Caracterizarea sistemelor NESECVENȚIALE

Anca Vasilescu, Oana Georgescu

În episoadele anterioare am luat în discuție aspecte generale cu privire la problematica sistemelor nesecvențiale: probleme practice, clasice, specifice calculului paralel și concurent și, respectiv, clasificări ale sistemelor nesecvențiale. În acest episod vom defini conceptele specifice, insistând asupra comunicării datelor și sincronizării proceselor. Toate aceste elemente se vor constitui ca un fundament pentru abordarea în episodul următor a mecanismelor de comunicare între procese.

Un program concurent poate fi privit ca un ansamblu de componente care pot fi executate simultan. La o primă abordare, aceste componente sunt procesele care comunică între ele. În acest context sunt corecte ambele definiții ale conceptului de proces și anume: *un proces este un calcul care se poate executa paralel sau concurent cu alte calcule*, respectiv *un proces este o secvență de program în execuție*.

O condiție fundamentală a execuției concurente a programelor se poate enunța astfel: *pentru fiecare proces și pentru fiecare două instrucțiuni mașină consecutive ale procesului, intervalul de timp dintre executările lor este neprecizat, dar finit*.

Execuția concurentă a proceselor poate crea o serie de surprize celui obișnuit cu programarea secvențială. Dintre acestea, poate cea mai mare surpriză este legată de caracterul ***nedeterminist*** al execuției concurente. Comportarea nedeterministă constă în faptul că un același program concurent, pe un același set de date de intrare poate furniza la ieșire "rezultate" diferite. De cele mai multe ori, diferențele dintre ieșiri sunt cauzate de ordinea în care intră în execuție procesele active, din diferențele care pot apărea între vitezele de execuție ale diferitelor entități fizice implicate în execuția programului curent.

De asemenea, un program concurent poate fi privit ca un program care, în timpul execuției sale, creează mai multe procese care se execută într-un paralelism abstract (nu neapărat pe procesoare distincte). Existența unui singur procesor fizic impune alocarea unor ***cuante de timp*** pentru fiecare proces activ. Politica de alocare poate fi aleatoare sau poate respecta disciplina unei anumite structuri de date (coadă, coadă de priorități etc.).

Între procesele unui program concurent apar interacțiuni determinate în special de nevoia proceselor de cooperare. Aceste interacțiuni sunt de tip comunicare și/sau sincronizare. În paragrafele următoare se va arăta că aceste două concepte sunt interdependente.

În continuare vom caracteriza sistemele paralele și concurente prin definirea și descrierea conceptelor fundamentale, specifice:

- interacțiunea proceselor;
- comunicarea datelor și sincronizarea proceselor;
- proprietățile de atomicitate și granularitate, respectiv siguranță și vivacitate;
- excluderea reciprocă;
- interblocarea.

Fiecare dintre aceste caracteristici a apărut sub diferite aspecte în episoadele anterioare, începând chiar cu enunțarea problemelor clasice de concurență. Mai mult, în episodul următor, aceste proprietăți vor sta la baza evaluării instrumentelor pentru programarea nesecvențială, considerate în diferite limbaje de programare.

Interacțiunea proceselor

Interacțiunea proceselor depinde pe de o parte de aspecte legate de cooperarea și/sau competiția între procesele respective, iar pe de altă parte, depinde de deosebirea arhitecturală, structurală între sistemele cu partajare de date și cele fără partajare de date. Interacțiunea proceselor este un

subiect deosebit de important în tematica sistemelor concurente și, de aceea, el se va regăsi atât în această secțiune, cât și în tratarea comunicării datelor și sincronizării proceselor concurente.

Concret, la nivelul sistemului nesecvențial care se analizează, interacțiunea proceselor depinde în mare măsură de localizarea proceselor în funcție de structura modulară a *soft*-ului sistemului respectiv.

