Operativni sistemi
- Sinhronizacija procesa -

Veljko Stanković

Uvod

- Kooperirajuci proces je proces koji moze da utice na druge procese ili na koga mogu da uticu drugi procesi.
- Kooperirajuci procesi mogu da dele adresni prostor (kao niti) ili mogu da dele samo otvorene fajlove.
- Konkurentni pristup podacima (otvorenim fajlovima) moze dovesti do problema konzistentnosti podataka.
  - Problem blagovremenog azuriranja i pristupa podacima po odgovarajucem redosledu.

Uvod

— Primer problema sinhronizacije procesa

- Komunikacija preko zajednicke memorije i bafera konacne velicine

```
repeat
    ...
    produce an item in nextp
    ...
    while in+1 mod n = out do no-op;
    buffer[in] := nextp;
    in := in+1 mod n;
until false;
```

```
repeat
    while in = out do no-op;
    nextc := buffer[out];
    out := out+1 mod n;
    ...
    consume the item in nextc
    ...
until false;
```

```
repeat
    ...
    produce an item in nextp
    ...
    while counter = n do no-op;
    buffer[in] := nextp;
    in := in + 1 mod n;
    counter := counter + 1;
until false;
```

```
repeat
    while counter = 0 do no-op;
    nextc := buffer[out];
    out := out + 1 mod n;
    counter := counter − 1;
    ...
    consume the item in nextc
    ...
until false;
```

Uvod

— Predpostavimo da se operacije azuriranja brojaca vrse sledecim setom naredbi.

$register_1 := counter;$	$register_2 := counter$
$register_1 := register_1 + 1;$	$register_2 := register_2 - 1;$
$counter := register_1$	$counter := register_2$

— Ove naredbe se mogu izvrsavati u potpuno proizvoljnom redosledu

T_0:	*producer*	**execute**	$register_1 := counter$	$\{register_1 = 5\}$
T_1:	*producer*	**execute**	$register_1 := register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	**execute**	$register_2 := counter$	$\{register_2 = 5\}$
T_3:	*consumer*	**execute**	$register_2 := register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	**execute**	$counter := register_1$	$\{counter = 6\}$
T_5:	*consumer*	**execute**	$counter := register_2$	$\{counter = 4\}$

— Uslovi trke

➥ Slucaj kada nekoliko procesa modifikuju iste podatke i rezultat obrade zavisi od redosleda kojim oni modifikuju te podatke.

☑ U prethodnom primeru, problem se moze resiti tako sto cemo obezbediti da samo jedan proces ima pristup podatku counter.

Problem kriticnog preklapanja

1010011
1110100
1100001
1010110

- Posmatramo sistem koji se sastoji iz *n* procesa: $\{P_0, P_1, \ldots, P_{n-1}\}$
- Svaki proces ima deo koda koji se naziva kriticni deo, u kome se moze desiti da proces menja neke zajednicke promenljive.
- Uslovi da nebi doslo do narusavanja konzistentnosti podataka
  - Veoma je vazno da kada jedan proces izvrsava neki svoj kriticni deo, nijedan drugi proces ne sme da izvrsava svoj kriticni deo.
  - Sinhronizacija procesa ne sme da zavisi od broja ili brzine CPU-a
  - Nijedan proces kojise izvrsava izvna kriticne sekcije, ne sme da blokira ostale procese
  - Nijedan proces ne sme previse dugo da ceka da bi izvrsio svou kriticnu sekciju
  - Svaki proces mora da trazi dozvolu da krene sa izvrsavanjem svog kriticnog dela.
    - ☑ Deo koda koji implementira ovaj zahtev se naziva **entry** sekcija.
    - ☑ Nakon kriticnog koda sledi **exit** sekcija

Problem kriticnog preklapanja

1010011
1110100
1100001
1010110

— Resenje problema izvrsavanja kriticnog dela mora da zadovolji sledece zahteve:

