OPERATIVNI SISTEMI
(problemi i struktura)

Miroslav Hajduković

NOVI SAD, 2020.

PREDGOVOR

U ovoj knjizi se izlažu struktura operativnih sistema, principi njihovog funkcionisanja, problemi vezani za pravljenje operativnih sistema, kao i načini rešavanja ovih problema. Deo objašnjenja, navedenih u ovoj knjizi, je u tradiciji operativnog sistema UNIX.

Podrazumeva se da čitalac ove knjige poznaje arhitekturu računara i programski jezik C++.

U ovom izdanju knjige su ispravljene greške, uočene u tekstu njenog prethodnog izdanja, a preuređen je i proširen njen tekst.

Autor se zahvaljuje kolegama dr Žarku Živanovu i Aleksandru Andrejeviću na pomoći prilikom tehničke pripreme ovog izdanja knjige.

1 UVOD

1.1 ZADATAK OPERATIVNOG SISTEMA

Operativni sistem je program koji objedinjuje u skladnu celinu raznorodne delove računara i sakriva od korisnika one detalje funkcionisanja ovih delova koji nisu bitni za korišćenje računara. Znači, operativni sistem ima dvostruku ulogu. S jedne strane, on upravlja sastavnim delovima računara, kao što su procesor, kontroleri i radna memorija, sa ciljem da oni budu što celishodnije upotrebljeni. S druge strane, operativni sistem stvara za krajnjeg korisnika računara pristupačno radno okruženje, tako što pretvara računar od mašine koja rukuje bitima, bajtima i blokovima u mašinu koja rukuje datotekama i procesima.

1.2 POJAM DATOTEKE

Pojam datoteke obuhvata **sadržaj** i **atribute** datoteke. Sadržaj datoteke predstavljaju korisnički podaci. U atribute datoteke spada, na primer, veličina datoteke ili vreme njenog nastanka. Atributi datoteke se čuvaju u **deskriptoru datoteke**.

1.2.1 Uloga datoteke

Datoteke su namenjene za trajno čuvanje podataka. Pristup ovim podacima se svodi na čitanje i pisanje sadržaja datoteka. Pre čitanja ili pisanja sadržaja datoteke, potrebno je omogućiti pristup podacima iz datoteke (na primer, prebaciti deskriptor datoteke iz masovne memorije u radnu memoriju). Zato čitanju i pisanju sadržaja datoteke obavezno prethodi njeno otvaranje, radi pripreme zahtevanog pristupa podacima. Iza čitanja i pisanja sadržaja datoteke obavezno sledi njeno zatvaranje (na primer, radi prebacivanja podataka i deskriptora datoteke iz radne memorije u masovnu memoriju). Time se sačuvaju atributi i sadržaj datoteke i ujedno onemogući pristupanje njenim podacima do njenog novog otvaranja.

1.3 POJAM PROCESA

Pojam procesa obuhvata **aktivnost**, **sliku** i **atribute** procesa. Aktivnost procesa odgovara angažovanju procesora na izvršavanju korisničkog programa. Slika procesa obuhvata adresni prostor procesa sa naredbama izvršavanog programa, stekom i podacima koji se obrađuju u toku izvršavanja programa. U atribute procesa spadaju, na primer, stanje procesa i njegov prioritet. Atributi procesa se čuvaju u **deskriptoru procesa**.

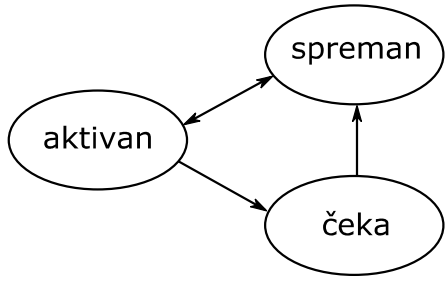
1.3.1 Stanje i prioritet procesa

Tipična stanja procesa su: "**aktivan**", "**čeka**" i "**spreman**". Proces je aktivan kada procesor izvršava program. On čeka kada nisu ispunjeni neophodni preduslovi za obradu podataka. Proces je spreman kada samo zauzetost procesora onemogućuje izvršavanje programa.

Od prioriteta procesa zavisi kada će spreman proces da postane aktivan. Podrazumeva se da je aktivan uvek proces sa najvišim prioritetom, jasno pod pretpostavkom da se razmatra jednoprocesorski računar (sa jednim procesorom namenjenim za izvršavanje korisničkih programa). Ako postoji nekoliko procesa najvišeg (istog) prioriteta, tada je bitna ravnomerna raspodela procesorskog vremena između njih. Ona se postiže ako aktivni proces prepušta procesor spremnom procesu istog (najvišeg) prioriteta čim istekne unapred određeni vremenski interval. Ovaj interval se naziva kvantum (*quantum*). Trenutke isticanja kvantuma označavaju prekidi sata. Obrada ovakvih prekida sata izaziva prevođenje aktivnog procesa u stanje "spreman" i preključivanje procesora na onaj od ostalih spremnih procesa najvišeg prioriteta koji je najduže u stanju "spreman". Pri tome aktivirani proces prelazi iz stanja "spreman" u stanje "aktivan".

Aktivan proces prelazi u stanje "čeka" kada postane nemoguć nastavak njegove aktivnosti. Nastavak njegove aktivnosti postaje moguć tek po prestanku važenja datog razloga čekanja. Kada se to desi, proces iz stanja "čeka" prelazi u stanje "spreman".

Moguće izmene stanja procesa prikazuje Slika 1.1.



Slika 1.1: Moguće izmene stanja procesa

1.3.2 Uloga procesa

Procesi omogućuju bolje iskorišćenje računara (procesora) i njegovu bržu reakciju na dešavanje vanjskih događaja.

Istovremeno postojanje više procesa omogućuje da se procesor preključi sa aktivnog procesa na spreman proces kada aktivni proces treba da pređe u stanje "čeka". Na taj način procesor ostaje iskorišćen dok god ima spremnih procesa.

Bržu reakciju na dešavanje vanjskih događaja omogućuje proces najvišeg prioriteta. Dok ovakav hitni proces čeka da se desi vanjski događaj, aktivan je neki drugi manje prioritetan proces. Kada se desi vanjski događaj, obrada odgovarajućeg prekida prevodi hitni proces iz stanja "čeka" u stanje "spreman". To dovodi do preključivanja procesora sa prekinutog procesa na hitni proces. Hitni proces u toku svoje aktivnosti odreaguje na dešavanje vanjskog događaja. Nakon toga on prepušta procesor prekinutom procesu, jer se vraća u stanje "čeka" dok se ne desi novi vanjski događaj, jasno, ako se u međuvremenu nisu desili novi vanjski događaji. Ako su se takvi događaji desili, njihova dešavanja se registruju u obradama odgovarajućih prekida. Tada se podrazumeva da hitni proces ne prelazi u stanje "čeka" nakon reakcije na prvi događaj, nego da ostaje aktivan

sve dok ne odreaguje na sve u međuvremenu registrovane događaje. Da bi prethodno bilo moguće, potrebno je da su prekidi omogućeni u toku aktivnosti hitnog procesa.

1.3.3 Pojam niti

Aktivnost procesa karakteriše redosled u kome se izvršavaju naredbe programa. Ovaj redosled se naziva **trag** (*trace*) procesa. Proces je **sekvencijalan** ako je njegov trag određen u vreme programiranja, odnosno ako zavisi samo od obrađivanih podataka. Trag sekvencijalnog procesa može da se prikaže kao **nit** (*thread*) koja povezuje izvršavane naredbe u redosledu njihovog izvršavanja. Nit sekvencijalnog procesa nasleđuje njegove atribute, znači njegovo stanje i njegov prioritet.

Mana sekvencijalnih procesa je da su neosetljivi na vanjske događaje. To se može pokazati na primeru editiranja teksta. Za editiranje teksta su važne, između ostalog, dve radnje. Prva je posvećena interakciji sa korisnikom, a druga je zadužena za čuvanje unesenog teksta (za periodično smeštanje unesenog teksta na masovnu memoriju, radi sprečavanja njegovog gubljenja u slučaju nestanka napajanja). Ako je editorski proces sekvencijalan, tada se pomenute radnje odvijaju jedna za drugom, znači sekvencijalno (Listing 1.1).

Listing 1.1: Sekvencijalni editorski proces

```
...
for(;;) {
   do_editor_command();
   if(time_to_save_data())
      save_data();
}
...
```

Za vreme trajanja druge radnje (**save_data()**) nije moguća interakcija sa korisnikom (**do_editor_command()**). To je ozbiljan nedostatak. On se može otkloniti, ako editorski proces postane nesekvencijalan. U tom slučaju pomenute dve radnje imaju razne prioritete i odvijaju se nesekvencijalno (Listing 1.2).

Listing 1.2: Nesekvencijalni editorski proces

```
...
for(;;) {
   do_editor_command_in_foreground();
}
...
for(;;) {
   if(time_to_save_data())
      save_data_in_background();
}
...
```

Prioritetnija, hitna radnja je posvećena interakciji sa korisnikom (**do_editor_command_in_foreground()**), a manje prioritetna pozadinska radnja je posvećena čuvanju teksta (**save_data_in_background()**). Pod pretpostavkom da je hitna radnja zaustavljena, jer nema komandi od korisnika, pozadinska radnja može da se odvija sve dok, na primer, vanjski događaj poput pritiska dirke na tastaturi ne najavi početak interakcije sa korisnikom. Tada se zaustavlja pozadinska radnja, radi nastavljanja hitne radnje. Kada se obavi korisnička komanda u okviru hitne radnje, a hitna radnja se zaustavi u očekivanju nove komande, nastavlja se pozadinska radnja. Zahvaljujući preplitanju hitne i pozadinske radnje, u toku editiranja nema perioda bez odziva. Podrazumeva se da opisanom nesekvencijalnom editorskom procesu odgovaraju dve niti. Prioritetnija nit je pridružena hitnoj radnji (ponavljanju izvršavanja funkcije **do_editor_command_in_foreground()**), a manje prioritetna nit je pridružena pozadinskoj radnji (ponavljanju izvršavanja funkcije **save_data_in_background()**). Takođe se podrazumeva da je prioritetna nit u stanju "čeka" dok je hitna radnja zaustavljena. Tada je aktivna manje prioritetna nit. Kada se desi vanjski događaj poput pritiska dirke na tastaturi, tada obrada odgovarajućeg prekida prevodi prioritetnu nit u stanje "spremna", pa se procesor preključuje sa prekinute manje prioritetne niti na prioritetnu nit. Tada manje prioritetna nit postaje spremna, a prioritena nit postaje aktivna. Po obavljanju interakcije sa korisnikom u okviru aktivnosti prioritetne niti, ona se vrati u stanje "čeka", što dovodi do preključivanja procesora na prekinutu manje prioritetnu nit. Da bi opisano rukovanje nitima bilo moguće, prioritet, stanje i stek se ne vezuju za proces, nego za njegove niti. Znači svaka nit procesa ima svoj prioritet, svoje stanje, svoj stek, pa i svoj deskriptor. Za niti istog procesa se podrazumeva da nisu potpuno nezavisne, odnosno da sarađuju razmenom podataka. Tako, u slučaju nesekvencijalnog editorskog procesa, manje prioritetna nit se brine o čuvanju teksta koga pripremi prioritetna nit.

Procesi sa više istovremeno (*concurrently*) postojećih niti se nazivaju **konkurentni procesi**. Oni odgovaraju izvršavanju **konkurentnih programa**. U slučaju jednoprocesorskog računara istovremeno postojanje više niti znači da je započeto, a da nije završeno izvršavanje više relativno nezavisnih delova istog programa. Svakom od

ovih izvršavanja odgovara posebna nit. Podrazumeva se da samo jedna od niti može biti u stanju "aktivna", a da su ostale ili u stanju "spremna" ili u stanju "čeka". Preključivanja procesora se jednog od ovih izvršavanja na drugo, zavisno od vanjskih događaja, uzrokuju da se odgovarajuće niti prepliću u redosledu koji nije određen u vreme programiranja, što znači da trag konkurentnog procesa zavisi od redosleda preplitanja njegovih niti, odnosno da zavisi od slučaja.

1.4 STRUKTURA OPERATIVNOG SISTEMA

Unutrašnji izgled ili struktura operativnog sistema se može lakše sagledati ako se zauzme stanovište da je zadatak operativnog sistema da upravlja **fizičkim i logičkim delovima** računara. Pošto fizički delovi računara obuhvataju procesor, kontrolere i radnu memoriju, a logički delovi računara obuhvataju datoteke i procese, sledi da se operativni sistem, odnosno, preciznije rečeno, njegovo jezgro (*kernel*), može raščlaniti na module namenjene rukovanju procesorom, kontrolerima, radnom memorijom, datotekama i procesima.

1.4.1 Modul za rukovanje procesorom

Zadatak modula za rukovanje procesorom je da preključuje procesor sa jedne niti na drugu. Pri tome ove niti mogu pripadati istom ili raznim procesima. Sa stanovišta modula za rukovanje procesorom ključna razlika između niti koje pripadaju istom procesu i niti koje pripadaju raznim procesima je da su prve niti u adresnom prostoru istog procesa (da bi mogle da sarađuju), dok su druge niti u adresnim prostorima raznih procesa. Zato, u toku preključivanja procesora između niti istog procesa ne dolazi do izmene adresnog prostora procesa, pa je ovakvo preključivanje brže (kraće) nego preključivanje procesora između niti raznih procesa.

Modul za rukovanje procesorom ostvaruje svoj zadatak tako što uvodi **operaciju preključivanja**. Pozivi ove operacije dovode do preključivanja procesora.

1.4.2 Modul za rukovanje kontrolerima

Zadatak modula za rukovanje kontrolerima je da upravlja raznim ulaznim i izlaznim uređajima koji su zakačeni za kontrolere. U ovakve ulazno-izlazne uređaje spadaju tastatura, miš, ekran, štampač, odnosno uređaji masovne memorije kao što su disk ili *CD* jedinice. Pošto upravljanje ulazno-izlaznim uređajima zavisi od vrste uređaja, modul za rukovanje kontrolerima se sastoji od niza komponenti, nazvanih **drajveri**. Svaki od drajvera je specijalizovan za jednu vrstu kontrolera, opslužuje jednu klasu ulazno-izlaznih uređaja i ima zadatak da konkretan ulazno-izlazni uređaj predstavi u apstraktnom obliku sa jednoobraznim i pravilnim načinom korišćenja. Tako, na primer, drajver diska stvara predstavu apstraktnog diska. Za razliku od stvarnog diska, apstraktni disk, umesto staza i sektora, sadrži niz blokova koji su označeni rednim brojevima. Drajver diska, preslikavajući blokove na staze i sektore, omogućuje pristup blokovima apstraktnog diska, radi ulaza (preuzimanja podataka), odnosno radi izlaza (pohranjivanja podataka).

Modul za rukovanje kontrolerima ostvaruje svoj zadatak tako što njegovi drajveri

uvode (drajverske) **operacije ulaza i izlaza**. Pozivi ovih operacija dovode do prenosa podataka na relaciji između radne memorije i ulazno-izlaznih uređaja. U okviru ovih operacija se, pozivom operacije preključivanja, zaustavlja aktivnost niti za čiji račun se operacije ulaza i izlaza obavljaju dok se zatraženi prenos podataka ne završi. Do nastavaka aktivnosti niti može doći čim ulazno-izlazni uređaj javi, posredstvom mehanizma prekida, da je zatraženi prenos podataka završen. Obradu ovakvih prekida preuzimaju obrađivači prekida, koji, takođe, ulaze u sastav drajvera. Oni omogućuju nastavak aktivnosti niti, radi čega se, eventualno, opet poziva operacija preključivanja. Pošto pozivi operacija preključivanja mogu biti posledica dešavanja prekida, trenutak preključivanja je u opštem slučaju nepredvidiv, kao što su nepredvidivi i trenuci dešavanja prekida.

1.4.3 Modul za rukovanje radnom memorijom

Zadatak modula za rukovanje radnom memorijom je da vodi evidenciju o slobodnoj radnoj memoriji radi zauzimanja zona slobodne radne memorije, odnosno radi oslobađanja prethodno zauzetih zona radne memorije. Kada podržava virtuelnu memoriju, ovaj modul se brine i o prebacivanju sadržaja stranica između radne i masovne memorije. To znači da se iz modula za rukovanje radnom memorijom pozivaju operacije ulaza i izlaza.

Modul za rukovanje radnom memorijom ostvaruje svoj zadatak tako što uvodi **operacije zauzimanja i oslobađanja**. Pozivi ovih operacija dovode do zauzimanja i oslobađanja zona radne memorije.

1.4.4 Modul za rukovanje datotekama

Zadatak modula za rukovanje datotekama je da omogući otvaranje i zatvaranje datoteka, odnosno čitanje i pisanje njihovog sadržaja. Radi toga ovaj modul vodi evidenciju o blokovima (masovne memorije) u kojima se nalaze sadržaji datoteka. On se, takođe, brine i o prebacivanju delova sadržaja datoteka između radne i masovne memorije. To znači da se iz modula za rukovanje datotekama pozivaju operacije ulaza i izlaza. Pošto su za pomenuto prebacivanje delova sadržaja datoteka potrebni baferi, iz modula za rukovanje datotekama se poziva i operacija zauzimanja, radi rezervisanja dovoljno velikog baferskog prostora.

Modul za rukovanje datotekama ostvaruje svoj zadatak tako što uvodi **operacije otvaranja, zatvaranja, čitanja i pisanja**. Pozivi poslednje dve operacije dovode do prenosa sadržaja datoteka na relaciji između radne i masovne memorije.

1.4.5 Modul za rukovanje procesima

Zadatak modula za rukovanje procesima je da omogući stvaranje i uništavanje procesa, kao i stvaranje i uništavanje njihovih niti. Na taj način postaje moguće da istovremeno postoji više procesa (višeprocesni režim rada, *multiprocessing*) i više niti (*multithreading*). To je preduslov: (1) za bolje iskorišćenje procesora, (2) za istovremenu podršku većeg broja korisnika (višekorisnički režim rada, *multiuser environment*) i (3) za bržu reakciju računara na vanjske događaje. Iz modula za rukovanje procesima se

poziva operacija čitanja, radi preuzimanja sadržaja izvršnih datoteka, koji su potrebni za svaranje slike procesa. Pošto je za stvaranje slike procesa potrebna radna memorija, iz modula za rukovanje procesima se pozivaju i operacije zauzimanja, odnosno oslobađanja.

Modul za rukovanje procesima ostvaruje svoj zadatak tako što uvodi **operacije stvaranja i uništavanja**. Pozivi ovih operacija dovode do stvaranja i uništavanja procesa, odnosno niti.

1.4.6 Slojeviti operativni sistem

Prethodno opisani moduli operativnog sistema formiraju hijerarhiju, sastavljenu od 5 slojeva. Svaki od slojeva je predodređen da sadrži jedan od modula operativnog sistema. Raspodelu modula po slojevima diktira pravilo koje nalaže da se iz svakog sloja pozivaju samo operacije uvedene u nižim slojevima hijerarhije. Primena pomenutog pravila dovodi do smeštanja modula za rukovanje procesima u sloj na vrhu hijerarhije. U prvi niži sloj dospeva modul za rukovanje datotekama, dok je sledeći niži sloj namenjen modulu za rukovanje radnom memorijom. Pretposlednji sloj hijerarhije sadrži modul za rukovanje kontrolerima, a u poslednjem sloju nalazi se modul za rukovanje procesorom. Na ovaj način je obrazovan **slojeviti operativni sistem** (Slika 1.2).

modul za rukovanje procesima
modul za rukovanje datotekama
modul za rukovanje radnom memorijom
modul za rukovanje kontrolerima
modul za rukovanje procesorom

Slika 1.2: Slojeviti operativni sistem

Svrha predstavljanja operativnog sistema kao hijerarhije slojeva je motivisana željom da se zadatak operativnog sistema raščlani na više jednostavnijih međusobno nezavisnih zadataka i zatim svaki od njih objasni zasebno. Iako je raslojavanje operativnog sistema u hijerarhiju slojeva moguće uspešno primeniti i prilikom pravljenja operativnog sistema, u praksi se uglavnom sreću **monolitni** (*monolithic*) operativni sistemi. Monolitni operativni sistemi nemaju hijerarhijsku strukturu kao slojeviti operativni sistemi, jer se sastoje od modula čija saradnja nije ograničena pravilima kao kod slojevitih operativnih sistema. To znači da se iz svakog od modula monolitnih operativnih sistema mogu slobodno pozivati operacije svih ostalih modula.

1.4.7 Sistemski pozivi

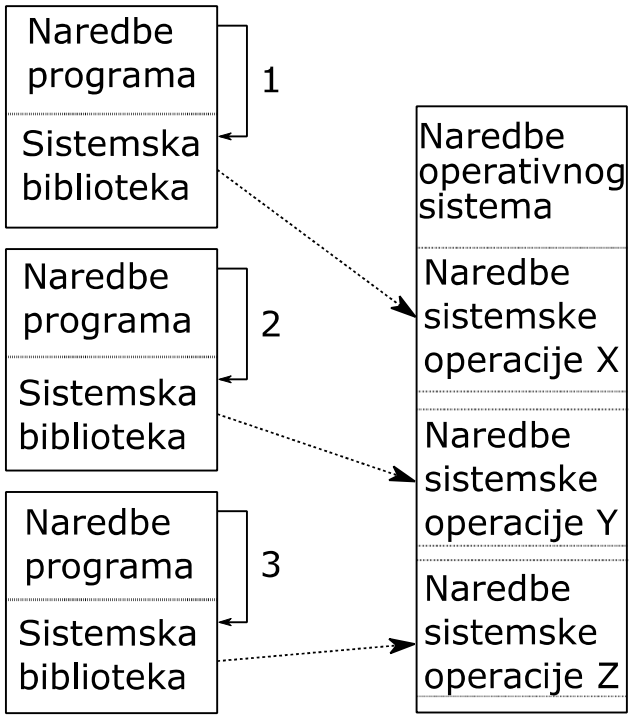
Slojeviti operativni sistem dozvoljava pozivanje (odnosno korišćenje) operacija stvaranja i uništavanja (procesa) samo iz sloja, smeštenog iznad hijerarhije njegovih slojeva. Prema tome, postojanje procesa je isključivo vezano za ovaj, korisnički sloj.

Iako su svi procesi locirani u korisničkom sloju, oni su međusobno jasno razdvojeni

zahvaljujući činjenici da svaki od procesa poseduje zaseban adresni prostor, koji se naziva **korisnički prostor** (*user space*). Na istom principu se zasniva i razdvajanje procesa od operativnog sistema, jer operativni sistem poseduje zaseban adresni prostor koji se naziva **sistemski prostor** (*kernel space*). Ali, razdvojenost korisničkih prostora od sistemskog prostora, sprečava da se pozivi operacija operativnog sistema zasnivaju na korišćenju poziva potprograma. Zato je neophodno uvođenje posebnog mehanizma **sistemskih poziva**, koji omogućuje prelazak iz korisničkog prostora u sistemski prostor, radi poziva operacija operativnog sistema. Sistemski pozivi zahtevaju korišćenje posebnih asemblerskih naredbi i zbog toga se sakrivaju unutar **sistemskih potprograma**. Neki od sistemskih potprograma, pored sistemskih poziva, sadrže i specifične obrade podataka, kao što je, na primer, pretvaranje brojeva iz znakovnog u binarni oblik i obrnuto kod formatiranog ulaza ili izlaza. Svaki od sistemskih potprograma je namenjen za pozivanje jedne od operacija operativnog sistema, namenjenih korisničkom sloju. Ovakve operacije se nazivaju **sistemske operacije**, da bi se razlikovale od ostalih (internih) operacija operativnog sistema, namenjenih samo za korišćenje unutar operativnog sistema. Za izvršavanje sistemskih operacija je potreban stek. Zato u sistemskom prostoru za svaki proces postoji poseban **sistemski stek**, namenjen izvršavanju sistemskih operacija koje, posredstvom sistemskih poziva, pozivaju pojedini procesi.

Sistemski potprogrami obrazuju **sistemsku biblioteku**. Znači pozivanje sistemskih operacija se svodi na pozivanje potprograma sistemske biblioteke. Sistemska biblioteka se isporučuje uz operativni sistem da bi se koristila u postupku linkovanja. U toku linkovanja sistemski potprogrami se vezuju za objektni oblik korisničkog programa, radi stvaranja izvršnog oblika korisničkog programa, koji tako postaje spreman za saradnju sa operativnim sistemom.

Zahvaljujući sistemskim potprogramima, odnosno sistemskoj biblioteci, operativni sistem predstavlja deo korisničkog programa, iako za njega nije direktno linkovan. Zato se može smatrati da obavljanje sistemskih operacija predstavlja sastavni deo izvršavanja korisničkog programa, odnosno da pripada aktivnosti procesa, vezanog za ovo izvršavanje. To, uz istovremeno postojanje više procesa i nepredvidivost preključivanja, uzrokuje da je moguće da istovremeno postoji više procesa, koji su započeli, a nisu završili svoju aktivnost u okviru operativnog sistema, odnosno, čija aktivnost je zaustavljena unutar sistemskih operacija operativnog sistema (Slika 1.3: pune linije sa strelicom označavaju poziv sistemskog potprograma, a isprekidane linije sa strelicom označavaju nezavršeni sistemski poziv). Iz prethodnog sledi da je operativni sistem konkurentni program, jer naizmenične aktivnosti raznih, istovremeno postojećih procesa u okviru sistemskih operacija operativnog sistema predstavljaju razne niti operativnog sistema.



Slika 1.3: Prikaz preplitanja izvršavanja tri sistemske operacije

1.4.8 Interakcija korisnika i operativnog sistema

Dok sistemska biblioteka omogućuje korišćenje operativnog sistema na **programskom nivou**, komande komandnog jezika omogućuju korišćenje operativnog sistema na **interaktivnom nivou**. Komande komandnog jezika precizno opisuju rukovanja procesima i datotekama. One tako lišavaju korisnika potrebe da poznaje mehanizme, koji se pokreću unutar operativnog sistema radi rukovanja procesima i datotekama. Za preuzimanje i interpretiranje komandi komandnog jezika zadužen je poseban proces iz korisničkog sloja, koji se naziva **interpreter komandnog jezika** (*shell*). Interpreter komandnog jezika posreduje između korisnika i operativnog sistema. U opštem slučaju u posredovanju obično učestvuju i procesi koje stvara interpreter komandnog jezika, da bi im prepustio izvršavanje pojedinih od prepoznatih komandi.

Interpreter komandnog jezika koristi operativni sistem na programskom nivou, jer u toku svog rada poziva sistemske operacije.

1.5 PITANJA

1. Koje poslove obavlja operativni sistem?
2. Šta obuhvata pojam datoteke?
3. Šta se nalazi u deskriptoru datoteke?
4. Šta omogućuju datoteke?
5. Šta obavezno prethodi čitanju i pisanju datoteke?

6. Šta sledi iza čitanja i pisanja datoteke?
7. Šta obuhvata pojam procesa?
8. Šta se nalazi u deskriptoru procesa?
9. Koja stanja procesa postoje?
10. Kada je proces aktivan?
11. Šta je kvantum?
12. Šta se dešava nakon isticanja kvantuma?
13. Po kom kriteriju se uvek bira aktivan proces?
14. Koji prelazi su mogući između stanja procesa?
15. Koji prelazi nisu mogući između stanja procesa?
16. Šta omogućuju procesi?
17. Šta karakteriše sekvencijalni proces?
18. Šta karakteriše konkurentni proces?
19. Šta ima svaka nit konkurentnog procesa?
20. Koju operaciju uvodi modul za rukovanje procesorom?
21. Po čemu se razlikuju preključivanja između niti istog procesa i preključivanja između niti raznih procesa?
22. Koje operacije uvodi modul za rukovanje kontrolerima?
23. Šta karakteriše drajvere?
24. Koje operacije uvodi modul za rukovanje radnom memorijom?
25. Koje operacije poziva modul za rukovanje radnom memorijom kada podržava virtuelnu memoriju?
26. Koje operacije uvodi modul za rukovanje datotekama?
27. Koje operacije poziva modul za rukovanje datotekama?
28. Šta omogućuju *multiprocesing* i *multithreading*?
29. Koje operacije uvodi modul za rukovanje procesima?
30. Koje operacije poziva modul za rukovanje procesima?
31. Koje module sadrži slojeviti operativni sistem?
32. Šta omogućuju sistemski pozivi?
33. Koje adresne prostore podržava operativni sistem?
34. Šta karakteriše interpreter komandnog jezika?
35. Koji nivoi korišćenja operativnog sistema postoje?

2 KONKURENTNO PROGRAMIRANJE

2.1 SVOJSTVA KONKURENTNIH PROGRAMA

Za izvršavanje konkurentnih programa na jednoprocesorskim računarima je tipično da procesor izvršava naredbe jedne niti dok je moguća njena aktivnost ili dok se ne desi prekid. Kada se desi prekid, procesor izvrši naredbe odgovarajućeg obrađivača prekida. Ako obrada prekida izazove preključivanje, u nastavku svog rada procesor izvrši naredbe potprograma preključivanja i zatim produži sa izvršavanjem naredbi druge niti. Ovakvo mešanje izvršavanja naredbi raznih niti, odnosno niti i obrađivača prekida se naziva **preplitanje** (*interleaving*). Preplitanje niti, odnosno preplitanje niti i obrada prekida imaju slučajan karakter, jer unapred nije poznato posle izvršavanja koje naredbe će se desiti prekid i eventualno preključivanje. To nije moguće odrediti čak ni za prekide pravilnog perioda, kao što su prekidi sata, jer izvršavanje konkurentnog programa može započeti u bilo kom trenutku između dva prekida sata. Prema tome, broj izvršenih naredbi konkurentnog programa pre prvog prekida sata varira od izvršavanja do izvršavanja, pa se i preplitanja niti, odnosno preplitanja niti i obrada prekida razlikuju od izvršavanja do izvršavanja.

Slučajna priroda preplitanja tera na razmišljanje o štetnom uticaju preplitanja na izvršavanje konkurentnih programa. Pod uticajem preplitanja rezultati izvršavanja konkurentnih programa mogu da budu stohastični, odnosno mogu da se menjaju od izvršavanja do izvršavanja. Štetan uticaj preplitanja na rezultate izvršavanja konkurentnih programa se može ilustrovati na primerima. U njima se koriste **klase** (*class*) programskog jezika C++, čije **operacije** (*function members*) opisuju rukovanje objektima ovih klasa. Umesto pojma **objekt** (*object - a region of memory containing class data members*) koristi se kao sinonim i pojam **promenljiva**.

2.2 PRIMERI ŠTETNIH PREPLITANJA

Primeri štetnih preplitanja niti se mogu pronaći u okviru operativnog sistema, jer on predstavlja tipičan konkurentni program. Operativni sistem ima osobine konkurentnog programa, jer (1) sistemskim pozivima, pokrenutim za račun procesa iz korisničkog sloja, odgovaraju niti operativnog sistema i jer (2) operativni sistem sadrži obrađivače prekida. Znači, u toku aktivnosti operativnog sistema moguća su međusobna preplitanja niti, ali i preplitanja niti i obrada prekida.

2.2.1 Rukovanje pozicijom kursora

Štetna preplitanja niti i obrada prekida u okviru operativnog sistema mogu da se ilustruju na primeru rukovanja pozicijom kursora. Poziciju kursora karakterišu dve koordinate (**x** i **y**), a rukovanje ovom pozicijom obuhvata promenu pozicije: **set()** i preuzimanje pozicije: **get()**. Rukovanje pozicijom kursora opisuje klasa **Position** (Listing 2.1).

Listing 2.1: Klasa **Position**

```
class Position {
   int x, y;
  public:
   Position();
   void set(int new_x, int new_y);
   void get(int* current_x, int* current_y);
};

Position::Position()
{
   x = 0;
   y = 0;
}

void
Position::set(int new_x, int new_y)
{
   x = new_x;
   y = new_y;
}

void
Position::get(int* current_x, int* current_y)
{
   *current_x = x;
   *current_y = y;
}
```

Neka u operativnom sistemu postoji objekat klase **Position**:

```
Position
position;
```

i neka operacija **position.set()** bude na raspolaganju samo obrađivaču prekida koji u okviru operativnog sistema registruje izmene pozicije kursora. Neka, zatim, operacija **position.get()** bude na raspolaganju procesima iz korisničkog sloja. Tada je moguće da u toku izvršavanja operacije **position.get()** proces bude prekinut radi obrade prekida, koja poziva operaciju **position.set()**. Ako se to desi nakon izvršavanja prvog iskaza dodele iz tela operacije **position.get()**, a pre izvršavanja drugog iskaza dodele iz njenog tela, tada će rezultat izvršavanja ove operacije biti pogrešan. Do greške dolazi, jer nakon

preuzimanja apscise, a pre preuzimanja ordinate, obrada prekida izmeni apscisu i ordinatu. Tako su preuzete stara apscisa i nova ordinata, pa je dobijena pozicija u kojoj se kursor ne nalazi. Znači, opisano preplitanje niti i obrade prekida, odnosno preplitanje izvršavanja operacija **position.get()** i **position.set()** je štetno.

2.2.2 Rukovanje slobodnim baferima

Štetna međusobna preplitanja niti raznih procesa u okviru operativnog sistema mogu da se ilustruju na primeru rukovanja slobodnim baferima. Rukovanje slobodnim baferima je locirano u modulu za rukovanje datotekama i zasniva se na pretpostavci da su slobodni baferi organizovani u listu. U pojednostavljenom slučaju, rukovanje listom bafera obuhvata uvezivanje: **link()** slobodnog bafera u listu i izvezivanje: **unlink()** slobodnog bafera iz liste. Rukovanje listom bafera opisuje klasa **List** (Listing 2.2).

Listing 2.2: Klasa **List**

```
struct List_member {
   List_member* next;
   char buffer[512];
};

class List {
   List_member* first;
  public:
   List() : first(0) {};
   void link(List_member* member);
   List_member* unlink();
};

void
List::link(List_member* member)
{
   member->next=first;
   first=member;
}
```

```
List_member*
List::unlink()
{
   List_member* unlinked;
   unlinked=first;
   if(first != 0)
      first=first->next;
   return unlinked;
}
```

Podrazumeva se da u operativnom sistemu postoji više primeraka strukture **List_member** koji su inicijalno uvezani u listu obrazovanu oko primerka klase **List**:

```
List
list;
```

Neka su operacije **list.link()** i **list.unlink()** na raspolaganju samo modulu za rukovanje datotekama i neka se pozivaju iz operacija ovog modula. Tada je moguće da izvršavanje operacije **list.unlink()** bude pokrenuto u toku aktivnosti niti nekog procesa u modulu za rukovanje datotekama. Kao rezultat izvršavanja ove operacije na steku pomenute niti nastane primerak njene lokalne promenljive **unlinked**. Izvršavanje iskaza:

```
unlinked = first;
```

smešta u ovaj primerak lokalne promenljive adresu prvog slobodnog bafera iz liste bafera, jasno, kada takav bafer postoji. Neka, nakon izvršavanja prethodnog iskaza, pod uticajem obrade prekida dođe do preključivanja procesora na nit drugog procesa. Tada, u toku aktivnosti niti ovog procesa, može doći do pokretanja još jednog izvršavanja operacije **list.unlink()**. Tako na steku i ove druge niti nastaje njen primerak lokalne promenljive **unlinked**. I u taj primerak dospeva adresa istog prvog slobodnog bafera iz liste bafera, jasno, kada takav bafer postoji. Posledica ovakvog sleda događaja je da posmatrana dva procesa, bez obzira na nastavak njihovih aktivnosti, nezavisno jedan od drugog koriste isti bafer. To neminovno dovodi do fatalnog ishoda. Prema tome, opisano međusobno preplitanje niti dva razna procesa, odnosno preplitanje dva izvršavanja operacije **list.unlink()**, je štetno. Isto važi i za preplitanje izvršavanja operacija **list.link()** i **list.unlink()**, kao i za preplitanja dva izvršavanja operacije **list.link()**.

2.2.3 Rukovanje komunikacionim baferom

Štetna međusobna preplitanja niti su moguća i za niti koje pripadaju istom procesu. Na primer, često je potrebno ostvariti međusobnu saradnju niti istog procesa. Ovakva saradnja se može ostvariti tako što jedna od niti šalje podatke drugoj niti. Takva razmena podataka između niti se obično obavlja posredstvom komunikacionog bafera. Nit koja

puni ovaj bafer podacima ima ulogu **proizvođača** (podataka), a nit koja prazni ovaj bafer ima ulogu **potrošača** (podataka). U pojednostavljenom slučaju, rukovanje komunikacionim baferom obuhvata punjenje: **put()** celog bafera, kao i pražnjenje: **get()** celog bafera. Rukovanje komunikacionim baferom opisuje klasa **Buffer** (Listing 2.3).

Listing 2.3: Klasa **Buffer**

```
const unsigned int
BUFFER_SIZE = 512;

class Buffer {
   char content[BUFFER_SIZE];
  public:
   Buffer() {};
   void put(char* c);
   void get(char* c);
};

void
Buffer::put(char* c)
{
   unsigned int i;
   for(i = 0; i < BUFFER_SIZE; i++)
      content[i] = *c++;
}

void
Buffer::get(char* c)
{
   unsigned int i;
   for(i = 0; i < BUFFER_SIZE; i++)
      *c++ = content[i];
}
```

Ako u konkurentnom programu postoji objekat klase **Buffer**:

```
Buffer
buffer;
```

tada operaciju **buffer.put()** poziva nit proizvođač (procesa koji odgovara izvršavanju posmatranog konkurentnog programa). Slično, operaciju **buffer.get()** poziva nit potrošač (istog procesa). U ovoj situaciji moguće je, na primer, da se desi prekid sata za vreme

aktivnosti proizvođača u operaciji **buffer.put()**. Obrada ovog prekida može izazvati preključivanje procesora na potrošač. Ako operacija **buffer.get()** bude pozvana u toku aktivnosti potrošača, tada postoji mogućnost da potrošač preuzme sadržaj delimično popunjenog bafera. Takođe su moguće situacije u kojima proizvođač višestruko puni bafer koga potrošač još nije ispraznio, odnosno u kojima potrošač višestruko preuzima isti sadržaj iz bafera. Prema tome, opisano preplitanje niti istog procesa, odnosno preplitanje izvršavanja operacija **buffer.put()** i **buffer.get()**, je štetno.

2.3 SPREČAVANJE ŠTETNIH PREPLITANJA

Konkurentni programi se zasnivaju na ideji preplitanja, ali njihova upotrebljivost zavisi od sprečavanja štetnih preplitanja. U primerima rukovanja pozicijom kursora, slobodnim baferima i komunikacionim baferom, štetna preplitanja su se javila u toku pristupanja promenljivim **position**, **list** i **buffer**. Pošto promenljivoj **position** pristupaju niti i obrade prekida, može se reći da niti i obrade prekida međusobno dele ovu promenljivu. Slično, promenljivim **list** i **buffer** pristupaju razne niti, pa se može reći da one međusobno dele ove promenljive. Zato se ovakve promenljive nazivaju **deljene** (*shared*) **promenljive**, a klase, koje opisuju rukovanje deljenim promenljivim, se nazivaju **deljene klase**.

2.3.1 Međusobna isključivost

Prethodno pomenute deljene klase su napravljene pod pretpostavkom da se rukovanja deljenim promenljivim obavljaju sekvencijalno, odnosno da se operacije deljenih promenljivih izvršavaju strogo jedna za drugom. To znači da novo izvršavanje bilo koje od operacija deljene promenljive može početi tek nakon završetka prethodno započetog izvršavanja neke od ovih operacija. Na taj način svako od ovih izvršavanja ostavlja i zatiče deljene promenljive u **konzistentnom** (predviđenom) stanju. Međutim, štetna preplitanja negiraju pomenutu pretpostavku, jer dopuštaju da novo izvršavanje neke operacije deljene promenljive započne pre nego što se završilo već započeto izvršavanje neke od operacija iste deljene promenljive. Na taj način, novo izvršavanje zatiče deljenu promenljivu u delimično izmenjenom, znači potencijalno nekonzistentnom stanju, što nije moguće kod sekvencijalnog izvršavanja operacija deljene promenljive. Problem štetnih preplitanja ne postoji, ako se obezbedi **međusobna isključivost** (*mutual exclusion*) izvršavanja operacija deljenih promenljivih. Međusobna isključivost izvršavanja operacija deljenih promenljivih garantuje serijalizaciju rukovanja deljenim promenljivim i na taj način sprečava štetna preplitanja.

2.3.2 Kritične sekcije

Prethodni primeri ukazuju da su štetna preplitanja vezana za pristupanje deljenim promenljivim. Kada nema pristupanja deljenim promenljivim, odnosno, kada se izvršavaju međusobno nezavisni delovi konkurentnog programa, tada nema mogućnosti za ugrožavanje konzistentnosti deljenih promenljivih, pa nema ni štetnih preplitanja. Tela operacija deljenih klasa ili delovi ovih tela, čije izvršavanje je kritično za konzistentnost deljenih promenljivih, se nazivaju **kritične sekcije** (*critical section*).

2.3.3 Sinhronizacija

Međusobna isključivost kritičnih sekcija se ostvaruje vremenskim usklađivanjem njihovih izvršavanja. Sprovođenje ovakvog usklađivanja se naziva **sinhronizacija** (*synchronization*). Sinhronizacija, zadužena za ostvarenje međusobne isključivosti, nije jedina vrsta sinhronizacije. Na primer, kod rukovanja komunikacionim baferom potrebno je osigurati naizmenično izvršavanje operacija **buffer.put()** i **buffer.get()**, da bi se sprečilo punjenje neispražnjenog komunikacionog bafera, odnosno da bi se sprečilo pražnjenje nenapunjenog komunikacionog bafera. Vrsta sinhronizacije, koja ostvaruje ovakav dodatni uslov u pogledu izvršavanja kritičnih sekcija, se naziva **uslovna sinhronizacija** (*condition synchronization*).

2.3.4 Atomski regioni

U primeru rukovanja pozicijom kursora, štetna preplitanja nastupaju kao posledica obrada prekida. Ako se onemoguće prekidi u kritičnim sekcijama odgovarajuće deljene promenljive, tada u toku izvršavanja ovih kritičnih sekcija nisu moguće ni obrade prekida, pa ni pomenuta štetna preplitanja. Zbog neprekidnosti izvršavanja (nedeljivosti), ovakvim kritičnim sekcijama pristaje ime **atomski regioni**. Neprekidnost atomskih regiona garantuje njihovu međusobnu isključivost. Ali, pošto onemogućenje prekida u atomskim regionima odlaže obradu novih prekida i usporava reakciju procesora na vanjske događaje, važno je da izvršavanja atomskih regiona budu što kraća. Ovakav zahtev sužava primenljivost onemogućenja prekida kao sredstva za osiguranje međusobne isključivosti kritičnih sekcija.

2.3.5 Propusnice i isključivi regioni

U primeru rukovanja slobodnim baferima, štetna preplitanja nastupaju kao direktna posledica preključivanja, dok su obrade prekida, kao inicijatori ovih preključivanja, samo posredan uzrok štetnih preplitanja. Zato, u ovom slučaju, ostvarenje međusobne isključivosti kritičnih sekcija nema potrebe zasnivati na onemogućenju prekida, nego na zaustavljanju aktivnosti niti, kada ona pokuša da uđe u kritičnu sekciju deljene promenljive kojoj već pristupa neka druga nit. Ovakav način ostvarenja međusobne isključivosti kritičnih sekcija podrazumeva da svaka deljena promenljiva poseduje jednu **propusnicu** za ulazak u njene kritične sekcije. Propusnica može biti **slobodna** (nedodeljena) ili **zauzeta** (dodeljena). Podrazumeva se da bez propusnice nije moguć ulazak u kritičnu sekciju deljene promenljive. Zato propusnicu traže sve niti koje se takmiče za ulazak u neku od kritičnih sekcija iste deljene promenljive. Međutim, propusnicu dobija samo jedna od njih i ona ulazi u kritičnu sekciju. Ostale niti zaustavljaju svoju aktivnost i prelaze u stanje "čeka". Svaka od njih ostaje u tom stanju do eventualnog dobijanja tražene propusnice. To se desi tek pošto nit, koja pristupa deljenoj promenljivoj, napusti njenu kritičnu sekciju i vrati propusnicu deljene promenljive. Nit, koja tada dobije propusnicu, odmah prelazi iz stanja "čeka" u stanje "spremna", ali u kritičnu sekciju ulazi tek kada postane aktivna (odnosno, kada se procesor preključi na nju).

Rukovanje propusnicom deljene promenljive je, takođe, ugroženo štetnim

preplitanjima. Zbog toga se konzistentnost propusnica mora zaštititi. Za to se obično koriste atomski regioni, jer su rukovanja propusnicama kratkotrajna.

Po načinu ostvarenja međusobne isključivosti, atomski regioni se razlikuju od onih kritičnih sekcija koje međusobnu isključivost ostvaruju korišćenjem propusnica. Zato ove druge sekcije treba drugačije nazvati, na primer, **isključivi regioni**.

Interesantno je uočiti da preključivanja u isključivim regionima omogućuju pojavu štetnih preplitanja, ali i omogućuju njihovo sprečavanje. Tako, ako nit, čiju aktivnost je omogućilo preključivanje, pokuša da uđe u isključivi region deljene promenljive sa zauzetom propusnicom, tada opet preključivanje dovodi do zaustavljanja aktivnosti ove niti. Prema tome, prvo preključivanje je stvorilo uslove za pojavu štetnog preplitanja, a drugo preključivanje je sprečilo tu pojavu.

Uslovna sinhronizacija se, takođe, zasniva na zaustavljanju aktivnosti niti, dok se ne ispuni uslov koji je neophodan za nastavak pomenute aktivnosti.

2.4 POŽELJNE OSOBINE KONKURENTNIH PROGRAMA

Za konkurentne programe je bitno da su sva njihova izvršavanja bez štetnih preplitanja. Zbirna tvrdnja da štetna preplitanja nisu moguća u bilo kom od izvršavanja konkurentnog programa se može raščlaniti na pojedinačne tvrdnje **isključivanja nepoželjnog** (*safety property*) i na pojedinačne tvrdnje **uključivanja poželjnog** (*liveness property*). Primer tvrdnje isključivanja nepoželjnog je tvrdnja da se u izvršavanjima konkurentnog programa ne javlja nekonzistentnost date deljene promenljive. Primer tvrdnje uključivanja poželjnog je tvrdnja da se u toku izvršavanja konkurentnog programa dese svi zatraženi ulasci u dati isključivi region. Važenje pojedinačnih tvrdnji znači da konkurentni program poseduje odgovarajuće poželjne osobine. Za konkurentni program se može reći da ima neku od poželjnih osobina samo ako se dokaže (na neformalan ili formalan način) da važi tvrdnja kojoj pomenuta osobina odgovara. Ovakvo rezonovanje o ispravnosti konkurentnog programa je neophodno, jer slučajna priroda preplitanja može da bude uzrok **nedeterminističkog** (nepredvidivog) **izvršavanja konkurentnog programa**. U takvoj situaciji, ponovljena izvršavanja konkurentnog programa, u toku kojih se obrađuju isti podaci, ne moraju da imaju isti ishod. Zbog toga, provera ispravnosti konkurentnog programa ne može da se zasniva samo na pokazivanju da pojedina izvršavanja konkurentnog programa imaju ispravan rezultat, jer tačan rezultat, dobijen u jednom ili više izvršavanja, ne isključuje mogućnost postojanja izvršavanja koja za iste ulazne podatke daju netačan rezultat.

2.5 PROGRAMSKI JEZICI ZA KONKURENTNO PROGRAMIRANJE

Konkurentno programiranje se razlikuje od sekvencijalnog po rukovanju nitima i deljenim promenljivima. Opisivanje ovakvih rukovanja se može zasnovati na korišćenju posebnih konkurentnih iskaza konkurentnog programskog jezika. Konkurentni programski jezik može nastati kao rezultat pravljenja potpuno novog programskog jezika ili kao rezultat proširenja postojećeg sekvencijalnog programskog jezika konkurentnim iskazima. U oba pristupa neizbežne su aktivnosti vezane za definisanje sintakse i semantike programskog jezika, kao i aktivnosti vezane za zahvate na kompajleru.

Ovakve aktivnosti se izbegavaju, ako se konkurentno programiranje zasniva na korišćenju konkurentne biblioteke. Njena dodatna prednost je što omogućuje da se za konkurentno programiranje koristi već postojeći, znači poznat programski jezik. Prema tome, posmatrano sa praktičnog stanovišta, opravdano je konkurentno programiranje osloniti na konkurentnu biblioteku. Ako je konkurentna biblioteka namenjena za objektno orijentisani programski jezik, tada ona može da sadrži definicije klasa koje opisuju rukovanje nitima i deljenim promenljivima. U tom slučaju, korišćenje ovih klasa predstavlja prirodan način za primenu baznih koncepata konkurentnog programiranja.

2.6 UVOD U KONKURENTNU BIBLIOTEKU KOJA IMPLEMENTIRA DEO MEĐUNARODNOG STANDARDA C++11

Međunarodni standard C++11 predviđa rukovanje nitima i deljenim promenljivim. U ovoj knjizi se izlaže samo podskup načina za rukovanje nitima i deljenim promenljivim koje predviđa međunarodni standard C++11. Za označavanje pomenutog podskupa u ovoj knjizi se koristi skraćnica **CppTss** (*C plus plus Thread subset*).

Rukovanje nitima omogućuje klasa **thread**. Njen konstruktor je zadužen za kreiranje (stvaranje i pokretanje) niti. Kao argument poziva ovog konstruktora navodi se adresa funkcije koja opisuje nit (njenu aktivnost). Tako funkcija:

```
void
thread_example()
{
   double pi;
   cout << "ZADAJ VREDNOST BROJA PI" << endl;
   cin >> pi;
   cout << endl << "PI = " << pi << endl;
}
```

opisuje nit u toku čije aktivnosti se zatraži zadavanje vrednosti broja PI, preuzme se i prikaže ta vrednost (prikazivanje konstante **endl** dovodi do pomeranja kursora na početak sledeće linije ekrana). Predviđeno je da kraj aktivnosti ovakve niti nastupi kada se izvrši telo funkcije koja opisuje dotičnu nit. Kreiranje ovakve niti je prikazano u sledećem primeru:

```
int
main()
{
   thread example(thread_example);
}
```

Nakon kreiranja niti neophodno je odrediti njen odnos prema funkciji **main()**. Kraj izvršavanja ove funkcije (odnosno, kraj aktivnosti njoj korespondentne niti) označava

kraj aktivnosti njoj odgovarajućeg procesa, a time i svih niti ovog procesa. Da ne bi došlo do prevremenog kraja aktivnosti kreirane niti, klasa **thread** nudi operaciju **join()**. Ona zaustavlja aktivnost svog pozivaoca sve dok se ne završi aktivnost niti na koju se odnosi ova operacija. Tako primer:

```
int
main()
{
   thread example(thread_example);
   example.join();
}
```

prikazuje kreiranje jedne niti i obezbeđuje da nakon završetka njene aktivnosti nastupi kraj izvršavanja funkcije **main()** (odnosno, kraj aktivnosti njoj korespondentne niti).

Klasa **thread** nudi i operaciju **detach()** pomoću koje se saopštava da je dozvoljen nasilni (prevremeni) kraj aktivnosti niti (da regularan kraj aktivnosti niti, na koju se odnosi ova operacija, može da nastupi i kao posledica kraja aktivnosti njenog procesa).

Kada se kreira više niti, moguća je pojava štetnog preplitanja. Primer:

```
int
main()
{
   thread example1(thread_example);
   thread example2(thread_example);
   example1.join();
   example2.join();
}
```

prikazuje kreiranje dve niti (podrazumeva se da sve niti procesa imaju isti prioritet). U toku njihovih aktivnosti moguća su štetna preplitanja njihovih izlaznih i ulaznih operacija, jer nema garancije da će u razmatranom primeru biti izvršene sve izlazne i ulazne operacije prvokreirane niti, pa tek onda sve izlazne i ulazne operacije drugokreirane niti. Da bi se sprečilo mešanje izlaznih i ulaznih operacije raznih niti, svaka od njih treba da zauzme terminal pre obavljanja svojih izlaznih i ulaznih operacija, a nakon njihovog obavljanja da oslobodi terminal. Pošto pomenuto korišćenje terminala predstavlja kritičnu sekciju, međusobna isključivost ovih kritičnih sekcija se može zasnovati na upotrebi propusnice koja reprezentuje terminal. To omogućuje klasa **mutex**. Njen objekat predstavlja propusnicu. Operacija **lock()** ove klase omogućuje zauzimanje propusnice, a operacija **unlock()** ove klase omogućuje oslobađanje propusnice. Na ovaj način su u primeru:

```
mutex terminal;

void
thread_example()
{
   double pi;
   terminal.lock();
   cout << "ZADAJ VREDNOST BROJA PI" << endl;
   cin >> pi;
   cout << endl << "PI = " << pi << endl;
   terminal.unlock();
}
```

sprečena štetna preplitanja prilikom obavljanja izlaznih i ulaznih operacija raznih niti.

Zasnivanje međusobne isključivosti na klasi **mutex** podrazumeva pozivanje njene operacije **lock()** na početku kritične sekcije i pozivanje njene operacije **unlock()** na kraju kritične sekcije. Isti efekat se može ostvariti, ako se koristi templejt klasa **unique_lock**, jer njen konstruktor poziva operaciju **lock()**, a destruktor operaciju **unlock()**. Da bi se to postiglo, kao argument templejt klase **unique_lock** mora se navesti klasa **mutex**, a kao argument konstruktora templejt klase **unique_lock** mora se navesti objekat klase **mutex**. Ostvarenje međusobne isključivosti izlaznih i ulaznih operacija raznih niti primenom klase **unique_lock** ilustruje primer:

```
mutex terminal;

void
thread_example()
{
   double pi;
   unique_lock<mutex> lock(terminal);
   cout << "ZADAJ VREDNOST BROJA PI" << endl;
   cin >> pi;
   cout << endl << "PI = " << pi << endl;
}
```

U prethodnom primeru stvaranje objekta klase **unique_lock** označava početak isključivog regiona, koji se završava na kraju složenog iskaza u kome je stvoren pomenuti objekat.

Klasa koja kao svoje polje sadrži propusnicu (objekat klase **mutex**) predstavlja deljenu klasu. Pomenuta propusnica je potrebna za obrazovanje isključivih regiona u telima operacija deljene klase, radi zaštite konzistentnosti njenih primeraka, odnosno,

radi zaštite konzistentnosti deljenih promenljivih. Deljena klasa, čiju konzistentnost štite isključivi regioni, se može nazvati **isključiva klasa**.

Ulazak u isključivi region nije moguć, ako je propusnica zauzeta. U tom slučaju aktivnost niti se zaustavlja. Isto treba da se desi i tokom aktivnosti niti u isključivom regionu, ako se ustanovi da traženi uslov, neophodan za njenu aktivnost, nije ispunjen. Kada taj uslov ispuni druga nit, ona objavi ispunjenje traženog uslova i time omogući nastavak aktivnosti prve niti. Za ostvarenje ovakve uslovne sinhronizacije zadužena je klasa **condition_variable**. Ova klasa nudi operacije **wait()** i **notify_one()**. Podrazumeva se da se ove operacije koriste u isključivom regionu. Operacija **wait()** omogućuje zaustavljanje aktivnosti niti, koja pozove ovu operaciju, dok se ne ispuni dati uslov. Tome prethode prevođenje niti u stanje "čeka", oslobađanje propusnice na koju ukazuje argument poziva ove operacije i preključivanje procesora na drugu spremnu nit. Kao argument poziva ove operacije služi objekat klase **unique_lock** pomoću koga je obrazovan isključivi region u kome se nalazi pomenuti poziv. Operacija **notify_one()** omogućuje objavu ispunjenja datog uslova, radi nastavka aktivnosti jedne od niti koje očekuju ispunjenje dotičnog uslova. Ova nit može da nastavi aktivnost tek kada nit, koja je pozvala operaciju **notify_one()**, oslobodi odgovarajuću propusnicu.

Nakon aktiviranja niti koja je očekivala ispunjenje nekog uslova, uputno je da nit ponovo proveri da li taj uslov važi, jer je, u opštem slučaju, moguće lažno (*spurious*) aktiviranje niti.

2.6.1 Sprečavanje štetnih preplitanja prilikom rukovanja slobodnim baferima

U primeru rukovanja slobodnim baferima, štetna međusobna preplitanja niti mogu da se spreče, ako tela operacija klase **List** obrazuju isključive regione. U tom slučaju je osigurana međusobna isključivost ovih operacija. Izmenjenu definiciju klase **List** sadrži Listing 2.4.

Listing 2.4: Sinhronizovana neblokirajuća klasa **List**

```
struct List_member {
   List_member* next;
   char buffer[512];
};

class List {
   mutex mx;
   List_member* first;
  public:
   List() : first(0) {};
   void link(List_member* member);
   List_member* unlink();
};
```

```
void
List::link(List_member* member)
{
   unique_lock<mutex> lock(mx);
   member->next=first;
   first=member;
}

List_member*
List::unlink()
{
   List_member* unlinked;
   {
      unique_lock<mutex> lock(mx);
      unlinked=first;
      if(first != 0)
         first=first->next;
   }
   return unlinked;
}
```

Operacija **unlink()** spada u neblokirajuće operacije, jer, u situaciji, kada je lista bafera prazna, ona ne zaustavlja aktivnost niti svog pozivaoca, nego vraća vrednost 0. Ova operacija postaje blokirajuća, ako se osloni na uslovnu sinhronizaciju, odnosno, ako iskoristi mogućnosti koje nudi klasa **condition_variable**. U nastavku je navedena verzija klase **List**, sa blokirajućom operacijom **unlink()** (Listing 2.5)

Listing 2.5: Sinhronizovana blokirajuća klasa **List**

```
struct List_member {
   List_member* next;
   char buffer[512];
};
```

```
class List {
   mutex mx;
   List_member* first;
   condition_variable nonempty;
  public:
   List() : first(0) {};
   void link(List_member* member);
   List_member* unlink();
};

void
List::link(List_member* member)
{
   unique_lock<mutex> lock(mx);
   member->next=first;
   first=member;
   nonempty.notify_one();
}

List_member*
List::unlink()
{
   List_member* unlinked;
   {
      unique_lock<mutex> lock(mx);
      while(first == 0)
         nonempty.wait(lock);
      unlinked=first;
      first=first->next;
   }
   return unlinked;
}
```

Operacija **unlink()** zaustavlja aktivnost niti svog pozivaoca, ako je lista bafera prazna. Operacija **link()** nastavlja aktivnost pomenute niti, čim uveže novi bafer u listu bafera.

2.6.2 Sprečavanje štetnih preplitanja prilikom rukovanja komunikacionim baferom

U primeru rukovanja komunikacionim baferom, štetna međusobna preplitanja niti mogu da se spreče, ako se operacije klase **Buffer** oslone na uslovnu sinhronizaciju. U tom slučaju je osigurano naizmenično pristupanje proizvođača i potrošača komunikacionom baferu. Izmenjenu definiciju klase **Buffer** sadrži Listing 2.6.

Listing 2.6: Sinhronizovana klasa **Buffer**

```
const unsigned int
BUFFER_SIZE = 512;

enum
Buffer_states { EMPTY, FULL };

class Buffer {
   mutex mx;
   char content[BUFFER_SIZE];
   Buffer_states state;
   condition_variable full;
   condition_variable empty;
 public:
   Buffer() { state = EMPTY; };
   void put(char* c);
   void get(char* c);
};

void
Buffer::put(char* c)
{
   unsigned i;
   unique_lock<mutex> lock(mx);
   while(state == FULL)
      empty.wait(lock);
   for(i = 0; i < BUFFER_SIZE; i++)
      content[i] = *c++;
   state = FULL;
   full.notify_one();
}
```

```
void
Buffer::get(char* c)
{
   unsigned i;
   unique_lock<mutex> lock(mx);
   while(state == EMPTY)
      full.wait(lock);
   for(i = 0; i < BUFFER_SIZE; i++)
      *c++ = content[i];
   state = EMPTY;
   empty.notify_one();
}
```

U početku je komunikacioni bafer prazan, pa se jedino može izvršiti operacija **put()**. Nakon njenog izvršavanja bafer je pun, pa se može izvršiti samo operacija **get()**. Posle njenog izvršavanja bafer je opet prazan, pa se može izvršiti samo operacija **put()**. Naizmenično izvršavanje operacija klase **Buffer** je osigurano izmenama stanja bafera, zaustavljanjem aktivnosti niti kada bafer nije u potrebnom stanju i omogućavanjem nastavljanja aktivnosti niti kada bafer pređe u potrebno stanje.

2.6.3 Komunikacioni kanal kapaciteta jedne poruke

Saradnja niti proizvođača i niti potrošača, u toku koje prva od njih prosleđuje rezultate svoje aktivnosti drugoj niti, može da se prikaže kao razmena poruka. U toku ove razmene proizvođač odlaže poruku u poseban pregradak iz koga tu poruku preuzima potrošač. Takvu razmenu poruka podržava templejt klasa **Message_box** (Listing 2.7). Njen parametar **MESSAGE** određuje tip poruka koje se razmenjuju. Pošto polje **content** ove klase, koje ima funkciju pregratka, može da prihvati samo jednu poruku, neophodno je da slanja i prijemi poruka budu naizmenični. Samo tako se može garantovati da će sve poslane poruke biti uvek ispravno primljene. Zato templejt klasa **Message_box** sadrži polje **state** koje određuje stanje polja **content**. Pomoću polja **state** se definišu uslovi, od kojih zavise aktivnosti niti pošiljalaca i primalaca poruka. Templejt klasa **Message_box** sadrži i polja **full** i **empty**, koja su potrebna radi uslovne sinhronizacije. Slanje poruke omogućuje operacija **send()**, a prijem poruke omogućuje operacija **receive()**. Prva od njih omogućuje smeštanje poruke u polje **content**, ako je ono prazno. Inače ona zaustavlja aktivnost niti pozivaoca (proizvođača) dok se ovo polje ne isprazni. Druga od njih omogućuje preuzimanje poruke iz polja **content**, ako je ono puno. Inače ona zaustavlja aktivnost niti pozivaoca (potrošača) dok se ovo polje ne napuni.

Listing 2.7: Templejt klasa **Message_box** (datoteka **box.hh**)

```
template<class MESSAGE>
class Message_box {
   mutex mx;
   enum Message_box_states { EMPTY, FULL };
   MESSAGE content;
   Message_box_states state;
   condition_variable full;
   condition_variable empty;
 public:
   Message_box() : state(EMPTY) {};
   void send(const MESSAGE* message);
   MESSAGE receive();
};

template<class MESSAGE>
void
Message_box<MESSAGE>::send(const MESSAGE* message)
{
   unique_lock<mutex> lock(mx);
   while(state == FULL)
      empty.wait(lock);
   content = *message;
   state = FULL;
   full.notify_one();
}

template<class MESSAGE>
MESSAGE
Message_box<MESSAGE>::receive()
{
   unique_lock<mutex> lock(mx);
   while(state == EMPTY)
      full.wait(lock);
   state = EMPTY;
   empty.notify_one();
   return content;
}
```

Templejt klasa **Message_box** omogućuje uspostavljanje komunikacionog kanala između niti pošiljaoca i niti primaoca. Njene operacije **send()** i **receive()** omogućuju

asinhronu razmenu poruka, jer se pošiljalac i primalac na sreću prilikom razmene poruka (aktivnost pošiljaoca se zaustavlja pri slanju poruka samo kada je komunikacioni kanal pun, dok se aktivnost primaoca zaustavlja pri prijemu poruka samo kada je ovaj kanal prazan). Ako se kapacitet komunikacionog kanala poveća na dve ili više poruka, tada svakom prijemu mogu da prethode dva ili više slanja. S druge strane, uz zadržavanje kapaciteta komunikacionog kanala na jednoj poruci, razmena poruka postaje **sinhrona**, ako se uvek zaustavlja aktivnost niti koja prva započne razmenu poruka, bez obzira da li se radi o pošiljaocu ili primaocu. Aktivnost ove niti ostaje zaustavljena dok i druga nit ne započne razmenu poruka (dok se pošiljalac i primalac na sretnu). Pri tome se podrazumeva da pošiljalac nastavlja svoju aktivnost tek kada primalac preuzme poruku. Prethodno dozvoljava da se u komunikacionom kanalu ne čuva poruka, nego njena adresa. To doprinosi brzini sinhrone razmene poruka, jer primalac može direktno preuzeti poruku od pošiljaoca. Time se izbegava potreba da se poruka prepisuje u komunikacioni kanal, što je neizbežno kod asinhrone razmene poruke. Iako na ovaj način primalac pristupa lokalnoj promenljivoj pošiljaoca, u kojoj se nalazi poruka, to ne predstavlja problem dok god mehanizam sinhrone razmene poruka osigurava međusobnu isključivost pristupanja pomenutoj lokalnoj promenljivoj. Pošto sinhrona razmena poruka zahteva da se pošiljalac i primalac poruke sretnu, ona se naziva i **randevu** (*rendezvous*).

Prethodno opisana sinhrona razmena poruka se oslanja na sinhrono slanje i sinhroni prijem poruke. Sinhrono slanje podrazumeva smeštanje adrese poruke u pregradak, koji je inicijalno anuliran, objavljivanje da je moguće preuzimanje poruke i čekanje da poruka bude preuzeta. Sinhroni prijem započinje proverom da li pregradak sadrži adresu poruke. Ako ne sadrži (ako je pregradak anuliran) neizbežno je čekanje da preuzimanje poruke postane moguće. Kada preuzimanje poruke postane moguće, poruka se preuzima, pregradak se anulira i objavljuje se da je poruka preuzeta.

Saradnja niti (procesa), zasnovana na razmeni poruka je primamljiva, jer eliminiše potrebu za eksplicitnim korišćenjem deljenih promenljivih. One su sakrivene u komunikacionom kanalu, a rukovanje ovim kanalom obezbeđuje da pošiljalac može da pristupi poruci samo pre njenog slanja, a primalac samo nakon njenog slanja.

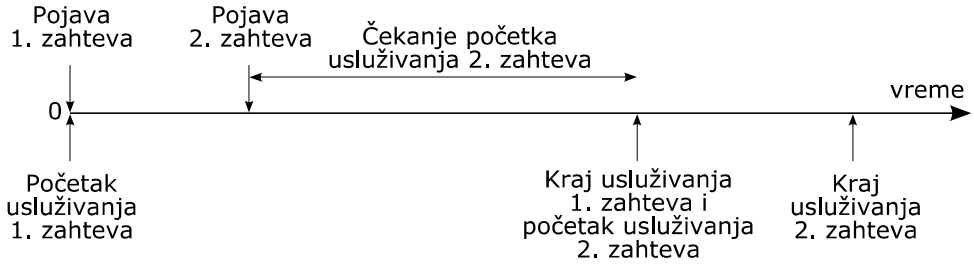
2.6.4 Primer simulacije

Proučavanje sistema podrazumeva izgradnju njihovih modela. Ovakvi modeli mogu da nastanu ako se bitni elementi proučavanih sistema predstave nitima, a međusobni odnosi pomenutih elemenata izraze kao međusobna saradnja ovih niti. Ako su pažljivo napravljeni, ovakvi modeli verno oponašaju proučavane sisteme. Takvo oponašanje sistema se naziva **simulacija**, a modeli koji omogućuju simulaciju sistema se nazivaju **simulacioni modeli**.

Simulacioni modeli su veoma važni za sisteme u čijoj osnovi se nalaze slučajne pojave, jer su simulacije nezamenljiv način proučavanja ponašanja ovakvih sistema u stanjima u koje je proučavane sisteme praktično teško dovesti, odnosno, u kojima je praktično teško pratiti njihovo ponašanje.

Primer sistema, za koje su razvijeni uspešni simulacioni modeli, predstavljaju sistemi

čiji elementi se nalaze u odnosu korisnika i uslužioca. Kod ovakvih sistema, korisnik upućuje uslužiocu zahteve za uslugom, pri čemu su razmaci između pojava susednih zahteva za uslugom, kao i dužine njihovog usluživanja, slučajne veličine. Ovakvi simulacioni modeli omogućuju, na primer, određivanje srednjeg vremena čekanja početka usluživanja pojedinih zahteva, ako su poznate: raspodela verovatnoća razmaka između pojava susednih zahteva i raspodela verovatnoća dužina njihovog usluživanja. Potrebno je samo po prvoj od ovih raspodela generisati seriju slučajnih brojeva, koji predstavljaju razmake između pojava susednih zahteva, a po drugoj od njih, generisati seriju slučajnih brojeva, koji predstavljaju dužine usluživanja zahteva. Na osnovu ovih serija formira se vremenska osa (Slika 2.1), koja prikazuje trenutke pojava zahteva, odnosno trenutke početaka i krajeva njihovog usluživanja.



Slika 2.1: Vremenska osa

Srednje vreme čekanja početka usluživanja pojedinih zahteva se dobije kada se ukupnim brojem zahteva podeli suma vremena koja su protekla između pojava pojedinih zahteva i početaka njihovog usluživanja.

U prethodnom primeru simulacionog modela, korisniku odgovara nit korisnik koja, na osnovu zadane raspodele, određuje trenutke pojava pojedinih zahteva. Te trenutke ona prosleđuje niti uslužiocu. Nit uslužilac na osnovu ovih trenutaka i dužina usluživanja pojedinih zahteva (koje, takođe, diktira zadana raspodela), određuje: trenutke početaka usluživanja pojedinih zahteva, dužine čekanja ovih početaka i srednje vreme čekanja početka usluživanja pojedinih zahteva. Važno je zapaziti da se nit korisnik i nit uslužilac nalaze u odnosu proizvođač i potrošač.

Komunikacioni kanal između proizvođača i potrošača uspostavlja deljena promenljiva **box**. Kroz ovaj kanal se mogu slati poruke koje se sastoje od jednog celog broja (**int**).

U implementiranom primeru simulacionog modela se ne koriste slučajni brojevi, radi jednostavnosti.

U simulacionom modelu korisnika i uslužioca ponašanje korisnika opisuje funkcija **thread_user()** (Listing 2.1). Korisnik koristi polje **request_time** koje sadrži, jedan za

drugim, trenutke pojava pojedinih zahteva. Podrazumeva se da je pojava prvog zahteva vezana za trenutak nula. Trenutak pojave narednog zahteva nastaje dodavanjem na trenutak pojave prethodnog zahteva razmaka između ovih trenutaka, određenog konstantom **USER_INTERVAL**. Trenutak pojave svakog zahteva, koji nastupi pre vremenske granice (određene poljem **time_limit**), se šalje uslužiocu, pozivanjem operacije **box.send()**. Pojava prvog zahteva, trenutak čije pojave pada iza ove vremenske granice, izaziva slanje oznake kraja simulacije (konstanta **TERMINATION**).

Ponašanje uslužioca opisuje funkcija **thread_server()**. Uslužilac koristi polje **service_end_time** koje sadrži, jedan za drugim, trenutke kraja usluživanja pojedinih zahteva. Podrazumeva se da je početna vrednost ovog polja nula. Trenutak kraja usluživanja datog zahteva se određuje dodavanjem na trenutak početka njegovog usluživanja dužine ovog usluživanja, određenog konstantom **SERVER_INTERVAL**. Trenutak početka usluživanja narednog zahteva je jednak trenutku kraja usluživanja prethodnog zahteva, ako trenutak pojave narednog zahteva nastupi pre trenutka kraja usluživanja prethodnog zahteva. U tom slučaju, javlja se i čekanje na početak usluživanja. Dužina ovog čekanja se određuje kao razlika između trenutka kraja usluživanja prethodnog zahteva i trenutka pojave narednog zahteva. Ova razlika se dodaje vrednosti polja **mean_waiting_time**. Ako trenutak pojave narednog zahteva sledi iza trenutka kraja usluživanja prethodnog zahteva, tada nema čekanja na početak usluživanja, a trenutak početka usluživanja narednog zahteva je jednak trenutku njegove pojave. Trenutak pojave narednog zahteva se preuzima, pozivanjem operacije **box.receive()**, i smešta u polje **new_request_time**. Po preuzimanju oznake kraja simulacije (konstanta **TERMINATION**) izračunava se i prikazuje srednje vreme čekanja početka usluživanja pojedinih zahteva. Pri tome, polje **request_count** sadrži ukupan broj zahteva.

U funkciji **main()** se kreiraju nit korisnika i nit uslužioca, a zatim se sačeka kraj njihove aktivnosti.

