
4. Što je potpuni zastoj?
5. Navesti nužne uvjete za nastajanje potpunog zastoja.
6. Što je monitor?
7. Navesti jezgrine strukture podataka koje se koriste za ostvarenje monitora.
8. Navesti jezgrine funkcije za ostvarenje monitora.
9. Kada se može dogoditi da se dvije dretve nađu u monitoru?
10. U pseudokodu napisati jezgrine funkcije za ostvarenje monitora: Ući_u_monitor, Izaći_iz_monitora, Uvrstiti_u_red_uvjeta i Oslobođiti_iz_reda_uvjeta.
11. Kojim jezgrinim mehanizmom moraju biti zaštićene korisničke monitorske funkcije?
12. U čemu se razlikuje monitorski semafor od binarnog semafora?

Analiza vremenskih svojstava računalnog sustava

7.1. Uvodna razmatranja

U dosadašnjim poglavljima obradivali smo uglavnom probleme povezane s organiziranjem rada računala. Pritom smo ustanovili da suvremeni operacijski sustavi podržavaju istodobno odvijanje više dretvi i više procesa i istodobno nadziru obavljanje raznovrsnih ulazno-izlaznih operacija. Utvrđili smo da se dretva može smatrati jedinkom aktivnosti unutar računalnog sustava. Do sada smo dretvu promatrali samo s logičkog stanovišta i ustanovili da, u pojednostavljenom modelu, ona u trenutku kada započinje svoje izvođenje iz svoje domene dohvaca ulazne podatke, a u trenutku kada ga završava pohranjuje rezultate u svoju kodomenu. Nismo se pitali koliko su ti trenuci međusobno razmaknuti, tj. koliko je trajanje izvođenja dretve.

S obzirom na to da dretvu obrađuje procesor, i to tako da redom obavlja njezine instrukcije koje su pohranjene u dretvenom dijelu radnog spremnika, trajanje izvođenja dretvi ovisit će o broju instrukcija dretve i o brzini kojom procesor obavlja instrukcije. Znamo da je brzina procesora određena frekvencijom generatora takta koji potiče na rad sklopovlje procesora te da frekvencija takta neposredno određuje broj instrukcija koje procesor može izvesti u jedinici vremena (u 2. smo se poglavljju time detaljnije pozabavili i znamo da je osnovna mjerda za brzinu procesora MIPS). Prema tome, ako povećamo brzinu procesora, skratit ćemo trajanje izvođenja dretve.

Vremensko ponašanje cijelog računalnog sustava ne ovisi samo o brzini procesora već i o brzini ostalih računalnih sredstava, kao i o brzini rada ulazno-izlaznih naprava i stoga ga je vrlo teško ocijeniti. No već i približni model ponašanja omogućuje nam shvaćanje nekih vremenskih svojstava sustava, što nam može pomoći pri donošenju odluka o izgradnji ili dogradnji sustava.

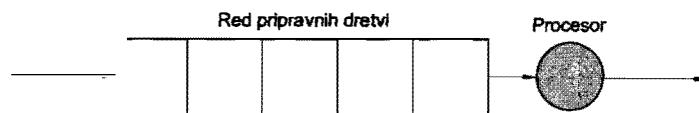
Ocjena ponašanja sustava može se zasnivati na:

- matematičkom modelu koji nam omogućuje matematičku analizu sustava;
- simulacijskom modelu koji nam omogućuje izračunavanje parametara ponašanja s pomoću računala;

- mjerenuju ponašanja stvarnog sustava.

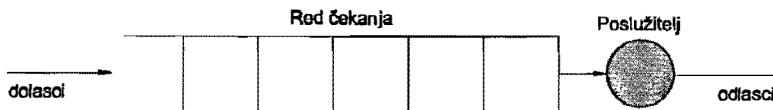
Ovdje ćemo se zadovoljiti izradom jednostavnog matematičkog modela, kako bismo dobili osnovnu sliku o ponašanju sustava i razumjeli podlogu pojedinih postupaka dodjele procesora pojedinim dretvama.

Najjednostavniji model sustava na kojem se mogu izučavati vremenska svojstva sastoji se od jednog reda u kojem dretve čekaju na izvođenje i procesora koji te dretve izvodi. U 5. smo poglavlju opisali osnovni model jezgre u kojem sve dretve koje su pripravne za izvođenje stavljam u red pripravnih dretvi. Ako je red pripravnih dretvi organiziran po redu prispjeća, onda će se u procesor priupustiti ona dretva koja je prva došla. Kažemo da procesor poslužuje dretve po redu prispjeća. Ako nam, nadalje, nije važno odakle se opisnici premještaju u red pripravnih dretvi, taj se red zajedno s procesorom koji će ih izvoditi može prikazati slikom 7.1.



Slika 7.1. Pojednostavljeni prikaz reda pripravnih dretvi jednostavnog modela jezgre

Ovaj je slikovni prikaz sličan slici 6.2. iz poglavlja 6. koja nam prikazuje red poruka koje čekaju pred potrošačem. Tamo smo rekli da proizvođač (ili više njih) stavlja poruke u red, a potrošač ih iz reda uzima po redu prispjeća i zatim ih obraduje. Svaka poruka zahtijeva određeno vrijeme obrade. Očigledno je da se ta dva zbivanja mogu modelirati jednim popoćenim modelom prema slici 7.2.



Slika 7.2. Model jednostavnog sustava s jednim redom čekanja i jednim poslužiteljem

Jednostavni sustav prikazan na slici sastoji se od *reda čekanja* i *poslužitelja*. Uz svaki posao (izvođenje dretve ili obrada poruke) povezana su dva događaja:

- događaj dolaska, ili kraće samo: dolazak i
- događaj odlaska, ili kraće samo: odlazak.

U nekom trenutku t_d posao ulazi u sustav, tj. zbiva se događaj dolaska. Ako je red čekanja prazan i poslužitelj sloboden, posao će odmah biti prihvati za posluživanje i nakon trajanja posluživanja napustiti će sustav. Ako je poslužitelj zauzet, posao se zadržava u redu čekanja i tek kada dođe na red prihvatiće ga poslužitelj, te će nakon završetka posluživanja napustiti sustav u trenutku t_n .

Trajanje zadržavanja u sustavu T prema tome je jednako:

$$T = t_n - t_d.$$

Označimo li trajanje posluživanja s T_p , onda možemo odrediti trajanje zadržavanja u redu T_r . Ono će biti jednako:

$$T_r = T - T_p.$$

Uočite da smo trenutke označavali s malim slovom t , a vremenske intervale s velikim slovima T . Svaki interval možemo odrediti kao razliku dvaju trenutaka.

Ako su svi trenuci dolazaka novih poslova poznati i ako su trajanja posluživanja tih poslova (kraće: trajanja poslova) određena, onda govorimo o determinističkom ponašanju sustava. Ponašanje takvog sustava može se u potpunosti predvidjeti.

PRIMJER 7.1.

Pretpostavimo da sustav prema slici 7.2. obraduje pet poslova koji u njega dolaze periodno s periodom od 20 jedinica vremena. Trenuci dolazaka u prvoj periodi koja započinje s $t = 0$ i trajanja poslova navedeni su u tablici 7.1.



Tablica 7.1. Trenuci dolazaka i trajanja skupine poslova

Posao	t_d	T_p
D_1	2	3
D_2	3	5
D_3	4	2
D_4	6	2
D_5	7	2

Ta će se skupina poslova ponavljati s periodom od 20 jedinica vremena, tj. sa svim trenucima dolazaka uvećanim za $k \cdot 20$. Periodnost nam omogućuje da promatranjem samo jedne periode donosimo zaključke o ponašanju sustava koji vrijede tijekom cijelog izvodenja poslova.

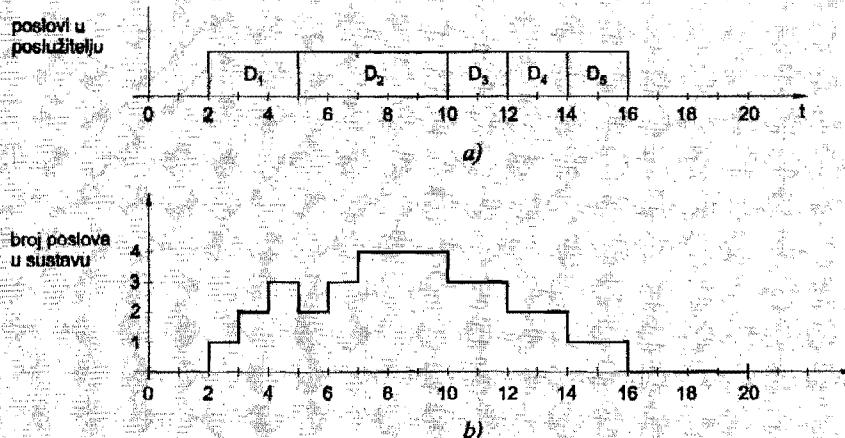
Na osnovi tablice 7.1. može se nacrtati tijek obrade poslova. Prvi posao D_1 dolazi u trenutku 2, kada je red prazan i stoga ga poslužitelj odmah prihvata. Njegova obrada traje 3 jedinice vremena i on napušta sustav u trenutku 5. Do toga trenutka već su pristigli poslovi D_2 i D_3 koji moraju čekati u redu. Istog trena kada posao D_1 napusti sustav, poslužitelj iz reda preuzima posao D_2 i počinje ga obradivati. Nastavimo li tako razmatrati odvijanje poslova dobit ćemo sliku 7.3.a). Iz te se slike mogu očitati trenuci odlazaka t_n pojedinih poslova iz sustava, a zatim odrediti i njihova trajanja zadržavanja u sustavu T , odnosno zadržavanja u redu T_r .

Ti su brojevi prikazani tablicom 7.2. u kojoj su u prva dva stupca ponovljeni podaci iz tablice 7.1.

Tablica 7.2. Vremensko ponašanje skupine poslova

Posao	t_d	T_p	t_n	T	T_r
D_1	2	3	5	3	0
D_2	3	5	10	7	2
D_3	4	2	12	8	6
D_4	6	2	14	8	6
D_5	7	2	16	9	7

Iz podataka tablice 7.2. možemo odrediti prosječno zadržavanje poslova u sustavu tako da zbrojimo pojedinačna vremena i podijelimo ih s brojem poslova, te dobivamo $\bar{T} = 7$. Isto tako, iz tablice zaključujemo da je prosječno vrijeme zadržavanja u poslužitelju $\bar{T}_p = 2.8$ i prosječno vrijeme zadržavanja u redu $\bar{T}_r = 4.2$.



Slika 7.3. a) Posluživanje poslova prema tablici 7.1.

b) Broj poslova u sustavu tijekom perioda od 20 jedinica vremena

Na temelju tablice 7.2. u kojoj možemo pročitati trenutke dolazaka i odlazaka poslova iz sustava može se nacrtati slika 7.3.b) u kojoj je prikazana vremenska zavisnost broja poslova koji se nalaze u sustavu. Iz te se slike može zaključiti koliki je prosječni broj poslova u sustavu. Taj se broj dobiva tako da se površina ispod izlomljene crte koja prikazuje trenutačni broj poslova u sustavu podijeli s trajanjem periode (ovdje dolazi do izražaja prije spominjanja periodnost), tako da se dobiva:

$$\bar{n} = \frac{35}{20} = 1.75.$$

Nadalje, možemo ustanoviti da u 20 jedinica vremena dolazi 5 poslova te je prema tome prosječni broj dolazaka u jedinici vremena jednak:

$$\alpha = \frac{5}{20} = 0.25.$$

Iz navedenog se primjera vidi da je:

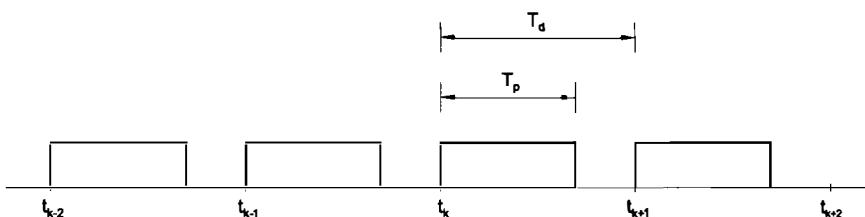
$$\bar{n} = \alpha \cdot \bar{T}.$$

Taj je izraz poznat kao *Littleovo pravilo* ili *Littleova formula*. Može se pokazati da ono vrijedi sasvim općenito. Mi to ovdje nećemo dokazivati, ali ćemo ponuditi jedno intuitivno obrazloženje.

Odaberimo trenutak t_0 kada jedan posao ulazi u sustav. U prosjeku bi taj posao trebao izaći iz sustava u trenutku $t_0 + \bar{T}$. Prije njega su iz sustava izašli svi poslovi koji su u sustav ušli jer se obavljuju redom prispijeća. To znači da se u trenutku kada posao napušta sustav u njemu nalaze samo poslovi koji su ušli poslije njega. Ti novi poslovi ulazili su u razdoblju dok je on bio u sustavu, a to je upravo vrijeme prosječnog zadržavanja \bar{T} . S obzirom na to da u jedinici vremena ulazi α poslova, u sustav je u tom razdoblju ušlo $\alpha \cdot \bar{T}$ poslova. Prema tome, Littleova formula vrijedi.

7.1.1. Periodni poslovi

Promotrimo jednostavan slučaj kada sustav periodno obavlja posao koji uvijek ima jednako trajanje. U takvom su sustavu trenuci dolazaka jednakom razmaku i svi su poslovi jednake duljine, kao što to prikazuje slika 7.4. Razmak između dvaju dolazaka označit ćemo s T_d , a trajanje poslova s T_p .



Slika 7.4. Periodični dolasci poslova jednog trajanja

Sasvim je razumljivo da trajanje posluživanja mora biti kraće od razmaka između dvaju dolazaka jer bi se inače u sustavu poslovi nagomilali. Prema tome, mora biti zadovoljen uvjet:

$$T_p \leq T_d.$$

Vrijeme zadržavanja u sustavu jednako je vremenu posluživanja, odnosno:

$$T = T_p,$$

a vrijeme zadržavanja u redu jednako je nuli:

$$T_r = 0.$$

Red je, naime, uvijek prazan i u trenutku dolaska svaki se posao odmah prosljeđuje poslužitelju. U graničnom slučaju, kada je $T_p = T_d$, vrijeme zadržavanja u sustavu jednako je periodi dolazaka.

Poslužitelj je zaposlen samo dio vremena. U jednoj periodi trajanja T_d on je zaposlen u razdoblju trajanja T_p . Nazvat ćemo omjer tih dvaju vremena *faktorom iskorištenja poslužitelja ρ* , tj.

$$\rho = \frac{T_p}{T_d}.$$

Često se iskorištenje η poslužitelja izražava u postocima, odnosno faktorom iskorištenja pomnoženim sa sto, odnosno:

$$\eta = \frac{T_p}{T_d} \cdot 100 = \rho \cdot 100.$$

Uz $T_p = T_d$ iskorištenje poslužitelja će iznositi 100%. U tom slučaju poslužitelj uvijek dobiva novi posao u trenutku kada obavljeni prethodni posao napušta sustav.

Uvest ćemo posebne oznake za recipročne vrijednosti vremena T_d i T_p . Recipročna vrijednost vremenskog razmaka između dvaju dolazaka bit će označena s:

$$\alpha = \frac{1}{T_d},$$

pri čemu je α broj dolazaka novih poslova u jedinici vremena.

Recipročna vrijednost trajanja posluživanja bit će označena s

$$\beta = \frac{1}{T_p},$$

pri čemu β predstavlja broj poslova koje bi poslužitelj mogao obaviti u jedinici vremena. Naime, poslužitelj bi mogao obaviti sve te poslove trajanja T_p kada bi oni dolazili s vremenskim razmakom $T_d = T_p$.

S obzirom na to da mora biti ispunjen uvjet $T_p \leq T_d$, slijedi da mora biti:

$$\frac{1}{\beta} \leq \frac{1}{\alpha},$$

odnosno:

$$\frac{\alpha}{\beta} = \rho \leq 1.$$

Utvrđimo još jedanput da faktor iskorištenja ρ u ovom potpuno determinističkom slučaju može poprimiti vrijednost 1.

Postavlja se pitanje kako se ponaša ovaj jednostavni sustav kada vremena između dvaju dolazaka nisu uvijek jednaka i kada poslovi koji ulaze u sustav nemaju svi jednako trajanje posluživanja.



Za pojedinačne konkretne slučajeve s poznatim trajanjima posluživanja i danim trenucima dolazaka može se o ponašanju sustava zaključivati kao što smo to činili u primjeru 7.1. Pokazalo se da se neki opći zaključci mogu načiniti prikladnjem modeliranjem promjenljivih razmaka između uzastopnih dolazaka i promjenljivih trajanja posluživanja.

Mi ćemo prethodno opisani model potpuno determinističkog ponašanja sustava prevesti u nedeterministički model tako da:

- umjesto jednakih razmaknutih trenutaka dolazaka kojih ima α u jedinici vremena pretpostavimo da je broj dolazaka u jedinici vremena podvrgnut Poissonovoj razdiobi s očekivanjem jednakim α ;
- umjesto jednakih trajanja posluživanja $1/\beta$ svih poslova mi pretpostavimo da su trajanja poslova podvrgnuta eksponencijalnoj razdiobi s očekivanjem jednakim $1/\beta$.

U *teoriji redova* (engl. *Queueing theory*) ili *teoriji masovnog posluživanja* uz te se pretpostavke izvode najjednostavniji modeli ponašanja sustava. Za one koji se do sada nisu sretali s teorijom redova ukratko ćemo opisati način izgradnje modela.

U sljedećem ćemo odjeljku ukratko opisati međusobnu povezanost eksponencijalne i Poissonove razdiobe i dovesti ih u vezu s našim problemom modeliranja posluživanja.

7.2. Povezanost Poissonove i eksponencijalne razdiobe

7.2.1. Poissonova razdioba

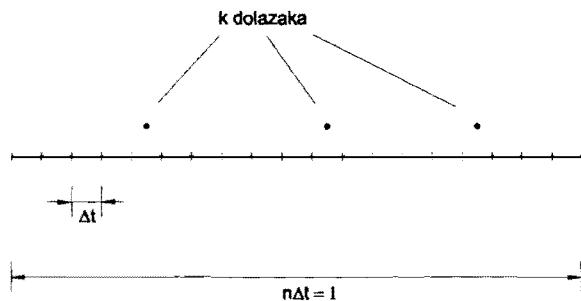
Podsetimo se da Poissonova razdioba može biti razmatrana kao aproksimacija binomne razdiobe. Ukratko ćemo ponoviti način njezina izvođenja, tako da on bude neposredno oslonjen na naš problem modeliranja trenutaka dolazaka novih poslova.

Na slici 7.5. prikazan je jedinični vremenski interval podijeljen na n pretinaca širine δt , tako da je $n \cdot \Delta t = 1$.

Ako se u tom jediničnom intervalu dogodi k dolazaka, oni mogu biti razvrstani na ukupno $\binom{n}{k}$ načina. Ako nadalje pretpostavimo da je p vjerojatnost prisustva događaja i q vjerojatnost odsustva događaja (gdje je $q = 1 - p$), onda je vjerojatnost da u jediničnom intervalu vremena imamo k događaja jednaka:

$$b(k, n, p) = \binom{n}{k} p^k q^{n-k}.$$

U gornjem je izrazu slučajna varijabla k . Ona može poprimati samo cjelobrojne vrijednosti. Vrijednosti n i p parametri su razdiobe.



Slika 7.5. Vremenski interval duljine 1 podijeljen na n vremenskih pretinaca

Vjerojatnost da u jediničnom intervalu nema ni jednog događaja dobiva se uz $k = 0$ te je:

$$b(0,n,p) = q^n = (1-p)^n.$$

Suprotna vjerojatnost je vjerojatnost da postoji barem jedan događaj u jediničnom intervalu i može se pisati:

$$b(k > 0, n, p) = 1 - q^n.$$

Poissonova aproksimacija binomne razdiobe dobiva se kada broj pretinaca n teži u beskonačnost (s tim da širina pretinaca teži k nuli) i vjerojatnost p teži k nuli s tim da umnožak $\lambda = np$ ima konačnu vrijednost.

Vjerojatnost da nema ni jednog događaja može se prepisati u sljedeći oblik:

$$\begin{aligned} b(0, n, p) &= (1-p)^n \\ &= \left(1 - \frac{\lambda}{n}\right)^n \\ &= \left[\left(1 + \frac{1}{-\frac{n}{\lambda}}\right)^{-\frac{n}{\lambda}}\right]^{-\lambda}. \end{aligned}$$

Kada $n \rightarrow \infty$, gornji izraz u uglatoj zagradi teži prema e pa dobivamo:

$$b(0, n, p) = e^{-\lambda}.$$

Iako smo n i p nadomjestili jednim parametrom λ , za sada ćemo privremeno i dalje pisati parametre n i p .

Ako načinimo omjer $b(k,n,p)$ i $b(k-1,n,p)$ dobivamo:

$$\begin{aligned}\frac{b(k,n,p)}{b(k-1,n,p)} &= \frac{\binom{n}{k} p^k q^{n-k}}{\binom{n}{k-1} p^{k-1} q^{n-k+1}} \\ &= \frac{n-k+1}{k} \cdot \frac{p}{q} \\ &= \frac{np}{kq} - \frac{(k-1)p}{kq} \\ &= \frac{\lambda}{kq} - \frac{(k-1)p}{kq}.\end{aligned}$$

Ako sada pustimo da $p \rightarrow 0$, drugi razlomak teži k nuli i preostaje:

$$\frac{b(k,n,p)}{b(k-1,n,p)} = \frac{\lambda}{k}$$

odnosno:

$$b(k,n,p) = \frac{\lambda}{k} \cdot b(k-1,n,p).$$

S obzirom na to da nam je $b(0,n,p)$ poznato, postupnim uvrštavanjem dobivamo:

$$\begin{aligned}b(0,n,p) &= e^{-\lambda}, \\ b(1,n,p) &= \lambda e^{-\lambda}, \\ b(2,n,p) &= \frac{\lambda^2}{2} e^{-\lambda}, \\ &\vdots \\ b(k,n,p) &= \frac{\lambda^k}{k!} e^{-\lambda}.\end{aligned}$$

Sada možemo zamijeniti oznaće tako da umjesto $b(k,n,p)$ pišemo $p(k,\lambda)$ te je:

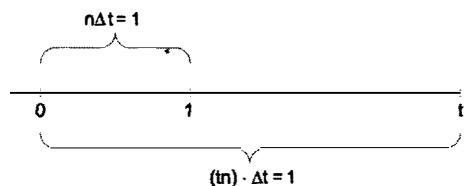
$$p(k,\lambda) = \frac{\lambda^k}{k!} e^{-\lambda}.$$

Podsjetimo se – ovdje je slučajna varijabla k . Ona može poprimiti vrijednosti iz skupa \mathbb{N}_0 . Gornji nam izraz kaže kolika je vjerojatnost da u jedinici vremena uz dani λ imamo k dolazaka novih poslova. Postavlja se pitanje što u našem modelu predstavlja parametar λ ? Odgovor na to pitanje dobivamo kada odredimo očekivanje slučajne varijable k :

$$\begin{aligned}E(k) &= \sum_{k=0}^{\infty} kp(k,\lambda) = e^{-\lambda} \sum_{k=0}^{\infty} \frac{k\lambda^k}{k!} = e^{-\lambda} \lambda \sum_{k=1}^{\infty} \frac{\lambda^{k-1}}{(k-1)!} \\ &= e^{-\lambda} \lambda e^{\lambda} = \lambda.\end{aligned}$$

S obzirom na to da očekivanje možemo protumačiti kao "prosječnu vrijednost", parametar u Poissonovoj razdiobi λ možemo shvatiti kao prosječni broj događaja u jedinici vremena.

Mi smo do Poissonove razdiobe došli na taj način da smo aproksimirali binomnu razdiobu koja nam je govorila kolika je vjerojatnost da u vremenskom intervalu jedinične duljine imamo k dolazaka novih poslova. Razmatranje smo započeli na temelju slike 7.5. u kojoj je jedinični interval vremena podijeljen na n vremenskih pretinaca širine Δt . Pogledajmo što bismo dobili ako umjesto jediničnog intervala vremena uzmememo interval duljine t (ovdje ćemo iznimno interval označavati malim slovom). Pogledom na sliku 7.6. možemo zaključiti da ćemo u tom intervalu imati nt pretinaca. Nas će zanimati kolika je vjerojatnost da se u k pretinaca od tog ukupnog broja zbio dolazak nekog novog posla.



Slika 7.6. Vremenski interval t podijeljen na nt pretinaca širine Δt

Opet možemo poći od binomne razdiobe:

$$b(k(t), nt, p) = \binom{nt}{k} p^k q^{nt-k}$$

koju treba aproksimirati Poissonovom. Označavat ćemo k događaja u intervalu t tako da uz k u zagradi pišemo t , tj. kao $k(t)$.

Lako se može zaključiti da će cijeli izvod biti potpuno jednak samo što u svim izrazima umjesto λ treba pisati λt jer uz $\lambda = np$ treba pisati $\lambda t = (nt)p$.

U Poissonovoj ćemo razdiobi umjesto λ pisati λt , pa imamo:

$$p(k(t), \lambda) = \frac{(\lambda t)^k}{k!} e^{-\lambda t}.$$

Ovaj nam izraz, dakle, kaže kolika je vjerojatnost da u intervalu t , uz dani parametar λ , imamo k događaja. Napomenimo da parametar λ ima još uvijek isto značenje, tj. predstavlja nam prosječni broj događaja u jedinici vremena.

Nas će u dalnjim razmatranjima posebno zanimati kolika je vjerojatnost da u nekom intervalu t nema ni jednog događaja. Iz gornjeg izraza uz $k(t) = 0$ dobivamo:

$$p(k(t) = 0, \lambda) = e^{-\lambda t}.$$

Suprotna ovoj vjerojatnosti jest vjerojatnost da se u intervalu t dogodio barem jedan događaj. Tu ćemo vjerojatnost dobiti tako da od jedan oduzmemo vjerojatnost odsustva događaja, odnosno:

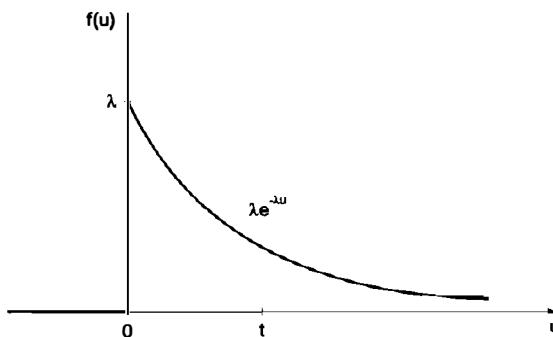
$$p(k(t) > 0, \lambda) = 1 - e^{-\lambda t}.$$

7.2.2. Eksponencijalna razdioba i njezina veza s Poissonovom

Eksponencijalna razdioba definirana je *funkcijom gustoće vjerojatnosti* sljedećeg oblika:

$$\begin{aligned} f(u) &= \lambda e^{-\lambda u} && \text{za } u \geq 0, \\ f(u) &= 0 && \text{za } u < 0. \end{aligned}$$

Ona je predstavljena slikom 7.7. Podsjetimo se da za kontinuirane slučajne varijable nema smisla govoriti o vjerojatnosti da varijabla poprimi neku vrijednost, tako da nas zanimaju vjerojatnosti da varijabla poprima vrijednosti iz nekog njezina intervala.



Slika 7.7. Funkcije gustoće vjerojatnosti eksponencijalne razdiobe

Za neku slučajnu varijablu T (ovdje je iznimno označavamo velikim slovom) možemo odrediti vjerojatnost s kojom će ona poprimiti vrijednosti iz intervala t_a do t_b tako da izračunamo površinu ispod krivulje $f(u)$ u granicama od t_a do t_b , odnosno:

$$p(t_a \leq T \leq t_b) = \int_{t_a}^{t_b} f(u) du.$$

Mi ćemo eksponencijalnom razdiobom modelirati slučajnu varijablu T koja će nam predstavljati *razmake između dvaju uzastopnih dolaska* u naš jednostavni sustav. To znači da u trenutku kada se zbio jedan događaj dolazaka počinjemo mjeriti vrijeme koje će proteći do sljedećeg događaja.

Može nas zanimati kolika je vjerojatnost da to vrijeme bude veće od nekog vremena t , tj. zanima nas kolika je vjerojatnost da T bude u granicama $t \leq T \leq \infty$. U skladu s gornjim izrazom dobivamo:

$$\begin{aligned} p(t \leq T \leq \infty) &= \int_t^{\infty} \lambda e^{-\lambda u} du \\ &= \lambda \left(-\frac{1}{\lambda} \right) e^{-\lambda u} \Big|_t^{\infty} \\ &= e^{-\lambda t}. \end{aligned}$$

Jednak smo izraz dobili iz Poissonove razdiobe kada smo odredili vjerojatnost da u intervalu t nema ni jednog događaja. Ova dva tumačenja znače zapravo istu činjenicu: naime, vjerojatnost da u intervalu t nema ni jednog događaja mora biti jednaka vjerojatnosti da je razmak između dvaju događaja veći od t .

Suprotna je vjerojatnost da je vremenski razmak između dva događaja manji od t jednaka:

$$p(0 \leq T \leq t) = 1 - e^{-\lambda t}.$$

Taj je izraz jednak vjerojatnosti da postoji barem jedan događaj u intervalu t , koji smo dobili razmatrajući Poissonovu razdiobu.

Pogledajmo sada još kakvo fizikalno značenje ima ovdje parametar λ . To ćemo načiniti tako da odredimo očekivanje slučajne varijable T :

$$E(T) = \int_0^{\infty} uf(u)du = \int_0^{\infty} ue^{-\lambda u}du = \frac{1}{\lambda}.$$

Očekivanje slučajne varijable tumačimo kao prosječnu vrijednost vremenskog razmaka između dva događaja. Parametar λ u Poissonovoj razdiobi označavao nam je prosječni broj događaja u jedinici vremena. Njegova recipročna vrijednost je onda prosječni razmak između dva uzastopna događaja.

Prema tome, možemo zaključiti da između Poissonove razdiobe, koja vrijedi za diskretne varijable (i služi nam za modeliranje slučajnih dolazaka u sustav), i eksponencijalne razdiobe koja vrijedi za kontinuirane varijable (i služi nam za modeliranje slučajnih vremenskih razmaka između dva događaja) postoji jasna dualnost koja je istaknuta tablicom 7.3.

Tablica 7.3. Povezanost Poissonove i eksponencijalne razdiobe

Poissonova razdioba	Eksponencijalna razdioba
Vjerojatnost da nema ni jednog događaja u intervalu t je $e^{-\lambda t}$	Vjerojatnost da je vrijeme između dva događaja veće od t je $e^{-\lambda t}$
Vjerojatnost da postoji barem jedan događaj u intervalu t je $1 - e^{-\lambda t}$	Vjerojatnost da je vrijeme između dva događaja manje od t je $1 - e^{-\lambda t}$
Prosječni broj događaja u jedinici vremena je λ	Prosječno vrijeme između dva događaja je $\frac{1}{\lambda}$



7.3. Analiza sustava s Poissonovom razdiobom dolazaka i eksponencijalnom razdiobom trajanja obrade

Pogledajmo kako će se ponašati sustav predstavljen slikom 7.2. kada su događaji dolazaka podvrgnuti Poissonovoj razdiobi, a trajanje obrade pridošlih poslova ima eksponencijalnu razdiobu. U prethodnom smo odjeljku razmatrali te dvije razdiobe i zbog istraživanja njihove povezanosti za obje smo razdiobe upotrebljavali isti parametar λ .

Ovdje ćemo za modeliranje dolazaka upotrijebiti Poissonovu razdiobu s parametrom α . Prema tome mi prepostavljamo da je prosječni broj dolazaka novih poslova u jedinici vremena jednak α .

Nadalje, prepostaviti ćemo da su trajanja poslova koji ulaze u sustav podvrgnuta eksponencijalnoj razdiobi s parametrom β . Taj nam parametar zapravo opisuje sposobnost poslužitelja. On će obradivati poslove tako da prosječno trajanje obrade bude jednak $1/\beta$. Možemo reći da poslužitelj ima sposobnost ostvariti odlaske čiji prosječni broj nije veći od β odlazaka u jedinici vremena. Kada bismo poslužitelj (procesor) ubrzali, parametar β bi se povećao i prosječno trajanje obrade skratilo.

U našem modelu prepostavljamo da poslužitelj uzima novi posao iz reda čekanja odmah čim se oslobodi prethodnog posla. Kada bi red čekanja uvijek bio pun, tj. kada bi poslužitelj mogao trajno raditi, onda bi (zbog povezanosti Poissonove i eksponencijalne razdiobe) trenuci odlazaka bili Poissonovi događaji s parametrom β . Takav poslužitelj može nam, dakle, poslužiti kao generator Poissonove razdiobe.

Intuitivno nam je jasno da prosječni broj dolazaka (kažemo i gustoća dolazaka) ne smije biti veći od tog broja, tj. da mora biti:

$$\alpha < \beta$$

ili:

$$\frac{\alpha}{\beta} < 1.$$

Pri opisu deterministički određene nakupine poslova, koji dolaze s jednakim razmakom i svi traju jedнако, ustanovili smo da se smije dopustiti puna zaposlenost poslužitelja. Međutim, pokazat će se da omjer parametara α i β ovdje ne smije biti jednak 1.

Analizom želimo odrediti koliko je u sustavu:

- prosječno trajanje zadržavanja \bar{T} i
- prosječni broj poslova \bar{n} .

U primjeru 7.1. ukazali smo na to da su te dvije veličine povezane Littleovom formulom koja vrijedi sasvim općenito i glasi:

$$\bar{n} = \alpha \bar{T}.$$

Prema tome, bit će nam dovoljno odrediti jednu od tih dviju veličina. Neka to bude \bar{n} .

Promatrat ćemo sustav u tzv. stohastičkom ravnotežnom stanju. To, u prvom redu, znači da su parametri α i β konstantni te da sustav promatramo dovoljno dugo i da su stoga sve vjerojatnosti u sustavu konstantne.

Iako znamo da su vjerojatnosti konstantne, označimo s $p_i(t)$ vjerojatnost da se u trenutku t u sustavu nalazi i poslova. U stacionarnom stanju mora biti:

$$p_i(t) = \text{konstanta},$$

što znači da je:

$$\frac{dp_i(t)}{dt} = 0.$$

Ova će nam činjenica pomoći pri određivanju vjerojatnosti $p_i(t)$.

Napomenimo da je slučajna varijabla broj poslova, tj. varijabla i , iako je ona ovdje malo neuobičajeno napisana u obliku indeksa.

Promatrat ćemo moguće promjene stanja sustava u vrlo malom intervalu vremena Δt .

Vjerojatnost da u vrlo kratkom intervalu Δt nema niti jednog dolaska bit će:

$$p(k(\Delta t) = 0, \alpha) = e^{-\alpha \Delta t}.$$

Uz mali Δt , razvojem u Taylorov red i odbacivanjem članova viših od linearног dobivamo:

$$p(k(\Delta t) = 0, \alpha) = 1 - \alpha \Delta t + \frac{(\alpha \Delta t)^2}{2!} - \frac{(\alpha \Delta t)^3}{3!} + \dots \approx 1 - \alpha \Delta t.$$

Prema tome, suprotna vjerojatnost da ćemo imati barem jedan dolazak u sustav bit će jednaka $\alpha \Delta t$. Mi ćemo smatrati da je interval Δt toliko malen da se umjesto "barem jedan dolazak" može smatrati da je to vjerojatnost za "samo jedan dolazak".

Slično tako, odredit ćemo vjerojatnost da nema niti jednog odlaska u intervalu Δt (vjerojatnost da je trajanje obrade veće od Δt). Ona je jednaka:

$$e^{-\beta \Delta t} = 1 - \beta \Delta t + \frac{(\beta \Delta t)^2}{2!} - \frac{(\beta \Delta t)^3}{3!} + \dots \approx 1 - \beta \Delta t.$$

Suprotna vjerojatnost da imamo barem jedan odlazak bit će jednaka $\beta \Delta t$, što ćemo mi smatrati vjerojatnošću upravo jednog odlaska.

Prema tome, mi ćemo smatrati da je za vrlo mali interval vremena Δt :

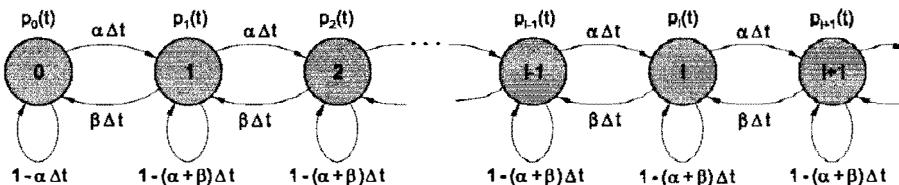
- vjerojatnost jednog dolaska jednaka $\alpha \Delta t$;
- vjerojatnost jednog odlaska jednaka $\beta \Delta t$.

Vjerojatnost da nema niti jednog odlaska ni jednog dolaska bit će jednaka umnošku:

$$(1 - \alpha \Delta t) \cdot (1 - \beta \Delta t) = 1 - (\alpha + \beta) \Delta t + \alpha \beta (\Delta t)^2 \approx 1 - (\alpha + \beta) \Delta t$$

u kojem smo opet zanemarili kvadratni član.

Ponašanje sustava možemo promotriti na slici 7.8. koja nam predstavlja graf mogućih stanja sustava (poznat kao Markovljev graf ili Markovljev lanac). Svaki čvor u tom grafu odgovara broju poslova u sustavu u trenutku t . Čvorovi su označeni brojevima $0, 1, 2, \dots, i-1, i, i+1, \dots$. Vjerojatnost da u sustavu u trenutku t ima i poslova jest $p_i(t)$, što je napisano uz svaki čvor.



Slika 7.8. Vjerojatnosni graf stanja sustava (Markovljev lanac)

Streljice između čvorova predstavljaju moguće promjene stanja u intervalu Δt . Uz svaku streljicu navedena je vjerojatnost prijelaza opisanog tom strelicom. Ako smatramo da je interval Δt tako malen da se u njemu mogu dogoditi najviše jedan odlazak ili dolazak, onda streljice mogu pokazivati samo na susjedne čvorove. Ako se nikakva promjena nije dogodila, streljica čini petlju i vraća se u isti čvor. Zbroj vjerojatnosti uz sve izlazne streljice jednog čvora mora biti jednak 1 (jer se nešto od mogućeg mora dogoditi).

Sustav se u trenutku $t + \Delta t$ može naći u stanju i na jedan od tri moguća načina:

- kada je u trenutku t bio u stanju $i - 1$ (za što postoji vjerojatnost $p_{i-1}(t)$) i u intervalu Δt se dogodio jedan dolazak (za što postoji vjerojatnost $\alpha\Delta t$);
- kada je u trenutku t bio u stanju $i + 1$ (za što postoji vjerojatnost $p_{i+1}(t)$) i u intervalu Δt se dogodio jedan odlazak (za što postoji vjerojatnost $\beta\Delta t$);
- kada je u trenutku t bio u stanju i (za što postoji vjerojatnost $p_i(t)$) i u intervalu se nije dogodio niti jedan odlazak i niti jedan dolazak (za što postoji vjerojatnost $1 - (\alpha + \beta)\Delta t$).

Prema tome, za neki $i > 0$ može se gore izrečeno zapisati u sljedećem obliku:

$$p_i(t + \Delta t) = [1 - (\alpha + \beta)\Delta t]p_i(t) + \alpha\Delta t p_{i-1}(t) + \beta\Delta t p_{i+1}(t).$$

Iznimka je stanje $i = 0$ u kojem ne može biti odlazaka i u koje se može doći samo iz stanja $i = 1$ te vrijedi:

$$p_0(t + \Delta t) = (1 - \alpha\Delta t)p_0(t) + \beta\Delta t p_1(t).$$

Ove se dve jednadžbe mogu prepisati u sljedeće oblike:

$$\frac{p_i(t + \Delta t) - p_i(t)}{\Delta t} = -(\alpha + \beta)p_i(t) + \alpha p_{i-1}(t) + \beta p_{i+1}(t)$$

i:

$$\frac{p_0(t + \Delta t) - p_0(t)}{\Delta t} = -\alpha p_0(t) + \beta p_1(t).$$

Ako sada pustimo da Δt teži k nuli, s lijeve strane jednadžbi dobit ćemo derivacije vjerojatnosti. Te derivacije u stacionarnom stanju moraju biti jednake nuli te dobivamo:

$$0 = -(\alpha + \beta)p_i(t) + \alpha p_{i-1}(t) + \beta p_{i+1}(t)$$

i:

$$0 = -\alpha p_0(t) + \beta p_1(t).$$

Iz druge jednadžbe slijedi:

$$p_1(t) = \frac{\alpha}{\beta} p_0(t) = \rho p_0(t)$$

gdje možemo omjer:

$$\rho = \frac{\alpha}{\beta}$$

nazvati, slično kao kod determinističkog periodnog opterećenja, faktorom iskorištenja poslužitelja.

Iz druge jednadžbe može se eksplisitno izraziti vjerojatnost za stanje $i + 1$ i dobiva se:

$$\begin{aligned} p_{i+1}(t) &= \left(1 + \frac{\alpha}{\beta}\right)p_i(t) - \frac{\alpha}{\beta}p_{i-1}(t) \\ &= (1 + \rho)p_i(t) - \rho p_{i-1}(t). \end{aligned}$$

Postupnim uvrštavanjem dobivamo redom sljedeće vjerojatnosti:

$$\begin{aligned} p_2(t) &= (1 + \rho)\rho p_0(t) - \rho p_0(t) = \rho^2 p_0(t) \\ p_3(t) &= (1 + \rho)\rho^2 p_0(t) - \rho^2 p_0(t) = \rho^3 p_0(t), \end{aligned}$$

odnosno:

$$p_i(t) = \rho^i p_0(t).$$

Ovim smo izrazom već gotovo odredili traženu vjerojatnost. Preostaje nam samo određivanje vjerojatnosti $p_0(t)$. Ona proizlazi iz zahtjeva da zbroj svih vjerojatnosti mora biti jednak 1 pa dobivamo:

$$\sum_{i=0}^{\infty} p_i(t) = \sum_{i=0}^{\infty} \rho^i p_0(t) = p_0(t) \sum_{i=0}^{\infty} \rho^i = 1.$$

Zbroj koji smo dobili u gornjem izrazu zbroj je geometrijskog reda koji konvergira samo uz $\rho < 1$, pri čemu je:

$$\sum_{i=0}^{\infty} \rho^i = \frac{1}{1 - \rho}.$$



Uvrštenjem tog zbroja u gornji izraz konačno se dobiva:

$$p_0(t) = 1 - \rho$$

te je vjerojatnost da se u sustavu nalazi i poslova jednaka:

$$p_i(t) = (1 - \rho)\rho^i.$$

Iz izraza vidimo da ona i nije funkcija vremena, kao što smo uostalom i polazno pretpostavili.

Sada smo već vrlo blizu izračunavanju prosječnog broja poslova u sustavu. Potrebno je samo odrediti očekivanje slučajne varijable i . Ono iznosi:

$$\bar{n} = \sum_{i=0}^{\infty} ip_i(t) = (1 - \rho) \sum_{i=0}^{\infty} i\rho^i.$$

Suma u gornjem izrazu jednaka je:

$$\sum_{i=0}^{\infty} i\rho^i = \frac{\rho}{(1 - \rho)^2},$$

pa slijedi da je:

$$\bar{n} = \frac{\rho}{1 - \rho}.$$

Ako brojnik i nazivnik pomnožimo s β , dobivamo i drugi izraz:

$$\bar{n} = \frac{\alpha}{\beta - \alpha}.$$

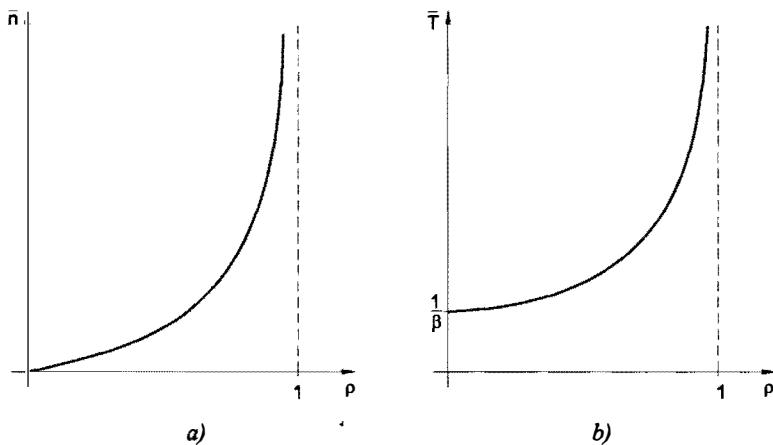
Na osnovi Littleova teorema slijedi da je prosječno trajanje zadržavanja u sustavu:

$$T = \frac{1}{\alpha} \cdot \frac{\rho}{1 - \rho}$$

ili:

$$\bar{T} = \frac{1}{\beta - \alpha}.$$

Podsjetimo se da promatramo sustav u kojem su događaji dolazaka podvrgnuti Poissonovoj razdiobi, a trajanje obrade je eksponencijalno distribuirano. Pokušamo li takav sustav opteretiti tako da faktor iskorištenja bude blizu jedinici, broj poslova u sustavu počet će rasti u beskonačnost, a vrijeme zadržavanja u sustavu također će postati suviše veliko, kao što to ilustriraju slike 7.9.a) i 7.9.b). Na slici 7.9.b) pretpostavlja se da ρ teži k nuli na takav način da β ostaje konstantno, tj. da α teži k nuli.



Slika 7.9. a) Prosječni broj poslova u sustavu π kao funkcija ρ
b) Prosječno vrijeme zadržavanja posla u sustavu kao funkcija ρ

Ta spoznaja koju smo stekli razmatrajući ovaj jednostavni sustav posluživanja ukazuje nam da se ni u kojem slučaju ne smi je računati sa stopostotnim iskorištenjem poslužitelja. Štoviše, taj zaključak možemo poopćiti na sustave u kojima su dolasci i trajanja obrade poslova podvrgnuti i drugim razdiobama. Formule za izračunavanje prosječnog broja poslova u sustavu (a prema tome i prosječnog trajanja zadržavanja u sustavu) pritom su malo drugačije i sadrže parametre tih drugih razdioba, ali se u svim slučajevima dobiva beskonačno duge redove čekanja i beskonačna trajanja zadržavanja u sustavu kada faktor iskorištenja ρ teži prema jedan.

Veliki faktor iskorištenja može se ostvariti samo u potpuno determinističkom slučaju. Čim vremena između dolazaka počinju malo varirati ili trajanja poslova nisu sva jednaka (što je vrlo realna pretpostavka), pokušaj potpunog iskorištavanja sustava dovest će do njegova zagušenja.

Zbog važnosti ovog zaključka ovdje je provedeno šire razmatranje načina njegova izvođenja. U literaturi koja se bavi operacijskim sustavima obično se to razmatranje može naći u prilozima. Uistinu, dvije završne formule i njihova vizualizacija slikom 7.9. mogu se shvatiti i bez njihova izvođenja, pa čitatelj koji ne želi pratiti cijeli izvod može cijeli ovaj odjeljak i preskočiti. Vjerojatno će i za te čitatelje zaključak biti upečatljiviji, čak i ako letimično utvrde njegovu matematičku zasnovanost.

Mi ćemo u sljedećem odjeljku iskoristiti ovaj krajnji rezultat analize za ocjenu načina dodjeljivanja procesora dretvama, a ovdje pogledajmo još jednu korisnu uporabu dobivenog izraza za vjerojatnost nalaženja *i* poslova u sustavu.

PRIMJER 7.2.

Pretpostavimo da proizvođač i potrošač komuniciraju preko ograničenog spremnika koji se sastoji od N pretinaca. U sustavu se tada može nalaziti najviše $M = N+1$ poruka (N poruka nalazi se u redu, a jednu troši potrošač). Pretpostavimo, nadalje, da proizvođač proizvodi poruke tako da su dogodaji njihovih stavljanja u spremnik podvragnuti Poissonovoj razdiobi, te da potrošač na obradu poruka troši vremena podvragnuta eksponencija noj razdiobi. Zanima nas koliko međuspremnik mora imati pretinaca da, uz dani faktor ρ , vjerovatnost blokiranja potrošača bude manja od neke zadane vrijednosti.

Proizvođač neće biti blokiran ako pronalazi u međuspremniku prazno mjesto, tj. ako je $i < M$, odnosno $i \leq N$. Prema tome, vjerovatnost $p(i \leq N)$ da proizvođač ne bude blokiran dobit ćemo kao sljedeći zbroj vjerovatnosti:

$$p(i \leq N) = \sum_{i=0}^N p_i(t),$$

Vjerovatnost da će proizvođač biti blokiran suprotna je ovoj vjerovatnosti pa dobivamo:

$$\begin{aligned} p(i > N) &= 1 - p(i \leq N) = \sum_{i=0}^{\infty} p_i(t) - \sum_{i=0}^N p_i(t) = \sum_{i=N+1}^{\infty} p_i(t) \\ &= \sum_{i=N+1}^{\infty} (1-\rho)\rho^i = (1-\rho)\rho^N \sum_{j=1}^{\infty} \rho^j = (1-\rho)\rho^N \frac{\rho}{1-\rho} \\ p(i > N) &= \rho^{N+1}. \end{aligned}$$

Na temelju ovog izraza zaključujemo da se, uz zadatu vjerovatnost $p(i > N)$ i uz dani ρ , traženi broj pretinaca N može odrediti tako da se izračuna najmanji cijeli broj koji će zadovoljiti nejednadžbu:

$$p(i > N) \geq \rho^{N+1}.$$

Zanimljivo je pogledati neke brojčane vrijednosti koje su navedene u tablici 7.4.

Tablica 7.4. Vjerovatnosti blokiranja proizvođača $p(i > N)$ u ovisnosti o N i ρ

N	M	ρ			
		0.2	0.4	0.6	0.8
0	1	0.2	0.4	0.6	0.8
1	2	0.04	0.16	0.36	0.64
2	3	0.008	0.064	0.216	0.512
4	5	0.00032	0.01024	0.07776	0.32768
8	9	5.12E-07	2.62E-04	0.0101	0.13422
16	17	1.31E-12	1.72E-07	1.69E-04	0.02252

7.4. Osnovni načini dodjeljivanja procesora dretvama

7.4.1. Dodjeljivanje po redu prispijeća

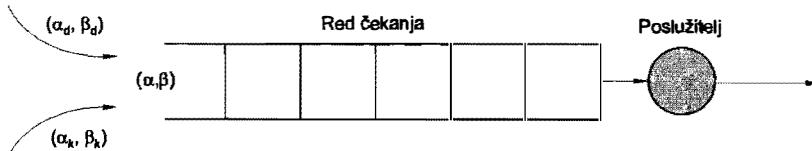
U 5. smo poglavlju opisali način djelovanja jezgre i ustanovili smo da sve dretve koje su pripravne za izvođenje smještamo u red pripravnih dretvi. U načelu smo pretpostavili da se pripravne dretve stavlaju u redove po redu prispijeća ili po pripisanim im prioritetima. Mi ćemo ovdje na temelju rezultata koje smo dobili u prethodnom odjeljku pokušati doći do nekih osnovnih spoznaja o ponašanju sustava za neke načine raspoređivanja procesora dretvama.

Kako bismo olakšali razmatranje pretpostaviti ćemo da se u sustav pripuštaju dvije skupine poslova:

- dugi poslovi s parametrima (α_d, β_d) i
- kratki poslovi s parametrima (α_k, β_k) .

Ako pretpostavimo da su dugi poslovi mnogo dulji od kratkih, onda je β_d mnogo manje od β_k (prosječno trajanje poslova jednako je recipročnim vrijednostima tih parametara).

Ako te dvije skupine poslova pomiješamo i pustimo u zajednički red, dobit ćemo mješavinu poslova koja se može karakterizirati zajedničkim parametrima (α, β) , kao što to ilustrira slika 7.10.



Slika 7.10. Miješanje kratkih i dugih poslova

Mješavina poslova imat će dolaske također raspodijeljene po Poissonovoj razdiobi s prosječnim brojem dolazaka u jedinici vremena jednakim:

$$\alpha = \alpha_k + \alpha_d.$$

Isto tako, mješavina poslova imat će eksponencijalnu razdiobu trajanja poslova, s tim da se njihovo prosječno trajanje određuje zbrajanjem prosječnog trajanja skupina pomnoženih s udjelom pojedine skupine u ukupnom broju poslova. Tako se dobiva:

$$\frac{1}{\beta} = \frac{\alpha_k}{\alpha_k + \alpha_d} \cdot \frac{1}{\beta_k} + \frac{\alpha_d}{\alpha_k + \alpha_d} \cdot \frac{1}{\beta_d}$$

ili:

$$\frac{1}{\beta} = \frac{1}{\alpha} \left(\frac{\alpha_k}{\beta_k} + \frac{\alpha_d}{\beta_d} \right) = \frac{1}{\alpha} (\rho_k + \rho_d),$$

iz čega slijedi da je:

$$\frac{\alpha}{\beta} = \rho = \rho_k + \rho_d.$$

PRIMJER 7.3.

Promotrimo dvije skupine poslova koje možemo nazvati kratkim i dugim poslovima. Svojstva tih dvaju skupina prikazana su tablicom 7.5. U tablici se osim toga nalaze i parametri mješavine tih dvaju skupina poslova.



Opravdanje za nazine kratki i dugi poslovi vidljivo je iz vremena prosječnog trajanja poslova $1/\beta$. Za duge poslove ono je 2500 puta dulje.

Tablica 7.5. Parametri skupine kratkih i dugih poslova i njihove mješavine

	samo kratki poslovi	samo dugi poslovi	mješavina poslova
α	10	0.01	10.01
β	50	0.02	14.3
$1/\beta$	0.02	50	0.06993
ρ	0.2	0.5	0.7
\bar{T}	0.025	100	0.233
\bar{n}	0.25	1	2.33

Usporedimo trajanje zadržavanja u sustavu samo za kratke poslove, samo za duge poslove te za mješavinu poslova. Prosječno vrijeme zadržavanja u sustavu za same duge poslove iznosi 100 jedinica vremena, za kratke poslove 0.025 jedinica vremena, dok je prosječno zadržavanje u sustavu mješavine poslova 0.233 jedinica vremena. Pogrešno je iz toga zaključiti da će se vrijeme obrade dugih poslova skratiti unošenjem kratkih poslova u sustav. Mi naprsto računamo prosjek zadržavanja u sustavu i za duge i za kratke poslove (kojih ima tisuću puta više).

Iz gornjeg primjera može se zaključiti kako djeluje umetanje dugih poslova u skupinu kratkih. Učinak umetanja dugih poslova u skupinu kratkih jest povećanje prosječnog vremena zadržavanja u sustavu za gotovo deset puta. Međutim, kada promatramo zadržavanje pojedinačnih kratkih poslova u sustavu, taj je učinak još mnogo drastičniji. Tako će, primjerice, kratki posao koji uđe u sustav neposredno iza nekog dugog posla biti obrađen tek nakon tog dugog posla i njegovo se vrijeme zadržavanja može produžiti i tisuću puta.

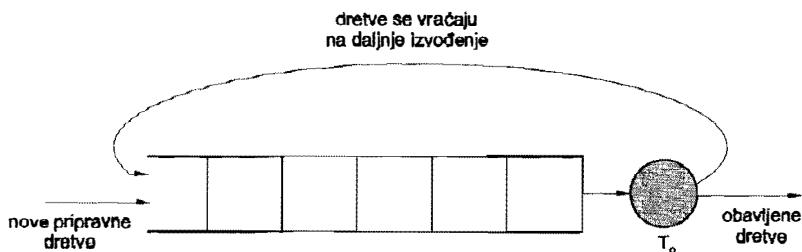
7.4.2. Kružno dodjeljivanje procesora

U stvarnim se uvjetima u računalu može istodobno naći više dretvi različita trajanja za koje mi unaprijed i ne moramo znati koliko će trajati njihovo izvođenje. Prethodno nam razmatranje ukazuje da će sve dretve koje postaju pripravne nakon neke dretve duga trajanja biti drastično odgodene u svom izvođenju. Međutim, znamo da u ovom slučaju, kada su poslovi koje promatramo dretve i poslužitelj je procesor, ne moramo dopustiti dretvama izvođenje do njihova završetka. Svaka promjena stanja dretvi i svaki prekid i tako zahtijeva "oduzimanje" procesora dretvi koja se izvodi kako bi se obavili kućanski poslovi jezgre. Tako imamo priliku organizirati redove pripravnih dretvi i dodjeljivanje procesora i po nekom drugom redu, a ne samo po redu prispjeća.

U odjeljku 5.2. ustanovili smo da dretve možemo razvrstati po redu prvenstva ako su dretvama pridruženi odgovarajući prioriteti. U našem razmatraju kratkih i dugih poslova mogli bismo kratke poslove smatrati hitnim dretvama, a duge poslove normalnim dretvama. Normalne dretve prekidaju se kada neka hitna dretva postaje pripravna.

Ovakva disciplina posluživanja može postati "nepravedna" prema dugim poslovima jer sve kratke dretve čim postaju pripravne prekidaju dugi posao. Duga dretva će se nastaviti izvoditi tek kada u redu više nema ni jedne kratke pripravne dretve. Prema tome, sve nove kratke dretve odgađaju dugu dretvu i njezino izvođenje.

Jedan od vrlo prikladnih načina dodjele procesora jest kružno posluživanje dretvi (engl. *round robin*). Osnovna je zamisao tog načina posluživanja da se procesor dodjeljuje jednoj dretvi (općenito: poslužitelj dodjeljuje jednom poslu) samo za određeni kvant vremena T_q . Ako je dretva u tom kvantu vremena obavljena, ona napušta procesor. Međutim, ako ona nije izvedena do kraja, ona se vraća na kraj reda pripravnih dretvi.



Slika 7.11. Kružno posluživanje dretvi

Slika 7.11. ilustrira i opravdava takav način posluživanja dretvi. U red pripravnih dretvi ulaze nove dretve i dretve vraćene na daljnju obradu. Time će se kratke dretve malo usporiti u svom izvođenju, ali duga dretva ima mogućnost dobiti procesor i prije nego li sve kratke dretve budu obavljene.

Pogledat ćemo način djelovanja kružnog posluživanja na raspoređivanje mješavine determinističkih dugih i kratkih poslova.

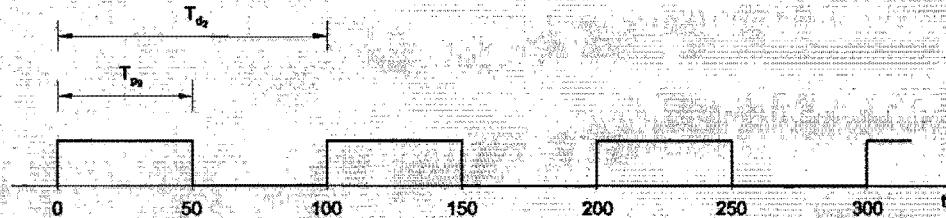
**PRIMJER 7.4.**

Pretpostavit ćemo da u sustav ulaze i miješaju se dvije skupine poslova. Dolasci i trajanja neka budu deterministički, i to tako da su im dolasci periodni i trajanja konstantna. Takve se skupine poslova mogu opisati vremenima između dva dolaska T_d i trajanjem posla T_p .

Odabrat ćemo ta vremena tako da odgovaraju prosječnim vremenima za kratke i dugе poslove iz primjera 7.3. tako da je:

$$\begin{aligned} T_{d1} &= 0.1, & T_{p1} &= 0.02, \\ T_{d2} &= 100, & T_{p2} &= 50. \end{aligned}$$

Ocjenu ponašanja sustava možemo načiniti tako da promatramo jednu periodu dugih poslova koja je ujedno i perioda za mješavinu kratkih i dugih poslova.



Slika 7.12. Perioda dugih poslova ujedno je i perioda mješavine

Faktori iskorištenja poslužitelja za pojedine skupine poslova jesu:

$$\rho_1 = \frac{T_{p1}}{T_{d1}} = \frac{0.02}{0.1} = 0.2, \quad \rho_2 = \frac{T_{p2}}{T_{d2}} = \frac{50}{100} = 0.5.$$

Za mješavinu poslova dobit ćemo zajednički faktor iskorištenja:

$$\rho = \rho_1 + \rho_2 = 0.2 + 0.5 = 0.7.$$

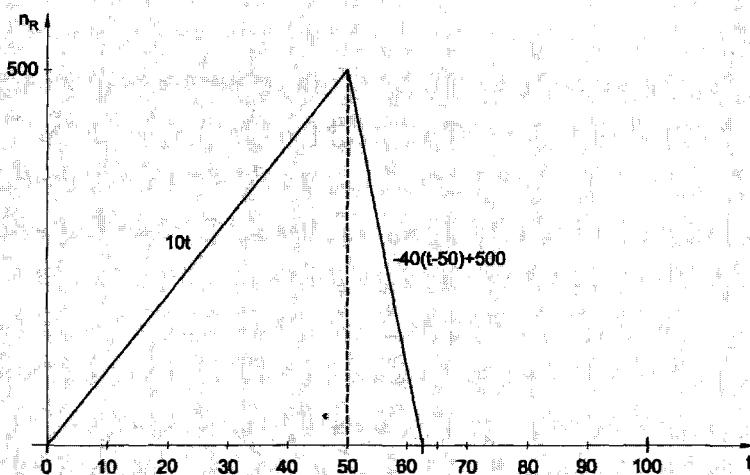
Jednaku vrijednost dobit ćemo i promatranjem jedne periode mješavine trajanja $T_{d2} = 100$. U toj periodi poslužitelj mora obraditi jedan dugi posao trajanja 50 i 1000 poslova trajanja 0.02, te je

$$\rho = \frac{50 + 1000 \cdot 0.02}{100} = 0.7.$$

Promotrimo najprije način posluživanja poslova po redu prispijeća.

Neka u trenutku $t = 0$ u prazni sustav uđe dugi posao i odmah iza njega prvi kratki posao. Dugi posao prolazi kroz prazni red i odmah ga poslužitelj počinje obradivati. Sljedećih 50 jedinica vremena ni jedan posao neće napustiti sustav, a u njega će ulaziti kratki poslovi s razmakom od 0.1 jedinica vremena ili 10 poslova u jedinici vremena. Prema tome, broj poslova u redu n_r rast će linearno s vremenom te je:

$$n_r = 10t.$$



Slika 7.13. Broj poslova u redu

Na slici 7.13. taj je prirast prikazan dijeom pravca iako je prirast diskretan i trebao biti prikazan stepeničastom funkcijom. U trenutku $t = 50$ u redu imamo 500 poslova.

Nakon što dugi posao napusti sustav poslužitelj počinje prazniti red brzinom od 50 poslova u jedinici vremena, no u sustav i dalje pristižu novi kratki poslovi brzinom od 10 poslova u jedinici vremena. Prema tome, red će se prazniti brzinom od 40 poslova u jedinici vremena. On će se potpuno isprazniti za:

$$\Delta t = \frac{500}{40} = 12.5 \text{ jedinica vremena}$$

te će nakon trenutka $t = 62.5$ jedinica vremena red biti prazan. Naime, u razdoblju od toga trenutka do kraja perioda, tj. do trenutka 100 u sustav ulaze samo kratki poslovi i oni odmah kroz prazni red ulaze u poslužitelj, tako da je red u tom razdoblju prazan.

Prosječna duljina reda u jednoj periodi može se dobiti tako da se trokutna površina podijeli s trajanjem periode, te se dobiva:

$$\bar{n}_r = \frac{500 \cdot 62.5}{2 \cdot 100} = 156.25$$

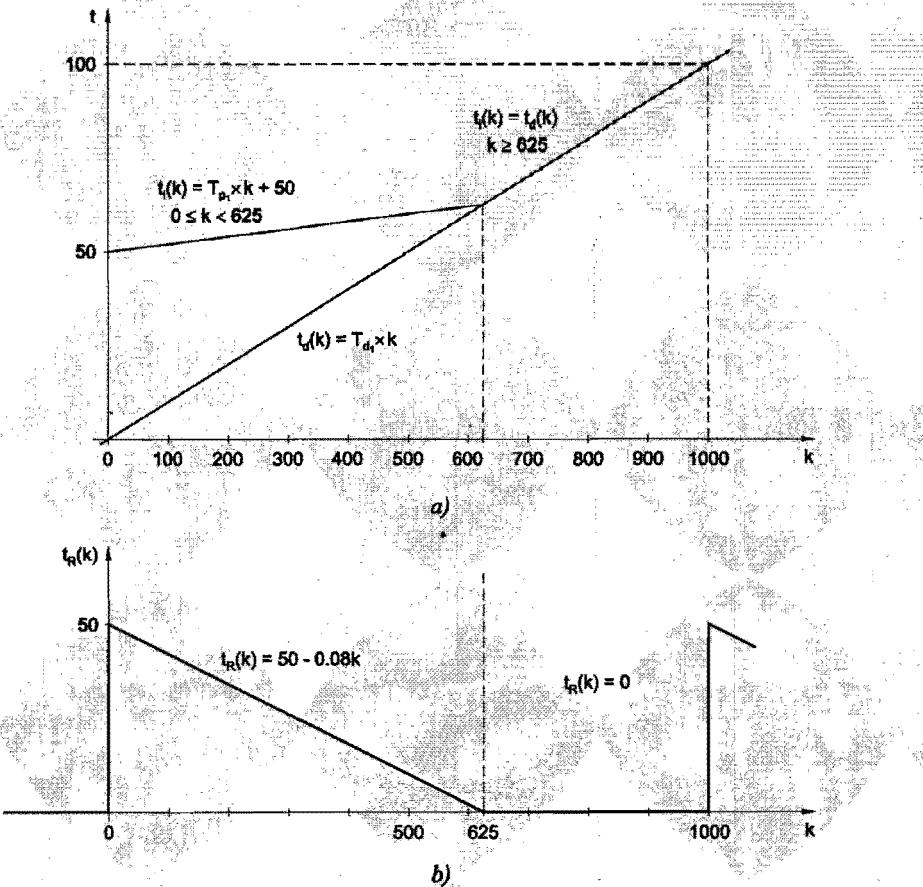
Kako bismo dobili prosječni broj poslova u sustavu moramo tom prosječnom broju pribrojiti i prosječni broj poslova u procesoru. U razdoblju trajanja 62.5 jedinice vremena, kada u redu ima poslova, uvek se u poslužitelju nalazi jedan posao. Nakon što se red isprazni u sustav će do kraja periode ući još $37.5 \cdot 10 = 375$ kratkih poslova, te će procesor biti zauzet još dalnjih $375 \cdot 0.02 = 7.5$ jedinica vremena, tako da će se tijekom cijele periode od 100 jedinica vremena u procesoru nalaziti jedan posao tijekom $62.5 + 7.5 = 70$ jedinica vremena, što znači da je prosječni broj poslova u procesoru tijekom cijele periode jednak 0.7. Taj broj već znamo jer je on jednak faktoru iskorištenja poslužitelja.

Prema tome, prosječni broj poslova u sustavu jednak je:

$$\bar{n} = \bar{n}_r + 0.7 = 156.95.$$

Preostaje nam odrediti još i trajanje zadržavanja poslova u redu, odnosno u sustavu. Pretpostavimo da dugi posao, koji ulazi na početku svake periode, uvijek preteke kratki posao koji ulazi zajedno s njim i tako odmah prolazi kroz prazni red bez zadržavanja. Treba još odrediti remena zadržavanja u redu za kratke poslove.

S obzirom na to da trajanje zadržavanja u redu nije jednako za sve kratke poslove, morat ćemo ih nekako razlikovati te ćemo im pridjeliti indeks k po redu prispjeća u sustav. U periodi od 100 jedinica vremena ući će ih u sustav ukupno 1000. Kratke poslove možemo podijeliti u dvije skupine: skupinu poslova koji će ulaziti u sustav tijekom razdoblja kada u redu ima poslova i skupinu poslova koji u sustav ulaze tijekom razdoblja kada je red prazan. Druga skupina poslova imat će vrijeme zadržavanja u redu jednako nuli jer će odmah nakon ulaska u sustav biti prihvatićeni od poslužitelja.



Slika 7.14. Zadržavanje kratkih poslova u redu
a) način određivanja
b) zadržavanje u redu za dvije skupine kratkih poslova

Slika 7.14. prikazuje način određivanja zadržavanja u redu. Na slici 7.14.a) je vidljivo da su trenuci dolaza pojedinih poslova određeni pravcem (i ovdje ćemo dis etne vrijednosti aproksimirati kontinuiranim):

$$t_d(k) = T_{d1} \cdot k.$$

Poslovi ne izlaze iz reda 50 jedinica vremena dok se u poslužitelju obrađuje dugi posao, a nakon toga odlaze iz reda brzinom određenom trajanjem posluživanja T_{p1} , tj. trenuci izlazaka iz reda određeni su pravcem:

$$t_i(k) = 50 + T_{p1} \cdot k.$$

Ta se dva pravca sijeku za vrijednost $k = 625$, nakon koje se trenuci izlaska iz reda poklapaju s trenucima dolazaka.

Prema tome, zadržavanje kratkih poslova u redu može se prikazati kao razlika tih dvaju pravaca i za naš primjer dobivamo sliku 7.14.b). Iz te se slike može zaključiti da je prosječno trajanje zadržavanja u redu jednako:

$$\bar{T}_R = \frac{50 \cdot 625}{2 \cdot 1001} = 15.61.$$

Prosječno zadržavanje u sustavu možemo dobiti tako da ovom zadržavanju u redu pridodamo prosječno zadržavanje u poslužitelju. Poslužitelj obrađuje u periodi od 100 jedinica vremena jedan dugi posao i tisuću kratkih, što na daje:

$$\bar{T} = \frac{1 \cdot 50 + 1000 \cdot 0.02}{1001} + \bar{T}_R \approx 15.68.$$

Može se utvrditi da vrijedi Littleovo pravilo uzimemo li da je za mješavinu poslova:

$$\alpha = \frac{1}{T_{d1}} + \frac{1}{T_{d2}} = 10 + 0.01 = 10.01.$$

Sada ćemo pogledati kakve ćemo rezultate dobiti ako na ovaj sustav primijenimo kružno posluživanje. Pretpostavimo da je kvant vremena koji se dodjeljuje jednom poslu jednak:

$$T_q = 0.01.$$

Prema tome, kratki poslovi utrošit će za svoje izvođenje 2 kvanta vremena, a dugi poslovi 5000 kvanta. Mi ćemo ovdje zanemariti trajanje promjene konteksta. Pretpostaviti ćemo, nadalje, zbog jednostavnosti razmatranja da se početak perioda kvantiziranja poklapa s trenutkom dolaska novog posla. Jednu periodu od 100 jedinica vremena možemo opet podijeliti na dva razdoblja: prvo razdoblje u kojem se zajedno s kratkim poslovima obrađuje i dugi posao i drugo razdoblje u kojem se obrađuju samo kratki poslovi.

Na slici 7.15. prikazan je interval između dvaju dolazaka kratkih poslova. Ako se novoprdošli posao odmah pusti u poslužitelj za jedan kvant vremena (a dugi se posao prekida i stavlja u red), onda se između dvaju dolazaka kratkih poslova u prvom raz-

dobiju podjela poslužitelja na kratke poslove i jedan dugi obavlja prema slici 7.15.a). S obzirom na to da dugi posao traje ukupno 50 jedinica vremena, a u periodi određenoj kao razmak između dva dolaska kratkih poslova dobiva 0.08 jedinica vremena, dugi posao bit će obavljen za 625 takvih perioda.

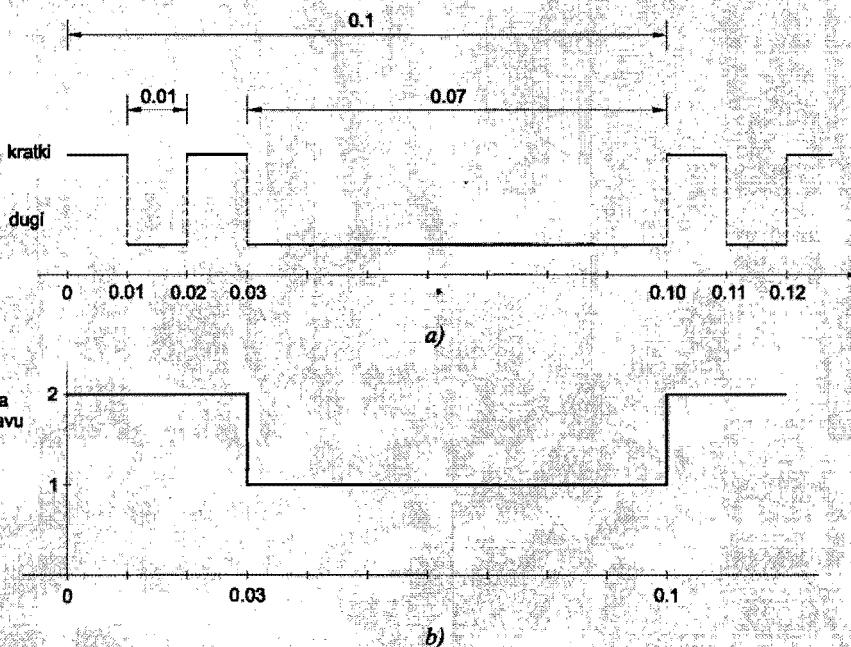
Iz slike je vidljivo da će se 625 kratkih poslova zadržavati u sustavu 0.03 jedinice vremena. Dugi posao napustit će sustav kada protekne tih 625 perioda od po 0.1 jedinice vremena, tj. zadržavat će se u sustavu 62.5 jedinica vremena. Ostatak od 375 kratkih poslova potrošit će dva uzastopna kvanta poslužitelja i zadržat će se u sustavu samo onoliko vremena koliko je potrebno za njihovu obradu, tj. 0.2 jedinice vremena.

Prema tome, dobit ćemo prosječno zadržavanje poslova u sustavu ovako:

$$\bar{T} = \frac{1 \cdot 62.5 + 625 \cdot 0.03 + 375 \cdot 0.02}{1001} = \frac{88.75}{1001} = 0.08866.$$

Prosječni broj poslova u sustavu dobit ćemo sljedećim razmatranjem. U prvom razdoblju od 625 perioda od po 0.1 jedinice vremena tijekom 0.03 jedinice vremena u sustavu se nalaze dva posla, a u ostalih 0.07 jedinica jedan posao, kao što prikazuje slika 7.15.b). Nakon tog razdoblja u preostalih 37.5 jedinica vremena u sustav će ući još 375 poslova i zadržati se po 0.02 jedinice vremena. Prema tome, dobivamo:

$$\bar{n} = \frac{625 \cdot (0.03 \cdot 2 + 0.07 \cdot 1) + 375 \cdot 0.02}{100} = 0.8875.$$



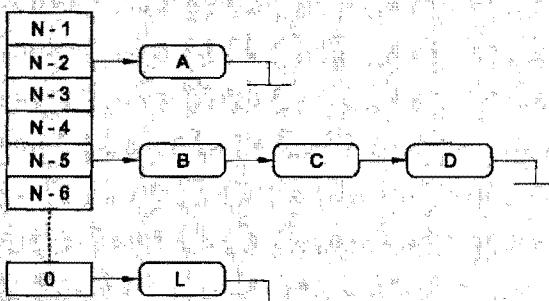
Slika 7.15. a) Obrada kratkih poslova i dugog posla između dva dolaska kratkih poslova
b) Broj poslova u sustavu

Nedostatak je kružnog posluživanja intenzivna promjena konteksta, koju smo mi u gornjem primjeru zanemarili. Međutim, ipak je to jedna od najprikladnijih disciplina posluživanja. U operacijskim sustavima ona se upotrebljava za raspoređivanje većine dretvi. Iznimku čine dretve koje obavljaju vremenski kritične funkcije, koje moraju biti obavljene u nekom zadanom vremenskom intervalu. Takve se dretve obrađuju s najvišim prioritetom, i to bez prekidanja. Dretve koje se prekidaju i ulaze u kružno posluživanje mogu također imati različite prioritete. Viši prioritet može doći do izražaja tako da se dretvi dodjeljuje dva ili više uzastopnih kvanta procesorskog vremena. U suvremenim se operacijskim sustavima na temelju praćenja odvijanja poslova dretvama prioriteti namještaju i dinamički.

PRIMJER 7.5.



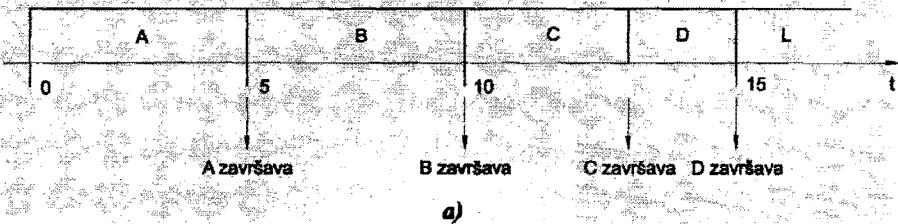
Raspoređivanje ostvareno u većini operacijskih sustava koristi već spomenuto raspoređivanje po redu prispijeća (FIFO) i kružno posluživanje (RR), ali uzimajući u obzir i prioritete dretvi. Kao što je spomenuto u 5. poglavljiju, pripravne dretve razvrstane su u liste prema njihovu prioritetu – za svaki prioritet po jedna lista (koja može biti i prazna). Neka je stanje nekog sustava prikazano pripravnim dretvama A, B, C i D različita prioriteta prema slici 7.16.

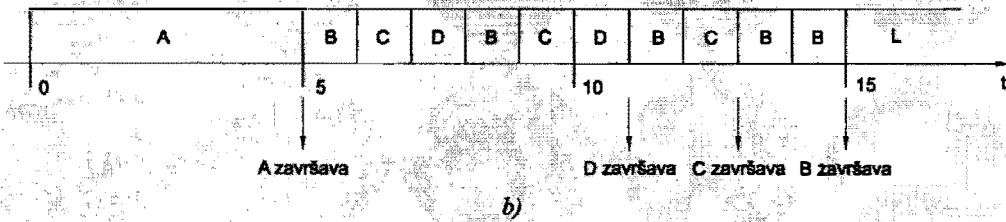


Slika 7.16. Pripravne dretve razvrstane prema prioritetu

Dretva L jest latentna dretva koja će se izvoditi kada nema niti jedne druge pripravne dretve u sustvu, te joj kao takvoj pripada najmanji prioritet.

Raspoređivanje po redu prispijeća kao i kružno posluživanje u prethodno opisanom načinu koristi se samo za dretve istog prioriteta. Modifikacija raspoređivanja u sustavu s dretvama različita prioriteta sastoji se u tome da se najprije raspoređuju dretve najvećeg prioriteta. Tek kada takvih dretvi ponestane u redu pripravnih, odabiru se pripravne dretve prvog nižeg prioriteta. Uz prepostavku da su trajanja izvođenja dretvi sa slike 7.16. $T_P(A) = 5$, $T_P(B) = 5$, $T_P(C) = 3$ te $T_P(D) = 2$ jedinice vremena, FIFO i RR raspoređivanje će za taj sustav izgledati kao na slici 7.17.a) i 7.17.b).





Slika 7.17. FIFO i RR rasporedišvanje

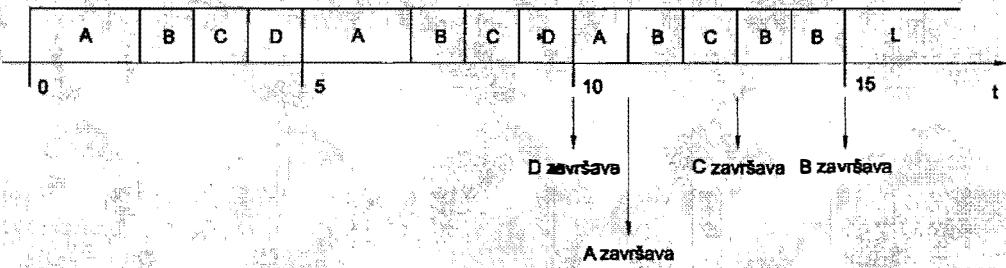
- a) Rasporedišvanje po redu prispijeća (FIFO)
b) Kružno rasporedišvanje (RR)

Kod raspredišanja po redu prispijeća, nakon što dretva A završi, uzima se dretva B i izvodi do njenog završetka. Tek potom dretva C pa D. Nakon toga ostaje jedino latentna dretva L.

Kod kružnog rasporedišvanja, nakon što dretva A završi, dretve B, C i D kružno se poslužuju, dobivajući po kvantu vremena (u primjeru uzeto $T_Q = 1$ jedinica vremena) sve dok ne završe.

Osim navedenih načina rasporedišvanja, u suvremenim operacijskim sustavima primjenjuje se i rasporedišvanje kod kojeg prioritet dretve označava udio procesorskog vremena koji će dretva dobiti. Veći prioritet ne znači, kao u prethodnom primjeru, sigurni dovršetak prije neke druge dretve manjeg prioriteta, već samo veći dio procesorskog vremena od dijela koji će dobiti dretva manjeg prioriteta. Ostvarenje navedenog načela može se napraviti na razne načine: dajući dretvama višeg prioriteta veći broj kvanata vremena nego manje prioritetskim dretvama. Ili, koristiti različita trajanja kvanata za dretve različitih prioriteta: dretvama višeg prioriteta dati dulji kvant vremena nego dretvama manjeg prioriteta.

Slika 7.18 prikazuje jedno moguće ostvarenje takvog sustava, gdje se prema prioritetu određuje duljina kvanata vremena za pojedine dretve: $T_Q(A) = 2$, $T_Q(B) = 1$, $T_Q(C) = 1$ i $T_Q(D) = 1$.



Slika 7.18. Rasporedišvanje raspodjeljom vremena prema prioritetu

Dretva A, ja oima najveći prioritet, neće zauzeti svo procesorsko vrijeme, već će dio vremena prepustiti i dretvama manjeg prioriteta. Latentna dretva i dalje ne dobiva procesorsko vrijeme dok ima drugih pripravnih dretvi jer ona ne obavlja nikakav koristan posao. Za potrebe razmatranja ovakvog načina rasporedišvanja možemo reći da je njen kvant jednak nuli.

