Analiza situațiilor de blocare

- Un sistem care este format din sisteme de sarcini combinate paralel

$$\mathbf{C}=\mathbf{C}_1 \mid \mathbf{C}_2 \mid \ldots \mid \mathbf{C}_n$$

este *blocat* dacă evoluţia sa este suspendată indefinit din cauză că mai multe sisteme de sarcini $\mathbf{C}_i$ sunt active simultan, fiecare utilizând, fără a putea fi întrerupte, resurse necesare altor sisteme din **C**.

- O resursă deţinută de o sarcina neinteruptibila poate fi eliberată numai de sistemul de sarcini care o utilizează la un moment dat.

- Pentru studiul situaţiilor de blocare vom privi sistemul fizic ca o colecţie de resurse R de tip:

  $R = R_1\ R_2 ... R_m$, fiecare tip conţinând:

  $W = W_1\ W_2 ... W_m$ exemplare

  Vectorul $W=(W_1,W_2,...,W_m)$ reprezintă capacitatea sistemului.

Exemplu de resurse R_i:

- unităţi centrale de prelucrare;
- coprocesoare matematice;
- pagini de memorie;
- magistrale de interconectare;
- înregistrări fizice (sectoare, piste, cilindri) la discuri magnetice;
- fişiere de date;
- zone de mesaje;
- tabele de descriere a diferitelor sarcini;
- variabile globale, etc.

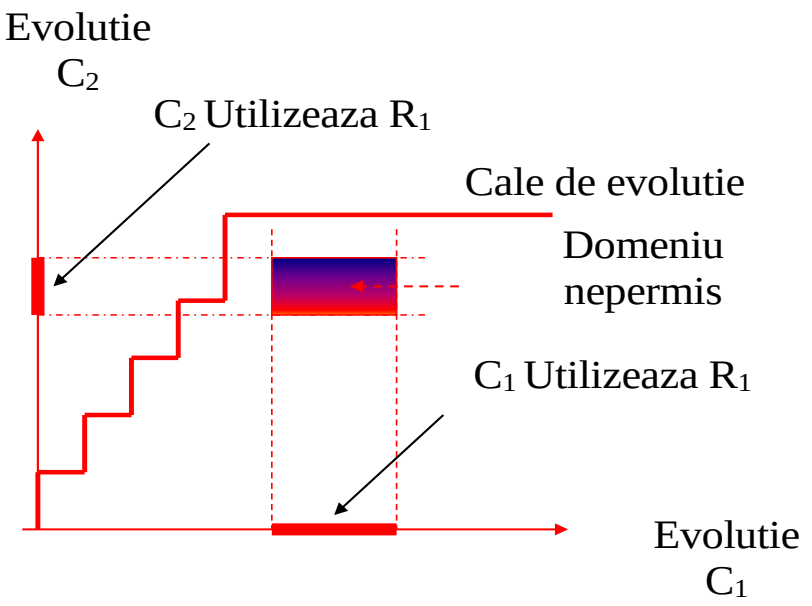
- Să considerăm un exemplu simplu de blocare:

  Fie două sisteme **C**$_1$ şi **C**$_2$ şi câte o resursă de tip R_1 şi R_2.

Evoluţia întregului sistem poate fi reprezentată în spaţiul cu două dimensiuni definit spaţiul de evoluţie.

Evoluţia sistemului **C**=**C**$_1$ | **C**$_2$ este reprezentată de succesiunea de puncte (x_1y_1) (x_2y_2)...(x_ky_k)... în acest spaţiu

Presupunem că originea este în punctul (0,0).
Evoluţia este ireversibilă deci $x_k \leq x_{k+1}$, $y_k \leq y_{k+1}$ şi
Nu orice cale de evoluţie, este realizabilă.
Evoluţia în care **C**$_1$ şi **C**$_2$ asftel sa utilizeaze simultan aceeaşi resursă R_1 este nepermisă
Zona hasurată reprezintă un domeniu nepermis, care conduce la blocarea sistemului.