Putem prezenta această structură *software* ținând cont de proprietățile zonelor de memorie în care se execută procesele curente. Din acest punct de vedere se deosebesc următoarele situații:
- procesele pot partaja spațiu de adresă;
- procesele se pot executa în spații de adresă separate, distincte și complet disjuncte de spațiile de adresă ale altor procese.

Dacă procesele pot partaja spațiu de adresă, atunci avem mai multe variante de organizare a execuției proceselor, în funcție de necesitatea cooperării între procesele curente și de interacțiunea între procesele care pot partaja simultan, prin acces direct, un spațiu de adresă. Astfel, se deosebesc două variante principale:
- toate procesele partajează un același spațiu de adresă și de aici rezultă că se vor executa cu partajare de date;
- procesele se execută în spații de adresă diferite (fiecare proces poate avea spațiu de adresă propriu), dar este permisă partajarea unora dintre segmentele codurilor proceselor (segmentul de cod și/sau segmentul de date); dacă segmentul de date este unul care nu poate fi partajat, atunci procesele cooperante urmează să stabilească alt protocol de comunicare, nu prin partajarea datelor; după localizarea fizică a spațiilor de adresă proprii proceselor, se pot deosebi procesele care se execută pe aceeași mașină de cele care se execută pe mașini distincte.

Un exemplu de sistem care implementează aceste variante de gestiune a proceselor este un sistem *multiuser* în care se impun cerințe de protecție a memoriei când unul dintre utilizatori are restricții de scriere sau de citire la nivelul zonelor de memorie ale altui utilizator. De asemenea, într-un sistem *multiuser* trebuie impuse restricții legate de protecția asupra segmentelor de cod comun ale componentei *software* a sistemului.

Un alt exemplu îl constituie procesele distribuite care partajează date sau cod cu alte procese din sistem. Acestea sunt procese care ocupă spații de adresă distincte, chiar situate pe mașini diferite. Orice comunicare între procesele distribuite folosește mijloacele de comunicare din rețeaua mașinilor din sistem, prin intermediul interfețelor *hardware* și pe baza protocoalelor de comunicare recunoscute în sistemul distribuit respectiv.

Dacă procesele se pot executa în spații de adresă separate, distincte și complet disjuncte de spațiile de adresă ale altor procese atunci putem deosebi două variante specifice și anume:
- toate procesele se execută în spații disjuncte;
- sunt procese care se execută în spații disjuncte.

Primul caz reprezintă o contradicție cu definiția concurenței, deci nu poate fi luat în considerare dacă se dorește implementarea unui sistem concurent.

În cel de-al doilea caz, un proces care se execută într-un spațiu de adresă disjunct de cel al oricărui alt proces este un proces care nu va partaja nici segmente de date, nici segmente de cod cu alte procese din sistem. Un astfel de proces poate doar să intermedieze execuția unei instrucțiuni prin scrierea sau citirea unui operand din propriul spațiu de adresă. Acesta este cazul proceselor distribuite care se execută pe mașini diferite într-un sistem distribuit.

Pentru detalierea acestui subiect vă propunem să consultați [1].

Toate aceste variante de comunicare între procese fac obiectul mecanismelor ***IPC*** (***I**nter **P**rocess **C**ommunication*), pe care le vom aborda în episodul următor.

Cerințe pentru interacțiunea proceselor

Cerințele pentru interacțiunea proceselor rezultă, pe de o parte, din necesitatea ca procesele să interacționeze între ele și, pe de altă parte, din felul în care aceste interacțiuni pot fi suportate de sistemul gazdă.

Procesele pot coopera pentru îndeplinirea unei sarcini comune. Concret, un proces se poate afla în poziția de a cere altui proces furnizarea unui serviciu și eventual, ar putea să aștepte să primească acel serviciu. Exemple pentru această situație ar putea fi:
- aspectele cooperative care reies din diferite variante de implementare a localizării unui proces asociat unui anumit modul furnizor de servicii;
- cazul în care procesul de gestionare a discurilor trebuie să se sincronizeze cu procesele discurilor respective;
- prelucrarea proceselor în *pipeline* în care fiecare proces de fază trebuie să aștepte până când procesul fazei anterioare își încheie execuția.

