Sisteme de Operare

Sincronizarea proceselor
partea a II-a

Cristian Vidrașcu

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

Cuprins

Am discutat despre:

- Introducere
- Problema secțiunii critice
- Interblocajul și înfometarea

Continuăm cu:

- Probleme clasice de sincronizare
  - Problema Producător-Consumator
  - Problema Cititori și Scriitori
  - Problema Cina Filozofilor
  - Problema Bărbierului Adormit
- Monitoare (și alte abordări ale problemei SC)

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)

Probleme clasice de sincronizare (1)

- Problema Producător-Consumator
  - Enunțată de Dijkstra în '65
  - Este o problemă reprezentativă pentru ilustrarea conceptului de *procese cooperante*:

    Un proces, cu rol de *producător,* produce informații ce sunt consumate de un alt proces, cu rol de *consumator*.

Producător-Consumator

- Enunțul problemei:
  - Un proces produce niște date (e.g. informații, mesaje, ...) și le depune într-o zonă tampon (*buffer*), de unde le ia un al doilea proces și le consumă.
  - Accesul la zona tampon se face în mod exclusiv.
  - Zona tampon are capacitate nelimitată.
  - Consumatorul trebuie să aștepte când bufferul este gol.
  - Altă variantă a acestei probleme este cu zona tampon de capacitate finită. În acest caz și producătorul trebuie să aștepte și, anume, atunci când zona tampon este plină.
  - *Notă*: problema aceasta se poate formula și cu mai mulți producători și consumatori (ce utilizează un singur buffer).

Producător-Consumator

- Soluția problemei (varianta cu buffer nelimitat):
  - un semafor binar mutex – care va controla secțiunea critică (i.e., accesul exclusiv la zona tampon)
  - un semafor (binar) delay – care va bloca consumatorul dacă zona tampon este goală
  - o variabilă întreagă n – care va număra elementele din zona tampon
  - Inițializări: n = 0, mutex = 1, delay = 0

Producător-Consumator

Procesul Producător:

```
repeat
   producere_element();
   wait(mutex);
   adaugare_element_in_buffer();
   n:=n+1;
   if n=1 then signal(delay);
   signal(mutex);
forever
```

Producător-Consumator

Procesul Consumator:

```
wait(delay);
repeat
   wait(mutex);
   extragere_element_din_buffer();
   n:=n-1; nlocal:=n;
   signal(mutex);
   consumare_element();
   if nlocal=0 then wait(delay);
forever
```

Producător-Consumator

Producător:

```
repeat
   producere_element();
   wait(mutex);
   adaugare_element_in_buffer();
   n:=n+1;
   if n=1 then signal(delay);
   signal(mutex);
forever
```

Consumator:

```
wait(delay);
repeat
   wait(mutex);
   extragere_element_din_buffer();
   n:=n-1; nlocal:=n;
   signal(mutex);
   consumare_element();
   if nlocal=0 then wait(delay);
forever
```

- Instrucțiunea if din producător poate fi scoasă în afara SC (i.e. asemănător ca în consumator, folosind o variabilă nlocal), în schimb if-ul din consumator nu poate fi pus în SC (i.e. asemănător ca în producător) deoarece ar putea apare interblocaj.

Producător-Consumator

- Soluția problemei (varianta cu buffer limitat):
  - Se poate adapta soluția de la versiunea cu buffer nelimitat, adăugând un semafor binar delayPro – care va bloca producătorul dacă zona tampon este plină

    (Observație: if-ul din producător trebuie scos în afara SC, asemănător ca în consumator, folosind o variabilă nlocal, pentru ca altfel poate să apară interblocaj)

Temă: încercați să scrieți soluția pe baza acestei idei.

