2.3. Mecanisme de control a concurenței, comunicare şi sincronizare

În practică, ***aplicațiile concurente***, atât la ***nivel de sistem de operare***, cât şi la ***nivel de program*** s-au proiectat şi implementat folosindu-se diverse ***mecanisme de control al concurenței.***

Se vor defini, la **nivel conceptual**, câteva astfel de mecanisme şi se vor analiza relațiile dintre aceste mecanisme.

2.3.1. Semafoare

Conceptul de ***semafor*** a fost introdus de Dijkstra, pentru a facilita ***sincronizarea proceselor***, prin ***protejarea secțiunilor critice*** şi ***asigurarea accesului* exclusiv** la ***resursele*** pe care procesele le accesează.

Secțiunile critice sunt secvențe de instrucțiuni care pot genera ***race – condition*** şi a căror execuție de către mai multe procese trebuie controlată.

Formal, un semafor se poate defini ca o pereche ***(v(s),c(s))*** unde:

-***v(s)*** este *valorea semaforului*- un nr. întreg a cărui valoare poate varia pe durata execuției diferitelor procese. Inițializabil cu 1 sau 0 (TRUE sau FAISE);

-***c(s)*** o coadă de aşteptare la semafor - conține *referințe la procesele* care aşteaptă la semaforul ***s***. Inițial coada este vidă, iar disciplina cozii depinde de sistemul de operare (LIFO, FIFO, priorități, etc.).

Fiecărei secțiuni critice trebuie să i se aloce un semafor.

Semafoare

Gestiunea semafoarelor: prin ***2 operații indivizibile***

P(s) –este apelată de către procese care doresc să acceseze o regiune critică pt a obține acces.

Efect: - dacă v(s) este 1(TRUE), execuția lui P(s) are ca efect accesul procesului apelant la secțiunea critică şi trecerea semaforului pe zero.

- dacă v(s) este 0 (False), procesul ce doreşte execuția lui P(s) aşteaptă

V(s)

Efect : modificarea valorii semaforului şi trecerea sa din 0 (FALSE) în 1 (TRUE). Ac funcție se apelează la sfârşitul secțiunii critice şi semnifică eliberarea acesteia pt. alte procese.

IMPLEMENTAREA SEMAFOARELOR: El. Hardware ale CPU

El. A sist. de operare

Succesiune instrucț.:

P(s)

regiune critică

V(s)

Rest. procesului

Operațiile P şi V pot fi exprimate în pseudo cod

```
P(s)

{
v(s)=v(s)-1;
        if (v(s)<0){
                stare_proces=blocat;
                procesul este trecut in coada c(s)
        }
}
```

Semnificatie: Procesul curent poate executa zona critica doar daca valoarea semaforului este 1 sau mai mare. Altfel este blocat de catre sistemul de operare. Deoarece valoarea semaforului se decrementeaza inainte de a se sti daca regiunea critica poate fi executata, valoarea contorului arata cate procese asteapta sa execute regiunea critica respectiva.

Operațiile P şi V pot fi exprimate în pseudo cod

Dacă procesul execută regiunea critică înseamnă că la intrarea sa în proces avem: v(s)=0 sau mai mare.

```
V(s)

{
v(s)=v(s)+1;
        if (v(s)<=0){
                selecteaza un proces din coada c(s)
                stare_proces_selectat=activ;
        }
}
```

Daca pe durata sectiunii critice semaforul nu este modificat, imediat dupa incrementare el va avea valoarea 1 sau mai mare ceea ce elibereaza regiunea critica si continua cu instructiunile succesive.

În UNIX

- Noțiunea de semafor a fost generalizată prin posibilitatea de a executa mai multe operații asupra unui semafor inclusiv incrementare decrementare cu valori diferite de 1.

-Se lucreaza cu multimi de semafoare care sunt descrise ***de structuri de date*** ce contin o alta ***structura*** reprezentind permisiunile proceselor la semafoare, un pointer la primul semafor din set, momentul de timp la care a avut loc ultima operatie asupra setului de semafoare.

- Accesul unui proces la un set de semafoare este controlat; doar procesele cu anumite caracteristici fiind capabile sa modifice semafoarele.

