

# Sinhronizacija procesa

- **Pozadina**
- **Problem kritične sekcije**
- **Hardverska sinhronizacija**
- **Semafori**
- **Klasični problemi sinhronizacije**
- **Kritični regioni**
- **Monitori**

# Pozadina

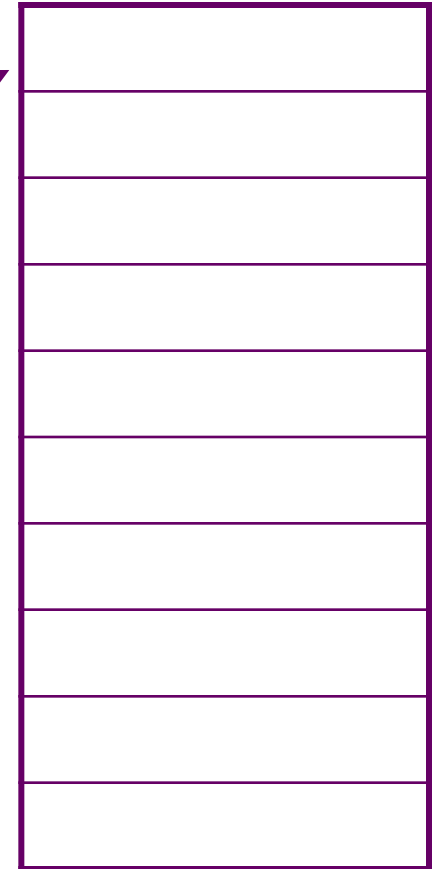
- **Konkurentni pristup deljenim podacima**
- može **dovesti** do **nekonzistencije podataka**
- **Održavanje konzistencije podataka**
  - ☞ **zahteva mehanizme** koji će da **osiguraju**
  - ☞ **izvršavanje kooperativnih procesa u poretku (in order)**
- **Solucija deljene memorije za problem bounded-buffer**
  - ☞ dozvoljava **više  $n - 1$**  elemenata u baferu
  - ☞ **u isto vreme**
- **Rešenje, gde se koriste svih  $N$  bafera, nije jednostavno**
  - ☞ **Pretpostavimo da želimo da izmenimo kôd za proizvođač-potrošač:**
  - ☞ **dodajući promenljivu **counter****
  - ☞ **inicijalizovan na 0**
  - ☞ **inkrementiran svaki put kada se novi element doda u bafer**
  - ☞ **dekrementiran svaki put kada je elemenat izbačen iz bafera**

# Bounded-Buffer

## ■ Deljeni podaci (shared data)

- ☞ **in** pokazuje na sledeći slobodan bafer
- ☞ **out** pokazuje na prvi pun bafer
- ☞ **counter++** proizvođač; **counter--** potrošač;

```
#define BUFFER_SIZE 10  
  
typedef struct {  
    ...  
} item;  
  
item buffer[BUFFER_SIZE];  
    int in = 0;  
    int out = 0;  
    int counter = 0;
```



# Bounded-Buffer

- Proces proizvođač

**item** nextProduced;

**while (1) {**

**while (counter == BUFFER\_SIZE)**

        ; /\* ništa ne radi, bafer je pun \*/

**buffer[in] = nextProduced;**

**in = (in + 1) % BUFFER\_SIZE;**

**counter++;**

**}**

# Bounded-Buffer

## ■ Proces potrošač

```
item nextConsumed;
```

```
while (1) {
```

```
    while (counter == 0)
```

```
        ; /* ništa ne radi, bafer je prazan */
```

```
    nextConsumed = buffer[out];
```

```
    out = (out + 1) % BUFFER_SIZE;
```

```
    counter--;
```

```
}
```

# Bounded Buffer

- Naredbe

**counter++;**

**counter--;**

moraju se izvršavati **atomski**

- **Atomska operacija znači**
  - ☞ operacija koja je u celosti izvršena
  - ☞ bez prekida