Producător-Consumator

- Altă soluție pentru varianta cu buffer limitat:
  - un semafor binar mutex – care va controla secțiunea critică (i.e., accesul exclusiv la zona tampon)
  - un semafor general empty – care va număra locațiile goale din zona tampon
  - un semafor general full – care va număra locațiile pline din zona tampon
  - Inițializări: mutex = 1, full = 0, empty = n

Producător-Consumator

Procesul Producător:

```
repeat
   producere_element();

   wait(empty);
   wait(mutex);
   adaugare_element_in_buffer();
   signal(mutex);
   signal(full);
forever
```

Producător-Consumator

Procesul Consumator:

```
repeat
   wait(full);
   wait(mutex);
   extragere_element_din_buffer();
   signal(mutex);
   signal(empty);

   consumare_element();
forever
```

Producător-Consumator

Producător:

```
repeat
   producere_element();
   wait(empty);
   wait(mutex);
   adaugare_element_in_buffer();
   signal(mutex);
   signal(full);
forever
```

Consumator:

```
repeat
   wait(full);
   wait(mutex);
   extragere_element_din_buffer();
   signal(mutex);
   signal(empty);
   consumare_element();
forever
```

- Ordinea celor două apeluri wait(), atât în producător, cât și în consumator, este esențială; prin schimbarea ordinii ar putea apare interblocaj.

Probleme clasice de sincronizare (2)

- Problema Cititori și Scriitori (CREW)
  - Enunțată de Courtois, Heymans și Parnas în '71
  - Este o problemă reprezentativă pentru accesul la o bază de date

    (e.g. un sistem de rezervare a biletelor de avion, cu multe procese competitoare dorind să citească și să scrie în baza de date)
  - CREW: este acceptabil să avem mai multe procese care să citească baza de date în același timp, dar, dacă un proces actualizează (i.e. scrie) baza de date, nici un alt proces nu trebuie să aibă acces la ea, nici măcar în citire.

Cititori și Scriitori

- Enunțul problemei:
  - Un obiect (i.e. o resursă, e.g. un fișier, o zonă de memorie, etc.) trebuie să fie partajat de mai multe procese concurente
  - Unele dintre aceste procese ar putea dori doar să citească conținutul obiectului partajat : Cititorii
  - Alte procese ar putea dori însă să actualizeze (citire și scriere) conținutul obiectului partajat : Scriitorii
  - Se cere ca scriitorii să aibă acces exclusiv la obiectul partajat, în schimb cititorii să îl poată accesa în mod concurent (non-exclusiv)

Cititori și Scriitori

– Versiunea #1: nici un cititor nu va fi ținut în așteptare decât dacă un scriitor a obținut deja permisiunea de acces la obiectul partajat

- Cu alte cuvinte, nici un cititor nu trebuie să aștepte alți cititori să termine de citit, doar din cauză că un scriitor așteaptă deja permisiunea de acces
- La acces simultan, cititorii sunt mai prioritari decât scriitorii
- *Notă*: unele procese (scriitorii în acest caz) pot deveni înfometate

Cititori și Scriitori

– Versiunea #2: o dată ce un scriitor este gata de scriere, el va executa acea scriere cât mai curând posibil

- Cu alte cuvinte, dacă un scriitor așteaptă deja permisiunea de acces, nici un nou cititor (i.e. care cere permisiunea de acces după scriitor) nu trebuie să primească permisiunea de acces
- La acces simultan, scriitorii sunt mai prioritari decât cititorii
- *Notă*: unele procese (cititorii în acest caz) pot deveni înfometate. De asemenea, această soluție permite un grad de concurență mai scăzut și, deci, o performanță mai slabă decât prima soluție.

Cititori și Scriitori

– Soluția pentru versiunea #1:

Variabile partajate:

- Procesele cititori partajează două semafoare binare mutex și wrt, precum și o variabilă întreagă readcount ; semaforul wrt este partajat și de către procesele scriitori.
- Inițializări: mutex = wrt = 1 , readcount = 0 .
- readcount ține evidența numărului de procese cititori ce sunt în cursul citirii obiectului partajat
- mutex este folosit pentru a asigura excluderea mutuală când este actualizată variabila readcount
- wrt este utilizat pentru a asigura excluderea mutuală pentru scriitori la accesul obiectului partajat

Cititori și Scriitori

– Structura proceselor scriitori:

```
repeat
   ...
   wait(wrt);
    scriere_obiect();
   signal(wrt);
   ...
forever
```

– Dacă un scriitor este în SC și n cititori așteaptă, atunci un cititor este în așteptare la wrt, iar ceilalți n-1 cititori așteaptă la mutex.