Listing 2.1: Simulacija korisnika i uslužioca (datoteka **p01.cpp**)

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include"box.hh"

Message_box<int>
box;
```

```
const int
TERMINATION = -1;

const int
USER_INTERVAL = 1;

const int
SERVER_INTERVAL = 2;

void
thread_user()
{
   int request_time = 0;
   int time_limit = 3;
   cout << endl << "USER-SERVER SIMULATION" << endl;
   while(request_time < time_limit) {
      box.send(&request_time);
      request_time += USER_INTERVAL;
   }
   box.send(&TERMINATION);
}

void
thread_server()
{
   int service_end_time = 0;
   int new_request_time;
   int request_count = 0;
   int mean_waiting_time = 0;
   while((new_request_time = box.receive()) != TERMINATION) {
      request_count++;
      if(new_request_time < service_end_time)
         mean_waiting_time += service_end_time - new_request_time;
      else
         service_end_time = new_request_time;
      service_end_time += SERVER_INTERVAL;
   }
   mean_waiting_time /= request_count;
   cout << endl << "mean waiting time = " << mean_waiting_time << '\n';
}
```

```
int
main()
{
   thread user(thread_user);
   thread server(thread_server);
   user.join();
   server.join();
}
```

Listing 2.1 sadrži potpun konkurentni program. U toku njegovog izvršavanja dolazi do kreiranja dve niti. Ovaj program nije namenjen za simulaciju nekog stvarnog sistema, nego za ilustraciju kako se takva simulaciju može da se izrazi pomoću konkurentnog programa.

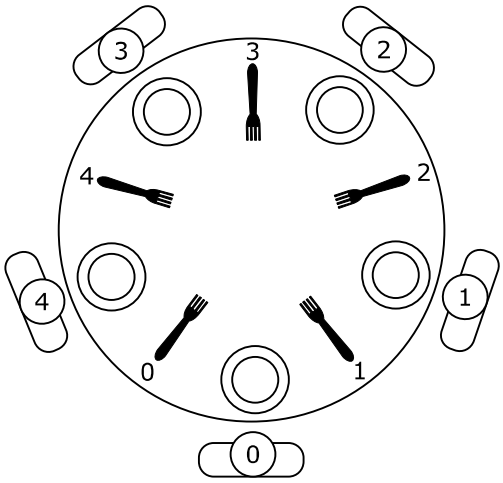
Način, na koji je u prethodnom programu određeno srednje vreme čekanja početka usluživanja pojedinih zahteva, može biti iskorišćen i za određivanje drugih interesantnih parametara ponašanja simuliranog sistema (kao što je, na primer, iskorišćenje uslužioca).

2.6.5 Uspavljivanje niti

Uspavljivanje niti omogućuje funkcija **sleep_for()**. Njen parametar dopušta zadavanje broja milisekundi koji određuje najkraći period odlaganja aktivnosti niti pozivaoca funkcije **sleep_for()**. Poziv funkcije **sleep_for()** sa argumentom većim od nula dovodi do uspavljivanja niti pozivaoca i do aktiviranja najprioritetnije spremne niti. Buđenje tako uspavane niti nastupa najranije nakon isticanja zadatog perioda odlaganja njene aktivnosti.

2.6.6 Problem pet filozofa

Zauzimanje više primeraka resursa iste vrste, neophodnih za aktivnost svake niti iz neke grupe niti, predstavlja tipičan problem konkurentnog programiranja. On se, u literaturi, ilustruje primerom problema pet filozofa (*dining philosophers*). Svaki od njih provodi život razmišljajući u svojoj sobi i jedući u zajedničkoj trpezariji. U njoj se nalazi pet stolica oko okruglog stola sa pet tanjira i pet viljuški između njih (Slika 2.2). Pošto se, po želji filozofa, u trpezariji služe uvek špagete, svakom filozofu su za jelo potrebne dve viljuške (one predstavljaju resurse koje filozofi zauzimaju). Ako svi filozofi istovremeno ogladne, uđu u trpezariju, sednu na svoje mesto za stolom i uzmu viljušku levo od sebe, tada nastupa **mrtva petlja** (*deadlock*), s kobnim ishodom po život filozofa.



Slika 2.2: Trpezarija

Ponašanje svakog filozofa opisuje funkcija **thread_philosopher()** (Listing 2.2). Razmišljanje filozofa se predstavlja kao odlaganje aktivnosti niti koja reprezentuje filozofa. Trajanje ovog odlaganje određuje konstanta **THINKING_PERIOD**. Na sličan način se predstavlja objedovanje filozofa. Trajanje obroka filozofa određuje konstanta **EATING_PERIOD**. Uzimanje viljuške pre jela i njeno vraćanje posle jela, opisuju operacije **take_fork()** i **release_fork()** klase **Dining_table**. Svaki filozof uzima viljuške jednu po jednu samo ako su one slobodne. U suprotnom, filozof čeka da svaka viljuška postane raspoloživa. Prilikom vraćanja viljušaka filozof oslobađa viljuške jednu po jednu i omogući nastavak aktivnosti svojih suseda.

Operacije **take_fork()** i **release_fork()** klase **Dining_table**, dopuštaju pojavu mrtve petlje. Da bi se ona sigurno desila uvedena su odlaganja aktivnosti niti, koja reprezentuje filozofa, u trajanju koje određuje konstanta **MEANTIME**.

Polje **fork_available** deljene klase **Dining_table** omogućuje očekivanje ispunjenja uslova da je viljuška raspoloživa, kao i objavljivanje ispunjenosti ovog uslova. Klasa **Dining_table** sadrži i polja **philosopher_state** i **fork_state**. Prvo od njih izražava stanja filozofa (**THINKING**, **WAITING_LEFT_FORK**, **HOLDING_ONE_FORK**, **WAITING_RIGHT_FORK**, **EATING**), a drugo stanja viljuški (**FREE**, **BUSY**).

Operacija **show()** klase **Dining_table** omogućuje prikazivanje svake promene stanja filozofa. Za svakog filozofa se u zagradama navode njegova numerička oznaka i njegovo stanje.

Funkcija **mod5()** podržava modulo aritmetiku.

U funkciji **main()** se kreiraju niti filozofi, a zatim se sačeka kraj njihove aktivnosti. Svaka od ovih niti preuzme svoj identitet (**Dining_table::take_identity()**): **0**, **1**, **2**, **3** i **4** koji omogućuje razlikovanje filozofa.

Listing 2.2: Pet filozofa (datoteka **p02.cpp**)

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

int
mod5(int a)
{
   return (a > 4 ? 0 : a);
}

enum
Philosopher_state {THINKING = 'T',
                   WAITING_LEFT_FORK = 'L',
                   HOLDING_ONE_FORK = 'O',
                   WAITING_RIGHT_FORK = 'R',
                   EATING = 'E'};

enum
Fork_state {FREE, BUSY};

class Dining_table {
   mutex mx;
   int philosopher_identity;
   Philosopher_state philosopher_state[5];
   Fork_state fork_state[5];
   condition_variable fork_available[5];
   void show();
  public:
   Dining_table();
   int take_identity();
   void take_fork(int fork, int philosopher,
              Philosopher_state waiting_state,
              Philosopher_state next_state);
   void release_fork(int fork, int philosopher,
                Philosopher_state next_state);
};
```

```
Dining_table::Dining_table()
{
   philosopher_identity=0;
   for(int i = 0; i < 5; i++) {
      philosopher_state[i] = THINKING;
      fork_state[i] = FREE;
   }
}

void
Dining_table::show()
{
  for(int i = 0; i < 5; i++) {
     cout << '(' << (char)(i+'0') << ':'
        << (char)philosopher_state[i] << ") ";
  }
  cout << endl;
}

int
Dining_table::take_identity()
{
   unique_lock<mutex> lock(mx);
   return philosopher_identity++;
}

void
Dining_table::take_fork(int fork, int philosopher,
                        Philosopher_state waiting_state,
                        Philosopher_state next_state)
{
   unique_lock<mutex> lock(mx);
   if(fork_state[fork] == BUSY) {
      philosopher_state[philosopher] = waiting_state;
      show();
      do {fork_available[fork].wait(lock);} while(fork_state[fork] == BUSY) ;
   }
   fork_state[fork] = BUSY;
   philosopher_state[philosopher] = next_state;
   show();
}
```

```
void
Dining_table::release_fork(int fork, int philosopher,
                           Philosopher_state next_state)
{
   unique_lock<mutex> lock(mx);
   fork_state[fork] = FREE;
   philosopher_state[philosopher] = next_state;
   show();
   fork_available[fork].notify_one();
}

Dining_table
dining_table;

const milliseconds
THINKING_PERIOD(10);

const milliseconds
MEANTIME(5);

const milliseconds
EATING_PERIOD(10);

void
thread_philosopher()
{
   int philosopher = dining_table.take_identity();
   int fork = philosopher;
   for(;;) {
      sleep_for(THINKING_PERIOD);
      dining_table.take_fork(fork, philosopher,
                             WAITING_LEFT_FORK, HOLDING_ONE_FORK);
      sleep_for(MEANTIME);
      dining_table.take_fork(mod5(fork+1), philosopher,
                             WAITING_RIGHT_FORK, EATING);
      sleep_for(EATING_PERIOD);
      dining_table.release_fork(fork, philosopher, HOLDING_ONE_FORK);
      sleep_for(MEANTIME);
      dining_table.release_fork(mod5(fork+1), philosopher, THINKING);
   }
}
```

```
int
main()
{
   cout << endl << "DINING PHILOSOPHERS" << endl;
   thread philosopher0(thread_philosopher);
   thread philosopher1(thread_philosopher);
   thread philosopher2(thread_philosopher);
   thread philosopher3(thread_philosopher);
   thread philosopher4(thread_philosopher);
   philosopher0.join();
   philosopher1.join();
   philosopher2.join();
   philosopher3.join();
   philosopher4.join();
}
```

Listing 2.2 sadrži potpun konkurentni program. U toku njegovog izvršavanja dolazi do kreiranja pet niti. Rezultat izvršavanja ovog programa je hronološki prikaz svih promena stanja filozofa koji daje uvid u ponašanje programa (Listing 2.3).

Listing 2.3: Prikaz svih izmena stanja filozofa

```
DINING PHILOSOPHERS
(0:T)  (1:T)  (2:0)  (3:T)  (4:T)
(0:0)  (1:T)  (2:0)  (3:T)  (4:T)
(0:0)  (1:T)  (2:0)  (3:0)  (4:T)
(0:0)  (1:0)  (2:0)  (3:0)  (4:T)
(0:0)  (1:0)  (2:0)  (3:0)  (4:0)
(0:0)  (1:0)  (2:R)  (3:0)  (4:0)
(0:R)  (1:0)  (2:R)  (3:0)  (4:0)
(0:R)  (1:0)  (2:R)  (3:R)  (4:0)
(0:R)  (1:R)  (2:R)  (3:R)  (4:0)
(0:R)  (1:R)  (2:R)  (3:R)  (4:R)
```

U prethodnom rešenju problema pet filozofa mrtva petlja se može sprečiti, ako se filozofi ponašaju asimetrično, odnosno ako parni uzimaju prvo levu, a neparni prvo desnu viljušku.

2.6.7 Problem čitanja i pisanja

Problem čitanja i pisanja (*readers-writers problem*) se može objasniti na primeru kao što je rukovanje bankovnim računima. Bankovni računi pripadaju komitentima banke i sadrže ukupan iznos novčanih sredstava svakog od komitenata. U najjednostavnijem slučaju, rukovanje bankovnim računima se svodi: na prenos sredstava (s jednog računa na drugi) i na proveru (stanja svih) računa. Prenos sredstava obuhvata četiri koraka. Prvi korak sadrži čitanje stanja računa s koga se prenose sredstva. Drugi korak obuhvata

pisanje novog stanja na ovaj račun. Novo stanje se dobije umanjivanjem pročitanog stanja za prenošeni iznos. Treći korak sadrži čitanje stanja računa na koji se prenose sredstva. Četvrti korak obuhvata pisanje novog stanja na ovaj račun. Novo stanje se dobije uvećanjem pročitanog stanja za prenošeni iznos. Pri tome se podrazumeva da nisu dozvoljeni prenosi sredstava iza kojih ostaje negativno stanje računa. Uz pretpostavku da su prenosi sredstava mogući samo između posmatranih bankovnih računa, ukupna suma njihovih stanja je nepromenljiva. Prema tome, provera računa se svodi na čitanja, jedno za drugim, stanja svih računa, radi njihovog sumiranja.

Ispravnost prenosa sredstava zavisi od očuvanja konzistentnosti stanja svih računa, za šta je neophodna međusobna isključivost raznih prenosa sredstava. U suprotnom, moguće su razne greške. Na primer, pokušaj istovremenog obavljanja dva ili više prenosa sredstava sa istog računa bi mogao da dovede do čitanja istog stanja u toku prvih koraka istovremenih prenosa sredstava. Tada umanjenja pročitanog stanja ne bi bila kumulativna, odnosno, samo bi poslednje od pisanja iz drugih koraka istovremenih prenosa sredstava odredilo novo stanje posmatranog računa, a umanjenja pročitanog stanja iz ostalih prenosa bi bila izgubljena. Sem međusobne isključivosti raznih prenosa sredstava, neophodna je i međusobna isključivost prenosa sredstava i provera računa. U suprotnom, provera računa bi mogla da pročita stanje računa s kog se prenose sredstva, i to nakon obavljanja drugog koraka prenosa sredstava, kao i da pročita stanje računa na koji se prenose sredstva, i to pre obavljanja četvrtog koraka prenosa sredstava. Tada bi suma stanja svih računa bila pogrešna, jer ne bi sadržala prenošena sredstva.

Iz prethodne analize sledi: (1) da je za ispravnost prenosa bitno da prenosi budu međusobno isključivi i (2) da je za ispravnost provera bitno da provere i prenosi budu međusobno isključivi. Pošto prenosi sadrže pisanja, a provere samo čitanja, sledi da operacije sa pisanjem moraju biti međusobno isključive, kao što moraju biti međusobno isključive operacije sa pisanjem i operacije sa čitanjem. Za operacije koje sadrže samo čitanja međusobna isključivost nije potrebna.

Bankovne račune predstavlja polje **accounts** deljene klase **Bank** (Listing 2.4). Funkcija **thread_reader()** opisuje aktivnost niti čitača koje proveravaju račune. Funkcije **thread_writer0to1()** i **thread_writer1to0()** opisuju aktivnost niti pisača koje prenose sredstva sa računa 0 na račun 1 i obratno. Potrebnu međusobnu isključivost obezbeđuju privatne operacije **reader_begin()** i **reader_end()**, odnosno privatne operacije **writer_begin()** i **writer_end()** deljene klase **Bank**. Prvi par prethodnih operacija koristi operacija **audit()** klase **Bank**, namenjena nitima čitačima, a drugi par prethodnih operacija koristi operacija **transaction()** klase **Bank**, namenjena nitima pisačima. Status banke određuju broj započetih čitanja (**readers_number**), broj započetih pisanja (**writers_number**), broj odloženih čitanja (**readers_delayed_number**) i broj odloženih pisanja (**writers_delayed_number**). Podrazumeva se da se daje prednosti pisanju nad čitanjem, pa čitanja ne mogu započeti, ako ima odloženih pisanja.

Polje **readers_q** deljene klase **Bank** omogućuje nitima čitačima da sačekaju ispunjenje uslova za početak čitanja, a polje **writers_q** deljene klase **Bank** omogućuje nitima pisačima da sačekaju ispunjenje uslova za početak pisanja.

Operacija **show()** klase **Bank** omogućuje prikazivanje svake promene statusa banke (promene brojeva započetih i odloženih čitanja, kao i započetih odloženih pisanja).

U funkciji **main()** se kreira pet niti (3 čitača i 2 pisača) u redosledu koji omogućuje proveru ispunjenosti uslovne sinhronizacije, a zatim se sačeka kraj njihove aktivnosti.

Listing 2.4: Čitači i pisači (datoteka **p03.cpp**)

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

const unsigned
ACCOUNTS_NUMBER = 10;

const int
INITIAL_AMOUNT = 100;

class Bank {
   mutex mx;
   int accounts[ACCOUNTS_NUMBER];
   short readers_number;
   short writers_number;
   short readers_delayed_number;
   short writers_delayed_number;
   condition_variable readers_q;
   condition_variable writers_q;
   void show();
   void reader_begin();
   void reader_end();
   void writer_begin();
   void writer_end();
  public:
   Bank();
   void audit();
   void transaction(unsigned source, unsigned destination);
};
```

```
Bank::Bank()
{
   for(unsigned i = 0; i < ACCOUNTS_NUMBER; i++)
      accounts[i] = INITIAL_AMOUNT;
   readers_number = 0;
   writers_number = 0;
   readers_delayed_number = 0;
   writers_delayed_number = 0;
}

void
Bank::show()
{
  cout << "RN: " << readers_number << "   RDN: " << readers_delayed_number
      << "   WN: " << writers_number << "   WDN: " << writers_delayed_number
      << endl;
}

void
Bank::reader_begin()
{
   unique_lock<mutex> lock(mx);
   if((writers_number > 0) || (writers_delayed_number > 0)){
      readers_delayed_number++;
      show();
      do { readers_q.wait(lock); }
      while((writers_number > 0) || (writers_delayed_number > 0));
   }
   readers_number++;
   show();
   if(readers_delayed_number > 0){
      readers_delayed_number--;
      show();
      readers_q.notify_one();
   }
}
```

```
void
Bank::reader_end()
{
   unique_lock<mutex> lock(mx);
   readers_number--;
   show();
   if((readers_number == 0) && (writers_delayed_number > 0)){
      writers_delayed_number--;
      show();
      writers_q.notify_one();
   }
}

void
Bank::writer_begin()
{
   unique_lock<mutex> lock(mx);
   if((readers_number > 0) || (writers_number > 0)){
      writers_delayed_number++;
      show();
      do { writers_q.wait(lock); }
      while((readers_number > 0) || (writers_number > 0));
   }
   writers_number++;
   show();
}
```

```
void
Bank::writer_end()
{
   unique_lock<mutex> lock(mx);
   writers_number--;
   show();
   if(writers_delayed_number > 0){
      writers_delayed_number--;
      show();
      writers_q.notify_one();
   } else if(readers_delayed_number > 0) {
      readers_delayed_number--;
      show();
      readers_q.notify_one();
    }
}

const milliseconds
READING_PERIOD(1);

void
Bank::audit()
{
   int sum = 0;
   reader_begin();
   sleep_for(READING_PERIOD);
   for(unsigned i = 0; i < ACCOUNTS_NUMBER; i++)
      sum += accounts[i];
   reader_end();
   if(sum != ACCOUNTS_NUMBER*INITIAL_AMOUNT) {
      unique_lock<mutex> lock(mx);
      cout << " audit error " << endl;
   }
}
```

```
const milliseconds
WRITING_PERIOD(1);

void
Bank::transaction(unsigned source, unsigned destination)
{
   int amount;
   writer_begin();
   sleep_for(WRITING_PERIOD);
   amount = accounts[source];
   accounts[source] -= amount;
   accounts[destination] += amount;
   writer_end();
}

Bank
bank;

void
thread_reader()
{
   bank.audit();
}

void
thread_writer0to1()
{
   bank.transaction(0, 1);
}

void
thread_writer1to0()
{
   bank.transaction(1, 0);
}
```

```
int
main()
{
   cout << endl << "READERS AND WRITERS" << endl;
   thread reader0(thread_reader);
   thread reader1(thread_reader);
   thread writer0(thread_writer0to1);
   thread reader2(thread_reader);
   thread writer1(thread_writer1to0);
   reader0.join();
   reader1.join();
   writer0.join();
   reader2.join();
   writer1.join();
}
```

Listing 2.4 sadrži potpun konkurentni program. U toku njegovog izvršavanja dolazi do kreiranja pet niti. Rezultat izvršavanja ovog programa je hronološki prikaz svih promena statusa banke koji daje uvid u ponašanje programa (Listing 2.5).

Listing 2.5: Prikaz svih izmena statusa banke

```
READERS AND WRITERS
RN: 1   RDN: 0   WN: 0   WDN: 0
RN: 2   RDN: 0   WN: 0   WDN: 0
RN: 2   RDN: 0   WN: 0   WDN: 1
RN: 2   RDN: 1   WN: 0   WDN: 1
RN: 2   RDN: 1   WN: 0   WDN: 2
RN: 1   RDN: 1   WN: 0   WDN: 2
RN: 0   RDN: 1   WN: 0   WDN: 2
RN: 0   RDN: 1   WN: 0   WDN: 1
RN: 0   RDN: 1   WN: 1   WDN: 1
RN: 0   RDN: 1   WN: 0   WDN: 1
RN: 0   RDN: 1   WN: 0   WDN: 0
RN: 0   RDN: 1   WN: 1   WDN: 0
RN: 0   RDN: 1   WN: 0   WDN: 0
RN: 0   RDN: 0   WN: 0   WDN: 0
RN: 1   RDN: 0   WN: 0   WDN: 0
RN: 0   RDN: 0   WN: 0   WDN: 0
```

Prethodno rešenje problema čitanja i pisanja nije dobro, ako je važno da čitanja imaju prednost u odnosu na pisanja, odnosno, da provere računa imaju prednost u odnosu na prenose sredstava.

2.7 RIZICI KONKURENTNOG PROGRAMIRANJA

Opisivanje obrada podataka je jedini cilj sekvencijalnog, a osnovni cilj konkurentnog programiranja. Bolje iskorišćenje računara i njegovo čvršće sprezanje sa okolinom su dodatni ciljevi konkurentnog programiranja, po kojima se ono i razlikuje od sekvencijalnog programiranja. Od suštinske važnosti je da ostvarenje dodatnih ciljeva ne ugrozi ostvarenje osnovnog cilja, jer je on neprikosnoven, pošto je konkurentni program upotrebljiv jedino ako iza svakog od njegovih izvršavanja ostaju samo ispravno obrađeni podaci.

Ostvarenja dodatnih ciljeva konkurentnog programiranja uzrokuju da se za konkurentni program vezuje više poželjnih osobina, nego za sekvencijalni program. Tako, tipične poželjne osobine sekvencijalnog, a to znači i konkurentnog programa obuhvataju tvrdnje uključivanja poželjnog, kao što je tvrdnja da nakon izvršavanja programa ostaju ispravno obrađeni podaci, i tvrdnje isključivanja nepoželjnog, kao što je tvrdnja da program ne sadrži beskonačne petlje. Tipične dodatne poželjne osobine konkurentnog programa obuhvataju tvrdnje uključivanja poželjnog, kao što je tvrdnja da su, u toku izvršavanja programa, deljene promenljive stalno konzistentne, i tvrdnje isključivanja nepoželjnog, kao što je tvrdnja da u toku izvršavanja programa ne dolazi do trajnog zaustavljanja aktivnosti niti. To znači, u toku pravljenja konkurentnog programa neophodno je obratiti posebnu pažnju na način ostvarenja sinhronizacije i na proveru da li ima propusta zbog kojih se može javiti nekonzistentnost deljenih promenljivih ili doći do trajnog zaustavljanja aktivnosti pojedinih niti.

Konkurentne biblioteke nude sredstva koja, sem što pomažu ostvarenje dodatnih ciljeva konkurentnog programiranja, omogućuju i sprečavanje štetnih preplitanja. Da bi se štetna preplitanja sprečila i u toku izvršavanja konkurentnih programa izbegle zagonetne greške, čija pojava ima slučajan karakter, neophodna je pažljiva upotreba sredstava za konkurentno programiranje i usredsređenost na otkrivanje i izbegavanje situacija bremenitih štetnim preplitanjima. Bez opreznog i odmerenog korišćenja ovih sredstava, pravljenje upotrebljivog konkurentnog programa nije izvesno, jer pravila primene ovih sredstava ostavljaju dovoljno prostora za raznovrsne propuste. Otkrivanje i otklanjanje ovakvih propusta je potpuno u nadležnosti programera. U nastavku su navedeni tipični primeri pomenutih propusta.

Ispravnu obradu podataka ugrožava narušavanje konzistentnosti deljenih promenljivih u toku izvršavanja konkurentnog programa. Do narušavanja konzistentnosti deljenih promenljivih dolazi, ako na kraju isključivog regiona deljena promenljiva nije u konzistentnom stanju ili ako se operacija **wait()** pozove pre nego je deljena promenljiva dovedena u konzistentno stanje:

```
{
  unique_lock<mutex> lock(mx);
    //< exclusive region 1 >
  some_condition.wait(lock);
    //< exclusive region 2 >
}
```

Prethodni poziv operacije **wait()** predstavlja (prikriveni) kraj isključivog regiona, jer operacija **wait()** vraća propusnicu i izaziva preključivanje na novu nit. Ova nit može da dobije vraćenu propusnicu i da uđe u neki od isključivih regiona iste deljene promenljive. Jasno, na programeru je da oceni da li rukovanje deljenom promenljivom sme da se rasporedi u više isključivih regiona, odnosno, da oceni kada je nastupio momenat za pozivanje operacije **wait()**.

Upotrebljivost konkurentnih programa ugrožava i pojava međuzavisnosti niti, poznata pod nazivom mrtva petlja. Ona dovodi do trajnog zaustavljanja aktivnosti niti, a to ima za posledicu da izvršavanje konkurentnog programa nema kraja. Konkurentni program, u toku čijeg izvršavanja je moguća pojava mrtve petlje, nije upotrebljiv, jer pojedina od njegovih izvršavanja, koja nemaju kraja, ne dovode do uspešne obrade podataka. To znači da iza svakog izvršavanja ovakvog konkurentnog programa ne ostaju ispravno obrađeni podaci. Do mrtve petlje može da dođe, na primer, ako se iz jedne deljene klase pozivaju operacije druge deljene klase, pod uslovom da je bar jedna od pozivanih operacija blokirajuća. U nastavku su navedene pojednostavljene definicije ovakve dve deljene klase (Listing 2.6).

Listing 2.6: Klase **Activity** i **Manager**

```
class Activity {
   mutex mx_activity;
   condition_variable activity_permission;
  public:
   void stop();
   void start();
};

void
Activity::stop()
{
   unique_lock<mutex> lock(mx_activity);
   activity_permission.wait(lock);
}

void
Activity::start()
{
   unique_lock<mutex> lock(mx_activity);
   activity_permission.notify_one();
}
```

```
class Manager {
   mutex mx_manager;
   Activity activity;
  public:
   void disable_activity();
   void enable_activity();
};

void
Manager::disable_activity()
{
   unique_lock<mutex> lock(mx_manager);
   activity.stop();
}

void
Manager::enable_activity()
{
   unique_lock<mutex> lock(mx_manager);
   activity.start();
}

Manager
manager;
```

Nit, koja pozove operaciju **manager.disable_activity()** i dobije propusnicu za ulazak u isključivi region deljene promenljive **manager**, poziva blokirajuću operaciju **activity.stop()**. Ako ova nit zatim dobije i propusnicu za ulazak u isključivi region deljene promenljive **activity**, tada ona zaustavlja svoju aktivnost u okviru poziva operacije **activity_permission.wait()**. Pre zaustavljanja aktivnosti ove niti, poziv operacije **activity_permission.wait()** vraća propusnicu deljene promenljive **activity**. Na taj način se stvara mogućnosti da neka druga nit dobije ovu propusnicu i pozove operaciju **activity_permission.notify_one()**, radi objave ispunjenosti uslova za nastavak aktivnosti prve niti. Međutim, pošto propusnica deljene promenljive **manager** nije vraćena, nema mogućnosti za izvršavanje opercije **manager.enable_activity()**, pa ni za pozivanje operacije **activity.start()**. To znači da pomenuta objava ispunjenja uslova nije moguća, pa je aktivnost prve niti trajno zaustavljena. Opisana situacija predstavlja tipičan primer očekivanja ispunjenja uslova koje neće uslediti.

Mrtve petlje, koje ilustruje prethodni primer, se mogu sprečiti, ako se blokirajuća operacija ne poziva iz isključivog regiona. U tom slučaju ne ostaje zauzeta propusnica, pa nema zapreke ni da se objavi ispunjenje očekivanog uslova.

Sprečavanje mrtve petlje zahteva pažljivo programiranje i analizu programa, radi otkrivanja mogućnosti njene pojave i načina njenog otklanjanja.

Nenamerno izazivanje konačnog, ali nepredvidivo dugog zaustavljanja aktivnosti niti u toku isključivog regiona može da ima negativne posledice na izvršavanje programa. To se, na primer, desi, kada se iz isključivog regiona pozivaju potencijalno blokirajuće operacije, poput funkcije **sleep_for()**, jer nit, čija aktivnost je zaustavljena, zadržava propusnicu, koja je potrebna za pristupanje deljenoj promenljivoj čiji je pomenuti isključivi region. Na taj način, ona privremeno onemogućuje pristupanje ovoj deljenoj promenljivoj.

U situaciji, u kojoj je aktivnost svih niti trajno ili privremeno zaustavljena, o angažovanju procesora brine se operativni sistem.

Globalne **const** promenljive, koje služe za smeštanje podataka, raspoloživih svim nitima, ne spadaju u deljene promenljive.

2.8 PITANJA

1. Šta je preplitanje?
2. Da li preplitanje ima slučajan karakter?
3. Šta izaziva pojavu preplitanja?
4. Da li preplitanje može uticati na rezultat izvršavanja programa?
5. Šta su deljene promenljive?
6. Šta je preduslov očuvanja konzistentnosti deljenih promenljivih?
7. Šta su kritične sekcije?
8. Šta je sinhronizacija?
9. Koje vrste sinhronizacije postoje?
10. Šta je atomski region?
11. Šta sužava primenu atomskih regiona?
12. Čemu služi propusnica?
13. Šta se dešava sa niti koja zatraži, a ne dobije propusnicu?
14. Šta se dešava kada nit vrati propusnicu?
15. Kako se štiti konzistentnost propusnica?
16. Šta je isključivi region?
17. Šta uvode poželjne osobine konkurentnih programa?
18. Po čemu se konkurentno programiranje razlikuje od sekvencijalnog?
19. Koje prednosti ima konkurentna biblioteka u odnosu na konkurentni programski jezik?
20. Kako se opisuju niti?
21. Kako se kreiraju niti?
22. Kada se zauzima propusnica deljene promenljive?
23. Kada se oslobađa propusnica deljene promenljive?
24. Kakvu ulogu ima klasa mutex?
25. Kakve operacije sadrži klasa mutex?
26. Kakvu ulogu ima klasa unique_lock?
27. Kakve operacije sadrži klasa unique_lock?

28. Kakvu ulogu ima klasa condition_variable?
29. Kakve operacije sadrži klasa condition_variable?
30. U pozivu koje od operacija klase condition_variable se vraća propusnica?
31. Koje vrste razmene poruka postoje?
32. U čemu se razlikuju sinhrona i asinhrona razmena poruka?
33. Šta omogućuje funkcija sleep_for()?
34. Po kojim ciljevima se konkurentno programiranje razlikuje od sekvencijalnog programiranja?
35. Zašto operacija condition_variable::wait() predstavlja prikriveni kraj isključivog regiona?
36. Šta je mrtva petlja?

2.9 ZADACI

1. Izmeniti klasu **Message_box** tako da podrži sinhronu ramenu poruka.
2. Izmeniti program **p01.cpp** tako da prikazuje iskorišćenje uslužioca.
3. Izmeniti program **p02.cpp** tako da mrtva petlja ne bude moguća.
4. Izmeniti program **p03.cpp** tako da čitanja imaju prednostu u odnosu na pisanja.

3 SINHRONIZACIJA POMOĆU SEMAFORA

3.1 SEMAFORI

Isključivi regioni ne predstavljaju jedini pristup za ostvarivanje sinhronizacije niti. Sinhronizacija niti može da se zasnuje i na ideji saobraćajnog semafora koji reguliše ulazak vozova u stanicu sa jednim kolosekom. Stanični kolosek predstavlja saobraćajnu kritičnu sekciju. Na njega treba da dolaze vozovi samo jedan po jedan. Kada se jedan voz nalazi na staničnom koloseku, pred semaforom se moraju zaustaviti svi vozovi koji treba da dođu na stanični kolosek. Po analogiji sa saobraćajnim semaforom prolaz niti kroz (softversku) kritičnu sekciju bi regulisao (softverski) **semafor**.

Sinhronizacija niti, koju omogućuje semafor, se zasniva na zaustavljanju aktivnosti niti, kao i na omogućavanju nastavljanja njihove aktivnosti. Ulazak niti u kritičnu sekciju zavisi od stanja semafora. Kada stanje semafora dozvoli ulazak niti u kritičnu sekciju, pri ulasku se semafor prevodi u stanje koje onemogućuje ulazak druge niti u kritičnu sekciju. Ako se takva nit pojavi, njena aktivnost se zaustavlja pred kritičnom sekcijom. Pri izlasku niti iz kritične sekcije semafor se prevodi u stanje koje dozvoljava novi ulazak u kritičnu sekciju i ujedno omogućuje nastavak aktivnosti niti koja najduže čeka na ulaz u kritičnu sekciju (ako takva nit postoji).

Semafori se obično implementiraju u okviru operativnog sistema i tada se njihova implementacija obično zasniva na (kratkotrajnom) onemogućenju prekida. Ali da bi se pokazala ekvivalencija isključivih regiona i semafora, kao i da bi se pojasnilo značenje semafora, njihovu implementaciju ovde opisuje deljana klasa **Semaphore** (Listing 3.1). Stanje semafora je sadržano u polju **state** klase **Semaphore**. Kada semafor reguliše međusobnu isključivost niti, tada pozitivna vrednost ovoga polja pokazuje da je moguć ulazak u kritičnu sekciju. Podrazumevajuća početna vrednost ovoga polja je 1. Klasa **Semaphore** sadrži polje **queue** koje omogućuje zaustavljanje aktivnosti niti, ako stanje semafora ne dozvoljava ulazak u kritičnu sekciju, i nastavak aktivnosti niti, kada stanje semafora dozvoli ulazak u kritičnu sekciju. Klasa **Semaphore** nudi operacije **stop()** i **resume()**. Operacija **stop()** se poziva na početku kritične sekcije, radi provere da li je moguć ulazak u kritičnu sekciju. U okviru ove provere, stanje semafora se menja da bi se onemogućio novi ulazak u kritičnu sekciju. Ako ulazak u kritičnu sekciju nije moguć (ako stanje semafora nije veće od nula), aktivnost niti pozivaoca ove operacije se zaustavlja. Operacija **resume()** se poziva na kraju kritične sekcije, radi izmene stanja semafora i omogućavanja da u kritičnu sekciju uđe jedna od niti koje čekaju pred kritičnom sekcijom (ako takve niti postoje).

Listing 3.1: Klasa **Semaphore** (datoteka **sem.hh**)

```
class Semaphore {
   mutex mx;
   int state;
   condition_variable queue;
  public:
   Semaphore(int value = 1) : state(value) {};
   void stop();
   void resume();
};

void
Semaphore::stop()
{
   unique_lock<mutex> lock(mx);
   while(state < 1)
      queue.wait(lock);
   state--;
}

void
Semaphore::resume()
{
   unique_lock<mutex> lock(mx);
   state++;
   queue.notify_one();
}
```

Semafori i isključivi regioni predstavljaju dva različita pristupa sinhronizaciji. Isključivi regioni su prilagođeni objektno orijentisanom programiranju, dok su semafori prilagođeni procedurnom programiranju.

3.2 VRSTE I UPOTREBA SEMAFORA

Semafor čije stanje ne može preći vrednost 1 se zove **binarni semafor** (*binary semaphore*). On omogućuje ostvarenje sinhronizacije međusobne isključivosti, ako se njegovo stanje inicijalizuje na vrednost 1. Tada su njegove operacije **stop()** i **resume()** slične operacijama **lock()** i **unlock()** klase **mutex**. Ako se stanje binarnog semafora inicijalizuje na vrednost 0, tada su njegove operacije **stop()** i **resume()** slične operacijama **wait()** i **notify_one()** klase **condition_variable**. Ali, između upotrebe ovih operacija postoji bitna razlika. Operacije klase **condition_variable** su namenjene za ostvarenje uslovne sinhronizacije u okviru kritičnih sekcija u kojima je međusobna

isključivost ostvarena pomoću operacija klase **mutex**. Međutim, upotreba operacija binarnog semafora (sa stanjem inicijalizovanim na vrednost 0) po uzoru na operacije klase **condition_variable** u okviru kritičnih sekcija u kojima je međusobna isključivost ostvarena pomoću operacija drugog binarnog semafora (sa stanjem inicijalizovanim na vrednost 1) izaziva mrtvu petlju. Zato se uvodi posebna vrsta binarnog semafora, nazvana **raspodeljeni binarni semafor** (*split binary semaphore*). On se realizuje pomoću više binarnih semafora, za koje važi ograničenje da suma njihovih stanja ne može preći vrednost 1. Pomoću raspodeljenog binarnog semafora se ostvaruje uslovna sinhronizacija tako što se na ulazu u svaku kritičnu sekciju poziva operacija **stop()** jednog od njegovih binarnih semafora, a na izlazu iz nje operacija **resume()** tog ili nekog od preostalih binarnih semafora. Na taj način najviše jedna nit se može nalaziti najviše u jednoj od pomenutih kritičnih sekcija, jer su stanja svih semafora manja od vrednosti 1 za vreme njenog boravka u dotičnoj kritičnoj sekciji.

Nova verzija templejt klase **Message_box** (Listing 3.2) sadrži primer upotrebe raspodeljenih binarnih semafora. Naizmenično slanje i prijem ispravnih poruka se zasniva na korišćenju raspodeljenih binarnih semafora **empty** i **full**, odnosno na unakrsnom pozivanju njihovih operacija **stop()** i **resume()**. Ključ za uspešnu uslovnu sinhronizaciju je inicijalizacija početnog stanja semafora **full** na vrednost 0, koja sprečava primaoca poruke da primi poruku pre nego je ona poslana. Kod slanja poruke, poziv operacije **empty.stop()** onemogućuje novo slanje poruke, dok prethodno poslana poruka ne bude preuzeta, a poziv operacije **full.resume()** omogućuje prijem poslane poruke. Simetrično tome, kod prijema poruke, poziv operacije **full.stop()** onemogućuje novi prijem poruke, dok nova poruka ne bude poslana, a poziv operacije **empty.resume()** omogućuje slanje nove poruke. Na taj način se ostvaruje naizmenično slanje i prijem poruka.

Listing 3.2: Razmena poruka, zasnovana na raspodeljenom binarnom semaforu (datoteka **sembox.hh**)

```
#include "sem.hh"

template<class MESSAGE>
class Message_box {
   MESSAGE content;
   Semaphore empty;
   Semaphore full;
  public:
   Message_box() : full(0) {};
   void send(const MESSAGE* message);
   MESSAGE receive();
};
```

```
template<class MESSAGE>
void
Message_box<MESSAGE>::send(const MESSAGE* message)
{
   empty.stop();
   content = *message;
   full.resume();
}

template<class MESSAGE>
MESSAGE
Message_box<MESSAGE>::receive()
{
   MESSAGE message;
   full.stop();
   message = content;
   empty.resume();
   return message;
}
```

Semafor, čije stanje može sadržati vrednost veću od 1, se naziva **generalni semafor** (*general semaphore*). On omogućuje ostvarenje uslovne sinhronizacije prilikom rukovanja resursima. Pozitivno stanje generalnog semafora može predstavljati broj slobodnih primeraka nekog resursa. Zahvaljujući tome, zauzimanje primerka resursa se može opisati pomoću operacije **stop()**, a njegovo oslobađanje pomoću operacije **resume()** generalnog semafora. Upotrebu generalnih semafora ilustruje nova verzija klase **List** (Listing 3.3). Ova klasa sadrži binarni semafor **mex** i generalni semafor **list_member_count**. Binarni semafor omogućuje međusobnu isključivost prilikom uvezivanja i izvezivanja slobodnog bafera. Generalni semafor omogućuje uslovnu sinhronizaciju, jer njegovo stanje pokazuje broj slobodnih bafera (ono je na početku inicijalizovano na 0). Nakon uvezivanja slobodnog bafera, operacija **link()** poziva operaciju **resume()** generalnog semafora, radi uvećavanja njegovog stanja za 1 i radi eventualnog pokretanja niti koja čeka pojavu slobodnog bafera u operaciji **unlink()**. Do tog čekanja dovodi poziv operacije **stop()** generalnog semafora. Ovaj poziv umanjuje stanje generalnog semafora za 1, čime se zauzima slobodan bafer, ako on postoji. Ako ne postoji, aktivnost niti se zaustavi. Kada se slobodan bafer pojavi, nit nastavi svoju aktivnost izvezivanjem bafera.

Listing 3.3: Klasa **List**, zasnovana na generalnom semaforu

```
#include "sem.hh"

struct List_member {
   List_member* next;
   char buffer[512];
};

class List {
   List_member* first;
   Semaphore list_member_count;
   Semaphore mex;
  public:
   List () : first(0), list_member_count(0) {};
   void link(List_member* member);
   List_member* unlink();
};

void
List::link(List_member* member)
{
   mex.stop();
   member->next=first;
   first=member;
   mex.resume();
   list_member_count.resume();
}

List_member*
List::unlink()
{
   List_member* unlinked;
   list_member_count.stop();
   mex.stop();
   unlinked=first;
   first=first->next;
   mex.resume();
   return unlinked;
}
```

3.3 REŠENJE PROBLEMA PET FILOZOFA POMOĆU SEMAFORA

Rešenje problema pet filozofa pomoću semafora (Listing 3.4) sprečava pojavu mrtve petlje, jer za parne filozofe zauzima prvo levu, a za neparne filozofe zauzima prvo desnu viljušku. Viljuške predstavljaju binarni semafori **forks[5]**, koji omogućuju uslovnu sinhronizaciju prilikom zauzimanja viljuški. Međusobnu isključivost omogućuje binarni semafor **mex**. Stanja filozofa su promenjena, jer nije moguća mrtva petlja, pa nije bitno koju viljušku filozof čeka.

Listing 3.4: Rešenje problema pet filozofa, zasnovano na semaforima (datoteka **p04.cpp**)

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include "sem.hh"

int
mod5(int a)
{
   return (a > 4 ? 0 : a);
}

enum
Philosopher_state { THINKING = 'T', HUNGRY = 'H', EATING = 'E' };

Philosopher_state philosopher_state[5];
Semaphore       forks[5];
Semaphore       mex;

void
show()
{
   for(int j = 0; j < 5; j++) {
      cout << '(' << (char)(j+'0') << ':'
          << (char)(philosopher_state[j]) << ") ";
   }
   cout << endl;
}
```

```
int
philosopher_identity(0);

const milliseconds
THINKING_PERIOD(10);

const milliseconds
EATING_PERIOD(10);

void
thread_philosopher()
{
   mex.stop();
   int pi = philosopher_identity++;
   philosopher_state[pi] = THINKING;
   mex.resume();
   for(;;) {
      sleep_for(THINKING_PERIOD);
      mex.stop();
      philosopher_state[pi] = HUNGRY;
      show();
      mex.resume();
      forks[pi%2 == 0 ? pi : mod5(pi+1)].stop();
      forks[pi%2 == 0 ? mod5(pi+1) : pi].stop();
      mex.stop();
      philosopher_state[pi] = EATING;
      show();
      mex.resume();
      sleep_for(EATING_PERIOD);
      mex.stop();
      philosopher_state[pi] = THINKING;
      show();
      mex.resume();
      forks[pi].resume();
      forks[mod5(pi+1)].resume();
   }
}
```

```
int
main()
{
   cout << endl << "DINING PHILOSOPHERS" << endl;
   thread philosopher0(thread_philosopher);
   thread philosopher1(thread_philosopher);
   thread philosopher2(thread_philosopher);
   thread philosopher3(thread_philosopher);
   thread philosopher4(thread_philosopher);
   philosopher0.join();
   philosopher1.join();
   philosopher2.join();
   philosopher3.join();
   philosopher4.join();
}
```

3.4 REŠENJE PROBLEMA ČITANJA I PISANJA POMOĆU SEMAFORA

Rešenje problema čitanja i pisanja pomoću semafora se oslanja na raspodeljeni binarni semafor koga obrazuju semafori **readers**, **writers** i **mex** (Listing 3.5).

Listing 3.5: Rešenje problema čitanja i pisanja, zasnovano na semaforima (datoteka **p05.cpp**)

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include "sem.hh"

const int
ACCOUNTS_NUMBER = 10;

const int
INITIAL_AMOUNT = 100;
```

```
class Bank {
   Semaphore mex;
   int accounts[ACCOUNTS_NUMBER];
   short readers_number;
   short writers_number;
   short readers_delayed_number;
   short writers_delayed_number;
   Semaphore readers;
   Semaphore writers;
   void show();
   void reader_begin();
   void reader_end();
   void writer_begin();
   void writer_end();
  public:
   Bank();
   void audit();
   void transaction(unsigned source, unsigned destination);
};

Bank::Bank() : mex(1), readers(0), writers(0)
{
   for(int i = 0; i < ACCOUNTS_NUMBER; i++)
      accounts[i] = INITIAL_AMOUNT;
   readers_number = 0;
   writers_number = 0;
   readers_delayed_number = 0;
   writers_delayed_number = 0;
}

void
Bank::show()
{
  cout << "RN: " << readers_number << "   RDN: " << readers_delayed_number
      << "   WN: " << writers_number << "   WDN: " << writers_delayed_number
      << endl;
}
```

```
void
Bank::reader_begin()
{
   mex.stop();
   if((writers_number > 0) || (writers_delayed_number > 0)) {
      readers_delayed_number++;
      show();
      mex.resume();
      readers.stop();
   }
   readers_number++;
   show();
   if(readers_delayed_number > 0) {
      readers_delayed_number--;
      show();
      readers.resume();
   } else
      mex.resume();
}

void
Bank::reader_end()
{
   mex.stop();
   readers_number--;
   show();
   if((readers_number == 0) && (writers_delayed_number > 0)) {
      writers_delayed_number--;
      show();
      writers.resume();
   } else
      mex.resume();
}
```

```
void
Bank::writer_begin()
{
   mex.stop();
   if((readers_number > 0) || (writers_number > 0)) {
      writers_delayed_number++;
      show();
      mex.resume();
      writers.stop();
   }
   writers_number++;
   show();
   mex.resume();
}

void
Bank::writer_end()
{
   mex.stop();
   writers_number--;
   show();
   if(writers_delayed_number > 0) {
      writers_delayed_number--;
      show();
      writers.resume();
   } else {
      if(readers_delayed_number > 0) {
         readers_delayed_number--;
         show();
         readers.resume();
      } else
         mex.resume();
    }
}

const milliseconds
READING_PERIOD(1);
```

```
void
Bank::audit()
{
   int sum = 0;
   reader_begin();
   sleep_for(READING_PERIOD);
   for(unsigned i = 0; i < ACCOUNTS_NUMBER; i++)
      sum += accounts[i];
   reader_end();
   if(sum != ACCOUNTS_NUMBER*INITIAL_AMOUNT) {
      mex.stop();
      cout << " audit error " << endl;
      mex.resume();
   }
}

const milliseconds
WRITING_PERIOD(1);

void
Bank::transaction(unsigned source, unsigned destination)
{
   int amount;
   writer_begin();
   sleep_for(WRITING_PERIOD);
   amount = accounts[source];
   accounts[source] -= amount;
   accounts[destination] += amount;
   writer_end();
}

Bank
bank;

void
thread_reader()
{
   bank.audit();
}
```

```
void
thread_writer0to1()
{
   bank.transaction(0, 1);
}

void
thread_writer1to0()
{
   bank.transaction(1, 0);
}

int
main()
{
   cout << endl << "READERS AND WRITERS" << endl;
   thread reader0(thread_reader);
   thread reader1(thread_reader);
   thread writer0(thread_writer0to1);
   thread reader2(thread_reader);
   thread writer1(thread_writer1to0);
   reader0.join();
   reader1.join();
   writer0.join();
   reader2.join();
   writer1.join();
}
```

Konkurentno programiranje se u praksi često oslanja samo na mehanizam semafora. Za to je dovoljno ponuditi biblioteku koja sadrži definiciju semafora i operacije za rukovanje semaforom. Mehanizam semafora je jednostavan i efikasan, ali je njegova mana što raspodeljeni binarni semafori nisu baš najpodesnije sredstvo za opisivanje uslovne sinhronizacije. To ilustruje prethodni primer rešenja problema čitanja i pisanja (Listing 3.5).

3.5 PITANJA

1. Šta karakteriše semafor?
2. Koje operacije su vezane za semafor?
3. Kako semafor obezbeđuje sinhronizaciju međusobne isključivosti?
4. Kako se obično implementira semafor?
5. U čemu se semafori razlikuju od isključivih regiona?
6. Koji semafori postoje?

7. Šta karakteriše binarni semafor?
8. Šta karakteriše raspodeljeni binarni semafor?
9. Šta karakteriše generalni semafor?
10. Šta omogućuje raspodeljeni binarni semafor?
11. Šta omogućuje binarni semafor?
12. Šta omogućuje generalni semafor?
13. Koje su prednosti i mane semafora?

4 IZVEDBA CppTss-A

4.1 SPECIFIČNOSTI IZVEDBE CppTss-A

Izvedba CppTss-a se razlikuje od konkurentne biblioteke koju podrazumeva međunarodni standard C++11 po tome što ne predviđa da se konkurentni program izvršava iznad operativnog sistema, nego da se izvršava samostalno (*stand alone*). To znači da ova izvedba uključuje one delove operativnog sistema koji su potrebni za samostalno izvršavanje konkurentog programa. Na taj način ona sadrži rešenja tipičnih problema koji se nalaze u nadležnosti operativnog sistema. Izvedba CppTss-a je namenjena za jednoprocesorski računar i podrazumeva da nije moguće lažno (*spurious*) aktiviranje niti.
Izvedba CppTss-a obuhvata ulazno-izlazne module, izvršioca (*kernel*) i virtuelnu mašinu. Radi izvršioca ona mora da podrži stvaranje atomskih regiona i pisanje drajvera, pa u tom smislu proširuje međunarodni standard C++11.

4.1.1 Atomski regioni

Stvaranje atomskih regiona omogućuje klasa **Atomic_region**. Njen konstruktor onemogućuje prekide, a destruktor vraća prekide u stanje koje je prethodilo akciji konstruktora. Tako, na primer, iskazi:

```
{
  Atomic_region ar;
  ...
}
```

uspostavljaju atomski region od mesta definisanja lokalne promenljive **ar**, pa do kraja složenog iskaza u kome ona postoji.

Upotrebu atomskog regiona ilustruje primer rukovanja pozicijom kursora u kome nisu moguća štetna preplitanja niti i obrada prekida, jer telo operacije **get()** klase **Position** obrazuje atomski region. Pošto se operacija **set()** poziva samo iz obrada prekida, njeno telo po definiciji obrazuje atomski region (jer su prekidi onemogućeni u toku obrade prekida), pa je tako osigurana međusobna isključivost operacija klase **Position**. Izmenjena definicija ove klase je navedena u nastavku (Listing 4.1).

Listing 4.1: Sinhronizovana klasa **Position**

```
class Position {
   int x, y;
  public:
   Position();
   void set(int new_x, int new_y);
   void get(int* current_x, int* current_y);
};

Position::Position()
{
   x = 0;
   y = 0;
}

void
Position::set(int new_x, int new_y)
{
   x = new_x;
   y = new_y;
}

void
Position::get(int* current_x, int* current_y)
{
   Atomic_region ar;
   *current_x = x;
   *current_y = y;
}
```

4.1.2 Klasa Driver

Pisanje drajvera olakšava klasa **Driver**. Nju nasleđuju klase koje opisuju ponašanje drajvera. Drajvere karakteriše saradnja obrađivača prekida i pozadinskih niti.

U okviru klase, koja opisuje ponašanje drajvera, obrađivač prekida se predstavlja u obliku funkcije bez povratne vrednosti i bez parametara. Ova funkcija mora biti **static**, da bi se mogla koristiti njena adresa. Takva moraju biti i sva polja klase kojima ona pristupa.

Operacija **start_interrupt_handling()** klase **Driver** omogućuje smeštanje adrese obrađivača prekida u tabelu prekida. Prvi argument poziva ove operacije predstavlja broj

vektora prekida, a drugi adresu obrađivača prekida.

Saradnja obrađivača prekida i pozadinskih niti podrazumeva da pozadinske niti čekaju dešavanje vanjskih događaja, a da obrađivači prekida objavljuju dešavanje vanjskih događaja. Zato klasa **Driver** sadrži klasu **Event**. Operacija **expect()** klase **Event** omogućuje zaustavljanje aktivnosti niti, koja pozove ovu operaciju, dok se ne desi vanjski događaj koga reprezentuje objekat klase **Event**. Tome prethode prevođenje ove niti u stanje "čeka" i preključivanje procesora na spremnu nit. Predviđeno je da se operacija **expect()** poziva samo iz atomskog regiona. Operacija **signal()** klase **Event** omogućuje objavu dešavanja nekog vanjskog događaja, radi nastavka aktivnosti niti koja (najduže) očekuje dešavanje dotičnog događaja. Operaciju **signal()** ima smisla pozivati samo iz obrađivača prekida, jer jedino oni opisuju reakciju na dešavanje vanjskih događaja. Izvršavanje operacije **signal()** prevodi nit, koja je dočekala dešavanje vanjskog događaja, u stanje "spremna". Podrazumeva se da se operacija **signal()** poziva samo jednom i to na kraju obrade prekida.

Upotrebu klase **Driver** ilustruje primer drajvera koji omogućuje rukovanje vremenom (zadavanje i preuzimanje vremena, predstavljenog brojem sati, minuta i sekundi). Ovaj drajver registruje proticanje vremena u računaru brojanjem otkucaja sata. Otkucaji, čiji period zavisi od takta procesora, nisu uvek podesni za određivanje vremena u danu, predstavljenog brojem sati, minuta i sekundi, jer sekunda ne može uvek da se izrazi celim brojem perioda ovakvih otkucaja. Zato je zgodno uvesti dodatni sat, čiji otkucaji (prekidi) imaju period od tačno jedne sekunde. Rukovanje ovim satom, odnosno rukovanje vremenom opisuje klasa **Timer_driver** (Listing 4.2). Njena tri celobrojna polja: **hour**, **minute** i **second** sadrže broj proteklih sati, minuta i sekundi. Početni sadržaj ovih polja je 0. Konstruktor klase **Timer_driver** u element tabele prekida, koga indeksira konstanta **TIMER** (broj vektora prekida dodatnog sata), smešta adresu njene operacije **interrupt_handler()**, koja je zadužena za periodičnu izmenu sadržaja polja **hour**, **minute** i **second**, sa periodom od jedne sekunde. Pri tome se sadržaj polja **second** povećava za 1 i to samo ako je manji od 59. Inače, on postaje 0, a povećava se sadržaj polja **minute** za 1, opet samo ako je manji od 59. U suprotnom slučaju, on postaje 0, a povećava se sadržaj polja **hour** za 1, ali samo ako je manji od 23. Inače, on postaje 0.

Klasa **Timer_driver** sadrži i operacije **set()** i **get()** za zadavanje i preuzimanje sadržaja njenih polja. To su osetljive operacije, čije preplitanje sa obradom prekida sata je štetno. Na primer, ako se obrada prekida sata desi nakon preuzimanja sadržaja polja **hour**, a pre preuzimanja sadržaja polja **minute**, i ako je, uz to, broj minuta pre periodične izmene bio na granici od 59, tada preuzeto vreme kasni iza stvarnog za 60 minuta (jer je preuzet stari sadržaj polja **hour** i novi sadržaj polja **minute**). U prethodnom primeru opisano je samo jedno od mogućih štetnih preplitanja. Da bi se ovakva štetna preplitanja sprečila, preuzimanja i zadavanja sadržaja njenih polja moraju da budu u atomskim regionima.

Listing 4.2: Klasa **Timer_driver** (datoteka **time.hh**)

```
class Timer_driver : public Driver {
   static int hour;
   static int minute;
   static int second;
   static void interrupt_handler();
  public:
   Timer_driver() { start_interrupt_handling(TIMER, interrupt_handler); };
   void set(const int h, const int m, const int s);
   void get(int* h, int* m, int* s) const;
};

int
Timer_driver : :hour = 0;

int
Timer_driver : :minute = 0;

int
Timer_driver : :second = 0;

void
Timer_driver::interrupt_handler()
{
   if(second < 59)
      second++;
   else {
      second = 0;
      if(minute < 59)
         minute++;
      else {
         minute = 0;
         if(hour < 23)
            hour++;
         else
            hour = 0;
      }
   }
}
```

```
void
Timer_driver::set(const int h, const int m, const int s)
{
   Atomic_region ar;
   hour = h;
   minute = m;
   second = s;
}

void
Timer_driver::get(int* h, int* m, int* s) const
{
   Atomic_region ar;
   *h = hour;
   *m = minute;
   *s = second;
}
```

Prethodno opisani način praćenja proticanja vremena, predstavljenog brojem sati, minuta i sekundi, se može primeniti, bez suštinskih izmena, i kada ne postoji dodatni sat, čiji otkucaji imaju period tačno jednu sekundu. U ovakvom slučaju, jedina izmena se odnosi na operaciju **interrupt_handler()**, u kojoj do povećanja broja sekundi dolazi tek nakon odgovarajućeg broja otkucaja sata.

Upotrebu klase **Driver** ilustruje i primer drajvera koji omogućuje uspavljivanje jedne niti dok ne protekne zadani broj otkucaja sata. Ovo uspavljivanje može da se prikaže kao očekivanje dešavanja zadanog broja otkucaja sata. To opisuje klasa **Sleep_driver** (Listing 4.3). Njena operacija **simple_sleep_for()** omogućuje jednoj niti da zaustavi svoju aktivnost dok se ne desi zadani broj otkucaja sata. Zadatak obrađivača prekida (operacije **interrupt_handler()**) je da odbroji zadani broj otkucaja sata i da nakon toga signalizira da je moguć nastavak aktivnosti uspavane niti.

Listing 4.3: Klasa **Sleep_driver**

```
class Sleep_driver: public Driver {
   static unsigned long countdown;
   static Event alarm;
   static void interrupt_handler();
  public:
   Sleep_driver() { start_interrupt_handling(TIMER, interrupt_handler); };
   void simple_sleep_for(unsigned long duration);
};
```

```
unsigned long
Sleep_driver: :countdown = 0;

void
Sleep_driver::interrupt_handler()
{
  if((countdown > 0) && ((--countdown) == 0))
    alarm.signal();
}

void
Sleep_driver::simple_sleep_for(unsigned long duration)
{
  Atomic_region ar;
  if(duration > 0) {
    countdown = duration;
    alarm.expect();
  };
}
```

4.2 PITANJA

1. Šta obuhvata izvedba CppTss-a?
2. Koja klasa omogućuje stvaranje atomskih regiona?
3. Koju klasu nasleđuju klase koje opisuju ponašanje drajvera?
4. Šta omogućuje operacija **start_interrupt_handling()** klase **Driver**?
5. Koje operacije sadrši klasa **Event**?
6. Šta omogućuje operacija **expect()** klase **Event**?
7. Šta omogućuje operacija **signal()** klase **Event**?

5 ULAZNO-IZLAZNI MODULI CppTss-A

5.1 PODELA ULAZNO-IZLAZNIH MODULA CppTss-A

U ulazno-izlazne module CppTss-a spadaju znakovni i blokovski ulazno-izlazni moduli. Znakovni ulazno-izlazni modul se oslanja na drajver ekrana i drajver tastature. Blokovski ulazno-izlazni modul se oslanja na drajver diska.

5.2 DRAJVERI TASTATURE I EKRANA

Klasa **Display_driver** (Listing 5.1) sadrži drajver ekrana koji upravlja kontrolerom ekrana. Kontroler ekrana (objekat **display_controller**) sadrži registar stanja (**display_controller.status_reg**) i registar podataka (**display_controller.data_reg**). Prikaz znaka na ekranu je moguć ako registar stanja sadrži konstantu **DISPLAY_READY**. U tom slučaju se kod znaka smešta u registar podataka, a u registar stanja se smešta konstanta **DISPLAY_BUSY**. Ova konstanta ostaje u registru stanja dok traje prikaz znaka. Po prikazu znaka, kontroler ekrana smešta u registar stanja konstantu **DISPLAY_READY** (podrazumeva se da se ova vrednost nalazi u registru stanja na početku rada kontrolera ekrana). Pokušaj niti da prikaže znak, dok je u registru stanja konstanta **DISPLAY_BUSY** (dok kontroler ekrana prikazuje prethodni znak), zaustavlja aktivnost niti. Nastavak aktivnosti niti usledi nakon obrade prekida ekrana, koja objavljuje da je prikaz prethodnog znaka završen. Zaustavljanje i nastavak aktivnosti niti omogućuje polje **displayed_char** klase **Display_driver**. Opisano ponašanje drajvera ekrana ostvaruju operacije **character_put()** i **interrupt_handler()** klase **Display_driver**. Broj vektora prekida ekrana određuje konstanta **DISPLAY**.

Listing 5.1: Klasa **Display_driver** (datoteka **char_drivers.cpp**)

```
class Display_driver : public Driver {
   static Event displayed_char;
   static void interrupt_handler();
   Display_driver(const Display_driver&);
   Display_driver& operator=(const Display_driver&);
  public :
   Display_driver() { start_interrupt_handling(DISPLAY, interrupt_handler); };
   void character_put(const char c);
};

Display_driver::Event
Display_driver::displayed_char;
```

```
void
Display_driver::interrupt_handler()
{
   displayed_char.signal();
}

void
Display_driver::character_put(const char c)
{
   Atomic_region ar;
   if(display_controller.status_reg == DISPLAY_BUSY)
      displayed_char.expect();
   display_controller.data_reg = c;
   display_controller.status_reg = DISPLAY_BUSY;
}

static Display_driver
display_driver;
```

Klasa **Keyboard_driver** (Listing 5.2) sadrži drajver tastature koji upravlja kontrolerom tastature. Kontroler tastature (objekat **keyboard_controller**) sadrži registar podataka (**keyboard_controller.data_reg**). Podrazumeva se da pritisak dirke na tastaturi (1) dovede do smeštanja koda odgovarajućeg znaka u registar podataka i (2) izazove prekid tastature. Pomenuti kod znaka se preuzima iz registra podataka u obradi prekida tastature i smešta u cirkularni bafer, ako on nije pun. Cirkularnom baferu odgovara polje **buffer** klase **Keyboard_driver**. Njeno polje **count** određuje popunjenost ovog bafera. Indekse cirkularnog bafera sadrže polja **first_full** i **first_empty** klase **Keyboard_driver**. Pokušaj niti da preuzme znak, kada je cirkularni bafer prazan, zaustavlja njenu aktivnost. Nastavak aktivnosti niti usledi nakon obrade prekida tastature. Zaustavljanje i nastavak aktivnosti niti omogućije polje **pressed** klase **Keyboard_driver**. Opisano ponašanje drajvera tastature ostvaruju operacije **character_get()** i **interrrupt_handler()** klase **Keyboard_driver**. Broj vektora prekida tastature određuje konstanta **KEYBOARD**.

Listing 5.2: Klasa **Keyboard_driver** (datoteka **char_drivers.cpp**)

```
const unsigned
KEYBOARD_BUFFER_SIZE = 1024;

class Keyboard_driver : public Driver {
   static Event pressed;
   static char buffer[KEYBOARD_BUFFER_SIZE];
   static unsigned count;
   static unsigned first_full;
   static unsigned first_empty;
   static void interrupt_handler();
   Keyboard_driver(const Keyboard_driver&);
   Keyboard_driver& operator=(const Keyboard_driver&);
  public:
   Keyboard_driver() { start_interrupt_handling(KEYBOARD, interrupt_handler); };
   char character_get();
};

Keyboard_driver::Event
Keyboard_driver::pressed;

char
Keyboard_driver::buffer[KEYBOARD_BUFFER_SIZE];

unsigned
Keyboard_driver::count = 0;

unsigned
Keyboard_driver::first_full = 0;

unsigned
Keyboard_driver::first_empty = 0;
```

```
void
Keyboard_driver::interrupt_handler()
{
   if(count<KEYBOARD_BUFFER_SIZE) {
      buffer[first_empty++] = keyboard_controller.data_reg;
      if(first_empty == KEYBOARD_BUFFER_SIZE)
         first_empty = 0;
      count++;
      pressed.signal();
   }
}

char
Keyboard_driver::character_get()
{
   char c;
   Atomic_region ar;
   if(count==0)
      pressed.expect();
   c = buffer[first_full++];
   if(first_full == KEYBOARD_BUFFER_SIZE)
      first_full = 0;
   count--;
   return c;
}

static Keyboard_driver
keyboard_driver;
```

5.3 ZNAKOVNI ULAZ-IZLAZ

Prilikom ulaza-izlaza znakova moguća su štetna preplitanja. Sprečavanje štetnih preplitanja ulaznih i izlaznih operacija podrazumeva da su one međusobno isključive. Njihova međusobna isključivost se može ostvariti, ako se tastatura, odnosno ekran zaključa (zauzme) pre i otključa (oslobodi) nakon korišćenja, prilikom svakog izvršavanja odgovarajuće operacije. Neuspešan pokušaj zaključavanja uređaja dovodi do zaustavljanja izvršavanja ovakve operacije, dok zaključavanje ne postane moguće. Zaključavanje tastature i ekrana, tokom izvršavanja ulaznih i izlaznih operacija, se zasniva na korišćenju propusnica. Posebne propusnice reprezentuju ekran i tastaturu. Zauzimanje propusnice odgovara zaključavanju njenog uređaja, a oslobađanje propusnice odgovara otključavanju dotičnog uređaja. Time se obezbeđuje međusobna isključivost pojedinačnih ulaznih i izlaznih operacija.

Znakovni ulaz-izlaz omogućuju klase **Terminal_out** i **Terminal_in**.

Klasa **Terminal_out** (Listing 5.3) omogućuje znakovni izlaz, odnosno prikaz znaknova na ekranu. Ona sadrži operacije koje omogućuju formatiranje prikazivanog podatka (njegovo pretvaranje u niz znakova). Oznake **%6d**, **%6u**, **%11d** i **%11u** određuju broj cifara u decimalnom formatu u kome se prikazuju cifre celih označenih (**d**) i neoznačenih (**u**) brojeva, a oznaka **% .3e** određuje broj cifara iza decimalne tačke u decimalnom formatu u kome se prikazuju cifre razlomljenih brojeva. Za prikaz niza znakova (znakovni izlaz) zadužena je operacija **string_put()** klase **Terminal_out**, koja se brine i o zaključavanju ekrana.

Listing 5.3: Klasa **Terminal_out** (datoteke **char_io.hh** i **char_io.cpp**)

```
const char
endl[] = "\n";

class Terminal_out : private mutex {
   Terminal_out(const Terminal_out&);
   Terminal_out& operator=(const Terminal_out&);
   void string_put(const char* string);
  public:
   Terminal_out() {};
   Terminal_out& operator<<(int number);
   Terminal_out& operator<<(unsigned int number);
   Terminal_out& operator<<(short number);
   Terminal_out& operator<<(unsigned short number);
   Terminal_out& operator<<(long number);
   Terminal_out& operator<<(unsigned long number);
   Terminal_out& operator<<(double number);
   Terminal_out& operator<<(char character);
   Terminal_out& operator<<(const char* string);
   friend class Terminal_in;
};

static const char*
SHORT_FORMAT = "%6d";
static const char*
UNSIGNED_SHORT_FORMAT = "%6u";
static const char*
INT_FORMAT = "%11d";
static const char*
UNSIGNED_FORMAT = "%11u";
static const char*
DOUBLE_FORMAT = "% .3e";
```

```
static const int
SHORT_SIGNIFICANT_FIGURES_COUNT = 5;
static const int
INT_SIGNIFICANT_FIGURES_COUNT = 10;
static const int
DOUBLE_SIGNIFICANT_FIGURES_COUNT = 10;

void
Terminal_out::string_put(const char* string)
{
   lock();
   while(*string != '\0')
            display_driver.character_put(*string++);
   unlock();
}

Terminal_out&
Terminal_out::operator<<(int number)
{
   char string[INT_SIGNIFICANT_FIGURES_COUNT + 2];
   sprintf(string, INT_FORMAT, number);
   string_put(string);
   return *this;
}

Terminal_out&
Terminal_out::operator<<(unsigned int number)
{
   char string[INT_SIGNIFICANT_FIGURES_COUNT + 2];
   sprintf(string, UNSIGNED_FORMAT, number);
   string_put(string);
   return *this;
}

Terminal_out&
Terminal_out::operator<<(short number)
{
   char string[SHORT_SIGNIFICANT_FIGURES_COUNT + 2];
   sprintf(string, SHORT_FORMAT, number);
   string_put(string);
   return *this;
}
```

```
Terminal_out&
Terminal_out::operator<<(unsigned short number)
{
   char string[SHORT_SIGNIFICANT_FIGURES_COUNT + 2];
   sprintf(string, UNSIGNED_SHORT_FORMAT, number);
   string_put(string);
   return *this;
}

Terminal_out&
Terminal_out::operator<<(long number)
{
   char string[INT_SIGNIFICANT_FIGURES_COUNT + 2];
   sprintf(string, INT_FORMAT, number);
   string_put(string);
   return *this;
}

Terminal_out&
Terminal_out::operator<<(unsigned long number)
{
   char string[INT_SIGNIFICANT_FIGURES_COUNT + 2];
   sprintf(string, UNSIGNED_FORMAT, number);
   string_put(string);
   return *this;
}

Terminal_out&
Terminal_out::operator<<(double number)
{
   char string[DOUBLE_SIGNIFICANT_FIGURES_COUNT + 2];
   sprintf(string, DOUBLE_FORMAT, number);
   string_put(string);
   return *this;
}
```

```
Terminal_out&
Terminal_out::operator<<(char character)
{
   char string[2];
   string[0] = character;
   string[1] = '\0';
   string_put(string);
   return *this;
}

Terminal_out&
Terminal_out::operator<<(const char* string)
{
   string_put(string);
   return *this;
}

Terminal_out
cout;
```

Klasa **Terminal_in** (Listing 5.4) omogućuje preuzimanje znakova (znakovni ulaz). Ona obezbeđuje zaključavanje tastature dok se izvršava njena operacija **string_get()**, kao i zaključavanje ekrana, radi eha znakova, preuzetih u ovoj operaciji. Operacija **string_get()** poziva operaciju **edit()** klase **Terminal_in** koja je zadužena za eho znakova na ekranu i njihovo primitivno editiranje. Preostale operacije klase **Terminal_in** koriste operaciju **string_get()** za preuzimanje nizova znakova koji odgovaraju raznim tipovima podataka. One zatim konvertuju te znakove u odgovarajući tip podataka.

Listing 5.4: Klasa **Terminal_in** (datoteke **char_io.hh** i **char_io.cpp**)

```
const int
INPUT_BUFFER_LENGTH = 128;

class Terminal_in : private mutex {
   Terminal_in(const Terminal_in&);
   Terminal_in& operator=(const Terminal_in&);
   unsigned index;
   char c;
   bool pressed_enter;
   char buff[INPUT_BUFFER_LENGTH];
   inline void edit();
   void string_get(unsigned figures_count);
  public:
   Terminal_in() {};
   Terminal_in& operator>>(int &number);
   Terminal_in& operator>>(short &number);
   Terminal_in& operator>>(long &number);
   Terminal_in& operator>>(double &number);
   Terminal_in& operator>>(char &character);
};

static const int
SHORT_SIGNIFICANT_FIGURES_COUNT = 5;
static const int
INT_SIGNIFICANT_FIGURES_COUNT = 10;
static const int
DOUBLE_SIGNIFICANT_FIGURES_COUNT = 10;

#define CHAR_ESC (27)
#define CHAR_LF  ('\n')
#define CHAR_BS1 ('\b')
#define CHAR_BS2 (127)
```

```
void
Terminal_in::edit()
{
   switch(c) {
    case CHAR_ESC:
            buff[index++]=c;
            display_driver.character_put('^');
            break;
    case CHAR_LF:
            pressed_enter = true;
            break;
    case CHAR_BS1:
    case CHAR_BS2:
            if(index>0) {
               buff[--index]='\0';
               display_driver.character_put('\b');
               display_driver.character_put(' ');
               display_driver.character_put('\b');
            }
            break;
    default:
            buff[index++]=c;
            display_driver.character_put(c);
            break;
   }
   buff[index]='\0';
}
```

```
void
Terminal_in::string_get(unsigned figures_count)
{
   lock();
   index = 0;
   pressed_enter = false;
   c = keyboard_driver.character_get();
   cout.lock();
   edit();
   while((index < (figures_count - 1)) && !pressed_enter) {
      c = keyboard_driver.character_get();
      edit();
   }
   cout.unlock();
   unlock();
}

Terminal_in&
Terminal_in::operator>>(int& number)
{
   string_get(INT_SIGNIFICANT_FIGURES_COUNT);
   number = (int)strtol(buff, 0, 10);
   return *this;
}

Terminal_in&
Terminal_in::operator>>(short& number)
{
   string_get(SHORT_SIGNIFICANT_FIGURES_COUNT);
   number = (int)strtol(buff, 0, 10);
   return *this;
}

Terminal_in&
Terminal_in::operator>>(long& number)
{
   string_get(INT_SIGNIFICANT_FIGURES_COUNT);
   number = (int)strtol(buff, 0, 10);
   return *this;
}
```

```
Terminal_in&
Terminal_in::operator>>(double& number)
{
   string_get(DOUBLE_SIGNIFICANT_FIGURES_COUNT);
   number = strtod(buff, 0);
   return *this;
}

Terminal_in&
Terminal_in::operator>>(char& character)
{
   string_get(1);
   character = buff[0];
   return *this;
}

Terminal_in
cin;
```

Zahvaljujući nasleđivanju klase **mutex** operacije klasa **Terminal_out** i **Terminal_in** su blokirajuće.

5.4 DRAJVER DISKA

Klasa **Disk_driver** (Listing 5.5) sadrži drajver diska koji upravlja DMA kontrolerom diska. DMA kontroler diska (objekat **disk_controller**) sadrži registar bloka (**disk_controller.block_reg**), registar bafera (**disk_controller.buffer_reg**), registar smera prenosa (**disk_controller.operation_reg**) i registar stanja (**disk_controller.status_reg**). Podrazumeva se da nit pokreće prenos bloka tako što: (1) u registar bloka smesti broj prenošenog bloka, (2) u registar bafera smesti adresu bafera koji učestvuje u prenosu, (3) u registar smera prenosa smesti konstantu **DISK_READ** ili **DISK_WRITE**, a (4) u registar stanja konstantu **DISK_STARTED**. Nakon toga aktivnost niti se zaustavi dok traje prenos bloka. Kraj prenosa bloka objavi prekid diska. Zaustavljanje i nastavak aktivnosti niti omogućuje polje **ready** klase **Disk_driver**. Opisano ponašanje drajvera diska ostvaruju operacije **block_transfer()** i **interrupt_handler()** klase **Disk_driver**. Broj vektora prekida diska određuje konstanta **DISK**.

Listing 5.5: Klasa **Disk_driver** (datoteka **block_drivers.cpp**)

```
class Disk_driver : public Driver {
   static Event ready;
   static void interrupt_handler();
   Disk_driver(const Disk_driver &);
   Disk_driver& operator=(const Disk_driver &);
  public:
   Disk_driver() { start_interrupt_handling(DISK, interrupt_handler); }
   inline int block_transfer(char* buffer, unsigned block,
                             Disk_operations operation);
};