# Bounded Buffer

- Naredba “**counter++**” može biti implementirana u mašinskom jeziku kao:

**register1 = counter**

**register1 = register1 + 1**

**counter = register1**

- Naredba “**counter--**” može biti implementirana kao:

**register2 = counter**

**register2 = register2 - 1**

**counter = register2**

# Bounded Buffer

- Ako obojica, proizvođač i potrošač
  - ☞ pokušaju da ažuriraju bafer (brojač) **konkurentno**,
  - ☞ asemblerske naredbe mogu imati preklapanje (interleaving)
  
- Interleaving **zavisi od toga**
  - ☞ **kako su proizvođač i potrošač procesi raspoređeni**

# Bounded Buffer

- Imamo **counter** inicijalizovan na 5

- **counter++;**  
**counter--;**

- **Jedan od redosleda** naredbi je:

proizvođač: **register1 = counter** (*register1 = 5*)

proizvođač: **register1 = register1 + 1** (*register1 = 6*)

- potrošač: **register2 = counter** (*register2 = 5*)

potrošač: **register2 = register2 - 1** (*register2 = 4*)

- proizvođač: **counter = register1** (*counter = 6*)

potrošač: **counter = register2** (*counter = 4*)

- Vrednost **counter** može biti **4 ili 6**,

- **gde tačan rezultat moze biti 5**

# Stanje trke (Race Condition)

## ■ Stanje trke:

- ☞ Situacija gde više procesa
- ☞ Pristupaju i upravljaju deljenim podacima konkurentno

## ■ Konačna vrednost deljenih podataka

- ☞ zavisi od toga
- ☞ koji proces završava **poslednji**

## ■ Da bi se izbeglo stanje trke,

- ☞ **konkurentni procesi**
- ☞ **moraju biti sinhronizovani**

# Problem kritične sekcije (Critical section)

- n procesa
- svi se takmiče da koriste neke od deljenih podataka
- **Svaki proces ima deo koda,**
  - ☞ zvani **kritična sekcija,**
  - ☞ **u kome pristupa deljenim podacima**
- **Problem:**
  - ☞ **kad je jedan proces udje u svoju kritičnu sekciju,**
  - ☞ **ni jedan drugi proces ne sme da udje u svoju kritičnu sekciju**

**only one**

# Rešenje problema kritične sekcije

## 1. Međusobno isključenje (Mutual Exclusion).

Ako se proces  $P_i$  izvršava u svojoj kritičnoj sekciji, onda ni jedan drugi proces ne može da se izvršava u istoj kritičnoj sekciji

## 2. Progres (Progress)

Ako se nijedan proces ne izvršava u svojoj kritičnoj sekciji a postoje neki procesi koji žele da uđu u svoju kritičnu sekciju, onda selekcija procesa koji hoće da udje u kritičnu sekciju nemože biti odložena u nedogled

## 3. Ograničeno čekanje (Bounded Waiting)

Ograničenje (bound) mora postojati od trenutka kada proces da zahtev da uđe u svoju kritičnu sekciju sve dok zahtev ne bude prihvaćen

- Pretpostavka je da se svaki proces izvršava na nonzero brzini
- Nema pretpostavke vezane za relativnu brzinu  $n$  procesa.

# Inicijalni pokušaj za rešenje problema

- Samo 2 procesa,  $P_0$  i  $P_1$
- Generalna struktura procesa  $P_i$  (drugi proces  $P_j$ )

```
do {  
    entry section  
    critical section  
    exit section  
    reminder section  
} while (1);
```

- Procesi mogu deliti neke zajedničke promenljive
- da sinhronizuju njihove akcije

# Algoritam 1 (strict alternation)

- Deljene promenljive:

- `int turn; initially turn = 0`

- ☞ `if turn = i ⇒ Pi može ući u svojoj kritičnoj sekciji`

- **Proces P0**

- `do {`

- `while (turn != 0) ; /*(entry section)*/`

- `critical section`

- `turn = 1; /*exit section*/`

- `.....reminder section`

- `} while (1);`

- **Proces P1**

- `do {`

- `while (turn != 1) ;`

- `critical section`

- `turn = 0;`

- `..... reminder section`

- `} while (1);`

- **Zadovoljava međusobno isključenje, ali ne progres (P0, P1, P0, P1)**

- ☞ (ako P1 ne uđe u svoj CS, P0 nikad neće ući)

# Algorithm 2 (No SA, but wrong)

- Deljene promenljive

- ☞ **boolean flag[2];**

- initially **flag [0] = flag [1] = false.**

- ☞ **flag [i] = true  $\Rightarrow$  P<sub>i</sub> spreman da uđe u kritičnoj sekciji**

- **Proces P<sub>i</sub>**

- do {

- flag[i] = true;**

- while (flag[j]) ;**

- critical section**

- flag [i] = false;**

- remainder section**

- } while (1);

- **Zadovoljava međusobno isključenje, ali ne progress**

- ☞ T0: P0 postavlja flag[0]=true; CPU je preempt-ovan u P1

- ☞ T1: P1 postavlja flag[1]=true; oba procesa se blokiraju zauvek

# Algoritam 3 Dekker-Peterson

- Kombinovane deljene promenljive algoritma 1 i 2.

- Proces  $P_i$

```
do {  
    flag [i] = true;  
    turn = j; /* daje šansu drugom procesu*/  
    while (flag [j] and turn = j) ;
```

**critical section**

```
flag [i] = false;
```

**remainder section**

```
} while (1);
```

- **Sva tri zahteva se ispunjavaju;**
- **rešava problem kritične sekcije za dva procesa.**

# Bakery Algorithm

## Kritična sekcija za n procesa

- Pre ulaska u kritičnu sekciju,
  - ☞ proces dobija broj
- Onaj koji ima najmanji broj ulazi u kritičnu sekciju
- Ako procesi  $P_i$  i  $P_j$  dobiju isti broj,
- if  $i < j$ , (i, j predstavlja vreme kreiranja procesa, ili PID)
- onda  $P_i$  se prvi izvršava; a ako ne  $P_j$  je se prvi izvršava
  - ☞ (prim. proces dobije broj,
  - ☞ onda je uspavan proces sa istim brojem probuđen)
- Šema brojeva uvek generiše brojeve od najmanjeg do najvećeg;
- prim., 1,2,3,3,3,3,4,5...

# Bakery Algorithm

- Notiranje  $\leq$  lexicographical order (**ticket #**, **process id #**)

- ☞ **(a,b) < (c,d)**

- ☞ **if a < c**

- ☞ ili

- ☞ **if a = c and b < d**

- ☞ **max (a<sub>0</sub>, ..., a<sub>n-1</sub>) je broj k, tako da k ≥ a<sub>i</sub> for i = 0, ..., n - 1**

- **Deljeni podaci:**

- boolean choosing[n];**

- int number[n];**

**Strukture podataka su inicijalizovane na false i 0 respektivno**

# Bakery Algorithm

do {

choosing[i] = true; *#čeka da dobije broj*  
number[i] = max(number[0], number[1], ..., number [n - 1])+1;  
choosing[i] = false;

for (j = 0; j < n; j++) {  
    while (choosing[j]) ;  
    while ((number[j] != 0) && (number[j],j) < (number[i],i)) ;  
}

**critical section**

number[i] = 0;

remainder section

} while (1);

# Hardverska sinhronizacija

- Testira i modifikuje sadržaj reči, atomski

```
boolean TestAndSet(boolean &target)
{
    boolean rv = target;

    target = true;

    return rv;
}
```

# Međusobno isključenje sa Test-and-Set

- Deljeni podaci:

```
boolean lock = false;
```

- Proces  $P_i$

```
do {  
    while (TestAndSet(lock)) ;  
    critical section  
    lock = false;  
    remainder section  
}
```