Semaforul este un instrument fundamental al concurenței

În UNIX noțiunea de semafor a fost generalizată prin posibilitatea de a executa mai multe operații asupra unui semafor inclusiv incrementare decrementare cu valori diferite de 1.
Notatie adoptata pentru definirea si atribuirea valorii initiale a unui semafor:

```
var semaphore s = x;
- - - - - - - - - - -
```

primeşte valoarea inițială $v_0(s)=x$

```
var semaphore s = n;
- - - - - - - - - - - - - - -
P(s);
Trenul trece între A si B pe una din cele n linii;
V(s);
```

2.3.2. Variabile mutex

Variabila mutex (**mut**ual **ex**clusion):

- un instrument util pentru protejarea unor resurse partajate, accesate concurent de mai multe thread-uri.
- sunt folosite, de asemenea, pentru implementarea ***secțiunilor critice*** şi a ***monitoarelor*** (notiuni pe care le vom defini în secțiunile imediat următoare).
- are două stări posibile: ***blocată*** (este proprietatea unui thread) sau ***neblocată*** (nu este proprietatea nici unui thread)**.** Un task care vrea să obțină o variabilă mutex blocată de alt task, trebuie să aştepte până când primul o eliberează.

Operațiile posibile asupra variabilelor mutex:

- inițializarea (static sau dinamic);
- blocarea (pentru obținerea accesului la resursa protejată);
- deblocarea (pentru eliberarea resursei protejate);
- distrugerea variabilei mutex.

Variabile mutex

Din punct de ***vedere conceptual***, o variabilă mutex este echivalentă cu un ***semafor s***, care poate lua două valori: 1 pentru starea neblocată şi 0 pentru starea blocată. (Un astfel de semafor se va numi ***semafor binar***).

Operațiile asupra unei variabile mutex m se definesc, cu ajutorul semafoarelor, astfel:

- ***Inițializare****:* se defineşte un semafor ***m*** astfel încât $v_0(m) = 1$.
- ***Blocare****:* (după o eventuală deblocare de către alt thread): ***P(m)***.
- ***Deblocare****:* ***V(m)***.
- ***Distrugere****:* distrugerea semaforului ***m*****.**

2.3.3. Variabile condiționale

Definiție, caracteristici

- **Sunt obiecte de sincronizare şi comunicare între task-urile care aşteaptă satisfacerea unei condiții şi task-ul care o realizează.**
- Au asociate: ***un predicat*** - dă condiția ce trebuie să se realizeze şi care de obicei implică date partajate;
  o variabilă mutex - asigură faptul că verificarea condiției şi intrarea în aşteptare, sau verificarea condiției şi semnalarea îndeplinirii ei să fie executate ca şi operații atomice.

Operațiile posibile asupra variabilelor condiționale sunt:

- ***Inițializare****:* care poate fi statică sau dinamică;
- ***Aşteptare*** *(wait):* threadul este pus în aşteptare până când i se va semnala din exterior îndeplinirea condiției;
- ***Semnalare*** *(notify, broadcast, notifyall):* threadul curent anunță unul dintre thread-urile ce aşteaptă îndeplinirea condiției, sau toate thread-urile ce aşteaptă îndeplinirea condiției;
- ***Distrugere****;*

2.3.4. Conceptul de monitor

- **Un *monitor*** este o construcție similară cu **un tip de date abstract**. Scopul său principal este de a încapsula ***variabilele partajate*** şi ***operațiile*** asupra acestora. Astfel, toate secțiunile critice sunt concentrate în această structură la care, la un moment dat, are acces unul singur dintre task-uri.

  Secțiunile critice sunt extrase din task-urile obişnuite şi devin proceduri sau funcții ale monitorului.

- In modelul lui Hoare (1974), un monitor poate fi descris ca un obiect care conține:
  1. datele partajate;
  2. procedurile care accesează aceste date;
  3. o metodă de inițializare a monitorului;

Conceptul de monitor

OBS

- Fiecare grup de proceduri este controlat de un monitor;

- În momentul rulării programului multi-thread, monitorul permite unui singur thread să execute o procedură controlată de el. In această situație, vom spune că threadul a ocupat monitorul;

- Dacă alte thread-uri invocă monitorul în timp ce acesta este ocupat, ele sunt suspendate până când procedura monitor apelată de thread-ul respectiv îşi încheie activitatea, ceea ce coincide cu eliberarea monitorului de către thread.

Conceptul de monitor

Implementări ale conceptului de monitor

Pascal Concurent

Mesa

Java - o variantă de monitor cu ajutorul modificatorului *synchronised.*

Monitorul este un concept mai uşor de manevrat decât semaforul.

