

# Sisteme de Operare

## Sincronizarea proceselor partea I-a

**Cristian Vidrașcu**

<https://profs.info.uaic.ro/~vidrascu>

- Introducere
- Problema secțiunii critice
  - Enunțul problemei și cerințele de rezolvare
  - Soluții pentru cazul a două procese
  - Soluții pentru cazul a  $n > 2$  procese
  - Soluții hardware
  - Soluții concrete: *spinlock*-uri și semafoare
- Interblocajul și înfometarea
- Probleme clasice de sincronizare
  - Problema Producător-Consumator
  - Problema Cititori și Scriitori
  - Problema Cina Filozofilor
  - Problema Bărbierului Adormit
- Monitoare (și alte abordări ale problemei SC)

# Introducere

- Procesarea paralelă și concurentă este fundamentul sistemelor de operare multiprogramate
- Pentru a asigura execuția ordonată, în paralel și concurent, a mai multor programe, SO-ul trebuie să ofere mecanisme pentru sincronizarea și comunicația proceselor

## ➤ “race conditions”:

Uneori procesele cooperante pot partaja un spațiu de stocare pe care fiecare proces îl poate citi și scrie (acest spațiu comun poate fi e.g. o zonă în memoria principală, ori un fișier pe disc).

În acest context, rezultatul final al execuției acelor procese poate depinde nu numai de datele de intrare, ci și de “cine execută exact ce anume și când anume” (i.e. de ordinea exactă de execuție a instrucțiunilor atomice ale acelor procese).

Asemenea situații sunt denumite “race conditions”.

*Notă:* uneori mai sunt numite și “data races”, sau “Heisenbugs”.  
(A se vedea <https://en.wikipedia.org/wiki/Heisenbug>.)

# Introducere

## ➤ Exemplu de “race condition”:

Două procese A și B care partajează o variabilă  $v$ , inițializată cu valoarea 0. Procesul A execută operația  $v := v + 1$ , iar B execută operația  $v := v - 1$ . Rezultatul firesc al execuției celor două procese ar fi ca variabila  $v$  să aibă în final tot valoarea 0.

Cele două operații constau de fapt din următoarele secvențe de instrucțiuni atomice (în limbaj mașină):

A: LD reg1, adr\_v  
INC reg1  
ST reg1, adr\_v

B: LD reg2, adr\_v  
DEC reg2  
ST reg2, adr\_v

Dacă ambele instrucțiuni Load sunt executate înaintea ambelor instrucțiuni Store, atunci rezultatul final este eronat → justificare:

# Introducere

## ➤ Exemplu de “race condition” (cont.):

Când cele două secvențe de cod A și B sunt executate în paralel (prin paralelism real, i.e. pe multi-CPU, sau doar aparent, i.e. pe 1 CPU prin *time-slicing*), sunt posibile următoarele 6 ordini de execuție:

- 1)  $LD_A \rightarrow ST_A \rightarrow LD_B \rightarrow ST_B$  ; în acest caz rezultatul final este  $v=0$
- 2)  $LD_B \rightarrow ST_B \rightarrow LD_A \rightarrow ST_A$  ; și în acest caz rezultatul final este  $v=0$
- 3)  $LD_A \rightarrow LD_B \rightarrow ST_A \rightarrow ST_B$  ; în acest caz rezultatul final este  $v=-1$  (!)
- 4)  $LD_B \rightarrow LD_A \rightarrow ST_B \rightarrow ST_A$  ; în acest caz rezultatul final este  $v=+1$  (!)
- 5)  $LD_A \rightarrow LD_B \rightarrow ST_B \rightarrow ST_A$  ; și în acest caz rezultatul final este  $v=+1$  (!)
- 6)  $LD_B \rightarrow LD_A \rightarrow ST_A \rightarrow ST_B$  ; și în acest caz rezultatul final este  $v=-1$  (!)

Important: cele 6 cazuri descriu ordinele temporale diferite în care se pot efectua cele 4 operații LD și ST. Aceste 4 operații lucrează cu regiștri diferiți, dar cu ACEEAȘI adresă de memorie (i.e., adresa unde este stocată variabila  $v$  în memoria RAM).

Cazurile în care rezultatul final este eronat sunt caracterizate prin: “ambele instrucțiuni Load sunt executate înaintea ambelor instrucțiuni Store”.

