Operativni sistemi

Konkurentnost: uzajamno isključivanje i sinhronizacija

Prof. dr Dragan Stojanović

Katedra za računarstvo
Univerzitet u Nišu, Elektronski fakultet

Literatura

- *Operating Systems: Internals and Design Principles*, edition, W. Stallings, Pearson Education Inc., 7th – 2012, (5th -2005, 6th - 2008, 8th – 2014 , 9th – 2017)
  - http://williamstallings.com/OperatingSystems/
  - http://williamstallings.com/OperatingSystems/OS9e-Student/

- Poglavlje 5: Konkurentnost: uzajamno isključivanje i sinhronizacija
- Dodatak A: Teme o konkurentnosti

OS i konkurentnost

- Centralna uloga OS je upravljanje procesima i nitima
  - Multiprogramiranje - Upravljanje više procesa unutar jednoprocesorskog sistema
  - Multiprocesiranje - Upravljanje više procesa unutar multiprocesora
  - Distribuirano procesiranje - Upravljanje više procesa koji se izvršavaju na distribuiranim računarskim sistemima
- Konkurentnost predstavlja značajni element u dizajnu OS, u okviru:
  - Dodele procesorskog vremena procesima
  - Podele resursa i nadmetanje za resursima
  - Sinhronizacije izvršavanja više procesa
  - Komunikacije između procesa

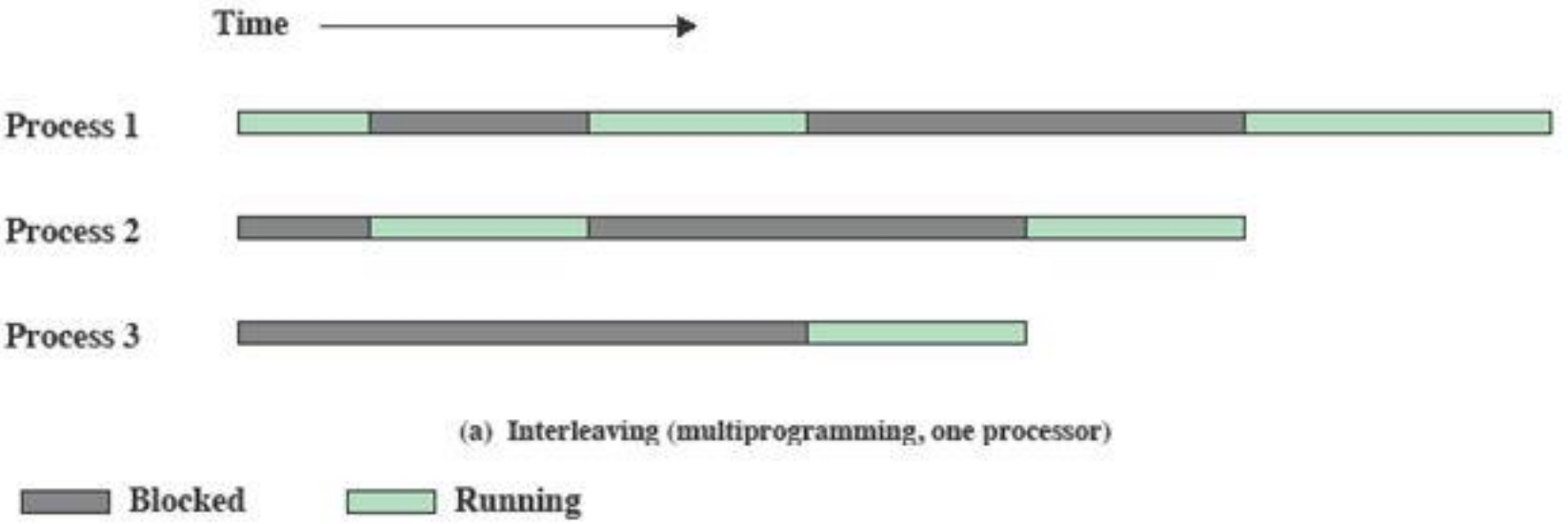
Kada se javlja konkurentnost?

Konkurentnost se javlja u tri različita konteksta:

- Višestruke aplikacije
  - Multiprogramiranje obezbeđuje deljenje vremena obrade između aktivnih aplikacija
- Struktuirane aplikacije
  - Aplikacije mogu biti programirane kao skup konkurentnih procesa
- Struktura operativnog sistema
  - Operativni sistem se implementira kao skup procesa ili niti

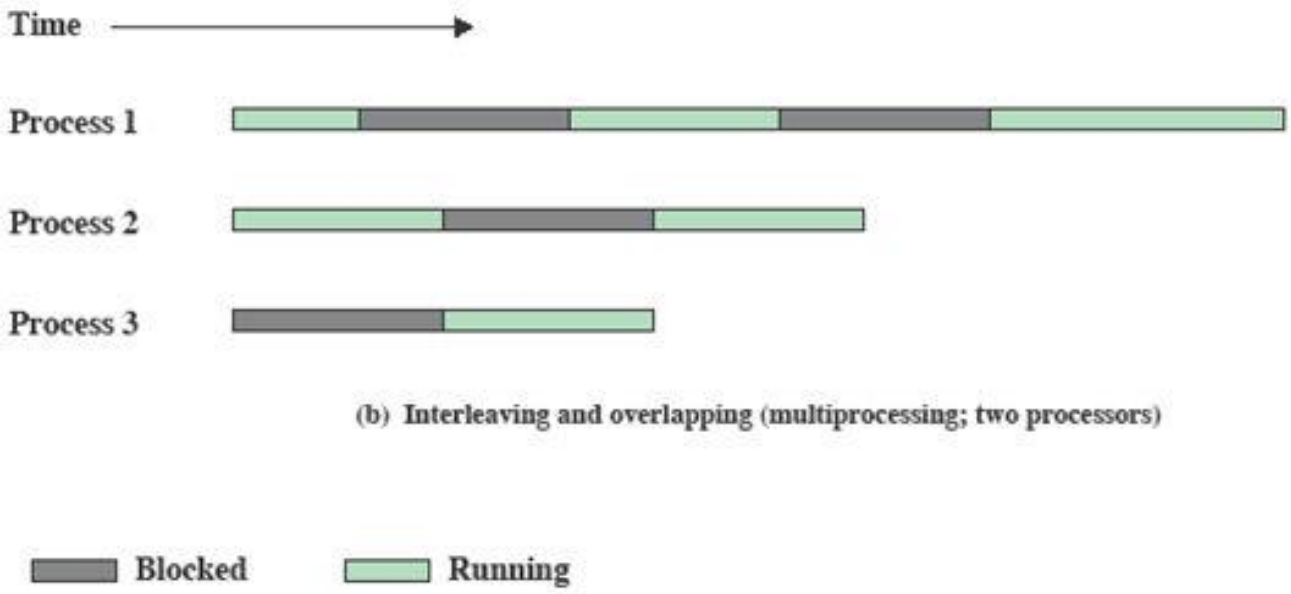
Principi konkurentnosti

- Preplitanje izvršenja procesa na jednoprocesorskom sistemu



Pirncipi konkurentnosti (2)

- Preklapanje izvršenja procesa na multiprocesorskom sistemu (2 procesora)



(b) Interleaving and overlapping (multiprocessing; two processors)

Problemi sa konkurentnošću

- Problemi potiču od osnovnih karakteristika multiprogramskih i multiprocesorskih sistema:
  - Relativna brzina izvršavanja procesa se ne može predvideti
- Problemi su:
  - Deljenje globalnih resursa
  - Operativnom sistemu je teško da optimalno upravlja dodelom resursa
  - Teško je otkrivanje grešaka u programiranju
- Rešenje za konkurentnost treba da obuhvati:
  - Komunikaciju među procesima
  - Deljenje resursa
  - Sinhronizaciju više procesa
  - Alokaciju vremena procesa za korišćenje resursa

