Univerzitet u Nišu
Prirodno-Matematički fakultet

Marko D. Petković

OSNOVI KONKURENTNOG PROGRAMIRANJA SA ZBIRKOM ZADATAKA

Niš, 2010

Dr Marko D. Petković, docent Prirodno-Matematičkog fakulteta u Nišu
Godina rođenja autora: 1984

OSNOVI KONKURENTNOG PROGRAMIRANJA SA ZBIRKOM ZADATAKA

Recenzenti: Dr Predrag S. Stanimirović,
redovni profesor Prirodno-matematičkog fakulteta u Nišu
Dr Predrag V. Krtolica,
docent Prirodno-matematičkog fakulteta u Nišu

Izdavač: Prirodno-matematički fakultet u Nišu,
Višegradska 33, 18000 Niš
www.pmf.ni.ac.rs

Odlukom Nastavno-naučnog veća Prirodno-Matematičkog fakulteta u Nišu br. 687/3-01 od 09.09.2009 godine rukopis je odobren za štampu kao pomoćni univerzitetski udžbenik.

ISBN: **978-86-83481-69-9**

Tiraž: 100 primeraka

Format: B5

Tehnička obrada: autor

Štampa: Osa Nova, Niš

SADRŽAJ

PREDGOVOR

Konkurentno programiranje je nastalo kao potreba da se, u to vreme, veliki i skupi računarski resursi prilagode za rad sa više korisnika. Ta potreba je naročito bila izražena u velikim kompanijama u kojima je jedan (mainframe) računar morao da obsluži veći broj korisnika sa različitim zahtevima i različitim pravima pristupa. Sa razvojem računarskih sistema, odnosno personalnih računara, konkurentno programiranje je našlo primenu i na ovoj vrsti računara. Naime, korisnici su imali sve veću potrebu da na svojim personalnim računarima konkurentno izvršavaju odjednom više poslova (slušanje muzike, pisanje teksta, rezanje CDa, itd.). Danas velika većina modernih operativnih sistema (Windows, Linux, Solaris, itd.) podržava različite oblike konkurentnog programiranja.

Oblast konkurentnog programiranja najčešće se proučava (u okviru univerzitetske nastave, a i uopšte) kao sastavni deo kursa vezanog za operativne sisteme. Ne može se reći da je ova podoblast najvažnija za razvoj operativnih sistema ali je definitivno jedna od najizazovnijih i ujedno i najinteresantnijih podoblasti. Sa druge strane, konkurentno programiranje je u tesnoj vezi sa paralelnim programiranjem, samim tim što se mnoge metode konkurentnog programiranja mogu sa malim modifikacijama primeniti na paralelno programiranje i obrnuto.

Zadatak konkurentnog programiranja je prilagodjenje nezavisnih programa (procesa) konkurentnom izvršavanju na istoj mašini. To prilagodjenje se odnosi na medjusobnu sinhronizaciju i podelu zajedničkih resursa. Potrebno je definisati odredjena pravila po kojima programi pristupaju zajedničkim resursima računara ali tako da ne remete jedan drugog i da svaki program posle odredjenog vremena čekanja dobije traženi pristup resursima. Problemi koji tom prilikom nastaju su veoma slični nekim problemima iz svakodnevnog života. Za većinu problema može se veoma lako konstruisati jednostavno, logično ali i netačno rešenje. Da situacija bude još gora, ovo rešenje korektno rešava problem osim u nekom vrlo specijalnom slučaju kada nastaju problemi (krah celokupnog sistema, gubitak podataka, itd.). Cilj konkurentnog programiranja je da se otkloni i najmanja sumnja u korektnost konkurentnog izvršavanja programa.