# Introducere

## ➤ Cum evităm “race conditions” ?

Ideea de evitare a acestor situații, ce sunt nedorite în orice context ce implică o resursă/informație partajată (e.g. memorie partajată, ori fișiere partajate, sau orice alt fel de resursă partajată), constă în a găsi un mecanism pentru a împiedica să avem mai mult de un singur proces, în același timp, care să citească sau să scrie resursa/informația partajată.

# Introducere

## ➤ Cum evităm “race conditions” ? (cont.)

Cu alte cuvinte, avem nevoie de **excludere mutuală**, adică de o tehnică care să ne garanteze că dacă un proces utilizează o resursă partajată, atunci toate celelalte procese vor fi împiedicate să o utilizeze.

*Notă:* alegerea operațiilor primitive adecvate pentru realizarea excluderii mutuale este o problemă de proiectare majoră la toate nivelurile: arhitectură hardware, sistem de operare, aplicații.

Vom studia în continuare această problemă, punând accent pe soluțiile de la nivelul SO-ului.



# Problema Secțiunii Critice

- Enunțul problemei secțiunii critice
- Cerințele de rezolvare
- Soluții pentru cazul a două procese
- Soluții pentru cazul a  $n > 2$  procese
- Soluții hardware: instrucțiuni hardware specializate
- Soluții concrete în SO: spinlock-uri și semafoare

# Problema Secțiunii Critice

Enunțul problemei secțiunii critice:

- Avem  $n$  procese  $P_i$ ,  $i=0, \dots, n-1$ , cu viteze de execuție necunoscute
- Fiecare proces are o zonă de cod, numită **secțiune critică (SC)**, în care efectuează diverse operații asupra unei resurse partajate
- Execuția secțiunilor critice de către procese trebuie să se producă **mutual exclusiv** în timp: la orice moment de timp cel mult un proces să se afle în SC proprie
- Oprirea oricărui proces are loc numai în afara SC a acestuia
- Fiecare proces trebuie să ceară permisiunea să intre în propria lui SC. Secvența de cod ce implementează această cerere este numită **secțiunea de intrare**
- Secțiunea critică poate fi urmată de o **secțiune de ieșire**
- Restul codului din fiecare proces este **secțiunea rest**

# Problema Secțiunii Critice

Soluția problemei SC trebuie să satisfacă următoarele 3 cerințe:

- **Excluderea mutuală:** dacă procesul  $P_i$  execută instrucțiuni în SC proprie, atunci nici un alt proces nu poate executa în propria SC.
- **Progresul:** dacă nici un proces nu execută în SC proprie, și unele dintre procese sunt în “conflict” la intrare (i.e., doresc simultan să intre în propriile lor SC), atunci selecția unuia dintre ele, singurul căruia i se va permite intrarea, nu poate fi amânată la infinit.  
(*Consecință:* numai acele procese care nu execută în secțiunile lor rest, vor participa la luarea deciziei care va fi următorul proces ce va intra în SC proprie.)
- **Așteptarea limitată:** trebuie să existe o limită a numărului de permisiuni acordate, între momentul când un proces a cerut accesul în propria SC și momentul când va primi permisiunea de intrare, altor procese de a intra în SC proprii.

# Problema Secțiunii Critice

Opțiuni de implementare a excluderii mutuale:

- **Dezactivarea întreruperilor**

(posibilă doar pentru sistemele uniprocessor și eficientă doar pentru secvențe critice scurte)

- **Soluții de așteptare ocupată**

- execută o buclă while de așteptare dacă SC este ocupată
- folosirea unor instrucțiuni atomice specializate

- **Sincronizare blocantă**

- sleep (inserarea în coada de așteptare) cât timp SC este ocupată

Primitivele de sincronizare (diverse abstracții, precum lacătele pe fișiere), ce sunt puse la dispoziție de un sistem, pot fi implementate prin aceste tehnici sau prin combinații ale unora dintre aceste tehnici.

# Problema Secțiunii Critice

Un șablon tipic de proces:

**repeat**

*secțiunea de intrare*

*secțiunea critică*

*secțiunea de ieșire*

*secțiunea rest*

**forever**

# Problema Secțiunii Critice

Soluții pentru cazul a  $n=2$  procese:

- Două procese  $P_0$  și  $P_1$  ce execută fiecare într-o buclă infinită câte un program ce constă din două secțiuni: secțiunea critică  $c_0$ , respectiv  $c_1$ , și restul programului – secțiunea necritică  $r_0$ , respectiv  $r_1$ . Execuția secțiunilor  $c_0$  și  $c_1$  nu trebuie să se suprapună în timp.
- Când se va prezenta procesul  $P_i$ , se va utiliza  $P_j$  pentru a ne referi la celălalt proces ( $j=1-i$ ).