**PITANJA ZA PROVJERUZNANJA 7**

1. Objasniti parametre sustava i njihov odnos: iskoristivost, broj poslova koji ulazi u sustav, broj poslova koje poslužitelj može obaviti u jedinici vremena, prosječni broj poslova u sustavu, prosječno zadržavanje posla u sustavu.
2. Navesti Littleovo pravilo.
3. Skicirati Markovljev lanac gdje stanja predstavljaju broj poslova u sustavu. Sustav neka ima Poissonovu razdiobu dolazaka s parametrom α i eksponencijalnu razdiobu trajanja obrade s parametrom $1/\beta$.
4. Neka u nekom sustavu zahtjevi za obradu podliježu Poissonovoj razdiobi, a vrijeme obrade neka ima eksponencijalnu razdiobu. Kolika je vjerojatnost da u nekom trenutku u sustavu bude točno 7 poslova ako je iskoristivost sustava $\rho = 0.7$?
Koliki je prosječni broj poslova u takvom sustavu?
Koliko je prosječno vrijeme zadržavanja posla u sustavu ako u sustav ulazi prosječno 5 poslova u jedinici vremena?
5. U kojem se slučaju može dopustiti veliki faktor iskorištenja (čak i 1)?
6. Neka u sustav ulazi dvije vrste poslova: kratki s parametrima: vrijeme između dva dolaska $T_{D1} = 0.1$ ms i trajanje posla $T_{P1} = 0.02$ ms te dugački poslovi s parametrima $T_{D2} = 100$ ms i $T_{P2} = 50$ ms.
Koliko za takav sustav iznose parametri α , β i ρ ?
Koliki je prosječni broj poslova u sustavu i prosječno zadržavanje poslova u sustavu ako poslovi dolaze u sustav prema Poissonovoj razdiobi, a vrijeme njihove obrade ima eksponencijalnu razdiobu?

8.

Gospodarenje spremničkim prostorom

8.1. Uvodna razmatranja

U drugom smo poglavlju, pri razmatranju Von Neumannova modela računala, ustanovili da je radni ili središnji spremnik računala stjecište svih informacija koje kolaju između ostalih dijelova računala. Podsetimo se da se u Von Neumannovu modelu računala sve adrese generiraju unutar procesora (iznimno se adrese generiraju u pristupnim sklopovima s neposrednim pristupom spremniku). U načelu adrese koje se generiraju unutar procesora jesu:

- adrese instrukcija koje dolaze iz programskog brojila,
- stogovne adrese koje dolaze iz registra kazaljke stoga i
- adrese operanada i rezultata operacija (adrese podataka) koje se stvaraju na temelju sadržaja adresnih dijelova instrukcija.

Nadalje, u četvrtom smo poglavlju, pri opisivanju procesa i dretvi procesa, ustanovili da svaki proces djeluje u svom vlastitom spremničkom prostoru. Ustanovili smo isto tako da se svaki cijeloviti program izvodi kao računalni proces, koji mora imati barem jednu dretvu. Ako se proces sastoji od više dretvi, onda se spremnički prostor procesa dijeli na dretvene prostore i jedan zajednički prostor u koji mogu pristupiti sve dretve. Unutar svakog dretvenog potprostora može se prepoznati instrukcijski dio dretve, stogovni dio dretve i podatkovni dio dretve. Zajednički spremnički prostor procesa sadrži zajedničke podatke, kao i jedan "prazni" dio – hrpu spremničkih lokacija (engl. *heap*) odakle dretve mogu dinamički dobavljati potrebne spremničke lokacije i kuda se vraćaju ispraznjene spremničke lokacije.

Veličina adresnog prostora cijelokupnog procesa određena je predviđenim brojem adresnih bitova m koji je utvrđen arhitekturom upotrijebljenog procesora. U odjeljku 2.1. spomenuto je da se u današnjim osobnim računalima pretežito upotrebljavaju procesori s arhitekturom koja podržava tridesetdvobitovno adresiranje, a radne stanice zasnovane su na šezdesetčetverobitovnoj arhitekturi. Slika 2.5. ilustrira model tridesetdvobitovne arhitekture u kojoj je prepostavljeno da je cijeli adresni prostor "popunjeno" fizičkim lokacijama

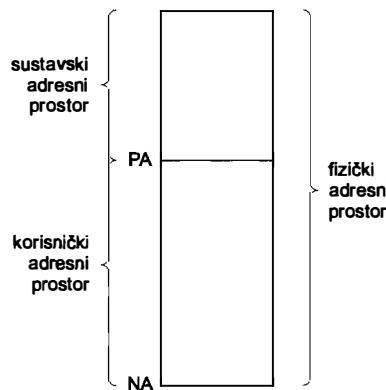
– bajtovima – u koje se mogu pohranjivati okteti bitova. Ostvarenje tog modela stvarnim računalom zahtijevalo bi veličinu radnog spremnika od 4 GB.

Ako je stvarno ugrađeni radni spremnik računala, možemo reći fizički spremnik, manji od maksimalno mogućeg (primjerice: 512 MB ili 1 GB), onda u prvi mah zaključujemo da se u takvom računalu mogu izvoditi samo oni procesi čiji adresni prostor ne premašuje tu veličinu. Drugim riječima, od mogućih 32 bita adrese tridesetdvobitovne arhitekture koristilo bi se u takvim računalima 29, odnosno 30 bitova ($2^{29} = 512 \cdot 2^{20} = 512\text{ M}$, odnosno $2^{30} = 1024 \cdot 2^{20} = 1\text{ GB}$).

Ne zaboravimo da svi programi, bez obzira na programski jezik u kojem su izvorno pripremljeni, moraju prije izvođenja biti prevedeni u strojni oblik i mora ih se pohraniti u radni spremnik u obliku niza strojnih instrukcija koje prepoznaće procesor.

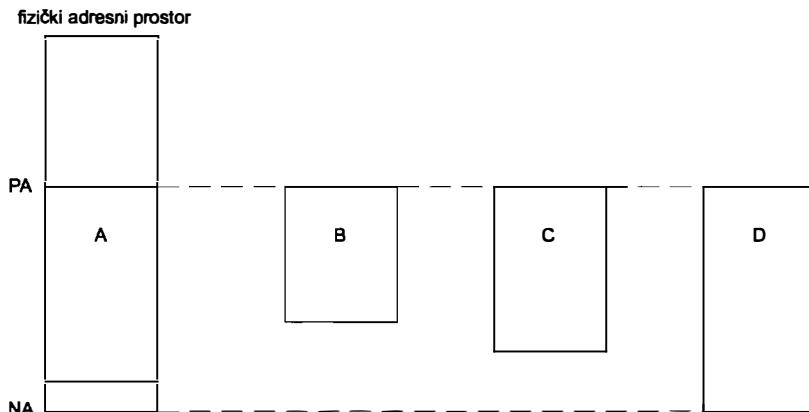
Prepostavit ćemo da promatramo najjednostavniji slučaj kada se u računalu izvode pojedinačni programi.

Dio radnog spremnika mora se rezervirati za strukture podataka i funkcije operacijskog sustava. Taj smo dio prostora nazvali *sustavskim adresnim prostorom*. Prema tome, samo jedan dio fizičkog spremnika preostaje za smještanje programa. Taj preostali dio fizičkog spremnika ostaje za uporabu u korisničkom načinu rada i određuje maksimalni mogući procesni adresni prostor, kao što prikazuje slika 8.1.



Slika 8.1. Fizički adresni prostor stvarnog računala

Svi programi koji će se izvoditi u takvom računalu moraju biti pripremljeni tako da se mogu smjestiti u *korisnički adresni prostor*. Osim toga, sve adrese unutar programa moraju biti u granicama između *PA* (početna adresa korisničkog adresnog prostora) i *NA* (najveća adresa korisničkog adresnog prostora) kako prikazuje slika 8.2. Program *A* smješten je u fizički spremnik i njegov se proces obavlja. Programi *B*, *C* i *D* čekaju na izvođenje. Jedan od njih bit će prvi odabran za izvođenje i smješten u radni spremnik kada program *A* bude završio svoje izvođenje. Svi programi moraju se u obliku pripravnom za izvođenje pohraniti u neki *pomoćni spremnik* (engl. *backing store*) ili *dopunski vanjski spremnik*, kako bi se mogli što brže premjestiti u radni spremnik kada to bude potrebno.



Slika 8.2. Programi pripremljeni za izvođenje

S obzirom na to da već i u ovom najjednostavnijem obliku jednoprogramskog rada uočavamo važnost pomoćnog spremnika, prije opisa složenijih načina gospodarenja spremničkim prostorom proučit ćemo svojstva pomoćnih spremnika.

Danas se kao pomoći spremnici u prvom redu upotrebljavaju *magnetski diskovi*. Magnetski se diskovi koriste i za smještanje datoteka, te su osnovna sklopovska podloga za ostvarenje baza podataka, a time i informacijskih sustava.

Magnetski diskovi, prema tome, imaju dvojaku ulogu te služe:

- kao pomoći ili dopunski spremnici koji pri izvođenju programa nadopunjaju radni spremnik računala;
- kao skladište za trajno čuvanje svih vrsta datoteka.

Osim magnetskih diskova za trajno pohranjivanje i prenošenje velikih količina informacija upotrebljavaju se i *optički diskovi*. Međutim, oni se zbog načina pristupa pohranjenim podacima s današnjim tehnološkim rješenjima ne mogu upotrebljavati kao djelotvorni pomoći spremnici. Isto tako, za arhiviranje podataka upotrebljavaju se i magnetne vrpce koje zbog načina pristupa podacima također ne mogu poslužiti kao pomoći spremnici.

8.2. Osnovna svojstva magnetskih diskova

Magnetski diskovi proizvode se u dvije izvedbe:

- jednoj koju na hrvatskom zovemo *diskom* (govori se u skladu s engleskim i *tvrdi disk* ili *čvsti disk*) i
- drugoj koju na hrvatskom zovemo *disketom* (govori se skladu s engleskim i *meki disk* ili *savitljivi disk*) (engl. *hard disk* i *floppy disk*).

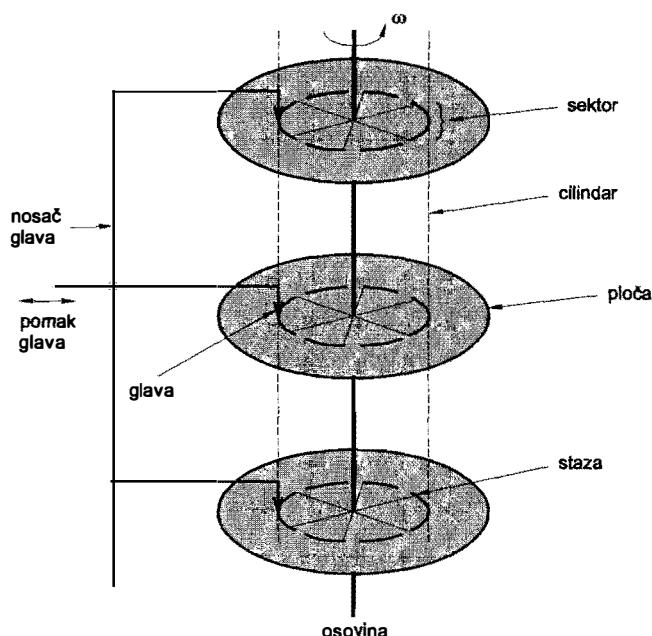
Diskovi se na računalo priključuju s pomoću prikladnog upravljačkog sklopovlja. Jedinka magnetskog diska (engl. *magnetic disk drive*) sastoji se od dvije komponente:

- elektromehaničkog dijela koji čine jedna okrugla ploča ili više njih presvučenih magnetskim materijalom koje se vrte konstantnom brzinom i mehanizma magnetskih glava koje se mogu pomicati približno radijalno iznad tih ploča;
- upravljačkog sklopa sastavljenog od mikroprocesora (mikroupravljača), spremnika, sučelja prema elektromehaničkom dijelu i sučelja prema sabirnici računala koje se ponaša kao pristupni sklop s neposrednim pristupom radnom spremniku (vidi odjeljak 3.4.).

Diskete imaju manji kapacitet i mogu se u disketnoj jedinki zamjenjivati. Njihovi su upravljački skloovi jednostavniji. One imaju samo povijesno značenje, jer se praktički više i ne koriste.

8.2.1. Organizacija zapisivanja sadržaja na disku

Detaljnije ćemo pogledati neka osnovna svojstva jedinki magnetskih diskova. Izgled magnetskih diskova prikazan je na slikama 8.3. i 8.4.

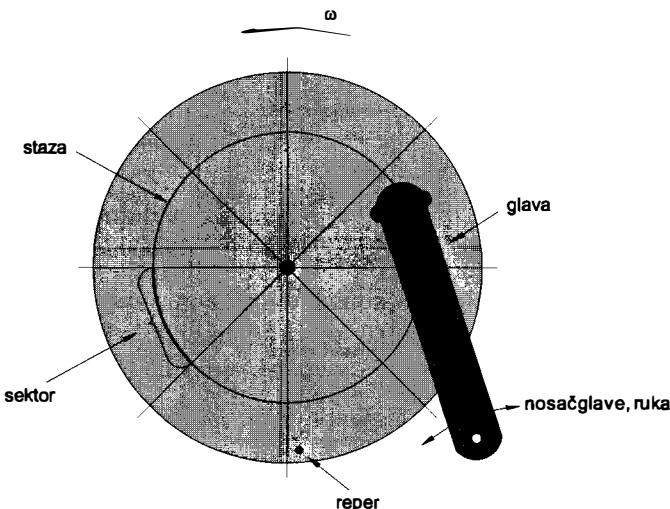


Slika 8.3. Mehanički izgled diska

Na slici 8.3. prikazan je disk s više ploča. Sve ploče učvršćene su na istu osovINU i vrte se konstantnom brzinom ω . U smjeru radiusa ploča (ili približno u smjeru radijusa kao što se vidi na slici 8.4., gdje se glava pomiče preko ploča u blagom luku) pomiču se glave za pisanje i čitanje. Glave se mogu vrlo precizno postaviti na jedno mjesto i tada ispod njih prolazi vrlo uski prsten ploče koji zovemo *stazom* (engl. track). Staze jednakih polumjera



svih diskova leže na zamišljenom plaštu valjka i zovemo ih zajedničkim imenom *cilindar* (engl. *cylinder*). Svaka staza podijeljena je na jednake dijelove (kružne lukove s jednim središnjim kutovima koje zovemo *sektorima* (engl. *sector*). S pomoću magnetskih glava promjenom se magnetskog toka upisuju bitovi u magnetski materijal površine ploče. Taj se sadržaj kasnije može pročitati jer magnetizirana površina diska inducira električke signale u glavi za čitanje iz kojih se razaznaje upisani sadržaj.



Slika 8.4. Gornji pogled na disk

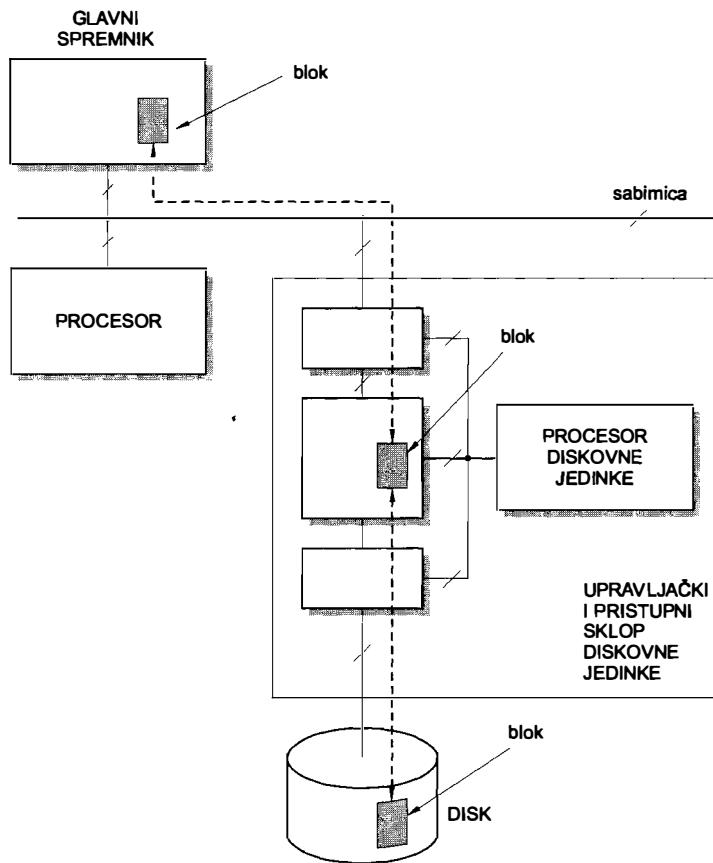
U jedan sektor može se pohraniti određeni broj bajtova. Uobičajeno je 256, 512 ili 1024 bajtova. Svaki sektor na disku ima svoju jedinstvenu adresu koja se izračunava iz:

- rednog broja ploče,
- rednog broja staze na ploči i
- rednog broja sektora na stazi.

Ploča se odabire aktiviranjem odgovarajuće glave, redni broj staze određuje se na temelju položaja glave, a redni broj sektora dobiva se tako da se mijere kutovi vrtnje počevši od posebne oznake na ploči koja označava početni kut.

Na disku se ne adresiraju pojedinačni bajtovi nego *blokovi bajtova*. Blokovi bajtova sastoje se od jednog ili više sektora. U uobičajenim izvedbama diskova svaka ploča ima svoju glavu za pisanje, odnosno čitanje, ali se u jednom času može pristupiti samo do jedne ploče, tj. može se čitati ili pisati samo u jednu stazu cilindra.

Upravljački sklop diskovne jedinice mora dakle omogućiti prijenos blokova. Skica upravljačkog sklopa diskovne jedinice zajedno s pristupnim sklopom prema sabirnici prikazana je slikom 8.5.



Slika 8.5. Priklučivanje diska

Upravljački program za upravljanje diskom prenosi iz glavnog spremnika na disk, ili obrnuto, blokove bajtova koji su jednake veličine kao i sektori diska ili im je veličina cje-lobrojni višekratnik veličine sektora. Pritom se programski mora odrediti početna adresa bloka u radnom spremniku s jedne strane, te redni broj sektora na disku s druge strane. Nakon toga može početi prijenos kojega pristupni sklop obavlja neposrednim pristupom do spremnika.

Gledano se strane sabirnice disk se može promatrati kao niz sektora definirane veličine do kojih se može pristupiti navođenjem njihova rednog broja – adrese sektora. Možemo reći da disk čini linearni adresni prostor sektora definirane veličine. Rekli smo da se ta jedinstvena adresa može preračunati u položaj sektora na disku određen rednim brojem ploče, staze i sektora. Međutim, u upravljačkom sklopu jedinke mogu se obaviti i složenija preslikavanja jedinstvene adrese u fizički sektor, tako da se pritom mogu sakriti i neki neispravni sektori ili cijele staze. Na taj se način prilikom završnih proizvodnih ispitivanja mogu isključiti neki loši dijelovi površine diskova.

Svojstva diskova s godinama se poboljšavaju. Spomenimo neka od važnijih svojstava diskova¹:

- promjeri ploča kreću se u granicama od 1 do 8 inča (tipični promjeri su: 1.0, 2.5, 3.5 i 5.25 inča);
- gustoća staza iznosi od 3000 do 250 000 staza po jednom inču radijusa;
- diskovna jedinka može imati od 1 do 20 ploča;
- broj staza na disku kreće se od 1500 do 400 000;
- kapaciteti diskova kreću se u granicama od nekoliko stotina megabajta do nekoliko terabajta;
- ploče diska čvrsto su ugrađene u diskovne jedinke i ne mogu se zamjenjivati.

8.2.2. Vremenska svojstva diskova

Vremenska svojstva diskova u pojednostavljenom se prikazu izražavaju nekim prosječnim vremenima za dohvrat jednog sektora na disku. Takav jedinstveni parametar može poslužiti kao osnovna mjera za uspoređivanje diskova.

Međutim, za malo složenije promišljanje ponašanja diskova i za eventualno pronalaženje rješenja za poboljšanje vremenskih svojstava diskovnih podsustava, korisno je razmotriti i malo detaljniji model vremenskog ponašanja diskova.

Vremenska svojstva diskova određena su pretežito brzinama pokretanja njegovih mehaničkih dijelova. Vrijeme potrebno za prijenos sadržaja nekog sektora s diska ili pohranjivanje sadržaja u taj sektor može se podijeliti na dva dijela, i to na:

- trajanje postavljanja glave (engl. *head positioning time*) i
- trajanje prijenosa podataka (engl. *data transfer time*).

Trajanje postavljanja glave može se nadalje podijeliti na dvije komponente, i to:

- trajanje traženja staze, ili kraće: trajanje traženja (engl. *seek time*) i
- rotacijsko kašnjenje (engl. *rotational latency*).

Trajanje traženja ovisi o razmaku između početnog položaja glave i staze na koju se glava mora postaviti. To trajanje ima četiri komponente: trajanje ubrzavanja ručice glave, kretanje konstantnom brzinom, trajanje usporavanja, trajanje finog pozicioniranja. Ako se glava pomiče samo za nekoliko staza, trajanje se uglavnom svodi na fino pozicioniranje. Kod kratkih pomaka ručica se u prvom dijelu ubrzava i mora se početi usporavati prije nego li dosegne maksimalnu brzinu. Za veće se pomake nakon početnog ubrzavanja dolazi u razdoblje konstantne brzine, nakon kojeg slijedi usporavanje i konačno fino pozicioniranje. Za pojedine od ovih pomaka mogu se koristiti pripadne formule za računanje trajanja pomaka. Uobičajeno se za vrijeme traženja uzima vrijeme koje je potrebno za prijelaz preko trećine ukupnog broja staza.

¹ Uobičajeno se za mjeru duljine pri opisu diskova koristi inč. Kod nas je zakonom propisana uporaba metričkog sustava, ali se ovdje iznimno odstupilo od preračunavanja u metrički sustav jer bi to moglo uzrokovati neke zabune.



Rotacijsko kašnjenje nastaje zbog toga što se nakon postavljanja glave na odabranu stazu mora prije početka prijenosa pričekati da se ispod glave za čitanje pojavi traženi sektor. U najlošijem slučaju može se dogoditi da je adresirani sektor upravo prolazio ispod glave kad je ona prispjela na stazu i tada se mora čekati puni okretaj. Najpovoljniji je slučaj onaj kada traženi sektor upravo nailazi ispod glave u trenutku njezina postavljanja na stazu. U tom je slučaju rotacijsko kašnjenje jednako nuli. Razumno je dakle za rotacijsko kašnjenje uzeti prosječnu vrijednost tih dvaju krajnjih slučajeva i prepostaviti da je rotacijsko kašnjenje prosječno jednako polovini trajanja jednog okretaja diska.

Trajanje prijenosa podataka određeno je brzinom prijenosa (engl. *data transfer rate*) i količinom prenesenih podataka. Brzina prijenosa određena je brzinom kojom ispod glave za čitanje ili pisanje promiču bajtovi sektora. Brzina prijenosa može se, dakle, izračunati tako da se broj sektora na stazi pomnoži s brojem bajtova u sektoru i taj umnožak podijeli s trajanjem jednog okretaja. Ovdje treba naglasiti da je unutar jednog sektora broj zapisanih bitova mnogo veći od broja bitova pohranjenih bajtova jer se uz informacijske bitove na disk pohranjuju i dodatni bitovi zaštitnog kodiranja. Pri određivanju brzine prijenosa, međutim, promatramo samo korisne informacijske bitove, odnosno bajtove.

S obzirom na to da se na disk nastoji pohraniti što je moguće više podataka, neki tipovi diskova izvode se tako da se na stazama većih promjera smješta više sektora jer se na taj način bolje iskorištava površina diska (uz istu gustoću zapisa na duljem kružnom luku može se pohraniti više bitova) pa i brzina prijenosa može biti promjenljiva.

Napredak u tehnologiji proizvodnje diskova omogućio je da se tijekom zadnjih desetak godina zamjetno povećao kapacitet diskova, ali su poboljšanja vremenskih svojstava mnogo skromnija. Prosječno trajanje postavljanja glava opalo je s vrijednosti od približno 20 ms na ispod 10 ms. Isto tako, tipična brzina okretaja povećana je s 3600 okretaja u minuti na 5400 do 15 000 okretaja u minuti. Brzine prijenosa povećale su se do oko 100 MB/s.

Na trajanje prijenosa djeluje i ponašanje upravljačkog sklopa diska. Kao što je vidljivo na slici 8.5., upravljački sklop diskovne jedinke mora, s jedne strane, podržavati komunikaciju s radnim spremnikom i s druge strane upravljati mehanizmom diska, te zapisivati i čitati podatke na ploču. Spremnik upravljačkog sklopa služi kao međuspremnik između radnog spremnika i samog magnetskog diska. Upravljački sklop obavlja sve detalje komuniciranja s diskom. U njemu se može obaviti i zaštitno kodiranje, odnosno dekodiranje i eventualno ponovno čitanje s diska kako bi se eliminirale moguće pogreške nastale od trenutačnih smetnji. Vidjet ćemo u sljedećem poglavlju, u kojem ćemo se baviti datotekama, da upravljački sklopoli diskova mogu obaviti i još mnogo drugih funkcija. Brzina prijenosa iz upravljačkog sklopa diska u radni spremnik ovisi i o brzini prijenosa na sabirnici, kao i o trenutačnom intenzitetu prometa na sabirnici². U najgorem slučaju upravljački sklop može djelovati tako da pridonosi ukupnom trajanju prijenosa dijelova milisekunde i mi ćemo ga u pojednostavljenom gledanju jednostavno zanemariti.

² U današnjim se računalima, uz glavnu sabirnicu, koristi posebna ulazno-izlazna sabirnica.

**PRIMJER 8.1.**

Za ocjenu utjecaja diskova na ponašanje sustava koristit ćemo jedan primjer diska. Čitatelj ože odgovarajućom zamjenom brojeva prilagoditi model svojim potrebama.

Pretpostaviti ćemo da disk ima:

- broj površina (glava)
- broj cilindara (saza na površini)
- broj sektora na stazi
- veličinu sektora

$$\begin{aligned}P &= 10, \\C &= 2000, \\S &= 100, \\M &= 512 \text{ B}.\end{aligned}$$

Prema tome, kapacitet K ovog diska bio bi jednak:

$$K = P \cdot C \cdot S \cdot M = 10 \cdot 2000 \cdot 100 \cdot 512 \text{ B} = 1 \text{ GB}.$$

Ako je brzina vrtnje diska jednaka:

$$\omega = 4800 \text{ okretaj/min} = 80 \text{ okretaj/s},$$

onda trajanje jednog okretaja T_R iznosi:

$$T_R = \frac{1}{\omega} = \frac{1}{80} \text{ s} = 12.5 \text{ ms}.$$

Prosječno rotacijsko kašnjenje za takav disk bilo bi jednako polovini tog vremena, tj. iznosilo bi 6.25 ms.

Brzinu prijenosa V_P možemo izračunati tako da broj bajtova jedne staze koji u jednom okretaju produ ispod glave za čitanje, odnosno pisanje podijelimo s trajanjem okretaja, pa za naš model diska dobivamo:

$$V_P = \frac{M \cdot S}{T_R} = M \cdot S \cdot \omega = 512 \cdot 100 \cdot 80 = 4096000 \text{ B/s} = 4.096 \text{ MB/s}.$$

Treba naglasiti da kod brzine prijenosa prefiks M ne označava faktor 2^{20} već uobičajeni faktor 10^6 .

Rekli smo da trajanje traženja staze možemo jednostavno izraziti kao neko prosječno trajanje prelaženja glava preko trećine svih staza diska ili ga preciznije opisati s nekoliko jednadžbi, ovisno o tome koliko se detaljno želi analizirati ponašanje diska (za mnoge se diskove i ne mogu pronaći odgovarajući podaci).

U našem ćemo modelu pretpostaviti da se trajanje traženja T_S može opisati s tri formule, ovisno o broju staza D koje glave moraju prijeći k ećući se od staze iznad koje se upravo nalaze da staze odakle moraju prenijeti sadržaj sljedećeg sektora, i to:

$$\begin{aligned}T_S &= 1.5 \cdot D \text{ ms} && \text{za } D \leq 4, \\T_S &= 4.0 + 0.5 \cdot \sqrt{D} \text{ ms} && \text{za } 4 \leq D \leq 400, \\T_S &= 10.0 + 0.01 \cdot D \text{ ms} && \text{za } D \geq 400.\end{aligned}$$

Prosječno vrijeme traženja za prelaženje preko trećine ukupnog broja staza dobili bismo iz treće formule i ono iznosi:

$$T_S = 10.0 + 0.01 \cdot \frac{2000}{3} \text{ ms} = 16.66 \text{ ms.}$$

Ovaj će nam model diska polužiti i u nekim kasnijim primjerima.

8.2.3. Disk kao dopunski spremnik radnom spremniku

U ovom ćemo poglavlju razmotriti samo uporabu diska kao pomoćnog ili dopunskog spremnika radnom spremniku. Način uporabe diska za pohranjivanje datoteka obradit ćemo u sljedećem poglavlju.

Pretpostavimo da su svi programi pripravni za izvođenje smješteni na vanjski pomoćni spremnik. Svaki od njih prije početka izvođenja mora biti prebačen u radni spremnik. Program će time postati proces sa svojim procesnim adresnim prostorom. Sve dretve koje pripadaju tom procesu koriste taj adresni prostor i mogu se u njemu nesmetano odvijati. Međutim, procesni adresni prostor ne može biti veći od fizički ostvarenog dijela spremnika predviđenog za korisnički način rada. Ako sve dretve procesa koji zauzima radni spremnik postanu blokirane (ili se zbog kružnog posluživanja želi oduzeti procesor tom procesu), trebat će pokrenuti neki drugi proces. U tom će se slučaju sadržaj radnog spremnika morati prebaciti na vanjski spremnik, a u radni spremnik iz vanjskog spremnika prebaciti niz instrukcija i podaci drugog procesa. U radnom se spremniku mora obaviti *zamjena programa* (engl. *swap*). Kasnije ponovno pokretanje prvog procesa zahtijevat će ponovnu promjenu sadržaja radnog spremnika.

Taj nas mehanizam podsjeća na promjenu konteksta kod prebacivanja procesora s izvođenja jedne dretve na drugu. Tamo smo morali pohraniti sadržaje registara procesora u opisnik dretve koja se prekida i napuniti registre procesora iz opisnika dretve koja započinje izvođenje. Ovdje moramo zamijeniti sadržaje svih spremničkih lokacija: pohraniti u pomoćni spremnik sadržaje lokacija procesa koji se prekida i napuniti lokacije sadržajima koji pripadaju procesu koji će započeti ili nastaviti s izvođenjem.

Ako zamislimo da su programi označeni s *A*, *B*, *C* i *D* na slici 8.2. zapravo procesni adresni prostori četiriju procesa koji se naizmjence izvode, onda u jednom trenutku samo jedan od njih može biti smješten u radni spremnik, a ostali moraju biti pohranjeni na disku. Svaki program na disku mora biti smješten u odgovarajući broj sektora (postoji mala vjerojatnost da je veličina procesnih adresnih prostora cjelobrojni višekratnik veličine sektora te zadnji sektor neće biti do kraja popunjeno).

U tablicama operacijskog sustava za svaki se proces mora nalaziti popis sektora diska u kojima se nalazi smješten program. Tablicu možemo zamisliti kao poredak u kojem su redom pohranjeni brojevi sektora dopunskog spremnika. Na samom disku sektori jednog programa ne moraju biti smješteni jedan uz drugog. Važno je samo da oni budu čitani s

diska onim redom kako je to zapisano u tablicama procesa i smješteni u radni spremnik pravilnim redoslijedom.

Međutim, način na koji su sektori smješteni na disku može tako utjecati na trajanje prebacivanja programa. Pokazat ćemo to na sljedećem primjeru.

PRIMJER 8.2.



Pretpostavimo da treba obaviti punjenje radnog spremnika programa s procesnim adresnim prostorom veličine 1 MB s diska koji ima svojstva opisana u zadatku 8.1. Želimo procijeniti vrijeme potrebno za punjenje programa ako je:

- a) program smješten na disku kompaktno (tj. u uzastopnim sektorima smještenim u jedan susjedni cilindar ili više njih) i
- b) program podijeljen na blokove veličine 1 KB koji su smješteni u dva uzastopna sektora, s tim da su ti parovi sektora raspršeni po cijelom disku.

Pri ocjeni trajanja prijenoa zanemarit ćemo vrijeme utrošeno u upravljačkom sklopu diskovne jedinke.

Program veličine 1 MB = 1024 KB dijeli se na 1024 bloka veličine 1 KB te za njihov smještaj trebamo 2048 sektora od po 512 B.

Model diska iz zadatka 8.1. ima cilindar sastavljen od 10 staza s po 100 sektora. Za kompaktni smještaj programa trebat ćemo dakle dva puna cilindra i još 48 sektora u trećem cilindru (što je manje od jedne staze).

Prepostaviti ćemo da se glave postavljaju na prvi od cilindara s prosječnim trajanjem traženja. Nakon prosječnog rotacijskog kašnjenja započinje prijenos cijele prve staze i zatim bez dodatnog rotacijskog kašnjenja prijenos sljedećih staza istog cilindra (svaka staza ima svoju glavu za čitanje, tako da se prebacivanje čitanja s jedne na drugu stazu može obaviti trenutačno). Prema tome, prijenos 10 cijelih staza obaviti će se u 10 okretaja diska. Nakon toga slijedi premještanje na susjedni cilindar (uz $D = 1$) i nakon prosječnog rotacijskog kašnjenja opet prijenos 10 cijelih staza. Konačno, nakon premještanja na sljedeći susjedni cilindar i prosječnog rotacijskog kašnjenja trebat će prenijeti još samo 48 sektora jedne staze. Opisani način izračunavanja trajanja prebacivanja daje:

traženje 1. cilindra	\bar{T}_S	16.66 ms
rotacijsko kašnjenje	$0.5T_R$	6.25 ms
prijenos 10 staza (1000 sektora)	$10T_R$	125.00 ms
premještanje na susjedni cilindar	$T_S = 1.5D$	1.50 ms
rotacijsko kašnjenje	$0.5T_R$	6.25 ms
prijenos 10 staza (1000 sektora)	$10T_R$	125.00 ms
premještanje na susjedni cilindar	$T_S = 1.5D$	1.50 ms
rotacijsko kašnjenje	$0.5T_R$	6.25 ms
prijenos preostalih 48 sektora	$\frac{48}{100}T_R$	6.00 ms
Ukupno	T_a	294.41 ms

Kod raspršenog smještaja 1024 blokova u parove sektora prijenos programa obavit će se tako da se 1024 puta obavlja postavljanje glava s prosječnim trajanjem traženja i prosječnim rotacijskim kašnjenjem nakon čega slijedi prijenos dvaju sektora. Za prijenos svakog bloka imat ćemo:

traženje cilindra	\bar{T}_S	16.66 ms
rotacijsko kašnjenje	$0.5T_R$	6.25 ms
prijenos 2 sektora	$\frac{2}{100} T_R$	0.25 ms
Ukupno		23.16 ms

Ukupno vrijeme prijenosa za 1024 bloka, dakle, iznosi

$$T_b = 1024 \cdot 23.16 \text{ ms} = 32715.84 \text{ ms} = 23.72 \text{ s.}$$

Bez obzira na moguće pogreške u ocjeni vremena prijenosa koje smo unijeli pojednostavljenjem modela diska, nedvojbeno se može utvrditi da kompaktni smještaj omogućuje mnogo brži prijenos programa. Vidjet ćemo, međutim, kasnije da takav smještaj nije jednostavno ostvariti.

Prethodni nam primjer ukazuje da čak i kod najbržeg prijenosa trajanje zamjene programa traje jako dugo u odnosu na trajanje promjene konteksta dretvi. Naime, ako se pri promjeni konteksta pohranjuje čak i nekoliko desetaka registara procesora, trajanje te zamjene obavit će se u dijelovima milisekundi, što je zanemarivo u odnosu na više stotina milisekundi potrebnih za zamjenu programa u radnom spremniku. Uočimo da u razdoblju promjene sadržaja spremnika procesor ne može ništa raditi i stoji nezaposlen.

Prema tome, bit će potrebno gospodarenje spremničkim prostorom obavljati tako da se u radnom spremniku istovremeno nalazi više programa. Bilo bi, nadalje, poželjno kada bi u svakom trenutku barem jedna od dretvi iz bilo kojeg procesa čiji je adresni prostor smješten u radni spremnik bila pripravna za izvođenje. Tijekom izvođenja takve dretve jednog procesa, može se obavljati izbacivanje programa nekog drugog procesa iz radnog spremnika i njegovo nadomještanje programom nekog trećeg procesa.

8.2.4. Procesni informacijski blok

Iz ovog najjednostavnijeg opisa zaključujemo da gospodarenje spremničkim prostorom nije jednostavni zadatak operacijskog sustava. On će se još više usložniti zahtjevom da se u računalu mogu izvoditi i programi čiji su procesni adresni prostori veći od fizičkog adresnog prostora računala. U današnjim se računalnim sustavima koriste mehanizmi tzv. virtualnog spremnika (engl. *virtual memory*), s pomoću kojih se na prikladan način kombinirano koriste radni i pomoćni spremnici.

Naglasimo ovdje još jedanput da se program koji je potpuno pripremljen za izvođenje smješta na disk, gdje čeka na početak izvođenja. Svaki će program postati proces kada bude krenuo u izvođenje i izvodić će se u svom procesnom adresnom prostoru.

Nadalje, ustanovimo još jedanput da svaki proces mora imati barem jednu dretvu kako bi se mogao izvoditi. Znamo da u sustavskom adresnom prostoru za svaku dretvu postoji opisnik dretve i da jezgra operacijskog sustava premješta taj opisnik iz jedne liste u drugu u ovisnosti o stanju dretve. Jednako tako, u sustavskom adresnom prostoru moraju se čuvati u posebnim tablicama sve informacije o svakom procesu. Za svaki se proces oblikuje *procesni informacijski blok* ili *procesni kontrolni blok* (engl. *process information block*, *process control block*)³. Unutar tog bloka nalaze se, kao što mu i ime govori, sve informacije važne za odvijanje procesa. Između ostalog, u procesni informacijski blok pohranjavat će se i sve informacije povezane s gospodarenjem spremničkim prostorom. U procesnom informacijskom bloku nalazit će se već spomenute tablice sektora u kojima je program procesa pohranjen na disku, a tu će se naći i podatke o smještanju programa u radnom spremniku. Vidjet ćemo kasnije da procesni informacijski blok mora sadržavati i mnoge druge podatke ili kazaljke na druge informacijske blokove.

Mi ćemo u sljedećem odjelu najprije razmotriti načine gospodarenja spremnicima koji danas imaju uglavnom povjesno značenje, ali ukazuju na način razmišljanja koji je važan i za druge mehanizme unutar računalnih sustava i opravdava razmjerno složen način ostvarenja virtualnog spremnika. Isto tako, razmatranje ukazuje i na razvitak sklopovske podloge kojom današnji procesori podupiru ostvarenje virtualnih spremnika. Pokazat će se da nam ta (uglavnom povjesna) promišljanja povezana s gospodarenjem radnog spremnika pomažu i pri osmišljavanju raspodjele spremničkog prostora diskova.

8.3. Pregled razvita načina dodjeljivanja radnog spremnika

8.3.1. Statičko raspoređivanje radnog spremnika

U najjednostavnijoj Von Neumannovoj arhitekturi računala unutar procesora neposredno se generiraju adrese koje se preko adresnih vodiča dovode do spremnika. Ponovno pogledajmo sliku 2.5. u kojoj je označeno da sve adrese koje izlaze iz procesora prolaze kroz adresni međuregistar. Već smo utvrdili da te adrese:

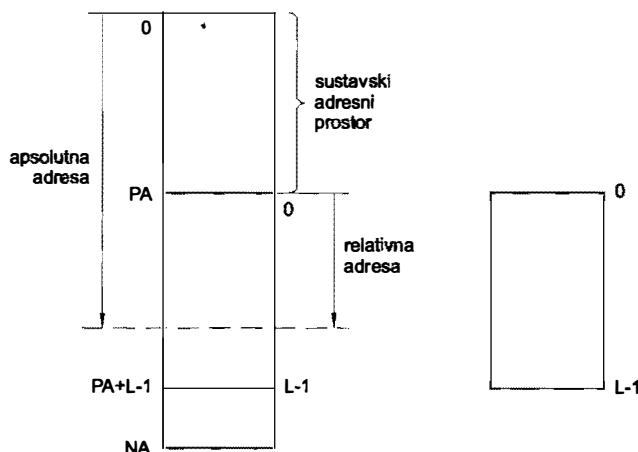
- dolaze iz programske brojila kada se iz radnog spremnika dohvataju instrukcije;
- dolaze iz registra kazaljke stoga kada se adresira stog;
- oblikuju se iz adresnog dijela instrukcija kada se adresiraju podaci.

Kompilator prilikom prevođenja programa mora generirati niz strojnih instrukcija i poslagati ih jednu iza druge. U okviru nekih instrukcija morat će biti zapisane stvarne fizičke adrese. Ako zamislimo da takav strojni program moramo smjestiti u uzastopne bajtove spremnika, onda će u nekim bajtovima (odnosno nizu od četiri bajta ako je riječ o tridesetdvobitnoj adresi) morati pisati brojevi koji su stvarne adrese u spremniku. Za takav program kažemo da je napisan u apsolutnom obliku. *Apsolutne adrese*, koje treba zapisati u takvom programu, ovise o tome gdje će program biti smješten u radnom spremniku kada bude pripremljen za izvođenje. Ako se predviđa da će prva adresa programa biti

³ Operacijski sustav Windows NT objektno je orijentiran te je procesni informacijski blok sadržan unutar procesnog objekta.

smještena na početnoj adresi PA , onda kompilator mora tu adresu uzeti u obzir pri određivanju apsolutnih adresa koje će se zapisati unutar strojnog programa. To znači da bi se početna adresa PA morala staviti na raspolaganje kompilatoru prije početka stvaranja strojnog programa.

Međutim, prikladniji je drugi način pripreme programa, kod kojeg se prevođenje obavlja tako da se pretpostavi početna adresa programa jednaka nuli i generira strojni oblik u kojem su sve adrese izračunate s obzirom na tu početnu nullu adresu. Te adrese zovemo *relativnim adresama*. Uz takav strojni oblik programa s relativnim adresama kompilator mora generirati posebne oznake uz one lokacije u kojima treba promijeniti adresu ovisno o stvarnoj početnoj adresi adresnog prostora u koji će program biti smješten. Veza između relativnih i apsolutnih adresa ilustrirana je slikom 8.6.



Slika 8.6. Prevođenje relativnih adresa u apsolutne adrese pri statičkom dodjeljivanju spremnika

Pretpostavimo, zbog jednostavnosti, da je program smješten u poredak od L riječi duljine od po 32 bita (četiri bajta), koji možemo deklarirati simbolom $R[L]$. Istodobno kompilator može generirati poredak bitova deklariran s $C[L]$ u kojem $C[I] == 0$ označava da riječ $R[I]$ ne treba mijenjati, a $C[I] == 1$ označava da sadržaju riječi $R[I]$ treba pribrojiti početnu adresu PA kako bi se dobio program s apsolutnim adresama. Ako za smještanje apsolutnog oblika programa deklariramo poredak $A[L]$, onda se prevođenje programa iz oblika s relativnim adresiranjem u taj apsolutni oblik dobiva sljedećim programskim odsječkom:

```

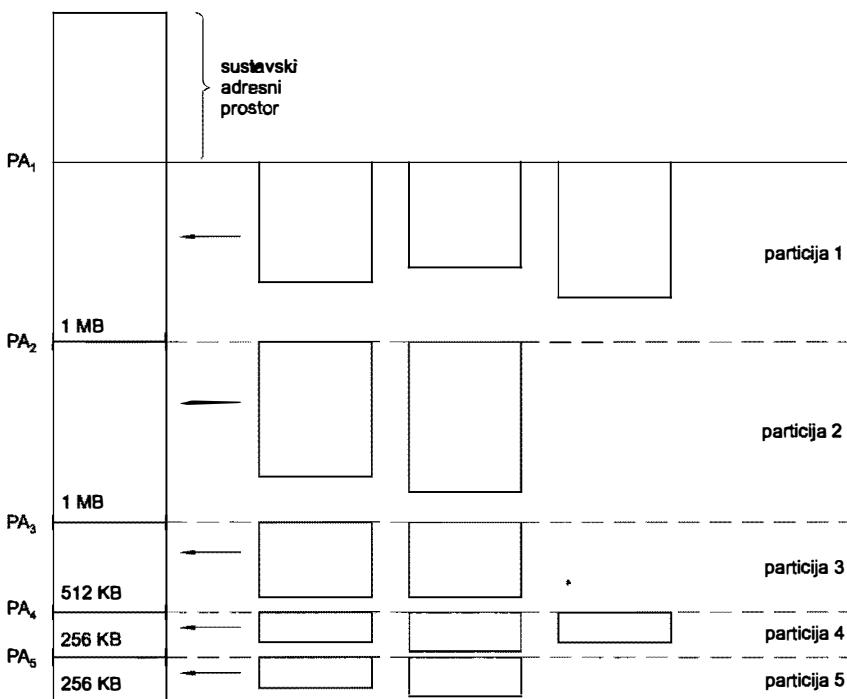
za (I = 0, J < L, I++) {
    ako je (C[I] == 1) {
        A[I] = R[I] + PA;
    }
    inače {
        A[I] = R[I];
    }
}

```



Ako se taj isti program želi smjestiti u radni spremnik na neku drugu početnu adresu, morat će se ponovno pripremiti novi apsolutni oblik toga programa.

Već smo ustanovili da je u radnom spremniku poželjno držati istodobno više programa. Oni moraju biti smješteni u različite dijelove spremnika s različitim početnim adresama. U povijesnom razvitku gospodarenja spremničkim prostorom jedan od zapaženih načina dodjele spremnika zasnovao se na podjeli spremnika na dijelove stalne veličine, tzv. *stalne particije* (engl. *fixed partitions*). Particije su mogle biti jednake veličine ili različitih veličina. Particije različitih veličina omogućivale su bolje iskorištenje radnog spremnika jer su se apsolutni oblici manjih programa mogli prirediti za manje particije a veći programi smjestiti u veće particije. Programi pripremljeni za jednu particiju nisu se jednostavno mogli preseliti u drugu particiju jer to seljenje zahtijeva novo generiranje apsolutnog oblika programa.



Slika 8.7. Podjela spremnika na particije

Na slici 8.7. prikazana je podjela korisničkog adresnog prostora na particije. Uz pojedinu particiju simbolički su prikazani programi koji su pripremljeni za izvođenje u toj particiji. Oni se nalaze na disku i mogu se po ukazanoj potrebi napuniti u radni spremnik. Ako ih se privremeno izbaciti iz radnog spremnika, oni će se moći kasnije vratiti samo u istu particiju. Otuda dolazi i naziv *statičko raspoređivanje* – programi se tijekom svog boravka u sustavu nalaze uvijek u istom dijelu radnog spremnika.

Uočimo da tijekom rada spremnik neće biti potpuno iskorišten, i to zbog dva razloga:

- Prvo, programi neće biti potpuno jednake veličine kao particije te će dijelovi particija ostati neiskorišteni. Taj način neiskorištenja dobio je naziv *unutarnja fragmentacija* (engl. *internal fragmentation*).
- Drugo, tijekom rada može se dogoditi da svi procesi čiji su programi smješteni u istu particiju bivaju blokirani pa ta particija radnog spremnika ostaje prazna. Pritom može postojati više procesa čiji programi čekaju na dodjelu radnog spremnika, ali oni ne mogu biti napunjeni u radni spremnik jer nisu pripremljeni za tu particiju. Taj način neiskorištenja spremnika dobio je naziv *vanska fragmentacija* (engl. *external fragmentation*).

Naglasimo još jedanput i dodatno ograničenje da adresni prostori programa nisu mogli biti veći od najveće particije fizičkog spremnika.

8.3.2. Dinamičko raspoređivanje radnog spremnika

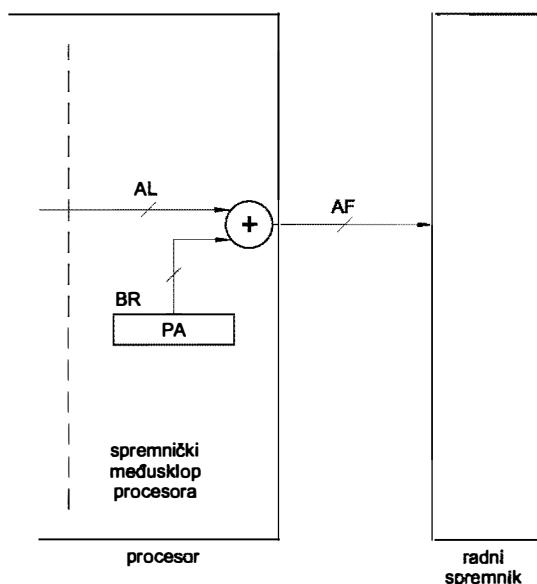
Vidjeli smo da se generiranje apsolutnog oblika programa svodi na pribrajanje početne adrese one particije u koju će program biti smješten sadržajima nekih označenih riječi programa.

S obzirom na to da je jedina operacija koju treba provoditi pri preračunavanju adresa pribrajanje početne adrese *PA*, nameće se pomisao da bi se to moglo obaviti sklopovski, i to uvijek kada se adresa iz procesora šalje prema spremniku.

Ta se jednostavna zamisao može ostvariti uz sljedeće pretpostavke:

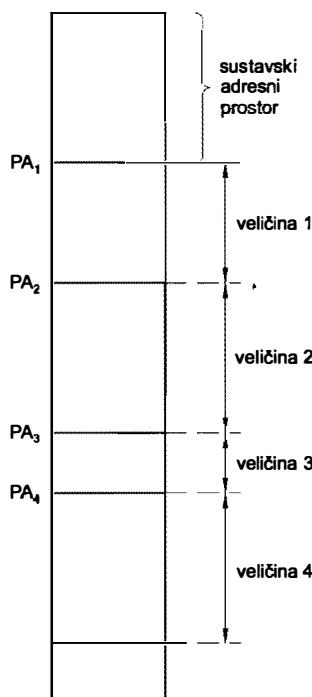
- Program će zauvijek ostati u obliku s relativnim adresama, tj. sve adrese unutar programa izračunate su kao da je početna adresa jednaka nuli. Taj adresni prostor nazvat ćemo *logičkim adresnim prostorom*.
- Program će se smjestiti u radni spremnik u prostor koji započinje s početnom adresom *PA*. Taj ćemo prostor nazivati *fizičkim adresnim prostorom*.
- U procesoru, na putu prema sabirnici, iza adresnog međuregistra (vidi slike 2.5. i 2.6.) treba dodati jedno sklopovsko zbrajalo i uz njega još jedan registar u koji će se zapisati početna adresa fizičkog adresnog prostora. Taj se registar naziva *baznim registrom* (engl. *base register*).

Ovakvo proširenje osnovnog modela procesora prikazno je slikom 8.8. Unutar procesora (u njegovim registrima) generira se logička adresa *AL*. Toj se adresi pribraja početna adresa *PA* koja prije početka izvođenja dretve mora biti pohranjena u bazni registar označen na slici s *BR*. Iz procesora prema radnom spremniku "izlazi" sada fizička adresa *AF* koja se dovodi do spremnika. Uočimo da se početna adresa pribraja svakoj adresi koja se u procesoru generira pa tako i adresama koje dolaze iz programskog brojila i registra kazaljke stoga. Time otpada briga o posebnom gledanju na adrese koje dolaze iz dva registra. Ovo sklopovsko proširenje procesora začetak je tzv. *spremničkog međusklopa* (engl. *memory management unit – MMU*) koji postaje sastavni dio svakog procesora.



Slika 8.8. Sklopovska pretvorba logičke adrese u fizičku adresu

Uz ovaku sklopovsku potporu svi se programi pripadaju tako da svaki program ima svoj logički adresni prostor koji smo za progamne koji se izvode nazvali procesnim adresim



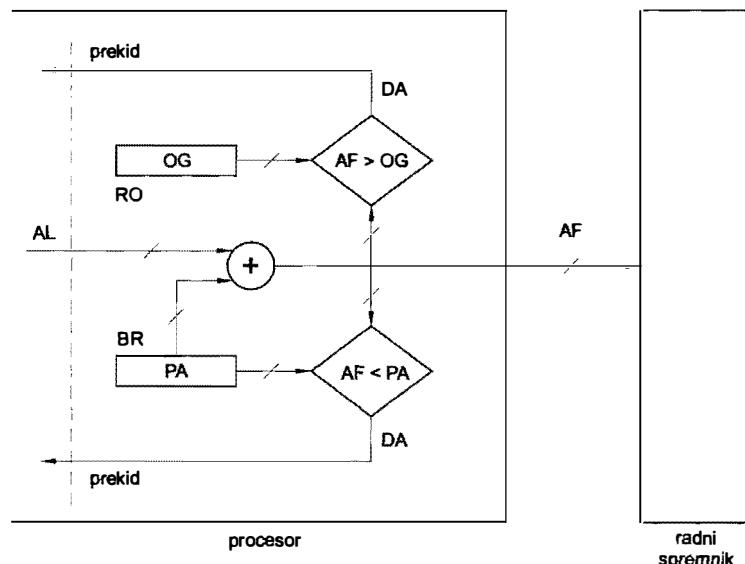
Slika 8.9. Smještanje programa uz dinamično raspoređivanje radnog spremnika

prostorom. Logički adresni prostor ne mijenja se tijekom izvođenja programa, bez obzira na mjesto kuda će program pri višekratnoj zamjeni biti premještan. Važno je samo da se prije početka njegova izvođenja uvijek u bazni registar pohrani njegova aktualna početna adresa u fizičkom adresnom prostoru. Za razliku od statičkog raspoređivanja spremnika ovdje govorimo o *dinamičkom raspoređivanju spremnika*.

U ovakvo sklopovsko rješenje nisu nam potrebne stalne particije i za njih pripremljeni programi. Programi se mogu "naslagati" u spremnik neposredno jedan iza drugoga, kao što prikazuje slika 8.9. U procesnom informacijskom bloku svakog procesa mora biti zapisana početna adresa PA_1 procesnog adresnog prostora.

U ovakvom radnom okruženju možemo jednostavno ostvariti višeprogramske rad. U baznom registru morat će se nalaziti početna adresa onog procesa čija se dretva upravo izvodi. To znači da se sadržaj bavnog registra mora mijenjati kada se izvođenje prebacuje s dretve jednog procesa na dretvu drugog procesa. Promjena sadržaja bavnog registra postaje tako sastavni dio promjene konteksta koju treba obaviti kada se izvođenje prebacuje s neke dretve unutar jednog procesa na dretvu drugog procesa. Ako se obavlja zamjena konteksta za dretve unutar istog procesa, onda se sadržaj bavnog registra ne mijenja.

Pri ovakvom načinu uporabe radnog spremnika postoji velika opasnost da procesi jedan drugomu na nedopušteni način zasmetaju. Naime, nikako se ne bi smjelo pretpostaviti da su svi programi uvijek potpuno ispravni i da će disciplinirano adresirati samo unaprijed za njih predviđeni adresni prostor. U tom se radnom okruženju, uostalom, moraju izvoditi i programi koji se još razvijaju i ispituju i koji će s vrlo velikom vjerojatnošću pisati u prostor predviđen za druge proceze.



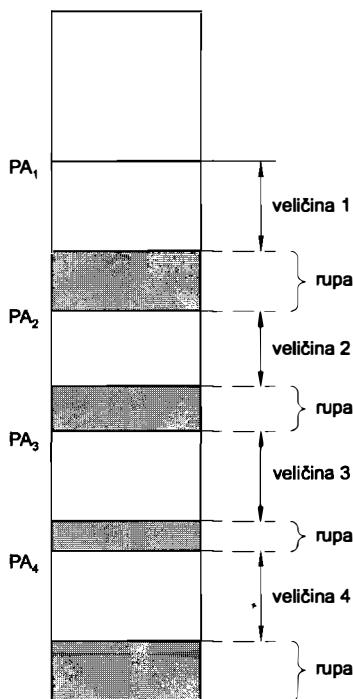
Slika 8.10. Spremnički međusklop s baznim registrom i registrom ograda

Pokazalo se da takve pogreške nije moguće izbjegći nikakvim programskim zaštitama i

da treba potražiti neko sklopovsko razrješenje toga problema. Osnovna zamisao takve sklopovske potpore prikazana je slikom 8.10. Uz bazni registar u spremnički međusklop dodan je još jedan registar koji možemo nazvati registrom ograda RO . U taj se registar pohranjuje najveća adresa koju program smije dohvaćati. Tu adresu možemo izračunati tako da početnoj adresi PA_I pribrojimo veličinu programa te je:

$$OG_I = PA_I + VELIČINA_I.$$

U spremničkom se međusklopu sklopovskim uspoređivačima ustanavljuje je li generirana fizička adresa AF veća od početne adrese i manja od adrese ograde. Ako ona prelazi te granice, spremnički međusklop izaziva sklopovski prekid jer ta pojava ukazuje da program nije ispravan. Instrukcija koja je izazvala prekid neće se izvesti do kraja i pozvat će se funkcija operacijskog sustava koja će zaustaviti izvođenje tog neispravnog procesa.



Slika 8.11. Fragmentacija pri dinamičnom raspoređivanju spremnika

S ovakvim spremničkim međusklopolom moguće je svladati neke nedostatke statičkog raspoređivanja radnog spremnika. Međutim, i dalje postoji problem fragmentacije spremničkog prostora. Naime, tijekom vremena neki procesi završavaju izvođenje i oslobađaju svoj spremnički prostor. Drugi procesi bivaju blokirani i izbacuju se iz radnog spremnika pa tako također oslobađaju svoj procesni prostor kako bi načinili mesta za programe procesa koje se mogu izvoditi. Ti će se procesi kasnije vratiti u radni spremnik, ali najvjerojatnije na neko drugo mjesto. S obzirom na to da veličine programa nisu jednake, u spremniku će se nakon nekog vremena između procesnih adresnih prostora pojaviti prazni dijelovi

spremnika koje nazivamo *rupama* (engl. *holes*). U nekome bi trenutku popunjenoš spremnika mogla izgledati kao na slici 8.11. Takav se izgled radnog spremnika u sustavu koji dulje radi jednostavno ne može izbjegći.

Spremnički međusklopovi oblika prema slici 8.10. poslužili su kao osnova gospodarenja spremničkim prostorom mnogih operacijskih sustava. Spremnički međusklopovi proširivani su tako da se procesni adresni prostor može podijeliti na posebne dijelove, tzv. segmente:

- segment u koji se pohranjuju instrukcije programa ili programski kôd te se taj segment naziva *kodnim segmentom* i adresira se iz programskog brojila;
- *stogovni segment* u koji se pohranjuje stog i adresira iz registra kazaljke stoga;
- *podatkovni segment* u koji se pohranjuju podaci i čije adrese potječu iz adresnih dijelova instrukcija.

Za svaki od segmenata u spremničkom međusklopu predviđeni su posebni bazni registri i registri ograde, kako bi se cijeli radni spremnik mogao podijeliti na posebne segmente: kodni, stogovni i podatkovni. Međutim, ni gospodarenje zasnovano na takvim sklopovima bitno ne olakšava problem fragmentacije.

Osnovni način suzbijanja fragmentacija svodi se na to da se rupe održavaju što većima kako bi se u njih mogli smjestiti novi programi. To se može postići tako da se:

- pri svakom oslobođanju nekog procesnog prostora novonastala rupa spaja s eventualnim susjednim rupama u novu veću rupu;
- pri svakom novom zahtjevu za spremničkim prostorom potraži najmanja rupa u koju se može smjestiti novi program.

Preostaje, nadalje, mogućnost da se u nekom trenutku, kada se ustanovi da je fragmentacija postala prevelika, privremeno obustavi izvođenje dretvi i “presloži” programe u kompaktni prostor. Time se na kraju spremnika možda dobiva dovoljno velika rupa u koju se može smjestiti program koji čeka na dodjelu radnog spremnika.⁴

Mi se time ovdje nećemo dalje baviti. Pokazat ćemo samo još sljedećim zanimljivim primjerom (koji se pri čitanju može i preskočiti) da se i pri takvom dodjeljivanju spremnika postojanje rupa ne može izbjegći.

PRIMJER 8.3.



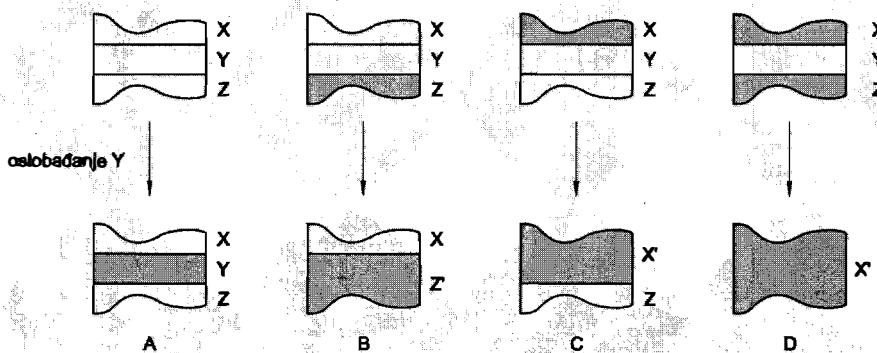
U ovom primjeru koji potječe od D. E. Knutha pretpostavlja se da promatraš sustav kod kojeg je vjerojatnost oslabuđanja spremničkog prostora jednaka vjerojatnosti zahtjeva. Ako takav sustav radi dovoljno dugo, uspostavit će se u njemu stohastičko ravnotežno stanje. To znači da će prosječni broj punih blokova i prosječni broj rupa biti stalni. Postavlja se pitanje koliko je u spremniku rupa ako se u njemu nalazi određeni broj punih blokova.

⁴ Ta se tehnika “sakupljanja otpadaka” (engl. *garbage collection*) upotrebljava i u nekim programskim okruženjima za izvođenje programa koji intenzivno koriste dinamičke strukture podataka.

Pretpostaviti ćemo da je:

- spremnik dovoljno velik tako da možemo zanemariti rubne uvjete i pretpostaviti da iznad i ispod svakog punog bloka postoji još jedan dio spremnika (prazan ili pun);
- da se nova rupa nastala oslobođanjem prostora spaja sa susjednim rupama (ako postoje);
- da je vjerojatnost pronalaženja rupe koja je jednaka veličini zahtijevanog prostora, nazovimo je q , zanemariva, a suprotna vjerojatnost $p = 1 - q$ približno jednaka jedinici.

Promotrimo na slici 8.12. što se događa s brojem rupa kada se oslobađa puni blok Y .



Slika 8.12. Stanja rupa prije i poslije oslobođanja prostora Y

Ovisno o tome što se nalazi u njegovu susjedstvu razlikujemo četiri tipa punih blokova:

- tip A ima dva puna susjedna bloka i oslobođanjem bloka Y nastat će jedna rupa;
- tip B ima za donjeg susjeda rupu Z i oslobođeni prostor spojiti će se s tom rupom u veću rupu Z' te broj rupa ostaje isti;
- tip C ima za gornjeg susjeda rupu X i oslobođeni prostor spojiti će se s tom rupom u veću rupu X' te broj rupa ostaje isti;
- tip D ima dvije susjedne rupe X i Y i oslobođeni prostor spojiti će se s njim u jednu rupu, nazovimo je X' , te će se broj rupa smanjiti za jedan.

Ako pretpostavimo da u sustavu nalazimo a punih blokova tipa A , b punih blokova tipa B , c punih blokova tipa C i d punih blokova tipa D , onda je broj punih blokova jednak:

$$m = a + b + c + d.$$

Broj rupa u sustavu jednak je:

$$n = \frac{b + c + 2 \cdot d}{2}$$

Najime, uz svaki od b punih blokova tipa B postoji jedna donja rupa, uz svaki od c punih blokova tipa C postoji jedna gornja rupa, a uz svaki od d blokova tipa D postoje dvije rupe: gornja i donja. Uz pune blokove tipa A nema susjednih rupa. Međutim, svaka rupa koja je jednom punom bloku gornja drugom je bloku donja i u opisanom promišljanju broji se dva puta pa se zbroj u gornjem izrazu dijeli s dva. Nadalje, zaključujemo da u sustavu mora biti $c = b$ jer svaka rupa koja je jednom punom bloku gornja mora biti jednom bloku donja, tako da dobivamo:

$$n = b + d.$$

S obzirom na to da promatramo stacionarno stanje zaključujemo da je vjerojatnost porasta broja rupa jednaka vjerojatnosti smanjivanja njihova broja. To ćemo izraziti tako da izjednačimo vjerojatnost nastanka jedne rupe s vjerojatnošću nestanka jedne rupe.

Broj rupa može porasti za jedan samo onda ako se oslobodi jedan puni blok, i to blok tipa A . Vjerojatnost da se to dogodi dobiva se tako da se vjerojatnost oslobađanja pomnoži s omjerom a/m te se dobiva:

$$(\text{vjerojatnost oslobađanja}) \cdot \frac{a}{m}.$$

Broj rupa može se smanjiti za jedan onda ako se oslobodi jedan puni blok, i to blok tipa D ili postoji jedan zahtjev koji je po veličini potpuno jednak veličini rupe u koji se novi program može smjestiti. Vjerojatnost da je veličina zahtjeva jednaka veličini neke postojeće rupe označili smo s q . Vjerojatnost da se broj rupa smanji za jedan jednak je, dakle:

$$(\text{vjerojatnost oslobađanja}) \cdot \frac{d}{m} + (\text{vjerojatnost zahtjeva}) \cdot q.$$

Vjerojatnosti povećavanja i smanjivanja broja rupa za jedan moraju biti jednake pa gornja dva izraza treba izj dnačiti. Prepostavili smo da su vjerojatnosti oslobađanja i zahtjeva međusobno jednakе te se njihovim dokidanjem dobiva:

$$\frac{a}{m} = \frac{d}{m} + q$$

odnosno, s obzirom na to da je $q = 1 - p$:

$$a = d + (1 - p)m.$$

Taj se izraz za a može uvrstiti u $m = a + b + c + d$ te slijedi:

$$m = d + (1 - p)m + b + c + d.$$

Iz ovog se izraza, uz uvažavanje da je $c = b$ i $n = b + d$, nakon sredivanja dobiva:

$$n = \frac{1}{2}pm,$$



odnosno, s obzirom na to da p teži prema 1:

$$n = \frac{1}{2}m.$$

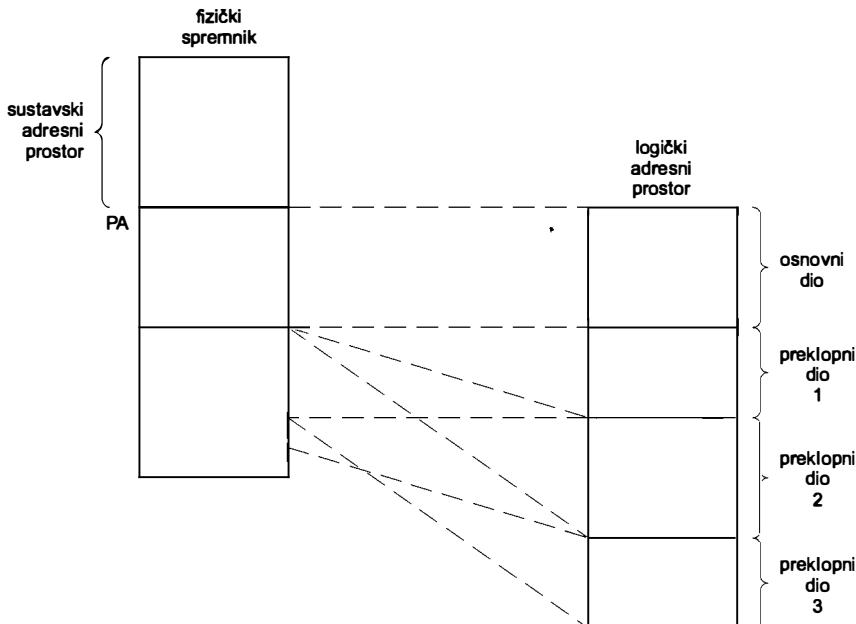
Prema tome, u stacionarnom će se stanju u spremniku naći broj rupa koji je jednak polovini broja punih blokova. Ta se činjenica spominje kao "Knuthovo pedesetpostotno pravilo".

Fragmentacija se, dakle, u ovakovom načinu raspoređivanja spremničkog prostora nikako ne može izbjegći.

Uz opisanu fragmentaciju ozbiljan je nedostatak svih do sada opisanih načina dodjeljivanja radnog spremnika činjenica da programi nikako ne mogu biti veći od fizički raspoloživog spremničkog prostora. To je ograničenje uvjetovano zahtjevom da se prije početka izvođenja cijeloviti program, tj. cijeli adresni prostor procesa mora nalaziti u radnom spremniku.

8.3.3. Preklopni način uporabe radnog spremnika

Osnovna zamisao koja omogućuje izvođenje programa čiji je logički adresni prostor veći od fizičkog spremnika zasniva se na podjeli programa na dijelove koji svi ne moraju biti istovremeno smješteni u fizičkom spremniku. U radnom spremniku mora se u svakom trenutku nalaziti onaj dio programa koji sadrži instrukcije koje se upravo izvode, kao i dio podatkovnog prostora koji te instrukcije adresiraju.



Slika 8.13. Preklopni način uporabe radnog spremnika

Program pritom treba podijeliti na jedan osnovni dio koji se uvijek nalazi u radnom spremniku i na dijelove koji se naizmjence smještaju u preostali dio korisničkog dijela fizičkog spremnika. Zamjenljivi dijelovi programa se u preklopu (engl. *overlay*) smještaju u radni spremnik, kao što prikazuje slika 8.13., pa taj način nazivamo preklopnim načinom uporabe spremnika.

Ovaj način uporabe spremnika zahtjeva, međutim, vrlo pomnjuvu pripremu programa. Programer mora podijeliti program na odgovarajuće dijelove i u osnovni dio programa ugraditi mehanizme za donošenje odluke o pravodobnom prebacivanju preklopnih dijelova u radni spremnik. Unutar programa mora se, dakle, voditi računa o načinu korištenja pojedinih dijelova programa. Takav način programiranja zahtjeva detaljnu analizu svakog programa jer je način njegove podjele na preklopne dijelove ovisan o njegovim željenim funkcijskim svojstvima. Posao podjele na preklopne dijelove bitno se usložnjava u više programskom radu jer se u radnom spremniku mogu naći pojedini preklopni dijelovi različitih programa.

Međutim, zamisao podjele programa na dijelove koji se svi ne moraju nalaziti istodobno u radnom spremniku pokazala se vrlo korisnom. Pritom je postalo razumljivo da postupak podjele programa na dijelove i način njihova smještanja u radni spremnik treba automatizirati, kako bi se olakšala priprema programa i izbjeglo pogreške programera.

U današnjim računalnim sustavima upotrebljavaju se procesori s prikladnim spremničkim međusklopovima koji omogućuju da operacijski sustavi ostvare vrlo djelotvorno gospodarenje spremničkim prostorom.

8.4. Dodjeljivanje spremnika straničenjem

8.4.1. Sklopovska podloga straničenju

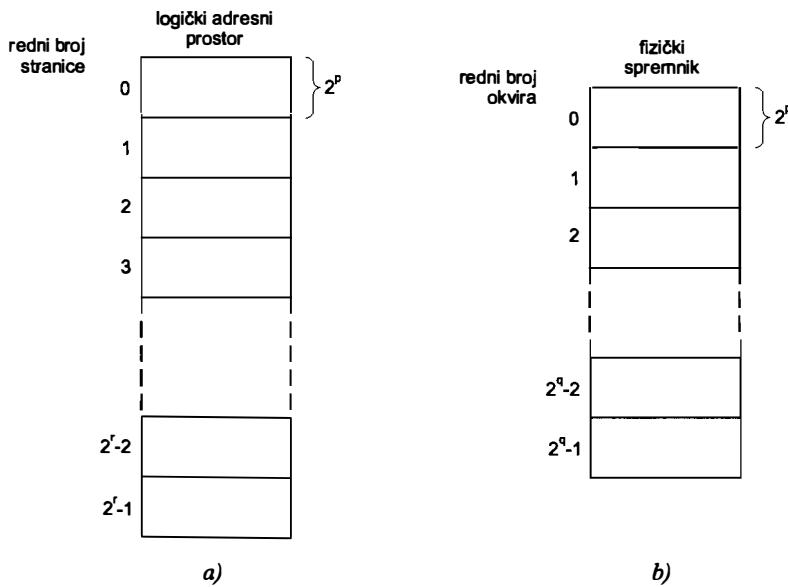
Stranice logičkog adresnog prostora i okviri fizičkog adresnog prostora

Prepostavimo da je cijeloviti korisnički program pripremljen za izvođenje na način koji smo opisali u opisu dinamičkog dodjeljivanja spremnika u prethodnoj točki. Logički adresni prostor programa zovemo i logičkim adresnim prostorom procesa. Adrese unutar tog adresnog prostora jesu logičke adrese koje započinju s nultom adresom. Procesor će u svojim registrima tijekom izvođenja dretvi procesa generirati te logičke adrese. Na putu iz registara procesora do fizičkog radnog spremnika iz tih logičkih adresa spremnički međusklop mora odrediti fizičku adresu. Vidjeli smo da se to, pri dinamičkom dodjeljivanju spremnika, obavlja pribrajanjem početne adrese smještene u bazni registar. Nadalje, ustavili smo da se u fizičkom spremniku mogu odvijati programi čiji je procesni adresni prostor veći od prostora fizičkog spremnika ako se procesni adresni prostor podijeli na dijelove koji ne moraju svi biti istovremeno smješteni u fizičkom spremniku. U današnjim računalnim sustavima zasnovanim oko suvremenih procesora ustalio se način takvog gospodarenja spremničkim prostorom koje nazivamo straničenjem (engl. *paging*).

Logički adresni prostor zamišljeno se dijeli na jednakove velike dijelove koje nazivamo stranicama (engl. *page*). Zbog pojednostavljenja te podjele prikladno je da stranice imaju veličinu koja je cijelobrojna potencija broja dva. U tom se slučaju m adresnih bitova može podijeliti na dvije skupine:

- p bitova koji će odrediti adresu unutar pojedine stranice i
- $r = m - p$ bitova koji određuju redni broj stranice u logičkom adresnom prostoru.

Veličina stranice je, prema tome, jednaka 2^p , a adrese unutar stranice kreću se u granicama od 0 do 2^{p-1} . U logičkom adresnom prostoru moguće je adresirati ukupno 2^r stranica čiji se redni brojevi kreću u granicama od 0 do 2^{r-1} . Tako podijeljeni logički adresni prostor ilustriran je slikom 8.14.a).



Slika 8.14. a) Logički adresni prostor podijeljen na stranice
b) Fizički spremnik podijeljen na okvire

Stvarni fizički ostvareni radni spremnik možemo zamišljeno podijeliti na dijelove koji su jednakih veličini stranice logičkog adresnog prostora. Te zamišljene dijelove nazivamo *okvirima* (engl. *frames*). U jedan okvir fizičkog radnog spremnika može se smjestiti jedna stranica logičkog adresnog prostora. Veličina ostvarenog fizičkog spremnika računala određuje broj postojećih okvira. Pretpostavimo li da je broj postojećih okvira cijelobrojna potencija broja dva, okvire ćemo moći adresirati s q bitova, i to tako da se redni brojevi kreću u granicama od 0 do 2^{q-1} . Na slici 8.14. b) prikazana je podjela fizičkog adresnog prostora podijeljenog na okvire.

PRIMJER 8.4.

Pogledajmo moguće podjеле logičkog adresnog prostora u računalnim sustavima s 32-bitovnom i 64-bitovnom arhitekturom.

U 32-bitovnoj arhitekturi je $m = 32$ i logički adresni prostor je velik:

$$2^{32} = 2^2 \cdot 2^{30} = 4 \text{ GB.}$$

Pretpostavimo li da je $p = 12$, veličina stranice bit će jednaka:

$$2^{12} = 2^2 \cdot 2^{10} = 4 \text{ KB.}$$

Preostalih $r = m - p = 20$ bitova adrese služi za adresiranje stranice te je, prema tome, u 32-bitovnoj arhitekturi moguće adresirati $2^{20} = 1 \text{ M}$ stranica.

U 64-bitovnoj arhitekturi logički adresni prostor velik je

$$2^{64} = 2^4 \cdot 2^{60} = 16 \cdot (2^{10})^6 \approx 16 \cdot (10^3)^6 = 16 \cdot 10^{18} = 16 \text{ EB.}$$

Uz jednaku veličinu stranice od 4 KB broj bitova za adresiranje stranica je $r = m - p = 52$, što daje broj mogućih stranica jednak:

$$2^{52} = 2^2 \cdot 2^{50} = 4 \cdot (2^{10})^5 \approx 4 \cdot (10^3)^5 = 4 \cdot 10^{15} = 4 \text{ PB stranica.}$$

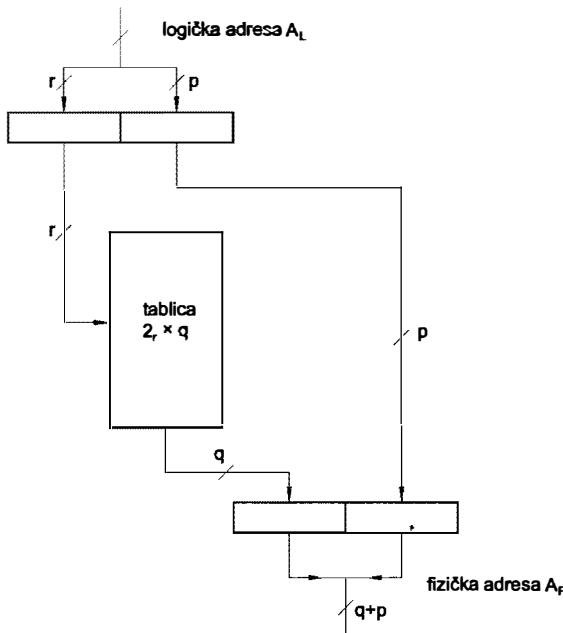
Uz veličinu stranica od 8 KB u 32-bitovnoj arhitekturi logički adresni prostor može imati najviše 500 K stranica, a u 64-bitovnoj arhitekturi logički adresni prostor od 16 EB (*eksabajta*) podijeljen je na najviše 2 P stranica (*petna stranica*).

PRIMJER 8.5.

Fizički radni spremnik ne mora biti ni izdaleka tako velik pa prema tome sadrži mnogo manje okvira od mogućeg broja stranica logičkog adresnog prostora. Tako radni spremnik od 128 MB ima ukupno $2^7 \cdot 2^{20} \text{ B} = 2^{27} \text{ B}$. Kada taj prostor podijelimo na okvire od 2^{12} B , dobivamo ukupno $2^{15} = 32 \text{ K}$ okvira. Fizički spremnik s 256 MB imat će, prema tome, 64 K okvira.

Kako bi se pri dodjeli okvira fizičkog spremnika izbjegao problem fragmentacije, osnovna postavka straničenja svodi se na to da stranice logičkog adresnog prostora mogu biti smještene u okvire fizičkog spremnika proizvoljnim redoslijedom. U posebnoj se tablici pritom mora voditi evidencija o tome u kojem je okviru smještena pojedina stranica. Ta se tablica koristi pri svakom pristupu fizičkom spremniku. Prema tome, ona bi morala biti sastavni dio spremničkog međusklopa jer se prevodenje logičke adrese u fizičku adresu mora obaviti što je moguće brže. Međutim, tablice zbog veličine nije moguće u potpunosti pohraniti unutar spremničkog međusklopa i obično je samo jedan dio tablice pohranjen u priručnim registrima međusklopa.

Prevodenje logičke adrese u fizičku adresu vrlo je jednostavno jer su veličine stranica (odnosno okvira) cijelobrojne potencije broja dva. Naime, redni brojevi stranica izraženi s pomoću r bitova nadopunjeni s desne strane s p bitova s vrijednostima 0 čine zapravo logičku adresu prvog bajta u stranici, a sadržaji donjih p bitova daju relativnu adresu unutar stranice. Ti donji bitovi neposredno se mogu dovesti do fizičkog spremnika i tamo će oni odrediti relativnu adresu unutar okvira. Početna adresa okvira dobiva se tako da se rednom broju okvira izraženog s q bitova pridoda s desne strane p bitova s vrijednošću 0. Prema tome, prevodenje logičke adrese u fizičku adresu obavlja se tako da se donjih p bitova logičke adrese propusti neposredno prema fizičkom spremniku, a gornjih r bitova logičke adrese (tj. redni broj stranice) zamjeni s q bitova fizičke adrese (tj. s rednim brojem okvira). Slika 8.15. ilustrira najprikladniji način prevodenja adrese. S gornjih r bitova logičke adrese adresira se tablica prevodenja koja može imati 2^r redaka, s tim da svaki redak sadrži q bitova koji određuju redni broj (odnosno početnu adresu) okvira u kojoj je smještena stranica.

