

# Sisteme de Operare

---

- **Gestiunea proceselor**
  - Sincronizarea proceselor
  - Secțiunea critică
    - semafoare, așteptare activă, monitoare, bariere
  - Paradigme ale programării concurente

# Sincronizarea proceselor

---

- ❑ Pentru execuție procesele au nevoie de acces la resursele sistemului : CPU, memorie, disc etc.
- ❑ Resursele pot fi:
  - locale sau globale,
  - private sau comune (partajabile).
- ❑ Toate resursele sunt critice (accesibile numai unui proces la un moment dat).
- ❑ Procesele concurează pentru obținerea accesului la resurse.

# Sincronizarea proceselor

---

- **Definiție: Sincronizarea proceselor** reprezintă acțiunea de coordonare a activității mai multor procese.
  - Sincronizarea presupune posibilitatea de a modifica starea unui proces la un moment dat până la apariția unor evenimente ce nu se afla sub controlul acestuia.
  - Sincronizarea impune aplicarea excluderii mutuale și o ordonare a evenimentelor.
- **Definiție:** Zona de program prin care se apelează o resursă critică se numește **secțiune critică (SC)**.
- Accesul unor procese la resursele critice se face printr-un protocol de **excludere mutuală (EM)**, ce presupune o sincronizare ce permite modificarea stării proceselor și eventual comunicații.

# Accesul la secțiunea critică

---

- intrarea în secțiunea critică
- tratarea resursei critice din secțiunea critică
- ieșirea din secțiunea critică.

# Condițiile pentru realizarea excluderii mutuale într-o secțiune critică

---

- Un singur proces la un moment dat trebuie să execute instrucțiunile din secțiunea critică ;
- Dacă mai mult de un proces sunt blocate la intrarea în secțiunea critică și nici un alt proces nu execută instrucțiunile din secțiunea critică, într-un timp finit unul din procesele blocate se deblochează și va intra în secțiunea critică (pe rând vor intra și celelalte).
- Blocarea unui proces în afara secțiunii critice să nu împiedice intrarea altui proces în secțiunea critică.
- Să nu existe procese privilegiate (mecanismul să fie echitabil pentru toate procesele).

# Excluderea mutuală - implementare

- ❑ Semafoare
- ❑ Așteptare pe condiție (așteptare activă)
- ❑ Monitoare
- ❑ Bariere

# Semafoare

- **Definitie:** Semaforul este un **mecanism de sincronizare** a execuției proceselor care acționează în mod concurent asupra unor resurse partajate.
  - Poate fi o variabilă sau o structură de date abstractă
  - Este utilizat pentru controlul accesului la o resursă comună într-un mediu multiproces, multiutilizator
- Semafoarele se caracterizează prin:
  - Valoare:  $val(s)$
  - coadă de așteptare:  $Q(s)$
- Semafoarele sunt de două tipuri :
  - binare (lucrează cu valori de 0 și 1)
  - întregi (lucrează cu valori între  $-n$  și  $n$ ).
- Cozile de așteptare sunt de tip FIFO.
- **Operațiile cu semafoare:**
  - creare
  - distrugere
  - operația de intrare în secțiunea critică :  $p(s)$ ,  $p$  = cerere de intrare în S.C.
  - operația de ieșire din secțiunea critică :  $v(s)$ ,  $v$  = cerere de ieșire în S.C.

# Semafoare

---

**P(s) :**

val(s) = val(s)-1 ;

if ( val(s) < 0 )

{

*//trece procesul în coada de așteptare a semaforului respectiv*

*// schimbă starea procesului în blocat*

}

**V(s):**

val(s) = val(s) +1 ;

if ( val(s) ≤ 0 )