# Problema Secțiunii Critice

## Soluția 1 (o primă idee de rezolvare)

- Cele două procese vor partaja o variabilă întreagă comună **turn** inițializată cu 0 (sau cu 1).
- Dacă **turn = i**, atunci procesul  $P_i$  este cel căruia  $i$  se permite să-și execute SC.

# Problema Secțiunii Critice

## Soluția 1

Procesul  $P_i$  :

**repeat**

**while**  $\text{turn} \neq i$  **do** *nothing*;

*secțiunea critică*

$\text{turn} := j$ ;

*secțiunea rest*

**forever**



Așteptare ocupată



# Problema Secțiunii Critice

## Soluția 1 este incompletă !

- Motivul: ea satisface condiția de excludere mutuală și cea de așteptare limitată, în schimb cerința de progres nu-i îndeplinită

E.g., dacă `turn=0`, după ce procesul  $P_1$  a trecut ultimul prin SC, și  $P_1$  vrea să intre din nou în SC proprie, în tot acest timp procesul  $P_0$  fiind “ocupat” la infinit în secțiunea sa rest (fie execută o buclă infinită, fie își termină execuția), atunci selecția procesului  $P_1$  este amânată la infinit.

Cu alte cuvinte, se poate manifesta fenomenul de *starvation* în cazul acestei soluții.

```
P0 :  
repeat  
  while turn ≠ 0 do nothing;  
  secțiunea critică  
  turn := 1;  
  secțiunea rest  
forever
```

```
P1 :  
repeat  
  while turn ≠ 1 do nothing;  
  secțiunea critică  
  turn := 0;  
  secțiunea rest  
forever
```

# Problema Secțiunii Critice

## Soluția 2 (o a doua idee de rezolvare)

- Variabila comună **turn** este înlocuită cu un tablou comun **flag[]**, inițializat cu valoarea false:  
**flag[0] = false** , **flag[1] = false** .
- Prin **flag[i] = true** se indică faptul că procesul  $P_i$  dorește să-și execute SC.

# Problema Secțiunii Critice

## Soluția 2

Procesul  $P_i$  :

**repeat**

flag[i] := true;

**while** flag[j] **do** *nothing*;

*secțiunea critică*

flag[i] := false;

*secțiunea rest*

**forever**

Așteptare ocupată



# Problema Secțiunii Critice

Și soluția 2 **este incompletă !**


- Motivul: condițiile de excludere mutuală și de așteptare limitată sunt satisfăcute, în schimb cerința de progres nu este îndeplinită

Soluția este dependentă de *timing*-ul execuției proceselor: în situația de “conflict”, dacă cele 2 atribuiri se execută înaintea celor 2 while-uri, atunci selecția unuia dintre P<sub>0</sub> și P<sub>1</sub> este amânată la infinit, deoarece fiecare îl așteaptă pe celălalt să-și reseteze flagul.

Cu alte cuvinte, în cazul acestei soluții se pot manifesta simultan fenomenul de *deadlock* și cel de *starvation*.

```
P0:  
repeat  
    flag[0] := true;  
    while flag[1] do nothing;  
    secțiunea critică  
    flag[0] := false;  
    secțiunea rest  
forever
```

```
P1:  
repeat  
    flag[1] := true;  
    while flag[0] do nothing;  
    secțiunea critică  
    flag[1] := false;  
    secțiunea rest  
forever
```



# Problema Secțiunii Critice

## Soluția 3 (completă !) (Peterson '81)

- Este o combinație a soluțiilor 1 și 2.
- Procesele partajează variabila `turn` și tabloul `flag[]`.
- Inițializări: `flag[0] = flag[1] = false`, `turn = 0` (sau `1`).
- Pentru a intra în SC, procesul  $P_i$  setează `flag[i] = true` și apoi îi dă voie celui alt proces  $P_j$  să intre în propria sa SC, dacă dorește acest lucru (`turn = j`).
- Dacă ambele procese doresc să intre în același timp, valoarea lui `turn` va decide căruia dintre cele două procese îi este permis să intre primul în SC proprie.
- *Notă*: istoric, prima soluție completă a fost cea datorată lui Dekker ('65)

# Problema Secțiunii Critice

## Solutia 3

**P<sub>i</sub>: repeat**

flag[i] := true;

turn := j;

**while** (flag[j] **and** turn=j)

**do** *nothing*;

*secțiunea critică*

flag[i] := false;

*secțiunea rest*

**forever**

← Așteptare ocupată

# Problema Secțiunii Critice

Soluția 3 este corectă și completă !

