Konkurentost

Operativni sistemi 1

Institut za matematiku i informatiku
Prirodno-matematički fakultet, Kragujevac

doc. dr Miloš Ivanović
prof. dr Dragan Stojanović

Novembar 2026. god.

O čemu će biti reči?

1. Pojmovi
2. Softverska realizacija k.s.
3. Hardverska realizacija k.s.
4. Semafori
   - Proizvođač/potrošač
5. Monitori
6. Poruke
7. Čitaoci/pisci

Ključni pojmovi

- **Multiprogramiranje** - upravljanje sa više procesa u jednoprocesorskom sistemu
- **Multiprocesiranje** - upravljanje sa više procesa unutar multiprocesora
- **Distribuirana obrada** - procesi na više računarskih sistema

Ključni pojmovi

1. **Kritična sekcija** - Deo koda koji zahteva pristup deljenim resursima i ne može se izvršiti dok je drugi proces u svojoj kritičnoj sekciji.
2. **Uzajamna blokada** (*deadlock*) - Zastoj jer svaki proces čeka da onaj drugi nešto uradi
3. **Živa blokada** (*livelock*) - Dva ili više procesa konstantno menjaju svoje stanje kao odgovor na akcije drugog procesa ne radeći ništa korisno
4. **Uzajamno isključivanje** (*mutual exclusion*) - Zahtev da nijedan proces ne može biti u svojoj kritičnoj sekciji kada je jedan proces u svojoj krit. sekciji
5. **Stanje trke** (*race condition*) - Rezultat zavisi od relativnog vremena izvršavanja
6. **Gladovanje** (*starvation*) - Raspoređivač (dispečer) uporno preskače dati proces iako se on nalazi u *Ready* stanju

Primer stanja trke (haotično stanje)

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

Process P1	Process P2
•	•
`chin = getchar();`	•
•	`chin = getchar();`
`chout = chin;`	`chout = chin;`
`putchar(chout);`	•
•	`putchar(chout);`
•	•

Pitanja

1. Šta se dešava ako dva procesa pokušaju da pristupe proceduri `echo()` na jednoprocesorskom sistemu?
2. A ako proceduru `echo` proglasimo kritičnim resursom (sekcijom)?
3. Šta se dešava ako su dostupna 2 procesora?

Uzajamno delovanje procesa

1. **Procesi nisu svesni drugih procesa** - Postoji odnos nadmetanja. Mogu se javiti problemi uzajamnog isključivanja, uzajamne blokade i gladovanja (*stravation*).
2. **Procesi su indirektno svesni postojanja drugih procesa (npr. deljeni objekti)** - Rezultati jednog procesa zavise od informacija koje daje neki drugi. Problemi su uzajamno isključivanje, blokada, gladovanje i povezanost podataka.
3. **Procesi su direktno svesni drugih procesa** - Procesi su projektovani tako da kooperiraju na istom zadatku. Problemi koji se mogu javiti su uzajamna blokada i gladovanje.

Načini kooperacije procesa

1. **Deljenje** - Preko deljenih promenljivih (objekata) i fajlova
2. **Putem komunikacije** - Procesi međusobno komuniciraju putem poruka

Primer kooperacije procesa deljenjem

```
P1:
    a+=1;
    b+=1;
P2:
    b*=2;
    a*=2;
Sekvenca:
    a+1;
    b*=2;
    b+=1;
    a*=2;
```

Na kraju sekvence $a \neq b$.

Problem konkurentnosti

Dve niti inkrementiraju counter

```c
#include <stdio.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

static volatile int counter = 0;

// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n",
           counter);
    return 0;
}
```

```
prompt> gcc -o main main.c -Wall -pthread; ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Uzrok problema?

Uzrok problema

Inkrementiranje counter nije atomično

counter = counter +1