{

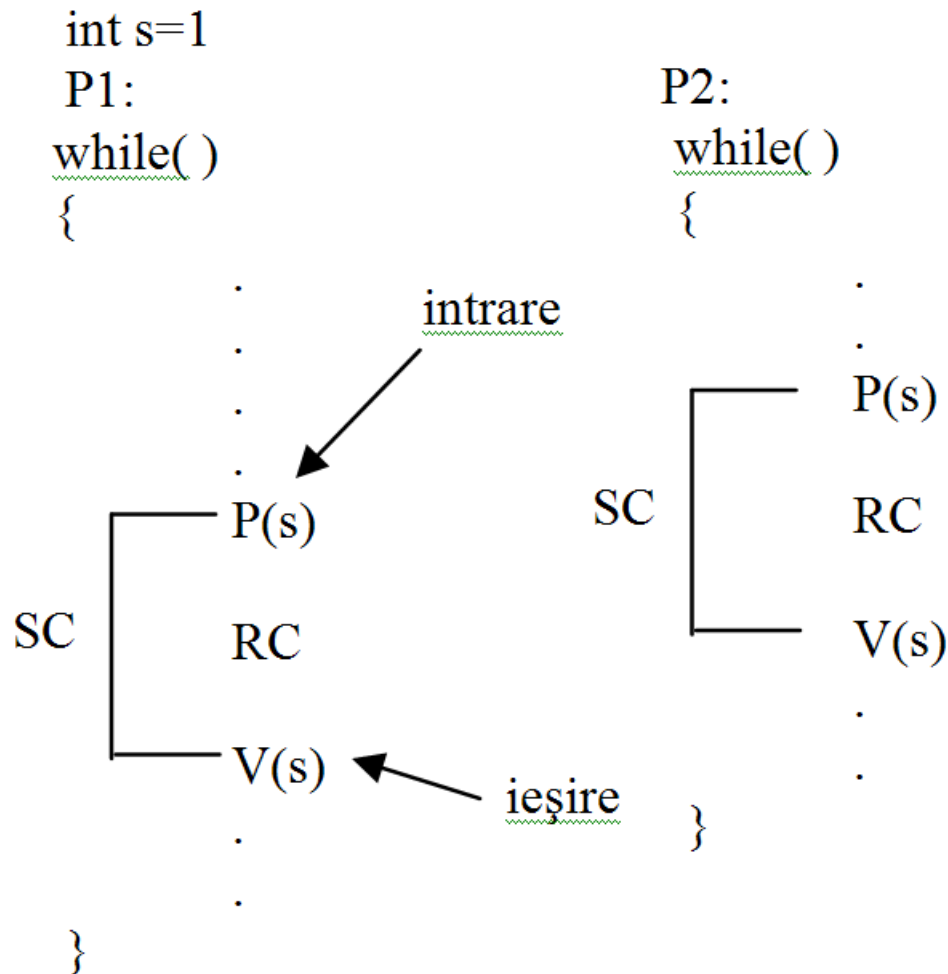
*//scoate primul proces din coada de așteptare a semaforului*

*// schimbă starea procesului în gata de execuție*

}



# Excluderea mutuală pentru 2 procese



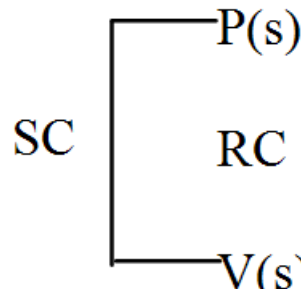
# Valoarea unui semafor la un moment dat

- **$V(s) = V_0(s) + nv(s) - np(s)$** 
  - $V(s)$  este valoarea unui semafor la un moment dat;
  - **$np$**  este numărul de treceri prin  $P(s)$
  - **$nv$**  este numărul de treceri prin  $V(s)$
- Dacă  $V(s) > 0$ , acest număr ne dă numărul de procese ce pot executa instrucțiunile din secțiunea critică.
- Dacă  $V(s) < 0$ , atunci acest număr ne dă numărul de procese blocate la semafor.
- Secțiunile critice pot fi atât imbricate cât și întrețesute.
- Numărul de procese trecute de semaforul  **$s$**  este:
  - **$nt(s) = \min\{V_0(s) + nv(s), np(s)\}$**

# Semafoare private

```
int s=0
P1:
while( )
{
    .
    .
    .
    .
    P(s)
    RC
    V(s)
    .
    .
}

P2:
while( )
{
    .
    .
    if (cond)
        V(s)
    .
    .
}
```



- numai un singur proces poate aplica P și V asupra lui
- celelalte procese putând executa numai V.
- Valoarea inițială a unui semafor privat este 0.
- Semaforul privat permite sincronizarea execuției unui proces cu o condiție externă lui (eventual cu un alt proces).

# Proiectarea semafoarelor

---

- ❑ Secțiunile critice trebuie încadrate de P și V ceea ce uneori este mai greu de urmărit ;
- ❑ Un proces nu poate fi distrus în SC ;
- ❑ Verificarea corectitudinii programelor nu este ușoară ;
- ❑ Gestiunea cozii poate duce la apariția unor probleme de proiectare.