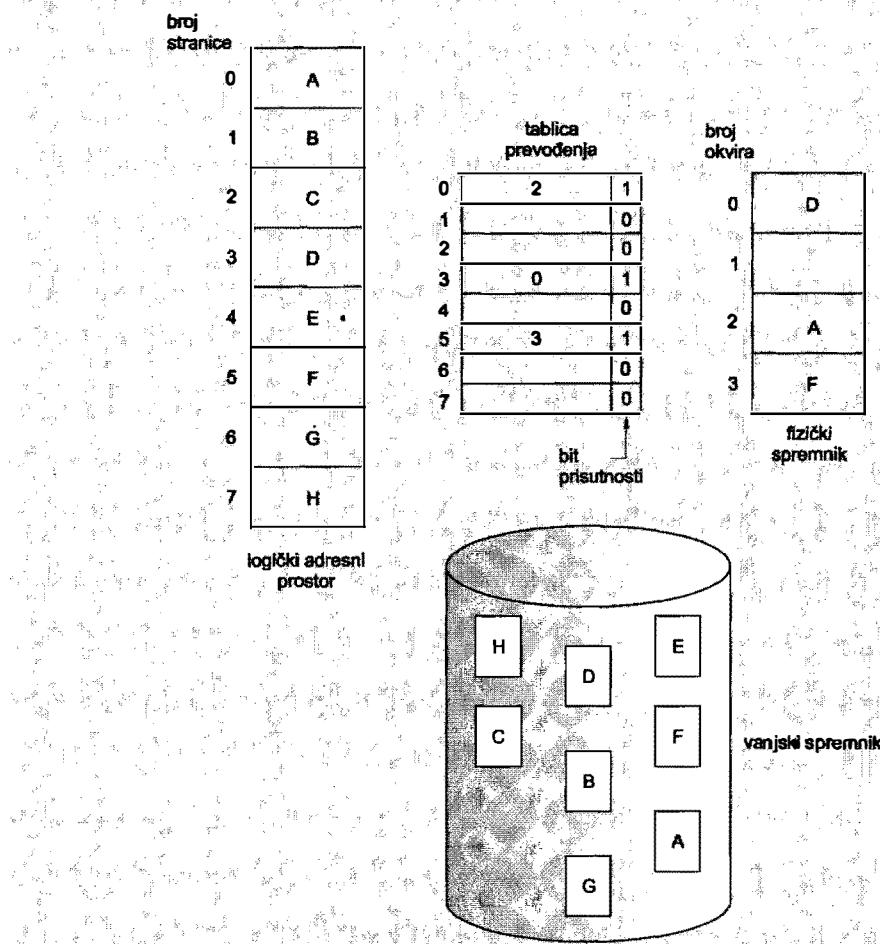


Slika 8.15. Tablično prevodenje fizičke u logičku adresu

U opisanom načinu prevodenja adresa šutke smo prepostavili da se sve stranice programa nalaze u okvirima fizičkog spremnika. U stvarnosti će samo neke od stranica biti smještene u fizički spremnik, a sve ostale bit će pohranjene samo u vanjskom spremniku. U tablici prevodenja stanje se stranice može označiti posebnim *bitom prisutnosti* koji se dodaje svakom retku. Vrijednost 1 neka označava prisutnost stranice u okviru fizičkog spremnika, a vrijednost 0 neka označava da stranica nije u fizičkom spremniku.

PRIMJER 8.6.

Na slici 8.16. prikazan je primjer programa s osam stranica koje su smještene na disku.



Sliku 8.16. Tablica prevođenja nadopunjena bitom prisutnosti

Redni brojevi stranica jesu vrijednosti indeksa tablice prevodenja adresa. U fizički spremnik sa samo četiri okvira smještene su tri stranice te se u pripadnim redcima tablice nalaze redni brojevi okvira i bitovi prisutnosti imaju vrijednost 1. U ostalim redcima tablice bit prisutnosti ima vrijednost 0, a sadržaj na mjestu predviđenom za pohranjivanje rednog broja okvira nema nikakvo značenje.

U fizičkom se spremniku mogu istovremeno nalaziti samo četiri stranice programa. Prema tome, tijekom izvođenja programa morat će se stranice zamjenjivati u raspoloživim okvirima. S time u vezi postavljaju se dva pitanja. Prvo je pitanje kako se ustanavljuje da stranica nije u radnom spremniku i, drugo, kako se odreduje stranica koju treba izbaciti iz nekog okvira kako bi se u njega smjestila nova stranica.

Određivanje stranice koja nije u radnom spremniku riješeno je sklopovljem spremničkog međusklopa, i to tako da adresiranje stranice koja u tablici prevođenja adresa ima u bitu prisutnost zapisanu vrijednost 0 izazove prekid. Taj prekid je poziv funkcije u dijelu operacijskog sustava koji se bavi gospodarenjem spremničkog prostora. Posljedice tog prekidanja opisat će o nakon što razmotrimo još neka svojstva sklopovlja koje podupire straničenje.

8.4.2. Opisnik virtualnog adresnog prostora

Do sada smo, dakle, ustanovili da svaki program koji je pripremljen za izvođenje mora biti podijeljen na stranice i da za svaki program mora postojati tablica prevođenja adresa. Program prilikom izvođenja nazivamo procesom i čitav adresni prostor nazvat ćemo *procesnim adresnim prostorom* ili *procesnim spremničkim prostorom*. U 4. smo poglavljju jednostavno pretpostavili da taj prostor postoji i da se unutar njega može načiniti podjela na dretvene potprostore. Nadalje, u 5. smo poglavljju ustanovili da se u opisniku svake dretve između ostalih podataka nalaze i podaci o dretvenom potprostoru.

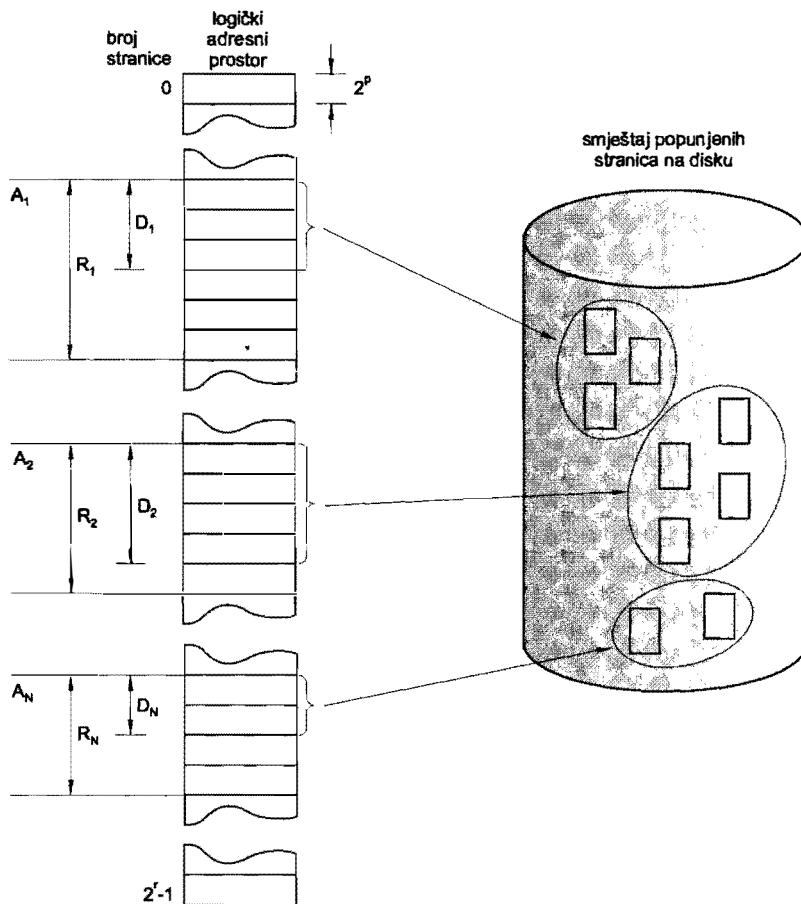
Već smo u uvodnom odjeljku ovoga poglavlja ustanovili da za svaki program (koji se pri izvođenju oblikuje u proces) u strukturi podataka operacijskog sustava postoji procesni informacijski blok. U taj se informacijski blok uz opisnike dretvi mora uključiti i *opisnik virtualnog procesnog adresnog prostora*.

Iako je teorijski moguće da procesni adresni prostor u m -bitovnoj arhitekturi može zauzimati svih 2^m adresa, u praktičnim postupcima gospodarenja spremničim prostorom prikladno je ograničiti veličinu virtualnog adresnog prostora koji će se stvarno koristiti. Naime, ako se cijeli adresni prostor trajno stavlja na raspolaganje programu, onda se trajno mora podržavati velike tablice za prevođenje logičkih u fizičke adrese. Iz primjera 8.4. vidljivo je da bi u 32-bitovnoj arhitekturi uz veličine stranica od 4 KB tablica prevođenja morala imati 1 M redaka. Nadalje, za smještanje svih mogućih stranica logičkog adresnog prostora na disk trebalo bi svakom procesu osigurati prostor od 4 GB. Zbog toga je razumno logički adresni prostor ograničiti u skladu sa stvarnim potrebama pojedinih programa.

Programska pomagala za pripremanje programa osigurat će osnovni adresni prostor za instrukcije i stog svake dretve te neki minimalni prostor za lokalne podatke dretve, kao i zajednički procesni podatkovni prostor. Uobičajeno se podatkovni prostor dijeli na tzv. staticki dio koji se tijekom odvijanja procesa neće mijenjati i dinamički dio koji se tijekom izvođenja koristi za podatke koji nastaju i nestaju. Takvi se podaci mogu naizmjence smještati u isti adresni prostor. Pritom se posebnim API funkcijama predviđenim za gospodarenje spremnikom programske može tražiti rezerviranje područja logičkih adresa te oslobodanje nekih područja logičkog adresnog prostora.

Nadalje, ne moraju sve stranice rezerviranog prostora biti u uporabi. Posebnim API funkcijama mogu se unutar rezerviranog adresnog prostora pojedine stranice stvarno dodjeljivati i oslobođavati. Smještaj na disku potrebno je osigurati samo za dodijeljene stranice,

tj. za popunjeni dio logičkog adresnog prostora. Prema tome, logički adresni prostor jednog procesa možemo simbolički prikazati slikom 8.17.



Slika 8.17. Rezervirani i dodijeljeni dio logičkog adresnog prostora

Za proces se od teorijski mogućih 2^r stranica rezervira ukupno:

$$R = \sum_{i=1}^N R_i$$

stranica, od kojih je trenutačno dodijeljeno njih ukupno

$$D = \sum_{i=1}^N D_i.$$

U opisniku virtualnog adresnog prostora mora se nalaziti tablica koja opisuje rezervirani adresni prostor. Ta tablica može biti jednostavna i opisivati samo granice rezerviranih adresa. To znači da za skupinu rezerviranih stranica koja započinje stranicom s rednim

brojem A_i iima R_i stranica jednostavno možemo zapisati granice pojedinih područja rezerviranih adresa, tj. $A_i \cdot 2^p$ i $(A_i + R_i) \cdot 2^{p-1}$ (gdje je p broj bitova relativne adrese a 2^p veličina stranice). Pokušaj pristupa do adresa izvan rezerviranih područja izazvat će prekidanje programa.

U tablici popunjenoj logičkog adresnog prostora mora se još dodatno za svaku stranicu nalaziti redak koji sadrži adresu sektora na disku u kojem je stranica pohranjena (odnosno adresu prvog od nakupine sektora ako je stranica pohranjena u više uzastopnih sektora).

Dakle, sve stranice programa pripremljenog za izvođenje moraju biti smještene na disku, a opisnik virtualnog adresnog prostora za svaku stranicu sadrži jedan redak. Iz tog se opisnika može pročitati u koje su sektore na disku pohranjene sve popunjene stranice. Vidjet ćemo u sljedećem poglavlju da se na sličan način na diskove pohranjuje i ostale sadržaje u obliku datoteka. Kada se program pohranjen u datoteku pokreće, onda će se uz tablicu koja opisuje smještaj na disku unutar operacijskog sustava stvoriti i tablica prevodenja logičkih adresa u fizičke adrese. Kao što smo ustanovili u opisu slike 8.16., za sve stranice koje su smještene u neki od okvira radnog spremnika u tablici prevodenja nalazi se broj okvira fizičkog spremnika u koji je stranica smještena. Uz to, u svakom retku tablice prevodenja nalazi se i bit prisutnosti koji ukazuje da je stranica premještena s diska u radni spremnik. Tijekom izvođenja programa opisnik virtualnog adresnog prostora nadopunjuje se dakle tablicom prevodenja, tako da u svakom trenutku za svaku stranicu jednoznačno znamo gdje je ona smještena na disku i gdje u radnom spremniku (ako se u njemu nalazi). Na taj je način potpuno opisan procesni adresni prostor.

8.4.3. Priručni međuspremnik za prevodenje adresa

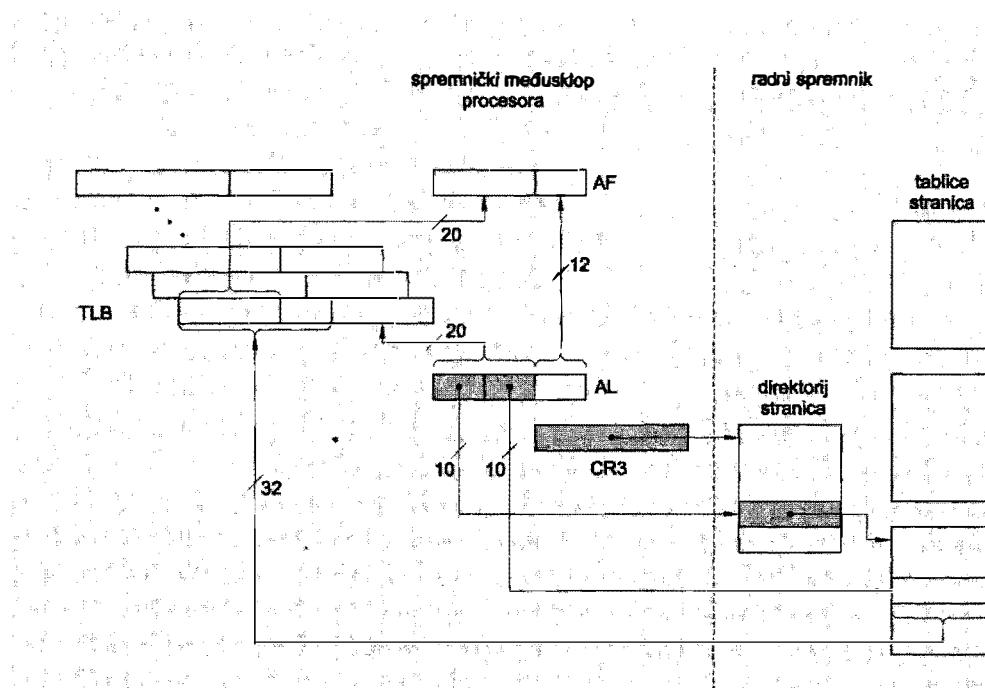
Operacijski sustav također organizira svoj dio spremničkog prostora straničenjem. Tako su i tablice prevodenja pohranjene u stranice logičkog adresnog prostora dodijeljenog operacijskom sustavu. Prilikom pokretanja programa operacijski sustav će iz podataka opisnika virtualnog adresnog prostora stvoriti tablicu prevodenja i dodijeliti programu (koji time postaje proces) neke početne okvire kako bi on mogao početi svoje izvođenje. Pojedini procesori sklopovski podupiru organizaciju tablica prevodenja i tako olakšavaju njihovo oživotvorenenje.

Pokazat ćemo na primjeru arhitekture *Intel x86* sklopovsku podlogu za ostvarenje straničenja.

PRIMJER 8.7.

U arhitekturi *Intel x86* organizacija tablica potpomognuta je na način ilustriran slikom 8.18. Unutar procesora nalazi se jedan 32-bitovni registar koji služi za smještanje kazaljke koja pokazuje na tablicu prevodenja (taj registar označen je u opisu procesora sa *CR3*). Svaki proces ima svoju tablicu prevodenja. Prema tome, ako procesor izvodi dretve unutar istog procesnog adresnog prostora, sadržaj tog registra ne treba mijenjati. Međutim, kada se procesor prebacuje na izvođenje dretve iz nekog drugog procesa, tada pri promjeni konteksta dretvi treba mijenjati i sadržaj tog registra.





Slika 8.18. Sklopovska podloga za ostvarenje straničenja arhitekture Intel x86

Tablica prevodenja organizira se kao dvorazinska tablica adresa koja je također organizirana po stranicama veličine 4 KB. U prvoj se razini nalazi jedna stranica s 1024 kazaljki od po 32 bita. U arhitekturi Intel x86 tu stranicu nazivaju direktorijem stranica (engl. *page directory*). Kazaljke direktorija pokazuju na stranice u kojima je opet smješteno po 1024 riječi od po 32 bita. Te se stranice nazivaju tablicama stranica (engl. *page tables*). Gornjih 20 bitova riječi sadržavat će početnu adresu (redni broj) okvira ako se stranica nalazi u radnom spremniku. Ako stranica nije u radnom spremniku, sadržaj tih dvadeset bitova nema nikakvo značenje. Donjih 12 bitova može se upotrijebiti za označavanje različitih stanja stranice. Jedan od tih bitova već je spomenut bit prisutnosti koji ukazuje je li stranica smještena u radnom spremniku.

Na slici 8.18. simbolički je prikazano prevodenje logičke adrese u fizičku adresu. Registar CR3 pokazuje na direktorij stranica procesa. Gornjih 10 bitova logičke adrese (koja se smješta u registar AL) izabire iz tog direktorija kazaljku na jednu od tablica stranica. Sljedećih 10 bitova logičke adrese odabire iz te tablice adresu stranice. Svaki redak tablice stranica ima također 32 bita. Ta se 32 bita prepisuju u jedan od registara *priručnog meduspremnika za prevodenje adresa* koji se nalazi na procesorskom čipu (engl. *translation lookaside buffer – TLB*). Gornjih 20 bitova iz tog registra čine gornjih 20 bitova fizičke adrese, dok donjih 12 bitova fizičke adrese dolazi neposredno iz logičke adrese.

Ustanovimo da je pri opisanom načinu oblikovanja fizičke adrese za pristup do neke adresirane lokacije potrebno tri puta pristupiti radnom spremniku:

- u prvom se pristupu s pomoću sadržaja registra *CR3* i gornjih 10 bitova logičke adrese pristupa do direktorija stranica i dobavlja kazaljka na tablicu stranica;
- u drugom se pristupu s pomoću adrese dobivene iz direktorija stranica i pomaknuća određenog sa sljedećih 10 bitova logičke adrese dobavlja u procesor odgovarajući redak tablice prevođenja;
- konačno, tek u trećem pristupu spremniku dohvata se adresirana lokacija.

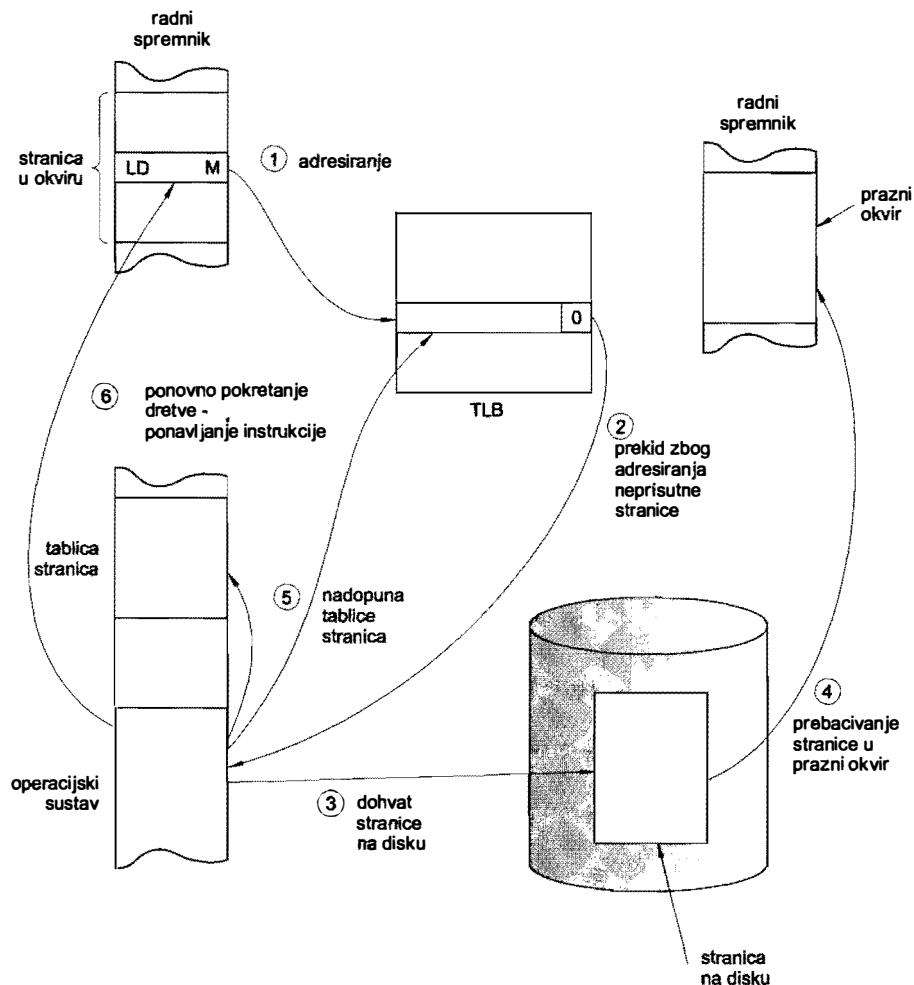
Drugim riječima, uz ovakav način adresiranja samo bi jedna trećina širine pojasa sabirnice bila korisno iskorištena.

Zbog toga priručni spremnik za prevođenja adresa *TLB* ima više registara u koji se pohranjuje više rednih brojeva okvira koji su zadnji dohvaćani. Uz svaki od tih registara postoji i dodatni 20-bitovni registar u koji se prepisuje gornjih dvadeset bitova logičke adrese iz registra *AL* (tj. redni broj stranice). Tako se u priručnom međuspremniku *TLB* pohranjuju parovi rednih brojeva stranica i pripadnih okvira u kojima se pojedine stranice nalaze. Postupak oblikovanja fizičke adrese znatno se ubrzava ako se ponovno adresira neka od stranica čiji se redni broj već nalazi u *TLB*-u. Naime, sklopolje spremničkog međusklopa najprije uspoređuje gornjih 20 bitova svake nove adrese koja dolazi u registar *AL* sa svim sadržajima pohranjenim u priručni spremnik *TLB* te, bez i jednog pristupa do radnog spremnika, oblikuje fizičku adresu ako se redni broj stranice i pripadni redni broj okvira u međuspremniku već nalaze. U supotnom sklopolje će započeti gore opisani postupak pribavljanja odgovarajućeg retka stranice prevođenja iz radnog spremnika. Prema tome, ako procesor uzastopno generira adrese unutar jedne stranice (ili unutar nekoliko stranica čiji se podaci o prevođenju nalaze unutar priručnog međuspremnika *TLB*), pristup do neke virtualne adrese može se obaviti u jednom spremničkom ciklusu.

Naglasimo da se opisani postupak prevođenja adresa obavlja potpuno sklopovski i da se punjenje međuspremnika *TLB* obavlja automatski bez ikakvog programskog djelovanja. Međutim, to je istinito samo onda kada se stranica već nalazi u nekom od okvira radnog spremnika, tj. kada bit prisutnosti stranice ima vrijednost 1. Ako se stranica ne nalazi u radnom spremniku, tada će sklopolje izazvati prekid čime započinje programska intervencija operacijskog sustava.

8.4.4. Straničenje na zahtjev

Već smo ustanovili da se pri pokretanju nekog programa mora uspostaviti opisnik virtualnog adresnog prostora i rezervirati odgovarajući prostor na disku. U radni se spremnik mora smjestiti barem jedna stranica, i to ona u kojoj se nalaze početne instrukcije strojnog programa. Sve ostale stranice mogu se dobavljati u trenutku kada se za njima ukaže potreba, tj. kada se unutar procesora bude generirala adresa kojom se zahtijeva pristup do neke stranice. Takav način straničenja dobio je naziv straničenje na zahtjev (engl. *demand paging*).



Slika 8.19. Aktivnosti obrade prekida zbog adresiranja neprisutne stranice

Na slici 8.19. označen je zaokruženim brojevima redoslijed aktivnosti koje je potrebno obaviti prilikom obrade prekida zbog adresiranja neprisutne stranice. Radni spremnik nacrtan je u dijelovima kako bi se pojednostavio izgled slike. Prepostavimo da jedna instrukcija adresira podatak na adresi **M** u stranici koja nije prisutna u radnom spremniku. Jezgra operacijskog sustava prepoznat će uzrok prekida, blokirati dretvu koja se izvodila i pokrenuti operaciju punjenja stranice. U tablici popunjenoj adresnog prostora stranica pronaći će broj sektora na disku u kojem je stranica pohranjena i u posebnoj tablici praznih okvira jedan prazan okvir radnog spremnika (u sljedećem ćemo odjeljku raspraviti što se zbiva kada praznih okvira nema). Operacijski sustav zatim prenosi upravljačkom sklopu diskovne jedinke nalog za prijenos stranice s diska u prazni okvir. Podsjetimo se da je trajanje prijenosa određeno vremenskim svojstvima diska i da će operacijski sustav prekidom od upravljačkog sklopa diskovne jedinke biti obaviješten o završetku prijenosa.

Nakon punjenja stranice mora se nadopuniti tablica stranica i prekinuta dretva se može pokrenuti ponavljanjem instrukcije koja je pokušala adresirati neprisutnu stranicu.

Napomenimo da se procesor pri obradi ovog prekida mora ponašati malo drukčije nego li kod obrada drugih vrsta prekida. Naime, procesor uobičajeno ispituje postojanje prekida pri kraju svake instrukcije i, nakon što tekuću instrukciju izvede do kraja, prihvata prekid. Programsko brojilo pritom sadrži adresu sljedeće instrukcije i ta se vrijednost pohranjuje u kontekstu dretve. Ponovno pokretanje dretve započinje, dakle, sljedećom instrukcijom. Međutim, nakon obrade prekida zbog adresiranja nepostojeće stranice mora se *ponoviti instrukcija* unutar koje se pojavio prekid. Prema tome, u kontekst dretve treba pohraniti one sadržaje registara procesora koji su u njima pohranjeni *prije početka* izvođenja instrukcije koja je izazvala prekid. Zbog toga procesori predviđeni za ostvarivanje virtualnog spremnika imaju posebne skrivene kopije registara u kojima se čuvaju nepromijenjene vrijednosti registara tijekom izvođenja cijele instrukcije. Vrijednosti se tih registara obnavljaju tek na kraju izvođenja svake instrukcije. Pri pojavi prekida zbog adresiranja neprisutne stranice u kontekst dretve pohranjuju se skrivene kopije registara, dakle, vrijednosti koje odgovaraju stanju procesora prije početka izvođenja instrukcije. Ponovno pokretanje prekinute dretve započet će ponavljanjem instrukcije koja je izazvala prekid.

Očigledno je da dobavljanje stranica u radni spremnik usporava izvođenje dretvi. Podsjetimo se da je trajanje prebacivanja stranice s diska u radni spremnik određeno vremenskim svojstvima diska i da preveliki broj prekida zbog adresiranja neprisutnih stranica može zamjetno usporiti izvođenje programa.

PRIMJER 8.8.

Za ilustraciju mogućeg usporavanja programa zbog straničenja pretpostaviti ćemo da dretva tijekom izvođenja pristupa do radnog spremnika u svakom spremničkom ciklusu čije je trajanje T_B . Pretpostaviti ćemo, nadalje, da se u svakih N pristupa do radnog spremnika dogodi jedan prekid zbog adresiranja neprisutne stranice. Neka se za punjenje stranice u radni spremnik utroši vrijeme T_D . Prema tome, za izvođenje N pristupa do radnog spremnika utrošiti će se ukupno vrijeme od:

$$NT_B + T_D,$$

te će prosječno trajanje jednog pristupa spremniku biti jednak:

$$T_P = \frac{NT_B + T_D}{N} = T_B + \frac{1}{N}T_D.$$

Prepostavimo da spremnički ciklus iznosi:

$$T_B = 100 \text{ ns} = 0.1 \mu\text{s},$$

a da se diskom s vremenskim svojstvima kao u primjeru 8.1. može postići prosječno vrijeme:

$$T_D = 15 \text{ ms} = 15000 \mu\text{s}.$$



Prosječno trajanje jednog pristupa spremniku iznosit će u tom slučaju:

$$T_P = \left(0.1 + \frac{15\,000}{N} \right) \mu s.$$

Pogledajmo koliko iznosi prosječno trajanje pristupa za neke vrijednosti N .

N	T_P	μs
10^3	15.1	μs
10^4	1.6	μs
10^5	0.25	μs
10^6	0.115	μs
10^7	0.01015	μs

Iz ove je tablice vidljivo da se prosječno trajanje pristupa spremniku povećava 150 puta ako se na svakih 1000 pristupa spremniku dogodi prekid zbog adresiranja nepostojecne stranice. Ako se, međutim, dobavljanje neprisutne stranice događa jedanput pri milijunu pristupa, pogoršanje prosječnog trajanja pristupa iznosit će samo prihvatljivih 15%.

8.4.5. Strategije zamjene stranica

Čiste i nečiste stranice

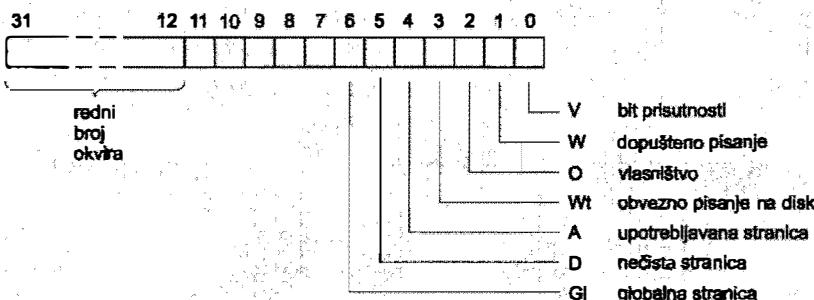
U prethodnom smo odjeljku prepostavili da u fizičkom spremniku uvijek postoji prazni okvir u koji se može smjestiti novootražena stranica. Međutim, ako je veličina programa veća od raspoloživog fizičkog adresnog prostora, onda će se s vremenom svi okviri popuniti i punjenje nove stranice moći će se obaviti samo tako da se neki od punih okvira isprazni. Podsetimo se da sve stranice procesnog adresnog prostora imaju svoj smještaj na disku. Prema tome, ako neki okvir treba isprazniti, onda stranicu koja je u njemu smještena moramo prepisati natrag na njezino mjesto na disku.

Prosječno trajanje pražnjenja bit će jednako prosječnom trajanju punjenja stranice. No, ako se sadržaj stranice tijekom njezina boravka u radnom spremniku, tj. u razdoblju od trenutka njezina smještanja do trenutka njezina izbacivanja iz radnog spremnika nije mijenjao, onda stranicu ne treba stvarno prepisivati na disk jer na disku postoji njezina vjerna kopija. Prema tome, u okvir u kojem se nalazi takva *čista stranica* možemo jednostavno smjestiti novu stranicu. U tablicu stranica zbog toga se uvodi posebni bit koji označava je li stranica čista ili nečista.

Pokazalo se korisnim da se u tablicu stranica uvedu još neki bitovi iz kojih se može zaključiti u kakvom je stanju stranica.

**PRIMJER 8.9.**

Kao što smo spomenuli u primjeru 8.7., u arhitekturi porodice Intel x86 donjih 12 bitova svakog retka tablice stranica može se upotrijebiti za označavanje različitih stanja stranice. Slika 8.20. prikazuje neke od tih bitova koje možemo nazvati zastavicama⁵. Spremnik međusklop procesora neposredno djeluje na postavljanje nekih od tih zastavica, a isto tako uskladjuje svoje djelovanje u skladu s nekim vrijednostima zastavica.



Slika 8.20. Zastavice tablice stranica u arhitekturi Intel x86

Pogledajmo funkcije tih osnovnih zastavica:

- Zastavica *V* je bit prisutnosti stranice u radnom spremniku i njezina je uloga već detaljno opisana.
- Zastavica *D* pokazuje je li stranica čista ili nečista. Ta se zastavica postavlja u vrijednost 0 prilikom svakog punjenja stranice u neki okvir radnog spremnika. Pri prvom pisanju u stranicu zastavica se postavlja u vrijednost 1 i time označava da je stranica nečista.
- Zastavica označena s *A* označava je li se u stranicu pristupalo (čitanjem ili pisanjem). Taj se bit također pri punjenju stranice u neki okvir postavlja u vrijednost 0, a pri pristupanju u stranicu postavlja se u vrijednost 1. Za razliku od zastavice *D*, ta se zastavica može programski pobrisati tijekom boravka stranice u radnom spremniku, čime se omogućuju neki algoritmi odabira stranice koju treba izbaciti iz njezina trenutačnog okvira.
- Postavljena zastavica *W* označava da se u stranicu ne smije pisati. Pokušaj pisanja u tako označenu stranicu izazvat će prekid koji ukazuje na pogrešnu uporabu stranice (ta zastavica može imati posebnu ulogu u više procesorskim sustavima).
- Zastavica *O* ukazuje da li se do stranice smije pristupiti u korisničkom ili sustavskom načinu rada procesora (na taj se način mogu posebno zaštititi stranice koje pripadaju ježgri).
- Uz postavljenu zastavicu *W*, svako pisanje u stranicu izazvat će prekid kojim se inicira trenutačno pohranjivanje promijenjene stranice na disk. Ako ta zastavica nije postavljena, sva pisanja mijenjaju samo sadržaj kopije stranice u radnom spremniku, dok njezin izvorni oblik na disku ostaje nepromijenjen. Promjena sadržaja na disku dogodit će se u tom slučaju samo kod izbacivanja stranice.

- Zastavica *Gl* ukazuje da se stranica koja se nalazi u tom okviru smatra globalnom, tj. da pripada adresnim prostorima više procesa. Preko tako označenih stranica operacijski sustav može organizirati razmijenu podataka između procesa.

8.4.6. Teorijske strategije zamjene stranica

Praktična ostvarenja postupaka odabira stranice koja će biti izbačena iz svog okvira zasnovaju se na nekim teorijski zasnovanim strategijama od kojih valja spomenuti:

- *FIFO* strategiju kod koje se za izbacivanje odabire stranica koja je najdulje u radnom spremniku (*FIFO* dolazi od engl. *first-in, first-out*);
- *LRU* strategiju kod koje će izbacuje stranica koja se u najdaljoj prošlosti nije upotrebljavala s nadom da takva stranica više neće biti potrebna (*LRU* dolazi od engl. *least recently used*);
- optimalnu strategiju (*OPT*) kod koje se za izbacivanje odabire stranica koja se u najdaljoj budućnosti neće upotrebljavati.

Kao mjera za usporedbu strategija može poslužiti broj prekida zbog adresiranja neprisutnih stranica (taj se prekid može nazvati *promašajem stranice*). S teorijskog bi stanovišta bila najprikladnija optimalna strategija jer ona odabire stranice koje se najdalje u budućnosti neće upotrebljavati, a među njima su i takve koje se više uopće neće upotrebljavati. Međutim, ta se strategija ne može praktično upotrebljavati jer bi se unaprijed moralo poznavati ponašanje cijelog procesa (koje i ne mora biti jednoznačno). Optimalna strategija stoga može samo poslužiti kao teorijska granica mogućeg ponašanja programa.

FIFO i *LRU* strategija mogu se također samo približno praktično primijeniti. Jasno je da broj prekida zbog adresiranja neprisutnih stranica ovisi o broju raspoloživih okvira. Kod straničenja na zahtjev možemo prepostaviti da se i punjenje prve stranice broji kao promašaj stranice.

PRIMJER 8.10.



Pretpostavimo da procesni adresni prostor ima ukupno šest stranica koje možemo označiti njihovim rednim brojevima 0, 1, 2, 3, 4, 5. Neka se pri izvođenju procesa te stranice dohvataju sljedećim redoslijedom: 0, 1, 2, 3, 4, 0, 1, 5, 0, 1, 2, 3, 4, 5. Pogledajmo kako broj promašaja ovisi o broju raspoloživih okvira, i to najprije na slici 8.21. za *FIFO* strategiju i na slici 8.22. za *LRU* strategiju.

Na slici je vidljivo da uz jedan raspoloživi okvir dobivamo 14 promašaja jer svaki put kada trebamo novu stranicu nju moramo smjestiti u taj jedini okvir. Kada na raspolažanju imamo više okvira, onda se na zahtjev postupno popunjavaju svi okviri i tek tada kada su oni svi puni započinje izbacivanje stranica. U ovom se primjeru uz dva raspoloživa okvira broj promašaja nije smanjio. Tek kod tri okvira broj promašaja smanjio se na 12 jer se pri trećem pristupanju do stranica 0 i 1 one nalaze u okvirima radnog spremnika.

redoslijed dohvata stranica	0	1	2	3	4	0	1	5	0	1	2	3	4	5	broj promašaja
1 okvir	0	1	2	3	4	0	1	5	0	1	2	3	4	5	14
2 okvira	0	0	2	2	4	4	1	1	0	0	2	2	4	4	14
3 okvira	0	0	0	3	3	3	1	1	1	1	3	3	3	12	
4 okvira	0	0	0	0	4	4	4	4	1	1	2	2	2	1	11
5 okvira	0	0	0	0	0	1	1	5	5	5	5	5	4	4	12
6 okvira	0	0	0	0	0	1	1	1	1	0	0	0	0	5	6
	-	1	1	1	1	-	-	-	-	-	-	-	-	-	
	-	-	2	2	2	2	2	1	1	-	-	1	1	4	
	-	-	-	3	3	3	3	3	5	-	-	5	5	5	
	-	-	-	-	4	1	1	4	4	4	4	3	3	3	
	-	-	-	-	-	-	-	-	-	-	-	-	-	-	

Slika 8.21. Ponašanje sustava virtualnog spremnika uz FIFO strategiju izbacivanja stranica

Na slici 8.21. prazni su okviri (prije prvog smještanjā neke stranice) označeni s horizontalnim crticama, a vertikalne crtice u svim okvirima označavaju da im je svima sadržaj ostao nepromijenjen, tj. da se dohaća stranica koja se već nalazi u radnom spremniku i da pritom nema promašaja. Ovakav način označavanja olakšava brojenje promašaja.

Broj promašivanja s povećanjem broja raspoloživih okvira u načelu bi trebao opadati, ali iz našeg se primjera vidi da to i ne mora biti tako. S četiri okvira broj promašivanja smanjuje se na 11, ali s pet raspoloživih okvira dobivamo opet 12 promašaja. Po čovjeku koji je prvi ukazao na takvo moguće ponašanje virtualnog spremnika taj se fenomen naziva Beladyjevom anomalijom.

Jasno je da nema smisla procesu dodijeliti broj okvira veći od broja stranica programa jer će neki okviri ostati prazni. Ukupni broj promašaja jednak je broju stranica jer, kada su sve stranice smještene u dodijeljene im okvire, dalnjih promašivanja više nema.

Slika 8.22. prikazuje ponašanje sustava pri primjeni *LRU* strategije. U ovom jednostavnom primjeru ne vide se neke bitnije razlike u broju promašivanja.

redoslijed dohvata stranica	0	1	2	3	4	0	1	5	0	1	2	3	4	5	broj promašaja
1 okvir	0	1	2	3	4	0	1	5	0	1	2	3	4	5	14
2 okvira	0	0	2	2	4	4	1	1	0	0	2	2	4	4	14
	-	1	1	3	3	0	0	5	5	1	1	3	3	5	
3 okvira	0	0	0	3	3	3	1	1	1	1	1	1	4	4	12
	-	1	1	1	4	4	4	5	1	1	2	2	2	5	
	-	-	2	2	2	0	0	0	1	1	0	3	3	3	
4 okvira	0	0	0	0	4	4	4	4	1	1	2	2	2	1	12
	-	1	1	1	1	0	0	0	1	1	0	0	4	4	
	-	-	2	2	2	2	1	1	1	1	1	1	1	5	
	-	-	-	3	3	3	3	5	1	1	5	3	3	3	
5 okvira	0	0	0	0	0	0	0	0	0	0	0	0	0	5	10
	-	1	1	1	1	1	1	1	1	1	1	1	1	1	
	-	-	2	2	2	2	1	1	1	1	5	5	4	4	
	-	-	-	3	3	3	3	3	1	1	2	2	2	2	
	-	-	-	-	4	1	1	4	1	1	4	3	3	3	
6 okvira	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6
	-	1	1	1	1	1	1	1	1	1	1	1	1	1	
	-	-	2	2	2	2	1	1	2	1	1	1	1	1	
	-	-	-	3	3	3	3	3	1	1	1	1	1	1	
	-	-	-	-	4	1	1	4	1	1	1	1	1	1	
	-	-	-	-	-	5	1	1	1	1	1	1	1	1	

Slika 8.22. Ponašanje virtualnog spremnika uz *LRU* strategiju izbacivanja stranica

Konačno, slika 8.23. pokazuje kako bi na ponašanje sustava djelovala optimalna strategija izbacivanja stranica. Ponašanje sustava nije sasvim jednoznačno. Naime, kada se u okvirima radnog spremnika nađe više od jedne stranice koje se više uopće neće upotrebljavati, tada se može izbaciti bilo koja od njih. U primjeru na slici 8.23. odabранa je za izbacivanje stranica iz gornjeg okvira.

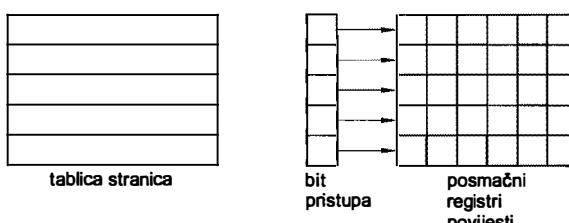
redoslijed dohvata stranica	0	1	2	3	4	0	1	5	0	1	2	3	4	5	broj promašaja
1 okvir	0	1	2	3	4	0	1	5	0	1	2	3	4	5	14
2 okvira	0	0	0	0	0	0	0	0	1	2	3	4	1	1	11
	-	1	2	3	4	1	1	5	1	5	5	5	5	1	

3 okvira	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td></td><td></td><td>0</td><td></td><td></td><td>2</td><td>3</td><td>4</td><td></td><td>9</td></tr><tr><td>-</td><td>1</td><td>1</td><td>1</td><td>1</td><td></td><td></td><td>1</td><td></td><td></td><td>1</td><td>1</td><td>1</td><td></td><td></td></tr><tr><td>-</td><td>-</td><td>2</td><td>3</td><td>4</td><td></td><td></td><td>5</td><td></td><td></td><td>5</td><td>5</td><td>5</td><td></td><td></td></tr></table>	0	0	0	0	0			0			2	3	4		9	-	1	1	1	1			1			1	1	1			-	-	2	3	4			5			5	5	5																																															
0	0	0	0	0			0			2	3	4		9																																																																													
-	1	1	1	1			1			1	1	1																																																																															
-	-	2	3	4			5			5	5	5																																																																															
4 okvira	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td></td><td></td><td>0</td><td></td><td></td><td>3</td><td>4</td><td></td><td></td><td>8</td></tr><tr><td>-</td><td>1</td><td>1</td><td>1</td><td>1</td><td></td><td></td><td>1</td><td></td><td></td><td>1</td><td>1</td><td></td><td></td><td></td></tr><tr><td>-</td><td>-</td><td>2</td><td>2</td><td>2</td><td></td><td></td><td>2</td><td></td><td></td><td>2</td><td>2</td><td></td><td></td><td></td></tr><tr><td>-</td><td>-</td><td>-</td><td>3</td><td>4</td><td></td><td></td><td>5</td><td></td><td></td><td>5</td><td>5</td><td></td><td></td><td></td></tr></table>	0	0	0	0	0			0			3	4			8	-	1	1	1	1			1			1	1				-	-	2	2	2			2			2	2				-	-	-	3	4			5			5	5																																	
0	0	0	0	0			0			3	4			8																																																																													
-	1	1	1	1			1			1	1																																																																																
-	-	2	2	2			2			2	2																																																																																
-	-	-	3	4			5			5	5																																																																																
5 okvira	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td></td><td></td><td>0</td><td></td><td></td><td></td><td></td><td>4</td><td></td><td>7</td></tr><tr><td>-</td><td>1</td><td>1</td><td>1</td><td>1</td><td></td><td></td><td>1</td><td></td><td></td><td></td><td></td><td>1</td><td></td><td></td></tr><tr><td>-</td><td>-</td><td>2</td><td>2</td><td>2</td><td></td><td></td><td>2</td><td></td><td></td><td></td><td></td><td>2</td><td></td><td></td></tr><tr><td>-</td><td>-</td><td>-</td><td>3</td><td>3</td><td></td><td></td><td>3</td><td></td><td></td><td></td><td></td><td>3</td><td></td><td></td></tr><tr><td>-</td><td>-</td><td>-</td><td>-</td><td>4</td><td></td><td></td><td>5</td><td></td><td></td><td></td><td></td><td>5</td><td></td><td></td></tr></table>	0	0	0	0	0			0					4		7	-	1	1	1	1			1					1			-	-	2	2	2			2					2			-	-	-	3	3			3					3			-	-	-	-	4			5					5																	
0	0	0	0	0			0					4		7																																																																													
-	1	1	1	1			1					1																																																																															
-	-	2	2	2			2					2																																																																															
-	-	-	3	3			3					3																																																																															
-	-	-	-	4			5					5																																																																															
6 okvira	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td></td><td></td><td>0</td><td></td><td></td><td></td><td></td><td></td><td></td><td>6</td></tr><tr><td>-</td><td>1</td><td>1</td><td>1</td><td>1</td><td></td><td></td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>-</td><td>-</td><td>2</td><td>2</td><td>2</td><td></td><td></td><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>-</td><td>-</td><td>-</td><td>3</td><td>3</td><td></td><td></td><td>3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>-</td><td>-</td><td>-</td><td>-</td><td>4</td><td></td><td></td><td>4</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td><td></td><td></td><td>5</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	0	0	0	0	0			0							6	-	1	1	1	1			1								-	-	2	2	2			2								-	-	-	3	3			3								-	-	-	-	4			4								-	-	-	-	-			5							
0	0	0	0	0			0							6																																																																													
-	1	1	1	1			1																																																																																				
-	-	2	2	2			2																																																																																				
-	-	-	3	3			3																																																																																				
-	-	-	-	4			4																																																																																				
-	-	-	-	-			5																																																																																				

Slika 8.23. Ponašanje virtualnog spremnika uz optimalnu strategiju izbacivanja stranica

8.4.7. Praktične aproksimacije strategija zamjene stranica

Praktično ostvarenje strategija izbacivanja stranica mora biti takvo da se izbor stranice koja će biti izbačena obavi što je moguće brže. Bilo bi prikladno da odabir bude što je više moguće poduprto sklopoljcem. Jedna od mogućih zamisli takve sklopovske potpore ilustrirana je slikom 8.24.



Slika 8.24. Moguća sklopovska potpora ostvarenja aproksimacije LRU strategije

Na slici su posebno istaknuti bitovi pristupa stranici. Podsjetimo se da se bitovi pristupa prilikom punjenja stranica u neki od okvira postavljaju u vrijednost 0 i da se postavljuju prilikom svakog pristupa stranici (već prvi pristup mijenja početnu vrijednost u 1). Uz svaki redak tablice stranica možemo zamisliti i jedan posmačni registar u koji će se posmiciti bitovi pristupa.

Dio operacijskog sustava za gospodarenje spremničkim prostorom mogao bi djelovati na sljedeći način:

- Pri punjenju stranice u neki okvir briše se pripadni bit pristupa i registar povijesti.
- Prekid od sata periodno (primjerice, svaku sekundu) posmakne sve bitove pristupa u registre povijesti i zatim ih sve pobriše. Ponovni pristup u neku stranicu postavit će opet bit pristupa, a za one stranice u koje se u tekućoj periodi vremena nije pristupalo ostaje u bitu pristupa upisana vrijednost 0. Nakon nekoliko perioda sadržaj registara povijesti može se očitati kao binarni broj. Što je taj broj manji, u stranicu se dulje nije pristupalo.
- Kada se pojavi potreba za okvirom, izbacuje se stranica s najmanjim brojem u registru povijesti (ili jedna od stranica koje imaju jednake najmanje brojeve).

Ako registri povijesti imaju po $n - 1$ bitova, oni zajedno s bitom pristupa mogu pohraniti binarni broj od n bitova, pa se stranice "po starini" mogu razvrstati u 2^n razreda.

Međutim, ostvarenja spremničkih međusklopova današnjih procesora nemaju registara povijesti, te nam za razvrstavanje stranica preostaje samo jedan bit – bit pristupa (zastavica A u arhitekturi *Intel x86*). Prema tome, stranice možemo razvratiti samo u dva razreda:

- stranice u koje se u prethodnoj periodi otkucaja sata nije pristupalo i
- stranice u koje se u prethodnoj periodi pristupalo.

Kada se pojavi potreba pražnjenja jednog okvira, najprije se pokušava izbaciti stranica s vrijednošću 0 u zastavici A, i samo ako takvih stranica nema, onda se izbacuje stranica u kojoj je vrijednost te zastavice jednaka 1.

Nadalje, ustanovili smo da je iz okvira razumno izbacivati čiste stranice jer se time štedi jedan pristup do diska. U tablici stranica čistoća se stranice može ustanoviti na temelju posebne zastavice – bita čistoće (zastavica D u arhitekturi *Intel x86*). Prema tome, uz postojeću sklopovsku podlogu moguće je stranice prisutne u radnom spremniku razvrstati, umjesto u dva, u četiri razreda u skladu s tablicom 8.1. Kada se pojavi potreba za praznim okvirom, onda se pokušava pronaći stranicu tipa 0, onda se redom pokušava pronaći stranice viših tipova.

Tablica 8.1. Razredi stranica u radnom spremniku

Tip stranice	Bit pristupa A	Bit čistoće	Napomena
0	0	0	u prethodnoj periodi nije bilo pristupa i stranica je čista
1	0	1	u prethodnoj periodi nije bilo pristupa i stranica je nečista
2	1	0	u prethodnoj periodi bilo je pristupa i stranica je čista
3	1	1	u prethodnoj periodi bilo je pristupa i stranica je nečista

Svojevrsna inaćica *LRU* strategije upotrebljava se u više izvedaba *UNIX* operacijskih sustava i u operacijskom sustavu *Windows NT*. Tu inaćicu nazivaju *satnim algoritmom* (engl. *clock algorithm*). Sve stranice koje su smještene u okvire radnog spremnika razvrstavaju se u listu onim redom kako su ulazile u radni spremnik. Ta se lista može promatrati kao *FIFO* red i njome bi bilo moguće ostvariti *FIFO* strategiju. Međutim, lista se obilazi posebnom kazaljkom koja se s kraja liste vraća na njezin početak i tako se ostvaruje kružni obilazak liste (od tuda dolazi i naziv algoritma) i u neku ruku slijedi *LRU* strategija s dva razreda stranica. Kada se pojavi potreba za praznim okvirom, izbacuje se stranica na koju pokazuje kazaljka ako je njezin bit pristupa *A* jednak 0 i kazaljka se u listi pomiče za jedno mjesto. Stranica će se preskočiti ako je bit pristupa *A* jednak 1. Ako kazaljka kružno obide cijelu listu i pronalazi sve bitove pristupa jednake 1, onda će početi izbacivati stranice u koje se pristupalo.

U praktičnoj izvedbi satnog algoritma bitovi pristupa ne brišu se pod utjecajem posebnog prekida od sata koji smo spominjali pri opisu aproksimacije *LRU* strategije. Brisanje bitova obavlja se prilikom nailaska kazaljke na dotičnu stranicu. Naime, kada kazaljka nađe na stranicu za koju je bit pristupa jednak 1, taj će se bit pobrisati i kazaljka će se pomaknuti dalje za jedno mjesto. Kazaljka se pomiče tako dugo dok ne nađe na prvu stranicu s pobrisanim bitom pristupa. Ta će se stranica izbaciti iz okvira i tako načiniti mjesto za smještaj nove stranice. Prema tome, ako u kružnoj listi sve stranice imaju postavljene bitove pristupa, svi će oni biti pobrisani i kazaljka će se vratiti u početni položaj te za izbacivanje odabratи stranicu na koju je početno pokazivala. Podsjetimo se da tijekom zamjena stranica procesor ne izvodi dretvu korisničkog procesa i prema tome nije moglo doći do promjene u stanjima stranica.

Način zamjene stranica malo je drukčiji u višeprocesorskim sustavima. Naime, brisanje bita pristupa nije praktično izvoditi jer se može dogoditi da pojedinu stranicu dohvaca više dretvi istog procesa koje se izvode na različitim procesorima. U tom bi slučaju, dakle, u svim procesorima trebalo promjeniti bit pristupanja u njihovim priručnim međuspremnicima za prevodenje adresa (*TLB*), što je vrlo nepraktično. U tom slučaju satni algoritam izbacuje stranice temeljem čiste *FIFO* strategije.

8.4.8. Raspodjela okvira u višeprogramskom radu

Gospodarenje okvirima

Pri dosadašnjem razmatranju straničenja pretpostavljali smo da sve stranice pripadaju jednom procesu. Pritom smo ustanovili da ponašanje programa ovisi o broju raspoloživih okvira fizičkog radnog spremnika. U višeprogramskom radu raspoloživi se okviri moraju služiti za ostvarenje više međusobno razdvojenih virtualnih procesnih prostora.

Ustanovili smo u odjeljku 8.4.1. da za svaki proces postoje na disku sve stranice popunjene logičkog adresnog prostora. U opisniku adresnog prostora koji se nalazi u procesnom informacijskom bloku nalaze se adrese sektora na disku u koje su sve stranice smještene. Na disku su, dakle, procesni prostori potpuno razdvojeni.

Dio operacijskog sustava koji se bavi gospodarenjem spremničkog prostora mora osigurati da pojedini okviri fizičkog adresnog prostora budu dodijeljeni samo jednom od procesa. U primjeru 8.7. vidjeli smo da u arhitekturi *Intel x86* postoji posebni registar (označen simbolom *CR3*), koji služi za smještanje kazaljke koja pokazuje na tablice prevođenja pojedinih procesa. Operacijski će sustav za sve programe koji se izvode iz njihovih procesnih informacijskih blokova oblikovati tablice prevođenja i sve ih smjestiti u svoj dio adresnog prostora (koji je također ostvaren straničenjem). Za svaku tablicu mora biti poznata početna adresa koja će se smještati u registar *CR3*.

Dobro gospodarenje spremničkim prostorom zahtijeva poznavanje trenutačnog stanja svakog pojedinačnog okvira fizičkog radnog spremnika. Najprikladnije je sve potrebne podatke o okviru smjestiti u jedan opisnik. Opisnik mora imati i mjesta za kazaljke kako bi se lako mogao premještati iz liste u listu.

Tijekom rada pojedini okviri mogu se nalaziti u sljedećim osnovnim stanjima:

- u *aktivnom stanju* okvir je dodijeljen jednom od procesa i njegov se redni broj (kazaljka koja pokazuje na njega) nalazi u odgovarajućem retku tablice prevođenja tog procesa;
- u *slobodnom stanju* okvir se nalazi u povezanoj listi slobodnih okvira i može se dodjeljivati pojedinim procesima;
- u *slobodnom stanju s pobrisanim sadržajem* okvir je u posebnoj listi slobodnih okvira ali s pobrisanim sadržajem (engl. *zeroed*) jer se u načelu zbog sigurnosnih razloga okviri ne bi smjeli dodjeljivati novim procesima koji bi mogli neovlašteno pročitati stare sadržaje.

Osim toga, moguća su i neka međustanja okvira. Primjerice, kada se neki okvir oduzima procesu, nečistu stranicu koja se u njemu nalazi treba najprije prepisati na disk i tek onda se okvir može premjestiti u listu slobodnih okvira. Okvir će se nalaziti u nekom prije-laznom stanju i kada se iz liste slobodnih okvira seli u listu slobodnih pobrisanih okvira (posebna dretva operacijskog sustava vrlo niskog prioriteta može obavljati taj posao kada procesor nije zaposlen dretvama višeg prioriteta).

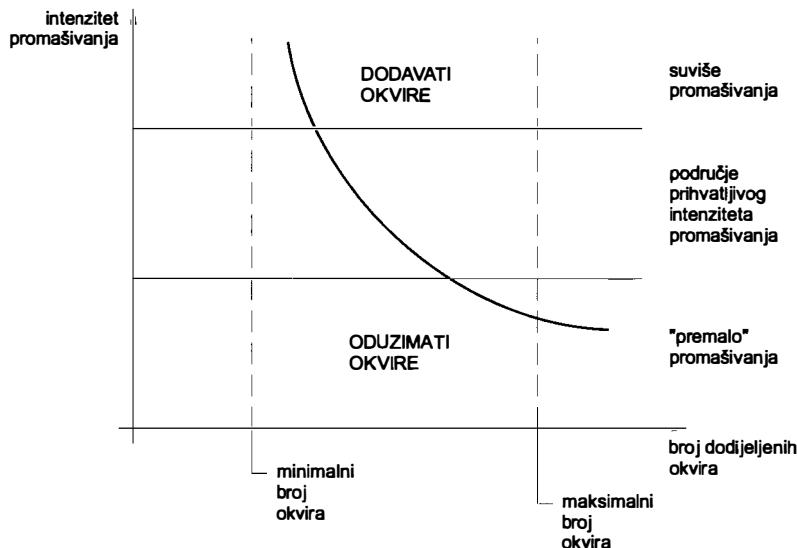
Tijekom rada neki okviri mogu pokazati i određene neispravnosti (koje se sklopovski mogu utvrditi). U tom se slučaju okviri mogu svrstati u posebnu listu *pokvarenih okvira*, koji više nikada neće biti dodjeljivani procesima.

Okvir prelazi iz stanja u stanje posredstvom pojedinih funkcija operacijskog sustava. Po-jedini se okvir u jednom trenutku može nalaziti samo u jednom od mogućih stanja.

8.4.9. Podjela okvira procesima

U višeprogramskom radu moguće je okvire raspodjeljivati ili tako da se svim procesima bez ograničenja dodjeljuju okviri u skladu s njihovim zahtjevima ili tako da se ograniči broj okvira koje pojedini proces može dobiti.

Prvi način dodjele okvira nije prikladan jer pojedini procesi koji intenzivno zahtijevaju nove stranice mogu usurpirati fizički adresni prostor i djelovati ograničavajuće na sve preostale procese.



Slika 8.25. Dinamičko namještanje raspodjele okvira procesima

Pokazalo se da djelotvornijim ograničiti broj okvira koje pojedini procesi mogu dobiti. Ako nam za korisničke programe na raspolaganju stoji M okvira i operacijski sustav dopušta istodobno odvijanje N procesa, onda se za svaki proces može predvidjeti M/N okvira. Tijekom izvođenja može se mjeriti intenzitet promašivanja stranica (prosječni broj promašaja u jedinici vremena). S obzirom na to da intenzitet promašivanja u načelu opada s povećanjem broja raspoloživih okvira, procesima koji suviše promašuju trebalo bi dodavati okvire, a procesima koji "premalo" promašuju mogu se okviri oduzimati. Dinamičko namještanje raspodjele broja okvira na procese zasnovano na tom načelu ilustrira slika 8.25. Razumljivo je da nema smisla smanjivati broj okvira nekom procesu ispod nekog minimalnog broja niti dopustiti nekom procesu uporabu suviše velikog broja okvira. Iskustveno se mogu odrediti donja i gornja granica raspodjele okvira. Ako svi procesi počinju suviše promašivati, onda novih okvira za raspodjelu više nema i jedini način razrješenja toga problema jest obustava nekih od procesa te preraspodjela svih oduzetih mu okvira preostalim procesima.

Sam operacijski sustav treba za svoje strukture podataka i sustavske dretve veći broj okvira koji su trajno zauzeti. Tako, primjerice, struktura podataka jezgre, kao i jezgrine funkcije moraju biti trajno smještene u radni spremnik. Isto tako, stranice koje služe kao međuspremnici za obavljanje ulazno-izlaznih operacija neposrednim pristupom spremniku (*DMA*) moraju biti stalno prisutne u radnom spremniku.

PRIMJER 8.11.

U operacijskom sustavu *Windows* inačice *NT 4.0* broj okvira koje će sustav početno dodijeliti procesu ovisi o veličini radnog spremnika i te predefinirane vrijednosti nije bilo moguće mijenjati. Novije inačice operacijskog sustava *Windows* dopuštaju dinamičku promjenu minimalnog i maksimalnog broja okvira koji mogu biti dodijeljeni jednom procesu. Primjerice, pretpostavljeni minimalni broj okvira koji proces dobiva iznosi 50, dok je maksimalna pretpostavljena vrijednost 345 okvira.

Odgovarajućim API funkcijama procesi mogu zatražiti promjenu ovih graničnih vrijednosti te time utjecati na ukupan broj dodijeljenih okvira. Pritom vrijedi pravilo prvenstva: procesu koji prvi zatraži povećanje maksimalnog broja okvira taj će zahtjev biti i odobren (ako ne prelazi broj raspoloživih okvira umanjen za 512). Svim ostalim procesima koji naknadno zatraže povećanje maksimalnog broja okvira, a koje bi narušilo prethodno odobrene vrijednosti, taj će zahtjev biti odbijen. Operacijski sustav u svakom slučaju neće dopustiti da broj slobodnih okvira padne ispod neke donje granice.

8.4.10. Radni skup

Skup okvira dodijeljen pojedinom procesu u operacijskom se sustavu *Windows* naziva radnim skupom (engl. *working set*). U literaturi se pri analizi straničenja inače upotrebljava i naziv *radni skup stranica*. Taj radni skup čine one stranice koje su se upotrebljavale unutar jednog razdoblja mjerenoj od trenutka promatranja unatrag. To se razdoblje može nazvati vremenskim prozorom. U skladu s osnovnim načelom *LRU* strategije, radni skup stranica za odabrani vremenski prozor treba zadržati u okvirima fizičkog spremnika pri oduzimanju okvira procesu. (Podsetimo se da je u aproksimaciji *LRU* strategije koja koristi samo bit pristupa za razredbu stranica širina prozora jednaka jednoj periodi prekida od sata.) U operacijskom sustavu *Windows* riječ je o *radnom skupu okvira* koji su dodijeljeni pojedinom procesu, a ne o radnom skupu stranica.

PRIMJER 8.12.

Kako bismo uočili razliku između ta dva pojma koja se susreću u literaturi promotri- mo ponašanje procesa čiji logički adresni prostor ima šest stranica: 0, 1, 2, 3, 4, 5. Pretpostavimo da se vrijeme mjeri diskretno i da se tijekom izvođenja dretvi procesa u pojedinim periodama prekida od sata upotrebljavaju stranice logičkog adresnog prostora kako je prikazano slikom 8.26. Radni skup stranica označit ćemo s $RS(t_i, \Delta)$, gdje je t_i trenutak i Δ širina vremenskog prozora mjerenoj u periodama prekida od sata.

Za trenutke t_1 i t_2 možemo sa slike odrediti *radne skupove stranica*:

$$RS(t_1, 1) = \{0, 3, 5\}$$

$$RS(t_1, 2) = \{0, 1, 3, 5\}$$

$$RS(t_1, 3) = \{0, 1, 2, 3, 5\}$$

$$RS(t_1, 4) = \{0, 1, 2, 3, 4, 5\}$$

$$RS(t_1, 5) = \{0, 1, 2, 3, 4, 5\}$$

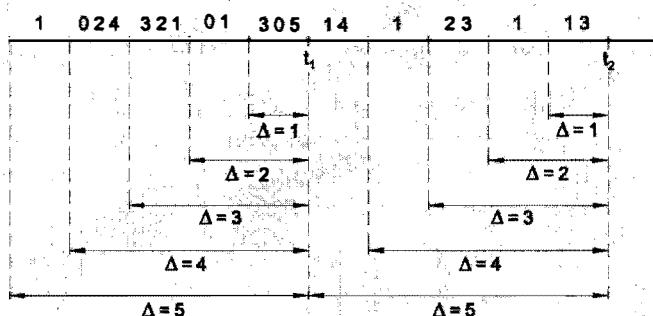
$$RS(t_2, 1) = \{1, 3\}$$

$$RS(t_2, 2) = \{1, 3\}$$

$$RS(t_2, 3) = \{1, 2, 3\}$$

$$RS(t_2, 4) = \{1, 2, 3\}$$

$$RS(t_2, 5) = \{1, 2, 3, 4\}$$



Slika 8.26. Ilustracija radnih skupova stranica

Radni skupovi okvira neka imaju redom 3, 4, 5 odnosno 6 okvira. Ako se stranice zahtijevaju redoslijedom predstavljenim na slici 8.26., onda će se sadržaji radnog skupa okvira mijenjati u skladu sa slikom 8.27., ako se izbacivanje stranica obavlja *FIFO* strategijom.

3 okvira	1	1	1	4	4	4	0	0	0	1	1	1
	—	0	0	0	3	3	3	5	5	5	2	2
	—	—	2	2	2	1	1	1	4	4	4	3

4 okvira	1	1	1	1	3	3	3	3	4	4	4	4
	—	0	0	0	0	1	1	1	1	2	2	2
	—	—	2	2	2	2	0	0	0	0	3	3
	—	—	—	4	4	4	4	5	5	5	5	1

5 okvira	1	1	1	1	1	5	5
	—	0	0	0	0	0	1
	—	—	2	2	2	2	2
	—	—	—	4	4	4	4
	—	—	—	—	3	3	3

6 okvira	1	1	1	1	1	1
	—	0	0	0	0	0
	—	—	2	2	2	2
	—	—	—	4	4	4
	—	—	—	—	3	3
	—	—	—	—	—	5

Slika 8.27. Stanja radnog skupa okvira pri izvođenju procesa sa slike 8.26.

8.5. Zaključne napomene o gospodarenju spremničkim prostorom

Već i ovaj pojednostavljeni prikaz koji smo proveli u ovom poglavlju može nas uvjeriti da postupci gospodarenja spremničkim prostorom u velikoj mjeri određuju ponašanje računalnog sustava. Može se ustvrditi da u velikoj većini računalnih sustava prevladava gospodarenje zasnovano na virtualnom spremniku.

Uvjerili smo se u sve prednosti koje nam pruža takav način ostvarenja spremničkog prostora. Procesni adresni prostor može biti mnogo veći od fizičkog spremnika i ne postoji problem fragmentacije radnog spremnika. Međutim, trajanje izvođenja procesa ovisi o ponašanju programa i broju promašivanja stranica koje će se dogoditi tijekom izvođenja. Štoviše, trajanje jednog procesa može bitno ovisiti o ponašanju drugih programa, jer operacijski sustav uravnotežuje njihova ponašanja dinamičkom preraspodjelom okvira. Zbog toga se u takvim sustavima ne može sa sigurnošću predvidjeti trenutke završavanja pojedinih procesa. Prema tome, za sustave u kojima se poslovi moraju obaviti uz strogo zadana vremenska ograničenja (za takve sustave kažemo da rade u stvarnom vremenu, engl. *real-time*) virtualni spremnik nije uvijek pravo rješenje.

Ponašanje sustava može se donekle poboljšati, barem za neke vremenski kritične dretve, ako se dobro upotrebljava. U današnjim operacijskim sustavima *API* sučelje dopušta da se iz programa utječe na neka svojstva straničenja. Tako se uobičajeno može dinamički rezervirati i oslobađati dijelove virtualnog adresnog prostora i neke se stranice u rezerviranom prostoru mogu (vidi odjeljak 8.4.1.) stvarno dodjeljivati ili oslobađati. Nadalje, posebnim *API* funkcijama može se zahtijevati da neke stranice logičkog adresnog prostora ostaju proizvoljno dugo u okvirima radnog spremnika (takve stranice operacijski sustav neće izbacivati iz njihovih okvira). Pomnijivim pisanjem programa može se s pomoću takve i sličnih *API* funkcija ciljano djelovati na odvijanje programa.

Virtualni spremnik, s druge strane, olakšava pisanje programa čija je veličina ograničena samo arhitektonskim svojstvima procesora. Gospodarenje spremničkim prostorom događa se automatski, i to uz pomoć sklopovla spremničkog međusklopa procesora. Međutim, onima koji pišu profesionalne programe preporučuje se proučavanje načina provođenja straničenja u njihovu konkretnom operacijskom sustavu, kao i izučavanje svih *API* funkcija povezanih s gospodarenjem spremničkim prostorom. Dobro poznavanje i pravilna uporaba tih funkcija može u velikoj mjeri unaprijediti ponašanje procesa.

PRIMJER 8.13.



Potrijebit ćemo ovu posljednju tvrdnju u tu svrhu često navođenim primjerom. Pretpostavimo da treba inicializirati s nulama matricu $A[1024][1024]$ čiji elementi su 32-bitovni cijeli brojevi pohranjeni u 4 bajta. Matrica se pohranjuje u spremnik po redcima pa će jedan redak zauzeti upravo jednu stranicu logičkog adresnog prostora veličine 4 KB, a cijela matrica zauzeti će 1024 stranice. Pretpostavimo nadalje da u fizičkom spremniku imamo na raspolaganju samo jedan okvir.

Programski odsječak:

```
za (I = 0; I < N; I++) {
    za (J = 0; J < N; J++) {
        C[I][J] = 0;
    }
}
```

izazvat će 1024 promašaja jer će se svaki put u okvir dobaviti jedan redak matrice i popuniti nulama.

Međutim, netko bi mogao načiniti samo jednu malu promjenu u programu i napisati odsječak ovako:

```
za (J = 0; J < N; J++) {
    za (I = 0; I < N; I++) {
        C[I][J] = 0;
    }
}
```



Ovaj odsječak popunjava redom stupce matrice, pri čemu će se za popunjavanje jednog stupca dogoditi 1024 promašaja. Ovaj će odsječak, prema tome, izazvati ukupno 1048 576 promašaja.

Ako nam kao vanjski spremnik služi disk sa svojstvima približno jednakim onima opisanim u primjerima 8.1. i 8.2., onda možemo ocijeniti da za jedno dobavljanje stranice u okvir fizičkog spremnika i njezino vraćanje na disk trebamo prosječno 30 ms.

Za obavljanje posla na prvi način program će utrošiti

$$1024 \cdot 30 \text{ ms} = 30\,720 \text{ ms} = 30.720 \text{ s.}$$

Ako se punjenje nulama obavlja na drugi način, trajanje posla će biti:

$$1\,048\,576 \cdot 30 \text{ ms} = 31\,457\,280 \text{ ms} = 31\,457.280 \text{ s} = 8 \text{ h } 44 \text{ m } 17.280 \text{ s.}$$

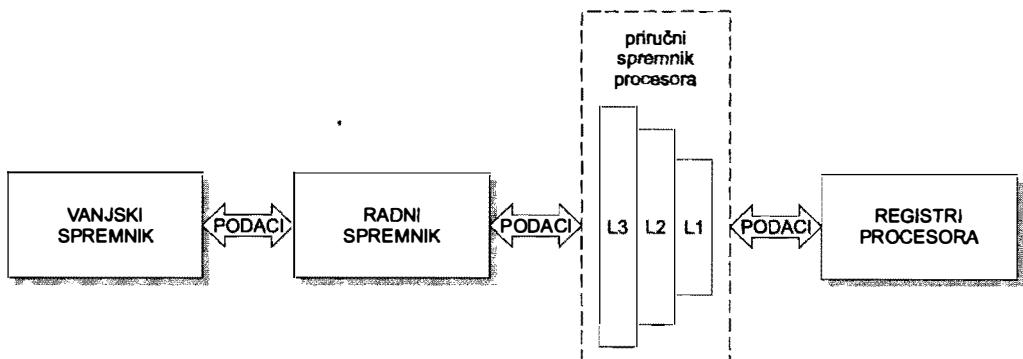
Primjer je u neku ruku pretjeran, ali zorno pokazuje da je vrijedno pomnjiće pisati programe za sustave s virtualnim spremnikom.

U ovom poglavlju posebno su razmotreni načini ubrzanja prijenosa podataka između tvrdog diska i radnog spremnika. Pritom ne treba zaboraviti da je cilj prenijeti podatke ne samo do radnog spremnika već do registara procesora. Kako bi se podaci iz radnog spremnika što je moguće brže dobavili u registre procesora, danas se u mnogim arhitekturama koristi priručni spremnik procesora kao što je opisano u odjeljku 2.1.6. Pristup podacima u priručnom spremniku mnogo je brži od pristupa podacima pohranjenima u radnom spremniku. U priručni spremnik procesora dinamički se pohranjuju podaci koji su pohranjeni u radnom spremniku. Mehanizmi prenošenja slični su onima koji se koriste u ostvarenju virtualnog spremnika straničenjem, samo što u njima djeluje par priručni spremnik – radni



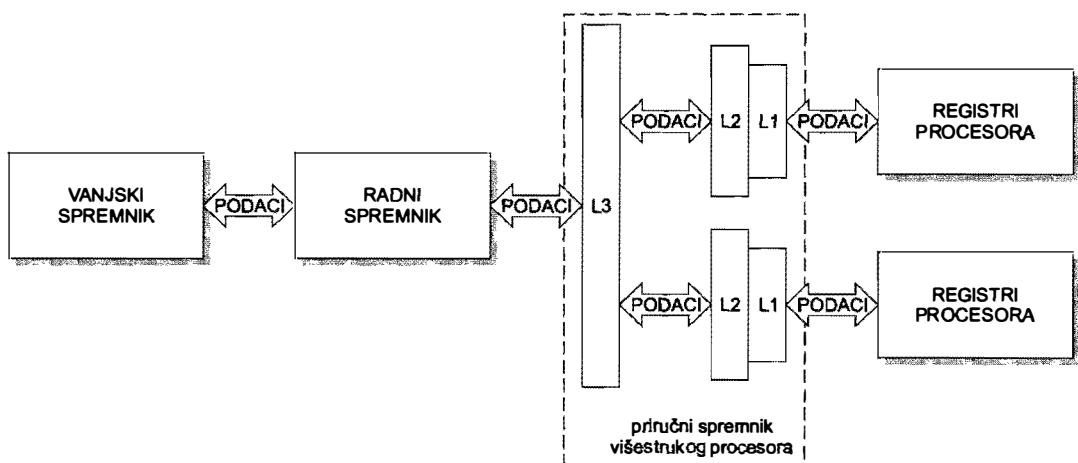
spremnik, a ne par radni spremnik – vanjski spremnik. Priručni spremnik procesora može biti ili jednolik (uniformni) ili može biti podijeljen na dio rezerviran za instrukcije i dio rezerviran za podatke.

Za korisnika i za operacijski sustav uglavnom je riječ o nevidljivom spremničkom sloju jer su prijenos i briga o koherenciji podataka u cijelosti osigurani na sklopovskoj razini. Priručni spremnik procesora obično se sastoji od nekoliko razina pa govorimo o višerazinskom priručnom spremniku. Na slici 8.27. prikazan je put koji podaci trebaju prevaliti od vanjskog spremnika do registara procesora ili obrnuto.



Slika 8.27. Hijerarhijska organizacija spremničkog prostora

Višerazinski priručni spremnik na slikama 8.27. i 8.28. sastoji se od triju razina. Uvijek se najprije provjerava postoje li traženi podaci u najnižoj, najmanjoj i najbržoj razini L1 priručnog spremnika. Zatim, ako nema traženih podataka u toj razini, traže se podaci u sljedećoj, višoj, većoj, ali i sporijoj razini L2 itd. Primjerice, kapaciteti razina L1, L2 i L3 mogu redom biti 64 KB, 1 MB i 16 MB.



Slika 8.28. Primjer arhitekture višerazinskog spremnika višestrukog procesora



Registri procesora mogu se promatrati kao priručni spremnik procesora najniže, nulte razine i najmanjeg kapaciteta.

U višestrukom procesoru svaka procesorska jedinka ima svoj L1 i L2 priručni spremnik, a priručni spremnik L3 dijele sve procesorske jedinice kako je to prikazano na slici 8.28.

Ustanovimo još jedanput na kraju ovoga poglavlja da procesi imaju potpuno razdvojene adresne prostore i jedan s drugim nemaju nikakve veze.

Podsjetimo se da dretve nesmetano dijele adresni prostor procesa s obzirom na to da je zamjena konteksta unutar jezgre operacijskog sustava dobro riješena.

Jednako tako, gospodarenje spremničkim prostorom mora osigurati da isti okviri fizičkog spremnika budu nesmetano korišteni od raznih procesa. U neku se ruku zamjena stranica može smatrati "velikom zamjenom konteksta" koja omogućuje da više procesa dijele isto spremničko sklopovlje. Kao što svaka dretva "vidi" samo svoj virtualni procesor, tako svaki proces "vidi" samo svoj virtualni spremnik.

U šestom smo poglavlju razmatrali kako dretve međusobno razmjenjuju podatke. Ustavili smo da one to mogu činiti jer djeluju u istom adresnom prostoru pa mogu nesmetano pristupati do istih spremničkih lokacija. Štoviše, zbog te velike slobode pristupanja morali smo izgraditi i koristiti i neke zaštitne i sinkronizacijske mehanizme. Postavlja se pitanje kako procesi mogu međusobno komunicirati. S obzirom na to da su im adresni prostori potpuno razdvojeni, komunikacija se mora obavljati na neki drugi organizirani način.

Podsjetimo se načina razmjene poruka između dretvi. Dretva proizvođač stavlja poruke u međuspremnik, smješten na nekoj adresi zajedničkog spremničkog prostora, a dretva potrošač čita poslanu joj poruku iz tog međuspremnika. Tako bi i *proces proizvođač* mogao oblikovati poruku i poslati je na neku izlaznu napravu. Neki *proces potrošač* morao bi znati da je poruka pohranjena u toj ulazno-izlaznoj napravi i da je on može pročitati. Ulazno-izlazna naprava morala bi imati mogućnost pohranjivanja više poruka jer proces proizvođač može u nekim razdobljima intenzivno proizvoditi poruke, koje proces potrošač ne stigne istodobno prihvatići.

Pokazalo se da je jedan od mogućih odabira za ulazno-izlaznu napravu disk, s tim da je međuspremnik na disku organiziran kao *datoteka* (engl. *file*)⁶. Zbog toga ćemo odgoditi razmatranje komunikacije između procesa i razmotriti najprije način ostvarivanja jednog važnog dijela operacijskog sustava koji nazivamo datotečnim sustavom (engl. *file system*). Pokazat će se da datoteke imaju i mnoge druge važne uloge u organizaciji rada računalnih sustava, kao i to da postoje i druge izvedenice ove osnovne zamisli komunikacije procesa posredstvom datoteka.

⁶ Komunikacija između procesa ostvaruje se i tako da se međuspremnik poruka smjesti u adresni prostor operacijskog sustava (koji je izvan adresnih prostora komunicirajućih procesa). Taj se međuspremnik može smatrati svojevrsnom datotekom u koju jedan proces može pisati, a drugi čitati. Ovaj mehanizam komunikacije razmotrit ćemo nakon što upoznamo datoteke.

**PITANJA ZA PROVJERUZNANJA 8**

1. Gdje se generiraju adrese unutar procesora?
2. Kako je podijeljen procesni adresni prostor?
3. Opisati organizaciju smještaja sadržaja na magnetskom disku (cilindri, staze, sektori).
4. Čime je određena jedinstvena adresa svakog sektora na disku?
5. Koliko iznosi ukupno trajanje prijenosa podataka tvrdi disk – radni spremnik?
6. Od čega se sastoji trajanje traženja staze (*seek time*)?
7. Koliko iznosi prosječno vrijeme traženja u odnosu na vrijeme koje je potrebno za prelaz preko svih staza?
8. Zbog čega nastaje rotacijsko kašnjenje i koliko ono iznosi?
9. Čime je određena brzina prijenosa podataka s diska u spremnik diskovne upravljačke jedinice?
10. Neka je trajanje traženja staze T_s nekog diska s 2000 staza opisano sa sljedeće tri formule, gdje je D udaljenost između trenutačnog položaja glave i tražene staze:
 - $T_s = 1.5D$ ms za $D \leq 4$,
 - $T_s = 4.0 + 0.5 \cdot D^{1/2}$ ms za $4 < D \leq 400$,
 - $T_s = 10.0 + 0.01 \cdot D$ ms za $D > 400$.
 Koliko iznosi prosječno vrijeme traženja staze?
11. Navesti sadržaj procesnog informacijskog bloka.
12. Opisati postupke statičkog i dinamičkog dodjeljivanja spremnika.
13. Navesti vrste fragmentacije prilikom statičkog dodjeljivanja spremnika.
14. Problem fragmentacije prilikom dinamičkog dodjeljivanja spremnika ne može se izbjegći, ali se može ublažiti. Kako?
15. Unatoč tomu što se problem fragmentacije prilikom dinamičkog dodjeljivanja spremnika može ublažiti, fragmentacija može postati prevelika. Što tada treba učiniti?
16. Navesti i dokazati Knuthovo pedesetpostotno pravilo.

17. Kako treba podijeliti program (koji u cijelosti ne stane u radni spremnik) u prekllopnom načinu uporabe radnog spremnika?
18. Kako je podijeljen logički, a kako fizički adresni prostor u sustavu sa straničenjem?
19. O čemu ovisi veličina fizičkog i logičkog adresnog prostora?
20. Mogu li stranice logičkog adresnog prostora biti smještene u okvire fizičkog spremnika proizvoljnim redoslijedom?
21. Čemu služi tablica prevodenja? Od kojih se elemenata sastoji?
22. U čemu se razlikuju prekidi izazvani zbog promašaja stranice kod straničenja na zahtjev od ostalih vrsta prekida?
23. Opisati sklopošku potporu za ostvarenje straničenja u Intel x86 arhitekturi. Što sadrži i čemu služi TLB (*Translation Lookaside Buffer*)? Opisati tablicu prevodenja. Gdje se ona nalazi? Koliko je puta potrebno pristupiti radnom spremniku ako se stranica ne nalazi u TLB međuspremniku, a koliko ako se nalazi u TLB međuspremniku?
24. Koju informaciju nosi bit čistoće? Gdje se on nalazi?
25. Čemu služi posmačni registar povijesti?
26. Opisati sljedeće strategije za izbacivanje stranica: FIFO, LRU, OPT te satni algoritam.
27. Opisati strategiju izbacivanja stranica u Intel x86 arhitekturi kada se u obzir uzimaju dvije zastavice za označavanje stanja stranice.
28. U kojim se stanjima mogu nalaziti pojedini okviri tijekom rada?

9.

Datotečni podsustav

9.1. Uloga datoteka u računalnim sustavima

Svi sadržaji koji se u računalu trebaju trajno čuvati pohranjuju se u datoteke. Praktički svi poslovi koje izvodimo računalom oslonjeni su na uporabu i pretvorbu datoteka (engl. *file*). Zbog toga se korisnik već u svom prvom pristupu računalu susreće s datotekama. Taj se pristup obavlja posredstvom dijela operacijskog sustava koji nazivamo *datotečnim sustavom* (engl. *file system*).

S obzirom na to da se sadržaji radnog spremnika gube pri svakom isključivanju računala, datoteke se čuvaju na vanjskim spremnicima. Najčešće korišteni vanjski spremnici su *diskovi*. Na svaki se disk, osim samih datoteka, pohranjuju i *tablice* ili *direktoriji* (engl. *directories*) u koje se smještaju imena datoteka s kazaljkama koje omogućuju pristup do datoteka. Tablice se uobičajeno organiziraju po odjeljcima i pododjeljcima (engl. *sub-directory*) kako bi se olakšalo pronalaženje datoteka. Takve se tablice mogu slikovno prikazati kao razgranato stablo u čijoj se posljednjoj razini nalaze nazivi datoteka.