Din punct de vedere conceptual, un monitor poate fi descris simplu folosind un singur semafor binar, cu valoarea inițială 1. Fiecare procedură a monitorului începe cu ***P(s)*** şi se încheie cu ***V(s)***:

```
var semaphore s = 1;
- - - - - - - - - -
Pentru fiecare procedura a monitorului:
P(s)
codul corpului procedurii
V(s)
```

2.3.5. Secțiune şi resursă critică; excludere mutuală

- Prin ***excludere mutuală*** a două procese se exprimă faptul că în fiecare moment numai ***unul*** dintre procese poate să fie ***activ***.
- Prin ***resursă critică*** este indicată ***o resursă*** *care poate fi ocupată şi folosită la un moment dat numai de către* ***un singur proces.***
- Prin ***secțiune critică*** se indică o porțiune de cod care nu poate fi executată la un moment dat decât de un singur task şi care secțiune, odată inițiată, execuția ei trebuie să fie terminată fără a fi întreruptă de un alt task.

  OBS

  - Secțiunile critice trebuie să fie, de regulă, cât mai scurte posibil deoarece toate celelalte task-uri sunt întârziate până la terminarea execuției unei asemenea secțiuni critice.

  -Problema secțiunii critice prezintă un interes deosebit, atât din punct de vedere teoretic, cât şi din punct de vedere practic. Majoritatea "subtilităților" programării concurente se leagă, într-un fel sau altul de ea.

Secțiune şi resursă critică; excludere mutuală

O secțiune critică bine definită trebuie să îndeplinească următoarele condiții:

1. la un moment dat, numai un singur proces este în secțiunea critică; orice alt proces solicită accesul la ea, îl va primi numai după ce procesul ocupant a terminat de executat instrucțiunile secțiunii critice;
2. vitezele relative ale proceselor care accesează secțiunea critică sunt necunoscute;
3. oprirea oricărui proces trebuie să aibă loc numai în afara secțiunii critice;
4. nici un proces nu va aştepta indefinit pentru a intra în secțiunea critică.

Există diverse modele de implementare a secțiunii critice. Implicit, toate acestea rezolvă excluderea mutuală şi resursele critice.

O secțiune critică trebuie implementată, la fel ca şi la monitor, folosind un *semafor binar* ***s***, cu valoarea inițială 1.

```
var semaphore s = 1;
- - - - - - - - - -
P(s)
codul sectiunii critice
V(s)
```

2.3.6. Regiuni critice condiționale

Prin ***regiune critică condițională*** se înțelege ***o secțiune critică*** plus o ***resursă critică*** plus ***verificarea unei condiții*** înainte de execuția efectivă.

- Fiecărei regiuni critice i se asociază o ***resursă*** constând din toate variabilele care trebuie protejate în regiune. Declararea ei se face astfel:

```
resource r :: v1 , v2 , ..., vn
```

unde ***r*** este numele resursei, iar ***v1 , ..., vn*** sunt numele variabilelor de protejat.

- O ***regiune critică condițională*** se specifică astfel:

```
region r [ when B ] do S
```

- unde ***r*** este numele unei resurse declarate ca mai sus, ***B* este o expresie booleană**, iar ***S*** este secvența de instrucțiuni corespunzătoare regiunii critice.

- dacă este prezentă opțiunea **when**, atunci *S* este executată numai dacă *B* este adevărată.

Descrierea prin semafoare a unei regiuni critice condiționale este dată în figura 2.10. Pentru descriere sunt necesare două semafoare şi un număr întreg.

```
var  semaphore sir = 0;
     exclus = 1;  int nr = 0;
- - - - - - - - - - - - - - - -
P(exclus);
nr:=nr+1;
while not B do begin
    V(exclus);  P(sir);  P(exclus);
end;
nr:=nr-1;
S;
for i:=1 to nr do V(sir);
V(exclus);
```

Figura 2.10 **Codul regiunii critice condiționale**

Semaforul *sir* reține in coada lui toate procesele care solicită acces la regiune.
Semaforul *exclus* asigură accesul exclusiv la anumite secțiuni ale implementării.
Intregul *nr* reține câte procese au cerut acces la regiune, **la un moment dat.**

2.3.7. Conceptul de întâlnire (rendez-vous)

Conceptul de ***întâlnire*** *(rendez-vous)* este introdus de limbajul ***Ada*** pentru a facilita comunicarea între două task-uri.

a) Procesul B este gata să transmită informațiile, dar procesul A nu le-a cerut încă. In acest caz, procesul B rămâne în aşteptare până când procesul A i le cere.

b) Procesul B este gata să transmită informațiile cerute, iar procesul A cere aceste date. In acest caz, se realizează un *rendez-vous*, cele două procese lucrează sincron până când îşi termină schimbul, după care fiecare îşi continuă activitatea independent.

c) Procesul A a lansat o cerere, dar procesul B nu este în măsură să-i furnizeze informațiile solicitate. In acest caz, A rămâne în aşteptare până la întâlnirea cu B.