# Sincronizarea proceselor cu semafoare

---

- ❑ Semafoarele întregi se folosesc pentru a gestiona un nr. de resurse identice (componentele unei zone tampon și resurse fizice).
- ❑ Resursele se alocă la cerere și se eliberează după folosirea lor.
- ❑ Când nu mai există copii disponibile procesele se blochează.

# Problemă:

## 3 resurse identice, un semafor și 3 procese

---

- execuția primitivelor P este critică

```
int s = 3;
```

```
p1()
```

```
{
```

```
.....
```

```
P(s);
```

```
(*) – se ocupa prima SC
```

```
P(s);
```

```
(*) – se ocupa a II-a SC
```

```
V(s);
```

```
V(s);
```

```
.....
```

```
}
```

```
p2()
```

```
{
```

```
.....
```

```
P(s);
```

```
(*) – se ocupa prima SC
```

```
P(s);
```

```
(*) – se ocupa a II-a SC
```

```
V(s);
```

```
V(s);
```

```
.....
```

```
}
```

```
p3()
```

```
{
```

```
.....
```

```
P(s);
```

```
(*) – se ocupa prima SC
```

```
P(s);
```

```
(*) – se ocupa a II-a SC
```

```
V(s);
```

```
V(s);
```

```
.....
```

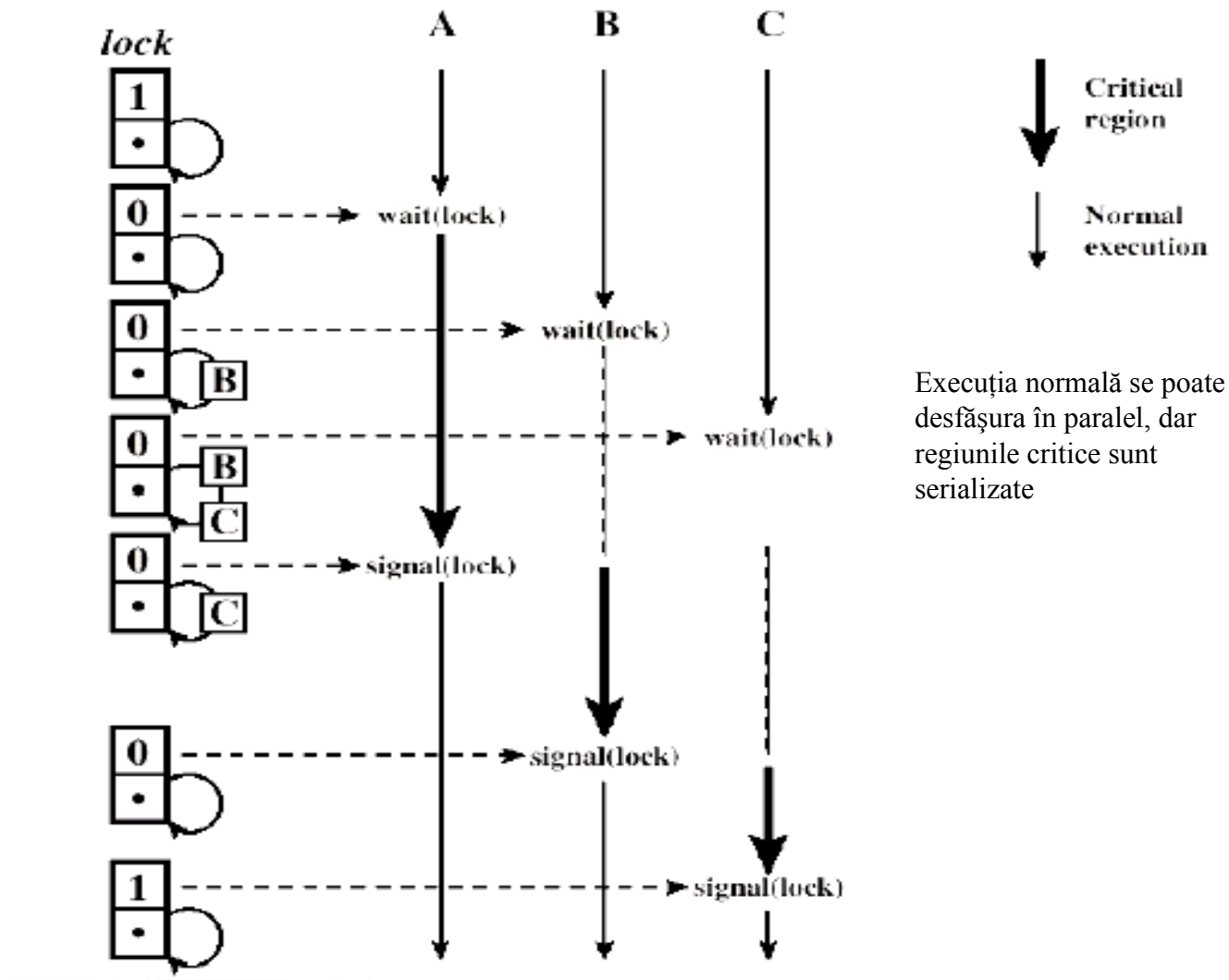
```
}
```

# Rezolvare

---

```
int s =3, s1 =2;
  p1()
  {
    .....
    P(s1);
    P(s);
    // – se ocupa prima RC
    P(s);
    // – se ocupa a II-a RC
    V(s);
    V(s);
    V(s1);
  }
```

# Exemplu de execuție pentru 3 procese





# Sincronizarea prin așteptare activă

---

- condiții:
  - există variabile comune care sunt testate;
  - testul și modificarea variabilelor să fie operații indivizibile.

```
boolean flag [2] = {false, false};
```

```
int turn;
```

```
void P0( )
```

```
{  