Ova knjiga je proistekla iz računskih vežbi iz predmeta *Paralelna obrada* koje sam držao na odseku za matematiku i informatiku na Prirodno-matematičkom fakultetu u Nišu. Samim tim, ova knjiga prvenstveno treba da posluži kao udžbenik, odnosno pomoćni udžbenik iz ovog predmeta, kao i iz predmeta *Napredni kurs iz računarskih arhitektura*. Pored toga, ova knjiga može da se koristi delom i za

predmete *Operativni sistemi* i *Uvod u operativne sisteme* na odseku za matematiku i informatiku Prirodno-Matematičkog fakulteta u Nišu. Naravno, knjiga može biti od koristi studentima drugih fakulteta na kojima se proučava ova problematika kao i programerima pri rešavanju konkretnih zadataka.

Knjiga je podeljena u 6 poglavlja. Svako poglavlje najpre sadrži teorijske osnove vezane za problematiku koja se proučava u tom poglavlju a nakon toga veći broj kompletno uradjenih zadataka. Odredjen broj zadataka (naročito u 2. i 3. poglavlju) predstavljaju poznate probleme iz odgovarajućih oblasti, formulisane u obliku zadatka. Rešenje svakog zadatka (osim u 1. poglavlju) sastoji se iz dva dela. Najpre je dato teorijsko objašnjenje sa posebnim osvrtom na ključne ideje u rešenju kao i na sinhronizacione strukture koje su pritom korišćene. Nakon toga sledi implementacija rešenja u programskom jeziku `BACI` (drugo, treće i četvrto poglavlje) odnosno programskom jeziku `ADA` (peto i šesto poglavlje) Nakon svakog poglavlja dato je i nekoliko zadataka za vežbu.

Zadaci su najčešće formulisani kao problemi iz svakodnevnog života. Time se čitalac ne opterećuje nepotrebnim detaljima vezanim za konkretne probleme u računarstvu a takodje se i pažnja čitalaca usredsredjuje na sam problem. Rešenja su tako formulisana da čitalac rešavajući zadatke stekne uvid u neke osnovne sinhronizacione strukture koje se najčešće koriste kao i ideje za rešavanje novih problema.

U prvom poglavlju razmatraju se performanse računarskog sistema. Dati su izrazi pomoću kojih se izračunavaju veličine kao što su CPI, MIPS, MFLOPS, itd. Kroz veći broj konkretnih zadataka pokazano je kako se ti izrazi primenjuju u praktičnim proračunima. U drugom poglavlju definišu se osnovni pojmovi vezani za konkurentno programiranje. Najpre je na jednom prostom primeru (sortiranje niza) pokazano koji to sve problemi mogu nastati pri konkurentnom izvršavanju dva ili više procesa. Date su definicije pojmova kao što su zaključavanje (*deadlock*) i umiranje od gladi (*lockout*) a razmatrani su i uslovi kada je neka konkurentna implementacija korektna. Nakon toga je detaljno razmatran osnovni a ujedno i najvažniji problem konkurentnog programiranja - problem kritične sekcije. Nekoliko različitih varijanti rešenja problema kritične sekcije formulisano je u obliku zadataka. Tema trećeg poglavlja su semafori, najvažnija sinhronizaciona struktura u konkurentnom programiranju. Data je definicija semafora i pokazano je kako se semafori koriste u različitim sinhronizacionim problemima. Tridesetak problema je kompletno uradjeno u vidu zadataka. Rešenje svakog problema je detaljno obrazloženo i data je implementacija u programskom jeziku `BACI` (konkurentna verzija programskog jezika `PASCAL`). Izmedju ostalih, razmatrani su i fundamentalni sinhronizacioni problemi kao što su problem kritične sekcije, problem proizvodjača i potrošača, problem filozofa koji večeraju, problem čitalaca i pisaca, itd. Četvrto poglavlje posvećeno je monitorima, takodje veoma važnoj sinhronizacionoj strukturi. Data je definicija monitora i pokazano je kako se monitori mogu upotrebiti u rešavanju sinhronizacionih problema. Kao i u predhodnom poglavlju, i ovde je kompletno rešeno nekoliko sinhronizacionih