```
100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c
```

OS	Thread 1	Thread 2	(after instruction) PC	eax	counter
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

Kritična sekcija

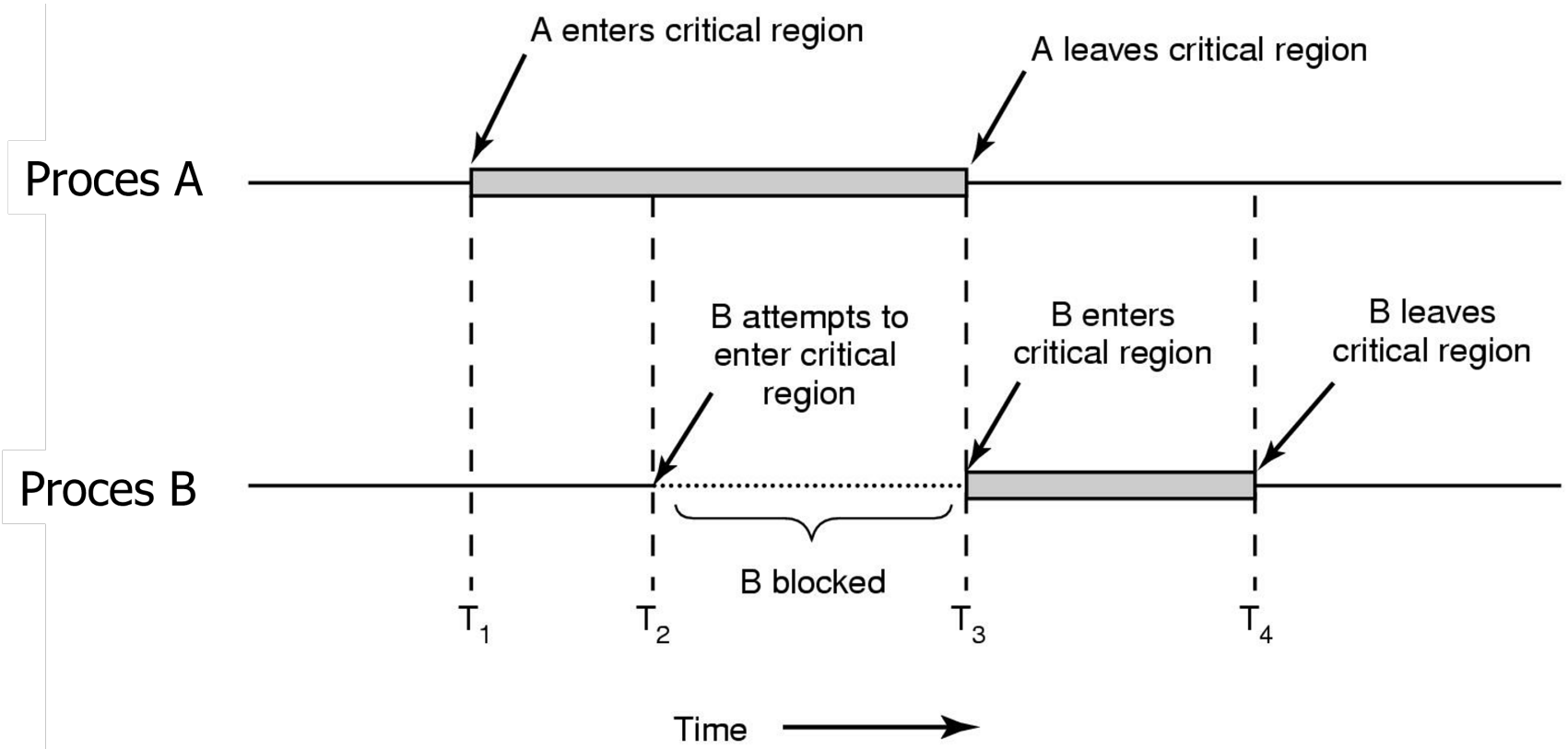
```
do {
    entry section
        critical section
    exit section
    /* ostatak koda */
} while (1);
```

Kako se kritična sekcija može realizovati?

1. softverski
2. hardverski
3. pomoću semafora
4. pomoću OS-a (sistemskim pozivima)
5. višim programskim strukturama za sinhronizaciju (monitorima)

Uzajamno isključivanje

Dva procesa u izvršavaju kod kritične sekcije



Softverska realizacija kritične sekcije

Zahtevi

1. Međusobno isključenje
2. Proces koji je izvan svoje k. sekcije ne može sprečiti druge procese da uđu u svoje k. sekcije
3. Proces ne sme neograničeno dugo čekati da uđe u svoju k. sekciju
4. Proces ne sme neograničeno dugo ostati u svojoj k. sekciji

Algoritam striktne alternacije

Softverska realizacija kritične sekcije

Proces 0:

```
do {
    while (turn!=0);    // ulazna sekcija
    /* kriticna sekcija */
    turn = 1;
    /* ostatak koda */
} while (true);
```

Proces 1:

```
do {
    while (turn!=1);    // ulazna sekcija
    /* kriticna sekcija */
    turn = 0;
    /* ostatak koda */
} while (true);
```

Nedostaci

1. Zaposleno čekanje troši procesorsko vreme
2. Moguća je blokada ako neki od procesa zaglavi u ostatku koda

Kritična sekcija bez stroge alternacije

Softverska realizacija kritične sekcije