Za svaku se datoteku svi važni podaci, tzv. *atributi* datoteke (engl. *attributes*) smještaju u jedan zapis koji nazivamo *opisnikom datoteke* (engl. *file descriptor*). Podatke iz tog zapisa koristi operacijski sustav, a ne korisnički programi. Takav smještaj opisnika omogućuje da se ista datoteka može nazvati različitim imenima i da se do nje može pristupiti iz različitih ograna direktorija.

Datoteke uz svoj naziv uobičajeno dobivaju sufiks (odvojen od imena točkom) kojim se opisuje tip datoteke. Datoteke mogu sadržavati primjerice:

- programe u strojnom obliku pripremljenom za izvođenje (uobičajeno sa sufiksom *exe*);
- izvorne programe napisane u višem programskom jeziku (uobičajeno sa sufiksom koji ukazuje na jezik: *c*, *cpp*, *pas* i sl.);
- datoteke koje sadrže tekstove i dokumente (uobičajeno sa sufiksima *txt*, *asc* ili *doc*).

Datotečni sustav omogućuje provođenje različitih operacija nad datotekama. Za određeni tip datoteka dopuštene su samo određene operacije; tako se, primjerice, datotekom koja sadrži program pripremljen za izvođenje može obaviti samo operacija pokretanja programa.

Znakovna se datoteka ne može pokrenuti kao program (čak ako je u datoteci pohranjen program napisan nekim višim programskim jezikom!).

Za pojedine tipove datoteka može se odrediti dopuštene načine njihove uporabe (samo izvođenje, samo čitanje, samo pisanje). Općenito, pristup do određenih datoteka može se sigurnosno zaštititi, tj. dopustiti ga samo za to ovlaštenim korisnicima.

Datoteke, osim trajnog pohranjivanja nekih sadržaja, imaju još jednu važnu svrhu u računalnim sustavima: one služe i za razmjenu informacija između programa. Datoteka može biti stvorena i popunjena od strane jednog programa i nakon toga njezin sadržaj može dohvatiti drugi program.

Prema tome, može se reći da datoteke u računalnim sustavima imaju dvojaku ulogu:

- one služe za trajno pohranjivanje svih oblika sadržaja i
- s pomoću njih se može obavljati razmjena informacija između različitih programa.

9.2. Struktura datoteka

Datotečni sustav u načelu se ne brine o sadržaju datoteka. Ipak je, zbog potpunijeg razumijevanja datotečnih operacija, prikladno razmotriti neke od mogućih načina strukturiranja datoteka.

Binarna datoteka

Najjednostavniji oblik datoteke koristi se za pohranjivanje strojnih programa pripravnih za izvođenje. Takva se datoteka može promatrati kao niz bitova (pohranjenih u bajtove) koji se kao cjelina prenose iz jednog spremnika u drugi, i to bez ikakvih promjena.

Nestrukturirana datoteka

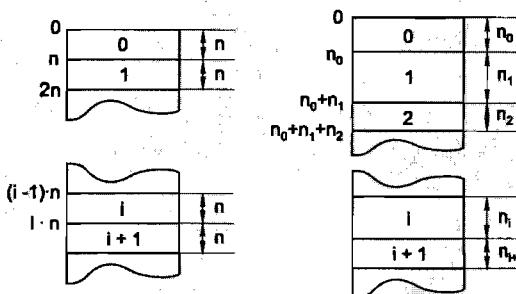
U današnjim se operacijskim sustavima pretežito upotrebljavaju datoteke koje se sastoje od niza bajtova i nemaju nikakvu strukturu. Datotečni sustav omogućuje pristup do pojedinog bajta s pomoću *datotečne kazaljke* (engl. *file pointer*). Pri otvaranju datoteke datotečna kazaljka postavlja se na početak datoteke. Datotečni sustav svojim funkcijama omogućuje čitanje niza bajtova iz datoteke i pisanje niza bajtova u datoteku. Datotečna se kazaljka pri čitanju i pisanju pomiče za broj pročitanih, odnosno upisanih bajtova. Posebnom se oznakom obilježava kraj datoteke (engl. *End of File*). Pokušaj čitanja iza kraja datoteke izaziva pogrešku, dok pisanje u datoteku može izazvati produljenje datoteke i pomicanje označke novog kraja datoteke.

U sučeljima predviđenim za pojedine programske jezike nalaze se *API* funkcije koje omogućuju određeno formatiranje podataka. Čitanje i pisanje osnovnih tipova podataka u skladu s deklariranim formatom uzrokuje dohvat, odnosno pohranjivanje određenog broja bajtova u datoteku i pomicanje datotečne kazaljke usklađeno s brojem dohvaćenih ili pohranjenih bajtova. Nadalje, postoje funkcije koje eksplicitno djeluju na pomicanje datotečne kazaljke. Datoteka se u neku ruku može promatrati kao linearni adresni prostor čije adresiranje obavlja datotečna kazaljka.

Potpore formatiranju postoji u *ASCII* skupu znakova u kojemu postoje posebni znakovi za odvajanje zapisa (primjerice, znakovi *CR* od engl. *Carriage Return* i *LF* od engl. *Line Feed*), kao i znak za kraj retka (*EOL* od engl. *End of Line*). Ti se znakovi mogu koristiti za svojevrsno strukturiranje datoteka pri čemu se datoteke tada uobičajeno nazivaju znakovnim datotekama (engl. *text file*). Velika je prednost nestrukturiranih datoteka što one omogućuju jednostavnu razmjenu sadržaja između različitih programa, i to čak i onih koji se izvode u potpuno različitim programskim okruženjima i uz potporu različitih operacijskih sustava.

PRIMJER 9.1.

Na razini primjenskog programa potpuno nestrukturirani bajtovi mogu se organizirati u različite strukture. Tako se datoteka može strukturirati u podnizove bajtova stalne ili promjenljive duljine (slika 9.1.).



Slika 9.1. Strukturiranje nestrukturirane datoteke

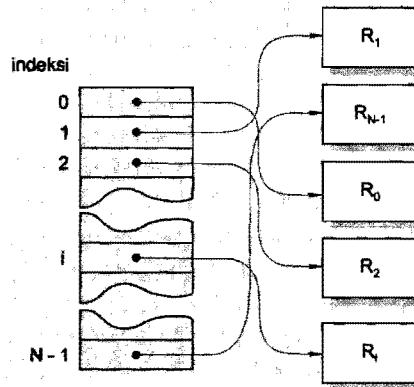
Unutar korisničkog programa moraju se nalaziti parametri (veličina n za datoteku sa stalnim duljinama zapisa, tablica veličina n_i za datoteke s promjenljivim duljinama zapisa) koji određuju logičku strukturu datoteke. Uporabom *API* funkcije za postavljanje datotečne kazaljke na izračunati početni bajt nekog logičkog zapisa izvorna se nestrukturirana datoteka može pretvoriti u datoteku koja omogućuje neposredni pristup do svakog zapisa.

Oponašanje indeksne strukture

U nekim ranijim operacijskim sustavima datotečni je sustav podržavao i operacije s tzv. indeksnim datotekama. Danas se uporaba indeksno organiziranih struktura zapisa prepusta sustavima *baza podataka*. Programski sustavi za podržavanje baza podataka omogućuju pohranjivanje i dohvaćanje zapisa uskladeno s vrijednostima pojedinih polja tih zapisa. Međutim, indeksna se struktura može organizirati unutar korisničkog programa uporabom tablica za pohranjivanje parametara strukture te izgradnjom funkcija za pristupanje do pojedinih elemenata strukture uporabom postojećih *API* funkcija za pristupanje nestrukturiranoj datoteci.

PRIMJER 9.2.

Prepostavimo da se datoteka sastoji od N zapisa R_i ($0 \leq i \leq N - 1$) jednake duljine n , koje želimo dohvaćati s pomoći indeksa. U izvorno nestrukturiranoj datoteci može se oponašati indeksni pristup tako da se redne brojeve početnih bajtova s vremenom smjesti u posebni poredak (smješten također unutar datoteke, slika 9.2.).



Slika 9.2. Indeksno-sekvencijska organizacija datoteke

Zapisi mogu u datoteci biti razmješteni proizvoljnim redoslijedom. Redoslijed pristupa određen je razmještajem kazaljki u poredak. Ovakva se organizacija može nazvati indeksno-sekvencijskom. Uporabom postojećih API funkcija može se pripremiti funkcije za organizaciju takve datoteke.

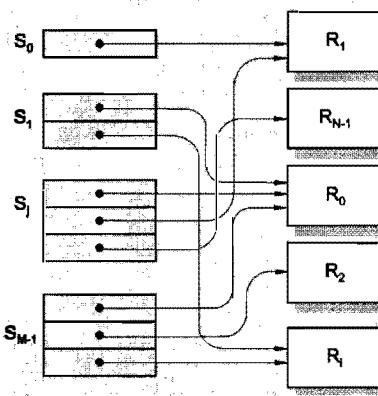
PRIMJER 9.3.

Prepostavimo da N zapisa iz prethodnog primjera mogu biti razvrstani u skladu s M binarnih svojstava S_j ($0 \leq j \leq M - 1$). Pojedini zapisi R_i ili imaju ili nemaju dano svojstvo, što se može izraziti tablično (slika 9.3.).

svojstvo	zapis					
	R_0	R_1	R_2	R_i	R_{N-1}	
S_0	0	1	0		0	0
S_1	1	0	0		1	0
S_j	1	1	0		0	1
S_{M-1}	1	0	1		1	0

Slika 9.3. Tablični prikaz svojstava zapisa datoteke

Zapisi se mogu dohvaćati po svojstvima ako se za svako svojstvo izgradi poredak kazaljki onih zapisa koje to svojstvo zadovoljavaju (slika 9.4.).



Slika 9.4. Indeksna organizacija zapisa po zadanim svojstvima

9.3. Smještanje datoteka na disku

Kao što je već spomenuto u točki 9.1., svaka je datoteka opisana u datotečnom sustavu svojim opisnikom. Opisnik sadrži sve atribute datoteke. Tipični sadržaji opisnika jesu:

- naziv datoteke,
- tip datoteke,
- lozinka,
- ime vlasnika datoteke,
- prava pristupa,
- vrijeme stvaranja datoteke,
- vrijeme zadnje uporabe datoteke,
- ime posljednjeg korisnika i slično,
- opis smještaja datoteke na disku.

Naziv svake datoteke mora biti jedinstven kako bi je se moglo nedvosmisleno prepoznati. Vlasnik datoteke koji poznaje lozinku može mijenjati neke atribute datoteke, primjerice, pravo pristupa do datoteke. Datoteke mogu upotrebljavati i drugi korisnici, ali samo na način kako je to propisao vlasnik. Pristup do datoteke može biti sigurnosno zaštićen pa se u opisniku mogu naći i elementi sigurnosne zaštite, odnosno kazaljke na zaštitni podopisnik.

Jedan od najvažnijih sadržaja opisnika datoteke jest opis njezina smještaja u vanjskom spremniku. Najprikladniji vanjski spremnik za smještanje datoteka je magnetski disk. Osnovna svojstva diskova opisana su u točki 8.2.

Opis diskovnog prostora

Podsjetimo se da se na disku ne mogu adresirati pojedinačni bajtovi nego sektori. Prema tome, datoteke je smisleno smještati na disk tako da ih se podijeli na *blokove bajtova* koji su po veličini jednaki veličini sektora, odnosno cjelobrojnom višekratniku te veličine (pri čemu je višekratnik najčešće potencija broja dva) kada se za smještanje blokova odabere nakupina (engl. *cluster*) od više susjednih sektora. Primjerice, uz veličinu sektora od 512 bajtova blokovi mogu imati duljinu od 1 KB, 2 KB, 4 KB, 8 KB itd., a nakupine sektora čine 2, 4, 8, odnosno 16 uzastopnih sektora.

Jedan fizički disk može se upotrebljavati ili kao jedan jedinstveni adresni prostor ili se on može podijeliti na logički razdvojene adresne potprostore, tzv. sveske (engl. *volume*). Svaki svezak ima svoj direktorij, odnosno *datotečnu tablicu* u kojoj su sadržani opisnici svih datoteka smještenih u tom svesku. Osim toga, sastavni je dio svake datotečne tablice i opis slobodnog prostora sveska. Evidencija o zauzetim i slobodnim sektorima, odnosno nakupinama sektora može se najsazeti obaviti *bitovnim prikazom*.

U nizu bitova svakom sektoru, odnosno nakupini sektora pripada jedan bit čija je vrijednost jednak 1 kada je sektor, odnosno nakupina zauzeta ili jednak 0 u suprotnom slučaju. Eventualni neispravni sektori mogu se pri početnom ispitivanju diska označiti s 1 i tako ih isključiti iz daljne uporabe, čime se omogućuje uporaba i djelomično neispravnog diska. Sektori označeni kao da su zauzeti neće se nikada dodijeliti niti jednoj datoteci i, prema tome, ne mogu narušiti funkcionalnost diska.

Slobodni se prostor sveska može opisati i svojevrsnim dinamičkim strukturama u kojima se neposredno može odrediti adresa slobodnog sektora, odnosno nakupine sektora (iz bitovnog se prikaza adrese dobivaju brojenjem bitova).

PRIMJER 9.4.

 Pretpostavimo da je sljedećim bitovnim prikazom označena zauzetost početnih sektora nekog sveska:

sektor 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...
zauzetost	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	1	1	0	0	1	...

Slika 9.5: Bitovni prikaz zauzeća sektora diska

Jednaku informaciju sadrži i lista u kojoj svaki zapis sadrži početnu adresu skupine slobodnih sektora te broj sektora u toj skupini kako to prikazuje sljedeća slika:



Slika 9.6. Prikaz slobodnog prostora u obliku liste

Način smještanja datoteka na disku

Na temelju opisa dinamičkih svojstava diskova u odjeljku 8.2. može se zaključiti da bi sa stanovišta brzine prijenosa sadržaja datoteke s diska u radni spremnik i obrnuto bilo prikladno cjelovite datoteke smjestiti kompaktno u uzastopne sektore na disku. Međutim, na temelju razmatranja uporabe radnog spremnika (vidjeti primjer 8.3.) možemo zaključiti da bi takav način smještanja datoteke izazvao fragmentaciju prostora jednog sveska. Naime, taj je način smještanja datoteke na disk usporediv s dinamičkim dodjeljivanjem radnog spremnika s jednim segmentom. Zbog toga se kao rješenje nameće način smještanja datoteke koji podsjeća na dodjeljivanje radnog spremnika straničenjem.

Datoteka se može promatrati kao "logički adresni prostor" koji započinje nultom adresom i završava adresom koja je za jedan manja od duljine datoteke. Znamo da unutar tog adresnog prostora "adresu" određuje datotečna kazaljka. Datoteka se može podijeliti na *blokove* (procesni logički adresni prostor dijeli se na stranice), čija je veličina jednaka veličini sektora, odnosno nakupine sektora diska. Blokovi se razmještaju u nakupine sektora na disku na jednak način kao što se stranice smještaju u okvire radnog spremnika.

U djelu opisnika datoteke koji sadrži opis razmještaja datoteke na disku mora se nalaziti tablica kazaljki na nakupine sektora koja podsjeća na indeksno-sekvencijsku organizaciju opisanu u primjeru 9.2. pri čemu bi zapisi bili po veličini jednakim nakupinama sektora. Kazaljke se redom dohvataju s pomoću indeksa.

Fragmentacija prostora sveska određena je veličinom nakupine sektora. Naime, pri podjeli datoteke na blokove posljednji blok bit će samo u iznimnim slučajevima popunjeno. U najgorem slučaju on može sadržavati samo jedan bajt i tada je posljednja nakupina sektora praktički prazna. Očekivati je da će u prosjeku datoteke popuniti polovinu posljednjeg bloka. Fragmentacija diskovnog prostora ovisi, dakle, o veličini nakupine sektora.

PRIMJER 9.5.

U tradicionalnim datotečnim sustavima operacijskog sustava **UNIX** opisnik datoteke naziva se *I-node* (indeksni čvor) jer sadrži indeksno dohvatljive kazaljke na nakupine sektora. Uz pretpostavku da je sektor velik 1024 B te da kazaljke imaju 32 bita, u jedan sektor može se smjestiti 256 kazaljki. U opisniku se nalazi 13 kazaljki, i to:

- 10 neposrednih kazaljki,
- 1 jednostruko indirektna kazaljka,
- 1 dvostruko indirektna kazaljka i
- 1 trostruko indirektna kazaljka.



Pristup do datoteke obavlja se na sljedeći način:

- prvih deset sektora dohvata se neposrednim kazaljkama, čime se može dohvatiti sve sektore datoteke koje su manje od 10 KB;
- jedanaesta kazaljka pokazuje na sektor u kojem se nalazi sljedećih 256 kazaljki (dovoljno za datoteku veličine do 266 KB);
- dvanaesta kazaljka pokazuje na sektor u kojem su kazaljke na 256 sektora svaki s 256 kazaljki – dovoljno za datoteke manje od $(266 + 256 \cdot 256)$ KB, što je približno 64 MB;

- trinaesta kazaljka pokazuje na trorazinsko stablo kazaljki i omogućuje pristup do ukupno:

$$10 + 256 + 256 \cdot 256 + 256 \cdot 256 \cdot 256$$

ili približno"

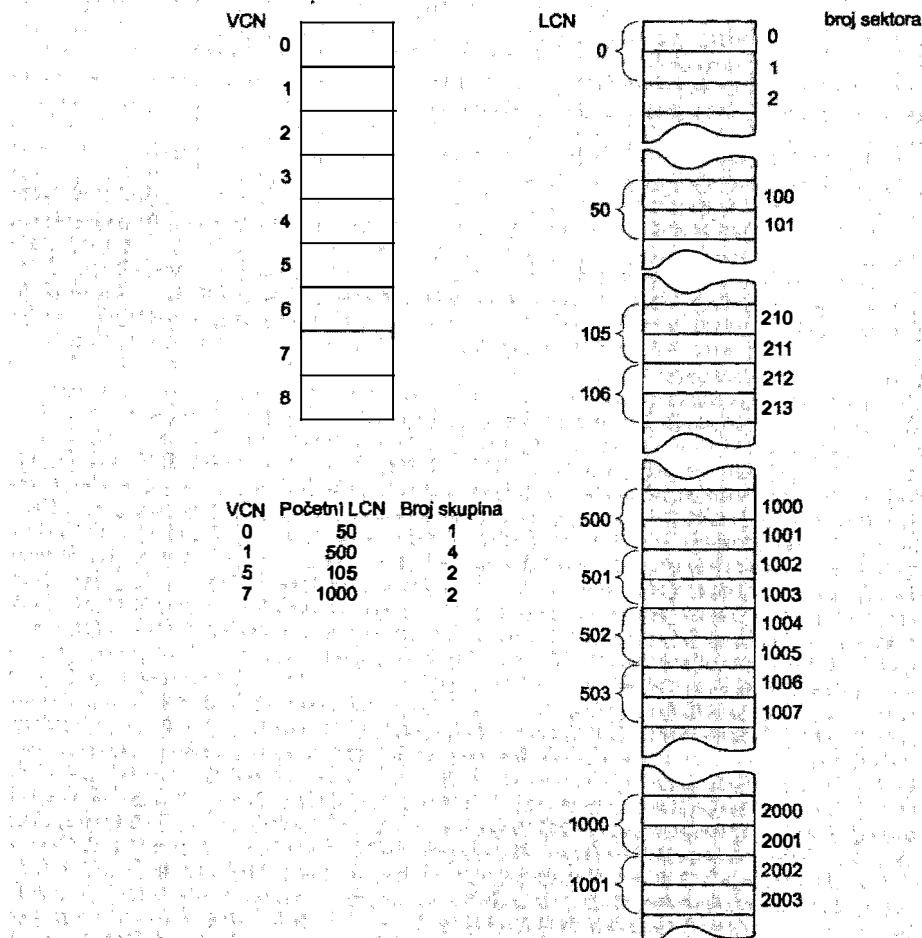
$$2^{24} \text{ sektora} = 16 \text{ M sektora},$$

odnosno datoteke veličine 16 GB.

PRIMJER 9.6.



Datotečni sustav *NTFS* upotrebljava se od operacijskog sustava *Windows NT* na dalje. Diskovni prostor dodjeljuje se po skupinama sektora (engl. *cluster*). Skupine uobičajeno imaju 1, 2, 4 ili 8 sektora. Veličine skupina određuju se u skladu s veličinom diska. Uobičajeno je za veće diskove koji su danas pretežito u uporabi da veličina skupine bude 4096 B (uz veličinu sektora od 512 B skupina se sastoji od 8 sektora).



Slika 9.7. Opis smještaja datoteke u NTFS datotečnom sustavu

NTFS se referira na disk uporabom logičkog broja skupine (engl. *Logical Cluster Number – LCN*). *LCN* je jednostavno rečeno redni broj skupine od početka do kraja diskovnog prostora predviđenog za smještaj datoteka. NTFS sadrži datoteku koja se naziva glavnom tablicom datoteka (engl. *Master File Table – MFT*). Svaka datoteka (uključujući i *MFT* datoteku) ima u toj datoteci jedan zapis koji je opisnik datoteke. Jedan *MFT* zapis ima veličinu jedne skupine sektora (primjerice 4 KB).

Male datoteke mogu se smjestiti unutar *MFT* zapisa, dok za veće datoteke zapis sadrži indeks skupina sektora i, ako je potrebno, i dodatno proširenje zapisa. Datoteka se dijeli na dijelove koji su jednak veliki kao nakupine sektora i nazivaju se *virtualnim skupinama*. Datoteka je dakle podijeljena na "stranice" koje su jednake veličini "okvira". Redni broj "stranice" naziva se *Virtual Cluster Number – VCN*.

Za smještaj datoteka pronalazi se što više uzastopnih skupina sektora i dodjeljuje datoteci. U najgorem slučaju, ako uzastopnih sektora nema, dodjeljuju se i pojedinačne skupine (slika 9.7.).

9.4. Načela ostvarenja datotečnih funkcija

Datotečni sustavi s pomoću zbirke *API* funkcija omogućuju obavljanje tipičnih operacija s datotekama. Tipične operacije jesu:

- stvoriti i uništiti;
- otvoriti i zatvoriti;
- čitati i pisati.

Za razumijevanje ostvarenja datotečnih funkcija treba se podsjetiti na sve što je do sada rečeno o datotekama:

- svi podaci o datotekama smješteni su u opisniku datoteke;
- opisnici su smješteni u datotečnoj tablici;
- datoteke su smještene u sektore ili nakupine sektora na disku.

Stvaranje datoteke

Stvaranje datoteke podrazumijeva uvođenje novog opisnika u datotečnu tablicu i oblikovanje svih potrebnih podataka za dohvrat datoteke. Skica funkcije za stvaranje datoteke prikazana je u nastavku:

```
funkcija Stvoriti_datoteku(Novo_ime, atributi) {
    pregledati datotečnu tablicu;
    ako je (Novo_ime već u tablici) {
        dojaviti pogrešku;
    }
}
```



```

    inače {
        uvesti novi opisnik u tablicu;
        dodijeliti početni prostor datoteke;
        zapisati sve podatke u opisnik;
    }
}

```

Otvaranje datoteke

Prije uporabe datoteka se mora otvoriti. Pri otvaranju se imenu datoteke pridružuje identifikator ID (ručica, engl. *handle*) s pomoću kojeg će se iz adresnog prostora procesa pristupati otvorenoj datoteci. Pri otvaranju treba:

- u adresni prostor operacijskog sustava prenijeti s diska aktivnu kopiju opisnika datoteke;
- u adresnom prostoru operacijskog sustava rezervirati međuspremnik za smještaj dijelova datoteka;
- uspostaviti vezu između adresnog prostora procesa i otvorene datoteke s pomoću identifikatora.

Samو se otvorene datoteke mogu koristiti. Pri otvaranju se datotečna kazaljka postavlja na početak. U nastavku je prikazana skica funkcije za otvaranje datoteke:



```

funkcija Otvoriti_datoteku(Ime,Način_pristupa, ID) {
    pregledati datotečnu tablicu;
    ako je (Ime nađeno && Način_pristupa == Pravo_pristupa) {
        stvoriti aktivnu kopiju opisnika u adresnom prostoru operacijskog
        sustava;
        rezervirati međuspremnik u adresnom prostoru operacijskog sustava;
        uspostaviti vezu između aktivnog opisnika i identifikatora;
    }
    inače {
        dojaviti pogrešku;
    }
}

```

Čitanje datoteke

Biblioteke pojedinih jezičnih procesora sadrže funkcije koje obavljaju formatizirano čitanje. Te se funkcije izgrađuju s pomoću ovakve osnovne funkcije čitanja:



```

funkcija Čitati_datoteku(ID, Logička_adresa, Broj_bajtova) {
    s pomoću ID pristupiti do aktivnog opisnika;
    provjeriti je li datoteka otvorena za čitanje i postoji li traženi broj
    bajtova;
}

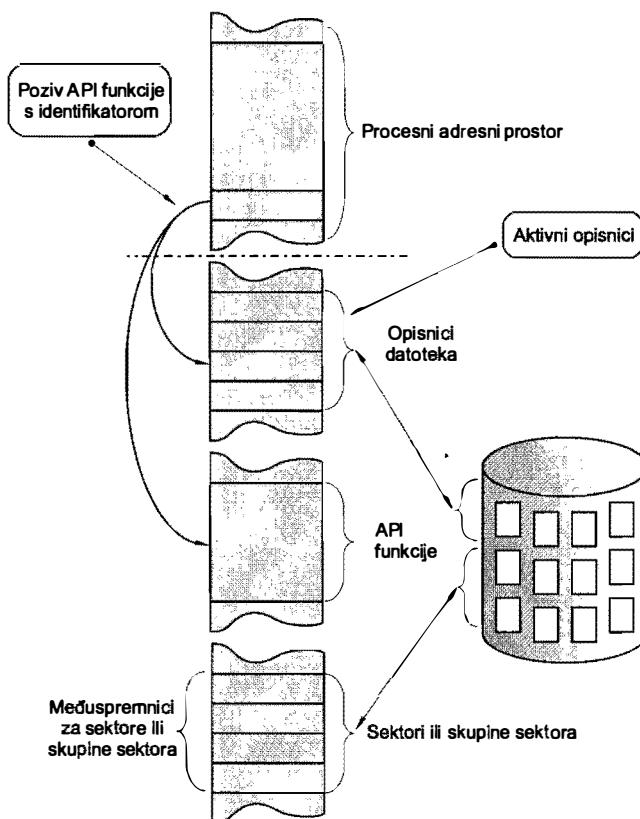
```

```

ako je (sve u redu) {
    na osnovi trenutačne vrijednosti datotečne kazaljke i Broj_bajtova u
    međuspremnik prenijeti potrebne sektore;
    prepisati u adresni prostor procesa s početnom adresom Logička_adresa
    traženi broj bajtova;
    povećati sadržaj datotečne kazaljke za preneseni broj bajtova;
}
inače {
    dojaviti pogrešku;
}
}

```

Na sličan su način izgrađene i ostale funkcije datotečnog sustava. Treba naglasiti da se pri pisanju u datoteku pišu u kopiju skupine sektora u radnom spremniku. Postoji opasnost da se pri nekim nepredviđenim prekidima rada promijenjeni sadržaji iz radnog spremnika ne prepisuju na disk. Time se u transakcijskim postupcima može pojaviti nekonzistentnost. Izbjegavanje nekonzistentnosti relativno je složeno i zahtijeva posebno razmatranje.



Slika 9.8. Rad s datotekama uz pomoć zbirke API funkcija

9.5. Metode posluživanja zahtjeva za pristup datotekama

Svaki pristup sadržaju datoteke odvija se posredovanjem operacijskog sustava, putem zbirke API funkcija kako je prikazano u prethodnom odjeljku ili putem sučelja operacijskog sustava. U višezadačnom sustavu moguće je nakupljanje zahtjeva za pristup datotečnom sustavu od strane više procesa budući da se u jednom trenutku može posluživati samo jedan zahtjev. Svi ostali neposluženi zahtjevi moraju se stoga staviti u red čekanja.

Operacijski sustav može utjecati na prosječno trajanje čekanja tako da nakupljene zahtjeve slaže u red po zadanim pravilima, tj. nekom metodom *određivanja redoslijeda* posluživanja. Pritom se mora voditi računa o fizikalnim svojstvima magnetskih diskova (opisanima u točki 8.2.) zbog činjenice da trajanje posluživanja ovisi o smještaju traženih podataka na disku te trenutačnoj poziciji glave za čitanje.

Posluživanje redom prispijeća

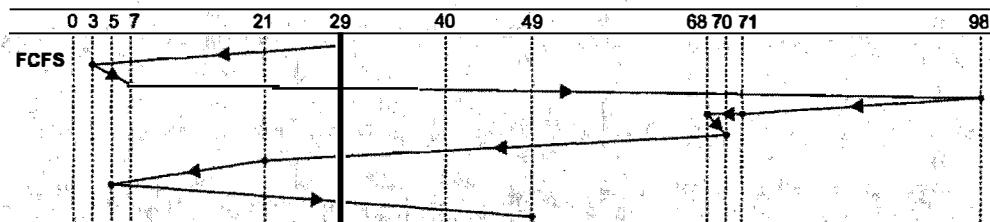
Najjednostavnija metoda rasporedivanja jest posluživanje redom prispijeća (engl. *first-come, first-served, FCFS*) i u njoj se zahtjevi poslužuju onim redom kojim su dolazili. Iako je na prvi pogled najpravednija, ova metoda često uzrokuje veliko prosječno trajanje čekanja. Naime, glava za čitanje mora prelaziti velik broj staza ako su zahtjevi povezani s podacima na udaljenim stazama na disku.

PRIMJER 9.7.

Pretpostavimo da su različiti zahtjevi svedeni samo na redni broj staze na kojoj se nalaze traženi podaci te da su pristigli sljedećim redoslijedom:

3, 7, 40, 98, 71, 68, 70, 21, 5, 49.

Pretpostavimo također da se glava za čitanje trenutno nalazi na stazi 29. Ako koristimo posluživanje redom prispijeća, kretanje glave za čitanje prilikom posluživanja ovih zahtjeva prikazano je slikom 9.9.:

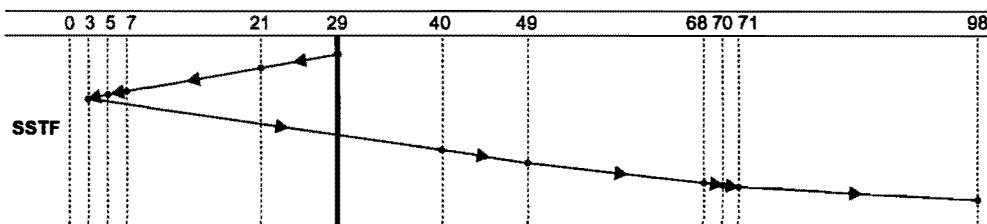


Slika 9.9. Posluživanje redom prispijeća

Posluživanje s najkraćim vremenom premještanja

Kako se velika količina vremena troši samo na pomicanje glave za čitanje, logično je poslužiti one zahtjeve koji su bliže trenutačnoj poziciji glave prije posluživanja udaljenih zahtjeva. Takva strategija vodi nas do metode posluživanja s najkraćim vremenom premještanja glave (engl. *shortest-seek-time-first*, *SSTF*). Svaki put kada je potrebno poslužiti neki od pristiglih zahtjeva, uvijek se odabire onaj koji je najbliži trenutačnoj poziciji glave.

Tijek posluživanja zahtjeva iz primjera 9.7. ovom metodom prikazuje slika 9.10.:



Slika 9.10. Posluživanje s najkraćim vremenom premještanja

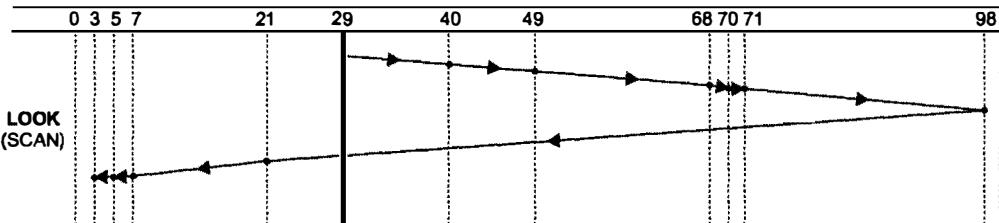
Primjena ove metode može izazvati *izgladnjivanje* nekih zahtjeva. U stvarnom radu sustava, naime, novi zahtjevi mogu dolaziti u red prilikom posluživanja prethodno zaprimljenih zahtjeva. Novoprdošli zahtjev može biti poslužen i prije zahtjeva koji dulje vrijeme čeka u redu ako je broj staze vezan uz novi zahtjev bliži trenutačnoj poziciji glave. Primjerice, ako je tijekom posluživanja zahtjeva na stazi 71 u našem primjeru došao novi zahtjev za stazom 65, zahtjev za stazom 98 privremeno će biti odgođen. U teoriji, stalan niz zahtjeva na bližim stazama (oko 65) može neodređeno dugo odgadati posluživanje udaljenog zahtjeva (na stazi 98). Naravno, ova je pojava statistički malo vjerojatna, ali je moguća.

Posluživanje pregledavanjem

Uvažavanje dinamičke prirode procesa posluživanja vodi nas do metode pregledavanjem (engl. *SCAN*). U ovoj metodi glava za čitanje pomicće se u jednom smjeru i pritom poslužuje sve zahtjeve na čije brojove staza nailazi. Dolaskom do krajnje staze glava mijenja smjer i nastavlja s posluživanjem novoprdošlih zahtjeva. Ova metoda može znatno poboljšati prosječno trajanje posluživanja, no također je moguća pojava neujednačenih posluživanja: primjerice, novoprdošli zahtjev čija se staza nalazi neposredno ispred glave za čitanje bit će poslužen gotovo trenutačno, dok će zahtjev koji pristigne neposredno *iza* glave za čitanje morati čekati promjenu smjera i ponovni dolazak do tražene staze.

Potrebno je napomenuti kako se u stvarnosti kretanje glave obično ne izvodi tako da se glava kreće do rubne staze diska, već se smjer kretanja mijenja ako ispred trenutačne pozicije glave ne postoji novi neposluženi zahtjevi (sa stazama koje se nalaze ispred glave u trenutačnom smjeru). U tom slučaju govorimo o *LOOK* metodi posluživanja.

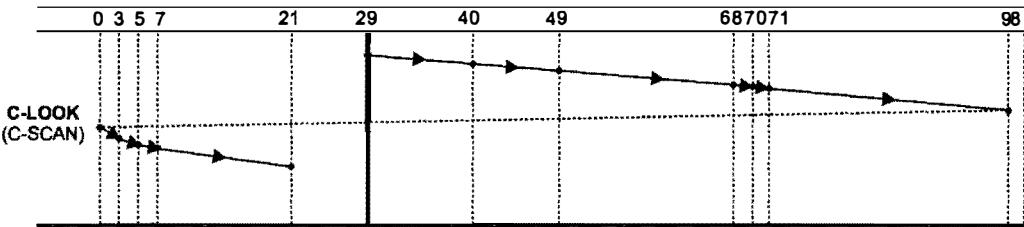
Ako se koristi metoda pregledavanja te ako se u početnom trenutku glava za čitanje kreće prema stazama s većim brojem, tijek posluživanja zahtjeva iz primjera 9.7. izgleda kako slijedi (ispredidanom linijom prikazan je dodatni pomak glave za *SCAN* metodu):



Slika 9.11. Posluživanje pregledavanjem

Posluživanje kružnim pregledavanjem

Inačica metode pregledavanja kojom se postiže ujednačenje trajanje posluživanja naziva se kružno pregledavanje (engl. *circular SCAN*, *C-SCAN*). U ovoj metodi zahtjevi se također poslužuju onim redom kako se glava pomiče po stazama u zadanom smjeru, no prilikom dolaska do krajnje staze, glava za čitanje pomiče se ponovno na početnu stazu i posluživanje se nastavlja istim zadanim smjerom kao u prethodnom prolazu. Treba naglasiti da se vraćanje glave na početni položaj obavlja mnogo većom brzinom te je utrošak vremena za povratak glave zanemariv. U praktičnim se ostvarenjima također glava ne kreće do rubne staze diska već samo do posljednjeg neposluženog zahtjeva u zadanom smjeru i u tom slučaju govorimo o *C-LOOK* metodi. Tijek posluživanja zahtjeva iz primjera 9.7. ovom metodom prikazan je na sljedećoj slici (ispredidanom linijom prikazan je dodatni pomak za *C-SCAN* metodu):



Slika 9.12. Posluživanje kružnim pregledavanjem

Odabir metode posluživanja

Važno je napomenuti da niti jedna od navedenih metoda ne jamči optimalni raspored posluživanja u vezi s najmanjim prijeđenim brojem staza (što je približno proporcionalno ukupnom trajanju posluživanja) za svaki skup zahtjeva. Moguće je optimalni redoslijed odrediti, no dodatni vremenski trošak izračunavanja redoslijeda (i ostvarenja algoritma) najčešće nije sumjerljiv mogućem poboljšanju u odnosu na metodu s najkraćim vremenom premještanja ili metodu pregledavanja.

Odabir metode posluživanja ovisit će u većini slučajeva o prirodi promatranog sustava i načinu uporabe datotečnog sustava. Učinak pojedine metode ovisi ponajviše o brojnosti i vrsti zahtjeva u sustavu (primjerice, višekorisnički pristup u poslužiteljskom sustavu).

U svakom slučaju, metoda posluživanja zahtjeva ostvaruje se kao posebni modul unutar operacijskog sustava koji se po potrebi može prilagoditi ili promijeniti.

Današnji diskovni uređaji također mogu uključivati neku od metoda posluživanja na koju operacijski sustav i ne mora imati utjecaj. U praktičnim ostvarenjima broj neposluženih zahtjeva na temelju kojih se donosi odluka o redoslijedu često je ograničen nekom vrijednošću. Primjerice, u redu operacijskog sustava za pristup disku trenutačno se može nalaziti desetak zahtjeva, no metoda posluživanja će odluku o sljedećem zahtjevu donositi samo na temelju tri najstarija zahtjeva u redu.



PITANJA ZA PROVJERU ZNANJA 9

1. Navesti sadržaj opisnika datoteke.
2. Gdje su pohranjeni:
 - a) opis smještaja datoteke,
 - b) opisnik datoteke i
 - c) datotečna tablica?
3. Navesti sadržaj datotečne tablice.
4. Na koji se način može prikazati slobodan prostor na disku?
5. Opisati bitovni prikaz slobodnog prostora na disku.
6. Opisati prikaz slobodnog prostora u obliku liste slobodnih blokova.
7. Opisati način smještaja datoteka u UNIX datotečnim podsustavima (*i-node*).
8. Opisati način smještaja datoteka u NTFS datotečnom podsustavu. Što je MFT? Kako se pohranjuju "male" datoteke?

10.

Komunikacija između procesa

Mogućnost komunikacija između procesa često je nužna funkcionalnost koju trebaju pružiti operacijski sustavi. Česti razlozi mogu biti u sudjelovanju procesa u obavljanju zajedničkog posla ili sinkronizacija radi korištenja dijeljenih sredstava. Komunikacija između procesa unutar istog računalnog sustava i komunikacija između procesa koji se nalaze u različitim računalnim sustavima obavlja se na različite načine te su zato u nastavku ovi načini komunikacije zasebno razmatrani.

10.1. Komunikacija između procesa unutar istog računalnog sustava

Komunikacija između dretvi koje djeluju unutar jednog procesa razmatrana je u poglavlju 6. Podsetimo se da sve dretve jednog procesa djeluju u njima zajedničkom adresnom prostoru. Prema tome, sve su lokacije u tom prostoru dostupne svim dretvama te govorimo da dretve djeluju u dijeljenom adresnom prostoru. To znači da se iz svake dretve mogu pohranjivati i dohvaćati sadržaji svih lokacija. Ipak se i u takvim uvjetima međudretvena komunikacija nastoji ostvarivati nekim sustavnim mehanizmima. Tako smo pokazali da se komunikacija između dretvi potrošača i proizvođača može obavljati preko međuspremnika te korištenjem jezgrinih funkcija za sinkronizaciju.

Procesi, međutim, imaju namjerno razdvojene adresne prostore kako jedan drugomu ne bi smetali. Mehanizmi za razdvajanje adresnih prostora koje je potrebno provesti unutar jednog računala razmatrani su u 8. poglavlju. Želimo li ostvariti komunikaciju između procesa tada bi u suštini te mehanizme trebalo namjerno oslabiti.

Najjednostavniji način komuniciranja između procesa jest razmjena datoteka:

- proces proizvođač proizvede datoteku i pohrani je na disk,
- proces potrošač otvorit će datoteku i preuzme njezin sadržaj.

Međutim, razmjena sadržaja preko datoteka nije uvek najprikladnija. Ona u komunikaciju unosi određenu staticnost. Zbog toga se uvode mehanizmi koji omogućuju dinamičku komunikaciju.

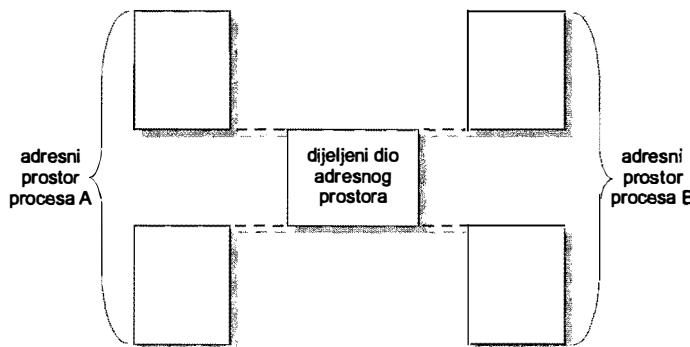
U načelu su moguća dva načina komuniciranja:

- zajednička uporaba izvornih podataka u dijeljenom spremničkom prostoru;
- uporaba kopija podataka zasnovana na razmjeni poruka.

10.1.1. Dijeljeni spremnički prostor

Znamo da svaki proces mora imati svoj vlastiti spremnički prostor. Uloga je operacijskog sustava da spremničke prostore procesa održava razdvojenim. Međutim, sve dretve istog procesa mogu slobodno pristupati svim lokacijama adresnog prostora samo svojeg procesa.

Suvremeni operacijski sustavi omogućuju da se dio virtualnog adresnog prostora više procesa (dva ili više) proglaši zajedničkim prostorom za te procese, kao što je prikazano slikom 10.1.



Slika 10.1. Dijeljeni adresni prostor

U tu svrhu operacijski sustav rezervira dio svog adresnog prostora i uvodi ga (preslikava) u adresne prostore procesa kao dijeljeni adresni prostor.

Sve dretve iz procesa A i procesa B mogu pristupiti do dijeljenog dijela adresnog prostora. To je najjednostavniji način razmjene podataka između procesa unutar jednog računala. Međutim, pri toj se razmjeni moraju uvažavati sva načela određenosti te je potrebno provesti sinkronizaciju procesâ, odnosno dretvi iz oba procesa korištenjem prikladnih sinkronizacijskih mehanizama (npr. semafora ili monitora).

Pojednostavljenog gledano, dretve svih procesa čije se domene ili kodomene (ili oboje) nalaze u dijeljenom dijelu adresnog prostora moraju se podvrgnuti pravilima komuniciranja opisanim u poglavljju 6.

Ostvarenja dijeljenog spremničkog prostora se u različitim operacijskim sustavima međusobno razlikuju.

**PRIMJER 10.1.**

U zbirci Win32 API nalaze se funkcije koje omogućuju da se cijele datoteke preslikaju u virtualni adresni prostor procesa i zatim je dijele s drugim procesima. U tu su svrhu predviđene funkcije:

- `CreateFileMapping()`,
- `MapViewOfFile()`,
- `MapViewOfFileEx()`,
- `OpenFileMapping()`.



U načelu se dijeljenje adresnog prostora svodi na dinamičko dijeljenje datoteke preslikane u virtualni adresni prostor procesa. Datoteka koja je preslikana u adresni prostor naziva se preslik datoteke (engl. *file view*).

Funkcija `MapViewOfFileEx()` čak dopušta da se preslika datoteke smjesti u željeni dio adresnog prostora. Ako je traženi dio adresnog prostora već dodijeljen, onda će funkcija dojaviti neuspjeli pokušaj preslikavanja. Drugi proces može dijeliti taj isti prostor ako preuzme presliku datoteke koju je kreirao prvi proces. Dijeljenje prostora može se načiniti i za više procesa.

Na UNIX operacijskim sustavima dijeljeni spremnički prostor ostvaruje se tako da se jezgrinim funkcijama najprije dohvati dijeljeni spremnički segment (funkcija `shmget`) koji se potom mapira u adresni prostor procesa (funkcija `shmat`). Ako novi procesi nastaju kopiranjem trenutačnog (funkcija `fork`), onda ti procesi nasljeđuju mapirani dio i preko njega mogu komunicirati. U protivnom, ako drugi procesi nastaju pokretanjem drugih programa, oni trebaju dohvatiti isti segment kao i prvi proces (funkcijom `shmget`), ali moraju poznavati ključ (prvi parametar funkcije `shmget`) te potom mapirati taj segment u svoj adresni prostor. Nakon mapiranja različiti procesi u svojim adresnim prostorima nalaze i izravno koriste zajednički spremnički prostor.

10.1.2. Razmjena poruka između procesa

Razmjena poruka između procesa može se ostvariti na način vrlo sličan razmjeni poruka između dretvi unutar jednog procesa.

Ostvarenje problema proizvođača i potrošača razmotrili smo u odjeljku 6.3. korištenjem dviju monitorskih funkcija:

- `poslati_poruku(p)` i
- `prihvati_poruku(r)`.

Pritom smo strukturu podataka potrebnu za ostvarenje tih funkcija te međuspremnik za razmjenu poruka smjestili u adresni prostor procesa. Koristili smo jezgrine funkcije:

- `ući_u_monitor`,
- `izaći_iz_monitora`,

- `uvrstiti_u_red_uvjeta,`
- `osloboditi_iz_reda_uvjeta.`

Na sličan bi se način moglo izgraditi *API* funkcije za komunikaciju između procesa, ali ne u adresnom prostoru procesa već u adresnom prostoru operacijskog sustava.

10.2. Komunikacija između procesa u raspodijeljenim sustavima

10.2.1. Osnove umrežavanja

Mnoge se ljudske aktivnosti zbivaju prostorno rasprostranjeno i stoga se od samih početaka uporabe računala nastoji pristup do odgovarajućih računalnih usluga ostvariti na onim mjestima gdje su one potrebne pa se računala međusobno povezuju u mrežu.

Velika je većina današnjih računala umrežena. Računala djeluju pod utjecajem svojih vlastitih operacijskih sustava koji ne moraju biti jednaki. Operacijski sustavi, međutim, moraju imati komunikacijski podsustav koji podržava razmjenu poruka između računala.

Svrha međusobnog povezivanja računala jest razmjena informacija. Svi oblici informacija (tekstovi, zvuk i slike) u računalima se pohranjuju u digitaliziranom obliku u *datotekama*.

Prema tome, pojednostavljeni gledano, povezivanje računala moglo bi se svesti na uspostavljanje načina razmjene datoteka, pri čemu se mora čvrsto dogovoriti detalje prenošenja koje svi sudionici moraju poštovati. Utvrđena se pravila komuniciranja nazivaju *protokolima*. Za uspostavljanje funkcionalnog povezivanja potrebno je utvrditi skup protokola, tzv. protokolni slog (engl. *protocol stack, protocol suite*).

Mrežni podsustav operacijskog sustava koji ostvaruje potrebni protokolni slog vrlo je složen podsustav koji zaslužuje zasebno razmatranje. Detaljnije razmatranje prelazi granice ove knjige te će u nastavku mrežni podsustav biti prikazan samo površno, prikazujući samo osnovne elemente podsustava i osnovne principe korištene u ostvarenju komunikacije.

Međunarodna telekomunikacijska unija (engl. *International Telecommunication Union – ITU*) u ime Međunarodne organizacije za normizaciju (engl. *International Standard Organization – ISO*) propisala je protokolni slog kojim se uspostavlja sedmerslojni referentni model povezivanja otvorenih sustava (engl. *Open System Interconnection*) poznat kao *ISO/OSI model*.

Međutim, u današnjoj globalnoj mreži *Internetu* prevladao je protokolni slog nastao postupno tijekom razvitka mreže kao skup *de facto* normi. O tehničkom aspektu Interneta vodi brigu konzorcij *Internet Engineering Task Force (IETF)*. IETF utvrđuje i protokolni slog koji se danas pretežito upotrebljava pri povezivanju računala u mrežu.

Taj protokolni slog ima četiri razine:

- primjensku (engl. *application layer*),

- prijenosnu (engl. *transport layer*),
- mrežnu (engl. *internet layer*) i
- razinu prijenosnih putova (engl. *link layer*).

U najvišoj *primjenskoj razini* utvrđena su pravila ostvarenja i uporabe pojedinih mrežnih usluga. Primjeri protokola primjenske razine su:

- prijenos hiperteksta (engl. *Hyper Text Transfer Protocol – HTTP*),
- elektronička pošta (engl. *Simple Mail Transfer Protocol – SMTP*),
- prijenos datoteka na daljinu (engl. *File Transfer Protocol – FTP*),
- diskusjske skupine (engl. *Newsgroups*) i sl.

Protokoli primjenske razine definiraju formate podataka i naredbi potrebnih u raspodijeljenim primjenskim programima. Tako npr. *HTTP* definira načine na koje klijent (Web preglednik) traži određene datoteke od poslužitelja (Web poslužitelja) te načine odgovora poslužitelja. Sama razmjena podataka obavlja se korištenjem *API* funkcije sljedeće niže, tzv. prijenosne razine.

U toj su razini utvrđeni *prijenosni protokoli* čiji su zadaci prijenos podataka između raspodijeljenih primjenskih programa. Najznačajniji protokoli u ovoj razini su *TCP* (engl. *Transmission Control Protocol*) i *UDP* (engl. *User Datagram Protocol*).

TCP za razliku od *UDP*-a nudi pouzdan prijenos podataka, uključujući postupke za oporavak od pogrešaka, isporučujući potpunu informaciju u originalnom redoslijedu. On određuje kako se početna informacija, koja se može sastojati od proizvoljno dugog niza okteta, po potrebi dijeli na dijelove i oblikuje u tzv. *prijenosne jedinke* (pakete), te način dodavanja rednog broja i dodatnih nadzornih i zaštitnih okteta toj jedinki.

S druge strane, *UDP* je jednostavniji i češće se koristi kada se prenose manje količine podataka (npr. cijela se komunikacija svodi na par poruka zahtjev / odgovor).

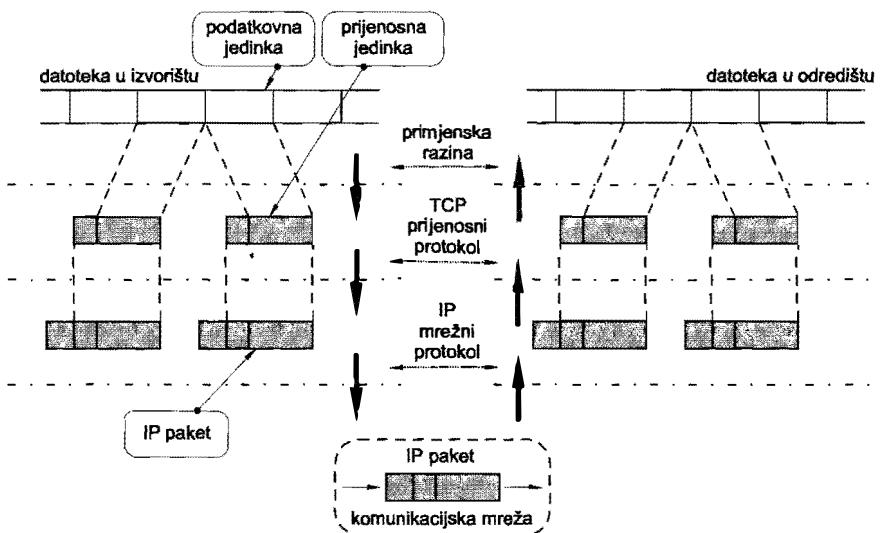
Prijenosni protokoli idućem nižem protokolu predaju (i od njih preuzimaju) prijenosne jedinke koje osim izvornih informacija, zaštitnih i ostalih bitova sadrže i adresu odredišnog primjenskog programa (adresa uključuje adresu odredišnog računala i identifikaciju programa na tom računalu). Drugim riječima, prijenosni protokoli logički povezuju primjenske programe koji su smješteni na različitim računalima povezanim u mrežu.

Nadopunjene prijenosne jedinke prijenosnog sloja dobivaju u sljedećoj nižoj *mrežnoj razini* dodatno zaglavje unutar kojeg se nalaze i adrese izvorišnog računala (engl. *source*) i odredišnog računala (engl. *destination*). Najznačajniji protokol mrežne razine je Internet protokol, kratice *IP*. Nadopunjena prijenosna jedinka u mrežnoj razini tada postaje *IP paket*, koji će posredstvom prijenosnih putova stići u odredište.

Prijenosni putovi čine četvrtu, najnižu razinu protokola koja se dijeli na dva sloja: *podatkovni* i *fizički*. Ova razina nije posebice normirana i omogućuje razne načine povezivanja u različitim mrežama. Primjerice može biti ostvarena žičnim vezama (npr. korištenjem *ethernet* protokola), ili korištenjem optike, ili bežičnom mrežom (engl. *wireless LAN*), ili modemskim vezama i slično.

Velik je uspjeh IP polučio upravo zbog toga što je prijenos paketa moguć uporabom različitih mrežnih tehničkih rješenja.

Slika 10.2. prikazuje opisani prijenos datoteke s jednog računala na drugo.

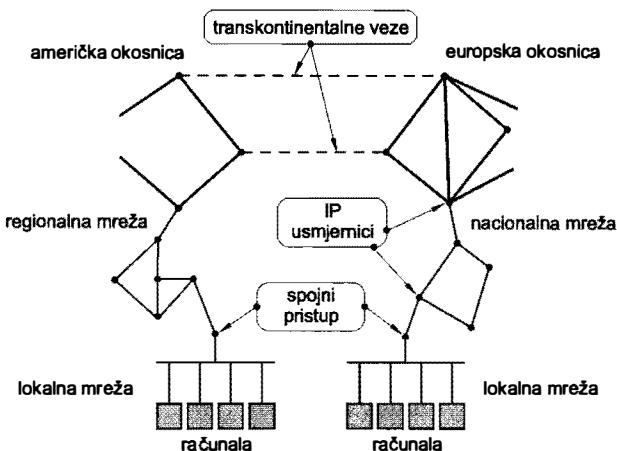


Slika 10.2. Prijenos datoteke prikazan u protokolarnom slogu Internet

Iz slike je vidljivo da se pri slanju početna informacija (datoteka) dijeli u podatkovne jedinke kojima se u sljedećoj nižoj razini dodaju zaglavlja *TCP* protokola. *TCP* paket prihvatom u mrežni protokol – *IP* dobiva dodatno zaglavlje te se kao *IP* paket proslijedi u iduću razinu. *IP* paket će tako korištenjem prijenosnih putova (komunikacijske mreže) prispjeti do odredišnog računala. Komunikacijska mreža ovisno o početnom i odredišnom računalu može se sastojati od više čvorova, tj. paket može proći kroz nekoliko računala prije nego li dode do odredišta. Na odredišnoj će strani *IP* razina iz prispjelog paketa otkloniti svoje zaglavlje i predati prijenosnu jedinku *TCP* razini unutar koje se na temelju rednih brojeva uspostavlja izvorni oblik datoteke. U toj se razini provjerava ispravnost prispjelih jedinki i traži od izvořišta ponovno slanje neispravnih te predaje uređene podatkovne jedinke u primjensku razinu. U primjenskoj će se razini uspostaviti izvorni oblik prenesene datoteke.

10.2.2. Struktura Interneta

Struktura globalne mreže računala nije čvrsto definirana, ali se u njoj može prepoznati nekoliko kontinentalnih *okosnica* (engl. *backbones*) sastavljenih od vrlo brzih komunikacijskih veza i *IP usmjernika* (engl. *IP routers*). Slika 10.3. prikazuje dio strukture globalne mreže koja uključuje američku i europsku okosnicu.

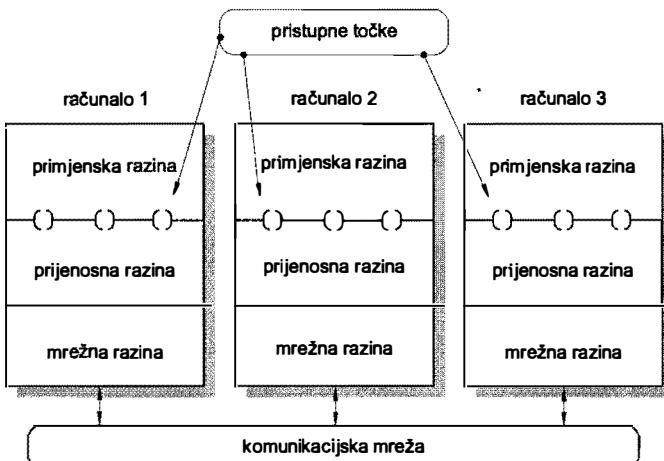


Slika 10.3. Primjer strukture globalne mreže

Okosnice su međusobno povezane međukontinentalnim vezama. Na okosnice su povezane nacionalne ili regionalne mreže, također sastavljene od međusobno povezanih usmjernika, a na te su mreže preko *spojnih pristupa* (engl. gateways) povezane različite lokalne mreže načinjene od radnih računala (engl. hosts).

Svako računalo ima svoju jedinstvenu adresu koju čini broj iz kojeg se može jednoznačno ustanoviti kojoj lokalnoj mreži, kojoj regionalnoj ili nacionalnoj mreži i kojoj kontinentalnoj okosnici ono pripada. U okviru konzorcija IETF postoji posebna radna skupina *Internet Assigned Numbers Authority (IANA)* koja uspostavlja sustav dodjele brojeva.

Paketi koji kreću iz izvođačnog računala takvim se načinom adresiranja mogu djelotvorno usmjeravati kroz mrežu. Paketi od izvođača do odredišta mogu biti usmjeravani i različi-



Slika 10.4. Pristupne točke

tim putovima i u odredište pristizati različitim redoslijedima. Protokol prijenosne razine (npr. TCP) mora se pobrinuti da se paketi na odredišnoj strani ispravno poredaju kao i da se predaju odgovarajućem primjenskom programu. Tome služe posebne skupine bitova u zaglavlju paketa koje adresiraju odgovarajuće *pristupne točke* (engl. *servis access points – SAP*), kao što ilustrira slika 10.4.

Primjenski programi koriste mrežni podsustav preko tih pristupnih točaka.

10.2.3. Komunikacija između procesa

Prvi način komuniciranja – komunikacija razmjenom poruka

S obzirom na to da procesi u raspodijeljenim sustavima ne dijele spremnički prostor u takvim se sustavima kao osnovni model komuniciranja nameće razmjena poruka. U primjenskoj razini može se iznad *TCP/IP* razina izgraditi komunikacijske mehanizme kojima će se uspostaviti protokol razmjene poruka. Komunikacijski mehanizmi mogu se aktivirati odgovarajućim *API* funkcijama.

Poruke će se oblikovati kao *IP* paketi. Kratke poruke smjestit će se unutar jedne podatkovne jedinke, a poruke većih dimenzija se, po analogiji s prijenosom datoteka, mogu podijeliti na veći broj podatkovnih jedinki i poslati kao niz *IP* paketa.

Kada proces želi komunicirati, on mora uspostaviti komunikacijsku priključnicu (engl. *socket*).

Funkcijom:

`uspostaviti_priklučnicu (identifikator, tip, protokol)`

proces uspostavlja vezu s prijenosnom razinom unutar svog računala kroz jednu pristupnu točku. Uspostavljena priključnica mora se imenovati, tj. povezati s imenom:

`povezati (identifikator, adresa).`

Ako dva računala uspostave kompatibilne priključnice, onda ih se može međusobno povezati. Povezivanje se može obaviti asimetričnom funkcijom:

`povezati (identifikator, adr_vlastita, adr_partnera).`

Funkcija je asimetrična jer jedan proces pokušava uspostaviti vezu dok drugi čeka na postavljanja zahtjeva. Ovakav je način uspostavljanja veze prikladan za komuniciranje klijenata i poslužitelja. Poslužitelj stvara priključnicu, povezuje s njom ime i čeka da se neki klijent poveže na nju. Kada poslužitelj uspostavi vezu, tada može početi slanje poruka na sličan način kako se to obavlja preko dvostranog cjevovoda unutar jednog računala.



PRIMJER 10.2.

U prethodnom primjeru podrazumijeva se postojanje programske potpore u primjenskoj razini koja uključuje mehanizme razmjene poruka između procesa u raspodijeljenom sustavu. Jedno ostvarenje opisane funkcionalnosti dostupno je uz pomoć norme *MPI* (engl. *Message Passing Interface*¹).

Norma *MPI* definira način razmjene poruka između procesa koji se mogu izvoditi na istom ili različitim računalima. Danas je *MPI* dostupan na gotovo svim sustavima za paralelno računanje i praktički je sinonim za biblioteku razmjene poruka. Norma omogućuje razvoj prenosivih primjenskih programa koji se izvode u raspodijeljenoj okolini, a posebno je prikladan za korištenje u okruženju računalnog grozda (engl. *cluster*) ili spleta računala (engl. *grid*). Različite implementacije norme mogu iskoristiti značajke računalne okoline u kojoj se izvode i na taj način postići vrlo visoku razinu učinkovitosti u međuprocesnoj komunikaciji.

U *MPI* okruženju paralelizam je eksplicitan, što znači da je programer odgovoran za ostvarenje paralelnog algoritma. U većini primjena to uključuje oblikovanje raspodjele podataka i komunikacije među procesima.

Norma nudi sljedeće razine funkcionalnosti:

- komunikaciju između dvaju procesa (engl. *point-to-point*);
- komunikaciju unutar grupe procesa (engl. *collective communication*);
- asinkronu komunikaciju (engl. *asynchronous communication*);
- grupiranje procesa u virtualne topologije.

Kako bi ikakva komunikacija mogla biti uspostavljena, norma definira funkcije koje svakom procesu dodjeljuju jedinstveni redni broj u grupi procesa. U okviru komunikacije između dvaju procesa, proces *pošiljatelj* mora pozvati odgovarajuću funkciju za slanje, a proces *primatelj* funkciju za primanje poruke. Jedan je od uvjeta uspješne komunikacije podudaranje rednih brojeva procesa pošiljatelja i primatelja. Proses primatelj ima dodatnu mogućnost primanja poruke od bilo kojeg pošiljatelja, čime se omogućuje izvedba nedeterminističkog programskog modela. S druge strane, redoslijed poruka od jednog procesa pošiljatelja jednom procesu primatelju očuvan je i jednak u obama procesima.

Globalna komunikacija uključuje funkcije u kojima jedan proces šalje podatke ostalim procesima, jedan proces skuplja podatke od više procesa te druge inačice ovih komunikacijskih modela. U okviru asinkrone komunikacije norma omogućuje procesima dinamičko ispitivanje postojanja poruka, čekanje na poruke ili preplitanje računanja i primanja ili slanja poruka.

Mehanizmi virtualne topologije omogućuju definiranje *susjedskih* relacija među procesima koji odgovaraju različitim oblicima (graf, višedimenzijsko polje, piramida itd.).

¹ Norma MPI dostupna je na <http://www.mcs.anl.gov/mpi/>. Za korištenje norme potrebna je neka od programskih implementacija od kojih su najpoznatije MPICH (<http://www.mcs.anl.gov/mpi/mpich/>) ili OpenMPI (<http://www.open-mpi.org/>).

U primjenama na specijaliziranoj računalnoj opremi, primjerice računalima s procesorima povezanim po modelu *hiperocke* (engl. *hypercube*), normom je moguće uskladiti programski i računalni komunikacijski model te time postići veću učinkovitost paralelnog programa.

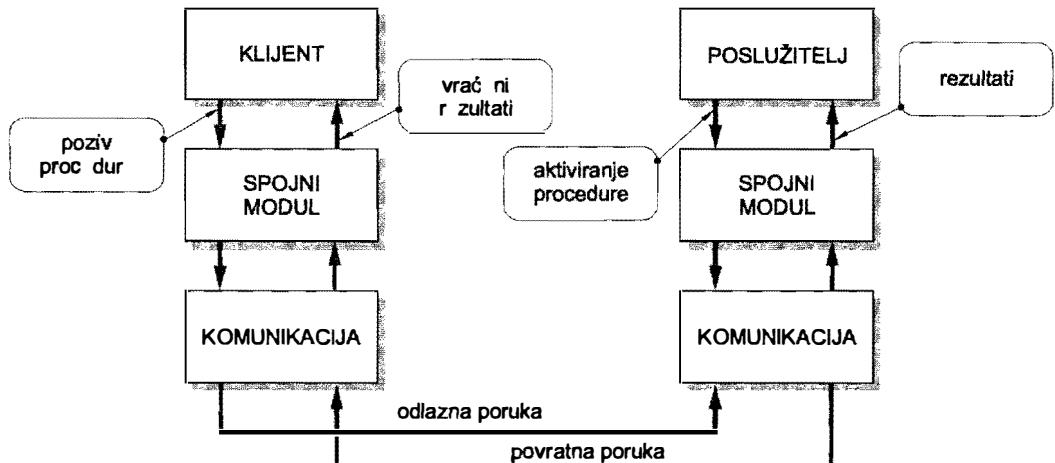
Drugi način komuniciranja – poziv udaljenih procedura

Poziv udaljenih procedura (engl. *Remote Procedure Call – RPC*) svojevrsna je nadgradnja mehanizama razmjene poruka. Ta je nadgradnja razrađena kako bi se programerima olakšala priprema programa u raspodijeljenom okruženju. Mehanizam oponaša dobro poznate načine uporabe potprograma koji se pripremaju u adresnom prostoru procesa.

Koncepcija potprograma opisana je u poglavlju 2. Potprogram čini niz instrukcija koje obavljaju često korištenu funkciju. Prilikom poziva potprograma ulazni se podaci preko jasno specificiranih lokacija prenose potprogramu, a nakon završetka niza instrukcija potprograma rezultati se vraćaju u nadređeni program. U potprogram se mogu prenositi vrijednosti (engl. *call by value*) ili adrese (engl. *call by reference*) ulaznih podataka, a na jednaka dva načina mogu se vraćati i rezultati.

Mehanizam poziva udaljenih procedura jest proširenje toga koncepta, s tim da se instrukcije procedure ne nalaze u adresnom prostoru procesa iz kojeg se pozivaju. S obzirom na to da se proces odakle dolazi poziv i proces kuda je poziv procedure upućen ne nalaze u istom adresnom prostoru, razmjena ulaznih podataka i rezultata mora se obaviti razmjenom poruka, i to *prenošenjem vrijednosti*.

Slika 10.5. prikazuje način ostvarenja poziva udaljenih procedura korištenjem spojnih modula izgrađenih iznad TCP prijenosnog protokola.



Slika 10.5. Pozivi udaljenih procedura

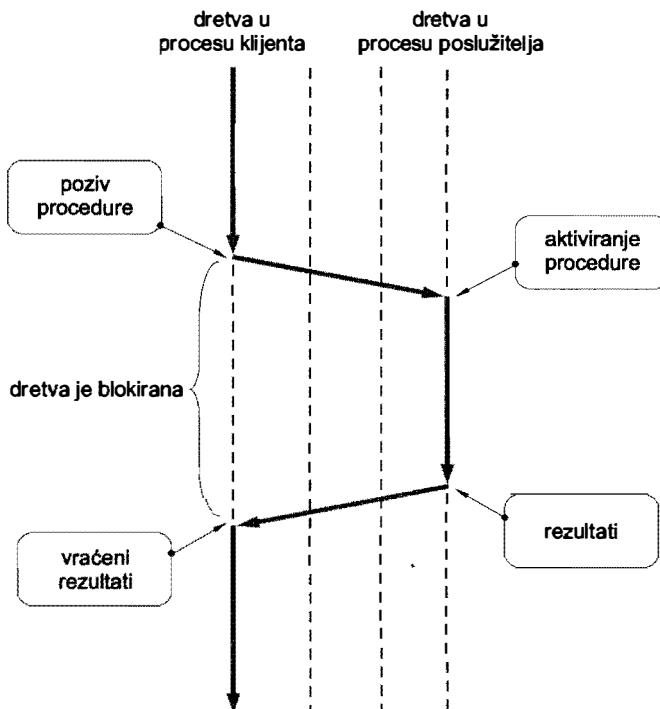
Mehanizam poziva udaljenih procedura ostvaruje se iznad TCP razine tako da poziv udaljene procedure izgleda kao uobičajeni poziv potprograma. Taj poziv prihvata *spojni modul* (engl. *stub*) koji oblikuje poruku i prepusta poruku komunikacijskom sustavu.

Poruka se upućuje računalu koje će izvesti proceduru. U njegovu spojnom modulu u poruci će se prepoznati ime procedure i njezine ulazne vrijednosti. Procedura nakon svog završetka predaje rezultate svom spojnom modulu, koji će oblikovati povratnu poruku i predati je komunikacijskom sustavu. Na strani koja je pozvala proceduru spojni modul rezultate prispjele povratnom porukom vraća u proces koji je pozvao proceduru.

Proces koji poziva proceduru zapravo je *klijent*, a proces koji izvodi proceduru je *poslužitelj*. Pokazuje se da je odnos *klijent – poslužitelj* vrlo koristan pristup pri rješavanju mnogih zadataka.

Ostvarenje protokola može se ostvariti izgradnjom monitora na način opisan u poglavlju 6.

Najjednostavniji protokol poziva udaljenih procedura koji prikazuje samo jedan poziv udaljene procedure prikazan je na slici 10.6.



Slika 10.6. Ovdjivanje poziva udaljene procedure

Proces koji poziva potprogram treba nastaviti izvođenje dretve na mjestu iza poziva potprograma. Protokol poziva udaljenih procedura trebao bi dakle blokirati izvođenje dretve procesa klijenta koja je pozvala udaljenu proceduru dok se ne vrate rezultati.

U višedretvenom procesu klijenta moglo bi se naći dretve koje bi mogle biti aktivirane u vremenu čekanja na rezultat.

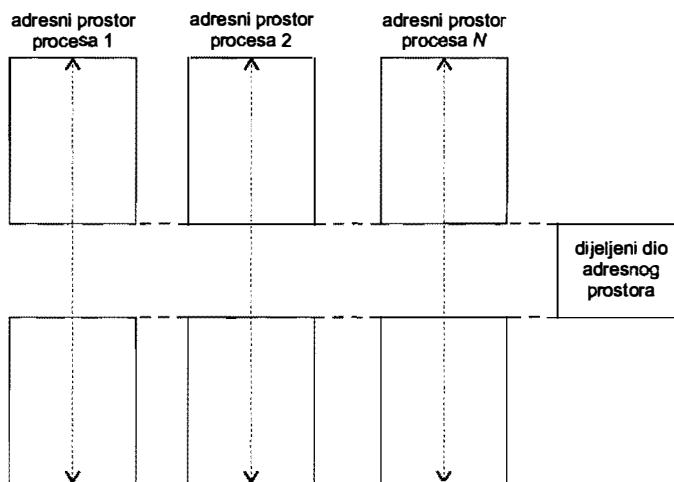
Treći način komuniciranja – raspodijeljeni dijeljeni spremnički prostor

Unutar jednog računala procesi mogu dijeliti dio spremničkog prostora kao što je opisano u odjeljku 10.1.1. U takvom načinu dijeljenja spremničkog prostora pojedine instrukcije dvaju ili više procesa mogu adresirati pojedinačne spremničke lokacije dijeljenog dijela spremnika.

Virtualni se spremnički prostor ostvaruje tako da se logički adresni prostor procesa dijeli na stranice koje se dinamički smještaju u radni spremnik, i to onda kada je to potrebno. Ako procesor adresira stranicu koja trenutačno nije u radnom spremniku, generira se prekid zbog promašaja stranice i aktivira se postupak dobavljanja stranice. Programer se ne mora brinuti o detaljima ostvarenja virtualnog spremnika. On jednostavno upotrebljava veliki adresni prostor cijelog procesa. Time smo se detaljno bavili u poglavlju 8.

Postavlja se pitanje može li se na sličan način razmišljati o jednom dijeljenom spremniku kojeg će jednostavno moći dohvaćati svi procesi koji se odvijaju u različitim računalima mreže.

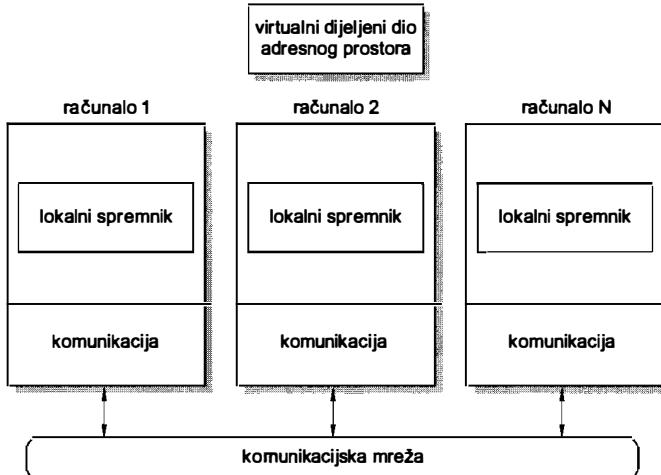
Zamislimo da se u svakom od N računala povezanih u mrežu izvodi po jedan proces koji djeluje u svom adresnom prostoru. Mi želimo da jedan dio adresnog prostora dijele svi ti procesi prema slici 10.7.



Slika 10.7. Dijeljeni spremnički prostor

Taj dijeljeni dio prostora može biti samo virtualan jer procesi mogu stvarno komunicirati samo razmjenom poruka kroz komunikacijsku mrežu, kako je to ilustrirano slikom 10.8.

Prepostavimo da se u svim računalima upotrebljava virtualni spremnik te da je i dijeljeni dio adresnog prostora podijeljen na stranice jednakih veličina.



Slika 10.8. Virtualni dijeljeni spremnički prostor

Ako se u jednom od računala dogodi promašaj stranice dijeljenog dijela spremnika, onda se može dogoditi:

- da se tražena stranica nalazi u računalu (na njegovu disku) u kojem se dogodio promašaj ili
- da se stranica nalazi u nekom drugom računalu.

U prvom slučaju obavit će se uobičajena zamjena stranica, a u drugom treba pokrenuti postupak dobavljanja stranice kroz komunikacijsku mrežu. Zamjena stranice potrajat će u drugom slučaju mnogo duže jer se mora pokrenuti čitav mehanizam razmjene poruka.

Ostvarenje dijeljenog raspodijeljenog spremnika zahtijeva pomnjuvu analizu mnogih detalja. Primjerice, kako bi se povećala djelotvornost sustava poželjno bi bilo da se kopije pojedinih stranica koje se često upotrebljavaju istodobno nalaze u više računala. Ako se iz tih stranica samo čita, onda nema nikakvih problema. Međutim, ako jedan od procesa piše u takvu stranicu tada treba načiniti promjene u svim njezinim kopijama. Tako dugo dok postoje razlike u sadržajima stranica dijeljeni spremnik nije *koherentan*, odnosno *konzistentan* (u uporabi su oba termina).

Postoji nekoliko protokola kojima se podržava koherencija, ali oni ovdje neće biti dalje razmatrani. Ostvarenje dijeljenog adresnog prostora u raspodijeljenim sustavima još je uvijek predmet istraživanja.

U praktičnoj su primjeni prva dva opisana načina komuniciranja: komunikacija razmjenom poruka i poziv udaljenih procedura.

10.3. Međusobno isključivanje u raspodijeljenim sustavima

10.3.1. Međusobno isključivanje – osnovni mehanizam ostvarenja funkcija operacijskog sustava

Međusobno isključivanje jezgrinih funkcija jednog računala

U 4. smu poglavlju ustanovili da je međusobno isključivanje osnovni mehanizam koji omogućuje ostvarenje svih funkcija jezgre. Podsetimo se da je međusobno isključivanje poduprto nekim sklopovskim rješenjima. Tako je u jednoprocесorskim sustavima međusobno isključivanje moguće postići *zabranom prekidanja*. Dretva koja želi ući u kritični odsječak instrukcijom zabrane prekidanje spriječit će svoje prekidanje. Na završetku kritičnog odsječka ona će dopustiti prekidanje. S obzirom na to da je ulazak u jezgru izazvan sklopovskim ili programskim prekidom, svaka se jezgrina funkcija obavlja međusobno isključeno s drugom jezgrinom funkcijom.

Dvije najvažnije jezgrine funkcije odnose se na binarni semafor. Jedna ga ispituje, a druga postavlja. U popisu API funkcija taj se semafor obično naziva *mutex* (od engl. *mutual exclusion*). Binarni semafor propustit će u kritični odsječak onu dretvu koja prva pozove funkciju ispitivanja prolaznog semafora. Sljedeće dretve koje ispituju zauzeti semafor bit će blokirane. Kada dretva napušta kritični odsječak ona poziva funkciju postavljanja semafora koja će semafor vratiti u prolazno stanje (ako ni jedna dretva u međuvremenu nije tražila prolaz) ili će propustiti u kritični odsječak prvu dretvu koja čeka u redu na semaforu.

U višeprsesorskim sustavima mehanizam zabrane prekidanja nije dovoljan za ostvarenje međusobnog isključivanja. Zbog toga je uveden novi mehanizam *nedjeljivog čitanja i pisanja*, kojim se omogućuje da procesori u čvrsto povezanom sustavu (s jednom sabirnicom i dodjeljivačem sabirnice) mogu nesmetano ispitati zastavicu smještenu u dijeljenom spremničkom prostoru. Ulazak u jezgru moguće je samo ako se uspješno obavi ispitivanje zastavice (koju smo u poglavlju 5. nazvali *Ograda_jezgre*). Ukoliko zastavica nije prolazna procesor iz kojeg je iniciran poziv jezgre obavlja radno čekanje. Na taj je način jednostavno riješeno pitanje međusobnog isključivanja jezgrinih funkcija i u višeprsesorskim sustavima.

Međusobno isključivanje na razini primjenskih programa unutar jednog računala

Na primjenskoj razini problem međusobnog isključivanja rješava se jednostavnim pozivanjem jezgrinih funkcija. Na toj se razini ne vide mehanizmi primijenjeni za ostvarivanje tih funkcija.

Binarni se semafor može gledati kao objekt i reći da je *binarni semafor* ili *mutex* objekt (možemo ga nazvati i značkom), koji može biti ili slobodan ili pripadati nekoj dretvi. Dretva koja prva dohvati taj objekt može nastaviti svoje izvođenje. Dretva koja ga ne

uspije dobiti mora obustaviti svoje izvođenje i čekati dok se objekt ne oslobodi. Dretve koje posjeduju objekt moraju ga u nekom konačnom vremenu i vratiti.

Za uspješno pisanje primjenskih programa potrebno je upoznati pripadne API funkcije kojima se može kreirati i uništavati takve objekte te obavljati nad njima osnovne operacije. Dretve koje sudjeluju u natjecanju za ulazak u kritične odsječke mogu se nalaziti unutar jednog procesa ili u različitim procesima. Ako se dretve nalaze u različitim procesima, onda semafori moraju biti deklarirani tako da su dostupni svim zainteresiranim procesima.

Međusobno isključivanje u raspodijeljenom sustavu

U raspodijeljenom sustavu ne postoji mogućnost sklopoške potpore međusobnog isključivanja. U sustavu se mogu samo razmjenjivati poruke. To donekle podsjeća na stanje u višeprcesorskom sustavu kod kojeg ne bi bilo mehanizma nedjeljivog čitanja i pisanja. U odjeljku 4.4. opisan je Lamportov protokol koji u tim uvjetima uspješno ostvaruje međusobno isključivanje. Postavlja se pitanje može li se nešto slično ostvariti i u raspodijeljenim sustavima. Prije nego li se razmotre neki protokoli međusobnog isključivanja opisat će se mogućnost praćenja vremenskog uređenja događaja u raspodijeljenim sustavima.

10.3.2. Vremensko uređenje događaja u raspodijeljenim sustavima

Prepostavit ćemo da promatramo N računala povezanih komunikacijskom mrežom u kojoj je razrađen protokol razmjene poruka. Nazovimo svako takvo računalo *čvorom* i prepostavimo da se u svakom računalu odvija jedan proces. Indeks i procesa P_i jednak je indeksu čvora.

Promatramo dakle N procesa koji međusobno nesmetano komuniciraju razmjenom poruka.

Nadalje, neka je sustav potpuno povezan i neka svaki proces može poslati poruku neposredno svakom drugom procesu. Također, neka sve poruke stižu u nekom konačnom vremenu do svog odredišta bez pogrešaka kako se razmatranja ne bi otežala postupcima oporavaka od pogrešaka. Nadalje, prepostavljat ćemo da poruke poslane iz jednog čvora prema drugom na odredište stižu onim redom kako su odašiljane (ta prepostavka ne mora biti ispunjena u stvarnim mrežama!).

Unutar svakog čvora može se mjeriti vrijeme jer u njemu postoji lokalni sat. Međutim, sat koji bi vrijedio za cijelu mrežu ne postoji. U načelu bi se za cijelu mrežu moglo primijeniti vrijeme koje emitira GPS sustav, ali nas i ne zanima apsolutno vrijeme već samo vremensko uređenje pojedinih događaja u sustavu.

Lokalni logički sat

U svakom čvoru i može postojati *lokalni logički sat* C_i koji se ponaša kao brojilo. On povećava svoju vrijednost nakon svakog karakterističnog događaja unutar procesa P_i .

Unutar jednog procesa može se jednoznačno utvrditi redoslijed događaja.

Nekom događaju a pripada vrijednost logičkog sata $C_i(a)$, a događaju b vrijednost logičkog sata $C_i(b)$.

Ako je $C_i(a) < C_i(b)$, onda se događaj a dogodio prije događaja b, što možemo pisati:

$$a \rightarrow b,$$

gdje \rightarrow označava relaciju "dogodilo se prije".

Relaciju "dogodilo se prije" možemo jednoznačno odrediti:

- za dva događaja *unutar jednog procesa* ili
- za događaj slanja poruke od strane jednog procesa i primjeka te *iste poruke* u drugom procesu.