    while (true)  
    {
```

```
        flag [0] = true;
```

```
        turn = 1;
```

```
        while (flag [1] && turn == 1);
```

```
        /* RC */;
```

```
        flag [0] = false;
```

```
        ....
```

```
    }  
}
```

```
void P1( )
```

```
{  
    while (true)  
    {
```

```
        flag [1] = true;
```

```
        turn = 0;
```

```
        while (flag [0] && turn == 0);
```

```
        /* RC */;
```

```
        flag [1] = false;
```

```
        ....
```

```
    }  
}
```

SC are 3 componente :

– protocolul de intrare

– corpul

– protocolul de iesire

— protocolul de intrare

— corpul RC

— protocolul de ieșire

# Dezavantajele aşteptării active

- ❑ consumarea inutilă de timp CPU pentru un proces care aşteaptă.
- ❑ Există 2 operaţii care trebuie realizate: **testarea** şi **modificarea** `flag` (de aceea au fost introduse variabilele `flag` şi `turn`).
- ❑ Gradul ridicat de dificultate în elaborarea protoalelor de intrare şi ieşire lucru ce poate duce pe lângă neclarităţi şi la apariţia de erori.

# Sincronizarea folosind monitoare

- Un monitor reprezintă o structură de date formată din:
  - **variabile de sincronizare** numite și **variabile condiții**,
  - **resurse partajate**
  - **proceduri de acces** la resurse. Procedurile pot fi: externe sau interne.

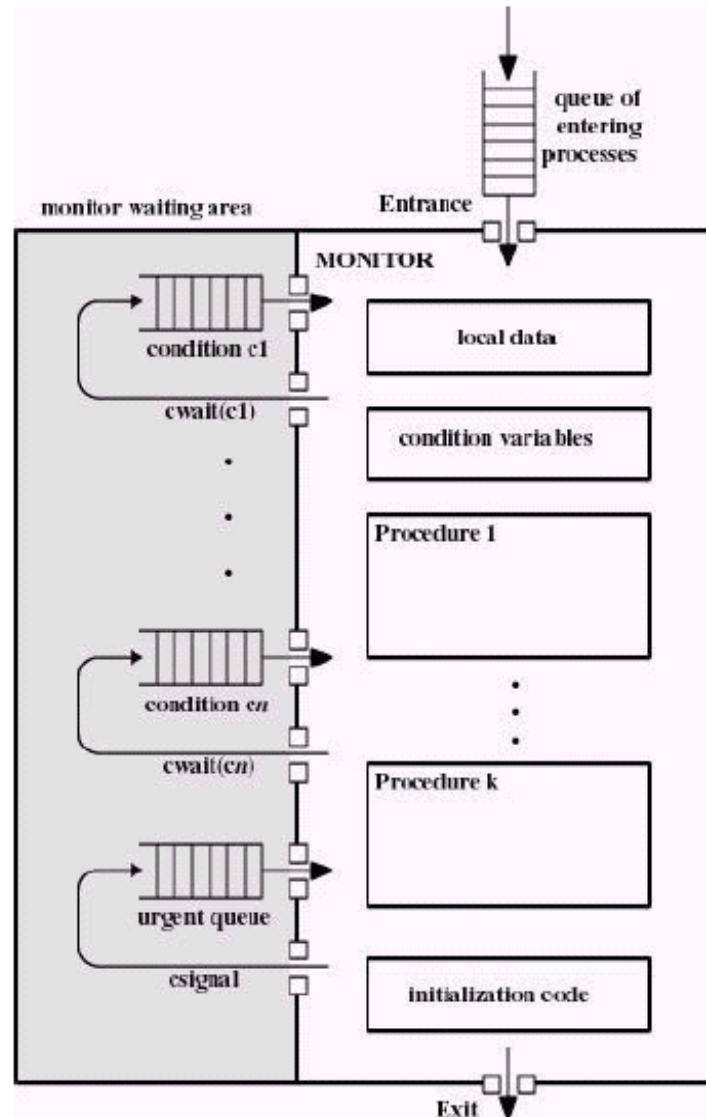
# Sincronizarea folosind monitoare

```
monitor monitor-name
{
    delarații variabile locale;

    procedure P1(...) {
        ...
    }
    ...
    procedure Pn() {
        ...
    }
}
```