Cititori și Scriitori

– Structura proceselor cititori:

```
repeat
   ...
   wait(mutex);
    readcount:=readcount+1;
    if readcount=1 then wait(wrt);
   signal(mutex);
   citire_obiect();
   wait(mutex);
    readcount:=readcount-1;
    if readcount=0 then signal(wrt);
   signal(mutex);
   ...
forever
```

Cititori și Scriitori

- Soluția versiunii #1

Proces scriitor:

```
repeat
   ...
   wait(wrt);
   scriere_obiect();
   signal(wrt);
   ...
forever
```

Proces cititor:

```
repeat
   ...
   wait(mutex);
   readcount:=readcount+1;
   if readcount=1 then wait(wrt);
   signal(mutex);
   citire_obiect();
   wait(mutex);
   readcount:=readcount-1;
   if readcount=0 then signal(wrt);
   signal(mutex);
   ...
forever
```

Temă: proiectați o soluție pentru vers. #2 (Atenție: nu este simetrică!)

Cititori și Scriitori

– Soluția pentru versiunea #2

Remarcă: întâi încercați să rezolvați singuri această problemă!

Variabile partajate:

- Variabila întreagă readcount (inițializată cu 0) va reprezenta numărul de cititori activi
- Variabila întreagă writecount (inițializată cu 0) va reprezenta numărul de scriitori activi
- Două semafoare binare mutex-rdc și mutex-wrc (inițializate cu 1) vor fi folosite pentru a proteja accesul la variabilele partajate readcount și respectiv writecount
- Semaforul binar wrt (inițializat cu 1), folosit la fel ca în soluția versiunii #1
- Încă două semafoare binare mutex_RW și rd (inițializate cu 1) folosite astfel încât cititorii să aștepte scriitorii activi

Cititori și Scriitori

Procesele cititori:

```
repeat
   wait(mutex-RW);
      wait(rd);
         wait(mutex-rdc);
            readcount := readcount + 1;
            if readcount = 1 then wait(wrt);
         signal(mutex-rdc);
      signal(rd);
   signal(mutex-RW);
   citeste_resursa();
   wait(mutex-rdc);
      readcount := readcount - 1;
      if readcount = 0 then signal(wrt);
   signal(mutex-rdc);
forever
```

Cititori și Scriitori

Procesele scriitori:

```
repeat
   wait(mutex-RW);
      wait(mutex-wrc);
         writecount := writecount + 1;
         if writecount = 1 then wait(rd);
      signal(mutex-wrc);
   signal(mutex-RW);

   wait(wrt);
      scrie_resursa();
   signal(wrt);

   wait(mutex-wrc);
      writecount := writecount - 1;
      if writecount = 0 then signal(rd);
   signal(mutex-wrc);
forever
```