Prema tome, relacija $a \rightarrow b$ vrijedi:

- ako su a i b događaji unutar jednog procesa i događaj a se zbio prije događaja b ili
- ako je a događaj slanja poruka od strane jednog procesa i b događaj primjeka te iste poruke od strane drugog procesa.

Nadalje, relacija $a \rightarrow b$ je tranzitivna pa vrijedi:

$$(a \rightarrow b) \wedge (b \rightarrow c) \text{ povlači da je } (a \rightarrow c).$$

Ako ni $(a \rightarrow b)$ ni $(b \rightarrow a)$ nije istinito, onda događaji a i b nisu vremenski uređeni.

Globalni logički sat

Na razini sustava može se uspostaviti skup pravila vremenskog uređenja koji nazivamo *globalni logički sat*. Globalni sat zasniva se na sljedećim pravilima:

- proces P_i povećava svoj logički sat C_i između svaka svoja dva događaja;
- kada proces P_i šalje poruku m, on uz nju pridodaje vremensku oznaku $T_m = C_i$;
- kada proces P_j primi poruku, on postavlja u svoj logički sat C_j vrijednost koja je veća od njegove prethodne vrijednosti i veća od prispeje vremenske oznake T_m .

Pri ovakovom ostvarenju globalnog sata može se dogoditi:

- da dva lokalna sata C_i i C_j imaju jednakе vrijednosti i
- da se njihove poruke koje si međusobno šalju presretnu na putu noseći jednakе vrijednosti vremenskih oznaka.

U tom slučaju dobit ćemo jednakе vremenske oznake za dva različita događaja. Po uzoru na Lamportov protokol opisan u odjeljku 4.4. uređenje možemo postići usporedbom vrijednosti indeksa koji su unaprijed pridodeljeni svakom čvoru. Prema tome, u sustavu se može podržavati globalni sat koji čine nakupine lokalnih satova.

Relacija vremenskog uređaja:

$$a \Rightarrow b \quad (a \text{ "prethodi" } b)$$

za događaj a iz procesa P_i i događaj b iz procesa P_j zadovoljena je kada je:

$$(C_i(a) < C_j(b))$$

ili:

$$(C_i(a) = C_j(b)) \wedge (i < j).$$

Utvrđivanje redoslijeda događaja olakšat će nam ostvarenje mehanizama međusobnog isključivanja u raspodijeljenim sustavima.

10.3.3. Protokoli međusobnog isključivanja u raspodijeljenim sustavima

Centralizirani protokol

U potpuno centraliziranom protokolu jedan od čvorova u mreži može se proglašiti odgovornim za ostvarenje međusobnog isključivanja. U njemu se nalaze svi podaci i ostali čvorovi moraju od tog čvora tražiti dozvolu za ulazak u kritični odsječak.

Proces čvora P_i (jedna dretva unutar tog procesa) koji želi ući u kritični odsječak mora poslati poruku *zahtjev(i)* centralnom čvoru. Centralni čvor će prihvati zahtjev i poslati poruku *odgovor(i)* čvoru P_i kada ustanovi da on smije ući u kritični odsječak.

Proces P_i (odnosno njegova dretva) mora pričekati da dobije odgovor prije nego što uđe u kritični odsječak. Nakon što dretva u čvoru i obavi kritični odsječak ona šalje poruku *izlaz(i)* centralnom čvoru.

Centralni čvor organizira red prisjelih zahtjeva i dopušta ulazak u kritični odsječak po redu prispijeća nakon što primi poruku *izlaz(i)*.

Za ostvarenje ulaska u kritični odsječak za jedan proces potrebne su, dakle, samo dvije poruke!

Na taj se protokol može gledati i na drugi način. Može se reći da centralni čvor posjeduje jedan objekt (značku) koji će slati čvorovima koji traže ulazak u kritični odsječak. Značka se može poslati samo jednomu od čvorova. Čvor mora vratiti značku u konačnom vremenu centralnom čvoru.

Osnovni nedostatak ovoga protokola jest velika ovisnost o centralnom čvoru. Ako se centralni čvor pokvari, protokol će u potpunosti zatajiti.

Protokol s putujućom značkom

Pokazuje se da nam centralni čvor nije ni potreban. U raspodijeljenom se sustavu definira jedna značka koja kao poruka ciklički putuje kroz sve čvorove.

Kada poruka značke prispije u čvor i proces P_i će:

- zadržati značku ako želi ući u kritični odsječak te poslati značku sljedećem čvoru tek nakon što ga završi ili
- prosljediti značku sljedećem čvoru ako ne želi ulaziti u kritični odsječak.

Ovaj ciklički protokol po svojoj je zamisli sličan mehanizmu dodjele sabirnice u više-procesorskom čvrsto povezanom sustavu. Tamo se sabirnica redom ciklički nudi svim procesorima, a pravo pristupa koriste samo oni procesori koji to žele.

Nedostaci ovakvog protokola uključuju neotpornost na ispad bilo kojeg čvora koji će prekinuti lanac prosljeđivanja značke. Nadalje, i u trenucima kada niti jedan čvor ne želi ući

u kritični odsječak, značka se i dalje prosljeđuje kroz čvorove, nepotrebno opterećujući komunikacijski sustav.

Lamportov raspodijeljeni protokol

Lamportov raspodijeljeni protokol zasniva se na uvažavanju vremenskog uređenja temeljenog na globalnom logičkom satu.

Kada proces P_i želi ući u kritični odsječak, generirat će poruku $\text{zahtjev}(i, T(i))$ i poslat će je svim ostalim procesima, gdje je $T(i)$ jednaka vrijednosti logičkog sata C_i u trenutku slanja poruke.

Unutar svakog procesa nalazi se red poruka u kojem se svi zahtjevi za ulazak u kritični odsječak svrstavaju u skladu s relacijom “ \Rightarrow ” (*prethodi*) vremenskog uređenja u sustavu.

Protokol se izvodi u skladu s pet pravila koja se moraju ostvariti kao lokalne operacije pojedinih čvorova:

- a) Kada proces P_i zahtjeva ulazak u kritični odsječak on:
 - stavlja poruku $\text{zahtjev}(i, T(i))$ u svoj vlastiti red čekanja i
 - šalje tu poruku svim ostalim procesima.
- b) Kada proces P_j primi poruku $\text{zahtjev}(i, T(i))$, on:
 - uskladi svoj lokalni sat C_j u skladu s pravilima uspostave globalnog sata,
 - stavlja u svoj red čekanja poruku $\text{zahtjev}(i, T(i))$ te
 - šalje poruku $\text{odgovor}(i, T(j))$ procesu P_i , gdje je $T(j)$ jednako novoj vrijednosti logičkog sata C_j .
- c) Proces P_i smije ući u kritični odsječak:
 - kada se njegov vlastiti zahtjev nalazi na početku reda i
 - kada je proces P_i primio poruke odgovora svih ostalih procesa.
- d) Proces P_i obavlja izlazak iz kritičnog odsječka tako da:
 - ukloni iz svog reda svoj $\text{zahtjev}(i, T(i))$ i
 - pošalje svim ostalim procesima poruku $\text{izlazak}(i, T(i))$, gdje je $T(i)$ jednaka vrijednosti iz prvotno poslanog zahtjeva.
- e) Kada proces P_j primi poruku $\text{izlazak}(i, T(i))$, on iz svog reda čekanja uklanja zahtjev procesa P_i .

Za ulazak i izlazak iz kritičnog odsječka u sustavu se razmijeni $3 \cdot (N - 1)$ poruka, i to:

- ($N - 1$) poruka zahtjev ,
- ($N - 1$) poruka odgovor i
- ($N - 1$) poruka izlazak .

10.3.4. Protokol Ricarta i Agrawala

Ricart i Agrawala pokazali su da se ukupni broj poruka u sustavu s N čvorova može svesti na $2 \cdot (N - 1)$ poruku. To se pojednostavljenje protokola može postići tako da procesi šalju odgovore na primljene poruke samo onda ako ustanove da ne žele ulaziti u kritični odsječak ili ustanove da proces koji je postavio zahtjev ima pravo prvenstva.

Protokol se ostvaruje sljedećim pravilima:

a) Kada proces P_i zahtjeva ulazak u kritični odsječak, on:

- šalje poruku $\text{zahtjev}(i, T(i))$ svim ostalim procesima (gdje je $T(i)$ trenutačna vrijednost lokalnog sata u čvoru i).

b) Kada proces P_j primi poruku $\text{zahtjev}(i, T(i))$, on:

- uskladi svoj lokalni sat C_j u skladu s pravilima uspostave globalnog sata;
- šalje poruku $\text{odgovor}(j, T(i))$, ako ne želi ulaziti u kritični odsječak ili ako je zahtjev procesa P_j za ulazak došao kasnije;
- proces P_j , dakle, neće poslati odgovor ako postoji njegov zahtjev čija vremenska oznaka $(j, T(j))$ prethodi vremenskoj oznaci $(i, T(i))$.

c) Kada proces P_i primi odgovore svih ostalih čvorova, on smije ući u kritični odsječak.

d) Kada proces P_i izlazi iz kritičnog odsječka, on će poslati poruku $\text{odgovor}(j, T(i))$ svim procesima čiji zahtjevi kod njega čekaju na odgovor.

Za ulazak i izlazak iz kritičnog odsječka u sustavu se razmijeni $2 \cdot (N - 1)$ poruka, i to:

$(N - 1)$ poruka zahtjev i

$(N - 1)$ poruka odgovor.

PRIMJER 10.3.

Razmotrimo sustav s tri procesa (svaki u različitom čvoru) s početnim vrijednostima lokalnog logičkog sata: $C_1 = 11$, $C_2 = 21$ i $C_3 = 29$. Neka u početnom stanju sva tri procesa budu u nekriticnom dijelu. U trenutku t_1 proces P_1 traži ulaz u kritični odsječak. Protokol mu treba nakon razmjena poruka to i omogućiti jer ostali procesi nisu još ni tražili ulaz. Nakon što P_1 započne s radom u kritičnom odsječku, neka procesi P_2 i P_3 istovremeno zatraže ulaz u kritični odsječak. Promotrimo poruke koje će se razmijeniti u sustavu. Tablica 10.1. prikazuje razmijenu poruka i akcije pojedinih procesa u različitim vremenskim trenucima (intervalima).



Tablica 10.1. Primjer međusobnog isključivanja protokolom Ricarta i Agrawala

proces P_1	proces P_2	proces P_3
$C_1 = 11$	$C_2 = 21$	$C_3 = 28$
P_1 šalje zahtjev(1, 11) procesima P_2 i P_3		

(početno stanje)

(t_1) P_1 traži ulaz u K.O.

	P_2 prima zahtjev(1, 11), šalje odgovor(2, 11) i ažurira $C_2 = 22$	P_3 prima zahtjev(1, 11), ažurira $C_3 = 29$ i šalje odgovor(3, 11)
P_1 prima odgovor(2, 11) i odgovor(3, 11) te ulazi u K.O.		
	P_2 šalje zahtjev(2, 22) procesima P_1 i P_3	P_3 šalje zahtjev(3, 29) procesima P_1 i P_2
P_1 prima zahtjev(3, 29) ($C_1 = 30$) i zahtjev(2, 22) ($C_1 = 31$)	P_2 prima zahtjev(3, 29) i ažurira $C_2 = 30$, ali ne šalje odgovor	P_3 prima zahtjev(2, 22), ažurira $C_3 = 30$ i šalje odgovor(3, 22)
	P_2 prima odgovor(3, 22) i ažurira $C_2 = 31$ (i čeka još jedan odgovor)	
P_1 šalje odgovore: odgovor(1, 22) za P_2 i odgovor(1, 29) za P_3		
	P_2 prima odgovor(1, 22) i ažurira $C_2 = 32$, i ulazi u K.O.	P_3 prima odgovor(1, 29) i ažurira $C_3 = 31$, (i čeka još jedan odgovor)
	P_2 šalje odgovor(2, 29)	
		P_3 prima odgovor(2, 29), ažurira $C_3 = 32$ i ulazi u K.O.

 P_1 ulazi u K.O. $(t_2) P_2$ i P_3 traže ulaz u K.O.stanje: P_1 u K.O. P_2 čeka odgovor od P_1 P_3 čeka odgovore od P_1 i P_2 P_1 izlazi iz K.O. P_2 ulazi u K.O. P_2 izlazi iz K.O. P_3 ulazi u K.O.

Opisana pravila za ostvarenje protokola trebalo bi pretvoriti u lokalne funkcije svakog čvora koje se obavljuju nedjeljivo. To se može riješiti ili dodavanjem novih jezgrinih funkcija ili izgradnjom odgovarajućeg monitora.

PRIMJER 10.4.

Razmotrimo mogući način ostvarenja protokola izgradnjom prikladnog monitora. Podsetimo se da smo u našem modelu jezgre za ostvarenje monitora u poglavlju 6. uveli četiri jezgrine funkcije, i to:

```
Uči_u_monitor(M);
Izači_iz_monitora(M);
Uvrstiti_u_red_uvjeta(M,K);
Osloboditi_iz_reda_uvjeta(M,K).
```



Protokol se može ostvariti sa sljedeće četiri monitorske funkcije:

```
Postaviti_zahajev;
Prihvati_ogovor(j,T(j));
Prihvati_zahajev(j,T(j));
Prijaviti_izlazak;
```



Za njegovo je ostvarenje potrebno deklarirati jedan monitor, nazovimo ga **Monitor [i]** i jedan red uvjeta pridružen tom monitoru, tj. **Red_uvjeta[i,1]**.



Slika 10.9. Struktura podataka za primjer 10.2.

Unutar monitora predviđej ćemo strukturu podataka prema tablici 10.3.

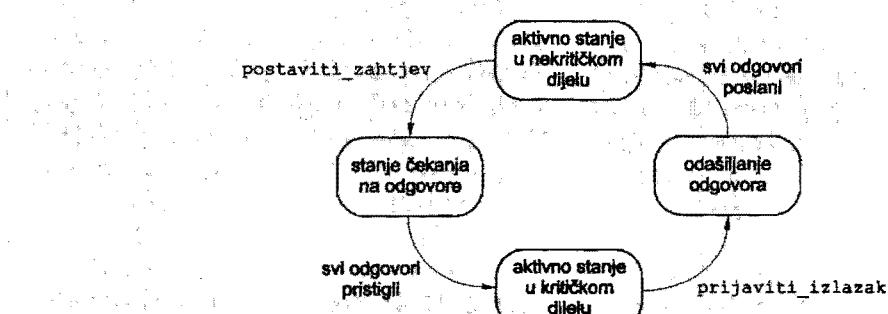
Tablica 10.2. Struktura podataka za primjer 10.3. za svaki čvor

lokalni_sat	lokalni logičk sat
odgovori_na_zahajev	brojilo odgovora na zahtjev
traži_ulaz	= 0 ako proces ne traži ulaz = 1 ako proces traži ulaz ili prolazi kroz kritični odsječak
odgođen_ogovor [1..N]	odgođen_ogovor [j]=1 ko je odgođen ogovor procesu j nače je odgođen_ogovor [j]=0

Dretva u procesu P_i koja će sudjelovati u protokolu međusobnog isključivanja može se nalaziti u tri stanja (prema slici 10.10.):

- aktivna u nekritičnom dijelu,
- u stanju čekanja na ulaz u kritični odsječak,
- aktivna u kritičnom odsječku.

Osim toga, pri izlasku iz kritičnog odsječka ona se može zadržati unutar monitora dok odašilje po rebne odgovore ostalim procesima.



Slika 10.10. Stanja dretve u raspodijeljenom sustavu

Dretva procesa koji želi ući u kritični odsječak pozvat će sljedeću monitorsku funkciju:

```

m-funkcija Postaviti_zahtjev {
    Uči_u_monitor(i);
    T(i) = lokalni_sat;
    odgovori_na_zahtjev = 0;
    traži_ulaz = 1;
    za j = 1 do N {
        ako je (i <> j) {
            poslati procesu P[j] poruku zahtjev(i, T(i));
        }
    }
    Uvrstiti_u_Red_uvjeta(i,1);
}
  
```

Dretva će ostati blokirana u Red_uvjeta(i,1) dok ne pristignu svi odgovori.

Kada kroz komunikacijski sustav pristigne poruka odgovor(j, T(j)), bit će pozvana monitorska funkcija Prihvati_odgovor.

```

m-funkcija Prihvati_odgovor(j, T(j)) {
    Uči_u_monitor(i);
    lokalni_sat = max(lokralni_sat, T(j)) + 1;
    odgovori_na_zahtjev++;
    ako je (odgovori_na_zahtjev == N - 1) {
        Osloboditi_iz_Reduvjeta(i, 1);
    }
    inače {
        Izači_iz_monitora(i);
    }
}
  
```

Ako u proces P_i dode poruka $zahtjev(j, T(j))$, komunikacijski sustav će je proslijediti monitoru pozivom monitorske funkcije $Prihvati_zahtjev(j, T(j))$.

Parametar j je indeks procesa koji je postavio zahtjev i $T(j)$ je prispjela vremenska oznaka.

```
m-funkcija Prihvati_zahtjev(j, T(j)) {
    Uči_u_monitor(i);
    lokalni_sat = max(lokralni_sat, T(j)) + 1;
    ako je ((traži_ulaz == 1)  $\wedge$  (T(j) > T(i)  $\vee$  (T(j) == T(i)  $\wedge$  j > i)) {
        odgođen_odgovor[j] = 1;
    }
    inače {
        poslati procesu P[j] poruku odgovor(i, T(j));
    }
    Izaći_iz_monitora(i);
}
```

Kada dretva procesa želi izaći iz kritičnog odsječka, onda ona mora poslati poruku $odgovor$ svim procesima koji kod nje čekaju na odgovor.

Dretva koja to obavlja nalazit će se unutar monitora sve dok svi odgovori ne budu odaslani.

```
m-funkcija Prijaviti_izlazak {
    Uči_u_monitor(i);
    traži_ulaz = 0;
    zaj = 1 doN {
        ako je (i  $\neq$  j  $\wedge$  odgođen_odgovor[j] == 1) {
            poslati procesu P[j] poruku odgovor(i, T(j));
            odgođen_odgovor[j] = 0;
        }
    }
    Izaći_iz_monitora(i);
}
```

Neka ciklička dretva unutar procesa P_i koja se izvodi ciklički djelomice u nekritičnom dijelu i u kritičnom odsječku imala bi uz ovako pripremljene monitorske funkcije sljedeći izgled:

```
dok je (1) {
    Postaviti_zahtjev;
    kritični odsječak;
    Prijaviti_izlazak;
    nekritični dio;
}
```



Analiza protokola ukazuje na sljedeće:

- svi su zahtjevi vremenski uređeni jer:
- svakom se novom zahtjevu pripisuje lokalno vrijeme lokalni sat čija je vrijednost veća od vrijednosti viđene u vremenskim oznakama od bilo kojeg zahtjeva koji je prije toga stigao u čvor;
- problem jednakih brojeva koji se mogu pojaviti zbog istovremenog generiranja zahtjeva u različitim čvorovima rješavaju se indeksom čvora (u skladu s pravilima logičkog sata);
- odlučivanje je potpuno raspodijeljeno;
- ne može se pojaviti potpuni zastoj;
- izgladnjivanje također nije moguće;
- algoritam se obavlja razmjenom $2 \cdot (N - 1)$ poruka.

PITANJA ZA PROVJERU ZNANJA 10

1. Navesti mehanizme za međusobnu komunikaciju između procesa na jednom računalu.
2. Kako mogu komunicirati procesi na udaljenim računalima?
3. Opisati mehanizam poziva udaljenih procedura (RPC).
4. Navesti mehanizme koji omogućuju međusobno isključivanje dretvi i procesa na jednom računalu.
5. Opisati centralizirani protokol međusobnog isključivanja u raspodijeljenim sustavima.
6. Opisati protokol s putujućom značkom.
7. Opisati lokalni i globalni logički sat.
8. Opisati raspodijeljeni Lamportov protokol.
9. Opisati protokol Ricarda i Agrawala.

11.1. Uvod

11.1.1. Uvodne napomene

Sigurnost računalnih sustava postaje sve važnija jer u današnjem svijetu *sve više korisnika na sve više načina* koristi *sve više informacija* koje su raspršene u raspodijeljenim sustavima. U takvom svijetu postoji sve veća opasnost od neovlaštene uporabe informacija, podmetanja krivih informacija ili uništavanja informacija. Zbog toga postaju sve zanimljiviji različiti zaštitni mehanizmi koji osiguravaju *sigurnost računalnih sustava*.

Sigurnosni zahtjevi ovise o vrsti informacija koje želimo zaštititi. Po važnosti sigurnosne zaštite može se načiniti približni redoslijed informacijskih sustava:

- vojni informacijski sustavi,
- bankovni informacijski sustavi,
- zdravstveni i bolnički informacijski sustavi,
- informacijski sustavi državnih institucija,
- informacijski sustavi osiguravajućih društava,
- poslovni informacijski sustavi gospodarskih subjekata.

Opća koncepcija sigurnosti ima svoje moralne i pravne aspekte koji se reguliraju zakonodavstvom i odgovarajućim kaznenim mjerama.

Mi ćemo se ovdje pozabaviti samo dijelom mogućih načina postizanja veće sigurnosti sustava.

Ugrožavanje sigurnosti računalnih sustava moguće je klasificirati na različite načine. Jedna od mogućih podjela sigurnosnih mehanizama je sljedeća:

- zaštita od vanjskih utjecaja,
- zaštita ostvarena sučeljem prema korisniku,
- unutarnji zaštitni mehanizmi,

- komunikacijski zaštitni mehanizmi.

Zaštita od vanjskih utjecaja

Vanjski utjecaji obuhvaćaju:

- mehaničko oštećenje naprava,
- oštećenje nastalo požarom ili poplavom,
- krađu uređaja i medija na kojima su informacije pohranjene.

Zaštitne mjere od tih utjecaja jednake su mjerama zaštite ostale imovine i ovdje se neće posebno razmatrati. One se svode na ograničavanje pristupa prostorima u koje su naprave smještene te na čuvanje kopija informacija na različitim sigurnim mjestima. Organizirana stražarska služba smije dopuštiti pristup samo povjerljivim ovlaštenim osobama.

Zaštita ostvarena sučeljem prema korisniku

Sučelje prema korisniku mora biti izvedeno tako da se kroz njega omogući uporaba računala samo ovlaštenim osobama i ne dopusti korištenje računala neovlaštenim osobama. Samo povjerljive osobe smiju upotrebljavati računalni sustav. One moraju unaprijed biti ovlaštene za pojedini način uporabe računalnog sustava. Pravo pristupa utvrđuje se provjerom identiteta, tj. utvrđivanjem vjerodostojnosti (autentičnosti) njegova identiteta. Korisnik se prilikom pristupa računalu predstavlja postupkom *identifikacije* (engl. *identification*).

Računalni sustav mora provesti postupak provjere identifikacije. Taj se postupak naziva *autentifikacijom*¹ (engl. *authentication*). Autentifikacija se provodi prilikom prijave za rad na sustavu (engl. *login*). Pritom se mora utvrditi autentičnost samo jedne strane, tj. korisnika. Korisnik, naime, ne mora provjeravati identitet računala. Međutim, u raspodijeljenim sustavima autentifikacija se često mora provoditi *dvostrano* jer obje strane u komunikaciji moraju dokazivati svoj identitet.

Nadalje, ako je riječ o postupku identifikacije ili autentifikacije, može se zaključiti i prema upitu u bazu podataka. Identifikacija može biti i složeniji proces od procesa autentifikacije ukoliko se pretražuje cijela baza korisnika. Primjerice, otisak prsta treba se usporediti sa svima u bazi podataka kako bi se identificirala osoba koja je dala samo otisak prsta, a nije se i predstavila. S druge strane, u postupku se autentifikacije samo provjerava ispravnost lozinke: nakon predstavljanja se otisak prsta uspoređuje samo s jednim podatkom u bazi.

Unutarnji zaštitni mehanizmi

Kada korisnik prođe postupak autentifikacije, on se "nalazi unutar sustava" i može početi upotrebljavati računalna sredstva. Unutarnji zaštitni mehanizmi, međutim, pojedinom korisniku moraju omogućiti pristup do pojedinih sredstava za koje on ima dopuštenje pristupa (za koje mu je pristup *autoriziran*). Mehanizmi *dopuštanja pristupa* (engl. *access control*) pojedinim sredstvima nazivaju se *autorizacijom* pristupa (engl. *authorization*).

¹ U hrvatskom se jeziku upotrebljava i neposredan prijevod s engleskog: *autentikacija*. Međutim, na osnovi tvorbe naziva identifikacija (dolazi od lat. *idem* – isti te ... fikacija od lat. glagola *facio* – činiti) može se načiniti kovanica autentifikacija (od grč. *authente s* – početnik, uzročnik i jednakog nastavka ... fikacija).

identifikacija = predstavljanje
 autentifikacija = identifikacija + verifikacija
 autorizacija = autentifikacija + provjera ovlasti, tj. provjera prava pristupa

Slika 11.1. Razlika između identifikacije, autentifikacije i autorizacije

U postupcima autorizacije uobičajilo se sljedeće nazivlje:

- *subjekti* u postupcima autorizacije jesu pojedini korisnici, odnosno njihovi procesi ili čak neke dretve unutar tih procesa;
- sva sredstva koja se zaštićuju jesu *objekti zaštite* (objekti su kako apstraktne tvorevine, kao što su procesi, strukture podataka, datoteke, tako i fizička sredstva, kao što su procesori, dijelovi spremničkog prostora ili pojedine naprave sustava);
- zaštitna pravila (engl. *protection rules*) moraju za svaki par subjekt-objekt odrediti pravo pristupa koje obuhvaća i način na koji se objekt smije upotrebljavati.

Komunikacijski zaštitni mehanizmi

U raspoloženim računalnim sustavima informacije se prenose raznovrsnim otvorenim i nesigurnim komunikacijskim putovima. Pristup do tih putova ne može se fizički zaštititi (to je posebice razumljivo za komunikacijske putove ostvarene radiovezama). Prema tome, svaki neprijateljski nastrojeni *napadač*, možemo ga zvati *uljezom* (engl. *attacker*, *intruder*), može vrlo lako narušiti sigurnost raspoloženog sustava. Zbog toga u raspoloženim sustavima komunikacijski zaštitni mehanizmi postaju najvažniji oblik ostvarenja sigurnosti.

U raspoloženom sustavu sve se informacije prenose u obliku poruka. Osnovni problem komunikacijskih zaštitnih mehanizama jest zaštita poruka. Pokazuje se da je najdjelotvornija zaštita poruka njihovo *kriptiranje*. Većina se narušavanja sigurnosti može razmotriti na mehanizmu protoka poruka od *izvorišta* do *odredišta*. Prema tome, najprije se može razmotriti modele zaštite komunikacijskog kanala od izvorišta do odredišta. Ovakvi se modeli mogu primijeniti i na ostale oblike zaštite jer se međusobno djelovanje subjekata i objekata (primjerice, u postupcima autentifikacije i autorizacije) može poistovjetiti s razmjenom poruka.

11.1.2. Vrste napada na sigurnost

Vrste napada na sigurnost mogu se najbolje ilustrirati modelom u kojem je izvorište informacija povezano s odredištem komunikacijskim kanalom.

Izvorište i odredište mogu se nalaziti:

- u jednom računalu (primjerice, pri razmjeni sadržaja između radnog i vanjskog spremnika ili pri prenošenju podataka iz jedne datoteke u drugu) ili
- u različitim računalima raspoloženog sustava (pri čemu su u komunikacijski kanal uključene telekomunikacijske naprave i sustavi).

Pojedine vrste napada uzrokuju različite oblike narušavanja sigurnosti.

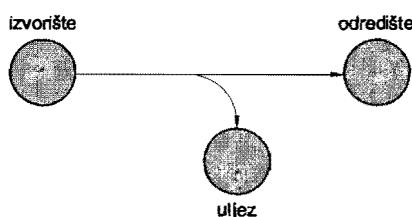
U normalnim uvjetima komunikacijski kanal prenosi nesmetano informacije iz izvorišta u odredište.



Slika 11.2. Prijenos informacija

1. Prisluškivanje

Najjednostavniji način napada na sigurnost jest *prisluškivanje* (engl. *eavesdropping*) ili *presretanje* (engl. *interception*). Napadač može čitati pakete koji su namijenjeni nekom drugom te na taj način doći do osjetljivih informacija. Ovo je *pasivni napad* (engl. *passive attack*) jer uljez ne djeluje aktivno na informacije.



Slika 11.3. Prisluškivanje

Prisluškivanjem se djeluje na *povjerljivost* (engl. *confidentiality*), odnosno *tajnost* (engl. *secrecy*) informacija.

U ostalim načinima napada uljez mora djelovati na informacije te se oni nazivaju *aktivnim napadima* (engl. *active attacks*).

2. Prekidanje

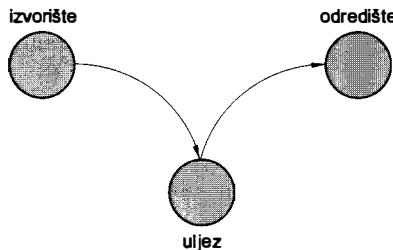
Uljez može djelovati tako da prekine komunikacijski kanal između izvorišta i odredišta. *Prekidanje* (engl. *interruption*) narušava *raspoloživost* (engl. *availability*) informacija.



Slika 11.4. Prekidanje

3. Promjena sadržaja poruka

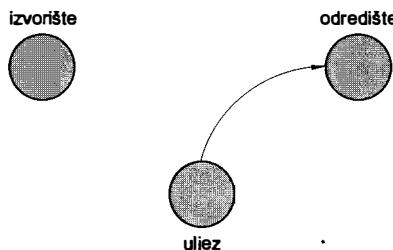
Uljez može prekinuti komunikacijski kanal i, lažno se predstavljajući kao izvořište, promijeniti sadržaj poruke. Promjena sadržaja (engl. *modification*) narušava *besprijekornost* ili *integritet* (engl. *integrity*) informacija.



Slika 11.5. Promjena sadržaja poruka

4. Izmišljanje poruka

Uljez može uspostaviti komunikacijski kanal s odredištem i, lažno se predstavljajući kao izvořište, slati mu izmišljene poruke ili snimljene stare poruke. *Izmišljanje* poruka (engl. *fabrication*) narušava, kao i promjena sadržaja, *besprijekornost* ili *integritet* (engl. *integrity*) informacija.



Slika 11.6. Izmišljanje poruka

5. Lažno predstavljanje

Napadač se predstavlja kao neki drugi korisnik (primjerice, provalivši na tuđi korisnički račun) ili napadač može namjestiti računalo tako da se ono pretvara kao da je neko drugo i tako vara druga računala kojima se na taj način lažno predstavlja.

6. Poricanje (engl. repudiation)

Nakon što je poslana poruka, korisnik se može predomisliti i poricati autorstvo poruke te tvrditi da se to netko lažno predstavio i poslao poruku u njegovo ime.

11.1.3. Sigurnosni zahtjevi

Na temelju opisanih mogućih napada na sigurnost proizlazi da se sigurnost računalnih sustava zasniva na ispunjavanju triju osnovnih sigurnosnih zahtjeva. To su:

- **Povjerljivost ili tajnost**

Informacije u sustavu smiju biti pristupačne samo ovlaštenim korisnicima.

- **Raspoloživost**

Informacije moraju uвijek biti na raspolaganju ovlaštenim korisnicima usprkos mogućim neočekivanim i nepredvidljivim događajima kao što su primjerce nestanak struje, prirodna nepogoda, nesreća ili zlonamjerni napad.

- **Bespriјekornost**

Informacije u sustavu mogu mijenjati samo za to ovlašteni korisnici. Dakle, treba se osigurati jamstvo da su informacije poslane, primljene ili pohranjene u izvornom i nepromijenjenom obliku.

- **Autentičnost**

Ovlašteni se korisnici moraju jednoznačno moći prepoznati. Jednoznačno prepoznavanje ovlaštenih korisnika obavlja se već opisanim postupkom *autentifikacije*.

- **Autorizacija**

Dodatno, za osiguranje bespriјekornosti autentificiranim se ovlaštenim korisnicima postupkom *autorizacije* dopušta pristup samo do nekih sadržaja.

- **Neporecivost**

Posebni oblik narušavanja sigurnosti mogao bi nastati tako da ovlašteni korisnik opovrgava poruku koju je ranije poslao tvrdeći da je ona izmišljotina uljeza. Zbog toga se kao sigurnosni zahtjev postavlja i mogućnost zaštite od opovrgavanja, odnosno *neporicanje* (engl. *nonrepudiation*).

11.1.4. Utjecaj pojedinih komponenti računalnih sustava na sigurnost

Pogledajmo kako pojedine komponente računalnih sustava mogu utjecati na ispunjavanje ili neispunjavanje navedenih sigurnosnih zahtjeva.

Sklopovlje računala

Sklopovlje računala može najviše utjecati na *raspoloživost* informacija. Sklopovlje je najviše podložno vanjskim utjecajima pa se unapređenje sigurnosti može postići odgovarajućim održavanjem sustava i ograničavanjem pristupa.



Povećanje raspoloživosti može se postići izgradnjom sustava sa zalihošću koji može djelotvorno raditi i u slučaju kvarova pojedinih njegovih dijelova (engl. *fault tolerant system*).

Programi

Programi mogu utjecati na sva tri osnovna sigurnosna zahtjeva. Nedopuštena uporaba ili krađa programa može imati utjecaja na *tajnost* informacija, a neovlaštena modifikacija programa može djelovati na *besprijeckornost* i *raspoloživost*.

Najpoznatiji način neovlaštenog djelovanja na programe jesu napadi *virusima*. Virusi su programski odsječci koji se komunikacijom ili razmjenom spremničkih medija unose u računalni sustav. Izgrađeni su tako da se sami pridodaju u neke postojeće programe i pri njihovu pokretanju djeluju štetno.

Neki su virusi izgrađeni tako domišljato da inficiraju mnogo programa u računalnom sustavu. Jedanput zaraženo računalo teško je vratiti u normalno radno stanje te je stoga mnogo djelotvornije djelovati preventivno. Preventivno se djelovanje svodi na uporabu samo legalno nabavljenih programa i uspostavu komunikacije samo s pouzdanim partnerima, te uporabu zaštitnih protuvirusnih programa.

Drugi, manje razvikanji zločudni programi jesu programski *crvi* (engl. *worms*). Za razliku od virusa, crvi su cjeloviti programi koji sami sebe kroz komunikacijsku mrežu prenose s računala na računalo i pritom djeluju destruktivno.

Treći oblik zločudnih programa jesu *trojanski konji*. Trojanski konj je program koji obavlja neki koristan posao, ali mu je pridodata neka funkcija koja štetno djeluje. Primjerice, trojanski konj može djelovati tako da uspostavi slobodan pristup datotekama koje su inače zaštićene.

Postoji i četvrti oblik zločudnih programa koji svoje djelovanje skrivaju na poseban način – tako da nadomješćuju sustavske procese i podatke, odnosno datoteke operacijskog sustava (engl. *rootkit*).

Podaci

Podaci smješteni u datoteke ili baze podataka podložni su svima trima osnovnim oblicima narušavanja sigurnosti. *Raspoloživost* može biti ugrožena neovlaštenim brisanjem datoteka, *tajnost* neovlaštenim čitanjem sadržaja datoteka, a *besprijeckornost* neovlaštenom promjenom sadržaja. Povećanje se sigurnosti može postići kontrolom pristupa datotekama, primjerice autorizacijom korisnika za njihovo korištenje.

Komunikacije

Već je uvodno naglašeno da su komunikacije sa stanovišta sigurnosti najosjetljiviji dio računalnih sustava. U komunikacijskom sustavu mogu nastati sva tri oblika narušavanja sigurnosti.

11.2. Osnove kriptografije

Iz dosadašnjih je razmatranja vidljivo kolika je važnost uspostave sigurnosnih mehanizama u umreženim računalnim sustavima. Moglo bi se reći da mnoga korisna ostvarenja uporabe računalnih mreža u praktički svim područjima ljudske djelatnosti dobrim dijelom ovise o razvitu pouzdanih zaštitnih mehanizama koji će osigurati primjerenu sigurnost sustava. Pokazat će se da se svi sigurnosni zahtjevi, osim raspoloživosti, mogu zadovoljiti uvođenjem kriptiranja sadržaja koji će razmjenjivati u umreženim računalnim sustavima.

S obzirom na to da je u komunikacijskom sustavu nemoguće spriječiti prisluškivanje podataka, pokazalo se razumnim načinu podatke nerazumljivima neovlaštenim uljezima. Podaci koji u svom izvornom obliku predstavljaju neku korisnu informaciju mogu se postupkom *kriptiranja* prevesti u oblik u kojem se ta informacija više ne prepoznaće.

Kriptiranje

Budući da su počeci kriptiranja povezani s prenošenjem pisanih informacija u obliku tekstova, u kriptografskoj se terminologiji izvorni oblik podataka naziva *razgovjetnim ili jasnim tekstrom* (engl. *plaintext, cleartext*).

Postupkom *kriptiranja* (engl. *encrption, enciphering*) jasni tekst se prevodi u *kriptirani tekst* (engl. *ciphertext*). Obrnuti postupak prevođenja kriptiranog teksta u jasni tekst naziva se *dekriptiranjem* (engl. *decryption, deciphering*).

U današnje se vrijeme, kada nam na raspolaganju stoje vrlo moćna računala, ne mogu primjenjivati naivne stare metode kriptiranja koje su se, u načelu, zasnivale na zamjeni znakova prema nekim složenim pravilima. Današnji kriptografski postupci su *parametarske matematičke funkcije*, odnosno *algoritmi* kojima se *nizovi bitova jasnog teksta* preračunavaju u *nizove bitova kriptiranog teksta* i obrnuto.

Funkcija kriptiranja može se zapisati u sljedećem obliku:

$$C = E(P, K_E),$$

gdje je:

P – jasni tekst,

C – kriptirani tekst,

E – funkcija kriptiranja,

K_E – parametar ili ključ kriptiranja (engl. *encryption key*).

Funkcija dekriptiranja neka je:

$$P = D(C, K_D),$$

gdje je:

D – funkcija dekriptiranja,

K_D – parametar ili ključ dekriptiranja (engl. *decryption key*).

Funkcija dekriptiranja mora biti inverzna funkciji kriptiranja, tako da vrijedi:

$$P = D(E(P, K_E), K_D),$$

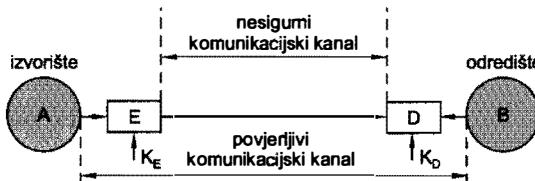
tj. dekriptiranjem kriptiranog jasnog teksta dobiva se opet jasni tekst.

Ako se kriptiranje obavi u izvořištu i dekriptiranje u odredištu, onda se kriptiranjem komunikacijski kanal štiti od prisluskivanja te se može postići povjerljivost informacija.

Kriptosustav, povjerljivi komunikacijski kanal

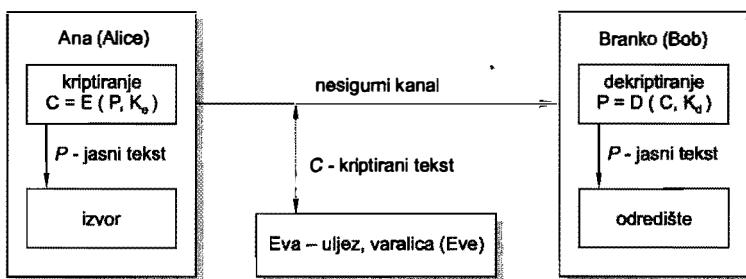
Funkcije kriptiranja E i dekriptiranja D čine *kriptosustav*. Korisnik A (umjesto korisnik A ili strana A uobičajeno se u literaturi o kriptiranju koristi ime *Alica*, a mi ćemo ovdje koristiti ime *Ana*) izvořište je informacija. On obavlja kriptiranje razgovijetnog teksta svojim ključem K_E .

Kriptirana poruka šalje se preko komunikacijskog kanala koji nije zaštićen od uljeza i zbog toga ga nazivamo *nesigurnim komunikacijskim kanalom* (engl. *unsecure channel*).



Slika 11.7. Sigurni i nesigurni komunikacijski kanal

Korisnik B (kojeg u literaturi obično nazivaju *Bob*, a mi ćemo koristiti ime *Branko*) nakon dekriptiranja poruke svojim ključem K_D dobiva razgovijetni tekst. Međutim, uljez koji ne zna ključ dekriptiranja poruku neće razumjeti. Uz pretpostavku da nitko osima Branka ne zna ključ dekriptiranja, kriptosustav transformira nesigurni komunikacijski kanal u *povjerljivi informacijski kanal* (engl. *trusted channel*) između *Ane* i *Branka*.



Slika 11.8. Zaštita povjerljivih informacija od nepovlašnjeg korisnika koji prisluskuje otvoreni kanal

Zaštita povjerljivih informacija provodi se zbog opasnosti da nepovlašnji korisnik (koji se u literaturi obično naziva *Eva*², u našem slučaju *Eva*) prisluskuje otvoreni komunikacijski kanal.

² prisluskivač, engl. *eavesdroper*

Današnji kriptosustavi zasnivaju se na postupcima koji se efikasno mogu izvoditi na računalima, i to bilo sklopovski ili programski. Ti se postupci zasnivaju na algoritmima koji su u pravilu općepoznati, ali s ključevima koji imaju *vrlo velik broj mogućih vrijednosti*, što omogućuje stvaranje vrlo velikog broja različitih oblika kriptiranog teksta. Štoviše, u modernoj kriptografiji poštuje se Kerchoffov zakon koji kaže da: *kriptosustav mora biti siguran i onda kada su sve informacije o kriptosustavu, osim tajnog ključa, javno poznate.*³

Ne zaboravimo da je osnovni razlog kriptiranja stvaranje nerazgovijetnog teksta za sve uljeze koji ne znaju ključ dekriptiranja. Prema tome, "razbijanje" kriptiranja svodi se na pronalaženje ključa dekriptiranja. Jasno je da je postupak pronalaženja toga ključa teži ako on može poprimit veliki broj vrijednosti. Dobrota kriptosustava određena je težinom otkrivanja ključa dekriptiranja K_D . Pri utvrđivanju dobrote kriptosustava mora se voditi računa o tome kako napadač ili uljez (kojeg u ovom slučaju nazivaju kriptoanalitičarom) pokušava otkriti ključ K_D .

Kriptoanalitičar može pokušavati otkriti ključ dekriptiranja uz poznavanje:

- samo kriptiranog teksta;
- samo ograničene količine kriptiranog i razgovijetnog teksta;
- neograničene količine kriptiranog i njemu pripadajućeg razgovijetnog teksta.

Razumljivo je da je posljednje spomenuti način za kriptoanalitičara najpovoljniji i stoga se tim načinom i ispituje otpornost kriptosustava. Smatra se da je najbolji način ispitivanja kriptosustava njegova uporaba. Ako određeno vrijeme nije potvrđen ni jedan uspješan pokušaj njegova razbijanja, on se može smatrati razmjerno sigurnim.

Danas su u uporabi dva osnovna oblika kriptosustava:

- simetrični kriptosustavi i
- asimetrični kriptosustavi.

Asimetrični kriptosustavi imaju različite ključeve kriptiranja K_E i dekriptiranja K_D i mogu se opisati već navedenim izrazima:

$$\begin{aligned} C &= E(P, K_E), \\ P &= D(C, K_D), \\ P &= D(E(P, K_E), K_D). \end{aligned}$$

U simetričnim kriptosustavima ključ kriptiranja K_E jednak je ključu dekriptiranja K_D . Zajednički se ključ može označiti jednim simbolom K . Prema tome, za takav sustav vrijedi:

$$\begin{aligned} C &= E(P, K), \\ P &= D(C, K), \\ P &= D(E(P, K), K). \end{aligned}$$

³ Prvi je taj princip uveo A. Kerchoff još u 19. stoljeću.

11.3. Simetrični kriptosustavi

Simetrični kriptosustavi koriste isti ključ K i za kriptiranje i za dekriptiranje. Oni se, u načelu, zasnivaju na uporabi logičke operacije *isključivo ILI* (engl. *exclusive OR* ili kraće: *XOR*). Označimo li dvije moguće vrijednosti logičke varijable s 0 i 1, operacija *XOR* dat će kao rezultat vrijednost 1 onda kada samo jedna od ulaznih varijabli ima vrijednost 1.

Prepostavimo da je jasni tekst P kodiran tako da se sastoji od n bitova i da imamo ključ K koji također ima n bitova. Pojedine bitove od P i K možemo promatrati kao logičke varijable P_i i K_i .

Kada na sve parove bitova primijenimo operaciju *XOR*, za koju možemo upotrijebiti simbol \oplus , dobit ćemo niz bitova C_i kriptiranog teksta. Operacijom *XOR* primjenjenom na bitove kriptiranog teksta i jednake bitove ključa dobit ćemo bitove izvornog razgovijetnog teksta:

P_i	K_i	$C_i = P_i \oplus K_i$
0	0	0
0	1	1
1	0	1
1	1	0

C_i	K_i	$P_i = C_i \oplus K_i$
0	0	0
1	1	0
1	0	1
0	1	1

Prema tome, funkcija kriptiranja jednaka je funkciji dekriptiranja i obavlja se s istom vrijednošću ključa. Ako prepostavimo da operacija \oplus djeluje paralelno na sve bitove teksta i ključa, možemo pisati:

$$C = P \oplus K, \quad P = C \oplus K,$$

odnosno:

$$P = (P \oplus K) \oplus K.$$

Jedan od mogućih načina kriptiranja bio bi, dakle, taj da se kao ključ upotrijebi neki dogovoreni tekst s odgovarajućim brojem znakova, tj. odgovarajućom duljinom ključa. Takav kriptosustav naziva se *jednokratnom bilježnicom* (engl. *One Time Pad*).

Ako razgovijetni tekst koji želimo kriptirati ima više znakova, onda bi se on morao podijeliti na *blokove* koji su jednakini duljini ključa i kriptirati svaki blok posebno. Ključ kriptiranja smiju znati samo Ana i Branko i nitko drugi. Ključ, dakle, mora biti *tajan* i mora se nekim sigurnim komunikacijskim kanalom (recimo, osobnim prenošenjem) razmijeniti.

Ako u nekom sustavu N sudionika želi uspostavljati povjerljive komunikacijske kanale, onda bi za sve parove u komunikaciji trebalo na siguran način razmijeniti $N(N - 1)/2$ tajnih ključeva. Svaki korisnik mora imati $(N - 1)$ tajnih ključeva.

Vidjet ćemo kasnije da postoje protokoli kojima se uspostavljanje tajnih ključeva može obaviti i kroz nesigurne komunikacijske kanale.

11.3.1. Data Encryption Standard (DES)

DES je najrasprostranjeniji simetrični kriptosustav. On je normiran u S.A.D., a međunarodna normizacija provedena je kroz neke institucije *ISO*. Zasniva se na permutaciji bitova i operaciji *XOR*. Postoji nekoliko njegovih varijanti. Ovdje ćemo kratko opisati osnovnu varijantu.

U *DES* sustavu kriptira se blokove duljine 64 bita (tj. osam bajtova). Ključ K ima 56 bitova i iz njega se određuje šesnaest parametara K_1, K_2, \dots, K_{16} .

Postupak se provodi na sljedeći način:

- Najprije se 64 bita jednog bloka razgovijetnog teksta P permutira na specificirani način i dobiva $IP(P)$,
- Rezultat permutacije podijeli se na dvije polovine L_0 i R_0 svaka s 32 bita, što možemo pisati kao:

$$L_0 R_0 = IP(P).$$

- Zatim se obavlja 16 koraka u kojima se, uz $i = 1, 2, \dots, 16$, obavlja sljedeće operacije:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, K_i), \end{aligned}$$

pri čemu funkcija $f(R_{i-1}, K_i)$ obavlja “preslagivanje” bitova u R_{i-1} ovisno o parametru K_i .

- Na kraju se obavlja inverzna permutacija od IP , koju možemo označiti s $IP^{-1}(R_{16} L_{16})$.

Kriptirani blok od 64 bita možemo, dakle, zapisati kao:

$$C = IP^{-1}(R_{16} L_{16}).$$

Skraćeno čitavu funkciju kriptiranja možemo zapisati kao:

$$C = DES(P, K).$$

Parovi simetričnih parametara K_i i K_{16-i+1} se iz ključa K određuju tako da uvjetuju međusobno inverzna preslagivanja bitova. Dekriptiranje se stoga obavlja na jednak način kao i kriptiranje s tim da se u i -toj iteraciji upotrebljava parametar K_{16-i+1} . Dekriptiranje se može opisati inverznom funkcijom, odnosno:

$$P = DES^{-1}(C, K).$$

Postupci kriptiranja i dekriptiranje opisani su detaljno u literaturi i mogu se programski ostvariti. Isto tako, postoje deseci proizvođača čipova za sklopovsko kriptiranje i dekriptiranje. Brzina sklopovskog kriptiranja doseže od 10^9 bit/s do 10^{10} bit/s. Programski se može ostvariti brzina kriptiranja koja je reda veličine 10^5 bit/s.

Ključ od 56 bitova može poprimiti $2^{56} = 7.2 \cdot 10^{16}$ vrijednosti. To je razumno velik broj ključeva tako da je u normalnim uvjetima vrlo teško otkriti upotrijebjeni ključ. Međutim, postoje pokušaji koji su uspjeli otkriti 56-bitovni ključ u roku od nekoliko sati. Zbog toga je uveden novi sustav s duljinom ključeva od 128 bitova, koji omogućuje odabir jednog od $3.4 \cdot 10^{38}$ različitih ključeva.

Treba naglasiti da je *DES* kriptiranje zaštićeno patentom i, osim toga, podložno je izvoznim dozvolama američke vlade, tako da mu je uporaba time ograničena. Zbog toga su u uporabi inačice toga postupka koje se mogu slobodno distribuirati. Pojedini isporučitelji takvih kriptosustava često svoja rješenja tretiraju kao poslovnu tajnu.

11.3.2. Utrostručeni DES, 3DES

Razbijanje *DES* kriptiranja može se otežati uporabom višestrukog *DES* kriptiranja. Umjesto jednog ključa K upotrebljavaju se tri ključa: K_1 , K_2 i K_3 (ključ K je ukupne duljine od 168 bita). Kriptiranje se obavlja u tri koraka:

$$\begin{aligned} C_1 &= DES(P, K_1), \\ C_2 &= DES^{-1}(C_1, K_2) = DES^{-1}(DES(P, K_1), K_2), \\ C &= DES(C_2, K_3) = DES(DES^{-1}(DES(P, K_1), K_2), K_3). \end{aligned}$$

Funkcija kriptiranja *3DES* glasi:

$$3DES(P, K_1, K_2, K_3) = DES(DES^{-1}(DES(P, K_1), K_2), K_3).$$

Dekriptiranje se obavlja u sljedeća tri koraka:

$$\begin{aligned} P_1 &= DES^{-1}(C, K_3), \\ P_2 &= DES(P_1, K_2) = DES(DES^{-1}(C, K_3), K_2), \\ P &= DES^{-1}(P_2, K_1) = DES^{-1}(DES(DES^{-1}(C, K_3), K_2), K_1). \end{aligned}$$

Funkcija je dekriptiranja *3DES*, dakle, ovakva:

$$3DES^{-1}(C, K_1, K_2, K_3) = DES^{-1}(DES(DES^{-1}(C, K_3), K_2), K_1).$$

Ovakav način trostrukog kriptiranja omogućuje da se *3DES* može upotrijebiti kao *DES* ako se sva tri ključa odaberu tako da budu jednaka. Postoji i inačica *3DES* koja koristi duljinu ključa od 112 bita (prvi i zadnji ključ su jednaki).

11.3.3. Izbijeljeni DES, DESX

Jedan od načina za otežavanje razbijanja *DES* kriptosustava tzv. je bijeljenje teksta (engl. *whitening*). Uz 56-bitovni ključ K_1 kriptiranja upotrebljavaju se još dva 64-bitovna ključa K_2 i K_3 koji "izbijeljuju" blokove tekstova.

Prije kriptiranja blokovi razgovijetnog teksta duljine 64 bita podvrgavaju se *XOR* operaciji s ključem K_2 , a nakon *DES* kriptiranja obavlja se dodatna *XOR* operacija s ključem K_3 .

Kriptiranje se, dakle, obavlja ovako:

$$\begin{aligned}C_1 &= P \oplus K_2, \\C_2 &= DES(C_1, K_1) = DES(P \oplus K_2, K_1), \\C &= C_2 \oplus K_3 = DES(P \oplus K_2, K_1) \oplus K_3.\end{aligned}$$

Prema tome, funkcija kriptiranja je:

$$DESX(P, K_1, K_2, K_3) = DES(P \oplus K_2, K_1) \oplus K_3.$$

Dekriptiranje se obavlja na sljedeći način:

$$\begin{aligned}P_1 &= C \oplus K_3, \\P_2 &= DES^{-1}(P_1, K_1) = DES^{-1}(C \oplus K_3, K_1), \\P &= P_2 \oplus K_2 = DES^{-1}(C \oplus K_3, K_1) \oplus K_2.\end{aligned}$$

Funkcija dekriptiranja, dakle, izgleda ovako:

$$DESX^{-1}(C, K_1, K_2, K_3) = DES^{-1}(C \oplus K_3, K_1) \oplus K_2.$$

11.3.4. Kriptosustav IDEA

Kriptosustav *IDEA* (od engl. *International Data Encryption Algorithm*) nastao je početkom devedesetih godina i poprimio svoj današnji oblik 1992. godine. Kao i u *DES* kriptosustavu kriptiraju se blokovi od 64 bita (8 bajtova). Pri kriptiranju se blok dijeli na četiri podbloka od po 16 bitova.

Algoritam se provodi u devet koraka. U prvih se osam koraka s četiri podbloka i šest 16-bitovnih potključeva obavljaju sljedeće operacije:

- isključivo ILI (*XOR*);
- zbrajanje po modulu 2^{16} ;
- množenje po modulu $2^{16} + 1$.

Deveti je korak jednostavniji i u njemu se upotrebljavaju samo četiri potključa i ne upotrebljava se operacija *XOR*. Ključ ima duljinu od 128 bitova. Iz tog se ključa mora odrediti ukupno $8 \cdot 6 + 4 = 52$ potključa duljine 16 bitova.

Potključevi se određuju tako da se iz izvornog ključa dijeljenjem na skupine od 16 bitova dobije prvih osam ključeva. Nakon toga se 128-bitovni ključ rotira u lijevo za 25 mesta i ponovno dijeli na sljedećih osam 16-bitovnih ključeva. Taj se postupak ponavlja još šest puta s tim da se nakon zadnje rotacije uzimaju samo četiri ključa. Na taj način dobivamo ukupno $6 \cdot 8 + 4 = 52$ potključa. Označit ćemo potključeve s K_{ij} , gdje je:

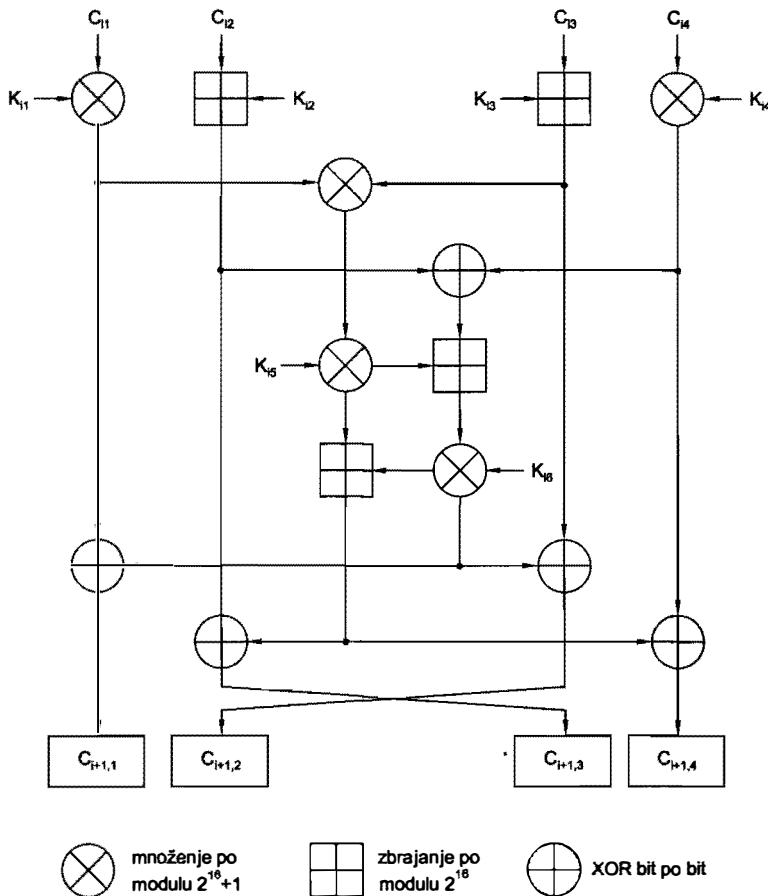
i – indeks koraka ($i = 1, 2, \dots, 9$),

j – indeks potključa u koraku ($j = 1, 2, \dots, 6$ za prvih osam koraka te $j = 1, 2, \dots, 4$ za posljednji deveti korak).

Označimo 16-bitovne podblokove 64-bitovnog bloka razgovijetnog teksta s $P_{01}, P_{02}, P_{03}, P_{04}$, a pripadne izlazne podblokove i -tog koraka s C_{i1}, C_{i2}, C_{i3} i C_{i4} , s tim da je:

$$C_{01} = P_{01}, C_{02} = P_{02}, C_{03} = P_{03}, C_{04} = P_{04}.$$

Slika 11.9. prikazuje kako se obavlja prvih osam koraka.



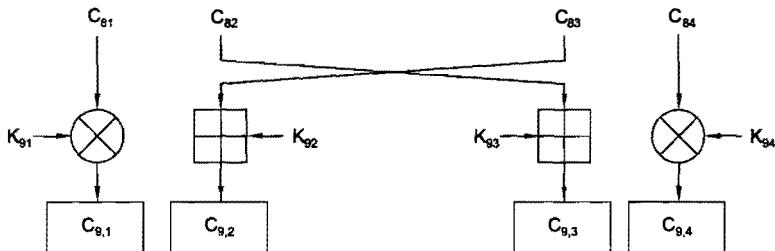
Slika 11.9. Prvih osam koraka algoritma IDEA

U devetom koraku nema operacija XOR .

Kriptirani 64-bitovni blok dobiva se nadovezivanjem 16-bitovnih blokova:

$$C = C_{91}C_{92}C_{93}C_{94}.$$

Dekriptiranje se obavlja jednakim postupkom s tim da se potključevi izračunavaju obrnutim redoslijedom i čine aditivne ili multiplikativne inverze potključeva kriptiranja. Brzina



Slika 11.10. Deveti korak algoritma IDEA

programskog kriptiranja IDEA kriptiranja dvostruko je brža od DES kriptiranja. Brzina je reda veličine 10^9 b/s. Postoje i sklopovska rješenja kriptiranja s brzinama reda veličine od 10^{11} b/s.

11.3.5. Napredni kriptosustav AES

Na natječaju za napredni kriptosustav (*AES – Advanced Encryption Standard*) koji je raspisao 1997. godine američki institut za normiranje i tehnologiju (*National Institute of Standards and Technology, NIST*) odabran je 2000. godine algoritam *Rijndael*. U međuvremenu je 3DES (*Triple DES*) proglašen kao privremen standard. Na natječaj su se mogli prijaviti samo algoritmi sa sljedećim svojstvima:

- simetrični blok algoritmi s javnim izvornim tekstrom programa,
- blok podataka koji se kriptira minimalne je veličine 128 bitova i
- veličina ključa od 128, 192 i 256 bitova.

Na prvom skupu, koji je nazvan *AESI*, NIST je objavio 15 prihvaćenih kandidata: *CAST-256*, *CRYPTON*, *DEAL*, *DFC*, *E2*, *FROG*, *HPC*, *LOKI97*, *MAGENTA*, *MARS*, *RC6TM*, *Rijndael*, *SAFER⁺*, *Serpent* te *Twofish*. Na drugom skupu (*AES2*) izbor je sužen na sljedećih pet kandidata: *MARS*, *RC6TM*, *Rijndael*, *Serpent* te *Twofish*. Svi su oni podjednako dobri pa, kada bi pobjednik bio kompromitiran, na raspolaganju su ostali finalisti.

Konačno polje $GF(2^8)$

Napredni simetrični kriptosustav (*AES*) koristi konačno Galoisovo polje $GF(2^8)$. Elementi toga polja su polinomi oblika $a_7x^7 + a_6x^6 + \dots + a_1x + a_0$, $a_i \in \{0, 1\}$. Odnosno, svaki bajt $a_7a_6a_5a_4a_3a_2a_1a_0$ (niz od 8 bitova) predstavljen je odgovarajućim polinomom. Nad elementima konačnog polja $GF(2^8)$ definirane su operacije zbrajanja i množenja. Zbrajanje je, zapravo, logička binarna operacija *isključivo ILI*. Množenje je binarno množenje polinoma modulo fiksni ireducibilni polinom $g(x) = x^8 + x^4 + x^3 + x + 1$.

PRIMJER 11.1.**Primjer množenja u $GF(2^8)$**

$$57_H \cdot 83_H = C1_H,$$

odnosno:

$$57_H = 01010111_2 \equiv x^6 + x^4 + x^2 + x + 1$$

$$83_H = 10000011_2 \equiv x^7 + x + 1$$

$$\begin{array}{r} (x^6 + x^4 + x^2 + x + 1) \cdot (x^7 + x + 1) = \\ \hline x^{13} + x^{11} + x^9 + x^8 + x^7 \\ \oplus \quad \quad \quad x^7 \quad + x^5 \quad + x^3 + x^2 + x \\ \hline x^{13} + x^{11} + x^9 + x^8 \quad + x^6 + x^5 + x^4 + x^3 \quad + 1 \end{array}$$

Još samo preostaje pronaći ostatak pri dijeljenju dobivenog rezultata s fiksnim irreducibilnim polinomom $g(x)$:

$$\begin{array}{r} (x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1) : (x^8 + x^4 + x^3 + x + 1) = x^5 + x^3 \\ x^{13} \quad + x^9 + x^8 + x^6 + x^5 \\ \hline x^{11} \quad \quad \quad + x^4 + x^3 + 1 \\ x^{11} \quad + x^7 + x^6 \quad + x^4 + x^3 \\ \hline x^7 + x^6 \quad + 1 \equiv C1_H \end{array}$$

Navedeni postupak nije prikladan za djelotvorno sklopovsko ostvarenje množenja. Međutim, u postupku množenja dvaju polinoma može se iskoristiti sljedeće svojstvo: množenje nekog polinoma $b(x)$ s polinomom $x \equiv 02_H$ zapravo je posmak polinoma $b(x)$ za jedan bit u lijevo. Ako prilikom posmaka dođe do preliva, tada se od rezultata mora oduzeti već spomenuti irreducibilni polinom $g(x) \equiv 11B_H$, što se opet svodi na logičku operaciju *isključivo ILI*. Množenje nekog polinoma s polinomom x neka obavlja funkcija *xputa()*.

PRIMJER 11.2.**Primjer množenja u $GF(2^8)$ uz pomoć funkcije *xputa()***

$$57_H \cdot 83_H = 57_H \cdot (01_H + 02_H + 80_H) = 57_H + AE_H + 38_H = C1_H$$

uz:

$$57_H \cdot 01_H = 57_H$$

$$57_H \cdot 02_H = AE_H$$

$$57_H \cdot 04_H = AE_H \cdot 02_H = 47_H$$

$$57_H \cdot 08_H = 47_H \cdot 02_H = 8E_H$$

$$57_H \cdot 10_H = 8E_H \cdot 02_H = 07_H$$

$$57_H \cdot 20_H = 07_H \cdot 02_H = 0E_H$$

$$57_H \cdot 40_H = 0E_H \cdot 02_H = 1C_H$$

$$57_H \cdot 80_H = 1C_H \cdot 02_H = 38_H$$



Osim toga, AES koristi i operacije nad polinomima s elementima iz $GF(2^8)$ stupnja manjeg od 4. Takvi polinomi mogu se predstaviti kao četvorke bitova. U ovom slučaju množenje dvaju polinoma definirano je kao binarno množenje polinoma modulo polinom $x^4 + 1$. Množenje $a(x) \cdot b(x)$ odvija se u dva koraka. U prvom se koraku dobiva polinom šestog stupnja: $c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$.

$$\begin{array}{r} (a_3x^3 + a_2x^2 + a_1x + a_0) \cdot (b_3x^3 + b_2x^2 + b_1x + b_0) = c(x) \\ \hline a_3b_3x^6 + a_2b_3x^5 + a_1b_3x^4 + a_0b_3x^3 \\ a_3b_2x^5 + a_2b_2x^4 + a_1b_2x^3 + a_0b_2x^2 \\ a_3b_1x^4 + a_2b_1x^3 + a_1b_1x^2 + a_0b_1x \\ a_3b_0x^3 + a_2b_0x^2 + a_1b_0x + a_0b_0 \end{array}$$

Dakle, koeficijenti polinoma $c(x)$ iznose:

$$\begin{aligned} c_0 &= a_0b_0 \\ c_1 &= a_0b_1 \oplus a_1b_0 \\ c_2 &= a_0b_2 \oplus a_1b_1 \oplus a_2b_0 \\ c_3 &= a_0b_3 \oplus a_1b_2 \oplus a_2b_1 \oplus a_3b_0 \\ c_4 &= a_1b_3 \oplus a_2b_2 \oplus a_3b_1 \\ c_5 &= a_2b_3 \oplus a_3b_2 \\ c_6 &= a_3b_3 \end{aligned}$$

U drugom se koraku dobiveni rezultat reducira po modulu polinoma $x^4 + 1$. Uzveši u obzir da vrijedi:

$$x^i \bmod (x^4 + 1) = x^{i \bmod 4}$$

koeficijenti polinoma $d(x) = a(x) \cdot b(x)$ su:

$$\begin{aligned} d_0 &= a_0b_0 \oplus a_1b_3 \oplus a_2b_2 \oplus a_3b_1 \\ d_1 &= a_0b_1 \oplus a_1b_0 \oplus a_2b_3 \oplus a_3b_2 \\ d_2 &= a_0b_2 \oplus a_1b_1 \oplus a_2b_0 \oplus a_3b_3 \\ d_3 &= a_0b_3 \oplus a_1b_2 \oplus a_2b_1 \oplus a_3b_0 \end{aligned}$$

Pazeći na redoslijed koeficijenata polinoma $b(x)$ isti koeficijenti d_i mogu se zapisati i ovako:

$$\begin{aligned} d_0 &= a_0b_0 \oplus a_3b_1 \oplus a_2b_2 \oplus a_1b_3 \\ d_1 &= a_1b_0 \oplus a_0b_1 \oplus a_3b_2 \oplus a_2b_3 \\ d_2 &= a_2b_0 \oplus a_1b_1 \oplus a_0b_2 \oplus a_3b_3 \\ d_3 &= a_3b_0 \oplus a_2b_1 \oplus a_1b_2 \oplus a_0b_3 \end{aligned}$$

što se može i zapisati u matričnom obliku:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

AES je Rijndael

Pobjednik na natječaju za napredni simerični algoritam (AES) bio je Rijndael. Autori algoritma *Joan Daemen* i *Vincent Rijmen* predlažu da se izvorni naziv algoritma izgovara "Rain Doll". Međutim, danas je uobičajeno zвати ga jednostavno AES.

AES kriptira blokove jasnog teksta veličine 128 bitova. Moguće duljine ključeva su 128, 192 ili 256 bitova.

Blok koji se kriptira smješten je u pravokutni niz bajtova u četiri retka, dok broj stupaca ovisi o duljini bloka (4, 6, ili 8) i nosi oznaku N_b . Na isti se način tretira i ključ kako je to prikazano na slici (slika 11.11.). Broj stupaca bloka ključa nosi oznaku N_k .

a ₀₀	a ₀₁	a ₀₂	a ₀₃
a ₁₀	a ₁₁	a ₁₂	a ₁₃
a ₂₀	a ₂₁	a ₂₂	a ₂₃
a ₃₀	a ₃₁	a ₃₂	a ₃₃

b ₀₀	b ₀₁	b ₀₂	b ₀₃	b ₀₄	b ₀₅
b ₁₀	b ₁₁	b ₁₂	b ₁₃	b ₁₄	b ₁₅
b ₂₀	b ₂₁	b ₂₂	b ₂₃	b ₂₄	b ₂₅
b ₃₀	b ₃₁	b ₃₂	b ₃₃	b ₃₄	b ₃₅

Slika 11.11. Primjer 128-bitnog ključa (a) i 192-bitnog bloka podataka (b)

Postupak kriptiranja i dekriptiranja obavlja se u koracima. Broj koraka ovisi o veličini bloka podataka i veličini bloka ključa. Tablica 11.1. prikazuje broj koraka N_r u ovisnosti o broju stupaca bloka teksta N_b i broju stupaca bloka ključa N_k . Unatoč tomu što je izvorni kriptosustav *Rijndael* predviđao i veće blokove od 128 bitova, za standard je prihvaćena samo veličina bloka od 128 bitova, tj. $N_b = 4$.

Tablica 11.1. Broj koraka N_r u ovisnosti o brojevima N_k i N_b

a) izvorni *Rijndael*

N_r	$N_b = 4$	$N_b = 6$	$N_b = 8$
$N_k = 4$	10	12	14
$N_k = 6$	12	12	14
$N_k = 8$	14	14	14

b) AES

N_r	$N_b = 4$
$N_k = 4$	10
$N_k = 6$	12
$N_k = 8$	14

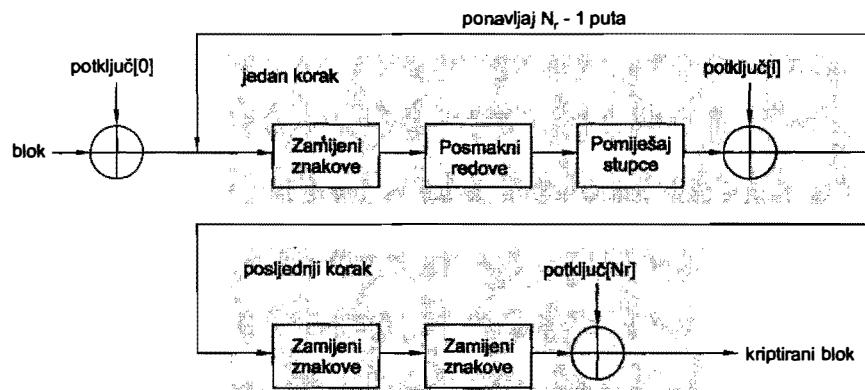
U svakom koraku obavljaju se četiri trasformacije: *Zamjeni znakove*, *Posmakni redove*, *Pomiješaj stupce* i *Dodaj potključ*, osim u zadnjem koraku u kojem se ne obavlja transformacija *Pomiješaj stupce*⁴.

⁴ Ostali detalji algoritma opisani su u izvornom dokumentu autora algoritma [Daemen, 1998] iz 1998. godine. Međutim, autorima se u tom dokumentu potkralo nekoliko pogrešaka koje su ispravljene prilikom definiranja norme pa se za literaturu preporučuje [normaAES, 2001] iz 2001. godine.



Potključevi su po veličini jednaki veličini bloka i dobivaju se iz izvornog ključa. Svi potključevi čine jedan *prošireni ključ*. Prošireni ključ dobiva se tako da se izvorni ključ kopira na početak proširenog, a ostatak se gradi koristeći osim logičke funkcije *isključivo ILI*, rotaciju bitova (kružni posmak), zamjenu bajtova uz pomoć supstitucijskih tablica te dodavanje konstanti. Prošireni ključ ima $(N_r + 1) \cdot N_b$ bitova.

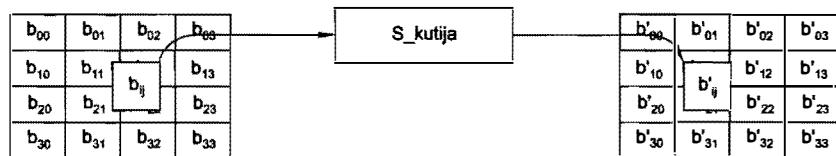
Međurezultat kriptiranja naziva se stanje (engl. *state*). Slika 11.12. prikazuje postupak kriptiranja gdje početni blok jasnog teksta prelazi iz stanja u stanje. Zadnje stanje bloka u postupku kriptiranja zapravo je kriptirani blok.



Slika 11.12. Postupak kriptiranja AES algoritmom

Funkcija *Zamjeni znakove* mijenja znak po znak koristeći supstitucijsku tablicu (*Sbox* ili *S_kutija*):

$$\text{znak} = \text{Skutija}[\text{znak}].$$



Slika 11.13. Transformacija *Zamjeni znakove* odvija se nad svakim znakom zasebno

Funkcija *Posmakni redove* rotira (kružno posmiče) znakove u lijevo, i to u drugom, trećem i četvrtom redu bloka (C_1 , C_2 i C_3) za unaprijed poznat broj mesta koji ovisi o broju N_b (broju stupaca bloka). Prvi red (C_0) se ne posmiče.

Tablica 11.2. Broj znakova za koji se posmiče redak bloka

N_b	C_1	C_2	C_3
4	1	2	3
6	1	2	3
8	1	3	4

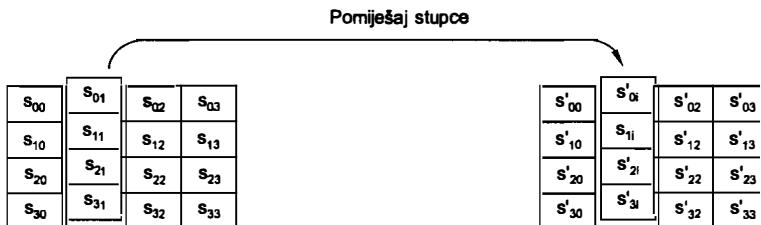
U funkciji *Pomiješaj stupce* množi se stupac po stupac bloka (tako da se svaki stupac promatra kao četveročlani polinom) s fiksnim polinomom $a(x) = 03_Hx^3 + 01_Hx^2 + 01_Hx + 02_H$ modulo $x^4 + 1$, što se matrično može prikazati ovako:

$$\overline{s'(x)} = \overline{a(x)} \otimes \overline{s(x)}.$$

Drugim riječima, za svaki stupac bloka računa se stupac novog stanja bloka prema izrazu:

$$\begin{bmatrix} s'_{0i} \\ s'_{1i} \\ s'_{2i} \\ s'_{3i} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0i} \\ s_{1i} \\ s_{2i} \\ s_{3i} \end{bmatrix}$$

na način kako je to već opisano u primjeru množenja $GF(2^8)$ na stranici 298.

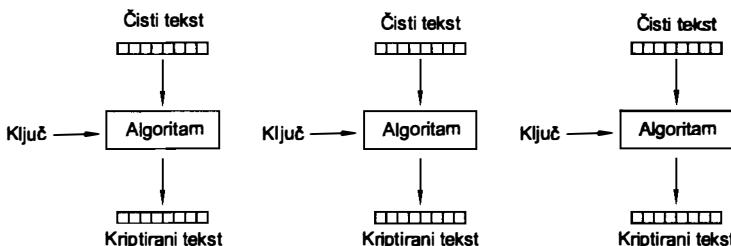


Slika 11.14. Slikoviti prikaz transformacije *Pomiješaj stupce*

11.4. Načini kriptiranja

11.4.1. Elektronička bilježnica

Elektronička bilježnica (engl. *Electronic Notebook, ECB*) najjednostavniji je i uobičajeni način kriptiranja. Svaki blok se kriptira, odnosno dekriptira zasebno.



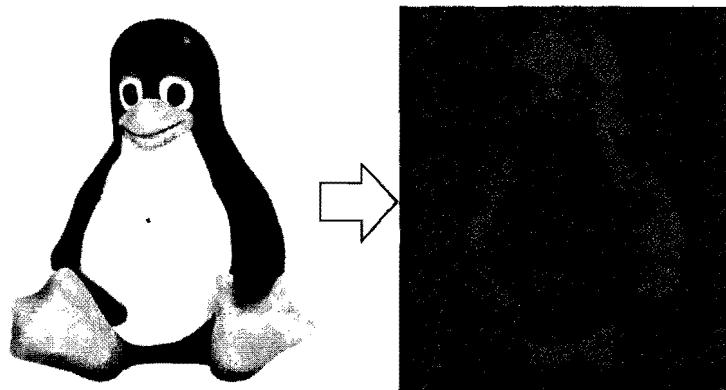
Slika 11.15. Kriptiranje u ECB načinu

Svojstva *ECB* načina rada:

- Idenični slijedni nekriptirani blokovi rezultiraju identičnim kriptiranim blokovima.

- Blokovi su kriptirani neovisno o ostalim blokovima.
- Pogreška unutar jednog bloka utjecat će na dekriptiranje samo tog bloka.

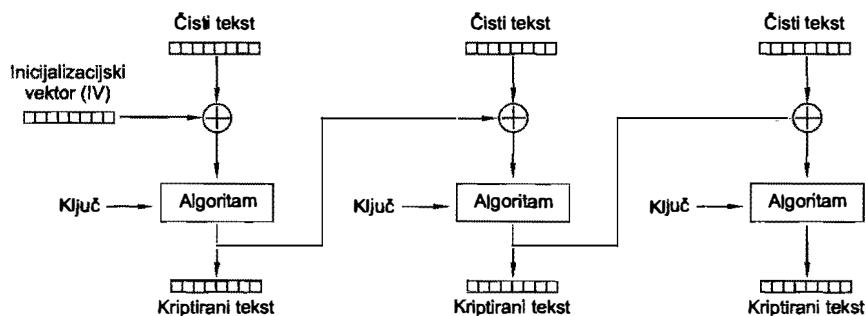
Slika 11.16. prikazuje kako se kriptiranjem ne može djelotvorno sakriti sadržaj slike u *bitmap* formatu ako se primjenjuje najjednostavniji *ECB* način kriptiranja. Korišten je kriptografski algoritam *AES*.



Slika 11.16. Otkrivanje strukture u ECB načinu kriptiranja

11.4.2. Ulančavanje

Ulančavanje (engl. *Cipher Block Chaining*, *CBC*) najpopularniji je način rada algoritma za kriptiranje blokova jasnog teksta. Jasni tekst se zbraja (*XOR*) s kriptiranim blokom i tada se primjenjuje algoritam na rezultirajući blok.

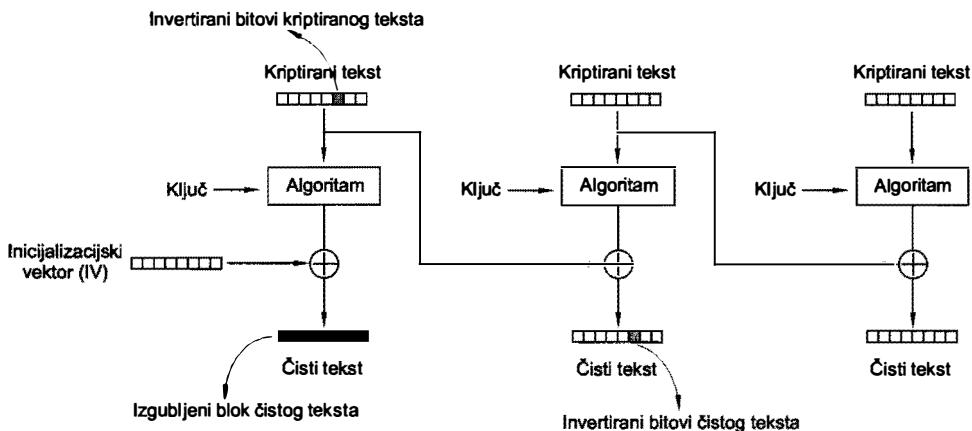


Slika 11.17. Kriptiranje ulančavanjem

Svojstva CBC načina kriptiranja:

- Blok kriptiranog teksta ovisi o svim prethodnim blokovima.
- Želimo li povećati razinu sigurnosti treba izbjegavati korištenje istog inicijalizacijskog vektora s istim ključem.

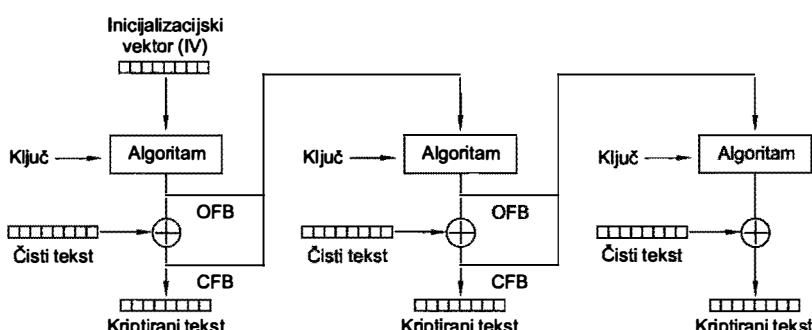
- Mehanizam ulančavanja uzrokuje da kriptirani blok c_j ovisi o bloku jasnog teksta x_j i svim nekriptiranim blokovima koji su prethodili.
- Jedan bit pogreške u kriptiranom bloku c_j utječe na dekriptiranje blokova c_j i c_{j+1} (zato što x_j ovisi o c_j i c_{j-1}).
- Ovakav je način rada *samosinkronizirajući*. Kada se pojavi greška u nekom bloku c_j ali ne i u c_{j+1} , blok c_{j+2} ispravno je dekriptiran u x_{j+2} . Isto razmatranje vrijedi u slučaju gubitka jednog ili više blokova (slika 11.18.).