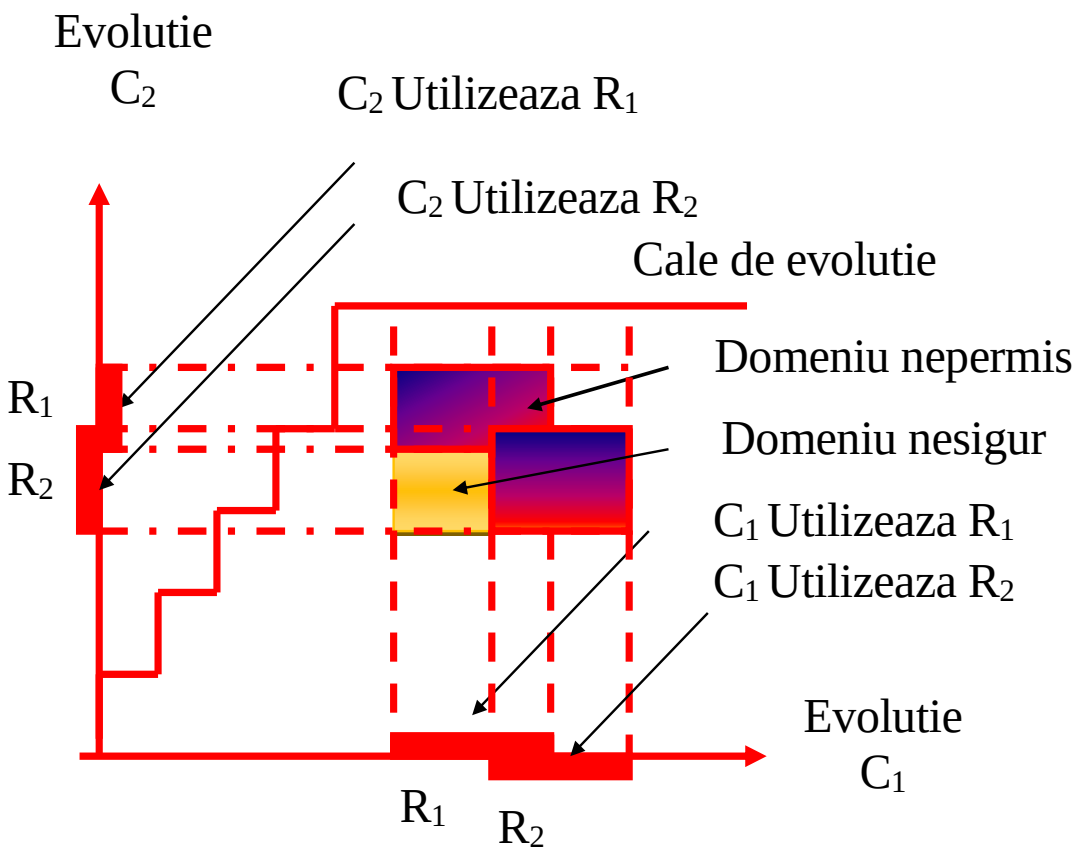
Exemplu de utilizare a doua resurse

Fie utilizarea resurselor R1 şi R2 de către C1 şi C2.

Zona haşurată reprezintă un domeniu nepermis, care conduce la blocarea sistemului.

Zona portocalie reprezintă un domeniu nesigur, care permite în continuare evoluţia sistemului, însă pentru un timp limitat.

Datorită ireversibilităţii procesului de evoluţie se va ajunge în mod sigur la blocarea sistemului.



- Blocarea este definită ca o stare de *aşteptare* circulară nerezolvabilă din cauza condiţiilor de excludere mutuală şi neintreruptibilitate.
- Pentru a exista o cale de evoluţie care conduce la blocare sunt necesare următoarele condiţii:

- **Utilizare exclusivă**: Fiecare sistem de sarcini doreşte
- controlul exclusiv asupra resursei pe care o utilizează;
- **Neîntreruptibilitate**: O resursă nu este eliberată până la terminarea completă a utilizării acesteia;
- **Aşteptare circulară**: Fiecare sistem de sarcini ţine ocupate resurse în timp ce aşteaptă ca altele să elibereze resurse pe care să le preia.

- Relativ la blocare se pun trei probleme:

1. Prevenirea

Una sau mai multe din condiţiile care conduc la blocare sunt înlăturate printr-o proiectare adecvată sau prin restricţii în utillizarea resurselor.

2. Detectarea şi înlăturarea

Sistemul de operare si planificatorul supraveghează evoluţia tuturor sistemelor de sarcini luând măsuri speciale de corecţie a evolutiei la detectarea situaţiilor de blocare.

3. Evitarea

Pe baza unor informaţii preliminare despre cerinţele de resurse, sistemul de operare / planificatorul dirijează evoluţia pentru evitarea zonelor nesigure.

Considerând că fiecare tip de resursă conţine un singur element $w_1=w_2=...=w_n=1$, putem alege o reprezentare sub forma de grafuri, fiecare nod reprezentând o resursă iar arcele reprezinta cererea de resurse.

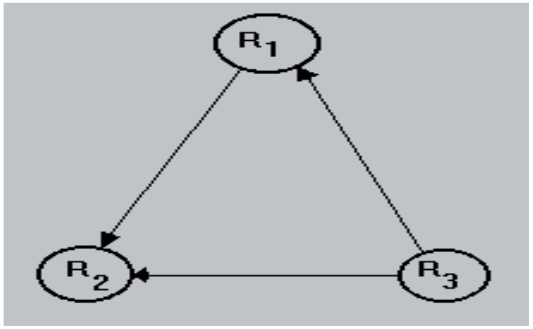
Arcele pot fi etichetate cu necesarul de resurse.

- Arcele sunt orientate şi sunt definite astfel:

  dacă la un moment dat, după evenimentul a_k, sistemul **C** ţine ocupată resursa R_i în timp ce aşteaptă preluarea lui R_j, graful conţine un arc orientat de la nodul R_i la nodul R_j.
- În exemplul urmator numărul de tipuri de resurse este n=3;

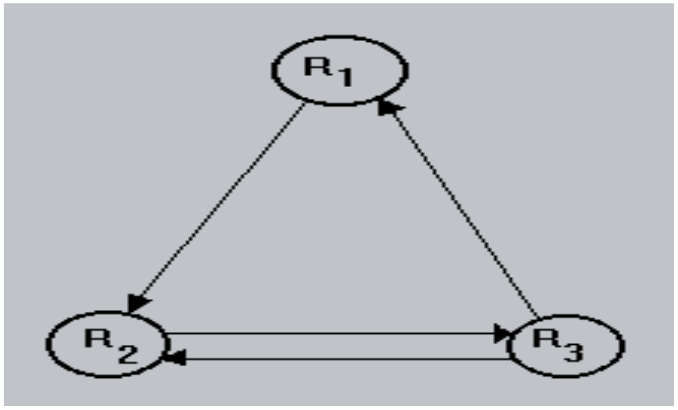
  O sarcina utilizează R_1 şi o cere pe R_2 ,

  Alta sarcina utilizează pe R_3 şi cere pe R_1 şi R_2.



Graf de cereri fara bucle

Dacă cele trei condiţii de blocare sunt posibile (operative) condiţia necesară şi suficientă pentru blocare este existenţa unei bucle (circuit, ciclu) în graful de cereri.
Exemplu , de graf de cereri în care se realizează condiţia de aşteptare circulară cu resurse ocupate.



Graf de cereri cu bucle

- Pentru cazul general în care există mai multe resurse de un anumit tip, fiecare nod reprezintă un tip de resursă.
- Cererile pentru resurse R_i trebuie să nu depăşească, ca număr, w_i.
- Un arc de la R_i la R_j arată că există cel puţin o sarcina care cere una sau mai multe resurse de tip R_j şi deţine una sau mai multe resurse de tip R_i.
- În cazul acesta o buclă în graful de cereri reprezintă o condiţie necesară dar nu şi suficientă pentru existenţa situaţiei de blocare.