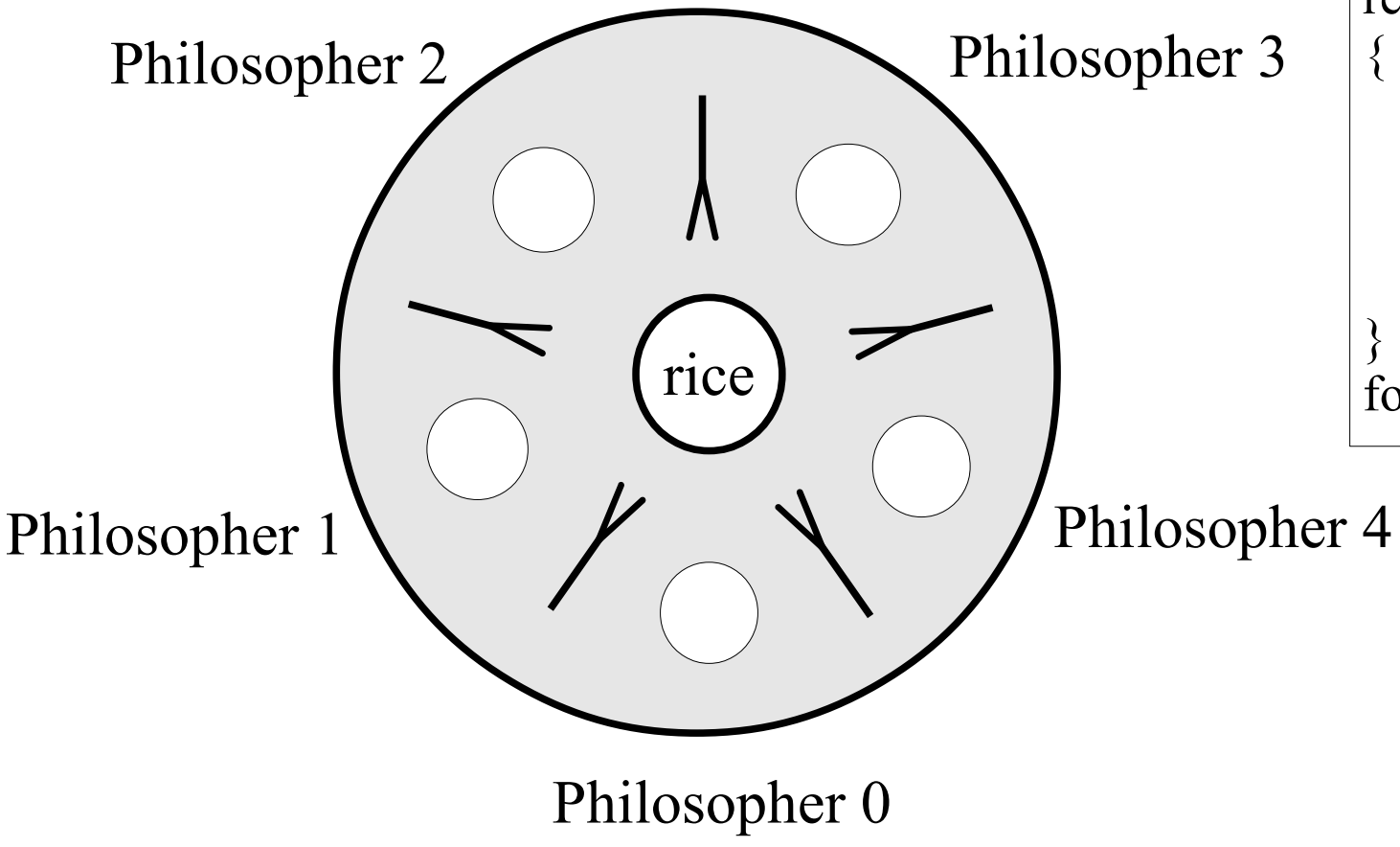
Probleme clasice de sincronizare (3)

- Problema Cina Filozofilor Chinezi
  - Enunțată de Dijkstra în '65
  - Este o problemă reprezentativă pentru nevoia de a aloca un număr limitat de resurse (nepartajabile) la mai multe procese, ce concurează pentru acces exclusiv la aceste resurse, alocare care să se facă fără să apară fenomenul de interblocaj sau cel de înfometare.

Cina Filozofilor

- Enunțul problemei:
  - 5 filozofi chinezi își petrec viețile gândind și mâncând
  - Ei partajează o masă circulară comună, înconjurată de 5 scaune, fiecare aparținând unuia dintre ei
  - În centrul mesei este un platou cu (foarte mult!) orez și mai există pe masă 5 farfurii și (doar!) 5 bețișoare pentru mâncat
  - Când un filozof gândește, el nu interacționează cu colegii lui
  - Din când în când, unui filozof i se face foame și vrea să mănânce, încercând să ridice cele 2 betișoare din dreptul lui
  - Fiecare filozof poate ridica un singur bețișor o dată
  - După ce termină de mancat, pune bețișoarele înapoi pe masă și începe să filozofeze din nou

Cina Filozofilor



Philosopher life:

```
repeat
{
   pick up the forks;
   eat awhile;
   put down the forks;
   think awhile;
}
forever
```

Cina Filozofilor

- Soluția problemei:
  - Fiecare bețișor (i.e. resursă nepartajabilă) este reprezentat printr-un semafor binar
  - Un filozof încearcă să ridice bețișorul executând operația wait() pe acel semafor, respectiv lasă bețișorul jos pe masă executând operația signal()
  - Datele partajate de procesele filozofi sunt: chopstick[0..4] of semaphore;
  - Inițializări: chopstick[i] = 1, i=0,...,4