Cooperarea între procese presupune sincronizarea lor, fapt care reiese din aspectele enumerate mai sus. Cu alte cuvinte, un proces poate fi nevoit să execute o operație de tipul **`WAIT`**`(PROCES)` pentru a se sincroniza cu alte procese. Sincronizarea între procese poate fi privită ca o extindere a sincronizării proceselor cu funcționarea componentelor *hardware* ale sistemului.

Comunicarea de date între procesele cooperante se realizează, de obicei, prin operații **`SIGNAL`**`(PROCES)`, deci prin semnale sau întreruperi.

Procesele pot concura pentru acces exclusiv la servicii sau la resurse. Din acest punct de vedere se disting două aspecte diferite și anume:
- mai mulți clienți adresează simultan cereri pentru anumite servicii;
- mai multe procese încearcă să acceseze simultan o resursă.

Implementarea sistemului concurent trebuie să prevadă și să gestioneze astfel de situații conflictuale prin stabilirea unei ordini predefinite, prin întârzierea execuției unora dintre procese, prin stabilirea unor priorități de acces la

resursele care pot deservi numai serial cererile pe care le primesc.

Procesele concurente pot să fie nevoite să execute operații **WAIT**(PROCES) înainte de a accesa o resursă partajată și trebuie să poată transmite semnale prin operații **SIGNAL**(PROCES) pentru a indica momentul în care au părăsit resursa partajată în sistem.

Procesele care se execută în spații de adrese distincte vor fi cunoscute sistemului de operare. În majoritatea cazurilor, codul acestor procese provine dintr-o sursă scrisă într-un limbaj de programare secvențial. În acest caz, operațiile **WAIT** și **SIGNAL** sunt recunoscute ca fiind *apeluri sistem*.

Procesele care partajează spații de adrese de memorie devin părți ale aceluiași program concurent și pot să fie sau nu cunoscute la nivelul sistemului de operare.

Dacă procesele nu sunt cunoscute de sistemul de operare atunci operațiile **WAIT**(PROCES) și **SIGNAL**(PROCES) vor putea fi lansate numai de la nivelul limbajului de programare.

Dacă procesele cooperează, atunci se pot dezvolta diferite mecanisme bazate pe operațiile **WAIT**(PROCES) și **SIGNAL**(PROCES) care să asigure comunicarea și sincronizarea prin construcții specifice pentru dezvoltarea aplicațiilor concurente în limbaje de programare de nivel înalt.

În cele ce urmează vom arăta că aceste două operații elementare (**WAIT**(PROCES) și **SIGNAL**(PROCES)) nu sunt suficiente pentru descrierea completă a unei scheme reale de comunicare între procese de tip *IPC*.

Comunicarea datelor și sincromizarea proceselor

Fiecare limbaj paralel trebuie să recunoască elemente specifice paralelismului fie explicit, fie implicit [2]. Practic, sunt mai accesibile limbajele de programare care nu cer construcții de limbaj pentru paralelism, dar permit procesarea paralelă printr-un paralelism implicit.

Aceasta este mai ușor de realizat în limbajele declarative, adică cele logice (*Prolog*) sau funcționale (*Lisp*), deoarece informația procedurală minimală trebuie paralelizată de un compilator "inteligent" care funcționează pe baza unor reguli de deducere și are o interacțiune foarte limitată cu programatorul.

Reprezentarea declarativă a cunoștințelor sau a problemei de rezolvat poate specifica fără echivoc o soluție. Totuși, poate fi destul de dificil să se convertească aceste cunoștințe într-un program paralel imperativ [3].

Indiferent dacă paralelismul este implicit sau explicit, trebuie să existe o cale de a crea procese paralele și o cale de a coordona activitatea acestor procese.