Prevenirea blocării

- Pentru a proiecta un sistem în care posibilitatea blocării este exclusă trebuie să existe certitudinea că în orice moment de timp cel puţin una din condiţiile necesare nu este îndeplinită.
- Prima din cele trei condiţii (***excluderea mutuală***) este uneori impusă de restricţii fizice privind accesul la resurse nepartajabile şi deci nu poate fi eliminată pentru orice fel de resurse.

Pentru celelalte condiţii vom enumera câteva posibilităţi:

- Dacă o sarcina care utilizează o resursă, cere o altă resursă *trebuie* să o elibereze pe prima şi dacă este necesară în continuare să o ceară din nou împreună cu resursele adiţionale de care are nevoie (***interuptibilitate***).
- Fiecare sarcina să ceară toate resursele de care are nevoie, putând continua numai după ce le are pe toate (***aşteptare cu resurse***).

- ***Ordonarea liniară*** a tipurilor de resurse pentru toate sarcinile. Dacă o sarcina utilizează resurse de tip R_i, la un moment dat, ea poate cere numai resurse din tipurile următoare lui R_i în acea ordonare,

  exemplu: R_j cu j>i. În acest caz, graful de cereri nu are circuite.

*Observaţie:

- **Prima soluţie** se pretează numai pentru resurse interuptibile, a căror stare poate fi uşor salvată şi restaurată. De exemplu UCP.
- **A doua soluţie** poate deveni costisitoare dacă unele resurse sunt alocate unor sarcini şi sunt neutilizate o perioadă lungă de timp.
- **A treia soluţie** poate fi utilizată numai pentru anumite componente, de exemplu lansarea job-urilor într-un sistem cu multiprogramare prin alocarea memoriei şi a dispozitivelor de I/E într-o ordine fixă.

Detectarea şi înlăturarea blocării

- Pentru cazul în care există o singură resursă de fiecare tip, mecanismul de detectare a blocării constă:
  - dintr-o *procedură* care actualizează graful de cereri ori de câte ori apare o cerere, o preluare sau eliberare de resurse şi
  - o *procedură* de detectare a circuitelor (buclelor) în acest graf.
- Pentru cazul general, la fiecare moment de timp t, corespunzător secvenţei de execuţie $\alpha = a_1 a_2 ... a_k$, se definesc
- $x_{ij}[k]$ reprezintă numărul de resurse de tip R_j alocate, respectiv şi
- $y_{ij}[k]$ cerute de $\mathbf{C}_i$, după evenimentul a_k din α, unde:

  $\mathbf{C}=\mathbf{C}_1 \mid \mathbf{C}_2 \mid ... \mid \mathbf{C}_n$, $1 \le i \le n$ iar $1 \le j \le m$,

  m fiind numărul de tipuri de resurse

Se obţin astfel:

$$X[k] = \begin{bmatrix} x_1(k) \\ x_2(k) \\ . \\ . \\ x_n(k) \end{bmatrix} \text{ matrice de alocare}$$

cu

$$Y[k] = \begin{bmatrix} y_1(k) \\ y_2(k) \\ . \\ . \\ y_n(k) \end{bmatrix} \text{ matrice de cereri}$$

$X_i[k] = (x_{i1}[k], x_{i2}[k],..., x_{im}[k])$

$Y_i[k] = (y_{i1}[k], y_{i2}[k],..., y_{im}[k]), \quad 1 \le i \le n$

- Starea s_k corespunzătoare secvenţei α după evenimentul a_k este dată de perechea [X[k], Y[k]].

Să notăm cu:

$V[k]=(v_1[k],v_2[k],...,v_m[k])$ vectorul resurselor disponibile.

Elementul $v_j[k] \le w_j$, $1 \le j \le m$, numărul de resurse de tip R_j disponibile la un moment dat după evenimentul a_k este:

$$V_j(k) = W_j - \sum_{i=1}^{n} X_{ij}[k], \quad 1 \le j \le m$$

- Pentru doi vectori X şi Y, relaţia $X \le Y$ este adevarată, dacă şi numai dacă relaţia este adevărată pentru fiecare pereche de elemente corespondente din X şi Y.
- Utilizând informaţia conţinută în starea s_k dată de perechea de matrice [X[k], Y[k]] vom elabora un algoritm de detectare a blocării.

Definiţie:

Fie $\mathbf{C} = \mathbf{C}_1 \mid \mathbf{C}_2 \mid ... \mid \mathbf{C}_n$ unde $\mathbf{C}_i$, $1 \le i \le n$, este de forma

$$\mathbf{C}_i = S_i[1]\ S_i[2].....S_i[l_i], \qquad l_i \ge 1.$$

Fie

$\alpha = a_1 a_2 ... a_k$ o secvenţă de execuţie parţială a lui **C** iar

$\sigma = s_0\ s_1 ... s_k$ secvenţa parţială de stări, corespunzătoare.

Dacă există o mulţime nevidă D de indici ai lui **C** astfel că pentru orice i din D nu este adevărată relaţia:

$\mathbf{Y_i[k]} \le$	$\mathbf{V[k]} +$	$\mathbf{\Sigma\ X_j[k]}$ $\mathbf{j \notin D}$
vector de cereri	*vector de resurse disponibile*	*vector de resurse allocate catre (deţinute de) sarcini ce nu aparţin lui D*

atunci σ este blocată (sau s_k este o stare de blocare).

De asemenea putem spune că orice sistem de sarcini $\mathbf{C}_i$ cu i ε D este blocat.