- **Variabilele locale** ale unui monitor nu sunt vizibile decât pentru procedurile lui.
- **Procedurile unui monitor** sunt puncte de intrare (pot fi accesate din exterior) și ele se executa prin excludere mutuală.
- **Primitivele de sincronizare**
  - **wait** – suspendă procesul care execută wait pe o variabilă de condiție și face disponibil monitorul pentru un alt apel;
  - **signal** – reactivează un proces în așteptare pe o variabilă de condiție.

# Structura unui monitor



# Cozile de așteptare ale unui monitor

- Fiecare variabilă  $c_i$  are atașată o coadă și odată accesată o procedură a monitorului, un proces în așteptare pe o condiție va ceda accesul unui alt proces în așteptare la intrarea în monitor.
- Procesele suspendate după execuția lui *signal* ( $c_i.signal$ ) nu eliberează excluderea mutuală în monitor deoarece:
  - fie există un proces în execuție a unei proceduri în monitor, eventual în așteptare pe o condiție;
  - fie este același proces care-si continuă execuția când nu mai sunt altele de reactivat.
- Coada proceselor suspendate este mai prioritara față de aceea de la intrarea în monitor și a cozilor de așteptare pe condiție

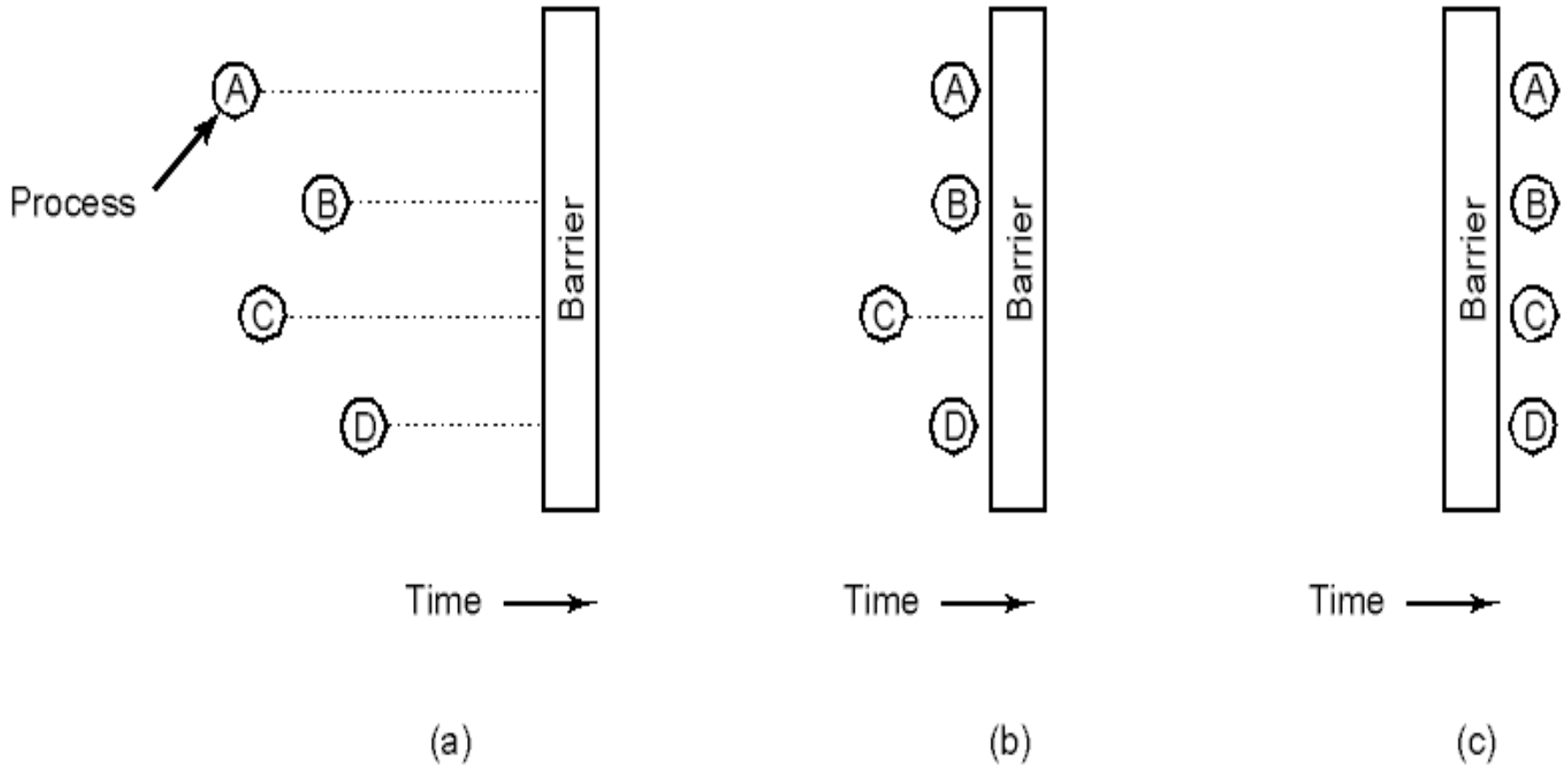
# Stările unui proces la accesarea unei proceduri din monitor

---

- aşteptare în coada de intrare a monitorului ;
- aşteptare într-o coadă pe o variabilă de condiție (`wait`) ;