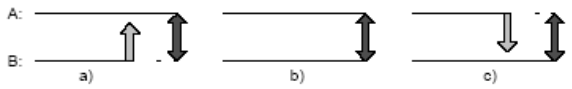

Figura 2.11 **Scenarii ale unui rendez-vouz**

Mecanismul se poate descrie folosind un semafor, ***s.*** Procesul A execută o operație ***P(s)*** înainte de punctul de întâlnire, iar procesul B execută o operație ***V(s)*** înainte de punctul de întâlnire.

2.4 Implementări ale mecanismului de excludere reciprocă

Modalitățile practice de implementare a mecanismului de excludere reciprocă operează prin:

- inhibarea întreruperilor,
- excluderea reciprocă prin programare,
- instrucțiunea de interschimbare,
- semafoare binare,
- regiuni critice simple,
- monitoare.

2.4.1 Inhibarea întreruperilor

- *Este o **soluție hardware**. Când un program ajunge într-o secțiune definită critică, se generează o întrerupere care, prin hardware, inhibă celelalte întreruperi (prin mascare), pe durata execuției secțiunii critice.*
- *Este o metodă simplă şi eficientă, însă presupune o inhibare neselectivă a tuturor întreruperilor ce poate duce la întârzieri semnificative în executarea celorlalte task-urilor.*

2.4.2 Excluderea reciprocă prin programare

Este o metodă ***software*** bazată pe ipoteza că accesele individuale la locațiile de memorie sunt indivizibile.

Soluții de implementare software a excluderii reciproce între două task-uri ciclice, fiecare având câte o secțiune critică:

1 - prin utilizarea unei ***variabile comune speciale*** pentru protejarea secțiunilor critice, variabilă denumită **"*poartă*"**, care poate avea două stări (valori) **"*deschis*"** şi, respectiv, **"*închis*".** Dacă valoarea variabilei este "închis" înseamnă că unul dintre task-uri se găseşte în secțiunea ei critică;

-**"deschis"** - înseamnă că task-ul poate executa secțiunea sa critică;

-**simultan "deschis"** ambele task-uri inițiază execuția secțiunii critice proprii.

OBS: nu garantează ***excluderea reciprocă corectă*** între două task-uri concurente.

2 - prin utilizarea unei ***variabile comune speciale*** pentru a dirija cele două task-uri concurente în momentul în care ele încearcă să execute secțiunile lor critice. Astfel, o variabilă întreagă, numit **"*comutator*",**

valoarea 1 = task-ul 1 poate executa secțiunea sa critică

valoarea 2 = task-ul 2 poate executa secțiunea sa critică

se garantează ***excluderea reciprocă*** a celor două task-uri concurente

OBS: impune o execuție alternantă obligatorie a secțiunilor critice ale celor două task-uri (tot timpul în ordinea 1,2,1,2,...).

Excluderea reciprocă prin programare

3 - *P*revederea pentru fiecare task în parte a unei ***variabile proprii locale*** care poate avea doar valorile **"*interior*"** şi respectiv **"*exterior*".**

-**"interior"** = task-ul respectiv doreşte să execute secțiunea sa critică;

-"exterior" = task-ul respectiv este în afara secțiunii sale critice;

- fiecare dintre task-urile concurente poate ***examina*** valoarea variabilei corespunzătoare task-ului concurent înainte de a trece la execuția secțiunii critice.

-nu există nici o variabilă manipulată în comun;

OBS - asignarea simultană a valorii ***"interior"*** variabilelor proprii = buclă infinită

4 - task-ul care detectează faptul că ambele task-uri concurente încearcă să treacă la execuția secțiunilor critice proprii, ***va modifica valoarea variabilei locale de control*** = atunci bucla infinită va fi evitată.

OBS –Situație critică dacă cele două task-uri ***derulează exact în acelaşi ritm*** (doi abonați telefonici se apelează reciproc în acelaşi moment de timp).

Excluderea reciprocă prin programare

5 - Combinarea ultimelor două propuneri.

Variabilă proprie, care indică dorința de a trece la execuția secțiunii critice iar în cazul în care ambele task-uri doresc simultan acest lucru.

Se utilizează o ***variabilă întreagă auxiliară*** pentru rezolvarea acestui conflict.

Fiecare task acționează doar asupra variabilei celuilalt task concurent doar dacă variabila proprie are valoarea "interior".

El va trece la executarea secțiunii critice proprii numai dacă task-ul concurent este în afara secțiunii sale critice.

Variabila întreagă "comutator" este folosită pentru a rezolva conflictele între task-urile concurente permițând task-ului, al cărui număr este în acel moment atribuit ca valoarea variabilei "comutator", să intre în execuția secțiunii sale critice.