# Hardverska sinhronizacija

- Atomska razmena dve promenljive

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

# Međusobno isključenje sa razmenom

- Deljeni podaci (inicijalizovani na **false**):  
**boolean lock;**

Proces  $P_i$

do {

**key = true;**

**while (key == true) Swap(lock, key);**

**critical section**

**lock = false;**

**remainder section**

}

# Semafori

- Alat za sinhronizaciju koji ne zahteva **busy waiting**
- Semafor **S**, integer promenljiva
- može biti dostupan samo preko
- dve atomske operacije:

**wait (S):**

```
while  $S \leq 0$  do no-op;  
S--;
```

**signal (S):**

```
S++;
```

# Kritična sekcija $n$ Procesa

- **Deljeni podaci:**

  - semaphore **mutex**; //initially mutex = 1

- **Proces  $P_i$ :**

  - do {









      - wait(mutex);**









      - critical section**

      - signal(mutex);**

      - remainder section

  - } while (1);

# Implementacija semafora

- Definirati **semafor** kao zapis

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

- Pretpostavlja **dve jednostavne** operacije:

-  **block**

-  prekida proces.

-  **wakeup(*P*)**

-  nastavlja izvršavanje blokiranog procesa *P*

# Implementacija

- Semafor operacije su sad definisane kao:

**wait(S):**

**S.value--;**

**if (S.value < 0)**

**{**

**add this process to S.L;**

**block;**

**}**

**signal(S):**

**S.value++;**

**if (S.value <= 0)**

**{**

**remove a process P from S.L;**

**wakeup(P);**

**#but which one {FIFO, LIFO, priority}**

**}**

# Semafor kao glavni alat za sinhronizaciju

- Izvršava se **B** u  $P_j$  samo kad se **A** izvrši u  $P_i$
- Koristi se semafor **flag** inicijalizovan na 0
- Kod:

$P_i$	$P_j$
<b>A</b>	<b>wait(flag)</b>
<b>signal(flag)</b>	<b>B</b>

# Zastoj i zakucavanje

- **Deadlock** – dva ili više procesa beskonačno čekaju
- na događaj koji se nikada neće dogoditi.

- **Neka S i Q budu dva semafora inicijalizovana na 1**

☞ Zastoj nastupa ako  $P_0$  (`wait(S)`), onda  $P_1$  (`wait(Q)`), sledeće čekanje stavlja proces u stanje zastoja

$P_0$   
1. `wait(S)`;  
`wait(Q)`;  
.....  
`signal(S)`;  
`signal(Q)`

$P_1$   
2. `wait(Q)`;  
`wait(S)`;  
.....  
`signal(Q)`;  
`signal(S)`;

- **Zakucavanje** – beskonacno blokiranje

- ☞ Proces nikad nemože da se pomeri
- ☞ iz semafor queue
- ☞ u kojem je bio suspendovan

# Dva tipa semafora

## ■ Brojački semafor:

- integer vrednost nema ograničenje u domenu

## ■ Binarni semafor:

- integer vrednost je ograničen samo između 0 i 1;
- može biti jednostavniji za implementaciju

## ■ Može se implementirati

- brojački semafor  $S$
- kao binarni semafor

# Klasični problemi sinhronizacije

- **Problem ograničenog bafera**
- **Problem čitalaca i pisaca**
- **Večera filozofa**

# Problem ograničenog bafera

- Deljeni podaci

**semaphore** **full**, **empty**, **mutex**;

Initially:

**full** = 0, **empty** = n, **mutex** = 1

- **empty**: broji prazne bafere
- **full**: broji pune bafere

# Problem ograničenog bafera

## Proizvođač-proces

```
do {
```

```
    ...
```

```
    produce an item in nextp
```

```
    ...
```

```
    wait(empty); /*buffer full*/
```

```
    wait(mutex); /* ulaz u kritičnu sekciju*/
```

```
    ...
```

```
    add nextp to buffer
```

```
    ...
```

```
    signal(mutex); /*oslobodi bafer i kritičnu sekciju*/
```

```
    signal(full); /*uvećaj broj popunjenih elemenata u  
baferu*/
```

```
} while (1);
```