problema formulisanih u obliku zadataka. Pokazana je i ekvivalentnost semafora i monitora. U petom poglavlju, čitalac se upoznaje sa osnovama programiranja u programskom jeziku `ADA`. Sobzirom da jezik `ADA` nije mnogo rasprostranjen medju programerima, cilj ovog poglavlja je upoznavanje čitalaca sa osnovnim programerskim konstrukcijama ovog programskog jezika da bi mogao lakše da prati materiju izloženu u poslednjem, šestom poglavlju. Tema šestog poglavlja je mehanizam prosledjivanja poruka *Rendez-Vous*. Ova sinhronizaciona struktura je karakteristična za programski jezik `ADA`. Detaljno je objašnjen tip `task` kojim se implementiraju konkurentni procesi u programskom jeziku `ADA` kao i komunikacija izmedju taskova putem mehanizma *Rendez-Vous*. Posebna pažnja posvećena je `select` naredbi i njenim oblicima. Na kraju ovog poglavlja kompletno je uradjeno nekoliko primera sinhronizacije pomoću sistema *Rendez-Vous*, formulisanih u obliku zadataka. Rešenje svakog zadatka detaljno je obrazloženo i data je implementacija u programskom jeziku `ADA`.

Koristim ovu priliku da se zahvalim recenzentima prof. dr Predragu Stanimiroviću i dr Predragu Krtolici na pažljivom čitanju rukopisa i veoma korisnim primedbama i savetima. Zahvalnost dugujem kolegi i prijatelju Aleksandru Iliću koji je pročitao dobar deo rukopisa i dao niz korisnih primedbi i sugestija. Zahvaljujem se i svojim studentima sa odseka za matematiku i informatiku Prirodno-matematičkog fakulteta u Nišu koji su takodje doprineli poboljšanju kvaliteta teksta.

Kako je ovo prvo izdanje, verovatno postoje propusti i greške. Stoga se autor unapred izvinjava čitaocima i moli ih za razumevanje. Takodje, autor se zahvaljuje svim čitaocima koji mu ukažu na greške. Za sve komentare, primedbe, sugestije, itd. koji su više nego dobrodošli, čitaoci mogu kontaktirati autora putem e-maila na adresu **dexterofnis@gmail.com**.

U Nišu, Autor
Maj 2010.

- Semafori nameću prirodne uslove koji u mnogome mogu pomoći programerima da izbegnu greške u svojim programima.
- Rešenja sinhronizacionih problema pomoću semafora su pregledna i lako se dokazuje korektnost samog programa.
- Semafori mogu biti efikasno implementirani u mnogim okruženjima i na mnogim sistemima, pa rešenja koja koriste semafore mogu raditi na različitim platformama i efikasna su.

3.2. Osnovni sinhronizacioni šabloni pomoću semafora

U ovom poglavlju izložićemo nekoliko osnovnih sinhronizacionih problema koje ćemo rešiti pomoću semafora.

Zadatak 1. (problem signalizacije, odnosno serijalizacije) Dva procesa `A` i `B` se izvršavaju konkurentno i treba da izvrše redom procedure `a1` i `b1`. Pri tome, procedura `a1` mora biti izvršena pre procedure `b1` i njihova izvršenja ne smeju se poklapati. Napisati program kojim se ovaj problem rešava pomoću semafora i dokazati njegovu ispravnost.

Rešenje:

Rešenje je dato sledećim kodom:

```
program serialization;
var s:semaphore;

procedure A;
begin
   a1;
   signal(s);
end;

procedure B;
begin
   wait(s);
   b1;
end;
begin
   initialsem(s,0);
   cobegin
      A; B;
   coend;
end.
```

Ako procedura B dodje do funkcije wait pre nego što je procedura A došla do funkcije signal, ona se blokira (pošto je vrednost semafora s jednaka 0), i ostaje blokirana sve dok procedura A ne dodje do funkcije signal. Procedura A pre izvršenja funkcije signal prvo izvrši proceduru a1, a zatim probudi proceduru B koja izvrši b1. Ako A izvrši funkciju signal pre nego što B izvrši wait, tada B prolazi bez čekanja i izvršava b1 (pri čemu je a1 već gotova). Prema tome u oba slučaja se b1 izvršava posle a1.