La ieşirea din secțiunea critică, task-ul va modifica valoarea variabilei "comutator" şi celălalt task va putea, astfel, să treacă la execuția secțiunii sale critice.

Excluderea reciprocă prin programare

Condiții generale pentru realizarea corectă a excluderii reciproce a task-urilor concurente:

• La un moment dat, cel mult un singur task se poate găsi în secțiunea sa critică;

• Oprirea unui task în afara secțiunii sale critice nu trebuie să afecteze celelalte task-uri;

• Nu se poate face nici o ipoteză asupra vitezei relative de desfăşurare a task-urilor;

• Task-urile ce sunt pe cale de a trece la execuția secțiunilor lor critice, nu trebuie să se blocheze reciproc la infinit.

2.4.3 Instrucțiunea de interschimbare

Metodele anterioare ***de rezolvare prin software a problemei excluderii reciproce,*** presupun doar ***indivizibilitatea acceselor singulare la locațiile de memorie.***

????? S-ar putea executa 2 operatii anumite (interschimbarea continutului unei locatii de memorie cu cel al unui registru local) fara a fi intrerupte

2.4.3 Instrucțiunea de interschimbare

Soluția: Variabila globală "excludere" este inițializată cu o valoare egală cu unitatea.

Înainte de a se executa secțiunea critică, task-ul trebuie să obțină valoarea unității memorate de variabila "excludere" şi să stabilească valoarea zero acestei variabile, ambele acțiuni printr-o singură operație indivizibilă.

La sfârşitul acțiunii critice, task-ul va returna valoarea unitate variabilei "excludere".

Deoarece există o singură valoare unitate în sistem, cel mult un singur task îl poate obține.

Dacă un task nu este capabil să treacă la execuția secțiunii sale critice, el va trebui să stea într-o buclă de aşteptare (în mod continuu). Când valoarea ***unitate*** a variabilei "excludere" este eliberată, una singură dintre buclele de aşteptare va putea să o folosească.

OBS Metoda este acceptabilă dacă există o solicitare moderată a resurselor şi dacă secțiunile critice sunt scurte.

2.4.4 Semafoare binare

Caracteristică comună a metodelor anterioare:

- Dcă un task doreşte să execute o secțiune a sa critică şi nu i se permite acest lucru, atunci el pierde dreptul de a utiliza CPU.

- Este preferabil ca un asemenea task să fie trecut în starea de **"*blocat*"** urmând să fie adus în starea **"*gata de execuție*"** în momentul în care este posibil să treacă la execuția secțiunii sale critice.

Semafoare binare (***booleene sau variabile mutex***)

2.4.4 Semafoare binare

OPERARE:

1. Fiecărei secțiuni critice i se asociază un semafor (inițial având valoarea 1) iar task-ul care doreşte să execute secțiunea sa critică, va ***efectua o operație P***; task-ului i se va permite execuția secțiunii sale critice numai dacă valoarea actualizată (decrementată) a semaforului are valoarea zero. În caz contrar, task-ul este trecut în starea "blocat".

2. La ieşirea din secțiunea critică, task-ul va efectua ***o operație de tip V*** care va provoca incrementarea cu 1 a valorii variabilei semafor; dacă există alte task-uri blocate, unul dintre aceste task-uri va putea trece la executarea secțiunii sale critice. ***Deci, un task va rămâne blocat până când un alt task îi va indica posibilitatea să continue.***

Semafoare binare

Operațiile P şi V sunt indivizibile şi un singur task poate executa la un moment dat una dintre ele asupra aceluiaşi semafor.

Operațiile P şi V asupra semafoarelor se pot implementa şi prin hardware dar, de regulă, ele sunt implementate prin software, fiind protejate prin inhibarea întreruperilor.

Pentru a evita aşteptarea, prin buclare, se foloseşte de regulă ***un şir de aşteptare*** în care se inserează şi se extrag task-urile ce se executau în momentul inhibării întreruperilor.

Dezavantajul major al utilizării semafoarelor este legat de necesitatea unei programări foarte atente; dacă, de exemplu, din neatenție se scrie o operație primitivă P în loc de V, atunci task-urile se vor bloca reciproc la infinit.

2.4.5 Regiuni critice simple

Pentru a uşura programarea corectă a semafoarelor este posibil să se utilizeze şi o construcție de limbaj specifică limbajelor de programare de nivel înalt, cum este, de exemplu, ***regiunea critică simplă propusă de Hoare***:

```
with R do S
```

Permite ca instrucțiunile critice *S* să opereze asupra variabilei comune *R*. ***Compilatorul respectiv va traduce această construcție în instrucțiuni de program***.