- Medjusobna iskljucivost
  - ☑ Ako neki proces Pi izvrsava svoju kriticnu sekciju, tada nijedan drugi proces ne sme da izvrsava svoju kriticnu sekciju
- Progres
  - ☑ Ako nijedan proces ne izvrsava svoju kriticnu sekciju, tada samo oni procesi koji ne izvrsavaju deo koda nakon kriticne sekcije mogu da se nadmecu za izvrsavanje svoje kriticne sekcije.
- Ograniceno cekanje
  - ☑ Broj pristupa drugih procesa svojim kriticnim sekcijama nakon sto je posmatrani proces dao zahtev da pristupi svojoj kriticnoj sekciji do trenutka kada je taj zahtev odobren je ogranicen.

Medjusobno iskljucenje sa uposlenim cekanjem

- Medjusobno iskljucenje sa uposlenim cekanjem (Mutual Exclusion with Busy Waiting)
  - Najjednostavniji nacin da se resi problem preklapanja kriticnih sekcija procesa, jeste da proces pre izvrsenja svoje kriticne sekcije suspenduje sve interupte (pa i timing interupt koji dovodi do promene konteksta CPU) i ponovo ih aktivira neposredno pre zavrsetka izvrsavanja kriticne sekcije.
  - Nijedan drugi proces ne moze da se izvrsava i pristupi zajednickim podacima.
  - Resenje nije dobro jer se korisnickom procesu daje mogucnost da sam suspenduje i aktivira interupte.
  - Elegantno resenje za izvrsavanje sistemskih procesa.

Uposleno cekanje

— Umesto da suspenduje interupte, korisnicki proces pre izvrsenja kriticne sekcije proverava vrednost celobrojne promenljive da li je 0

— Drugi proces proverava vrednost ove promenljive sve dok ona ne promeni vrednost 0->1, tj. proces koji trenutno izvrsava svoju kriticnu sekciju ne promeni njenu vrednost neposredno pre zavrsetka izvrsavanja svoje kriticne sekcije.

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Uposleno cekanje

- Stalna provera neke promenljive dok ona ne promeni svoju vrednost se naziva uposleno cekanje.
  - Treba izbegavati jer se uzaludno gube CPU ciklusi.
- Naizmenicno izvrsavanje kriticnih sekcija nije dobro resenje ako je jedan proces mnogo sporiji od drugog, tj ako je razlika u trajanjima CPU ciklusa velika.
  - Primer: Predpostavimo da je trjanje CPU ciklusa procsa 1 duze nego trajanje ciklusa procesa.0. Proces 0 ce nakon izvrsavanja svog kriticnog ciklusa promniti vrednost signalne promenljive 0->1 i nastaviti sa izvrsavanjem nekriticnih ciklusa. Proces 1 ce nakon izvrsavanja svog kriticnog ciklusa postaviti vrednost signalne promenljive 1->0 i nastaviti sa izvrsavanjem nekriticnog ciklusa. Sada proces 0 ponovo moze da izvrsava svoj kriticni cklusa. Medjutim, nakon zavrsetka izvrsavanja kriticnog ciklusa procesa 0, proces 1 jos uvek izvrsava svoje nekriticne cikluse i proces 0 mora da saceka da proces 1 zavrsi sa svojim nekriticnim i kriticnim ciklusima.

Petersonov algoritam

- Resava problem sinhronizacije dva procesa
- Procesi su oznaceni brojevima 0 i 1
- Algoritam zahteva da procesi dele dve promenljive
  - int turn
    - ☑ pokazuje proces koji je na redu da izvrsava svoj kriticni ciklus
  - boolean flag[2]
    - ☑ pokazuje da li je odredjeni proces spreman da izvrsava svoj kritican ciklus
- Pre izvrsenja odredjeni proces (na primer proces P0) postavlja vrednost parametra turn na 0 i flag[0] na true
  - Proces P1 mora da ceka sve dok proces P0 ne zavrsi svoj kriticni ciklus i postavi flag[0] na false
  - Proces P1 moze da promeni parametar turn na 1