Napomena: Na ovom primeru se lepo vidi zašto su osnovne funkcije wait i signal dobile baš ta imena. U nekim implementacijama, ove funkcije se označavaju redom sa decrement i increment, ili tradicionalno (kako ih je Dijkstra prvi put označavao) sa P i V. Takodje, ime s semaforske promenljive u ovom slučaju nije baš pametan izbor. Mnogo logičniji izbor bi bio npr. aArrived.

Zadatak 2. (susret) Dva procesa A i B se izvršavaju konkurentno, pri čemu treba da izvrše redom procedure a1 i a2, odnosno b1 i b2. Pri tome, a1 mora da se izvrši pre b2 i b1 mora da se izvrši pre a2. Napisati program kojim se obezbedjuje ovaj uslov korišćenjem semafora. Programom ne sme biti definisan redosled izvršenja procedura a1 i b1, kao i a2 i b2.

Rešenje:

Koristićemo dva semafora aArrived i bArrived, koji će da označavaju redom da su procesi A i B izvršili procedure a1 i b1. Kod je sličan kodu iz prethodnog zadatka, zato navodimo samo kodove funkcija A i B.

```
procedure A;
begin
   a1;
   signal(aArrived);
   wait(bArrived);
   a2;
end;

procedure B;
begin
   b1;
   signal(bArrived);
   wait(aArrived);
   b2;
end;
```

Kada proces A završi proceduru a1, on signalizira semafor aArrived i time dopušta procesu B da izvrši b2. Dalje, on pozivom funkcije wait čeka B da izvrši b1 i da ga o tome obavesti pomoću semafora bArrived. Kad se to dogodi, A izvršava a2.

Napomena: Ukoliko zamenimo redosled poziva funkcija wait i signal u jednoj od procedura A i B, rešenje je ponovo korektno. Medjutim, ukoliko zamenimo redoslede u obe funkcije, dolazi do zastoja.

Zadatak 3. (kritična sekcija) Procesi `A` i `B` se izvršavaju konkurentno, i svaki treba da izvrši proceduru `crit` koja predstavlja kritičnu sekciju (procesi `A` i `B` ne smeju u isto vreme izvršavati proceduru `crit`). Napisati program kojim se ovaj uslov realizuje. Redosled kojim `A` i `B` izvršavaju kritičnu sekciju nije bitan i ne sme biti odredjen programom.

Rešenje:

Ovo je još jedan od osnovnih sinhronizacionih problema koji se veoma efikasno rešavaju pomoću semafora. Koristićemo jednu semaforsku promenljivu `mutex`.

Pretpostavimo da proces `A` prvi dodje do poziva `wait` funkcije. Tada on izvršava kritičnu sekciju. Proces `B` je za to vreme blokiran sve dok `A` ne dodje do poziva funkcije `signal`. Tada `B` ulazi u kritičnu sekciju. Sledi implementacija ovog rešenja.

```
procedure A;
begin
   wait(mutex);
   crit;
   signal(mutex);
end;

procedure B;
begin
   wait(mutex);
   crit;
   signal(mutex);
end;
```

Napomena: Primetimo da su kodovi funkcija A i B identični. Rešenje kod kojeg je ovo slučaj se naziva simetričnim. U suprotnom, rešenje se naziva asimetričnim. Simetrična rešenja su naročito pogodna pošto se najčešće jednostavno i efikasno generalizuju u slučaju više od dva konkurentna procesa.