Ključni pojmovi koji se odnose na konkurentnost

- Kritična sekcija (*critical section*)
  - Deo koda unutar procesa koji zahteva pristup deljenim resursima i ne može se izvršiti dok je drugi proces u odgovarajućem delu koda
- Uzajamno blokiranje, zastoj (*deadlock*)
  - Situacija u kojoj dva ili više procesa ne mogu da nastave sa radom jer svaki čeka da jedan od preostalih nešto uradi
- Zaključavanje uživo (*livelock*)
  - Situacija u kojoj dva ili više procesa konstantno menjaju svoje stanje kao odgovor na promene drugih procesa i pri tom ne rade ništa korisno
- Uzajamno isključivanje (*mutual exclusion*)
  - Zahtev po kome nijedan proces ne može biti u kritičnoj sekciji koja pristupa bilo kom deljenom resursu kada je jedan proces u kritičnoj sekciji pristupio deljenim resursima
- Uslov trke (*race condition,* haotično stanje)
  - Situacija u kojoj više niti ili procesa čita i upisuje stavke deljenih podataka, a krajnji rezultat zavisi od relativnog vremena izvršavanja
- Gladovanje (*starvation*)
  - Situacija u kojoj dispečer neprekidno preskače proces koji je spreman za izvršavanje, iako je spreman za rad, taj proces nikada neće biti odabran

Jednostavan primer konkurentnosti

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

- Posmatraćemo jednoprocesorski multiprogramski sistem koji podržava jednog korisnika
- Svaki proces može ponavljati poziv ove deljene procedure kako bi prihvatao ulaz sa tastature i prikazivao ga na ekranu korisnika
  - Jedan primerak ove procedure se učitava u glavnu memoriju i zajednička je za sve aplikacije
  - Promenljiva *chin* je globalna
- Korisnik se može kretati između više aplikacija, ali su tastatura i ekran zajednički za sve apikacije
  - Svaka aplikacija poziva jedanput, ili više puta ovu proceduru

Jednostavan primer problema koje konkurentnost donosi

Proces P1

```
.
chin = getchar();
.
.
chout = chin;
putchar(chout);
.
```

Proces P2

```
.
.
chin = getchar();
chout = chin;
.
.
putchar(chout);
```

1. Proces P1 poziva proceduru **echo** i biva prekinut nakon prve instrukcije
2. Aktivira se proces P2 i poziva proceduru **echo** koja se izvršava do kraja
3. Nakon procesa P2 nastavlja se izvršavanje procesa P1

- **Rezultat:** prvi znak je izgubljen, drugi znak se prikazuje 2 puta
- **Srž problema** je deljena globalna promenljiva **chin**

Jednostavan primer – rešenje problema

Proces P1	Proces P2
.	.
chin = getchar();	.
.	chin = getchar();
.	chout = chin;
chout = chin;	.
putchar(chout);	.
.	putchar(chout);

Rešenje: Postaviti ograničenje da samo jedan proces u jednom tenutku može biti u proceduri **echo**

1. Proces P1 poziva **echo** i biva prekinuta nakon prve instrukcije (nakon unosa znaka)
2. Aktivira se proces P2 i poziva **echo**, ali kako je proces P1 u proceduri echo, proces P2 se blokira
3. Kada P1 nastavi sa radom, on nalazi svoj znak u **chin** i prikazuje ga na ekranu. Kada P1 napusti proceduru **echo**, uklanja se blokada sa P2 i on ulazi u proceduru **echo**

- **Rezultat:** oba znaka su uspešno prikazana

Uslov trke

- Uslov trke (*race condition*) nastaje kada:
  - Više procesa ili niti čitaju ili upisuju deljene podatke
  - Oni to obavljaju na način da finalni rezultat zavisi od redosleda izvršenja procesa.
- Konačna vrednost deljenih podataka zavisi od toga koji proces završi "trku" poslednji
- Primer:
  - Globalne promenljive b=1, c=2
  - P3: b=b+c
  - P4: c=b+c
  - Ako je P3 prvi, rezultat je b=3, c=5
  - Ako je P4 prvi, rezultat je b=4, c=3

Problem

```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

counter = counter +1

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

OS	Thread 1	Thread 2	PC	eax (after instruction)	counter (after instruction)
	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**

Poslovi operativnog sistema u vezi konkurentnosti

- OS mora biti sposoban da prati razne procese
- OS mora dodeljivati i oduzimati razne resurse za svaki aktivan proces
  - Procesorsko vreme
  - Memoriju
  - Datoteke
  - U/I uređaje
- OS mora zaštiti podatke i fizičke resurse jednog procesa od nenamernog narušavanja od drugog
- Izlaz mora biti nezavistan od brzine izvršenja drugih konkurentnih procesa

Uzajamno delovanje procesa

- Procesi nisu svesni postojanja drugih procesa
  - Nezavisni procesi za koje **nije predviđeno da rade zajedno**
  - OS vodi računa o njihovom **nadmetanju** za resurse
- Procesi su indirektno svesni postojanja drugih procesa
  - Procesi koji dele pristup nekom objektu (npr. U/I baferu)
  - Pri deljenu zajedničkog objekta pokazuju **kooperaciju preko deljenja**
- Procesi koji su direktno svesni postojanja drugih procesa
  - Procesi koji mogu komunicirati preko PID-a i rade zajednički na nekoj aktivnosti
  - Takvi procesi pokazuju **kooperaciju putem komunikacije**

Uzajamno delovanje procesa

Stepen svesnosti	Odnos	Uticaj koji jedan proces ima na drugi	Potencijalni problemi upravljanja
Procesi nisu svesni postojanja drugih procesa	Nadmetanje	• Rezultati jednog procesa nezavisni od akcije ostalih procesa • Može uticati na vreme izvršavanja procesa	• Uzajamno isključivanje • Uzajamno blokiranje (resurs koji se može obnoviti) • Gladovanje
Procesi su indirektno svesni postojanja drugih pricesa	Kooperacija deljenjem	• Rezultati jednog procesa mogu zavisiti od informacija dobijenih od drugi procesa • Može uticati na vreme izvršavanja procesa	• Uzajamno isključivanje • Uzajamno blokiranje (resurs koji se može obnoviti) • Gladovanje • Povezanost podataka
Procesi koji su direktno svesni postojanja drugih procesa	Kooperacija komunika-cijom	• Rezultati jednog procesa mogu zavisiti od informacija dobijenih od drugih procesa • Može uticati na vreme izvršavanja procesa	• Uzajamno blokiranje (resurs koji se može obnoviti) • Gladovanje

Nadmetanje procesa za resursima

- Konkurentni procesi se **nadmeću** za korišćenje istog resursa
- Pri nadmetanju procesa postoje tri problema upravljanja:
  1. **Uzajamno isključivanje** pri korišćenju nedeljivog resursa
     - Takav resurs nazivamo **kritični resurs**, a deo programa koji ga koristi **kritična sekcija**
     - Kritična sekcija
       - Samo jedan program u datom trenutku može biti u kritičnoj sekciji
       - **Primer:** samo jedan proces u jednom trenutku može slati komandu štampaču
  2. Uzajamno blokiranje
  3. Gladovanje

Ilustracija uzajamnog isključivanja

- *N* procesa pristupa deljenom resursu u okviru svoje kritične sekcije
  - Funkcije zaključavanja/otključavanja (*entercritical/exitcritical*)

```
PROCESS 1 */

void P1
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

```
/* PROCESS 2 */

void P2
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

. . .