Disk_driver::Event
Disk_driver::ready;

void
Disk_driver::interrupt_handler()
{
   ready.signal();
}

int
Disk_driver::block_transfer(char* buffer, unsigned block,
                            Disk_operations operation)
{
   int r = -1;
   if(block < DISK_BLOCKS) {
         Atomic_region ar;
         disk_controller.block_reg = block;
         disk_controller.buffer_reg = buffer;
         disk_controller.operation_reg = operation;
         disk_controller.status_reg = DISK_STARTED;
         ready.expect();
         r = 0;
   }
   return r;
}

static Disk_driver
disk_driver;
```

5.5 BLOKOVSKI ULAZ-IZLAZ

Klasa **Disk** (Listing 5.6) opisuje rukovanje virtuelnim diskom, za koga se podrazumeva da ima hiljadu blokova, čija veličina je 512 bajta. Ova klasa definiše operacije **block_get()** i **block_put()**, koje omogućuju preuzimanje bloka sa diska i smeštanje bloka na disk. Prvi parametar ovih operacija pokazuje na niz od 512 bajta radne memorije koji učestvuje u prebacivanju bloka, a drugi parametar određuje broj bloka (u rasponu od 0 do 999). Obe operacije su blokirajuće.

Disk je elektro-mehanički uređaj čija brzina rada je ograničena brzinom pomeranja njihovih mehaničkih delova. On sadrži podatke u obliku magnetnih zapisa, raspoređenih u koncentričnim kružnim stazama. Za pristup podacima je neophodno da glava diska, koja se pomera od centra i ka centru rotacije ploča diska, dođe iznad tražene staze. Pošto brzina pomeranja glave diska ograničava ukupnu brzinu diska, važno je skratiti put koji glava diska prelazi. Optimizacija kretanja glave diska je moguća kada se skupi, zbog sporosti diska, više zahteva za čitanjem ili pisanjem blokova (podataka sa diska). Ovakva optimizacija se svodi na opsluživanje zahteva u redosledu staza na koje se oni odnose, a ne u hronološkom redosledu pojave zahteva. Na taj način se izbegava da glava diska osciluje između vanjskih i unutrašnjih staza diska, prelazeći tako višestruko put, koji bi prešla samo jednom, ako bi, krećući se od unutrašnjih ka vanjskim stazama, opslužila sve zahteve. Da bi se ovo ostvarilo, neophodno je zahteve razvrstati u dva skupa, uređena u rastućem redosledu staza na koje se zahtevi odnose. Pri tome, u jedan skup dolaze zahtevi čije staze se nalaze između trenutnog položaja glave diska i oboda rotirajućih ploča diska, a u drugi skup dolaze zahtevi čije staze se nalaze između centra rotirajućih ploča diska i trenutnog položaja glave diska. Optimizacija kretanja glave diska se može nalaziti u nadležnosti drajvera operativnog sistema.

Da bi optimizacija kretanja glave diska bila moguća, neophodno je uticati na redosled zahteva za čitanjem ili pisanjem blokova. To postavlja specifične zahteve na implementaciju klase **condition_variable**. U izvedbi CppTss-a objekat klase **condition_variable** predstavlja listu uslova u koju se u hronološkom redosledu uvezuju deskriptori niti nakon poziva operacije **wait()** pomenutog objekta. Da bi bilo moguće uticati na redosled u kome se deskriptori niti uvezuju u listu uslova, izvedba CppTss-a proširuje međunarodni standard C++11 tako da predviđa dodatnu funkcionalnost klase **condition_variable**. Podrazumeva se da odabrani redosled deskriptora niti u listi uslova nastaje na osnovu privezaka koji se dodeljuju svakom deskriptoru prilikom njegovog uvezivanja u ovu listu. Privesci imaju oblik neoznačenih celih brojeva, a njihovo dodeljivanje deskriptoru omogućuje dodatni, drugi parametar operacije **wait()**. Privezak se dodeljuje deskriptoru niti, kada ona pozove ovu operaciju. Podrazumevajući privezak je 0. Uticaj na redosled deskriptora niti u listi uslova omogućuju operacije **first()**, **next()** i **last()** klase **condition_variable**.

Operacija **first()** omogućuje pozicioniranje pre prvog deskriptora u listi uslova. Do pozicioniranja dolazi samo ako lista uslova nije prazna i tada poziv ove operacije vraća vrednost **true**, a njen parametar omogućuje preuzimanje priveska deskriptora koji se nalazi iza tačke pozicioniranja.

Operacija **next()** omogućuje pozicioniranje pre narednog ili iza poslednjeg deskriptora u listi uslova. Do pozicioniranja dolazi samo ako lista uslova nije prazna i ako je prethodno izvršeno pozicioniranje pre nekog deskriptora u ovoj listi. Poziv ove operacije vraća vrednost **true** samo kada, nakon pozicioniranja, iza tačke pozicioniranja postoji deskriptor. Tada parametar operacije **next()** omogućuje preuzimanje priveska tog deskriptora.

Operacija **last()** omogućuje pozicioniranje iza poslednjeg deskriptora u listi uslova. Do pozicioniranja dolazi samo ako lista uslova nije prazna i tada poziv ove operacije vraća vrednost **true**.

Izvršavanje operacije **wait()** nakon izvršavanja sekvence, koju sačinjavaju jedna ili više od prethodno opisane tri operacije, dovodi do uvezivanja deskriptora u tački, koju su odredila uzastopna pozicioniranja u toku izvršavanja pomenute sekvence operacija. Ako izvršavanju operacije **wait()** nije prethodilo pozicioniranje, tada se deskriptor niti pozivaoca ove operacije uvezuje na kraj liste uslova.

Podrazumeva se da operacija **notify_one()** klase **condition_variable** uvek izvezuje deskriptor sa početka liste uslova.

Optimizaciju kretanja glave diska omogućuje operacija **optimize()** klase **Disk**. U situaciji kada je disk slobodan (kada polje **state** klase **Disk** sadrži konstantu **FREE**), poziv operacije **optimize()** dovodi do poziva blokirajuće operacije **disk_driver.block_transfer()**. U toku njenog izvršavanja disk je zaposlen (polje **state** sadrži konstantu **BUSY**), a aktivnost pozivajuće niti je zaustavljena. Ako u ovoj situaciji više niti, jedna za drugom, pozove operaciju **optimize()**, njihova aktivnost se zaustavlja, a njihovi deskriptori se uvezuju u jednu od dve liste uslova, koje odgovaraju polju **q** klase **Disk**. Polje **index** ove klase indeksira ili listu uslova namenjenu za deskriptore niti koje čitaju blokove između trenutnog položaja glave diska i njegovog oboda ili listu uslova namenjenu za deskriptore niti koje čitaju blokove između centra rotacije ploče diska i trenutnog položaja njegove glave. Podatak o trenutnom položaju glave diska (odnosno, o bloku koji se upravo čita) sadrži polje **boundary** klase **Disk**. Polje **index** može imati vrednost 0 ili 1. U pomenutim listama uslova deskriptori su poređani u rastućem redosledu brojeva blokova koje niti čitaju (što je u skladu sa optimizacijom kretanja glave diska). Pomenuti brojevi blokova predstavljaju priveske deskriptora iz listi uslova.

Telo operacije **optimize()** sadrži dva isključiva regiona da bi poziv blokirajuće operacije **disk_driver.block_transfer()** bio van isključivog regiona.

Predložena optimizacija kretanja glave diska predviđa usluživanje zahteva samo u smeru kretanja glave diska od unutrašnjih ka vanjskim stazama. Pređeni put glave diska bi se još više skratio, kada bi se zahtevi usluživali u oba smera kretanja glave diska. To bi zahtevalo da skupovi zahteva budu uređeni na razne načine, jedan u rastućem, a drugi u opadajućem redosledu brojeva blokova, a usluživanje bi više čekali zahtevi vezani za krajnje staze.

Listing 5.6: Klasa **Disk** (datoteke **block_io.hh** i **block_io.cpp**)

```
const int
DISK_ERROR = -1;

class Disk {
   mutex mx;
   enum Optimized_disk_state { FREE, BUSY };
   Optimized_disk_state state;
   unsigned boundary;
   condition_variable q[2];
   int index;
   Disk(const Disk&);
   Disk& operator=(const Disk&);
   int optimize(char* buffer, unsigned block, Disk_operations operation);
  public:
   Disk() : state(FREE), boundary(0), index(0) {};
   int block_get(char* buffer, unsigned block);
   int block_put(char* buffer, unsigned block);
};
```

```
int
Disk::optimize(char* buffer, unsigned block, Disk_operations operation)
{
   unsigned tag;
   int i;
   int status;
   {
      unique_lock<mutex> lock(mx);
      if(state == BUSY) {
         i = index;
         if(block < boundary)
            i = ((i == 0) ? (1) : (0));
         if(q[i].first(&tag))
            do {
               if(block < tag)
                  break;
            } while(q[i].next(&tag));
         q[i].wait(lock, block);
      }
      state = BUSY;
      boundary = block;
   }
   status = disk_driver.block_transfer(buffer, block, operation);
   {
      unique_lock<mutex> lock(mx);
      state = FREE;
      if(!q[index].first())
         index = ((i == 0) ? (1) : (0));
      q[index].notify_one();
   }
   return status;
}

int
Disk::block_get(char* buffer, unsigned block)
{
   int status;
   status = optimize(buffer, block, DISK_READ);
   return status;
}
```

```
int
Disk::block_put(char* buffer, unsigned block)
{
   int status;
   status = optimize(buffer, block, DISK_WRITE);
   return status;
}

Disk
disk;
```

5.6 PITANJA

1. Na koje drajvere se oslanjaju ulazno-izlazni moduli CppTss-a?
2. Šta se dešava u obradi prekida ekrana?
3. Do čega dovodi pokušaj niti da prikaže novi znak dok kontroler ekrana prikazuje prethodni znak?
4. Šta se dešava u obradi prekida tastature?
5. Do čega dovodi pokušaj niti da preuzme znak kada je cirkularni bafer drajvera tastature prazan?
6. Šta se desi kada se napuni cirkularni bafer drajvera tastature?
7. Koje klase nasleđuje klasa **Terminal_out**?
8. Koje klase nasleđuje klasa **Terminal_in**?
9. Šta rade operacije klase **Terminal_out** za prikaz brojeva?
10. Šta rade operacije klase **Terminal_in** za preuzimanje brojeva?
11. Šta zaključava operacija **Terminal_in::string_get()**?
12. Šta otključava operacija **Terminal_in::string_get()**?
13. Šta zaključava operacija **Terminal_out::string_put()**?
14. Šta otključava operacija **Terminal_out::string_put()**?
15. Šta obuhvata primitivno editiranje koje podržava klasa **Terminal_in**?
16. Šta se desi u obradi prekida diska?
17. Operacije koje klase ulazno-izlaznih modula CppTss-a vraćaju poruku greške?

6 VIRTUELNA MAŠINA CppTss-A

6.1 ZADATAK VIRTUELNE MAŠINE CppTss-A

Razvojna verzija konkurentne biblioteke CppTss sadrži virtuelnu mašinu koja za potrebe ostatka ove biblioteke:

1. emulira mehanizam prekida,
2. emulira kontrolere tastature, ekrana i diska,
3. podržava okončanje izvršavanja konkurentnog programa,
4. podržava rukovanje pojedinim bitima memorijskih lokacija,
5. podržava rukovanje numeričkim koprocesorom (*Numeric Processor Extension - NPX*) i
6. podržava rukovanje stekom.

Emulacija mehanizma prekida se ostvaruje posredstvom periodičnog pokretanja emulacije kontrolera radi oponašanja rada kontrolera tastature, ekrana i diska. Sve sistemske biblioteke koje se koriste u emulaciji hardvera su pobrojane u Listingu 6.1.

Listing 6.1: Sistemske biblioteke korišćene u emulaciji hardvera (datoteka **includes.cpp**)

```
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <signal.h>
#include <sys/time.h>
#include <string.h>
#include <stdio.h>
```

Zahvaljujući emulaciji hardvera konkurentni program ne pristupa stvarnom, nego emuliranom hardveru, pa se izvršava kao običan (neprivilegovan) *Linux* proces. Zato, u slučaju grešaka, on ne može ugroziti funkcionisanje operativnog sistema u okviru koga se izvršava.

6.2 EMULACIJA MEHANIZMA PREKIDA

Emulacija mehanizma prekida se zasniva na: (1) uvođenju (emulirane) tabele prekida, (2) uvođenju (emuliranog) bita prekida i (3) obezbeđenju nezavisnosti (asinhronosti) između (emuliranih) prekida i izvršavanja konkurentnog programa.

Potrebe CppTss biblioteke su uzrokovale da (emulirana) tabela prekida sadrži pet elemenata. Prvi od njih je namenjen za vektor obrađivača hardverskog izuzetka (*FLOATING POINT EXCEPTION*), a drugi je rezervisan za vektor obrađivača prekida

sata. Preostala tri su predviđena za vektore obrađivača prekida tastature, ekrana i diska. Listing 6.6 sadrži brojeve vektora podržanih prekida.

Listing 6.6: Brojevi vektora podržanih prekida (datoteka **interrupts.hh**)

```
static const unsigned
INTERRUPT_TABLE_VECTOR_COUNT = 5;

enum
Vector_numbers { FP_EXCEPTION, TIMER, KEYBOARD, DISPLAY, DISK };
```

Svrha (emuliranog) bita prekida je da označi da li je ili nije omogućena obrada prekida. Emulacija bita prekida je ostvarena pomoću promenljive **interrupts_enabled** i funkcija **ad__disable_interrupt()** i **ad__restore_interrupts()** (Listing 6.7). Promenljiva **interrupts_enabled** sadrži (emulirani) bit prekida. Njena vrednost određuje da li su (emulirani) prekidi omogućeni (konstanta **true**) ili ne. Na ovu promenljivu utiču funkcije **ad__disable_interrupts()** i **ad__restore_interrupts()**. Prva od njih, izmenom (emuliranog) bita prekida, onemogućuje (emulirane) prekide, a druga poništava efekte prve, vraćajući (emulirani) bit prekida u prethodno stanje. Kada operacija **ad__restore_interrupts()** utvrdi da je došlo do odlaganja obrade (emuliranih) prekida (**Interrupt::pending**), ona pokreće prethodno odloženu emulaciju kontrolera pozivom operacije **controller_emulator()** klase **Interrupt**. Nakon toga se registruje da nema više odloženih obrada (emuliranih) prekida.

Listing 6.7: Emulacija bita prekida (datoteka **interrupts.hh**)

```
bool
interrupts_enabled = true;

inline static bool
ad__disable_interrupts()
{
   bool saved_interrupts_enabled = interrupts_enabled;
   interrupts_enabled = false;
   return saved_interrupts_enabled;
}
```

```
inline void
ad__restore_interrupts(bool saved_interrupts_enabled)
{
  if(saved_interrupts_enabled && interrupt.pending) {
        interrupt.controller_emulator();
        interrupt.pending = false;
  }
  interrupts_enabled = saved_interrupts_enabled;
}
```

Nezavisnost (emuliranih) prekida od izvršavanja konkurentnog programa se ostvaruje pomoću **mehanizma signala** *Linux*-a. Ovaj mehanizam omogućuje da se na pojavu signala reaguje izvršavanjem odabrane funkcije (*user level exception handling*). Ova funkcija se naziva obrađivač signala. Signali su unapred definisani, a svaki od njih je pridružen jednoj vrsti događaja, kao što je isticanje zadanog vremenskog intervala (**SIGVTALRM**) ili pojava hardverskog izuzetka (**SIGFPE**). Kada se takav događaj desi, mehanizam signala zaustavi izvršavanje programa (u toku koga se desio dotični događaj), radi pokretanja izvršavanja odgovarajućeg obrađivača signala. Nakon obrade dotičnog signala moguć je nastavak zaustavljenog izvršavanja programa. Obrađivač signala je funkcija koja opisuje korisničku reakciju na pojavu odabranog signala. Funkcija postaje obrađivač signala kada se poveže sa odgovarajućim signalom.

Pojava signala **SIGVTALRM** nije zavisna od izvršavanja konkurentnog programa. Zadatak obrađivača signala **SIGVTALRM** je da pozove emulaciju kontrolera i tako izazove obradu nekog od (emuliranih) prekida, a zadatak obrađivača signala **SIGFPE** je da izazove obradu izuzetka. Za razliku od obrade izuzetaka, koja se uvek obavlja bez odlaganja, obrada (emuliranih) prekida se obavezno odlaže ako su (emulirani) prekidi onemogućeni. Do obrade prethodno onemogućenog (emuliranog) prekida dolazi tek nakon omogućenja (emuliranih) prekida.

U slučaju konkurentnog programa, pojava signala potakne mehanizam signala da zaustavi zatečenu aktivnost niti i pokrene odgovarajućeg obrađivača signala. Ako u sklopu obrade (emuliranog) prekida, koju izazove ovaj obrađivač signala, dođe do preključivanja na drugu nit, započeta obrada signala će biti završena tek nakon ponovnog preključivanja na prethodno zaustavljenu nit. Da bi se u međuvremenu mogli obraditi novi signali, neophodno je da se razna izvršavanja obrađivača signala mogu preklapati. Za takve obrađivače signala se kaže da su višeulazni (*reentrant*).

Klasa **Linux_signals** (Listing 6.8) opisuje reakciju na *Linux* signale. Njen konstruktor koristi njeno polje **sa** i sistemski poziv **sigaction()** da saopšti da njena operacija **signal_handler()** ima ulogu obrađivača signala **SIGVTALRM** i **SIGFPE**. Ovaj konstruktor koristi konstantu **SA_NODEFER** (koju upisuje u polje **sa.sa_flags**) da saopšti da nema odlaganja obrada signala (da je obrađivač signala višeulazni). Destruktor klase **Linux_signals** poništava akcije njenog konstruktora. Obrađivač signala **Linux_signals::signal_handler()** u slučaju signala **SIGVTALRM** pozove emulaciju prekida

(**Interrupt::emulation()**), a u slučaju signala **SIGFPE** pozove obrađivača izuzetka (**Interrupt::handler()**) koji opslužuje hardverski izuzetak.

Listing 6.8: Klasa **Linux_signals** (datoteke **interrupt_classes.cpp** i **interrupt_classes_functions.cpp**)

```
class Linux_signals {
   struct sigaction sa;
   static void signal_handler(int signal);
   Linux_signals(const Linux_signals&);
   Linux_signals& operator=(const Linux_signals&);
  public:
   Linux_signals();
   ~Linux_signals();
};

void
Linux_signals::signal_handler(int signal)
{
   switch(signal) {
     case SIGVTALRM:
        interrupt.emulation();
        break;
     case SIGFPE:
        interrupt.handler(FP_EXCEPTION);
        break;
   }
}

Linux_signals::Linux_signals()
{
   sa.sa_handler=signal_handler;
   sigemptyset(&(sa.sa_mask));
   sa.sa_flags=SA_NODEFER;
   sigaction(SIGVTALRM, &sa, 0);
   sigaction(SIGFPE,    &sa, 0);
}
```

```
Linux_signals::~Linux_signals()
{
   sa.sa_handler = SIG_DFL;
   sigaction(SIGVTALRM, &sa, 0);
   sigaction(SIGFPE,    &sa, 0);
}

Linux_signals
linux_signals;
```

Konstruktor klase **Linux_timer** (Listing 6.9) obezbedi da se signal **SIGVTALRM** dešava svakih 10 milisekundi. Za to on koristi polje **itimer** i sistemski poziv **setitimer()**. Destruktor ove klase poništava akciju njenog konstruktora.

Listing 6.9: Klasa **Linux_timer** (datoteke **interrupt_classes.cpp** i **interrupt_classes_functions.cpp**)

```
const int
LINUX_TIMER_INTERVAL = 10;

class Linux_timer {
   struct itimerval itimer;
   Linux_timer(const Linux_timer&);
   Linux_timer& operator=(const Linux_timer&);
  public:
   Linux_timer();
   ~Linux_timer();
};

Linux_timer::Linux_timer()
{
   itimer.it_interval.tv_sec = 0;
   itimer.it_interval.tv_usec = LINUX_TIMER_INTERVAL * 1000;
   itimer.it_value.tv_sec = 0;
   itimer.it_value.tv_usec = LINUX_TIMER_INTERVAL * 1000;
   setitimer(ITIMER_VIRTUAL, &itimer, 0);
}
```

```
Linux_timer::~Linux_timer()
{
   itimer.it_value.tv_sec = 0;
   itimer.it_value.tv_usec = 0;
   setitimer(ITIMER_VIRTUAL, &itimer, 0);
}

static Linux_timer
linux_timer;
```

Klasa **Interrupt (**Listing 6.10**)**:

1. uvodi (emuliranu) tabelu prekida (sadržanu u nizu **vector**),
2. omogućuje registrovanje odlaganja obrade (emuliranog) prekida (polje **pending**) i
3. reguliše redosled pozivanja obrađivača pojedinih (emuliranih) prekida (polja **controller_turn i timer_turn**).

(Emulirana) tabela prekida se inicijalizuje tako da njeni elementi sadrže vektor podrazumevajućeg obrađivača prekida: **default_interrupt_handler()**. Operacija **handler()** klase **Interrupt** posreduje u pozivu obrađivača (emuliranog) prekida, a operacija **emulation()** registruje odlaganje obrade prekida ili pokreće emulaciju kontrolera pozivom operacije **controller_emulator()**. U svakoj parnoj emulaciji kontrolera poziva se obrađivač (emuliranog) prekida sata (sa brojem vektora **TIMER**), a u svakoj neparnoj emulaciji kontrolera obavlja se, u kružnom redosladu, emulacija samo jednog od kontrolera (**display_controller.output()**, **keyboard_controller.input()** ili **disk_controller.transfer()**). U okviru emulacije pojedinih kontrolera poziva se obrađivač odgovarajućeg (emuliranog) prekida.

Operacija **ad_set_vector()** omogućuje izmenu vektora prekida.

Listing 6.10: Klasa **Interrupt** (datoteke **interrupt_classes.cpp** i **interrupt_classes_functions.cpp**)

```
class Interrupt {
   static void (*vector[INTERRUPT_TABLE_VECTOR_COUNT])();
   bool pending;
   int controller_turn;
   bool timer_turn;
   Interrupt(const Interrupt&);
   Interrupt& operator=(const Interrupt&);
 public:
   Interrupt();
   inline void handler(unsigned index);
   inline void emulation();
   inline void controller_emulator();
   friend inline void ad__restore_interrupts(bool new_interrupt_status);
   friend inline void ad__set_vector(int index, void (*handler)());
};

void
default_interrupt_handler()
{
}

void
(*Interrupt::vector[INTERRUPT_TABLE_VECTOR_COUNT])()
   = {default_interrupt_handler};

Interrupt::Interrupt()
   : pending(false), controller_turn(0), timer_turn(true)
{
}

void
Interrupt::handler(unsigned index)
{
   (vector[index])();
}
```

```
void
Interrupt::emulation()
{
   if(!interrupts_enabled)
      interrupt.pending = true;
   else {
      interrupts_enabled = false;
      controller_emulator();
      interrupts_enabled = true;
   }
}

void
Interrupt::controller_emulator()
{
   bool interrupt_emulated;
   int counter = 0;
   if(timer_turn) {
      timer_turn = false;
      handler(TIMER);
   } else {
               timer_turn = true;
          do {
             switch(controller_turn) {
              case 0:
                     interrupt_emulated = display_controller.output();
                     controller_turn = 1;
                     break;
              case 1:
                     interrupt_emulated = keyboard_controller.input();
                     controller_turn = 2;
                     break;
              case 2:
                     interrupt_emulated = disk_controller.transfer();
                     controller_turn = 0;
                     break;
             }
          } while(!interrupt_emulated && ++counter < 3);
   }
}
```

```
inline void
ad__set_vector(int index, void (*handler)())
{
   Interrupt::vector[index] = handler;
}

Interrupt
interrupt;
```

6.3 EMULACIJA KONTROLERA

Klasa **Keyboard_controller** (Listing 6.2) opisuje kontroler tastature. Njeno polje **data_reg** predstavlja registar podataka kontrolera, a njena operacija **input()** opisuje ponašanje kontrolera tastature. Ako postoji znak za preuzimanje, on se preuzima u okviru operacije **input()** posredstvom odgovarajućeg *Linux* sistemskog poziva. Ujedno se poziva obrađivač prekida tastature posredstvom operacije **handler()**:

```
interrupt.handler(KEYBOARD)
```

klase **Interrupt** koja emulira tabelu prekida. Operacija **input()** se periodično poziva u toku emulacije kontrolera.

Listing 6.2: Klasa **Keyboard_controller** (datoteke **controllers.hh** i **controllers.cpp**)

```
class Keyboard_controller {
   char data_reg;
   bool input();
   Keyboard_controller(const Keyboard_controller&);
   Keyboard_controller& operator=(const Keyboard_controller&);
public:
   Keyboard_controller() {};
   friend class Interrupt;
   friend class Keyboard_driver;
};
```

```
bool
Keyboard_controller::input()
{
   bool interrupt_emulated = false;
   unsigned read_count;
   char c;
   read_count = read(STDIN_FILENO, &c, 1);
   if(read_count > 0) {
      data_reg = c;
      interrupt.handler(KEYBOARD);
      interrupt_emulated = true;
   }
   return interrupt_emulated;
}

Keyboard_controller
keyboard_controller;
```

Klasa **Display_controller** (Listing 6.3) opisuje kontroler ekrana. Njena polja **data_reg** i **status_reg** predstavljaju registre podataka i stanja kontrolera, a njena operacija **output()** opisuje ponašanje kontrolera ekrana. Ako postoji znak za prikazivanje, on se prikazuje u okviru operacije **output()** posredstvom odgovarajućeg *Linux* sistemskog poziva. Ujedno se poziva obrađivač prekida ekrana posredstvom operacije **handler()**:

```
interrupt.handler(DISPLAY)
```

klase **Interrupt** koja emulira tabelu prekida. Operacija **output()** se periodično poziva u toku emulacije kontrolera.

Listing 6.3: Klasa **Display_controller** (datoteke **controllers.hh** i **controllers.cpp**)

```
enum
Display_status { DISPLAY_READY, DISPLAY_BUSY };
```

```
class Display_controller {
   char data_reg;
   Display_status status_reg;
   bool output();
   Display_controller(const Display_controller&);
   Display_controller& operator=(const Display_controller&);
  public:
   Display_controller() : status_reg(DISPLAY_READY) {};
   friend class Interrupt;
   friend class Display_driver;
};

bool
Display_controller::output()
{
   bool interrupt_emulated = false;
   if(status_reg == DISPLAY_BUSY) {
      write(STDOUT_FILENO, &data_reg, 1);
      status_reg = DISPLAY_READY;
      interrupt.handler(DISPLAY);
      interrupt_emulated = true;
   }
   return interrupt_emulated;
}

Display_controller
display_controller;
```

Prethodne dve klase koriste tastaturu i ekran *Linux* terminala. Za potrebe emulacije neophodno je isključiti eho (*echo*) *Linux* terminala, prevesti *Linux* terminal u režim rada bez linijskog editiranja (*raw mode*) i obezbediti da poziv čitanja znaka sa terminala bude neblokirajući. Sve prethodno obezbeđuje konstruktor klase **Linux_terminal** (Listing 6.4), koji koristi polje **ots** ove klase da sačuva zatečeni režim rada *Linux* terminala, a **ts** polje da zada njegov novi režim rada. Prethodno zatečeni režim rada *Linux* terminala ponovo uspostavlja destruktor klase **Linux_terminal**. Ovaj destruktor se poziva na kraju aktivnosti procesa i radi provere da li postoje niti za koje nije regularno da kraj njihove aktivnosti nastupi kao posledica kraja aktivnosti procesa kome one pripadaju i radi obaveštenja o prevremenom završetku ovakvih niti.

Listing 6.4: Klasa **Linux_terminal** (datoteka **controllers.cpp**)

```
class Linux_terminal {
   struct termios ts;
   struct termios ots;
   Linux_terminal(const Linux_terminal&);
   Linux_terminal& operator=(const Linux_terminal&);
  public:
   Linux_terminal();
   ~Linux_terminal();
};

Linux_terminal::Linux_terminal()
{
   tcgetattr(STDIN_FILENO, &ts);
   ots = ts;
   ts.c_lflag &= ~ECHO;
   ts.c_lflag &= ~ICANON;
   ts.c_cc[VTIME] = 0;
   ts.c_cc[VMIN] = 0;
   tcsetattr(STDIN_FILENO, TCSANOW, &ts);
}

Linux_terminal::~Linux_terminal()
{
   if(undetached_threads())
      write(STDOUT_FILENO, &"\nERROR: ABORTION OF UNDETACHED
            THREADS!\n", 40);
   else
      write(STDOUT_FILENO, &"\n", 1);
   tcsetattr(STDIN_FILENO, TCSANOW, &ots);
}

static Linux_terminal
linux_terminal;
```

Klasa **Disk_controller** (Listing 6.5) opisuje ponašanje kontrolera diska. Njena polja **operation_reg**, **buffer_reg**, **block_reg** i **status_reg** odgovaraju registrima smera prenosa, bafera, bloka i stanja kontrolera, a njena operacija **transfer()** opisuje ponašanje kontrolera diska. Konstruktor klase **Disk_controller** koristi sistemski poziv **calloc()** radi zauzimanja radne memorije, u kojoj se čuvaju blokovi emuliranog diska. Inercija diska

se emulira brojanjem poziva operacije **transfer()**. Kada broj poziva ove operacije postane jednak procenjenom broju vremenskih jedinica, potrebnom za prenos bloka, tada se obavi prenos bloka u okviru ove operacije i ujedno se pozove obrađivač prekida diska posredstvom operacije **handler()**:

```
interrupt.handler(DISK)
```

klase **Interrupt** koja emulira tabelu prekida. Operacija **transfer()** se periodično poziva u toku emulacije kontrolera.

Listing 6.5: Klasa **Disk_controller** (datoteke **controllers.hh** i **controllers.cpp**)

```
enum
Disk_operations { DISK_READ, DISK_WRITE };

enum
Disk_status { DISK_STARTED, DISK_ACTIVE, DISK_STOPPED };

const unsigned
BLOCK_SIZE = 512;

const unsigned
DISK_BLOCKS = 1000;

typedef char
Disk_block[BLOCK_SIZE];
```

```
class Disk_controller {
   Disk_block* disk_space;
   unsigned accessed_last;
   unsigned transfer_time;
   Disk_operations operation_reg;
   char* buffer_reg;
   unsigned block_reg;
   Disk_status status_reg;
   bool transfer();
   Disk_controller(const Disk_controller&);
   Disk_controller& operator=(const Disk_controller&);
  public:
   Disk_controller();
   ~Disk_controller();
   friend class Interrupt;
   friend class Disk_driver;
};

Disk_controller::Disk_controller()
   : accessed_last(0), transfer_time(0), status_reg(DISK_STOPPED)
{
   disk_space = (Disk_block*) calloc(DISK_BLOCKS, sizeof(Disk_block));
}

Disk_controller::~Disk_controller()
{
   free(disk_space);
}

const int
SECTORS_PER_TRACK = 10;

const int
TRANSFER_TIME_AND_ROTATIONAL_DELAY = 2;
```

```
bool
Disk_controller::transfer()
{
   bool interrupt_emulated = false;
   Disk_block* block_pointer;
   int block_distance;
   if(status_reg == DISK_STARTED) {
      status_reg = DISK_ACTIVE;
      block_distance = accessed_last - block_reg;
      if(block_distance < 0)
         block_distance = -block_distance;
      transfer_time = TRANSFER_TIME_AND_ROTATIONAL_DELAY;
      transfer_time += block_distance / SECTORS_PER_TRACK;
      accessed_last = block_reg;
   }
   if((status_reg == DISK_ACTIVE) && (--transfer_time == 0)) {
      block_pointer = disk_space + block_reg;
      if(operation_reg == DISK_WRITE)
         bcopy(buffer_reg, block_pointer, BLOCK_SIZE);
      else
         bcopy(block_pointer, buffer_reg, BLOCK_SIZE);
      status_reg = DISK_STOPPED;
      interrupt.handler(DISK);
      interrupt_emulated = true;
   }
   return interrupt_emulated;
}

Disk_controller
disk_controller;
```

6.4 OKONČANJE IZVRŠAVANJA KONKURENTNOG PROGRAMA

Izvršavanje konkurentnog programa se okončava sistemskim pozivom **exit()**. To omogućuje funkcija **ad__report_and_finish()** (Listing 6.11).

Listing 6.11: Okončanje izvršavanja konkurentnog programa (datoteka **end_functions.cpp**)

```
inline void
ad__report_and_finish(const char* message_string)
{
   int length = 0;
   while(message_string[length] != 0)
         length++;
   write(STDERR_FILENO, message_string, length);
   write(STDERR_FILENO, "\n", 1);
   exit(1);
}
```

6.5 RUKOVANJE POJEDINIM BITIMA MEMORIJSKIH LOKACIJA

Emulacija hardvera je namenjena za platforme zasnovane na i386 (i novijim) procesorima koji podržavaju asemblerske naredbe za dobijanje indeksa najznačajnijeg postavljenog bita u reči: **bsr**, za postavljanje datog bita reči: **bts** i za njegovo čišćenje: **btr**. Funkcije **ad__get_index_of_most_signifcant_set_bit()**, **ad__set_bit()** i **ad__clear_bit()** posreduju u korišćenju ovih asemblerskih naredbi (Listing 6.12).

Listing 6.12: Rukovanje pojedinim bitima memorijskih lokacija (datoteka **bit_functions.cpp**)

```
inline static int
ad__get_index_of_most_significant_set_bit(unsigned priority_bits)
{
   int index;
   asm ("bsr %1, %0"
         : "=r"(index)
         : "r"(priority_bits)
        );
   return index;
}
```

```
inline static unsigned
ad__set_bit(unsigned priority_bits, int index)
{
  asm ("bts %1, %0"
       : "=r"(priority_bits)
       : "r"(index), "0"(priority_bits)
      );
  return priority_bits;
}

inline static int
ad__clear_bit(unsigned priority_bits, int index)
{
  asm ("btr %1, %0"
       : "=r"(priority_bits)
       : "r"(index), "0"(priority_bits)
      );
  return priority_bits;
}
```

Listing 6.12 pokazuje da se asemblerske naredbe zadaju između malih zagrada koje slede iza rezervisane reči **asm**. Da bi kompajler na najbolji način uključio ovako zadane asemblerske naredbe u generisani kod, kompajleru se prepušta da oblikuje operande pomenutih asemblerskih naredbi. Zato se u ovim asemblerskim naredbama umesto operanada koriste redni brojevi operanada. Podrazumeva se da ovim rednim brojevima prethodi znak procenta (**%**). Opis operanada se navodi iza asemblerskih naredbi. Prvo se navodi opis izlaznih, a onda se navodi opis ulaznih operanada. Tako, za jedinu asemblersku naredbu iz prve funkcije važi da izlazni operand sa rednim brojem **%0** odgovara lokalnoj promenljivoj **index**, a ulazni operand sa rednim brojem **%1** odgovara parametru **priority_bits**. Kod druge i treće funkcije za jedinu asemblersku naredbu važi da izlazno-ulazni operand sa rednim brojem **%0** odgovara parametru **priority_bits**, a ulazni operand sa rednim brojem **%1** odgovara parametru **index**.

6.6 RUKOVANJE NUMERIČKIM KOPROCESOROM

Preključivanje procesora sa jedne niti na drugu podrazumeva da se sačuva kontekst (sadržaj registara) prve niti i uspostavi kontekst druge niti. Kontekst se čuva na steku niti i obuhvata i registre numeričkog koprocesora. Klasa **I387_npx** (Listing 6.13) omogućuje pripremu i preuzimanje inicijalnog sadržaja registara numeričkog koprocesora. Konstruktor klase **I387_npx** inicijalizuje registre numeričkog koprocesora i smešta njihov inicijalni sadržaj u polje **initial_context** ove klase pomoću asemblerskih naredbi **fninit** i **fnsave**. Jedini ulazni operand druge asemblerske naredbe je pokazivač na **initial_context**. Operacija **initial_context_get()** klase **I387_npx** omogućuje preuzimanje

inicijalnog sadržaja registara numeričkog koprocesora.

Listing 6.13: Rukovanje numeričkim koprocesorom (datoteka **i387_npx.cpp**)

```
const unsigned
i387_SAVE_REGION_SIZE = 0x6c;

class I387_npx {
   static char initial_context[i387_SAVE_REGION_SIZE];
   I387_npx(const I387_npx&);
   I387_npx& operator=(const I387_npx&);
 public:
   I387_npx();
   inline void initial_context_get(Stack_item* stack_top);
};

char
I387_npx::initial_context[i387_SAVE_REGION_SIZE] = {0};

I387_npx::I387_npx() {
   asm volatile(
         "fninit      \n\t"
         "fnsavel %0   \n\t"
         :
         : "m"(*initial_context)
   );
}

void
I387_npx::initial_context_get(Stack_item* stack_top) {
   for(unsigned i = 0; i < i387_SAVE_REGION_SIZE; i++)
            ((char*)stack_top)[i] = initial_context[i];
}

static I387_npx
i387_npx;
```

6.7 RUKOVANJE STEKOM

Preključivanje procesora sa jedne niti na drugu se temelji na izmeni važećeg steka. Ova izmena je posledica promene sadržaja registra koji služi kao pokazivač steka i dešava se u toku izvršavanja funkcije preključivanja. Znači, pre poziva funkcije preključivanja koristi se stek prve niti, na koji dospeva povratna adresa prve niti. U toku

izvršavanja funkcije preključivanja dođe do izmene važećeg steka, pa se nakon toga koristi stek druge niti, na kome je ranije pripremljena povratna adresa druge niti. Tako funkciju preključivanja pozove jedna nit, a funkcija se vraća u drugu nit. Za uspeh preključivanja je neophodno da funkcija preključivanja na steku druge niti zatekne ispravan frejm (u istom obliku u kome je ona ostavila frejm na steku prve niti). To je obezbeđeno kada se procesor preključuje na nit koja je već bila aktivna. Ali, ako se procesor prvi put preključuje na neku nit, on na njenom steku mora zateći frejm sa ranije pripremljenim njenim inicijalnim kontekstom.

Podrazumeva se da prvo preključivanje na neku nit dovodi do početka izvršavanja funkcije koja opisuje dotičnu nit. Da bi izvršavanje ove funkcije bilo moguće, neophodno je na steku niti pripremiti frejm njenog poziva sa odgovarajućom povratnom adresom. Kao povratna adresa služi adresa funkcije **destroy()**. Tako se, prilikom prirodnog završetka aktivnosti niti, automatski poziva njeno uništenje. Do izvršavanja funkcije koja opisuje nit dolazi nakon povratka iz funkcije preključivanja, ako se na steku niti pripremi i frejm poziva funkcije preključivanja u kome se kao povratna adresa koristi adresa funkcije koja opisuje nit. Pomenuta dva frejma (frejm poziva funkcije koja opisuje nit i frejm poziva funkcije preključivanja) na steku stvarane niti pripremi funkcija **ad__stack_init** (Listing 6.14).

Listing 6.14: Inicijalizacija steka (datoteka **stack.cpp**)

```
const int
INITIAL_FLAGS = 0x0200;

static inline void
ad__stack_init(Stack_item** stack_top, unsigned thread_function)
{
   *(--(*stack_top)) = (Stack_item) destroy;
   *(--(*stack_top)) = (Stack_item) thread_function;
   *(--(*stack_top)) = (Stack_item) 0;
   *(--(*stack_top)) = (Stack_item) true;
   *(--(*stack_top)) = (Stack_item) INITIAL_FLAGS;
   *(--(*stack_top)) = (Stack_item) 0;
   *(--(*stack_top)) = (Stack_item) 0;
   *(--(*stack_top)) = (Stack_item) 0;
   *(--(*stack_top)) = (Stack_item) 0;
   *(--(*stack_top)) = (Stack_item) 0;
   *(--(*stack_top)) = (Stack_item) 0;
   *(--(*stack_top)) = (Stack_item) 0;
   *(--(*stack_top)) = (Stack_item) 0;
   *stack_top = (Stack_item*)(((size_t)(*stack_top)) - i387_SAVE_REGION_SIZE);
   i387_npx.initial_context_get(*stack_top);
}
```

```
extern "C"
void
ad__stack_swap(Stack_item** const old_stack, const Stack_item* new_stack);
```

Funkcija **ad__stack_init** na stek smesti: (1) adresu funkcije **destroy()** kao povratnu adresu funkcije koja opisuje stvaranu nit, (2) adresu funkcije **thread_function()**) kao povratnu adresu funkcije preključivanja i (3) kontekst niti na koju se procesor prvi put preključuje. Ovaj kontekst obuhvata pokazivač prethodnog frejma, početno stanje (emuliranog) bita prekida, početno stanje status (**flag**) registra procesora i početni sadržaj registara procesora. Kao pokazivač prethodnog frejma se koristi 0, jer nema prethodnog frejma. Početno stanje (emuliranog) bita prekida i status registra određuju konstante **true** i **INITIAL_FLAGS**. Registri procesora obuhvataju registre opšte namene i registre njegovog numeričkog koprocesora. Kao početni sadržaj registara opšte namene koriste se 0, dok inicijalni sadržaj numeričkog koprocesora obezbeđuje poziv operacije **initial_context_get()** klase **I387_npx**.

Funkcija **ad__stack_swap()** ima ulogu funkcije preključivanja. Pošto je ona napisana asemblerskim jezikom (Listing 6.15), radi provere ispravnosti njenih poziva uvedena je njena C deklaracija (Listing 6.14).

Listing 6.15: Funkcija preključivanja (datoteka **stack_swap.s**)

```
    .text
    .align  2
.globl ad__stack_swap

ad__stack_swap:
    pushl   %ebp
    movl    %esp,%ebp
    movl    8(%ebp),%edx

    mov     interrupts_enabled,%eax
    push    %eax
    pushf
    pusha
    sub     i387_SAVE_REGION_SIZE,%esp
    fnsavel (%esp)
    mov     %esp,(%edx)
    mov     12(%ebp),%esp
    frstorl (%esp)
    add     i387_SAVE_REGION_SIZE,%esp
    popa
    popf
    pop     %eax
    mov     %eax,interrupts_enabled

    popl    %ebp
    ret
```

Iz C deklaracije funkcije **ad__stack_swap()** (Listing 6.14) sledi da kao prvi argument njenog poziva služi adresa lokacije sa adresom vrha steka iz deskriptora niti sa koje se procesor preključuje. Kao drugi argument njenog poziva služi adresa vrha steka iz deskriptora niti na koju se procesor preključuje.

Funkcija preključivanja prvo na stek aktivne niti smesti njen kontekst. Zatim, ona kao važeći stek postavi stek novoaktivirane niti. Na kraju ona sa ovog steka preuzme kontekst novoaktivirane niti.

Funkcija preključivanja sačuva zatečeni pokazivač frejma iz registra **%ebp** (prva asemblerska naredba), postavi novi pokazivač frejma (druga asemblerska naredba) i u registar **%edx** smesti adresu lokacije sa adresom vrha steka iz deskriptora niti sa koje se procesor preključuje (treća asemblerska naredba). Ona zatim na stek aktivne niti smesti zatečeno stanje (emuliranog) bita prekida (**interrupts_enabled** - Listing 6.7), pa zatečeni

sadržaj status (**flag**) registra (**pushf**), pa zatečeni sadržaj registara opšte namene (**pusha**) i zatečeni sadržaj registara numeričkog koprocesora (**fnsavel**). Potom se trenutni sadržaj pokazivača steka (adresa vrha steka) smesti u deskriptor do tada aktivne niti u lokaciju koju pokazuje registar **%edx**. Zatim se u pokazivač steka prebaci adresa vrha steka iz deskriptora novoaktivirane niti. Na kraju se sa novog steka preuzmu novi sadržaji registara numeričkog koprocesora (**frstorl**), registara opšte namene (**popa**), status registra (**popf**), (emuliranog) bita prekida (**interrupts_enabled**) i frejm pointera. Time se priključivanje završava.

Funkcija preključivanja je napisana asemblerskim jezikom, jer koristi arhitekturu naredbi procesora na način koji nije dostupan iz programskih jezika visokog nivoa.

6.8 PITANJA

1. Koji zadatak ima virtuelna mašina CppTss-a?
2. Šta omogućuje emulacija hardvera biblioteci CppTss?
3. Na čemu se zasniva emulacija mehanizma prekida?
4. Šta važi za mehanizam *Linux* signala?
5. Šta omogućuje klasa **Linux_signal**?
6. Šta omogućuje klasa **Linux_timer**?
7. Šta podržava klasa **Interrupt**?
8. Koja funkcija pokreće odloženu obradu (emuliranih) prekida?
9. Šta omogućuje klasa **Keyboard_controller**?
10. Šta omogućuje klasa **Display_controller**?
11. Šta omogućuje klasa **Disk_controller**?
12. Šta omogućuje klasa **Linux_terminal**?
13. Koja klasa omogućuje prevođenje *Linux* terminala u režim rada bez linijskog editiranja?
14. Koja klasa omogućuje isključivanje eha *Linux* terminala?
15. Koja klasa obezbeđuje da čitanje znakova sa *Linux* terminala bude neblokirajuće?
16. Kojim *Linux* sistemskim pozivom se okončava zvršavanje konkurentnog programa?
17. Šta omogućuje rukovanje pojedinim bitima memorijskih lokacija?
18. Šta omogućuje rukovanje numeričkim koprocesorom?
19. Šta omogućuje rukovanje stekom?
20. Šta obuhvata kontekst niti?

7 CppTss IZVRŠILAC

7.1 DELOVI CppTss IZVRŠIOCA

Deo CppTss-a koji ima ulogu jezgra operativnog sistema se naziva izvršilac. Funkcionalna sličnost CppTss izvršioca sa operativnim sistemom ima za posledicu sličnost njihovih izvedbi. Znači, struktura CppTss izvršioca se može predstaviti pomoću istih slojeva kao i struktura operativnog sistema. Ključna razlika je da CppTss izvršilac ne sadrži modul za rukovanje datotekama, jer CppTss ne podržava pojam datoteke (Slika 7.1).

sistemske niti	**thread_wake_up_deamon()** **thread_destroyer_deamon()** **thread_zero()** klasa **Delta**
modul za rukovanje procesima	klasa **thread** klasa **Thread_image**
modul za rukovanje datotekama	-
modul za rukovanje radnom memorijom	klasa **Memory_fragment**
modul za rukovanje kontrolerima	klasa **Timer_driver** klasa **Exception_driver** klasa **Driver**
modul za rukovanje procesorom	klasa **condition_variable** klasa **unique_lock** klasa **mutex** klasa **Kernel** klasa **Atomic_region** klasa **Ready_list** klasa **Descriptor** klasa **Permit** klasa **List_link** klasa **Failure**

Slika 7.1: Slojevi CppTss izvršioca

U prvoj koloni (Slika 7.1) su navedeni moduli operativnog sistema, a u drugoj koloni se nalaze imena klasa i funkcija koje sadrže pojedini moduli CppTss izvršioca. Na vrhu hijerarhije slojeva se nalaze sistemske niti, koje ulaze u sastav CppTss izvršioca.

U nastavku se opisuju moduli CppTss izvršioca i to u redosledu odozdo na gore.

7.2 KLASA Failure

Klasa **Failure** (Listing 7.1) omogućuju rukovanje izuzecima (greškama), nastalim u

toku izvršavanja konkurentnog programa. Svaki objekat klase **Failure** sadrži: (1) string sa imenom odgovarajućeg izuzetka i (2) numeričku oznaku (kod) izuzetka. Operacije **name()** i **code()** ove klase omogućuju preuzimanje odgovarajućeg imena i koda izuzetka.

Listing 7.1: Klasa **Failure** i njeni objekti (datoteke **failure.hh** i **failure.cpp**)

```
enum
Failure_codes { MEMORY_SHORTAGE, NOTIFY_OUTSIDE_EXCLUSIVE_REGION };

class Failure {
 protected:
  const char* f_name;
  Failure_codes f_code;
 public:
  Failure(const char* fname, const Failure_codes fcode)
    : f_name(fname), f_code(fcode) {};
  inline const char* name() const { return f_name; };
  inline Failure_codes code() const { return f_code; };
};

Failure
failure_memory_shortage("MEMORY SHORTAGE!",
                        MEMORY_SHORTAGE);
Failure
failure_notify_outside_exclusive_region("NOTIFY OUTSIDE EXCLUSIVE REGION!",
                                        NOTIFY_OUTSIDE_EXCLUSIVE_REGION);
```

7.3 KLASA List_link

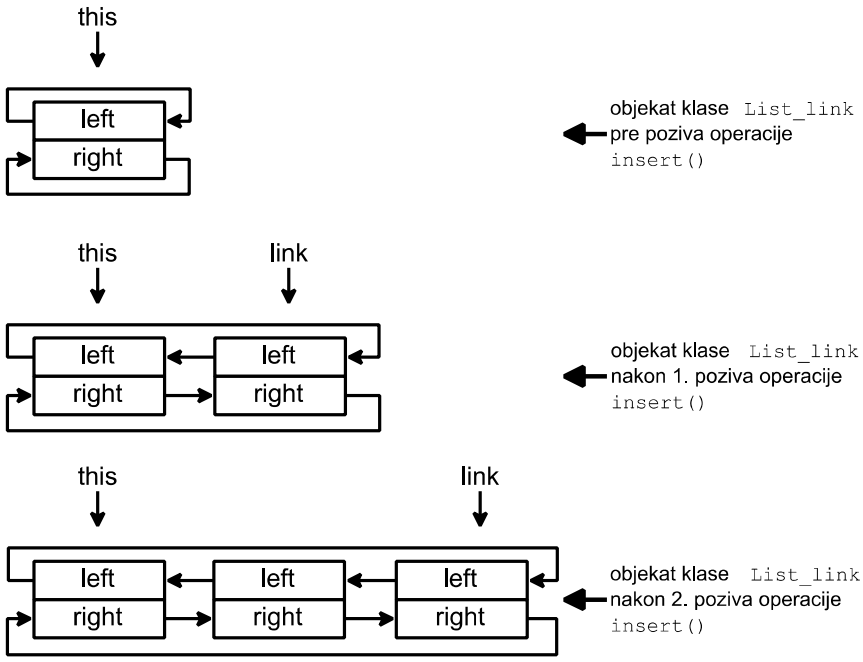
Klasa **List_link** (Listing 7.2) omogućuje obrazovanje dvosmerne cirkularne liste. Nju predstavlja objekat ove klase, koji tada istovremeno služi kao njen početak i kraj. Podrazumeva se da ovakva lista kao svoje elemente sadrži objekte klase **List_link**. Takođe se podrazumeva da se oni uvezuju na njen kraj i da se izvezuju sa njenog početka. Dvosmerna cirkularna lista se obrazuje pomoću polja **left** i **right**. Operacije klase **List_link** omogućuju preuzimanje vrednosti ovih polja: **left_get()**, **right_get()**, uvezivanje novog elementa na kraj liste: **insert()**, izvezivanje elementa sa početka liste: **extract()**, proveru da li u listi ima uvezanih elemenata: **not_empty()**, odnosno proveru da li je lista prazna: **empty()**.

Za deljene promenljive, nastale na osnovu klase **List_link**, se podrazumeva da je na svim mestima njihovog korišćenja na odgovarajući način zaštićena njihova konzistentnost.

Listing 7.2: Klasa **List_link** (datoteke **list.hh** i **list.cpp**)

```
class List_link {
   List_link* left;
   List_link* right;
   List_link(const List_link&);
   List_link& operator=(const List_link&);
  public:
   List_link() { right = this; left = this; };
   List_link* left_get() const { return left; };
   List_link* right_get() const { return right; };
   void insert(List_link* const link);
   List_link* extract();
   bool empty() const { return (this == right); };
   bool not_empty() const { return !empty(); };
};

void
List_link::insert(List_link* const link)
{
   link->left = left;
   link->right = this;
   left->right = link;
   left = link;
}

List_link*
List_link::extract()
{
   List_link* p = right;
   right->right->left = this;
   right = right->right;
   return p;
}
```

Slika 7.2 sadrži rezultate uzastopnih poziva operacije **insert()** objekta klase **List_link**.



Slika 7.2: Rezultati poziva operacije **insert()** objekta klase **List_link**

7.4 KLASA Permit

Klasa **Permit** (Listing 7.3) opisuje rukovanje propusnicama. Polje **free** klase **Permit** čuva stanje propusnice. Pokazivačko polje **previous** klase **Permit** omogućuje obrazovanje liste propusnica. Ova lista je vezana za nit koja je dobila pomenute propusnice. Propusnice se uvezuju u ovu listu u redosledu u kome ih je nit dobila, a izvezuju iz nje u obrnutom redosledu, jer se u obrnutom redosledu propusnice vraćaju. Oko polja **admission_list** klase **Permit** se obrazuje lista deskriptora niti koje čekaju na propusnicu da bi ušle u isključivi region, a oko njenog polja **fulfiled_list** se obrazuje lista deskriptora niti koje čekaju na propusnicu nakon ispunjenja uslova.

Operacije klase **Permit** omogućuju proveru da li je propusnica isključive promenljive zauzeta: **not_free()**, zauzimanje ove propusnice: **take()**, njeno oslobađanje: **release()** i rukovanje listama deskriptora niti: **admission_insert()**, **admission_extract()**, **admission_not_empty()**, **fulfilled_insert()**, **fulfilled_extract()**, **fulfilled_not_empty()**.

Poziv operacije provere da li je propusnica zauzeta i poziv operacije zauzimanja propusnice moraju da budu u istom atomskom regionu, inače se može desiti da više niti, jedna za drugom, proverom ustanovi da je ista propusnica slobodna i da zatim, jedna za drugom, zauzme istu propusnicu. Pozivi operacija za rukovanje listama deskriptora niti moraju biti u atomskom regionu, radi zaštite konzistentnosti ovih listi.

Listing 7.3: Klasa **Permit** (datoteke **permit.hh** i **permit.cpp**)

```
class Permit {
   bool free;
   Permit* previous;
   List_link admission_list;
   List_link fulfilled_list;
 public:
   Permit() { free = true; previous = 0; };
   inline bool not_free() const { return(free == false); };
   inline void take() { free = false; };
   inline void release() { free = true; };
   inline void admission_insert(List_link* link)
      { admission_list.insert(link); };
   inline void fulfilled_insert(List_link* link)
      { fulfilled_list.insert(link); };
   inline List_link* admission_extract()
      { return admission_list.extract(); };
   inline List_link* fulfilled_extract()
      { return fulfilled_list.extract(); };
   inline bool admission_not_empty()
      { return admission_list.not_empty(); };
   inline bool fulfilled_not_empty()
      { return fulfilled_list.not_empty(); };
   friend class Descriptor;
};
```

7.5 KLASA Descriptor

Klasa **Descriptor** (Listing 7.4) određuje deskriptor niti. Ona nasleđuje klasu **List_link**, da bi bilo moguće deskriptore niti uvezivati u liste. Polje **stack_top** klase **Descriptor** sadrži pokazivač (adresu) vrha steka niti. Prioritet niti je sadržan u polju **priority** ove klase. Polje **last** klase **Descriptor** sadrži adresu poslednje dobijene propusnice. Polje **tag** ove klase je namenjeno za smeštanje priveska deskriptora niti.

Klasa **Descriptor** nudi operacije za pristup nekim od njenih polja: **tag_get()**, **tag_set()**, kao i operacije za uvezivanje propusnice u listu propusnica niti prilikom njenog ulaska u kritični region: **link_permit()**, odnosno za izvezivanje propusnice iz liste propusnica niti prilikom njenog izlaska iz kritičnog regiona: **unlink_permit()**.

Listing 7.4: Klasa **Descriptor** (datoteke **descriptor.hh** i **descriptor.cpp**)

```
typedef int Stack_item;

class Descriptor : private List_link {
  protected:
    Stack_item* stack_top;
    int priority;
    Permit* last;
    unsigned tag;
  public:
    Descriptor();
    inline unsigned tag_get() const { return tag; };
    inline void tag_set(unsigned t) { tag = t; };
    inline void link_permit(Permit* const permit);
    inline Permit* unlink_permit();
    friend class Ready_list;
    friend class Kernel;
    friend class thread;
};

Descriptor::Descriptor()
{
    stack_top = 0;
    priority = 0;
    last = 0;
    tag = 0;
}

void
Descriptor::link_permit(Permit* const permit)
{
    permit->previous = last;
    last = permit;
}
```

```
Permit*
Descriptor::unlink_permit()
{
  Permit* permit = last;
  last = permit->previous;
  return permit;
}
```

7.6 KLASA Ready

Klasa **Ready_list** (Listing 7.5) omogućuje rukovanje spremnim nitima. Primer takvog rukovanja je brzo pronalaženje najprioritetnije niti među spremnim nitima, što je osnov za ispunjenje zahteva da je uvek aktivna najprioritetnija nit. Radi toga, svakom od prioriteta niti se dodeljuje posebna lista spremnih niti i podrazumeva se da se deskriptor spremne niti uvek uvezuje na kraj liste spremnih niti koja odgovara prioritetu dotične niti. Takođe se podrazumeva da se deskriptor spremne niti uvek izvezuje sa početka odabrane liste spremnih niti. Na ovaj način spremne niti istog prioriteta se uvek aktiviraju u redosledu u kome su postajale spremne.

Sve liste spremnih niti zajedno formiraju multi-listu (**Ready::ready**). Rukovanje ovom multi-listom obuhvata:

1. dobijanje prioriteta najprioritetnije neprazne liste spremnih niti: **Ready::highest()**,
2. uvezivanje deskriptora niti na kraj odgovarajuće liste spremnih niti: **Ready::insert()**,
3. izvezivanje deskriptora niti sa početka najprioritetnije neprazne liste spremnih niti: **Ready::extract()** i
4. poređenje prioriteta najprioritetnije neprazne liste spremnih niti sa prioritetom zadane niti: **Ready::higher_than()**.

Pošto je multi-lista niz listi spremnih niti, deskriptor spremne niti se uvezuje na kraj liste spremnih niti koju direktno indeksira prioritet ove niti. Međutim, za operaciju izvezivanja deskriptora iz najprioritetnije neprazne liste spremnih niti, potrebno je prvo pronaći najprioritetniju nepraznu listu spremnih niti. Brzo pronalaženje najprioritetnije neprazne liste spremnih niti se ostvaruje tako da se svaka lista spremnih niti reprezentuje jednim bitom koji sadrži 1 ako je lista neprazna, a 0 ako je lista prazna. Ove bite sadrži **Ready::priority_bits**, tako da se na značajnijim pozicijama nalaze biti prioritetnijih listi spremnih niti. Na najmanje značajnoj poziciji je bit nulte liste spremnih niti, sa prioritetom 0. U njoj se nalazi posebna nulta nit, sa prioritetom 0. Ona angažuje procesor kada nema drugih spremnih niti. Kada je nulta nit u stanju "spremna", tada je njen deskriptor uvezan u nultu listu spremnih niti. Nulta nit može biti još samo u stanju "aktivna". Ona u to stanje prelazi kada ne postoji neka druga nit koja može da zaposli procesor.

Najniži prioritet spremnih niti 0 (**ZERO**) je rezervisan za nultu nit, a najviši prioritet

spremnih niti 31 (**SYSTEM**) je rezervisan za sistemske niti (Listing 7.5). Između se nalaze prioriteti korisničkih niti (**PR01**, **PR02**, . . . , **PR30**). Za uništavane niti, koje čekaju da budu uništene i koje više ne mogu biti spremne (a ni aktivne), uveden je poseban završni prioritet -1 (**TERMINAL**) koji omogućuje njihovo posebno tretiranje u okviru operacije **Ready::insert()**.

Listing 7.5: Klasa **Ready_list** (datoteke **ready.hh** i **ready.cpp**)

```
const unsigned
PRIORITY_NUMBER = 32;

enum Priority {TERMINAL = -1,
               ZERO = 0,
               PR01, PR02, PR03, PR04, PR05, PR06, PR07, PR08, PR09, PR10,
               PR11, PR12, PR13, PR14, PR15, PR16, PR17, PR18, PR19, PR20,
               PR21, PR22, PR23, PR24, PR25, PR26, PR27, PR28, PR29, PR30,
               SYSTEM = 31};

class Ready_list {
   unsigned priority_bits;
   List_link ready[PRIORITY_NUMBER];
   Ready_list(const Ready_list&);
   Ready_list& operator=(const Ready_list&);
  public:
   Ready_list() : priority_bits(0) {};
   int highest() const;
   void insert(Descriptor* d);
   Descriptor* extract();
   bool higher_than(Descriptor* d) const;
};

int
Ready_list::highest() const
{
   int n = 0;
   if(priority_bits != 0)
      n = ad__get_index_of_most_significant_set_bit(priority_bits);
   return n;
}
```

```
void
Ready_list::insert(Descriptor* d)
{
   if(d->priority != TERMINAL) {
      priority_bits = ad__set_bit(priority_bits, d->priority);
      ready[d->priority].insert(d);
   }
}

Descriptor*
Ready_list::extract()
{
   Descriptor* d;
   d = (Descriptor*)(ready[highest()].extract());
   if(ready[d->priority].empty())
      priority_bits = ad__clear_bit(priority_bits, d->priority);
   return d;
}

bool
Ready_list::higher_than(Descriptor* d) const
{
   return(highest() > d->priority);
}

static Ready_list
ready;
```

Rukovanje multi-listom podrazumeva i rukovanje bitima koji reprezentuju pojedine liste spremnih niti. Tako, na primer, pronalaženje najprioritetnije neprazne liste spremnih niti se svodi na određivanje indeksa najznačajnijeg postavljenog bita među bitima koji reprezentuju pojedine liste spremnih niti. U toku rukovanja multi-listom neophodno je zaštititi njenu konzistentnost.

7.7 KLASA Atomic_region

Klasa **Atomic_region** (Listing 7.6) omogućuje stvaranje atomskih regiona. Njen konstruktor onemogućuje prekide, a njen destruktor poništava akciju konstruktora. **Atomic_region::flags** čuva stanje o[ne]mogućenosti prekida koje je zatekao konstruktor i koje treba da vrati destruktor ove klase.

Listing 7.6: Klasa **Atomic_region** (datoteke **atomic_region.hh** i **atomic_region.cpp**)

```
class Atomic_region {
   bool flags;
   Atomic_region(const Atomic_region&);
   Atomic_region& operator=(const Atomic_region&);
  public:
   Atomic_region();
   ~Atomic_region();
};

Atomic_region::Atomic_region()
{
   flags = ad__disable_interrupts();
}

Atomic_region::~Atomic_region()
{
   ad__restore_interrupts(flags);
}
```

7.8 KLASA Kernel

Klasa **Kernel** (Listing 7.7) omogućuje rukovanje procesorom. Rukovanje procesorom se svodi na preključivanje procesora sa jedne niti na drugu.

Za preključivanje je neophodno imati adresu deskriptora aktivne niti sa koje se procesor preključuje (**active**), adresu deskriptora niti na koju se procesor preključuje (**pretender**), kao i adresu deskriptora niti sa koje se procesor preključio (**former**). Operaciju preključivanja poziva privatna operacija **switch_to()**, koja postavlja pokazivač deskriptora aktivne niti **active** i pokazivač deskriptora niti sa koje se procesor preključio **former**.

Preključivanje je nužno vezano za raspoređivanje (*scheduling*), odnosno za izbor niti na koju se procesor preključuje. Ciljevi raspoređivanja kod CppTss izvršioca su da uvek bude aktivna najprioritetnija spremna nit i da se ravnomerno deli vreme procesora između spremnih niti istog prioriteta. Do raspoređivanja dolazi:

1. kada se pojavi spremna nit sa višim prioritetom od aktivne niti i
2. na kraju kvantuma.

Pomenuta dva slučaja raspoređivanja su podržana, respektivno, sledećim operacijama klase **Kernel**:

1. **schedule()** i

2. **periodic_schedule()**.

Klasa **Kernel** nasleđuje klasu **Deskriptor** da bi jedini objekat klase **Kernel** reprezentovao deskriptor **main()** niti. Konstruktor ove klase proglašava aktivnom **main()** nit. Pošto **main()** nit koristi stek konkurentnog programa kao svoj stek, za nju nije potrebno zauzeti stek.

U nadležnosti klase **Kernel** nije samo preključivanje, nego i podrška viših slojeva iz hijerarhijske strukture CppTss izvršioca.

Aktivnost niti omogućava operacija **make_ready()**.

Očekivanje dešavanja događaja omogućuje operacija **expect()**, a objavu dešavanja događaja omogućuje operacija **signal()**.

Operacije **exclusive_in()** i **exclusive_out()** ostvaruju ulazak u isključivi region i izlazak iz njega.

Očekivanje ispunjenja uslova omogućuje operacija **wait()**, dok operacija **notify_one()** omogućuje objavu ispunjenja uslova.

U operacijama deljene promenljive **kernel** se koriste atomski regioni radi zaštite njene konzistentnosti, kao i konzistentnosti argumenata iz poziva ovih operacija. To je neophodno da bi polja ove deljene promenljive bila konzistentna, odnosno da bi rezultat uvezivanja u razne liste i izvezivanja iz njih bio ispravan. Isto važi i za funkcije koje pristupaju deljenoj promenljivoj **kernel**. Atomski regioni nisu korišćeni samo u operacijama koje se uvek pozivaju iz atomskih regiona i u funkcijama koje ne ugrožavaju konzistentnost.

Prethodno pomenuto korišćenje atomskih regiona je prihvatljivo, jer su operacije deljene promenljive **kernel** i funkcije koje joj pristupaju kratkotrajne i ne dovode do značajnog odlaganja obrada prekida.

Listing 7.7: Klasa **Kernel** (datoteke **kernel.hh** i **kernel.cpp**)

```
class Kernel : public Descriptor {
   Descriptor* active;
   Descriptor* pretender;
   Descriptor* former;
   Kernel(const Kernel&);
   Kernel& operator=(const Kernel&);
   inline void switch_to(Descriptor* const d);
   inline void schedule();
   inline void periodic_schedule();
  public:
   Kernel() : active(0),pretender(0), former(0)
         { priority = PR15; active = this; };
   inline void make_ready(Descriptor* const d);
   inline void expect(List_link* const waiting_list);
   inline void signal(List_link* const waiting_list);
   inline void exclusive_in(Permit* const permit);
   inline void wait(const unsigned t, List_link* const waiting_list);
   inline void notify_one(List_link* const waiting_list);
   inline void exclusive_out();
   inline Descriptor* active_get() const { return active; };
   friend void yield();
   friend class Timer_driver;
};

void
Kernel::switch_to(Descriptor* const d)
{
   former = active;
   active = d;
   ad__stack_swap(&(former->stack_top), active->stack_top);
}
```

```
void
Kernel::schedule()
{
   if(ready.higher_than(active)) {
      ready.insert(active);
      pretender = ready.extract();
      switch_to(pretender);
   }
}

void
Kernel::periodic_schedule()
{
   ready.insert(active);
   pretender = ready.extract();
   if(active != pretender)
      switch_to(pretender);
}

void
Kernel::make_ready(Descriptor* const d)
{
   Atomic_region ar;
   ready.insert(d);
}

void
Kernel::expect(List_link* const waiting_list)
{
   waiting_list->insert(active);
   pretender = ready.extract();
   switch_to(pretender);
}
```

```
void
Kernel::signal(List_link* const waiting_list)
{
   if(waiting_list->not_empty()) {
      pretender =(Descriptor*) waiting_list->extract();
      ready.insert(pretender);
      schedule();
   }
}

void
Kernel::exclusive_in(Permit* const permit)
{
   Atomic_region ar;
   if(permit->not_free()) {
      permit->admission_insert(active);
      pretender = ready.extract();
      switch_to(pretender);
   } else {
      permit->take();
      active->link_permit(permit);
   }
}
```

```
void
Kernel::wait(const unsigned t, List_link* const waiting_list)
{
   Atomic_region ar;
   Permit* permit = active->unlink_permit();
   active->tag = t;
   if(permit->fulfilled_not_empty()) {
      pretender = (Descriptor*)permit->fulfilled_extract();
      pretender->link_permit(permit);
      ready.insert(pretender);
   } else if(permit->admission_not_empty()) {
      pretender = (Descriptor*)permit->admission_extract();
      pretender->link_permit(permit);
      ready.insert(pretender);
   } else
      permit->release();
   waiting_list->insert(active);
   pretender = ready.extract();
   switch_to(pretender);
}

void
Kernel::notify_one(List_link* const waiting_list)
{
   Permit* permit = active->last;
   if(permit == 0)
      throw &failure_notify_outside_exclusive_region;
   Atomic_region ar;
   if(waiting_list->not_empty()) {
      pretender =(Descriptor*) waiting_list->extract();
      permit->fulfilled_insert(pretender);
   }
}
```

```
void
Kernel::exclusive_out()
{
   Atomic_region ar;
   Permit* permit = active->unlink_permit();
   if(permit->fulfilled_not_empty()) {
      pretender = (Descriptor*)permit->fulfilled_extract();
      pretender->link_permit(permit);
      ready.insert(pretender);
   } else if(permit->admission_not_empty()) {
      pretender = (Descriptor*)permit->admission_extract();
      pretender->link_permit(permit);
      ready.insert(pretender);
   } else
      permit->release();
   schedule();
}

static Kernel
kernel;

void
yield()
{
   Atomic_region set_up;
   kernel.periodic_schedule();
}
```

7.9 KLASA mutex

Klasa **mutex** (Listing 7.8) omogućuje zaključavanje i otključavanje resursa koje reprezentuju njeni objekti. Objekat ove klase je propusnica čije zauzimanje omogućuje operacija **mutex::lock()**, a oslobađanje operacija **mutex::unlock()**.

Listing 7.8: Klasa **mutex** (datoteke **mutex.hh** i **mutex.cpp**)

```
class mutex : private Permit {
   mutex(const mutex&);
   mutex& operator=(const mutex&);
  public:
   mutex() {};
   void lock();
   void unlock();
};

void
mutex::lock()
{
   kernel.exclusive_in(this);
}

void
mutex::unlock()
{
   kernel.exclusive_out();
}
```

7.10 KLASA unique_lock

Klasa **unique_lock** (Listing 7.9) omogućuje stvaranje isključivih regiona. Njen konstruktor omogućuje ulazak u isključivi region, a njen destruktor omogućuje izlazak iz isključivog regiona.

Listing 7.9: Klasa **unique_lock** (datoteke **unique_lock.hh** i **unique_lock.cpp**)

```
template<class MUTEX>
class unique_lock {
   unique_lock(const unique_lock&);
   unique_lock& operator=(const unique_lock&);
  public:
   unique_lock(MUTEX& mx);
   ~unique_lock();
};
```

```
template<class MUTEX>
unique_lock<MUTEX>::unique_lock(MUTEX& mx)
{
   kernel.exclusive_in((Permit*)&mx);
}

template<class MUTEX>
unique_lock<MUTEX>::~unique_lock()
{
   kernel.exclusive_out();
}
```

7.11 KLASA condition_variable

Klasa **condition_variable** (Listing 7.10) omogućuje zaustavljanje aktivnosti niti do ispunjenja uslova i objavu ispunjenja uslova, kao i uticanje na redosled deskriptora niti koje čekaju ispunjenje uslova.

Listing 7.10: Klasa **condition_variable** (datoteke **condition_variable.hh** i **condition_variable.cpp**)

```
class condition_variable {
      List_link list_head;
      List_link* position;
   public:
      condition_variable() { position = &list_head; };
      void wait(unique_lock<mutex>& lock, unsigned t = 0);
      void notify_one();
      bool first(unsigned* t = 0);
      bool last();
      bool next(unsigned* t = 0);
      bool attach_tag(unsigned t);
};

void
condition_variable::wait(unique_lock<mutex>& lock, unsigned t)
{
   kernel.wait(t, position);
   position = &list_head;
}
```

```
void
condition_variable::notify_one()
{
   kernel.notify_one(&list_head);
   position = &list_head;
}

bool
condition_variable::first(unsigned* t)
{
   bool r = false;
   if(list_head.not_empty()) {
      position = list_head.right_get();
      if(t != 0)
         *t = ((Descriptor*) position)->tag_get();
      r = true;
   }
   return(r);
}

bool
condition_variable::last()
{
   bool r = false;
   if(list_head.not_empty()) {
      position = &list_head;
      r = true;
   }
   return(r);
}
```

```
bool
condition_variable::next(unsigned* t)
{
   bool r = false;
   if(position != &list_head) {
      position = position->right_get();
      if(position != &list_head) {
         if(t != 0)
            *t = ((Descriptor*) position)->tag_get();
         r = true;
      }
   }
   return(r);
}

bool
condition_variable::attach_tag(unsigned t)
{
   bool r = false;
   if(position != &list_head) {
      ((Descriptor*) position)->tag_set(t);
      r = true;
   }
   return(r);
}
```

7.12 KLASA Driver

Klasa **Driver** (Listing 7.11) omogućuje smeštanje adrese obrađivača prekida u tabelu prekida: **Driver::start_interrupt_handling()**. Pored toga, ova klasa uvodi definiciju klase **Event** koja omogućuje zaustavljanje aktivnosti niti do dešavanja događaja i objavu dešavanja događaja.

Listing 7.11: Klasa **Driver** (datoteke **driver.hh** i **driver.cpp**)

```
class Driver {
 protected:
   void start_interrupt_handling(Vector_numbers vector_number,
                        void (*handler)());
   class Event {
      List_link list_head;
    public:
      void expect();
      void signal();
   };
};

void
Driver::start_interrupt_handling(Vector_numbers vector_number,
                        void (*handler)())
{
   Atomic_region ar;
   ad__set_vector(vector_number, handler);
}

void
Driver::Event::expect()
{
   kernel.expect(&list_head);
}

void
Driver::Event::signal()
{
   kernel.signal(&list_head);
}
```

7.13 KLASE Exception_driver I Timer_driver

Iz klase **Driver** su izvedene klase **Exception_driver** i **Timer_driver** (Listing 7.12). Prva od njih omogućuje reakciju na pojavu hardverskih izuzetaka, radi izazivanja prevremenog kraja konkurentnog programa. Druga od ovih klasa omogućuje rukovanje vremenom.

Klasa **Exception_driver** uvodi operaciju **interrupt_handler()**. Ova operacija zaustavlja izvršavanje konkurentnog programa. Na osnovu klase **Exception_driver**

nastaje drajver **exception_driver**.

Rukovanje vremenom obuhvata:

1. brojanje otkucaja sata, radi praćenja proticanja sistemskog vremena,
2. odbrojavanje otkucaja sata preostalih do kraja kvantuma aktivne niti kao i
3. odbrojavanja otkucaja sata preostalih do buđenja uspavane niti.

Kada broj otkucaja, preostalih do isticanja kvantuma aktivne niti, padne na nulu, potrebno je pokrenuti periodično raspoređivanje. Takođe, kada broj otkucaja, preostalih do buđenja uspavane niti, padne na nulu, potrebno je probuditi sve niti za koje je nastupio trenutak buđenja. Svi prethodno pobrojani poslovi se nalaze u nadležnosti operacije **interrupt_handler()** koju uvodi klasa **Timer_driver**. Polje **current_ticks** ove klase sadrži sistemsko vreme, polje **countdown** sadrži broj otkucaja do buđenja, a polje **rest** broj otkucaja do isticanja kvantuma.

Funkcija **now()** vraća sadržaj polja **current_ticks**, odnosno vraća sistemsko vreme (Listing 7.12).

Na osnovu klase **Timer_driver** nastaje drajver **timer_driver**.

Listing 7.12: Klase **Exception_driver** i **Timer_driver** (datoteke **drivers.hh** i **drivers.cpp**)