Uneori, procesele lucrează asupra datelor proprii și nu interacționează. Dar, atunci când procesele interschimbă rezultate, ele trebuie să comunice și să se sincronizeze între ele.

Comunicarea și sincronizarea pot fi realizate prin variabile partajate sau prin transfer de mesaje.

Comunicarea datelor în sisteme concurente

Comunicarea constă în schimbul de date (informații) între procesele care cooperează. Comunicarea poate fi identificată cu transmiterea de mesaje între procesele concurente.

Se pot identifica trei criterii de clasificare a tipurilor de comunicare:
- mijlocul de comunicare;
- gradul de sincronizare;
- modul de precizare a sursei și destinației comunicării.

În continuare vom detalia aceste clasificări. În funcție de mijlocul de comunicare avem:
- comunicare prin memorie comună;
- comunicare prin mecanisme specifice de tip monitor, semafor, resursă, canal.

Datorită faptului că o zonă de memorie utilizată în comun este o resursă *critică*, rezultă că, în acest caz, se impune programarea de secțiuni critice și implicit rezolvarea problemei excluderii mutuale.

În funcție de gradul de sincronizare avem cele două clase naturale de comunicare și anume:
- comunicarea fără sincronizare, care poate fi identificată cu transmiterea asincronă de mesaje, deci comunicare fără semnal de recepție;
- comunicarea cu sincronizare, care poate fi identificată cu transmiterea sincronă de mesaje; la rândul ei, comunicarea cu sincronizare poate fi realizată prin:
  - ♦ sincronizare simplă;
  - ♦ invocare la distanță.

În categoria *comunicare prin sincronizare simplă* putem încadra emiterea mesajului cu semnal de recepție, comunicarea prin canale sau cu punct de întâlnire de tip *rendez-vous* (vezi episodul următor). În cazul invocării la distanță ***recepția*** mesajului înseamnă atât *semnal de recepție*, cât și un răspuns de la destinatar la expeditor. Chiar și în sintaxa unor limbaje de programare concurente, invocarea la distanță apare ca și generalizare a comunicării prin canale sau ca un *rendez-vous* extins.

Pe de altă parte, invocarea la distanță reprezintă mecanismul ***RPC***, (***R**emote **P**rocedure **C**all*) de implementare a modelului *client/server* de interacțiune între procese: mai multe procese (active) acționează în calitate de clienți asupra unui proces server (pasiv) ce urmărește în principal satisfacerea cererilor clienților (vezi episodul următor).

În funcție de modul de precizare a sursei și destinației, comunicarea se poate face prin precizare:
- explicită;
- indirectă a partenerilor implicați în comunicare.

Prin precizare indirectă se înțelege folosirea unui intermediar între procesele care comunică. De exemplu, un astfel de intermediar ar putea fi un canal de comunicare care preia mesajul de la procesul sursă, în timp ce procesul destinație așteaptă să preia mesajul de pe acel canal.

Sincronizarea proceselor în sisteme concurente

Sincronizarea este o restricție impusă în evoluția în timp a proceselor. Ea constă în planificarea execuției a două pro-

cese, în cazul în care continuarea unuia depinde de execuția celuilalt.

Pe de altă parte, conform [4], sincronizarea proceselor constă în serializarea accesului proceselor concurente la resursele partajate.

Reluând o idee din descrierea comunicării, putem clasifica sincronizarea astfel:
- cu transmitere de mesaje (comunicare);
- fără transmitere de mesaje (fără comunicare).

În a doua situație sincronizarea se realizează la nivelul zonelor de memorie comune proceselor concurente.

Modalitatea naturală pentru rezolvarea sincronizării se dovedește a fi cooperarea logică prin diverse structuri de date. Limbajele de programare oferă programatorului posibilitatea de a opta între utilizarea diverselor tipuri abstracte de date pentru sincronizare, cu sau fără comunicare.

Din punctul de vedere al nivelului de sincronizare, programatorul poate opta pentru:
- sincronizare pe generație;
- sincronizare pe condiție.