Petersonov algoritam


```
int turn;                                   /* whose turn is it? */
int interested[N];                          /* all values initially 0 (FALSE) */

void enter_region(int process);             /* process is 0 or 1 */
{
      int other;                            /* number of the other process */

      other = 1 – process;                  /* the opposite of process */
      interested[process] = TRUE;           /* show that you are interested */
      turn = process;                       /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)              /* process: who is leaving */
{
      interested[process] = FALSE;          /* indicate departure from critical region */
}
```

Test and Set Lock (TSL)

1010011
1110100
1100001
1010110

- Broj procesa veci od 2
- Dve zajednicke promenljive
  - lock
  - waiting[n]
- Algoritam
  - Proces testira lock dok ona ne promeni svoju vrednost u false
  - Proces menja svoj waiting u false i postavlja lock u true
  - Zapocinje sa izvrsavanjem kriticnog ciklusa
  - Bira naredni proces koji ceka na ulazak u svoju kriticnu sekciju
  - Drugi proces preuzima lock
- Zahteva implemetaciju operacija kao sto su TestAndSetLock ili Swap
  - Implementacija nije jednostavna.

Test and Set Lock (TSL)

1010011
1110100
1100001
1010110

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
}while (TRUE);
```

Swap naredba

1010011
1110100
1100001
1010110

```
void Swap(boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock, &key);

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Semafori

- Jednostavan koncept i mehanizam za programiranje medjusobne iskljucivosti i kriticnog preklapanja.
- Sva prethodna resenja za sinhronizaciju procesa zahtevaju uposleno cekanje i posebne instrukcije za proveru i azuriranje lock/turn parametra.
  - Ciklusi koji se gube uposlenim cekanjem se mogu korisnije iskoristiti.
- Prethodni metodi za sinhronizaciju procesa se mogu posmatrati kao vrsta binarnih semafora.
  - Ova vrsta semafora se jos zove i spinlock jer se proces “vrti” (spin) dok ceka da se oslobodi pristu kriticnoj sekciji (lock – brava).
  - Pristup sa uposlenim cekanjem je koristan u slucajevima kad procesi/niti ne cekaju mnogo na pristup kriticnoj oblasti, kao sto je slucaj sa multiprocesorskim sistemima.

Semafori

- Semafor *S* predstavlja ceolobrojnu promenljivu kojoj se moze pristupiti samo preko dve standardne atomicne operacije: *wait* i *signal*.
- Jedna opcija jeste da proces umesto da izvrsava uposleno cekanje na pristup kriticnoj sekciji, blokira svoje izvrsavanje kada biva smesten u red cekanja (waiting) koji je pridruzen odredjenom semaforu.
  - Kontrolu preuzima CPU alokator koji odlucuje koji proces ce sledeci da izvrsava svoju kriticnu sekciju.
- Svaki semafor je definisan:
  - Celim brojem S
  - listom procesa

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

Semafori

— Kada proces mora da ceka na semafor, on se dodaje u red cekanja:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

— Operacija *signal*() uklanja proces iz reda cekanja na odredjeni semafor i aktivira njegovo izvrsavanje

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

➥ Operacije *block*() i *wakeup*() predstavljaju osnovne sistemske pozive OS-a.

Semafori

- Klasicno uposleno cekanje predstavlja semafor kod koga vrednost S nije nikada negativna.
- Ukoliko je vrednost semafora negativna, tada njegova apsolutna vrednost odgovara broju procesa u redu cekanja na taj semafor.
- Operacije nad semaforima wait() i signal() su atomicne
  - Ne smeju da postoje dva procesa koji istovremeno izvrsavaju ove dve opseracije isotvremeno nad istim semaforom.
  - Problem se resava jednostvnim zamrzavanjem interapta za vreme izvrsavanja operacija wait() i signal().
    - ☑ Ovo resenje funkcionise samo na sistemima sa jednim procesorom.
    - ☑ Na sistemima sa vise procesora morali bi da onesposobimo interapte na svim CPU kako bi sprecili da procesi na razlicitim CPU izvrsavaju istovremeno operacije wait() i signa() nad istim semaforom.