```
int flag[2];
flag[0] = flag[1] = 0;

do {
    flag[i]=1;
    while (flag[j]);   /ulazna sekcija
    /* kriticna sekcija */
    flag[i] = 0;
    /* ostatak koda */
} while (true);
```

Nedostaci

1. Moguća uzajamna blokada koja procese ostavlja u beskonačnoj petlji

Dekker-Petersenov algoritam

Softverska realizacija kritične sekcije

1. Proces P_j želi da uđe u svoju kritičnu sekciju ($flag[j] = 1$)
2. Procesu P_j data je šansa da uđe u svoju kritičnu sekciju ($turn = j$)

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn==j);
    /* kriticna sekcija */
    flag[i] = 0;
    /* ostatak koda */
} while (true);
```

Zadovoljeni uslovi

1. Ispoštovano međusobno isključenje preko `turn`
2. Nema striktne alternacije pa time ni blokade

Pekarski algoritam

Uopštenje Dekker-Petersena na sistem sa N procesa

Pravila pekarskog algoritma

1. Sekvenca brojeva se generiše u rastućem redosledu i dodeljuje procesima (recimo, 1,2,3,3,4,5,5,...)
2. Procesi se opslužuju redom od najmanjeg do najvećeg dodeljenog broja.
3. Ako dva procesa dobiju iste brojeve, prvo se opslužuje onaj proces sa nižim indeksom.

Leksikografski poredak kod pekarskog algoritma

1. $(a, b) < (c, d)$ ako je $a < c$ ili ako je $a = c \wedge b < d$
2. $max(a_0, ..., a_{n-1})$ je broj a_k, takav da je $a_k \geq a_i$ za svako $i \in [0, n-1]$

Zajednički podaci za pekarski algoritam

- `int biranje[n]; // logicka promenljiva za medjusobno iskljucivanje`
- `int broj[n]; // za smestanje dodeljenih brojeva`

Pekarski algoritam - nastavak

Softverska realizacija kritične sekcije

```
do {
    biranje[i] = 1;
    broj[i] = max (broj[0],broj[1],...,broj[n-1])+1;
    biranje[i] = 0;
    for (j=0; j<n; j++) {
        while (biranje[j]);
        while ( (broj[j]!=0) && ((broj[j],j) < (broj[i],i)) );
    }
    /* kriticna sekcija */
    broj[i] = 0;
    /* ostatak koda */
} while (true);
```

Proces ostaje van kritične sekcije sve dok postoji i jedan proces sa većim brojem po leksikografskom poretku. Po izlasku iz kritične sekcije, anulira se broj i proces se više ne takmiči za k.s.

Hardverska realizacija kritične sekcije

Realizacija prekidima

Moguće je realizovati kritičnu sekciju i onemogućavanjem prekida, ali je takva realizacija neefikasna. Zašto?

```
boolean testset(int i) {
    if (i==0) {
        i=1;
        return true;
    } else
        return false;
}
```

Atomske instrukcije

Instrukcija `testset()` je atomska, tj. izvršava se u jednom komadu i ne može se prekinuti.

Hardverska realizacija kritične sekcije

Atomska instrukcija razmene `exchange()`

Neki procesori imaju nedeljivu instrukciju razmene podataka u 2 registra ili između registra i memorije. Na primer, Intelovi procesori imaju instrukciju **XCHG**.

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (!testset(bolt))
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));