Justificare:

- Motivul #1: condiția de excludere mutuală este satisfăcută. Într-adevăr:

Fiecare proces  $P_i$  poate intra în SC proprie doar dacă fie  $\text{flag}[j]=\text{false}$ , fie  $\text{turn}=i$ .

Presupunând că ambele procese ar putea să execute în SC proprie în același timp, atunci am avea  $\text{flag}[0]=\text{flag}[1]=\text{true}$ .

Dar aceasta ar însemna că  $P_0$  și  $P_1$  nu ar fi putut să-și execute cu succes instrucțiunile *while* proprii în același timp.


# Problema Secțiunii Critice

Soluția 3 **este corectă și completă !** Justificare:

- Motivul #2: condițiile de progres și de așteptare limitată sunt satisfăcute. Într-adevăr:

```
P0:  
repeat  
  flag[0] := true;  
  turn := 1;  
  while (flag[1] and turn=1)  
    do nothing;  
  secțiunea critică  
  flag[0] := false;  
  secțiunea rest  
forever
```

```
P1:  
repeat  
  flag[1] := true;  
  turn := 0;  
  while (flag[0] and turn=0)  
    do nothing;  
  secțiunea critică  
  flag[1] := false;  
  secțiunea rest  
forever
```





# Problema Secțiunii Critice

Soluția 3 **este corectă și completă !** Justificare:

– Motivul #2 (cont.):

Un proces  $P_i$  poate fi împiedicat să intre în SC (i.e., să **progreseze**) doar dacă este blocat în bucla *while*. În acest caz, dacă  $P_j$  nu-i gata să intre în SC, atunci **flag[j]=false** și  $P_i$  poate intra în SC proprie.

Dacă însă  $P_j$  a setat deja **flag[j]=true** și-și execută și el bucla *while*, atunci avem **turn=i** sau **turn=j**. Dacă **turn=i**, atunci  $P_i$  va intra în SC.

Dacă însă **turn=j**, atunci  $P_j$  va intra în SC, iar după ce  $P_j$  va ieși din SC, își va reseta **flag[j]** la **false**, permițându-i lui  $P_i$  să intre în SC.

Aceasta deoarece, chiar dacă apoi  $P_j$  setează **flag[j]** la **true** pentru a intra din nou în SC, el va seta de asemenea și **turn=i**.

Astfel,  $P_i$  va termina bucla *while* și va intra în SC (**progres**) după cel mult o intrare a lui  $P_j$  în SC (**așteptare limitată**).

# Problema Secțiunii Critice

Prima soluție corectă și completă: Dekker '65

- Inițial propusă de Dekker într-un context diferit, a fost aplicată de Dijkstra pentru rezolvarea problemei SC.
- Dekker a adus ideea unui proces favorit și a permiterii accesului în SC a oricărui dintre procese în cazul când o cerere de acces este necontestată de celălalt
- În schimb, dacă este un conflict (i.e. ambele procese vor să intre simultan în SC-urile proprii), unul dintre procese este favorizat, iar prioritatea se inversează după execuția cu succes a SC.
- Procesele partajează variabila `turn` și tabloul `flag[]`.
- Inițializări: `flag[0] = flag[1] = false`, `turn = 0` (sau `1`).

# Problema Secțiunii Critice

**$P_i$  : repeat**

    flag[i] := true;

**while** (flag[j]) **do**

**if** (turn=j) **then begin**

            flag[i] := false;

**while** (turn=j) **do** *nothing*;

            flag[i] := true;

**end**

*secțiunea critică*

    turn := j;

    flag[i] := false;

*secțiunea rest*

**forever**

↖ Așteptare ocupată

Temă: încercați să justificați corectitudinea acestui algoritm.