Slika 11.18. Jedna greška u kriptiranom tekstu uzrokuje dvije greške u čistom tekstu u CBC načinu kriptiranja

11.4.3. CFB i OFB načini kriptiranja

Za kriptiranje toka podataka, slično kao i kod kriptografskog algoritma jednokratna bilježnica, duljina ključa treba biti jednaka duljini poruke koja se kriptira. Ključ proizvoljne duljine (engl. *keystream*) moguće je postići uzastopnim kriptiranjem neke početne vri-



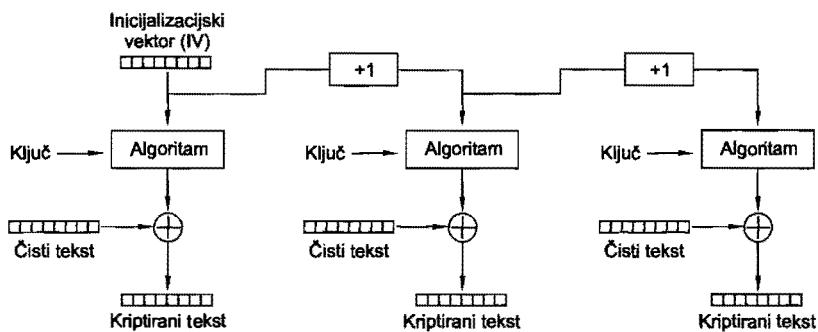
Slika 11.19. CFB i OFB načini kriptiranja

jednosti (OFB način kriptiranja) ili ulančanim kriptiranjem već kriptiranih blokova (CFB način kriptiranja). Kriptirani tekst dobiva se jednostavnim zbrajanjem (logičkom funkcijom *isključivo ILI*) jasnog teksta s tako dobivenim ključem koji je jednak duljine kao i jasni tekst. Inicijalizirajući vektor ne treba biti tajan.

U CFB načinu kriptiranja pogreška u kriptiranom tekstu propagira se na sljedeći blok (greška u jednom kriptiranom bloku daje dva neispravna bloka čistog teksta). To nije slučaj kod OFB načina kriptiranja jer sljedeći kriptirani blok ne zavisi od prethodnog.

11.4.4. Brojač

Način kriptiranja s brojačem (engl. *Counter*) ili CTR način kriptiranja sličan je OFB načinu kriptiranja samo što se niz bitova ključa (engl. *keystream*) dobiva uzastopnim kriptiranjem rastuće vrijednosti brojača. Umjesto brojača može se koristiti i bilo koja druga funkcija koja ne ponavlja vrijednosti kroz duži period, ali brojač je najjednostavnije, najučinkovitije pa time i najpopularnije rješenje.



Slika 11.20. CTR način kriptiranja

11.5. Asimetrični kriptosustavi, sustavi s javnim ključem

11.5.1. Neke činjenice i algoritmi iz teorije brojeva

Asimetrični kriptosustavi zasnivaju se na određenim svojstvima brojeva koja se istražuju u teoriji brojeva. Pri kriptiranju se razgovjetni tekst kodira kao niz prirodnih brojeva koji se odabranom funkcijom kriptiranja i ključem kriptiranja K_E preračunavaju u niz brojeva kriptiranog teksta. Funkcija kriptiranja mora biti takva da iz niza brojeva kriptiranog teksta napadač samo s velikim naporima može odrediti izvorni niz brojeva. Međutim, poznavanje ključa dekriptiranja K_D omogućuje lako izračunavanje izvornog niza brojeva.

Podsjetit ćemo se na neke činjenice iz teorije brojeva.

Djeljivost

Broj a djeljiv je s brojem d kada je a višekratnik od d . Postoji nekoliko načina označavanja djeljivosti:

$$\begin{array}{ll} d | a & d \text{ dijeli } a, d \text{ je djelitelj od } a; \\ a = k \cdot d & a \text{ je višekratnik od } d. \end{array}$$

Najmanji djelitelj od a je $d = 1$, a najveći djelitelj je $d = a$. To su trivijalni djelitelji. Netrivijalni djelitelji zovu se faktori.

PRIMJER 11.3.

Broj $a = 24$ ima sljedeće djelitelje: 1, 2, 3, 4, 6, 8, 12, 24.

Trivijalni djelitelji su 1 i 24, a faktori 2, 3, 4, 6, 8, 12.



Prosti ili prim brojevi

Broj $a > 1$ koji nema faktora (ima samo djelitelje 1 i a) zove se prosti ili prim broj.

Teorem dijeljenja

Za svaki cijeli broj a i bilo koji pozitivni cijeli broj n postoje jedinstveni cijeli brojevi:

- kvocijent, količnik q i
- reziduum, ostatak r (uz $0 \leq r < n$),

tako da vrijedi:

$$a = q \cdot n + r.$$

Možemo pisati:

$$q = \left\lfloor \frac{a}{n} \right\rfloor \quad \text{i} \quad r = a \bmod n,$$

odnosno:

$$a = \left\lfloor \frac{a}{n} \right\rfloor \cdot n + (a \bmod n),$$

$$a \bmod n = a - \left\lfloor \frac{a}{n} \right\rfloor \cdot n.$$

Ekvivalentnost po modulu, kongruentnost

Broj a je ekvivalentan broju b po modulu n ako je:

$$a \bmod n = b \bmod n.$$

Kaže se da su a i b kongruentni po modulu n i piše se:

$$a \equiv b \pmod{n}.$$

Relativno prosti brojevi

Brojevi a i b relativno su prosti ako je najveći zajednički djelitelj brojeva a i b jednak 1, tj. brojevi a i b nemaju zajedničkih faktora ili $\text{nzd}(a, b) = 1$.

Eulerova phi funkcija

Neka je $Z_n = \{0, 1, 2, \dots, n - 1\}$ prsten u kojem su definirane operacije zbrajanja, oduzimanja i množenja po modulu n .

Neka je Z_n^* podskup koji se sastoji od elemenata skupa Z_n koju su relativno prosti u odnosu na n , tj.:

$$Z_n^* = \{a \in Z_n, \text{ nzd}(a, n) = 1\}.$$

Broj elemenata skupa Z_n^* , tj. kardinalnost skupa $|Z_n^*|$ jednaka je Eulerovoj *phi* ili *totient funkciji* $\varphi(n)$. Ako je $n = p$ prosti broj, onda je $\varphi(p) = p - 1$.

Ako n ima rastav na proste faktore $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, onda je:

$$\varphi(n) = n \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_k}\right).$$

Ako je $n = p \cdot q$, gdje su p i q prosti brojevi, onda je:

$$\varphi(n) = n \left(1 - \frac{1}{p}\right) \left(1 - \frac{1}{q}\right) = (p - 1)(q - 1).$$

Dokaz za $n = p$ trivijalan je jer je:

$$Z_p = \{0, 1, 2, \dots, p - 1\} \quad \text{i} \quad Z_p^* = \{1, 2, \dots, p - 1\}.$$

Dokaz za $n = p \cdot q$ je sljedeći:

Skup $Z_n = \{0, 1, 2, \dots, p \cdot q - 1\}$ s ukupno $p \cdot q$ elemenata je unija podskupova:

$\{0\}$	s 1 elementom,
$\{p, 2 \cdot p, \dots, (q - 1) \cdot p\}$	s $q - 1$ elemenata,
$\{q, 2 \cdot q, \dots, (p - 1) \cdot q\}$	s $p - 1$ elemenata,
Z_n^*	sa $ Z_n^* $ elemenata.

Prema tome je:

$$p \cdot q = 1 + (q - 1) + (p - 1) + |Z_n^*|,$$

što daje:

$$|Z_n^*| = (p - 1)(q - 1).$$

PRIMJER 11.4.

Neka je $n = 15 = 3 \cdot 5$, tj. $p = 3$ i $q = 5$.

Tada je:

$$Z_{15} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\} \quad \text{i} \quad Z_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}.$$

Prema tome je $|Z_{15}^*| = 8$, što se dobiva i iz $\varphi(15) = (5 - 1)(3 - 1) = 8$.



Modularno potenciranje

Potenciranje s velikim eksponentima prikladno je obaviti uzastopnim kvadriranjem.

Želimo izračunati $d = b^a \pmod{n}$. Neka je $a_m, a_{m-1}, a_{m-2}, \dots, a_1, a_0$ binarni prikaz od a . Potenciranje će obaviti sljedeći algoritam složenosti $O(\log a)$:

```

d = 1;
i = m;
dok je (i >= 0) {
    d = (d * d) mod n;
    ako je (a[i] == 1) {
        d = (d*b) mod n;
    }
    i--;
}

```



PRIMJER 11.5.

Pogledajmo što se dobiva za $n = 734$, $b = 5$ i $a = 635$.

Binarni prikaz eksponenta je: $a = 1001111011$ te se dobiva:



i	9	8	7	6	5	4	3	2	1	0
a[i]	1	0	0	1	1	1	1	0	1	1
d	5	25	625	685	261	29	535	699	253	21

Eulerov teorem

Za svaki prirodni broj $n > 1$ vrijedi:

$$a^{\varphi(n)} \equiv 1 \pmod{n} \quad \text{za sve } a \in Z_n^*.$$

Mali Fermatov teorem

Posebno za proste brojeve p vrijedi:

$$a^{p-1} \equiv 1 \pmod{p} \quad \text{za sve } a \in Z_p^*$$

S obzirom na to da je $Z_p^* = \{1, 2, \dots, p-1\}$, Fermatov teorem vrijedi za sve brojeve iz Z_p s iznimkom broja 0.

Posljedica toga teorema je da za sve $a \in Z_p$ vrijedi da je:

$$a^p \equiv a \pmod{p}.$$

(Postoje i neki složeni brojevi (Charmicleovi brojevi) kod kojih za mnoge vrijednosti od a vrijedi gornji izraz, primjerice 561, 1105.)

PRIMJER 11.6.



Promotrimo nizove potencija po modulu za neke od brojeva iz skupa Z_p^* . Neka je $p = 11$.

i	0	1	2	3	4	5	6	7	8	9	10	11
$2^i \bmod 11$	1	2	4	8	5	10	9	7	3	6	1	2
$5^i \bmod 11$	1	5	3	4	9	1	5	3	4	9	1	5
$7^i \bmod 11$	1	7	5	2	3	10	4	6	9	8	1	7

Zanimljivo je da se kod nekih nizova potencija ne pojavljuju svi elementi skupa Z_p^* . Tako za $a = 5$ potencije generiraju podskup $\{1, 3, 4, 5, 9\}$. Taj podskup ima 5 članova što se označava s:

$$\text{ord}_{11}(5) = 5.$$

Ako je $\text{ord}_n(a) = |Z_n^*|$, onda je svaki element od Z_n^* potencija od a po modulu n . Broj a tada je *osnovni korijen* ili *generator* od Z_n^* . U primjeru 11.6. brojevi 2 i 7 su osnovni korijeni od Z_{11}^* , dok broj 5 nije osnovni korijen.

Ako Z_n^* ima osnovni korijen, onda se kaže da Z_n^* čini cikličku grupu. Dokazano je da je Z_n^* ciklička grupa za $n > 1$ koji poprimaju sljedeće vrijednosti: $2, 4, p^e, 2p^e$, gdje su p prosti brojevi i e prirodni brojevi.

Diskretni logaritam ili indeks

Neka je a osnovni korijen od Z_n^* i b jedan element od Z_n^* . Postoji takav x tako da je:

$$a^x \equiv b \pmod{n}.$$

Broj x je *diskretni logaritam* ili *indeks* broja $b \pmod{n}$ u odnosu na bazu a . Može se pisati:

$$x = \text{ind}_{n,a}(b).$$

Primjena kineskog teorema ostataka

Kineski teorem ostataka (Sun-Tsu, oko 100. godine) dovodi u vezu sustav jednadžbi po modulima koji su međusobno po parovima relativno prosti s jednadžbom po modulu koji je jednak njihovu umnošku.

Neka je $n = n_1 \cdot n_2 \cdot \dots \cdot n_k$, gdje su svi parovi faktora relativno prosti. Teorem kaže da je struktura Z_n identična Kartezijevu produktu $Z_{n_1} \cdot Z_{n_2} \cdot \dots \cdot Z_{n_k}$.

Konkretno, za $n_1 = p$ i $n_2 = q$ to znači da za bilo koja dva cijela broja x i a vrijede jednadžbe:

$$x \equiv a \pmod{p},$$

$$x \equiv a \pmod{q}$$

onda i samo onda ako je:

$$x \equiv a \pmod{n}.$$

11.5.2. Asimetrični kriptosustav RSA

Kao što je već rečeno, asimetrični kriptosustavi imaju različite ključeve kriptiranja K_E i dekriptiranja K_D i mogu se opisati već navedenim izrazima:

$$\begin{aligned} C &= E(P, K_E), \\ P &= D(C, K_D), \\ P &= D(E(P, K_E), K_D). \end{aligned}$$

Jedan od takvih sustava (koji je postao *de facto* standard u svijetu) razradili su autori *Ron Rivest, Adi Shamir i Len Adleman*, po čijim je početnim slovima prezimena sustav dobio svoj naziv *RSA*. On je razrađen na osnovi svojstava brojeva koje smo razmotrili u prethodnom odjeljku.

Postupak izgradnje RSA sustava

Kriptosustav RSA izgrađuje se na temelju prethodno opisanih postavki iz teorije brojeva na sljedeći način:

1. Odabiru se dva velika prosta broja p i q ($p > 10^{100}$, $q > 10^{100}$).
2. Izračunava se umnožak $n = p \cdot q$.
3. Izračunava se umnožak $\varphi(n) = (p - 1) \cdot (q - 1)$.
4. Odabire se broj $e < \varphi(n)$ i relativno prost u odnosu na $\varphi(n)$, tj. koji nema zajedničkih faktora s $\varphi(n)$.
5. Izračunava se broj $d < \varphi(n)$ tako da bude $e \cdot d \equiv 1 \pmod{\varphi(n)}$, što se može napisati drukčije kao $e \cdot d = k \cdot \varphi(n) + 1$.
6. Par $K_E = (e, n)$ obznanjuje se i proglašava javnim ključem (engl. *public key*).
7. Par $K_D = (d, n)$ se taj i postaje privatni ključ (engl. *private key*).

Kriptiranje se obavlja funkcijom kriptiranja:

$$C = E(P, K_E) = RSA(P, K_E) = P^e \pmod{n},$$

a dekriptiranje funkcijom dekriptiranja:

$$P = D(C, K_D) = RSA^{-1}(C, K_D) = C^d \pmod{n}.$$

Kriptiranje i dekriptiranje može se obaviti algoritmom modularnog potenciranja opisanim u primjeru 11.3.

Provjera korektnosti RSA kriptosustava

Pogledajmo što se dobiva dekriptiranjem kriptiranog teksta, tj. što daje:

$$RSA^{-1}(RSA(P, K_E), K_D),$$

odnosno:

$$(P^e \bmod n)^d \bmod n = P^{e \cdot d} \bmod n.$$

S obzirom na to da je:

$$e \cdot d = k \cdot \varphi(n) + 1 = k(p-1)(q-1) + 1,$$

može se za one P koji nisu kongruentni s 0 (mod p) uporabom Fermatova teorema pisati:

$$\begin{aligned} P^{e \cdot d} &\equiv P^{k(p-1)(q-1)+1} & (\text{mod } p) \\ &\equiv P(P^{(p-1)})^{k(q-1)} & (\text{mod } p) \\ &\equiv P(1)^{k(q-1)} & (\text{mod } p) \\ &\equiv P & (\text{mod } p). \end{aligned}$$

To je, također, trivijalno ispunjeno i za:

$$P \equiv 0 \pmod{p}.$$

Jednako tako vrijedi i:

$$\begin{aligned} P^{e \cdot d} &\equiv P^{k(p-1)(q-1)+1} & (\text{mod } q) \\ &\equiv P(P^{(q-1)})^{k(p-1)} & (\text{mod } q) \\ &\equiv P(1)^{k(p-1)} & (\text{mod } q) \\ &\equiv P & (\text{mod } q). \end{aligned}$$

Dakle, vrijedi:

$$\begin{aligned} P^{e \cdot d} &\equiv P & (\text{mod } p) \\ P^{e \cdot d} &\equiv P & (\text{mod } q), \end{aligned}$$

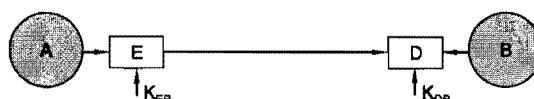
što je u skladu s kineskim teoremom ostataka ispunjeno samo ako je:

$$P^{e \cdot d} \equiv P \pmod{n}$$

Prema tome, RSA sustav je korektan.

11.5.3. Komuniciranje uporabom kriptosustava RSA

Branko koji želi komunicirati s drugim sudionicima obznanjuje svoj javni ključ kriptiranja K_{EB} i čuva samo za sebe svoj privatni ključ dekriptiranja K_{DB} . Ana, kada želi poslati Branku poruku, P saznaje njegov javni ključ K_{EB} , kriptira poruku tim ključem i šalje ju Branku. Jedino Branko zna svoj privatni ključ dekriptiranja K_{DB} i jedino on može dekriptirati poruku.



Slika 11.21. Sigurna komunikacija uporabom asimetričnog kriptosustava

Kriptiranje bi se moglo obaviti tako da se razgovijetni tekst P podijeli na niz brojeva jednake bitovne duljine:

$$P = P_0 P_1 P_2 \dots P_i \dots$$

Kriptirani tekst dobio bi se tako da se kriptira svaki P_i :

$$C_i = RSA(P_i, K_E) = P_i^e \bmod n$$

i dobije:

$$C = C_0 C_1 C_2 \dots C_i \dots$$

Na odredišnoj bi se strani tada obavilo dekriptiranje svakog C_i i ponovno uspostavilo razgovijetni tekst.

PRIMJER 11.7.

Odaberimo $p = 17$ i $q = 19$. Dobivamo $n = 323$ i $\varphi(n) = 16 \cdot 18 = 288$.

Moramo odabrati d i e , tako da oni budu relativno prosti u odnosu na $\varphi(n)$ i da bude zadovoljeno:

$$e \cdot d = k \cdot \varphi(n) + 1.$$

Pogledajmo sljedeće vrijednosti:

k	$k \cdot \varphi(n) + 1$	rastav na proste faktore
0	1	
1	289	$= 17 \cdot 17$
2	577	prosti broj
3	865	$= 5 \cdot 173$

Javni ključ kriptiranja je $K_{EB} = (17, 323)$ i privatni ključ dekriptiranja $K_{DB} = (17, 323)$. U ovom su primjeru iznimno javni i tajni ključ isti. Razgovijetni tekst možemo podijeliti na niz brojeva tako da bude $P_i < n$. Tako, primjerice, dobivamo:

P_i	17	98	62	22	73
$C_i = P_i^{17} \bmod 323$	85	13	232	260	158
$P_i = C_i^{17} \bmod 323$	17	98	62	22	73

Izaberemo li $K_{EB} = (5, 323)$ i $K_{DB} = (173, 323)$, dobivamo:

P_i	17	98	62	22	73
$C_i = P_i^5 \bmod 323$	272	319	180	167	99
$P_i = C_i^{173} \bmod 323$	17	98	62	22	73



PRIMJER 11.8.

Odaberimo $p = 43$ i $q = 47$. Dobivamo $n = 2021$ i $\varphi(n) = 42 \cdot 46 = 1932$.

Moramo odabrati d i e , tako da oni budu relativno prosti u odnosu na $\varphi(n)$ i da vrijedi:

$$e \cdot d = k \cdot \varphi(n) + 1.$$

Pogledajmo sljedeće vrijednosti:

k	$k \cdot \varphi(n) + 1$
0	1
1	1933
2	3865
3	5797

$= 5 \cdot 773$
 $= 31 \cdot 187 = 17 \cdot 341$

Možemo odabrat:

$$K_{EB} = (31, 2021) \quad \text{i} \quad K_{DB} = (187, 2021)$$

ili:

$$K_{EB} = (17, 2021) \quad \text{i} \quad K_{DB} = (341, 2021).$$

Tako, primjerice, dobivamo:

P_i	1723	981	2009	162
$C_i = P_i^{31} \bmod 2021$	334	305	812	1482
$P_i = C_i^{187} \bmod 2021$	1723	981	2009	162

odnosno:

P_i	1723	981	2009	162
$C_i = P_i^{17} \bmod 2021$	1574	821	228	1281
$P_i = C_i^{341} \bmod 2021$	1723	981	2009	162

Tajnost RSA kriptosustava postiže se odabirom prostih brojeva p i q s više od 100 dekadskih znamenki. Stoga se kriptiranje i dekriptiranje svodi na potenciranje s velikim eksponentima što je relativno dugotrajno. Brzina kriptiranja i dekriptiranja je reda veličine od 10^3 do 10^5 bit / s. U usporedbi sa simetričnim kriptiranjem to je za nekoliko redova veličina sporije i nema ga smisla upotrebljavati za prenošenje većih količina informacija.

11.5.4. Dobrota RSA kriptosustava

Dobrota RSA kriptosustava zasniva se na teškoći faktoriziranja velikih brojeva. Naime, uz objavljeni javni ključ $K_E = (e, n)$ uljez bi mogao odrediti privatni ključ $K_D = (d, n)$ ako uspije faktorizirati broj n , tj. saznati proste brojeve p i q . Tada bi on mogao izračunati

$\varphi(n)$ i odrediti pripadni d iz uvjeta:

$$e \cdot d = k \cdot \varphi(n) + 1.$$

Međutim, faktoriziranje velikih brojeva vrlo je teško. Do danas nema drugog načina do dijeljenja nizom brojeva $2, 3, \dots, \sqrt{n}$. U najgorem slučaju kada je n prosti broj, imat ćemo $O(\sqrt{n})$ operacija. Ako broj n ima m bitova, onda je:

$$\begin{aligned} n &\approx 2^m \\ \sqrt{n} &\approx 2^{m/2} \end{aligned}$$

te je složenost faktoriziranja $O(2^{m/2})$.

Do sada, osim faktoriziranja broja n , nisu pronađeni drugi načini za razbijanje RSA kriptosustava. Pokazuje se da je s današnjom računalnom snagom moguće faktorizirati 512 bitovne brojeve, ali je već nemoguće u razumnom vremenu faktorizirati 1024-bitovne brojeve.

To bi značilo da uz $n = p \cdot q$ zadovoljavaju prosti brojevi koji imaju po 512 bitova. S obzirom na to da je $2^{512} \approx 10^{150}$, to znači da bi trebalo pronalaziti proste brojeve s više od 150 znamenaka.

Srećom prosti brojevi nisu previše rijetki i može ih se pronaći mnogo. Funkcija gustoće prostih brojeva $\pi(n)$ funkcija je koja daje broj prostih brojeva manjih od n . Tako je:

$$\begin{aligned} \pi(10) &= 4 && (\text{prosti brojevi manji od } 10 \text{ su: } 2, 3, 5, 7), \\ \pi(15) &= 6 && (\text{prosti brojevi manji od } 15 \text{ su: } 2, 3, 5, 7, 11, 13). \end{aligned}$$

Za tu funkciju vrijedi:

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n / \ln n} = 1.$$

Za velike n može se, dakle, odrediti približni broj prostih brojeva manjih od n izrazom:

$$\pi(n) = n / \ln n.$$

Tako se dobiva:

n	10^{120}	10^{130}	10^{140}	10^{150}
$n / \ln n$	$3.62 \cdot 10^{117}$	$3.34 \cdot 10^{127}$	$3.3 \cdot 10^{137}$	$2.89 \cdot 10^{147}$

Prema tome, u intervalu između 10^{140} i 10^{150} ima $2.89 \cdot 10^{147} - 3.3 \cdot 10^{137} \approx 2.89 \cdot 10^{147}$ prostih brojeva.

S obzirom na to da u svemiru ima manje od 10^{80} atoma to znači da bismo svakom atomu u svemiru mogli pridijeliti 10^{67} prostih brojeva!

Pronalaženje prostih brojeva

Za velike n funkcija gustoće prostih brojeva približno je jednaka:

$$\pi(n) = n / \ln n.$$

Prema tome, vjerojatnost da slučajno odabrani veliki broj n bude prosti približno je jednaka:

$$1 / \ln n.$$

Dručije rečeno, u okolini slučajno odabranog broja n treba ispitati približno $\ln n$ brojeva i utvrditi jesu li oni prosti. Zapravo treba ispitati samo $4/10$ od tih brojeva jer su u jednoj dekadi (nizu uzastopnih brojeva čije su znamenke jedinice 0, 1, 2, 3, 4, 5, 6, 7, 8, 9) kandidati za proste brojeve samo oni čije su znamenke 1, 3, 7 i 9.

Tako dobivamo:

n	10^{120}	10^{130}	10^{140}	10^{150}
$4 \ln(n)/10$	112	121	130	140

To znači da bi u okolini jednog slučajno odabranog broja sa 120 znamenaka trebalo ispitati oko 112 brojeva.

Najjednostavnije ispitivanje prostosti opet je dijeljenje s nizom brojeva $2, 3, \dots, \sqrt{n}$. To je već opisani postupak faktorizacije i njegova je složenost za m bitovne brojeve jednaka $O(2^{m/2})$. Srećom postoje postupci heurističkog ispitivanja složenosti kod kojeg se ne provodi faktorizacija već se samo dobiva odgovor je li broj složen ili nije složen.

Heurističko ispitivanje po Miller-Rabinu

Heurističko ispitivanje po *Miller-Rabinu* zasniva se na Fermatovu teoremu koji kaže da za proste brojeve vrijedi:

$$a^{p-1} \equiv 1 \pmod{p} \quad \text{za sve } a \in Z_p^*.$$

Nažalost, postoje i neki složeni brojevi (Charmicleovi brojevi) kod kojih je gornji izraz zadovoljen za mnoge vrijednosti baze a . Primjerice, to su brojevi 561, 1105, 1729. Oni su, istina, vrlo rijetki tako da ih ima samo 255 manjih od 10^8 . Ipak bi se moglo dogoditi da lažno utvrdimo lažnu prostost broja koji je složen.

PRIMJER 11.9.

Pogledajmo za prvih nekoliko baza a potencije Charmichelovih brojeva 561 i 1105.

Za Charmichelov broj $561 = 3 \cdot 11 \cdot 17$ dobivamo:

$$2^{560} \pmod{561} = 1$$

$$8^{560} \pmod{561} = 1$$

$$3^{560} \pmod{561} = 375$$

$$9^{560} \pmod{561} = 375$$

$$4^{560} \pmod{561} = 1$$

$$10^{560} \pmod{561} = 1$$

$$5^{560} \pmod{561} = 1$$

$$11^{560} \pmod{561} = 154$$

$$6^{560} \pmod{561} = 375$$

$$12^{560} \pmod{561} = 375$$

$$7^{560} \pmod{561} = 1$$

$$13^{560} \pmod{561} = 1.$$



Za Charmichleov složeni broj $1105 = 5 \cdot 13 \cdot 17$ dobivamo:

$$2^{1104} \bmod 1105 = 1$$

$$3^{1104} \bmod 1105 = 1$$

$$4^{1104} \bmod 1105 = 1$$

$$5^{1104} \bmod 1105 = 885$$

$$6^{1104} \bmod 1105 = 1$$

$$7^{1104} \bmod 1105 = 1$$

$$8^{1104} \bmod 1105 = 1$$

$$9^{1104} \bmod 1105 = 1$$

$$10^{1104} \bmod 1105 = 885$$

$$11^{1104} \bmod 1105 = 1$$

$$12^{1104} \bmod 1105 = 1$$

$$13^{1104} \bmod 1105 = 936.$$

Ispitivanje na prostost Fermatovim teoremom otežano je za Charmicloove brojeve. Stoga bi ispitivanje bilo dobro obaviti s nekoliko vrijednosti baze a . Time bi se smanjila vjerojatnost lažnog utvrđivanja prostosti za složeni broj.

Provjera složenosti može se obaviti modifikacijom algoritma za modularno potenciranje iz primjera 11.3. S obzirom na to da želimo provjeru obaviti za nekoliko vrijednosti od a napisat ćemo funkciju koja će za jednu vrijednost od a utvrditi je li zadovoljen Fermatov teorem.

Heurističko ispitivanje složenosti po Rabinu i Milleru uzima u obzir još jedno svojstvo prirodnih brojeva. Naime, broj x netrivijalni je drugi korijen od:

$$1 \bmod n$$

ako je zadovoljena jednadžba:

$$x^2 \equiv 1 \pmod{n},$$

a x nije niti jednak 1 niti $n - 1$ (odnosno +1 ili -1).

Ako postoji netrivijalni drugi korijen od $1 \bmod n$, onda je n sigurno složeni broj! Funkcija će utvrditi složenost kada se ustanovi:

- da za dani n Fermatov teorem nije zadovoljen ili
- da za dani n postoji netrivijalni drugi korijen od $1 \bmod n$.

Funkcija će u varijabli G vratiti vrijednost 1 ako je utvrđena složenost ili vrijednost 0 ako ne utvrdi složenost. Vjerodostojnost tih dvaju odgovora nije jednaka. Ako je utvrđena složenost, onda je broj sigurno složen. Ako se ne utvrdi složenost, postoji vjerojatnost, istina mala, da je broj ipak složen.

Neka je $b_m, b_{m-1}, b_{m-2}, \dots, b_1, b_0$ binarni prikaz od $n - 1$. Složenost izvođenja te funkcije bit će $O(\log n)$, tj. $O(m)$.

Pomoćna funkcija programa za određivanje složenosti:

```
funkcija provjera_slozenosti(a,n,G) {
    G = 0;
    d = 1;
    c = n-1;
    i = -1;
```



```

dok je c > 0 {
    i++;
    b[i] = c mod 2;
    c = c div 2;
}
dok je ((i >= 0) ∧ (G == 0)) {
    d_s = d;
    d = (d * d) mod n;
    ako je ((d == 1) ∧ (d_s != 1) ∧ (d_s != n-1)) {
        G = 1;
    }
    ako je (b[i] == 1) {
        d = (d*a) mod n;
    }
    i--;
}
ako je ((i == -1) ∧ (d != 1)) {
    G = 1;
}
}

```

Program za utvrđivanje prostosti može se napisati tako da se generira k nasumičnih brojeva a i zatim k puta poziva funkcija za utvrđivanje prostosti.

Neka funkcija `random(1,n-1)` vraća nasumični broj veći od 1 i manji od $n-1$. Program koji će s velikom pouzdanošću odrediti je li broj prost može, dakle, glasiti ovako:



```

G = 0;
i = k;
dok je ((i >= 0) ∧ (G == 0)) {
    a = random (1,n-1);
    provjera_slozenosti(a,n,G);
    i--;
}
ako je (G == 1) {
    n je složeni broj // sigurno!
}
inače {
    n je prosti broj // skoro sigurno!
}

```

Prema tome, ako u blizini nekog nasumično odabranog velikog broja moramo ispitati $4/10 \ln n$ brojeva i svako od tih ispitivanja je složenost $O(\log n)$ te pritom odabiremo k

baza a , možemo procijeniti da je složenost pronalaženja prostog broja jednaka:

$$O(k \cdot \log^2 n) = O(k \cdot m^2) = O(m^2).$$

Uspješnost RSA postupka zasniva se upravo na bitnoj različitosti složenosti postupka pronalaženja prostih brojeva $O(m^2)$ i postupka faktorizacije koji ima eksponencijalnu složenost $O(2^{m/2})$.

11.6. Sažetak poruke, utvrđivanje besprijeckornosti

11.6.1. Digitalna omotnica

Jedno od praktičnih rješenja je da Ana koja želi poslati Branku tekst P postupi ovako:

- odabere proizvoljni simetrični ključ K ;
- kriptira tekst P simetričnom funkcijom, primjerice DES, i dobiva:

$$C_1 = DES(P, K),$$

- kriptira Brankovim javnim ključem tajni ključ K i dobiva:

$$C_2 = RSA(K, K_{EB}),$$

- šalje poruku $M = (C_1, C_2)$.

Branko nakon primitka poruke M :

- najprije dekriptira C_2 svojim privatnim ključem K_{DB} i tako sazna tajni ključ:

$$K = RSA^{-1}(RSA(K, K_{EB}), K_{DB})$$

- nakon toga tajnim ključem K dekriptira tekst C_1 i dobiva izvorni tekst

$$P = DES^{-1}(DES(P, K)).$$

Poruku M možemo nazvati *digitalnom omotnicom*. Digitalnom se omotnicom postiže ispunjenje zahtjeva *tajnosti* slanja poruke. Samo Branko može pročitati poruku jer jedino on može svojim privatnim ključem dekriptirati C_2 i sazнати tajni ključ K . Međutim, ovakvim se slanjem poruke ne ispunjavaju ostali sigurnosni zahtjevi. Tako se, primjerice, svatko može lažno predstaviti kao Ana i poslati poruku M . Trebat će, dakle, ugraditi mehanizam provjere identiteta pošiljatelja poruke, tj. postupak autentifikacije. Nadalje, primatelj nema mogućnost provjere besprijeckornosti (integriteta) dobivene poruke. Za ostvarenje tih zahtjeva trebat će nam još neki matematički postupci koje ćemo razmotriti kasnije.

Digitalni potpis

Ustanovili smo da Ana i Branko mogu komunicirati uporabom *RSA* sustava ako oboje obzname svoje javne ključeve. Oni mogu razmjenjivati veće količine tajnih informacija tako da koriste digitalnu omotnicu. U nekim primjenama tajnost ne mora biti osnovni sigurnosni zahtjev, već je bitno da bude zadovoljen uvjet besprijecknosti (integriteta).

Branko, koji poznaje Anin javni ključ K_{EA} , želi biti siguran:

- da je neki javni tekst P koji mu Ana šalje uistinu njezina poruka te
- da neki napadač nije tijekom komunikacije mijenjao sadržaj poruke, tj. da je primljena poruka besprijeckorna.

Osnove postupka za utvrđivanje besprijecknosti čini matematički postupak za izradu *kriptografskog sažetka* poruke (engl. *cryptographic digest*). Kriptografski sažetak izrađuje se funkcijom za izradu sažetka. To je *jednosmjerna funkcija* koja iz poruke proizvoljne duljine izračunava sažetak stalne duljine. Funkcija je jednosmjerna stoga što je izračunavanje sažetka vrlo lako, dok je iz sažetka praktički nemoguće izračunati izvorni tekst. Najjednostavniji oblik jednosmjerne funkcije jest uzastopna uporaba *XOR* funkcije na niz nakupina bitova koji se dobivaju dijeljenjem izvorne poruke na dijelove jednakе duljine.

Neka izvorni razgovijetni tekst bude podijeljen na dijelove P_i jednakе duljine:

$$P = P_0 P_1 P_2 \dots P_i \dots P_M.$$

Tada se sažetak S , duljine jednakе duljini dijelova P_i , dobiva uzastopnom primjenom funkcije *XOR*:

$$S = P_M \oplus (P_{M-1} \oplus (P_{M-2} \oplus (P_1 \oplus P_0))) = H(P).$$

Iz sažetka S praktički je nemoguće odrediti izvorni tekst P . Ako se u razgovijetnom tekstu promijeni ma i jedan bit, to će utjecati na vrijednost sažetka. Napadač koji bi htio promijeniti tekst P morao bi promijeniti i sažetak $H(P)$.

Ana može kriptirati sažetak poruke svojim privatnim ključem K_{DA} (koji inače služi za dekriptiranje!) i poslati poruku:

$$M = (P, RSA(H(P), K_{DA})).$$

Branko Aninim javnim ključem K_{EA} (koji inače služi za kriptiranje!) obavi dekriptiranje:

$$H(P) = RSA^{-1}(RSA(H(P), K_{DA}), K_{EA}).$$

Ovdje je iskorišteno svojstvo RSA kriptiranja da se ključevi kriptiranja i dekriptiranja mogu zamijeniti!

Nakon toga Branko može iz razgovijetnog teksta P koji je primio sam izračunati sažetak i usporediti ga s dekodiranim sažetkom S koji je poslala Ana.

Branko je na taj način saznao dvije činjenice:



- da je poruku uistinu poslala Ana jer samo ona i nitko drugi ne poznaje njezin privatni ključ K_{DA} ;
- da je prispjela poruka P besprijekorna.

Kriptirani sažetak čini svojevrsni potpis, tzv. *digitalni potpis* (engl. *digital signature*) koji je Ana poslala uz poruku. Detaljni oblik ovog digitalnog potpisa ovisi o sadržaju poruke. On je za svaku poruku drukčiji. Vidjet ćemo kasnije da se u javnim sustavima upotrebljava normirani oblik digitalnog potpisa. Ovaj se način potpisivanja poruke može kombinirati s mehanizmom digitalne omotnice kako bi se uz besprijekornost zadovoljilo i svojstvo tajnosti.

11.6.2. Digitalni pečat

Digitalni pečat je potpisana poruka u digitalnoj omotnici, odnosno zapečaćena digitalna omotnica.

Kada Ana želi poslati potpisušanu poruku u digitalnoj omotnici, ona:

- odabere proizvoljni simetrični ključ K ;
- kriptira tekst P simetričnom funkcijom, primjerice DES , i dobiva

$$C_1 = DES(P, K),$$

- kriptira Brankovim javnim ključem tajni ključ K i dobiva:

$$C_2 = RSA(K, K_{EB}),$$

- izradi sažetak $S = H(C_1, C_2)$ ⁵, kriptira ga svojim privatnim ključem i dobiva

$$C_3 = RSA(S, K_{DA}),$$

- šalje Branku poruku $M = (C_1, C_2, C_3)$.

Branko nakon primitka poruke M :

- svojim privatnim ključem K_{DB} dekriptira C_2 i dobiva tajni ključ:

$$K = RSA^{-1}(C_2, K_{DB}),$$

- dobivenim tajnim ključem dekriptira C_1 i dobiva razgovijetni tekst:

$$P = DES^{-1}(C_1, K),$$

- dekriptira C_3 Aninim javnim ključem K_{EA} i saznaće:

$$S = RSA^{-1}(C_3, K_{EA}),$$

⁵ Poruka (C_1, C_2) digitalna je omotnica, dakle računa se sažetak digitalne omotnice pa je digitalni pečat digitalno potpisana digitalna omotnica.

- izračunava sažetak dobivenog teksta $H(C_1, C_2)$ i uspoređuje ga s dobivenim sažetkom S ;
- ako je izračunati sažetak $H(P)$ jednak pristiglom sažetku S , Branko je siguran da mu je upravo Ana poslala tajni vjerodostojni tekst.

11.6.3. Funkcije za izračunavanje sažetka, funkcije sažimanja

XOR funkcija izrađuje vrlo jednostavan sažetak. Za kratke bi tekstove izrada sažetka zahtijevala samo nekoliko koraka njezine primjene tako da bi se iz sažetka možda moglo rekonstruirati izvorni tekst. Zbog toga su mnogi autori konstruirali složenije funkcije za izradu sažetka. Takve se funkcije nazivaju i *funkcijama sažimanja* (engl. *hash function* (engl. *hash* – sjeckati, kosati, sitno izrezati, zbrkati, smješati)). Spomenut ćemo samo dvije od njih, i to: *MD5* i *SHA*.

Funkcija sažimanja MD5

Ta je funkcija kosanja jedna od varijanti funkcijâ s kraticom *MD* (*MD* dolazi od engl. *Message Digest* – sažetak poruke). *MD5* proizvodi sažetak duljine 128 bitova.

Izvorni tekst dijeli se na blokove duljine 512 bitova. Zadnji blok teksta koji ne mora biti potpun nadopunjuje se na 512 bitova tako da se:

- iza zadnjeg bita teksta dodaje jedna jedinica;
- nakon te jedinice upisuje se toliko nula da u bloku preostanu 64 bita;
- u te se preostale bitove upisuje bitovna duljina izvorne poruke (bez nadopunjujućih bitova).

Svaki se blok dijeli na 16 podblokova duljine 32 bita koji se mogu nazvati:

$$M_0, M_1, M_2, \dots, M_{15}.$$

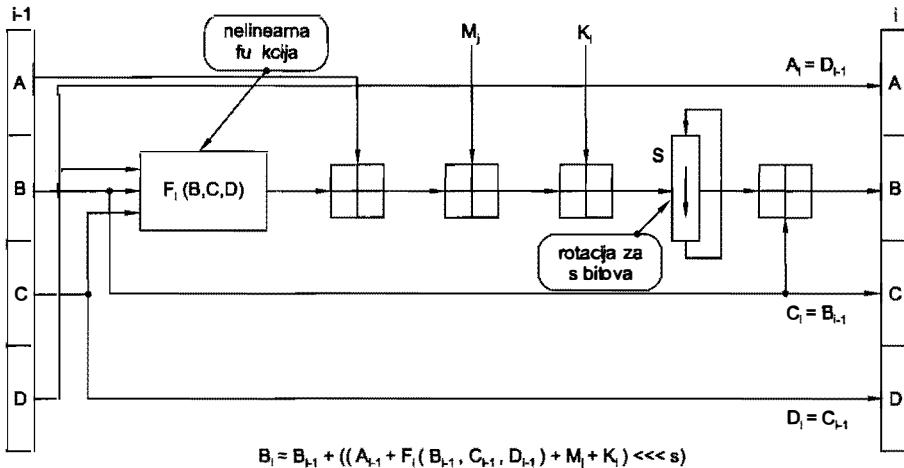
Svaki podblok M_j sudjeluje u izračunavanju sažetka četiri puta te se izračunavanje obavlja u 64 koraka podijeljena u četiri kruga. U svakom se koraku i ($1 \leq i \leq 64$) u izračunavanje uključuje i 32-bitovna konstanta K_i , koja se dobiva kao:

$$K_i = 2^{32} \cdot \text{abs}(\sin(i)).$$

Sažetak S od 128 bitova sastoji se od nadovezane četiri 32-bitovne varijable A , B , C i D . One se početno inicijaliziraju s konstantama:

$$A_0 = 01234567_{16}, \quad B_0 = 89ABCDEF_{16}, \quad C_0 = FEDCBA98_{16}, \quad D_0 = 76543210_{16}.$$

U i -tom se koraku izračunava:



Slika 11.22. MD5 postupak sažimanja poruke u i -tom koraku

U prvom se krugu ($1 \leq i \leq 16$) upotrebljava nelinearna funkcija

$$F_i(X, Y, Z) = (X \wedge Y) \vee ((\neg X) \wedge Z).$$

Podblokovi se dovode redom:

$$\begin{aligned} &M_0, M_1, M_2, M_3, M_4, M_5, M_6, M_7, \\ &M_8, M_9, M_{10}, M_{11}, M_{12}, M_{13}, M_{14}, M_{15}. \end{aligned}$$

a rotacije u lijevo obavljaju se za $s = 5, s = 19, s = 14, s = 20$ bitova za po četiri uzastopna koraka, što se zbiva četiri puta.

U drugom se krugu ($17 \leq i \leq 32$) upotrebljava funkcija:

$$F_i(X, Y, Z) = (X \wedge Z) \vee (Y \wedge (\neg Z)).$$

Podblokovi se dovode redom:

$$\begin{aligned} &M_1, M_6, M_{11}, M_0, M_5, M_{10}, M_{15}, M_4, \\ &M_9, M_{14}, M_3, M_8, M_{13}, M_2, M_7, M_{12}, \end{aligned}$$

a rotacije u lijevo obavljaju se za $5, 19, 14$ i 20 bitova za po četiri uzastopna koraka (i to četiri puta).

U trećem se krugu ($33 \leq i \leq 48$) upotrebljava funkcija:

$$F_i(X, Y, Z) = X \oplus Y \oplus Z.$$

Podblokovi se dovode redom:

$$\begin{aligned} &M_5, M_8, M_{11}, M_{14}, M_1, M_4, M_7, M_{10}, \\ &M_{13}, M_0, M_3, M_6, M_9, M_{12}, M_{15}, M_2, \end{aligned}$$

a rotacije u lijevo obavljaju se za 4, 11, 16 i 23 bitova za po četiri uzastopna koraka (i to četiri puta).

U četvrtom se krugu ($49 \leq i \leq 64$) upotrebljava funkcija:

$$F_i(X, Y, Z) = Y \oplus (X \vee (\neg Z)).$$

Podblokovi se dovode redom:

$$\begin{aligned} M_0, M_7, M_{14}, M_5, M_{12}, M_3, M_{10}, M_1, \\ M_8, M_{15}, M_6, M_{13}, M_4, M_{11}, M_2, M_9, \end{aligned}$$

a rotacije u lijevo obavljaju se za 6, 10, 15 i 21 bit za po četiri uzastopna koraka (četiri puta). Konačnim vrijednostima varijabli A , B , C i D , koje se nadovezuju u 128-bitovni sažetak, pribrajaju se na kraju konstante A_0 , B_0 , C_0 i D_0 , što daje:

$$\begin{aligned} A &= A_{64} + A_0, & B &= B_{64} + B_0, \\ C &= C_{64} + C_0, & D &= D_{64} + D_0. \end{aligned}$$

Rezultat prethodne iteracije ($A_{i-1}B_{i-1}C_{i-1}D_{i-1}$) je ulaz u sljedeću iteraciju ($A_iB_iC_iD_i$).

Funkcija sažimanja SHA

Funkcija sažimanja *Secure Hash Algorithm (SHA)* proizvodi 160-bitovni sažetak. Izvorni tekst dijeli se, kao i kod *MD5* postupka, na blokove duljine 512 bitova. Zadnji blok teksta, koji ne mora biti potpun, nadopunjuje se na 512 bitova na jednak način kao i kod *MD5* postupka. Sažetak S od 160 bitova sastoji se od 5 nadovezanih 32-bitovnih varijabli A , B , C , D i E koje se inicijaliziraju s vrijednostima:

$$\begin{aligned} A_0 &= 67452301_{16}, & B_0 &= EFC DAB89_{16}, \\ C_0 &= 98BADC F E_{16}, & D_0 &= 10325476_{16}, & E_0 &= C3D2E1F0_{16}. \end{aligned}$$

MD5 postupak sastoji se od 4 kruga od po 16 koraka, dok *SHA* postupak ima također 4 kruga ali s 20 koraka, tj. ukupno 80 koraka. Iz podblokova duljine 32 bita:

$$M_0, M_1, M_2, \dots, M_{15}$$

izračunava se 80 riječi W_i duljine 32 bita na sljedeći način:

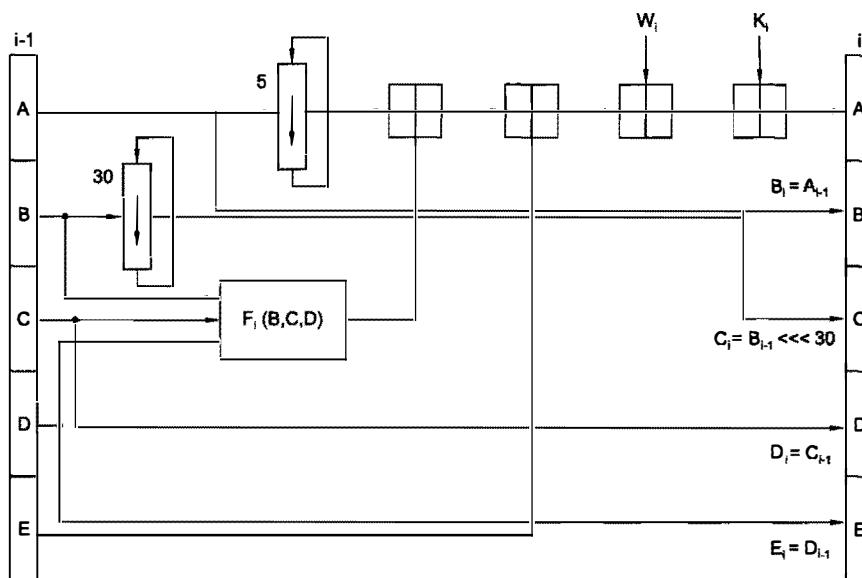
$$\begin{aligned} W_i &= M_{i-1} & \text{za } 1 \leq i \leq 16, \\ W_i &= (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) <<< 1 & \text{za } 17 \leq i \leq 80. \end{aligned}$$

Nadalje, u postupku se upotrebljavaju samo četiri konstante, i to:

$$\begin{aligned} K_i &= 5A827999_{16} & \text{za } 1 \leq i \leq 20, \\ K_i &= 6ED9EBA1_{16} & \text{za } 21 \leq i \leq 40, \\ K_i &= 8F1BBCDC_{16} & \text{za } 41 \leq i \leq 60, \\ K_i &= CA62C1D6_{16} & \text{za } 61 \leq i \leq 80. \end{aligned}$$

SHA koristi sljedeće četiri nelinearne funkcije:

$$\begin{aligned} F_i(X, Y, Z) &= (X \wedge Y) \vee ((\neg X) \wedge Z) & \text{za } 1 \leq i \leq 20, \\ F_i(X, Y, Z) &= X \oplus Y \oplus Z & \text{za } 21 \leq i \leq 40, \\ F_i(X, Y, Z) &= (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) & \text{za } 41 \leq i \leq 60, \\ F_i(X, Y, Z) &= X \oplus Y \oplus Z & \text{za } 61 \leq i \leq 80. \end{aligned}$$



Slika 11.23. SHA postupak sažimanja poruke u i -tom koraku

U i -tom se koraku obavlja sljedeće:

$$\begin{aligned} A_i &= (A_{i-1} <<< 5) + F_i(B_{i-1}, C_{i-1}, D_{i-1}) + E_{i-1} + W_i + K_i \\ B_i &= A_{i-1} \\ C_i &= B_{i-1} <<< 30 \\ D_i &= C_{i-1} \\ E_i &= D_{i-1}. \end{aligned}$$

Sažetak poruke čine nadovezane vrijednosti:

$$S = A_{80}B_{80}C_{80}D_{80}E_{80}.$$

Sažeci poruka poslužit će i za uspostavljanje *infrastrukture javnih ključeva* (engl. *Public Key Infrastructure – PKI*) o kojoj ćemo govoriti kasnije.

11.6.4. Važna svojstva funkcija za izračunavanje sažetka poruke

Funkcije sažimanja ocjenjuju se po sljedećim svojstvima:

Otpornost na izračunavanje originala (engl. preimage resistance)

Prvi zahtjev koji se postavlja na funkciju za izračunavanje sažetka poruke jest nemožnost izračunavanja originalnog teksta poruke M iz sažetka poruke H . Odnosno, za

poznati sažetak $H = h(M)$ ne smije postojati inverzna funkcija h^{-1} s pomoću koje je moguće izračunati izvornu poruku $M = h^{-1}(H)$.

Za velike poruke taj je zahtjev ispunjen na trivijalan način jer je intuitivno jasno da iz malog sažetka nije moguće izračunati originalnu poruku koja je mnogo veća od sažetka. Međutim, taj zahtjev mora biti ispunjen za sve poruke pa i za one male (primjerice i za manje poruke od sažetka).

Idealna funkcija za izračunavanje sažetka poruke morala bi za svake dvije različite poruke dati dva različita sažetka. S druge strane, sažetak je ograničene duljine od n bitova. Dakle, broj različitih sažetaka ograničen je na 2^n mogućnosti. Budući da postoji neograničen broj različitih poruka, postoji i beskonačan broj različitih poruka koje daju isti sažetak. Unatoč tomu, funkcija za izračunavanje sažetka poruke mora zadovoljavati dva dodatna zahtjeva: otpornost na izračunavanje poruke koja daje isti sažetak te otpornost na kolizije.

Otpornost na izračunavanje poruke koja daje isti sažetak (engl. 2nd preimage resistance)

Za poznati par poruka M i njen sažetak $H = h(M)$ nije nemoguće pronaći neku drugu poruku M' koja daje isti sažetak H .

Otpornost na kolizije (engl. collision resistance)

Ne smije biti moguće pronaći dvije različite poruke M_1 i M_2 za koje se dobiva isti sažetak $h(M_1) = h(M_2)$.

Na prvi se pogled može pogrešno zaključiti da su zadnja dva zahtjeva jednaka jer u oba se slučaja ne smije moći pronaći različite poruke koje daju isti sažetak. Međutim, s gledišta napadača mnogo je jednostavnije pronaći bilo koje dvije različite poruke (pa izgledale one kao slučajni niz bitova), nego pronaći neku drugu poruku koja daje isti sažetak, ako je originalna poruka zadana, a time i njen sažetak. Dakle, lakše je izvesti napad koji narušava svojstvo otpornosti na kolizije od napada koji izračunava poruku koja daje isti sažetak.

11.7. Sigurnosni protokoli

11.7.1. Diffie–Hellmanov postupak za razmjenu tajnog ključa

Kod opisa digitalne omotnice ustanovili smo da Ana, kada to zaželi, može uspostaviti s Brankom simetrični kriptosustav tako da:

- odabere tajni ključ K ;
- kriptira ga Brankovim javnim ključem $RSA(K, K_{EB})$ i pošalje to kao poruku Branku.

Branko nakon primjeka poruke saznaće tajni ključ K tako da svojim privatnim ključem dekriptira primljenu poruku:

$$K = RSA^{-1}(RSA(K, K_{EB}), K_{DB}).$$

Međutim, ako Ana i Branko nisu uključeni u RSA asimetrični kriptosustav, onda takav način razmjene ključeva nije moguć.

Diffie i Hellman objavili su dvije godine prije pojave RSA sustava, još 1976. godine svoj postupak za uspostavu simetričnog kriptosustava između dva partnera. Postupak se zasniva na bitno različitoj složenosti izračunavanja modularne potencije i izračunavanja diskretnog logaritma.

Dva se sudionika u komunikaciji unaprijed slože o dva vrlo velika broja n i g , ali takva da je g relativno prost u odnosu na n , tj. najveći im je zajednički djelitelj $\text{ndz}(g, n) = 1$. Najpraktičnije je za n odabrati veliki prosti broj p . Ta dva broja ne moraju biti tajna. Oni se mogu javno objaviti. Postupak uspostave provodi se na sljedeći način:

- Ana koja želi uspostaviti komunikaciju s Brankom odabire veliki nasumični prirodni broj x i šalje Branku rezultat izračunavanja:

$$X = g^x \bmod p.$$

- Branko odabire veliki nasumični prirodni broj y i šalje Ani rezultat izračunavanja:

$$Y = g^y \bmod p.$$

- Ana izračunava:

$$K = Y^x \bmod p = (g^y)^x \bmod p = g^{xy} \bmod p.$$

- Slično, Branko izračunava:

$$K = X^y \bmod p = (g^x)^y \bmod p = g^{xy} \bmod p.$$

Izračunata vrijednost K je tajni ključ koji Ana i Branko mogu upotrebljavati za simetrično kriptiranje. Čak ako neki napadač prislушкиje komunikaciju i sazna vrijednosti g , p , X i Y , on ne može izračunati ključ K , osim ako mu ne uspije izračunati diskretni logaritam od X ili Y . S obzirom na to da je to izražavanje vrlo teško, ključ K ostaje tajnim.

PRIMJER 11.10.

Neka se Ana i Branko slože i objave brojeve $p = 47$ i $g = 40$. Ana odabire broj $x = 35$ i pošalje Branku:

$$X = 40^{35} \bmod 47 = 30.$$



Branko odabire broj $y = 27$ i pošalje Ani:

$$Y = 40^{27} \bmod 47 = 43.$$

Nakon toga, Ana izračuna:

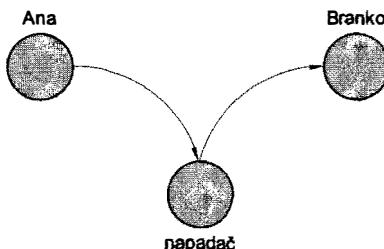
$$K = Y^x \bmod p = 43^{35} \bmod 47 = 45.$$

Isto tako, Branko izračuna:

$$K = X^y \bmod p = 30^{27} \bmod 47 = 45.$$

Na taj su se način Ana i Branko složili da im je tajni ključ jednak $K = 45$.

Diffie-Helmanov postupak osjetljiv je na napade kada uljez – čovjek u sredini (engl. *man in the middle*) – može presresti Anine i Brankove poruke s mreže i nadomjestiti ih svojim porukama.



Slika 11.24. Napad "čovjek u sredini"

Napadač odabire nasumični broj z i poznavajući objavljene brojeve p i g izračuna:

$$Z = g^z \bmod p.$$

Kada Ana pošalje vrijednost X , napadač kod sebe pohrani tu vrijednost i pošalje Branku vrijednost Z . Jednako tako on presretne Brankovu poruku Y , zadrži je za sebe i Ani također pošalje poruku Z . Ana i Branko misle da su dobili informacije jedan od drugog, ali su ih zapravo dobili od napadača. Napadač sebi izračuna ključeve K_A i K_B :

$$\begin{aligned} K_A &= X^z \bmod p = (g^x)^z \bmod p = g^{xz} \bmod p \\ K_B &= Y^z \bmod p = (g^y)^z \bmod p = g^{yz} \bmod p. \end{aligned}$$

Ana sebi izračuna ključ:

$$K_A = Z^x \bmod p = (g^z)^x \bmod p = g^{xz} \bmod p,$$

a Branko ključ:

$$K_B = Z^y \bmod p = (g^z)^y \bmod p = g^{yz} \bmod p.$$

Ako nakon razmjene ključeva Ana i Branko nastave komunicirati koristeći izračunate tajne ključeve, napadač može narušiti i tajnost i besprijekornost komuniciranja.

Kada Ana pošalje kriptiranu poruku:

$$M_1 = E(P, K_A),$$

napadač je presretne, dekriptira i saznaje njezin sadržaj:

$$P = D(E(P, K_A), K_A).$$

On sada može taj isti tekst P ili neki izmišljeni tekst P' kriptirati i poslati Branku poruku:

$$M_2 = E(P', K_B).$$

Branko misli da je dobio poruku od Ane i svojim ključem K_B dobiva:

$$P' = D(E(P', K_B), K_B).$$

Jednako može proteći i komunikacija u suprotnom smjeru. Niti Ana niti Branko neće otkriti da se u njihovu komunikaciju umiješao napadač. Ovakve i slične razmjene ključeva treba dodatno zaštiti postupkom autentifikacije sudionika u komunikaciji.

Time smo zaključili kratki pregled dijela matematičke podloge potrebne za razumijevanje osnovnih kriptografskih sustava. U daljnjim čemo odlomcima pogledati neke njihove primjene u ostvarivanju sigurnosnih protokola.

11.7.2. Raspodjela ključeva u zatvorenom simetričnom kriptosustavu

Raspodjela ključeva prema Needhamu i Schroederu

Ako su dva sudionika u komunikaciji koja žele koristiti simetrični kriptosustav moraju najprije razmijeniti tajni ključ kriptiranja. Vidjeli smo u odjeljku 11.5.2. da se tajni ključ može prenijeti ako su sudionici uključeni u asimetrični kriptosustav i poznati su im javni ključevi drugog sudionika. Razmjena ključeva može se obaviti i Diffie-Hellmanovim postupkom ako se u njega ugradi dodatna zaštita od napadača u sredini.

Ako se želi uspostaviti sigurna komunikacija između manje skupine sudionika potrebno je ključeve razmijeniti na sigurni način za sve moguće parove sudionika (recimo, osobnim raznošenjem). Za N sudionika trebalo bi imati ukupno $N(N - 1)/2$ tajnih ključeva, a svaki bi sudionik morao pohraniti $N - 1$ ključeva. Time je, međutim, ozbiljno ugrožena sigurnost!

Jedno od mogućih rješenja jest uspostava centra za raspodjelu ključeva (engl. *Key Distribution Center – KDC*). Centar za raspodjelu ključeva jedan je pouzdani poslužitelj koji je zaštićen od vanjskih opasnosti i u kojeg svi ostali sudionici imaju povjerenje. Svi potencijalni sudionici u komuniciranju moraju se unaprijed prijaviti i pritom im se dodjeljuje tajni ključ za komuniciranje s centrom KDC. Dakle, sudionik I dobiva svoj identifikator ID_I i tajni ključ K_I . KDC obznanjuje identifikatore svih prijavljenih sudionika, a zadržava u tajnosti pripadnu tablicu tajnih ključeva.

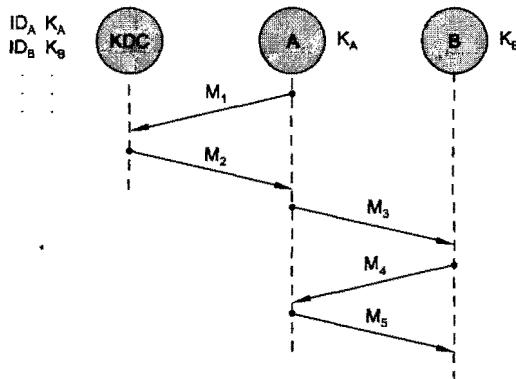
Kada sudionik A zaželi komunicirati sa sudionikom B , tada će KDC prema protokolu koji su predložili Needham i Schroeder dodijeliti tajni ključ K_{AB} . Protokol se sastoji od sljedećih šest koraka u kojima se razmjenjuje pet poruka:

1. Sudionik A šalje u KDC u razgovijetnom obliku poruku:

$$M_1 = (R_A, ID_A, ID_B),$$

gdje je:

- R_A – kôd zahtjeva za dodjelu ključa,
- ID_A – identifikator sudionika A,
- ID_B – identifikator sudionika B.



Slika 11.25. Protokol prema Needhamu i Schroederu

2. KDC stvara ključ K_{AB} , u svojoj tablici na temelju ID_A i ID_B pronađi ključeve K_A i K_B te:

- uporabom ključa K_B kriptira par (ID_A, K_{AB}) u kriptirani tekstu:

$$C_1 = E((ID_A, K_{AB}), K_B),$$

- uporabom ključa K_A kriptira četvorku R_A, ID_A, K_{AB}, C_1 u poruku:

$$M_2 = E((R_A, ID_A, K_{AB}, C_1), K_A).$$

i šalje je sudioniku A.

3. Sudionik A svojim ključem K_A dekriptira poruku M_2 i utvrđuje na temelju R_A i ID_A da je to odgovor na poruku M_1 te:

- zadržava ključ K_{AB} ;
- šalje sudioniku B poruku $M_3 = C_1$.

4. Sudionik B svojim ključem K_B dekriptira C_1 i saznaće:

$$(ID_A, K_{AB}) = D(C_1, K_B) = D(E((ID_A, K_{AB}), K_B), K_B),$$

tj. zna da sudionik A želi s njime komunicirati i da je KDC za tu komunikaciju do-dijelio tajni ključ K_{AB} . Međutim, sudionik B želi provjeriti da li sudionik A uistinu znade tajni ključ K_{AB} te:

- generira slučajni broj R ;
- šalje sudioniku A poruku $M_4 = E(R, K_{AB})$.

5. Sudionik A nakon primitka dekriptira poruku M_4 i saznaje:

$$R = D(E(R, K_{AB}), K_{AB}).$$

On zatim unaprijed dogovorenom funkcijom F izračunava $F(R)$, kriptira rezultat i šalje poruku:

$$M_5 = E(F(R), K_{AB}).$$

6. Sudionik B dekriptira poruku M_5 i saznaje:

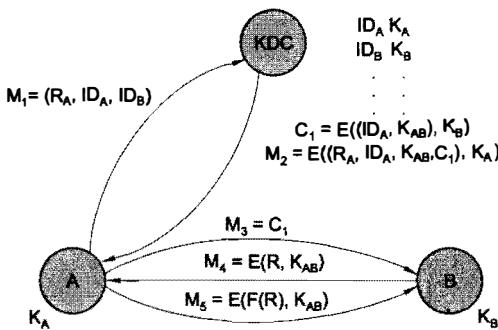
$$F(R) = D(E(F(R), K_{AB}), K_{AB}).$$

On zatim inverznom funkcijom izračunava $F^{-1}(R)$ i uspoređuje taj rezultat s R . Ako je:

$$F^{-1}(R) = R,$$

sigurni kanal s tajnim ključem K_{AB} je uspostavljen.

Opisani protokol može se sažeto prikazati slikom 11.26. Pritom krugove i strelice na slici ne treba interpretirati kao stanja i prijelaze iz stanja u stanje. U ovom slučaju krugovi predstavljaju sudionike u komunikaciji, a strelica događaj slanja poruke.



Slika 11.26. Raspodjela ključeva u zatvorenom simetričnom kriptosustavu prema Needhamu i Schroederu

Broj R koji se upotrebljava u svojevrsnoj međusobnoj autentifikaciji sudionika naziva se *nonce*, što je posebno skovana prigodna riječ od engleskog *number used only once* – broj upotrijebljen samo jednom.

Ključ K_{AB} može se uspostaviti za svaku komunikaciju posebno ili se može upotrebljavati neko propisano vrijeme. Uvođenjem vremenske oznake T (engl. *time stamp*) može se trajanje valjanosti ključa svesti ne neko dogovoreno razdoblje.

Centar za raspodjelu ključeva KDC mora pri oblikovanju poruke u nju ugraditi i vremensku oznaku T , tako da oblikuje:

$$C_1 = E((ID_A, K_{AB}, T), K_B),$$

odnosno:

$$M_2 = E((R_A, ID_A, K_{AB}, C_1, T), K_A).$$

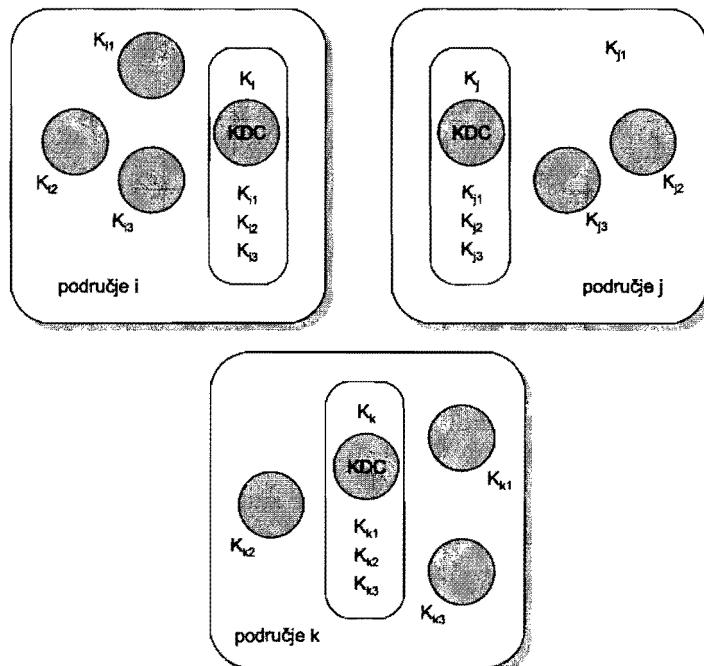
Tako i sudionik A i sudionik B tako uz ključ K_{AB} saznaju i T .

Uvođenje vremenske oznake povećava sigurnost. Naime, ako neki napadač uhvati poruku M_3 i uspije ju nekako dekriptirati, on će saznati ključ K_{AB} . Nakon toga on se može višekratno lažno predstaviti kao sudionik A . Ta se opasnost smanjuje ako se ograniči trajanje valjanosti ključa na neko unaprijed dogovoren vrijeme.

Nedostatak uporabe jednog centra za raspodjelu ključeva jest smanjenja pouzdanost i preopterećenost. Ako čvor poslužitelja zakaže, čitav se sustav raspodjele ključeva raspada. Najveća se pouzdanost (ali ne nužno i sigurnost!) postiže kada svi sudionici unaprijed znaju ključeve za komunikaciju sa svim ostalim sudionicima. Već smo rekli da u sustavu s N sudionika moramo imati ukupno $N(N - 1)/2$ tajnih ključeva, a svaki bi sudionik morao pohraniti $N - 1$ ključeva. U slučaju kvara jednog čvora svi ostali sudionici mogu nastaviti komunicirati. Međutim, nedostaci ovog rješenja jesu prevelik broj ključeva i praktički nemoguće ostvarenje njihova vremenskog ograničavanja.

Raspodijeljena raspodjela ključeva

Jedno od mogućih rješenja za ublažavanje tih poteškoća jest ostvarenje raspodijeljenog centra za dodjelu ključeva. Sudionici se mogu podijeliti na područja od kojih svako ima svoj centar za raspodjelu ključeva KDC_i . U sustavu se moraju na siguran način uspostaviti tajni ključevi za komuniciranje svakog sudionika sa svojim KDC -om i tajni ključevi za svaki par KDC . Dakle, svaki KDC čuva tajne ključeve za komuniciranje sa sudionicima iz svog područja te tajne ključeve za komuniciranje sa svim ostalim centrima.



Slika 11.27. Raspodjela sudionika na područja

**PRIMJER 11.11.**

Pretpostavimo da u zatvorenom sustavu imamo 100 sudionika u komuniciranju. Koliko tajnih ključeva treba unaprijed podijeliti:

- bez centra za raspodjelju ključeva,
- s jednim KDC centrom,
- s deset područnih centara kojima je pridruženo po deset sudionika?

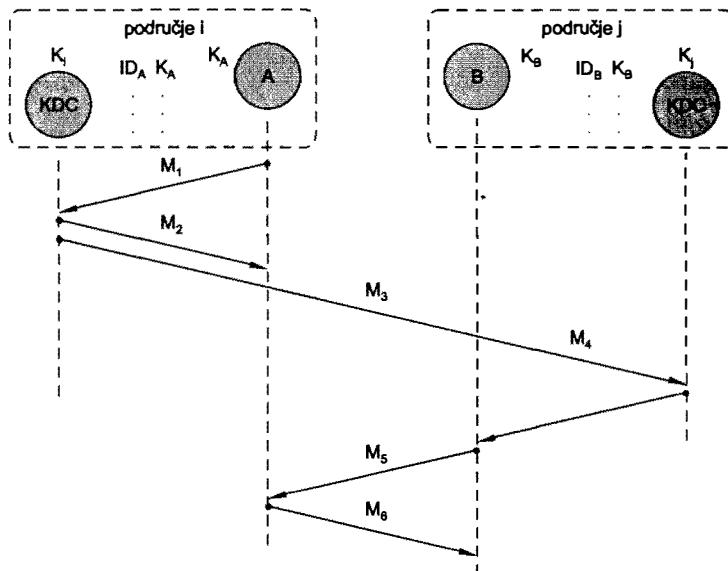
Odgovor:

a) Bez centra treba unaprijed podijeliti $100 \cdot 99/2 = 4950$ ključeva. Svaki sudionik mora čuvati 99 ključeva.

b) S jednim KDC-om treba unaprijed podijeliti 100 ključeva. Svaki sudionik čuva samo svoj (1) ključ, a KDC 100 ključeva.

c) U raspodijeljenom sustavu postoji $10 \cdot 9/2 = 45$ ključeva za komunikaciju između centara. Svaki KDC čuva njih 9. Nadalje, u svakom područnom KDC-u postoji 10 ključeva za komuniciranje sa sudionicima unutar područja. Svaki sudionik čuva samo svoj (1) ključ za komuniciranje s područnim centrom. Područni KDC mora čuvati i tih 10 ključeva, tako da on čuva $9 + 10 = 19$ ključeva. U tom se sustavu koristi ukupno $10 \cdot 10 + 45 = 145$ ključeva.

Identifikatori svih sudionika moraju biti poznati i dostupni unutar cijelog sustava. Protokol za dodjelu ključa obavlja se na jednak način kao u centraliziranom protokolu ako se oba sudionika nalaze u istom području. Ako se sudionici nalaze u različitim područjima,



Slika 11.28. Sekvencijski dijagram raspodjele ključeva u raspodijeljenom sustavu

onda se uspostavljanje sigurnog kanala može obaviti razmjenom šest poruka. Protokol se sastoji od sljedećih šest koraka u kojima se razmjenjuje šest poruka:

1. Sudionik A šalje u svoj KDC_i u razgovijetnom obliku poruku:

$$M_1 = (R_A, ID_A, ID_B)$$

gdje je:

R_A –kôd zahtjeva za dodjelu ključa,
 ID_A –identifikator sudionika A ,
 ID_B –identifikator sudionika B .

2. KDC_i kreira ključ K_{AB} , u svojoj tablici na temelju ID_A ključ pronalazi K_A i na temelju ID_B pronalazi da se sudionik B nalazi u području j te:

- sudioniku A šalje poruku:

$$M_2 = E((R_A, ID_A, K_{AB}), K_A),$$

- uporabom ključa K_j kriptira trojku K_{AB}, ID_A, ID_B u poruku:

$$M_3 = E((K_{AB}, ID_A, ID_B), K_j)$$

i šalje ju u KDC_j .

3. KDC_j svojim ključem K_j dekriptira dobivenu poruku i dobiva:

$$(K_{AB}, ID_A, ID_B) = D(E((K_{AB}, ID_A, ID_B), K_j), K_j).$$

On u svojoj tablici temeljem ID_B pronalazi ključ K_B te njime oblikuje poruku:

$$M_4 = E((ID_A, K_{AB}), K_B)$$

koju šalje sudioniku B .

4. Sudionik B svojim ključem K_B dekriptira M_4 i saznaće:

$$(ID_A, K_{AB}) = D(E((ID_A, K_{AB}), K_B), K_B),$$

tj. zna da sudionik A želi s njime komunicirati i da je za tu komunikaciju dodijeljen tajni ključ K_{AB} .

Sudionik B želi provjeriti da li sudionik A uistinu znade tajni ključ K_{AB} te:

- generira slučajni broj R ;
- šalje sudioniku B poruku:

$$M_5 = E(R, K_{AB}).$$

5. Sudionik A nakon primitka dekriptira poruku M_5 i saznaće:

$$R = D(E(R, K_{AB}), K_{AB}).$$

On zatim unaprijed dogovorenom funkcijom F izračunava $F(R)$, kriptira rezultat i šalje poruku:

$$M_6 = E(F(R), K_{AB}).$$

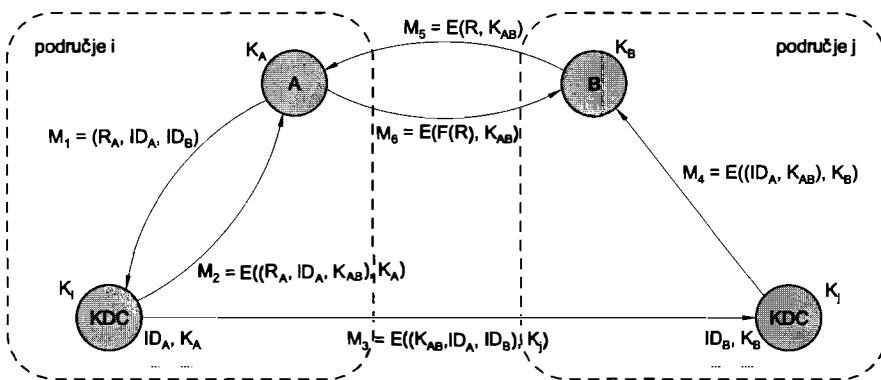
6. Sudionik B dekriptira poruku M_6 i saznaće:

$$F(R) = D(E(F(R), K_{AB}), K_{AB}).$$

On zatim inverznom funkcijom izračunava $F^{-1}(R)$ i uspoređuje taj rezultat s R . Ako je:

$$F^{-1}(R) = R,$$

sigurni kanal s tajnim ključem K_{AB} je uspostavljen.



Slika 11.29. Rasподjela ključeva u raspodijeljenom sustavu

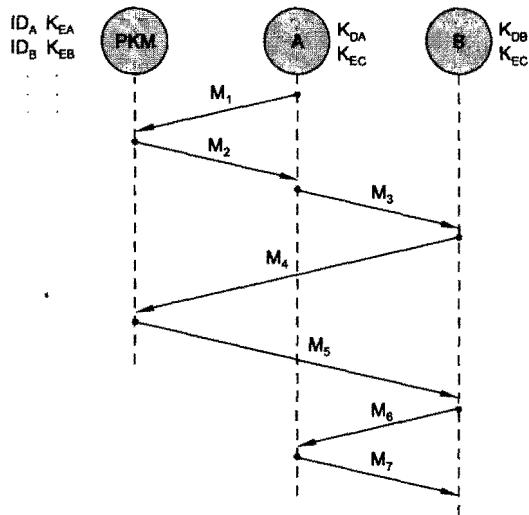
11.7.3. Rasподjela ključeva u zatvorenom asimetričnom kriptosustavu

U asimetričnom kriptosustavu raspodjeluju se samo javni ključevi. Ti ključevi ne moraju biti tajni i zbog toga se na prvi pogled čini da s njihovim prijenosom nema problema. U jednom zatvorenom sustavu svi potencijalni sudionici moraju se prijaviti. Prilikom prijava njima se dodjeljuje par ključeva. Svoj privatni ključ oni čuvaju kod sebe dok se pripadni javni ključ pohranjuje zajedno s njihovim identifikatorom u tablicama pouzdanog poslužitelja koji možemo nazvati centrom za raspodjelu javnih ključeva (engl. *public key manager* – *PKM*).

Kada sudionik A želi komunicirati sa sudionikom B sigurnim kanalom, on će zatražiti od *PKM*-a njegov javni ključ. Time se izbjegava mogućnost da neki napadač generira svoj par ključeva i ponudi svoj javni ključ te uspostavi prividno sigurni komunikacijski kanal sa sudionikom A . Dodatno je potrebno u postupak raspodjele, odnosno obznanjivanja javnih ključeva sudionika ugraditi autentifikacijske mehanizme.

Dakle, svakom sudioniku I prijavljenom u sustav dodjeljuje se javni K_{EI} i privatni K_{DI} ključ. Centar za raspodjelu ključeva također ima svoj javni ključ K_{EC} i privatni ključ

K_{DC} . Svaki sudionik sustava čuva svoj privatni ključ i javni ključ centra za dodjelu ključeva. Centar čuva svoj privatni ključ i tablicu u kojoj su uz identifikatore pohranjeni javni ključevi svih prijavljenih sudionika.



Slika 11.30. Sekvencijski dijagram raspodjele ključeva u zatvorenom asimetričnom kriptosustavu

Protokol za uspostavu sigurnog kanala između sudionika A i B , na zahtjev sudionika A , jest sljedeći:

1. Sudionik A šalje u PKM poruku:

$$M_1 = E((R_A, T_1, ID_A, ID_B), K_{EC}),$$

gdje je:

- | | |
|----------|-----------------------------------|
| R_A | – kôd zahtjeva za dodjelu ključa, |
| T_1 | – vremenska oznaka, |
| ID_A | – identifikator sudionika A , |
| ID_B | – identifikator sudionika B , |
| K_{EC} | – javni ključ centra. |

2. PKM s K_{DC} dekriptira poruku M_1 i saznaje R_A , T_1 , ID_A , ID_B . On zatim iz svoje tablice očita javne ključeve K_{EA} i K_{EB} te kriptira poruku:

$$M_2 = E((R_A, T_1, K_{EB}), K_{EA}),$$

i šalje je sudioniku A .

3. Nakon primitka poruke sudionik A svojim ključem K_{DA} dekriptira poruku M_2 i saznaje:

$$(R_A, T_1, K_{EB}),$$

te:

- usporedbom s izvornim vrijednostima R_A, T_1 utvrđuje da je poruka M_2 odgovor na njegovu poruku M_1 ;
- generira slučajni broj N_A i kriptira par (ID_A, N_A) javnim ključem sudionika B i šalje mu poruku:

$$M_3 = E((ID_A, N_A), K_{EB}).$$

4. Sudionik B svojim ključem K_{DB} dekriptira M_3 i saznaće:

$$(ID_A, N_A) = D(E((ID_A, N_A), K_{EB}), K_{DB}),$$

tj. zna da sudionik A želi s njime uspostaviti sigurni komunikacijski kanal. Sudionik B nakon toga šalje u centar sljedeću poruku kriptiranu javnim ključem centra:

$$M_4 = E((R_B, T_2, ID_A, ID_B), K_{EC}),$$

5. PKM s K_{DC} dekriptira poruku M_4 i saznaće R_B, T_2, ID_A, ID_B . On zatim iz svoje tablice očita javne ključeve K_{EA} i K_{EB} te kriptira poruku:

$$M_5 = E((R_B, T_2, K_{EA}), K_{EB}),$$

i šalje je sudioniku B .

6. Nakon primitka poruke sudionik B svojim ključem K_{DB} dekriptira poruku M_5 i saznaće:

$$(R_B, T_2, K_{EA}),$$

te:

- usporedbom s izvornim vrijednostima R_B, T_2 utvrđuje da je poruka M_5 odgovor na njegovu poruku M_4 ;
- generira slučajni broj N_B i kriptira par (N_A, N_B) javnim ključem sudionika A i šalje mu poruku:

$$M_6 = E((N_A, N_B), K_{EA}).$$

7. Sudionik A svojim ključem K_{DA} dekriptira M_6 i saznaće:

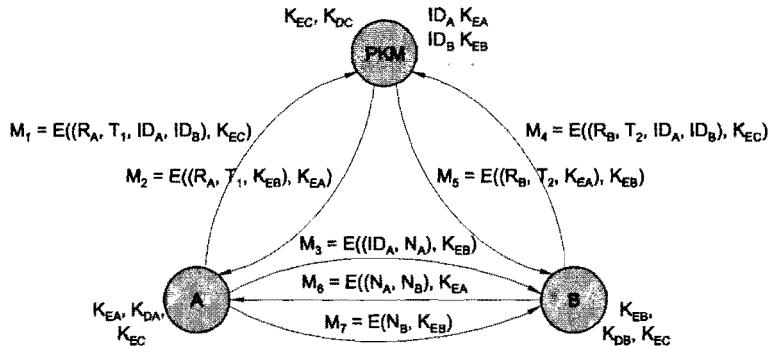
$$(N_A, N_B) = D(E((N_A, N_B), K_{EA}), K_{DA})$$

te:

- uspoređuje dobiveni N_A s izvorno odaslanim i ako se oni poklapaju, siguran je da je B autentičan;
- šalje sudioniku B kriptiranu poruku:

$$M_7 = E(N_B, K_{EB}).$$

8. Sudionik B svojim ključem K_{DB} dekriptira M_7 i dobiva $N_B = D(E(N_B, K_{EB}), K_{DB})$. Ako se dobiveni N_B podudara s izvornom vrijednošću, B je siguran da je A autentičan i sigurni komunikacijski kanal je uspostavljen.



Slika 11.31. Raspodjela ključeva u zatvorenom asimetričnom kriptosustavu

11.7.4. Autentifikacija u zatvorenim sustavima

Opisani modeli raspodjele ključeva u zatvorenim sustavima zasnovani na simetričnim ili asimetričnim kriptosustavima mogu poslužiti i za uspostavu sustava autentifikacije. Ustanovimo još jedanput da u zatvorenim sustavima svi sudionici moraju biti unaprijed prijavljeni te da prilikom prijave:

- u simetričnom sustavu dobivaju tajni ključ koji osim njih samih poznaje samo još centar za raspodjelu ključeva;
- u asimetričnom sustavu dobivaju svoj par javnog i privatnog ključa te javni ključ centra za raspodjelu ključeva (s tim da samo centar raspodjele ključeva ima pohranjene javne ključeve svih prijavljenih sudionika).