```
/* PROCESS n */

void Pn
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

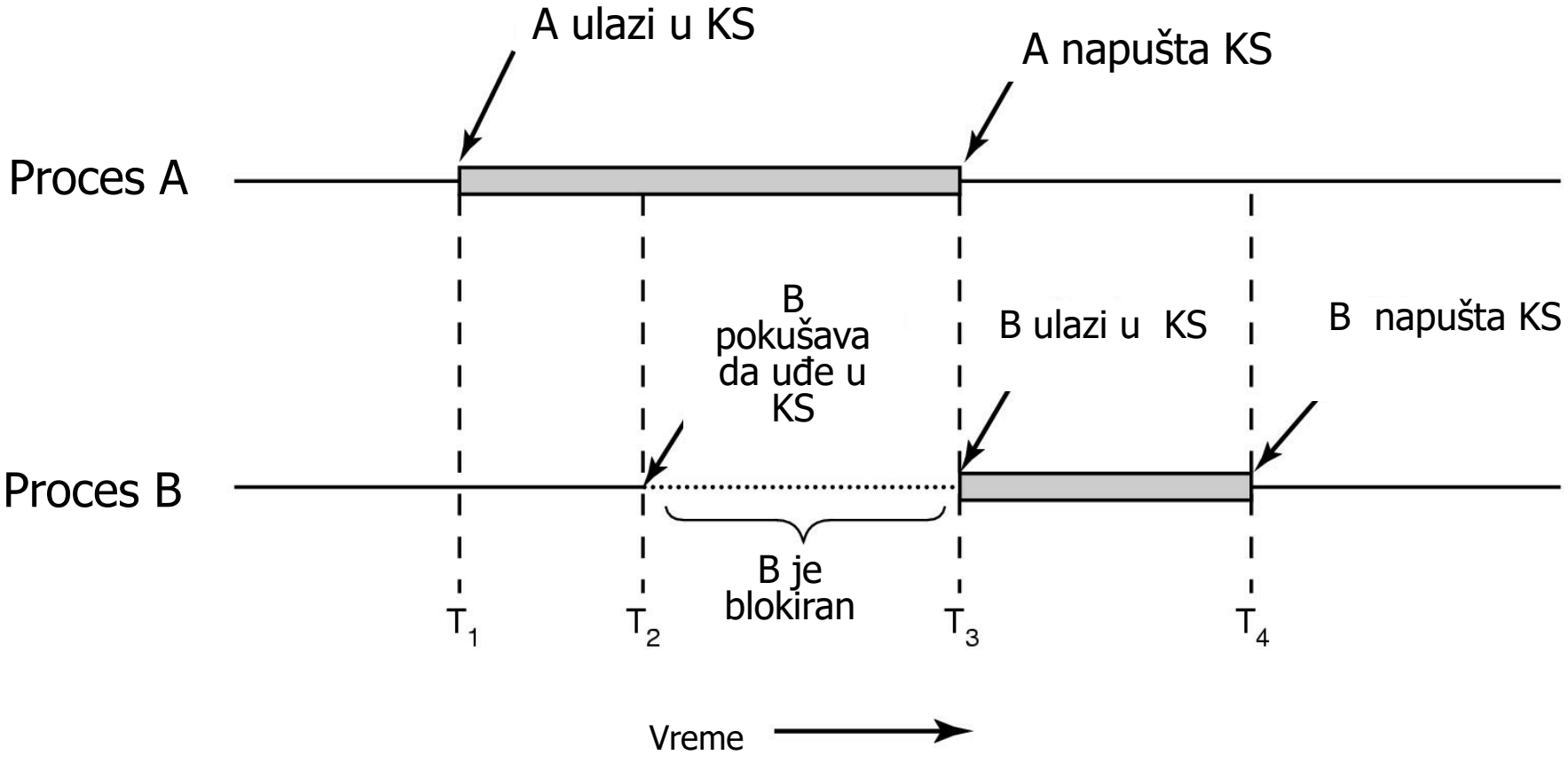
Kooperacija procesa deljenjem i komunikacijom

- Konkurentni procesi (ili niti) često imaju potrebe da dele podatke (koji se održavaju u deljenoj memoriji, ili u datotekama) i resurse, ili razmenjuju poruke
- Ako ne postoji kontrolisani pristup deljenim resursima/podacima, tj. ukoliko nema uzajamnog isključivanja neki procesi će dobiti nekonzistentan pogled na podatke
- Kod kooperacije komunikacijom, nema uzajamnog isključivanja već je moguće uzajamno blokiranje i gladovanje
- Akcije koje izvode konkurentni procesi zavisi će od redosleda kojim se izvršenja prepliću/preklapaju –uslov trke
- Zbog toga procese treba sinhronizovati

Zahtevi za uzajamno isključivanje

- Samo jednom procesu je dozvoljeno da uđe u kritičnu sekciju (KS) radi pristupa deljenom resursu
  - Dva i više procesa ne mogu biti istovremeno u kritičnoj sekciji
- Proces koji se zaustavi u sekciji koja nije kritična mora to uraditi bez uticaja na druge procese
- Proces koji zahteva ulazak u KS ne može biti beskonačno zadržan – nema zastoja i gladovanja
- Kada nijedan proces nije u KS, bilo kom procesu koji zahteva ulaz u KS to mora biti dozvoljeno
- Ne prave se pretpostavke o relativnoj brzini procesa niti o broju procesora
- Proces ostaje u kritičnoj sekciji samo konačno vreme

Uzajamno isključivanje u izvršavanju kritične sekcije



Rešenje uzajamnog isključivanja

- SW rešenja (za samostalni rad, Dodatak A)
  - Softverski algoritmi imaju značajno dodatno procesiranje i rizik od nastanka logičkih grešaka
- HW rešenja
  - Vezana su za neke mašinske instrukcije i prevenciju prekida
- OS rešenja
  - OS obezbeđuje sistemske funkcije i strukture podataka koje programeri mogu koristiti za programiranje kritičnih sekcija

Uzajamno isključivanje: hardverska podrška

- Postoje dva pristupa:
  - Zabrana (onemogućavanje, *disable*) prekida
  - Specijalne mašinske instrukcije

Zabrana prekida

- Na jednoprocesorskom sistemu:
  - Obezbeđeno uzajamno isključivanje, ali efikasnost izvršenja degradirana.
  - Dok je proces u KS, nijedan drugi proces se ne može izvršavati
- Na višeprocesorskom sistemu ovaj pristup ne radi:
  - Uzajamno isključivanje nije obezbeđeno
- Generalno, rešenje nije prihvatljivo

```
Proces Pi:
while(true)
{
    zabrana prekida
    KRITIČNA SEKCIJA
    dozvola prekida
    OSTATAK
}
```

Specijalne mašinske instrukcije

- Na hardverskom nivou, pristup nekoj memorijskoj lokaciji isključuje ostale pristupe toj lokaciji
- Projektanti HW su predložili mašinske instrukcije koje nad istom memorijskom lokacijom izvode dve akcije atomično (nedeljivo, u jednom koraku koji ne može biti prekinut)
  - Na primer, čitanje i upis, ili čitanje i testiranje
- Izvršenje takve instrukcije je uzajamno isključivo (čak i kod multiprocesora)
  - Tokom izvršenja instrukcije, pristup memorijskoj lokaciji je blokiran za sve druge instrukcije koje referenciraju tu lokaciju
- Implementirane instrukcije:
  - *Test and Set* instrukcija
  - *Compare&Swap* instrukcija
  - *Exchange* instrukcija