Semafori

- Implementacijom semafaora na ovaj nacin nismo u potpunosti uklonili uposleno cekanje, vec smo ga samo ogranicili na kriticne sekcije operacija wait() i signal() koje su obicno veoma kratke (10-ak instrukcija).
  - Kao posledica, kriticne sekcije su veliki deo vremena slobodne pa je i trajanje uposlenog cekanja kratko.
- Posmatramo set problema vezanih za sinhronizaciju koji se obicno koriste pri testiranju novih algoritama.

Semafori
- Primer: Pristup baferu konacne duzine -

— Pristup baferu konacne duzine

- Pristup baferu duzine *n*
- Koriste se 3 semafora
  - ☑ mutex = 1
    - kontrolise pristup baferu
  - ☑ full = 0
    - kontrolise upis podataka u bafer
    - pokazuje broj upisanih podataka u bafer
  - ☑ empty = n
    - kontrolise citanje podataka iz bafera
    - pokazuje broj praznih likacija u baferu

Semafori
- Primer: Pristup baferu konacne duzine -

1010011
1110100
1100001
1010110

```
do {
      . . .
   // produce an item in nextp
      . . .
   wait(empty);
   wait(mutex);
      . . .
   // add nextp to buffer
      . . .
   signal(mutex);
   signal(full);
}while (TRUE);
```

```
do {
   wait(full);
   wait(mutex);
      . . .
   // remove an item from buffer to nextc
      . . .
   signal(mutex);
   signal(empty);
      . . .
   // consume the item in nextc
      . . .
}while (TRUE);
```

Semafori
- Problem citanja – upisa -

- Posmatra se baza podataka koju deli *n* procesa
  - Neki procesi zele samo da ocitaju podatke iz baze podataka (read)
  - Neki procesi zele da azururaju podatke u bazi (read, write)
- Problem se javlja kada proces koji ima pravo citanja i upisa (read&write – writer) pokusa da istovremeno pristupi podatku kada i neki drugi proces (nebitno da li ima ili ne pravo upisa)
  - Problem se resava tako sto se procesima sa pravom upisa se daje ekskluzivno pravo pristupa podacima.

Semafori
- Problem citanja – upisa -

— Postoji vise varijanti ovog problema

- Prvi problem citanja-upisa
  - ☑ Nijedan proces sa pravima citanja ne sme da ceka na pristup sem ako proces sa pravom upisa nije vec dobio dozvolu da pristupi zajednickom objektu
- Drugi problem citanja-upisa
  - ☑ Ako je neki proces sa pravom upisa sprema da pristupi objektu, tada mu se treba omoguciti da pristupi u najkracem mogucem roku.
    - Ako proces sa pravom upisa ceka na objekat, tada tom objektu ne sme da pristupi nijedna drugi proces koji ima samo pravo citanja

— Resenje prvog problema citanja-upisa

- Procesi sa pravom citanja dele sledece strukrure podataka

```
semaphore mutex, wrt;
int readcount;
```

- Smafori mutex i wrt su inicijalizovani na 1, a readcount je inicijalizovan na 0.
- Semafor wrt je zajednicki za procese sa pravom upisa i procese sa pravom upisa i citanja.
- Semafor mutex se koristi da bi osigurao medjusobnu iskljucivost kada se azurira promenljiva readcount.
- Promenljiva readcount predstavlja broj procesa koji trenutno ocitavaju zajednicki objekat.
- Semafor wrt sluzi da obezbedi medjusobnu isklucivost za procese sa pravom citanja i upisa.