```
const unsigned long
QUANTUM = 2;

class Exception_driver : public Driver {
   static void interrupt_handler();
  public:
   Exception_driver() { start_interrupt_handling(FP_EXCEPTION,
                                                 interrupt_handler); };
};

void
Exception_driver::interrupt_handler()
{
   ad__report_and_finish("\nHARDWARE EXCEPTION!\n");
}

static Exception_driver
exception_driver;
```

```
class Timer_driver : public Driver {
   static unsigned long current_ticks;
   static unsigned long countdown;
   static unsigned long rest;
   static unsigned long quantum;
   static Event alarm;
   static void interrupt_handler();
  public:
   Timer_driver() { start_interrupt_handling(TIMER, interrupt_handler); };
   friend unsigned long now();
   friend class Delta;
};

unsigned long
Timer_driver::current_ticks = 0;

unsigned long
Timer_driver::countdown = 0;

unsigned long
Timer_driver::rest = QUANTUM;

unsigned long
Timer_driver::quantum = QUANTUM;

Timer_driver::Event
Timer_driver::alarm;

void
Timer_driver::interrupt_handler()
{
   current_ticks++;
   if((--rest) == 0)
      rest = quantum;
   if((countdown > 0) && ((--countdown) == 0))
      alarm.signal();
   else if(rest == quantum)
      kernel.periodic_schedule();
}

static Timer_driver
timer_driver;
```

```
unsigned long
now()
{
   Atomic_region ar;
   return timer_driver.current_ticks;
}
```

7.14 KLASA Memory_fragment

Klasa **Memory_fragment** (Listing 7.13) omogućuje rukovanje slobodnom radnom memorijom. Slobodnu radnu memoriju obrazuje celi broj jedinica sastavljenih od **UNIT** bajta. Rukovanje slobodnom radnom memorijom podrazumeva da se uvek zauzima, odnosno da se uvek oslobađa celi broj ovih jedinica. Zauzimanja ovakvih zona slobodne radne memorije, odnosno njihova oslobađanja uzrokuju iscepkanost slobodne radne memorije u odsečke. Odsečci se zato uvezuju u jednosmernu listu, uređenu u rastućem redosledu njihovih početnih adresa. Radi toga, početak svakog odsečka sadrži svoju veličinu, izraženu u pomenutim jedinicama od po **UNIT** bajta, i pokazivač narednog odsečka. Veličinu odsečka i pokazivač narednog odsečka sadrže polja **size** i **next** klase **Memory_fragment**.

Konstruktor klase **Memory_fragment** opisuje obrazovanje liste odsečaka slobodne radne memorije, sastavljene od stalnog odsečka čija veličina je 0 i od odsečka koji obuhvata raspoloživu slobodnu radnu memoriju. Stalnom (prvom) odsečku odgovara objekt **memory** klase **Memory_fragment**, koji je jedini objekt ove klase. Dodavanju drugog odsečka prethodi provera da li je obezbeđeno dovoljno radne memorije za potrebe konkurentnog programa. Ako nije, izvršavanje konkurentnog programa se završava uz poruku **INITIAL MEMORY SHORTAGE**. Pošto je **UNIT** jednak veličini stranice, uvek se zauzima toliko radne memorije da u nju može da stane traženi broj stranica, a da početak raspoložive slobodne radne memorije bude postavljen na početak prve stranice.

Zauzimanje slobodne radne memorije omogućuje operacija **take()** klase **Memory_fragment**. Zauzima se jedna jedinica od **UNIT** bajta više nego što je traženo. Ona prethodi preostalim zauzetim jedinicama memorije i sadrži ukupnu veličinu zauzete memorije. Ova veličina se koristi prilikom kasnijeg oslobađanja zauzete memorije. Zauzimanju prethodi pretraživanje liste odsečaka, radi pronalaženja prvog dovoljno velikog odsečka. Pretraživanje uvek počinje od stalnog odsečka. Ako se pronađe dovoljno velik odsečak, traženi bajti se zauzimaju s njegovog kraja. Ako pronađeni odsečak obuhvata baš traženi broj bajta, tada se on isključuje iz liste i zauzimaju se svi njegovi bajti.

Oslobađanje prethodno zauzete radne memorije omogućuje operacija **free()** klase **Memory_fragment**. Za oslobađanje je neophodno u listi odsečaka pronaći odsečak iza koga će se oslobađani odsečak uvezati u ovu listu. Pre uvezivanja proverava se da li oslobađani odsečak može da se spoji u jedan odsečak sa svojim prethodnikom i sa svojim

sledbenikom. Odsečak se uvezuje u pomenutu listu samo ako ovo spajanje nije moguće.

Prethodno opisane operacije klase **Memory_fragment** su namenjene za zauzimanje i oslobađanje radne memorije prilikom stvaranja i uništavanja objekata pojedinih klasa. Da bi se njihova namena ostvarila, neophodno je da ove operacije pozivaju globalni operatori **new()** i **delete()**. Ali, tada razne niti mogu da pozivaju operacije klase **Memory_fragment** posredstvom prethodna dva operatora i da tako ugroze konzistentnost liste odsečaka. Da bi se to sprečilo, ova klasa nasleđuje klasu **mutex** i tako omogućuje zaključavanje i otključavanje njenog jedinog objekta **memory**. To je obezbeđeno u definicijama funkcija **operator new()** i **operator delete()** (Listing 7.13), radi ostvarenja međusobne isključivosti različitih pristupanja listi odsečaka.

Listing 7.13: Klasa **Memory_fragment** (datoteke **memory.hh** i **memory.cpp**)

```
const size_t
UNIT = PAGE_SIZE;

class Memory_fragment : public mutex {
  size_t size;
  Memory_fragment* next;
  Memory_fragment(const Memory_fragment&);
  Memory_fragment& operator=(const Memory_fragment&);
 public:
  Memory_fragment();
  void* take(size_t size);
  void free(void* address);
};

Memory_fragment::Memory_fragment() :
  size(0), next(this)
{
  size_t free_memory = 2000 * UNIT;
  size_t beginning =(size_t) malloc(free_memory + UNIT-1);
  if(beginning == 0)
    ad__report_and_finish("INITIAL MEMORY SHORTAGE");
  else {
    beginning = (beginning + UNIT-1) & ~(UNIT-1);
    next =(Memory_fragment*) beginning;
    next->size = free_memory;
    next->next = this;
  }
}
```

```
void*
Memory_fragment::take(size_t size)
{
   size += 2 * UNIT - 1;
   size -= size % UNIT;
   Memory_fragment* m = 0;
   Memory_fragment* p = this;
   while(p->next != this) {
      if((p->next->size) < size)
         p = p->next;
      else if(p->next->size == size) {
         m = p->next;
         p->next = p->next->next;
         break;
      } else {
         p->next->size -= size;
         m =(Memory_fragment*) ((size_t)(p->next) + (p->next->size));
         break;
      }
   }
   if(m == 0)
      throw &failure_memory_shortage;
   m->size = size;
   return (void*) ((size_t)m + UNIT);
}
```

```
void
Memory_fragment::free(void* address)
{
   if(address != 0) {
      Memory_fragment* a = (Memory_fragment*) ((size_t)address - UNIT);
      Memory_fragment* p = this;
      while(p->next != this)
         if(a > p->next)
            p = p->next;
         else
            break;
      if((((size_t) p) + (p->size)) == ((size_t) a)) {
         p->size += a->size;
         if((((size_t) p) + (p->size)) == ((size_t)(p->next))) {
            p->size += p->next->size;
            p->next = p->next->next;
         }
      } else if((((size_t) a) + (a->size)) == ((size_t)(p->next))) {
         a->size += p->next->size;
         a->next = p->next->next;
         p->next = a;
      } else {
         a->next = p->next;
         p->next = a;
      }
   }
}

static Memory_fragment
memory;
```

```
void*
operator new(size_t size)
{
   void* memory_block;
   memory.lock();
   try {
      memory_block = memory.take(size);
   }
   catch(...) {
      memory.unlock();
      throw;
   }
   memory.unlock();
   return memory_block;
}

void operator delete(void* address)
{
   memory.lock();
   memory.free(address);
   memory.unlock();
}
```

7.15 KLASE Thread_image i thread

Rukovanje nitima omogućuju klase **Thread_image** i **thread**, kao i funkcije **thread_destroyer_deamon()**, **destroy()** i **undetached_threads()** (Listing 7.14). Klasa **Thread_image** opisuje sliku niti, sastavljenu od deskriptora niti, uslova (**ended**) koji omogućuje čekanje završetka aktivnosti niti, oznake da je dozvoljen nasilni kraj aktivnosti niti (**detached**) i steka niti. Klasa **thread** omogućuje međusobnu isključivost svojih operacija (**mx**), brojanje niti za koje nije dozvoljen nasilni kraj njihove aktivnosti (**undetached_threads_number**), uništavanje niti (**termination**, **terminating**), kao i pristup slici niti (**ti**). Konstruktor klase **thread** omogućuje kreiranje niti. U toku kreiranja niti pripremi se njen stek za preključivanje, da bi automatski započelo izvršavanje funkcije koja opisuje ponašanje niti nakon prvog preključivanja na nit. Završetak aktivnosti niti se otkriva u okviru operacija **join()** i **detach()** na osnovu završnog prioriteta niti (**TERMINAL**).

Na završetku aktivnosti niti, u toku izvršavanja funkcije **destroy()**, omogućuje se nastavak aktivnosti niti koja čeka dotični završetak i oslobađanje prostora koga zauzima slika niti, što je u nadležnosti sistemske niti **thread_destroyer_deamon().**

Funkcija **undetached_threads()** omogućuje proveru da li postoje niti za koje nije dozvoljen nasilni kraj njihove aktivnosti, da bi se na kraju aktivnosti procesa ukazalo na prevremeni završetak ovakvih niti.

Listing 7.14: Klase **Thread_image** i **thread** (datoteke **thread.hh** i **thread.cpp**)

```
const unsigned
DEFAULT_STACK_SIZE = 4096;

class Thread_image: public Descriptor {
   condition_variable ended;
   bool detached;
   Stack_item stack[DEFAULT_STACK_SIZE];
   Thread_image (const Thread_image &);
   Thread_image & operator=(const Thread_image &);
   public:
   Thread_image(void (*thread_function)(), Priority p);
   friend class thread;
   friend void destroy();
};

Thread_image::Thread_image(void (*thread_function)(), Priority p) :
   detached(false)
{
   stack_top = &(stack[DEFAULT_STACK_SIZE]);
   ad__stack_init(&stack_top, (unsigned)thread_function);
   priority = p;
}
```

```
class thread {
   static mutex mx;
   static unsigned undetached_threads_number;
   static condition_variable termination;
   static Thread_image* terminating;
   Thread_image* ti;
   thread (const thread &);
   thread & operator=(const thread &);
   public:
   thread(void (*thread_function)(), Priority p = PR15);
   void join();
   void detach();
   friend void thread_destroyer_deamon();
   friend void destroy();
   friend bool undetached_threads();
};

mutex
thread::mx;

unsigned
thread::undetached_threads_number = 0;

condition_variable
thread::termination;

Thread_image*
thread::terminating;

thread::thread(void (*thread_function)(), Priority p)
{
   ti = new Thread_image(thread_function, p);
   unique_lock<mutex> lock(mx);
   undetached_threads_number++;
   kernel.make_ready(ti);
}
```

```
void
thread::join()
{
   unique_lock<mutex> lock(mx);
   if(ti->priority != TERMINAL)
      ti->ended.wait(lock);
}

void
thread::detach()
{
   unique_lock<mutex> lock(mx);
   if((ti->priority != TERMINAL) && (!ti->detached)) {
      ti->detached = true;
      undetached_threads_number--;
   }
}

void
thread_destroyer_deamon()
{
  for(;;)  {
     unique_lock<mutex> lock(thread::mx);
     thread::termination.wait(lock);
     delete thread::terminating;
  }
}

void
destroy()
{
   unique_lock<mutex> lock(thread::mx);
   thread::terminating = (Thread_image*)kernel.active_get();
   while(thread::terminating->ended.last())
      thread::terminating->ended.notify_one();
   if(!thread::terminating->detached)
      thread::undetached_threads_number--;
   thread::terminating->priority = TERMINAL;
   thread::termination.notify_one();
}
```

```
bool
undetached_threads()
{
  return (thread::undetached_threads_number > 0);
}
```

7.16 KLASA Delta

Klasa **Delta** i funkcije **thread_wake_up_deamon()**, **sleep_for()** i **sleep_until()** zajedno omogućuju uspavljivanje i buđenje niti, a funkcija **thread_zero()** opisuje aktivnost nulte (sistemske) niti (Listing 7.15). Funkcija **sleep_for()** omogućuje uspavljivanje aktivne niti dok ne protekne zadani broj otkucaja sata, a funkcija **sleep_until()** omogućuje uspavljivanje aktivne niti dok ne nastupi zadani trenutak sistemskog vremena. Do uspavljivanja ne dolazi, ako je dotični trenutak prošao u vreme poziva funkcije.

Oko polja **list** klase **Delta** se formira lista deskriptora uspavanih niti. Da se za svaku uspavanu nit ne bi proveravalo, nakon svakog otkucaja, da li je nastupilo vreme njenog buđenja, deskriptori uspavanih niti se uvezuju u listu u hronološkom redosledu buđenja niti. Svakom od ovih deskriptora je dodeljen privezak koji pokazuje relativno vreme buđenja (relativni broj otkucaja do buđenja) u odnosu na prethodnika u listi. Ovakva lista se zove delta lista. Zahvaljujući delta listi, nakon svakog otkucaja potrebno je proveriti da li je nastupio trenutak buđenja samo za nit koja se najranije budi, odnosno samo za prvi deskriptor iz delta liste. Pošto može da bude više niti, čije buđenje je vezano za isti trenutak, unapred nije poznato koliko niti treba probuditi nakon otkucaja sata.

Operaciju **awake()** klase **Delta** poziva sistemska nit **Wake_up_daemon()**. Vreme njenog buđenja je uvek jednako najranijem vremenu buđenja korisničkih niti. Nakon buđenja, sistemska nit budi sve korisničke niti sa početka delta liste, za koje je nastupio trenutak buđenja. Čekanje buđenja omogućuje poziv operacije **Timer_driver::alarm.expect()**. Dužinu čekanja određuje vrednost lokalne promenljive **tag**, kada ima uspavanih korisničkih niti (na čije prisustvo ukazuje vrednost lokalne promenljive **sleeping**). Dužina čekanja se skraćuje za vrednost lokalne promenljive **passed_ticks**, koja registruje vreme proteklo na buđenju korisničkih niti.

Operaciju **sleep()** klase **Delta** poziva, posredstvom funkcije **sleep_for()**, korisnička nit, da bi se njen deskriptor uključio u delta listu, a njena aktivnost privremeno zaustavila.

Konstruktor klase **Delta** omogućuje kreiranje sistemskih niti korišćenjem bezimenih (privremenih) objekata klase **thread**.

Konzistentnost delta liste štite isključivi regioni u telima operacija **awake()** i **sleep()** klase **Delta**.

Listing 7.15: Klasa **Delta** (datoteke **delta.hh** i **delta.cpp**)

```
void
thread_zero();

void
thread_wake_up_deamon();

typedef unsigned long milliseconds;

class Delta {
   mutex mx;
   condition_variable list;
   inline void sleep(milliseconds duration);
   inline void awake();
   Delta (const Delta &);
   Delta & operator=(const Delta &);
  public:
   Delta();
   friend void thread_wake_up_deamon();
   friend void sleep_for(milliseconds duration);
};

Delta::Delta()
{
   thread (thread_zero, ZERO).detach();
   thread (thread_destroyer_deamon, SYSTEM).detach();
   thread (thread_wake_up_deamon, SYSTEM).detach();
}
```

```
void
Delta::awake()
{
   bool sleeping = false;
   unsigned long begining_moment = 0;
   unsigned long passed_ticks = 0;
   unsigned tag = 0;
   for(;;) {
      { Atomic_region ar;
       if(!sleeping)
          Timer_driver::alarm.expect();
       else {
          passed_ticks = Timer_driver::current_ticks - begining_moment;
          if(tag > passed_ticks) {
             Timer_driver::countdown = tag - passed_ticks;
             Timer_driver::alarm.expect();
          }
       }
       begining_moment = Timer_driver::current_ticks;
      }
      { unique_lock<mutex> lock(mx);
       do {
          passed_ticks = passed_ticks - tag;
          list.notify_one();
          sleeping = list.first(&tag);
       } while(sleeping && (tag <= passed_ticks));
      }
   }
}
```

```
void
Delta::sleep(milliseconds duration)
{
   unsigned new_tag = duration;
   unsigned old_tag;
   {
     Atomic_region ar;
     old_tag = Timer_driver::countdown;
   }
   unique_lock<mutex> lock(mx);
   if(list.first())
      do {
         if(old_tag > duration) {
            list.attach_tag(old_tag - duration);
            break;
         } else if(old_tag == duration) {
            duration = 0;
            list.next();
            break;
         } else
            duration -= old_tag;
      } while(list.next(&old_tag));
   if(duration == new_tag) {
      Atomic_region ar;
      Timer_driver::countdown = duration;
   }
   list.wait(lock, duration);
}

static Delta
delta;

void
thread_zero()
{
   for(;;)
      ;
}
```

```
void
thread_wake_up_deamon()
{
   delta.awake();
}

void
sleep_for(milliseconds duration)
{
   if(duration > 0)
      delta.sleep(duration);
}