Sincronizarea pe generație constă în întârzierea unui proces până când toate procesele din aceeași generație cu el și-au încheiat activitatea. Acest model se regăsește în rezolvarea problemei *jocul vieții* sau în soluțiile paralele/concurente pentru calculul unor valori numerice date de expresii sumă.

Sincronizarea pe condiție constă în întârzierea unui proces până când este îndeplinită o anumită condiție. Această variantă este mult mai des folosită în aplicațiile practice și datorită faptului că mecanismele de implementare a sincronizării pe condiție sunt mai bine reprezentate în majoritatea limbajelor de programare cu facilități pentru concurență.

Quinn [2] afirmă că există două metode de sincronizare:
- sincronizarea de precedență;
- sincronizarea pentru excludere mutuală.

Sincronizarea de precedență asigură faptul că un eveniment nu începe până când un altul nu s-a încheiat. *Sincronizarea pentru excludere mutuală* asigură faptul că, la un moment dat, numai un proces intră în secțiunea critică a codului cu care se prelucrează datele partajate.

Proprietăți specifice

Atomicitate și granularitate

Abordarea sistemelor concurente ridică o serie de probleme care pot fi rezolvate numai apelând la entități specifice. Din acest punct de vedere trebuie spus că tratarea concurenței trebuie să facă distincție între acțiuni concurente singulare (*single concurrent actions*) și acțiuni concurente compuse (*concurrent composite actions*). Putem defini o acțiune singulară ca fiind o operație de citire/scriere din/în memoria principală, dar problematica sistemelor concurente nu se poate folosi numai de acest tip de operații. Un motiv ar fi simplul fapt că procesele concurente sunt corect și complet descrise prin astfel de operații singulare numai dacă se poate presupune că orice operație este garantată de resursele *hardware* că este indivizibilă, adică atomică. Această condiție este prea restrictivă în condițiile unui sistem concurent general. De aceea, definiția unei acțiuni concurente singulare trebuie extinsă la operații de nivel mai înalt decât citirea/scrierea din/în memorie.

Potrivit lui ***Andrews*** [5], o acțiune atomică este o secvență de una sau mai multe stări care apare ca fiind indivizibilă, în sensul că nici un proces extern ei nu are acces la nici una dintre stările intermediare. Detaliind, *Andrews* definește două tipuri de acțiuni atomice:
- o acțiune atomică este de granularitate fină dacă poate fi implementată direct printr-o instrucțiune mașină unică;
- o acțiune atomică poate fi o secvență de acțiuni atomice de granularitate fină, indivizibilă.

O acțiune concurentă compusă rezultă prin combinarea acțiunilor singulare într-o operație de nivel înalt. Considerentele legate de aceste aspecte ale concurenței deschid două direcții distincte și anume:
- legarea acestor noțiuni de structura modulară a componentei *software* a unui sistem, în particular de tipurile abstracte de date recunoscute la nivelul sistemului respectiv;
- stabilirea legăturilor între acțiunile singulare astfel încât să se poată decide dacă ele pot forma sau nu acțiuni compuse.

În [5] se afirmă că o acțiune atomică realizează o transformare indivizibilă de stare. Aceasta înseamnă că orice stare intermediară care ar putea să existe în implementarea acțiunii nu trebuie să fie vizibilă pentru alte procese.

Proprietatea de granularitate reprezintă numărul de procesoare din sistem sau, dacă se referă la o aplicație, atunci granularitatea reprezintă numărul de procese care compun aplicația respectivă.

O acțiune atomică ***fine-grained*** este implementată direct de componenta *hardware* a sistemului pe care se execută programul concurent.

Uneori este nevoie să folosim un mecanism de sincronizare pentru a construi o acțiune atomică ***coarse-grained***, care este o secvență de acțiuni atomice *fine-grained* care apare ca fiind indivizibilă.