Această construcție evită necesitatea de a încadra cu operații *P()* şi *V()* toate secțiunile critice şi elimină necesitatea verificării lor de la începutul şi sfârşitul secțiunilor critice.

Compilatorului verifică totodată faptul că ***variabilele comune sunt utilizate doar în interiorul regiunii critice***, actualizarea valorilor variabilelor comune fiind astfel protejată.

2.4.6 Monitoare

Dacă un ***task*** doreşte să ***execute secțiunea sa critică*** va face apel la ***procedura corespunzătoare a monitorului***;

Condiții de lucru:

- unul singur dintre task-uri se admite la un moment dat să se bucure de serviciile monitorului;
- toate celelalte apeluri la monitor aştepaptă eliberarea monitorului;
- într-un monitor asupra variabilelor partajate nu se pot executa decât operații "***cunoscute***" (***declarate o dată cu definirea monitorului***) pentru care se asigură în mod automat condițiile de excludere mutuală;
- nu există posibilitatea accesului din exterior la datele locale monitorului decât prin intermediul operațiilor;
- din acest motiv nu pot să apară erori de sincronizare;

Monitoare

În programarea cu monitoare se consideră că un program conține două tipuri de module:

- procese active
- procese pasive.

Toate variabilele partajate sunt ***variabile locale monitoarelor***.

Interacțiunea dintre procese are loc numai prin intermediul monitoarelor.

Un monitor operează asupra a trei tipuri de variabile:

- **variabile *permanente*** - sunt de fapt ***variabilele partajate***. Aceste variabile îşi păstrează valoarea între apelurile operațiilor asigurate de monitor. Inițializarea acestor variabile se face la crearea monitorului.
- **variabile *locale - sunt variabile utilizate pentru realizarea operațiilor. Au acelaşi tip de proprietăți ca al variabilelor locale în proceduri.***

- ***parametrii de apel*** - sunt parametrii operațiilor realizate de către monitor.

Monitoare

Trecerea unui proces printr-un monitor trebuie să dureze cât mai puțin.

Ce se întâmpla însă dacă un proces care începe execuția unei operații dintr-un monitor descoperă că trebuie să se blocheze în aşteptarea unui eveniment extern ?

Abordări posibile

-Se poate interzice apariția unor astfel de situații;

-Dacă un proces se blochează (este în aşteptarea unui eveniment) în timp ce se află în execuția unei operații dintr-un monitor atunci un alt proces poate să fie lăsat sa utilizeze monitorul.

Implicații: ***să existe o "evidență" a proceselor care aşteaptă apariția unui eveniment.***

Pentru a asigura ***semnalizarea blocării în aşteptarea unui eveniment*** se utilizează ***variabile condiție***.

Monitoare-variabile condiție

Variabila condiție=variabilă partajată, asupra căreia se pot executa două tipuri de operații primitive (atomice):

- ***wait*** *(****delay****) : produce blocarea procesului care o invocă;*

- ***signal*** *(****resume****): anunță posibilitatea deblocării unuia dintre procesele care a executat* ***operația wait*** *pentru variabila respectivă;*

OBS!!!- Atunci când un proces este blocat ca urmare a execuției unei operații *wait* în corpul unei operații dintr-un monitor, alte procese pot să utilizeze monitorul.

-Operațiile wait şi signal par să "semene" cu operațiile P şi V.

- Deosebire importantă. Operația signal nu are nici un efect dacă nu exista un proces care aşteaptă (ceea ce nu este cazul cu operația V care incrementează valoarea semaforului).

- Operația wait blochează întotdeauna procesul care o execută.

2.5 Sincronizarea explicită

Se consideră situațiile în care ***task-urile concurente doresc să coopereze între ele fiind astfel, într-un fel, interesate de desfăşurarea celorlalte task-uri***

Task-urile continuă să se concureze pentru a obține dreptul de a intra în execuția secțiunii lor critice.

Odată obținut acest drept, acțiunea task-ului în timpul execuției secțiunii critice poate conduce la realizarea unei condiții care, anterior, determinase suspendarea (sau întârzierea) execuției unui alt task.

REZULTĂ:

-necesitatea ***existenței unei metode care să permită unui task să indice (să semnaleze) că un anumit eveniment a avut loc sau să aştepte până când are loc un anumit eveniment.***

- asigurarea unor forme de cooperare între task-uri, spre avantajul lor reciproc; ***SINCRONIZARE. Două programe se consideră sincronizate nu numai dacă sunt lansate în execuție concomitent ci şi dacă se pot stabili relații reciproce temporale predictibile între anumite momente ale desfăşurării lor.***

Sincronizarea explicită

Sensul general al sincronizării este ***cel de coordonare în timp, de corelare***, presupunând o relație reciprocă între task-uri, care au un caracter temporal şi stabil.