Zadatak 4. (multipleks) Posmatrajmo modifikovani problem kritične sekcije za više od dva konkurentna procesa. Pri tome pretpostavimo da funkciju `crit` mogu da izvršavaju više procesa istovremeno, pri čemu je gornja granica za taj broj procesa ograničena brojem n. Drugim rečima, ne više od n procesa mogu da izvršavaju kritičnu sekciju paralelno (u isto vreme). Napisati program kojim se to obezbedjuje, ako je ukupan broj procesa $N = 10$ i $n = 5$.

Rešenje:

Kod je identičan kao i u prethodnom zadatku, samo što se sada vrednost semaforske promenljive inicijalizuje na vrednost *n*.

```
program MutualExclusion;
const n=5;
var mutex:semaphore;

procedure P(i:integer);
begin
   wait(mutex);
   crit(i);
   signal(mutex);
end;

begin
   initialsem(mutex,n);
   cobegin
     P(1); P(2); ... P(N);
   coend;
end.
```

Primetimo da je vrednost semaforske promenljive `mutex` u ovom slučaju broj procesa koji trenutno mogu da udju u kritičnu sekciju. Ukoliko je ovaj broj nula, proces čeka sve dok drugi proces koji izvršava kritičnu sekciju ne završi. Tada on budi jedan od procesa koji čekaju na izvršenje. Kada svi procesi završe kritičnu sekciju, vrednost semaforske promenljive ponovo postaje jednaka broju **n**.

Napomena: U realnom životu, ovakva situacija se često sreće kod šaltera u pošti, banci, itd. Najčešće imamo nekoliko šaltera na kojima radi po jedan službenik koji može da radi samo sa jednom osobom istovremeno. Osobama koje čekaju u redu je u principu svejedno na kom šalteru će obaviti svoj posao, i čim se jedan od šaltera oslobodi, prva sledeća osoba u redu pristupa tom šalteru. Na početku i na kraju, svi šalteri su slobodni.

Zadatak 5. (barijera) Posmatrajmo modifikovani problem susreta iz zadatka 2. Neka sada N procesa mora da izvrši dve procedure: `before` i `after`, pri čemu proceduru `after` proces može da izvrši tek kad svi ostali procesi izvrše proceduru `before`. Napisati program koji rešava ovaj problem, pri čemu osim pomenutog uslova nikakav drugi redosled izvršenja procedura ovih procesa nije bitan i ne sme biti definisan programom. Neka je broj procesa jednak 10.

Rešenje:

Koristićemo dve semaforske promenljive, `barrier` i `mutex`, kao i brojač `count` koji označava koliko je procesa završilo proceduru `before` i spremno je na

izvršenje procedure `after`. Ukoliko `count` nije jednako nuli, proces se blokira na semaforu `barrier`, a ako jeste on "otključava" barijeru, i svi procesi nastavljaju sa radom. Setimo se da operacija dekrementiranja broja `count` predstavlja kritičnu sekciju, pa prema tome moramo je obezbediti semaforom `mutex`.

```
program Barrier;

var mutex, barrier:semaphore;
    count:integer;

procedure P(i:integer);
begin
   before(i);
   wait(mutex);
      count:=count-1;
   signal(mutex);

   if count=0 then signal(barrier);
   wait(barrier);
   after(i);
end;
begin
   initialsem(mutex,0);
   initialsem(barrier,0);
   count:=10;
   cobegin
     P(1); ... P(10);
   coend;
end.
```

Ovo rešenje, iako na prvi pogled izgleda korekno, **zapravo je skroz pogrešno**. Naime, ukoliko prvih 9 procesa stignu do funkcije `wait(barrier)` (pri čemu poslednji proces za to vreme ne radi ništa), oni će biti blokirani. Kada poslednji proces dodje do `if` naredbe, on tada izvrši funkciju `signal(barrier)` i jedan od uspavanih procesa se budi. Posle toga on izvršava `wait(barrier)`, blokira se i ostaje blokiran zajedno sa preostalih 8 procesa. Prema tome, u ovom scenariju samo jedan proces nastavlja sa radom, tj nastaje zastoj.