- suspendarea prin execuția `signal`, care reactivează un proces în aşteptare pe o variabilă de condiție ;
- execuția normală a instrucțiunilor unei proceduri din monitor.



# Sincronizarea folosind bariere



# Sincronizarea prin transmiterea de mesaje

---

- ❑ Mesajele pot avea dimensiune fixă sau nu.
- ❑ Dacă două procese P și Q vor să transmită mesaje trebuie să stabilească o legătură între ele.
- ❑ Detalii de implementare și gestionare a comunicațiilor sunt lăsate în grija sistemului de operare.
- ❑ Sistemul de transmitere a mesajelor trebuie să ofere funcții de tip:
  - `send(destinație, mesaj)`
  - `receive(sursă, mesaj)`

# Sincronizarea prin transmiterea de mesaje

---

- Atât procesul care trimite mesajele, cât și cel care le primește pot fi blocate sau nu
- **Send/receive blocante**: ambele procese sunt blocate până la terminarea comunicării ;
- **Send neblokant, receive blocant** : procesul ce execută send își continuă execuția imediat ce a trimis mesajul, iar procesul ce execută receive este blocat până la sosirea mesajului ;
- **Send/receive neblocante**: nici un proces nu așteaptă terminarea operațiilor de comunicație;

# Sincronizarea prin transmiterea de mesaje

## Adresarea

---

- ❑ **directă**: în acest caz, funcțiile send/receive includ identificatorul procesului destinație ;
- ❑ **indirectă**: în acest caz mesajele sunt trimise unei structuri de date partajate ce constau în cozi numite mailbox (casuțe poștale)