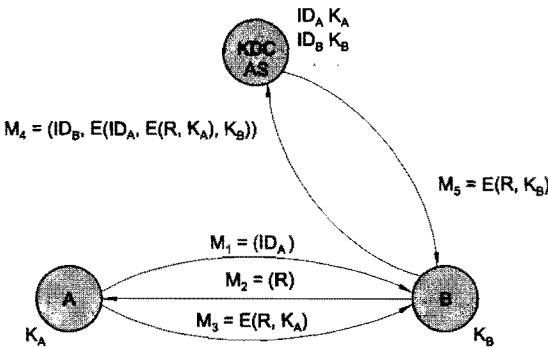
Centri za raspodjelu ključeva mogu se preimenovati u autentifikacijske poslužitelje (engl. *authentication server – AS*), a umjesto protokola za raspodjelu ključeva mogu se primijeniti (u načelu jednostavniji) protokoli za utvrđivanje autentičnosti.

Napomenimo da se u prethodno opisanim modelima za raspodjelu ključeva obavlja i autentifikacija.

Jednostrana autentifikacija u zatvorenom simetričnom kriptosustavu

Prepostavimo da postoji autentifikacijski poslužitelj *AS* koji, jednako kao *KDC* u sustavu za raspodjelu ključeva opisanom u odjeljku 11.7.2., sadrži identifikatore i tajne ključeve svih prijavljenih sudionika.

Kada sudionik *A* zaželi komunicirati sa sudionikom *B*, potonji će pokrenuti provjeru njegova identiteta, što će se obaviti s ukupno pet poruka.



Slika 11.32. Postupak jednostrane autentifikacije u zatvorenom simetričnom kriptosustavu

Protokol se sastoји од sljedećih šest koraka u kojima se obavlja razmjena pet poruka:

1. Sudionik *A* šalje u razgovijetnom obliku svoj identifikator sudioniku *B* u poruci:

$$M_1 = (ID_A).$$

2. Nakon primitka M_1 sudionik *B* generira slučajni broj R i šalje u razgovijetnom obliku sudioniku *A* poruku:

$$M_2 = (R).$$

3. Nakon primitka M_2 sudionik *A* kriptira R svojim tajnim ključem K_A i šalje poruku:

$$M_3 = E(R, K_A).$$

4. Sudionik *B*, nakon primitka M_3 , kriptira svojim tajnim ključem par $ID_A, E(R, K_A)$ i šalje autentifikacijskom poslužitelju *AS* poruku:

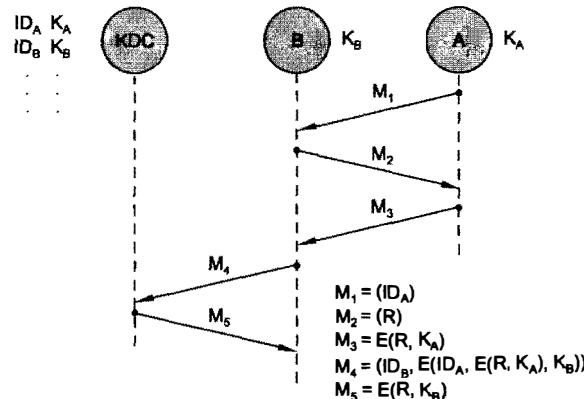
$$M_4 = (ID_B, E(ID_A, E(R, K_A), K_B)).$$

5. Nakon primitka M_4 poslužitelj *AS*:

- iz svoje tablice na temelju ID_B pročita K_B te dekriptira M_4 i saznaće ID_A , $E(R, K_A)$;
- pronalazi u svojoj tablici K_A i dekriptira $E(R, K_A)$ te saznaće $R = D(E(R, K_A), K_A)$;
- kriptira R s pomoću ključa K_B i šalje poruku:

$$M_5 = E(R, K_B).$$

6. Sudionik *B* dekriptira M_5 svojim tajnim ključem K_B i saznaće $R = D(E(R, K_B), K_B)$. Dobiveni R uspoređuje s originalom te prihvaća ili odbija sudionika *A*.



Slika 11.33. Sekvencijski dijagram u zatvorenom simetričnom kriptosustavu

Po uzoru na postupak raspodjele tajnih ključeva može se ovaj centralizirani postupak autentifikacije prerađiti u raspodijeljeni sustav, gdje više autentifikacijskih poslužitelja koji međusobno poznaju svoje tajne ključeve sudjeluje u prenošenju nasumičnog broja R .

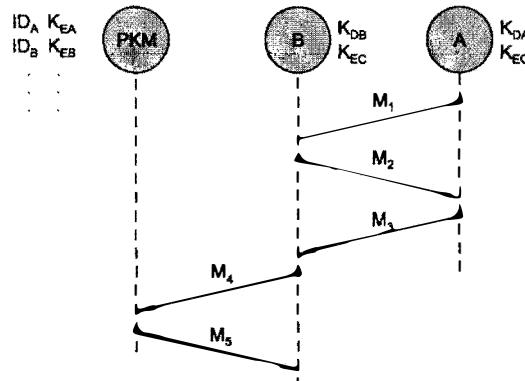
Jednostrana autentifikacija u zatvorenom asimetričnom sustavu

Prepostavimo da postoji autentifikacijski poslužitelj AS koji, jednako kao PKM u sustavu za raspodjelu ključeva opisanom u odjeljku 11.7.3., sadrži identifikatore i javne ključeve svih prijavljenih sudionika te svoj privatni ključ. Svaki prijavljeni sudionik sustava čuva svoj privatni ključ te javni ključ autentifikacijskog poslužitelja AS .

Protokol se sastoji od sljedećih šest koraka u kojima se obavlja razmjena pet poruka:

1. Sudionik A šalje u razgovijetnom obliku svoj identifikator sudioniku B u poruci:

$$M_1 = (ID_A).$$



Slika 11.34. Sekvencijski dijagram jednostrane autentifikacije u zatvorenom asimetričnom sustavu

2. Nakon primitka M_1 sudionik B generira slučajni broj R i šalje u razgovijetnom obliku sudioniku A poruku:

$$M_2 = (R).$$

3. Nakon primitka M_2 sudionik A kriptira svojim privatnim ključem R i šalje poruku:

$$M_3 = E(R, K_{DA}).$$

(Ovdje se koristi svojstvo asimetričnog sustava da se ključevi kriptiranja i dekriptiranja mogu zamijeniti!)

4. Sudionik B , nakon primitka M_3 , šalje autentifikacijskom poslužitelju AS u razgovijetnom obliku poruku:

$$M_4 = (ID_A, R_B)$$

gdje je R_B kôd kojim sudionik B zahtijeva od poslužitelja javni ključ sudionika s identifikatorom ID_A .

5. Nakon primitka M_4 poslužitelj AS :

- iz svoje tablice na temelju ID_A pročita K_{EA} ;
- s pomoću svog privatnog ključa K_{DC} kriptira par (ID_A, K_{EA}) i šalje poruku:

$$M_5 = E((ID_A, K_{EA}), K_{DC}).$$

6. Nakon primitka M_5 sudionik B :

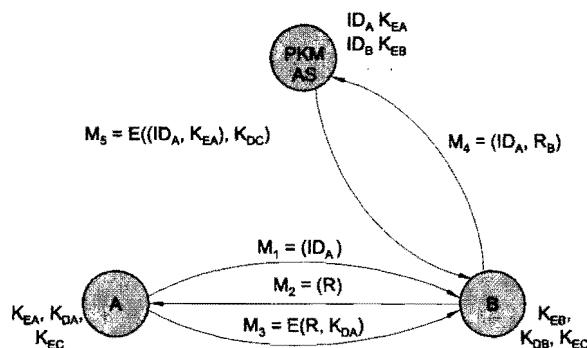
- javnim ključem poslužitelja dekriptira M_5 i dobiva:

$$(ID_A, K_{EA}) = D(E((ID_A, K_{EA}), K_{DC}), K_{EC}),$$

- s pomoću dobivenog ključa K_{EA} dekriptira poruku M_3 i saznaće:

$$R = D(E(R, K_{DA}), K_{EA}),$$

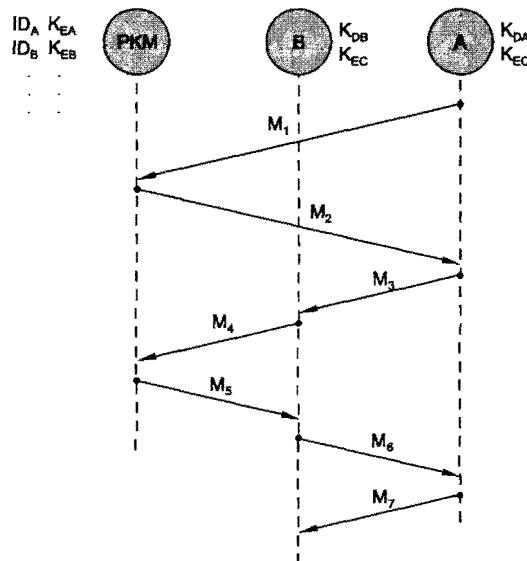
- dobiveni R uspoređuje s originalom te prihvaca ili odbacuje sudionika A .



Slika 11.35. Postupak jednostrane autentifikacije u zatvorenom asimetričnom kriptosustavu

Obostrana autentifikacija u zatvorenom asimetričnom sustavu

Obostrana autentifikacija mogla bi se obaviti tako da se najprije utvrdi autentičnost jednog sudionika i zatim na jednak način autentičnost drugog sudionika. Takvo bi rješenje zahtijevalo ukupno 10 poruka. Pokazuje se, međutim, da je obostranu autentifikaciju moguće provesti s ukupno 7 poruka.



Slika 11.36. Sekvenički dijagram obostrane autentifikacije u zatvorenom asimetričnom sustavu

Sustav je uspostavljen na jednak način kao i za jednostranu autentifikaciju i u njemu se odvija razmjena 7 poruka. Protokol se odvija u sljedećih osam koraka:

1. Kada sudionik A želi uspostaviti međusobnu vezu sa sudionikom B , on šalje auten-

tifikacijskom poslužitelju AS u razgovijetnom obliku poruku:

$$M_1 = (ID_A, ID_B, R_A),$$

gdje je:

R_A – kôd zahtjeva sudionika A .

2. Nakon primitka M_1 poslužitelj AS :

- iz svoje tablice na temelju ID_B dobavlja K_{EB} ;
- s pomoću svog privatnog ključa K_{DC} kriptira par (ID_B, K_{EB}) i šalje sudioniku A poruku:

$$M_2 = E((ID_B, K_{EB}), K_{DC}).$$

3. Nakon primitka M_2 sudionik A :

- s pomoću javnog ključa K_{EC} dekriptira M_2 i dobiva:

$$(ID_B, K_{EB}) = D(E((ID_B, K_{EB}), K_{DC}), K_{EC}),$$

- generira nasumični broj N_A ,
- s pomoću javnog ključa K_{EB} kriptira par (ID_A, N_A) i šalje sudioniku B poruku:

$$M_3 = E((ID_A, N_A), K_{EB}).$$

4. Nakon primitka M_3 sudionik B :

- s pomoću svog privatnog ključa K_{DB} dekriptira M_3 i dobiva:

$$(ID_A, N_A) = D(E((ID_A, N_A), K_{EB}), K_{DB}),$$

- s pomoću javnog ključa poslužitelja K_{EC} kriptira četvorku (ID_A, N_A, ID_B, R_B) i šalje poslužitelju poruku:

$$M_4 = E((ID_A, N_A, ID_B, R_B), K_{EC}),$$

gdje je R_B kôd zahtjeva sudionika B .

5. Nakon primitka M_4 autentifikacijski poslužitelj AS :

- s pomoću privatnog ključa K_{DC} dekriptira M_4 i dobiva:

$$(ID_A, N_A, ID_B, R_B),$$

- generira novi ključ za simetrično kriptiranje K ;
- oblikuje tri kriptirana teksta:

$$C_1 = E((ID_A, K_{EA}), K_{DC}),$$

$$C_2 = E((ID_B, K, N_A), K_{DC}),$$

$$C_3 = E(C_2, K_{EB}),$$

- šalje sudioniku B poruku:

$$M_5 = (C_1, C_3).$$

6. Nakon primitka M_5 sudionik B :

- s pomoću javnog ključa poslužitelja K_{EC} dekriptira C_1 i dobiva:

$$(ID_A, K_{EA}) = D(E((ID_A, K_{EA}), K_{DC}), K_{EC}),$$

- generira nasumični broj N_B ;
- s pomoću svog privatnog ključa K_{DB} dekriptira drugi dio poruke i dobiva $C_2 = D(E(C_2, K_{EB}), K_{DB})$;
- s pomoću javnog ključa K_{EA} kriptira par (C_2, N_B) i šalje sudioniku A poruku:

$$M_6 = E((C_2, N_B), K_{EA}).$$

7. Nakon primitka M_6 sudionik A :

- s pomoću svog privatnog ključa K_{DA} dekriptira M_6 i dobiva:

$$(C_2, N_B) = D(E((C_2, N_B), K_{EA}), K_{DA}),$$

- s pomoću javnog ključa poslužitelja K_{EC} dekriptira C_2 i dobiva

$$(ID_B, K, N_A) = D(E((ID_B, K, N_A), K_{DC}), K_{EC}),$$

- uspoređuje dobiveni N_A s izvornim i, ako su jednaki, utvrđuje autentičnost sudionika B ;
- kriptira N_B s pomoću dobivenog tajnog ključa K (primjerice DES funkcijom) i šalje sudioniku A poruku:

$$M_7 = DES(N_B, K).$$

8. Nakon primitka M_7 sudionik B :

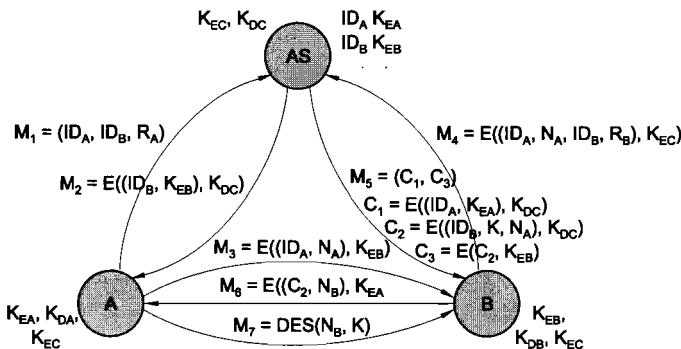
- s pomoću javnog ključa poslužitelja K_{EC} dekriptira C_2 kojeg je sačuvao iz točke 6. i dobiva trojku:

$$(ID_B, K, N_A) = D(E((ID_B, K, N_A), K_{DC}), K_{EC}),$$

- iz dekriptirane trojke treba mu samo tajni ključ K kojim dekriptira M_7 i dobiva trojku:

$$N_B = DES^{-1}(DES(N_B, K), K),$$

- uspoređuje dobiveni N_B s izvornim i, ako su jednaki, utvrđuje autentičnost sudionika A ;



Slika 11.37. Postupak obostrane autentifikacije u zatvorenom asimetričnom sustavu

Ovim se postupkom autentifikacije ujedno dodjeljuje i tajni ključ K koji u daljnjoj komunikaciji omogućuje simetrično kriptiranje.

11.8. Prijava za rad

Uobičajeni postupci za ograničavanje pristupa računalnom sustavu upotrebljavaju *ime korisnika* i *lozinku* (engl. *user name*, *password*). Pristup sustavu dopušta se samo ako korisnik ispravno navede svoje ime i lozinku. Iz imena se izvodi *identifikator* korisnika (engl. *user identifier*).

Taj način pristupa ima mnoge slabosti:

- korisnici mogu sami lako otkriti svoje podatke jer ih obično zapisuju;
- napadači su vrlo domišljati pri otkrivanju identifikatora i lozinki;
- datoteke s identifikatorima i lozinkama meta su napadača.

Koriste se različite metode za povećanje sigurnosti takvog pristupa sustavu:

- broj mogućih lozinki mora biti velik, čime se smanjuje vjerojatnost pogađanja lozinke;
- postupak prijave mora biti takav da se dopušta samo ograničeni broj ponavljanja netočne lozinke (sustav prijave može se namjestiti tako da ne prihvata višekratno neuspješno prijavljivanje s istim identifikatorom);
- operacijski sustav mora pohranjivati pokušaje neovlaštenog pristupa kako bi se olakšala naknadna istraga.

11.8.1. Kriptiranje lozinki

Izvan računalnog sustava tajnost lozinke mora čuvati svaki korisnik i o tome se ne mora brinuti operacijski sustav. Međutim, unutar sustava također se mora sačuvati tajnost. Napadač bi mogao doći do tablica lozinki i narušiti sigurnost. Sigurnost sustava može

se povećati ako se tablica lozinki kriptira. Kriptiranje se obavlja prilikom registracije korisnika, odnosno prilikom svake promijene lozinke.

U postupku autentifikacije lozinku se kriptira i u kriptiranom obliku uspoređuje s pohranjenom lozinkom u tablici. Prema tome, sadržaj tablice nije nikada potrebno dekriptirati. Napadač će iz tablice kriptiranih lozinki vrlo teško moći odrediti njihov izvorni tekst. Ako s P označimo izvorni tekst lozinke, onda će kriptirani tekst biti:

$$C = E(P, K).$$

Izvorni tekst bit će još teže prepoznati ako se ključ kriptiranja K načini promjenljivim. Jedan od načina za to je da lozinka P posluži kao ključ tako da se dobiva:

$$C = E(P, P).$$

Umjesto funkcije kriptiranja mogla bi se upotrijebiti i funkcija sažimanja.

11.8.2. Otežavanje pogodanja lozinke

Korisnici vrlo često smisljavaju lozinke koje im je lako zapamtiti. Zbog toga sustav registracije mora potaknuti korisnika na odabir lozinki koje se teže pogodaju. Otkrivanje lozinke može se otežati tako da:

- sustav na prihvaća kratke lozinke već zahtijeva da one budu propisane duljine;
- prihvaćenu lozinku sustav prije kriptiranja nadopuni nasumičnim brojem koji se dodjeljuje pri registraciji i čuva u posebnoj tablici (on se može mijenjati pri svakoj promjeni lozinke);
- sustav nudi pomoć pri odabiru lozinke s tim da odbija lako prepoznatljive ili iznosi svoj prijedlog za lako pamtljive nakupine slova ili pak potupno nasumično odabran niz znakova.

U raspoljjenim sustavima ovaj način prijavljivanja nije dovoljno siguran i mora se primjeniti jedan od postupaka autentifikacije opisan u prethodnom odjeljku. Ostvarenje te zamisli je sustav *Kerberos* o kojem ćemo govoriti kasnije.

11.8.3. Zaštita pristupa pojedinim sredstvima – autorizacija

U odjeljku 11.1. ustanovili smo da je uz autentifikaciju korisno raspolažati i unutarnjim zaštitnim mehanizmima koji će autentificiranom korisniku omogućiti pristup samo do onih sredstava za koje on ima dopuštenje pristupa. Mehanizmi *dopuštanja pristupa* (engl. *access control*) pojedinim sredstvima nazivaju se *autorizacijom* pristupa (engl. *authorization*). Ustanovili smo da su *subjekti* u postupcima autorizacije pojedini korisnici, odnosno njihovi procesi ili čak neke dretve unutar tih procesa. Sredstva koja se zaštićuju jesu *objekti* zaštite.

Zaštitna pravila (engl. *protection rules*) moraju za svaki par subjekt-objekt odrediti pravo pristupa koje obuhvaća i način na koji se objekt smije upotrebljavati. Zaštitna se pravila

mogu prikazati u obliku matrice pristupa (engl. *access matrix*) u kojoj svaki subjekt dobiva svoj redak i svaki objekt svoj stupac. Neprazni elementi pojedinog retka označavaju one objekte koje pojedini subjekt smije upotrebljavati, a prazni elementi u retku pokazuju da subjekt ne smije pristupiti dotičnom objektu.

Matrica pristupa rijetko je popunjena i može se praktičnije prikazati s pomoću lista.

Liste prava pristupa objektu (engl. *access control list*) čine neprazni elementi stupaca matrice pristupa. Lista sadrži sve subjekte koji imaju pravo pristupa do dotičnog objekta. Neki objekti mogu biti javni (engl. *public*), tj. dostupni svim subjektima.

		OBJEKTI			
		1	2	3	m
SUBJEKTI		1	Č, P		Č
		2	Č	I	P
		3	I		
		n	Č, P		P

Slika 11.38. Matrica pristupa

Neprazni elementi pojedinih redova matrice pristupa čine **listu dozvola za pristup objektima** (engl. *capability tickets*). Svaki subjekt dobit će, kada to zatraži, dozvolu (engl. *ticket*) samo za one objekte koji se nalaze u njegovoj listi dozvola.

U sustavu *Kerberos* dodjeljuju se u fazi autorizacije dozvole za pristup pojedinim objektima.

11.9. Autentifikacijski protokol Kerberos

Protokol *Kerberos* počeo se razvijati 1978. godine na *Massachusetts Institute of Technology (MIT)* u okviru projekta *Athena*. Naziv je dobio po troglavom psu *Kerberu* iz grčke mitologije koji čuva ulaz u podzemni svijet *Had*.

Pojedini operacijski sustavi upotrebljavaju različite varijante *Kerberos* protokola. Ovdje će se razmotriti samo njegovi osnovni mehanizmi. Taj je protokol u prvom redu projektiran za ne prevelike raspodijeljene sustave zasnovane na *TCP/IP* protokolu, ali se on može prilagoditi i drugim okruženjima te čak djelovati i unutar samo jednog računala. U *Kerberosu* se upotrebljava simetrični kriptosustav (izvorno: *DES*). Pretpostavlja se da su sva računala koja sudjeluju u protokolu sigurna i da je samo mreža nesigurna.

11.9.1. Struktura sustava u kojem djeluje Kerberos protokol

Model *Kerberos* sustava možemo zamisliti tako da su svi subjekti koji žele upotrebljavati sredstva sustava (objekte) predstavljeni kao *procesi klijenti* (engl. *client process*). Do svih se objekata sustava može doći isključivo preko *procesa poslužitelja* (engl. *server*). Tako postoji poslužitelji za pristup datotekama, poslužitelji za pristup bazama podataka, poslužitelji za pristup pisačima i sl. Ti su poslužitelji *davatelji usluga*.

Možemo zamisliti da su svi procesi klijenti i svi procesi poslužitelji smješteni unutar jednog računala ili da je svaki od njih smješten u svom čvoru s tim da je između njih uspostavljana razmjena poruka. Zbog pojednostavljenja, u dalnjem ćemo razmatranju prepostaviti da postoje čvorovi *klijenti* i čvorovi *poslužitelji*.

Posebni *Kerberos* čvor sadrži sve potrebne podatke za odvijanje protokola koje on čuva u *Kerberos bazi podataka*. Taj čvor mora biti posebno fizički zaštićen i u njega moraju svi imati povjerenje. Kerberos čvor sastoji se od dva poslužitelja:

- autentifikacijskog poslužitelja (engl. *authentication server*) *AS* i
- poslužitelja za dodjelu dozvola (engl. *ticket-granting server*) *TGS*.

On sadrži i bazu podataka u kojoj su smješteni svi podaci potrebni za odvijanje protokola. U bazi se nalaze tajni ključevi *svih klijenata* i *svih poslužitelja*. Ti se ključevi moraju na siguran način (fizičkim donošenjem) pohraniti u bazu.

S obzirom na to da se na jednoj radnoj stanicu, odnosno osobnom računalu korisnici mogu izmjenjivati, svaki će korisnik imati svoj identifikator i lozinku. Tajni ključ klijenta K_C izračunat će se na temelju lozinke jednosmernom funkcijom kriptiranja ili funkcijom sažimanja. U bazi podataka Kerberos čvora taj se kriptirani ključ pohranjuje na siguran način (fizičkim upisivanjem i izračunavanjem lozinke u *Kerberos* čvoru). Taj se ključ, čak i u kriptiranom obliku neće nikada prenosi mrežom. Dakle, lozinka se u izvornom obliku nikada ne prenosi mrežom. Osjetljivi podaci prenose se u kriptiranom obliku.

Prema tome, sustav se sastoji od sljedećih čvorova:

- **čvora klijenta** (engl. *client node*),
- **čvora poslužitelja** (engl. *application server node*) koji obavlja traženu uslugu i
- **čvora Kerberos poslužitelja** koji se sastoji od:
 - baze podataka (identifikatori, lozinke i tajni ključevi svih poslužitelja u sustavu);
 - poslužitelja za utvrđivanje autentičnosti (*authentication server – AS*);
 - poslužitelja (ili procesa) za dodjelu ulaznica za pristup pojedinim uslugama (*ticket granting server – TGS*).

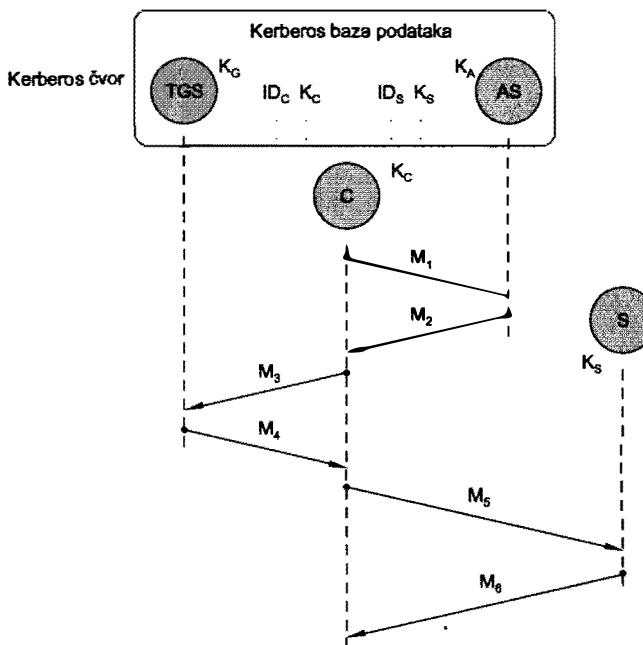
11.9.2. Kerberos protokol

Protokol se provodi sljedećim koracima:

- Kada se korisnik prijavljuje za rad:

- on unosi svoje ime;
- iz njegova se imena određuje identifikator ID_C (lozinka će se tražiti u trećem koraku);
- generira se nasumični broj N_1 i šalje se razgovijetnom obliku autentifikacijskom poslužitelju poruka:

$$M_1 = (ID_C, N_1).$$



Slika 11.39.

- Nakon primitka M_1 autentifikacijski poslužitelj AS :

- generira nasumični broj koji će poslužiti kao *sjednički ključ* K_1 ;
- stvara *ulaznicu*, odnosno *ulaznu dozvolu* koja se sastoji od:

- | | |
|----------|------------------------------------------------------------------------------------------------------|
| ID_C | - identifikatora klijenta, |
| ID_G | - identifikatora TGS, |
| T_{S1} | - trenutka početka valjanosti dozvole, |
| T_{E1} | - <i>trenutka završetka valjanosti dozvole</i>
(trajanje valjanosti može iznositi nekoliko sati), |
| K_1 | - tajnog sjedničkog ključa; |

- s pomoću tajnog ključa poslužitelja za dodjelu dozvola K_G kriptira ulaznu dozvolu u:

$$C_1 = E((ID_C, ID_G, T_{S1}, T_{E1}, K_1), K_G),$$

- na temelju ID_C pronalazi u svojoj bazi podataka lozinku klijenta i izračunava ključ K_C ;
- s pomoću ključa K_C kriptira trojku (N_1, K_1, C_1) i šalje klijentu poruku:

$$M_2 = E((N_1, K_1, C_1), K_C).$$

3. Nakon primitka poruke M_2 klijent:

- traži unošenje lozinke i na temelju nje izračunava tajni ključ K_C (treba uočiti da se lozinka ne prenosi mrežom, nakon izračunavanja ključa ona se može pobrisati);
- s pomoću ključa K_C dekriptira M_2 i dobiva:

$$(N_1, K_1, C_1) = D(E((N_1, K_1, C_1), K_C), K_C),$$

- uspoređuje dobiveni N_1 s izvornom vrijednošću, ustanavljuje autentičnost poslužitelja AS te čuva dobivene K_1 i C_1 za daljnje uporabe.

4. Kada klijent želi neku uslugu od nekog poslužitelja s identifikatorom ID_S , on:

- oblikuje tzv. *autentifikator* (ID_C, T_1) gdje je T_1 trenutak postavljanja zahtjeva (autentifikator vrijedi samo vrlo kratko vrijeme – nekoliko minuta);
- s pomoću sjedničkog ključa K_1 kriptira autentifikator u:

$$C_2 = E((ID_C, T_1), K_1),$$

- generira nasumični broj N_2 i šalje poslužitelju za dodjelu dozvola TGS poruku

$$M_3 = (ID_S, N_2, C_1, C_2).$$

5. Nakon primitka poruke M_3 poslužitelj TGS :

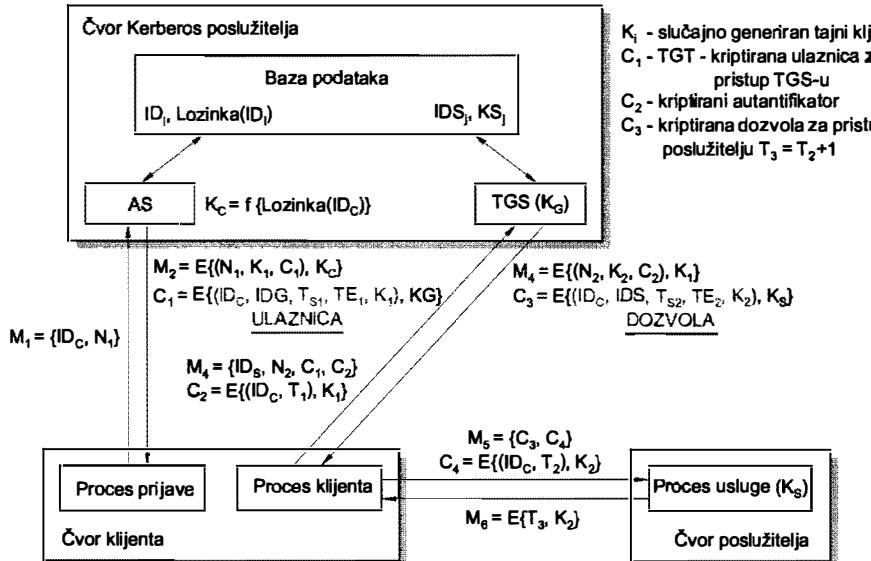
- s pomoću svog tajnog ključa K_G dekriptira C_1 i dobiva: $ID_C, ID_G, T_{S1}, T_{E1}, K_1$;
- s pomoću sjedničkog ključa K_1 dekriptira C_2 i dobiva: ID_C, T_1 ;
- provjerava jesu li iz C_1 i C_2 dobiveni jednaki identifikatori ID_C , je li trenutak T_1 unutar vremenskog intervala valjanosti ulazne dozvole ograničenog s T_{S1} i T_{E1} te da li je T_1 blizak stvarnom vremenu;
- nakon uspješnih provjera generira nasumični broj K_2 koji će postati novi tajni ključ i s pomoću tajnog ključa poslužitelja K_S kriptira novu *dozvolu za pristup poslužitelju* ili kraće *dozvolu*:

$$C_3 = E((ID_C, ID_S, T_{S2}, T_{E2}, K_2), K_S),$$

gdje su T_{S2} i T_{E2} granice važenja ključa K_2 ;

- kriptira trojku (N_2, K_2, C_3) s pomoću ključa K_1 i šalje klijentu poruku:

$$M_4 = E((N_2, K_2, C_3), K_1).$$



Slika 11.40. Komunikacijski dijagram protokola Kerberos

6. Nakon primitka poruke M_4 klijent:

- s pomoću svog ključa K_1 dekriptira M_4 i dobiva: (N_2, K_2, C_3) ;
- uspoređuje dobiveni N_2 s izvornom vrijednošću i ustanavljuje autentičnost poslužitelja TGS ;
- stvara novi autentifikator (ID_C, T_2) i kriptira ga s pomoću K_2 u

$$C_4 = E((ID_C, T_2), K_2),$$

- šalje poslužitelju usluge poruku:

$$M_5 = (C_3, C_4).$$

7. Nakon primitka poruke M_5 poslužitelj:

- s pomoću svog tajnog ključa K_s dekriptira C_3 i dobiva: $(ID_C, ID_G, T_{S2}, T_{E2}, K_2)$;
- s pomoću dobivenog ključa K_2 dekriptira C_4 i dobiva autentifikator (ID_C, T_2) ;
- nakon toga povećava T_2 za jedan i dobiva $T_3 = T_2 + 1$;
- šalje klijentu kriptiranu poruku:

$$M_6 = E(T_3, K_2).$$

8. Nakon primitka poruke M_6 klijent:

- s pomoću ključa K_2 dekriptira M_6 i dobiva T_3 ;

- nakon što ustanovi da je dobiveni T_3 jednak $T_2 + 1$, što će provjeriti uporabom sačuvanog T_2 , klijent je provjerio autentičnost poslužitelja usluga;
- klijent i poslužitelj mogu za obavljanje posla koristiti tajni ključ K_2 .

Postupak autentifikacije obavlja se samo jedanput prilikom prijavljivanja za rad (engl. *single sign-on*). Autentifikacijski poslužitelj AS će nakon provjere identiteta klijentu dodijeliti *ulaznu dozvolu*, tzv. dozvolu za dodjelu dalnjih dozvola (engl. *ticket-granting ticket*) s kojom će klijent od Kerberos poslužitelja za dodjelu dozvola TGS tražiti daljnje dozvole za pristup pojedinim poslužiteljima. Sve će se dozvole izdavati s ograničenim trajanjem tako da je u dozvoli označen početni trenutak T_s i završni trenutak T_e razdoblja valjanosti dozvole.

Nakon što je autentifikacija uspješno provedena klijent može razmjenjivati poruke s poslužiteljem koristeći jednu od tri razine zaštite:

- bez zaštite: provjera autentičnosti obavlja se samo na početku;
- “sigurne poruke” – uz poruku u jasnom obliku šalje se i kriptirani autentifikator;
- “privatne poruke” – kriptirani su i poruka i autentifikator (najviša razina zaštite).

Ograničenja i nedostaci *Kerberos* protokola:

- svaki program treba biti “*kerberiziran*”, odnosno prilagođen za rad u *Kerberos* okruženju;
- Kerberos* protokol ne obuhvaća postupak autorizacije;
- Kerberos* poslužitelj mora biti fizički zaštićen;
- Kerberos* protokolom nije riješen problem sigurne pohrane tajnih ključeva;
- protokol podliježe strogim američkim zakonima o izvozu kriptotehnologije.

Kao što je već rečeno, u različitim se operacijskim sustavima upotrebljavaju različite izvedenice ovog osnovnog modela *Kerberos* protokola.

11.10. Infrastruktura javnih ključeva

11.10.1. Digitalni certifikat

U odjeljku 11.5. ustanovili smo da pri uporabi asimetričnog kriptosustava postoji potreba svojevrsne raspodjele javnih ključeva. Naime, potencijalni sudionici u komunikaciji moraju na neki način doznati javne ključeve svojih partnera. Osim toga, oni se moraju uvjeriti da partneri nisu uljezi tj. sudionici koji se lažno predstavljaju.

Ustanovili smo da se u jednom zatvorenom sustavu svi potencijalni sudionici moraju prijaviti i tada im se dodjeljuje par ključeva. Svoj privatni ključ oni čuvaju kod sebe, a njihov se javni ključ pohranjuje zajedno s njihovim identifikatorom u tablicama pouzdanog poslužitelja koji smo nazvali centrom za raspodjelu javnih ključeva (engl. *public key manager*

– *PKM*). Kada sudionik *A* želi uspostaviti vezu sa sudionikom *B* sigurnim kanalom, on će zatražiti od *PKM*-a njegov javni ključ.

Razmotrili smo i modele protokola za saznavanje javnih ključeva te protokole za jednostranu ili dvostranu autentifikaciju uz pomoć *PKM*-a. Taj su pristup predložili 1976. godine W. Diffie i M. Hellman. Tablicu u *PKM*-u koja identifikatorima (imenima) pridružuje pri-padne javne ključeve nazvali su javnom datotekom (engl. *Public File*). Ovakvo je rješenje prikladno za manje zatvorene sredine (primjerice: jednu banku, jednu tvrtku, sveučilište).

Međutim, ako se u komunikaciju želi uključiti sudionike iz različitih okruženja i uspostaviti sigurno komuniciranje u širim razmjerima, onda jedan jedini centar za rasподjelu ključeva sigurno nije dobro rješenje problema. Kako bi se otklonile poteškoće zatvorenog sustava, godine 1978. L. Kohnfelder predložio je koncepciju digitalnog certifikata.

Svaki se sudionik *I* prijavljuje u jedan certifikacijski centar (engl. *Certification Authority – CA*) pri čemu mu se dodjeljuje javni ključ K_{EI} i privatni ključ K_{DI} . Certifikacijski centar *C* također ima svoj javni ključ K_{EC} i privatni ključ K_{DC} . U postupku prijave certifikacijski centar izrađuje i potpisuje certifikat sudionika koji izgleda ovako:

$$CER_I^C = (ID_I, K_{EI}, E(H(ID_I, K_{EI}), K_{DC})),$$

gdje je H neka funkcija sažimanja (primjerice *SHA*), a E neka funkcija kriptiranja (pri-mjerice, *RSA*). Ovaj osnovni model certifikata sastoji se, dakle, od para ID_I , K_{EI} i digi-talnog potpisa kojim certifikacijski centar *C* jamči da javni ključ K_{EI} pripada sudioniku *I*.

Dakle, certifikat povezuje javni ključ sudionika s njegovim imenom. Istinitost te veze može se provjeriti na temelju digitalnog potpisa, za što je potrebno poznavati javni ključ K_{EC} certifikacijskog centra.

Neka infix operator \bullet označava utvrđivanje ključa sudionika s pomoću ključa certifikacijskog centra K_{EC} , tako da za prijavljene sudionike *A* i *B* možemo utvrditi:

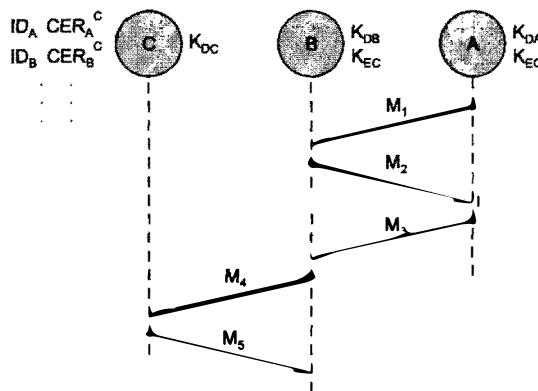
$$\begin{aligned} K_{EA} &= K_{EC} \bullet CER_A^C, \\ K_{EB} &= K_{EC} \bullet CER_B^C. \end{aligned}$$

Ovako oblikovane certifikate ne mora se više tajno čuvati. Oni mogu biti umnažani i smješteni u više datoteka. Uporabu certifikata može se ilustrirati nadopunom opisanog protokola za jednostranu autentifikaciju u asimetričnom sustavu opisanom u odjeljku 11.3.4. U certifikacijskom centru *C* čuva se tablica certifikata svih sudionika koji su u njemu prijavljeni i čiji je identitet prilikom prijave nedvojbeno utvrđen.

Protokol autentifikacije sudionika *A* sastoji se od sljedećih šest koraka u kojima se obavlja razmjena pet poruka:

1. Sudionik *A* šalje u razgovijetnom obliku svoj identifikator sudioniku *B* u poruci:

$$M_1 = (ID_A).$$



Slika 11.41. Sekvenčni dijagram jednostrane autentifikacije uz pomoć certifikata

2. Nakon primitka M_1 sudionik B generira slučajni broj N i šalje u razgovijjetnom obliku sudioniku A poruku:

$$M_2 = (N).$$

3. Nakon primitka M_2 sudionik A kriptira svojim privatnim ključem N i šalje poruku:

$$M_3 = E(N, K_{DA}).$$

4. Sudionik B , nakon primitka M_3 , šalje certifikacijskom centru C u razgovijjetnom obliku poruku:

$$M_4 = (ID_A, R_B)$$

gdje je R_B kôd kojim sudionik B zahtijeva od CA certifikat sudionika s identifikatorom ID_A .

5. Nakon primitka M_4 poslužitelj certifikacijskog centra C :

- iz svoje tablice na temelju ID_A pročita CER_A^C ;
- s pomoću svog privatnog ključa K_{DC} kriptira CER_A^C i šalje poruku:

$$M_5 = E(CER_A^C, K_{DC}).$$

6. Nakon primitka M_5 sudionik B :

- javnim ključem poslužitelja dekriptira M_5 i dobiva:

$$CER_A^C = D(E(CER_A^C, K_{DC}), K_{EC}),$$

- iz CER_A^C saznaće:

$$ID_A, K_{EA} \text{ i } E(H(ID_A, K_{EA}), K_{DC})),$$



- izračunava $H(ID_A, K_{EA})$ i dobiveni rezultat uspoređuje s dekriptiranom vrijednošću:

$$D(E(H(ID_A, K_{EA}), K_{DC})), K_{EC},$$

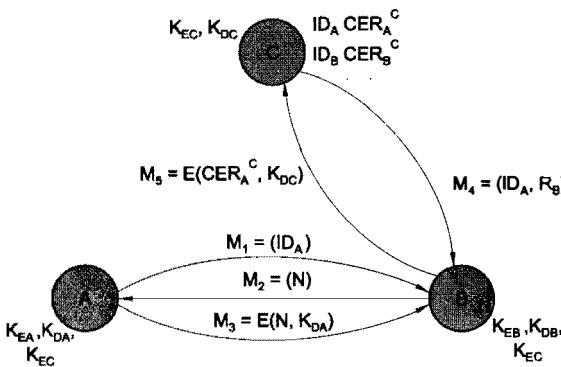
čime provjerava dobiveni K_{EA} , a zapravo je proveo operaciju:

$$K_{EA} = K_{EC} \bullet CER_A^C,$$

- s dobivenim K_{EA} dekriptira raniju poruku M_3 i dobiva:

$$N = D(E(N, K_{DA}), K_{EA}),$$

- dobiveni N uspoređuje s originalom te prihvaca ili odbacuje sudionika A .



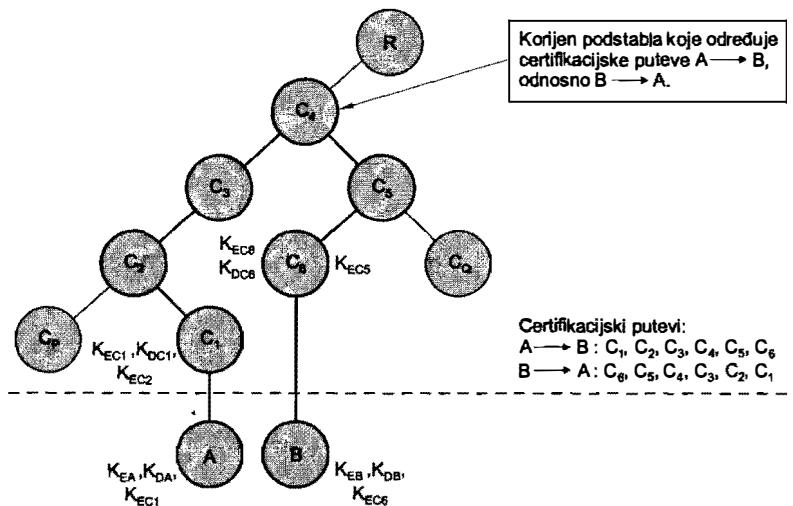
Slika 11.42. Postupak jednostrane autentifikacije uz pomoć certifikata

U ovom se slučaju protokol bitno ne razlikuje od onog iz odjeljka 11.3.4. jer su oba sudionika prijavljena u istom certifikacijskom centru.

11.10.2. Provjera certifikata u otvorenoj mreži

Uporaba certifikata omogućuje postojanje više certifikacijskih centara koji mogu biti povezani u raspodijeljeni sustav. Sudionici u komunikaciji mogu pripadati različitim centrima. Provjera certifikata obavlja se uvijek u centru u kojem je sudionik prijavljen.

Najprikladnije je certifikacijske centre povezati u hijerarhijski građenu stablastu strukturu. Takva se struktura prirodno može izgraditi iznad TCP razine globalne računalne mreže (opisane u odjeljku 10.2.1.). Svaki čvor u toj strukturi ima svoje jedinstveno ime, odnosno adresu.



Slika 11.43. Hjерархијски повезаниcertifikacijski centri

Svaki sudionik prilikom prijave:

- dobiva svoj jedinstveni identifikator (ime) te javni i privatni ključ (tako sudionici A i B imaju identifikatore ID_A , odnosno ID_B i ključeve K_{EA} i K_{DA} , odnosno K_{EB} i K_{DB});
- dobiva certifikat potpisani od certifikacijskog centra u kojem je obavio prijavu (tako sudionici A i B dobivaju certifikate:

$$CER_A^{C1} = (ID_A, K_{EA}, E(H(ID_A, K_{EA}), K_{DC1}))$$

odnosno:

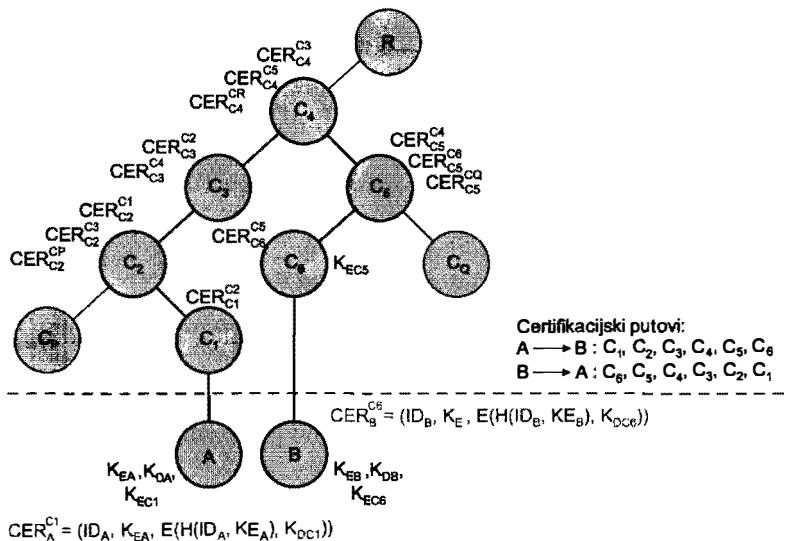
$$CER_B^{C6} = (ID_B, K_{EB}, E(H(ID_B, K_{EB}), K_{DC6})).$$

Nadalje, svaki certifikacijski centar C_I ima:

- svoj jedinstveni identifikator ID_{CI} te javni ključ K_{ECI} i privatni ključ K_{DCI} ;
- certifikate potpisane od svojih neposrednih susjeda u mreži (u stablasto građenoj mreži: od svog roditelja i sve svoje djece).

U primjeru na slici za pojedine se čvorove pri uspostavljanju mreže izrađuju sljedeći certifikati:

$$\begin{array}{ll} C_1: CER_{C1}^{C2} \text{ (čvor nema djece)}; & C_4: CER_{C4}^{C3}, CER_{C4}^{C5}, CER_{C4}^{CR}; \\ C_2: CER_{C2}^{C1}, CER_{C2}^{C3}, CER_{C2}^{CP}; & C_5: CER_{C5}^{C4}, CER_{C5}^{C6}, CER_{C5}^{CQ}; \\ C_3: CER_{C3}^{C2}, CER_{C3}^{C4}; & C_6: CER_{C6}^{C5} \text{ (čvor nema djece)}. \end{array}$$



Slika 11.44. Certifikati centara na putovima $A \rightarrow B$, odnosno $B \rightarrow A$

Sudionik A može provjeriti certifikat sudionika B iz certifikata CER_B^{C6} ako sazna ključ K_{EC6} . S obzirom na to da on poznaje samo ključ K_{EC1} , do ključa može doći tako da redom utvrdi javne ključeve svih centara na putu $A \rightarrow B$, odnosno:

$$\begin{aligned} K_{EC2} &= K_{EC1} \bullet CER_{C2}^{C1}, \\ K_{EC3} &= K_{EC2} \bullet CER_{C3}^{C2}, \\ K_{EC4} &= K_{EC3} \bullet CER_{C4}^{C3}, \\ K_{EC5} &= K_{EC4} \bullet CER_{C5}^{C4}, \\ K_{EC6} &= K_{EC5} \bullet CER_{C6}^{C5}, \\ K_{EB} &= K_{EC6} \bullet CER_B^{C6}, \end{aligned}$$

ili:

$$K_{EB} = K_{EC1} \bullet CER_{C2}^{C1} \bullet CER_{C3}^{C2} \bullet CER_{C4}^{C3} \bullet CER_{C5}^{C4} \bullet CER_{C6}^{C5} \bullet CER_B^{C6},$$

što možemo kraće zapisati ovako:

$$K_{EB} = K_{EC1} \bullet (A \rightarrow B) \bullet CER_B^{C6}.$$

Jednako tako, sudionik B može provjeriti certifikat CER_A^{C1} tako da redom obavi sljedeće provjere:

$$\begin{aligned} K_{EC5} &= K_{EC6} \bullet CER_{C5}^{C6}, \\ K_{EC4} &= K_{EC5} \bullet CER_{C4}^{C5}, \\ K_{EC3} &= K_{EC4} \bullet CER_{C3}^{C4}, \\ K_{EC2} &= K_{EC3} \bullet CER_{C2}^{C3}, \end{aligned}$$

$$K_{EC1} = K_{EC2} \bullet CER_{C1}^{C2}, \\ K_{EA} = K_{EC1} \bullet CER_A^{C1},$$

ili, kraće:

$$K_{EA} = K_{EC6} \bullet (B \rightarrow A) \bullet CER_A^{C1}.$$

Ako zbog općenitosti certifikacijski centar u kojem je registriran sudionik *A* imenujemo s *CA* i centar u kojem je registriran sudionik *B* sa *CB*, onda možemo pisati:

$$K_{EA} = K_{ECB} \bullet (B \rightarrow A) \bullet CER_A^{CA}$$

i:

$$K_{EB} = K_{ECA} \bullet (A \rightarrow B) \bullet CER_B^{CB}.$$

Treba naglasiti da putovi ne moraju nužni biti dijelovi stablaste strukture, već im struktura može biti proizvoljna!

11.10.3. Infrastruktura javnih ključeva zasnovana na X.509 modelu

Ostvarenja sustava u kojima se na opisani način upotrebljavaju certifikati razlikuju se i po namjenama i po detaljima izvedbe. Opisi tih sustava, kao što su *Pretty Good Privacy (PGP)*, *Simple Public Key Infrastructure / Simple Distributed Security Infrastructure (SPKI/SDSI)*, premašuju namjenu ovog teksta kojemu je svrha opis osnovnih modela postupaka za povećanje sigurnosti.

Ipak, treba makar pregledno spomenuti jedan od modela koji čini osnovu za ostvarenje većine današnjih sustava. Taj je model uspostavljen u okviru *Međunarodne telekomunikacijske unije (ITU)* i nosi naziv *X.509*.

X.509 certifikat

Certifikat *X.509* osim identifikatora i pridruženog mu javnog ključa sadrži sljedeće elemente:

- verziju *X.509* preporuke;
- serijski broj certifikata;
- naziv certifikacijskog centra koji izdaje certifikat;
- primijenjeni algoritam sažimanja i potpisivanja;
- ime i identifikator sudionika;
- primijenjeni kriptosustav (algoritam i parametri);
- razdoblje valjanosti certifikata (početni i završni dan);
- digitalni potpis certifikacijskog centra.

Preporučuje se uporaba različitih ključeva za potpisivanje, identifikaciju i kriptiranje teksta.

```

Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 75 (0x4b)
    Signature Algorithm: md5WithRSAEncryption
    Issuer: C=US, ST=WA, L=Seattle, O=Thawte Consulting cc,
             OU=Certification Services Division,
             CN=Thawte Server CA/emailAddress=certs@thawte.com
  Validity
    Not Before: May 13 23:33:08 2008 GMT
    Not After : Dec 31 23:59:59 2020 GMT
  Subject: C=HR, ST=Grad Zagreb, L=Zagreb, O=FER, OU=CIP,
            CN=webmail.fer.hr/emailAddress=korisnik@webmail.fer.hr
  Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
    RSA Public Key: (1024 bit)
      Modulus (2048 bit):
        00:cd:66:28:fb:b8:b3:b7:e0:72:77:48:2d:08:04:
        e1:6d:lc:c5:4f:57:73:0c:e6:db:3b:8e:cd:c6:25:
        61:7f:60:c9:da:a3:9f:1d:fa:d8:ef:00:7b:f9:54:
        65:ab:7e:9e:9b:6d:ff:d4:12:ad:f8:ac:87:6e:83:
        ec:65:5f:b4:2d:eb:b8:dc:1c:d7:32:b7:46:a5:e3:
        a1:6c:0b:4c:1b:0c:89:0a:fb:0e:3a:c0:0f:af:b2:
        62:1d:2f:60:e4:b1:27:b4:7c:59:00:2c:19:e9:f3:
        a3:88:fe:01:d6:56:be:26:c7:f8:42:b1:79:39:98:
        a1:b4:4a:84:dd:20:ca:e7:a9:db:6d:a6:73:88:e7:
        81:8b:3e:81:3d:00:e5:5d:7f:3d:9b:cd:ba:9b:28:
        88:88:7f:d7:69:2c:66:eb:8f:79:b8:ec:bc:bb:76:
        67:b1:00:2a:70:bd:f1:21:66:6f:ba:74:81:82:30:
        02:c0:a8:57:f8:9f:76:02:df:7f:49:44:4a:32:93:
        48:a4:25:73:47:10:21:20:fe:b6:d2:09:1a:60:4f:
        a5:d9:df:ea:55:49:43:c6:ce:96:0b:7d:a7:22:c1:
        3e:5b:28:2e:2c:04:7a:b2:93:89:db:d8:2b:59:86:
        a3:0a:c1:6f:f9:56:b2:a5:71:4c:4b:74:f3:b8:a1:
        b4:65
      Exponent: 65537 (0x10001)
  X509v3 extensions:
    X509v3 Basic Constraints: critical
      CA:TRUE
  Signature Algorithm: md5WithRSAEncryption
  07:fa:4c:69:5c:fb:95:cc:46:ee:85:83:4d:21:30:8e:ca:d9:
  a8:6f:49:1a:e6:da:51:e3:60:70:6c:84:61:11:a1:1a:c8:48:
  3e:59:43:7d:4f:95:3d:al:8b:b7:0b:62:98:7a:75:8a:dd:88:
  4e:4e:9e:40:db:a8:cc:32:74:b9:6f:0d:c6:e3:b3:44:0b:d9:
  8a:6f:9a:29:9b:99:18:28:3b:d1:e3:40:28:9a:5a:3c:d5:b5:
  e7:20:1b:8b:ca:a4:ab:8d:e9:51:d9:e2:4c:2c:59:a9:da:b9:
  b2:75:1b:f6:42:f2:ef:c7:f2:18:f9:89:bc:a3:ff:8a:23:2e:
  70:47

```

Slika 11.45. Primjer X.509 certifikata

Preporučeni X.509 autentifikacijski protokoli

X.509 preporučuje tri protokola za autentifikaciju: protokol s jednom porukom, protokol s dvije poruke i protokol s tri poruke.

- Protokol s jednom porukom (engl. *one-way protocol*) kojim se autentificiraju oba sudionika *A* i *B* osigurava integritet sadržaja koji se prenosi sudioniku *B* te uporabom vremenske oznake sprečava napad ponavljanjem poruke.

- Protokol s dvije poruke (engl. *two-way protocol*) kojim se pridodaje odgovor sudionika B , utvrđuje da je upravo sudionik B a ne neki napadač odgovorio na prvu poruku i uporabom vremenske oznake sprečava napad ponavljanjem druge poruke.
- Protokol s tri poruke (engl. *three-way protocol*) je onaj kojim sudionik A vraća treću poruku sudioniku B ali se u svim porukama ne upotrebljavaju vremenske oznake.

U **protokolu s jednom porukom** obavljaju se sljedeći koraci:

1. Kada sudionik A želi uspostaviti komunikaciju sa sudionikom B , on:
 - generira nasumični broj N_A ;
 - oblikuje vremensku oznaku T_A koja se sastoji od dvije komponente: početnog vremena valjanosti oznake i trajanja valjanosti oznake;
 - pronalazi u tablicama put $A \rightarrow B$ te iz certifikata CER_B^{CB} saznaće i utvrđuje javni ključ sudionika B :

$$K_{EB} = K_{ECA} \bullet (A \rightarrow B) \bullet CER_B^{CB},$$

- oblikuje četvorku (ID_B, T_A, N_A, D) gdje je D podatkovna komponenta koja može biti kriptirana s K_{EB} ;
- kriptira oblikovanu četvorku svojim privatnim ključem K_{DA} i šalje sudioniku B poruku:

$$M_1 = (CER_A^{CA}, E((ID_B, T_A, N_A, D), K_{DA})).$$

2. Kada sudionik B primi poruku M_1 , on:

- pronalazi u tablicama put $B \rightarrow A$ te iz certifikata CER_A^{CA} saznaće i utvrđuje javni ključ sudionika A :

$$K_{EA} = K_{ECB} \bullet (B \rightarrow A) \bullet CER_A^{CA},$$

- s pomoću ključa K_{EA} dobiva:

$$(ID_B, T_A, N_A, D) = D(E((ID_B, T_A, N_A, D), K_{DA}), K_{EA}),$$

- na temelju ID_B utvrđuje da je poruka stvarno upućena njemu;
- na temelju vremenske oznake T_A utvrđuje da je poruka svježa;
- dekriptira svojim privatnim ključem K_{DB} podatkovnu komponentu D ako je bila kriptirana;
- po želji, može usporediti dobiveni N_A s pohranjenim nasumičnim brojevima iz prethodnih poruka kako bi ustanovio da poruka nije ponovljena.

Protokol s dvije poruke nastavlja se na protokol s jednom porukom tako da se obavljaju daljnja dva koraka:

3. Sudionik B :

- generira svoj nasumični broj N_B ;

- generira vremensku oznaku T_B ;
- oblikuje petorku (ID_A, T_B, N_A, N_B, D) gdje je D podatkovna komponenta koja može biti kriptirana s K_{EA} ;
- kriptira oblikovanu petorku uporabom svog privatnog ključa K_{DB} i šalje sudioniku A poruku:

$$M_2 = E((ID_A, T_B, N_A, N_B, D), K_{DB}).$$

4. Kada sudionik A primi poruku M_2 , on:

- s pomoću ključa K_{EB} dekriptira M_2 i dobiva:

$$(ID_A, T_B, N_A, N_B, D) = D(E((ID_A, T_B, N_A, N_B, D), K_{DB}), K_{EB}),$$

- na temelju ID_A utvrđuje da je poruka stvarno upućena njemu;
- na temelju vremenske oznake T_B utvrđuje da je poruka svježa;
- dekriptira svojim privatnim ključem K_{DA} podatkovnu komponentu D ako je bila kriptirana;
- po želji, može usporediti dobiveni N_B s pohranjenim nasumičnim brojevima iz prethodnih poruka kako bi ustanovio da poruka nije ponovljena.

Protokol s tri poruke obavlja se na jednak način kao i protokol s dvije poruke, s tim da se pri odvijanju prethodnih četiriju točaka ignoriraju vremenske oznake (stavlja se: $T_A = T_B = 0$). Protokol se nastavlja na protokol s dvije poruke sa sljedeće dve točke:

5. Sudionik A :

- uspoređuje dobiveni N_A iz poruke M_2 s izvornom vrijednošću i utvrđuje da je poruka M_2 odgovor na M_1 ;
- s pomoću ključa K_{DA} kriptira dobiveni N_B i šalje poruku:

$$M_3 = E(N_B, K_{DA}).$$

6. Nakon primitka poruke M_3 sudionik B :

- s pomoću ključa K_{EA} dekriptira poruku M_3 i dobiva:

$$N_B = D(E(N_B, K_{DA}), K_{EA}),$$

- uspoređuje dobiveni N_B iz poruke M_3 s izvornom vrijednošću i utvrđuje da je M_3 odgovor na M_2 .

11.10.4. Problem opozivanja certifikata

Pri ostvarivanju infrastrukture javnih ključeva pojavljuje se problem opozivanja certifikata. Naime, nakon što je certifikat izdan u jednom od certifikacijskih centara, on se u

tablicama raspodijeljenog sustava može naći na više mesta. Kada se iz nekog razloga certifikat opoziva, onda bi se istovremeno moralo opozvati i sve njegove kopije.

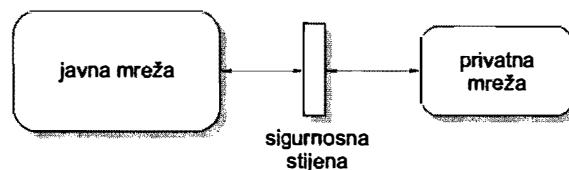
Ostvarenje te zamisli nije sasvim jednostavno. U načelu se ono može provesti na sličan način kao što se to čini s kreditnim karticama. Izdavatelj kartica šalje svim prodajnim mjestima popis opozvanih kartica, s tim da se na svakom mjestu prije odobrenja kupnje mora provjeriti valjanost kartice.

Certifikacijski centar koji je izdao certifikat mora uspostaviti popis opozvanih certifikata (engl. *certificate revocation list – CRL*). Protokol provjere certifikata mora se nadopuniti tako da se najprije provjeri *CRL* i tek nakon što se ustanovi da se certifikat u tom popisu ne nalazi provjerava potpis certifikata.

U različitim sustavima problem opoziva certifikata kao i zamjena certifikata novima (ili obnovljenim) obavlja se različitim protokolima i prelazi okvire razmatranja jednostavnih modela.

11.11. Sigurnosna zaštitna stijena

Sigurnosna zaštitna stijena ili kraće: *sigurnosna stijena* (i najkraće samo: *stijena*) jest računalo ili neka nakupina komunikacijskih naprava koje fizički razdvajaju dvije mreže. Uobičajeno, sigurnosna zaštitna stijena ograničava pristup nekoj privatnoj lokalnoj mreži (ili čak samo jednom računalu) iz javne mreže.



Slika 11.46. Sigurnosna zaštitna stijena fizički razdvaja javnu i privatnu mrežu

U engleskom se jeziku zaštitna sigurnosna stijena naziva protupožarnom stijenom (engl. *firewall*) jer se slikovito može zamisliti da stijena sprečava prođor "požara" nedopuštenog pristupa iz vanjske mreže u lokalnu mrežu. Sigurnosna stijena omogućuje zaštitu podataka i programa u lokalnoj mreži, ali i nadzor zaposlenika kompanije ili institucije i ograničavanje njihovoga pristupa do nekih mesta izvan mreže. Stijena, dakle, može obavljati svoju funkciju i dvostrano.

Jedna od mogućnosti uporabe stijena jest povezivanje dviju fizički udaljenih lokalnih mreža sigurnim informacijskim kanalom. Sigurnosne stijene lokalnih mreža pritom moraju obavljati kriptiranje i dekriptiranje svih poruka koje se prenose javnom mrežom (koja je nesigurni informacijski kanal).



Slika 11.47. Povezivanje dviju fizički udaljenih mreža sigurnim informacijskim kanalom uz pomoć sigurnosne stijene

Izvedbe sigurnosnih stijena vrlo su raznolike i ovise o tome što se njima želi postići. Dobro osmišljena i konfigurirana zaštitna stijena može dobro zaštiti od uljeza.

Zaštitne stijene koje zahtijevaju strogu autentifikaciju i autorizaciju pristupa vrlo su djelotvorne. Prethodno opisani protokoli mogu poslužiti za ostvarenje takvih stijena. Međutim, neke izvedbe sigurnosnih stijena koje su danas u uporabi ne obavljaju svoju zaštitnu ulogu tako temeljito i zbog toga predstavljaju samo djelomičnu zaštitu.

Po svom načinu djelovanja sigurnosne stijene mogu se podijeliti u tri skupine:

- stijene koje filtriraju komunikacijske pakete (engl. *packet filter*);
- stijene koje djeluju kao prividni poslužitelji (engl. *proxy server*);
- stijene koje djeluju kao stvarni poslužitelji (engl. *full server*).

Filtarske sigurnosne stijene djeluju na nižim razinama komunikacijskih protokola i obavljaju svoju funkciju na temelju podataka koje pronalaze u komunikacijskim paketima.

Na temelju nekih podataka kao što su, primjerice, adresa pošiljatelja, adresa primatelja i smjer kretanja paketa, stijena može neke pakete propušтati a neke ne propušтati, tj. blokirati. Takve se stijene nazivaju i *blokirajućim stijenama* (engl. *blocking firewall*). Blokiranje se može obaviti i na primjenskoj razini ili prijenosnoj razini tako da, primjerice, zaštitna stijena dopušta prolaz elektroničke pošte, ali blokira prijavu udaljenog korisnika.

Sigurnosne stijene koje djeluju kao *prividni poslužitelji* djeluju tako da prihvaćaju zahtjeve za obavljanje nekih usluga, obave sigurnosnu provjeru tih zahtjeva i zatim proslijeduju zahtjev za obavljanje te usluge stvarnom zaštićenom poslužitelju.

Sigurnosne stijene mogu djelovati i kao *stvarni poslužitelji*. One na kontrolirani način obavljaju usluge vanjskim klijentima i ne dopuštaju neposredni kontakt vanjskih klijenata i unutarnjih poslužitelja.

Dobro konfigurirana sigurnosna stijena može poslužiti kao zaštita od uljeza iz vanjske u lokalnu mrežu. Međutim, treba naglasiti da se zaštitne stijene ne mogu smatrati apsolutnom zaštitom. Sigurnosne stijene su samo jedna od komponenti sigurnosne slagalice. One ostvaruju zaštitu samo na granicama mreža i nemaju nikakav utjecaj na unutarnju sigurnost. Kombinacija sigurnosnih stijena i ostalih sigurnosnih mehanizama pridonijet će povećanju stvarne sigurnosti sustava.

11.12. Zaključne napomene

Sigurnost postaje sve važnija komponenta računalnih sustava. Ona je osnovni preduvjet ostvarenja mnogih primjena. To je područje računarstva u razdoblju intenzivnog razvitka i stoga se odabiru sigurnosnih mehanizama mora pristupiti vrlo pomnijivo. Ipak, neka od rješenja pokazala su u primjeni potpuno zadovoljavajuće rezultate. Niz sigurnosnih postupaka i međunarodno je normiran, a mnogi od njih postali su *de facto* norme.

Treba uočiti da suvremeni operacijski sustavi kao svoj sastavni dio nude vrlo dobra rješenja autentifikacije i autorizacije te da se u sučeljima prema korisničkim programima kroz API funkcije nude prikladni sigurnosni mehanizmi.

Novija radna okruženja pojedinih programskih jezika nadopunjaju se također postupcima koji se mogu ugrađivati u primjenske programe (tako se, primjerice, za programski jezik *Java* pronalazi API funkcije za ostvarenje digitalnog potpisivanja, izradu sažetaka poruka, raspodjelu ključeva i autorizacije).

Za raspodijeljena okruženja u širokoj je uporabi protokol *Secure Sockets Layer – SSL* (izvorno razvijen u tvrtki *Netscape*) koji djeluje kao medurazina između *TCP-a* i primjenske razine. On omogućuje odabir različitih kriptosustava za ostvarenje pojedinih sigurnosnih mehanizama.

U okviru *IETF-a* razrađen je skup protokola *Transport-Layer Security – TLS* koji je sličan skupu *SSL*. On se također oslanja na *TCP* razinu. *IETF* razrađuje skupinu *IPsec* protokola koji djeluje u načelu na mrežnoj razini.

Dva sudionika u komunikaciji moraju najprije uspostaviti sigurnosno udruživanje (engl. *security association*), pri čemu se u svakom smjeru generiraju jednosmjerni sigurni komunikacijski kanali. Nakon toga svi *IP* paketi dobivaju sigurnosna zaglavla i svaki od paketa digitalno je potpisani.

S primjenske je razine sva sigurnosna zaštita potpuno transparentna. Prema tome, izgradnja primjerenog sigurnosnog sustava zahtijeva pomniju sigurnosnu analizu sustava koji se želi ostvariti te odabir i prilagodbe odgovarajućih sigurnosnih mehanizama.

Problem ostvarenja odgovarajuće razine sigurnosti ostat će, prema svemu sudeći, trajni izazov.

Treba naglasiti da su neke osnovne zasade današnjih mehanizama zaštite nastale u ranim sedamdesetim godinama tako da će daljnji napredak u tehnologiji (posebice povećanje brzine procesora i brzine prijenosa u današnjim mrežama) zahtijevati stalno poboljšanje sigurnosnih mehanizama.

U ovom se poglavlju opisuju samo osnove kriptografskih postupaka i daje pregled osnovnih modela sigurnosnih protokola. Na tim se temeljima može pristupiti detaljnijem izučavanju kriptografskih postupaka i protokola.



PITANJA ZA PROVJERUZNANJA 11

1. Objasniti pojmove identifikacija, autentifikacija i autorizacija.
2. Navesti vrste napada na sigurnost računalnog sustava.
3. Navesti sigurnosne zahtjeve.
4. Navesti svojstva simetričnog i asimetričnog kriptosustava.
5. Na koji način se kriptira jasni tekst M koristeći jednokratnu bilježnicu (*one time pad*)?
6. Navesti nekoliko simetričnih kriptosustava.
7. Koje su moguće duljine ključa u kriptosustavima DES, IDEA i AES?
8. Navesti postupak kriptiranja i dekriptiranja utrostručenim DES kriptosustavom.
9. Navesti postupak kriptiranja i dekriptiranja izbijeljenim DES kriptosustavom.
10. Čemu služe supstitucijske (S) tablice u kriptosustavima DES i AES?
11. Skicirati ECB, CBC, CFB, OFB i CTR načine kriptiranja.
12. Koji je osnovni nedostatak ECB načina kriptiranja?
13. Koji su načini kriptiranja pogodni za kriptiranje toka podataka?
14. Navesti postupak generiranja ključeva u RSA kriptosustavu.
15. Navesti postupak kriptiranja koristeći RSA kriptosustav.
16. Što je digitalna omotnica?
17. Što je digitalni potpis?
18. Što je digitalni pečat?
19. Koje sigurnosne zahtjeve osigurava digitalna omotnica, koje digitalni potpis, a koje digitalni pečat?
20. Što se ispituje uz pomoć heurističkog ispitivanja po Milleru i Rabinu? Koji je rezultat ispitivanja?
21. Navesti nekoliko funkcija za izračunavanje sažetka poruke.

22. Nabrojiti svojstva koja mora zadovoljavati funkcija za izračunavanje sažetka poruke.
23. Pojasniti što znači da je neka funkcija sažimanja otporna na kolizije/ izračunavanje originala/ izračunavanje poruke koja daje isti sažetak?
24. Kolika je veličina sažetka ako se koristi MD5/SHA-1 postupak?
25. U čemu se razlikuju postupci sažimanja SHA-0 i SHA-1?
26. Opisati Diffie-Hellmanov postupak razmjene tajnog ključa.
27. Opisati napad čovjek u sredini (*man in the middle attack*) kada se koristi Diffie-Hellmanov postupak razmjene tajnog ključa.
28. Na koji se način obavlja zaštita pristupanja pojedinim sredstvima (autorizacija) koristeći matricu pristupa?
29. Matrica pristupa rijetko je popunjena i može se praktičnije prikazati s pomoću lista. Navesti i opisati te liste.
30. Od čega se sastoji čvor Kerberos poslužitelja?
31. U čemu se razlikuju ulazna dozvola i dozvola za pristup poslužitelju?
32. Što je digitalni certifikat? Navesti osnovni sadržaj digitalnog certifikata.
33. Navesti dijelove PKI sustava.
34. Opisati postupak jednostrane autentifikacije uz pomoć certifikata.
35. Navesti i opisati preporučene X.509 autentifikacijske protokole.
36. Navesti vrste sigurnosnih stijena.

12.

Višediskovni zalihosni spremnici



12.1. Osnovna razmatranja

Mikroelektronička tehnologija osigurava trajno unapređivanje svojstava procesora i poluvodičkih spremnika. Međutim, napredak svojstava magnetskih diskova mnogo je sporiji jer on ovisi o napretku elektromehaničkih svojstava diskovnih naprava. Stoga diskovlje postaje ograničavajući faktor u sveopćem unapređivanju računalnih sustava.

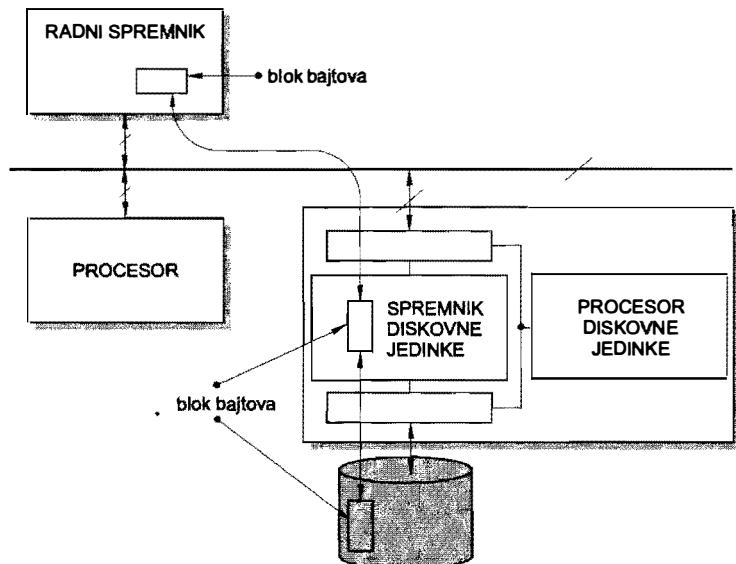
Pri razmatranju gospodarenja spremničkim prostorom upoznali smo osnovna svojstva diskova. Iz tog razmatranja proizlazi da je maksimalna brzina prijenosa podataka s diska ili na disk jednoznačno određena brzinom vrtnje ploča i kapacitetom staze diska.

Nadalje, dobavljanje nekog sadržaja s diska ili njegova pohranjivanja ovisi i o njegovu smještanju u sektore diska. Kompaktni smještaj pri kojem se minimizira trajanje premještanja glava s cilindra na cilindar omogućuje maksimalno iskorištenje širine pojasa pristupa disku. Raspršeni smještaj blokova podataka (koji minimizira fragmentaciju diska) smanjiće brzinu prijenosa jer se prijenos ne može obavljati u razdoblju premještanja glava i rotacijskog kašnjenja.

Upravljački sklopovi, koji su po brzini rada usporedivi s procesorom, nezamjetno utječu na brzinu prijenosa. Štoviše, upravljački sklopovi mogu poslužiti za neka poboljšanja u prijenosu više uzastopnih blokova jer se njihovi spremnici mogu upotrijebiti kao priručni spremnici.

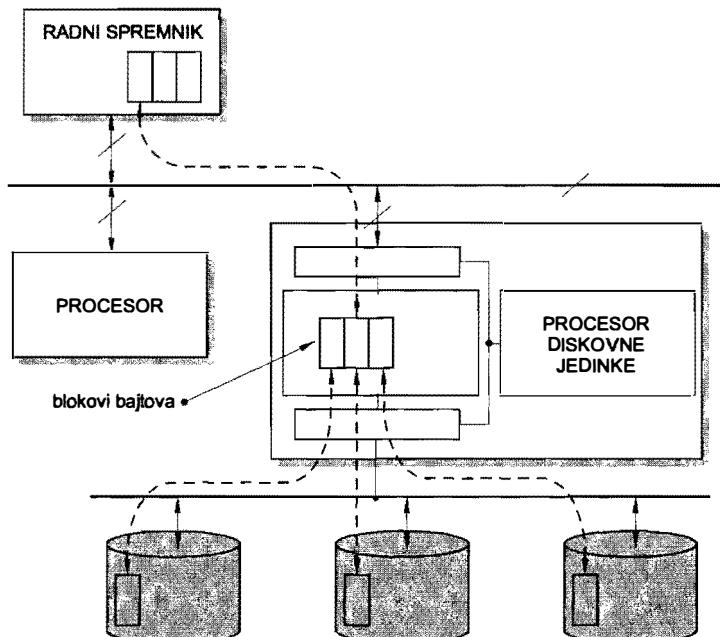
Ubrzanje pri prijenosu može se, primjerice, postići tako da se:

- pri čitanju jednog bloka u spremnik upravljačkog sklopa prenese sadržaj cijele staze i tako ubrza eventualno čitanje nekog drugog bloka s iste staze;
- posluživanje više zahtjeva koji su prispjeli u upravljački sklop ne obavlja redom prispijeća već onim redoslijedom koji minimira pomake glava.



Slika 12.1. Prijenos s diska u radni spremnik posredstvom upravljačkog sklopa

Međutim, sva takva nastojanja mogu ubrzati prijenos samo do njegova maksimalnog iznosa određenog elektromehaničkim svojstvima diskova. To se ograničenje može zaobići smještanjem podataka na više diskova kojima se može istodobno pristupati.



Slika 12.2. Višediskovni podsustav

Višediskovni se sustav sastoji od polja diskova (engl. *disk array*) kojima se pristupa paralelno. Podaci se na diskove mogu raspoređivati na različite načine.

Najjednostavnije je promatrati diskove kao nezavisne cjeline i svaki od njih adresirati nezavisno kao što je prikazano slikom 12.3. Adresni prostor diska čine adrese sektora pri čemu je adresa odredena rednim brojem cilindra, rednim brojem staze u cilindraru i rednim brojem sektora na stazi.

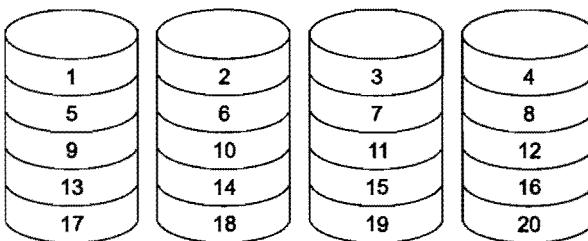
Disk 1	A0	A1	A2	A3	
Disk 2	B0	B1	B2	B3	
Disk 3	C0	C1	C2	C3	

Slika 12.3. Smještaj podataka na pojedinačne diskove

U drugom se načinu raspoređivanja cijelo polje diskova promatra kao jedan logički adresni prostor. Pritom je osnovna jedinica adresiranja *podatkovni pojas* ili, kraće, samo: *pojas* (engl. *data stripe, stripe*), koji se na diskove smješta kružnim (engl. *round robin*) adresiranjem.

Razlikujemo dvije vrste pojasne organizacije:

- sitno zrnatu pojasnu organizaciju (engl. *fine-grained striping*) i
- krupno zrnatu pojasnu organizaciju (engl. *coarse-grained striping*).



Slika 12.4. Pojasna organizacija višediskovnog adresnog prostora

U *sitno zrnatoj* organizaciji podaci se podjednako raspoređuju na sve diskove kako je to ilustrirano slikom 12.5. Razumno je za najmanju veličinu pojasa odabrati veličinu jednog sektora. Međutim, pri razmatranju sitno zrnate organizacije u literaturi se spominju bajtovi ili čak bitovi kao moguće minimalne veličine pojaseva. Ostvarenje takve ekstremne sitne zrnatosti može se postići uporabom međuspremnika upravljačkog sklopa u kojem je moguće razložiti sadržaje dobavljenih cijelih sektora.

Disk 1	A0 A1 A2 A3	B0 B1 B2 B3	C0 C1 C2 C3
Disk 2	A0 A1 A2 A3	B0 B1 B2 B3	C0 C1 C2 C3
Disk 3	A0 A1 A2 A3	B0 B1 B2 B3	C0 C1 C2 C3

Slika 12.5. Sitnozrnati pojasni smještaj

Osnovna je pretpostavka sitno zrnate organizacije da se pri pojedinim zahtjevima za pristupanje diskovima uviјek podjednako angažiraju svi diskovi. Pojednostavljenogledano, brzina posluživanja zahtjeva trebala bi se pritom smanjiti za faktor koji je jednak broju diskova u polju. S obzirom na to da su pri posluživanju zahtjeva zaposleni svi diskovi, svaki se zahtjev obavlja pojedinačno.

U *krupno zrnatoj* organizaciji prikazanoj na slici 12.6. jedan se pojas sastoji od nakupine sektora. U takvoj organizaciji nije nužno da svi diskovi sudjeluju u ispunjenju svakog pojedinačnog zahtjeva. Prema tome, polje diskova može istodobno ispunjavati više manjih pojedinačnih zahtjeva, dok se pri zahtjevima za prijenos velikih količina podataka opet angažira više diskova ili čak svi diskovi. Odabir veličine pojasa ovisi o predviđenoj primjeni.

Disk 1	A0	A3	B0	B3
Disk 2	A1	A4	B1	B4
Disk 3	A2	A5	B2	B5

Slika 12.6. Krupnozrnati pojasci smještaj

Međutim, uporaba većeg broja diskova povećava vjerojatnost pojave kvarova. Zbog toga se pri izgradnji višediskovnih sustava nužno nameće uvođenje stanovite zalihosti i tehnikâ za oporavak od kvarova. Uporaba zalihosnih diskova za povećanje pouzdanosti unosi potrebu za dodatnim pohranjivanjima i dobavljanjima podataka, što smanjuje efektivnu brzinu prijenosa. Prema tome, postizanje veće brzine i veće pouzdanosti dva su kontradiktorna zahtjeva.

Postoji veći broj mogućih načina ostvarivanja zalihosnih mehanizama u višediskovnim sustavima. Razmotrit ćemo ih nakon što obrazložimo osnovne pojmove o pouzdanosti.

12.2. Modeliranje zalihosnih sustava

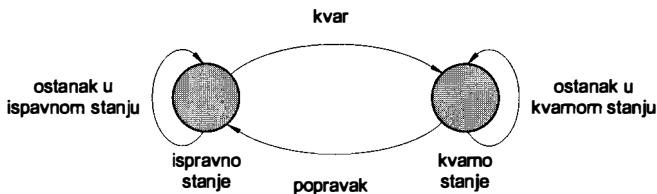
12.2.1. Pouzdanost i nepouzdanost sustava

Popravljive i nepopravljive komponente

Ocjena pouzdanosti može se načiniti za neku komponentu sustava ili za neki uređaj na temelju prikupljenih podataka o velikom broju istovrsnih komponenti, odnosno uređaja.

Pretpostavimo da komponenta ima dva stanja:

- ispravno stanje i
- kvarno stanje.



Slika 12.7. Ispravno i kvarno stanje

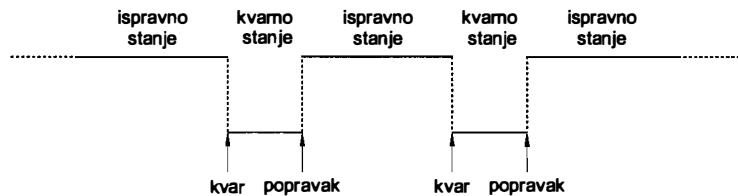
Slika 12.7. prikazuje ta dva stanja te prijelaz između dvaju stanja.