}
```

```
   /* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true) {
      do exchange (keyi, bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ..., P(n));
}
```

Osobine hardverske realizacije kritične sekcije

Ako je n broj procesa, važi da je:

$$bolt + \sum_i key_i = n$$

Nedostaci hardverske realizacije

1. **Upotrebljava se stanje zauzet na čekanju**, i time nepotrebno troši proc. vreme
2. **Moguće je gladovanje** - slučajno se bira proces koji će biti raspoređen na izlazu iz k.s.
3. **Moguće je uzajamno blokiranje** - recimo na jednoprocesorskom sistemu sa šemom prioriteta

Opšti i binarni semafor

Operacije

Semafori

Semafori su jednostavan mehanizam OS-a za razmenu signala između procesa. Da bi se poslao signal, izvršava se `SemSignal(s)`, a da bi se primio `SemWait(s)`.

Opšti semafor

1. Semafor se može inicijalizovati na pozitivnu vrednost
2. **semWait()** umanjuje vrednost semafora. Ako postane negativna, proces se blokira. Ako ostane pozitivna, nastavlja se.
3. **semSignalB()** uvećava vrednost semafora. Ako postane ≤ 0, vrši se deblokada procesa koji je blokiran pomoću **semWait()**

Binarni semafor (*mutex*)

1. Semafor se može inicijalizovati na 0 ili 1
2. **semWaitB()** proverava vrednost semafora. Ako je 0, blokira se proces koji je pozvao **semWaitB()**. Ako je 1, menja se na 0 i nastavlja se dalje.
3. **semSignalB()** proverava da li je neki proces na datom semaforu blokiran. Ako jeste, deblokira se. Ako nema blokiranih procesa, vrednost semafora se postavlja na 1.

Opšti i binarni semafor

Struktura-

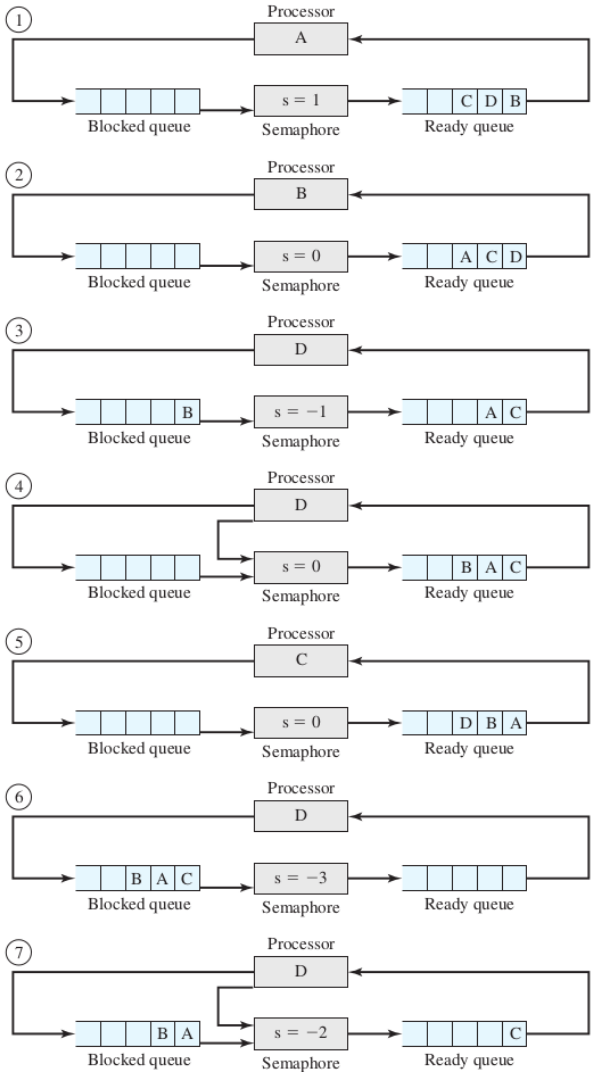
Slika: **Levo:** Struktura opšteg semafora **Desno:** Struktura binarnog semafora

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
           /* place this process in s.queue */;
           /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
           /* remove a process P from s.queue */;
           /* place process P on ready list */;
      }
}
```

```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};
void semWaitB(binary_semaphore s)
{
      if (s.value == one)
           s.value = zero;
      else {
                /* place this process in s.queue */;
                /* block this process */;
      }
}
void semSignalB(semaphore s)
{
      if (s.queue is empty())
           s.value = one;
      else {
                /* remove a process P from s.queue */;
                /* place process P on ready list */;
      }
}
```