Test and Set instrukcija

- Instrukcija za testiranje i postavljanje (Test and Set Instruction)

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

Exchange instrukcija

- Instrukcija razmene (*Exchange Instruction*)

```
void exchange(int register,
    int memory) {
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

- Instrukcija XCHG na procesorima Intel-32 (Pentium) i IA-64 (Itanium)

Uzajamno isključivanje korišćenjem mašinskih instrukcija

```
/* 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));

}
```

(a) Test and set instruction

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

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Compare & Swap instrukcija

```
int compare_and_swap (int
   *word,
   int testval,
   int newval)
{
    int oldval;
    oldval = *word;
    if (oldval == testval)
      *word = newval;
    return oldval;
}
```

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

}
```

(a) Compare and swap instruction

Svojstva pristupa sa mašinskim instrukcijama - Prednosti

- Može se primenjivati na bilo koji broj procesa i na jednom da procesoru i na multiprocesoru gde procesori dele glavnu memoriju
- Jednostavno je i lako se proverava
- Može se koristiti za podršku više kritičnih sekcija; svaka KS može se definisati svojom promenljivom

Svojstva pristupa sa mašinskim instrukcijama - Nedostaci

- Zaposleno čekanje
- Moguće je gladovanje
  - Kada jedan proces napusti KS, a više čeka na ulaz u KS, izbor sledećeg procesa je proizvoljan
- Moguće je uzajamno blokiranje
  - Ako je proces nižeg prioriteta P1 u KS, prekinut je i procesor je dodeljen procesu višeg prioriteta P2 koji želi da uđe u KS
    - P1 se prekida, P2 dobija procesor
    - ako P2 želi da koristi isti resurs kao P1 neće mu biti dozvoljeno
    - P2 čeka i drži procesor, tako da P1 neće nikad biti raspoređen na procesor jer je nižeg prioriteta

Semafori

- Definisani od strane E. Dijkstra, 1965
- Sinhronizacioni mehanizam koji obezbeđuje OS
- Semafor S je **integer promenljiva** nad kojom su definisane tri **atomične** operacije
  - **Inicijalizacija**
  - **semWait(S)**
    - alternativna imena **P** (Dijkstra, na holandskom *proberen*), **down** (Tanenbaum), **wait**
  - **semSignal(S)**
    - alternativna imena **V** (Dijkstra, na holandskom *verhogen*), **up** (Tanenbaum), **signal**
- Semafor može biti:
  - **Binarni** – uzima samo dve vrednosti 0 i 1
  - **Brojački** (**generalni**) – uzima vrednosti 0,1,2,...,n,

Operacije semafora – definicija

- Proces koji izvede operaciju ***semWait(S)*** **blokira** sam sebe, umesto da čeka u petlji
- Proces koji je blokiran na semaforu **biće probuđen** kada neki drugi proces izvede operaciju ***semSignal(S)***
- Ove dve operacije moraju biti:
  - **Atomične** – moraju se izvoditi u celini, tj. bez prekidanja
  - **Uzajamno isključive** - na istom semaforu ne mogu se istovremeno izvoditi

semWait(S):

if (S>0) **then**

    S=S-1

else

    {blokiraj_proces, aktiviraj_dispečer}

semSignal(S):

if (jedan ili više procesa čeka na S) **then**

    probudi jedan od procesa koji su blokirani na semaforu S

else

    S=S+1

Operacije binarnog semafora

```
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 */;
      }
}
```

Semafor – implementacija

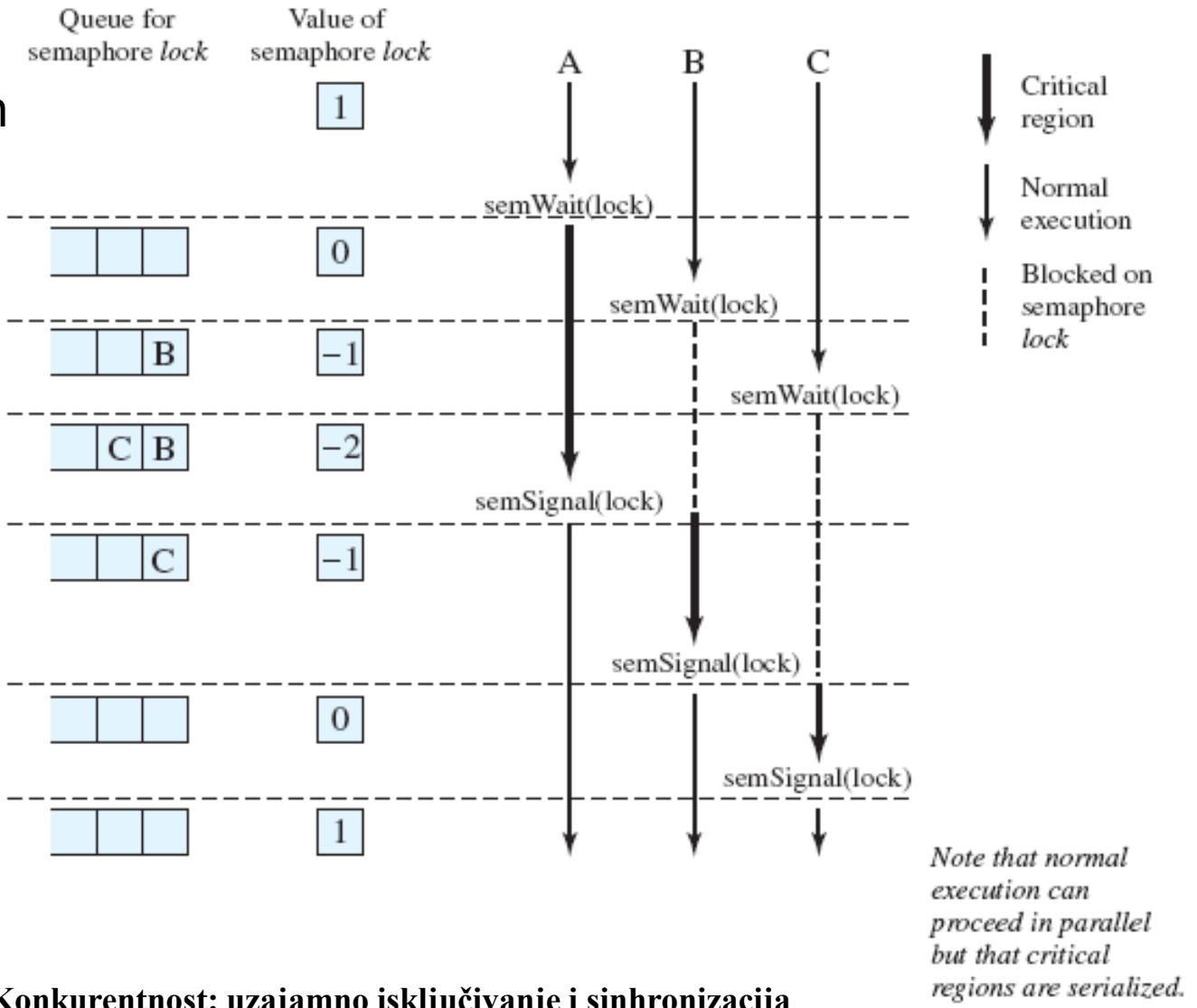
```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <=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

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));
}
```

Primer: Uzajamno isključivanje

- Tri procesa pristupaju deljenim podacima zaštićenim semaforom
- s .count ≥ 0 – broj procesa koji može da izvrši operaciju semWait(s) bez blokiranja (ukoliko u međuvremenu nije izvršena semSignal(s))
- s .count < 0 – broj procesa blokiranih u redu s.queue