Exemplu:

Fie **C_1** şi **C_2** două sisteme de sarcini:

$C_1 = S_1[1], S_1[2], S_1[3]$ şi $C_2 = S_2[1], S_2[2]$

care utilizează resursele $R=(R_1,R_2,R_3,R_4)$ cu $w = (3,3,3,3)$, în felul următor :

	C 1			C 2	
	resurse cerute	resurse eliberate		resurse cerute	resurse eliberate
$S_1[1]$	(0,1,2,0)	(0,0,2,0)	$S_2[1]$	(2,0,2,2)	(0,0,0,2)
$S_1[2]$	(1,0,0,1)	(0,0,0,1)	$S_2[2]$	(1,0,1,0)	(3,0,3,0)
$S_1[3]$	(0,0,2,0)	(1,1,2,0)			

Secvenţa de execuţie $\alpha = SI_1[1]$ $SF_1[1]$ $SI_2[1]$ $SF_2[1]$ $SI_1[2]$ $SF_1[2]$ $SI_2[2]$ .....

	X	Initial	$SI_1[1]$	$SF_1[1]$	$SI_2[1]$	$SF_2[1]$	$SI_1[2]$	$SF_1[2]$	$SI_2[2]$
		R_1,R_2,R_3,R_4	R_1,R_2,R_3,R_4	R_1,R_2,R_3,R_4	R_1,R_2,R_3,R_4	R_1,R_2,R_3,R_4	R_1,R_2,R_3,R_4	R_1,R_2,R_3,R_4	
Alocare	X_1	0 0 0 0	0 1 2 0	0 1 0 0	0 1 0 0	0 1 0 0	1 1 0 1	1 1 0 0	Blocare
X[k]	X_2	0 0 0 0	0 0 0 0	0 0 0 0	2 0 2 2	2 0 2 2	2 0 2 0	2 0 2 0	
Cereri	Y_1	0 1 2 0	0 0 0 0	1 0 0 1	1 0 0 1	1 0 0 1	0 0 0 0	0 0 2 0	
Y[k]	Y_2	2 0 2 2	2 0 2 2	2 0 2 2	0 0 0 0	1 0 1 0	1 0 1 0	1 0 1 0	
V[k]		3 3 3 3	3 2 1 3	3 2 3 3	1 2 1 1	1 2 1 3	0 2 1 2	0 2 1 3	

După evenimentul S1[2], atât Y1[k] ≤ V[k] cât şi Y2[k] ≤ V[k] astfel că P1[3] şi P2[2] nu pot avea loc cu valorile obţinute pentru V[k] înaintea acestor evenimente.

Sistemul s-a blocat deoarece nici C1 nici C2 nu pot continua fără ca unul din ele să fie întrerupt pentru a elibera resursele deţinute.

Algoritm pentru detectarea blocării

Etapele algoritmului sunt:

1°. Iniţializare : $D \leftarrow \{1,2,...,n\}$; $V \leftarrow V(k)$
2°. Caută indicii i din D pentru care $Y_i(k) \leq V$ şi execută pasul 3.
Dacă nu există nici un indice, algoritmul se termină.
3°. $D \leftarrow D \setminus \{i\}$; $V \leftarrow V + X_i(k)$ şi salt la pasul 2.

- Algoritmul se bazează direct pe definiţia de mai sus şi constă din simularea execuţiei sarcinilor până când rămâne un set de sisteme de sarcini $\mathbf{C}_i$ pentru care, în final, resursele disponibile nu sunt suficiente sau mulţimea de indici D devine vidă.
- În cel mai defavorabil caz algoritmul dat analizează la pasul i, întreaga mulţime de sisteme de sarcini din D, în număr de (n-i+1).

Timpul de execuţie al algoritmului este în acest caz proporţional cu

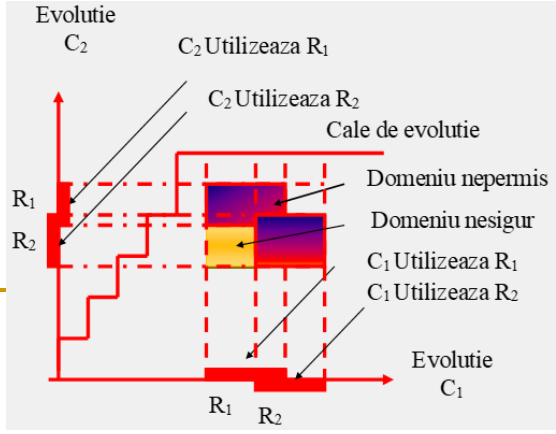
$$m \cdot \sum_{i=1}^{n} (n-i+1) = O(m.n^2)$$

- Printr-o aranjare a structurilor de date şi o reprezentare mai elaborată a stărilor resurselor se poate obţine un algoritm cu timpul de execuţie proporţional cu O(m.n).
- Complexitatea acestui algoritm face ca implementarea lui să fie justificată numai în sisteme în care apariţia unei blocări este intolerabilă.
- În multe sisteme se utilizează o soluţie simplă care se bazează pe limitarea timpului în care sistemul poate sta într-o anumită stare.
- Dacă s-a ajuns la această limită se consideră că s-a realizat o situaţie de blocare şi sistemul este scos din blocare şi trecut într-o anumită stare, predeterminată din care continuă evoluţia.
- Implementarea metodelor de detectare a situaţiilor de "time out" se face atât la nivelul maşinii de bază cât şi la nivelul sistemului de operare.

Evitarea blocării

- Pentru a analiza problema evitării blocării se presupune că o sarcină S constă dintr-o succesiune de paşi.
- Pe parcursul execuţiei unui pas utilizarea resurselor rămâne constantă şi aceasta implică achiziţionarea resurselor necesare pentru acest pas în plus faţă de cele primite la pasul precedent.
- Urmează apoi o perioadă de execuţie în care cerinţele de resurse nu se schimbă.
- În final când se termină execuţia, toate resursele care nu sunt cerute la pasul următor sunt eliberate şi trecute într-o colecţie de resurse disponibile.