# Problema Secțiunii Critice

Soluții pentru procese multiple:

- Trebuie dezvoltați algoritmi diferiți de cei anteriori, pentru a rezolva problema secțiunii critice pentru cazul a  $n$  procese ( $n > 2$ )

# Problema Secțiunii Critice

## Algoritmul brutarului (Lamport '74)

- La intrarea în magazin, fiecare client primește un număr de ordine.
- Clientul cu cel mai mic număr este servit primul.
- Dacă  $P_i$  și  $P_j$  primesc același număr și dacă  $i < j$ , atunci  $P_i$  este servit primul.
- Algoritmul este deterministic  
(numele proceselor sunt unice și total ordonate).
- *Notă*: istoric, prima soluție completă pentru cazul  $n > 2$  a fost cea datorată lui Eisenberg & McGuire ('72)

# Problema Secțiunii Critice

## Algoritmul brutarului (bakery alg.)

- Structurile de date comune (partajate de cele  $n$  procese) sunt tablourile `choosing[]` și `number[]`.
- Inițializări: `choosing[i] = false` , `number[i] = 0`.
- Notății:  
 $(a,b) < (c,d)$  dacă  $a < c$  sau ( $a = c$  și  $b < d$ ) ;  
 $\max(a_0, \dots, a_{n-1})$  = un număr  $k$  astfel încât  
 $k \geq a_i$  , pentru  $i=0, \dots, n-1$ .

# Problema Secțiunii Critice

$P_i$  ( $0 \leq i \leq n-1$ ):

**repeat**

(Alg. brutarului)

choosing[i] := true;

number[i] := max(number[0], ..., number[n-1]) + 1;

choosing[i] := false;

**for** j := 0 **to** n-1 **do begin**

**while** choosing[j] **do** *nothing*;

**while** number[j] ≠ 0 **and**

        (number[j], j) < (number[i], i) **do** *nothing*;

**end**

*secțiunea critică*

number[i] := 0;

*secțiunea rest*

**forever**

# Problema Secțiunii Critice

Prima soluție corectă: Eisenberg & McGuire '72

(Este practic o generalizare a soluției lui Peterson)

- Structurile de date comune (partajate de cele  $n$  procese) sunt variabila  $turn$  și tabloul  $flag[]$ , cu
$$turn \in \{0, 1, \dots, n-1\},$$
$$flag[i] \in \{idle, want-in, in-cs\}, 0 \leq i \leq n-1.$$
- Inițializări:
$$flag[i] = idle, 0 \leq i \leq n-1, \text{ și}$$
$$turn = 0 \text{ (sau orice valoare între } 0 \text{ și } n-1).$$



# Problema Secțiunii Critice

```
Pi: var k:0..n;
    repeat
        repeat
            flag[i] := want-in;
            k := turn;
            while k ≠ i do
                if flag[k]=idle then k := (k + 1) mod n;
                else k := turn;

            flag[i] := in-cs;
            k := 0;
            while (k<n) and (k=i or flag[k]≠in-cs) do k := k + 1;
        until (k≥n) and (turn=i or flag[turn]=idle);
        turn := i;
        secțiunea critică
        k := (turn + 1) mod n;
        while (flag[k]=idle) do k := (k + 1) mod n;
        turn := k;
        flag[i] := idle;
        secțiunea rest
    forever
```

Temă: încercați să justificați corectitudinea algoritmului.

Bibliografie: [https://en.wikipedia.org/wiki/Eisenberg\\_%26\\_McGuire\\_algorithm](https://en.wikipedia.org/wiki/Eisenberg_%26_McGuire_algorithm)

# Problema Secțiunii Critice

## Soluții hardware (1)

- Instrucțiunea atomică specializată **Test-and-Set**

Semantica ei (în pseudo-cod):

```
function Test-and-Set (var target: boolean): boolean;  
begin  
    Test-and-Set := target; // valoarea returnată  
    target := true;  
end;
```

Exemplu: majoritatea arhitecturilor de calcul multiprocesor posedă o instrucțiune de tipul

TSL *reg, adr*

e.g. pentru  $\mu$ P Intel x86 avem instrucțiunea

lock bts *op1, op2*

# Problema Secțiunii Critice

## Soluții hardware (1)

- $n$  procese; variabila comună **lock**, inițializată cu false.

```
Pi: repeat  
    while Test-and-Set (lock)  
        do nothing;  
    secțiunea critică  
    lock := false;  
    secțiunea rest  
forever
```

# Problema Secțiunii Critice

## Soluții hardware (2)

- Instrucțiunea atomică specializată **Swap**

Semantica ei (în pseudo-cod):

```
procedure Swap (var a, b: boolean);  
var temp: boolean;  
begin  
    temp := a;  
    a := b;  
    b := temp;  
end;
```

Exemplu: pentru  $\mu$ P Intel x86 avem instrucțiunea  
`xchg op1, op2`  
unde operanzii sunt doi regiștri, sau un registru și o adresă de memorie.