Cina Filozofilor

– Structura procesului filozof al i-lea (i=0,...,4):

```
repeat
   wait(chopstick[i]);
   wait(chopstick[(i+1) mod 5]);
   mananca();
   signal(chopstick[i]);
   signal(chopstick[(i+1) mod 5]);
   gandeste();
forever
```

Cina Filozofilor

- Comentarii
  - Soluția este incompletă ! Motivul:
    - Este posibil să se creeze un interblocaj
    - Unii filozofi pot deveni înfometați

- Temă:
  - Proiectați o altă soluție pentru problema cinei filozofilor, care să nu permită apariția interblocajului sau a înfometării

Probleme clasice de sincronizare (4)

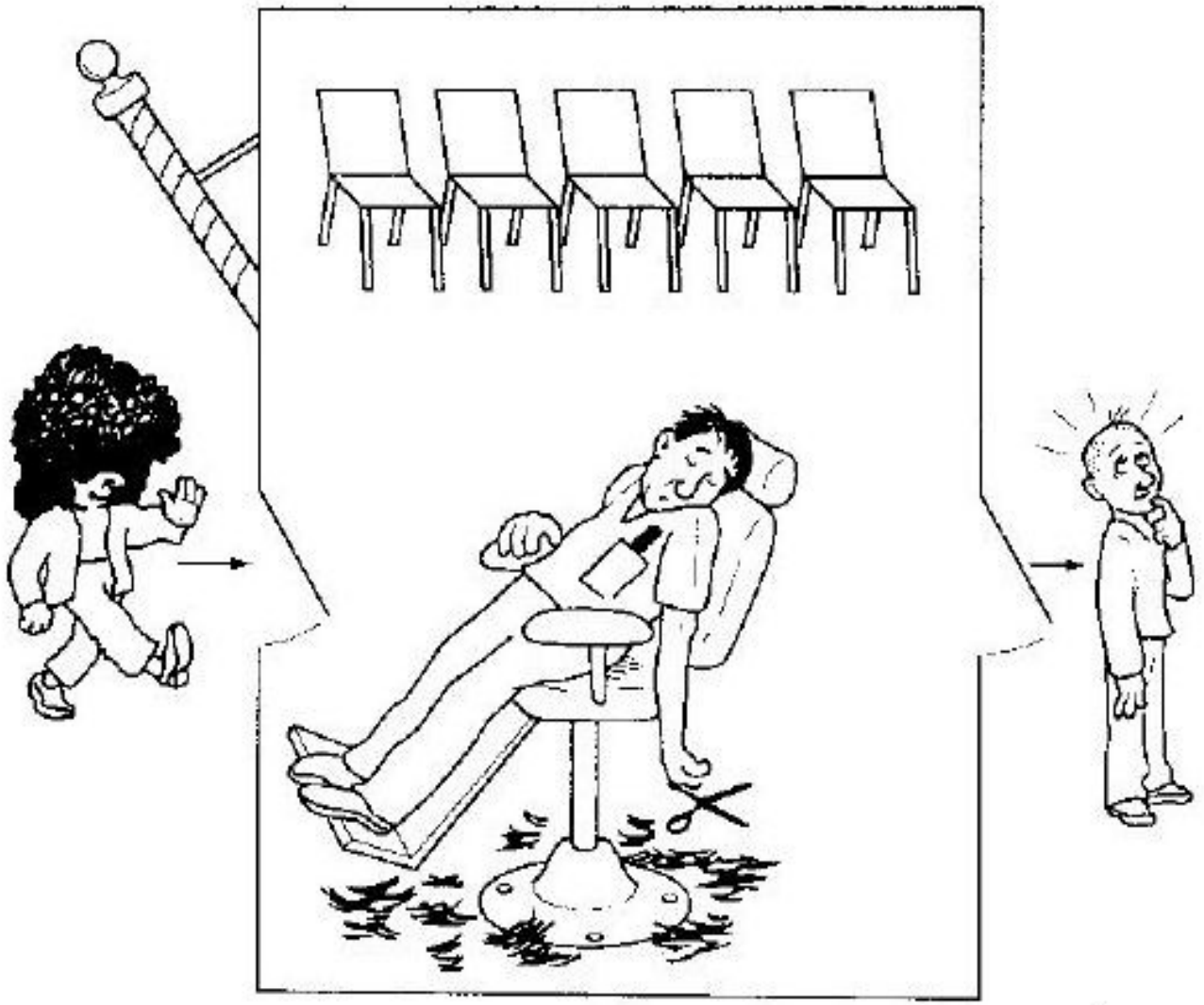
- Problema Bărbierului Adormit
  - Enunțată de Dijkstra în '65
  - Este o problemă reprezentativă pentru situații diverse de așteptare la coadă