Semafori
- Problem citanja – upisa -


```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
}while (TRUE);
```

The structure of a writer process.

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
}while (TRUE);
```

The structure of a reader process.

Semafori
- Problem citanja – upisa -

— Treba uociti

- Ako je objektu vec pristupio proces sa pravom upisa, tada n – 1 procesa sa pravom citanja ceka na semafor mutex a 1 na semafor wrt.
- Kada proces sa pravom upisa oslobodi zajednicki objekat, alokator odlucjuje koji ce sledeci objekat da pristupi objektu.
- Problem: proces sa pravom upisa moze da ceka veoma dugo na pristup zajednickom objektu.

Semafori

1010011
1110100
1100001
1010110

```
semWait(s)
{
   while (c                    TestAndSet(s.flag)==1                  )
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set
s.flag to 0) */;
   }
   s.flag = 0;
}

semSignal(s)
{
   while (c                    TestAndSet(s.flag)==1                  )
      /* do nothing */;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   s.flag = 0;
}
```

(a) Compare and Swap Instruction

```
semWait(s)
{
   inhibit interrupts;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow inter-
rupts */;
   }
   else
      allow interrupts;
}

semSignal(s)
{
   inhibit interrupts;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

(b) Interrupts

Monitori

- Iako semafori predstavljaju jednostavan i efektivan nacin za sinhornizaciju procesa, njihova nepravilna upotreba moze dovesti do gresaka koje je tesko otkriti.
- Da bi se ovi problemi prevazisli uvedene su konstrukcije u okviru HLL-a koje omogucavaju resavanje problema sinhronizacije procesa.
- Monitor predstavlja slozeni objekat koji enkapsulira zajednicke podatke i javne (public) metode za obradu tih podataka.
- Monitor sadrzi skup operacija koje zadovoljavaju uslov uzajamne iskljucivosti u okviru monitora.
  - Od procedura koje su definisane unutar monitora, u jednom trenutku se moze izvrsavati samo jedna od njih -> zadovoljen uslov medjusobne iskljucivosti.
  - Programer ne mora da brine o sinhronizaciji.