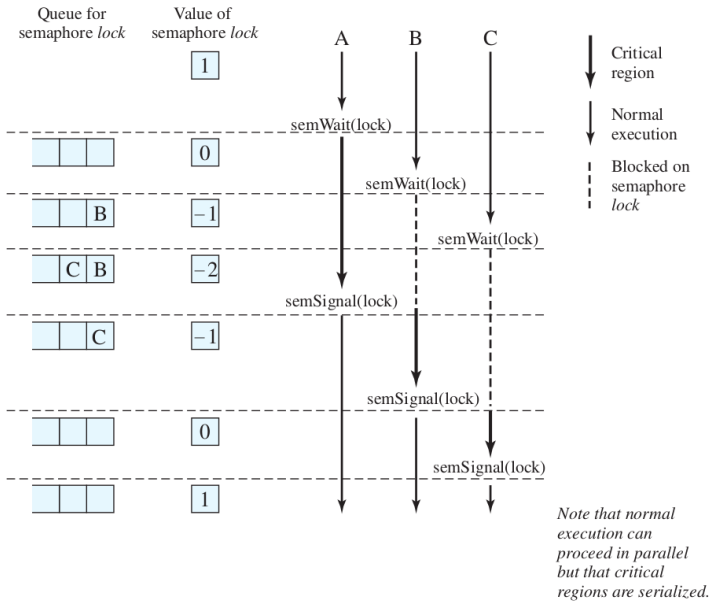
Primer korišćenja semafora

Procesi A,B i C čekaju podatke koje obezbeđuje proces D



Uzajamno isključivanje korišćenjem semafora

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
      while (true) {
            semWait(s);
            /* critical section    */;
            semSignal(s);
            /* remainder    */;
      }
}
void main()
{
      parbegin (P(1), P(2), . . ., P(n));
}
```



Pitanje

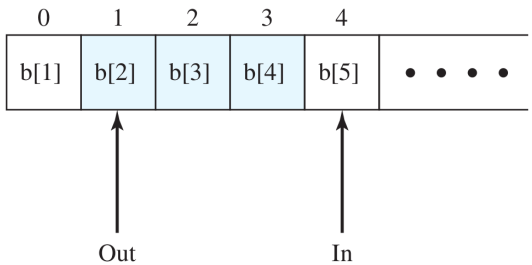
Kako se postiže međusobno isključivanje ako je zahtev da više od jednog procesa može da bude u svojoj krtičnoj sekciji?

Problem proizvođač/potrošač

Pretpostavka o beskonačnom baferu

```
producer:
while (true) {
   /* produce item v */;
   b[in] = v;
   in++;
}
```

```
consumer:
while (true) {
   while (in <= out)
        /* do nothing */;
   w = b[out];
   out++;
   /* consume item w */;
}
```



Producer/consumer

Da bi moglo da se izvrši "konzumiranje", potrebno je da je $in > out$. Broj objekata u baferu može se označiti sa $n(= in - out)$.

Problem proizvođač/potrošač - POGREŠNO!

Pretpostavka o beskonačnom baferu

```
/* program producerconsumer */
      int n;
      binary_semaphore s = 1, delay = 0;
      void producer()
      {
           while (true) {
                 produce();
                 semWaitB(s);
                 append();
                 n++;
                 if (n==1) semSignalB(delay);
                 semSignalB(s);
           }
      }
```

```
void consumer()
{
      semWaitB(delay);
      while (true) {
            semWaitB(s);
            take();
            n--;
            semSignalB(s);
            consume();
            if (n==0) semWaitB(delay);
   }
}
void main()
{
      n = 0;
      parbegin (producer, consumer);
}
```

Pitanje

Koji scenario izvršenja dovodi do konzumiranja nepostojećeg objekta?

Problem proizvođač/potrošač - POGREŠNO!

Scenario izvršenja koji dovodi do greške

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	–1	0
21		semiSignlaB(s)	1	–1	0

Pitanje

Da li se problem može rešiti uvođenjem `if (n==0) semWaitB()` u kritičnu sekciju?

Problem proizvođač/potrošač - ISPRAVNO!

Levo: Korišćenjem binarnih semafora, Desno: standardni semafori

```
/* program producerconsumer */
      int n;
      binary_semaphore s = 1, delay = 0;
      void producer()
      {
            while (true) {
                  produce();
                  semWaitB(s);
                  append();
                  n++;
                  if (n==1) semSignalB(delay);
                  semSignalB(s);
            }
      }
      void consumer()
      {
            int m; /* a local variable */
            semWaitB(delay);
            while (true)  {
                  semWaitB(s);
                  take();
                  n--;
                  m = n;
                  semSignalB(s);
                  consume();
                  if (m==0) semWaitB(delay);
            }
      }
      void main()
      {
            n = 0;
            parbegin (producer, consumer);
      }
```

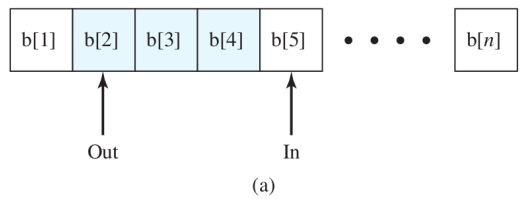
```
/* program  producerconsumer */
      semaphore n = 0, s = 1;
      void producer()
      {
            while (true) {
                  produce();
                  semWait(s);
                  append();
                  semSignal(s);
                  semSignal(n);
            }
      }
      void consumer()
      {
            while (true) {
                  semWait(n);
                  semWait(s);
                  take();
                  semSignal(s);
                  consume();
            }
      }
      void main()
      {
            parbegin (producer, consumer);
      }
```

Pitanje

Koje je rešenje elegantnije?

Problem proizvođač/potrošač - ograničeni kružni bafer

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed




```
producer:
while (true) {
   /* produce item v */
   while ((in + 1) % n == out)
       /* do nothing */;
   b[in] = v;
   in = (in + 1) % n;
}
```

```
consumer:
while (true) {
   while (in == out)
       /* do nothing */;
   w = b[out];
   out = (out + 1) % n;
   /* consume item w */;
}
```

Problem proizvođač/potrošač

Rešenje sa ograničenim baferom

```
/* program boundedbuffer */
      const int sizeofbuffer = /* buffer size */;
      semaphore s = 1, n = 0, e = sizeofbuffer;
      void producer()
      {
            while (true) {
                  produce();
                  semWait(e);
                  semWait(s);
                  append();
                  semSignal(s);
                  semSignal(n);
            }
      }
      void consumer()
      {
            while (true)  {
                  semWait(n);
                  semWait(s);
                  take();
                  semSignal(s);
                  semSignal(e);
                  consume();
            }
      }
      void main()
      {
            parbegin (producer, consumer);
      }
```

Kako implementirati semafore?

Levo: `testset()` isntrukcija, Desno: Onemogućavanje prekida

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

```
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;
}
```

Pitanje

Šta su slabosti `testset()` instrukcije, a šta slabosti onemogućavanja prekida?

Monitori - osnovne osobine

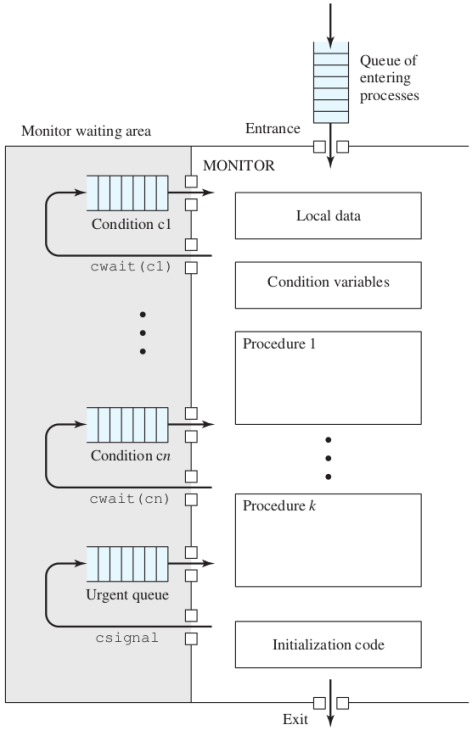
Šta su to monitori?

Semafori predstavljaju primitivan sistem kontrole uzajamnog isključivanja i sinhronizacije. Zašto? U nekim programskim jezicima postoje strukture koje se zovu monitorima koje rešavaju problem kapsulacijom funkcionalnosti. **Sastoji se iz procedura, init sekvence i lokalnih podataka.**

Osobine monitora

1. Lokalnim promenljivama se može pristupiti isključivo preko procedura monitora.
2. Proces ulazi u monitor pozivanjem jedne od svojih procedura.
3. U monitoru se istovremeno može izvršavati samo jedan proces.

Monitori - primer proizvođač/potrošač




```
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);
}
```

Kondicione varijable - operacije

1. **cwait(c)**: suspenduj izvršavanje procesa koji je pozvao. Monitor je sda na raspolaganju za bilo koji drugi proces.
2. **csignal(c)**: nastavi izvršavanje procesa koji je blokiran pozivom **cwait()**. Ako postoji više takvih procesa, izaberi jedan. Inače ne radi ništa.

Monitori - primer proizvođač/potrošač

Realizacija procedura u monitoru