- Fie $\sigma = s_0 s_1 ... s_k$ secvenţa de stări corespunzătoare secvenţei de execuţie $\alpha = a_1 ... a_k$.
- Dacă există cel puţin o secvenţa de execuţie completă, validă, care are ca prefix pe α atunci s_k este o stare sigură, altfel starea s_k este o stare nesigură cu pericol de blocare în evoluţia următoare.
- Cu alte cuvinte s_k este *sigură* dacă utilizând resursele disponibile V[k] şi cele care vor fi eliberate de paşii care se execută la momentul respectiv, este posibil să găsim o secvenţă validă de paşi încă neiniţiaţi, pentru sarcinile iniţiate dar neterminate astfel încât toate sarcinile din sistem să poată continua execuţia spre finalizare.
- Starea iniţială în care încă nu s-au alocat resurse, toate fiind disponibile, este intotdeauna o stare sigură.
- Deci există posibilitatea finalizării tuturor sarcinilor.



- Să considerăm în continuare problemele pe care trebuie să le rezolve un supervizor pentru evitarea blocării.
- Să presupunem că sistemul se găseşte în starea s_k sigură şi că există cererile de resurse date de Q[k].
- Supervizorul trebuie să determine dacă orice cerere din Q[k] poate fi satisfacută, $y_{ij}[k] \leq v_j[k]$.
- În general pot fi mai multe astfel de cereri şi deci se pot prevedea diferite *criterii de alocare* pentru a stabili care cerere să fie satisfacută mai întîi, de exemplu:

  FIFO - (First In First Out)-primul venit primul servit

  SJF - (Shortest Job First)- sarcina cea mai scurtă,prima servită;

  ROUND ROBIN - rulare prin rotaţie;

  Priorităţi atribuite sarcinilor;

  etc.

- Pentru a determina dacă o stare s_k este sigură ca urmare a satisfacerii unor cereri de resurse trebuie căutată o secvenţa adecvată de paşi.
- Pentru aceasta se presupune că resursele disponibile sunt completate cu cele alocate sarcinilor în curs de execuţie.
- Aceasta este echivalentă cu presupunerea că primul pas într-o secvenţă analizată nu începe până când toţi paşii în curs de execuţie au toate condiţiile pentru a se finaliza.

Excluderea mutuală
Definiții. Concepte de bază.

- Într-un sistem de sarcini concurente, acestea pot interacţiona în două moduri:
  - indirect, prin disputa pentru aceleaşi resurse;
  - direct, prin transmitere / recepţie de mesaje sau informaţii de stare.

- Când se execută sarcini independente evoluţia acestora poate fi controlată de către supervizor aşa cum am văzut la analiza situaţiilor de blocare.

- Totusi când sarcinile interactionează în mod direct mecanismul de control al evoluţiei lor trebuie să apară în mod explicit în programele care le definesc.

- Interacţiunea directă se efectuează prin mecanisme specifice contextului în care evoluează sistemul de sarcini concurente.

Contextul în care evoluează un sistem de sarcini concurente poate fi:

- centralizat, dacă există o memorie comună accesibilă pentru citire/scriere tuturor sarcinilor, sau
- distribuit dacă nu există memorie comună, ci doar memorie locală accesibilă unei sarcini şi inaccesibilă celorlalte sarcini.

- Excluderea mutuală se ocupă de problemele legate de controlul resurselor reutilizabile (acele resurse a căror stare internă este modificată în timpul utilizării dar care poate fi initializată la atribuirea lor altor sarcini) astfel ca la un moment dat să fie utilizate de o singură sarcină.

- Deoarece sarcinile modifică starea resursei în timpul utilizării, este necesar ca o resursă să fie alocată unei singure sarcini la un moment dat pentru a asigura o execuţie corectă a sarcinilor.
- Pe de altă parte, deoarece o sarcină iniţializează starea unei resurse înainte de a o utiliza, ordinea în care mai multe sarcini o utilizează este indiferentă.
- Astfel orice soluţie pentru excluderea mutuală care implică a priori constrîngeri privind ordinea de execuţie a sarcinilor este neadecvată.
- O soluţie corectă trebuie să se aplice unui sistem de sarcini independente.
- Dacă utilizarea resurselor este sub controlul unui sistem de operare centralizat, la construirea unui sistem de sarcini trebuie să se ţină seama numai de convenţiile prin care se comunică sistemului de operare cererile şi eliberările de resurse.

- Vom considera că excluderea mutuală trebuie rezolvată de însuşi sistemul de sarcini, utilizând *variabile globale* sau variabile de stare şi mesaje pentru comunicaţia între sarcini în funcţie de context: centralizat sau distribuit.

- Sarcinile care trebuie să se execute mutual exclusiv constituie secţiuni critice.

- O soluţie corectă pentru excluderea mutuală trebuie să elimine apariţia unor rezultate diferite sau blocari în tranziţiile sistemului în condiţiile unor timpi de execuţie a sarcinilor diferiţi, pentru diferite procesoare.
- De exemplu: Să considerăm posibilitatea ca două sarcini S_1 şi S_2 care se execută pe procesoare diferite să utilizeze în comun o resursă R.

- S_1 şi S_2 utilizează o variabilă globală care specifică disponibilitatea resursei R.
- S_1 testează variabila, găseşte R disponibilă şi înainte ca S_1 să schimbe valoarea variabilei pentru a arăta că R este ocupată, S_2 testează şi ea variabilă şi găseşte R disponibilă.
- Mai concret S_1 şi S_2 care evoluează într-un context centralizat au primit aceeaşi interpretare, citirea variabilei Semafor, incrementarea valorii citite şi scrierea noii valori.

Fie, de exemplu, secvenţa $SI_1\ SI_2\ SF_1\ SF_2$, deci

1°. S_1 citeşte variabila Semafor;

2°. S_2 citeşte variabila Semafor;

3°. S_1 incrementează şi scrie noua valoare;

4°. S_2 incrementează şi scrie noua valoare.

În urma acestei secvenţe de execuţie, Semafor primeşte valoarea Semafor + 1 în loc de Semafor + 2.

- Rezolvarea cererilor de acces la memorie pentru UCP şi DMA reprezintă un exemplu de excludere mutuală la nivel hardware.
- Scrierea unor texte la o imprimantă comună într-o reţea locală de calculatoare constituie un exemplu de excludere mutuală, la nivel de aplicaţii.
- Accesul la o înregistrare dintr-un fişier pentru a fi citită şi modificată de mai multe sarcini constituie deasemenea un exemplu de excludere mutuala în context distribuit.