Monitori

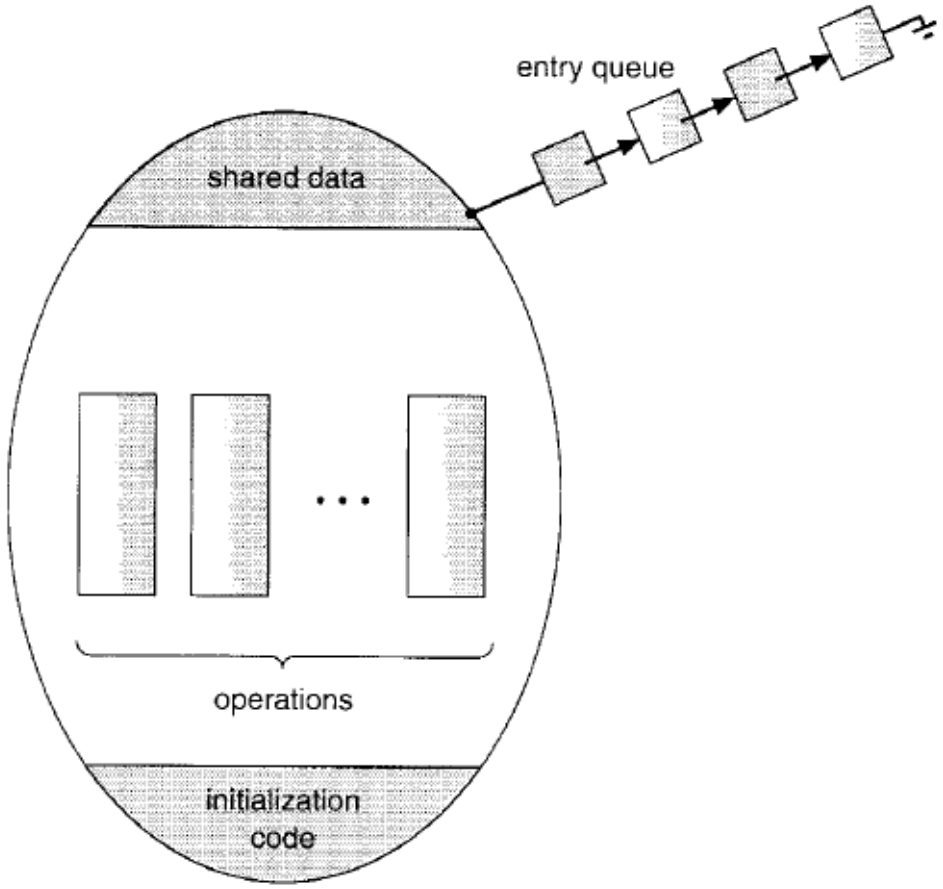

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```



Monitori

- Kada neki proces pozove monitor, on proverava da li neka druga procedura trenutno koristi monitor i ako je tako, suspenduje proces.
- Sta se desava kada proces koji koristi monitor ne moze da nastavi sa svojim izvrsavanjem?
  - Uvode se konstrukcije *condition*.
  - Programer definise jednu ili vise promneljivih tipa condition

    condition x, y;
  - Jedine operacije koje se mogu izvrsavati nad promenljivama ovog tipa jesu wait() i signal()
  - Proces koji izvrsava operaciju x.wait() ce biti suspendovan sve dok neki drugi proces ne izvrsi operaciju x.signal()
  - Pozivom operacije x.signal() se nastavlja sa izvrsavanjem samo jedne operacije
    - ☑ Ako nijedan proces nije suspendovan tada ova operacija nema nikakvog efekta.
    - ☑ U slucaju semafora, operacija signal() ucek utice na njegovu vrednost.

Monitori

- Proces koji koristi monitor i treba da suspenduje svoje izvrsavanje i prepusti monitor drugom procesu izvrsava operaciju *signal*.
  - Moze se desiti da imamo dva procesa koji istovremeno pristupaju monitoru.
  - Sta se desava nakon izvrsenja operacije signal?
    - ☑ Opcija 1. -> Drugi proces moze da nastavi sa svojim izvrsavanjem.
    - ☑ Opcija 2. -> Nakon izvrsenja operacije signal, proces mora da napusti monitor.
      - Opcija 2. je jednostavnija i laksa za implementaciju.
    - ☑ Opcija 3. -> Proces ceka na ispunjenje uslova za nastavak izvrsavanja i ne predaje monitor drugom procesu. Tek kada zavrsi sa izvrsavanjem predaje monitor drugom procesu.

Monitori

— Problem filozofa koji rucaju

- Svaki proces (filozof) izvrsava sledece operacije:

```
dp.pickup(i);
    ...
    eat
    ...
dp.putdown(i);
```

- Monitor koji kontrolise pristup podacima se definise na sledeci nacin -->

```
monitor dp
{
   enum {THINKING, HUNGRY, EATING}state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
         state[i] = EATING;
         self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

Monitori
-Producer/Consumer-


```
void producer()
{
      char x;
      while (true) {
      produce(x);
      append(x);
      }
}
void consumer()
{
      char x;
      while (true) {
      take(x);
      consume(x);
      }
}
void main()
{
      parbegin (producer, consumer);
}
```

Monitori
-Producer/Consumer-

1010011
1110100
1100001
1010110

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                /* buffer pointers */
int count;                                          /* number of items in buffer */
cond notfull, notempty;                 /* condition variables for synchronization */

void append (char x)
{
      if (count == N) cwait(notfull);           /* buffer is full; avoid overflow */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;
      /* one more item in buffer */
      csignal (nonempty);                          /*resume any waiting consumer */
}
void take (char x)
{
      if (count == 0) cwait(notempty);        /* buffer is empty; avoid underflow */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                       /* one fewer item in buffer */
      csignal (notfull);                           /* resume any waiting producer */
}
{                                                                  /* monitor body */
      nextin = 0; nextout = 0; count = 0;              /* buffer initially empty */
}
```