```
/* 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 */
}
```

Prosleđivanje poruka

Sva osnovna zahteva za procese su postizanje **sinhronizacije** i **komunikacije**.

Primitive

1. `send(odrediste, poruka)`
2. `receive(izvor, poruka)`

Pitanje

Koja je osnovna prednost prosleđivanja poruka nad prethodno pomenutim mehanizmima? Na kakvim sistemima sve može da radi?

Prosleđivanje poruka

Projektne karakteristike

Sinhronizacija
- Slanje
  - blokirano
  - neblokirano
- Primanje
  - blokirano
  - neblokirano
  - ispitivanje prijema

Format
- Sadržaj
- Dužina
  - fiksna
  - promenljiva

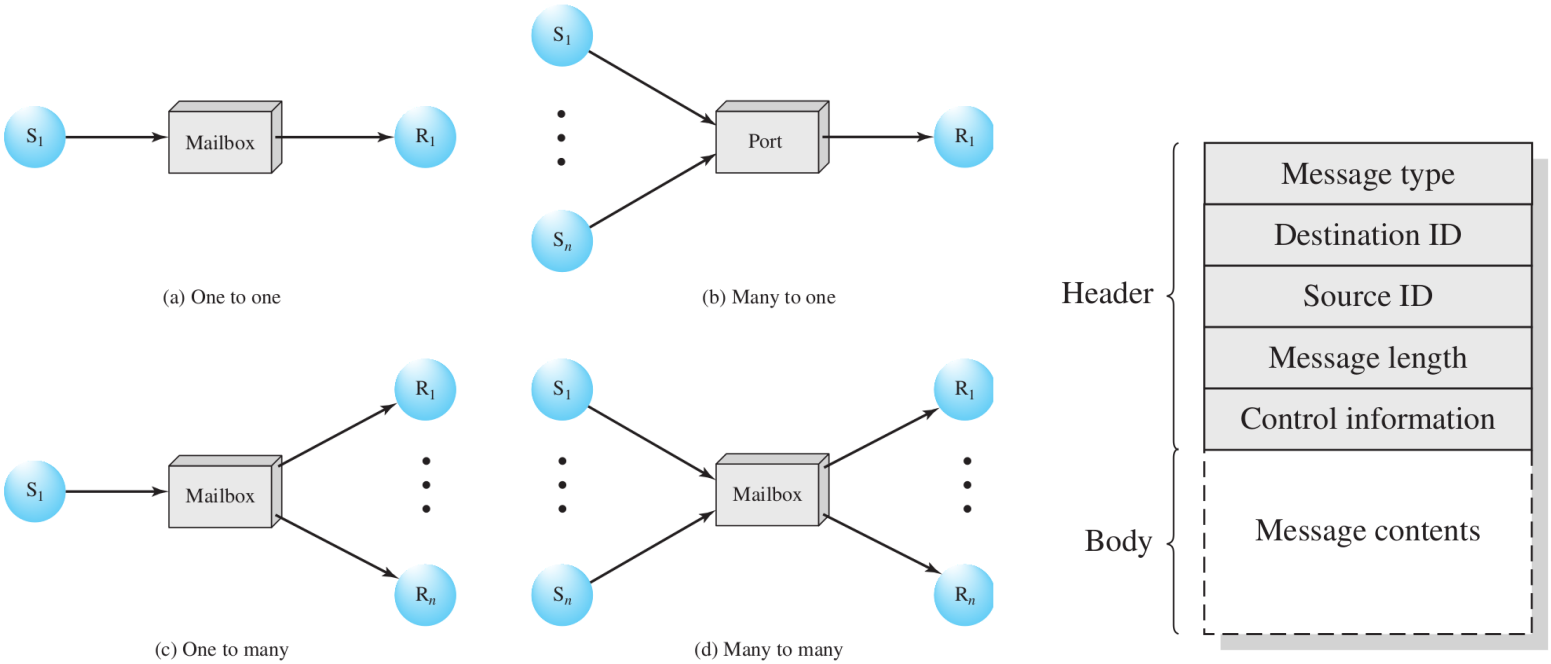
Adresiranje
- Direktno
  - slanje
  - primanje
    - eksplicitno
    - implicitno
- Indirektno
  - statičko
  - dinamičko
  - vlasništvo

Disciplina redova
- FIFO
- po prioritetu

Prosleđivanje poruka

Projektne karakteristike

Slika: Levo: Indirektna komunikacija procesa. Desno: Opšti format poruke



Uzajamno isključivanje tehnikom poruka

```
/* program uzajamno iskljucivanje */
const int n = /* broj procesa */
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);
       /* critical section */;
       send (box, msg);
      /* remainder */;
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Digresija

U oblasti računarskih mreža, prosleđivanje *tokena* se primenjuje u velikoj meri (*token ring*).

Problem proizvođača/potrošača tehnikom poruka

```
const int
capacity = /* kapacitet bafera */ ;
null = /* prazna poruka */ ;
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);
}
```

Problem čitalaca i pisaca

1. Bilo koji broj čitalaca može simultano da čita dokument
2. Samo jedan pisac u jednom trenutku može da upisuje u dokument
3. Ukoliko pisac upisuje, nijedan čitalac ne može da čita

Pitanja

1. Da li se ovaj problem može svesti na problem uzajamnog isključivanja?
2. Da li se ovaj problem može svesti na proizvođača/potrošača?