# Problema Secțiunii Critice

## Soluții hardware (2')

- Varianta condițională a instrucțiunii atomice **Swap** este instrucțiunea atomică **Compare-and-Swap**

Semantica ei (în pseudo-cod):

```
function Compare-and-Swap (var val:int; expected, newval:int);  
var temp: int;  
begin  
    temp := val;  
    if val = expected then val := newval;  
    Compare-and-Swap := temp; // valoarea returnată  
end;
```

Exemplu: pentru  $\mu$ P Intel x86 avem instrucțiunea

```
lock cmpxchg op1, op2
```

unde operanzii sunt doi regiștri, sau un registru și o adresă de memorie.

# Problema Secțiunii Critice

## Soluții hardware (2)

- $n$  procese; variabila comună **lock**, inițializată cu false.

$P_i$ : **var** *key*:boolean; // variabila locală *key*

**repeat**

*key* := true;

**repeat**

Sau: *key* = Compare-and-Swap(*lock*, false, true);

Swap(*lock*, *key*);

**until** *key* = false;

*secțiunea critică*

*lock* := false;

*secțiunea rest*

**forever**

# Problema Secțiunii Critice

## Soluții hardware (3)

- *Notă*: soluțiile anterioare satisfac condițiile de excludere mutuală și de progres, dar nu îndeplinesc și cerința de așteptare limitată.
- O soluție **completă** folosind **Test-and-Set** (sau **Swap**) :
  - $n$  procese;
  - variabila comună **lock**, inițializată cu false, și vectorul comun **waiting**[0.. $n-1$ ] , inițializat cu false
  - limita de așteptare:  $n-1$

# Problema Secțiunii Critice

## Soluții hardware (3)

$P_i$  ( $0 \leq i \leq n-1$ ):

**var**  $j:0..n-1$ ;  $key:boolean$ ;

**repeat**

$waiting[i] := true$ ;

$key := true$ ;

**while**  $waiting[i]$  **and**  $key$  **do**  $key := \text{Test-and-Set}(lock)$ ;

$waiting[i] := false$ ;

*secțiunea critică*

$j := i+1 \bmod n$ ;

**while**  $j \neq i$  **and not**  $waiting[j]$  **do**  $j := j+1 \bmod n$ ;

**if**  $j = i$  **then**  $lock := false$ ;

**else**  $waiting[j] := false$ ;

*secțiunea rest*

**forever**

Sau : ... **do**  $\text{Swap}(key, lock)$  ;



Temă: încercați să justificați corectitudinea acestui algoritm.

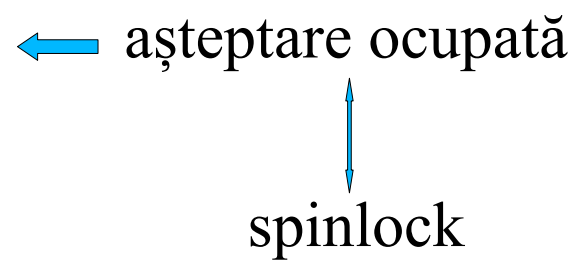


# Problema Secțiunii Critice

## Soluții concrete (1): Lacăte mutex (sau *spinlocks*)

- Cea mai simplă soluție software utilizată în nucleul SO pentru protecția unei SC: la intrare trebuie să dobândești lacătul, iar la ieșire să-l eliberezi, prin intermediul a două operații standard, atomice: operația `acquire()` și operația `release()`, lacătul având o valoare booleană *available*, ce arată dacă este disponibil sau nu.
- Semantica operației `acquire()` :  

```
while !available do nothing; ← așteptare ocupată  
available:=false;
```


- Semantica operației `release()` :  

```
available:=true;
```

spinlock
- Pentru a fi atomice, cele două operații se implementează folosind instrucțiunile atomice specializate (TSL sau Swap).

# Problema Secțiunii Critice

## Excluderea mutuală implementată cu *spinlocks*

- Problema SC cu  $n$  procese; variabila comună **lock** este un *spinlock* liber (i.e., inițializat cu true).

**P<sub>i</sub>: repeat**

```
    acquire(lock);  
    secțiunea critică  
    release(lock);  
    secțiunea rest
```

**forever**

- **Scenarii de utilizare:** în situațiile în care durata de execuție a secțiunii critice este scurtă (comparativ cu durata unui *context-switch*).

Ex.: inserția sau extracția unui proces din coada *ready* a planificatorului CPU.