Monitori

- Monitori predstavljaju konstrukciju unutar programskog jezika i njihovu implementaciju vrši kompajler.
  - Programer se oslobađa detalja implementacije.
- Monitori i semafori su projektovani da reše problem sinhronizacije na jednom ii više CPU koji dele zajedničku memoriju.
- Ova rešenja postaju neprimenljiva za slučaj da se radi o distribuiranom sistemu gde imamo nezavisne računarske sisteme sa nizavisnim memorijama.
  - Imeplemntacija semafora zahteva direktnu podršku u hardveru (zbog implementacije wait i signal funkcija).
  - Monitori zavise od izbora programskog jezika.

Razmena poruka

- Prednost primene mehanizama za razmenu poruka za sinhrnoizaciju procesa jeste mogućnost implementacije kako u distribuiranim isstemima, tako i na multiprocesorskim sistemima sa zajedničkom memorijom.
- Za komunikaciju se koriste dve funkcije koje su slično semofarima a za razliku od monitora implemeitrane u obliku sistemskog poziva:
  - send(destination, message)
    - ☑ Nakon slanja poruke, proces može da bude blokiran ili ne.
  - receive(source, message)
    - ☑ Ako je poruka prethodno poslata, proces prihvata poruku i nastavlja sa izvršavanjem.
    - ☑ Proces može da se blokira dok poruka ne stigne ili da nastavi sa izvršavanjem i odustane od pokušaja da primi poruku.

Razmena poruka

1010011
1110100
1100001
1010110

— Problemi:

- Kod neblokirajućeg send može doći ako dođe do greške kod procesa onda je moguće da generiše veili broj poruka i optereti resurse sistema.
- Programer mora da obezbedi prijem poruka.
  - ☑ Primena poruka za potvrdu prijema (ACK).
- Kod blokirajućeg receive nastaje problem u slučaju gubitka poruke. Proces može ostati veoma dugo blokiran.
- Kod neblokirajućeg receive može doći do gubitka poruke ako se instrukcija receive izvrši pre slanja poruke.
  - ☑ Uvodese mehanizmi da se prvo testira da li je poruka poslata pa se tek onda izvršava receive.

Razmena poruka

1010011
1110100
1100001
1010110

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build _message(&m, item);             /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Razmena poruka

- U slučaju indirektne komunikacije procesa, poruke se razmenjuju prko mailbox-a.
  - Kada proces pokuša da pošalje poruku u mailbox koji ej pun, proces se blokira dok se mailbox ne isprazni i oslobodi prostor za novu poruku.
- Problem producer/consumer možemo rešiti na sledeći način:
  - I Producer i Consumer imaju posebne mailboxovepreko kojih primaju poruke.
    - ☑ Mogu sltai/primati poruke od različitih procesa.
    - ☑ Mailbox sadrži poruke koje su primljene ali još nisu prihvaćene.

Razmena poruka

1010011
1110100
1100001
1010110

```
const int
    capacity = /* buffering capacity */ ;
    null = /* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce,pmsg);
     pmsg = produce();
     send (mayconsume,pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume,cmsg);
     consume (cmsg);
     send (mayproduce,null);
    }
}
void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1;i <= capacity;i++) send (mayproduce,null);
    parbegin (producer,consumer);
}
```

1010011
1110100
1100001
1010110

Semafori
- Blokiranje i Izgladnjivanje -

— Moze se desiti da dva procesa cekaju da onaj drugi izvrsi operaciju signal()
  - Njihovo izvrsavanje je blokirano

— Primer:
  - Posmatramo dva procesa koji pristupaju dva semafora S i Q cija je pocetna vrednost postavljena na 1.

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`