Korišćenje semafora za sinhronizaciju dva procesa

- Semafori se mogu koristiti **za sinhronizaciju dva procesa**
- Posmatramo dva procesa P1 i P2
- Naredba S1 procesa P1 treba da se izvede pre naredbe S2 procesa P2
- Koristi se binarni semafor **synch**
- Semafor **synch** se inicijalizira na 0

```
void P1()
{
   S1;
   semSignal(synch);
   …
}

void P2()
{
   ...
   semWait(synch);
   S2;
}

void main()
{
   semaphore  synch =0;
   parbegin( P1,P2);
}
```

Korišćenje semafora za sinhronizaciju procesa (2)

- Semafori se mogu koristiti za sinhronizaciju procesa tipa ***randezvous***
- Kod tzv. **sastanka (*randezvous*)** proces koji prvi dođe do KS mora čekati da drugi proces stigne do KS
- Koriste se dva binarna semafora **S1 i S2 koji se inicijalizuju na 0**

```
void P1()
{
   …
   semSignal(s1);
   semWait(s2); …
   …
}

void P2()
{
   …
   semSignal(s2);
   semWait(s1);
   …
}

void main()
{
   semaphore s1=0, s2=0;
   parbegin( P1,P2);
}
```

Problem proizvođač/potrošač (Producer/Consumer)

- Jedan ili više proizvođača generiše podatke (slogove, znakove) i stavlja ih u bafer
- Jedan potrošač uzima podatke iz bafera jednu po jednu
- Sistem mora obezbditi da se operacije nad baferom ne preklapaju
- U datom trenutku samo jedan proizvođač ili potrošač može pristupati baferu
- Pretpostavimo da je bafer neograničen i sadrži linearan niz elemenata

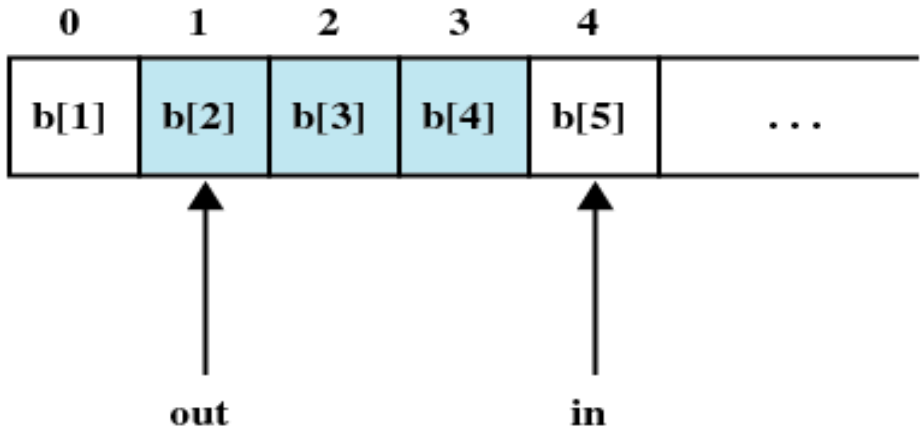
Proizvođač i potrošač

producer:

```
while (true) {
/*napravi stavku v*/;
   b[in] = v;
   in++;
}
```

consumer:

```
while (true) {
   while (in <= out)
   /*ne radi ništa*/;
   w = b[out];
   out++;
   /* potroši stavku w */
}
```



Tamna polja predstavljaju elemente bafera (niza) sa podacima

Rešenje korišćenjem binarnih semafora

- Neispravno rešenje problema proizvođač/potrošač sa beskonačnim baferom uz upotrebu binarnih semafora
- Broj elemenata u baferu
  `n = in - out`
- Semafor *s* obezbeđuje uzajamno isključivanje
- Semafor *delay* primorava potrošača da se blokira (izvršavanjem semWait) ukoliko je bafer prazan

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore 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);
}
```

Moguć scenario ovog rešenja

	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

Korigovano rešenje

- Korektno rešenje problema proizvođač/potrošač sa beskonačnim baferom uz upotrebu binarnih semafora
- Dodavanje pomoćne promenljive m u okviru kritične sekcije

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore 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);
}
```

Rešenje korišćenjem brojačkog semafora

- Korektno rešenje problema proizvođač/potrošač sa beskonačnim baferom uz upotrebu brojačkog semafora
- Promenljiva n je sada semafor i predstavlja broj elemenata u baferu
- Problem: ako se semWait operacije nad n i s zamene

```
/* program  producerconsumer */
semaphore n = 0;
semaphore 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);
}
```

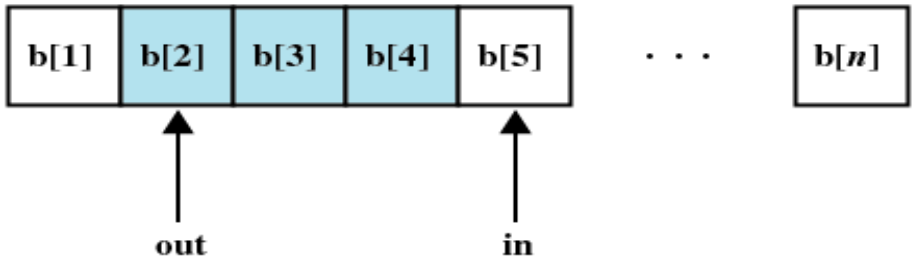
Problem proizvođač/potrošač sa kružnim baferom

producer:

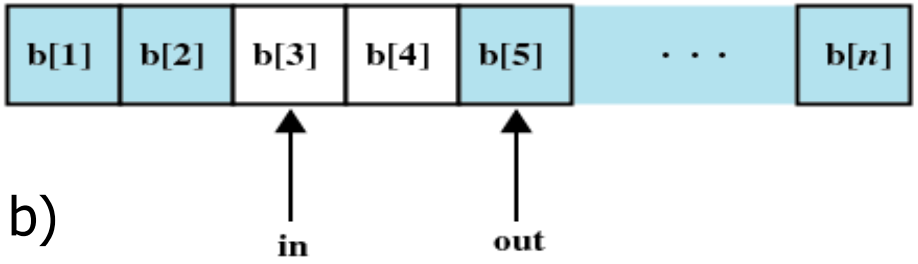
```
while (true) {
   /* napravi stavku v */
   while((in+1)%n==out)
      /* ne radi ništa */;
   b[in] = v;
   in = (in + 1) % n
}
```

consumer:

```
while (true) {
   while (in == out)
      /* ne radi ništa */;
   w = b[out];
   out = (out + 1) % n;
   /* potroši stavku w */
}
```



a)



b)

Rešenje sa ograničenim kružnim baferom

- Korektno rešenje problema proizvođač/potrošač sa **ograničenim** kužnim baferom uz upotrebu binarnog i brojačkog semafora
  - Proizvođač se blokira kad hoće da upiše element u pun bafer
  - Potrošač se blokira kad hoće da uzme element iz praznog bafera