Rešenje kada čitaoci imaju prednost

Problem čitalaca i pisaca

```
/* program readersandwriters */
int readcount;
semaphore x = 1,wsem = 1;
void reader()
{
    while (true){
     semWait (x);
     readcount++;
     if(readcount == 1)
         semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount;
     if(readcount == 0)
         semSignal (wsem);
     semSignal (x);
    }
}
```

```
void writer()
{
    while (true){
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader,writer);
}
```

Pitanje

Čemu služi semafor **x**, a čemu semafor **wsem**?

Rešenje kada pisci imaju prednost

Problem čitalaca i pisaca

Problem prethodnog rešenja je u tome što dok god ima i jedog čitaoca, pisac neće moći da pristupi. Može doći do **gladovanja**.

Novi semafori

1. Semafor **rsem** sprečava sve čitaoce dok god ima i jednog pisca koji pristupa dokumentu
2. Promenljiva **writecount** kontroliše postavljanje **rsem**
3. Semafor **y** kontroliše ažuriranje promenljive **writecount**
4. Semafor **z** je potreban da se ne bi stvorio dug red čitalaca ispred **rsem**. Samo jedan čitalac sme da čeka ispred **rsem**, dok svi ostali čekaju ispred **z**.

Rešenje kada pisci imaju prednost

Problem čitalaca i pisaca

```
/* program readersandwriters */
int readcount,writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true){
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1)
                         semWait (wsem);
                    semSignal (x);
               semSignal (rsem);
          semSignal (z);
          READUNIT();
          semWait (x);
               readcount--;
               if (readcount == 0) semSignal (wsem);
          semSignal (x);
    }
}
```

```
void writer ()
{
    while (true){
          semWait (y);
               writecount++;
               if (writecount == 1)
                    semWait (rsem);
          semSignal (y);
          semWait (wsem);
          WRITEUNIT();
          semSignal (wsem);
          semWait (y);
               writecount;
               if (writecount == 0) semSignal (rsem);
          semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Pitanje

Šta se dešava ako su prisutni samo čitaoci? A samo pisci?

Rešenje kada pisci imaju prednost

Problem čitalaca i pisaca

Samo čitaoci u sistemu	• postavljen semafor **wsem** • nema redova čekanja
Samo pisci u sistemu	• postavljeni semafori **wsem** i **rsem** • pisci prave red pred semaforom **wsem**
I čitaoci i pisci, prvo čitanje	• semafor **wsem** postavlja čitalac • semafor **rsem** postavlja pisac • pisci prave red pred semaforom **wsem** • jedan čitalac se nalazi ispred **rsem** • ostali čitaoci prave red ispred **z**
I čitaoci i pisci, prvo upisivanje	• semafor **wsem** postavlja pisac • semafor **rsem** postavlja pisac • jedan čitalac se nalazi ispred **rsem** • ostali čitaoci prave red ispred **z**

Rešenje problema čitalaca i pisaca preko poruka

Problem čitalaca i pisaca

Sistem se sastoji iz tri različite vrste komponenti: **čitalaca**, **pisaca** i jedinstvenog **kontrolera**.

Promenljiva *count*

1. Ako je $count > 0$, onda nema pisaca koji čekaju na svoj red.
2. Ako je $count = 0$, jedini zahtev koji čeka je zahtev pisca. Piscu treba dozvoliti pristup i sačekati poruku *finished*.
3. Ako je $count < 0$, tada je pisac napravio zahtev, ali mora da čeka da završe svi aktivni čitaoci. Iz tog razloga, primaju se samo *finished* poruke.

Rešenje problema čitalaca i pisaca preko poruka

Problem čitalaca i pisaca

```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
}
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = j;
      send (writerequest, rmsg);
      receive (mbox[j], rmsg);
      WRITEUNIT ();
      rmsg = j;
      send (finished, rmsg);
   }
}
```

```
void  controller()
{
      while (true)
      {
         if (count > 0) {
            if (!empty (finished)) {
               receive (finished, msg);
               count++;
            }
            else if (!empty (writerequest)) {
               receive  (writerequest,  msg);
               writer_id = msg.id;
               count = count - 100;
            }
            else if (!empty (readrequest)) {
               receive (readrequest, msg);
               count--;
               send (msg.id, "OK");
            }
         }
         if (count == 0) {
            send (writer_id, "OK");
            receive (finished, msg);
            count = 100;
         }
         while (count < 0) {
            receive (finished, msg);
            count++;
         }
      }
}
```