- Pentru simplitate vom considera în continuare o singură resursă R care poate fi utilizată de o singură sarcină la un moment dat.

Definiţie:
Fie $\alpha = a_1,a_2,...,a_k$ o secvenţă de execuţie parţială a sistemului de sarcini **C**=(***S***,<). Sarcinile S şi S' sunt mutual exclusive în α dacă şi numai dacă cel mult una din ele este activă după orice prefix a lui α.

Pentru a implementa excluderea mutuală vom considera că o soluţie este corectă dacă cel mult o sarcină este în secţiunea critică la un moment dat şi sunt satisfacute următoarele condiţii adiţionale:

- Nu se face nici o presupunere relativă la activitatea şi numărul de procesoare pe care se execută sistemul de sarcini.

  Desigur, aşa cum s-a arătat şi mai sus, se consideră ca citirea şi scrierea unei variabile de memorie sunt operaţii atomice, indivizibile. Lansarea simultană a unor astfel de operaţii va avea ca efect execuţia lor secvenţială, într-o ordine oarecare, necunoscută apriori.

- Soluţia implementată nu trebuie să fie afectată de vitezele relative ale sarcinilor care se exclud mutual, considerând că aceste viteze sunt diferite de zero.
- O sarcină care operează în afara secţiunii critice nu poate bloca altă sarcină de la intrarea în secţiunea critică.
- Două sarcini care doresc să intre în secţiunea critică nu pot fi într-o buclă de aşteptare reciprocă.

  Deci dacă se eliberează secţiunea critică şi există sarcini în aşteptare, este exclusă posibilitatea ca nici un proces să nu intre în secţiunea critică. În plus, o sarcină care doreşte să intre în secţiunea critică nu poate fi în stare de aşteptare un timp nedefinit.

- Toate sarcinile trebuie tratate la fel, fără priorităţi sau alte privilegii.

Algoritmi de excludere mutuală în mediu centralizat

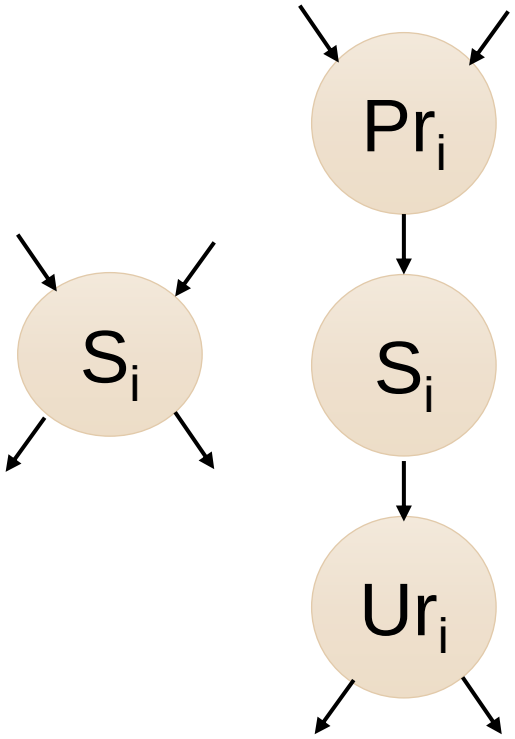
- Complexitatea soluţiei pentru excluderea mutuală depinde de tipurile de operaţii indivizibile care se pot efectua asupra variabilelor globale.
- Astfel dacă se lasă pe seama proiectantului hardware responsabilitatea implementării excluderii mutuale prin operaţii indivizibile mai elaborate, se obţine o soluţie mai simplă pentru excluderea mutuală decât dacă se consideră ca operaţii indivizibile simple citiri şi scrieri în memorie.
- Astfel, sunt calculatoare care au instrucţiuni de tipul Test and Set (Read Modify Write), sau chiar primitive de sincronizare indivizibile, implementate ca operaţii atomice.

- Dacă cel puţin două sarcini independente ale unui sistem **C** trebuie să utilizeze mutual exclusiv resursa R vom modifica **C** pentru a asigura această cerinţă.
- Pentru fiecare S_i care utilizează resursa R vom introduce sarcinile Pr_i şi Ur_i astfel că:

  Pr_i precede imediat sarcina S_i, iar

  Ur_i urmează imediat sarcinii S_i.
- Vom nota cu **C'** sistemul de sarcini extins. Cu toate că S_i rămâne neinterpretat Pr_i şi Ur_i vor fi complet specificate.
- Pr_i va fi astfel proiectat încât să poată fi terminat şi deci să permită iniţierea lui S_i dacă şi numai dacă pentru $\forall\ j \neq i$, Pr_j nu poate fi terminat.
- Funcţia principală a lui Ur_i este de a comunica celorlalte sarcini că S_i a terminat utilizarea lui R.

```
do{
   Entry Section
   Critical Section
   Exit Section
   } while (TRUE);
```



- Pentru a proiecta pe Pr_i şi Ur_i astfel ca să nu existe subsecvenţe de execuţie de forma:

$PrF_i \; PrF_j \, UrF_i$

vom considera ca variabile globale:

p - o variabilă de tip întreg.

Q - (Q[0],Q[1],...,Q[n]) - un vector de n+1 variabile de tip întreg.

Iniţial p=0 şi Q[i]=0, $0 \le i \le n$.

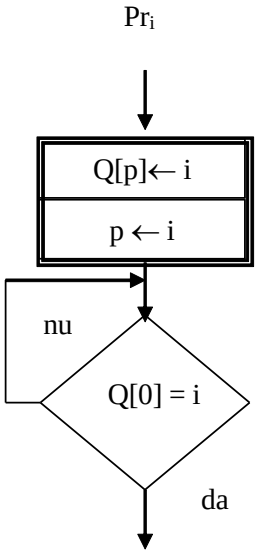
- Funcţia vectorului Q este de a memora într-o ordine FIFO lista sarcinilor ce aşteaptă utilizarea lui R. Acest lucru se realizează prin memorarea indicelui sarcinei care utilizează pe R în Q.
- Astfel, coada de aşteptare la resursa R este organizată ca o simplă listă înlănţuită, memorată în Q.
- *Variabila p* conţine indicele ultimului element din lista (ultimul element intrat în coada Q).
- Condiţia de coadă vidă, considerată ca şi condiţie iniţială este dată de p= 0 şi Q[0] = 0.