# Problem ograničenog bafera

## Potrošač proces

```
■ do {  
■ wait(full); /* bafer prazan*/  
■ wait(mutex); /*ulaz u kritičnu sekciju*/  
■ ...  
■ remove an item from buffer to nextc  
■ ...  
■ signal(mutex); /*oslobodi bafer i kritičnu sekciju*/  
■ signal(empty); /*uvećaj broj slobodnih mesta*/  
■ ...  
■ consume the item in nextc  
■ ...  
■ } while (1);
```

# Problem čitalaca i pisaca

- **Pravila:** (tipični deljeni fajlovi ili zapisi)
  - ☞ samo jedan pisac u jedinici vremena (pisac ima ekskluzivan pristup)
  - ☞ vise čitalaca, simultano
  - ☞ kad je pisac aktivan, nijedan čitalac nemože imati pristup
  - ☞ kad je neki čitalac aktivan, nijedan pisac nema pristup

## ■ Deljeni podaci

semaphore **mutex**, **wrt**;

Initially

**mutex = 1**, **wrt = 1**, **readcount = 0**

- **wrt = mutex za čitaoce i pisce**
- **mutex = mutex za readcount update**
- **readcount = broj procesa koji čita objekat**

# Problem čitalaca i pisca (writer proces)

```
wait(wrt); /* no readers*/
```

...

```
writing is performed
```

...

```
signal(wrt);
```

# Problem čitalaca i pisca (reader proces)

**entry**

**wait(mutex);**  
**readcount++;**

**if (readcount == 1) wait(wrt);** /\* prvi reader, čeka writer \*/  
**else continue** /\*nije prvi, ima readers\*/

**signal(mutex);**

...

reading is performed

...

**wait(mutex);**  
**readcount--;**

**if (readcount == 0) signal(wrt);**

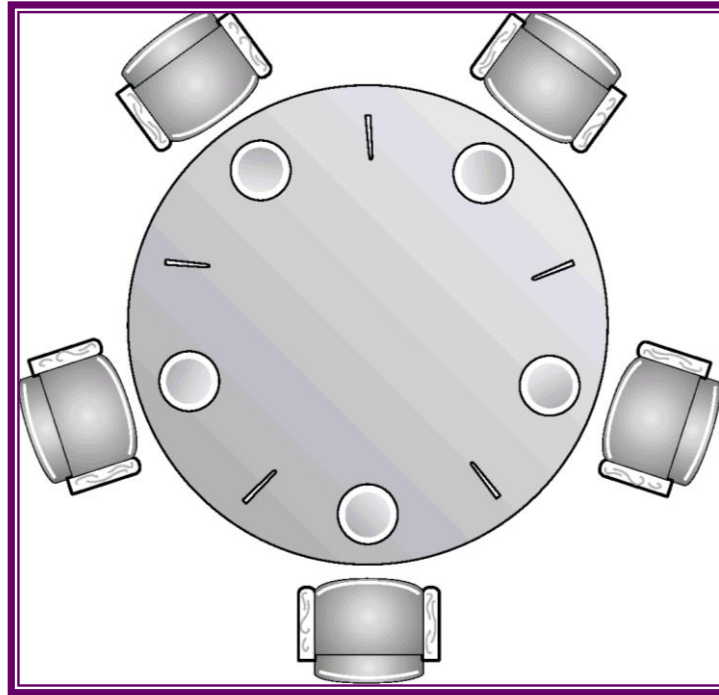
#no readers, freeing wrt

**signal(mutex);**

**go to transactions**

**exit**

# Večera filozofa



- Deljni podaci  
**semaphore chopstick[5];**

Inicijalno sve vrednosti su 1

# Večera filozofa

## ■ Filozof i:

```
do {  
    wait(chopstick[i])  
    wait(chopstick[(i+1) % 5])  
    ...  
    eat  
    ...  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    ...  
    think  
    ...  
} while (1);
```