# Excluderea mutuală folosind mesaje

---

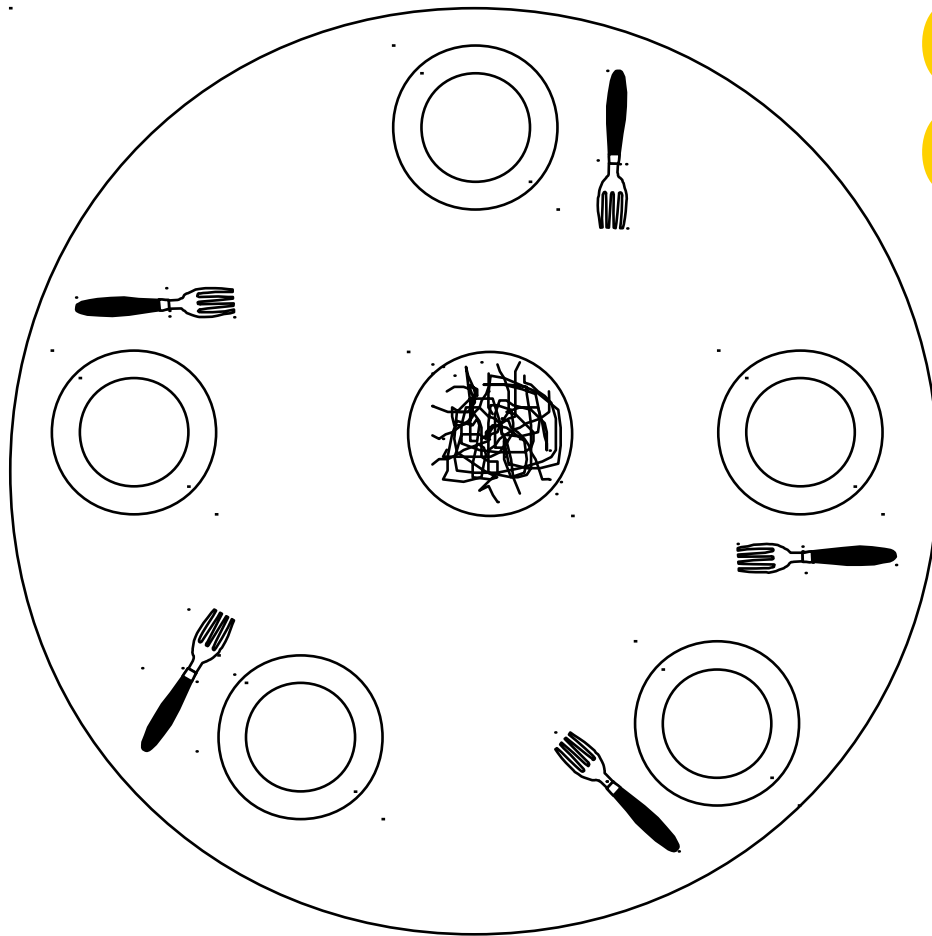
```
const int n /* number of processes */;
void P(int i)
{
    message msg;
    while (true)
    {
        receive (mutex, msg);
        /* critical section */;
        send (mutex, msg);
        /* remainder */;
    }
}
void main()
{
    create_mailbox (mutex);
    send (mutex, null);
    parbegin (P(1),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

# Paradigme ale programării concurente

---

- ❑ Problema celor 5 filosofi
- ❑ Problema producător – consumator
- ❑ Problema cititori/scriitori
- ❑ Problema bărbierului

# Problema celor 5 filosofi



- ▣ Resurse: furculite
- ▣ Procese: filosofi.

# Problema celor 5 filosofi

---