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1;
semaphore n= 0;
semaphore 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);
}
```

Problemi sa semaforima

- Semafori nude moćan alat za obezbeđenje uzajamnog isključivanja i za koordinaciju procesa
- Ali *semWait(S)* i *semSignal(S)* su razbacani među procesima, pa je teško razumeti njihove efekte
- Korišćenje mora biti korektno u svim procesima
- Jedan loš proces može uticati na zastoj čitave kolekcije procesa

Monitori

- Monitor je konstrukcija viših programskih jezika koja nudi funkcionalnost ekvivalentnu semaforima, ali je lakša za upravljanje
- Prvi put ga je formalno definisao Hoare, 1974
- Nalazi se u mnogim programskim jezicima
  - Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java
  - Takođe se može implementirati kao programska biblioteka

Monitori (2)

- Monitor je SW modul koji sadrži:
  - Jednu ili više procedura/funkcija
  - Inicijalizacionu sekvencu
  - Lokalne promenljive
- Karakteristike monitora:
  1. Lokalnim promenljivama može se pristupati samo iz procedura monitora i nijednom spoljnom procedurom
  2. Proces "ulazi" u monitor pozivom neke od njegovih procedura
  3. U jednom trenutku samo jedan proces se može izvršavati u monitoru; svaki drugi proces koji poziva proceduru monitora se blokira i čeka da monitor postane dostupan
- Prve dve karakteristike podsećaju na karakteristike objekata u OO jezicima
  - Monitor se može implementirati kao objekat
- Treća karakteristika obezbeđuje uzajamno isključivanje

Monitori (3)

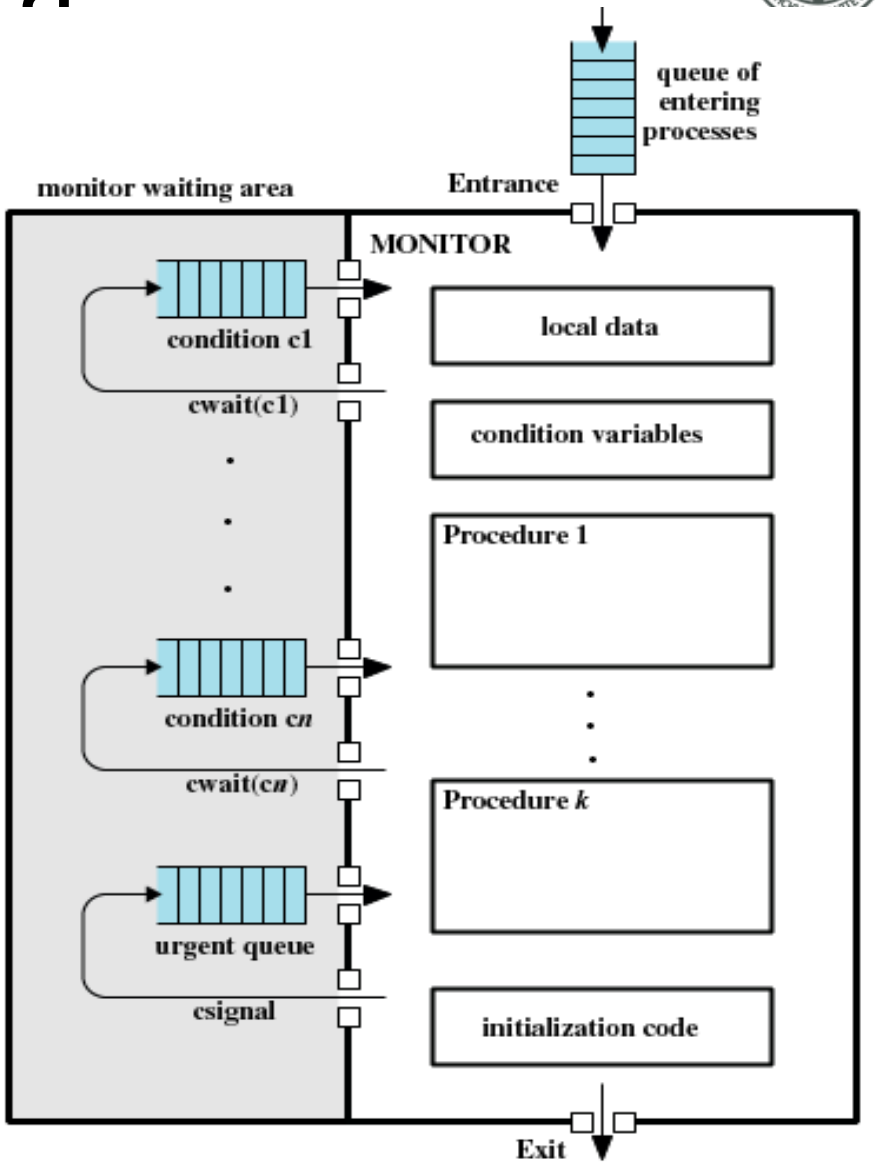
- Monitor osigurava **uzajamno isključivanje**
  - Nije potrebno programirati ovo ograničenje
- U jednom trenutku deljivim podacima u monitoru može pristupati samo jedan proces
  - Stoga su deljivi podaci zaštićeni njihovim smeštanjem u monitor
- Monitor takođe poseduje sinhronizacioni alat
  - To su **uslovne promenljive (*condition*)**
  - Sinhronizaciju procesa vrši programer korišćenjem uslovnih promenljivih

Uslovne promenljive monitora

- **Uslovne promenljive** su lokalne u monitoru
  - Može im se pristupati jedino unutar monitora
- Uslovne promenljive su specijalnog tipa podataka u monitoru kojima se može pristupati jedino pomoću dve funkcije:
  - **cwait(a):** blokira izvršenje procesa koji je pozvao ovu funkciju na uslovnoj promenljivoj **a**; monitor je sada na raspolaganju za bilo koji drugi proces
  - **csignal(a):** budi proces koji je blokiran na uslovnoj promenljivoj **a**
    - Ako postoji više takvih procesa – bira jedan
    - Ako nema takvih procesa – ne radi ništa
  - Treba uočiti da su operacije **cwait** i **csignal** monitora različite od istih operacija kod semafora
    - Ako proces u monitoru izda signal i ako nijedan proces ne čeka na toj uslovnoj promenljivoj signal je izgubljen

Struktura monitora

- Monitor ima jednu ulaznu i jednu izlaznu tačku
- Proces koji je u monitoru može privremeno sam sebe blokirati na uslovnoj promenljivoj **a** korišćenjem operacije **cwait(a)**
- On se tada stavlja u red procesa koji pokušavaju da ponovo uđu u monitor kada se uslov promeni i tada nastavljaju sa izvršenjem sa naredbom koja sledi iza **cwait(a)**
- Ako proces koji se izvršava u monitoru detektuje ispunjenje uslova na kojem su blokirani provesi, poziva **csignal(a)** koja budi (deblokira) proces iz reda procesa te uslovne promenljive **a**



Primena monitora za rešavanje problema proizvožač-potrošač

- Monitor **boundedbuffer** upravlja baferom koji se koristi za smeštanje i preuzimaje znakova
- Monitor sadži dve uslovne promenljive ***notfull*** (kada postoji mesto u baferu da se doda barem jedan znak) i ***notempty*** (kada u baferu postoji barem jedan znak)
- Proizvođač može dodati znak u bafer samo preko procedure monitora **append**
  - Proizvođač nema direktan pristup baferu
- Potrošač može uzeti znak iz bafera samo preko procedure monitora **take**
  - Potrošač nema direktan pristup baferu

Monitori: Proizvođač/potrošač

```
/* 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(notempty);                       /* 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 */
}
```

Lokalne promenljive monitora

procedura append

procedura take

inicijalizacija

Monitori: 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);
}
```

Prenos poruka (*Message Passing*)

- To je metoda komunikacije među procesima
- Koriste se dve primitive:

  send(odredište,poruka)

  receive(izvor,poruka)

- **send** – proces šalje **poruku** drugom procesu koji je označen kao **odredište**
- **receive** – proces prima **poruku** od procesa koji je označen kao **izvor**