# Kritični regioni

- Konstrukcija sinhronizacije visokog nivoa
- **Zajednička promenljiva v tipa T**, je deklarirana kao:

**v: shared T**

- Promenljivoj **v se može pristupati** samo unutar regiona **naredbi S**

**region v when B do S**

gde je **B logički izraz**

- Dok se naredba **S izvršava,**
- nijedan drugi proces nemože pristupiti **promenljivoj v**

# Kritični regioni

- **Regioni koje se oslanjaju na istu deljenu promenljivu ( $v$ )**
- **isključuju jedan drugog u vremenu**
- **Uslovni kritični regioni**
- **Kad proces pokuša da izvrši naredbu regiona,**
- **logički izraz  $B$  se ispituje:**
  - ☞ **Ako je  $B$  true**
    - 📄 naredba  **$S$  se izvršava**
  - ☞ **Ako je  $B$  false**
    - 📄 **proces se blokira dok  $B$  ne postane true**
    - 📄  **$i$**
    - 📄 **nema drugih proces u tom regionu koji je asociran sa  $v$**

# Primer – Bounded Buffer

- Deljeni podaci:

```
struct buffer           #all in structure is shared
{
    int pool[n];
    int count, in, out;
}
```

# Bounded Buffer (Proizvođač proces)

- Proizvođač proces ubacuje **nextp** u zajednički bafer

```
region buffer when( count < n)
{
    pool[in] = nextp;
    in := (in+1) % n;
    count++;
}
```

# Bounded Buffer (Potrošač proces)

- Proces potrošač pomera element iz zajedničkog bafera i postavlja u **nextc**

**region buffer when (count > 0)**

```
{
```

```
nextc = pool[out];
```

```
out = (out+1) % n;
```

```
count--;
```

```
}
```

# Monitori

- Konstrukcija sinhronizacije visokog nivoa koja omogućava sigurno deljenje jednog apstraktnog tipa podatka među konkurentnim procesima.