void
sleep_until(milliseconds moment)
{
   milliseconds difference = moment - now();
   if(difference <= (ULONG_MAX / 2))
      sleep_for(difference);
}
```

7.17 PITANJA

1. Koje izuzetke podržava klasa **Failure**?
2. Šta omogućuje klasa **List_link**?
3. Šta omogućuje klasa **Permit**?
4. Šta sadrže objekti klase **Permit**?
5. Šta je uslov konzistentnosti propusnice?
6. Šta sadrže objekti klase **Deskriptor**?
7. Koju klasu nasleđuje klasa **Deskriptor**?
8. Šta karakteriše nultu nit?
9. Šta karakteriše klasu **Ready_list**?
10. Koje operacije omogućuju rukovanje multi-listom?
11. Koju klasu nasleđuje klasa **Kernel**?
12. Šta reprezentuje jedini objekat klase **Kernel**?
13. Šta omogućuje klasa **Kernel**?
14. Kada dolazi do raspoređivanja u okviru klase **Kernel**?
15. Koju klasu nasleđuje klasa **mutex**?
16. Šta omogućuje klasa **Driver**?
17. Šta registruje **Timer_driver::interrupt_handler()**?
18. Šta omogućuje **Timer_driver::interrupt_handler()**?
19. Šta karakteriše odsečke slobodne radne memorije?

20. U kom redosledu su uvezani odsečci slobodne radne memorije u listi?
21. Koju klasu nasleđuje klasa **Memory_fragment**?
22. Šta važi za konzistentnost operacija **Memory_fragment::take()** i **Memory_fragment::free()**?
23. Gde se sve štiti konzistentnost zaključavanjem i otključavanjem jedinog objekta klase **Memory_fragment**?
24. Koje operacije sadrži klasa **thread**?
25. Šta koristi klasa **thread** za ostvarenje konzistentnosti?
26. Šta karakteriše deskriptore uspavanih niti?
27. Ko budi uspavane niti?

8 SVOJSTVA DATOTEKA

8.1 OZNAČAVANJE DATOTEKA

Svaka datoteka poseduje ime koje bira korisnik. Poželjno je da ime datoteke ukazuje na:

1. njen konkretan sadržaj i
2. na vrstu njenog sadržaja (radi klasifikacije datoteka po njihovom sadržaju).

Zato su imena datoteka dvodelna, tako da prvi deo imena datoteke označava njen sadržaj, a drugi deo označava vrstu njenog sadržaja, odnosno njen tip. Ova dva dela imena datoteke obično razdvaja tačka. Tako, na primer:

godina1.txt

može da predstavlja ime datoteke, koja sadrži podatke o studentima prve godine studija. Na to ukazuje prvi deo imena **godina1**, dok drugi deo imena **txt** ove datoteke govori da je datoteka tekstualna, odnosno da sadrži samo vidljive *ASCII* znakove.

Rukovanje datotekom obuhvata ne samo rukovanje njenim sadržajem, nego i rukovanje njenim imenom. Tako, na primer, stvaranje datoteke podrazumeva i zadavanje njenog sadržaja, ali i zadavanje njenog imena. To se dešava, na primer, u toku editiranja, kompilacije, kopiranja i slično. Takođe, izmena datoteke može da obuhvati ne samo izmenu njenog sadržaja, nego i izmenu njenog imena, što se dešava, na primer, u editiranju.

8.2 ORGANIZACIJA DATOTEKA

Datoteke se grupišu u skupove datoteka. Na primer, prirodno je da datoteke sa podacima o studentima pojedinih godina studija istog odseka pripadaju jednom skupu datoteka. Za svaki skup datoteka postoji **imenik** (*directory, folder*) koji sadrži imena svih datoteka koje pripadaju datom skupu. Radi razlikovanja imenika, svaki od njih poseduje ime koje bira korisnik. Za imenike su dovoljna jednodelna imena, jer nema potrebe za klasifikacijom imenika. Tako, na primer:

odsek

može da predstavlja ime imenika, koji obuhvata datoteke sa podacima o studentima svih godina studija istog odseka.

Razvrstavanjem datoteka uz pomoć imenika nastaje hijerarhijska organizacija datoteka, u kojoj su na višem nivou hijerarhije imenici, a na nižem nivou se nalaze datoteke, čija imena su sadržana u ovim imenicima. Ovakva hijerarhijska organizacija povlači za sobom i hijerarhijsko označavanje datoteka. Hijerarhijsku oznaku ili **putanju** (*path name*) datoteke obrazuju ime imenika za koji je datoteka vezana i ime datoteke.

Delove putanje obično razdvaja znak **/** (ili znak ****). Tako, na primer:

odsek/godina1.txt

predstavlja putanju datoteke, koja sadrži podatke o studentima prve godine studija sa prvog odseka.

Uobičajeno je da se ime imenika završava znakom **/**:

odsek/

Hijerarhijska organizacija datoteka ima više nivoa, kada jedan imenik sadrži, pored imena datoteka, i imena drugih imenika, odnosno obuhvata, pored datoteka, i druge imenike. Obuhvaćeni imenici se nalaze na nižem nivou hijerarhije. Na primer, imenik fakultet obuhvata imenike pojedinih odseka.

Na vrhu hijerarhijske organizacije datoteka se nalazi **korenski imenik** (*root*) koji obično nema ime.

U slučaju više nivoa u hijerarhijskoj organizaciji datoteka, putanju datoteke obrazuju imena imenika sa svih nivoa hijerarhije (navedena u redosledu od najvišeg nivoa na dole) kao i ime datoteke. Na primer:

/fakultet/odsek/godina1.txt

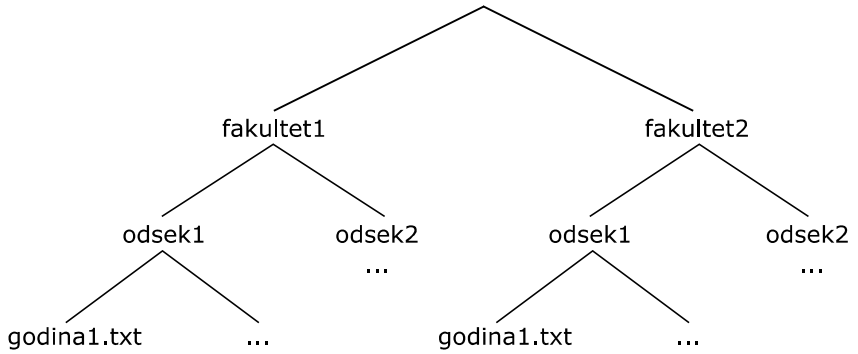
predstavlja putanju datoteke **godina1.txt**, koja pripada imeniku **odsek**. Ovaj imenik pripada imeniku **fakultet**, a on pripada korenskom imeniku. Pošto korenski imenik nema ime, njega označava prvi znak **/**.

Na prethodno opisani način se obrazuje i putanja imenika. Tako, na primer:

/fakultet/odsek/

predstavlja putanju imenika **odsek**, koji pripada imeniku **fakultet** iz korenskog imenika.

Hijerarhijska organizacija datoteka dozvoljava da postoje datoteke (imenici) sa istim imenima, pod uslovom da pripadaju raznim imenicima (Slika 8.1).



Slika 8.1: Grafička predstava hijerarhijske organizacije datoteka

U hijerarhijskoj organizaciji datoteka (Slika 8.1) korenskom imeniku pripadaju imenici **fakultet1** i **fakultet2**. Svaki od njih sadrži imenike **odsek1** i **odsek2**. Pri tome, oba imenika sa imenom **odsek1** sadrže datoteku **godina1.txt**. Putanje omogućuju razlikovanje istoimenih imenika, odnosno istoimenih datoteka. Tako, putanje:

/fakultet1/odsek1/
/fakultet2/odsek1/

omogućuju razlikovanje imenika sa imenom **odsek1**, a putanje:

/fakultet1/odsek1/godina1.txt

/fakultet2/odsek1/godina1.txt

omogućuju razlikovanje datoteka sa imenom **godina1.txt**.

Zahvaljujući hijerarhijskoj organizaciji datoteka, moguće je rukovanje skupovima datoteka. Na primer, moguće je kopiranje celog imenika, odnosno kopiranje svih datoteka i imenika, koji mu pripadaju.

Navođenje **apsolutne putanje** datoteke, sa svim prethodećim imenicima, je potrebno kad god je moguć nesporazum, zbog datoteka sa istim imenima, odnosno, zbog imenika sa istim imenima. Ali, ako postoji mogućnost određivanja nekog imenika kao **radnog** (*working directory*), tada se njegova putanja može podrazumevati i ne mora se navoditi. Na primer, ako se podrazumeva da je:

/fakultet1/odsek1/

radni imenik, tada:

godina1.txt

jednoznačno označava datoteku, koja pripada imeniku **odsek1** iz imenika **fakultet1**.

Radni imenik omogućuje korišćenje **relativnih putanja**. Na primer, ako se podrazumeva da je:

/fakultet1/

radni imenik, tada:

odsek1/godina1.txt

jednoznačno označava datoteku, koja pripada imeniku **fakultet1**.

Datoteke koje pripadaju istoj hijerarhijskoj organizaciji obrazuju **sistem datoteka**.

8.3 ZAŠTITA DATOTEKA

Za uspešnu upotrebu podataka, trajno pohranjenih u datotekama, neophodna je zaštita datoteka. Ona obezbeđuje da podaci, sadržani u datoteci, neće biti izmenjeni bez znanja i saglasnosti njihovog vlasnika. Takođe, ona obezbeđuje da podatke, sadržane u datoteci jednog korisnika, bez njegove dozvole drugi korisnici ne mogu da koriste. Podaci, sadržani u datoteci, ostaju neizmenjeni, ako se onemogući pristup datoteci radi **pisanja**, odnosno radi izmene njenog sadržaja. Takođe, podaci, sadržani u datoteci, ne mogu biti korišćeni, ako se onemogući pristup datoteci, radi **čitanja**, odnosno radi preuzimanja njenog sadržaja. Na ovaj način uvedeno **pravo pisanja** i **pravo čitanja** datoteke omogućuju da se za svakog korisnika jednostavno ustanovi koja vrsta rukovanja datotekom mu je dozvoljena, a koja ne. Tako, korisniku, koji ne poseduje pravo pisanja datoteke, nisu dozvoljena rukovanja datotekom, koja izazivaju izmenu njenog sadržaja. Ili, korisniku, koji ne poseduje pravo čitanja datoteke, nisu dozvoljena rukovanja datotekom, koja zahtevaju preuzimanje njenog sadržaja.

Za izvršne datoteke uskraćivanje prava čitanja je prestrogo, jer sprečava ne samo neovlašćeno uzimanje tuđeg izvršnog programa, nego i njegovo **izvršavanje**. Zato je uputno, radi izvršnih datoteka, uvesti posebno **pravo izvršavanja** programa, sadržanih u izvršnim datotekama. Zahvaljujući posedovanju ovog prava, korisnik može da pokrene izvršavanje programa, sadržanog u izvršnoj datoteci, i onda kada nema pravo njenog čitanja.

Pravo čitanja, pravo pisanja i pravo izvršavanja datoteke predstavljaju tri prava pristupa datotekama (*file access control*), na osnovu kojih se za svakog korisnika utvrđuje koje vrste rukovanja datotekom su mu dopuštene. Da se za svaku datoteku ne bi evidentirala prava pristupa za svakog korisnika pojedinačno, uputno je sve korisnike razvrstati u kategorije i za svaku od njih vezati pomenuta prava pristupa. Iskustvo pokazuje da su dovoljne tri kategorije korisnika. Jednoj pripada vlasnik datoteke, drugoj njegovi saradnici, a trećoj ostali korisnici. Nakon razvrstavanja korisnika u tri kategorije, evidentiranje prava pristupa datotekama omogućuje **matrica zaštite** (*protection matrix*, *access matrix*). Ona ima tri kolone (po jednu za svaku kategoriju korisnika) i onoliko

redova koliko ima datoteka (Slika 8.2). U preseku svakog reda i svake kolone matrice zaštite navode se prava pristupa datoteci iz posmatranog reda za korisnike koji pripadaju kategoriji iz posmatrane kolone.

	vlasnik	saradnik	ostali
datoteka_1.bin	pisanje čitanje izvršavanje	- čitanje izvršavanje	- - izvršavanje
datoteka_2.bin	- čitanje izvršavanje	- - izvršavanje	- - izvršavanje
	...	...	...
datoteka_n.txt	pisanje čitanje -	- čitanje -	- - -

Slika 8.2: Matrica zaštite (po jedna za svakog vlasnika)

U primeru matrice zaštite (Slika 8.2) vlasnik datoteke **datoteka_1.bin** ima sva prava pristupa, njegovi saradnici nemaju pravo pisanja, a ostali korisnici imaju samo pravo izvršavanja (pretpostavka je da je reč o izvršnoj datoteci). Ima smisla uskratiti i vlasniku neka prava, na primer, da ne bi nehotice izmenio sadržaj datoteke **datoteka_2.bin**, ili da ne bi pokušao da izvrši datoteku koja nije izvršna (**datoteka_n.txt**).

Prava pristupa iz matrice zaštite se mogu vezati (1) za datoteke i čuvati u deskriptorima datoteka, ili (2) vezati za korisnike. U prvom slučaju redovi matrice zaštite su raspoređeni po deskriptorima raznih datoteka, a u drugom slučaju elemente kolona matrice zaštite čuvaju pojedini korisnici.

Za uspeh izloženog koncepta zaštite datoteka neophodno je onemogućiti neovlašteno menjanje matrice zaštite. Jedino vlasnik datoteke sme da zadaje i menja prava pristupa sebi, svojim saradnicima i ostalim korisnicima. Zato je potrebno znati za svaku datoteku ko je njen vlasnik. Takođe, potrebno je i razlikovanje korisnika, da bi se među njima mogao prepoznati vlasnik datoteke. To se postiže tako što svoju aktivnost svaki korisnik započinje svojim **predstavljanjem** (*login*). U toku predstavljanja korisnik predočava svoje ime i navodi dokaz da je on osoba za koju se predstavlja, za šta je, najčešće, dovoljna lozinka. Predočeno ime i navedena lozinka se porede sa spiskom imena i spiskom za njih vezanih lozinki registrovanih korisnika. Predstavljanje je uspešno, ako se u spiskovima imena i lozinki registrovanih korisnika pronađu predočeno ime i navedena lozinka.

Predstavljanje korisnika se zasniva na pretpostavci da su njihova imena javna, ali da su im lozinke tajne. Zato je i spisak imena registrovanih korisnika javan, a spisak lozinki registrovanih korisnika tajan, znači, direktno nepristupačan korisnicima. Jedina dva slučaja, u kojima ima smisla dozvoliti korisnicima posredan pristup spisku lozinki, su: (1) radi njihovog predstavljanja i (2) radi izmene njihove lozinke.

Za predstavljanje korisnika uvodi se posebna operacija, koja omogućuje samo proveru da li se zadani par (ime, lozinka) može pronaći u spiskovima imena i lozinki registrovanih korisnika. Slično, za izmenu lozinki uvodi se posebna operacija, koja omogućuje samo promenu lozinke onome ko zna postojeću lozinku. Sva druga rukovanja spiskovima imena i lozinki registrovanih korisnika, kao što su ubacivanje u ove spiskove novih parova (ime, lozinka), ili njihovo izbacivanje iz ovih spiskova, nalaze se u nadležnosti poverljive osobe, koja se naziva **administrator** (*superuser*). Zaštita datoteka zavisi od odgovornosti i poverljivosti administratora.

Nakon prepoznavanja korisnika, odnosno, nakon njegovog uspešnog predstavljanja, uz pomoć matrice zaštite moguće je ustanoviti koja prava pristupa korisnik poseduje za svaku datoteku. Da bi se pojednostavila provera korisničkih prava pristupa, uputno je, umesto imena korisnika, uvesti njegovu **numeričku oznaku**. Radi klasifikacije korisnika zgodno je da ovu numeričku oznaku obrazuju dva redna broja. Prvi od njih označava korisnika, a drugi od njih označava grupu kojoj korisnik pripada. Podrazumeva se da su svi korisnici iz iste grupe međusobno saradnici. Prema tome, **redni broj korisnika** (*UID, User IDentification*) jednoznačno određuje vlasnika. Saradnici vlasnika su svi korisnici koji imaju isti **redni broj grupe** (*GID, Group IDentification*) kao i vlasnik. U ostale korisnike spadaju svi korisnici čiji je redni broj grupe različit od rednog broja grupe vlasnika. Posebna grupa se rezerviše za administratore.

Numerička oznaka korisnika pojednostavljuje proveru njegovog prava pristupa datoteci. Ipak, da se takva provera ne bi obavljala prilikom svakog pristupa datoteci, umesno je takvu proveru obaviti samo pre prvog pristupa. To je zadatak operacije otvaranja datoteke, koja prethodi operacijama, kao što su pisanje ili čitanje datoteke. Pomoću operacije otvaranja se saopštava i na koji način korisnik namerava da koristi datoteku. Ako je njegova namera u skladu sa njegovim pravima, otvaranje datoteke je uspešno, a pristup datoteci je dozvoljen, ali samo u granicama iskazanih namera. Iza operacija pisanja ili čitanja datoteke sledi operacija zatvaranja datoteke, pomoću koje korisnik saopštava da završava korišćenje datoteke. Nakon zatvaranja datoteke, pristup datoteci nije dozvoljen do njenog narednog otvaranja.

Numerička oznaka vlasnika datoteke i prava pristupa korisnika iz pojedinih klasa predstavljaju atribute datoteke.

Zaštita datoteka uvodi pojam **sigurnost** (*security*) koji se odnosi na uspešnost zaštite od neovlašćenog korišćenja ne samo datoteka, nego i ostalih delova računara kojima upravlja operativni sistem. Sigurnost se bavi načinima prepoznavanja ili identifikacije korisnika (*authentication*), kao i načinima provere njihovih prava pristupa (*authorization*). Operativni sistem nudi mehanizme sigurnosti pomoću kojih mogu da se ostvare različite politike sigurnosti.

8.4 PITANJA

1. Na šta ukazuje ime datoteke?
2. Od koliko delova se sastoji ime datoteke?
3. Od koliko delova se sastoji ime imenika?
4. Šta obuhvata rukovanje datotekom?

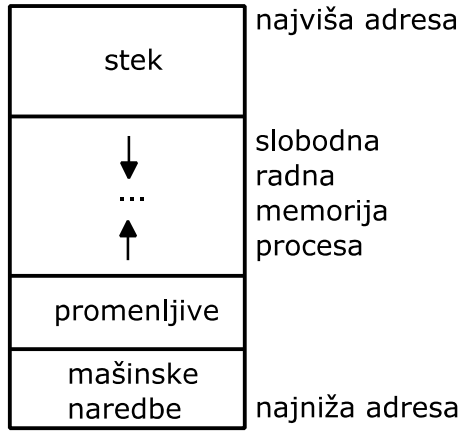
5. Šta karakteriše hijerarhijsku organizaciju datoteka?
6. Šta važi za apsolutnu putanju?
7. Šta važi za relativnu putanju?
8. Koje datoteke obrazuju sistem datoteka?
9. Koja su prava pristupa datotekama?
10. Koje kolone ima matrica zaštite?
11. Čemu je jednak broj redova matrice zaštite?
12. Gde se mogu čuvati prava pristupa iz matrice zaštite?
13. Šta je potrebno za sprečavanje neovlašćenog menjanja matrice zaštite?
14. Kada korisnici mogu posredno pristupiti spisku lozinki?
15. Koju dužnost imaju administratori?
16. Šta sadrži numerička oznaka korisnika?
17. Kakvu numeričku oznaku imaju saradnici vlasnika datoteke?
18. Kakvu numeričku oznaku imaju ostali korisnici?
19. Kada se obavlja provera prava pristupa datoteci?
20. Čime se bavi sigurnost?

9 SLOJ OPERATIVNOG SISTEMA ZA RUKOVANJE PROCESIMA

9.1 OSNOVNI ZADACI SLOJA ZA RUKOVANJE PROCESIMA

Osnovni zadaci sloja za rukovanje procesima su stvaranje i uništenje procesa. Stvaranje procesa obuhvata stvaranje njegove slike i njegovog deskriptora, kao i pokretanje njegove aktivnosti. Uništenje procesa obuhvata zaustavljanje njegove aktivnosti, kao i uništenje njegove slike i njegovog deskriptora. Pored sistemskih operacija stvaranja i uništenja procesa, potrebne su i sistemske operacije za izmenu atributa procesa, na primer, za izmenu radnog imenika procesa.

Slika procesa (Slika 9.1) obuhvata niz lokacija radne memorije sa uzastopnim (logičkim) adresama. Ona sadrži izvršavane mašinske naredbe, promenljive i stek.



Slika 9.1: Grafička predstava slike procesa

Podrazumeva se da slika procesa započinje od lokacije radne memorije sa najnižom adresom, od koje započinju mašinske naredbe, a završava na lokaciji radne memorije sa najvišom adresom, na kojoj započinje stek. Pri tome se podrazumeva da se stek širi (puni) u smeru nižih adresa. Iza mašinskih naredbi dolaze statičke promenljive (i to prvo inicijalizovane, pa neinicijalizovane). Između ovih promenljivih i steka se nalazi slobodna radna memorija procesa. Ona je na raspolaganju procesu za širenje (punjenje) steka, ali i za stvaranje dinamičkih promenljivih (deo slobodne radne memorije procesa, koji se koristi za stvaranje dinamičkih promenljivih, se zove na engleskom *heap*). Svi dinamički zahtevi za zauzimanjem radne memorije, postavljeni u toku aktivnosti procesa, se zadovoljavaju samo na račun slobodne radne memorije procesa. Ovakva organizacija slike procesa uslovljava da proces prvo ugrozi svoju aktivnost, kada njegovi zahtevi za radnom memorijom nadmaše njegovu raspoloživu slobodnu radnu memoriju. Na primer, ako ne postoji način da se automatski ustanovi, prilikom širenja steka, da su

zahtevi za radnom memorijom nadmašili slobodnu radnu memoriju procesa, tada dolazi do preklapanja steka i promenljivih, sa fatalnim ishodom po aktivnost procesa.

Pored slike, za aktivnost procesa je važan i deskriptor procesa, koji sadrži atribute procesa. Ovi atributi karakterišu aktivnost procesa. Oni obuhvataju:

1. stanje procesa ("spreman", "aktivan", "čeka"),
2. sadržaje procesorskih registara (zatečene u njima pre poslednjeg preključivanja procesora sa procesa),
3. numeričku oznaku vlasnika procesa,
4. oznaku procesa stvaraoca,
5. trenutak pokretanja aktivnosti procesa,
6. ukupno trajanje aktivnosti procesa (odnosno, ukupno vreme angažovanja procesora),
7. podatke o slici procesa (njenoj veličini i njenom položaju u radnoj i masovnoj memoriji),
8. podatke o datotekama koje proces koristi,
9. podatak o radnom imeniku procesa i
10. razne podatke neophodne za upravljanje aktivnošću procesa (poput prioriteta procesa ili položaja sistemskog steka procesa, koga koristi operativni sistem u toku obavljanja sistemskih operacija).

9.2 SISTEMSKE OPERACIJE ZA STVARANJE I UNIŠTENJE PROCESA

Za stvaranje procesa potrebno je pristupiti odgovarajućoj izvršnoj datoteci sa inicijalnom slikom procesa, koja, između ostalog, sadrži mašinske naredbe i početne vrednosti (inicijalizovanih) statičkih promenljivih programa, ali i podatak o veličini (pojedinih delova) slike procesa. Takođe, potrebno je zauzeti deskriptor procesa, kao i dovoljno veliku zonu radne memorije za sliku procesa. Sve to, kao i pravljenje slike procesa na osnovu njegove inicijalne slike, odnosno popunjavanje atributa u njegov deskriptor, spada u nadležnost sistemske operacije stvaranja procesa (**fork()** i **exec()**). Ovu operaciju poziva proces stvaralac i ona se obavlja u toku njegove aktivnosti. U okviru poziva sistemske operacije stvaranja procesa kao argument se navodi putanja odgovarajuće izvršne datoteke. Svi atributi stvaranog procesa ne moraju biti navedeni u okviru poziva sistemske operacije stvaranja procesa, jer se jedan njihov deo nasleđuje iz deskriptora procesa stvaraoca (na primer, numerička oznaka vlasnika procesa, podatak o radnom imeniku procesa ili njegov prioritet), a jedan deo nastaje u toku stvaranja procesa (na primer, podaci o slici procesa). Kada se, u okviru stvaranja procesa, stigne do pokretanja njegove aktivnosti, moguće je preključivanje procesora sa procesa stvaraoca na stvarani proces. To se desi, ako je prioritet stvaranog procesa viši od prioriteta procesa stvaraoca. U tom slučaju, proces stvaralac dospeva među spremne procese. Inače tamo dospeva stvarani proces.

Za uništenje procesa potrebno je osloboditi njegov deskriptor i zonu radne memorije sa njegovom slikom. Ovo spada u nadležnost sistemske operacije uništenja procesa (**exit()**). Nju automatski poziva proces na kraju svoje aktivnosti, čime izaziva svoje samouništenje. Uništenje procesa se završava preključivanjem procesora sa uništavanog

na neki od spremnih procesa. U okviru poziva sistemske operacije uništenja procesa uputno je predvideti argument, posredstvom koga uništavani proces može da saopšti svom stvaraocu svoje završno stanje, odnosno informaciju da li je aktivnost uništavanog procesa bila uspešna ili ne. Jasno, da bi proces stvaralac mogao iskoristiti ovakvu povratnu informaciju od stvorenog procesa, on mora pozivom posebne sistemske operacije (**wait()**) da zatraži zaustavljanje svoje aktivnosti i tako omogući preključivanje procesora na stvarani proces. U ovom slučaju, proces stvaralac ne dospeva među spremne procese, nego među procese u stanju čekanja, jer čeka kraj aktivnosti stvorenog procesa. Takođe, u ovom slučaju sistemska operacija uništenja stvorenog procesa ima i zadatak da prevede proces stvaralac među spremne procese i tako omogući nastavak njegove aktivnosti.

Sistemska operacija uništenja procesa se automatski poziva, kada se desi nepopravljiv prekid (izuzetak) u toku aktivnosti procesa.

9.3 ZAMENA SLIKA PROCESA

Najveći mogući broj slika procesa, koje mogu da istovremeno postoje u radnoj memoriji, se naziva **stepen multiprogramiranja** (*degree of multiprogramming*). Što je stepen multiprogramiranja viši, to je i veća verovatnoća da je procesor zaposlen, jer je veća verovatnoća da postoji spreman proces. Stepen multiprogramiranja zavisi od veličine radne memorije. Kada broj istovremeno postojećih procesa dostigne stepen multiprogramiranja, stvaranje novih procesa postaje problematično. U ovoj situaciji moguće rešenje je da se slika nekog od (manje prioritetnih) postojećih procesa privremeno izbaci u masovnu memoriju i tako oslobodi prostor u radnoj memoriji za sliku novog procesa. Predloženi pristup podrazumeva da je broj deskriptora procesa veći od stepena multiprogramiranja i da su svi deskriptori stalno prisutni u radnoj memoriji, što ne predstavlja problem, jer deskriptori procesa ne zauzimaju mnogo radne memorije. Prethodno opisani način oslobađanja prostora za sliku procesa u radnoj memoriji uzrokuje da, uz sliku procesa u radnoj memoriji, obavezno postoji i njena kopija u masovnoj memoriji. Pošto se, u toku aktivnosti procesa, menja samo deo njegove slike u radnoj memoriji, jer se menjaju samo vrednosti njegovih promenljivih i njegov stek, prilikom izbacivanja slike procesa potrebno je samo njen izmenjeni deo prebacivati u kopiju slike u masovnoj memoriji. Ali, pri vraćanju slike u radnu memoriju, prebacuje se cela njena kopija, da bi se u radnoj memoriji obnovila cela slika procesa. Do vraćanja slike procesa u radnu memoriju dolazi, kada se u njoj oslobodi prostor, pa je vraćanje slike jednog procesa vezano za uništavanje drugog procesa, jer se tada oslobađa prostor u radnoj memoriji.

Podaci o kopiji slike procesa (koja nastaje istovremeno sa nastankom procesa, odnosno sa nastankom njegove slike, ili prilikom njenog prvog izbacivanja, a nestaje istovremeno sa nestankom procesa, odnosno sa nestankom njegove slike) se čuvaju u deskriptoru procesa, zajedno sa podacima o slici procesa. Prilikom izbacivanja slike procesa, u deskriptoru procesa se naznačava da se njegova slika ne nalazi više u radnoj memoriji. Da procesi, čije su slike izbačene van radne memorije, ne bi bili dugo zapostavljeni, uputno je periodično vršiti **zamenu slika procesa** (*swapping*), znači

izbacivati sliku jednog procesa, radi ubacivanja slike drugog procesa. Za to je potrebna posebna operacija za zamenu slika procesa. Ona ne spada obavezno u sistemske operacije. U nadležnosti ove operacije je **dugoročno raspoređivanje** (*long term scheduling*), u okviru koga se odabira proces, čija slika se izbacuje, kao i proces, čija slika se ubacuje. Važno je uočiti da se dugoročno raspoređivanje razlikuje od običnog ili **kratkoročnog raspoređivanja** (*short term scheduling*), koje među spremnim procesima odabira proces na koga se prekljućuje procesor.

9.4 RUKOVANJE NITIMA

Rukovanje nitima može, ali i ne mora, biti u nadležnosti sloja za rukovanje procesima. Kada je rukovanje nitima povereno sloju za rukovanje procesima, tada operativni sistem nudi sistemske operacije za rukovanje nitima, koje omogućuju stvaranje, uništavanje i sinhronizaciju niti. U ovom slučaju, deskriptori i sistemski stek niti se nalaze u sistemskom prostoru, dok se sopstveni stek niti nalazi u korisničkom prostoru (unutar slike procesa).

U slučaju kada rukovanje nitima nije u nadležnosti operativnog sistema, brigu o nitima potpuno preuzima konkurentna biblioteka. Pošto ona pripada slici procesa, rukovanje nitima u ovom slučaju se potpuno odvija u korisničkom prostoru, u kome se nalaze i deskriptori niti, kao i stekovi niti.

Osnovna prednost rukovanja nitima van operativnog sistema je efikasnost, jer su pozivi potprograma konkurentne biblioteke brži (kraći) od poziva sistemskih operacija. Ali, kada operativni sistem ne rukuje nitima, tada poziv blokirajuće sistemske operacije iz jedne niti dovodi do zaustavljanja aktivnosti procesa kome ta nit pripada, jer operativni sistem pripisuje sve pozive sistemskih operacija samo procesima, pošto ne registruje postojanje niti. Na taj način se sprečava konkurentnost unutar procesa, jer zaustavljanje aktivnosti procesa sprečava aktivnost njegovih spremnih niti. Ova mana rukovanja nitima van operativnog sistema ozbiljno umanjuje praktičnu vrednost ovakvog pristupa. Zato savremeni operativni sistemi podržavaju rukovanje nitima. Tako, u okviru *POSIX* (*Portable Operating System Interface*) standarda (IEEE 1003 ili ISO/IEC 9945) postoji deo *pthread* (*POSIX threads*) koji je posvećen nitima.

Sa stanovišta ostalih slojeva operativnog sistema nema suštinske razlike između procesa i niti, pa se u nastavku izlaganja pominju samo procesi.

9.5 OSNOVA SLOJA ZA RUKOVANJE PROCESIMA

Sloj za rukovanje procesima se oslanja na sloj za rukovanje datotekama, radi pristupa sadržaju izvršne datoteke, ali i radi rukovanja kopijama slika procesa, jer se za njih rezerviše posebna datoteka. Pored toga, sloj za rukovanje procesima se oslanja i na sloj za rukovanje radnom memorijom, radi zauzimanja i oslobađanja zona radne memorije, potrebnih za smeštanje slika procesa. Na kraju, sloj za rukovanje procesima se oslanja i na sloj za rukovanje procesorom, jer do preključivanja dolazi prilikom stvaranja i uništenja procesa.

9.6 PITANJA

1. Šta omogućuju sistemske opercije za rukovanje procesima?
2. Šta obuhvata stvaranje procesa?
3. Šta obuhvata uništenje procesa?
4. Šta sadrži slika procesa?
5. Za šta se koristi slobodna radna memorije procesa?
6. Koji atributi procesa postoje?
7. Koje sistemske operacije za rukovanje procesima postoje?
8. Koji se atributi nasleđuju od procesa stvaraoca prilikom stvaranja procesa?
9. Koji se atributi procesa nastanu prilikom njegovog stvaranja?
10. U kojim stanjima može biti proces stvaraoc nakon stvaranja novog procesa?
11. Šta je stepen multiprogramiranja?
12. Šta karakteriše kopiju slike procesa?
13. Koje raspoređivanje je vezano za zamenu slika procesa?
14. Šta karakteriše rukovanje nitima unutar operativnog sistema?
15. Šta karakteriše rukovanje nitima van operativnog sistema?

10 SISTEMSKI PROCESI

10.1 ULOGA SISTEMSKIH PROCESA

Za obavljanje pojedinih zadataka operativnog sistema prirodno je koristiti procese. Ovakvi procesi se nazivaju **sistemski procesi** (*daemon*), jer su u službi operativnog sistema.

10.2 NULTI PROCES

Tipičan primer sistemskog procesa je **nulti** ili beskonačni (*idle*) proces, na koga se procesor prekljućuje, kada ne postoji drugi spreman proces. Znači, beskonačan proces ima zadatak da zaposli procesor, kada nema mogućnosti za korisnu upotrebu procesora. U toku aktivnosti beskonačnog procesa izvršava se beskonačna petlja, što znači da je beskonačan proces uvek ili spreman ili aktivan (on ne prelazi u stanje čekanja). Njegov prioritet je niži od prioriteta svih ostalih procesa, a on postoji za sve vreme aktivnosti operativnog sistema.

10.3 PROCES DUGOROČNI RASPOREĐIVAČ

Drugi primer sistemskog procesa je proces **dugoročni raspoređivač** (*swapper*), koji se brine o zameni slika procesa (jasno, kada za to ima potrebe). On se periodično aktivira, radi pozivanja operacije za zamenu slika procesa. Nakon toga, ovaj proces se uspava, odnosno, njegova aktivnost se zaustavlja, dok ne nastupi trenutak za njegovo novo aktiviranje. Da bi se proces uspavao, odnosno da bi se njegova aktivnost zaustavila do nastupanja zadanog trenutka, on poziva odgovarajuću sistemsku operaciju. Ona pripada sloju za rukovanje kontrolerima, jer proticanje vremena registruje drajver sata. I proces dugoročni raspoređivač postoji za sve vreme aktivnosti operativnog sistema (jasno, ako ima potrebe za dugoročnim raspoređivanjem).

10.4 PROCESI IDENTIFIKATOR I KOMUNIKATOR

U sistemske procese spada i **proces identifikator** (*login process*), koji podržava predstavljanje korisnika. Proces identifikator opslužuje terminal, da bi posredstvom njega stupio u interakciju sa korisnikom u toku predstavljanja, radi preuzimanja imena (koje se prikazuje na ekranu) i lozinke (koja se ne prikazuje na ekranu) korisnika. Po preuzimanju imena i lozinke, proces identifikator proverava njihovu ispravnost i, ako je prepoznao korisnika, tada stvara **proces komunikator**, koji nastavlja interakciju sa korisnikom. Pri tome, proces identifikator prepušta svoj terminal stvorenom procesu komunikatoru i zaustavlja svoju aktivnost. Ona se nastavlja tek nakon završetka aktivnosti procesa komunikatora. Tada proces identifikator opet preuzima opsluživanje terminala, da bi podržao novo predstavljanje korisnika. Prema tome, i proces identifikator postoji za sve vreme aktivnosti operativnog sistema.

Za proveru ispravnosti imena i lozinke korisnika, neophodno je raspolagati spiskovima imena i lozinki registrovanih korisnika. Ovi spiskovi se čuvaju u posebnoj

datoteci lozinki (*password file*). Svaki slog ove datoteke sadrži:

1. ime i lozinku korisnika,
2. numeričku oznaku korisnika,
3. putanju radnog imenika korisnika i
4. putanju izvršne datoteke, sa inicijalnom slikom korisničkog procesa komunikatora.

Nekada se lozinke korisnika čuvaju u posebnoj datoteci (*shadow file*).

Za stvaranje procesa komunikatora proces identifikator koristi putanju izvršne datoteke koja sadrži inicijalnu sliku korisničkog procesa komunikatora. Tom prilikom on upotrebi numeričku oznaku prepoznatog korisnika kao numeričku oznaku vlasnika stvaranog procesa komunikatora, a putanju radnog imenika prepoznatog korisnika upotrebi kao putanju radnog imenika stvaranog procesa komunikatora.

Slogovi datoteke lozinki se, zbog jednoznačnosti, međusobno razlikuju obavezno po imenima i po numeričkim oznakama korisnika. Međutim, oni se mogu razlikovati po putanjama korisničkih (početnih) radnih imenika, pa i po putanjama izvršnih datoteka (sa inicijalnim slikama korisničkih procesa komunikatora). Zahvaljujući tome, svaki korisnik može da ima poseban (sopstveni) radni imenik, ali i poseban (sopstveni) proces komunikator. Za procese komunikatore, koji su prilagođeni posebnim potrebama pojedinih korisnika (znači, kojima ne odgovara standardni interpreter komandnog jezika), se ne smatra da su sistemski procesi, jer oni ne predstavljaju sastavni deo operativnog sistema, iako obavljaju ulogu sloja za spregu sa korisnikom. Zbog ovakvih procesa komunikatora, moguće je zauzeti stanovište da sloj za spregu sa korisnikom i nije deo operativnog sistema, nego da pripada višem (korisničkom) sloju, kome, između ostalog, pripadaju i sistemski programi, kao što su tekst editor ili kompajler.

Iz funkcije procesa komunikatora sledi da on ostvaruje interaktivni nivo komunikacije korisnika i operativnog sistema. Proces komunikator prihvata sve komande, koje dolaze sa terminala i tretira ih kao komande prepoznatog korisnika (koga je prepoznao proces identifikator).

Za proces komunikator (kao i za svaki drugi proces) prava pristupa datotekama se određuju na osnovu numeričke oznake njegovog vlasnika. Procesi, koje stvara proces komunikator (radi izvršavanja pojedinih komandi korisnika) imaju ista prava kao i proces komunikator, jer se podrazumeva da stvoreni procesi nasleđuju numeričku oznaku vlasnika procesa stvaraoca. Prema tome, dok je u interakciji sa procesom komunikatorom, njegov vlasnik nema mogućnosti za narušavanje zaštite datoteka. Ali, ako, umesto vlasnika, u interakciju sa procesom komunikatorom stupi neki drugi korisnik, zaštita datoteka se narušava, jer ovaj drugi korisnik dobija priliku da uživa tuđa prava pristupa datotekama (koja pripadaju vlasniku procesa komunikatora). Zato, nakon predstavljanja, korisnik sme prepustiti svoj terminal drugom korisniku, tek nakon što je završio aktivnost svog procesa komunikatora. Radi toga, među komandama procesa komunikatora, obavezno postoji posebna komanda, namenjena baš procesu komunikatoru, a koja izaziva njegovo uništenje. Znači, svaki korisnik započinje rad **prijavom**, u toku koje se predstavi i pokrene aktivnost svog procesa komunikatora, a završi rad **odjavom**, u toku koje okonča aktivnost svog procesa komunikatora.

Za zaštitu datoteka ključno je onemogućiti neovlaštene pristupe datoteci lozinki. Prirodno je da njen vlasnik bude administrator i da jedino sebi dodeli pravo čitanja i pisanja ove datoteke. Pošto su procesi identifikatori sistemski procesi, koji nastaju pri pokretanju operativnog sistema, nema prepreke da njihov vlasnik bude administrator. Iako na taj način procesi identifikatori dobijaju i pravo čitanja i pravo pisanja datoteke lozinki, to pravo korisnici ne mogu da zloupotrebe, jer, posredstvom procesa identifikatora, jedino mogu proveriti da li su registrovani u datoteci lozinki. Pri tome je bitno da svoja prava proces identifikator ne prenosi na proces komunikator. Zato proces komunikator ne nasleđuje numeričku oznaku vlasnika od svog stvaraoca procesa identifikatora.

Administrator bez problema pristupa datoteci lozinki, radi izmene njenog sadržaja, jer je on vlasnik svih procesa, koje je stvorio i pokrenuo da bi izvršili njegove komande. Pošto korisnici nemaju načina da pristupe datoteci lozinki, javlja se problem kako omogućiti korisniku da sam izmeni svoju lozinku. Taj problem se može rešiti po uzoru na proces identifikator, koga koriste svi korisnici, iako nisu njegovi vlasnici. Prema tome, ako je administrator vlasnik izvršne datoteke sa inicijalnom slikom procesa za izmenu lozinki, dovoljno je naznačiti da on treba da bude vlasnik i procesa nastalog na osnovu ove izvršne datoteke (*SUID - Switch User IDentification program*). Zahvaljujući tome, vlasnik ovakvog procesa je administrator, bez obzira ko je stvorio proces. U tom slučaju, nema smetnje da korisnik, koji izazove stvaranje procesa za izmenu lozinke, pristupi datoteci lozinki. Pri tome, proces za izmenu lozinki dozvoljava korisniku samo da izmeni sopstvenu lozinku, tako što od korisnika primi njegovo ime, važeću i novu lozinku, pa, ako su ime i važeća lozinka ispravni, on važeću lozinku zameni novom.

Datoteka lozinki je dodatno zaštićena, ako su lozinke u nju upisane u izmenjenom, odnosno u **kriptovanom** (*encrypted*) obliku, jer tada administrator može da posmatra sadržaj datoteke lozinki na ekranu, ili da ga štampa, bez straha da to može biti zloupotrebljeno. Da bi se sprečilo pogađanje tuđih lozinki, proces identifikator (ili proces za izmenu lozinki) treba da reaguje na više uzastopnih neuspešnih pokušaja predstavljanja, obaveštavajući o tome administratora, ili odbijajući neko vreme da prihvati nove pokušaje predstavljanja. Takođe, korisnici moraju biti oprezni da sami ne odaju svoju lozinku lažnom procesu identifikatoru. To se može desiti, ako se njihov prethodnik ne odjavi, nego ostavi svoj proces da opslužuje terminal, oponašajući proces identifikator. Ovakvi procesi se nazivaju **trojanski konji** (*trojan horse*).

Na kraju, važno je uočiti da, zbog istovremenog postojanja više procesa i nepredvidivosti preključivanja, postoji nezanemarljiva mogućnost da više procesa istovremeno pokuša da pristupi datoteci lozinki. Ako su to samo procesi identifikatori, to i nije problematično, jer oni samo preuzimaju njen sadržaj. Ali, ako ovoj datoteci (ili bilo kojoj drugoj) istovremeno pokušaju pristupiti procesi koji menjaju njen sadržaj, ili procesi koji preuzimaju i/ili menjaju njen sadržaj, rezultat pristupa je nepredvidiv, znači zavisan od redosleda pristupa i različit od rezultata potpuno sekvencijalnog pristupa. Nepredvidivost rezultata pristupa je posledica činjenice da preključivanja mogu da učine vidljivim samo delimično izmenjen sadržaj datoteke, što je nemoguće u slučaju potpuno sekvencijalnog pristupa (kada su vidljive samo celovite izmene sadržaja datoteke). Vidljivost delimično izmenjenog sadržaja datoteke uzrokuje da ukupna izmena može da

bude posledica delimičnih izmena, napravljenih u toku aktivnosti raznih procesa, što je neprihvatljivo. Iz istog razloga moguće je preuzimanje dela novog (izmenjenog) i dela starog (neizmenjenog) sadržaja datoteke, što je, takođe, neprihvatljivo. Zato je potrebno sinhronizovati procese koji pristupaju sadržaju iste datoteke.

Proces identifikator je jedan od delova operativnog sistema koji su zaduženi za sigurnost, a kriptovanje je tehnika koja doprinosi povećanju sigurnosti.

10.5 POJAM KRIPTOGRAFIJE (*CRYPTOGRAPHY*)

Cilj kriptovanja teksta, poput, na primer, lozinke, je da tekst nakon kriptovanja postane nerazumljiv (nečitljiv) za neupućenu osobu. Kriptovanje menja tekst po unapred dogovorenom algoritmu kriptovanja, uz korišćenje zadatog ključa kriptovanja. Tako, na primer, ako algoritam kriptovanja vrši zamenu znakova teksta dvocifrenim brojevima, tada ključ kriptovanja ima oblik niza dvocifrenih brojeva, koji korespondiraju znakovima. Pretpostavka je da je redosled znakova unapred zadan. U ovom primeru ključeva kriptovanja ima koliko i različitih nizova dvocifrenih brojeva koji korespondiraju znakovima. Da bi kriptovani tekst postao razumljiv, potrebno ga je **dekriptovati** (*decrypt*), odnosno vratiti u prvobitni oblik. To omogućuju algoritam dekriptovanja i ključ dekriptovanja. Ako algoritam dekriptovanja direktno i jednoznačno sledi iz algoritma kriptovanja, a ključ dekriptovanja iz ključa kriptovanja, tada je reč o **simetričnoj kriptografiji** (*symmetric-key cryptography, secret-key cryptography*). Tako, iz prethodno pomenutog primera algoritma kriptovanja sledi da se algoritam dekriptovanja sastoji od zamene dvocifrenih brojeva odgovarajućim znakovima, a da ključ dekriptovanja ima oblik niza znakova koji korespondiraju dvocifrenim brojevima. Pretpostavka je da je redosled dvocifrenih brojeva unapred zadan.

Osobina simetrične kriptografije je da poznavanje ključa kriptovanja omogućuje i dekriptovanje. Zato, kod simetrične kriptografije, ključ kriptovanja predstavlja tajnu kao i ključ dekriptovanja.

Algoritmi kriptovanja i dekriptovanja ne predstavljaju tajnu, jer je praksa pokazala da se takva tajna ne može sačuvati. Prema tome, tajnost kriptovanja teksta se zasniva na činjenici da (1) komplikovanost algoritama kriptovanja i dekriptovanja, (2) velika dužina ključeva kriptovanja i dekriptovanja, kao i (3) veliki broj ovih ključeva čine praktično neizvodljivim pokušaj da se "na silu" dekriptuje kriptovani tekst probanjem, jednog po jednog, svih mogućih ključeva dekriptovanja.

Simetrična kriptografija nije podesna za kriptovanje poruka, jer tada ključ kriptovanja mora znati svaki pošiljalac poruke, što ga dovodi u poziciju da može da dekriptuje poruke drugih pošiljalaca. To nije moguće u **asimetričnoj kriptografiji** (*public-key cryptography*), jer je njena osobina da se iz ključa kriptovanja ne može odrediti ključ dekriptovanja, pa poznavanje ključa kriptovanja ne omogućuje dekriptovanje. Ovakav ključ kriptovanja se zove **javni ključ** (*public key*), jer je on dostupan svima. Njemu odgovarajući ključ dekriptovanja je privatan (tajan), jer je dostupan samo osobama ovlašćenim za dekriptovanje. Zato se on naziva **privatni ključ** (*private key*). Prema tome, svaki pošiljalac poruke raspolaže javnim ključem, da bi mogao da kriptuje poruke, dok privatni ključ poseduje samo primalac poruka, da bi jedini mogao da dekriptuje

poruke. Asimetrična kriptografija se temelji na korišćenju jednostavnih algoritama kriptovanja kojima odgovaraju komplikovani algoritmi dekriptovanja. Zato je asimterična kriptografija mnogo sporija od simetrične.

U praksi se simetrična kriptografija zasniva na DES (*Data Encryption Standard*) i IDEA (*International Data Encryption Algorithm*) pristupima, a asimetrična kriptografija na RSA algoritmu (čiji naziv potiče od prvih slova prezimena njegovih autora). Obično se asimetrična kriptografija koristi samo za razmenu ključeva potrebnih za simetričnu kriptografiju, pomoću koje se, zatim, kriptuju i dekriptuju poruke.

10.6 PITANJA

1. Šta karakteriše nulti proces?
2. Šta je karakteristično za proces dugoročni raspoređivač?
3. Šta radi proces identifikator?
4. Ko stvara proces komunikator?
5. Šta sadrži slog datoteke lozinki?
6. Šta označava SUID (*switch user identification*)?
7. Šta je neophodno za podmetanje trojanskog konja?
8. Šta karakteriše simetričnu kriptografiju?
9. Šta karakteriše asimetričnu kriptografiju?
10. Na čemu se temelji tajnost kriptovanja?

11 SLOJ OPERATIVNOG SISTEMA ZA RUKOVANJE DATOTEKAMA

11.1 ZADACI SLOJA ZA RUKOVANJE DATOTEKAMA

Zadatak sloja za rukovanje datotekama je da ozbezbedi punu slobodu rukovanja podacima, sadržanim u datotekama. Punu slobodu rukovanja podacima nudi predstava sadržaja datoteke kao niza bajta, pod uslovom da se ovaj niz može, po potrebi, menjati, kao i da se bajtima iz niza može pristupati u proizvoljnom redosledu, korišćenjem rednog broja bajta za njegovu identifikaciju. Ovakva predstava datoteke dozvoljava da se iznad nje izgrade različiti pogledi na datoteku, na primer, da se datoteka posmatra kao skup slogova iste ili različite dužine, koji se identifikuju pomoću posebnih indeksa. Oblikovanje ovakvih specijalizovanih pogleda na datoteke izlazi van okvira operativnog sistema, koji sadržaj datoteke predstavlja kao niz bajta.

11.2 KONTINUALNE DATOTEKE

Sadržaji datoteka se nalaze u blokovima masovne memorije. Za bilo kakvo rukovanje ovim sadržajem neophodno je da on dospe u radnu memoriju. Zato je rukovanje bajtima sadržaja datoteke neraskidivo povezano sa prebacivanjem blokova sa ovim bajtima između radne i masovne memorije. Pri tome se bajti prebacuju iz radne u masovnu memoriju radi njihovog trajnog čuvanja, a iz masovne u radnu memoriju radi obrade. Da bi ovakvo prebacivanje bilo moguće, neophodno je da sloj za rukovanje datotekama uspostavi preslikavanje rednih brojeva bajta u redne brojeve njima odgovarajućih blokova. Ovakvo preslikavanje se najlakše uspostavlja, ako se sadržaj datoteke nalazi u susednim blokovima (čiji redni brojevi su uzastopni). Ovakve datoteke se nazivaju **kontinualne** (*contiguous*). Kod kontinualnih datoteka redni broj bloka sa traženim bajtom se određuje kao količnik rednog broja bajta i veličine bloka, izražene brojem bajta koje sadrži svaki blok. Pri tome, ostatak deljenja ukazuje na relativni položaj bajta u bloku.

Kontinualne datoteke zahtevaju od sloja za rukovanje datotekama da za svaku datoteku vodi evidenciju ne samo o njenom imenu, nego i o rednom broju početnog bloka datoteke, kao i o dužini datoteke. Podatke o početnom bloku i dužini kontinualne datoteke čuva njen deskriptor. Dužina datoteke može biti izražena brojem bajta, ali i brojem blokova, koga obavezno prati podatak o popunjenosti poslednjeg zauzetog bloka. Pojava da poslednji zauzeti blok datoteke nije popunjen do kraja se naziva **interna fragmentacija** (*internal fragmentation*). Ova pojava je važna, jer nepopunjeni poslednji zauzeti blok datoteke predstavlja neupotrebljen (i potencijalno neupotrebljiv) deo masovne memorije.

Sloj za rukovanje datotekama obavezno vodi i evidenciju slobodnih blokova masovne memorije. Za potrebe kontinualnih datoteka bitno je da ova evidencija olakša pronalaženje dovoljno dugačkih nizova susednih blokova. Zato je podesna **evidencija u obliku niza bita** (*bit map*), u kome svaki bit odgovara jednom bloku i pokazuje da li je

on zauzet (0) ili slobodan (1). U slučaju ovakve evidencije, pronalaženje dovoljno dugačkih nizova susednih blokova se svodi na pronalaženje dovoljno dugačkog niza jedinica u nizu bita koji odslikava zauzetost masovne memorije. Ovakvo pronalaženje je neizbežno pri stvaranju kontinualnih datoteka, čija veličina se zadaje prilikom njihovog stvaranja, pa unapred mora biti poznata.

Rukovanje slobodnim blokovima masovne memorije zahteva sinhronizaciju (međusobnu isključivost procesa), da bi se, na primer, izbeglo da više procesa, nezavisno jedan od drugog, zauzme iste slobodne blokove masovne memorije.

Pojava iscepkanosti slobodnih blokova masovne memorije u kratke nizove susednih blokova otežava rukovanje kontinualnim datotekama. Ta pojava se zove **eksterna fragmentacija** (*external fragmentation*). Ona nastaje kao rezultat višestrukog stvaranja i uništenja datoteka u slučajnom redosledu, pa nakon uništavanja datoteka ostaju nizovi slobodnih susednih blokova, međusobno razdvojeni blokovima postojećih datoteka. Problem eksterne fragmentacije se povećava, kada se, prilikom traženja dovoljno dugačkog niza susednih blokova, pronađe niz duži od potrebnog, jer se tada zauzima (*allocation*) samo deo blokova u pronađenom nizu. To dovodi do daljeg drobljenja (skraćenja) nizova slobodnih susednih blokova, jer preostaju sve kraći nizovi slobodnih susednih blokova.

Eksterna fragmentacija je problematična, jer posredno izaziva neupotrebljivost slobodnih blokova masovne memorije. Na primer, eksterna fragmentacija onemogućuje stvaranje kontinualne datoteke, čija dužina je jednaka sumi slobodnih blokova, kada oni ne obrazuju niz susednih blokova. Problem eksterne fragmentacije se može rešiti **sabijanjem** (*compaction*) **datoteka**, tako da svi slobodni blokovi budu potisnuti iza datoteka i da tako obrazuju niz susednih blokova. Mana ovog postupka je njegova dugotrajnost.

U opštem slučaju produženje kontinualne datoteke je komplikovano, jer zahteva stvaranje nove, veće kontinualne datoteke, prepisivanje sadržaja produžavane datoteke u novu datoteku i uništavanje produžavane datoteke. Sve ovo je potrebno, jer se ne može očekivati da se neposredno iza produžavane datoteke uvek nađe dovoljno dugačak niz susednih slobodnih blokova.

Problem produženja kontinualne datoteke se ublažava, ako se dozvoli da se kontinualna datoteka sastoji od više kontinualnih delova. Pri tome se za svaki od ovih delova u deskriptoru datoteke čuvaju podaci o rednom broju početnog bloka dotičnog dela i o njegovoj dužini (*extent list*). Ovakav pristup je zgodan za veoma dugačke datoteke (namenjene za čuvanje zvučnog ili video zapisa).

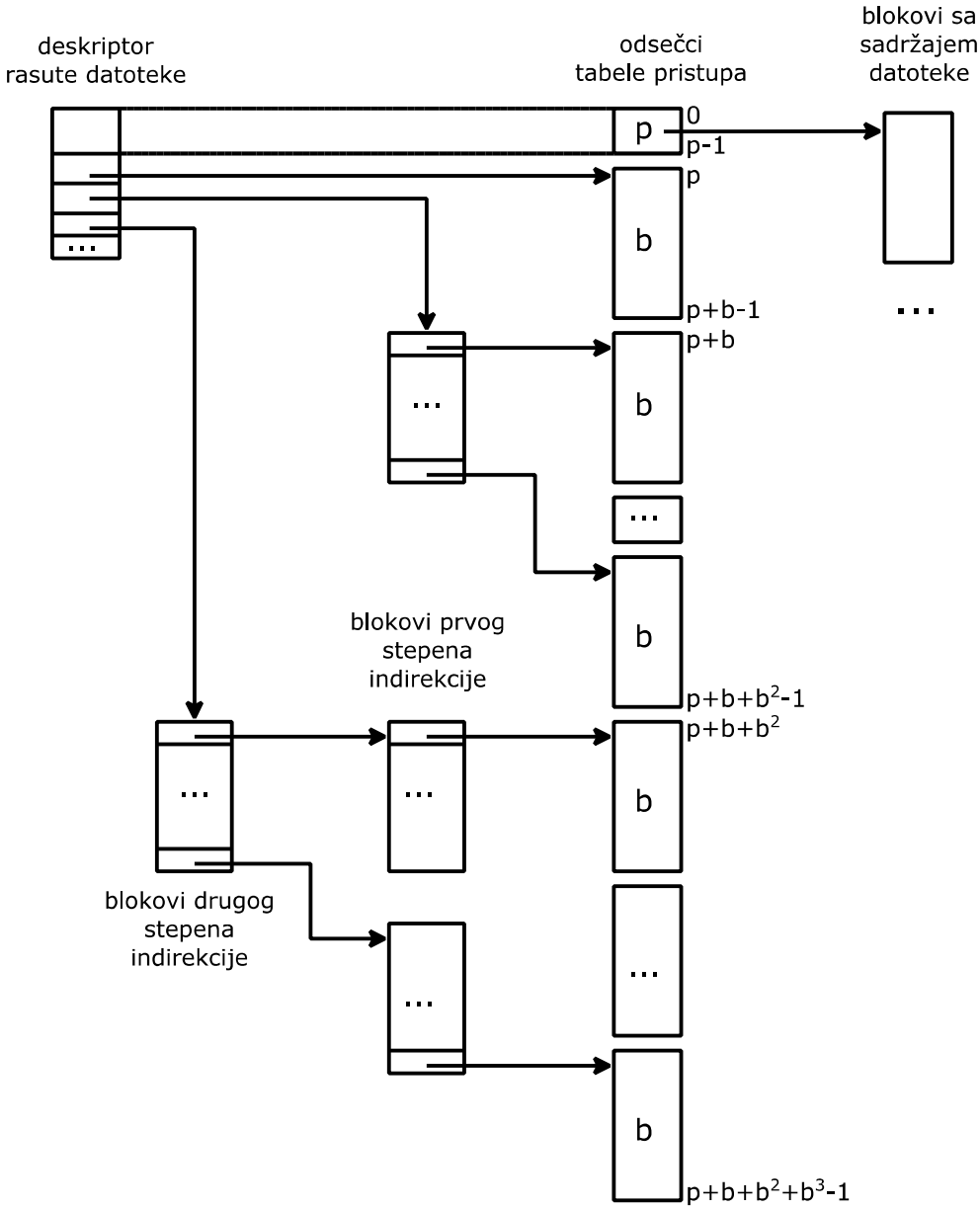
11.3 RASUTE DATOTEKE

Upotrebnu vrednost kontinualnih datoteka značajno smanjuju problem eksterne fragmentacije, potreba da se unapred zna veličina kontinualnih datoteka i teškoće sa njihovim produžavanjem. Zato se umesto kontinualnih koriste **rasute** (*noncontiguous*) **datoteke**, čiji sadržaj je smešten (rasut) u nesusednim blokovima masovne memorije. Kod rasutih datoteka redni brojevi bajta se preslikavaju u redne brojeve blokova pomoću

posebne **tabele pristupa** (*file allocation table*). Njeni elementi sadrže redne brojeve blokova. Indekse ovih elemenata određuje količnik rednog broja bajta i veličine bloka. Iz prethodnog sledi da dužinu rasutih datoteka ograničava veličina tabele pristupa. Zato se veličina tabele pristupa dimenzionira tako da zadovolji najveće praktične zahteve u pogledu dužine rasutih datoteka.

Tabele pristupa se čuvaju u blokovima masovne memorije (kao, uostalom, i sadržaji datoteka). Radi manjeg zauzeća, važno je da se u blokovima masovne memorije ne čuva uvek cela tabela pristupa, nego samo njen neophodan (stvarno korišćen) deo. Zato se tabela pristupa deli u odsečke. **Početni odsečak**, sa p elemenata tabele pristupa je uvek prisutan. On nije veći od bloka masovne memorije. **Dodatni odsečci** su prisutni samo kad su neophodni. Svaki dodatni odsečak je jednak bloku masovne memorije i može da sadrži b elemenata tabele pristupa ($b > p$). Prema tome, tabela pristupa svake rasute datoteke zauzima jedan blok (ili njegov deo), u kome se nalazi početni odsečak ove tabele sa p njenih elemenata. Za tabelu pristupa se, po potrebi, zauzima još jedan blok sa dodatnim odsečkom, u kome se nalazi narednih b njenih elemenata. Kada zatreba još dodatnih odsečaka, za tabelu pristupa se zauzima poseban **blok prvog stepena indirekcije**. On sadrži do b rednih brojeva blokova sa dodatnim odsečcima. U svakom od njih se nalazi b novih elemenata tabele pristupa. Na kraju, po potrebi, za ovu tabelu se zauzima poseban blok **drugog stepena indirekcije**. On sadrži do b rednih brojeva blokova prvog stepena indirekcije. Svaki od njih sadrži do b rednih brojeva blokova sa dodatnim odsečcima, a u svakom od njih se nalazi b novih elemenata tabele pristupa. Prema tome, ukupno ima $1+b+b^2$ dodatnih odsečaka, svaki sa b elemenata tabele pristupa.

Bitni elementi organizacije rasute datoteke se mogu prikazati grafički (Slika 11.1).



Slika 11.1: Grafički prikaz bitnih elemenata organizacije rasute datoteke

Deskriptor rasute datoteke sadrži početni odsečak tabele pristupa, redni broj njenog prvog dodatnog odsečka, redni broj bloka prvog stepena indirekcije i redni broj bloka drugog stepena indirekcije. Pored toga, ovaj deskriptor sadrži i dužinu rasute datoteke, da bi se znalo koliko blokova je zauzeto sadržajem i koliko je popunjen poslednji zauzeti blok.

Ideja, korišćena za organizaciju tabele pristupa, može da se upotrebi i za organizaciju

evidencije slobodnih blokova masovne memorije. U ovom slučaju ova evidencija ima oblik **liste slobodnih blokova**. Slobodni blokovi, uvezani u ovu listu, sadrže redne brojeve ostalih slobodnih blokova, pa podsećaju na blokove prvog stepena indirekcije.

11.4 KONZISTENTNOST SISTEMA DATOTEKA

Iza rukovanja datotekama krije se rukovanje blokovima masovne memorije, u kojima se nalaze i sadržaj i deskriptor i, eventualno, dodatni odsečci tabele pristupa svake rasute datoteke. Rukovanje ovim blokovima usložnjava činjenica da se međusobno zavisni podaci nalaze u raznim blokovima. Pošto se blokovi modifikuju u radnoj memoriji, a trajno čuvaju u masovnoj memoriji, prirodno je da u pojedinim trenucima postoji razlika između blokova u masovnoj memoriji i njihovih kopija u radnoj memoriji. Probleme izaziva gubitak kopija blokova u radnoj memoriji, na primer, zbog nestanka napajanja, pre nego su sve kopije blokova, sa međusobno zavisnim podacima, prebačene u masovnu memoriju. Tako, na primer, produženje rasute datoteke zahteva (1) izmenu evidencije slobodnih blokova, radi isključivanja pronađenog slobodnog bloka iz ove evidencije, kao i (2) izmenu tabele pristupa produžavane rasute datoteke, radi smeštanja rednog broja novog bloka u element ove tabele. Izmena evidencije slobodnih blokova dovodi do promene jedne od kopija njenih blokova u radnoj memoriji. Isti efekat ima i izmena tabele pristupa produžavane rasute datoteke. Ako obe izmenjene kopije budu prebačene u masovnu memoriju, tada je produženje rasute datoteke uspešno obavljeno. Ako ni jedna od kopija ne dospe u masovnu memoriju, tada produženje rasute datoteke nije obavljeno, jer nije registrovano u masovnoj memoriji. Ali, ako samo jedna od promenjenih kopija dospe u masovnu memoriju, tada se javljaju problemi konzistentnosti sistema datoteka. U jednom slučaju, kada samo promenjena kopija bloka evidencije slobodnih blokova dospe u masovnu memoriju, blok isključen iz ove evidencije postaje izgubljen, jer njegov redni broj nije prisutan niti u ovoj evidenciji, a niti u tabeli pristupa neke od rasutih datoteka. U drugom slučaju, kada samo promenjena kopija bloka tabele pristupa dospe u masovnu memoriju, blok, pridružen ovoj tabeli, ostaje i dalje uključen u evidenciju slobodnih blokova. Prvi slučaj je bezazlen, jer se izgubljeni blokovi mogu pronaći. Pronalaženje izgubljenih blokova se zasniva na traženju blokova koji nisu prisutni ni u evidenciji slobodnih blokova, ni u tabelama pristupa rasutih datoteka. Za razliku od prvog slučaja, drugi slučaj je neprihvatljiv, jer može da izazove istovremeno uključivanje istog bloka u više rasutih datoteka, čime se narušava njihova konzistentnost. Zato je neophodno uvek prebacivati u masovnu memoriju prvo promenjenu kopiju bloka evidencije slobodnih blokova, pa tek iza toga i promenjenu kopiju bloka tabele pristupa. Znači, potrebno je paziti na redosled u kome se izmenjene kopije blokova prebacuju u masovnu memoriju.

U opštem slučaju konzistentnost sistema datoteka može da se zasniva na vođenju **pregleda izmena** (*journal*). Pre bilo kakve izmene sistema datoteka, u pregledu izmena se registruje potpun opis nameravane izmene, na osnovu koga je moguće izvršiti oporavak sistema datoteka posle nedovršene izmene. Tek nakon toga se pristupa izmeni sistema datoteka. Po uspešno obavljenoj izmeni, u pregledu izmena se to i registruje. Ako u pregledu izmena nije registrovan potpun opis nameravane izmene, tada izmena nije ni započeta, pa je sistem datoteka u konzistentom stanju (nakon izbacivanja iz

pregleda izmena nepotpunog opisa nameravane izmene). Kada je u pregledu izmena registrovan potpun opis nameravane izmene, ali nije registrovano njeno uspešno obavljanje, tada je sistem datoteka moguće vratiti u konzistentno stanje. Ako su u pregledu izmena registrovani potpuni opis nameravane izmene i njeno uspešno obavljanje, tada je sistem datoteka u konzistentnom stanju. Ideja pregleda izmena može da bude osnova za organizovanje celog sistema datoteka (*log structured file system*). U ovom pristupu izmena svake datoteke se registruje samo u posebnom pregledu izmena, čijom kasnijom analizom se, po potrebi, rekonstruiše aktuelni sadržaj datoteke.

Nakon izmene kopije bloka u radnoj memoriji, važno je što pre izmenjenu kopiju prebaciti u masovnu memoriju, radi smanjenja mogućnosti da se ona izgubi (na primer, kao posledica nestanka napajanja). To je naročito važno, ako izmena nije rezultat automatske obrade, nego ljudskog rada (na primer, editiranja), jer se tada ne može automatski rekonstruisati.

11.5 BAFERSKI PROSTOR

Pristupi sadržaju datoteke mogu zahtevati prebacivanje više blokova u radnu memoriju, na primer, jednog bloka sa deskriptorom datoteke, pa, eventualno, jednog ili više dodatnih blokova tabele pristupa i, na kraju, bloka sa traženim bajtima sadržaja. Pošto je, sa stanovišta procesora, prenos blokova na relaciji radna i masovna memorija, spor (dugotrajan), dobra ideja je zauzeti u radnoj memoriji prostor za više bafera, namenjenih za čuvanje kopija korišćenih blokova (*block cache, buffer cache*). Pošto je radna memorija mnogo manja od masovne, njeni baferi mogu istovremeno da sadrže mali broj kopija blokova masovne memorije. Zato je važno da baferi sadrže kopije blokova, koje će biti korišćene u neposrednoj budućnosti, jer se samo tako značajno ubrzava obrada podataka. Problem se javlja kada su svi baferi napunjeni, a potrebno je pristupiti bloku masovne memorije, čija kopija nije prisutna u nekom od bafera. U tom slučaju neizbežno je oslobađanje nekog od bafera, da bi se u njega smestila kopija potrebnog bloka. Iskustvo pokazuje da je najbolji pristup osloboditi bafer za koga trenutak poslednjeg pristupa njegovom sadržaju prethodi trenucima poslednjeg pristupa sadržajima svih ostalih bafera (*Least Recently Used - LRU*). Za takav bafer se kaže da ima najstariju referencu. Pri oslobađanju bafera, njegov dotadašnji sadržaj se poništava, kada bafer sadrži neizmenjenu kopiju bloka, jer je ona identična bloku masovne memorije. U suprotnom slučaju, neophodno je sačuvati izmene, pa se kopija prebacuje u masovnu memoriju. U oslobođeni bafer se prebacuje kopija potrebnog bloka masovne memorije. Da bi se znalo koja od kopija ima najstariju referencu, baferi se uvezuju u listu. Na početak ovakve liste se uvek prebacuje bafer sa upravo korišćenom kopijom (sa najnovijom referencom), pa na njen kraj nužno dospeva bafer sa najstarijom referencom.

Baferovanje izmenjenih kopija blokova u radnoj memoriji zahteva da se odredi trenutak u kome se izmenjena kopija prebacuje u masovnu memoriju. Ako se izmenjena kopija prebacuje u masovnu memoriju odmah nakon svake izmene, tada se usporava rad, a ako se izmenjena kopija prebacuje u masovnu memoriju tek pri oslobađanju bafera, tada se povećava mogućnost gubljenja izmene (na primer, zbog nestanka napajanja). Rešavanje ovoga problema se može posredno prepustiti korisniku, ako se uvede posebna

sistemska operacija (**sync()**) za izazivanje prebacivanja sadržaja bafera (sa izmenjenim kopijama blokova) u masovnu memoriju. U tom slučaju izmenjene kopije dospevaju u masovnu memoriju ili kada se baferi oslobađaju ili kada to zatraži korisnik. Potreba da se kopija bloka što brže nakon izmene prebaci u masovnu memoriju je u suprotnosti sa nastojanjem da se blokovi što ređe prenose između radne i masovne memorije.

Na brzinu prebacivanja podataka između radne i masovne memorije važan uticaj ima i veličina bloka, jer, što je blok veći, u proseku se potroši manje vremena na prebacivanje jednog bajta između radne i masovne memorije. Međutim, što je blok veći, veća je i interna fragmentacija. Ta dva oprečna zahteva utiču na izbor veličine bloka, koja se kreće od 512 bajta do 8192 bajta.

11.6 DESKRIPTOR DATOTEKE

Deskriptor datoteke, pored atributa koji omogućuju preslikavanje rednih brojeva bajta u redne brojeve blokova, sadrži i:

1. numeričku oznaku vlasnika datoteke,
2. prava pristupa datoteci za njenog vlasnika, za njegove saradnike i za ostale korisnike,
3. podatak da li je datoteka zaključana ili ne,
4. SUID podatak da li numerička oznaka vlasnika datoteke postaje numerička oznaka vlasnika procesa stvorenog na osnovu sadržaja datoteke (važi samo za izvršne datoteke), kao i
5. datum poslednje izmene datoteke.

Činjenica, da deskriptor datoteke sadrži prava pristupa datoteci, podrazumeva da je sadržaj masovne memorije fizički zaštićen, tako da nema načina za neovlašteni pristup masovnoj memoriji. To podrazumeva da se centralni delovi računara (procesor, radna i masovna memorija, kontroleri i sabirnica) nalaze u zaštićenoj (sigurnoj) prostoriji, a da su samo periferni delovi računara (kao što su terminali ili štampači) direktno na raspolaganju korisnicima. Kada to nije moguće, alternativa je da sadržaj masovne memorije bude kriptovan.

Podatak da li je datoteka zaključana ili ne se čuva u kopiji deskriptora u radnoj memoriji. Ova kopija nastane prilikom otvaranja datoteke. Podatak da li je datoteka zaključana ili ne je uveden radi ostvarenja međusobne isključivosti procesa u toku pristupa datoteci. Pri tome se podrazumeva da su aktivnosti ovih procesa međusobno isključive i u toku obavljanja operacije zaključavanja datoteke. U ovoj operaciji se proverava da li je datoteka zaključana i eventualno obavi njeno zaključavanje. Sinhronizacija procesa u toku obavljanja ove operacije je neophodna, da bi se sprečilo da dva ili više procesa istovremeno zaključe da je ista datoteka otključana i da, nezavisno jedan od drugog, istovremeno zaključaju pomenutu datoteku. Pomenuta sinhronizacija obezbeđuje da uvek najviše jedan proces zaključa datoteku, jer samo on pronalazi otključanu datoteku, dok svi preostali istovremeno aktivni procesi pronalaze zaključanu datoteku. Ako je za nastavak aktivnosti ovih preostalih procesa neophodno da pristupe datoteci, tada se njihova aktivnost zaustavlja do otključavanja datoteke. Zbog ovakvih procesa potrebna je operacija otključavanja datoteke. Njeno izvršavanje omogućuje

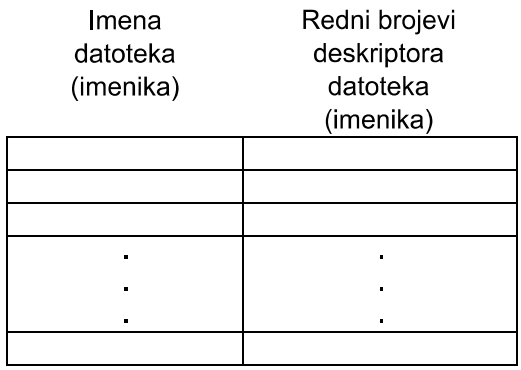
nastavak aktivnosti samo jednog od procesa, čija aktivnost je zaustavljena do otključavanja datoteke. Ako takav proces postoji, datoteka se i ne otključava, nego se samo prepušta novom procesu. Inače, datoteka se otključava. I operacija otključavanja datoteka zahteva sinhronizaciju procesa.

U slučaju zaključavanja datoteke, moguće je da proces nastavi svoju aktivnost i nakon neuspešnog pokušaja zaključavanja datoteke. Jasno, tada se podrazumeva da on neće pristupati pomenutoj datoteci. Prema tome, operacija zaključavanja datoteke je blokirajuća, kada, radi uslovne sinhronizacije, u toku njenog obavljanja dolazi do zaustavljanja aktivnosti procesa, dok se ne stvore uslovi za međusobno isključive pristupe zaključanoj datoteci. Ova operacija je neblokirajuća, kada njena povratna vrednost ukazuje na neuspešan pokušaj zaključavanja datoteke i na nemogućnost pristupa datoteci, koju je zaključao neki drugi proces.

Sinhronizaciju procesa moraju da obezbede ne samo operacije zaključavanja i otključavanja datoteke, nego i sve druge operacije za rukovanje deskriptorima datoteka. Jedino tako se može obezbediti očuvanje konzistentnosti deskriptora (i njima odgovarajućih datoteka).

11.7 IMENICI

Ime datoteke je prirodno vezano za njen deskriptor. Pošto se ime datoteke nalazi u imeniku, uz njega bi, u imenik, mogao da bude smešten i deskriptor datoteke. Međutim, tipičan način korišćenja imenika je njihovo pretraživanje, radi pronalaženja imena imenika ili imena datoteke, navedene u datoj putanji. Ovakvo pretraživanje prirodno prethodi pristupu sadržaju datoteke, odnosno sadržaju imenika. Brzina tog pretraživanja je veća što su imenici kraći, jer se tada manje podataka prebacuje između radne i masovne memorije. Znači uputno je da imenici sadrže samo imena datoteka i imenika, a ne i njihove deskriptore, pogotovo ako su deskriptori veliki. Zato se deskriptori (*inodes*) čuvaju na disku van imenika. Da bi se uspostavila veza između imena datoteka, odnosno imena imenika sa jedne strane i njihovih deskriptora sa druge strane, u imenicima se, uz imena datoteka, odnosno uz imena imenika, navode i redni brojevi njihovih deskriptora, koji jednoznačno određuju ove deskriptore. Prema tome, imenik je datoteka koja sadrži tabelu u čijim elementima su imena datoteka (odnosno, imena imenika) i redni brojevi njihovih deskriptora (Slika 11.2).



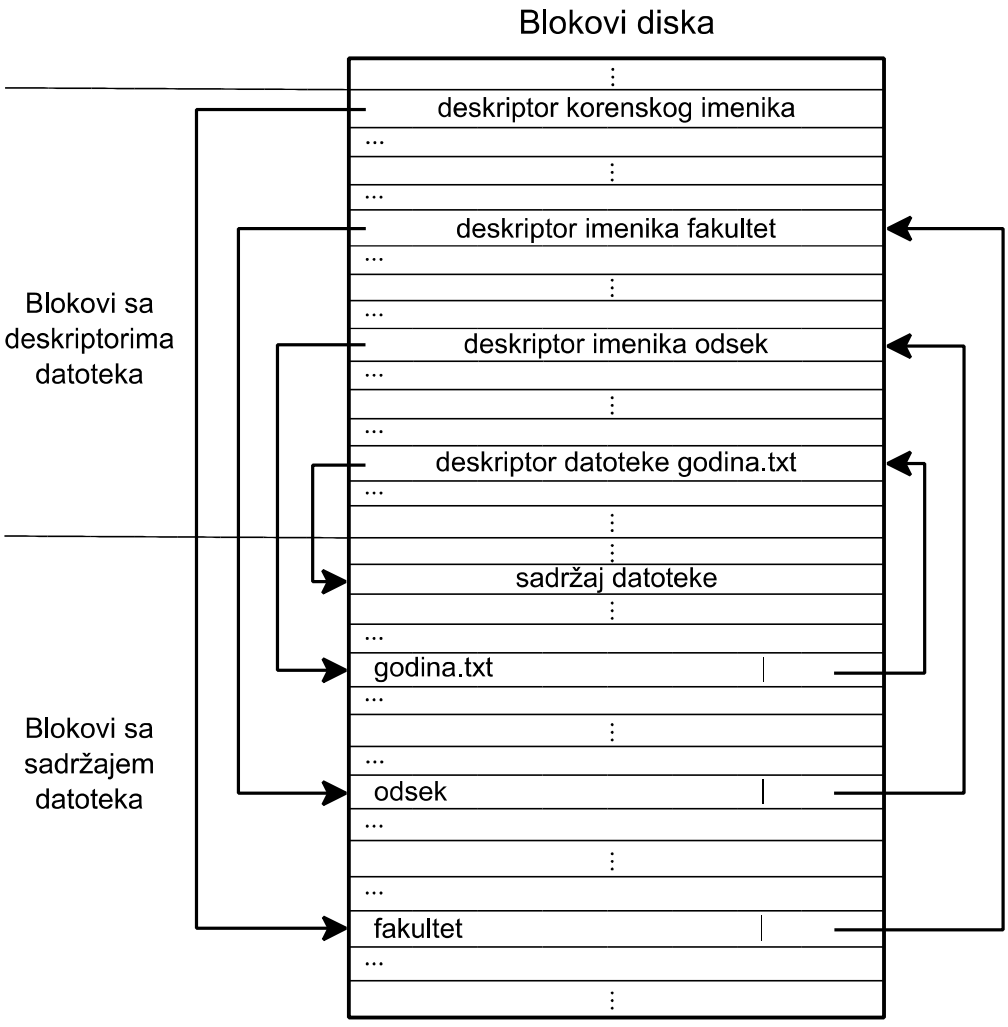
Slika 11.2: Sadržaj imenika

Iz rednog broja deskriptora se može odrediti redni broj bloka masovne memorije, u kome se deskriptor nalazi, ako se izvestan broj susednih blokova rezerviše samo za smeštanje deskriptora. Pod pretpostavkom da blok sadrži celi broj deskriptora, količnik rednog broja deskriptora i ukupnog broja deskriptora u bloku određuje redni broj bloka sa deskriptorom. Pri tome se podrazumeva da je deskriptor sa rednim brojem 0 rezervisan za korenski imenik. Zahvaljujući ovoj pretpostavci, moguće je uvek pronaći deskriptor korenskog imenika i od njega započeti pretraživanje imenika, što obavezno prethodi pristupu sadržaju datoteke. Tako, na primer, za pristup sadržaju datoteke sa putanjom:

/fakultet/odsek/godina.txt

potrebno je prebaciti u radnu memoriju blok sa deskriptorom korenskog imenika, koji je poznat, zahvaljujući činjenici da je redni broj (0) deskriptora korenskog imenika unapred zadan (Slika 11.3). U deskriptoru korenskog imenika se nalaze redni brojevi blokova sa sadržajem korenskog imenika. Nakon prebacivanja ovih blokova u radnu memoriju, moguće je pretražiti sadržaj korenskog imenika, da bi se ustanovilo da li on sadrži ime **fakultet**. Ako sadrži, uz ovo ime je i redni broj deskriptora odgovarajućeg imenika, iz koga se može odrediti redni broj bloka u kome se nalazi ovaj deskriptor. Po prebacivanju ovog bloka u radnu memoriju, u pomenutom deskriptoru se pronalaze redni brojevi blokova sa sadržajem imenika **fakultet**. Nakon prebacivanja ovih blokova u radnu memoriju, moguće je pretražiti sadržaj i ovog imenika, da bi se ustanovilo da li on sadrži ime **odsek**. Ako sadrži, uz ovo ime je i redni broj deskriptora odgovarajućeg imenika, iz koga se može odrediti redni broj bloka u kome se nalazi ovaj deskriptor. Po prebacivanju ovog bloka u radnu memoriju, u pomenutom deskriptoru se pronalaze redni brojevi blokova sa sadržajem imenika **odsek**. Nakon prebacivanja ovih blokova u radnu memoriju moguće je pretražiti sadržaj i ovog imenika, da bi se ustanovilo da li on sadrži ime datoteke **godina.txt**. Ako sadrži, uz nju je i redni broj deskriptora odgovarajuće datoteke, iz koga se može odrediti redni broj bloka u kome se nalazi ovaj deskriptor. Po prebacivanju ovog bloka u radnu memoriju, u pomenutom deskriptoru se pronalaze redni brojevi blokova sa sadržajem datoteke **godina.txt**. Tek tada je moguć pristup ovom

sadržaju.

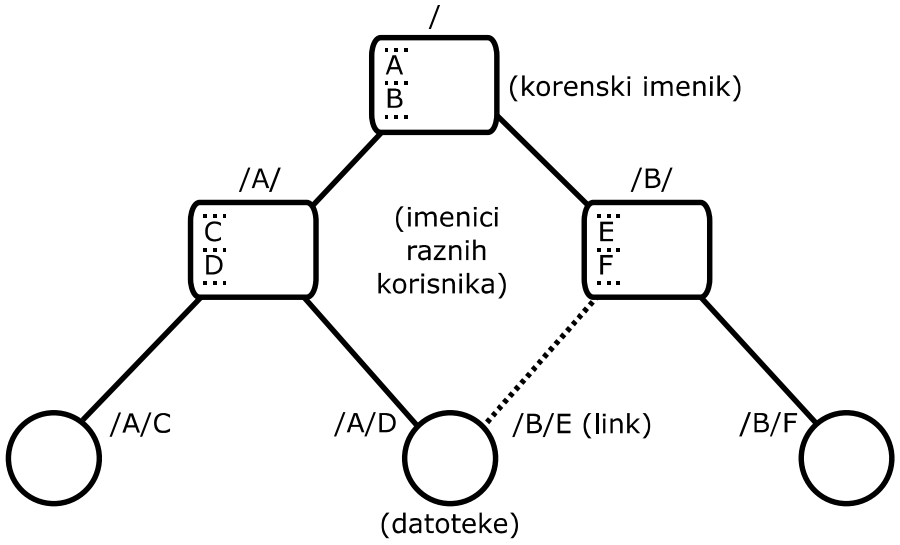


Slika 11.3: Organizacija imenika

U toku pristupa imenicima, neophodno je njihovo zaključavanje, radi obezbeđenja međusobne isključivosti pristupa raznih procesa istom imeniku, čime se obezbeđuje očuvanje konzistentnosti imenika.

Za imenike je važno pitanje da li ista datoteka može istovremeno biti registrovana u dva ili više imenika. Ako se to dozvoli, tada razne putanje mogu voditi do istog sadržaja. To je efikasan način da vlasnik, ali i više drugih korisnika istu datoteku mogu naći svaki u svom imeniku, a da ne moraju praviti sopstvenu kopiju datoteke. Pri tome, svaki od drugih korisnika može dotičnoj datoteci nadenuti novo ime, koje se naziva link (*link*). Slika 11.4 sadrži prikaz tri imenika (predstavljeni kvadratima) i tri datoteke

(predstavljene krugovima). Do srednje datoteke vode dve putanje (link je predstavljen isprekidanom linijom).



Slika 11.4: Primer linka

U imeniku uz link može biti naveden redni broj deskriptora odgovarajuće datoteke (*hard link*), ali može biti navedena putanja datoteke njenog vlasnika (*soft link*). U prvom slučaju, deskriptor datoteke mora da sadrži broj linkova. Ako se dozvoli da i imenici imaju linkove, tada postaju mogući ciklusi u hijerarhijskoj organizaciji datoteka (jer imenik može sadržati svoj link ili link imenika sa višeg nivoa hijerarhije).

11.8 SISTEMSKE OPERACIJE SLOJA ZA RUKOVANJE DATOTEKAMA

Prethodno opisano pretraživanje imenika se dešava u okviru izvršavanja sistemske operacije otvaranja datoteke (**open()**). Zato je putanja datoteke obavezni argument poziva ove operacije. Njeno izvršavanje prebacuje kopiju deskriptora datoteke u radnu memoriju u **tabelu deskriptora datoteka**. Ova kopija se ne uključuje u deskriptor procesa, u toku čije aktivnosti je inicirano njeno prebacivanje, jer ista datoteka može istovremeno biti otvorena u toku aktivnosti više procesa. Da bi svaki od njih koristio istu kopiju deskriptora datoteke, u deskriptoru svakog procesa postoji **tabela otvorenih datoteka**. Svaki njen element sadrži adresu kopije deskriptora odgovarajuće datoteke iz tabele deskriptora datoteka. Indeks elementa tabele otvorenih datoteka (u kome je adresa kopije deskriptora otvorene datoteke) predstavlja povratnu vrednost poziva sistemske operacije otvaranja datoteke. Ovaj indeks otvorene datoteke se koristi kao argument u pozivima drugih sistemskih operacija sloja za rukovanje datotekama. On određuje datoteku na koju se pomenuti poziv odnosi. Od veličine tabele otvorenih datoteka zavisi najveći mogući broj istovremeno otvorenih datoteka nekog procesa.

Kao dodatni argument poziva sistemske operacije otvaranja datoteke može se javiti oznaka nameravane vrste pristupa otvaranoj datoteci, koja pokazuje da li se datoteka

otvara samo za čitanje, ili za pisanje/čitanje. U toku izvršavanja sistemske operacije otvaranja datoteke proverava se da li proces, koji poziva ovu operaciju, poseduje pravo nameravanog pristupa otvaranoj datoteci. Za ovo se koristi, sa jedne strane, numerička oznaka vlasnika procesa iz deskriptora procesa, a sa druge strane, numerička oznaka vlasnika otvarane datoteke, kao i prava pristupa otvaranoj datoteci za njenog vlasnika, za njegove saradnike i za ostale korisnike, što sadrži deskriptor otvarane datoteke. Otvaranje datoteke je uspešno samo ako proces poseduje pravo nameravanog pristupa datoteci. Tada poziv sistemske operacije otvaranja datoteke vraća indeks otvorene datoteke. Inače, on vraća kod greške.

Oznaka vrste nameravanog pristupa datoteci se čuva u posebnom polju odgovarajućeg elementa tabele otvorenih datoteka, radi naknadne provere ispravnosti pristupa datoteci.

Pokušaj otvaranja nepostojeće datoteke dovodi do njenog stvaranja, ako se tako navede u argumentima poziva sistemske operacije otvaranja datoteke. Alternativa je da postoji posebna operacija (**create()**) za stvaranje datoteke (argument poziva ove operacije bi bila putanja stvarane datoteke).

Korišćenje datoteke se završava pozivom sistemske operacije zatvaranja datoteke (**close()**). Ona čisti odgovarajući element tabele otvorenih datoteka procesa, koji je pozvao ovu sistemsku operaciju, i prebacuje, eventualno, kopiju deskriptora i baferovane kopije blokova sadržaja zatvarane datoteke u masovnu memoriju (što je neophodno samo ako su ove kopije izmenjene). Kopija deskriptora zatvarane datoteke ostaje u radnoj memoriji, ako, pored procesa koji zatvara datoteku, postoje i drugi procesi koji joj pristupaju. Zato kopija deskriptora datoteke sadrži broj procesa koji pristupaju datoteci. Obavezni argument poziva sistemske operacije zatvaranja datoteke je indeks otvorene datoteke.

Važno je uočiti da u periodu dok je datoteka otvorena, znači, između uzastopnih poziva sistemskih operacija otvaranja i zatvaranja datoteke, proces nije pod uticajem izmena prava pristupa otvorenoj datoteci, jer izmena prava pristupa postaje delotvorna tek pri narednom otvaranju datoteke, pošto se tek tada ova prava ponovo proveravaju.

Korisna praksa je da se, na kraju aktivnosti procesa, automatski zatvore sve otvorene datoteke.

Otvorena datoteka se zaključava kada je neophodno ostvariti međusobnu isključivost u toku pristupa njenom sadržaju. Datoteka se otključava kada prestane potreba za međusobnom isključivošću u toku rukovanja njenim sadržajem. Zaključavanje i otključavanje datoteka se nalazi u nadležnosti posebne sistemske operacije (**flock()**) koja rukuje kopijom deskriptora zaključavane/otključavane datoteke. U opštem slučaju, operacija zaključavanja datoteke može biti blokirajuća ili neblokirajuća. U prvom slučaju aktivnost procesa se zaustavlja tokom pokušaja zaključavanja datoteke, dok zaključavanje ne postane moguće. U drugom slučaju, ako zaključavanje datoteke nije moguće, poziv sistemske operacije zaključavanja datoteke vraća kod greške koji objašnjava razlog neuspeha u zaključavanju datoteke. Obavezni argument poziva sistemske operacije zaključavanja datoteke je indeks otvorene datoteke.

Nakon otvaranja, sadržaj datoteke se može čitati pozivom sistemske operacije čitanja datoteke (**read()**) i pisati pozivom sistemske operacije pisanja datoteke (**write()**), ako je to u skladu sa namerama, izraženim u otvaranju datoteke. Obavezni argumenti ovih poziva su indeks otvorene datoteke i broj bajta (koji se čitaju ili pišu). Pored toga, poziv sistemske operacije čitanja sadrži, kao argument, adresu zone radne memorije u koju se smeštaju pročitani bajti, a poziv sistemske operacije pisanja sadrži, kao argument, adresu zone radne memorije iz koje se preuzimaju bajti za pisanje. Oba poziva vraćaju vrednost, koja ukazuje na uspešan poziv ili na grešku. Podrazumeva se da sistemske operacije pisanja i čitanja nude sekvencijalan pristup datotekama. To znači, da, ako jedan poziv sistemske operacije čitanja (pisanja) pročita (upiše) prvi bajt datoteke, naredni takav poziv datoteke će da pročita (upiše) drugi bajt datoteke. Radi podrške sekvencijalnom pristupu, svaki element tabele otvorenih datoteka sadrži i posebno polje **pozicije** u datoteci sa rednim brojem bajta od koga se primenjuje naredno čitanje ili pisanje. Svako čitanje ili pisanje pomera poziciju na prvi naredni nepročitani (neupisani) bajt. Da bi bili mogući direktni pristupi bajtima datoteke (u proizvoljnom redosledu), postoji sistemska operacija izmene pozicije u datoteci (**seek()**). Obavezni argumenti njenog poziva su indeks otvorene datoteke i podatak o novoj poziciji, dok povratna vrednost ovog poziva ukazuje da li je poziv bio uspešan ili ne. Tako, na primer, ako se želi pisati na kraj datoteke, neophodno je prvo pozvati sistemsku operaciju izmene pozicije u datoteci, radi pozicioniranja iza poslednjeg bajta datoteke, i zatim pozvati sistemsku operaciju pisanja. Prvi poziv izmeni poziciju u datoteci u odgovarajućem elementu tabele otvorenih datoteka. Drugi poziv, na osnovu ove pozicije i kopije deskriptora datoteke, odredi redni broj bloka, ako on postoji, i prebaci njegovu kopiju u radnu memoriju, ako ona već nije bila prisutna u radnoj memoriji. U ovu kopiju se smeste dopisivani bajti i ona se prebaci (odmah ili kasnije, zavisno od strategije baferovanja) u masovnu memoriju. Ako blok ne postoji, ili ako se upisivanje proteže na više blokova, upisivanju bajta prethodi zauzimanje blokova. Radi toga se menja (proširuje) tabela pristupa datoteke, što može da dovede i do izmene njenog deskriptora. Nakon toga se u radnoj memoriji oblikuje novi sadržaj blokova i oni se prebacuju u masovnu memoriju.

Sloj za rukovanje datotekama nudi posebne sistemske operacije za izmenu atributa datoteke, sadržanih u njenom deskriptoru (kao što su numerička oznaka vlasnika datoteke, prava pristupa datoteci za njenog vlasnika, za njegove saradnike i za ostale korisnike, ili SUID podatak da li numerička oznaka vlasnika datoteke postaje numerička oznaka vlasnika procesa, stvorenog na osnovu sadržaja datoteke). Obavezni argumenti poziva ovih sistemskih operacija su putanja datoteke i nova vrednost menjanog atributa datoteke. Njihova povratna vrednost ukazuje na uspešnost poziva. U okviru izvršavanja ovih sistemskih operacija pretražuju se imenici, radi prebacivanja u radnu memoriju kopije deskriptora datoteke. Zatim se proverava da li je ove sistemske operacije pozvao vlasnik datoteke, jer jedino on ima pravo da menja atribute datoteke. Ako jeste, tada se menja zadani atribut i blok sa izmenjenom kopijom deskriptora vraća u masovnu memoriju. U sistemske operacije za izmenu atributa datoteke spada i sistemska operacija za izmenu imena datoteke. Iako se u okviru ove operacije ne menja deskriptor datoteke, nego njen imenik, ovde je, pored pristupa deskriptoru njenog imenika, neophodan i pristup deskriptoru dotične datoteke, radi provere da li je ovu operaciju pozvao vlasnik

datoteke.

Za stvaranje linka (dodatnog imena datoteke) potrebna je posebna sistemska operacija (**link()**). Obavezani argumenti njenog poziva su putanja sa imenom datoteke i putanja sa linkom. U okviru ove operacije se pretražuju imenici radi ubacivanja linka.

Za uništenje datoteke neophodna je posebna sistemska operacija (**unlink()**). Obavezni argument njenog poziva je putanja uništavane datoteke. U okviru sistemske operacije uništenja datoteke, pretražuju se imenici, radi prebacivanja u radnu memoriju kopije deskriptora uništavane datoteke. Zatim se proverava da li je broj linkova za ovu datoteku nula i da li je ovu sistemsku operaciju pozvao vlasnik datoteke, pa ako jeste, oslobađaju se blokovi datoteke i njen deskriptor. Na kraju se poništava ime datoteke u odgovarajućem imeniku, što predstavlja izmenu njegovog sadržaja. Povratna vrednost ove operacije ukazuje da li je ona uspešno obavljena. Na primer, uništenje nepostojeće datoteke ne može biti uspešno obavljeno.

Prethodno opisane sistemske operacije omogućuju pristup svim datotekama, znači i imenicima. Ali za pristupe imenicima je potrebno poznavati detalje njihove organizacije, kao što su (1) broj bajta koji je rezervisan za imena datoteka i imenika, (2) broj bajta koji je rezervisan za redne brojeve njihovih deskriptora, ili (3) oznaku koja označava prethodni imenik u hijerarhiji i slično. Potreba za poznavanjem organizacije imenika se izbegava, ako se ponude posebne sistemske operacije za rukovanje imenicima, kao što su operacije za stvaranje imenika, za pregledanje i za izmene njegovog sadržaja, ili za uništenje imenika. Među sistemske operacije za rukovanje imenicima spada posebna operacija (**mount()**) koja omogućuje spajanje dva sistema datoteka, tako što korenski imenik jednog od sistema datoteka postane (zameni) običan imenik drugog sistema datoteka. Postoji i suprotna operacija (**umount()**) koja razdvaja dva prethodno spojena sistema datoteka.

11.9 SPECIJALNE DATOTEKE

Važno svojstvo pojma datoteke je da je on primenljiv i za opisivanje ulaznih i izlaznih uređaja. Tako, ulazni uređaj, kao što je tastatura, odgovara datoteci, čiji sadržaj se može samo čitati, a sastoji se od bajta, koji stižu sa tastature. Slično, izlazni uređaj, kao što je ekran, odgovara datoteci, čiji sadržaj se može samo pisati, a sastoji se od bajta, koji se upućuju na ekran. Takođe, ulazno izlazni uređaj, kao što je disk, odgovara datoteci, čiji sadržaj se može i pisati i čitati, a sastoji se od bajta iz pojedinih blokova diska. Datoteke, koje predstavljaju pojedine ulazne ili izlazne uređaje, se nazivaju **specijalne datoteke** (*special file*). Specijalne datoteke se dele na znakovne, koje odgovaraju uređajima kao što su tastatura, ekran ili štampač, i na blokovske, koje odgovaraju, na primer, diskovima. Znakovne specijalne datoteke podržavaju sekvencijalno čitanje ili pisanje znakova (koji dolaze sa odgovarajućeg urađaja, ili odlaze na odgovarajući uređaj). Za ove datoteke izmena pozicije nema smisla. Blokovske specijalne datoteke podržavaju čitanje ili pisanje blokova. Za njih izmena pozicije omogućuje određivanje rednog broja bloka na koga se primenjuje naredna operacija.

Blokovske specijalne datoteke omogućuju direktne pristupe blokovima diska, što je

važno, na primer, kod pronalaženja izgubljenih blokova, kod sabijanja datoteka, ili kod pripremanja disk jedinica za korišćenje. U poslednjem slučaju se određuje namena pojedinih blokova diska (Slika 11.5).

blok 0	**prvi (*boot*) blok**
blok 1	**drugi (*super*) blok**
blok 2	**blokovi namenjeni za deskriptore**
…	
blok n	**blokovi namenjeni za sadržaj datotka i za tabele pristupa**
…	

Slika 11.5: Namena pojedinih blokova diska

Prvi (*boot*) blok (Slika 11.5) je rezervisan za podatke, koji su potrebni za pokretanje računara (operativnog sistema), a drugi (*super*) blok sadrži:

1. podatke o nizu susednih blokova, koji su namenjeni za smeštanje deskriptora datoteka (ovi podaci obuhvataju redni broj prvog bloka iz ovog niza, kao i ukupan broj blokova u ovom nizu),
2. podatke o slobodnim mestima za deskriptore datoteka i
3. podatke potrebne za evidenciju (preostalih) slobodnih blokova (u ovu evidenciju se uključuju svi blokovi, koji nisu upotrebljeni za smeštanje sadržaja datoteka ili za tabele pristupa).

Blokovska specijalna datoteka nije nužno vezana za jedan celi disk. Ona se može odnositi na deo diska, koji se naziva **particija** (*partition*) i koji tada predstavlja **logičku disk jedinicu** (*logical volume*). Da bi to bilo moguće, neophodno je raspolagati sredstvima za rukovanje particijama. Rukovanje particijama dozvoljava i formiranje logičkih disk jedinica koje obuhvataju više particija na raznim fizičkim diskovima. To omogućava: (1) brži pristup blokovima logičke disk jedinice, jer istovremeno mogu biti prebacivani blokovi iz raznih particija sa raznih fizičkih diskova (*striping*, RAID0), (2) veću pouzdanost logičke disk jedinice, jer svaki blok može biti repliciran tako da svaka particija sa raznih fizičkih diskova sadrži potpunu kopiju logičke disk jedinice (*mirroring*, RAID1), ili (3) oboje (RAID5).

Pre korišćenja specijalnih datoteka, neophodno je njihovo otvaranje (radi provere prava pristupa uređaju, koga datoteka predstavlja), i eventualno njihovo zaključavanje (radi ostvarenja međusobne isključivosti pristupa uređaju, koga datoteka predstavlja). Nakon korišćenja, sledi eventualno otključavanje i zatvaranje specijalne datoteke. Za podršku otvaranja i zatvaranja, odnosno zaključavanja i otključavanja specijalnih datoteka, neophodno je da one poseduju svoje deskriptore. Deskriptori specijalnih datoteka obuhvataju atribute, kao što su, na primer numerička oznaka vlasnika datoteke, prava pristupa datoteci za njenog vlasnika, za njegove saradnike i za ostale korisnike, ili podatak da li je datoteka zaključana ili ne. Međutim, za operacije čitanja ili pisanja specijalnih datoteka nije potrebno preslikavanje rednih brojeva bajta u redne brojeve

blokova masovne memorije. Umesto toga za njih je potrebno pozivanje odgovarajućih operacija drajvera uređaja, koje ove datoteke predstavljaju. Zato se u deskriptorima specijalnih datoteka ne nalaze podaci o tabeli pristupa, nego podaci o odgovarajućem drajveru i primerku uređaja, koga on opslužuje. Za jednoznačno identifikovanje drajvera uvodi se **redni broj drajvera** (*major number*), koji služi kao indeks za posebnu **tabelu drajvera**. Polja indeksiranog elementa ove tabele sadrže adrese operacija dotičnog drajvera. Prema tome, ako se redni broj drajvera čuva u deskriptoru specijalne datoteke, na osnovu njega je moguće pronaći adrese operacija ovog drajvera (posredstvom odgovarajućeg elementa tabele drajvera). U pomenutom deskriptoru se čuva i **redni broj uređaja** (*minor number*) koga predstavlja specijalna datoteka, da bi se na njega mogla usmeriti odabrana operacija drajvera.

Za izmenu atributa specijalnih datoteka primenljive su sistemske operacije, koje su uvedene s tom namerom za obične datoteke. Isto važi i za sistemske operacije za izmenu imena datoteke i za njeno uništenje. Pri tome, kod uništenja specijalne datoteke, nema oslobađanja blokova, nego se oslobađa samo njen deskriptor. Sistemske operacije prave razliku između običnih i specijalnih datoteka na osnovu posebne oznake, navedene u deskriptoru datoteke.

11.10 STANDARDNI ULAZ I STANDARDNI IZLAZ

Pojmovi datoteke i procesa su čvrsto povezani, jer je aktivnost procesa posvećena obradi podataka, sadržanih u datotekama. Pri tome je tipično da obrađivani podaci stižu u proces iz jedne, ulazne datoteke, a da obrađeni podaci napuštaju proces, završavajući u drugoj, izlaznoj datoteci. Ovakav model obrade podataka je dovoljno čest, da opravda uvođenje naziva **standardni ulaz** (*standard input*) za ulaznu datoteku i **standardni izlaz** (*standard output*) za izlaznu datoteku. Pri tome se podrazumeva da se u toku stvaranja procesa otvore i njegov standardni ulaz i njegov standardni izlaz. Zahvaljujući tome, bez posebnog otvaranja se može čitati standardni ulaz i pisati standardni izlaz. Kao podrazumevajući standardni ulaz služi specijalna datoteka, koja predstavlja tastaturu, a kao podrazumevajući standardni izlaz služi specijalna datoteka, koja predstavlja ekran. U slučaju da proces stvaralac zaustavlja svoju aktivnost, dok traje aktivnost stvorenog procesa, tada je prirodno da stvoreni proces nasledi standardni ulaz i standardni izlaz od procesa stvaraoca i da tako, preuzimajući opsluživanje terminala, nastavi interakciju sa korisnikom. Kao indeks otvorene datoteke, koja odgovara standardnom ulazu, može da služi vrednost 0, a kao indeks otvorene datoteke, koja odgovara standardnom izlazu, može da služi vrednost 1.

11.11 SPAŠAVANJE DATOTEKA

U nadležnosti sloja za rukovanje datotekama nalazi se i podrška **spašavanju** (*backup*) datoteka, čiji cilj je da se redovno prave kopije postojećih (svih, ili samo u međuvremenu izmenjenih, odnosno stvorenih) datoteka. Na osnovu ovakvih kopija moguće je rekonstruisati (*restore*) sadržaj oštećenih datoteka. Do oštećenja datoteka dolazi na razne načine, kao što je, na primer, pojava loših (neispravnih) blokova. Kada se otkriju, loši blokovi se izbacuju iz upotrebe. Jedan način da se to postigne je da se, na primer, formira datoteka loših blokova.

11.12 OSNOVA SLOJA ZA RUKOVANJE DATOTEKAMA

Sloj za rukovanje datotekama se oslanja na operacije drajvera iz sloja za rukovanje kontrolerima. On može da koristi i operacije sloja za rukovanje radnom memorijom, radi zauzimanja bafera, namenjenih za smeštanje kopija blokova, ili kopija deskriptora datoteka, na primer.

11.13 PITANJA

1. Kako se predstavlja sadržaj datoteke?
2. Gde se javlja interna fragmentacija?
3. Šta karakteriše kontinualne datoteke?
4. Koji oblik evidencije slobodnih blokova masovne memorije je podesan za kontinualne datoteke?
5. Šta je eksterna fragmentacija?
6. Šta karakteriše rasute datoteke?
7. Šta karakteriše tabelu pristupa?
8. Šta ulazi u sastav tabele pristupa?
9. Kada rasuta datoteka ne zauzima više prostora na disku od kontinualne datoteke?
10. Koji oblik evidencije slobodnih blokova masovne memorije je podesan za rasute datoteke?
11. Kada dolazi do gubitka blokova prilikom produženja rasute datoteke?
12. Kada dolazi do višestrukog nezavisnog korišćenje istog bloka prilikom produženja rasute datoteke?
13. Kada pregled izmena ukazuje da je sistem datoteka u konzistentnom stanju?
14. Kako se ubrzava pristup datoteci?
15. Od čega zavisi veličina bloka?
16. Šta sadrži deskriptor kontinualne datoteke?
17. Kako se rešava problem eksterne fragmentacije?
18. Kako se ublažava problem produženja kontinualne datoteke?
19. Šta sadrži deskriptor rasute datoteke?
20. Šta je imenik?
21. Šta sadrže elementi tabela otvorenih datoteka?
22. Koje sistemske operacije za rukovanje datotekama postoje?
23. Koji su argumenti sistemskih operacija za rukovanje datotekama?
24. Šta karakteriše specijalne datoteke?
25. Šta sadrži deskriptor specijalne datoteke?
26. Šta omogućuju blokovske specijalne datoteke?
27. Šta omogućuje rukovanje particijama?

12 SLOJ OPERATIVNOG SISTEMA ZA RUKOVANJE RADNOM MEMORIJOM

12.1 ZADATAK SLOJA ZA RUKOVANJE RADNOM MEMORIJOM

Sloj za rukovanje radnom memorijom rukuje fizičkom radnom memorijom. Fizičkoj radnoj memoriji se pristupa posredstvom fizičkog adresnog prostora (koga koristi operativni sistem), ali i preko logičkih adresnih prostora (po jedan za svaki korisnički proces). Logički adresni prostori izoluju procese (međusobno i od operativnog sistema) tako što ograničavaju pristup procesa samo na deo fizičke radne memorije. To se postiže preslikavanjem logičkog adresnog prostora svakog od procesa na samo jedan deo fizičkog adresnog prostora, koji odgovara delu fizičke memorije, dodeljene dotičnom procesu. Preslikavanje obavlja MMU (*Memory Management Unit*) tako što logičke adrese pretvara (translira) u odgovarajuće fizičke adrese. Funkcionisanje i organizacija MMU zavisi od karaktera logičkog adresnog prostora. Logički adresni prostor može biti:

1. kontinualan,
2. sastavljen od segmenata raznih veličina,
3. sastavljen od stranica iste veličine i
4. sastavljen od segmenata raznih veličina koji se sastoje od stranica iste veličine.

12.2 KONTINUALNI LOGIČKI ADRESNI PROSTOR

Kontinualni logički adresni prostor se sastoji od jednog niza uzastopnih logičkih adresa, koji počinje od logičke adrese 0. Veličina kontinualnog logičkog adresnog prostora nadmašuje potrebe prosečnog procesa. Zato, da bi se mogao detektovati pokušaj izlaska procesa iz njegovog logičkog adresnog prostora, neophodno je poznavati najvišu ispravnu logičku adresu procesa. Ona se zove **granična adresa** procesa. Logičke adrese procesa ne smiju biti veće od njegove granične adrese.

Za kontinualni logički adresni prostor se podrazumeva da se niz uzastopnih logičkih adresa preslikava u niz uzastopnih fizičkih adresa. Ova dva niza se razlikuju po tome što niz uzastopnih fizičkih adresa ne počinje od 0 nego od neke adrese koja se zove **bazna adresa**. Znači, dodavanjem bazne adese logičkoj adresi nastaje fizička adresa. U toku translacije logičke adrese u fizičku, MMU poredi logičku i graničnu adresu. Ako je logička adresa veća, MMU izaziva izuzetak, inače sabira baznu adresu sa logičkom adresom da bi dobio fizičku adresu.

12.3 SEGMENTIRANI LOGIČKI ADRESNI PROSTOR

Segmentirani logički adresni prostor (*segmentation*) se sastoji od više nizova uzastopnih logičkih adresa (za svaki segment po jedan niz). Da bi se znalo kom segmentu pripada logička adresa, njeni značajniji biti sadrže adresu njenog segmenta, a manje značajni biti sadrže unutrašnju adresu u njenom segmentu. Pošto je svaki segment kontinualan, on ima svojstva kontinualnog logičkog adresnog prostora. Zato svaki segment karakterišu njegova granična i bazna adresa. U ovom slučaju za translaciju je

potrebna tabela segmenata. Broj njenih elemenata određuje najveći broj segmenata u segmentiranom logičkom adresnom prostoru. Svaki element ove tabele sadrži graničnu i baznu adresu odgovarajućeg segmenta. Adresa segmenta indeksira element tabele segmenata, a njegove graničnu i baznu adresu koristi MMU za translaciju unutrašnje adrese segmenta u fizičku adresu.

12.4 STRANIČNI LOGIČKI ADRESNI PROSTOR

Stranični logički adresni prostor (*paging*) se sastoji od jednog niza uzastopnih logičkih adresa (podeljenog u stranice iste veličine). Da bi se znalo kojoj stranici pripada logička adresa, njeni značajniji biti sadrže adresu njene stranice, a manje značajni biti sadrže unutrašnju adresu u njenoj stranici. Pošto je svaka stranica kontinualna, ona ima svojstva kontinualnog logičkog adresnog prostora. Razlika je da je veličina stranice unapred poznata i jednaka stepenu broja dva. Zato svaku stranicu karakteriše samo njena bazna adresa (graniča adresa nije potrebna, jer je unutrašnja adresa ograničena veličinom stranice). U ovom slučaju za translaciju je potrebna tabela stranica. Broj njenih elemenata jednak je najvećem broju stranica u straničnom logičkom adresnom prostoru. Svaki element ove tabele sadrži baznu adresu odgovarajuće stranice. Adresa stranice indeksira element tabele stranica, a njegovu baznu adresu koristi MMU za translaciju logičke adrese stranice u fizičku adresu.

12.5 STRANIČNO SEGMENTIRANI LOGIČKI ADRESNI PROSTOR

Stranično segmentirani logički adresni prostor (*paging segmentation*) se sastoji od više nizova uzastopnih logičkih adresa (za svaki segment po jedan niz, podeljen u stranice iste veličine). Da bi se znalo kom segmentu i kojoj stranici pripada logička adresa, njeni najznačajniji biti sadrže adresu njenog segmenta, njeni srednji biti sadrže adresu njene stranice, a njeni najmanje značajni biti sadrže unutrašnju adresu u njenoj stranici. Segment je kontinualan, kao i stranica, s tim da je njena veličina unapred poznata i jednaka stepenu broja dva. U ovom slučaju za translaciju je potrebna jedna tabela segmenata i više tabela stranica (za svaki segment po jedna). Elementi tabele segmenata sadrže graničnu adresu stranice u segmentu (koja određuje najveći broj stranica u segmentu) i baznu adresu tabele stranica segmenta (koja određuje početak tabele stranica segmenta). Adresa segmenta indeksira element tabele segmenata. Graniča adresa indeksiranog elementa omogućuje proveru ispravnosti adrese stranice, a njegova bazna adresa omogućuje pristup odgovarajućoj tabeli stranica. Njene elemente indeksira adresa stranice. Baznu adresu stranice iz indeksiranog elementa tabele stranica koristi MMU za translaciju logičke adrese stranice u fizičku adresu.

12.6 ODNOS PROCESA I LOGIČKOG ADRESNOG PROSTORA

Svaki proces mora da ima neophodne translacione podatke koji obuhvataju ili graničnu i baznu adresu, ili tabelu segmenata, ili tabelu stranica, ili tabelu segmenata sa pripadnim tabelama stranica. Translacione podatke pripremi operativni sistem, prilikom stvaranja procesa. Da bi mogao da obavlja translaciju, MMU mora da raspolaže sa odgovarajućim translacionim podacima. Translacioni podaci se menjaju prilikom preključivanja, pa utiču na trajanje preključivanja.

Veličina slike procesa određuje veličinu njegovog logičkog adresnog prostora.

12.7 UPOTREBA KONTINUALNOG LOGIČKOG ADRESNOG PROSTORA

Kontinualni logički adresni prostor se koristi kada je logički adresni prostor svakog procesa manji od raspoloživog fizičkog adresnog prostora. Da bi translacija logičkih adresa u fizičke bila moguća, neophodno je da se u lokacije fizičke radne memorije smesti slika procesa. U ovom slučaju, translacija logičkih adresa u fizičke adrese odgovara dinamičkoj relokaciji i olakšava zamenu slika procesa (*swapping*). Pokušaj procesa da izađe iz svog adresnog prostora izaziva izuzetak na koji reaguje operativni sistem, tako što uništi dotični proces.

12.8 UPOTREBA SEGMENTIRANOG LOGIČKOG ADRESNOG PROSTORA

Segmentacija (segmentirani logički adresni prostor) se koristi kada je važno racionalno korišćenje fizičke radne memorije. I ovde se pretpostavlja da je logički adresni prostor procesa manji od raspoloživog fizičkog adresnog prostora, kao i da smeštanje slike procesa u lokacije fizičke radne memorije prethodi translaciji logičkih adresa u fizičke. U ovom slučaju, segmenti logičkog adresnog prostora procesa sadrže delove njegove slike. Na primer, postoje segment za mašinske naredbe, segment za promenljive i segment za stek. Ako istovremeno postoji više procesa, nastalih na osnovu istog programa, tada oni mogu da dele iste mašinske naredbe. U ovom slučaju, tabele segmenata takvih procesa sadrže isti par granične i bazne adrese deljenog segmenta. Ovaj par omogućuje preslikavanje unutrašnjih adresa segmenta naredbi raznih procesa na iste fizičke adrese lokacija fizičke radne memorije sa deljenim mašinskim naredbama. Izdvajanje promenljivih i steka procesa u posebne segmente je korisno i zbog mogućnosti naknadnog proširenja ovih segmenata (kada to postane potrebno u toku aktivnosti procesa).

Prednost, segmentacije je da ona ubrzava zamenu slika procesa (*swapping*), jer se segment naredbi ne mora izbacivati, ako je ranije već izbačen u masovnu memoriju, niti ubacivati, ako već postoji u fizičkoj radnoj memoriji. Jasno, rukovanje segmentima mora voditi posebnu evidenciju o segmentima, prisutnim u fizičkoj radnoj memoriji, da bi bilo moguće otkriti kada se isti segment može iskoristiti u slikama raznih procesa. Ovakva evidencija obuhvata jednoznačnu oznaku segmenta, podatak o broju korisnika segmenta, dužinu segmenta, odnosno, njegovu graničnu i baznu adresu. Radi očuvanja konzistentnosti evidencije segmenata, operacije za rukovanje ovom evidencijom moraju obezbediti sinhronizaciju procesa.

Prethodno opisana (osnovna) segmentacija se razvija u punu segmentaciju, ako se dozvoli da svakom potprogramu ili promenljivoj odgovara poseban segment. To, na primer, dozvoljava deljenje potprograma između raznih procesa, ako se segment istog potprograma uključi u slike više procesa. U tom slučaju, stvaraju se uslovi za dinamičko linkovanje (povezivanje) potprograma za program. Ono se ne dešava u toku pravljenja izvršne datoteke, nego u trenutku prvog poziva potprograma, kada se u fizičkoj radnoj memoriji pronalazi njegov segment ili se stvara ako ne postoji. Dinamičko linkovanje

doprinosi racionalnom korišćenju masovne memorije, jer omogućava da se često korišćeni potprogrami ne multipliciraju u raznim izvršnim datotekama.

Puna segmentacija dozvoljava i deljenje promenljivih između raznih procesa, ako se segment iste promenljive uključi u slike više procesa. Jasno, ovakav način ostvarenja saradnje procesa zahteva njihovu sinhronizaciju, radi očuvanja konzistentnosti deljenih promenljivih. Za segmente sa deljenim promenljivim su važna i prava pristupa segmentu, jer nekim od procesa treba dozvoliti samo da čitaju deljenu promenljivu, a drugima treba dozvoliti i da pišu u deljenu promenljivu. Zato se elementi tabele segmenata proširuju oznakom prava pristupa segmentu. U pogledu prava pristupa, segmenti se ne razlikuju od datoteka. Prema tome, prava pristupa segmentu obuhvataju pravo čitanja, pravo pisanja i pravo izvršavanja segmenta. Prva dva prava se primenjuju na segmente, koji sadrže promenljive, a treće pravo se primenjuje na segmente sa mašinskim naredbama programa ili potprograma.

Na pokušaj narušavanja prava pristupa segmentu reaguje *MMU*, generisanjem prekida (izuzetka), što dovodi do uništenja aktivnog procesa.

Rukovanje segmentima se obavlja posredstvom sistemskih programa, kao što su kompajler ili linker.

12.9 UPOTREBA STRANIČNOG LOGIČKOG ADRESNOG PROSTORA

Stranični logički adresni prostor se koristi kada je logički adresni prostor tipičnog procesa veći od raspoloživog fizičkog adresnog prostora, pa slika procesa ne može da stane u fizičku radnu memoriju. Stranični logički adresni prostor se naziva i **virtuelni adresni prostor**. On pripada **virtuelnoj memoriji** i sadrži **virtuelne adrese**. Stranice virtuelnog adresnog prostora sa slikom procesa se nalaze na masovnoj memoriji. Postoji posebna evidencija o tome gde se one tačno nalaze na masovnoj memoriji. Pošto je za izvršavanje mašinske naredbe neophodno da u fizičkoj radnoj memoriji budu samo bajti njenog mašinskog formata, kao i bajti njenih operanada, u fizičkoj radnoj memoriji moraju da se nalaze samo kopije virtuelnih stranica koje sadrže pomenute bajte.

Podrazumeva se da se kopije neophodnih virtuelnih stranica, kada zatreba, automatski prebacuju iz masovne memorije u fizičku radnu memoriju i obrnuto. Do prebacivanja u obrnutom smeru dolazi, kada je potrebno osloboditi lokacije fizičke radne memorije sa kopijama u međuvremenu izmenjenih virtuelnih stranica. Ovo prebacivanje se obavlja, da bi se oslobodila fizička radna memorija i, istovremeno, obezbedilo da masovna memorija uvek sadrži ažurnu sliku procesa. Da bi se znalo koja kopija virtuelne stranice je izmenjena, neophodno je automatski registrovati svaku izmenu svake od kopija virtuelnih stranica.

Fizička radna memorija sadrži kopije pojedinih virtuelnih stranica, pa je prirodno da i ona bude izdeljena u fizičke stranice u kojima se nalaze kopije virtuelnih stranica. To znači da se fizička adresa sastoji od adrese fizičke stranice (u značajnijim bitima fizičke adrese) i od unutrašnje adrese (u manje značajnim bitima fizičke adrese). Pošto su virtuelne i fizičke stranice iste veličine, unutrašnja adresa (manje značajni biti) virtuelne adrese se poklapa sa unutrašnjom adresom (manje značajnim bitima) fizičke adrese. Zato

je za translaciju virtuelne adrese u fizičku potrebno samo zameniti adresu virtuelne stranice adresom fizičke stranice. Zbog toga se u elementu tabele stranica (koga indeksira adresa virtuelne stranice) kao bazna adresa nalazi adresa fizičke stranice (koja sadrži kopiju dotične virtuelne stranice). Ako virtuelna stranica nije kopirana u neku fizičku stranicu, tada translacija njenih virtuelnih adresa u fizičke nije moguća. U tom slučaju, to mora biti registrovano u odgovarajućem elementu tabele stranica. Takvu ulogu ima **bit prisustva**. Uz bit prisustva, elementi tabele stranica sadrže i **bit referenciranja** (koji pokazuje da li je bilo pristupanja kopiji virtuelne stranice) i **bit izmene** (koji pokazuje da li je bilo izmena kopije virtuelne stranice). Poslednja dva bita se automatski postavljaju.

Kada pokušaj translacije virtuelne adrese bude neuspešan (jer bit prisustva ukaže da se kopija odgovarajuće virtuelne stranice ne nalazi u fizičkoj radnoj memoriji), MMU izaziva **stranični prekid** (*page fault*). U obradi straničnog prekida se prebaci kopija potrebne virtuelne stanice u neku fizičku stranicu. Adresa ove fizičke stranice postaje bazna adresa i smesti se u odgovarajući element tabele stranica. U ovom elementu se istovremeno postavi bit prisustva, a očiste se bit referenciranja i bit izmene. Nakon toga, ponavlja se neuspešna translacija virtuelne adrese.

Važno je uočiti da se kopije virtuelnih stranica prebacuju na zahtev (*demand paging*), a ne unapred (*prepaging*), jer u opštem slučaju ne postoji način da se predvidi redosled korišćenja virtuelnih stranica.

Praktična upotrebljivost virtuelne memorije se temelji na svojstvu **lokalnosti izvršavanja programa**. Zahvaljujući ovome svojstvu, za izvršavanje programa je dovoljno da u fizičkoj radnoj memoriji uvek bude samo deo progama.

12.10 UPOTREBA STRANIČNO SEGMENTIRANOG LOGIČKOG ADRESNOG PROSTORA

Stranična segmentacija (stranično segmentirani logički adresni prostor) se koristi kada su segmenti potrebni radi racionalnog korišćenja fizičke radne memorije, a njihova veličina nadmašuje veličinu raspoložive fizičke radne memorije. U tom slučaju, svaki segment uvodi sopstveni virtuelni adresni prostor. Zahvaljujući tome, stranice virtuelnog adresnog prostora segmenta se nalaze u masovnoj memoriji, a u fizičkim stranicama se nalaze samo kopije neophodnih virtuelnih stranica segmenta.

Prednost stranične segmentacije je da ona omogućava dinamičko proširenje segmenata (dodavanjem novih stranica), što je važno za segmente promenljivih i steka.

Stranična segmentacija može otvorenim datotekama dodeljivati posebne segmente i na taj način ponuditi koncept **memorijski preslikane datoteke** (*memory mapped file*). Pristup ovakvoj datoteci ne zahteva sistemske operacije za čitanje, pisanje ili pozicioniranje, jer se direktno pristupa lokacijama sa odgovarajućim sadržajem datoteke. Mana koncepta memorijski preslikane datoteke je da se veličina datoteke izražava celim brojem stranica, jer nema načina da operativni sistem odredi koliko je popunjeno bajta iz poslednje stranice. Takođe, problem je i što virtuelni adresni prostor segmenta može biti suviše mali za pojedine datoteke.

12.11 ZADACI SLOJA ZA RUKOVANJE FIZIČKOM RADNOM MEMORIJOM

Zadatak sloja za rukovanje fizičkom radnom memorijom je da omogući zauzimanje zona susednih lokacija (sa uzastopnim adresama) slobodne fizičke radne memorije, kao i da omogući oslobađanje prethodno zauzetih zona fizičke radne memorije. Radi toga ovaj sloj nudi operacije zauzimanja i oslobađanja (fizičke radne memorije). Argument poziva operacije zauzimanja je dužina zauzimane zone (broj njenih lokacija), a povratna vrednost ovog poziva je adresa zauzete zone (adresa prve od njenih lokacija), ili indikacija da je operacija zauzimanja završena neuspešno. Argumenti poziva operacije oslobađanja su adresa oslobađane zone (adresa prve od njenih lokacija) i njena dužina (broj njenih lokacija).

Uspešno obavljanje zadatka sloja za rukovanje fizičkom radnom memorijom se temelji na vođenju evidencije o slobodnoj fizičkoj radnoj memoriji. Kada podržava virtuelnu memoriju, ovaj sloj mora svakom procesu dodeliti dovoljan broj fizičkih stranica za kopije potrebnih virtuelnih stranica.

12.12 RASPODELA FIZIČKE RADNE MEMORIJE

Fizička radna memorija se deli na lokacije koje stalno zauzima operativni sistem i na preostale slobodne lokacije koje su na raspolaganju za stvaranje procesa i za druge potrebe.

Operativni sistem obično zauzima lokacije sa početka i, eventualno, sa kraja fizičkog adresnog prostora, a između njih se nalaze lokacije slobodne fizičke radne memorije. Na početku fizičkog adresnog prostora su, najčešće, lokacije, namenjene za tabelu prekida. Iza njih slede lokacije sa naredbama i promenljivim operativnog sistema (koje obuhvataju i prostor za smeštanje deskriptora i sistemskih stekova procesa). Na kraju fizičkog adresnog prostora su, najčešće, lokacije, koje odgovaraju registrima kontrolera.

U slučaju da procesor podržava virtuelnu memoriju, praksa je da se samo donja polovina (sa nižim adresama) virtuelnog adresnog prostora stavi na raspolaganje svakom procesu, a da gornja polovina (sa višim adresama) virtuelnog adresnog prostora bude rezervisana za operativni sistem. Gornja polovina virtuelnog adresnog prostora se zato može nazvati sistemski virtuelni adresni prostor, a donja polovina virtuelnog adresnog prostora se može nazvati korisnički virtuelni adresni prostor. Za virtuelne adrese iz prve polovine sistemskog virtuelnog adresnog prostora se ne vrši translacija (na primer, tako što se deo manje značajnih bita virtuelne adrese koristi kao fizička adresa) i podrazumeva se da se tu nalazi rezidentni deo operativnog sistema, koji je stalno prisutan u fizičkoj radnoj memoriji. Virtulene adrese iz druge polovine sistemskog virtuelnog adresnog prostora se transliraju na uobičajeni način i podrazumeva se da se one odnose na nerezidentni deo operativnog sistema, koji se po potrebi prebacuje u fizičku radnu memoriju.

12.13 EVIDENCIJA SLOBODNE FIZIČKE RADNE MEMORIJE

Sloj za rukovanje fizičkom radnom memorijom obavezno vodi evidenciju o slobodnoj fizičkoj radnoj memoriji. Za potrebe kontinualnog ili segmentiranog logičkog

adresnog prostora ova evidencija može da bude u obliku niza bita (*bit map*), u kome svaki bit odgovara grupi susednih lokacija. Podrazumeva se da je broj lokacija u ovakvoj grupi unapred zadan. On ujedno predstavlja jedinicu u kojoj se izražava dužina zauzimane i oslobađane zone fizičke radne memorije. Ako je grupa lokacija slobodna, njoj odgovarajući bit sadrži 1. Inače, on sadrži 0. Mana ovakve evidencije je da su i operacija zauzimanja i operacija oslobađanja dugotrajne, jer prva pretražuje evidenciju, radi pronalaženja dovoljno dugačkog niza jedinica, a druga postavlja takav niz jedinica u evidenciju. Zato se češće evidencija slobodne fizičke radne memorije pravi u obliku liste slobodnih odsečaka fizičke radne memorije. Na početku rada operativnog sistema ovakva lista sadrži jedan odsečak, koji obuhvata celu fizičku slobodnu radnu memoriju. Ovakav odsečak se drobi u više kraćih odsečaka, kao rezultat višestrukog stvaranja i uništavanja procesa u slučajnom redosledu. Novonastali kraći odsečci se nalaze na mestu slika uništenih procesa, a između njih su slike postojećih procesa. Na početku svakog odsečka su njegova dužina i adresa narednog odsečka. Broj lokacija, potrebnih za smeštanje dužine dotičnog odsečka i adrese narednog odesečka, određuje najmanju dužinu odsečka i može da predstavlja jedinicu u kojoj se izražavaju dužine odsečaka (odnosno, dužine zauzimanih i oslobađanih zona fizičke radne memorije). Odsečci su uređeni u rastućem redosledu adresa njihovih početnih lokacija. To, prilikom oslobađanja zone fizičke radne memorije, olakšava operaciji oslobađanja:

1. da pronađe dva susedna odsečka, koji mogu da se spoje u jedan, kada se između njih ubaci oslobađana zona, ili
2. da pronađe odsečak, kome može da se doda (spreda ili straga) oslobađana zona, ili
3. da pronađe mesto u listi u koje će oslobađana zona biti uključena kao poseban odsečak.

Listu slobodnih odsečaka pretražuje i operacija zauzimanja, radi pronalaženja dovoljno dugačkog odsečka. Pri tome se zauzima samo deo odsečka, koji je jednak zauzimanoj zoni, dok preostali deo odsečka ostaje u listi kao novi odsečak. Na ovaj način se odsečci dalje usitnjavaju. To dovodi do eksterne fragmentacije. Ona posredno uzrokuje neupotrebljivost odsečaka, jer onemogućuje zauzimanje zone fizičke radne memorije, čija dužina je veća od dužine svakog od postojećih odsečaka, bez obzira na činjenicu da je suma dužina postojećih odsečaka veća od dužine zauzimane zone. Iskustvo pokazuje da se eksterna fragmentacija povećava, ako se, umesto traženja prvog dovoljno dugačkog odsečka (*first fit*), pokušava naći najmanji dovoljno dugačak odsečak (*best fit*), ili najveći dovoljno dugačak odsečak (*worst fit*). Poboljšanje ne nudi ni ideja da lista odsečaka bude ciklična i da se pretražuje ne od početka, nego od tačke u kojoj je zaustavljeno poslednje pretraživanje (*next fit*). Ideja da dužina zauzimanih zona bude uvek jednaka stepenu broja 2 (*quick fit*) i da postoji posebna lista odsečaka za svaku od mogućih dužina takođe ima manu, jer, pored eksterne, uvode i internu fragmentaciju, pošto se na ovaj način u proseku zauzimaju duže zone od stvarno potrebnih, čime nastaje neupotrebljiva radna memorija.

Problem eksterne fragmentacije se može rešiti sabijanjem (*compaction*) slika procesa, čime se sve slike procesa pomeraju na jedan kraj fizičke radne memorije, tako da na drugom kraju bude slobodna fizička radna memorija. U toku sabijanja, moraju se

menjati bazne adrese (segmenata) pojedinih procesa. Mana sabijanja je njegova dugotrajnost (sporost).

Za potrebe virtuelnog adresnog prostora, evidencija slobodne fizičke radne memorije može da bude u obliku niza bita, u kome svaki bit odgovara slobodnoj fizičkoj stranici. Alternativa je da se slobodne fizičke stranice vežu u listu. Međutim, atraktivna je i evidencija o slobodnim odsečcima veličine jednake multiplu fizičke stranice (*quick fit*, *buddy system*), jer se u virtuelnom adresnom prostoru uvek zauzima celi broj stranica.

Važno je uočiti da rukovanje evidencijom sistemske slobodne radne memorije zahteva sinhronizaciju procesa, u toku čije aktivnosti se istovremeno obavljaju operacije zauzimanja i oslobađanja. Sinhronizacija treba da obezbedi međusobnu isključivost obavljanja ovih operacija, radi očuvanja konzistentnosti pomenute evidencije.

Za virtuelnu memoriju je važno pitanje dužine stranice. Za dugačke stranice postaje izražen problem interne fragmentacije, jer sve stranice nisu uvek potpuno iskorišćene, pa se u njima javljaju neupotrebljive lokacije. Za kratke stranice postaje izražen problem veličine tabele stranica, jer tada virtuelni adresni prostor ima više stranica, pa zato i tabela stranica ima više elemenata. Praksa je veličinu stranice smestila između 512 i 8192 bajta.

12.14 DODELA FIZIČKIH STRANICA PROCESIMA

Za aktivnost procesa je potrebno da procesoru na raspolaganju budu kopije svih virtuelnih stranica koje su neophodne za izvršavanje pojedinih mašinskih naredbi. Za čuvanje kopija tih stranica procesu mora biti dodeljeno dovoljno fizičkih stranica koje obrazuju **minimalan skup**.

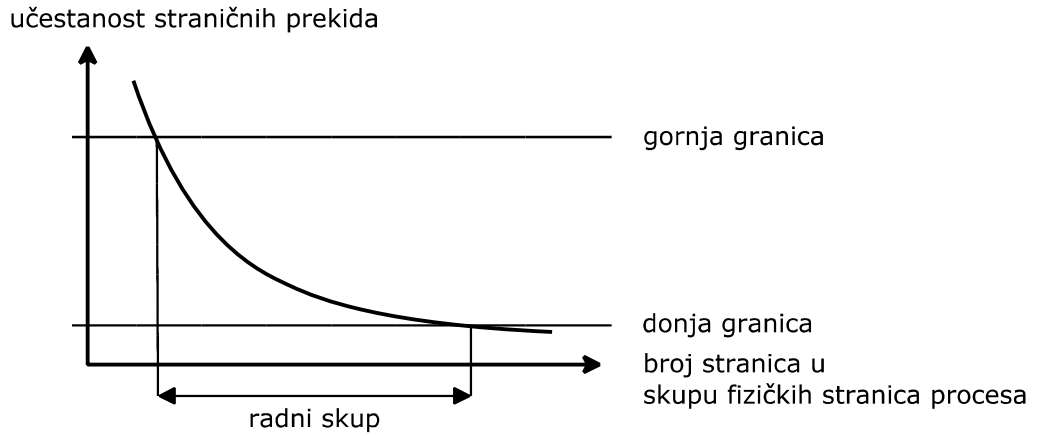
Na primer, za procesor, čije naredbe imaju najviše dva operanda, minimalni skup sadrži šest fizičkih stranica, jer se, u ekstremnom slučaju, i bajti mašinske naredbe, kao i bajti oba njena operanda, mogu nalaziti u različitim susednim fizičkim stranicama. Pošto se naredba može izvršiti samo kada su u fizičkoj radnoj memoriji prisutni svi bajti njenog mašinskog formata i svi bajti njenih operanada, prethodno pomenuti ekstremni slučaj uslovljava da je pridruživanje minimalnog skupa procesu preduslov bilo kakve njegove aktivnosti.

Kada se, u toku aktivnosti procesa, desi stranični prekid, koji zahteva prebacivanje kopije nove virtuelne stranice u radnu memoriju, pre zahtevanog prebacivanja neophodno je razrešiti dilemu da li uvećati skup fizičkih stranica procesa novom fizičkom stranicom i u nju smestiti kopiju nove virtuelne stranice, ili u postojećem skupu fizičih stranica procesa zameniti sadržaj neke od njih kopijom nove virtuelne stranice.

Uvećanje skupa fizičkih stranica procesa ima smisla samo ako to dovodi do smanjivanja učestanosti straničnih prekida (njihovog prosečnog broja u jedinici vremena). Znači, kada je, u toku aktivnosti procesa, učestanost straničnih prekida iznad neke (iskustveno određene) gornje granice, tada ima smisla uvećanje skupa fizičih stranica procesa, da bi se učestanost straničnih prekida svela na prihvatljiv nivo. To je važno, jer obrada svakog prekida troši procesorsko vreme, pa veliki broj obrada straničnih prekida može u potpunosti da angažuje procesor i da tako vrlo uspori, ili

potpuno spreči njegovu bilo kakvu korisnu aktivnost (*trashing*). Međutim, ako je učestanost straničnih prekida ispod neke (iskustveno određene) donje granice, tada ima smisla smanjenje skupa fizičkih stranica procesa, jer i sa manjim skupom fizičkih stranica učestanost straničnih prekida ostaje u prihvatljivom rasponu. U opštem slučaju učestanost straničnih prekida ne pada na nulu, ako sve kopije virtuelnih stranica procesa ne mogu stati u radnu memoriju. Smanjenje skupa fizičkih stranica procesa je važno, jer se tako omogućuje neophodni rast skupova stranica drugih procesa.

U slučaju da je učestanost straničnih prekida između pomenute dve granice, tada nema potrebe za izmenom broja fizičkih stranica u skupu fizičkih stranica aktivnog procesa (Slika 12.1). U ovom slučaju skup fizičkih stranica (odnosno, njima odgovarajući skup virtuelnih stranica) obrazuje **radni skup** (*working set*).



Slika 12.1: Odnos učestanosti straničnih prekida i broja stranica u skupu fizičkih stranica procesa

Radni skup procesa nije statičan. U proseku, on se sporo menja u toku aktivnosti procesa (iako su, povremeno, moguće značajne kratkotrajne varijacije radnog skupa). Važno je uočiti da se u toku aktivnosti procesa obavezno javlja *trashing*, kada njegov radni skup ne može da stane u radnu memoriju. U ovom slučaju pomaže izbacivanje (*swapping*) procesa, dok se ne oslobodi dovoljan broj fizičkih stranica. Ovaj pristup ima smisla samo ako je radni skup procesa manji od ukupne slobodne fizičke radne memorije.

Stepen multiprogramiranja kod virtuelne memorije zavisi od broja radnih skupova, koji se istovremeno mogu smestiti u raspoloživu fizičku radnu memoriju. Za uspeh koncepta virtuelne memorije važno je da se stalno prate radni skupovi istovremeno postojećih procesa i da se povremeno izbacuju procesi, čim fizička radna memorija postane pretesna za sve radne skupove (*load control*). Na ovaj način se oslobađaju fizičke stranice za preostale procese, neophodne za smeštanje njihovih radnih skupova. Prilikom kasnijeg ubacivanja procesa, uputno je ubacivati kopije svih virtuelnih stranica, koje obrazuju njegov radni skup.

12.15 OSLOBAĐANJE FIZIČKIH STRANICA PROCESA

Pad učestanosti straničnih prekida ispod donje granice ukazuje na mogućnost smanjenja radnog skupa. U ovoj situaciji je potrebno odlučiti koju virtuelnu stranicu izbaciti iz radnog skupa, odnosno osloboditi. Za oslobađanje kao kandidat se nameće virtuelna stranica, koja neće biti referencirana do kraja aktivnosti procesa, ili će biti referencirana iza svih ostalih virtuelnih stranica iz radnog skupa (pod referenciranjem se podrazumeva pristup bilo kojoj lokaciji stranice, radi preuzimanja ili izmene njenog sadržaja).

Povećanje učestanosti straničnih prekida preko gornje granice ukazuje na potrebu proširenja radnog skupa. U ovom slučaju, potrebno je procesu pridružiti novu fizičku stranicu, da bi se u nju smestila kopija potrebne virtuelne stranice. Ako nema slobodnih fizičkih stranica, tada se oslobađa, kada je to moguće, fizička stranica, koja je pridružena nekom drugom procesu. Time se radni skup ovog drugog procesa smanjuje. U oslobođenoj fizičkoj stranici zatečenu kopiju **zamenjuje** (*replacement*) kopija potrebne virtuelne stranice.

Do zamene kopija virtuelnih stranica dolazi i kada se javi potreba za ubacivanjem kopije nove stranice, a učestanost straničnih prekida je između gornje i donje granice, pa se radni skup aktivnog procesa ne proširuje, nego se oslobađa jedna od fizičkih stranica iz njegovog radnog skupa. U oba prethodna slučaja kandidat za zamenu je fizička stranica, koja sadrži kopiju virtuelne stranice sa najstarijom referencom.

I u slučaju pada učestanosti straničnih prekida ispod donje granice i u slučaju povećanja učestanosti straničnih prekida iznad gornje granice, kao i u slučaju kada je učestanost straničnih prekida između gornje i donje granice, javlja se potreba za oslobađanjem fizičkih stranica. Izbor fizičke stranice za oslobađanje se može vršiti na razne načine, po raznim algoritmima. Ovakvi algoritmi se nazivaju algoritmi zamene stranica (*page replacement algorithms*), jer se zatečeni sadržaj oslobađane fizičke stranice zamenjuje novim sadržajem.

Oslobađanje fizičkih stranica može biti zasnovano samo na upotrebi bita referenciranja i bita izmene. U ovom pristupu se pronalazi fizička **stranica koja nije korišćena u nekom prethodnom periodu** (*Not Recently Used - NRU*) tako što se posmatra dvobitni broj, čiji značajniji bit odgovara bitu referenciranja, a manje značajan bit odgovara bitu izmene (podrazumeva se da *MMU* automatski postavlja ove bite nakon svakog referenciranja kopije virtuelne stranice, odnosno nakon svake njene izmene, a da se u pravilnim vremenskim razmacima dešavaju vremenski prekidi, čiji obrađivač čisti bit referenciranja). Takođe se podrazumeva da se oba bita čiste i prilikom oslobađanja, odnosno prilikom zauzimanja fizičke stranice (prilikom zamene njenog sadržaja). Prema tome, pomenuti dvobitni broj sadrži 00, kada kopija virtuelne stranice nije korišćena nakon nekog od poslednjih vremenskih prekida (znači, niti referencirana niti izmenjena). Pomenuti dvobitni broj sadrži 01, kada kopija virtuelne stranice nije referencirana nakon nekog od poslednjih vremenskih prekida (ali je prethodno izmenjena). Ovaj broj sadrži 10, kada kopija virtuelne stranice nije izmenjena, ali je referencirana nakon poslednjeg vremenskog prekida. Pomenuti dvobitni broj sadrži 11, kada je kopija virtuelne stranice izmenjena i, uz to, referencirana nakon poslednjeg vremenskog prekida. Kandidati za

zamenu su stranice, čiji dvobitni broj je najmanji.

Prethodno opisani pristup oslobađanja fizičkih stranica je efikasan, ali nedovoljno precizno procenjuje koja fizička stranica će biti ubrzo referencirana.

U opštem slučaju nema načina da se precizno ustanovi da li će i kada neka virtuelna stranica biti referencirana. Ipak, zahvaljujući lokalnosti referenciranja, odnosno zapažanju da se pristupi lokacijama sa bliskim adresama dešavaju u bliskim trenucima, moguće je sa priličnom pouzdanošću zaključivati o referenciranju stranica u neposrednoj budućnosti na osnovu njihovog referenciranja u neposrednoj prošlosti. Prema tome, najmanju verovatnoću da bude referencirana u neposrednoj budućnosti ima stranica, čije poslednje referenciranje je najstarije, odnosno prethodi referenciranju svih ostalih stranica iz radnog skupa. Za pronalaženje **najmanje korišćene fizičke stranice** (*Least Recently Used - LRU*), odnosno stranice sa najstarijom referencom, neophodno je registrovanje starosti referenciranja. To je moguće, ako postoji brojač, koji se automatski uvećava za jedan nakon izvršavanja svake naredbe. Ako se njegova zatečena vrednost automatski pridružuje virtuelnoj stranici prilikom njenog svakog referenciranja, tada je stranici sa najstarijom referencom pridružena najmanja vrednost ovog brojača. Opisani pristup ima samo teoretsko značenje, jer se oslanja na hardver, koji u opštem slučaju nije raspoloživ. Međutim, moguće je i **softverski simulirati pronalaženje najmanje korišćene fizičke stranice** (*Not Frequently Used - NFU/aging*). Ovakva simulacija se zasniva na korišćenju bita referenciranja svake virtuelne stranice i na uvođenju **polja starosti referenci** u elemente tabele stranica. Ovo polje sadrži n bita, po jedan bit za svaku od vrednosti bita referenciranja u poslednjih n trenutaka. Polje starosti referenci se periodično ažurira, tako što se iz njega periodično izbacuje najstariji bit referenciranja. To je zadatak obrađivača vremenskog prekida koji (1) pomera u desno za jedan bit polje starosti referenci svake od virtuelnih stranica iz radnog skupa aktivnog procesa, (2) dodaje s leva, u upražnjenu bitnu poziciju ovog polja, zatečeni sadržaj bita referenciranja dotične virtuelne stranice i (3) zatim očisti bit referenciranja. Na ovaj način polje starosti referenci sadrži najmanju vrednost za virtuelnu stranicu, koja je najranije referencirana u prethodnih n trenutaka.

Polje starosti referenci se može iskoristiti i za određivanje radnog skupa. Kriterij može biti postavljenost nekog od k najznačajnijih bita iz polja starosti referenci. Po tom kriteriju radnom skupu pripadaju sve virtuelne stranice, koje imaju bar jedan bit postavljen u najznačajnijih k bita svog polja starosti referenci.

Prethodno opisani pristupi oslobađanja fizičkih stranica (*NRU, LRU, NFU*) obezbeđuju smanjenje učestanosti straničnih prekida nakon povećanja broja fizičkih stranica procesa. Znači, obavljanje liste istih zahteva za pristupe virtuelnim stranicama dovodi do pojave manje straničnih prekida nakon povećanja broja fizičkih stranica procesa, nego pre toga. Ovo je važno istaći, jer svi pristupi oslobađanja fizičkih stranica ne dovode obavezno do smanjenja učestanosti straničnih prekida nakon povećanja broja fizičkih stranica procesa. To je karakteristično, na primer, za pristup, kod koga se oslobađa fizička **stranica sa najstarijim sadržajem** (*First In First Out - FIFO*), bez obzira da li je ona skoro referencirana. Ovaj pristup ima tendenciju da ne oslobađa fizičke stranice, čiji sadržaj je svežiji, čak i ako one neće biti uskoro referencirane.

Međutim, mana poslednje pomenutog pristupa se otklanja, ako se on modifikuje, tako da se za oslobađanje prvo traži fizička stranica sa najstarijim nereferenciranim sadržajem, a ako su sadržaji svih fizičkih stranica referencirani, tek tada se oslobađa fizička stranica sa najstarijim sadržajem (*second chance* i *clock* pristupi). U sva tri poslednje pomenuta pristupa (*FIFO, second chance* i *clock*) fizičke stranice procesa se uvezuju u listu. Pri tome položaj u listi ukazuje na starost sadržaja fizičke stranice. U *clock* algoritmu zamene ovakva lista je kružna, a poseban pokazivač, kao kazaljka na satu, pokazuje na stranicu sa najstarijim sadržajem. Postoji i varijanta *clock* pristupa (*wsclock*) u kome se uz svaku fizičku stranicu iz liste čuva i podatak o trenutku poslednjeg referenciranja. Za sve fizičke stranice, za koje ovaj trenutak ispada iz unapred određenog vremenskog intervala (gledajući u prošlost), se smatra da ne spadaju u radni skup, pa su one kandidati za zamenu.

Pristup *NFU/aging* i pristup *wsclock* imaju najveću praktičnu važnost.

Virtuelna memorija pokazuje najbolje rezultate, kada uvek ima slobodnih fizičkih stranica. To se može postići, ako se uvede poseban **stranični sistemski proces**, koji se periodično aktivira, da bi oslobodio izvestan broj fizičkih stranica. On, pri tome, odabira fizičke stranice za oslobađanje po nekom od prethodno opisanih algoritama zamene. Zadatak straničnog sistemskog procesa je i da u masovnu memoriju prebacuje izmenjene kopije virtuelnih stranica i da tako čuva ažurnost masovne memorije.

Opisani pristupi oslobađanja fizičkih stranica pokazuju da se u praksi ne prati učestanost straničnih prekida. Umesto toga, broj fizičkih stranica procesa varira između minimalnog skupa i iskustveno određenog maksimalnog skupa. Pri tome stranični prekidi izazivaju povećanje broja fizičkih stranica procesa do maksimalne veličine, a stranični sistemski proces se brine o njegovom smanjivanju.

12.16 IMPLEMENTACIJA UPRAVLJANJA VIRTUELNOM MEMORIJOM

Sloj za rukovanje virtuelnom memorijom podržava operacije zauzimanja i oslobađanja, a oslanja se na stranični sistemski proces i obrađivače vremenskog i straničnog prekida. Obrađivač straničnog prekida se aktivira, kada je referencirana virtuelna stranica, čija kopija nije prisutna u fizičkoj radnoj memoriji, odnosno, u nekoj od njenih fizičkih stranica. Ako je, greškom, referencirana virtuelna stranica, koja uopšte ne postoji u slici procesa, obrađivač straničnog prekida završava aktivnost prekinutog procesa (uz odgovarajuću poruku). Inače, ovaj obrađivač odabira slobodnu fizičku stranicu i prema njoj usmerava prenos kopije potrebne virtuelne stranice sa masovne memorije. Kada se taj prenos završi, obrađivač straničnog prekida ažurira polja odgovarajućeg elementa tabele stranica i omogućuje nastavak aktivnosti prekinutog procesa. Kada nema slobodne fizičke stranice, ovaj obrađivač oslobađa neku od fizičkih stranica. Ako ona sadrži izmenjenu kopiju virtuelne stranice, on pokreće prenos ove kopije u masovnu memoriju, radi ažuriranja odgovarajuće virtuelne stranice. Po završetku ovoga prenosa, obrađivač straničnog prekida usmerava ka oslobođenoj fizičkoj stranici prenos kopije potrebne virtuelne stranice. Nakon završetka ovog prenosa i ažuriranja polja odgovarajućih elemenata tabele stranica, on omogućava nastavak aktivnosti prekinutog procesa. Za uspešno obavljanje posla, obrađivaču straničnog

prekida je potrebna evidencija o položaju virtuelnih stranica u masovnoj memoriji. On, takođe, koristi i evidenciju slobodnih fizičkih stranica. Nju koriste i operacija zauzimanja i operacija oslobađanja, pa rukovanje ovom evidencijom mora obezbediti sinhronizaciju procesa, i to onemogućenjem prekida. Operacija zauzimanja omogućuje zauzimanje bar minimalnog skupa, radi stvaranja procesa, a operacija oslobađanja oslobađa sve fizičke stranice iz radnog skupa unišavanog procesa.

12.17 OSNOVA SLOJA ZA RUKOVANJE VIRTUELNOM MEMORIJOM

Sloj za rukovanje virtuelnom memorijom se oslanja na operacije sloja za rukovanje kontrolerima, da bi obezbedio prenos kopija virtuelnih blokova na relaciji masovna i radna memorija i da bi smestio adrese svojih obrađivača prekida u odgovarajuće elemente tabele prekida.

12.18 PITANJA

1. Kakav može biti logički adresni prostor?
2. Šta karakteriše kontinualni logički adresni prostor?
3. Šta karakteriše segmentirani logički adresni prostor?
4. Šta karakteriše stranični logički adresni prostor?
5. Šta karakteriše stranično segmentirani logički adresni prostor?
6. Šta karakteriše translacione podatke?
7. Šta karakteriše translaciju logičkih adresa kontinualnog logičkog adresnog prostora u fizičke adrese?
8. Koji logički adresni prostor se koristi kada veličina fizičke radne memorije prevazilazi potrebe svakog procesa?
9. Šta karakteriše segmentaciju?
10. Koji logički adresni prostor se koristi kada je važno racionalno korišćenje fizičke radne memorije?
11. Koji logički adresni prostor se koristi kada veličina fizičke radne memorije nedovoljna za pokrivanje potreba tipičnog procesa?
12. Šta sadrže elementi tabele stranica?
13. Šta karakteriše virtuelni adresni prostor?
14. Po kom principu se prebacuju kopije virtuelnih stranica?
15. Šta karakteriše straničnu segmentaciju?
16. Koji logički adresni prostor se koristi kada je važno racionalno korišćenje fizičke radne memorije, a ona ima nedovoljnu veličinu?
17. Kako se deli fizička radna memorija?
18. Kako se deli virtuelni adresni prostor?
19. U kom obliku može biti evidencija slobodne fizičke memorije?
20. Kod kog adresnog prostora se javlja eksterna fragmentacija?
21. Kako se nazivaju skupovi fizičkih stranica, koji se dodeljuju procesima?
22. Kada treba proširiti skup fizičkih stranica procesa?
23. Kada treba smanjiti skup fizičkih stranica procesa?
24. Kada ne treba menjati veličinu skupa fizičkih stranica procesa?
25. Koji pristupi oslobađanja fizičkih stranica obezbeđuju smanjenje učestanosti

straničnih prekida nakon povećanja broja fizičkih stranica procesa?
26. Koji pristupi oslobađanja fizičkih stranica koriste bit referenciranja?
27. Koji pristupi oslobađanja fizičkih stranica koriste bit izmene?
28. Na šta se oslanja rukovanje virtuelnom memorijom?

13 SLOJ OPERATIVNOG SISTEMA ZA RUKOVANJE KONTROLERIMA

13.1 PODELA ULAZNIH I IZLAZNIH UREĐAJA RAČUNARA

Ulazni i izlazni uređaji računara se dele na blokovske i znakovne uređaje. Ovakva podela je uslovljena razlikama između ove dve vrste ulaznih i izlaznih uređaja u pogledu jedinice pristupa, u pogledu načina pristupa i u pogledu upravljanja. Tako je za blokovske uređaje jednica pristupa blok, a za znakovne uređaje jedinica pristupa je znak. Dalje, dok značajan broj blokovskih uređaja dozvoljava direktan pristup, znakovni uređaji podržavaju samo sekvencijalni pristup. Na kraju, za razliku od blokovskih uređaja, znakovni uređaji dozvoljavaju dinamičko podešavanje njihovih pojedinih funkcionalnih karakteristika, kao što je, na primer, brzina prenosa znakova od računara i ka računaru. Prethodne razlike utiču na oblikovanje drajvera, zaduženih za rukovanje kontrolerima.

Važno je uočiti da klasifikacija uređaja na znakovne i blokovske ne obuhvata sve uređaje. Na primer, mrežni kontroler, sat ili miš ne spadaju ni u znakovne ni u blokovske uređaje. Zato se drajveri ovakvih uređaja razlikuju od drajvera znakovnih i blokovskih uređaja.

13.2 DRAJVERI

Zajedničko svojstvo drajvera je da je svaki od njih namenjen za rukovanje određenom klasom uređaja. Pri tome, obično, jedan drajver može da opsluži više primeraka uređaja iste klase. Drajveri se nalaze u tesnoj saradnji sa kontrolerima ulaznih i izlaznih uređaja i kriju sve detalje i posebnosti funkcionisanja ovih kontrolera. Van drajvera su vidljive samo operacije, kao što su, na primer, operacije ulaza ili izlaza, koje omogućuju jednoobrazno korišćenje ulaznih i izlaznih uređaja. Tipične operacije drajvera blokovskih uređaja su:

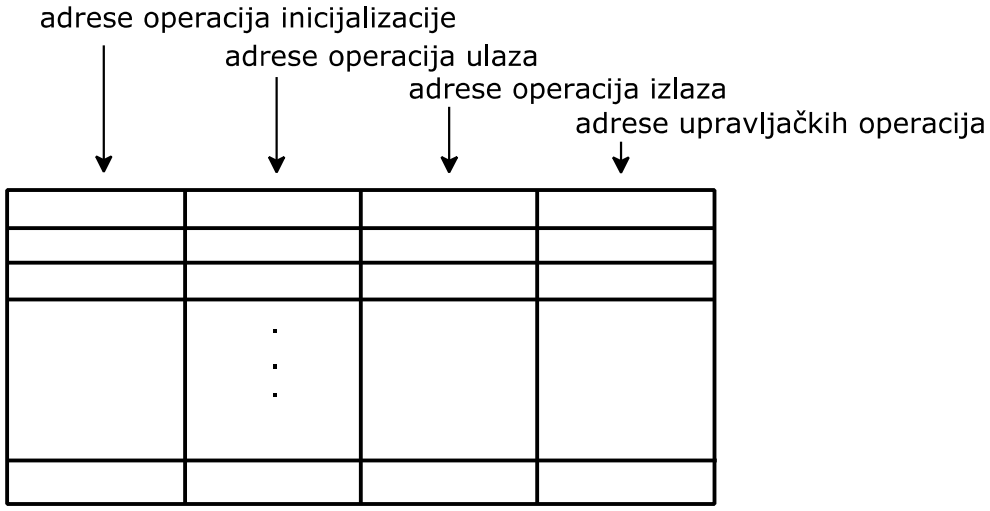
1. operacija inicijalizacije (koja se poziva u toku pokretanja operativnog sistema) i
2. operacije ulaza i izlaza blokova (koje koristi sloj za rukovanje datotekama).

Tipične operacije drajvera znakovnih uređaja su:

1. operacija inicijalizacije (koja se poziva u toku pokretanja operativnog sistema),
2. operacije ulaza i izlaza znakova (koje koristi sloj za rukovanje datotekama), kao i
3. upravljačka operacija (koja omogućuje dinamičko podešavanje funkcionalnih karakteristika znakovnih uređaja, na primer, njihove brzine prenosa znakova).

Za adresu svake od ovih operacija predviđeno je posebno polje u elementu tabele drajvera (Slika 13.1). Podrazumeva se da redni broj drajvera indeksira element ove tabele, koji sadrži polja sa adresama pojedinih operacija datog drajvera. Pri tome, polja, namenjena za adrese operacija, koje dotični drajver ne podržava, sadrže adresu posebne lažne operacije, čije obavljanje nema efekta. To važi, na primer, za upravljačku operaciju

kod drajvera diska, ili za operaciju ulaza kod drajvera štampača.



Slika 13.1: Tabela drajvera

Tabela drajvera nudi standardan način za povezivanje sloja za rukovanje datotekama i sloja za rukovanje kontrolerima, radi vezivanja operacija običnih i specijalnih datoteka za operacije drajvera ulaznih i izlaznih uređaja. Zahvaljujući ovoj tabeli, moguće je u operativni sistem dodavati (statički i dinamički) nove drajvere. Uslov za to je dopunjavanje tabele drajvera adresama operacija novog drajvera, dodavanje objektnog oblika novog drajvera objektnom obliku operativnog sistema i, u slučaju dinamičkog dodavanja, pozivanje operacije inicijalizacije radi aktiviranja drajvera.

U sklopu opsluživanja kontrolera, drajveri moraju da reaguju i na prekide, koji stižu od kontrolera. Prekidi, na primer, objavljuju da je završen prenos podataka ka kontroleru, ili od kontrolera. U ovakvom slučaju, obrada prekida obuhvata ili preuzimanje podataka, pristiglih od kontrolera, ili pripremu prenosa novih podataka ka kontroleru. Za ovakve obrade su zaduženi obrađivači prekida drajvera. Za razliku od operacija drajvera, koje se pozivaju iz slojeva iznad sloja za rukovanje kontrolerima, obrađivače prekida poziva mehanizam prekida, znači hardver ispod operativnog sistema. Zato operacije drajvera obrazuju gornji deo drajvera, a obrađivači prekida obrazuju donji deo drajvera. Uslov, da hardverski mehanizam prekida pozove nekog od obrađivača prekida, je da adresa ovog obrađivača dospe u odgovarajući element tabele prekida. To se ostvaruje u okviru drajverske operacije inicijalizacije.

13.3 DRAJVERI BLOKOVSKIH UREĐAJA

Aktivnost drajvera blokovskih uređaja započinje inicijalizacijom njihovih kontrolera. To se obavi pozivanjem drajverske operacije inicijalizacije. Nakon toga, aktivnost drajvera blokovskih uređaja se svodi na prenos blokova ka ovim uređajima i od njih. Zato su za drajverske operacije ulaza i izlaza blokova obavezni argumenti redni broj

prenošenog bloka i adresa bafera u koji, ili iz kog se prenosi blok. Pomenuti bafer pripada sloju za rukovanje datotekama. Drajversku operaciju ulaza bloka poziva sistemska operacija čitanja sloja za rukovanje datotekama. Ako se sistemska operacija čitanja odnosi na običnu datoteku, ona, kao prvi argument u pozivu drajverske operacije ulaza bloka, navodi izračunati redni broj bloka. U slučaju da se pomenuta sistemska operacija čitanja odnosi na specijalnu datoteku, ona, kao prvi argument u pozivu drajverske operacije ulaza bloka, navodi sadržaj polja pozicije u ovoj datoteci, koje se nalazi u elementu tabele otvorenih datoteka procesa pozivaoca pomenute sistemske operacije. U svakom slučaju, kao drugi argument pozivane drajverske operacije koristi se adresa nekog od slobodnih bafera sloja za rukovanje datotekama.

Drajverska operacija izlaza bloka se poziva kada se zahteva da izmenjeni sadržaj bafera sloja za rukovanje datotekama budu sačuvani na disku. Ovo se dešava i prilikom oslobađanja bafera, koji sadrži izmenjenu kopiju bloka. Međutim, poziv drajverske operacije izlaza bloka može da isprovocira i sistemska operacija pisanja sloja za rukovanje datotekama. Ako se sistemska operacija pisanja odnosi na običnu datoteku, ona, kao prvi argument u pozivu drajverske operacije izlaza bloka, navodi izračunati redni broj bloka u koji se smeštaju pisani bajti. U slučaju da se pomenuta sistemska operacija pisanja odnosi na specijalnu datoteku, ona, kao prvi argument u pozivu drajverske operacije izlaza bloka, navodi sadržaj polja pozicije u ovoj datoteci, koje se nalazi u elementu tabele otvorenih datoteka procesa pozivaoca pomenute sistemske operacije. U svakom slučaju, kao drugi argument pozivane drajverske operacije koristi se adresa nekog od bafera sloja za rukovanje datotekama, u kome je pripremljen novi sadržaj upisivanog bloka.

Za drajverske operacije ulaza ili izlaza bloka je prirodno da se oslone na mehanizam direktnog memorijskog pristupa, ako to omogućuje kontroler. U ovom slučaju, postavlja se pitanje šta učiniti sa aktivnošću procesa pozivaoca neke od ovih operacija, nakon pokretanja mehanizma direktnog memorijskog pristupa, radi prenosa bloka. Ako je za nastavak aktivnosti pomenutog procesa neophodno da prenos bloka bude završen, tada je neizbežno zustavljanje aktivnosti pomenutog procesa, nakon pokretanja mehanizma direktnog memorijskog pristupa, dok se zatraženi prenos bloka ne obavi. Drajverske operacije ulaza ili izlaza bloka, koje zaustavljaju aktivnost svog procesa pozivaoca, dok se ne obavi zatraženi prenos bloka, spadaju u klasu blokirajućih operacija. Zaustavljanje aktivnosti jednog procesa u okviru drajverske operacije ulaza ili izlaza bloka prirodno dovodi do preključivanja procesora na drugi proces. U toku aktivnosti ovog drugog procesa se može, takođe, javiti potreba za prenosom bloka. Ako se, u okviru drajverske operacije ulaza ili izlaza bloka, ustanovi da se zatraženi prenos ne može pokrenuti, jer je ulazni ili izlazni uređaj zauzet već pokrenutim prenosom bloka za potrebe prvog procesa, neizbežno je zaustavljanje aktivnosti i drugog procesa. Pri tome, mora ostati trag o zahtevu za prenosom novog bloka, da bi se taj prenos pokrenuo po završetku već pokrenutog prenosa. Sticaj okolnosti može dovesti do toga da postoji više ovakvih zahteva, jer je rad ulaznog ili izlaznog uređaja sporiji od rada procesora. Znači, moguće je da se, u toku prenosa jednog bloka, procesor više puta preključi na razne procese, čije aktivnosti se, jedna za drugom, zaustavljaju zbog zahteva za prenosom novih blokova. Uvezivanje svih istvoremeno postojećih zahteva za prenosom blokova u **listu zahteva**

omogućuje ne samo registrovanje svih zahteva, nego i registrovanje redosleda njihovog obavljanja. Pri tome, svaki zahtev u ovakvoj listi zahteva mora da sadrži:

1. smer zahtevanog prenosa bloka,
2. redni broj ovog bloka,
3. adresu bafera koji učestvuje u prenosu, kao i
4. adresu deskritpora procesa, čija aktivnost se zaustavlja do obavljanja zahtevanog prenosa bloka.

Drajverske operacije ulaza ili izlaza bloka započinju pripremanjem zahteva za prenos bloka i njegovim ubacivanjem u listu zahteva. Time se, ujedno, zaustavlja aktivnost procesa pozivaoca ovakve operacije, ako nije moguće pokrenuti zahtevani prenos bloka, jer je drugi prenos u toku. U suprotnom, pokreće se mehanizam direktnog memorijskog pristupa, radi obavljanja zahtevanog prenosa bloka, i opet se zaustavlja aktivnost procesa pozivaoca. Nastavak ove aktivnosti omogućuje odgovarajući obrađivač prekida. Njega pozove kontroler, izazivajući prekid nakon obavljanja zahtevanog prenosa bloka. Pomenuti obrađivač prekida prvo izbaci iz liste zahteva upravo opsluženi zahtev, pamteći, pri tome, adresu deskriptora procesa, čija aktivnost se može nastaviti. Pre omogućavanja nastavljanja ove aktivnosti, obrađivač prekida pokreće prenos novog bloka, ako lista zahteva nije prazna. Važno je uočiti da rukovanje listom zahteva u toku drajverskih operacija ulaza ili izlaza bloka mora biti pod onemogućenim prekidima, da bi obrađivač prekida uvek zaticao tu listu u ispravnom (konzistentnom) stanju.

Drajver blokovskog uređaja mora da poznaje karakteristike uređaja koga opslužuje. Na primer, kada opslužuje magnetni disk, drajver mora da zna koliko sektora ima u stazi, koliko staza ima u cilindru i koliko cilindara ima na disku. Na osnovu tih podatka, drajver preračunava redni broj bloka u redne brojeve cilindara, staza i sektora, da bi izazvao pozicioniranje glava diska na potrebni cilindar i usmerio prenos podataka na odgovarajuće staze i sektore. Jedan blok može da sadrži više sektora. Preračunavanje rednog broja bloka u redne brojeve cilindara, staza i sektora može biti i u nadležnosti kontrolera.

U nadležnosti drajvera blokovskog uređaja je i određivanje načina preslikavanja blokova u sektore, mada i to može obavljati kontroler. Ovo preslikavanje je bitno, jer od njega zavisi propusnost, odnosno broj blokova koji se mogu preneti u jedinici vremena do ili od uređaja, kao što je magnetni disk. Tako, na primer, zbog dužine prenosa jednog sektora između kontrolera i radne memorije, moguće je da glava diska pređe preko početka drugog sektora, koji prostorno sledi odmah iza prenošenog sektora, pre nego se pomenuti prenos završi. Tada se pristup drugom sektoru mora odložiti za jedan obrtaj, dok njegov početak ponovo ne dođe ispod glave diska. To znači da je moguć pristup samo jednom bloku u jednom obrtaju, ako se pretpostavi da sektor odgovara bloku i ako se pristupa uzastopnim blokovima, koji su preslikani u prostorno uzastopne sektore. Zato je bolje da uzastopni blokovi ne budu preslikani u prostorno uzastopne sektore (*interleaving*). Ako se uzastopni blokovi preslikavaju u sektore, međusobno razdvojene jednim sektorom (*interleaving factor* 1), i ako se prenos jednog sektora završi pre nego kraj njegovog prostornog sledbenika prođe ispod glave diska, tada se u toku jednog obrtaja može pročitati n/2 uzastopnih blokova, uz pretpostavku da staza sadrži n sektora i da sektor odgovara bloku. Broj sektora (*interleaving factor*), koji razdvajaju sektore,

dodeljene uzastopnim blokovima, zavisi od odnosa vremena prenosa bloka (između kontrolera i radne memorije) i vremena za koje sektor prođe ispod glave diska. Ako kontroler automatski prebacuje sve sektore staze, iznad koje se kreće glava diska, u svoju lokalnu radnu memoriju (*track_at_time caching*), tada nema smetnje da se uzastopni blokovi preslikaju u prostorno uzastopne sektore.

Propusnost magnetnog diska zavisi i od redosleda usluživanja zahteva za prenosom blokova, jer, sem vremena prenosa bloka (*transfer time*), vremena za koje staza prođe ispod glave diska (*rotational delay*), na propusnost diska značajno utiče i vreme premeštanja glave diska sa staze na stazu (*seek time*). Da bi se ovo vreme minimiziralo, potrebno je što manje pokretati glavu diska. To znači, da je bolje da se zahtevi za prenosom blokova ne uslužuju hronološki, nego u redosledu, koji obezbeđuje minimalno pokretanje glave diska. Zato se lista ovakih zahteva sortira u rastućem redosledu staza, na kojima se nalaze blokovi, čije prenošenje se zahteva, a glava diska se pomera iz početne pozicije samo u jednom smeru, dok svi zahtevi u smeru njenog kretanja ne budu usluženi. Posle toga, ona menja smer pomeranja, radi usluživanja zahteva, koji su pristigli nakon što je glava diska prešla preko staza na kojima se nalaze blokovi, čije prenošenje se zahteva. Na ovaj način se, pored optimizacije kretanja glave diska, obezbeđuje i pravedno usluživanje svih zahteva, jer nema mogućnosti za nepredvidivo dugo odlaganje usluživanja pojedinih zahteva. Prethodno opisani pristup se naziva **elevator algoritam**, jer se po njemu upravlja kretanjem liftovima u visokim zgradama. O optimizaciji kretanja glave diska može da se brine i kontroler. U ovom slučaju, drajver samo ubacuje zahteve u listu zahteva, a kontroler se brine o redosledu njihovog usluživanja. To je naročito važno, kada se kontroler brine o zameni loših sektora ispravnim rezervnim sektorima, koji se nalaze na posebnim rezervnim stazama. Pošto, u ovom slučaju, jedino kontroler zna kojoj stazi stvarno pripada koji sektor, jedino on može da optimizira kretanje glave diska.

Zadatak drajvera je da iskoristi sve mogućnosti kontrolera. Tako, ako kontroler podržava više magnetnih diskova i omogućuje istovremena nezavisna pozicioniranja glava raznih diskova, tada to drajver može da iskoristi za smanjenje srednjeg vremena premeštanja glave diska sa staze na stazu.

Drajver može da poveća pouzdanost diska, ako reaguje na prolazne greške u radu diska. Na primer, u slučaju pojave zrnca prašine između glave magnetnog diska i magnetne površine, ulaz ili izlaz neće biti uspešan. Međutim, višestrukim ponavljanjem operacije, drajver može da otkloni prethodnu grešku, jer je verovatno da će se zrnce prašine pomeriti u narednim pokušajima ulaza ili izlaza. Na sličan način drajver može da reaguje i na pozicioniranje glave diska na pogrešnu stazu, kao i na neke druge greške kontrolera. Drajver može i da smanji trošenje magnetnih disketa, kod kojih glava disketne jedinice klizi po površini sa magnetnim premazom, zaustavljanjem obrtanja diskete, čim nestanu zathevi za prenos blokova na nju i sa nje.

13.4 BLOKOVSKI I ZNAKOVNI UREĐAJI KAO SPECIJALNE DATOTEKE

Za blokovske uređaje, poput magnetnog diska, je tipično da ih koriste istovremeno postojeći procesi u toku pristupa (raznim) datotekama. Zato je i moguće da se u listi

zahteva istovremeno zateknu zahtevi raznih procesa. Blokovskim uređajima se retko pristupa kao specijalnim datotekama, a kada se to i desi, koristi ih samo jedan proces, čiji je zadatak najčešće formatiranje uređaja ili provera ispravnosti blokova, radi pronalaženja izgubljenih ili loših blokova. Za razliku od blokovskih uređaja, za znakovne uređaje je tipično da im procesi pristupaju kao specijalnim datotekama i da ih zaključavaju, da bi obezbedili međusobnu isključivost u toku njihovog korišćenja. Takav način upotrebe znakovnih uređaja je uobičajen ne samo za terminale, za koje je prirodno da ih opslužuje samo jedan proces, nego i za štampače, za koje je prirodno da ih koristi više procesa. Zato se, u slučaju znakovnih uređaja, kao što su štampači, uvode posebni **sistemski procesi posrednici** (*spooler*), koji posreduju u korišćenju pomenutih uređaja. Svaki od ovih procesa pristupa svom znakovnom uređaju kao specijalnoj datoteci, koju zaključava, da bi obezbedio međusobnu isključivost u toku njenog korišćenja. Istovremeno, uz svaki od sistemskih procesa posrednika postoji i poseban imenik. Kada neki korisnički proces želi da odštampa tekst, on prvo pripremi datoteku sa odgovarajućim sadržajem, a zatim tu datoteku ubaci u pomenuti imenik. Odgovarajući sistemski proces posrednik vadi datoteke iz svog imenika (jednu po jednu), da bi njihove sadržaje (jedan po jedan) uputio, posredstvom svoje specijalne datoteke, na željeni uređaj. Pri tome se korisnički procesi (s jedne strane) i sistemski proces posrednik (s druge strane) nalaze u odnosu proizvođač i potrošač, jer prvopomenuti procesi "proizvode" datoteke, koje "troši" drugopomenuti proces. Da bi ovakva saradnja procesa bila uspešna, neophodno je da sloj za rukovanje datotekama obezbedi sinhronizaciju procesa tokom njihovog pristupa imenicima, koji posreduju u razmeni datoteka.

13.5 DRAJVERI ZNAKOVNIH UREĐAJA

Blokovski i znakovni uređaji se razlikuju ne samo po načinu korišćenja, nego i po načinu aktiviranja. Tako, dok blokovske uređaje uvek aktiviraju procesi, aktivnost znakovnih uređaja zavisi i od aktivnosti korisnika. Na primer, prispeće znakova sa tastature ne zavisi od aktivnosti procesa, nego od aktivnosti korisnika. Slično, upućivanje znakova na ekran zavisi i od aktivnosti korisnika (eho), ali i od aktivnosti procesa. Zato u sastav drajvera znakovnih uređaja obavezno ulaze i baferi. Oni su namenjeni za smeštanje znakova, koji su, nezavisno od aktivnosti procesa, prispeli sa ovih uređaja, odnosno, koji su upućeni ka ovim uređajima. U ovakvim baferima znakovi se čuvaju dok ih procesi ili uređaji ne preuzmu. Tako, na primer, za drajver terminala je potreban par takvih bafera za svaki od terminala koje drajver opslužuje. Pri tome, jedan, **ulazni bafer** služi za smeštanje znakova, prispelih sa tastature, a drugi, **eho bafer** služi za smeštanje znakova, upućenih ka ekranu. Svaki pritisak dirke sa tastature izaziva prekid, koji aktivira obrađivača prekida tastature. Ako je ulazni bafer pun, obrađivač prekida tastature ignoriše prispeli znak. Inače, on ga preuzima i smešta u ulazni bafer. U ulaznom baferu znak čeka da neki proces zatraži njegovo preuzimanje. Preuzimanje znaka iz ulaznog bafera omogućuje drajverska operacija ulaza znaka. U okviru ove operacije se zaustavlja aktivnost procesa njenog pozivaoca, ako je ulazni bafer prazan. Tada nastavljanje ove aktivnosti omogućuje obrađivač prekida tastature, po smeštanju znaka u ulazni bafer. U svakom slučaju, drajverska operacija ulaza znaka vraća preuzeti znak iz ulaznog bafera kao svoju vrednost.

Obrađivač prekida tastature ima, takođe zadatak da proveri da li je eho bafer prazan. Ako ovaj bafer nije prazan, tada obrađivač prekida tastature smešta prispeli znak u eho bafer. Inače, obrađivač prekida tastature upućuje prispeli znak ka ekranu. Nakon prikazivanja znaka, ekran izaziva prekid, koji aktivira obrađivača prekida ekrana. Ako je eho bafer prazan, aktivost obrađivača prekida ekrana se odmah završava. Inače, on preuzima naredni znak iz eho bafera i upućuje ga ka ekranu.

Eho bafer je podesan i za čuvanje znakova, koje procesi žele da prikažu na ekranu. Prikazivanje znaka omogućuje drajverska operacija izlaza znaka. Jedini argument njenog poziva je prikazivani znak. U okviru ove operacije se zaustavlja aktivnost procesa njenog pozivaoca, ako je eho bafer pun. Tada nastavljanje ove aktivnosti omogućuje obrađivač prekida ekrana i to nakon pražnjenja jednog znaka (ili više znakova) iz ovog bafera. Ako drajverska operacija izlaza znaka zatekne eho bafer prazan, ona upućuje prikazivani znak ka ekranu. U slučaju da eho bafer nije ni pun ni prazan, drajverska operacija izlaza znaka samo smešta prikazivani znak u eho bafer.

Prethodno opisane drajverske operacije ulaza i izlaza znaka spadaju u blokirajuće operacije. Ove operacije se pozivaju iz sistemskih operacija čitanja i pisanja sloja za rukovanje datotekama, kada se čita, odnosno piše specijalna datoteka. Poziv drajverske operacije ulaza nema argumenata, a njegova povratna vrednost je pročitani znak (njegov kod). Za poziv drajverske operacije izlaza kao jedini argument služi pisani znak (njegov kod). Ovaj poziv nema povratnu vrednost.

Rukovanje ulaznim i eho baferom, u okviru drajverskih operacija ulaza i izlaza znaka, mora biti pod onemogućenim prekidima, da bi obrađivači prekida tastature i ekrana uvek zaticali bafere u ispravnom (konzistentnom) stanju. Inače, obrađivač prekida tastature i proces pozivalac drajverske operacije ulaza znaka se nalaze u odnosu proizvođač i potrošač. U istom odnosu se nalaze proces pozivalac operacije izlaza znaka i obrađivač prekida ekrana.

Za razliku od prethodno opisanih znakovnih terminala, za grafičke (memorijski preslikane) terminale nije potreban eho bafer, niti obrađivač prekida ekrana, jer ovakvi terminali poseduju video memoriju čiji sadržaj se periodično prikazuje prilikom osvežavanja ekrana. Zato je, kod grafičkog terminala, za prikazivanje znaka na ekranu dovoljno smestiti znak u odgovarajuću lokaciju video memorije. U slučaju da se želi podržati više prozora (*window*) na ekranu grafičkog terminala, za svaki od prozora je potreban poseban ulazni bafer. Znak prispeo sa tastature se smešta u ulazni bafer aktivnog prozora, a prikazivani znak se upućuje u deo video memorije prozora na kome znak treba da se pojavi. O aktivnom prozoru se brine rukovalac prozorima (*window manager*).

Drajver terminala, pored operacije inicijalizacije, namenjene za inicijalizaciju kontrolera terminala, i operacija ulaza i izlaza znakova, nudi i upravljačku operaciju. Argumenti njenog poziva utiču ne samo na funkcionisanje, na primer, terminala, nego i na funkcionisanje drajvera terminala. Upravljačka operacija omogućuje da se drajveru terminala saopšti da interpretira znakove koji dolaze sa tastature (*cooked mode*), ili da ih ne interpretira (*raw mode*). U prvom slučaju, u nadležnosti drajvera terminala se nalazi editiranje znakova prispelih sa tastature. U sklopu toga, drajver terminala mora, na

primer, da omogući brisanje poslednje prispelog znaka. Znači, kada primi odgovarajući upravljački znak (*delete*), drajver terminala, odnosno, njegov obrađivač prekida tastature, mora da prethodno prispeli znak izbaci iz ulaznog i iz eho bafera, kao i da obezbedi brisanje tog znaka sa ekrana, ako je on već prikazan. Takođe, drajver terminala se brine o interpretaciji upravljačkih znakova "prelazak na novu liniju" (*line feed*), "prelazak na početak linije" (*carriage return*), kao i drugih upravljačkih znakova (*escape*, *return* ili *enter* i slično). U slučaju kada ne interpretira znakove, drajver terminala samo prosleđuje znakove koji su pristigli u njegov ulazni bafer.

Eho znaka, pristiglog sa tastature, nije uvek poželjan. To je slučaj, na primer, kod zadavanja znakova lozinke. Zato upravljačka operacija omogućuje da se drajveru terminala saopšti kada da vrši, a kada da ne vrši eho pristiglih znakova. Drajver terminala mora da razdvoji eho znakova od prikazivanja znakova koje na ekran šalju procesi.

Drajver terminala može da interpretira znakove, koji su mu prosleđeni, radi pomeranja kursora, pomeranja linija, kao i drugih rukovanja ekranom, poput rukovanja prozorima.

13.6 DRAJVER SATA

U nadležnosti sloja za rukovanje kontrolerima se nalazi i praćenje proticanja vremena. Praćenje proticanja vremena se zasniva na brojanju periodičnih prekida, koje u pravilnim vremenskim intervalima generiše sat. Obrađivač prekida sata broji prekide sata, a njihov zbir predstavlja sistemsko vreme (lokalno vreme u računaru). Ovaj obrađivač prekida predstavlja donji deo drajvera sata. Gornji deo ovog drajvera predstavljaju sistemske operacije za preuzimanje ili izmenu sistemskog vremena i za uspavljivanje procesa, odnosno, za odlaganje njegove aktivnosti, dok ne istekne zadani vremenski interval.

Sistemsko vreme se može predstaviti (1) kao broj prekida sata ili (2) kao broj sekundi i broj prekida sata u tekućoj sekundi. Druga predstava zahteva manje prostora.

Kvantum se predstavlja kao celi broj prekida sata.

Važno je uočiti da, dok je sistemsko vreme precizno, jer je sat precizan, dotle pripisivanje procesorskog vremena procesima, odnosno, merenje trajanja aktivnosti procesa, ne mora biti precizno. Kada postoji ovakva nepreciznost, nju izaziva nemogućnost merenja dužine vremenskih intervala, koji su kraći od perioda prekida sata, a kojih ima u toku aktivnosti procesa. Na primer, trajanje obrade prekida je obično kraće od perioda prekida sata i pripisuje se prekinutom procesu, iako pomenuta obrada prekida ne mora biti sa njim povezana. Obrada prekida obično predstavlja deo aktivnosti prekinutog procesa, jer se u obradi prekida, zbog brzine, izbegava preključivanje sa prekinutog procesa, odnosno preključivanje na prekinuti proces. Slično, ako se u toku jednog perioda prekida sata desi više preključivanja, celi period se pripisuje kvantumu poslednjeg aktivnog procesa, koga je prekinuo prekid sata. Pomenute nepreciznosti se mogu izbeći, ako procesor broji svoje cikluse i njihovu sumu čuva u posebnom registru. Ako se sadržaj ovog registra preuzme na početku i na kraju perioda aktivnosti procesora koji je kraći od perioda sata, iz razlike ovih sadržaja se može ustanoviti koliko ciklusa je

potrošeno u ovom periodu i iz toga odrediti trajanje dotičnog perioda.

U nadležnosti obrađivača prekida sata nalazi se više poslova, kao što su:

1. održavanje sistemskog vremena,
2. praćenje isticanja kvantuma aktivnog procesa,
3. praćenje ukupnog korišćenja procesorskog vremena aktivnog procesa,
4. provera da li je nastupilo vreme buđenja uspavanog procesa (čija aktivnost se nastavlja tek kada istekne zadani vremenski interval), ili
5. skupljanje statistika o aktivnosti procesa (koje se svodi na registrovanje sadržaja programskog brojača, radi otkrivanja učestanosti izvršavanja pojedinih delova programa).

13.7 RUKOVANJE TABELOM PREKIDA

Sloj za rukovanje kontrolerima omogućuje i smeštanje adresa obrađivača prekida u tabelu prekida, čime dozvoljava da viši slojevi operativnog sistema mogu da reaguju na prekide. Za operaciju smeštanja adresa obrađivača prekida u tabelu prekida nije uputno da bude sistemska operacija, jer ona pruža mogućnost da se ugrozi funkcionisanje operativnog sistema i naruši njegov mehanizam zaštite.

13.8 OSNOVA SLOJA ZA RUKOVANJE KONTROLERIMA

Sloj za rukovanje kontrolerima se oslanja na sloj za rukovanje procesorom, jer su preključivanja sastavni deo aktivnosti drajvera. Za operacije sloja za rukovanje kontrolerima, odnosno za drajverske operacije, je zajedničko da se obavljaju pod onemogućenim prekidima, što je prihvatljivo, jer je reč o kratkotrajnim operacijama.

13.9 PITANJA

1. Šta karakteriše ulazne i izlazne uređaje?
2. Koja svojstva imaju drajveri?
3. Šta karakteriše tabelu drajvera?
4. Šta podrazumeva podela drajvera na gornji i na donji deo?
5. Kada se pozivaju operacije drajvera blokovskih uređaja?
6. Šta sadrži lista zahteva?
7. Šta spada u nadležnosti drajvera blokovskih uređaja, ali i kontrolera?
8. Kada se uzastopni blokovi preslikavaju u prostorno uzastopne sektore?
9. Na koji drajver se odnosi elevator algoritam?
10. Koju ulogu imaju sistemski procesi posrednici?
11. Kada se specijalna datoteka tipično zaključava?
12. Šta sadrži drajver terminala?
13. U kom slučaju nisu potrebni eho bafer i obrađivač prekida ekrana?
14. Šta omogućuje upravljačka operacija drajvera terminala?
15. Koje operacije sadrži gornji deo drajvera sata?
16. Šta ne može da meri drajver sata?
17. Šta omogućuje obrađivač prekida iz donjeg dela drajvera sata?

14 SLOJ OPERATIVNOG SISTEMA ZA RUKOVANJE PROCESOROM

14.1 RASPOREĐIVANJE

Osnovni zadatak rukovanja procesorom je preključivanje procesora sa aktivnog procesa na neki od spremnih procesa. O izboru spremnog procesa, na koga se preključuje procesor, brine **raspoređivanje** (*scheduling*). Ovaj izbor zavisi od cilja raspoređivanja. Tipični ciljevi raspoređivanja su, na primer, (1) poboljšanje iskorišćenja procesorskog vremena, (2) ravnomerna raspodela procesorskog vremena, (3) što kraći odziv na korisničku akciju ili neki drugi oblik postizanja potrebnog kvaliteta usluge (*Quality of Service*, *QoS*), kao što je rezervisanje procesorskog vremena radi obezbeđenja kvalitetne reprodukcije zvuka ili videa kod multimedijalnih aplikacija. Ovakvi ciljevi nisu saglasni, pa se ne mogu istovremeno ostvariti.

Za neinteraktivno korišćenje računara cilj raspoređivanja je poboljšanje iskorišćenja procesorskog vremena. Ovakav cilj se ostvaruje minimiziranjem preključivanja na neophodan broj (samo nakon pozivanja blokirajućih sistemskih operacija ili nakon kraja aktivnosti procesa).

Za interaktivno korišćenje računara (u višekorisničkom režimu rada) ciljevi raspoređivanja su ravnomerna raspodela procesorskog vremena između istovremeno postojećih procesa, odnosno, između njihovih vlasnika (korisnika, koji istovremeno koriste računar) i što kraći odziv na korisničku akciju. Ovakvi ciljevi se ostvaruju **kružnim raspoređivanjem** (*round robin scheduling*), koje svakom od istovremeno postojećih procesa dodeljuje isti vremenski interval, nazvan kvantum. Po isticanju kvantuma, aktivni proces prepušta procesor spremnom procesu, koji najduže čeka na svoj kvantum. Neophodan preduslov za primenu kružnog raspoređivanja je da se preključivanje vezuje za trenutak u kome se završava tekući kvantum. Zato je neophodno da se preključivanje poziva neposredno nakon obrade prekida sata (pre nastavka prekinutog procesa).

Kružno raspoređivanje se uspešno primenjuje i u situaciji kada hitnost svih procesa nije ista, pa se, zbog toga, procesima dodeljuju razni prioriteti. Pri tome se podrazumeva da kružno raspoređivanje važi u okviru grupe procesa sa istim prioritetom. Procesor se preključuje na procese sa nižim prioritetom samo kada se završi (zaustavi) aktivnost i poslednjeg od procesa sa višim prioritetom. Procesor se preključuje na proces sa višim prioritetom odmah po pojavi ovakvog procesa (*preemptive scheduling*), odnosno odmah po omogućavanju nastavka aktivnosti prioritetnijeg procesa.

Dinamička izmena prioriteta procesa doprinosi ravnomernosti raspodele procesorskog vremena između procesa, ako se uspostavi obrnuta proporcionalnost između prioriteta procesa i obima u kome je on iskoristio poslednji kvantum. Pri tome se periodično proverava iskorišćenje poslednjeg kvantuma svakog od procesa i, u skladu s tim, procesima se dodeljuju novi prioriteti. Takođe, dinamička izmena prioriteta procesa doprinosi ravnomernosti raspodele procesorskog vremena između korisnika,

ako se uspostavi obrnuta proporcionalnost između prioriteta procesa, koji pripadaju nekom korisniku, i ukupnog udela u procesorskom vremenu tog korisnika u toku njegove interakcije sa računarom. Znači, što je ukupan udeo korisnika više ispod željenog proseka, to prioritet njegovih procesa više raste.

Ravnomerna raspodela procesorskog vremena se može postići i bez izmena prioriteta, ako se uvede **lutrijsko raspoređivanje** (*lottery scheduling*). Ono se zasniva na dodeli procesima lutrijskih lozova. Nakon svakog kvantuma na slučajan način se izvlači broj loza, a procesor se preključuje na proces koji poseduje izvučeni loz. Tako, ako ukupno ima m lozova, proces, koji poseduje n od m lozova (n < m), u proseku koristi n/m kvantuma procesorskog vremena.

Za multimedijalne aplikacije, koje zahtevaju visoku propusnost podataka i njihovu isporuku sa pravilnim periodom, cilj raspoređivanja je garantovanje procesima potrebnog broja kvantuma u pravilnim vremenskim razmacima.

Ostvarenje raznih ciljeva raspoređivanja se može zasnovati na istim mehanizmima raspoređivanja. U tom slučaju razni načini primene tih mehanizama ili razne politike raspoređivanja dovode do ostvarenja raznih ciljeva raspoređivanja. Razdvajanje mehanizama raspoređivanja od politike raspoređivanja je važno zbog fleksibilnosti. Mehanizmi raspoređivanja omogućuju uticanje na dužinu kvantuma i na nivo prioriteta, a politika raspoređivanja određuje dužinu kvantuma i nivo prioriteta.

Uticanje na dužinu kvantuma je važno, jer od dužine kvantuma zavisi iskorišćenje procesora, ali i odziv računara, odnosno brzina kojom on reaguje na korisničku akciju sa terminala. Pri tome, skraćenje (do određene granice) kvantuma doprinosi poboljšanju odziva, ali i smanjenju iskorišćenja procesora, jer povećava broj preključivanja koja troše procesorsko vreme. Suviše kratak kvantum počinje da ugrožava i odziv, kada se prevelik procenat procesorskog vremena počne da troši na preključivanje. Sa stanovišta iskorišćenja procesora prihvatljiva su samo neophodna preključivanja (kada nije moguć nastavak aktivnosti procesa), odnosno, značajno smanjivanje učestanosti preključivanja. S tom idejom na umu moguće je iskoristiti dinamičku izmenu prioriteta procesa za (1) održavanje dobrog odziva za procese, koji su u interakciji sa korisnicima, kao i za (2) održavanje dobrog iskorišćenja procesora za pozadinske (*background*) procese, koji nisu u (čestoj) interakciji sa korisnicima. Pri tome se interaktivnim procesima dodeljuje najviši prioritet i najkraći kvantum. Pozadinskim procesima, koji su vrlo dugo aktivni bez ikakve interakcije sa korisnikom, se dodeljuje najniži prioritet i najduži kvantum. Procesu automatski opada prioritet i produžava se kvantum što je on duže aktivan i ima manju interakciju sa korisnikom. Povećanje interakcije sa korisnikom dovodi do porasta prioriteta procesa i smanjenja njegovog kvantuma. Dinamička izmena prioriteta se obavlja periodično i nalazi se u nadležnosti politike raspoređivanja, koja je zadužena i za vezivanje odgovarajućih dužina kvantuma za odgovarajuće prioritete.

Za operacije sloja za rukovanje procesorom je zajedničko da se obavljaju pod onemogućenim prekidima, što je prihvatljivo, jer je reč o kratkotrajnim operacijama. To je naročito značajno za operacije koje rukuju deskriptorima procesa, jer jedino onemogućenje prekida osigurava ispravnost rukovanja listama u koje se uključuju i iz kojih se isključuju deskriptori procesa u toku ovih operacija (odnosno, osigurava

konzistentnost ovih listi). Pod onemogućenim prekidima se obavljaju i operacija preključivanja (sa operacijom raspoređivanja), sistemska operacija za izmenu prioriteta procesa, kao i sistemske operacije za sinhronizaciju procesa.

Operacija raspoređivanja obuhvata bar dve radnje. Jedna ubacuje proces među spremne procese, tako što njegov deskriptor uvezuje na kraj liste deskriptora spremnih procesa, koja odgovara prioritetu dotičnog procesa. U ovom slučaju se podrazumeva da za svaki prioritet postoji posebna lista deskriptora spremnih procesa, na koju se primenjuje kružno raspoređivanje. Druga od ove dve radnje izvezuje iz listi deskriptora spremnih procesa deskriptor najprioritetnijeg spremnog procesa.

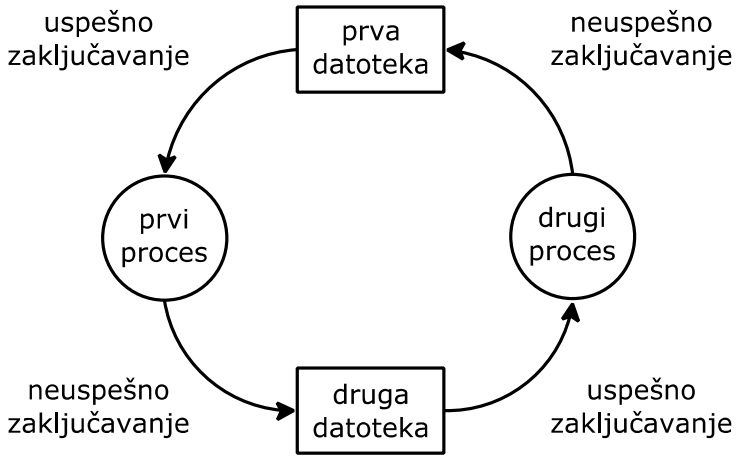
14.2 PITANJA

1. Šta karakteriše tipične ciljeve raspoređivanja?
2. Šta je cilj raspoređivanja za neinteraktivno korišćenje računara?
3. Šta je cilj raspoređivanja za interaktivno korišćenje računara?
4. Zašto je uvedeno kružno raspoređivanje?
5. Šta doprinosi ravnomernoj raspodeli procesorskog vremena?
6. Šta je cilj raspoređivanja za multimedijalne aplikacije?
7. Do čega dovodi skraćenje kvantuma?
8. Šta se postiže uticanjem na nivo prioriteta i na dužinu kvantuma?

15 MRTVA PETLJA

15.1 USLOVI ZA POJAVU MRTVE PETLJE

Mrtva petlja je problematična pojava trajnog zaustavljanja aktivnosti međusobno zavisnih procesa. Na primer, ona se javi kada dva procesa žele da u režimu međusobne isključivosti pristupaju dvema datotekama. Ako prvi od njih zaključa prvu datoteku, a zatim drugi od njih zaključa drugu datoteku, tada nema mogućnosti za nastavak aktivnosti tih procesa, bez obzira da li je sistemska operacija zaključavanja blokirajuća ili ne. U slučaju blokirajućih sistemskih operacija zaključavanja, pokušaj prvog procesa da zaključa drugu datoteku dovodi do trajnog zaustavljanja njegove aktivnosti. Isto se dešava sa drugim procesom prilikom njegovog pokušaja da zaključa prvu datoteku. Zaustavljanje aktivnosti ova dva procesa je trajno, jer je svaki od njih zauzeo datoteku koja treba onom drugom procesu za nastavak njegove aktivnosti i nema nameru da tu datoteku oslobodi (Slika 15.1).



Slika 15.1: Primer mrtve petlje

U slučaju neblokirajuće sistemske operacije zaključavanja, procesi upadaju u beskonačnu petlju (*starvation*), pokušavajući da zaključaju datoteku, koju je zaključao drugi proces. Ovakav oblik međusobne zavisnosti procesa se naziva i **živa petlja** (*livelock*). Ona se, po svom ishodu, suštinski ne razlikuje od mrtve petlje.

Pojava mrtve petlje je vezana za zauzimanje resursa, kao što su, na primer, prethodno pomenute datoteke. Pri tome, za pojavu mrtve petlje je potrebno da budu ispunjena četiri uslova:

1. zauzimani resursi se koriste u režimu međusobne isključivosti,
2. resursi se zauzimaju jedan za drugim, tako da proces, nakon zauzimanja izvesnog broja resursa, mora da čeka da zauzme preostale resurse,
3. resurse oslobađaju samo procesi koji su ih zauzeli i

4. postoji cirkularna međuzavisnost procesa (prvi proces čeka oslobađanje resursa koga drži drugi proces, a on čeka oslobađanje resursa koga drži treći proces, i tako redom do poslednjeg procesa iz lanca procesa, koji čeka oslobađanje resursa koga drži prvi proces).

15.2 TRETIRANJE MRTVE PETLJE

Postoje četiri pristupa tretiranja problema mrtve petlje:

1. sprečavanje (*prevention*) pojave mrtve petlje (onemogućavanjem važenja nekog od četiri neophodna uslova za njenu pojavu),
2. izbegavanje (*avoidance*) pojave mrtve petlje,
3. otkrivanje (*detection*) pojave mrtve petlje i oporavak (*recovery*) od nje i
4. ignorisanje pojave mrtve petlje.

Kod sprečavanja pojave mrtve petlje, važenje prvog uslova obično nije moguće sprečiti, jer se resursi najčešće koriste u režimu međusobne isključivosti. Važenje drugog uslova se može sprečiti, ako se unapred zna koliko treba resursa i ako se oni svi zauzmu pre korišćenja. Pri tome, neuspeh u zauzimanju bilo kog resursa dovodi do oslobađanja prethodno zauzetih resursa, što je moguće učiniti bez posledica, jer nije započelo njihovo korišćenje. Važenje trećeg uslova se obično ne može sprečiti, jer najčešće ne postoji način da se zauzeti resurs privremeno oduzme procesu. I konačno, važenje četvrtog uslova se može sprečiti, ako se resursi uvek zauzimaju u unapred određenom redosledu, koji isključuje mogućnost cirkularne međuzavisnosti procesa.

Izbegavanje pojave mrtve petlje zahteva poznavanje podataka (1) o maksimalno mogućim zahtevima za resursima, (2) o ukupno postavljenim zahtevima za resursima i (3) o stanju resursa. Podrazumeva se da se udovoljava samo onim zahtevima za koje se proverom ustanovi da, nakon njihovog ispunjavanja, postoji redosled zauzimanja i oslobađanja resursa u kome se mogu zadovoljiti maksimalno mogući zahtevi svih procesa. Praktična vrednost ovoga pristupa nije velika, jer se obično unapred ne znaju maksimalno mogući zathevi procesa za resursima, a to je neophodno za proveru da li se može udovoljiti pojedinim zahtevima. Sem toga, ovakva provera je komplikovana, a to znači i neefikasna.

Otkrivanje pojave mrtve petlje se zasniva na sličnom pristupu kao i izbegavanje pojave mrtve petlje. U ovom slučaju se proverava da li postoji proces, čijim zahtevima se ne može udovoljiti ni za jedan redosled zauzimanja i oslobađanja resursa. Pored komplikovanosti ovakve provere, problem je i šta učiniti, kada se i otkrije pojava mrtve petlje. Ako se resursi ne mogu privremeno oduzeti od procesa, preostaje jedino uništavanje procesa, radi oslobađanja resursa koje oni drže. Međutim, to nije uvek prihvatljiv zahvat. Zbog toga ni ovaj pristup nema veliki praktični značaj.

Ignorisanje pojave mrtve petlje je pristup koji je najčešće primenjen u praksi, jer se ovaj problem ne javlja tako često da bi se isplatilo da ga rešava operativni sistem. Prema tome, kada se mrtva petlja javi, na korisniku je da se suoči sa ovim problemom i da ga reši na način, koji je primeren datim okolnostima.

15.3 PITANJA

1. Šta je mrtva petlja?
2. Po čemu se živa petlja razlikuje od mrtve petlje?
3. Koji uslovi su potrebni za pojavu mrtve petlje?
4. Kako se u praksi tretira problem mrtve petlje?
5. Na čemu se temelji sprečavanje mrtve petlje?
6. Šta karakteriše izbegavanje mrtve petlje?
7. Šta karakteriše otkrivanje i oporavak od mrtve petlje?
8. Šta karakteriše ignorisanje mrtve petlje?

16 KOMUNIKACIJA SA OPERATIVNIM SISTEMOM

16.1 PROGRAMSKI NIVO KOMUNIKACIJE SA OPERATIVNIM SISTEMOM

Komunikaciju sa operativnim sistemom na programskom nivou omogućuju sistemske operacije. Da bi se u toku izvršavanja korisničkog programa dobila neka usluga od operativnog sistema, potrebno je pozvati odgovarajuću sistemsku operaciju. Na primer, da bi se preuzeo znak sa tastature, u korisničkom programu je neophodno navesti poziv odgovarajuće ulazne sistemske operacije.

16.2 INTERAKTIVNI NIVO KOMUNIKACIJE SA OPERATIVNIM SISTEMOM

Interaktivni nivo korišćenja operativnog sistema se ostvaruje pomoću komandi komandnog jezika. One, na primer, omogućuju rukovanje datotekama i procesima. Najjednostavniju komandu komandnog jezika predstavlja putanja izvršne datoteke. Kao operand ovakve komande se može, opet, javiti putanja datoteke, ako je komanda namenjena za rukovanje datotekama. Prema tome, na prethodni način oblikovana komanda započinje operatorom u obliku putanje izvršne datoteke, koja opisuje rukovanje, a završava operandom (ili operandima) u obliku putanja datoteka, kojima se rukuje. Tako:

```
kopiraj godina1.txt godina2.txt
```

predstavlja primer prethodno opisane komande znakovnog komandnog jezika. U ovom primeru se pretpostavlja da radni imenik obuhvata izvršnu datoteku sa imenom **kopiraj.bin** i tekst datoteku **godina1.txt**. Takođe se pretpostavlja da izvršavanje programa iz izvršne datoteke **kopiraj.bin** dovodi do stvaranja datoteke **godina2.txt**, koja je po sadržaju identična datoteci **godina1.txt** i koja pripada radnom imeniku. Prethodna komanda opisuje korisnu operaciju, ako svi studenti prve godine studija upisuju drugu godinu studija.

Prethodno opisani način zadavanja komandi odgovara **znakovnom komandnom jeziku**. Komandni jezik može olakšati zadavanje komandi, ako omogući korisniku da operator komande bira u spisku operatora (*menu*), umesto da ga pamti i u celosti navodi. Spisak operatora se prikazuje na ekranu, a izbor operatora se vrši pomoću namenskih dirki tastature ili miša. Nakon izbora operatora sledi, po potrebi, dijalog u kome korisnik navodi (ili opet bira) operand (operande) komande. Ovakvi komandni jezici se nazivaju **grafički komandni jezici** (*menu driven user interface, graphical user interface*). Oni još više pojednostavljuju zadavanje komandi, ako korisniku omogućuju da ne bira operator, nego samo operande komandi. U ovom slučaju, izbor operanda se svodi na izbor nekog od imena datoteka, prikazanih na ekranu, a operator se podrazumeva ili na osnovu tipa odabrane datoteke, ili, eventualno, na osnovu upotrebljene namenske dirke.

U svakom slučaju, zadatak komandnog jezika je da omogući korisniku da zada komandu, koja precizno određuje i vrstu rukovanja i objekat rukovanja, a zadatak

interpretiranja komande je da pokrene proces, u okviru čije aktivnosti usledi rukovanje, zatraženo komandom.

16.2.1 Znakovni komandni jezici

Izgled, način rada i mogućnosti interpretera znakovnog komandnog jezika (*command language interpreter, shell*) zavise od ciljeva, koje komandni jezik treba da ostvari. Ciljevi znakovnih komandnih jezika obuhvataju:

1. omogućavanje izvršavanja pojedinih (korisničkih) programa,
2. omogućavanje kombinovanja izvršavanja više (korisničkih) programa i
3. omogućavanje pravljenja **komandnih datoteka** (*command file, shell script*).

Interpreter znakovnog komandnog jezika ostvaruje prethodne ciljeve tako što sa standardnog ulaza prima niz znakova, koji obrazuju komandu, prepoznaje u tom nizu znakova operator komande (i, eventualno, njene operande) i preduzima zahtevanu akciju. Rezultat svoje akcije ovaj interpreter prikazuje na standardnom izlazu.

Prilikom preduzimanja zahtevane akcije, interpreter znakovnog komandnog jezika se oslanja na sistemske operacije. Pri tome, on koristi delove komandi, odnosno njen operator i njene operande, kao argumente sistemskih operacija. Na primer, do izvršavanja pojedinih korisničkih programa dolazi tako što interpreter znakovnog komandnog jezika poziva sistemsku operaciju stvaranja procesa, a kao njene argumente upotrebi operator i operande komande, odnosno putanju izvršne datoteke i putanje datoteka sa obrađivanim podacima. Ovi argumenti su namenjeni stvaranom procesu. Tako, na primer, kod programskog jezika C parametri funkcije **main()** omogućuju preuzimanje broja argumenata (**argc**) iz komandne linije, kao i stringova pojedinih argumenata (**argv**), jer su ovi smešteni na stek procesa prilikom njegovog stvaranja.

Interpreter znakovnog komandnog jezika obavlja obradu znakova komande, pre nego što ih iskoristi kao argumente neke sistemske operacije. Zahvaljujući tome, u okviru operanada komandi se mogu javiti **specijalni znakovi** (*magic character, wild cards*), kao što je, na primer, znak *****. Njegova upotreba je vezana, pre svega, za imena datoteka i namenjena je za skraćeno označavanje grupa datoteka. Tako, na primer:

```
*.obj
```

označava sve objektne datoteke u radnom imeniku, a

```
d*1.txt
```

označava sve tekst datoteke u radnom imeniku, čiji prvi deo imena započinju znakom **d**, a završava cifrom **1**. Zahvaljujući specijalnim znakovima, moguće je, na primer, jednom komandom uništiti sve objektne datoteke iz radnog imenika:

```
unisti *.obj
```

ili odštampati sve tekst datoteke iz radnog imenika, čiji prvi deo imena započinje znakom **d**, a završava znakom **1**:

```
stampaj d*1.txt
```

Za obavljanje ovakvih akcija, neophodno je pretraživanje imenika i provera imena datoteka.

Zahvaljujući obradi znakova komande, moguće je interpreteru znakovnog komandnog jezika saopštiti i da **preusmeri** (*redirect*) standardni ulaz i standardni izlaz sa tastature i ekrana na proizvoljno odabrane datoteke. Ovo je važno za pozadinske procese, koji nisu u interakciji sa korisnikom. Zahvaljujući preusmeravanju, pozadinski proces ne ometa interaktivni rad korisnika, jer, umesto tastature i ekrana, koristi odabrane datoteke. Međutim, da bi se korisnik upozorio na greške u toku aktivnosti pozadinskog procesa, uz standardni izlaz se uvodi i **standardni izlaz greške**. Pošto je standardni izlaz greške namenjen, pre svega, za prikazivanje poruka o greškama, kao podrazumevajući standardni izlaz greške služi specijalna datoteka, koja odgovara ekranu. Ova datoteka se otvara za vreme stvaranja procesa, a kao njen indeks za tabelu otvorenih datoteka može da služi vrednost 2. I standardni izlaz greške se može preusmeriti na proizvoljnu datoteku, čiji sadržaj tada ukazuje na eventualne greške u toku aktivnosti pozadinskog procesa.

Uobičajeno je da preusmeravanje standardnog ulaza najavljuje znak **<**, a da preusmeravanje standardnog izlaza (kao i standardnog izlaza greške) najavljuje znak **>**. Tako, na primer, komanda:

```
kompiliraj < program.c > program.obj
```

saopštava interpreteru znakovnog komandnog jezika da stvori proces na osnovu izvršne datoteke **kompiliraj.bin**, pri čemu kao standardni ulaz procesa služi datoteka **program.c**, a kao njegov standardni izlaz datoteka **program.obj**. U ovom primeru ekran i dalje služi kao standardni izlaz greške. Izvršavanje prethodne komande omogućuje kompilaciju C programa, sadržanog u datoteci **program.c**. Rezultat kompilacije se smešta u datoteku **program.obj**, a eventualne poruke o greškama kompilacije se prikazuju na ekranu.

Preusmeravanje standardnog ulaza i izlaza predstavlja osnovu za kombinovanje izvršavanja više korisničkih programa. Važnost pomenutog kombinovanja se može pokazati na primeru uređivanja (sortiranja) reči iz nekog rečnika po kriteriju rimovanja. Nakon sortiranja reči po ovom kriteriju, sve reči, koje se rimuju, nalaze se jedna uz drugu. Umesto pravljenja posebnog programa za sortiranje reči po kriteriju rimovanja, jednostavnije je napraviti program za obrtanje redosleda znakova u rečima (tako da prvi i poslednji znak zamene svoja mesta u reči, da drugi i pretposlednji znak zamene svoja mesta u reči i tako redom) i kombinovati izvršavanje ovog programa sa izvršavanjem postojećeg programa za sortiranje:

```
obrni < recnik.txt > obrnuti_recnik.txt
sortiraj < obrnuti_recnik.txt > sortirani_obrnuti_recnik.txt
obrni < sortirani_obrnuti_recnik.txt > rime.txt
```

Umesto preusmeravanja standardnog ulaza i standardnog izlaza, moguće je nadovezati standardni izlaz jednog procesa na standardni ulaz drugog procesa i tako obrazovati tok procesa (*pipe*). Nadovezivanje u tok se označava pomoću znaka **|**. Za prethodni primer ovakav tok bi izgledao:

```
obrni < recnik.txt | sortiraj | obrni > rime.txt
```

U ovom primeru su u tok nadovezana tri procesa. Prvi je nastao na osnovu komande:

```
obrni < recnik.txt
```

drugi je nastao na osnovu komande

```
sortiraj
```

a treći je nastao na osnovu komande:

```
obrni > rime.txt
```

Ako bi se reči zadavale sa tastature, a po rimama sortirani rečnik prikazivao na ekranu, prethodni tok bi izgledao:

```
obrni | sortiraj | obrni
```

Razmena podataka između dva procesa, koji su povezani u tok, se ostvaruje posredstvom posebne specijalne datoteke. Njoj odgovara bafer u radnoj memoriji. Ova baferovana specijalna datoteka služi prvom od ovih procesa kao standardni izlaz, a drugom od njih kao standardni ulaz. Znači, prvi proces samo piše u ovu datoteku, a drugi samo čita iz nje. Prilikom obrazovanja toka procesa, interpreter znakovnog komandnog jezika stvara procese, koji se povezuju u tok. Pri tome on koristi istu posebnu specijalnu datoteku kao standardni izlaz i ulaz za svaki od parova ovih procesa. Zatim interpreter znakovnog komandnog jezika čeka na kraj aktivnosti poslednjeg od ovih procesa.

Pozadinski procesi se razlikuju od običnih (interaktivnih) procesa po tome što interpreter znakovnog komandnog jezika, nakon stvaranja pozadinskog procesa, ne čeka kraj njegove aktivnosti, nego nastavlja interakciju sa korisnikom. Zato su pozadinski procesi u principu neinteraktivni. Na primer, komanda:

kompiliraj < program.c > program.obj &

omogućuje stvaranje pozadinskog procesa, koji obavlja kompilaciju programa, sadržanog u datoteci **program.c**, i rezultat kompilacije smešta u datoteku **program.obj**, a eventualne greške u kompilaciji prikazuje na ekranu. U prethodnom primeru znak **&** sa kraja komande je naveo interpreter znakovnog komandnog jezika na stvaranje pozadinskog procesa.

Svaka komanda, upućena interpreteru znakovnog komandnog jezika, ne dovodi do stvaranja procesa. Komande, koje se često koriste, pa je važno da budu brzo obavljene, interpreter znakovnog komandnog jezika obavlja sam. Za ostale komande, za koje se stvaraju procesi, interpreter znakovnog komandnog jezika čeka kraj aktivnosti stvorenog procesa da bi od njega dobio, kao povratnu informaciju, završno stanje stvorenog procesa. Ovo stanje se obično kodira celim brojem. Ako interpreter znakovnog komandnog jezika protumači ovaj broj kao logičku vrednost (0 - tačno, različito od 0 - netačno), tada on može da podrži uslovno izvršavanje programa. Tako, na primer, komanda:

```
if kompiliraj < program.c > program.obj
then
linkuj < program.obj > program.bin
fi
```

označava da do linkovanja dolazi samo nakon uspešne kompilacije. Pri tome su **if**, **then** i **fi** rezervisane reči za interpreter znakovnog komandnog jezika. Prva najavljuje komandu, na osnovu koje interpreter znakovnog komandnog jezika stvori proces. Za vreme aktivnosti ovog procesa usledi kompilacija programa, sadržanog u datoteci **program.c**. Ako kompilacija prođe bez grešaka, stvoreni proces vraća interpreteru znakovnog komandnog jezika vrednost 0 kao svoje završno stanje. U ovom slučaju, interpreter znakovnog komandnog jezika interpretira komandu (komande) između rezervisanih reči **then** i **fi** i stvara proces, čija aktivnost dovodi do linkovanja datoteke **program.obj** sa potprogramima iz sistemske biblioteke. Ime sistemske biblioteke se ne navodi, jer se podrazumeva. U suprotnom slučaju, ako je bilo grešaka u kompilaciji, pa je završno stanje procesa, zaduženog za kompilaciju, bilo različito od vrednosti 0, interpreter znakovnog komandnog jezika ne interpretira komandu (komande) između rezervisanih reči **then** i **fi** i ne stvara proces zadužen za linkovanje.

Pored prethodno opisane komande za uslovno izvršavanje programa, interpreteri znakovnih komandnih jezika podržavaju komandu za ponavljanje izvršavanja programa, ali i druge komande, tipične za procedurne programske jezike, koje omogućuju rukovanje promenljivim, konstantama, ulazom, izlazom i slično. To dozvoljava pravljenje komandnih datoteka. One opisuju okolnosti pod kojima se izvršavaju korisnički programi, a sadržaj komandne datoteke preuzima na interpretiranje interpreter znakovnog komandnog jezika. Zato komandne datoteke imaju poseban tip, da bi ih

interpreter znakovnog komandnog jezika mogao prepoznati. Zahvaljujući tome, ime svake komandne datoteke, uostalom, kao i ime svake izvršne datoteke, predstavlja ispravnu komandu znakovnog komandnog jezika. Iako broj i vrste ovakvih komandi nisu ograničeni, jer zavise samo od kreativnosti i potreba korisnika, ipak je moguće napraviti njihovu klasifikaciju i navesti neke neizbežne grupe komandi. Najgrublja podela komandi je na:

1. korisničke komande i
2. administratorske komande.

Korisničke komande, između ostalog, omogućuju:

1. rukovanje datotekama,
2. rukovanje imenicima,
3. rukovanje procesima i
4. razmenu poruka između korisnika.

Standardne komande za rukovanje datotekama omogućuju:

1. izmenu imena (kao i atributa) datoteke,
2. poređenje sadržaja datoteka,
3. kopiranje datoteka i
4. uništenje datoteka.

U komande za rukovanje imenicima spadaju:

1. komande za stvaranje i uništenje imenika,
2. komanda za promenu radnog imenika,
3. komanda za pregledanje sadržaja imenika (imena datoteka i imena imenika, sadržanih u njemu) i
4. komande za izmenu imena i ostalih atributa imenika.

Administratorske komande omogućuju:

1. pokretanje i zaustavljanje rada računara,
2. spašavanje (*backup*) i vraćanje (*restore*) datoteka,
3. rukovanje vremenom,
4. sabijanje (*compaction*) datoteka,
5. ažuriranje podataka o korisnicima računara i njihovim pravima,
6. generisanje izveštaja o korišćenju računara (o korišćenju procesorskog vremena ili o korišćenju prostora na disku),
7. rukovanje konfiguracijom računara (određivanje načina rada uređaja i programa koji ulaze u njegov sastav),
8. proveru ispravnosti rada računara i
9. pripremu diskova za korišćenje (ovo obuhvata pronalaženje oštećenih blokova i njihovo isključivanje iz upotrebe, pronalaženje izgubljenih blokova i njihovo uključivanje u evidenciju slobodnih blokova, formiranje skupa datoteka na disku i njegovo uključivanje u skup datoteka računara).

16.2.2 Grafički komandni jezici

Interpreteri grafičkih komandnih jezika omogućuju pozivanje bilo koje od prethodnih komandi, a da pri tome ne zahtevaju od korisnika da znaju napamet imena komandi, niti

da zadaju komande posredstvom tastature, uz obavezu strogog poštovanja sintakse znakovnog komandnog jezika. Umesto toga, grafički komandni jezici uvode grafičku predstavu komandi (*icon*), ili spiskove sa imenima komandi (*menu*), dozvoljavajući korisnicima da pozovu komandu izborom njene grafičke predstave, ili izborom njenog imena sa spiska imena komandi. Grafički komandni jezici dozvoljavaju i da se komanda automatski pokrene izborom nekog od prikazanih imena datoteka. Pretpostavka za ovo je da izabrana datoteka predstavlja podrazumevajući operand date komande.

Za komunikaciju sa grafičkim komandnim jezicima potreban je pokazivački uređaj kao što je miš. On omogućuje pokazivanje tačke ekrana. Zadatak interpretera grafičkog komandnog jezika je da, na osnovu pozicije (koordinata) pokazane tačke i pritisaka na odgovarajuću dirku pokazivačkog uređaja, odredi šta korisnik želi. Na primer, ako pokazana tačka pripada skupu tačaka zone ekrana koja sadrži grafičku predstavu komande ili njeno ime, tada dva uzastopna pritiska na odgovarajuću dirku pokazivačkog uređaja izazivaju obavljanje odabrane komande.

16.3 PITANJA

1. Od čega se sastoje komande znakovnog komandnog jezika?
2. Kako se zadaju komande grafičkih komandnih jezika?
3. Šta su ciljevi znakovnih komandnih jezika?
4. Šta omogućuju znakovni komandni jezici?
5. Šta omogućuju čarobni znakovi?
6. Šta omogućuje preusmeravanje?
7. Čemu služi *pipe*?
8. Čemu služi baferovana specijalna datoteka?
9. Šta karakteriše pozadinske procese?
10. Šta karakteriše komandne datoteke?
11. Šta omogućuju korisničke komande?
12. Šta omogućuju administratorske komande?

17 KLASIFIKACIJA OPERATIVNIH SISTEMA

17.1 KRITERIJUM KLASIFIKACIJE OPERATIVNIH SISTEMA

Jedan od mogućih kriterijuma za klasifikaciju operativnih sistema je vrsta računara kojim operativni sistem upravlja. Po tom kriteriju mogu se izdvojiti:

1. operativni sistemi realnog vremena
2. multiprocesorski operativni sistemi i
3. distribuirani operativni sistemi.

17.2 OPERATIVNI SISTEMI REALNOG VREMENA

Operativni sistemi realnog vremena (*real time operating system*) su namenjeni za primene računara u kojima je neophodno obezbediti reakciju na vanjski događaj u unapred zadanom vremenu. Ovakvi operativni sistemi su, zbog toga, podređeni ostvarenju što veće brzine izvršavanja korisničkih programa.

Za operativne sisteme realnog vremena je tipično da su, zajedno sa računarom, ugrađeni (*embedded*) u sistem, čije ponašanje se ili samo prati, ili čijim ponašanjem se upravlja. Zadatak operativnih sistema realnog vremena je da samo stvore okruženje za korisničke programe, jer komunikaciju sa krajnjim korisnikom obavljaju korisnički programi. Zato se operativni sistemi realnog vremena obično koriste samo na programskom nivou.

Modul za rukovanje procesima je podređen potrebi brzog stvaranja i uništenja procesa, njihove brze i lake saradnje, kao i brzog preključivanja procesora sa procesa na proces. Zato obično svi procesi dele isti fizički adresni prostor. To je moguće, jer ne postoji potreba za međusobnom zaštitom procesa, pošto oni imaju istog autora, ili njihovi autori pripadaju istom timu.

Modul za rukovanje datotekama nije obavezni deo operativnog sistema realnog vremena, jer sve primene realnog vremena ne zahtevaju masovnu memoriju. Kada rukovanje datotekama postoji, ono obično podržava kontinuirane datoteke, zbog brzine pristupa podacima. Mane kontinuiranih datoteka se ovde ne ispoljavaju, jer su unapred poznati svi zahtevi primene.

Modul za rukovanje radnom memorijom obično podržava efikasno zauzimanje memorijskih zona sa unapred određenom veličinom, da bi se izbegla ili umanjila eksterna fragmentacija.

Modul za rukovanje kontrolerima podržava tipične ulazne i izlazne uređaje i, uz to, omogućava jednostavno uključivanje novih drajvera za specifične uređaje. Pri tome se nude blokirajuće i neblokirajuće sistemske operacije, ali i vremenski ograničene blokirajuće sistemske operacije. Zahvaljujući neblokirajućim sistemskim operacijama, moguće je vremenski preklopiti aktivnosti procesora i kontrolera. Vremenski ograničene blokirajuće sistemske operacije omogućuju reakciju u zadanom vremenskom intervalu na izostanak željenog događaja, odnosno, na izostanak obavljanja pozvane sistemske

operacije.

Modul za rukovanje procesorom mora da obezbedi efikasno rukovanje vremenom. Za to se često koriste posebni satovi - tajmeri. U sklopu toga, mora se obezbediti da aktivnost procesa bude završena do graničnog trenutka (*deadline scheduling*). To se postiže sortiranjem deskriptora spremnih procesa po dužini preostalog vremena do graničnog trenutka (*earliest deadline first*). Podrazumeva se da ovo sortiranje dovodi na prvo mesto deskriptor procesa sa najkraćim preostalim vremenom do graničnog trenutka, tako da je njegov proces prvi na redu za aktiviranje. Ako je aktivnost procesa periodična sa unapred određenim i nepromenljivim trajanjem aktivnosti u svakom periodu, tada se procesima mogu dodeliti prioriteti jednaki broju njihovih perioda u jedinici vremena (*rate monotonic scheduling*). Tako se može obezbediti da aktivnost procesa bude obavljena pre kraja svakog od njegovih perioda. U svakom slučaju, postavljeni cilj raspoređivanja može da se ostvari samo ako ima dovoljno procesorskog vremena za sve procese.

Modul za rukovanje procesorom operativnog sistema realnog vremena obično podržava mehanizam semafora, jer je on jednostavan za korišćenje, brz i jer ne zahteva izmene kompajlera.

17.3 MULTIPROCESORSKI OPERATIVNI SISTEMI

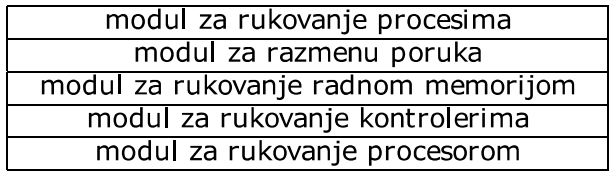
Multiprocesorski operativni sistemi upravljaju računarskim sistemom sa više procesora opšte namene, koji pristupaju zajedničkoj radnoj memoriji. Podrazumeva se da ove procesore i radnu memoriju povezuje sabirnica.

Specifičnosti multiprocesorskog operativnog sistema su vezane za modul za rukovanje procesorom i posledica su istovremene aktivnosti više procesa na raznim procesorima. Zato se sinhronizacija procesa u ovakvim okolnostima više ne može zasnivati na onemogućenju prekida, nego na zauzimanju sabirnice, jer je to jedini način da se spreči da više od jednog procesa pristupa istoj lokaciji radne memorije. Mogućnost istovremene aktivnosti više procesa na raznim procesorima usložnjava raspoređivanje, jer ono mora da odabere ne samo proces, koji će biti aktivan, nego i da odabere procesor, koji će da se preključi na odabrani proces.

17.4 DISTRIBUIRANI OPERATIVNI SISTEMI

Distribuirani operativni sistemi upravljaju međusobno povezanim računarima, koji su prostorno udaljeni. Potrebu za povezivanjem prostorno udaljenih (distribuiranih) računara nameće praksa. S jedne strane, prirodno je da računari budu na mestima svojih primena, na primer, uz korisnike ili uz delove industrijskih postrojenja, koje opslužuju. Na taj način računari mogu biti potpuno posvećeni lokalnim poslovima, koji su vezani za mesta njihove primene, pa mogu efikasno obavljati ovakve poslove. S druge strane, neophodno je omogućiti saradnju između prostorno udaljenih korisnika, odnosno obezbediti usaglašeni rad prostorno udaljenih delova istog industrijskog postrojenja. Za to je potrebno obezbediti razmenu podataka između računara, posvećenih pomenutim korisnicima, odnosno posvećenih pomenutim delovima industrijskog postrojenja. Radi toga, ovakvi, prostorno udaljeni računari se povezuju komunikacionim linijama, koje

omogućuju prenos (razmenu) podataka, organizovanih u poruke. Na ovaj način nastaje **distribuirani računarski sistem** (*distributed computer system*). Za svaki od računara, povezanih u distribuirani računarski sistem, je neophodno da sadrže procesor, radnu memoriju i mrežni kontroler. Prisustvo masovne memorije i raznih ulaznih i izlaznih uređaja u sastavu ovakvih računara zavisi od mesta njihove primene i, u opštem slučaju, nije obavezno. Zato nema ni potrebe da ih podržava operativni sistem, prisutan na računarima iz distribuiranog računarskog sistema. Ovakav operativni sistem ima smanjenu funkcionalnost u odnosu na "običan" operativni sistem, pa se naziva **mikrokernel** (*microkernel*). Hijerarhijska struktura mikrokernela je prikazana na slici (Slika 17.1).



Slika 17.1: Hijerarhijska struktura mikrokernela

Mikrokernel ne sadrži modul za rukovanje datotekama, jer on nije potreban za svaki od računara iz distribuiranog računarskog sistema. Zato se ovaj modul prebacuje u korisnički sloj (iznad mikrokernela), koji je predviđen za korisničke procese.

Modul za rukovanje procesima se oslanja na modul za razmenu poruka, da bi pristupio izvršnoj datoteci koja je locirana na nekom drugom računaru. Modul za razmenu poruka se oslanja na modul za rukovanje radnom memorijom, radi dinamičkog zauzimanja i oslobađanja bafera, namenjenih za privremeno smeštanje poruka. Modul za razmenu poruka se oslanja i na modul za rukovanje kontrolerima, u kome se nalazi drajver mrežnog kontrolera, posredstvom koga se fizički razmenjuju poruke. Na kraju, modul za razmenu poruka se oslanja i na modul za rukovanje procesorom. Ovo je potrebno, da bi se, na primer, privremeno zaustavila aktivnost procesa do prijema poruke, bez koje nastavak aktivnosti nije moguć, ali i da bi se moglo reagovati na dugotrajni izostanak očekivanog prijema poruke.

Modul za razmenu poruka nije samo na raspolaganju modulu za rukovanje procesima. On sadrži sistemske operacije, koje omogućuju razmenu poruka, odnosno saradnju između procesa, aktivnih na raznim računarima, kao i saradnju između procesa, aktivnih na istom računaru. Tipičan oblik saradnje procesa je da jedan proces traži uslugu od drugog procesa. To je potrebno, na primer, kada jedan proces želi da na svom računaru, koji je bez masovne memorije, stvori novi proces. On se, tada, posredstvom modula za rukovanje procesima, obraća drugom procesu, aktivnom na računaru sa masovnom memorijom, zahtevajući od njega, kao uslugu, da mu pošalje sadržaj odgovarajuće izvršne datoteke. Uobičajeni način traženja i dobijanja usluge se sastoji od pozivanja operacije, čije obavljanje dovodi do pružanja tražene usluge. Ako pozivana operacija ne odgovara potprogramu koji se lokalno izvršava u okviru aktivnosti procesa pozivaoca, nego odgovara potprogramu koji se izvršava u okviru aktivnosti drugog,

udaljenog procesa, aktivnog na udaljenom računaru, reč je o **pozivu udaljene operacije** (*Remote Procedure Call - RPC*). Proces, koji poziva udaljenu operaciju, se nalazi u ulozi **klijenta** (primaoca usluge), a proces, koji obavlja udaljenu operaciju, se nalazi u ulozi **servera** (davaoca usluge).

17.4.1 Poziv udaljene operacije

Poziv udaljene operacije liči na poziv (lokalne) operacije. Znači, on ima oblik poziva potprograma, u kome se navode oznaka (ime) operacije i njeni argumenti. Ovakav potprogram se naziva **klijentski potprogram** (*client stub*), jer je klijent njegov jedini pozivalac. U klijentskom potprogramu je sakriven niz koraka, koji se obavljaju, radi dobijanja zahtevane usluge. U ove korake spadaju:

1. pronalaženje procesa servera, koji pruža zahtevanu uslugu,
2. pakovanje (*marshalling*) argumenata (navedenih u pozivu klijentskog potprograma) u poruku zahteva,
3. slanje serveru ove poruke zahteva,
4. prijem od servera poruke odgovora sa rezultatom pružanja zahtevane usluge,
5. raspakivanje prispele poruke odgovora i
6. isporuka rezultata pružanja zahtevane usluge pozivaocu klijentskog potprograma.

Simetrično klijentskom potprogramu postoje dva **serverska potprograma** (*server stub*). Njih poziva jedino server, a oni kriju više koraka, koji se obavljaju, radi pružanja zahtevane usluge. Prvi od serverskih potprograma obuhvata:

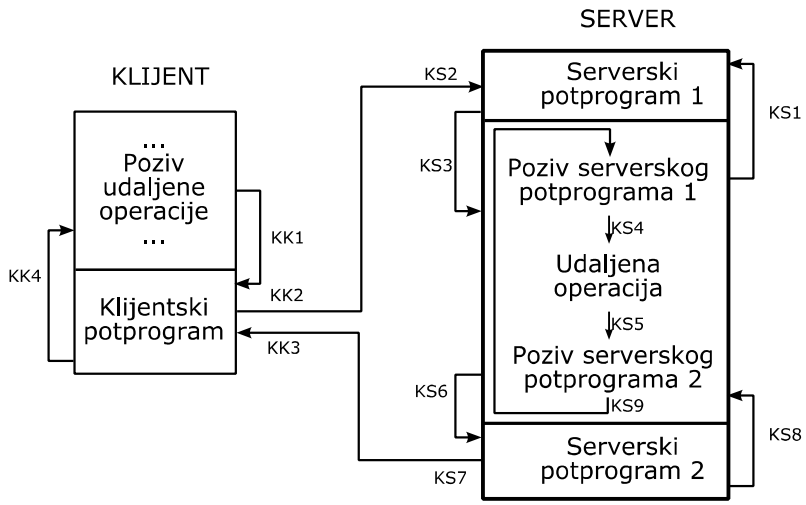
1. prijem poruke zahteva i
2. raspakivanje argumenata iz ove poruke.

Drugi od serverskih potprograma obuhvata:

1. pakovanje rezultata usluge (koju je pružio server) u poruku odgovora i
2. slanje klijentu ove poruke odgovora.

Između poziva ova dva serverska potprograma se nalazi lokalni poziv operacije, koja odgovara zahtevanoj usluzi, odnosno, programski tekst, koji opisuje aktivnost servera na pružanju zahtevane usluge.

Slika 17.2 prikazuje redosled koraka klijenta i servera tokom poziva udaljene operacije (kk1 do kk4 označavaju sekvencu koraka klijenta, a ks1 do ks9 označavaju sekvencu koraka servera).



Slika 17.2: Poziv udaljene operacije

Oslanjanje na poziv udaljene operacije olakšava posao programeru, jer od njega krije, na prethodno opisani način, detalje saradnje klijenta i servera. Pri tome, klijentski potprogram pripada biblioteci udaljenih operacija. Ova biblioteka sadrži po jedan klijentski potprogram za svaku od postojećih udaljenih operacija. Klijentski potprogram se generiše, zajedno sa serverskim potprogramima, prilikom prevođenja programa, koji odgovara serveru.

17.4.2 Problemi poziva udaljene operacije

Uprkos nastojanju da što više liči na poziv lokalne operacije, poziv udaljene operacije se značajno razlikuje od svog uzora. Te razlike su posledica koraka, sakrivenih u pozivu udaljene operacije, koji uzrokuju da se u toku poziva udaljene operacije mogu da pojave problemi, čija pojava nije moguća kod poziva lokalne operacije. Tako je moguće:

1. da se ne pronađe server, koji pruža zahtevanu uslugu,
2. da se, u toku prenosa, izgube ili poruka zahteva ili poruka odgovora, kao i
3. da dođe do otkaza ili servera, ili klijenta u toku njihovog rada.

Ako nema servera, tada nije moguće pružanje tražene usluge. To je nemoguća situacija kod poziva lokalne operacije. Do istog rezultata dovode smetnje na komunikacionim linijama, koje onemogućuju prenos bilo poruke zahteva, bilo poruke odgovora. Kada u očekivanom vremenu izostane prijem poruke odgovora, bilo zbog gubljenja poruke zahteva, bilo zbog gubljenja poruke odgovora, jedino što se na strani klijenta može uraditi je da se ponovo pošalje (retransmituje) poruka zahteva. Pri tome je broj retransmisija ograničen. Ako je izgubljena poruka zahteva, njenom retransmisijom se stvara mogućnost da ona stigne do servera i da on pruži traženu uslugu. Međutim, ako je izgubljena poruka odgovora, tada treba sprečiti da, po prijemu retransmitovane poruke zahteva, server ponovi pružanje već pružene usluge. Da bi server razlikovao retransmitovanu poruku od originalne, dovoljno je da svaka originalna poruka ima jedinstven redni broj i da server za svakog klijenta pamti redni broj poslednje primljene

poruke zahteva od tog klijenta. Prijem poruke sa zapamćenim rednim brojem ukazuje na retransmitovanu poruku, koja je već primljena.

Otkaz servera, izazvan kvarom računara, je neprijatan zbog teškoća da se ustanovi da li je do otkaza došlo pre, u toku, ili posle pružanja usluge. Zato je problematično da poziv udaljene operacije garantuje da će zahtevana usluga biti pružena samo jednom, kao kod poziva lokalne operacije. Na primer, na strani klijenta otkaz servera se ispoljava kao izostajanje poruke odgovora. Tada retransmisija poruke zahteva može navesti ponovo pokrenutog servera da još jednom pruži već pruženu uslugu, jer je, u ovom slučaju, server izgubio, zbog otkaza, evidenciju o rednim brojevima poslednje primljenih poruka zahteva od klijenata. Kod poziva lokalne operacije ovo se ne može desiti, jer otkaz računara znači i kraj izvršavanja celog programa, bez pokušaja njegovog automatskog oporavka. Garantovanje da će server pružiti zahtevanu usluga samo jednom podrazumeva upotrebu stabilne memorije (*stable storage*) koja je ima visoku pouzdanost, pa može ispravno funkcionisati i u okolnostima otkaza servera.

Otkaz klijenta znači da server uzaludno pruža zahtevanu uslugu. Ovo se izbegava tako što server obustavlja pružanje usluga klijentima, za koje ustanovi da su doživeli otkaz. To klijenti sami mogu da jave serveru, nakon svog ponovnog pokretanja, ili to server može sam da otkrije, periodičnom proverom stanja klijenata, koje opslužuje.

Poziv udaljene operacije praktično dozvoljava da argumenti budu samo vrednosti, a ne i adrese, odnosno pokazivači, zbog problema kopiranja pokazanih vrednosti sa klijentovog računara na računar servera i u obrnutom smeru. Pored toga, ako su ovi računari različiti, javlja se i problem konverzije vrednosti, jer se, na primer, predstava realnih brojeva razlikuje od računara do računara.

Činjenica da se u okviru klijentskog potprograma javlja potreba za pronalaženjem servera, ukazuje da u vreme pravljenja izvršnog oblika klijentskog programa nije poznato koji server će usluživati klijenta. U opštem slučaju, može biti više servera iste vrste i svaki od njih može istom klijentu da pruži zatraženu uslugu. Radi toga se uvodi poseban **server imena** (*name server, binder*). Njemu se, na početku svoje aktivnosti, obraćaju svi serveri i ostavljaju podatke o sebi, kao što je, na primer, podatak o vrsti usluge koje pružaju. Serveru imena se obraćaju i klijenti, radi pronalaženja servera, koji pruža zahtevanu uslugu. Na ovaj način se ostvaruje **dinamičko linkovanje** (*dynamic binding*) klijenta, koji zahteva uslugu, i servera, koji pruža zahtevanu uslugu.

17.4.3 Razmena poruka

Klijentski i serverski potprogrami, koji omogućuju poziv udaljene operacije, se oslanjaju na sistemske operacije modula za razmenu poruka. Prva od ovih operacija je sistemska operacija **zahtevanja usluge**, a druge dve su sistemske operacije **prijema zahteva** i **slanja odgovora**. Sistemska operacija zahtevanja usluge je namenjena klijentu i poziva se iz njegovog potprograma. Ona omogućuje slanje poruke zahteva i prijem poruke odgovora. Sistemske operacije prijema zahteva i slanja odgovora su namenjene serveru i omogućuju prijem poruke zahteva i slanje poruke odgovora. Sistemska operacija prijema zahteva se poziva iz prvog serverskog potprograma, a sistemska operacija slanja odgovora se poziva iz drugog serverskog potprograma. Ove tri sistemske

operacije ostvaruju poseban **protokol razmene poruka** (*request reply protocol*), koji je prilagođen potrebama poziva udaljene operacije.

Sistemske operacije zahtevanja usluge, prijema zahteva i slanja odgovora su blokirajuće. Prva zaustavlja aktivnost klijenta do stizanja odgovora, ili do isticanja zadanog vremenskog perioda. Druga zaustavlja aktivnost servera do stizanja zahteva, a treća zaustavlja aktivnost servera do isporuke odgovora ili do isticanja zadanog vremenskog intervala. Ove tri sistemske operacije su zadužene za prenos poruka. Pored slanja i prijema poruka, one (1) potvrđuju prijem poruka, (2) retransmituju poruke, čiji prijem nije potvrđen, (3) šalju upravljačke poruke, kojima se proverava i potvrđuje aktivnost servera (čime se omogućuje otkrivanje njegovog otkaza) i slično. U nadležnosti ovih operacija je i rastavljanje poruka u pakete, koji se prenose preko komunikacionih linija, sastavljanje poruka od paketa, pristiglih preko komunikacionih linija, potvrda prijema paketa i retransmisija paketa čiji prijem nije potvrđen, kao i prilagođavanje brzine slanja paketa brzini kojom oni mogu biti primani (*flow control*). Pomenute tri sistemske operacije koriste usluge drajvera sata, radi reagovanja na isticanje zadanih vremenskih intervala, nakon kojih je, na primer, potrebno ili retransmitovati poruku, ili poslati poruku potvrde. One pozivaju i (neblokirajuće) operacije gornjeg dela drajvera mrežnog kontrolera, radi fizičkog prenosa i prijema paketa. U donjem delu ovog drajvera se nalaze obrađivači prekida, zaduženi za registrovanje uspešnog slanja i uspešnog prijema paketa.

Sistemske operacije modula za razmenu poruka se brinu o baferima, namenjenim za (privremeno) smeštanje poruka. Na primer, ako server nije pozvao sistemsku operaciju prijema zahteva, jer je aktivan na usluživanju prethodno primljenog zahteva od jednog klijenta, a pristigla je poruka zahteva od drugog klijenta, ova poruka se smešta u slobodan bafer, da bi bila sačuvana i kasnije isporučena serveru. Ako ne postoji slobodan bafer, poruka zahteva se odbacuje, uz, eventualno, slanje odgovarajuće upravljačke poruke drugom klijentu.

Svaka od poruka, koje se razmenjuju između procesa, se sastoji:
1. od upravljačkog dela poruke i
2. od sadržaja poruke.

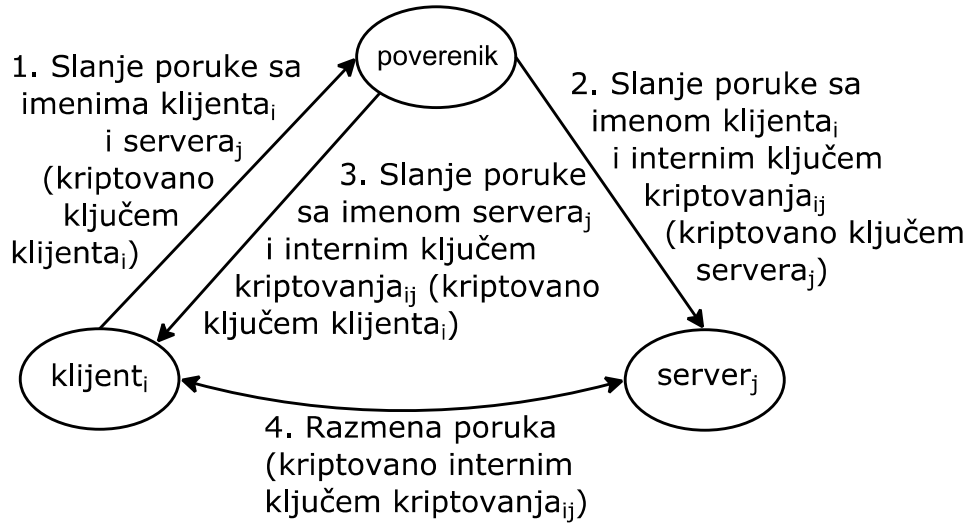
Upravljački deo poruke obuhvata:
1. adresu odredišnog procesa (kome se poruka upućuje),
2. adresu izvorišnog procesa (od koga poruka kreće, a kome se, eventualno, kasnije upućuje odgovor) i
3. opis poruke (njenu vrstu, njen redni broj i slično).

Adresa (odredišnog ili izvorišnog) procesa sadrži jedinstven redni broj računara, kome proces pripada (a po kome se razlikuju svi računari), kao i port (jedinstven redni broj po kome se razlikuju procesi, koji pripadaju istom računaru). Na osnovu rednog broja računara, mrežni kontroler utvrđuje da li prihvata ili propušta poruku, a na osnovu porta se određuje proces, kome se poruka isporučuje. U toku programiranja, zgodnije je, umesto ovih rednih brojeva, koristiti imena za označavanje i računara i procesa. Korespondenciju između imena i rednih brojeva uspostavlja već pomenuti server imena. Ove podatke o sebi ostavljaju svi serveri, kada se, na početku svoje aktivnosti, obrate

serveru imena.

17.4.4 Problemi razmene poruka

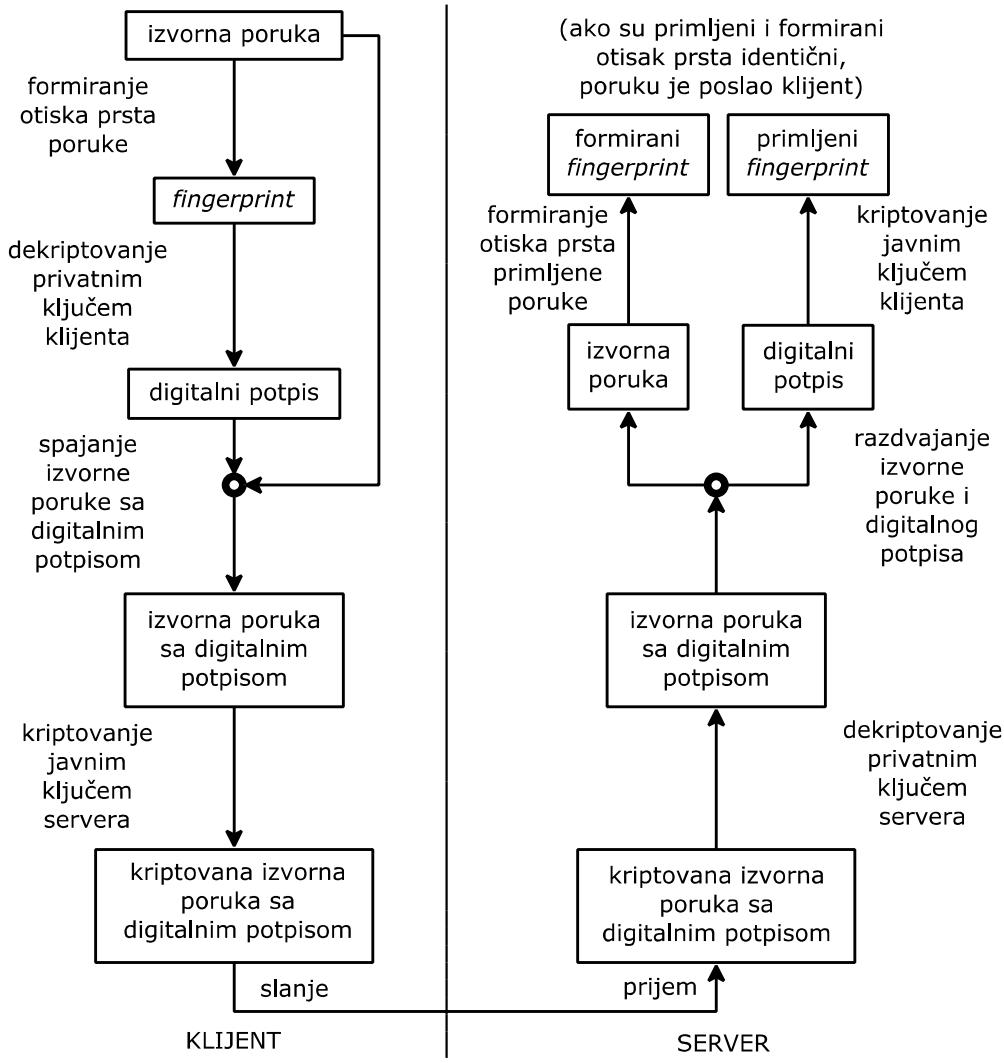
Slaba tačka razmene poruka je sigurnost, jer su komunikacione linije pristupačne svim korisnicima, pa je svaki od njih u poziciji da preuzima tuđe poruke i da šalje poruke u tuđe ime. Sprečavanje preuzimanja tuđih poruka se zasniva na kriptovanju (*encryption*) poruka, a sprečavanje slanja poruka u tuđe ime se zasniva na nedvosmislenoj međusobnoj identifikaciji procesa (*authentication*). U slučaju simetrične kriptografije, da bi klijent i server mogli da razmenjuju poruke sa kriptovanim sadržajima, oba moraju da znaju i algoritam kriptovanja i zajednički interni ključ kriptovanja. Pod pretpostavkom da je algoritam poznat svim procesima, a da interni ključ kriptovanja treba da znaju samo klijent i server, koji razmenjuju poruke, javlja se problem kako dostaviti interni ključ kriptovanja samo pomenutom klijentu i serveru. U tome može da pomogne poseban server, u koga svi procesi imaju poverenje i koji se, zato, naziva **poverenik** (Slika 17.3). Pri tome se podrazumeva da poverenik poseduje unapred dogovoren poseban ključ kriptovanja za komunikaciju sa svakim procesom. Zahvaljujući tome, klijent može da pošalje povereniku poruku, koja sadrži ime klijenta i ime servera sa kojim klijent želi da ostvari sigurnu komunikaciju. Sadržaj ove poruke je kriptovan ključem, koji je poznat samo klijentu i povereniku, tako da je razumljiv samo za poverenika, a on, na osnovu adrese izvorišnog procesa iz upravljačkog dela ove poruke, može da pronađe ključ za dekriptovanje njenog sadržaja. Poverenik tada odredi interni ključ kriptovanja i pošalje poruku serveru, koja sadrži interni ključ kriptovanja i ime klijenta. Sadržaj ove poruke je kriptovan ključem, koji je poznat samo povereniku i serveru, tako da je razumljiv samo za servera. Takođe, poverenik šalje poruku i klijentu, koja sadrži interni ključ kriptovanja i ime servera. Sadržaj ove poruke je kriptovan ključem, koji znaju samo poverenik i klijent, tako da je razumljiv samo za klijenta. Na ovaj način, samo klijent i samo server dobiju interni ključ kriptovanja za sigurnu međusobnu komunikaciju i ujedno se obavi njihova međusobna identifikacija, tako da se drugi procesi ne mogu neprimećeno umešati u njihovu komunikaciju. Opisano ponašanje klijenta i servera predstavlja oblik sigurnosnog protokola (*security protocol, cryptographic protocol, encryption protocol*).



Slika 17.3: Sigurna komunikacija klijenta i servera

Ako se sigurna razmena poruka zasniva na asimetričnoj kriptografiji, tada je uloga poverenika da čuva javne ključeve i tako osigura međusobnu identifikaciju procesa. Znači, kada je potrebno ostvariti sigurnu komunikaciju između dva procesa, oni se obraćaju povereniku, da bi dobili javni ključ svog komunikacionog partnera. Za komunikaciju sa poverenikom ovi procesi koriste unapred dogovoreni javni ključ poverenika, a za komunikaciju sa njima poverenik koristi njihove unapred dogovorene javne ključeve.

Asimetrična kriptografija, sa komutativnim algoritmima kriptovanja i dekriptovanja, omogućuje i digitalno potpisivanje poruka, radi neopozivog pripisivanja poruke njenom pošiljaocu (Slika 17.4). **Digitalni potpis** (*digital signature*) se šalje uz poruku. On sadrži podatke koji jednoznačno reprezentuju poruku, pa predstavljaju **otisak prsta poruke** (*fingerprint*, *cryptographic checksum*). Otisak prsta poruke formiraju **jednosmerne funkcije** (*one-way functions*) na osnovu sadržaja poruke. Digitalni potpis nastane kada se otisak prsta poruke dekriptuje (transformiše) primenom algoritma dekriptovanja i privatnog ključa. Primalac poruke kriptuje (retransformiše) digitalni potpis primenom algoritma kriptovanja i javnog ključa. Ako se rezultat kriptovanja digitalnog potpisa poklapa sa otiskom prsta primljene poruke, tada je poruka nedvosmisleno stigla od pošiljaoca.



Slika 17.4: Digitalno potpisivanje poruka

Sigurnu komunikaciju klijenta i servera mogu ometati drugi procesi zlonamernim retransmisijama starih poruka, ili izmenom sadržaja poruka. Ugrađivanjem u sadržaj poruke njenog rednog broja, mogu se otkriti retransmisije starih poruka, a ugrađivanjem u sadržaj poruke kodova za otkrivanje i oporavak od izmena sadržaja, mogu se otkriti, pa i ispraviti izmene sadržaja poruka.

17.4.5 Razlika klijenata i servera

Različita uloga, koju klijent i server imaju u toku međusobne komunikacije (saradnje), je prirodna posledica njihove namene. Iz toga proizlaze i razlike u njihovoj internoj organizaciji. Dok je za klijenta prihvatljivo da njegova aktivnost bude strogo

sekvencijalna, za servera stroga sekvencijalnost njegove aktivnosti znači manju propusnost i sporije pružanje usluga. To je najlakše ilustrovati na primeru servera datoteka, zaduženog za pružanje usluga, kao što je čitanje ili pisanje datoteke. Strogo sekvencijalna aktivnost ovoga servera bi izazvala zaustavljanje njegove aktivnosti, radi usluživanja jednog klijenta, dok kontroler ne prenese blok sa sadržajem datoteke između masovne i radne memorije. U međuvremenu ne bi bilo usluživanja drugih klijenata, čak i ako bi se njihovi zahtevi odnosili na blokove datoteka, prisutne u baferima radne memorije. Ovakva sekvencijalnost nije prisutna kod tradicionalnih operativnih sistema, jer nakon zaustavljanja aktivnosti jednog procesa u modulu za rukovanje datotekama, drugi proces može nastaviti aktivnost u istom modulu. Zato je za servere potrebno obezbediti više niti. Pri tome, svaka od niti, u okviru istog servera, opslužuje različitog klijenta, a broj ovih niti zavisi od broja postavljanih zahteva i menja se u vremenu. Postojanje više niti zahteva njihovu sinhronizaciju, dok pristupaju globalnim (statičkim) promenljivim servera. Iako je primena više niti tipična za servere, ona ima svoje opravdanje i kod klijenata, jer može poboljšati njihovo ponašanje.

17.4.6 Poziv operacije udaljenog objekta

Poziv udaljene operacije ima kao alternativu poziv operacije udaljenog objekta (*Remote Method Invocation - RMI*). Za poziv operacija nekog objekta je potrebno raspolagati njegovom referencom i poznavati operacije koje su za njega definisane. Pri tome ne smeta ako je objekat udaljen, odnosno ako se ne nalazi na istoj mašini kao i proces koji poziva operaciju dotičnog objekta. Dobijanje reference udaljenog objekta, postupak poziva njegove operacije i dobijanje rezultata izvršavanja pozvane operacije se suštinski ne razlikuju od rešavanja sličnih problema kod poziva udaljene operacije.

17.4.7 Distribuirani sistem datoteka

Distribuirani sistem datoteka obuhvata hijerarhijsku organizaciju datoteka čiji delovi se nalaze na raznim računarima. Ovakav distribuirani sistem datoteka se može oslanjati na više **servera imenika** i na više **servera datoteka**. Serveri imenika podržavaju hijerarhijsku organizaciju datoteka, a serveri datoteka podržavaju pristup sadržaju (običnih) datoteka. U imenicima, kojima rukuju serveri imenika, uz imena datoteka, odnosno uz imena imenika, ne stoje samo redni brojevi deskriptora datoteka, nego i redni brojevi servera datoteka, odnosno servera imenika, kojima pripadaju pomenuti deskriptori.

U distribuiranom sistemu datoteka pristup datoteci podrazumeva konsultovanje servera imena, radi pronalaženja servera imenika, od koga kreće pretraživanje imenika. Pretraživanje imenika može zahtevati kontaktiranje različitih servera imenika, dok se ne stigne do servera datoteka sa traženom datotekom.

Serveri imenika i datoteka mogu da ubrzaju pružanje usluga, ako kopiju često korišćenih podataka čuvaju u radnoj memoriji. Ubrzanju pružanja usluga doprinosi i **repliciranje datoteka**, da bi one bile fizički bliže korisnicima. Međutim, to stvara probleme, kada razni korisnici istovremeno menjaju razne kopije iste datoteke, jer se tada postavlja pitanje koja od izmena je važeća.

Za distribuirani sistem datoteka zaštitu datoteka je primerenije zasnovati na **dozvolama** (*capability*), nego na pravima pristupa. To znači da u okviru deskriptora datoteka, odnosno imenika, ne postoje navedena prava pristupa za pojedine grupe korisnika, nego se za svaku datoteku, odnosno imenik, generišu različite dozvole. One omogućuju razne vrste pristupa datoteci, odnosno imeniku. Da bi klijent dobio neku uslugu, on mora da poseduje odgovarajuću dozvolu, koju prosleđuje serveru u okviru zahteva za uslugom. Dozvola sadrži:

1. redni broj servera,
2. redni broj deskriptora datoteke (odnosno, deskriptora imenika),
3. oznaku vrste usluge i
4. oznaku ispravnosti dozvole.

Sadržaj dozvole je zaštićen kriptovanjem, tako da nije moguće, izmenom oznake vrste usluge prepraviti dozvolu. Pre pružanja usluge, server dekriptuje sadržaj dozvole, i proverava da li je ona ispravna i da li se njena oznaka vrste usluge podudara sa zatraženom uslugom. Dozvole deli server na zahtev klijenata, koji ih čuvaju i po potrebi prosleđuju jedan drugom. Pri tome se podrazumeva da klijent, stvaralac datoteke, po njenom stvaranju automatski dobije dozvolu za sve vrste usluga, koja uključuje i uslugu stvaranja drugih, restriktivnijih dozvola. Kada želi da poništi određenu dozvolu, server samo proglasi njenu oznaku ispravnosti nevažećom.

Prednost zasnivanja zaštite datoteka na dozvolama umesto na pravima pristupa je u tome da prvi pristup ne zahteva razlikovanje korisnika, niti njihovo označavanje. To je važno, jer rukovanje jedinstvenim i neponovljivim oznakama korisnika u distribuiranom računarskom sistemu nije jednostavno. Sem toga, dozvole omogućuju veću selektivnost, jer grupišu korisnike po kriteriju posedovanja dozvole određene vrste, a ne na osnovu njihovih unapred uvedenih (numeričih) oznaka.

Komercijalni distribuirani sistem datoteka nastaje spajanjem lokalnog i udaljenog sistema datoteka. Osnovu za stvaranje komercijalnog distribuiranog sistema datoteka nudi, na primer, NFS (*Sun Microsystem's Network File Service*). Pristup udaljenom sistemu datoteka podrazumeva mrežnu komunikaciju lokalnog klijenta i udaljenog mrežnog servera datoteka, o čijim detaljima korisnik ne mora da vodi računa.

17.4.8 Raspoređivanje procesa u distribuiranom računarskom sistemu

Cilj raspoređivanja procesa u distribuiranom računarskom sistemu je ostvarenje najboljeg iskorišćenja računara, ili ostvarenje najkraćeg vremena odziva, odnosno, najbržeg usluživanja korisnika. Zadatak raspoređivanja komplikuju razlike između računara, jer u opštem slučaju svaki računar ne može da prihvati svaki izvršni oblik programa, pošto su izvršni oblici programa vezani za procesor, za raspoloživu radnu memoriju i slično. Raspoređivanje komplikuje i zahtev za omogućavanje migracije procesa sa računara na računar, da bi se prezaposlen računar rasteretio, a nezaposlen zaposlio. Raspoređivanje je olakšano, ako su unapred poznate karakteristike opterećenja računara, odnosno vrsta i broj njihovih procesa. Kod raspoređivanja procesa po računarima, važno je voditi računa o saradnji procesa i procese, koji tesno međusobno sarađuju, raspoređivati na isti računar.

17.4.9 Distribuirana sinhronizacija

Saradnja procesa, aktivnih na raznim računarima, zatheva njihovu sinhronizaciju, što se ostvaruje razmenom poruka. Pri tome, najjednostavniji način za ostvarenje sinhronizacije se zasniva na uvođenju **procesa koordinatora**. Njemu se obraćaju svi procesi, zainteresovani za sinhronizaciju, a koordinator donosi odluke o njihovoj sinhronizaciji. Tako, ako je potrebno, na primer, ostvariti međusobnu isključivost procesa u pristupu istoj datoteci, svi procesi traže od koordinatora dozvolu za pristup, a on dozvoljava uvek samo jednom procesu da pristupi datoteci. Na sličan način se može ostvariti i uslovna sinhronizacija. Za razliku od ovakvog centralizovanog algoritma sinhronizacije, koji se zasniva na uvođenju koordinatora, postoje i distribuirani algoritmi sinhronizacije, koji se zasnivaju na međusobnom dogovaranju procesa, zainteresovanih za sinhronizaciju. Distribuirani algoritmi sinhronizacije zahtevaju sredstva za grupnu komunikaciju procesa, odnosno, efikasna sredstva koja omogućuju da jedan proces pošalje poruke svim ostalim procesima iz grupe procesa, zainteresovanih za sinhronizaciju, i da od njih primi odgovore. Pored veće razmene poruka, distribuirani algoritmi sinhronizacije su komplikovaniji od centralizovanih algoritama, a pri tome ne nude prednosti, tako da je njihov razvoj više od principijelnog, nego od praktičnog značaja.

Za distribuirane operativne sisteme nije samo bitno da omoguće efikasnu sinhronizaciju procesa, nego i da podrže poseban oblik sinhronizacije procesa, koji obezbeđuje da se obave ili sve operacije iz nekog niza pojedinačnih operacija, ili ni jedna od njih. Ovakav niz operacija se naziva transakcija, a transakcije, koje imaju svojstvo da se obave u celosti ili nikako, se nazivaju **atomske transakcije** (*atomic transaction*). Primer niza operacija, za koje je neophodno da obrazuju atomsku transakciju, je prebacivanje nekog iznosa sa računa jedne banke na račun druge banke. Pri tome, proces klijent, koji u ime korisnika obavlja ovo prebacivanje, kontaktira dva servera, koji reprezentuju dve banke, da bi obavio transfer iznosa sa jednog na drugi račun. Transfer se mora obaviti tako, da drugi klijenti mogu videti oba računa samo u stanju ili pre, ili posle transakcije. Znači, za atomske transakcije je neophodno da budu međusobno isključive, ako pristupaju istim podacima, ali i da njihovi rezultati budu trajni, da se jednom napravljena izmena ne može izgubiti.

Saradnja procesa, aktivnih na raznim procesorima, otvara mogućnost pojave mrtve petlje. Pri tome, u uslovima distribuiranog računarskog sistema, algoritmi za izbegavanje pojave mrtve petlje, odnosno za otkrivanje i za oporavak od pojave mtrve petlje su još neefikasniji i sa još manjim praktičnim značajem, nego u slučaju centralizovanog (jednoprocesorskog) računara. Zato, u uslovima distribuiranog računarskog sistema, kada nije prihvatljiv pristup ignorisanja problema mrtve petlje, preostaje da se spreči njena pojava, na primer, sprečavanjem ispunjenja uslova, neophodnih za pojavu mrtve petlje.

17.4.10 Svojstva distribuiranog računarskog sistema

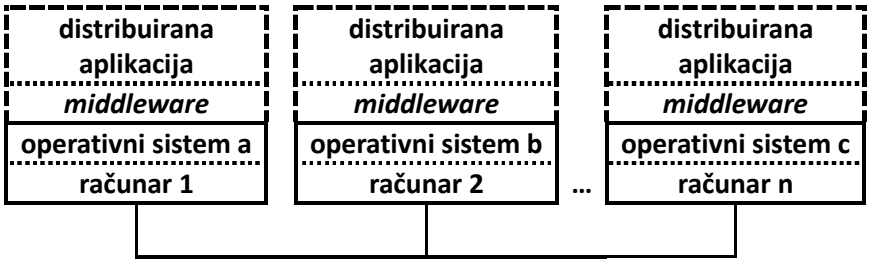
Distribuirani računarski sistem je zamišljen tako da integriše mnoštvo računara u moćan multiračunarski sistem. Na taj način je moguće od više jeftinih i malih računara

napraviti moćan multiračunarski sistem povoljne cene. Ovakav multiračunarski sistem, uz to, nudi i veću pouzdanost, jer kvar pojedinačnog računara nije fatalan za ceo sistem, kao i mogućnost proširenja, jer je moguće naknadno dodavanje računara u sistem. Pored integrisanja pojedinačnih računara u multiračunarski sistem, distribuirani računarski sistem omogućuje i deljenje skupih resursa ovakvog multiračunarskog sistema između više korisnika, a nudi i prilagodljivost zahtevima korisnika, željenu raspoloživost i predvidivost odziva, pa, čak, i veću sigurnost, jer korisnici mogu da čuvaju poverljive podatke na svom računaru, koga fizički štite i čije korišćenje mogu da kontrolišu.

17.4.11 Implementacija distribuiranog operativnog sistema

Zadatak distribuiranog operativnog sistema je da objedini sve računare distribuiranog računarskog sistema, tako da korisnik ne vidi pojedine računare, nego jedinstven sistem, koji pruža usaglašene usluge. Sve ovakve usluge se pružaju na uniforman način, koji zanemaruje mesto i druge specifičnosti pružanja usluge.

Uniforman način pružanja usluga podrazumeva sakrivanje različitosti (heterogenosti) računara od kojih je obrazovan distribuirani računarski sistem. To se može postići, ako se iznad raznorodnih operativnih sistema pojedinih računara distribuiranog računarskog sistema napravi posebna distribuirana softverska platforma (*middleware*) za razvoj distribuiranih softverskih sistema (Slika 17.5). Ona ima ulogu distribuiranog operativnog sistema.



Slika 17.5: Distribuirana softverska platforma

Distribuirana softverska platforma je obično specijalizovana tako da nudi konzistentan skup operacija, koje omogućuju razvoj željene vrste distribuiranih softverskih sistema. Ovakav skup operacija podržava saradnju komponenti koje obrazuju distribuirani softverski sistem. U tom pogledu se može govoriti o raznim distribuiranim softverskim platformama, poput one namenjene za podršku distribuiranih dokumenata (*World Wide Web*), ili one namenjene za podršku distribuiranom objektno orijentisanom programiranju (*CORBA*, *Java middleware*, *.NET*). U osnovi ovakvih sistema se krije klijent-server model. Server se nalazi na strani rukovaoca distribuiranim dokumentima (*web site*) ili rukovaoca objektima, a klijent se nalazi na strani korisnika distribuiranih dokumenata (*web browser*) ili pozivaoca operacija udaljenih objekata (pri čemu je pozivanje ovih operacija zasnovano na RPC mehanizmima).

17.5 PITANJA

1. Šta karakteriše operativne sisteme realnog vremena?
2. Šta karakteriše multiprocesorske operativne sisteme?
3. Koje module sadrži mikrokernel?
4. Šta karakteriše poziv udaljene operacije (RPC)?
5. Šta radi klijentski potprogram?
6. Za šta su zaduženi serverski potprogrami?
7. Koji problemi su vezani za poziv udaljene operacije?
8. Šta podrazumeva dinamičko linkovanje klijenta i servera?
9. Koje operacije podržava protokol razmene poruka između klijenta i servera?
10. Za šta su zadužene sistemske operacije koje ostvaruju protokol razmene poruka?
11. Šta sadrže poruke koje razmenjuju klijent i server?
12. Šta je potrebno za sigurnu razmenu poruka između klijenta i servera?
13. Šta karakteriše digitalni potpis?
14. Od čega zavisi propusnost servera?
15. Šta sadrže dozvole na kojima se zasniva zaštita datoteka u distribuiranom sistemu?
16. Šta karakteriše distribuiranu sinhronizaciju?
17. Šta karakteriše distribuirani računarski sistem?
18. Šta karakteriše distribuiranu softversku platformu?

18 PARALELNO PROGRAMIRANJE

18.1 CILJ PARALELNOG PROGRAMIRANJA

Cilj paralelnog programiranja je da skrati vreme obrade podataka korišćenjem paralelizma koga nude računari sa više procesora opšte namene. Taj cilj se ostvaruje pronalaženjem **relativno nezavisnih delova** ukupne obrade podataka i zaduživanjem posebnih niti da ih obavljaju. Ostvarenje pomenutog cilja podrazumeva da su aktivnosti ovih niti istovremene, odnosno da su vezane za različite procesore. Način dekompozicije obrade podataka u relativno nezavisne delove zavisi od karaktera obrade podataka, ali i od oblika raspoloživog paralelizma. Zato se paralelni programi specijalizuju za pojedine oblike paralelizma.

Paralelno programiranje se zasniva na konkurentnom programiranju, jer relativna nezavisnost delova obrade podataka znači da je, pre ili kasnije, neizbežna saradnja za njih zaduženih niti. Pored toga, zahvaljujući konkurentnom programiranju, paralelni programi se mogu razvijati i na jednoprocesorskom računaru, na kome, jasno, nije moguće demonstrirati skraćenje vremena obrade podataka koje paralelni programi nude.

18.2 PARALELNO PROGRAMIRANJE ZASNOVANO NA RAZMENI PORUKA

Saradnju niti paralelnog programa je najbolje zasnovati na razmeni poruka, jer takav oblik saradnje osigurava najveću nezavisnost niti. Međutim, razmena poruka je ograničena oblikom paralelizma koji određuje između kojih procesora (i njima pridruženih niti) postoji mogućnost za direktnu razmenu poruka. To znači da se paralelni program može izvršavati na jednoprocesorskom računaru, ako se obezbede komunikacioni kanali, saglasni sa oblikom paralelizma za koga je paralelni program specijalizovan. Zadatak ovih komunikacionih kanala je da podrže direktnu razmenu poruka samo između onih niti, između kojih je ta razmena moguća u pomenutom obliku paralelizma. Na taj način se niti vezuju, na primer, u niz (*array*), matricu (*mesh*) ili potpuno međusobno povezuju.

18.3 PARALELNO SUMIRANJE NIZA BROJEVA

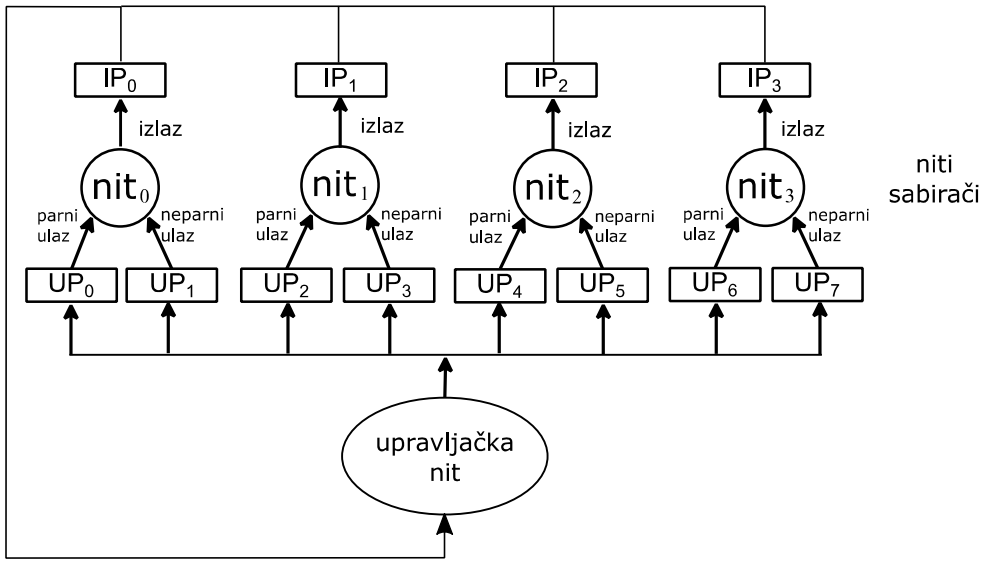
18.3.1 Zamisao paralelnog sumiranja niza brojeva

Paralelno određivanje zbira niza brojeva započinje istovremenim sabiranjem raznih parova brojeva iz niza, a nastavlja se istovremenim sabiranjem raznih parova prethodno paralelno određenih suma, dok se ne dobije zbir svih brojeva iz niza. Pravilnost prethodnog postupka nalaže da dužina niza bude jednaka nekom stepenu broja 2. Na početku sumiranja angažovani broj **niti sabirača** je jednak polovini dužine niza. Na svakom sledećem koraku se angažuje upola manje niti sabirača, dok u poslednjem koraku ne preostane aktivna samo jedna nit sabirač. U svakom koraku aktivne niti sabirači sabiraju po dva broja koja su im dodeljena i njihov zbir prosleđuju dalje. Posebna upravljačka nit pre svakog koraka upućuje brojeve nitima sabiračima i nakon

svakog koraka preuzima sume od angažovanih niti sabirača.

18.3.2 Komunikaciona osnova paralelnog sumiranja niza brojeva

Svaka nit sabirač koristi par ulaznih pregradaka (**UPi** i **UPi+1**) za preuzimanje sumiranih brojeva i jedan izlazni pregradak (**IPi**) za prosleđivaje njihove sume (Slika 18.1).



Slika 18.1: Niti sabirači i upravljačka nit

Paralelno sumiranje se obavlja u koracima. U svakom koraku sumiranja upravljačka nit upućuje brojeve iz niza u ulazne pregratke niti sabirača i preuzima parcijalne sume iz izlaznih pregradaka niti sabirača.

Templejt klasa **Adder_boxes** (Listing 18.1) omogućuje međusobno komunikaciono povezivanje niti sabirača sa upravljačkom niti. Dva parametra templejt klase **Adder_boxes** omogućuju definisanje tipa poruke i broja niti sabirača. Elementi polja **in_slots** odgovaraju ulaznim pregracima (**UPi**), a elementi polja **out_slots** odgovaraju izlaznim pregracima (**IPi**). Podrazumeva se da se kao oznake niti sabirača koriste redni brojevi: 0, 1, 2, 3 i tako dalje. Operacije **send()** i **receive()** omogućuju razmenu poruka. Prvi par ovih operacija omogućuje upravljačkoj niti da razmenjuje poruke sa nitima sabiračima posredstvom njihovih ulaznih i izlaznih pregradaka. Drugi par ovih operacija omogućuje nitima sabiračima da razmenjuju poruke sa upravljačkom niti posredstvom svojih ulaznih i izlaznih pregradaka. Konstante **EVEN_IN** i **ODD_IN** označavaju parni i neparni ulazni pregradak.

Listing 18.1: Klasa **Adder_boxes** (datoteka **adder_boxes.hh**)

```
#include "box.hh"

enum
In_adder_boxes { EVEN_IN = 0, ODD_IN = 1 };

template <class MESSAGE, int THREADS>
class Adder_boxes {
   Message_box<MESSAGE> in_slots[THREADS * 2];
   Message_box<MESSAGE> out_slots[THREADS];
 public:
   void send_in(int box, const MESSAGE message)
      { in_slots[box].send(&message); };
   MESSAGE receive_out(int box)
      { return out_slots[box].receive(); };
   void send_out(int sender, const MESSAGE message)
      { out_slots[sender].send(&message); };
   MESSAGE receive_in(int receiver, In_adder_boxes source)
      { return in_slots[(receiver) * 2 + source].receive(); };
};
```

18.3.3 Izvedba paralelnog sumiranja niza brojeva

Klasa **Thread_identity** (Listing 18.2) omogućuje jednoznačno numerisanje niti (sabirača).

Listing 18.2: Klasa **Thread_identity** (datoteka **thread_identity.hh**)

```
class Thread_identity {
   mutex mx;
   int identity;
 public:
   Thread_identity(int value = 0) : identity(value) {};
   int get();
};

int
Thread_identity::get()
{
   unique_lock<mutex> lock(mx);
   return identity++;
}
```

Funkcija **thread_adder** (Listing 18.3) opisuje ponašanje niti sabirača. Niti sabirači, nakon dobijanja svoje oznake, preuzimaju iz svojih ulaznih pregradaka brojeve za sabiranje, sabiraju ih i sumu prosleđuju u svoj izlazni pregradak. Niti sabirače stvara (korišćenjem bezimenih objekata klase **thread**) upravljačka nit, čiju aktivnost opisuje funkcija **thread_manager**. Upravljačka nit još preuzima brojeve za sabiranje, upućuje ih nitima sabiračima i zatim ponavlja preuzimanje parcijalnih suma od niti sabirača i upućivanje ka njima tih parcijalnih suma, dok ne preuzme i ne prikaže traženi zbir.

Listing 18.3: Paralelno sumiranje niza brojeva (datoteka **p06.cpp**)

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include "adder_boxes.hh"
#include "thread_identity.hh"

const int
THREADS = 4;

Adder_boxes<int, THREADS>
adder_boxes;

Thread_identity
thread_identity;

void
thread_adder()
{
  int tid = thread_identity.get();
  for(;;)
    adder_boxes.send_out(tid, adder_boxes.receive_in(tid, EVEN_IN) +
                adder_boxes.receive_in(tid, ODD_IN));
}
```

```
void
thread_manager()
{
   int number;
   int counter;
   int limit = THREADS;
   cout << endl << "PARALLEL SUMMATION"
        << endl << "input " << short(THREADS * 2)
        << " integers for summation" << "(integer values from 1 to 100)";
   for(counter = 0; counter < limit; counter++) {
      thread (thread_adder).detach();
   }
   for(counter = 0; counter < limit * 2; counter++) {
      do {
         cout << endl << ((short)counter) << ". integer: ";
         cin >> number;
      } while((number < 1) || (number > 100));
      adder_boxes.send_in(counter, number);
   }
   while(limit > 1) {
      for(counter = 0; counter < limit; counter++)
         adder_boxes.send_in(counter, adder_boxes.receive_out(counter));
      limit =  limit / 2;
   }
   cout << endl << "total sum: " << adder_boxes.receive_out(0) << endl;
}

int
main()
{
   thread manager(thread_manager);
   manager.join();
}
```

Sadržaj izvorne datoteke **p06.cpp** (Listing 18.3) predstavlja potpun konkurentni program. U toku njegovog izvršavanja nastane pet niti.

Dužina sumiranja, opisanog prethodnim programom, je proporcionalna logaritmu (baze dva) broja sumiranih brojeva, ako se svakoj niti sabiraču dodeli poseban procesor. Ako se prethodni program izvršava na jednoprocesorskom računaru, tada je dužina sumiranja proporcionalna broju sumiranih brojeva.

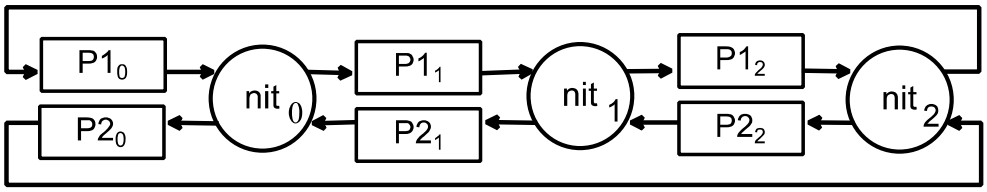
18.4 PARALELNO SORTIRANJE

18.4.1 Zamisao paralelnog sortiranja

Zadatak sortiranja niza znakova je njihovo uređenje u zadanom redosledu. Sortiranje znakova započinje tako da se na prvo mesto niza dovede znak koji prethodi svim preostalim znakovima. Radi toga se poredi znak, koji je zatečen na prvom mestu niza, sa znakovima, koji su zatečeni na preostalim mestima nesortiranog niza. Cilj ovih poređenja je otkrivanje parova znakova čiji redosled nije u skladu sa željenim uređenjem i eventualna zamena mesta ovih znakova. Analognim postupkom se dovode na drugo i sva ostala mesta znakovi koji prethode svim preostalim znakovima, tako da su na kraju svi znakovi poredani u zadanom redosledu. Postupci dovođenja željenih znakova na pojedina mesta niza su međusobno nezavisni kada se ne odnose na iste znakove, pa se mogu odvijati paralelno. Oni, znači, mogu biti predmet aktivnosti raznih niti, pod uslovom da ovakvih **sortnih niti** ima koliko i sortiranih znakova, kao i da su one uvezane u niz, odnosno da obrazuju protočnu strukturu. Tada znakovi, jedan za drugim, mogu da teku od prve ka poslednjoj sortnoj niti, tako da prva od sortnih niti može da izdvoji znak koji je na prvom mestu u uređenju, sortna nit iza nje znak koji je na drugom mestu u uređenju i tako dalje. Pri tome se podrazumeva da svaka sortna nit zadrži prvi znak koji primi. Ona zatim, poredi svaki novopridošli sa prethodno zadržanim znakom. Nakon poređenja, sortna nit zadržava uvek znak koji je u skladu sa željenim uređenjem, a šalje dalje preostali znak. Posebna upravljačka nit upućuje znakove na sortiranje, šaljući ih prvoj sortnoj niti i preuzima ih sa sortiranja, primajući ih prvo od prve, pa od druge i na kraju od poslednje sortne niti.