Exemple

Presupunem că o bază de date conține două valori x și y și că aceste valori trebuie să fie în fiecare moment egale. Cu alte cuvinte, nici un proces nu are acces la baza de date dacă $x \neq y$. Pentru a asigura această restricție, orice proces care modifică valoarea lui x trebuie să modifice și valoarea lui y, ca parte a aceleiași acțiuni atomice.

Fie un proces care inserează elemente într-o coadă reprezentată printr-o listă de legături. Un alt proces extrage elemente din listă, presupunând că există elemente în listă. Două variabile referă capul și, respectiv, coada listei. Valorile acestor variabile sunt modificate în timpul executării

operației de inserare, respectiv de extragere a elementelor din listă. Dacă lista conține un singur element, atunci o inserare și o extragere simultane pot crea conflicte, lista ajungând într-o stare instabilă. De aceea, inserarea și, respectiv, extragerea elementelor din listă trebuie să fie acțiuni atomice. Mai mult, dacă lista este vidă atunci trebuie să se întârzie executarea unei extrageri până când va fi executată o inserare.

Siguranță și vivacitate

Un atribut adevărat pentru oricare execuție a unui program este o proprietate a programului respectiv. Orice proprietate se poate exprima pe baza a două proprietăți speciale și anume:

- siguranța (*safety*);
- vivacitatea, viabilitatea (*liveness*) [5], [6], [1].

În general, un program este sigur dacă "execuția sa nu implică apariția unor evenimente neplăcute" și este viabil dacă "este de așteptat să apară un rezultat bun".

Un program secvențial este sigur dacă starea finală este corectă și este viabil dacă execuția lui se termină.

Caracterul nedeterminist al programelor concurente nu ne permite să extindem cu ușurință aceste fomulări de la programele secvențiale la programele concurente. De aceea, pentru un program concurent, definițiile proprietăților de siguranță și viabilitate vor folosi conceptele specifice aplicațiilor concurente.

Astfel, siguranța unui program concurent constă în asigurarea excluderii mutuale și absența interblocării (interblocarea apare când procesele așteaptă îndeplinirea unor condiții care nu pot fi satisfăcute), iar viabilitatea se exprimă prin faptul că un proces în așteptare va fi semnalizat în curând și astfel va putea să-și continue execuția.

Viabilitatea unui program concurent depinde de:

- corectitudinea alocării resurselor (*fairness*) - un proces va obține șansa de a intra în execuție, independent de starea altor procese;
- așteptarea activă (*starvation*) - un proces activ, care este blocat pe un timp nedefinit de unitatea de planificare (deși este teoretic capabil să se execute), va fi ales la un moment dat pentru a-și continua execuția. Această condiție este una esențială și pentru evitarea stării de blocare globală a sistemului (*deadlock*).

Excluderea reciprocă

Din considerente practice s-a ajuns la necesitatea ca, în contexte concrete, o secvență de instrucțiuni din programul de cod al unui proces să poată fi considerată ca fiind indivizibilă. Prin proprietatea de a fi indivizibilă înțelegem că după începerea execuției instrucțiunilor din această secvență, nu se va mai executa nici o acțiune a unui alt proces, până la terminarea execuției instrucțiunilor din acea secvență.

Trebuie să subliniem un aspect fundamental și anume că această secvență de instrucțiuni este o entitate logică care este accesată exclusiv de unul dintre procesele curente.

O astfel de secvență de instrucțiuni este o secțiune critică logică. Deoarece procesul care a intrat în secțiunea sa critică exclude de la execuție orice alt proces curent, rezultă că are loc o excludere reciprocă (mutuală) a proceselor între ele.

Dacă execuția secțiunii critice a unui proces impune existența unei zone de memorie pe care acel proces să o monopolizeze pe durata execuției secțiunii sale critice, atunci acea zonă de memorie devine o secțiune critică fizică sau resursă critică. De cele mai multe ori, o astfel de secțiune critică fizică este o zonă comună de date partajate de mai multe procese.