    (e.g. un secretariat cu un sistem computerizat de preluare a apelurilor telefonice și punere în așteptare, cu o capacitate limitată de păstrare a apelurilor în așteptare)

Bărbierul Adormit

- Enunțul problemei:
  - O frizerie cu un bărbier, un scaun de lucru și *n* scaune pentru clienții ce așteaptă să le vină rândul la tuns
  - Dacă nu are clienți, bărbierul stă în scaunul de lucru și doarme (i.e., se odihnește)
  - Când vine un prim client, el trebuie să-l trezească pe bărbier
  - Dacă mai vin și alți clienți cât timp bărbierul tunde un client, aceștia fie iau loc pe scaunele de așteptare (dacă mai sunt locuri libere), fie pleacă netunși (în caz contrar)
  - Problema constă în a programa bărbierul și clienții de așa manieră încât să nu apară blocaje datorită fenomenelor de tip "*race conditions*"
  - *Notă*: problema are și o variantă cu mai mulți bărbieri

Bărbierul Adormit



Bărbierul Adormit

- Soluția problemei:
  - Procesele bărbier și clienți partajează trei semafoare: mutex, customers și barber, și o variabilă întreagă: freeseatscount
  - mutex este folosit pentru a asigura excluderea mutuală când este accesată variabila freeseatscount
  - freeseatscount ține evidența numărului de scaune libere (i.e. neocupate de clienți ce stau în așteptare)
  - semaforul general customers ține evidența numărului de clienți în așteptare pe scaune (exclusiv cel ce este tuns)
  - semaforul binar barber este utilizat pentru a indica dacă bărbierul este ocupat (1) sau liber (0)
  - Inițializări: customers = barber = 0 , freeseatscount = n , mutex = 1

Bărbierul Adormit

– Structura procesului bărbier:

```
repeat
   wait(customers);
   wait(mutex);
   freeseatscount++;
   signal(barber);
   signal(mutex);
   tunde_client();
forever
```

Bărbierul Adormit

– Structura proceselor clienți:

```
wait(mutex);
if(freeseatscount > 0)
{
   freeseatscount--;
   signal(customers);
   signal(mutex);
   wait(barber);
   este_tuns_de_barbier();
}
else
   signal(mutex);
```

Bărbierul Adormit

Procesul bărbier:

```
repeat
   wait(customers);
   wait(mutex);
   freeseatscount++;
   signal(barber);
   signal(mutex);
   tunde_client();
forever
```

Procesele clienți:

```
wait(mutex);
if(freeseatscount > 0)
{
   freeseatscount--;
   signal(customers);
   signal(mutex);
   wait(barber);
   este_tuns_de_barbier();
}
else
    signal(mutex);
```

Monitoare (și alte abordări)

- Construcții de sincronizare în limbaje de nivel înalt

  La folosirea semafoarelor pot apare erori de sincronizare datorită unei ordini incorecte a apelurilor wait și signal; este suficient ca un singur proces să nu coopereze corect (fie datorită unei erori de programare, fie în mod intenționat), pentru a "strica" sincronizarea tuturor proceselor cooperante.

  De aceea, s-au introdus o serie de construcții de sincronizare în unele limbaje de programare de nivel înalt, care să "ascundă" programatorului detaliile legate de apelurile wait și signal (tratarea corectă a acestora cade în sarcina compilatorului).