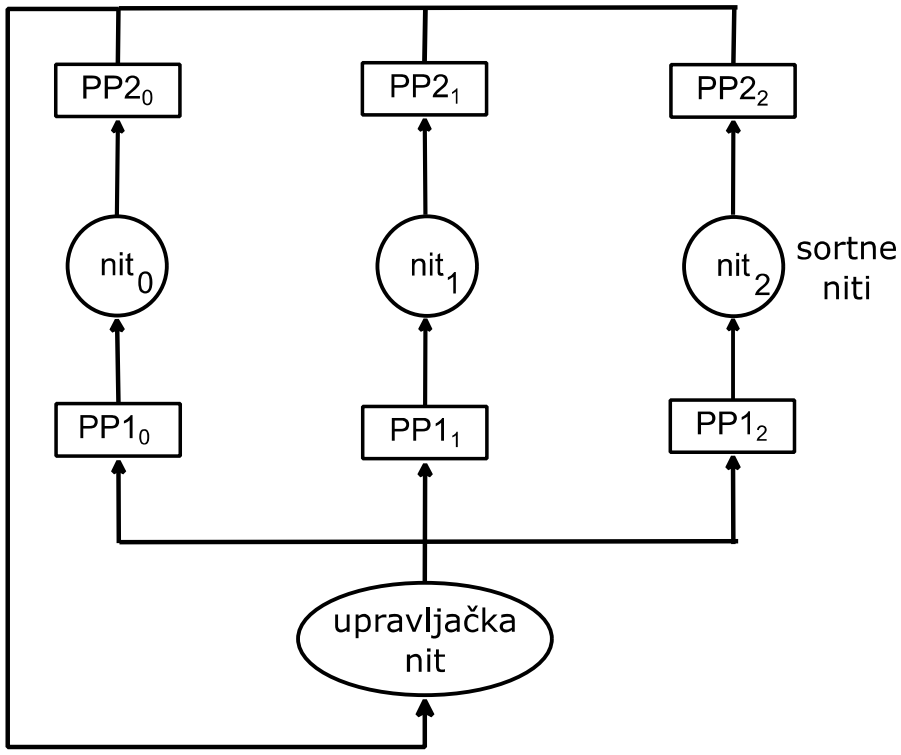
18.4.2 Komunikaciona osnova paralelnog sortiranja

Vezivanje sortnih niti u niz je inspirisano idejom jednodimenzionalne sistoličke arhitekture (*array processors*). Svaka sortna nit iz ovog niza ima levo i desno od sebe po dva pregratka (**P1i**) i (**P2i**) za poruke (Slika 18.2).



Slika 18.2: Niti uvezane u niz

Posmatrajući relativno, sa stanovišta svake sortne niti, pregraci levo uspostavljaju komunikacioni kanal od prethodnika i ka prethodniku iz niza. Slično pregraci desno uspostavljaju komunikacioni kanal ka sledbeniku i od sledbenika. Pošto je niz cirkularan, prva sortna nit je sledbenik poslednje sortne niti (odnosno poslednja sortna nit je prethodnik prve sortne niti). Podrazumeva se da uz svaku sortnu nit postoji i par posebnih pregradaka (**PP1i**) i (**PP2i**) za direktnu komunikaciju sortnih niti sa upravljačkom niti (Slika 18.3).



Slika 18.3: Sortne niti i upravljačka nit

Templejt klasa **Array** (Listing 18.4) omogućuje uspostavljanje komunikacionih kanala koji su potrebni za uvezivanje sortnih niti u niz. Njena dva parametra omogućuju definisanje tipa poruke i broja sortnih niti u nizu. Elementi polja **slots1** i **slots2** odgovaraju, respektivno, pregracima (**P1i**) i (**P2i**), a elementi polja **special1** i **special2** odgovaraju posebnim pregracima (**PP1i**) i (**PP2i**), za direktnu komunikaciju sortnih niti sa upravljačkom niti. Podrazumeva se da se kao oznake sortnih niti koriste brojevi: 0, 1, 2, 3 i tako dalje. Operacije **send()** i **receive()** omogućuju razmenu poruka. Prvi par ovih operacija omogućuje upravljačkoj niti da razmenjuje poruke sa sortnim nitima (koje su identifikovane svojim oznakama). Drugi par ovih operacija omogućuje sortnim nitima da razmenjuju poruke sa svojim prethodnikom (koga označava konstanta **LEFT**), sa svojim sledbenikom (koga označava konstanta **RIGHT**) i sa upravljačkom niti (koju označava konstanta **SPECIAL**).

Polje **threads** pokazuje koliko je angažovano sortnih niti. Njegovo postavljanje omogućuje operacija **threads_set()**.

Listing 18.4: Klasa **Array** (datoteka **array.hh**)

```
#include "box.hh"

enum
Array_relative_position { LEFT, RIGHT, SPECIAL };

template<class MESSAGE, int THREADS>
class Array {
   Message_box<MESSAGE> slots1[THREADS];
   Message_box<MESSAGE> slots2[THREADS];
   Message_box<MESSAGE> special1[THREADS];
   Message_box<MESSAGE> special2[THREADS];
   int threads;
 public:
   Array() : threads(THREADS) {};
   void send(int receiver, const MESSAGE* message)
      { special1[receiver].send(message); };
   MESSAGE receive(int sender) { return special2[sender].receive(); };
   void threads_set(int n) { threads = n; };
   void send(int sender, Array_relative_position relative_position,
         const MESSAGE* message);
   MESSAGE receive(int receiver, Array_relative_position relative_position);
};
```

```
template<class MESSAGE, int THREADS>
void
Array<MESSAGE, THREADS>::send(int sender,
                    Array_relative_position relative_position,
                    const MESSAGE* message)
{
   Message_box<MESSAGE>* destination;
   switch(relative_position) {
    case LEFT:
       destination = &(slots2[sender]);
       break;
    case RIGHT:
       if(sender < threads)
          destination = &(slots1[sender+1]);
       else
          destination = &(slots1[0]);
       break;
    case SPECIAL:
       destination = &(special2[sender]);
       break;
   }
   destination->send(message);
}
```

```
template<class MESSAGE, int THREADS>
MESSAGE
Array<MESSAGE, THREADS>::receive(int receiver,
                                 Array_relative_position relative_position)
{
  Message_box<MESSAGE>* source;
  switch(relative_position) {
   case LEFT:
     source = &(slots1[receiver]);
     break;
   case RIGHT:
     if(receiver < threads)
       source = &(slots2[receiver+1]);
     else
       source = &(slots2[0]);
     break;
   case SPECIAL:
     source = &(special1[receiver]);
     break;
  }
  return source->receive();
}
```

Templejt klasa **Array** se zasniva na asinhronoj razmeni poruka, ali se može osloniti i na sinhronu razmenu poruka.

18.4.3 Izvedba paralelnog sortiranja

Ponašanje sortnih niti opisuje funkcija **thread_sorter()** (Listing 18.5). Sortne niti stvara (korišćenjem bezimenih objekata klase **thread**) upravljačka nit, čiju aktivnost opisuje funkcija **thread_manager()**. Upravljačka nit još preuzima sa tastature znakove za sortiranje, upućuje ih u protočnu strukturu, iz nje preuzima sortirane znakove i prikazuje ih na ekranu. Podrazumeva se da sortirani niz znakova na svom kraju sadrži razmak (*space*).

Listing 18.5: Paralelno sortiranje (datoteka **p07.cpp**)

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;
```

```
#include "array.hh"
#include "thread_identity.hh"

const int
THREADS = 10;

const char
SPACE = ' ';

Array<char, THREADS>
array;

Thread_identity
thread_identity;

char
receive(int receiver)
{
   char received;
   if(receiver == 0)
      received = array.receive(receiver, SPECIAL);
   else
      received = array.receive(receiver, LEFT);
   return received;
}
```

```
void
thread_sorter()
{
   char old_character;
   char new_character;
   int tid = thread_identity.get();
   if((old_character = receive(tid)) > SPACE) {
      while((new_character = receive(tid)) > SPACE) {
         if(new_character < old_character) {
            array.send(tid, RIGHT, &old_character);
            old_character = new_character;
         } else
            array.send(tid, RIGHT, &new_character);
      }
      array.send(tid, RIGHT, &new_character);
   }
   array.send(tid, SPECIAL, &old_character);
}

void
thread_manager()
{
   int counter;
   char c;
   cout << endl << "PARALLEL SORTING"
        << endl << "unsorted array of characters "
        << "(enter " << THREADS-1 << " characters):" << endl;
   for(counter = 1; counter < THREADS; counter++) {
      thread (thread_sorter).detach();
      cin >> c;
      array.send(0, &c);
   }
   thread (thread_sorter).detach();
   array.send(0, &SPACE);
   cout << endl << "sorted array of characters  (in ascending order)" << endl;
   for(counter = 1; counter <= THREADS; counter++)
      cout << array.receive(counter-1);
   cout << endl;
}
```

```
int
main()
{
   thread manager(thread_manager);
   manager.join();
}
```

Sadržaj izvorne datoteke **p07.cpp** (Listing 18.5) predstavlja potpun konkurentni program. U toku njegovog izvršavanja nastane jedanaest niti.

Dužina sortiranja zavisi od broja poređenja. Kod sekvencijalnog sortiranja po prethodnom algoritmu broj poređenja je: **(n-1)+(n-2)+...+1=n(n-1)/2** (za prvi element sortiranog niza napravi se **n-1** poređenja, za drugi **n-2** poređenja, a za pretposlednji element **1** poređenje). Znači dužina sortiranja je proporcionalna kvadratu broja sortiranih znakova.

Paralelno sortiranje, opisano prethodnim programom, omogućuje preklapanje poređenja različitih parova znakova, ako se svakoj od sortnih niti dodeli poseban procesor. U tom slučaju prva sortna nit obavi **n-1** poređenja, druga sortna nit obavi **n-2** poređenja, ali sa kašnjenjem od 2 poređenja iza prve sortne niti. Do kašnjenja dolazi jer tek nakon dva poređenja druga sortna nit dobija od prve dva znaka i tek tada sama može započeti poređenja. To znači da se samo poslednje poređenje druge sortne niti ne poklapa sa poređenjima prve sortne niti. Isti odnos postoji između druge i treće sortne niti i tako redom. Prema tome, na dužinu sortiranja utiče **n-1** poređenje prve sortne niti i po jedno (poslednje) poređenje preostalih **n-2** sortnih niti. Sledi da je ukupan broj poređenja koja utiču na dužinu sortiranja: **(n-1)+(n-2)=2n-3**, pa je dužina sortiranja linearno zavisna od broja sortiranih znakova.

18.5 PARALELNO MNOŽENJE MATRICA

18.5.1 Zamisao paralelnog množenja matrica

Izračunavanja elemenata matrice, koja je jednaka proizvodu druge dve matrice, su dovoljno međusobno nezavisna da mogu biti paralelna. To pokazuje primer množenja dvodimenzionalnih matrica A i B, čiji proizvod je jednak dvodimenzionalnoj matrici C (Slika 18.4).

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{a_{11}b_{11}}+a_{12}b_{21} & a_{11}b_{12}+\mathbf{a_{12}b_{22}} \\ a_{21}b_{11}+\mathbf{a_{22}b_{21}} & \mathbf{a_{21}b_{12}}+a_{22}b_{22} \end{bmatrix}$$

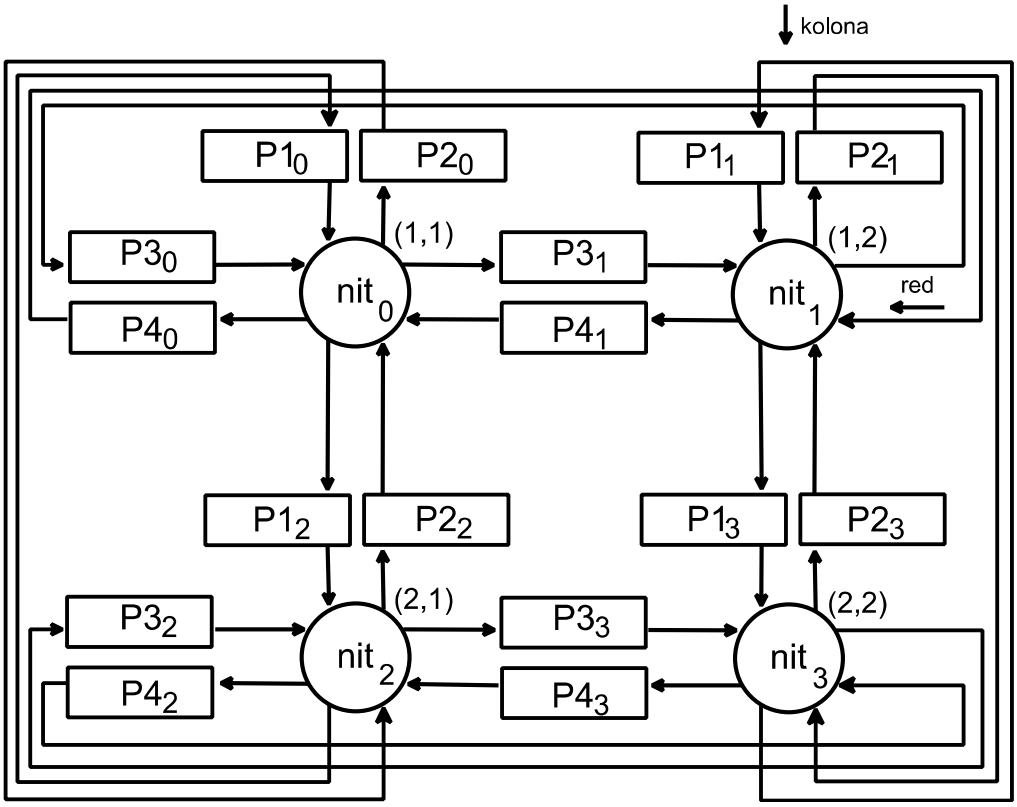
Slika 18.4: Proizvod dvodimenzionalnih matrica

Izračunavanju svakog od četiri elementa matrice C može se posveti posebna **nit**

množač. Tada, svaka od njih, nezavisno jedna od druge (znači paralelno), može da izračuna proizvod elemenata matrica A i B, koje su označene zadebljanim slovima (Slika 18.4), ako poseduje potrebne elemente. To se može postići, ako se niti množači prostorno rasporede kao elementi matrice C i tako komunikaciono povežu da svaka od njih može da šalje poruke nitima množačima desno i ispod sebe, a prima poruke od niti množača levo i iznad sebe. Podrazumeva se da na početku svaka nit množač primi od posebne upravljačke niti svoj element matrice A i svoj element matrice B. Pošto tada jedino nit množač (1,1) poseduje potreban par elemenata, ostale niti množači moraju da razmene svoje elemente. Tako niti množači (2,1) i (2,2) šalju desno raspoloživi element matrice A, a primaju s leva potrebni element iste matrice. Slično, niti množači (1,2) i (2,2) šalju dole raspoloživi element matrice B, a primaju odozgo potrebni element iste matrice. Nakon toga sve niti množači raspolažu potrebnim elementima matrica A i B, pa mogu istovremeno da odrede njihov proizvod. Za računanje preostalog proizvoda, potrebno je da niti množači međusobno razmene raspoložive elemente matrica A i B. Međusobna razmena elemenata matrica A i B je pravilna, pa sve niti množači šalju raspoloživi element matrice A desno, a raspoloživi element matrice B dole, i zatim primaju novi element matrice A sa leva, a novi element matrice B odozgo. Nakon ovakve razmene, sve niti množači raspolažu elementima neophodnim za računanje drugog proizvoda, pa mogu istovremeno da ga odrede. Po izračunavanju svog elementa matrice C, svaka nit ga šalje upravljačkoj niti.

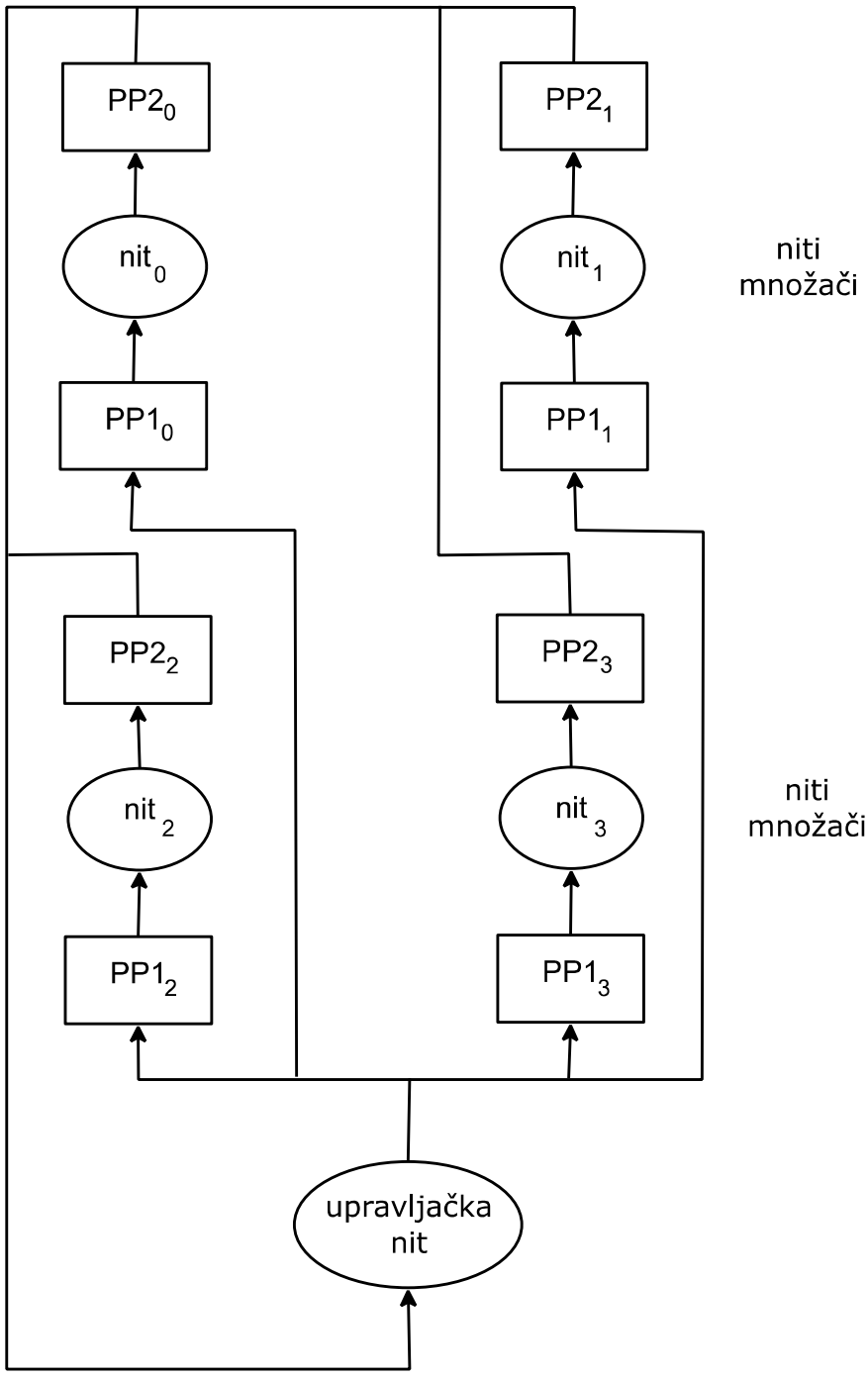
18.5.2 Komunikaciona osnova paralelnog množenja matrica

Vezivanje niti množača u matricu je inspirisano idejom dvodimenzionalne sistoličke arhitekture (*mesh*). Svaka nit množač iz ovog niza ima levo, desno, iznad i ispod sebe po dva pregratka za poruke (Slika 18.5).



Slika 18.5: Niti uvezane u matricu,Niti uvezane u matricu

Pregraci (Pij) levo uspostavljaju komunikacioni kanal od levog suseda i ka levom susedu iz matrice. Slično, pregraci desno uspostavljaju komunikacioni kanal od desnog suseda i ka desnom susedu iz matrice. Podrazumeva se da je nit množač sa levog kraja svakog reda matrice desni sused niti množača sa desnog kraja istog reda i obrnuto. Pregraci iznad uspostavljaju komunikacioni kanal od gornjeg suseda i ka gornjem susedu iz matrice. Slično, pregraci ispod uspostavljaju komunikacioni kanal ka donjem susedu i od donjeg suseda iz matrice. Podrazumeva se da je nit množač sa gornjeg kraja svake kolone matrice donji sused niti množača sa donjeg kraja iste kolone i obrnuto. Na ovaj način svaka nit množač može da razmenjuje poruke sa svojim horizontalnim i vertikalnim susedima. Podrazumeva se da uz svaku nit množač postoji poseban par pregradaka posredstvom koga niti množači mogu da direktno razmenjuju poruke sa upravljačkom niti (Slika 18.6).



Slika 18.6: Niti množači i upravljačka nit

Templejt klasa **Mesh** (Listing 18.6) omogućuje uspostavljanje komunikacionih

kanala koji su potrebni za uvezivanje niti u matricu. Njena tri parametra omogućuju definisanje tipa poruke i broja redova i kolona niti u matrici. Elementi polja **slots1**, **slots2**, **slots3** i **slots4** odgovaraju, respektivno, pregracima: **P1i**, **P2i**, **P3i** i **P4i**. Elementi polja **special1** i **special2** odgovaraju posebnim pregracima **PP1i**, **PP2i**, za direktnu komunikaciju između niti množača i upravljačke niti. Operacije **my_row()** i **my_column()** omogućuju nitima da dobiju redni broj svog reda i kolone iz matrice. Podrazumeva se da se kao oznake niti množača koriste redni brojevi: 0, 1, 2, 3 i tako dalje. Operacije **send()** i **receive()** omogućuje razmenu poruka. Prvi par ovih operacija omogućuje upravljačkoj niti da komunicira sa svakom od niti množača, za šta je neophodno korišćenje rednih brojeva njihovih redova i kolona. Drugi par ovih operacija omogućuje nitima množačima da komuniciraju sa svojim horizontalnim i vertikalnim susedima koje označavaju pomoću konstanti **UPPER**, **DOWN**, **LEFT** i **RIGHT**, kao i sa upravljačkom niti koju označavaju pomoću konstante **SPECIAL**.

Polja **rows** i **columns** pokazuju koliko je angažovano niti množača. Njihovo postavljanje omogućuju operacije **rows_set()** i **columns_set()**.

Listing 18.6: Klasa **Mesh** (datoteka **mesh.hh**)

```
#include "box.hh"

enum
Mesh_relative_position { UPPER, DOWN, LEFT, RIGHT, SPECIAL };

template<class MESSAGE, int ROWS, int COLUMNS>
class Mesh {
   Message_box<MESSAGE> slots1[ROWS * COLUMNS];
   Message_box<MESSAGE> slots2[ROWS * COLUMNS];
   Message_box<MESSAGE> slots3[ROWS * COLUMNS];
   Message_box<MESSAGE> slots4[ROWS * COLUMNS];
   Message_box<MESSAGE> special1[ROWS * COLUMNS];
   Message_box<MESSAGE> special2[ROWS * COLUMNS];
   int rows;
   int columns;
  public:
   Mesh() : rows(ROWS), columns(COLUMNS) {};
   void send(int row, int column, const MESSAGE* message);
   MESSAGE receive(int row, int column);
   void rows_set(unsigned r) { rows = r; };
   void columns_set(unsigned c) { columns = c; };
   void send(int sender, Mesh_relative_position relative_position,
         const MESSAGE* message);
   MESSAGE receive(int receiver, Mesh_relative_position relative_position);
   int my_row(int thread_identity);
   int my_column(int thread_identity);
};

template<class MESSAGE, int ROWS, int COLUMNS>
void
Mesh<MESSAGE, ROWS, COLUMNS>::send(int row, int column,
                                   const MESSAGE* message)
{
   int index = (row - 1) * columns + column - 1;
   special1[index].send(message);
}
```

```
template<class MESSAGE, int ROWS, int COLUMNS>
MESSAGE
Mesh<MESSAGE, ROWS, COLUMNS>::receive(int row, int column)
{
   int index = (row - 1) * columns + column - 1;
   return special2[index].receive();
}

template<class MESSAGE, int ROWS, int COLUMNS>
void
Mesh<MESSAGE, ROWS, COLUMNS>::send(int sender,
                        Mesh_relative_position relative_position,
                        const MESSAGE* message)
{
   Message_box<MESSAGE>* destination;
   switch(relative_position) {
    case UPPER:
       destination = &(slots2[sender]);
       break;
    case DOWN:
       sender += columns ;
       if(sender >= rows * columns)
          sender -= rows * columns;
       destination = &(slots1[sender]);
       break;
    case LEFT:
       destination = &(slots4[sender]);
       break;
    case RIGHT:
       if(((sender+1) % columns) == 0)
          sender -= columns;
       destination = &(slots3[sender+1]);
       break;
    case SPECIAL:
       destination = &(special2[sender]);
       break;
   }
   destination->send(message);
}
```

```
template<class MESSAGE, int ROWS, int COLUMNS>
MESSAGE
Mesh<MESSAGE, ROWS, COLUMNS>::receive(int receiver,
                        Mesh_relative_position relative_position)
{
   Message_box<MESSAGE>* source;
   switch(relative_position) {
     case UPPER:
        source = &(slots1[receiver]);
        break;
     case DOWN:
        receiver += columns ;
        if(receiver >= rows * columns)
           receiver -= rows * columns;
        source = &(slots2[receiver]);
        break;
     case LEFT:
        source = &(slots3[receiver]);
        break;
     case RIGHT:
        if(((receiver+1) % columns) == 0)
           receiver -= columns;
        source = &(slots4[receiver+1]);
        break;
     case SPECIAL:
        source = &(special1[receiver]);
        break;
   }
   return source->receive();
}

template<class MESSAGE, int ROWS, int COLUMNS>
int
Mesh<MESSAGE, ROWS, COLUMNS>::my_row(int thread_identity)
{
   return(thread_identity  / columns + 1);
}
```

```
template<class MESSAGE, int ROWS, int COLUMNS>
int
Mesh<MESSAGE, ROWS, COLUMNS>::my_column(int thread_identity)
{
   return(thread_identity  % columns + 1);
}
```

18.5.3 Izvedba paralelnog množenja matrica

Ponašanje niti množača opisuje funkcija **thread_multiplier()** (Listing 18.7). Ona opisuje inicijalno raspodeljivanje elemenata matrica A i B, kao i izračunavanje pojedinih elemenata matrice C.

Funkcija **input_matrix()** omogućuje preuzimanje elemenata matrica A i B sa tastature i njihovo upućivanje pojedinim nitima uvezanim u matricu. Operacija **thread_manager()** opisuje aktivnost upravljačke niti, koja stvara niti množače, isporučuje im odgovarajuće elemente matrica A i B, preuzima elemente matrice C i prikazuje ih.

Listing 18.7: Paralelno množenje matrica (datoteka **p08.cpp**)

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include "mesh.hh"
#include "thread_identity.hh"

const int
SQUARE_MATRIX_ORDER = 2;

const int
THREAD_ROWS = SQUARE_MATRIX_ORDER;

const int
THREAD_COLUMNS = SQUARE_MATRIX_ORDER;

Mesh<int, THREAD_ROWS, THREAD_COLUMNS>
mesh;
```

```
Thread_identity
thread_identity;

void
thread_multiplier()
{
   int counter;
   int matrix_a_element;
   int matrix_b_element;
   int matrix_c_element;
   int tid = thread_identity.get();
   matrix_a_element = mesh.receive(tid, SPECIAL);
   matrix_b_element = mesh.receive(tid, SPECIAL);
   counter = (THREAD_ROWS - mesh.my_row(tid) + 1) % THREAD_ROWS;
   while((counter--) > 0) {
      mesh.send(tid, RIGHT, &matrix_a_element);
      matrix_a_element = mesh.receive(tid, LEFT);
   }
   counter =(THREAD_COLUMNS - mesh.my_column(tid) + 1) % THREAD_COLUMNS;
   while((counter--) > 0){
      mesh.send(tid, DOWN, &matrix_b_element);
      matrix_b_element = mesh.receive(tid, UPPER);
   }
   counter = SQUARE_MATRIX_ORDER;
   matrix_c_element = 0;
   while((counter--) > 0) {
      matrix_c_element += matrix_a_element * matrix_b_element;
      mesh.send(tid, RIGHT, &matrix_a_element);
      mesh.send(tid, DOWN, &matrix_b_element);
      matrix_a_element = mesh.receive(tid, LEFT);
      matrix_b_element = mesh.receive(tid, UPPER);
   }
   mesh.send(tid, SPECIAL, &matrix_c_element);
}
```

```
void
input_matrix(char name)
{
   int matrix_element;
   cout << endl << "matrix " << name
       << " input (elements values from -100 to 100)";
   for(int row = 1; row <= THREAD_ROWS; row++)
      for(int column = 1; column <= THREAD_COLUMNS; column++) {
         do {
            cout << endl << name << '('
                << (short)row << ',' << (short)column << "):";
            cin >> matrix_element;
         }
         while((matrix_element < -100) || (matrix_element > 100));
         mesh.send(row, column, &matrix_element);
      }
}

void
thread_manager()
{
   int row;
   int column;
   cout << endl << "PARALLEL MATRIX-BY-MATRIX MULTIPLICATION";
   for(row = 1; row <= THREAD_ROWS; row++)
      for(column = 1; column <= THREAD_COLUMNS; column++) {
         thread (thread_multiplier).detach();
      }
   input_matrix('A');
   input_matrix('B');
   cout << endl << "matrix C=A*B";
   for(row = 1; row <= THREAD_ROWS; row++) {
      cout << endl;
      for(column = 1; column <= THREAD_COLUMNS; column++) {
         cout << mesh.receive(row, column) << " ";
      }
   }
   cout << endl;
}
```

```
int
main()
{
   thread manager(thread_manager);
   manager.join();
}
```

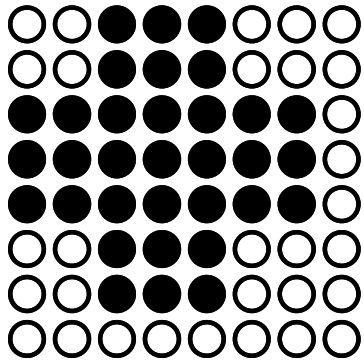
Sadržaj izvorne datoteke **p08.cpp** (Listing 18.7) predstavlja potpun konkurentni program. U toku njegovog izvršavanja nastane pet niti.

Računanje svih elemenata matrice C, opisano u prethodnom programu, traje koliko i računanje samo jednog od ovih elemenata, ako se svakoj od niti, zaduženih za računanje po jednog elementa matrice C, dodeli poseban procesor. Ali, ako se prethodni program izvršava na jednoprocesorskom računaru, tada je dužina njegovog izvršavanja proporcionalna broju elemenata matrice C.

18.6 PARALELNO IZDVAJANJE KONTURE

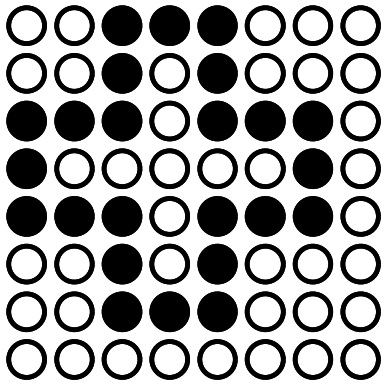
18.6.1 Zamisao paralelnog izdvajanja konture

Liku, predstavljenom crnim tačkama na crno–beloj slici (Slika 18.7):



Slika 18.7: Lik

odgovara kontura koju obrazuju crne tačke, u čijem susedstvu je bar jedna bela tačka (Slika 18.8):



Slika 18.8: Kontura

Izdvajanje konture ovakvog lika se zasniva na uparivanju boje svake tačke lika sa bojom tačaka iz njenog susedstva. Podrazumeva se da su svakoj tački **p** susedne tačke koje su za nju iznad (*up*: **u**), iznad desno (*up right*: **ur**), iznad levo (*up left*: **ul**), ispod (*down*: **d**), ispod desno (*down right*: **dr**), ispod levo (*down left*: **dl**), desno (*right*: **r**) i levo (*left*: **l**). Ovo važi i za rubne tačke slike, jer se smatra da su suprotne ivice slike spojene. Ako crnoj tački odgovara vrednost 1, a beloj vrednost 0, tada iskaz:

```
c = p&(~u)|p&(~ur)|p&(~ul)|p&(~d)|p&(~dr)|p&(~dl)|p&(~r)|p&(~l);
```

opisuje proveru da li je tačka **p** konturna.

Za razne tačke, uparivanja njihovih boja su nezavisna, pa se mogu odvijati paralelno. One, znači, mogu biti predmet aktivnosti raznih niti. Pri tome, svaka od ovih **konturnih niti** proverava za po jednu tačku slike da li dotična tačka pripada konturi lika. Podrazumeva se da su konturne niti raspoređene u prostoru kao i tačke koje su im dodeljene, znači uvezane u matricu. Za susedne konturne niti je važno da budu komunikaciono povezane, radi razmena boja tačaka koje su im dodeljene. Dovoljno je da svaka konturna nit bude komunikaciono povezana sa konturnim nitima iznad, ispod, levo i desno od sebe, jer tada, posredstvom svojih vertikalnih i horizontalnih suseda, ona može razmeniti boje tačaka i sa svojim dijagonalnim susedima.

Podrazumeva se da posebna upravljačka nit saopštava konturnim nitima boju njihove tačke i od njih prima podatak da li je njihova tačka konturna.

18.6.2 Komunikaciona osnova paralelnog izdvajanja konture

Templejt klasa **Mesh** (Listing 18.6) omogućuje komunikaciono povezivanje konturnih niti između sebe, kao i njihovu direktnu komunikaciju sa upravljačkom niti.

18.6.3 Izvedba paralelnog izdvajanja konture

Funkcija **thread_contour()** (Listing 18.8) opisuje aktivnost konturnih niti. Svaka od njih raspolaže bojom svoje tačke (**my_pixel**), preuzima boju susednih tačaka

(**foreign_pixel**) i izračunava da li je njena tačka konturna (**contour_pixel**). Stvaranje ovih niti (korišćenjem bezimenih objekata klase **thread**), uvezanih u matricu, je u nadležnosti upravljačke niti, čiju aktivnost opisuje operacija **thread_manager()**. Upravljačka nit šalje konturnim nitima tačke slike (**picture**) i preuzima i prikazuje konturu.

Listing 18.8: Paralelno izdvajanje konture (datoteka **p09.cpp**)

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include "mesh.hh"
#include "thread_identity.hh"

const int
SQUARE_MATRIX_ORDER = 8;

const int
THREAD_ROWS = SQUARE_MATRIX_ORDER;

const int
THREAD_COLUMNS = SQUARE_MATRIX_ORDER;

Mesh<char, THREAD_ROWS, THREAD_COLUMNS>
mesh;

Thread_identity
thread_identity;
```

```
void
thread_contour()
{
   char my_pixel;
   char foreign_pixel;
   char contour_pixel;
   int tid = thread_identity.get();
   my_pixel = mesh.receive(tid, SPECIAL);
   mesh.send(tid, DOWN, &my_pixel);
   contour_pixel = my_pixel &(~(foreign_pixel = mesh.receive(tid, UPPER)));
   mesh.send(tid, LEFT, &foreign_pixel);
   contour_pixel |= my_pixel &(~(mesh.receive(tid, RIGHT)));
   mesh.send(tid, RIGHT, &foreign_pixel);
   contour_pixel |= my_pixel &(~(mesh.receive(tid, LEFT)));
   mesh.send(tid, UPPER, &my_pixel);
   contour_pixel |= my_pixel &(~(foreign_pixel = mesh.receive(tid, DOWN)));
   mesh.send(tid, LEFT, &foreign_pixel);
   contour_pixel |= my_pixel &(~(mesh.receive(tid, RIGHT)));
   mesh.send(tid, RIGHT, &foreign_pixel);
   contour_pixel |= my_pixel &(~(mesh.receive(tid, LEFT)));
   mesh.send(tid, LEFT, &my_pixel);
   contour_pixel |= my_pixel &(~(mesh.receive(tid, RIGHT)));
   mesh.send(tid, RIGHT, &my_pixel);
   contour_pixel |= my_pixel &(~(mesh.receive(tid, LEFT)));
   mesh.send(tid, SPECIAL, &contour_pixel);
}

const char
picture[THREAD_ROWS][THREAD_COLUMNS] = { {0, 0, 1, 1, 1, 0, 0, 0},
                                         {0, 0, 1, 1, 1, 0, 0, 0},
                                         {1, 1, 1, 1, 1, 1, 1, 0},
                                         {1, 1, 1, 1, 1, 1, 1, 0},
                                         {1, 1, 1, 1, 1, 1, 1, 0},
                                         {0, 0, 1, 1, 1, 0, 0, 0},
                                         {0, 0, 1, 1, 1, 0, 0, 0},
                                         {0, 0, 0, 0, 0, 0, 0, 0}
};
```

```
void
thread_manager()
{
   int row;
   int column;
   cout << endl << "PARALLEL CONTOUR FINDING";
   for(row = 1; row <= THREAD_ROWS; row++) {
      cout << endl;
      for(column = 1; column <= THREAD_COLUMNS; column++) {
         cout << ((picture[row - 1][column - 1] == 0) ? ' ' : '.');
         thread (thread_contour).detach();
         mesh.send(row, column, &picture[row - 1][column - 1]);
      }
   }
   for(row = 1; row <= THREAD_ROWS; row++) {
      cout << endl;
      for(column = 1; column <= THREAD_COLUMNS; column++)
         cout << ((mesh.receive(row, column) == 0) ? ' ' : '.');
   }
}

int
main()
{
   thread manager(thread_manager);
   manager.join();
}
```

Sadržaj izvorne datoteke **p09.cpp** (Listing 18.8) predstavlja potpun konkurentni program. U toku njegovog izvršavanja nastane 65 niti.

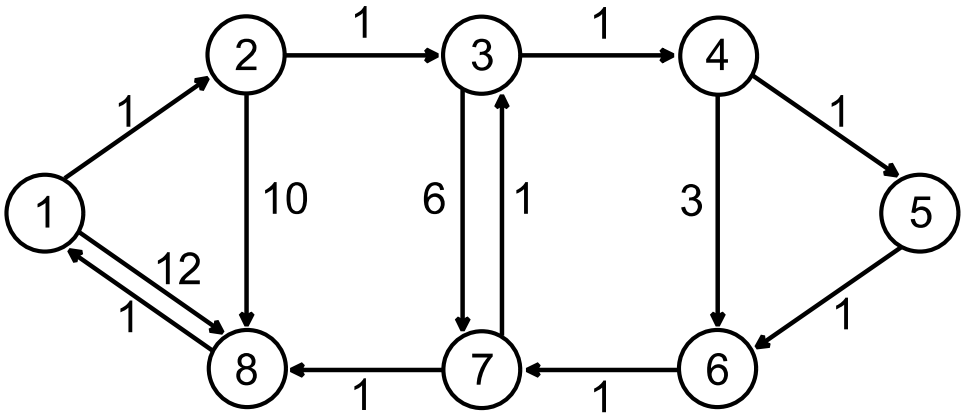
Izdvajanje konture, opisano u prethodnom programu, traje koliko i provera za jednu tačku da li je konturna, ako se svakoj od niti, zaduženoj za po jednu ovakvu proveru, dodeli poseban procesor. Ali, ako prethodni program izvršava jednoprocesorski računar, tada je vreme, potrebno za izdvajanje konture, proporcionalno broju tačaka slike.

Pristup, primenjen u prethodnom programu za izdvajanje konture, se može primeniti i za čišćenje slike od posledica šuma (koji se javlja pri telekomunikacionom prenosu slike). Ovaj šum izaziva izmenu boja pojedinih tačaka, pa se javljaju crne tačke na beloj pozadini i obrnuto. Čišćenje šuma podrazumeva izmenu boje svake tačke koja je okružena tačkama suprotne boje.

18.7 PARALELNO ODREĐIVANJE NAJKRAĆIH MEĐUSOBNIH UDALJENOSTI ČVOROVA USMERENOG GRAFA

18.7.1 Zamisao paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

Usmereni graf se sastoji od numerisanih čvorova, povezanih usmerenim spojnicama raznih nenultih dužina (Slika 18.9).



Slika 18.9: Usmereni graf

Određivanje najkraće udaljenosti čvorova usmerenog grafa od zadanog čvora se svodi na sabiranje dužina spojnica, koje vode od zadanog do pojedinih od preostalih čvorova. Za svaki od preostalih čvorova udaljenost se određuje relativno u odnosu na prethodnika, sabiranjem udaljenosti prethodnika i dužine spojnice koja dolazi od prethodnika. Ako od zadanog do nekog čvora vodi **k** spojnica, odnosno, ako se između ovih čvorova nalazi **k-1** drugih čvorova, tada se udaljenost posmatranog čvora od zadanog čvora može odrediti tek nakon **k** koraka, pošto se za svaki od **k-1** čvorova, koji prethode pomenutom čvoru, odrede njihove udaljenosti od zadanog čvora. Kada od zadanog do nekog od preostalih čvorova vodi više puteva, koji se razlikuju bar po jednoj spojnici, tada udaljenost pomenutog čvora od zadanog čvora, određena u nekom od koraka, može biti izmenjena u nekom od narednih koraka, ako je put preko više spojnica kraći.

Određivanja najkraće udaljenosti čvorova koji ne leže na istom putu, posmatrano od zadanog čvora, su međusobno nezavisna, pa se mogu odvijati paralelno. Ona, znači, mogu biti predmet aktivnosti raznih niti. Pri tome se svakoj od ovih **čvornih niti** dodeljuje po jedan čvor, a čvorne niti su raspoređene u prostoru kao čvorovi koji su im dodeljeni. Za čvorne niti je važno da budu međusobno komunikaciono povezane kao i pridruženi im čvorovi. To je potrebno da bi čvorne niti mogle međusobno sarađivati, radi slanja sledbenicima njihovih udaljenosti i radi prijema svojih udaljenosti od prethodnika. U opštem slučaju ovo podrazumeva potpuno međusobno povezivanje čvornih niti.

Na početku određivanja udaljenosti, zadanom čvoru se dodeli udaljenost 0, a svim

ostalim čvorovima beskonačna udaljenost. Saradnja čvornih niti se odvija u koracima. Svaki od njih obuhvata dve faze. U prvoj fazi, svaka čvorna nit šalje svojim sledbenicima njihovu udaljenost, koja je jednaka ili izračunatoj ili vrednosti beskonačno, zavisno od toga da li je čvorna nit primila ili ne svoju udaljenost od prethodnika. U drugoj fazi, svaka čvorna nit prima od svojih prethodnika svoju udaljenost. Primljena udaljenost zamenjuje postojeću samo ako je manja od nje. Nakon svakog koraka sledi provera da li se izmenila ijedna od udaljenosti. Prvi korak, u kome izostanu izmene određivanih udaljenosti, označava kraj postupka, jer su tada određene najkraće udaljenosti čvorova usmerenog grafa od zadanog čvora, pa nove izmene nisu moguće.

Provera kraja postupka nastupa kada sve čvorne niti završe započeti korak. Prelazak čvorne niti na novi korak ima smisla tek nakon što se u pomenutoj proveri ustanovi da postupak nije završen. Zato aktivnosti čvornih niti moraju biti usklađene sa pomenutom proverom. Ovakvo usklađivanje aktivnosti niti, u kome, pre prelaska ijedne niti na novi korak, sve niti moraju završiti započeti korak, se naziva **barijerna sinhronizacija** (*barrier synchronization*). Barijerna sinhronizacija se uspostavlja zahvaljujući posredovanju upravljačke niti. Radi ostvarenja barijerne sinhronizacije, svaka od čvornih niti, po završetku započetog koraka, šalje upravljačkoj niti, u okviru jedne poruke, broj svoga čvora i svoju važeću udaljenost. Nastavak aktivnosti čvornih niti zavisi od broja izmenjenih udaljenosti, koji je primila upravljačka nit. Ako je ovaj broj 0, tada niti završavaju svoju aktivnost, jer je postupak završen.

18.7.2 Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

Pravilan karakter saradnje niti opravdava korišćenje posebnog pregratka za svaki smer razmene poruka između bilo koje dve komunikaciono povezane niti. Ali, ako je saradnja niti sporadična, kao u slučaju paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa, tada postoji neizvesnost posredstvom kog pregratka dolazi poruka, pa je moguće da nit očekuje poruku iz jednog pregratka, a da poruka u međuvremenu pristigne u drugi pregradak. Zato je bolje, u situaciji kao što je prethodna, da svaka nit prima sve poruke posredstvom samo jednog pregratka. Pri tome je važno da pregradak sadrži više odeljaka da bi mogao da primi sve poruke, koje su upućene niti kojoj je pregradak dodeljen. U suprotnom slučaju, neizbežna je mrtva petlja, ako se svi pregraci napune, a sve niti nastave sa slanjem poruka.

Pregratke sa više odeljaka opisuje nova verzija templejt klase **Message_box** (Listing 18.9). Njeno polje **slots** sadrži **SLOTS** odeljaka. Svaki od njih prima po jednu poruku. Polja **first_empty_slot** i **first_full_slot** ukazuju na prvi prazan, odnosno na prvi pun odeljak. Njihove vrednosti se menjaju po modulo aritmetici, zbog ciklične organizacije odeljaka. Polje **full_slot_number** sadrži broj napunjenih pregradaka. Polja **not_full** i **not_empty** određuju uslove, od čijeg ispunjenja zavise aktivnosti niti primalaca i pošiljalaca poruka. Operacije **send()** i **receive()** omogućuju slanje i prijem poruka.

Listing 18.9: Klasa **Message_box** (datoteka **slot_box.hh**)

```
template<class MESSAGE, int SLOTS>
class Slot_box {
   mutex mx;
   MESSAGE slots[SLOTS];
   unsigned int first_empty_slot;
   unsigned int first_full_slot;
   unsigned int full_slot_number;
   condition_variable not_full;
   condition_variable not_empty;
 public:
   Slot_box() : first_empty_slot(0), first_full_slot(0),
                full_slot_number(0) {};
   void send(const MESSAGE* message);
   MESSAGE receive();
};

template<class MESSAGE, int SLOTS>
void
Slot_box<MESSAGE, SLOTS>::send(const MESSAGE* message)
{
   unique_lock<mutex> lock(mx);
   while(full_slot_number == SLOTS)
      not_full.wait(lock);
   slots[first_empty_slot] = *message;
   full_slot_number++;
   if(++first_empty_slot == SLOTS)
      first_empty_slot = 0;
   not_empty.notify_one();
}
```

```
template<class MESSAGE, int SLOTS>
MESSAGE
Slot_box<MESSAGE, SLOTS>::receive()
{
   unsigned int return_index;
   unique_lock<mutex> lock(mx);
   while(full_slot_number == 0)
      not_empty.wait(lock);
   return_index = first_full_slot;
   full_slot_number--;
   if(++first_full_slot == SLOTS)
      first_full_slot = 0;
   not_full.notify_one();
   return slots[return_index];
}
```

Paralelno određivanje najkraćih međusobnih udaljenosti čvorova usmerenog grafa zahteva mogućnost direktne razmene poruka između svih čvornih niti. U tom slučaju, potrebno je komunikaciono potpuno međusobno povezati sve čvorne niti. To se ostvaruje ako se uz svaku čvornu nit veže pregradak sa više odeljaka i ako se svim čvornim nitima dozvoli da šalju poruke u pregratke preostalih čvornih niti.

Templejt klasa **Any2any** (Listing 18.10) omogućuje uspostavljanje komunikacionih kanala koji su potrebni za potpuno međusobno povezivanje čvornih niti. Njena tri parametra omogućuju definisanje tipa poruke, broja pregradaka i broja odeljaka u svakom pregratku. Pojedinim pregracima odgovaraju elementi polja **boxes**. Operacije **send()** i **receive()** omogućuju razmenu poruka. Podrazumeva se da pregracima i oznakama čvornih niti odgovaraju isti redni brojevi: 1, 2, 3 i tako dalje. Takođe se podrazumeva da pregradak sa rednim brojem 0 (**boxes[0]**) odgovara upravljačkoj niti.

Listing 18.10: Klasa **Any2any** (datoteka **any2any.hh**)

```
#include "slot_box.hh"

template<class MESSAGE, int NODES, int SLOTS>
class Any2any {
   Slot_box<MESSAGE, SLOTS> boxes[NODES + 1];
  public:
   Any2any() {};
   void send(int node, const MESSAGE* message);
   MESSAGE receive(int node);
};
```

```
template<class MESSAGE, int NODES, int SLOTS>
void
Any2any<MESSAGE, NODES, SLOTS>::send(int node, const MESSAGE* message)
{
  boxes[node].send(message);
}

template<class MESSAGE, int NODES, int SLOTS>
MESSAGE
Any2any<MESSAGE, NODES, SLOTS>::receive(int node)
{
  return boxes[node].receive();
}
```

18.7.3 Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

U posmatranom primeru graf ima 8 čvorova i predstavljen je nizom **graph** (Listing 18.11). Njegovih prvih 8 elemenata sadrži dužine spojnica usmerenih od prvog čvora prema svim čvorovima, njegovih drugih 8 elemenata sadrži dužine spojnica usmerenih od drugog čvora prema svim čvorovima i tako dalje. Promenljiva **barrier** omogućuje ostvarenja barijerne sinhronizacije. Konstanta **TOP** simbolizuje beskonačnu udaljenost.

Funkcija **thread_node()** opisuje ponašanje čvornih niti. Njena lokalna promenljiva **successors** omogućuje smeštanje broja sledbenika čvora, lokalna promenljiva **links** omogućuje smeštanje oznaka sledbenika i udaljenosti čvora od svojih sledbenika, lokalna promenljiva **predecessors** omogućuje smeštanje broja prethodnika čvora, lokalna promenljiva **message** omogućuje smeštanje poruke koja se šalje sledbenicima čvora, lokalna promenljiva **new_distance** omogućuje smeštanje nove najmanje udaljenosti čvora od zadatog čvora i lokalna promenljiva **distance** omogućuje smeštanje aktuelne najmanje udaljnosti čvora od zadanog čvora.

Funkcija **thread_manager()** opisuje ponašanje upravljačke niti. Ona na početku, svim čvornim nitima šalje njihovu početnu udaljenost, broj sledbenika, oznake i udaljenosti tih sledbenika, kao i broj prethodnika, a zatim stvori i pokrene pomenute čvorne niti (korišćenjem bezimenih objekata klase **thread**), da bi nakon toga pratila izmene udaljenosti i na osnovu toga utvrdila kada su određene najkraće udaljenosti. Ova funkcija sadrži lokalnu promenljivu **distances** za smeštanje trenutnih udaljenosti svih čvorova od zadatog čvora, kao i lokalnu promenljivu **change_counter** u kojoj se čuva broj izmena udaljenosti u svakom koraku.

Listing 18.11: Paralelno određivanje najkraćih međusobnih udaljenosti čvorova usmerenog grafa (datoteka **p10.cpp**)

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include "box.hh"
#include "any2any.hh"
#include "thread_identity.hh"

struct Package {
   int node;
   int value;
};

const int
THREAD_NODES = 8;

const int
BOX_SLOTS = THREAD_NODES * 2;

const int
TOP = 127;

Message_box<int>
barrier[THREAD_NODES];

Any2any<Package, THREAD_NODES, BOX_SLOTS>
any2any;

Thread_identity
thread_identity(1);
```

```
void
thread_node()
{
   int successors;
   int i;
   Package links[THREAD_NODES - 1];
   int predecessors;
   Package message;
   int new_distance;
   int distance;
   int tid = thread_identity.get();
   distance = any2any.receive(tid).value;
   successors = any2any.receive(tid).value;
   for(i = 0; i < successors; i++)
      links[i] = any2any.receive(tid);
   predecessors = any2any.receive(tid).value;
   message.node = tid;
   do {
      for(i = 0; i < successors; i++) {
         message.value = (distance < TOP)
            ? (distance + links[i].value) : (TOP);
         any2any.send(links[i].node, &message);
      }
      for(i = 0; i < predecessors; i++) {
         new_distance = any2any.receive(tid).value;
         if(distance > new_distance)
            distance = new_distance;
      }
      message.value = distance;
      any2any.send(0, &message);
   }
   while(barrier[tid-1].receive() > 0);
}
```

```
const short
graph[THREAD_NODES][THREAD_NODES] = { {0, 1, 0, 0, 0, 0, 0, 12},
                                      {0, 0, 1, 0, 0, 0, 0, 10},
                                      {0, 0, 0, 1, 0, 0, 6, 0},
                                      {0, 0, 0, 0, 1, 3, 0, 0},
                                      {0, 0, 0, 0, 0, 1, 0, 0},
                                      {0, 0, 0, 0, 0, 0, 1, 0},
                                      {0, 0, 1, 0, 0, 0, 0, 1},
                                      {1, 0, 0, 0, 0, 0, 0, 0}
};

void
thread_manager()
{
   int i;
   short distances[THREAD_NODES];
   int j;
   Package message;
   int change_counter;
   cout << endl << "PARALLEL FINDING OF SHORTEST PATH IN "
      << "DIRECTED AND WEIGHTED GRAPH" << endl << "WEIGHT MATRIX:";
   for(i = 0; i < THREAD_NODES; i++) {
      message.node = 0;
      message.value = ((i == 0) ? 0 : TOP);
      any2any.send(i + 1, &message);
   }
```

```
for(i = 0; i < THREAD_NODES; i++) {
   distances[i] = 0;
   message.node = 0;
   message.value = 0;
   cout << endl;
   for(j = 0; j < THREAD_NODES; j++) {
      cout << graph[i][j] << ' ';
      if(graph[i][j] > 0)
         message.value++;
   }
   any2any.send(i + 1, &message);
   for(j = 0; j < THREAD_NODES; j++)
      if(graph[i][j] > 0) {
         message.node = j + 1;
         message.value = graph[i][j];
         any2any.send(i + 1, &message);
      }
}
for(i = 0; i < THREAD_NODES; i++) {
   message.node = 0;
   message.value = 0;
   for(j = 0; j < THREAD_NODES; j++)
      if(graph[j][i] > 0)
         message.value++;
   any2any.send(i + 1, &message);
}
for(i = 0; i < THREAD_NODES; i++) {
   thread (thread_node).detach();
}
cout << endl << "DISTANCES FROM NODE 1 TO" << endl;
```

```
   do {
      change_counter = 0;
      for(i = 0; i < THREAD_NODES; i++) {
         message = any2any.receive(0);
         if(distances[message.node - 1] != message.value) {
            change_counter++;
            distances[message.node - 1] = message.value;
         }
      }
      if(change_counter == 0)
         for(i = 1; i < THREAD_NODES; i++) {
            cout << "NODE " << i+1 << ": ";
            if(distances[i] == TOP)
               cout << " ";
            else
               cout << distances[i] << endl;
         }
      message.node = 0;
      message.value = change_counter;
      for(i = 0; i < THREAD_NODES; i++)
         barrier[i].send(&change_counter);
   }
   while(change_counter > 0);
}

int
main()
{
   thread manager(thread_manager);
   manager.join();
}
```

Sadržaj izvorne datoteke **p10.cpp** (Listing 18.11) predstavlja potpun konkurentni program. U toku njegovog izvršavanja nastane 9 niti.

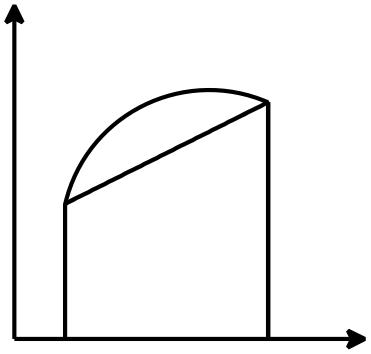
Paralelno određivanje najkraće udaljenosti čvorova usmerenog grafa od zadanog čvora, opisano u prethodnom programu, je, u najgorem slučaju, proporcionalno proizvodu najvećeg broja spojnica, koje dolaze u neki čvor i odlaze iz njega, i najvećeg broja koraka, ako se svakoj od čvornih niti dodeli poseban procesor. Ali, ako prethodni program izvršava jednoprocesorski računar, tada je vreme, potrebno za određivanje pomenutih udaljenosti, proporcionalno proizvodu najvećeg broja spojnica, koje dolaze u neki čvor i odlaze iz njega, najvećeg broja koraka i ukupnog broja čvorova.

Pristup primenjen za paralelno određivanje najkraće udaljenosti čvorova usmerenog grafa od zadanog čvora predstavlja prirodnu osnovu za paralelno određivanje najkraće udaljenosti svih parova čvorova usmerenog grafa. Potrebno je samo uočiti da se određivanje najkraće udaljenosti svih parova čvorova usmerenog grafa može razložiti na nezavisna određivanja najkraćih udaljenosti njegovih čvorova od različitih zadanih čvorova.

18.8 PARALELNO IZRAČUNAVANJE POVRŠINE ISPOD POLUKRUGA

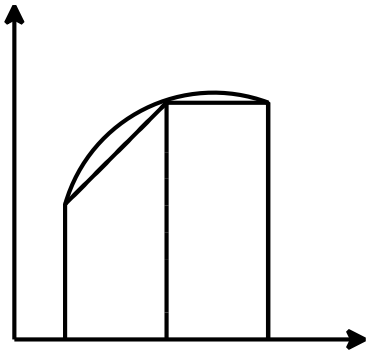
18.8.1 Zamisao paralelnog izračunavanja površine ispod polukruga

Površina ispod grafa funkcije na intervalu, na kome je ona neprekidna, nenegativna i neperiodična, se određuje približno, kao površina trapeza, čija visina se podudara sa posmatranim intervalom, a čije baze su jednake vrednostima funkcije u krajnjim tačkama ovog intervala (Slika 18.10).



Slika 18.10: Površina na intervalu

Površina ovog trapeza predstavlja traženu površinu, ako je dovoljno bliska sumi površina trapeza, konstruisanih na isti način iznad leve i desne polovine posmatranog intervala (Slika 18.11).



Slika 18.11: Površina na dva podintervala

Ako površina prvog trapeza nije bliska sumi površina druga dva trapeza, tada se za svaku od polovina posmatranog intervala na prethodni način određuje površina ispod grafa krive. Postupak se završava tek kada se za svaki od podintervala odredi površina. Suma površina svih podintervala daje traženu površinu.

Određivanja površina za pojedine podintervale su međusobno nezavisna, pa mogu da budu predmet aktivnosti raznih **niti integratora**. Svaka od njih, pri podeli [pod]intervala na dve polovine, odlaže podatke o jednoj polovini u skladište, a zatim nastavlja da se bavi drugom polovinom. Podatke o [pod]intervalima iz skladišta preuzimaju besposlene niti integratori. Ako na početku određivanja tražene površine upravljačka nit u skladište smesti podatke o posmatranom intervalu, tada je skladište prazno po uspešnom određivanju tražene površine, pa poslednja aktivna nit integrator, nakon neuspešnog preuzimanja podataka iz praznog skladišta, obaveštava upravljačku nit da je tražena površina određena. S druge strane, nakon neuspešnog pokušaja da odloži podatke u puno skladište, poslednja aktivna nit integrator obaveštava upravljačku nit da tražena površina nije određena.

18.8.2 Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

Rukovanje skladištem opisuje templejt klasa **Pool** (Listing 18.12). Njena tri parametra omogućuju saopštavanje tipa podataka koji se odlažu u skladište, broja niti integratora koje odlažu podatke u skladište (**CLIENTS**) i broja odeljaka za ove podatke po svakoj od ovih niti. Ova templejt klasa sadrži i polja **not_full** i **not_empty**, koja određuju uslove od čije ispunjenosti zavisi aktivnost niti integratora koje odlažu, odnosno preuzimaju podatke. Aktivnost svake od njih se zaustavlja: pri odlaganju podataka, ako je skladište puno, i pri preuzimanju podataka, ako je skladište prazno. Ovakvo neselektivno zaustavljanje aktivnosti uzrokuje mrtvu petlju, ako svaka od niti integratora proba: da odloži podatke u puno skladište, odnosno, da preuzme podatke iz praznog skladišta. Zato, radi sprečavanja ovakve mrtve petlje, templejt klasa **Pool** sadrži polja **not_full_await_count**, odnosno **not_empty_await_count**, sa brojem niti integratora zaustavljenih pri odlaganju, odnosno, pri preuzimanju podataka. Na osnovu poređenja

vrednosti parametra **CLIENTS** sa poljem **not_full_await_count**, odnosno sa poljem **not_empty_await_count**, moguće je ustanoviti kada poslednja nit integrator pokuša da odloži podatke u puno skladište, odnosno da preuzme podatke iz praznog skladišta. Tada se poslednjoj niti integratoru, umesto zaustavljanja njene aktivnosti, vraća vrednost **false**, koja označava neuspešno odlaganje, odnosno, neuspešno preuzimanje podataka. Preostale niti integratori se bude uz vraćanje vrednosti **false**, koja takođe označava neuspešno odlaganje, odnosno, neuspešno preuzimanje podataka. To omogućuju polja **end**, **pool_ending_state** i **exit**, kao i operacija **blocked()**, namenjena upravljačkoj niti.

Broj slobodnih odeljaka u skladištu za prijem podataka sadrži polje **free_slots_count** templejt klase **Pool**. Podrazumeva se da su ovi odeljci kružno raspoređeni (tako da iza poslednjeg odeljka sledi prvi). Indekse prvog praznog, odnosno prvog punog odeljka sadrže polja **first_empty_slot**, odnosno **first_full_slot**.

Odlaganje podataka u skladište omogućuje operacija **insert()**, a preuzimanje podataka iz skladišta operacija **extract()**.

Listing 18.12: Klasa **Pool** (datoteka **pool.hh**)

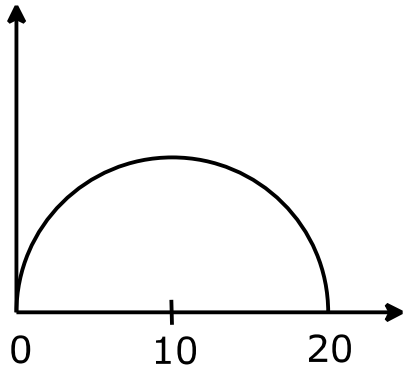
```
enum
Pool_ending_states { NOT_END, EMPTY, FULL };

template<class ITEM, int CLIENTS, int SLOTS_PER_CLIENT>
class Pool {
   mutex mx;
   ITEM slots[CLIENTS * SLOTS_PER_CLIENT];
   int free_slots_count;
   int first_empty_slot;
   int first_full_slot;
   condition_variable not_full;
   condition_variable not_empty;
   int not_full_wait_count;
   int not_empty_wait_count;
   condition_variable end;
   Pool_ending_states pool_ending_state;
   bool exit;
 public:
   Pool() : free_slots_count(CLIENTS * SLOTS_PER_CLIENT),
         first_empty_slot(0), first_full_slot(0),
         not_full_wait_count(0), not_empty_wait_count(0),
         pool_ending_state (NOT_END), exit(false) {};
   Pool_ending_states blocked();
   bool insert(ITEM* item);
   bool extract(ITEM * item);
};

template<class ITEM, int CLIENTS, int SLOTS_PER_CLIENT>
Pool_ending_states
Pool<ITEM, CLIENTS, SLOTS_PER_CLIENT>::blocked()
{
   unique_lock<mutex> lock(mx);
   do { end.wait(lock); } while(pool_ending_state == NOT_END);
   exit = true;
   if(pool_ending_state == FULL)
      not_full.notify_one();
   else
      not_empty.notify_one();
   return pool_ending_state;
}
```

```
template<class ITEM, int CLIENTS, int SLOTS_PER_CLIENT>
bool
Pool<ITEM, CLIENTS, SLOTS_PER_CLIENT>::insert(ITEM* item)
{
   unique_lock<mutex> lock(mx);
   if( (free_slots_count == 0) && (not_full_wait_count == (CLIENTS - 1)) ) {
      pool_ending_state = FULL;
      end.notify_one();
      return false;
   }
   if(free_slots_count == 0) {
      not_full_wait_count++;
      do { not_full.wait(lock); } while( (free_slots_count == 0) && !exit );
      if(exit) {
         not_full.notify_one();
         return false;
      }
      not_full_wait_count--;
   }
   slots[first_empty_slot] = *item;
   if((++first_empty_slot) == CLIENTS * SLOTS_PER_CLIENT)
      first_empty_slot = 0;
   free_slots_count--;
   not_empty.notify_one();
   return true;
}
```

```
template<class ITEM, int CLIENTS, int SLOTS_PER_CLIENT>
bool
Pool<ITEM, CLIENTS, SLOTS_PER_CLIENT>::extract(ITEM* item)
{
   unique_lock<mutex> lock(mx);
   if((free_slots_count == (CLIENTS * SLOTS_PER_CLIENT)) &&
     (not_empty_wait_count == (CLIENTS - 1)))  {
      pool_ending_state = EMPTY;
      end.notify_one();
      return false;
   }
   if(free_slots_count == CLIENTS * SLOTS_PER_CLIENT) {
      not_empty_wait_count++;
      do { not_empty.wait(lock); }
      while( (free_slots_count == CLIENTS * SLOTS_PER_CLIENT) && !exit );
      if(exit) {
         not_empty.notify_one();
         return false;
      }
      not_empty_wait_count--;
   }
   *item = slots[first_full_slot];
   if((++first_full_slot) == CLIENTS * SLOTS_PER_CLIENT)
      first_full_slot = 0;
   free_slots_count++;
   not_full.notify_one();
   return true;
}
```

18.8.3 Izvedba paralelnog izračunavanja površine ispod polukruga

U ovom primeru površina se određuje ispod polukruga (Slika 18.12).



Slika 18.12: Polukrug

Funkcija **semicircle()** (Listing 18.13) omogućuje izračunavanje ordinata tačaka ovog polukruga, a funkcija **square_root()** omogućuje računanje kvadratnog korena. On se uvek nalazi između nule i kvadrata, odnosno između nule i jedinice, pa se računa primenom postupka polovljenja intervala. Funkcija **trapezoid()** omogućuje računanje površine trapeza. Tip **Segment** opisuje podatke koji se odlažu u skladište, odnosno preuzimaju iz skladišta.

Funkcija **thread_numerical_integrator()** opisuje ponašanje niti integratora. Ove niti akumuliraju površine za pojedine [pod]intervale u polju **total** isključive promenljive **area**.

Funkcija **thread_manager()** određuje aktivnost upravljačke niti, koja preuzima podatke o preciznosti određivanja površine i o granicama posmatranog intervala, priprema podatke o ovom intervalu, odlaže ih u skladište (isključiva promenjiva **pool**), stvara i pokreće niti integratore (korišćenjem bezimenih objekata klase **thread**) i na kraju čeka ishod određivanja tražene površine.

Podatak o preciznosti određivanja površine se koristi kod računanja kvadratnog korena za određivanje kraja iteracije, ali i kod izračunavanja površine za utvrđivanje da li je površina trapeza sa celog [pod]intervala dovoljno bliska površinama trapeza sa polovina [pod]intervala.

Listing 18.13: Paralelno određivanje površine ispod polukruga (datoteka **p11.cpp**)

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include "pool.hh"
```

```
double
trapezoid(double height, double base1, double base2)
{
   return(height * (base1 + base2) / 2.);
}

double
absolute(double a)
{
   return((a < 0) ? (-a) : (a));
}

double
square_root(double square, double approximation)
{
   double upper_limit;
   double lower_limit = 0.;
   double old_root;
   double new_root;
   if(square < 1.)
      upper_limit = 1.;
   else
      upper_limit = square;
   new_root = (upper_limit + lower_limit) / 2.;
   do {
      old_root = new_root;
      if(new_root * new_root > square)
         upper_limit = new_root;
      else
         lower_limit = new_root;
      new_root = (upper_limit + lower_limit) / 2.;
   }
   while(absolute(new_root - old_root) > approximation);
   return new_root;
}

const double
RADIUS = 10.;
```

```
double
semicircle(double x, double approximation)
{
   return(square_root(RADIUS*RADIUS-(x-RADIUS)*(x-RADIUS), approximation));
}

class Area {
   mutex mx;
   double total;
  public:
   Area() : total(0) {};
   void increment(double part);
   void show();
};

void
Area::increment(double part)
{
   unique_lock<mutex> lock(mx);
   total += part;
}

void
Area::show()
{
   unique_lock<mutex> lock(mx);
   cout << endl << "area =" << total << endl;
};

Area
area;

struct Segment {
   double begin;
   double end;
   double begin_f;
   double end_f;
   double surface;
};
```

```
const int
THREAD_CLIENTS = 5;

const int
SLOTS_PER_THREAD_CLIENT = 5;

Pool<Segment, THREAD_CLIENTS, SLOTS_PER_THREAD_CLIENT>
pool;

enum
Numerical_integrator_states { IDLE, BUSY };

double approximation;
```

```
void
thread_numerical_integrator()
{
   Numerical_integrator_states state = IDLE;
   Segment all;
   Segment right;
   double left_surface;
   bool more = true;
   while(more) {
      if(state == IDLE) {
         if(!pool.extract(&all)){
            more = false;
            break;
         };
         state = BUSY;
      }
      right.begin = (all.begin + all.end) / 2.;
      right.begin_f = semicircle(right.begin, approximation);
      right.surface = trapezoid(all.end-right.begin, right.begin_f, all.end_f);
      left_surface = trapezoid(right.begin - all.begin, all.begin_f, right.begin_f);
      if(absolute(all.surface -(left_surface + right.surface)) > approximation) {
         right.end = all.end;
         right.end_f = all.end_f;
         if(!pool.insert(&right)){
            more = false;
            break;
         };
         all.end = right.begin;
         all.end_f = right.begin_f;
         all.surface = left_surface;
      } else {
         area.increment(all.surface);
         state = IDLE;
      }
   }
}
```

```
void
thread_manager()
{
   Segment all;
   int counter;
   cout << endl << "PARALLEL NUMERICAL INTEGRATION " << endl
        << "FOR SEMICIRCLE WITH RADIUS (" << RADIUS << ") "
        << "AND WITH CENTRE (" << RADIUS << ",0.)";
   do {
      cout << endl << "approximation (from 0.001 to 10.):";
      cin >> approximation;
   } while((approximation < 0.001) || (approximation > 10.));
   do {
      cout << endl << "left limit (from 0. to " << RADIUS * 2. << "):";
      cin >> all.begin;
   } while((all.begin < 0.) || (all.begin > RADIUS * 2.));
   do {
      cout << endl << "right limit (from " << all.begin
           << " to " << RADIUS * 2. << "):";
      cin >> all.end;
   } while((all.end < all.begin) || (all.end > RADIUS * 2.));
   all.begin_f = semicircle(all.begin, approximation);
   all.end_f = semicircle(all.end, approximation);
   all.surface = trapezoid(all.end - all.begin, all.begin_f, all.end_f);
   pool.insert(&all);
   for(counter = THREAD_CLIENTS; counter > 0; counter--) {
      thread (thread_numerical_integrator).detach();
   }
   if(pool.blocked() == EMPTY)
      area.show();
   else
      cout << endl << "POOL FULL!" << endl;
}

int
main()
{
   thread manager(thread_manager);
   manager.join();
}
```

Sadržaj izvorne datoteke **p11.cpp** (Listing 18.13) predstavlja potpun konkurentni

program. U toku njegovog izvršavanja nastane 6 niti.

U slučaju da se svakoj od niti integratora dodeli poseban procesor, tada je dužina određivanja ukupne površine, opisana prethodnim programom, proporcionalna broju koji se nalazi između logaritma (baze 2) broja podintervala i polovine broja podintervala. Pri tome se podrazumeva da broj procesora nije manji od najvećeg broja podintervala koji nastanu u nekom koraku. Ali, ako prethodni program izvršava jednoprocesorski računar, tada je dužina računanja proporcionalna broju podintervala.

Pri određivanju površine ispod grafa funkcije nije neophodno korišćenje skladišta, ako svaka nit integrator, pri polovljenju [pod]intervala pokuša da stvori i pokrene novu nit integrator, sa namerom da joj prepusti određivanje površine za jednu polovinu [pod]intervala. Podrazumeva se da se pokušaji stvaranja nove niti integratora ponavljaju (nakon kratkotrajnih odlaganja aktivnosti) dok se u tome ne uspe. U ovom slučaju, kraj određivanja tražene površine nastupi kada je suma dužina [pod]intervala, čije površine su određene, jednaka dužini polaznog intervala, ili kada su aktivnosti svih niti integratora zaustavljene pri pokušaju stvaranja novih niti integratora.

18.9 ZADACI

1. Izmeniti program **p07.cpp** tako da upravljačka nit prima od poslednje sortne niti sortirane znakove.
2. Izmeniti program **p08.cpp** tako da računa sumu svih elemenata matrice.
3. Izmeniti program **p09.cpp** tako da čisti sliku od posledica šuma.
4. Izmeniti program **p10.cpp** tako da odredi najkraću međusobnu udaljenost svih čvorova usmerenog grafa.
5. Izmeniti program **p11.cpp** tako da se pri određivanju površine ispod grafa funkcije ne koristi skladište.

LITERATURA

Potpunije upoznavanje materije vezane za operativne sisteme omogućuju knjige:
A.S. TANENBAUM, Modern Operating Systems, fourth edition, Prentice Hall, New Jersey, 2014,

A.S. TANENBAUM, M. VAN STEEN, Distributed Systems - Principles and Paradigms, second edition, Prentice Hall, New Jersey, 2006 i

J. Bacon, T. Harris, Operating Systems - Concurrent and Distributed Software Design, Addison Wesley, 2003.

Sveobuhvatnije upoznavanje konkurentnog programiranja omogućuju knjige:
G.R. ANDREWS, Concurrent Programming - Principles and Practice, Benjamin/Cummings, 1991 i

G.R. ANDREWS, Foundation of Multithreaded, Parallel and Distributed Programming, Addison Wesley, 2000.

Upoznavanje rigoroznog pristupa konkurentnom programiranju omogućuje knjiga:
F.B. SCHNEIDER, On Concurrent Programming, SpringerVerlag, New York, 1997.

Pregled međunarodnog standarda C++11 sadrži *Wikipedia*.

Detaljno upoznavanje programskog jezika C++ omogućuju knjige:
B. STROUSTRUP, The C++ Programing Language, fourth edition, Addison Wesley, 2013. i

B. STROUSTRUP, The Design and Evolution of C++, Addison Wesley, 1995.

Prethodno pomenute reference sadrže iscrpan pregled literature iz oblasti operativnih sistema i konkurentnog programiranja.

PRILOG: INDEKS SLIKA