Problema excluderii mutuale este legată de proprietatea secțiunii critice de a fi atomică, indivizibilă la nivelul de bază al execuției unui proces. Această atomicitate se poate atinge fie valorificând facilitățile primitivelor limbajului de programare gazdă, fie prin mecanisme controlate direct de programator (din această categorie fac parte algoritmii specifici pentru rezolvarea anumitor probleme concurente).

Problema excluderii mutuale apare deoarece în fiecare moment cel mult un proces poate să se afle în secțiunea lui critică. Potrivit lui *Andrews* [5], excluderea mutuală este o primă formă de sincronizare, ca o alternativă pentru sincronizarea pe condiție.

Problema excluderii mutuale este una centrală în contextul programării concurente. Rezolvarea problemei excluderii mutuale depinde de îndeplinirea a trei cerințe fundamentale și anume [6]:

- excluderea reciprocă propriu-zisă - la fiecare moment de timp cel mult un proces se află în secțiunea sa critică;
- competiția constructivă, neantagonistă - dacă nici un proces nu este în secțiunea critică și dacă există procese care doresc să intre în secțiunile lor critice atunci unul dintre acestea va intra efectiv;
- conexiunea liberă între procese - dacă un proces "întârzie" în secțiunea sa necritică atunci această situație nu trebuie să împiedice alt proces să intre în secțiunea sa critică (dacă dorește acest lucru).

Îndeplinirea celei de-a doua condiții asigură faptul că procesele nu se împiedică unul pe altul să intre în secțiunea critică și nici nu se invită la nesfârșit unul pe altul să intre în secțiunile critice.

Cea de-a treia condiție asigură faptul că, dacă "întârzierea" este definitivă, aceasta nu duce la blocarea întregului sistem (cu alte cuvinte, blocarea locală nu implică blocarea globală a sistemului).

Excluderea reciprocă (mutuală) fiind un concept central, esențial al concurenței, în paragrafele următoare vom reveni asupra rezolvării acestei probleme la diferite niveluri de abstractizare și folosind diferite primitive. Principalele abordări sunt:

- folosirea suportului *hardware* al sistemului gazdă: rezolvarea excluderii mutuale se face cu ajutorul ***TAS***-urilor (***T**est **A**nd **S**et*); aceste primitive testează și setează variabile *booleene* (o astfel de variabilă este asociată unei resurse critice); un *TAS* nu este o operație atomică;

- folosind arbitrajul memoriei comune: soluția vizează secțiunile critice și nu resursele critice (cum s-a lucrat la nivelul *TAS*-urilor); la acest nivel aplicațiile folosesc semafoare;
- folosind suportul *software*: rezolvarea excluderii mutuale se bazează pe facilitățile oferite de limbajele de programare pentru comunicarea și sincronizarea proceselor executate nesecvențial.

Interblocarea

Se spune că un grup de procese concurente sunt ***interblocate*** dacă fiecare deține resurse comune care trebuie să fie alocate de alte procese pentru a-și continua execuția [2].

Interblocarea poate apărea ori de câte ori mai multe procese partajează resurse într-o manieră nesupervizată. De aici rezultă că interblocarea poate să apară atât în sisteme care operează în multiprogramare, cât și în sistemele multicalculator și multiprocesor.

Când un proces își alocă o resursă se spune că blochează acea resursă. Operațiile **`LOCK`** și **`UNLOCK`** corespund operatorilor *P* și *V* definiți de *Dijkstra* pentru semafoarele binare (vezi episodul următor) [7].

Exemple

Într-un sistem multiprocesor poate apărea următoarea situație de interblocare în care sunt implicate două procese, *Proces* 1 și *Proces* 2 și două resurse, *A* și *B*:

Proces 1	*Proces* 2
. . .	. . .
`LOCK(A)`	`LOCK(B)`
. . .	. . .
`LOCK(B)`	`LOCK(A)`
. . .	. . .