Monitoare (și alte abordări)

- Monitorul

– Concept dezvoltat de B.Hansen '73 & C.A.R.Hoare '74
– Este o construcție de sincronizare de nivel înalt (i.e. implementată în unele limbaje de programare, cum ar fi *Concurrent Pascal*, C#, Java, ș.a.), introdusă pentru a ușura sarcina programatorilor: se elimină erorile de programare ce pot apare, la folosirea semafoarelor, datorită unei ordini incorecte a apelurilor wait și signal

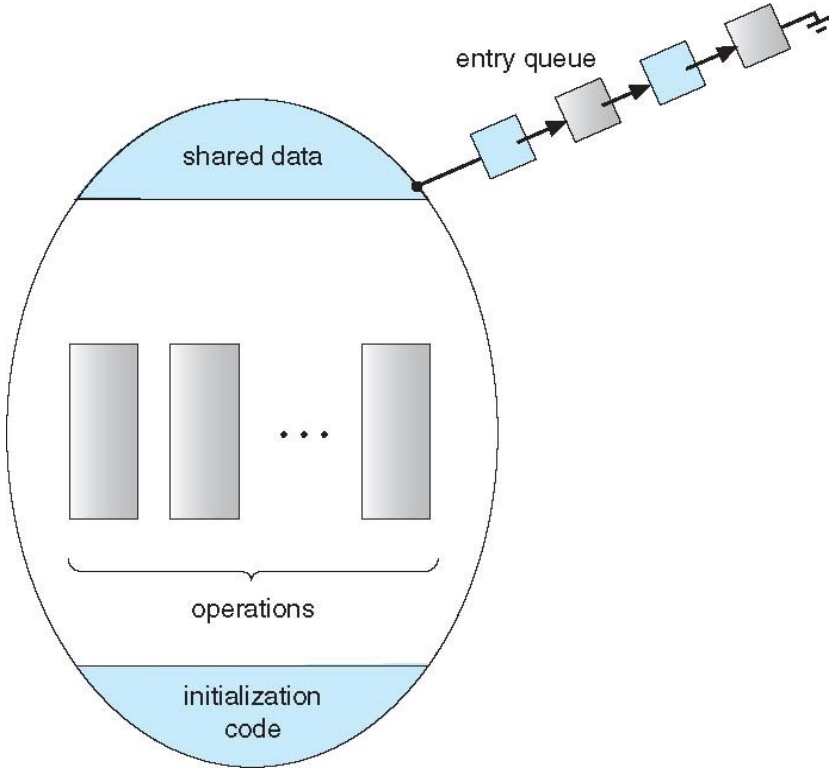
- Regiunea critică condițională – o altă construcție de acest gen, implementată în *Concurrent Pascal* (limbaj proiectat de B.Hansen):

var *var-shared* : shared *type* ;

region *var-shared* when *condiție* do *cod* ;

Monitoare (și alte abordări)

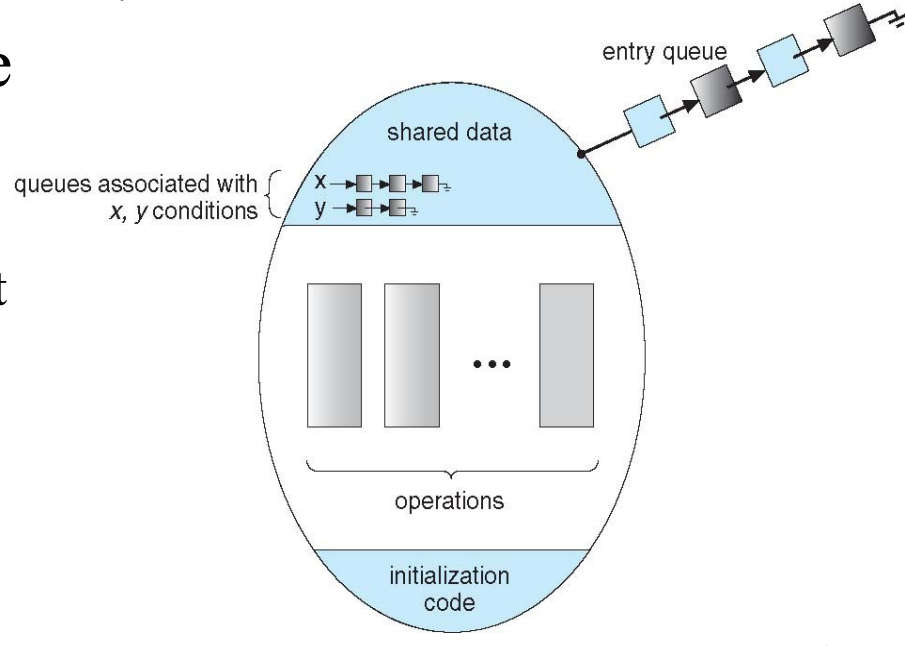
- Monitorul este un tip de dată abstract care:
  - Încapsulează date partajate împreună cu operații asupra lor ce se efectuează cu accesul mutual exclusiv la obiect (practic, un monitor = o "clasă" cu un *zăvor* asociat)
  - Variabilele interne sunt accesibile numai prin operațiile monitorului
  - În orice moment, cel mult un singur proces/thread poate fi *activ* în interiorul monitorului (i.e. poate executa vreo operație)