Noțiunea de sincronizare s-a extins şi mai mult, incluzând şi ***forma de sincronizare cu timpul absolut sau cel real*** (înțelegând prin aceasta că un anumit task este lansat în execuție în fiecare zi la o oră fixă sau ciclic, de exemplu, din oră în oră).

Sincronizarea trebuie să permită ***activarea*** şi respectiv, ***inhibarea*** desfăşurării unor programe (task-uri) atât prin cereri inițiate de alte task-uri cât şi prin comenzi ale utilizatorilor, introduse de la terminale.

Momentele de timp în care se inițiază aceste acțiuni sunt momentele de sincronizare efectivă.

Rezultă

Sunt necesare ***funcții referitoare la "evenimente",*** în sensul general al cuvântului, şi anume ***"aşteptarea"*** până la producerea evenimentului specificat şi ***"anunțarea"*** faptului că acesta s-a produs.

funcții mai complexe: aşteptarea primului eveniment dintr-o mulțime precizată de evenimente; aşteptarea şi tratarea selectivă a mai multor evenimente într-o ordine eventual diferită de cea a apariției lor;

Sincronizarea explicită

Metodele şi mecanismele folosite pentru realizarea sincronizării programelor sau task-urilor concurente se deosebesc sub mai multe aspecte:

- ***natura sincronizării***: sincronizare între programe sau task-uri sau sincronizare cu timpul;

- ***momentul sincronizării***: cu ***începutul*** unui program, cu ***sfârşitul*** unui program sau cu un"***punct***" (moment) oarecare din interiorul programului;

- ***implementarea sincronizării***: prin facilităţi specifice ale limbajelor de programare (şi ale compilatoarelor asociate), ale limbajului de comandă sau prin facilităţi (servicii) asigurate de un monitor.

Principalele tehnici şi metode de implementare a sincronizării dintre task-uri concurente sunt prin:

2.5.1 Semafoare generale

Un semafor general este o ***variabilă întreagă***, ce poate lua doar valori pozitive şi singurele operaţii ce se pot efectua asupra lui sunt operaţiile ***primitive P şi V***.

Dacă semaforul are valoarea zero, un program (task) ce încearcă să efectueze o operaţie P asupra acestuia va fi suspendat (blocat) şi va aştepta până când un alt program va efectua asupra aceluiaşi semafor o operaţie V.

Modul de utilizare a semafoarelor generale

Exemplu:mai multe programe sau task-uri denumite "***producătoare***", doresc să comunice o serie de date altor programe sau task-uri concurente, denumite şi "***consumatoare***" sau receptoare.

Necesar:- o ***zonă de memorie tampon***, de capacitate evident limitată, în care producătorii vor depune datele lor şi din care consumatorii le vor extrage când acestea devin disponibile.

- programele trebuiesc ***evident sincronizate***, astfel încât producătorii să nu depună date noi în zona tampon când aceasta este plină, iar consumatorii să nu extragă date din zona tampon când aceasta este goală.

Rezultă: ***înterpătrundere a mecanismelor de sincronizare cu cele de comunicare inter-programe (inter-proces) concurente.***

Sincronizarea necesară ***între programe*** se poate realiza în acest caz folosind ***semafoare generale*** care să reflecte condiţiile posibile de aşteptare (condiţia de "tampon plin" respectiv "tampon gol").

Pentru a evita suprascrierile sau săriturile peste anumite date din zona tampon, aceste operaţii trebuiesc efectuate prin excludere reciprocă. În acest scop, după cum s-a văzut, se poate introduce şi un ***semafor binar***.

Exemplu

Dacă se foloseşte, un semafor general ***"Plin"*** pentru a reprezenta numărul de locații încărcate (pline) ale zonei tampon, atunci un consumator va trebui întârziat dacă *Plin = 0*.

Dacă un task producător va efectua o operație V asupra semaforului "Plin", după ce a plasat o informație în zona tampon, un task-consumator va fi "servit" în urma efectuării unei operații P (va avea acces la tampon şi va putea "consuma" informația anterior introdusă).

În mod similar, un al doilea semafor general "*Gol*", poate fi folosit pentru a întârzia după necesități task-urile producător.

Concluzii

În general, dacă un task doreşte să aştepte până ce se realizează o anumită condiție, acestei condiții de aşteptare i se asociază un semafor şi orice task ce ar putea provoca realizarea condiției trebuie să aibă responsabilitatea efectuării unei operații V asupra acestui semafor.