Atâta timp cât fiecare proces are acces exclusiv la o resursă cerută (solicitată) de celălalt proces, nici unul nu-și poate continua activitatea.

Într-un sistem multicalculator poate să apară un caz particular de interblocare și anume cea de tip *buffer deadlock*. Considerăm un sistem multicalculator în care procesele comunică sincron. Aceasta înseamnă că atunci când un proces *E* trimite un mesaj către un alt proces *D*, expeditorul *E* se blochează în așteptarea confirmării de la destinatarul *D*. În această situație, la expediere, mesajul este depus într-un *buffer* de mesaje ale destinatarului.

Presupunem că mai multe procese au trimis mesaje către *D*, acesta nu le-a preluat la timp și *buffer*-ul este plin. Cât timp *D* nu citește mesaje (deci nu-și descarcă *buffer*-ul), nici un mesaj trimis către *D* nu va mai putea fi recepționat și toate procesele expeditor sunt blocate în așteptarea confirmării.

Fie *I* un proces care încearcă să trimită mesaje către *D* cât timp *buffer*-ul este plin. Dacă *D* așteaptă să citească ***cu prioritate*** mesajul de la *I*, atunci procesul *D* se va bloca în așteptarea mesajului de la *I*. La rândul lui, procesul *I* este deja blocat în așteptarea confirmării de la *D*. Rezultă că cele două procese sunt interblocate datorită umplerii *buffer*-ului de recepționare a mesajelor.

În general, interblocarea poate apărea dacă:
- sistemul trebuie să asigure excluderea mutuală a proceselor la accesul pe resurse comune;
- sistemul nu respectă condiția de *preemption* (previziune, prioritate): un proces nu eliberează resursele pe care le deține dacă nu le-a utilizat pe toate;
- sistemul trebuie să respecte condiția de *resource waiting*: fiecare proces deține resursele cât timp așteaptă ca alte procese să le elibereze pe ale lor;
- există un lanț de procese în așteptare: fiecare proces așteaptă să se elibereze resursele pe care le deține un alt proces din lanț.

Indiferent de tipul de sistem nesecvențial, acesta trebuie să fie prevăzut cu mecanisme specifice pentru detectarea interblocării. Cu alte cuvinte, interblocarea este o stare a sistemului care trebuie evitată.

Rezolvarea problemei interblocării poate fi privită în trei moduri:
- prima abordare este să se detecteze apariția interblocării și să se încerce refacerea contextului anterior interblocării;
- a doua abordare este să se evite interblocarea prin utilizarea ***în avans*** a informațiilor referitoare la cererile de resurse, astfel încât să se realizeze un control al alocării resurselor care să ***preîntâmpine*** apariția interblocării;
- a treia abordare este să se prevină interblocarea prin a interzice apariția ultimelor trei situații prezentate anterior.

Bibliografie

1. **Bacon J.**, *Concurrent Systems - operating systems, database and distributed systems: an integrated approach*, Addison-Wesley, 1998
2. **Quinn M.J.**, *Parallel computing - theory and practice*, McGraw-Hill, 1994
3. **Braunl T.**, *Parallel programming - an introduction*, Prentice-Hall, 1993
4. **Ionescu F.**, *Principiile calculului paralel*, Editura Tehnică București, 1999
5. **Andrews G.R.**, *Concurrent Programming - Principles and Practice*, Redwood City CA: Benjamin/Cummings, 1991
6. **Georgescu H.**, *Programare concurentă - teorie și aplicații*, Editura Tehnică București, 1996
7. **Dijkstra E.W,**. *The structure of THE operating system*, Comm. ACM 11(5), 1968

D-na Anca Vasilescu este asistent universitar la Universitatea Transilvania din Brașov și poate fi contactată prin e-mail la adresa **vasilex@info.unitbv.ro**. *D-na Oana Georgescu este asistent universitar la Universitatea Sextil Pușcariu din Brașov și poate fi contactată prin e-mail la adresa* **o.georgescu@info.unitbv.ro**.