# Problema Secțiunii Critice

## Soluții concrete (2): Semafoare

- Concept introdus de E.W. Dijkstra în '65
- Un **semafor**  $S$  este o variabilă întreagă care este accesată (exceptând operația de inițializare) numai prin intermediul a două operații standard, atomice:
  - operația  $P$  sau **wait()** (*proberen* = a testa), și
  - operația  $V$  sau **signal()** (*verhogen* = a incrementa).
- Semantica operației **wait(S)** :  
`while  $S \leq 0$  do nothing;  $S := S - 1$ ;`
- Semantica operației **signal(S)** :  `$S := S + 1$ ;`

# Problema Secțiunii Critice

## Semafoare – implementare la nivelul SO:

- Semafoarele pot suspenda și reporni procese/thread-uri, pentru a evita *așteptarea ocupată* (i.e. risipa de cicli CPU)
- Semaforul se definește ca un articol:

```
typedef struct {  
    int value;  
    struct thread *ListHead;  
} semaphore;
```

- Se consideră următoarele 2 operații:

`suspend()` : suspendă execuția thread-ului care-l apelează  
`resume(T)` : reia execuția unui thread T blocat anterior  
(printr-un apel `suspend()` )

# Problema Secțiunii Critice

## Semafoare – implementare la nivelul SO:

– Cele 2 operații atomice cu semafoare se definesc atunci astfel:

1) operația ***wait(S)*** :

```
S.value--;  
if (S.value < 0) {  
    add this thread to S.ListHead;  
    suspend();  
}
```

2) operația ***signal(S)*** :

```
S.value++;  
if (S.value <= 0) {  
    remove a thread T from S.ListHead;  
    resume(T);  
}
```

*Notă:* atomicitatea celor două secvențe de cod se realizează folosind *spinlock*-uri.

# Problema Secțiunii Critice

## Semafoare – implementare la nivelul SO:

- Mai multe detalii de implementare în S.O.-urile moderne: a se citi §6.6.2, pag. 274-276, din Silberschatz: “*Operating System Concepts*”, ediția 10 [OSC10]
- La nivelul aplicațiilor, semafoarele pot fi simulate prin diverse entități logice (e.g. fișiere, canale fifo, semnale, ș.a. )
- Biblioteca IPC (introdusă în UNIX System V) permite lucrul cu semafoare în aplicații (inclusiv în Linux)
- Despre spinlocks – a se vedea <https://wiki.osdev.org/Spinlock>

# Problema Secțiunii Critice

## Excluderea mutuală implementată cu semafoare

- Problema SC cu  $n$  procese; variabila comună **mutex** este un semafor binar (i.e, semafor inițializat cu 1).

**$P_i$ : repeat**

*wait(mutex) ;*

*secțiunea critică*

*signal(mutex) ;*

*secțiunea rest*

**forever**

- **Scenarii de utilizare:** în situațiile în care execuția secțiunii critice durează suficient de mult (comparativ cu durata unui *context-switch*).

# Problema Secțiunii Critice

## Semafoare – clasificare:

- După modul de utilizare, putem clasifica semafoarele în următoarele două tipuri:
  - **Semafoarele binare** (i.e., semafoare inițializate cu valoarea 1) **pot asigura excluderea mutuală** (e.g., pot soluționa problema secțiunii critice)
  - **Semafoarele generale** (i.e., semafoare inițializate cu valoarea  $n > 1$ ) **pot reprezenta o resursă cu instanțe multiple** (i.e., cu  $n$  instanțe) (e.g., pot soluționa problema producător-consumator)



# Problema Secțiunii Critice

## Semafoare – alte scenarii de utilizare

- Pot fi folosite pentru a rezolva diverse alte probleme de sincronizare între procese
- Exemplu: execută B în  $P_2$  numai după ce s-a executat A în  $P_1$
- Soluție: folosim un semafor *synch* inițializat cu 0, astfel:

$P_1$ :

```
.....  
    A  
signal(synch);  
.....
```

$P_2$ :

```
.....  
wait(synch);  
    B  
.....
```

# Problema Secțiunii Critice

## Semafoare – alte scenarii de utilizare (cont.)

Exemplul anterior se referea la situația: un proces trebuie să-l aștepte pe altul, într-un anumit punct din cod.

Întrebare: poate fi generalizat la mai multe procese, și cum anume?