Programatorul va trebui să introducă în fiecare task instrucțiuni de sincronizare cu celelalte task-uri, ceea ce presupune o atenție specială de lucru cu semafoarele generale.

Dacă mai multe task-uri consumator aşteaptă, ele vor trebui planificate într-un mod neutru sau după unele criterii de prioritate ale task-urilor consumator.

2.5.2 Regiuni critice condiționale

Este o structură de programare de forma :

```
with R when B do S
```

unde *B* este o expresie booleană iar *S* este o secțiune critică ce operează asupra variabilei comune *R*.

Această construcție specifică faptul că secțiunea critică de program S trebuie executată doar dacă condiția B este respectată.

2.5.3 Variabile de condiție

- ***dată*** ce poate fi utilizată ***doar*** în cadrul ***unui monitor***.
- sunt folosite pentru a identifica şirul de task-uri în aşteptare ce sunt manipulate prin operațiile ***"aşteptare"*** şi ***"semnalare".***

Operația ***"aşteptare"*** dezactivează task-ul şi îl trece într-un şir de aşteptare asociat variabilei de condiție respective, anulând excluderea reciprocă care ar fi interzis ca un alt task să obțină serviciile monitorului.

Operația ***"semnalare"*** va produce reactivarea primului task din şirul de aşteptare, asociat variabilei de condiție respective.

Sincronizarea explicită

Task-ul care efectuează o operație ***"semnalare"*** şi ***provoacă reactivarea*** unui alt task (trecut anterior în aşteptare) ***va fi***, la rândul său, ***suspendat*** până în momentul în care task-ul activat va ieşi din monitor sau va executa o operație "aşteptare".

OBS!!!

Prin detalierea specifică a condițiilor în care task-urile pot trece în aşteptare şi, respectiv, pot fi reactivate se poate realiza mai simplu şi mai explicit planificarea task-urilor decât în cazul regiunilor critice condiționale (unde erau nevoie evaluări repetate ale condițiilor) şi ***fiecare şir de aşteptare asociat variabilei de condiție*** este tratat după strategia "***primul - venit, primul - servit***" de fiecare dată când se execută o ***operație "semnalare"*** asupra variabilei de condiție respective.

2.5.4. Sincronizare prin comunicare

Comunicarea şi sincronizarea sunt în realitate doar fațete ale aceluiaşi fenomen

Programele comunică între ele nu numai pentru a-şi comunica informații sub formă de mesaje ci şi pentru a se sincroniza.

REZULTĂ

Un semnal de sincronizare poate fi considerat şi el că este un mesaj fără conținut ce se transmite între programe.

Cum se realizează acest lucru?

a)***Procese secvențiale comunicante. Rendez-vous simetric.***

task-ul A: o comandă de emitere mesaj → ***SUSPENDARE*** ← **task-ul B:** o comandă de **recepție de mesaj**;

task-ul B: o comandă de **recepție de mesaj** → ***SUSPENDARE*** ← **task-ul A:** o comandă de **emisie de mesaj**;

task-uri sincronizate→ datele (mesajul) sunt transferate

b) **Procese distribuite. Rendez-vous asimetric**

Comunicarea şi sincronizarea între programele concurente se realizează în acest caz ***similar cu apelarea prin nume de către programul emițător a unei proceduri incluse în programul receptor***, lista cu parametrii asociați acestui apel fiind folosită ca un "canal" de comunicare pentru transmiterea de date între cele două programe. Doar programul apelant trebuie să cunoască numele programului apelat, nu şi invers.

Implementare

*Pentru a putea provoca sincronizarea prin rendez-vous a celor două programe, în programul apelat vor trebui inserate **"comenzi de acceptare"**, care vor avea forma unor **proceduri de intrare apelabile de către programele apelante** (task-ul apelat este suspendat în aceste puncte de program).*

DEZAVANTAJE

Tipul de rendez-vous este unul asimetric.

Aceste forme de rendez-vous nu sunt suficient de adecvate pentru programarea aplicațiilor reale.

Sincronizarea mult prea strânsă a task-urilor împiedică orice operare asincronă a programelor şi astfel inhibă avantajele potențiale ale unui paralelism dezvoltat.

Procese distribuite. Rendez-vous asimetric

Eliminarea dezavantajelor:

Introducerea posibilității de selectare nedeterministă a comenzilor de acceptare. Este vorba despre un mecanism pe baza căruia un program receptor poate evita executarea unei comenzi de acceptare şi să fie suspendat. După necesități, se pot introduce condiții suplimentare de execuție a comenzii de selectare.

După cum vom constata, există şi ***mecanisme mixte***, care au atât caracter sincron, cât şi caracter asincron.