Definirea sarcinilor auxiliare Pr si Ur

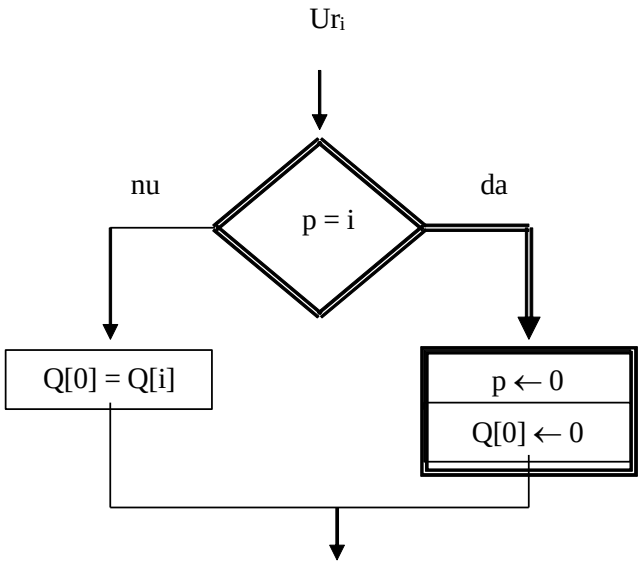
Procesele auxiliare Pr_i şi Ur_i pot fi specificate prin organigramele urmatoare

Sarcina Pr_i arată intenţia lui S_i de a utiliza resursa R, introduce pe S_i în coada Q şi aşteaptă într-o buclă până când S_i ajunge în capul listei Q[0].

Sarcina Ur_i înlătură pe S_i din coadă şi trece următoarea sarcină pentru execuţie în capul listei.
Dacă S_i a fost ultima sarcină introdusă în coadă
(p=i când se execută Ur_i)
atunci coada este vidă şi se trec în zero p şi Q[0].



a) Sarcina Pr_i



b) Sarcina Ur_i

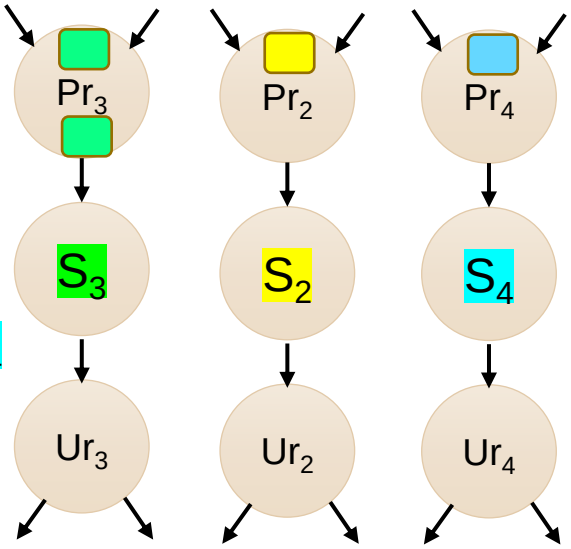
Pentru a putea elimina posibilitatea interferenţelor între Pr_i,Pr_j sau Pr_i,Ur_j ce se execută concurent trebuie ca anumite operaţii asupra variabilelor globale să **fie indivizibile**, cele în chenar dublu.

Fie un sistem de sarcini C=(S,<)

$S=\{S_1,S_2,S_3,S_4,S_5, \ldots S_n\}$

- Se arată modul de reprezentare în coada Q considerând **solicitările de a intra în secţiunea critică** în ordinea $S_3S_2S_4$.

α= .. Pr_{I_3} Pr_{I_2} Pr_{I_4} PrF_3 SI_3 SF_3 Ur_3 PrF_2 SI_2 SF_2 Ur_2 PrF_4 SI_4 SF_4 Ur_4 ..



p	0				
Q[0]	0				
Q[1]					
Q[2]					
Q[3]					
Q[4]					

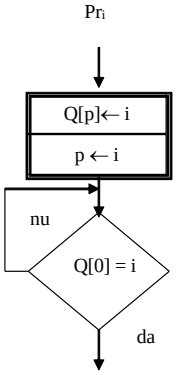
p	0	3			
Q[0]	0	3			
Q[1]					
Q[2]					
Q[3]					
Q[4]					

p	0	3	2		
Q[0]	0	3			
Q[1]					
Q[2]					
Q[3]	2				
Q[4]					

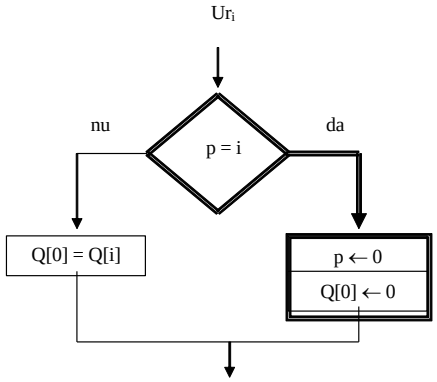
p	0	3	2	4	
Q[0]	0	3			
Q[1]					
Q[2]	4				
Q[3]	2				
Q[4]					

p	0	3	2	4	
Q[0]	0	3 S_3	2		
Q[1]					
Q[2]	4				
Q[3]	2				
Q[4]					

p	0	3	2	4	0
Q[0]	0	3	2 S_2	4	0
Q[1]					
Q[2]	4				
Q[3]	2				
Q[4]					



a) Sarcina Pr_i



b) Sarcina Ur_i

Initial p = 0 şi Q[0] = 0.