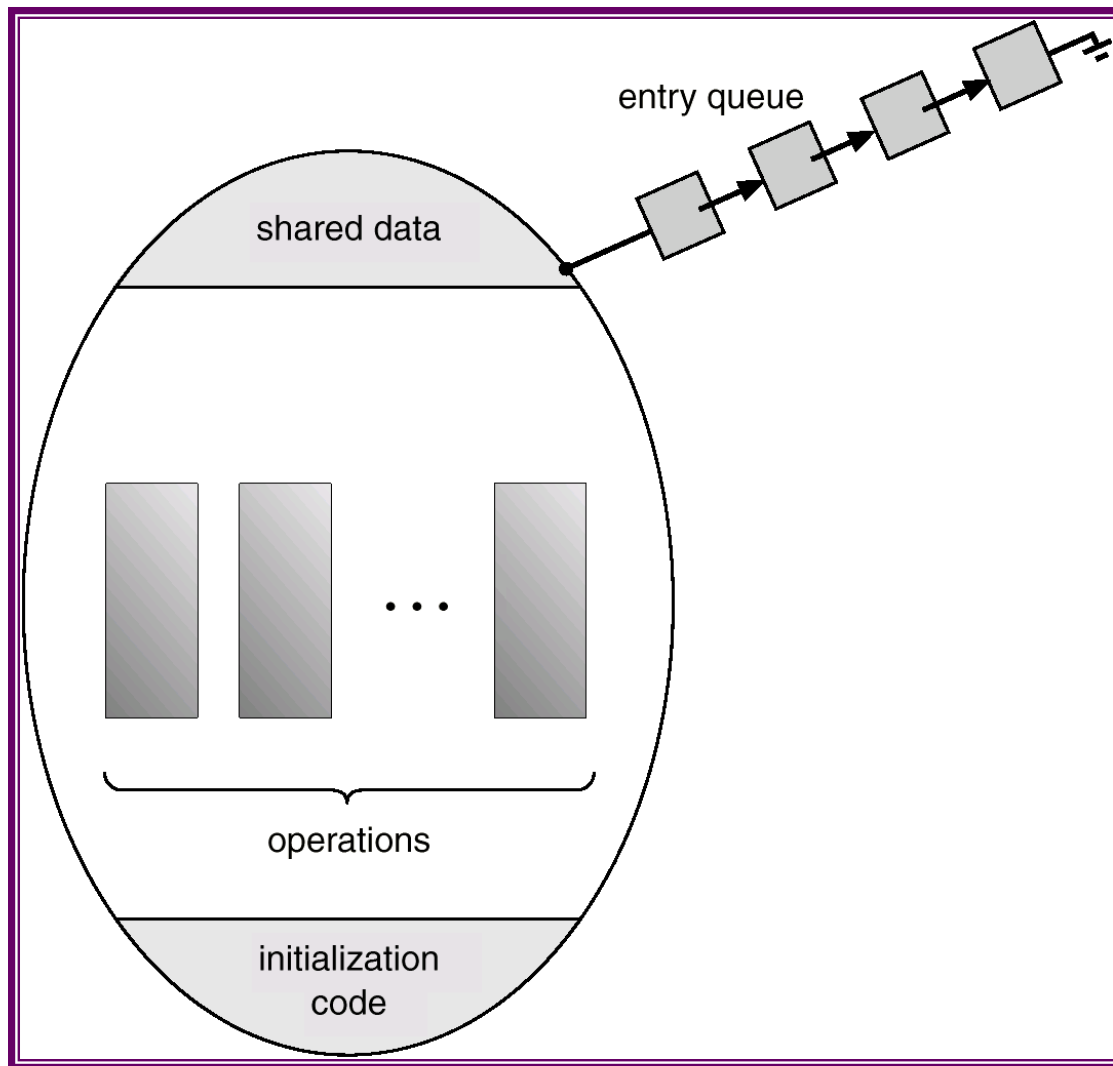
```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        ...
    }
    procedure body P2 (...) {
        ...
    }
    procedure body Pn (...) {
        ...
    }
    {
        initialization code
    }
}
```