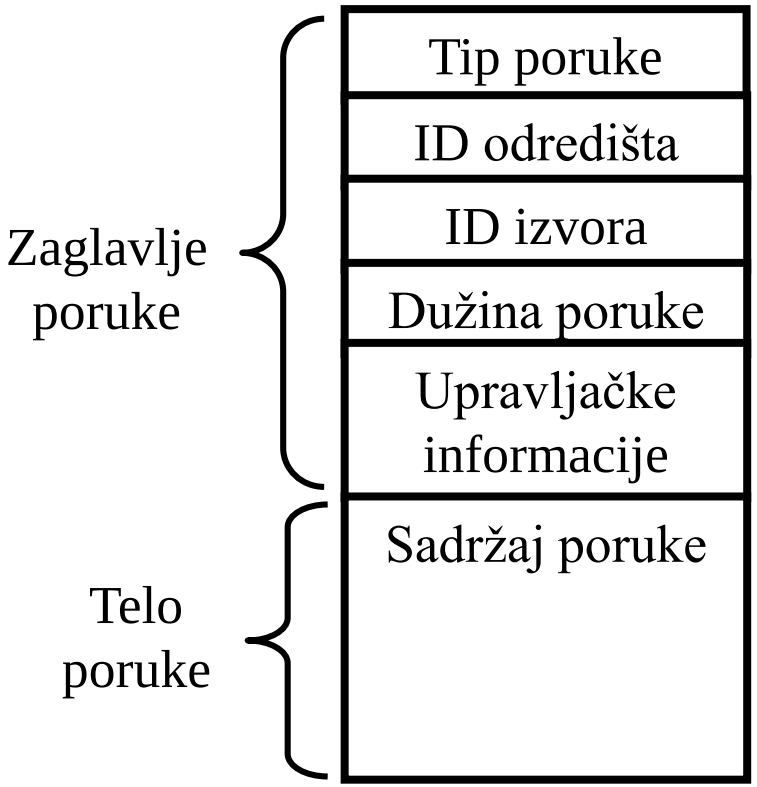
Prenos poruka (*Message Passing*)

- Problemi u prenosu poruka:
  - Format poruke
  - Sinhronizacija – procese koji komuniciraju treba na neki način sinhronizovati
  - Protokol za komunikaciju
    - Primalac i pošiljalac moraju dogovoriti neki protokol za komunikaciju
    - Npr. primalac po prijemu poruke šalje potvrdu, a pošiljalac ako potvrda ne stigne za neko vreme ponovo šalje poruku
    - Poruke se numerišu čime se izbegava zabuna ako poruka kasni
  - Imenovanje (adresiranje) procesa
    - Procesi koji komuniciraju moraju imati način da se međusobno referenciraju
  - Autentikacija – da pošiljalac i primalac znaju da komuniciraju sa pravim procesom

Format poruke

- Format poruke zavisi:
  - Od cilja sistema za prenos poruka
  - Od toga da li se sistem izvršava na jednom računaru ili distribuiranom sistemu
- Poruke mogu biti fiksne i promenljive dužine

- Opšti format poruke promenljive dužine



Sinhronizacija u prenosu poruka (1)

- Komunikacija dva procesa porukama podrazumeva neki nivo sinhronizacije između njih
  - Primalac ne može primiti poruku pre nego što je neki proces ne pošalje
- Važno je da se definiše šta se događa sa procesom nakon što pokrene **send** ili **receive** primitivu
- Proces primalac i proces pošiljalac mogu biti:
  - blokirani
  - neblokirani

Sinhronizacija u prenosu poruka (2)

- Moguće su tri kombinacije:
  - Blokirajući **send**, blokirajući **receive**
    - pošiljalac i primalac se blokiraju dok poruka ne bude isporučena
    - Naziva se ***randevouz***
  - Neblokirajući **send**, blokirajući **receive**
    - Pošiljalac nastavlja da se izvršava, a primalac se blokira dok ne stigne poruka
    - Omogućava se pošiljaocu da pošalje 1 ili više poruka na jednu ili više destinacija
    - Primer su serverski procesi
  - Neblokirajući **send**, neblokirajući **receive**
    - Ni pošiljalac ni primalac ne moraju da budu blokirani

Sinhronizacija u prenosu poruka (3)

- Za pošiljaoca: prirodno je da ne bude blokiran posle slanja poruke pozivom funkcije **send**
  - Može poslati nekoliko poruka na više destinacija
  - Ali pošiljalac obično očekuje potvrdu prijema poruke (za slučaj da je primalac neispravan)
- Za primaoca: prirodnije je da bude blokiran dok ne primi poruku pozivom funkcije **receive**
  - Primalac obično treba neku informaciju pre nego što nastavi obradu
  - Postoji opasnost da bude neograničeno blokiran ako proces pošiljalac završi pre nego što pošalje **send**

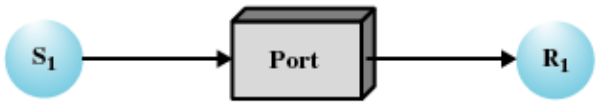
Adresiranje procesa

- **Direktno adresiranje**
  - Jedan proces direktno šalje poruku drugom procesu
  - Pošiljalac eksplicitno specificira kome šalje poruku preko argumenta **odredište**
  - Primalac eksplicitno specificira od koga prima poruku preko argumenta **izvor**
- **Indirektno adresiranje**
  - Poruke se ne šalju direktno od pošiljaoca primaocu, već se šalju deljenim strukturama podataka koje se sastoje od redova koji mogu privremeno da čuvaju poruke
  - Ovi redovi se nazivaju **poštanski sandučići (*mailboxes*)**
  - Proces pošiljalac šalje poruku u određeni mailbox, a drugi proces primalac uzima poruku iz tog mailbox-a

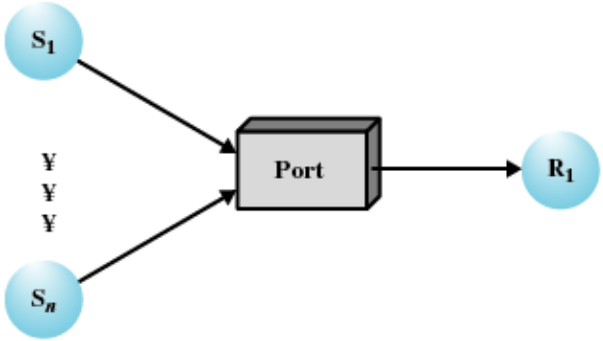
Adresiranje poruka

- Kod indirektnog adresiranja odnos između pošiljaoca i primaoca može biti:
  - 1:1 (jedan-prema-jedan)
    - privatni komunikacioni link između 2 procesa
  - N:1 (više-prema-jedan)
    - koristan za klijent-server interakciju kada jedan proces prima poruke od većeg broja drugih procesa
    - U ovom slučaju se mailbox naziva **port**
  - 1:N (jedan-prema-više)
    - Postoji 1 pošiljalac i više primaoca
    - Pogodan je u slučaju kada jedan proces emituje poruke većem broju procesa
  - N:M (više-prema-više)
    - Dopušta da više serverskih procesa konkurentno nudi servis većem broju klijenata