```
semaphore fork[5] = {1,1,1,1,1};
int i;
void philosopher(int i)
{
while(true)
{
    think( );
    P(fork[i]);
        P(fork[(i+1) mod 5]);
        eat( );
        V(fork[(i+1) mod 5]);
        V(fork[i]);
}
}
```

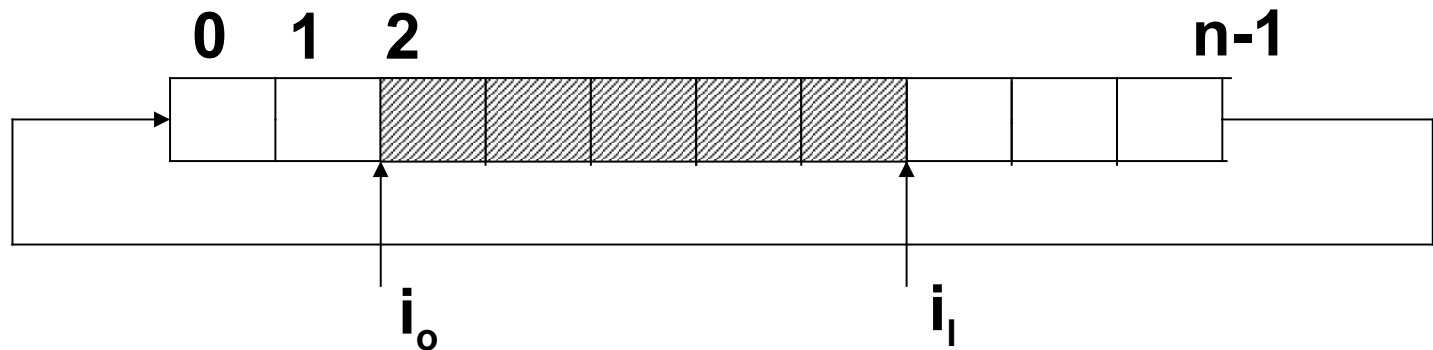
```
int fork[5] = {1,1,1,1,1};
int valet = 4; int i ;
void philosopher(int i)
{
    while (true)
    {
        P(valet); // - intră in cameră
        P(fork[i]); // - ridică furculița stânga
        P( fork[( i + 1) % 5]); // - ridică furculița
dreapta
        // - mănâncă
        V(fork[i]); // - eliberează furculița stânga
        V( fork[( i + 1) % 5]); // - eliberează
furculița dreapta
        V(valet); // - iese din cameră
    }
}
```



# Problema producător – consumator

- implementează un protocol de comunicație între 2 sau mai multe procese care utilizează 2 sau mai multe resurse duale (locațiile ocupate sau libere dintr-un buffer):
- unul sau mai mulți producători introduc datele în buffer;
- un singur consumator extrage datele din buffer.

# Problema producător – consumator buffer-ul circular cu $n$ locații



## □ Condiții statice:

- elementele din buffer sunt citite de consumator în aceeași ordine în care sunt scrise de producător
- nici un element nu trebuie pierdut sau introdus în plus.

## □ Constante:

- $i_0$  – locația unde este un mesaj care se poate citi;
- $i_1$  – locația de la care putem scrie.

# Problema producător – consumator

## Condiții de sincronizare:

---

- ❑ dacă buffer-ul este gol consumatorul se blochează până când există cel puțin un mesaj.
- ❑ dacă buffer-ul este plin producatorul se blochează până când există cel puțin o locație liberă.

# Problema producător – consumator

## 1 producător - 1 consumator

---

```
int io = ii = 0;
int liber = n, ocupat = 0;
int buf[n];
void producator (int mes)
{
    .....
    P(liber);
    buf [ii] = mes; //adaugă mesaj
    ii = (ii + 1) % n;
    V(ocupat);
}
```

```
int consumator()
{
    .....
    P(ocupat);
    mes = buf [io]; //citește mesaj
    io = (io + 1) % n;
    V(liber);
    .....
    consuma(mes);
}
```

## „k” producători și „l” consumatori

---

- $i_p$  este resursă critică pentru producători
- se introduce un semafor **liber** pe care îl inițializăm cu **n**.
- $i_c$  resursă critică pentru consumatori
- se introduce semaforul **ocupat** inițializat cu 0.
- Semaforul **s** asigură accesul la secțiunea critică și este inițializat cu 1.

# Problema producător – consumator

## „k” producători și „l” consumatori

---

```
int io = il=0;
int liber =n, ocupat =0, s =1;
void producator(int mes)
{
    .....
    P(liber);
    P(s);
    buf[il] =mes;    // adaugă mesaj
    il =(il +1) %n;
    V(s);
    V(ocupat);
    .....
}
```

```
int consumator()
{
    .....
    P(ocupat);
    P(s);
    mes=buf[io];    //citește mesaj
    io =(io +1) %n;
    V(s);
    V(liber);
    consuma(mes);
}
```

# Problema producător – consumator buffer infinit

---

- un semafor **s** pentru a realiza excluderea mutuală asupra buffer-ului;
- un semafor **n** pentru a sincroniza producătorul și consumatorul asupra dimensiunii buffer-ului

# Problema producător – consumator buffer infinit

---

```
int n = 0, s = 1; //semafoare
```

```
int io = ii = 0;
```

```
void producător ()
```

```
{  
    while (true)  
    {  
        produce();  
        P(s);  
        buf[ii] = mes; // adaugă mesaj  
        ii = (ii + 1) % n;  
        V(s);  
        V(n);  
    }  
}
```

```
void consumator()
```

```
{  
    while (true)  
    {  
        P(n);  
        P(s);  
        mes = buf[io]; //citește mesaj  
        io = (io + 1) % n;  
        V(s);  
        consume();  
    }  
}
```