Răspuns: în primul rând, generalizarea la mai multe procese nu este UNICĂ! Spre exemplu, cazurile extreme la generalizare ar fi: 1) N-1 procese care doresc să aștepte un anumit proces (altul decât cele N-1) ; respectiv 2) un proces vrea să aștepte simultan alte N-1 procese. Plus, toate cazurile intermediare posibile între cele 2 extreme.

Soluția cu acel semafor *synch* dată pe slide-ul anterior nu poate fi generalizată prea ușor la toate aceste cazuri generalizate de așteptare.

Spre exemplu, doar pentru cazul extrem 2) descris mai sus, o posibilă soluție cu semafoare ar putea fi: cele N-1 procese ce trebuie așteptate vor fi de forma  $P_i$  ( $1 \leq i \leq N-1$ ), iar procesul ce trebuie să le aștepte pe toate celelalte va fi de forma Q indicată mai jos ; folosim N-1 semafoare  $synch_1, \dots, synch_{N-1}$  inițializate cu 0, partajate de Q cu  $P_i$  :

$P_i$  :

Q :

.....

.....

$A_i$

$wait(synch_1); wait(synch_2); \dots ; wait(synch_{N-1});$

$signal(synch_i);$

B

.....

.....

# Problema Secțiunii Critice

## Semafoare – alte scenarii de utilizare (cont.)

Referitor la exemplele anterioare de așteptare, mai există și alte mecanisme de sincronizare, pe lângă semafoare, care sunt mai adecvate pentru anumite forme de sincronizare între un număr  $N > 2$  de procese.

Un astfel de mecanism de sincronizare este cel de "barieră" (*barrier*, în engleză) care rezolvă problema așteptării **simultane** pentru un număr  $N$  (oarecare) de procese, la (câte) un punct din codul fiecăruia.

*Modul de funcționare al barierei:* fiecare proces "ajunge la barieră" în ritmul său, și le așteaptă acolo pe toate celelalte. Abia după ce au ajuns toate, "se ridică bariera" **simultan** pentru toate procesele, care-și vor continua astfel execuția fiecare cu codul propriu ce urmează după apelul barierei.

Pentru mai multe detalii despre mecanismul de "barieră", consultați următoarea referință:

[https://en.wikipedia.org/wiki/Barrier\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Barrier_(computer_science))

# Interblocajul și înfometarea

- **Interblocajul** (*deadlock*)
  - O situație în care două sau mai multe procese așteaptă pe termen nelimitat producerea câte unui eveniment (e.g., execuția unei operații signal pe un semafor), eveniment ce ar putea fi cauzat doar de către unul dintre celelalte procese ce așteaptă.
  - Aceste procese se spune că sunt **interblocate**.
- **Blocajul nelimitat** sau **înfometarea** (*starvation*)
  - O situație în care un(ele) proces(e) așteaptă nelimitat (e.g., la un semafor: procesul ar putea sta suspendat în coada de așteptare asociată acelui semafor pe termen nelimitat).

# Interblocajul și înfometarea

- **Interblocajul** (*deadlock*)

- Exemplu: două procese folosesc 2 semafoare binare S și Q (i.e. inițializate cu 1) în ordinea de mai jos:

$P_1$ :	$P_2$ :
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
.....	.....
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

Se observă că este posibil să apară interblocaj. În ce situație apare? (Specificați un scenariu de execuție care duce la interblocaj...)

Soluție (pentru a nu fi posibil interblocajul): ambele programe ar trebui să execute operațiile `wait()` pe cele două semafoare în aceeași ordine !

# Probleme clasice de sincronizare

- Problema Producător-Consumator  
(Producer-Consumer or Sender-Receiver problem)
- Problema Cititori și Scriitori  
(Readers and Writers problem)
- Problema Cina Filozofilor  
(Dining-Philosophers problem)
- Problema Bărbierului Adormit  
(Sleeping Barber problem)

(va urma)

- **Bibliografie obligatorie**

capitolele despre *sincronizarea proceselor* din

- Silberschatz : “*Operating System Concepts*”

(cap.6 din [OSC10])

sau

- Tanenbaum : “*Modern Operating Systems*”

(a treia parte a cap.2 din [MOS4])

- Introducere
  - Problema secțiunii critice
    - Enunțul problemei și cerințele de rezolvare
    - Soluții pentru cazul a două procese
    - Soluții pentru cazul a  $n > 2$  procese
    - Soluții hardware
    - Soluții concrete: *spinlock*-uri și semafoare
  - Interblocajul și înfometarea
- (va urma)
- Probleme clasice de sincronizare
  - Monitoare (și alte abordări ale problemei SC)

Întrebări ?