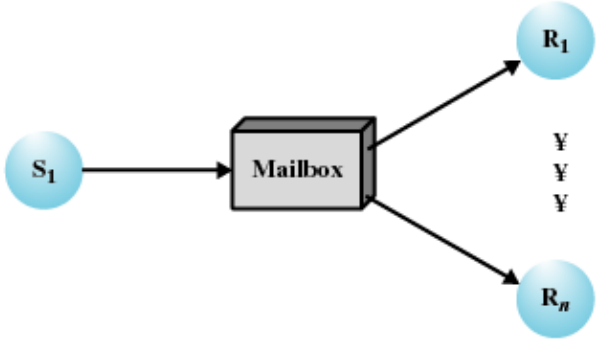
Indirektna komunikacija među procesima



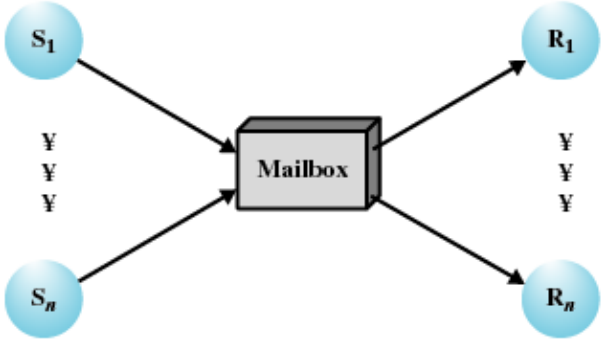
jedan:jedan



više:jedan



jedan:više



više:više

Poštansko sanduče i port

- Poštansko sanduče (mailbox) može biti
  - Privatno za par pošiljalac-primalac
  - Isto poštansko sanduče može deliti više pošiljaoca i primaoca
    - OS mora obezbediti tip podataka *message*
- Port je poštansko sanduče povezano sa 1 primaocem i više pošiljaoca
  - Koristi se za klijent-server aplikacije
- Vlasnici portova i mailbox-ova
  - Port obično kreira proces primalac i on je njegov vlasnik
  - Port se uništava kad se proces primalac terminira
  - Mailbox kreira OS u ime nekog procesa koji postaje njegov vlasnik
  - Takav mailbox se uništava na zahtev vlasnika ili kada se vlasnik terminira

Uzajamno isključivanje prenosom poruka

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true)
    {
     receive (mutex, msg);
     /* critical section    */;
     send (mutex, msg);
     /* remainder    */;
    }
}
void main()
{
    create_mailbox (mutex);
    send (mutex, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.20 Mutual Exclusion Using Messages

Uzajamno isključivanje prenosom poruka (2)

- Kreira se mailbox **mutex** deljiv za N konkurentnih procesa
- **send()** je neblokirajući
- **receive()** – proces se blokira kada je poštansko sanduče mutex prazno
- Inicijalizacija: šalje se prazna poruka **send(mutex,null);** null je poruka bez sadržaja
- Proces Pi koji želi da uđe u KS pokušava da primi poruku
  - Ukoliko je mailbocx mutex prazan, proces se blokira
  - Inače proces Pi ulazi u KS
  - Po izlasku iz KS vraća poruku u mailbox
- Prvi proces Pi koji izvrši **receive()** će ući u KS. Ostali će biti blokirani dok Pi ponovo ne pošalje poruku

Proizvođač-potrošač sa prenosom poruka

- Koriste se dva mailbox-a: *mayproduce* i *mayconsume*
  - Proizvođač generiše podatke i šalje ih u mailbox *mayconsume*
  - Potrošač uzima poruke iz mailbox-a *mayconsume* dok ima bar jedna poruka
- Mailbox *mayconsume* je **bafer**
  - Bafer je kapaciteta K poruka (globalna promenljiva capacity)
- Mailbox *mayproduce* se inicijalno puni sa K praznih (null) poruka
  - Broj poruka u mailbox-u *mayproduce* se smanjuje sa svakom proizvodnjom, a povećava sa svakom potrošnjom
- Može podržavati više proizvođača i potrošača koji imaju pristup do oba mailbox-a
- Sistem može biti distribuiran:
  - proizvođač i mailbox *mayproduce* na jednom mestu, a potrošač i mailbox *mayconsume* na drugom

Proizvođač-potrošač sa prenosom poruka

```
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);
}
```

Klasični sinhronizacioni problemi

- Proizvođač – potrošač
- Čitaoci – pisci
- Večera filozofa
- Uspavani berberin

Problem čitaoci-pisci

The Readers and Writers Problem

- Objekti podataka (npr. fajl, blok memorije, grupa registara procesora) su deljivi za više konkurentnih procesa
- Neki procesi samo čitaju sadržaj deljivih objekata (to su **čitaoci**), dok drugi samo upisuju u deljivi objekat (to su **pisci**)
  - Ako dva čitaoca istovremeno čitaju deljivi objekat nema neželjenih efekata
  - Međutim, ako pisac i neki drugi proces (čitalac ili pisac) pristupaju simultano deljivom objektu nastupiće problem
- Moraju biti zadovoljeni sledeći uslovi:
  - Bilo koji broj čitalaca može istovremeo čitati deljivi objekat
  - U jednom trenutku samo 1 pisac može upisivati u deljivi objekat
  - Ukoliko pisac upisuje, nijedan čitalac ne može da čita
- Ovaj sinhronizacioni problem se naziva **problem čitaoci-pisci**

Problem čitaoci-pisci (2)

- Da bi se izbegli problemi **treba obezbediti da pisci imaju ekskluzivan pristup deljivom objektu**
- Ima više rešenja
  - **Čitaoci imaju prioritet** – nijedan novi čitalac se ne drži na čekanju, ako je neki čitalac već dobio dozvolu da koristi deljivi objekat; tj. nijedan čitalac neće čekati da neki čitalac završi čitanje bez obzira da li neki pisac već čeka
  - **Pisci imaju prioritet** – jedan pisac je spreman da obavi upis čim bude to moguće; tj. ako neki pisac čeka za pristup objektu, nijedan novi čitalac ne može startovati čitanje
- Oba rešenja mogu dovesti do ***izgladnjivanja*** procesa:
  - Čitaoci imaju prioritet – gladuju *pisci*
  - Pisci imaju prioritet – gladuju *čitaoci*

Čitaoci imaju prioritet

```
/* 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);
     }
```

readcount – broj čitalaca

wsem – semafor za uzajamno isključivanje na deljivom objektu; dok pisac piše nijedan čitalac ili pisac ne mogu da koriste deljivi objekat

Ako neki čitalac čita ostali ne moraju da čekaju

Ako nijedan čitalac ne čita, prvi čitalac mora čekati na **wsem**

x – koristi se da bi se obezbedilo ispravno ažuriranje promenljive **readcount**

```
/*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);
}
```

Pisci imaju prioritet

rsem – sprečava sve čitaoce dok postoji bar jedan pisac koji želi da pristupi deljivom podatku

wsem – za uzajamno isključivanje na deljivom objektu; dok pisac piše nijedan čitalac ili pisac ne mogu da koriste deljivi objekat

x – semafor koji se koristi da bi se obezbedilo ispravno ažuriranje promenljive **readcount**

y – semafor koji se koristi da bi se obezbedilo ispravno ažuriranje promenljive **writecount**

z – semafor koji obezbeđuje da samo jedan čitalac može biti u redu na semaforu **rsem**, dok ostali ulaze u red na semaforu **z**

Čitaoci-pisci: rešenje prenosom poruka

```
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++;
         }
      }
}
```

Domaći zadatak

- Poglavlje **5 Konkrurentnost: uzajamno isključivanje i sinhronizacija**
  - 5.9 Ključni pojmovi, kontrolna pitanja i problemi
- Animacije
  - Mutual exclusion with a semaphore http://williamstallings.com/OS/Animation/Queensland/SEMA.SWF
  - Process Synchronization: Producer/Consumer problem https://apps.uttyler.edu/Rainwater/COSC3355/Animations/processsync.htm