Ovaj problem možemo rešiti na sledeći način. Kada jedan proces prodje barijeru, on probudi drugi proces koji je uspavan na semaforu. Ovaj proces budi sledeći, itd. Na ovaj način će se svi uspavani procesi probuditi. Razlika u kodu je samo što je dodat još jedan poziv funkcije `signal(barrier)`.

Drugi nedostatak predhodnog[4] koda je to što je `if` naredba kojom se ispituje vrednost promeljive `count` van kritične sekcije. Time je ostavljena mogućnost da više procesa sukcesivno izvrši tu `if` naredbu, odnosno da semafor `barrier` bude nekoliko puta signaliziran. Problem nastaje i u slučaju kada jedan proces dekrementira `count` a drugi izvršava `if`. Prema tome, na kraju programa, vrednost

[4] Koji zapravo ne remeti korektnost programa, ali zato skreće pažnju na neke stvari koje bi trebalo izbegavati u sličnim problemima.

semafora `barrier` može biti bilo koji broj izmedju 1 i 10. Pošto `barrier` nije resetovan na 0, ovaj kod je potpuno beskorisan npr. u slučaju kada procesi treba istu sekvencu da prodju 2 puta.

Sledi implementacija korektnog rešenja.

```
program Barrier;
var mutex, barrier:semaphore;
    count:integer;

procedure P(i:integer);
begin
  before(i);
  wait(mutex);
    count:=count-1;
    if count=0 then
       signal(barrier);
  signal(mutex);

  wait(barrier);
  signal(barrier);

  after(i);
end;

begin
   initialsem(mutex,0);
   initialsem(barrier,0);
   count:=10;
   cobegin
     P(1); ... P(10);
   coend;
end.
```

Zadatak 6. (barijera za više prolaza) Modifikujmo sada problem iz prethodnog zadatka tako što ćemo da predpostavimo da procesi moraju da izvršavaju procedure `before` i `after` više puta. Jedno rešenje ovog problema je dato sledećim kodom:

```
program ReusableBarrier;
var mutex, barrier:semaphore;
    count:integer;

procedure P(i:integer);
begin
   repeat
      before(i);
      wait(mutex);
         count:=count-1;
         if count=0 then signal(barrier);
      signal(mutex);

      wait(barrier);
      signal(barrier);
      after(i);
      wait(mutex);
         count:=count+1;
         if count=n then wait(barrier);
      signal(mutex);
   forever;
end;

begin
   initialsem(mutex,1);
   initialsem(barrier,0);
   count:=10;
   cobegin
      P(1); ... P(10);
   coend;
end.
```

Nažalost, ovo rešenje **nije korektno**. Pronaći scenario pri kome ovo rešenje ne radi korektno. Ispraviti ovaj program uvodjenjem nove semaforske promenljive `barrier2`.

Rešenje:

Posle poziva funkcije `after`, svaki proces signalizira semafor `barrier` i povećava vrednost promenljive `count`. Proces koji poslednji inkrementira promenljivu `count`, signalizira `barrier` još jednom, pa tada ovaj semafor ima vrednost 0 kao što je imao i na početku.

Medjutim ovo rešenje iako izgleda korektno ima nedostatak. Naime, posmatrajmo scenario u kome proces koji oslobodi barijeru (signalizira `barrier` u prvoj `if` naredbi) nastavi sa izvršenjem, završi jedan ciklus, i ponovo udje u barijeru. Proces koga je oslobodio signaliziranjem barijere ne radi za to vreme ništa, dok su ostali procesi još uvek blokirani na semaforu `barrier`. On će tada ponovo signalizirati `barrier` i ponovo proći ostatak koda sve do kraja ciklusa, itd.