Monitoare (și alte abordări)

- Monitorul poate avea asociate *variabile condiții*:
  - Doar excluderea mutuală nu este suficientă uneori:
    În timp ce un proces execută o operație în monitor, ar putea avea nevoie să aștepte până când o anumită condiție c (depinzând de variabilele monitorului) devine *true*. O soluție de genul:
    `while not(c) do nothing;` // așteptare ocupată
    nu funcționează în acest caz, pentru că procesul menține monitorul ocupat
  - O variabilă condiție x este o soluție bazată pe sincronizare blocantă

    Practic, x are asociate doar două operații:
    • x.wait() : procesul ce o invocă va fi suspendat
    • x.signal() : când un proces P o invocă, se va relua activitatea unui proces Q din coada de așteptare a celor suspendate prin x.wait() și doar unul dintre cele două, P sau Q, va fi activ în monitor (Care? Sunt mai multe opțiuni în acest sens: signal-and-wait vs. signal-and-continue)



Monitoare (și alte abordări)

- În limbajul C++ : noi extensii introduse în standardul C++11:
  – suport pentru programarea *multithreaded*
  – fire de execuție: clasa `std::thread` cu operatii specifice acestora (a se vedea detalii aici)
  – lacăte mutex: clasa `std::mutex` și alte variante
  – variabile condiții: clasa `std::condition_variable`și alte variante
  – operații atomice: clasa `std::atomic_flag` cu metodele `test_and_set()` și `clear()`

- În limbajul Java:
  – metode syncronized într-o clasă (a se vedea detalii aici)
  – *intrinsic lock* (sau *monitor lock*) asociat fiecărui obiect
  – obiecte *immutable* (i.e. după creare, sunt *read-only*)
  – high-level APIs in the `java.util.concurrent` packages

Monitoare (și alte abordări)

- Memorie tranzacțională:
  - o tranzacție cu memoria este o secvență de citiri/scrieri în memorie, executată în manieră atomică
  - STM: implementată la nivel software, în compilator
  - HTM: implementată la nivel hardware, în cache-urile CPU-ului
- Extensia OpenMP (pentru limbajele C/C++, Fortran, ș.a.):
  - un set de directive de compilare și un API pt. programare paralelă
  - paralelism *multithreaded*: `#pragma omp parallel { cod; }`
  - secțiune critică: `#pragma omp critical { cod; }`
- Limbaje de programare funcționale:
  - variabilele sunt *immutable* (o dată create și inițializate, nu mai pot fi modificate)
  - ca urmare, nu pot apare situații de *race conditions* sau *deadlocks*
  - ex. de limbaje funcționale populare: Erlang, Scala, ș.a.

Bibliografie

- **Bibliografie obligatorie**

  capitolele despre *sincronizarea proceselor* din

  – Silberschatz : "*Operating System Concepts*"
  (cap.6&7 din [OSC10])

  sau

  – Tanenbaum : "*Modern Operating Systems*"
  (a treia și a cincea parte a cap.2 din [MOS4])

Sumar

- Probleme clasice de sincronizare
  - Problema Producător-Consumator
  - Problema Cititori și Scriitori
  - Problema Cina Filozofilor
  - Problema Bărbierului Adormit
- Monitoare (și alte abordări ale problemei SC)

Întrebări ?