Nova komponenta započinje svoj životni vijek u ispravnom stanju. Kada se dogodi kvar, komponenta prelazi u kvarno stanje.

Komponente mogu biti:

- nepopravljive i
- popravljive.

Nepopravljive komponente nakon događaja kvara ostaju trajno u kvarnom stanju. Tim događajem završava njihov životni vijek. *Popravljive komponente* događajem popravka vraćaju se iz kvarnog stanja natrag u ispravno stanje.



Slika 12.8. Popravljive komponente nakon popravka započinju novi životni ciklus

Slikom 12.8. ilustrirano je vremensko ponašanje popravljive komponente.

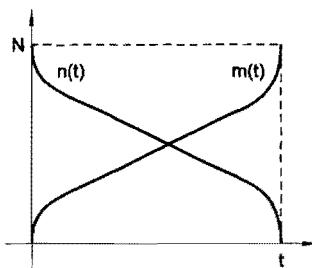
Pri modeliranju ponašanja popravljivih komponenti pretpostavljat će se da one nakon popravka započinju novi životni ciklus.

Pouzdanost i nepouzdanost nepopravljivih komponenti

Razmatranje ponašanja pojedine komponente može se provesti promatranjem velikog broja komponenti. Pretpostavimo da je u jednom trenutku $t = 0$ stavljena u pogon veća količina, recimo, N nepopravljivih komponenti. Možemo reći da je N početna populacija komponenti. Tijekom vremena pojedine komponente će se kvariti, broj ispravnih komponenti $n(t)$ smanjivat će se, a broj će neispravnih komponenti $m(t)$ rasti kao što prikazuje slika 12.9. Nakon određenog vremena pokvarit će se i zadnja komponenta i cijela će se početna populacija pokvariti.

U svakom je trenutku t zbroj ispravnih i neispravnih komponenti jednak početnoj populaciji:

$$n(t) + m(t) = N.$$



Slika 12.9. Broj ispravnih $n(t)$ i neispravnih komponenti $m(t)$ kao funkcije vremena

Na temelju promatranja vremenskog ponašanja cijele populacije ilustrirano slikom može se za pojedinu komponentu odrediti pouzdanost, odnosno nepouzdanost.

Pouzdanost (engl. *reliability*) $R(t)$ neke komponente jest vjerojatnost da se ona u trenutku t nalazi u ispravnom stanju (vjerojatnost da se do trenutka t nije dogodio događaj kvara).

Nepouzdanost (engl. *unreliability*) $F(t)$ neke komponente jest vjerojatnost da se ona u trenutku t nalazi u kvarnom stanju (vjerojatnost da se do trenutka t dogodio događaj kvara).

Promatranjem statističkih podataka za cijelu populaciju može se ustanoviti da je:

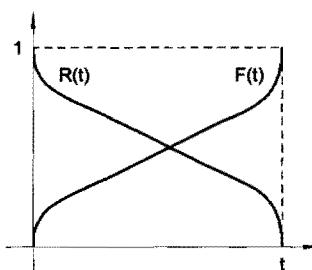
- pouzdanost $R(t)$ proporcionalna dijelu populacije koja će ostati u ispravnom stanju do trenutka t ;
- nepouzdanost $F(t)$ proporcionalna dijelu populacije koja će prijeći u kvarno stanje do trenutka t .

Ako izraz $n(t) + m(t) = N$ podijelimo s N , dobit ćemo:

$$\frac{n(t)}{N} + \frac{m(t)}{N} = 1,$$

odnosno:

$$R(t) + F(t) = 1.$$



Slika 12.10. Funkcija pouzdanosti $R(t)$ i nepouzdanosti $F(t)$

Funkcije pouzdanosti $R(t)$ i nepouzdanosti $F(t)$ prikazane su slikom 12.10.

Očigledno je da vrijedi:

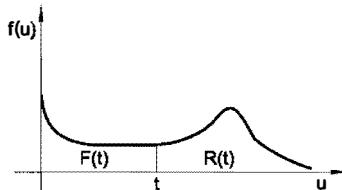
$$\begin{array}{ll} \lim_{t \rightarrow 0} R(t) = 1 & \lim_{t \rightarrow 0} F(t) = 0 \\ \lim_{t \rightarrow \infty} R(t) = 0 & \lim_{t \rightarrow \infty} F(t) = 1. \end{array}$$

Funkcija gustoće vjerojatnosti kvarenja dobit će se derivacijom funkcije $F(t)$:

$$f(t) = \frac{dF(t)}{dt}.$$

Uz danu funkciju gustoće vjerojatnosti kvarenja $f(u)$ dobivamo funkciju nepouzdanosti $F(t)$:

$$F(t) = \int_0^t f(u)du.$$



Slika 12.11. Funkcija gustoće vjerojatnosti kvarenja

Iz slike 12.11. je vidljivo da je $F(t)$ jednako površini ispod krivulje $f(u)$ u vrijednosti $u=t$.

S obzirom na to da je $R(t) + F(t) = 1$, vrijedi:

$$R(t) = 1 - F(t) = 1 - \int_0^t f(u)du = \int_t^\infty f(u)du.$$

Vjerojatnost da će se kvar dogoditi u intervalu (t_1, t_2) jednaka je:

$$F(t_2) - F(t_1) = \int_{t_1}^{t_2} f(u)du.$$

Pri razmatranju pouzdanosti definira se *prosječno vrijeme do pojave kvara* (engl. *Mean Time to Failure – MTTF*) kao očekivanje varijable u , odnosno

$$MTTF = \int_0^\infty uf(u)du.$$

Prema tome, za pojedinačnu komponentu ocjenjujemo pouzdanost, nepouzdanost i prosječno vrijeme do pojave kvara promatranjem statističkih podataka za cijelu populaciju.

Brzina kvarenja (engl. *failure rate*) $r(t)$ definira se kao vjerojatnost pojavljivanja kvara u jedinici vremena u trenutku t , i to za onaj dio populacije koji je do tогa trenutka preživio.

Promotrimo kratki vremenski interval $(t, t + \Delta t)$. U intervalu Δt broj kvarova bit će jednak:

$$r(t) \cdot \Delta t \approx \frac{\overbrace{m(t + \Delta t) - m(t)}^{\text{broj kvarova u intervalu } \Delta t}}{\underbrace{n(t)}_{\text{preživjele komponente do } t}}.$$

Kada se brojnik i nazivnik podijele s početnom veličinom populacije N , dobivamo:

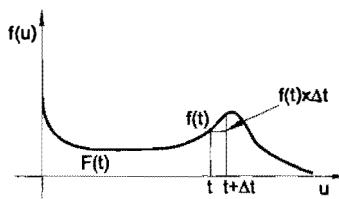
$$r(t) \cdot \Delta t \approx \frac{F(t + \Delta t) - F(t)}{R(t)} = \frac{\int_t^{t+\Delta t} f(u)du}{R(t)}.$$

Za vrlo mali interval Δt bit će:

$$\int_t^{t+\Delta t} f(u)du \approx f(t)\Delta t,$$

te je:

$$r(t) \cdot \Delta t = \frac{f(t)\Delta t}{R(t)}.$$



Slika 12.12. Broj kvarova u intervalu Δt

Kada se lijeva i desna strana podijele s Δt i uzme u obzir da je $R(t) = 1 - F(t)$, slijedi:

$$r(t) = \frac{f(t)}{1 - F(t)},$$

odnosno:

$$r(t) = \frac{dF(t)}{1 - F(t)} = -\frac{d}{dt} \ln[1 - F(t)].$$

Integracijom lijeve i desne strane dobivamo:

$$\int_0^t r(u)du = - \int_0^t \frac{d}{du} \ln[1 - F(u)]du = - \ln[1 - F(u)]|_0^t,$$

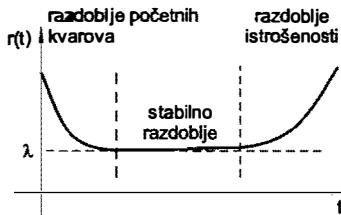
iz čega, s obzirom na to da je $F(0) = 0$, slijedi:

$$\int_0^t r(u)du = - \ln[1 - F(t)].$$

Dakle, može se pisati:

$$F(t) = 1 - e^{-\int_0^t r(u)du}.$$

Za mnoge tehničke komponente vrijedi da je brzina kvarenja $r(t)$ relativno velika na početku njihova korištenja, da je približno konstantna u nekom razdoblju te da se opet povećava pri kraju životnog vijeka komponenti.



Slika 12.13. Brzina kvarenja na početku i na kraju korištenja

U stabilnom je razdoblju $r(t) = \lambda$, tako da je:

$$\int_0^t r(u)du = \int_0^t \lambda du = \lambda t,$$

što vodi na jednostavni model ocjene nepouzdanosti:

$$F(t) = 1 - e^{-\lambda t}.$$

Pouzdanost je komponente, prema tome, jednaka:

$$R(t) = 1 - F(t) = e^{-\lambda t},$$

a funkcija gustoće vjerojatnosti kvarenja jednaka je:

$$f(u) = \lambda e^{-\lambda u}.$$

U stabilnom je razdoblju prosječno vrijeme do pojave kvara, dakle, jednako:

$$MTTF = \int_0^{\infty} u \lambda e^{-\lambda u} du = \frac{1}{\lambda}.$$

Pri analizi pouzdanosti najčešće se upotrebljava ovaj jednostavni model. Podsetimo se da on vrijedi za stabilno razdoblje, u kojem se više ne pojavljuju početni kvarovi (komponente s "dječjim bolestima" uklanjuju se najčešće završnim ispitivanjima nakon proizvodnje) i još se ne pojavljuju učestali kvarovi zbog istrošenosti. Proizvođači komponenti na temelju dugotrajnijih promatranja svojih proizvoda obznanjuju ocjenu za *MTTF*.

Modeliranje procesa popravljanja komponenti

Životni vijek popravljičkih komponenti može se popravljanjem produljiti. Takve će komponente pojedina razdoblja provesti u kvarnom stanju. Postavlja se pitanje kako modelirati postupak popravljanja.

Zbog pojednostavljenja analize često se koristi model jednak onomu za analizu kvarenja (iako za to nema nekog posebnog opravdanja, analiza daje barem osnovni osjećaj o ponašanju popravljičkih komponenti).

Prepostavlja se da vrijeme procesa popravka započinje istoga trenutka kada se dogodio kvar i taj trenutak $t = 0$ proglašavamo početkom popravka.

Neka funkcija $G(t)$ predstavlja vjerojatnost da je popravak obavljen prije trenutka t . Funkcija $G(t)$ ima slična svojstva kao funkcija $F(t)$, tako da je:

$$\lim_{t \rightarrow 0} G(t) = 0,$$

$$\lim_{t \rightarrow \infty} G(t) = 1.$$

Nadalje, funkcija gustoće vrijednosti popravka je:

$$g(t) = \frac{dG(t)}{dt},$$

a srednje vrijeme do popravka (engl. Mean Time to Repair – *MTTR*) dobiva se kao očekivanje:

$$MTTR = \int_0^{\infty} ug(u) du.$$

Brzina popravljanja bit će:

$$m(t) = \frac{g(t)}{1 - G(t)},$$

iz čega proizlazi:

$$G(t) = 1 - e^{-\int_0^t m(u) du}.$$

Ako pretpostavimo da je brzina popravljanja konstantna i iznosi $m(t) = \mu$, dobivamo:

$$G(t) = 1 - e^{-\mu \cdot t},$$

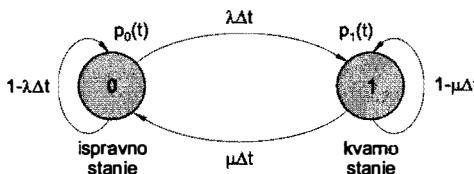
$$g(u) = \mu e^{-\mu \cdot t}.$$

Konačno, prosječno vrijeme trajanja popravka iznosi:

$$MTTR = \frac{1}{\mu}.$$

12.2.2. Model ponašanja popravljive komponente s konstantnim brzinama kvarenja i popravljanja

Ponašanje komponente s konstantnom brzinom kvarenja λ i konstantnom brzinom popravljanja μ jest Markovljev proces. Označimo ispravno stanje indeksom 0, a kvarno stanje indeksom 1.



$p_0(t)$ - vjerojatnost da je komponenta u trenutku t u ispravnom stanju

$p_1(t)$ - vjerojatnost da je komponenta u trenutku t u kvarnom stanju

$\lambda \Delta t$ - vjerojatnost da će se u malom intervalu vremena Δt dogoditi kvar

$\mu \Delta t$ - vjerojatnost da će se u malom intervalu vremena Δt dogoditi popravak

Slika 12.14. Ponašanje jedne komponente s konstantnim brzinama kvarenja i popravljanja

Vjerojatnosti da će se u trenutku $t + \Delta t$ komponenta naći u ispravnom, odnosno kvarnom stanju su sljedeće:

$$p_0(t + \Delta t) = (1 - \lambda \Delta t)p_0(t) + \mu \Delta t p_1(t)$$

$$p_1(t + \Delta t) = \lambda \Delta t p_0(t) + (1 - \mu \Delta t)p_1(t).$$

Nakon sređivanja dobivamo:

$$\frac{p_0(t + \Delta t) - p_0(t)}{\Delta t} = -\lambda p_0(t) + \mu p_1(t)$$

$$\frac{p_1(t + \Delta t) - p_1(t)}{\Delta t} = \lambda p_0(t) - \mu p_1(t).$$

Kada pustimo da $\Delta t \rightarrow 0$, s lijeve strane dobivamo derivacije i dobivamo sustav diferencijalnih jednadžbi:

$$\frac{dp_0(t)}{dt} = -\lambda p_0(t) + \mu p_1(t)$$

$$\frac{dp_1(t)}{dt} = \lambda p_0(t) - \mu p_1(t).$$

Jednadžbe se mogu napisati u matričnom obliku:

$$\begin{bmatrix} \frac{dp_0(t)}{dt} \\ \frac{dp_1(t)}{dt} \end{bmatrix} = \begin{bmatrix} -\lambda & \mu \\ \lambda & -\mu \end{bmatrix} \begin{bmatrix} p_0(t) \\ p_1(t) \end{bmatrix}.$$

Opći oblik rješenja jednadžbi glasi:

$$\begin{aligned} p_0(t) &= C_1 e^{\chi_1 t} + C_2 e^{\chi_2 t} \\ p_1(t) &= C_3 e^{\chi_1 t} + C_4 e^{\chi_2 t}, \end{aligned}$$

gdje su χ_1 i χ_2 svojstvene vrijednosti matrice sustava.

Svojstvene vrijednosti matrice sustava dobivamo iz karakteristične jednadžbe:

$$\left| \chi \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} -\lambda & \mu \\ \lambda & -\mu \end{bmatrix} \right| = \left| \begin{bmatrix} \chi + \lambda & -\mu \\ -\lambda & \chi + \mu \end{bmatrix} \right| = 0,$$

što daje:

$$\chi^2 + (\lambda + \mu)\chi = 0 \implies \chi_1 = 0 \quad \text{i} \quad \chi_2 = -(\lambda + \mu).$$

Prema tome, opći oblik rješenja glasi:

$$\begin{aligned} p_0(t) &= C_1 + C_2 e^{-(\lambda+\mu)t} \\ p_1(t) &= C_3 + C_4 e^{-(\lambda+\mu)t}. \end{aligned}$$

Koefficijente općeg rješenja dobivamo uvrštenjem početnih uvjeta:

$$p_0(0) = 1 \quad \text{i} \quad p_1(0) = 0.$$

Iz diferencijalnih jednadžbi dobiva se dodatno:

$$\frac{dp_0(0)}{dt} = -\lambda, \quad \frac{dp_1(0)}{dt} = \lambda.$$

Uvrštenjem tih početnih uvjeta u rješenja i derivacije rješenja dobivaju se koeficijenti:

$$C_1 = \frac{\mu}{\lambda + \mu}, \quad C_2 = \frac{\lambda}{\lambda + \mu}, \quad C_3 = \frac{\lambda}{\lambda + \mu}, \quad C_4 = -\frac{\lambda}{\lambda + \mu}.$$

Prema tome, konačno rješenje sustava diferencijalnih jednadžbi glasi:

$$\begin{aligned} p_0(t) &= \frac{\mu}{\lambda + \mu} + \frac{\lambda}{\lambda + \mu} e^{-(\lambda+\mu)t} \\ p_1(t) &= \frac{\lambda}{\lambda + \mu} - \frac{\lambda}{\lambda + \mu} e^{-(\lambda+\mu)t} = \frac{\lambda}{\lambda + \mu} \cdot [1 - e^{-(\lambda+\mu)t}]. \end{aligned}$$

Podsjetimo se da je $p_0(t)$ vjerojatnost da se komponenta nalazi u ispravnom stanju u trenutku t ako se u trenutku $t = 0$ nalazila u ispravnom stanju. Ta vjerojatnost podsjeća

na definiciju pouzdanosti $R(t)$ nepopravljivih komponenti. Za popravljive se komponente ona naziva raspoloživošću (engl. *availability*) i obilježava s $A(t)$ te je:

$$A(t) = \frac{\mu}{\lambda + \mu} + \frac{\lambda}{\lambda + \mu} \cdot e^{-(\lambda + \mu)t}.$$

Jednako tako, $p_1(t)$ je vjerojatnost da se komponenta nalazi u kvarnom stanju u trenutku t ako se u trenutku $t = 0$ nalazila u ispravnom stanju. Ta vjerojatnost podsjeća na definiciju nepouzdanosti $F(t)$ nepopravljivih komponenti. Za popravljive se komponente ona naziva neraspoloživošću (engl. *unavailability*) i obilježava s $Q(t)$ te je:

$$Q(t) = \frac{\lambda}{\lambda + \mu} - \frac{\lambda}{\lambda + \mu} e^{-(\lambda + \mu)t} = \frac{\lambda}{\lambda + \mu} [1 - e^{-(\lambda + \mu)t}].$$

Izrazi koji opisuju raspoloživost i neraspoloživost prelaze u izraze za pouzdanost i nepouzdanost kada se u njih uvrsti vrijednost $m = 0$.

Podsjetimo se da je:

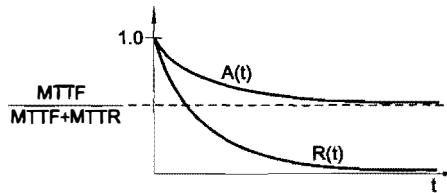
$$\lim_{t \rightarrow \infty} R(t) = 0$$

$$\lim_{t \rightarrow \infty} F(t) = 1.$$

Raspoloživost kod popravljivih komponenti neće pasti na nulu niti će neraspoloživost dosegnuti jedinicu, jer je:

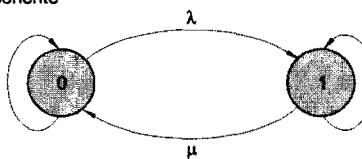
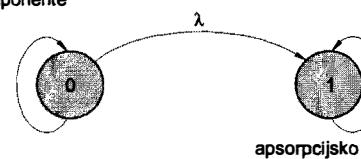
$$\lim_{t \rightarrow \infty} A(t) = \frac{\mu}{\lambda + \mu} = \frac{\frac{1}{MTTR}}{\frac{1}{MTTR} + \frac{1}{MTTF}} = \frac{MTTF}{MTTF + MTTR},$$

$$\lim_{t \rightarrow \infty} Q(t) = \frac{\lambda}{\lambda + \mu} = \frac{\frac{1}{MTTF}}{\frac{1}{MTTR} + \frac{1}{MTTF}} = \frac{MTTR}{MTTF + MTTR}.$$



Slika 12.15. Grafička usporedba raspoloživosti i pouzdanosti

Grafička usporedba pouzdanosti i raspoloživosti prikazana je slikom 12.15. U sljedećoj su tablici prikazani modeli ponašanja popravljivih i nepopravljivih komponenti.

popravljive komponente	nepopravljive komponente
 <p>Kvarenje: $r(t) = \lambda$ $R(t) = e^{-\lambda t}$ $F(t) = 1 - e^{-\lambda t}$ $f(u) = \lambda e^{-\lambda u}$ $MTTF = \frac{1}{\lambda}$ $Q(t) = \frac{\lambda}{\lambda + \mu} [1 - e^{-(\lambda + \mu)t}]$ $A(t) = \frac{\mu}{\lambda + \mu} + \frac{\lambda}{\lambda + \mu} e^{-(\lambda + \mu)t}$ $p'_0(t) = -\lambda p_0(t) + \mu p_1(t)$ $p'_1(t) = \lambda$</p> <p>Popravljanje: $m(t) = \mu$ $G(t) = 1 - e^{-\mu \cdot t}$ $g(u) = \mu e^{-\mu \cdot u}$ $MTTR = \frac{1}{\mu}$</p>	 <p>Kvarenje: $r(t) = \lambda$ $R(t) = e^{-\lambda t}$ $F(t) = 1 - e^{-\lambda t}$ $f(u) = \lambda e^{-\lambda u}$ $MTTF = \frac{1}{\lambda}$ $Q(t) = F(t) = 1 - e^{-\lambda t}$ $A(t) = R(t) = e^{-\lambda t}$ $p'_0(t) = -\lambda p_0(t)$ $p_0(t) - \mu p_1(t)$</p> <p>apsorpcijsko stanje</p>

12.2.3. Modeliranje višekomponentnih sustava

Modeliranje višekomponentnih sustava je relativno složeno. Zbog toga ćemo mi detaljnije razmotriti dvokomponentne sustave i samo specijalne ishode analize poopćiti za sustave s više komponenti.

Promatramo sustav s dvije komponente s parametrima (λ_1, μ_1) i (λ_2, μ_2) . Slika 12.16. prikazuje sva moguća stanja sustava i prijelaze između tih stanja.

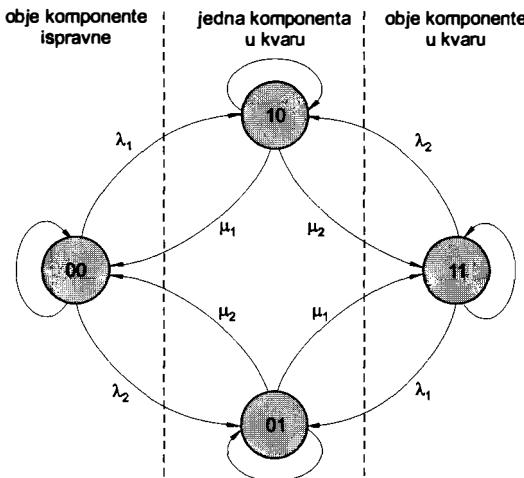
Moguća su sljedeća stanja sustava:

- | | |
|----|--------------------------|
| 00 | obje komponente ispravne |
| 10 | komponenta 1 u kvaru |
| 01 | komponenta 2 u kvaru |
| 11 | obje komponente u kvaru |

Na temelju Markovljeva grafa može se neposredno napisati sljedeći sustav četiriju diferencijalnih jednadžbi koji opisuju ponašanje sustava:

$$p'_{00}(t) = -(\lambda_1 + \lambda_2)p_{00}(t) + \mu_1 p_{10}(t) + \mu_2 p_{01}(t) \quad (1)$$

$$p'_{10}(t) = \lambda_1 p_{00}(t) - (\lambda_2 + \mu_1)p_{10}(t) + \mu_2 p_{11}(t) \quad (2)$$



Slika 12.16. Model dvokomponentnog sustava

$$p'_{00}(t) = \lambda_2 p_{00}(t) - (\lambda_1 + \mu_2)p_{01}(t) + \mu_1 p_{11}(t) \quad (3)$$

$$p'_{11}(t) = \lambda_2 p_{10}(t) + \lambda_1 p_{01}(t) - (\mu_1 + \mu_2)p_{11}(t). \quad (4)$$

Promatrat ćemo dva specijalna slučaja:

- a) sustav ćemo smatrati ispravnim samo ako su obje komponente ispravne, tj. kada se sustav nalazi u stanju 00;
- b) sustav ćemo smatrati ispravnim i onda kada je jedna komponenta neispravna, tj. sustav se smatra ispravnim kada se nalazi stanjima 00, 10 ili 01 (sustav je u kvaru samo kada je u stanju 11).

Slučaj a): Sustav je ispravan samo ako su obje komponente ispravne

S obzirom na to da je ispravno stanje 00, raspoloživost sustava $A_S(t)$ jednaka je vjerojatnosti $p_{00}(t)$. Pri razmatranju sustava ne zanima nas proces popravka komponenti, tako da stanja 10 i 01 postaju apsorpcijska stanja (smatramo da su $m_1 = 0$ i $m_2 = 0$).

Jedandžba (1) prelazi u oblik:

$$p'_{00}(t) = -(\lambda_1 + \lambda_2)p_{00}(t),$$

što uz $p_{00}(0) = 1$ daje:

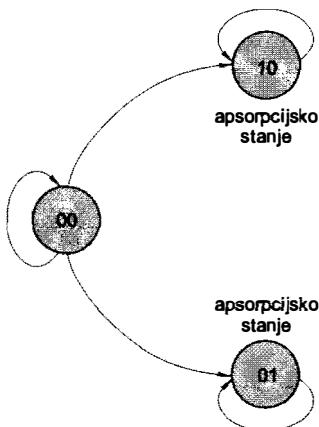
$$p_{00}(t) = e^{-(\lambda_1 + \lambda_2)t}.$$

Prema tome je:

$$A_S(t) = p_{00}(t) = e^{-(\lambda_1 + \lambda_2)t},$$

$$Q_S(t) = 1 - A_S(t) = 1 - e^{-(\lambda_1 + \lambda_2)t},$$

$$q_S(u) = (\lambda_1 + \lambda_2)e^{-(\lambda_1 + \lambda_2)u}.$$



Slika 12.17. Pojednostavljenje Markovljeva grafa

Prosječno vrijeme do pojave kvara sustava bit će, dakle:

$$MTTF_S = \int_0^\infty u(\lambda_1 + \lambda_2)e^{-(\lambda_1 + \lambda_2)u} du = \frac{1}{\lambda_1 + \lambda_2}.$$

Uz $\lambda_1 = \frac{1}{MTTF_1}$ i $\lambda_2 = \frac{1}{MTTF_2}$ dobivamo:

$$MTTF_S = \frac{1}{\frac{1}{MTTF_1} + \frac{1}{MTTF_2}} = \frac{MTTF_1 \cdot MTTF_2}{MTTF_1 + MTTF_2}.$$

U slučaju da je $MTTF_1 = MTTF_2 = MTTF$ bit će:

$$MTTF_S = \frac{1}{2}MTTF.$$

Ovaj se rezultat analize za sustave s dvije komponente može proširiti na sustave s više komponenti pa dobivamo:

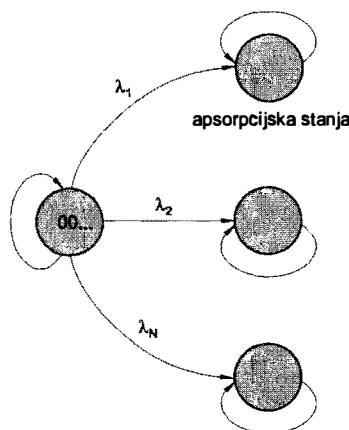
$$p'_{00\dots}(t) = -(\lambda_1 + \lambda_2 + \dots + \lambda_N)p_{00\dots}(t),$$

odnosno:

$$A_S(t) = e^{-(\lambda_1 + \lambda_2 + \dots + \lambda_N)t},$$

$$Q_S(t) = 1 - e^{-(\lambda_1 + \lambda_2 + \dots + \lambda_N)t},$$

$$q_S(u) = (\lambda_1 + \lambda_2 + \dots + \lambda_N)e^{-(\lambda_1 + \lambda_2 + \dots + \lambda_N)u}.$$



Slika 12.18. Sustav je ispravan sve dok se jedna komponenta ne pokvari

Prosječno vrijeme do pojave kvara za sustave s N komponenti bit će, dakle:

$$MTTF_S = \frac{1}{N}MTTF.$$

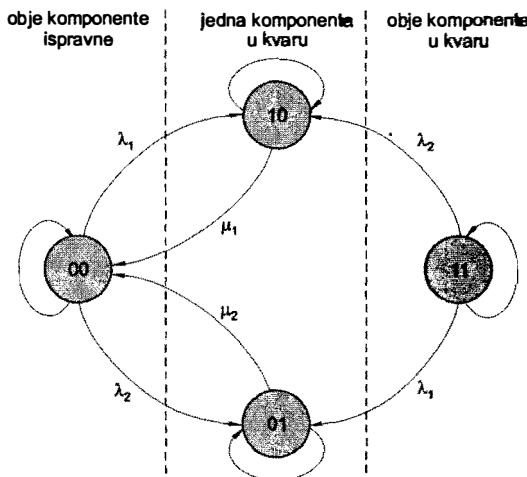
Slučaj b): Sustav je neispravan samo ako su obje komponente u kvarnom stanju

Ispravna su stanja sustava 00, 10 i 01, tako da je raspoloživost sustava jednaka:

$$A_S(t) = p_{00}(t) + p_{10}(t) + p_{01}(t),$$

a neraspoloživost je jednaka:

$$Q_S(t) = p_{11}(t).$$



Slika 12.19. Model dvokomponentnog sustava gdje je sustav u kvaru ako su obje komponente u kvaru

Sustav diferencijalnih jednadžbi dade se pojednostavniti tako da se uzme u obzir da je stanje *11 apsorpcijsko stanje*.

Sustav četiri diferenčialnih jednadžbi pojednostavljuje se i glasi:

$$p'_{00}(t) = -(\lambda_1 + \lambda_2)p_{00}(t) + \mu_1 p_{10}(t) + \mu_2 p_{01}(t) \quad (1b)$$

$$p'_{10}(t) = \lambda_1 p_{00}(t) - (\lambda_2 + \mu_1)p_{10}(t) \quad (2b)$$

$$p'_{01}(t) = \lambda_2 p_{00}(t) - (\lambda_1 + \mu_2)p_{01}(t) \quad (3b)$$

$$p'_{11}(t) = \lambda_2 p_{10}(t) + \lambda_1 p_{01}(t). \quad (4b)$$

Sustav se može dalje pojednoštavnići ako se prepostavi da su dvije komponente jednake, tj. da je:

$$\lambda_1 = \lambda_2 = \lambda \quad \text{i} \quad \mu_1 = \mu_2 = \mu.$$

Sustav diferenčialnih jednadžbi glasi:

$$p'_{00}(t) = -2\lambda p_{00}(t) + \mu[p_{10}(t) + p_{01}(t)] \quad (1c)$$

$$p'_{10}(t) = \lambda p_{00}(t) - (\lambda + \mu)p_{10}(t) \quad (2c)$$

$$p'_{01}(t) = \lambda p_{00}(t) - (\lambda + \mu)p_{01}(t) \quad (3c)$$

$$p'_{11}(t) = \lambda [p_{10}(t) + p_{01}(t)]. \quad (4c)$$

Uvedemo li, nadalje, nove označke za vjerojatnosti tako da je:

- $p_0(t)$ – vjerojatnost da su obje komponente ispravne,
- $p_1(t)$ – vjerojatnost da je jedna komponenta u kvaru,
- $p_2(t)$ – vjerojatnost da su obje komponente u kvaru,

dobivamo:

$$p_0(t) = p_{00}(t)$$

$$p_1(t) = p_{10}(t) + p_{01}(t)$$

$$p_2(t) = p_{11}(t).$$

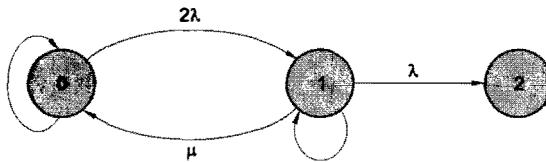
Iz jednadžbi (1c) i (4c) i iz zbroja jednadžbi (2c) i (3c) slijedi:

$$p'_0(t) = -2\lambda p_0(t) + \mu p_1(t) \quad p_0(0) = 1$$

$$p'_1(t) = 2\lambda p_0(t) - (\lambda + \mu)p_1(t) \quad p_1(0) = 0$$

$$p'_2(t) = \lambda p_1(t) \quad p_2(0) = 0.$$

Markovljev graf koji vodi na taj sustav jednadžbi izgleda ovako:



Slika 12.20. Pojednostavljenje Markovljeva grafa

Raspoloživost takvog sustava jednaka je vjerojatnosti da se do trenutka t nije pojavio dvostruki kvar, odnosno:

$$A_S(t) = p_0(t) + p_1(t).$$

Neraspoloživost je jednaka vjerojatnosti da se dogodio dvostruki kvar te je:

$$Q_S(t) = p_2(t).$$

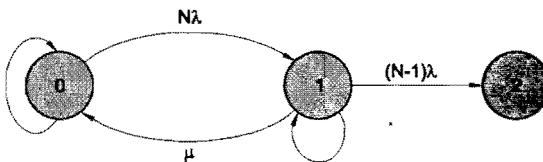
Funkcija gustoće vjerojatnosti dvostrukog kvara je:

$$q_S(t) = \frac{dQ_S(t)}{dt} = p'_2(t) = \lambda p_1(t)$$

te će prosječno vrijeme do pojave dvostrukog kvara biti:

$$MTTF_S = \int_0^{\infty} u \lambda p_1(u) du.$$

Nećemo tražiti prosječno vrijeme do pojave dvostrukog kvara za sustav od samo dvije jednake komponente već za sustav koji se sastoji od N komponenti. Markovljev graf za N komponenti dobit ćemo poopćenjem grafa sustava dviju komponenti.

Slika 12.21. Markovljev graf za N komponenti

Sustav jednadžbi koje se dobiju na temelju takvog grafa glasi:

$$\begin{aligned} p'_0(t) &= -N\lambda p_0(t) + \mu p_1(t) & p_0(0) &= 1 \\ p'_1(t) &= N\lambda p_0(t) - [(N-1)\lambda + \mu] p_1(t) & p_1(0) &= 0 \\ p'_2(t) &= (N-1)\lambda p_1(t) & p_2(0) &= 0. \end{aligned}$$

I ovdje vrijedi da je:

$$q_S(t) = \frac{dQ_S(t)}{dt} = p'_2(t) = (N-1)\lambda p_1(t)$$

tako da će nas zanimati rješenje za $p_1(t)$. Uočimo da prve dvije jednadžbe određuju rješenje. Prve dvije jednadžbe napisane u matričnom obliku izgledaju ovako:

$$\begin{bmatrix} p'_0(t) \\ p'_1(t) \end{bmatrix} = \begin{bmatrix} -N\lambda & \mu \\ N\lambda & -(N-1)\lambda + \mu \end{bmatrix} \begin{bmatrix} p_0(t) \\ p_1(t) \end{bmatrix}.$$

Rješenje sustava ima sljedeći oblik:

$$\begin{aligned} p_0(t) &= A_1 e^{\chi_1 t} + A_2 e^{\chi_2 t} \\ p_1(t) &= B_1 e^{\chi_1 t} + B_2 e^{\chi_2 t}, \end{aligned}$$

gdje su χ_1 i χ_2 svojstvene vrijednosti matrice sustava:

$$\begin{vmatrix} \chi + N\lambda & -\mu \\ -N\lambda & \chi[(N-1)\lambda + \mu] \end{vmatrix} = 0,$$

odnosno:

$$\chi^2 + [(2N-1)\lambda + \mu]\chi + N(N-1)\lambda^2 = 0.$$

Uočimo da su obje svojstvene vrijednosti realne i negativne:

$$\chi_{1,2} = \frac{-(2N-1)\lambda + \mu \pm \sqrt{\mu^2 + 2(2N-1)\lambda\mu + \lambda^2}}{2}.$$

Za dalji postupak bit će nam korisni izrazi za $\chi_1 + \chi_2$ i $\chi_1\chi_2$. Ako se opći oblik kvadratne jednadžbe uz realne korijene napiše ovako:

$$\begin{aligned} (\chi - \chi_1)(\chi - \chi_2) &= 0 \\ \chi^2 - (\chi_1 + \chi_2)\chi + \chi_1\chi_2 &= 0, \end{aligned}$$

iz karakteristične jednadžbe dobivamo:

$$\begin{aligned} \chi_1 + \chi_2 &= -(2N-1)\lambda + \mu, \\ \chi_1\chi_2 &= N(N-1)\lambda^2. \end{aligned}$$

Nakon ove pripreme pogledajmo rješenje za $p_1(t)$. Koeficijente B_1 i B_2 odredit ćemo iz početnih uvjeta:

$$p_1(0) = 0 \quad \text{i} \quad p'_1(0) = N\lambda.$$

Dobili smo:

$$p_1(t) = B_1 e^{\chi_1 t} + B_2 e^{\chi_2 t},$$

što deriviranjem daje:

$$p'_1(t) = B_1 \chi_1 e^{\chi_1 t} + B_2 \chi_2 e^{\chi_2 t}.$$

Za $t = 0$ dobivamo:

$$\begin{aligned} B_1 + B_2 &= 0 & B_1 &= \frac{N\lambda}{\chi_1 - \chi_2} \\ B_1 \chi_1 + B_2 \chi_2 &= N\lambda & B_2 &= -\frac{N\lambda}{\chi_1 - \chi_2}. \end{aligned}$$

Konačno je:

$$p_1(t) = \frac{N\lambda}{\chi_1 - \chi_2} (e^{\chi_1 t} - e^{\chi_2 t})$$

pa je:

$$q_S(t) = \frac{N(N-1)}{\chi_1 - \chi_2} \lambda^2 (e^{\chi_1 t} - e^{\chi_2 t}).$$

Prosječno vrijeme do pojave dvostrukog kvara u sustavu odredit ćemo ovako:

$$MTTF_S = \int_0^\infty u q_S(u) du = \frac{N(N-1)}{\chi_1 - \chi_2} \lambda^2 \left(\int_0^\infty ue^{\chi_1 u} du - \int_0^\infty ue^{\chi_2 u} du \right).$$

Lako se pokazuje da je:

$$\int_0^\infty ue^{\chi u} du = \frac{1}{\chi^2}$$

te je:

$$MTTF_S = \frac{N(N-1)}{\chi_1 - \chi_2} \lambda^2 \left(\frac{1}{\chi_1^2} - \frac{1}{\chi_2^2} \right) = \frac{N(N-1)}{\chi_1 - \chi_2 \lambda^2} \cdot \frac{\chi_2^2 - \chi_1^2}{\chi_1^2 \chi_2^2} = N(N-1) \lambda^2 \frac{-(\chi_1 + \chi_2)}{\chi_1^2 \chi_2^2}.$$

Uzmemo li u obzir da je:

$$\chi_1 + \chi_2 = -[(2N-1)\lambda + \mu] \quad i \quad \chi_1 \chi_2 = N(N-1)\lambda^2,$$

dobivamo:

$$MTTF_S = \frac{1}{N(N-1)} \frac{\mu \lambda^{-2}}{N(N-1)} + \frac{2N-1}{N(N-1)} \lambda^{-2},$$

odnosno:

$$MTTF_S = \frac{1}{N(N-1)} \cdot \frac{MTTF^2}{MTTR} + \frac{2N-1}{N(N-1)} \cdot MTTF.$$

PRIMJER 12.1.

Pretpostavimo da su za neki tip diskova deklarirana prosječna vremena:

$$MTTF = 200\,000 \text{ sati} = 22.8 \text{ godina} \quad i$$

$$MTTR = 1 \text{ sat.}$$



Pogledajmo koliko je srednje vrijeme do pojave kvara u $MTTF_S$ sustavu s N diskova, i to:

- a) ako se kvarom sustava smatra kvar jednog od diskova,
- b) ako se kvarom sustava smatra kvar dvaju diskova.

a) $MTTF_S = \frac{1}{N} MTTF.$

N	$\frac{1}{N} MTTF$
2	100 000 sati
3	66 666 sati
10	20 000 sati
100	2 000 sati

b) $MTTF_S = \frac{1}{N(N-1)} \cdot \frac{MTTF^2}{MTTR} + \frac{2N-1}{N(N-1)} MTTF.$

$$\left. \begin{aligned} \frac{MTTF^2}{MTTR} &= \frac{(2 \cdot 10^5)^2}{1} = 4 \cdot 10^{10} \\ MTTF &= 2 \cdot 10^5 \end{aligned} \right\} \text{U ovom se slučaju drugi član u izrazu može zanemariti}$$

te je $MTTF_S \approx \frac{1}{N(N-1)} \cdot \frac{MTTF^2}{MTTR}.$

N	$\frac{1}{N(N-1)} \cdot \frac{MTTF^2}{MTTR}$
2	$\frac{1}{2} \cdot 4 \cdot 10^{10} = 2 \cdot 10^{10}$ sati
3	$\frac{1}{6} \cdot 4 \cdot 10^{10} = 6.6 \cdot 10^9$ sati
10	$\frac{1}{90} \cdot 4 \cdot 10^{10} = 4.44 \cdot 10^8$ sati
100	$\frac{1}{9900} \cdot 4 \cdot 10^{10} = 4.04 \cdot 10^6$ sati

Prema tome, višediskovni sustav nužno se mora organizirati tako da se tolerira barem jednostruki kvar diska.

12.3. Načini zalihosne organizacije diskova

Ustanovili smo da postoji veći broj mogućih načina ostvarivanja zalihosnih mehanizama u višediskovnim sustavima koji omogućuju povećanje njihove raspoloživosti.

Polje diskova s ugrađenom zalihošću (redundancijom) dobilo je naziv *Redundant Array of Independent Disks* od čega je izvedena kratica *RAID* (u prvo vrijeme je naziv glasio *Redundant Array of Inexpensive Disks*, što je iz komercijalnih razloga napušteno). U literaturi se spominje šest osnovnih načina organizacije višediskovnih sustava: od *RAID 0* do *RAID 6*.

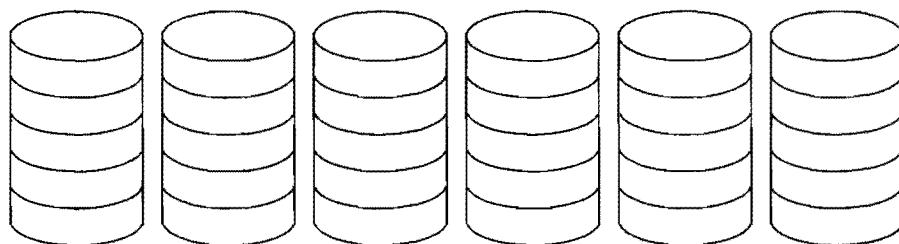
12.3.1. RAID 0 – nezalihosna organizacija

U organizaciji *RAID 0* nema zalihosti. Svi su diskovi podatkovni diskovi. Primjenjuje se pojšna organizacija, tj. podaci su razdijeljeni na sve diskove. Neka je, primjerice, jedan pojas veličine jedne nakupine sektora. Datoteka veličine nekoliko nakupina sektora pohranjena je na više diskova. Time se u idealnom slučaju postiže ubrzanje pristupa podacima za faktor koji je jednak broju diskova.

Prosječno vrijeme do pojave kvara sustava jednako je:

$$MTTF_S = \frac{1}{N}MTTF.$$

Ovakva je organizacija prikladna kada je raspoloživost manje važna od kapaciteta i brzine.



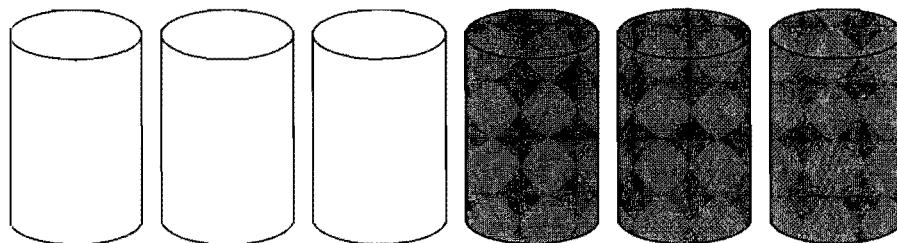
Slika 12.22. RAID 0 diskovna organizacija

Prosječno vrijeme $MTTF_S$ u višediskovnim se sustavima naziva i prosječnim vremenom do gubitka podataka (engl. *Mean Time to Data Loss* – *MTTDL*) pa je za *RAID 0*:

$$MTTDL = \frac{1}{N}MTTF.$$

12.3.2. RAID 1 – zrcaljena organizacija

U organizaciji *RAID 1* svaki disk ima svoju kopiju – zrcaljeni disk (engl. *mirrored disk*).



Slika 12.23. RAID 1 diskovna organizacija

Prosječno vrijeme do pojave gubitka podataka je:

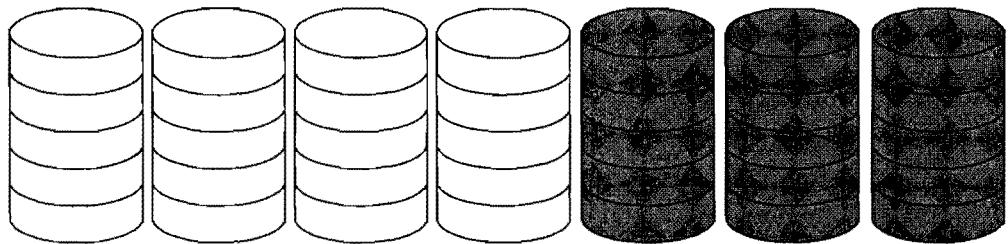
$$MTTDL = \frac{1}{N(N-1)} \cdot \frac{MTTF^2}{MTTR} + \frac{2N-1}{N(N-1)} \cdot MTTF.$$

Svojstva *RAID 1* organizacije:

- svako se pisanje mora obaviti dva puta;
- čitanje može biti brže nego kod *RAID 0* jer se može obaviti s diska koji postigne kraće vrijeme postavljanja glava;
- koristi se često za pohranjivanje baza podataka
- nedostatak: pola kapaciteta potroši se za postizanje zalihosti.

12.3.3. RAID 2 – organizacija zasnovana na Hammingovim kodovima

U organizaciji *RAID 2* za korekciju jednostrukih pogrešaka koristi se Hammingov kôd. Za korekciju nakupine od N bitova potrebno je dodati $\log_2 N + 1$ korekcijskih bitova. Tako su za $N = 4$ potrebna 3 korekcijska bita, a za $N = 128$ potrebno ih je 8.



Slika 12.24. RAID 2 diskovna organizacija

Prosječno vrijeme do pojave gubitka podataka je:

$$MTTDL = \frac{1}{N(N-1)} \cdot \frac{MTTF^2}{MTTR} + \frac{2N-1}{N(N-1)} \cdot MTTF.$$

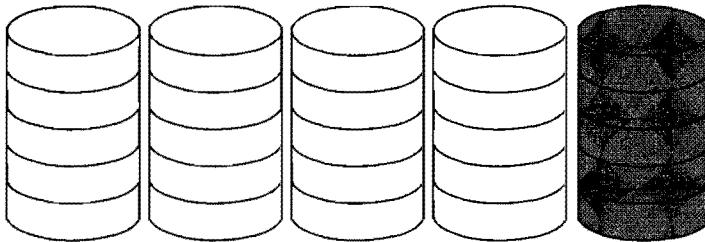
Korekcijski bitovi omogućuju:

- određivanje diska na kojem je nastala pogreška;
- ispravljanje pogrešnog bita (tj. određivanje vrijednosti bita na ustanovljenom mjestu).

U višediskovnom se sustavima, međutim, može na drugi način ustanoviti koji se disk pokvario tako da nam je mjesto pokvarenog bita poznato i dovoljan je samo jedan paritetni korekcijski bit za utvrđivanje vrijednosti bita na poznatom mjestu.

12.3.4. RAID 3 – paritetna organizacija sitne zrnatosti

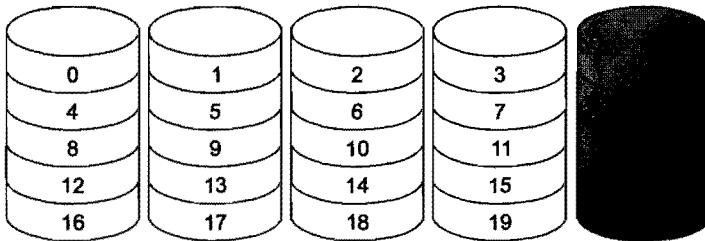
U organizaciji **RAID 3** za korekciju jednostrukih pogrešaka koristi se jedan paritetni disk (jer položaj pokvarenog diska znamo!). Pojasevi se svode na veličinu sektora tako da se praktički može govoriti o bitovnoj zrnatosti.



Slika 12.25. RAID 3 diskovna organizacija

12.3.5. RAID 4 – paritetna organizacija krupne zrnatosti

U organizaciji **RAID 4** za korekciju jednostrukih pogrešaka također se koristi jedan paritetni disk, ali su pojasevi veći.



Slika 12.26. RAID 4 diskovna organizacija

Pri kvaru jednog diska mogu se rekonstruirati podaci pokvarenog pojasa uz pomoć pri-padnog pojasa paritetnog diska (primjerice, paritetni pojас P_0 štiti podatkovne pojaseve 0, 1, 2 i 3).

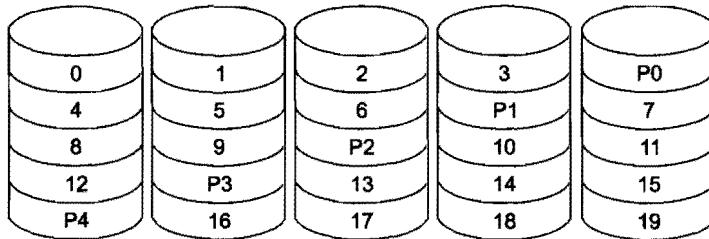
Pisanje u jedan od pojaseva zahtijeva četiri pristupa do diskova (zbog izračunavanja paritetnog zaštitnog sadržaja):

- čitanje starih podataka;
- čitanje iz paritetnog pojasa;
- pisanje novih podataka;
- pisanje u paritetni pojas.

Stoga paritetni disk može postati usko grlo!

12.3.6. RAID 5 – paritetna organizacija krupne zrnatosti s raspodijeljenim paritetnim pojasevima

U organizaciji *RAID 5* paritetni se pojasevi raspoređuju po svim diskovima.



Slika 12.27. RAID 5 diskovna organizacija

Najprikladniji je način razmještaja pojaseva prikazan slikom (zovu ga: *left-symmetric parity distribution*). Kada se pr takvom smještaju pojasevi čitaju sekvenčijski, onda će se pristupiti svim diskovima prije nego se ponovno pristupi prvom disku.

PRIMJER 12.2.



Promotrimo skupinu od G diskova koji su zaštićeni *RAID 5* načinom. Tada, ako se zanemari drugi pribrojnik, vrijedi:

$$MTTDL_G = \frac{1}{G(G-1)} \cdot \frac{MTTF^2}{MTTR}$$

Neka postoji K takvih skupina tako da ima ukupno $N = K \cdot G$ diskova. Ako se pojavi kvar jedne od K grupa (što znači dvostruki kvar unutar te grupe), onda imamo gubitak podataka te je:

$$MTTDL_N = \frac{1}{K} \cdot \frac{1}{G(G-1)} \cdot \frac{MTTF^2}{MTTR} = \frac{1}{N(G-1)} \cdot \frac{MTTF^2}{MTTR}$$

Uz:

$$MTTF = 200\,000 \text{ sati}$$

$$MTTR = 1 \text{ sat}$$

$$N = 96$$

$$G = 16$$

dobivamo:

$$MTTSL_{96} = \frac{1}{96 \cdot 15} \cdot \frac{2 \cdot 10^5}{1} = 2.78 \cdot 10^7 \text{ sati} = 3170 \text{ godina.}$$

Najčešće se upotrebljavaju *RAID 1* i *RAID 5*. Treba naglasiti da je iskoristivost kapaciteta mnogo veća u *RAID 5* organizaciji.

Označimo s C ukupni kapacitet sustava od N diskova.

RAID 1:

$$C = \frac{1}{2}N.$$

RAID 5 s K paritetnih diskova:

$$\begin{aligned} C &= N - K, \quad N = K \cdot G \\ \Rightarrow C &= N - \frac{N}{G} = \frac{G-1}{G}N. \end{aligned}$$

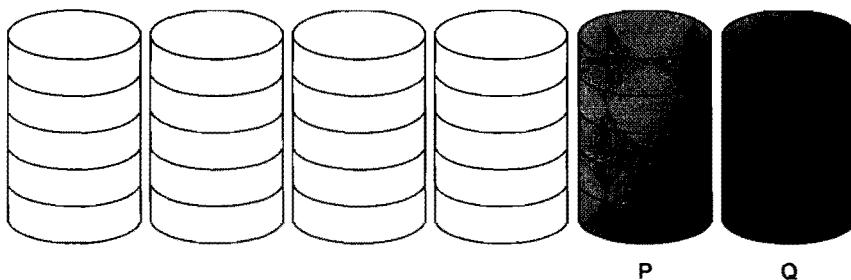
Primjerice, uz:

$$G = 16 \Rightarrow C = \frac{15}{16}N$$

$$\text{ili za } G = 2 \Rightarrow C = \frac{1}{2}N.$$

12.3.7. RAID 6 – organizacija sa zaštitom od dvostrukog kvara ($P + Q$ zalihost)

Podloga za RAID 6 organizaciju jesu *Reed-Solomonovi* kodovi. Postoje dva zaštitna diska (odnosno dva zaštitna pojasa za svaku zaštićenu skupinu diskova koji se ravnomjerno raspoređuju po svim diskovima).



Slika 12.28. RAID 6 diskovna organizacija

Ako se načini K skupine od po G diskova, tada imamo ukupno $N = K \cdot G$ diskova, a prosječno vrijeme do pojave gubitka podataka bit će:

$$MTTDL_N = \frac{1}{N(G-1)(G-2)} \cdot \frac{MTTF^3}{MTTR^2}.$$

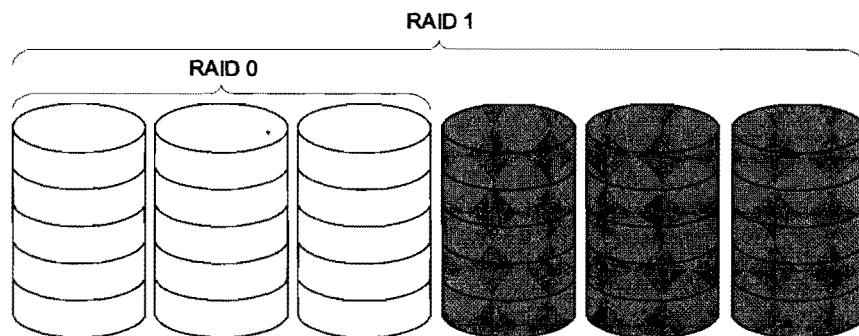
12.3.8. Višerazinski RAID sustavi

Opisani RAID sustavi mogu se kombinirati tvoreći složene ili višerazinske RAID sustave. Tako se, primjerice, RAID 0 i RAID 1 mogu kombinirati na dva načina, tj. uz pomoć

njih mogu se ostvariti sljedeći višerazinski RAID sustavi:

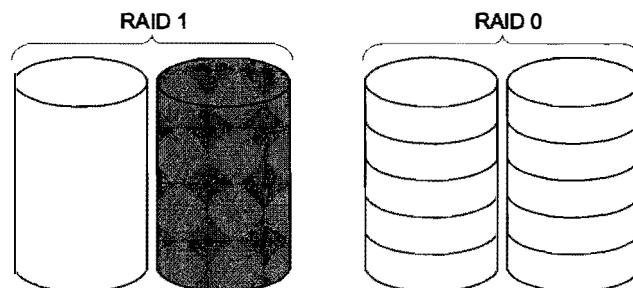
- RAID 0+1 i
- RAID 10.

Kako bi se ostvario RAID 0+1 sustav¹ potrebno je prvo ostvariti pojASNu organizaciju na polovici raspoloživih diskova, što čini RAID 0 diskovnu organizaciju. Druga polovica diskova je zrcalna slika pojASNe organizacije. Dakle, u višoj je razini ostvaren RAID 1 sustav. Za ostvarenje takvog sustava potrebna su minimalno četiri diska.



Slika 12.29. RAID 0 + 1 diskovna organizacija

Slično kao i u sustavu RAID 0+1, u RAID 10 sustavu² ostvarena je i pojASNa i zrcaljena organizacija diskova, ali obrnutim redoslijedom, kako je to prikazano na slici 12.30. Minimalni broj diskova i u ovom je slučaju 4.



Slika 12.30. RAID 10 diskovna organizacija

Na sličan se način mogu kombinirati i ostali RAID sustavi pa je tako moguće, primjerice, ostvariti RAID 30, RAID 50, RAID 0+5, RAID 51, RAID 1+5 sustave.

¹ U literaturi se još koriste nazivi RAID 01 ili RAID 0/1.

² U literaturi se još koriste nazivi RAID 1+0 ili RAID 1/0.

**PITANJA ZA PROVJERU ZNANJA 12**

- 1.** Na koje se sve načine može ubrzati prijenos podataka s diska u radni spremnik i obrnuto?
- 2.** Što se dobiva pojasnom organizacijom diskova?
- 3.** Opisati sitno i krupno zrnatu pojasnu organizaciju.
- 4.** Navesti prednosti i nedostatke uvođenja zalihosnih diskova.
- 5.** Objasniti pojmove: pouzdanost i raspoloživost. U čemu se oni razlikuju?
- 6.** Skicirati Markovljev lanac za trokomponentni sustav koji omogućuje oporavak od jednostrukih pogrešaka. Naznačiti vjerojatnosti prijelaza iz stanja u stanje u stacionarnom stanju s konstantnom brzinom kvarenja λ i konstantnim brzinom popravljanja μ .
- 7.** Navesti osnovne načine zalihosne organizacije diskova.
- 8.** Skicirati i opisati RAID 0/1/2/3/4/5/6 sustav.
- 9.** Skicirati i opisati višerazinski RAID 10 sustav sa 6 diskova.
- 10.** Skicirati i opisati višerazinski RAID 0 + 1 sustav s 8 diskova.

Literatura

- [Anderson, 2001] R. ANDERSON, "Security Engineering", J. Wiley & Sons, 2001.
- [Andrews, 2000] G. R. ANDREWS, "Foundations of multithreaded, parallel, and distributed programming", Addison-Wesley, 2000.
- [Biggerstaff, 1986] T. J. BIGGERSTAFF, "System software tools", Prentice-Hall, 1986.
- [Bovet, 2003] D. P. BOVET AND M. CESATI, "Understanding the Linux Kernel", O'Reilly & Associates, 2, 2003.
- [Brunson, 2002] R. BRUNSON, "Linux and Windows 2000 Integration Toolkit: A Complete Resource", J. Wiley & Sons, 2002.
- [Buttazzo, 2000] G. C. BUTTAZZO, "Hard real-time systems: predictable scheduling algorithms and applications", Kluwer Academic Publishers, 2000.
- [Comer, 1984] D. COMER, "Operating system design, the Xinu approach", Prentice-Hall, 1984.
- [Coulouris, 1994] G. COULOURIS, J. DOLLIMORE AND T. KINDBERG, "Distributed systems: concept and design", Addison-Wesley, 1994.
- [Custer, 1993] H. CUSTER, "Inside Windows NT", Microsoft Press, 1993.
- [Daemen, 1998] JOAN DAEMEN AND VINCENT RIJMEN, "AES Proposal: The Rijndael Block Cipher", dokument dostupan na Internet adresi:
<http://www.cryptosoft.de/docs/Rijndael.pdf>, 1998.
- [Engel, 1999] J. ENGEL, "Programming for the Java virtual machine", Addison-Wesley, 1999.
- [Fortier, 1988] P. J. FORTIER, "Design of Distributed Operating Systems", McGraw-Hill, 1988.
- [Foster, 1999] I. FOSTER AND C. KESSELMAN, "The Grid: Blueprint for a New Computing Infrastructure", Morgan Kaufmann, 1999.
- [Furber, 2000] S. FURBER, "ARM System-on-chip Architecture", Addison-Wesley, 2000.
- [Fussell, 1995] D. S. FUSSELL, M. MALEK, "Responsive computer systems: steps toward fault-tolerant real-time systems", Kluwer Academic Publishers, 1995.
- [Glossbrenner, 1990] A. GLOSSBRENNER AND N. ANIS, "Glossbrenner's Complete Hard Disk Handbook", Osborne McGraw-Hill, 1990.
- [Goswami, 2003] S. GOSWAMI, "Internet Protocols: Advances, Technologies and Applications", Kluwer Academic Publishers, 2003.
- [Habermann, 1976] A. N. HABERMANN, "Introduction to operating System Design", Science Research Associates, 1976.
- [Hailpern, 1982] B. T. HAILPERN, "Verifying Concurrent Processes Using Temporal Logic", Lecture Notes in Computer Science, vol. 129, Springer-Verlag, 1982.
- [Hoare, 1985] C. A. R. HOARE, "Communicating Sequential Processes", Prentice-Hall, 1985.
- [Hughes, 1997] C. HUGHES AND T. HUGHES, "Object-Oriented Multithreading Using C++", J. Wiley & Sons, 1997.
- [Kaisler, 1983] S. H. KAISLER, "The design of operating systems for small computer systems", J. Wiley & Sons, 1983.
- [Kleiman, 1996] S. KLEIMAN, D. SHAH AND B. SMAALDERS, "Programming with Threads", Prentice-Hall, 1996.
- [Kopetz, 1997] H. KOPETZ, "Real-Time Systems: Design Principles for Distributed Embedded Applications", Kluwer Academic Publishers, 1997.
- [Kou, 1977] W. KOU, "Networking security and standards", Kluwer Academic Publishers, 1977.
- [Langendorfer, 1994] H. LANGENDORFER AND B. SCHNOR, "Verteilte Systeme", Carl Hanser Verlag, 1994.
- [Laplante, 1993] P. A. LAPLANTE, "Real-Time Systems Design and Analysis – An Engineer's Handbook", IEEE Computer Society Press, 1993.
- [Leach, 1994] R. J. LEACH, "Advances topics in UNIX", J. Wiley & Sons, 1994.
- [Lemieux, 2001] J. LEMIEUX, "Programming in the OSEK/VD Environment", CMP Books, 2001.
- [Madnick, 1974] S. E. MADNICK AND J. J. DONOVAN, "Operating systems", McGraw-Hill, 1974.

- [Maekawa, 1987] M. MAEKAWA, A. E. OLDEHOEFT AND R. R. OLDEHOEFT, "Operating systems: advanced concepts", The Benjamin/Cummings Publishing Company, 1987.
- [Malek, 1994] M. MALEK, "Responsive computing", Kluwer Academic Publishers, 1994.
- [Miller, 1997] D. D. MILLER, "OpenVMS Operating System Concepts", Digital Press, 1997.
- [Nehmer, 1987] J. NEHMER, "Experiences with Distributed Systems", Lecture Notes in Computer Science, vol. 209, Springer-Verlag, 1987.
- [norma AES, 2001] "Federal Information Processing Standards Publication No. 197: Advanced Encryption Standard", dostupno na Internet adresi:
<http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>, 2001.
- [Nutt, 2001] G. NUTT, "Kernel Projects for Linux", Addison-Wesley, 2001.
- [Nutt, 1999] G. NUTT, "Operating System Projects Using Windows NT", Addison-Wesley, 1999.
- [Nutt, 2001] G. J. NUTT, "Operating Systems: A Modern Perspective", Addison-Wesley Longman Publishing, 2001.
- [Ogorman, 2003] J. O'GORMAN, "The Linux process manager: the internals of scheduling, interrupts and signals", J. Wiley & Sons, 2003.
- [Olsen, 1998] D. R. OLSEN, "Developing User Interfaces", Macmillan Publishing Company, 1998.
- [Peterson, 1983] J. L. PETERSON AND A. SILBERSCHATZ, "Operating system concepts", Addison-Wesley, 1983.
- [Pfaffenberger, 2001] B. PFAFFENBERGER, "Linux Networking Clearly Explained", Morgan Kaufmann, 2001.
- [Pieper, 1977] F. PIEPER, "Einführung in die Programmierung paralleler Prozesse", R. Oldenbourg Verlag, 1977.
- [Rajkumar, 1991] R. RAJKUMAR, "Synchronization in real-time systems: A Priority Inheritance Approach", Kluwer Academic Publishers, 1991.
- [Ribarić, 1990b] S. RIBARIĆ, "Arhitektura mikroprocesora", Tehnička knjiga, 4, 1990.
- [Ribarić, 1990a] S. RIBARIĆ, "Arhitektura računala", Školska knjiga, 1990.
- [Ribarić, 1996] S. RIBARIĆ, "Arhitektura računala RISC i CISC", Školska knjiga, 1996.
- [Ribarić, 1990c] S. RIBARIĆ, "Naprednije arhitekture mikroprocesora", Školska knjiga, 1990.
- [Saulpaugh, 1999] T. SAULPAUGH AND C. MIRHO, "Inside the JavaOS Operating System", Addison-Wesley, 1999.
- [Schneier, 1996] B. SCHNEIER, "Applied Cryptography", J. Wiley & Sons, 1996.
- [Shaw, 1974] A. C. SHAW, "The Logical Design of Operating Systems", Prentice-Hall, 1974.
- [sigurnost, 2009] Studentski radovi iz područja računalne sigurnosti, ur. M. Golub, dostupno na Internet adresi <http://sigurnost.zemris.fer.hr>
- [Silberschatz, 2000] A. SILBERSCHATZ, P. B. GALVIN AND G. GAGNE, "Applied operating system", J. Wiley & Sons, 1, 2000.
- [Silberschatz, 1994] A. SILBERSCHATZ AND P. GALVIN, "Operating System Concepts", Addison-Wesley, 4, 1994.
- [Silberschatz, 2003] A. SILBERSCHATZ, P. B. GALVIN AND G. GAGNE, "Operating system concepts", J. Wiley & Sons, 6, 2003.
- [Stallings, 1992] W. STALLINGS, "Operating Systems", Macmillan Publishing Company, 1992.
- [Standard, 1986] "IEEE Trial-Use Standard: Portable Operating System for Computer Environments", The Institute of Electrical and Electronics Engineers, Inc., 1986.
- [Stevens, 1999] W. R. STEVENS, "UNIX network programming", Prentice-Hall, 2, vol. 1,2, 1999.
- [Tanenbaum, 1996] A. S. TANENBAUM, "Computer Networks", Prentice-Hall International, 3, 1996.
- [Tanenbaum, 1995] A. S. TANENBAUM, "Moderne Betriebssysteme", Prentice-Hall, 2, 1995.
- [Tanenbaum, 1986] A. S. TANENBAUM, "Operating systems: Design and Implementation", Prentice-Hall, 1986.
- [Thies, 1994] K. D. THIES, "Multitasking: Grundlagen, Betriebssystem-Kern-Funktionen für INTEL-Prozessoren, parallele Programmierung, Realzeit-Systeme", Carl Hanser Verlag, 1994.
- [Vahid, 2002] F. VAHID AND T. GIVARGIS, "Embedded System Design: A Unified Hardware/ Software Introduction", J. Wiley & Sons, 2002.
- [Weikum, 2002] G. WEIKUM AND G. VOSSEN, "Transactional Information Systems", Morgan Kaufmann, 2002.
- [Werner, 1992] D. WERNER, "Theorie der Betriebssysteme", Carl Hanser Verlag, 1992.



Kazalo pojnova

- 3DES, 293
 adresa, 12
 adresni dio sabirnice, 13
 adresni m^{od}uregistar, 17
 adresni prostor, 12, 90, 187
 AES, 296, 299
 aktivno stanje dretve, 93
 API, (ii), 6, 150
 absolutna adresa, 199
 arhitekture ARM, 36
 aritmetičko-logička jedinka, 9, 16, 18
 AS, 346
 asimetrični kriptosustav, 290, 304
 atributi datoteke, 241
 autentičnost, 286
 autentifikacija, 282
 autentifikacijski poslužitelj, 346
 autentifikator, 348
 autorizacija, 282, 286
 bajt, 11
 barijera, 145
 bazni registar, 202
 bespriječnost, 285, 286
 binarna datoteka, 241
 binarni semafor, 97, 103, 137
 binomna razdioba, 164
 bit prisutnosti, 213, 223
 bitovni prikaz, 246
 blokirana stanja dretvi, 97
 broj dolazaka, 162
 brojač, 304
 brojački semafor, 109, 120
 brojilo događaja, 109
 brzina kvarenja, 372
 CA, 351
 CBC, 302
 centralizirani protokol, 273
 certifikacijski centar, 351
 CFB, 303
 ciklička dretva, 69, 117
 ciklus dobavljanja, 15
 ciklus pohranjivanja, 14
 cilindar, 191
 cjevodovna obrada podataka, 62
 CRL, 360
 CTR, 304
 čista stranica, 222
 čitanje datoteke, 250
 čvrsto povezani višeprocesorski sustav, 56
 Charmicleovi brojevi, 314
 datotečna tablica, 246, 249
 datotečna kazaljka, 242
 datotečni podsustav, (iv)
 datotečni sustav, 241
 datoteka, 2, 241
 Dekkerov postupak, 76
 dekodiranje instrukcije, 19
 dekriptiranje, 288
 DESX, 293
 determinističko ponašanje sustava, 159
 Diffie-Hellmanov postupak, 324
 digitalna omotnica, 317
 digitalni certifikat, 350
 digitalni pečat, 319
 digitalni potpis, 317, 319
 dijeljeni spremnički prostor, 258
 dijeljenje spremnika, 58
 dinamičko podešavanje raspodjele okvira, 231
 dinamičko raspoređivanje spremnika, 204
 direktorij, 241
 direktorij stranica, 218
 diskretni logaritam, 308
 dodjeljivač sabirnice, 58
 dodjeljivanje po redu prispijeda, 176
 događaj dolaska, 158
 događaj odlaska, 158
 dohvati instrukcije, 19
 domena, 65
 dopunski vanjski spremnik, 188
 dopuštanje pristupa, 344
 dozvola, 348
 dozvola za pristup poslužitelju, 348
 dretva, (iii), 29, 64, 91
 ECB, 301
 eksponencijalna razdioba, 163, 167
 elektronička bilježnica, 301
 Eulerov teorem, 307
 Eulerova phi funkcija, 306
 faktor iskorištenja, 162, 172, 174, 179
 FIFO strategija, 224
 fizički adresni prostor, 202, 211
 fragmentacija, 205, 209, 247
 funkcija, 24
 funkcija gustoće vjerovatnosti, 167
 generator, 308
 generator takta, 18
 globalni logički sat, 272
 gospodarenje okvirima, 229
 heurističko ispitivanje po Miller-Rabinu, 314
 hijerarhijska izgradnja sustava, 4

hijerarhijski pristup, (iii)
 homogeni višeprocesorski sustav, 58, 112
 IDEA, 294
 identifikacija, 282
 indeks, 308
 infrastruktura javnih ključeva, 350
 instrukcije, 15
 instrukcije dretve, 64
 instrukcije procesora, 21
 instruksijska dretva, 30
 instruksijski registar, 17
 instruksijski skup, 20, 21
 integritet, 285
 Internet protokol, 261
 inverzija prioriteta, 145
 IP, 261
 ispitati i postaviti, 84, 112
 ispitni lanac, 43
 izgladnjivanje, 145, 253
 izlazak iz jezgre, 102
 izlazne naprave, 33
 izmišljanje poruka, 285
 jasni tekst, 288
 jednokratna bilježnica, 291
 jednostavna autentifikacija, 336, 338
 jezgra, (iv), 89
 jezgrevi način rada, 39
 jezgrina funkcija, 53, 90
 kazaljka stoga, 39
 Kerberos, 345
 Kerchoffov princip, 290
 kineski teorem ostataka, 308
 klijent, 267
 kodni segment, 206
 kodomena, 65
 kompilator, 3
 komunikacija razmjenom poruka, 264
 komunikacijski kanal, 289
 konačno polje, 296
 kongruentnost, 305
 kontekst dretve, 32, 40, 92
 korisnički adresni prostor, 188
 korisnički način rada, 39
 korisničko sučelje, 2
 kriptirani tekst, 288
 kriptiranje, (iv), 288
 kriptoanalitičar, 290
 kriptosustav, 289
 kritični odsječak, 69, 103
 krupno zrnata pojasma organizacija, 367
 kružno dodjeljivanje procesora, 178
 kućanski posao, 41
 kvant vremena, 178, 182
 kvartet, 12
 labavo povezani sustav, 59
 Lamportov protokol, 80

Lamportov raspodijeljeni protokol, 274
 lažno predstavljanje, 285
 lista dozvola za pristup objektima, 345
 lista postojećih dretvi, 93
 lista prava pristupa objekta, 345
 Littleovo pravilo, 161
 logički adresni prostor, 202, 211
 lokalni logički sat, 271
 lozinka, 343
 LRU strategija, 224
 magnetski diskovi, 189
 mali Fermatov teorem, 307
 Markovljev lanac, 171
 MD5, 320
 međudretvena komunikacija, 118
 međunarodna telekomunikacijska unija, 260
 međunarodne organizacije za normizaciju, 260
 međuregistri, 14
 međusobno isključivanje, (iii), 63, 68, 103
 međuspremnik, 120
 mikrojezgra, 151
 model jezgre, 89
 model sklopovskih komponenti, 7
 modularno potenciranje, 307
 monitor, (iv), 133, 142
 monolitne jezgre, 150
 MPI, 265
 MTTF, 371
 MTTR, 374
 napadač, 290
 napredni kriptosustav, 296
 nedeterministički model, 163
 nedjeljivi sabirnički ciklusi, 84
 nehomogeni višeprocesorski sustav, 112
 neporecivost, 286
 neposredni pristup spremniku, 54
 nepouzdanost, 370
 nestrukturirana datoteka, 242
 nezavisni podzadaci, 66
 nezavisnost, (iii), 66
 obavljanje operacije, 19
 obavljanje ulazno-izlaznih operacija, 110
 objektni model jezgre, 115
 obnoviti kontekst, 40
 obostrana autentifikacija, 340
 obrada prekida, 40, 42, 45, 50
 očekivanje slučajne varijable, 165, 168, 173
 odgađanje izvođenja, 98
 OFB, 303
 ograda jezgre, 113
 ograničeni spremnik, 175
 okosnice, 263
 oktet, 11
 okvir, 211
 omogućiti prekidanje, 41

opći semafor, 98, 106
 operacijski sustav, 1
 opis diskovnog prostora, 246
 opisnik datoteke, 241, 245, 247, 249
 opisnik ili deskriptor dretve, 92
 opisnik virtualnog procesnog adresnog prostora, 215
 oponašanje indeksne strukture, 243
 optički diskovi, 189
 optimalna strategija, 224
 osnovni korijen, 308
 otkucaj sata, 90
 otpornost na izračunavanje originala, 323
 otpornost na izračunavanje poruke koja daje isti sažetak, 324
 otpornost na kolizije, 324
 otvaranje datoteke, 250

paralelizam, 68
 paralelni putovi, 67
 pasivno stanje, 93
 PC, 18
 pekarski algoritam, 80
 periferijske naprave, 33
 periodni poslovi, 161
 Petersonov postupak, 78
 petlje čekalice, 38
 PKM, 350
 podatkovni dio, 13
 podatkovni međuregistar, 17
 podatkovni pojas, 367
 podatkovni segment, 206
 podsustav za prihvat prekida, 42
 pohraniti kontekst, 40
 Poissonova razdioba, 163
 pomoćni spremnik, 188
 ponašanje sustava, (iv)
 popis opozvanih certifikata, 360
 poravnate adrese, 14
 poricanje, 285
 POSIX, 153
 pošiljatelj, 265
 poslužitelj za dodjelu dozvola, 346
 poslužitelj, 120, 158, 267
 posluživanje po redu prispjeća, 179, 252
 posluživanje pregledavanjem, 253
 posluživanje kružnim pregledavanjem, 254
 posluživanje s najkratim vremenom premještanja, 253
 potprogram, 24
 potpuni zastoj, (iv), 75, 127, 145, 149
 potrošač, 117, 139, 175
 pouzdanost, 370
 povjerljivi informacijski kanal, 289
 povjerljivost, 286
 poziv jezgre, 53
 poziv sustava, 53
 poziv udaljenih procedura, 266

prekid, 44, 90
 prekid od sata, 90
 prekid od ulazno-izlaznih naprava, 90
 prekidanje, 284
 prekidni način rada, 39
 prekidni rad, (iii)
 prekidni signal, 39
 preklopni način uporabe radnog spremnika, 209
 prenošenje vrijednosti parametara, 28
 prijava za rad, 343
 prijenosni protokoli, 261
 primatelj, 265
 primjenska razina, 261
 primjenski program, 4
 prioritet, 44
 prioriteti dretvi, 184
 pripravno stanje dretve, 94
 priručni spremnik, 19, 235
 prisluškivanje, 284
 pristupne točke, 264
 pristupni sklop, 34
 problem pet filozofa, 130, 141
 procedura, 24
 proces, (iii), 30, 61
 procesni adresni prostor, 215
 procesni informacijski blok, 199
 procesni kontrolni blok, 199
 procesor, 13
 procesorski čip, 21
 program, 2, 30, 61
 programska petlja, 24
 programski prekid, 53, 90
 programsko brojilo, 15, 17
 proizvođač, 117, 139, 175
 promašaj stranice, 224
 promjena konteksta, 32, 65, 184, 204
 promjena sadržaja poruka, 285
 prosječni broj događaja, 166
 prosječni broj dolazaka, 160, 169
 prosječni broj poslova, 160, 169, 173
 prosječno trajanje obrade, 169
 prosječno trajanje zadržavanja, 169, 173
 prosječno vrijeme do pojave kvara, 371
 prosječno zadržavanje poslova, 159
 protokol "dvožičnog rukovanja", 37
 protokol nasljeđivanja prioriteta, 148
 protokol Ricarta i Agrawala, 275
 protokol s putujućom značkom, 273
 protokol stropnog prioriteta, 148
 protokoli, 260
 protokolni slog, 260

računalni proces, 30
 radni skup okvira, 232
 radni skup, 232
 radni skup stranica, 232

radni spremnik, (iv), 11, 13
 radno čekanje, 35, 145
 RAID 0+1, 392
 RAID 0, 386
 RAID 1, 387
 RAID 10, 392
 RAID 2, 388
 RAID 3, 389
 RAID 4, 389
 RAID 5, 390
 RAID 6, 391
 RAID, 386
 raspodijeljeni dijeljeni spremnički prostor, 268
 raspodijeljeni sustav, (iv), 260
 raspodjela ključeva, 327
 raspoloživost, 286
 razmjena datoteka, 257
 razmjena poruka, (iv)
 razmjena poruka između procesa, 259
 red čekanja, 99, 120, 158
 red odgođenih dretvi, 99
 red poruka, 123
 red pripravnih dretvi, 94, 95, 158
 red semafora, 98
 redni broj okvira, 213
 redni broj stranice, 213
 registar kazaljke stoga, 17
 registar ograde, 205
 registar stanja, 17
 registri, 17
 relativne adrese, 200
 Rijndael, 299
 rotacijsko kašnjenje, 193
 RSA, 309
 rutina, 24
 sabirnički ciklus, 11
 sabirnički sustav, 10
 sakupljanje otpadaka, 206
 satni algoritam, 229
 sažetak poruke, 317
 sektor, 191, 246
 SHA, 322
 sigurnosna stijena, 360
 sigurnosni zahtjevi, 286
 sigurnost računalnih sustava, (iv)
 simetrični kriptosustavi, 290
 sinkronizacija, 63
 sinkronizacija dretvi, 126
 sinkronizacijski semafor, 108
 sitno zrnata pojasna organizacija, 367
 sklop za prihvrat prekida, 49
 slikovno korisničko sučelje, (i)
 slučajna varijabla, 163
 spojni modul, 266
 spojni pristupi, 263
 spremnički međusklop, 202
 srednje vrijeme do popravka, 374

stalne particije, 201
 stanje dretve, 92, 100
 stanje okvira, 230
 statičko raspoređivanje, 201
 staza, 190
 stog, 25, 28
 stog dretve, 64
 stogovni segment, 206
 straničenje na zahtjev, 219
 straničenje, 210
 stranice, 211
 strategije zamjene stranica, 222, 224
 strojni oblik programa, 6
 strojni program, 3, 18
 struktura podataka jezgre, 91
 stvaranje datoteke, 249
 sučelje, 2
 sučelje operacijskog sustava za primjenske programe, (ii), 150
 sučelje prema primjenskim programima, 6
 sučelje primjenskih programa, 2
 sučelje programa, (ii)
 sustav dretvi, 66
 sustav podzadataka, 66
 sustavski adresni prostor, 90, 188
 sustavski način rada, 39
 sustavski registar kazaljke stoga, 90
 svežak, 246
 širina pristupa, 13
 tablica, 241
 tablica prevođenja, 213
 tablice stranica, 218
 tajnost, 286
 TAS, 84
 TCP, 261
 teorem dijeljenja, 305
 teorija masovnog posluživanja, 163
 teorija redova, 163
 TGS, 346
 TGT, 349
 trajanje poslova, 159
 trajanje posluživanja, 159, 161
 trajanje postavljanja glave, 193
 trajanje prijenosa podataka, 193
 trajanje traženja staze, 193
 trajanje zadržavanja u redu, 159
 trajanje zadržavanja u sustavu, 158
 UDP, 261
 ulančavanje, 302
 ulazak u jezgru, 91, 102, 151
 ulazna dozvola, 347
 ulazna naprava, 33
 ulaznica, 347
 ulazno-izlazna naprava, (iii), 33
 uljez, 290
 unutarnja fragmentacija, 202
 upravljačka jedinka, 9, 16, 17

upravljački dio, 13
upravljački elektronički sklop, 34
utrostručeni DES, 293
vanjska fragmentacija, 202
virtualni spremnik, (iv)
višediskovni sustav, 367
višedretveni proces, (iii)
višedretveni rad, 31
višedretvenost, 62
višeprocesorska računala, 31
višeprocesorski sustav, (iii), 57, 111
višepramski rad, 1, 61
višezadačni rad, 61

višezadačnost, (iii)
viši programski jezik, 61
Von Neumannov model, 9, 16
vratiti se iz prekidnog načina rada, 41
vremenska svojstva diskova, 193
vrste napada na sigurnost, 283
X.509 certifikat, 356
zadatak, 60
zaglavna listi, 92
zamjena programa, 196
zaštitna pravila, 344
zastavica, 17
zavisnost, (iii), 66