}

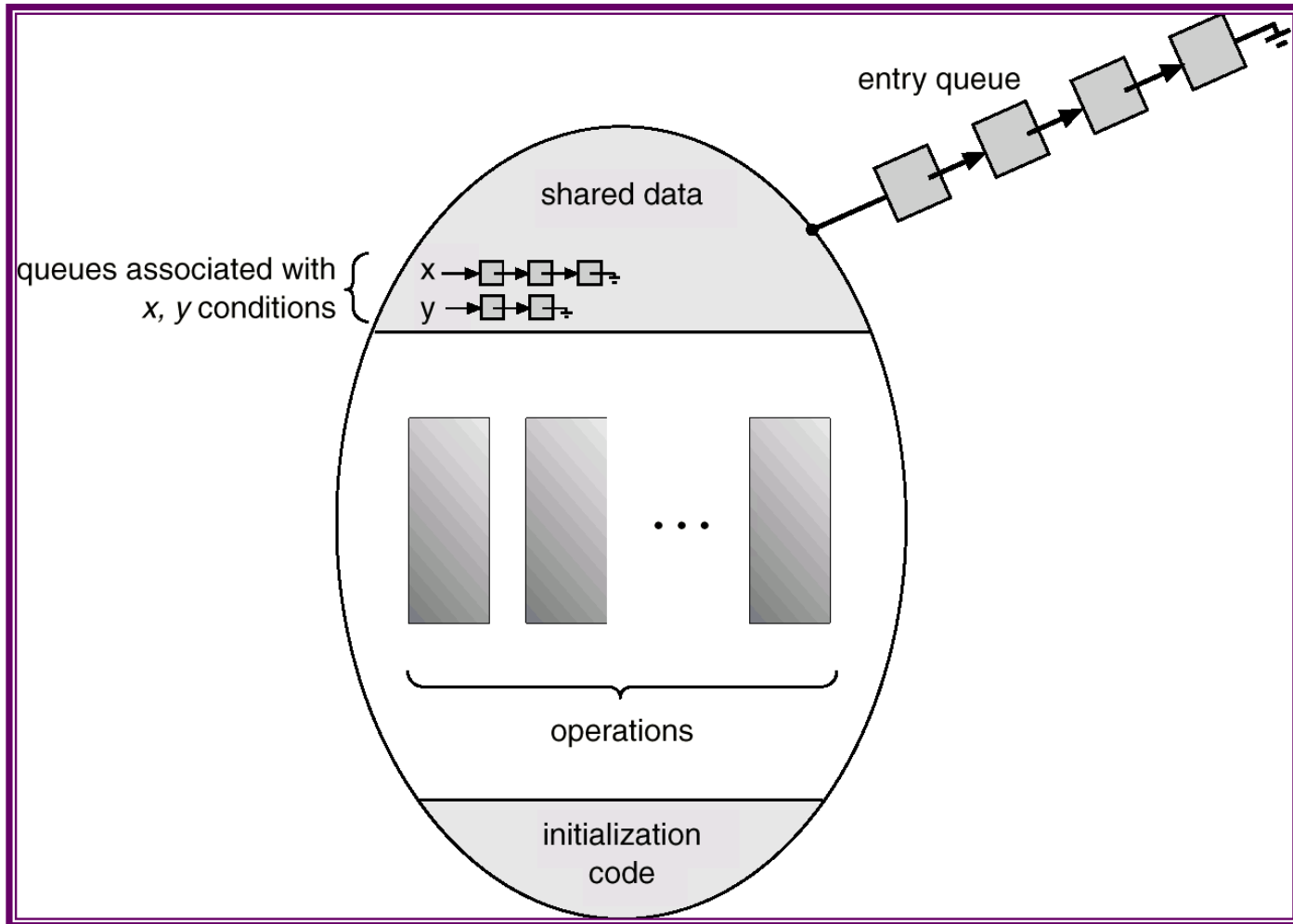
# Monitori

- Da bi se dozvolilo procesu da čeka u monitoru,
- uslovna promenljiva mora biti deklarirana, kao
- **condition** `x, y;`
- Uslovna promenljiva može biti upotrebljena samo sa operacijama **wait** i **signal**:
  - ☞ Operacija  
**`x.wait();`**  
znači da proces koji je pozvao operaciju se suspenduje sve dok **drugi proces** ne pozove kontra operaciju  
**`x.signal();`**
    - ☞ **`x.signal`** operacija će odblokirati tačno jedan suspendovan proces.
    - ☞ Ako nema suspendovanih procesa, onda **signal** operacija nema efekta.

# Šematski prikaz na monitor



# Monitor With Condition Variables



# Primer - Večera filozofa

**monitor dp**

{

enum {thinking, hungry, eating} state[5];

condition self[5];

void pickup(int i)

// following slides

void putdown(int i)

// following slides

void test(int i)

// following slides

void init()

{

for (int i = 0; i < 5; i++)

state[i] = thinking; //init code

}

}

# Večera filozofa

```
void pickup(int i)
```

```
{  
    state[i] = hungry;  
    test[i];  
    if (state[i] != eating)  
        self[i].wait();  
}
```

```
void putdown(int i)
```

```
{  
    state[i] = thinking;  
    // testira levog i desnog komsiju  
    test((i+4) % 5); /*daje šansu levom komšiji da jede*/  
    test((i+1) % 5); /*daje šansu desnom komšiji da jede */  
}
```

# Večera filozofa

```
void test(int i)
```

```
{
```

```
if((state[(i + 4)%5] != eating) && (state[i] == hungry)&&(state[(i + 1) % 5] != eating))
```

```
{
```

```
state[i] = eating;
```

```
self[i].signal();
```

```
}
```

```
}
```

# Večera filozofa

each philosopher do it:

**dp.pickup(i)**

.....

**eat**

**dp.putdown(i)**