Exemplu de interferenţă care conduce la o funcţionare incorectă:

Să presupunem că p=Q[0]=0 şi că Pr_j începe execuţia.

- **Situația 1.** După ce Pr_j se execută Q[0]←j şi înainte de a executa p←j, o sarcină Pr_k cu viteza mai mare în execuţie (pe un procesor mai rapid) execută ambele operaţii **Q[p] ←k ; p ← k**

S_k se va executa imediat, dar cererea pentru Sj s-a pierdut deoarece Q[0] a fost rescris de Pr_k.

Considerând *cele două operaţii indivizibile* **Q[p] ←i ; p ← i**

se elimină astfel de interferenţe.

- **Situația 2.** Să presupunem că p a fost testat de Ur_i şi a fost găsit egal cu i (p=i).

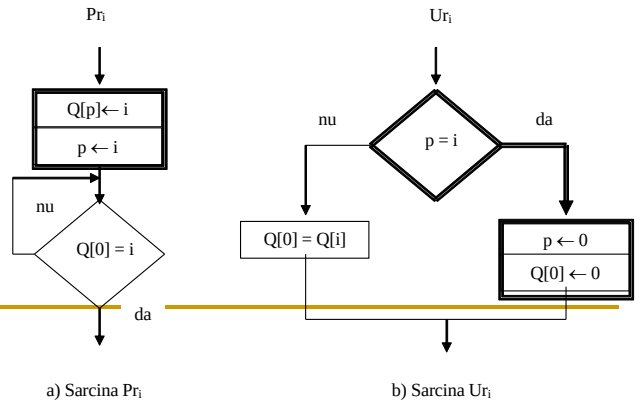
Inainte ca Ur_i să şteargă Q[0] şi p **Q[0] ←0 ; p ← 0** se execută sarcina Pr_k.

Înainte de a se testa egalitatea Q[0]=k în cadrul sarcinii Pr_k, Ur_i se execută mai departe trecând pe Q[0] şi p în zero.

Este evient că Pr_k nu se va executa niciodată,

deci operaţiile implicate de Ur_i

trebuie să fie indivizibile.



- În procedura de realizare a excluderii mutuale arătate anterior **primitiva de aşteptare** Pr_i ocupă procesorul, ceea ce duce la utilizarea ineficientă a acestuia în sisteme în care un număr mare de sarcini intră în competiţie pentru resursa R.
- În acelaşi timp apare o puternică interferenţă la memorie prin accesul la variabilele globale; operaţiile indivizibile fiind destul de lungi.
- Soluţia de excludere mutuală prezentată are marele dezavantaj că necesită execuţia unor operaţii indivizibile complicate, ceea ce de cele mai multe ori nu este posibil prin arhitectura procesoarelor sau chiar dacă este posibil, duce la scăderea performanţelor sistemului.

În continuare vom prezenta o suită de algoritmi, în ordinea dezvoltării lor, care presupun că singurele operaţii indivizibile sunt citirea şi scrierea unei locaţii de memorie.

- Dacă există mai multe cereri de citire sau scriere, ele se vor executa serial, într-o ordine oarecare, nespecificată.
- Prezentarea acestor algoritmi se va face, pe cât posibil, în forma în care au fost publicaţi, desprinderea sarcinilor Pr_i şi Ur_i fiind banală.

Algoritmi de excludere mutuală în mediu centralizat

- **Algoritmul lui Dekker**
- Algoritmul lui Dijkstra
- Algoritmul incorect al lui Hyman
- Algoritmul lui Knuth
- Algoritmul lui De Brujin
- Algoritmul lui Eisenberg şi Mac Quire
- **Algoritmul lui Peterson**
- Algoritmul lui Burns

Algoritmul lui Dekker

- Algoritmul lui Dekker se aplică pentru 2 sarcini S_0 şi S_1 care partajază variabilele:

```
wants_to_enter : array of 2 booleans
turn : integer
wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0 // or 1
```

Semnificaţia acestor variabile este următoarea:

`wants_to_enter[] ← false` S_i nu doreşte să intre în secţiunea critică.

`wants_to_enter[] ← true` S_i doreşte să intre în secţiunea critică.

`turn = i;` Este rândul lui S_i să intre în secţiunea critică.

- Prin variabilele `wants_to_enter` sarcinile îşi dispută accesul la secţiunea critică.
- Dacă ele doresc simultan să intre în secţiunea critică, vor utiliza variabila comună turn pentru a arbitra această situaţie de simultaneitate.

```
variables
wants_to_enter : array of 2 booleans
turn : integer
wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0 // or 1
```

S0:

```
wants_to_enter[0] ← true
while wants_to_enter[1] {
  if turn ≠ 0 {
     wants_to_enter[0] ← false
     while turn ≠ 0 {
       // busy wait
     }
     wants_to_enter[0] ← true
   }
}

// critical section
...
turn ← 1
wants_to_enter[0] ← false
// remainder section
```

S1:

```
wants_to_enter[1] ← true
while wants_to_enter[0] {
  if turn ≠ 1 {
     wants_to_enter[1] ← false
     while turn ≠ 1 {
       // busy wait
     }
     wants_to_enter[1] ← true
   }
}

// critical section
...
turn ← 0
wants_to_enter[1] ← false
// remainder section
```

https://en.wikipedia.org/wiki/Dekker%27s_algorithm

Comentarii

- Sarcinile indică intenția de a intra în secțiunea critică care este testată de bucla exterioară while.
- Dacă celălalt proces nu a marcat intenția, secțiunea critică poate fi introdusă în siguranță, indiferent de tura curentă.
- Excluderea reciprocă va fi în continuare garantată, deoarece niciuna dintre sarcini nu poate intra in zona critică înainte de a-și seta indicatorul (ceea ce implică cel puțin o sarcina va intra în bucla while).
- Alternativ, dacă variabila celuilalt proces a fost setată, bucla while este activă și variabila turn va stabili cine are voie să devină critic.
- Sarcinile fără prioritate își vor retrage intenția de a intra în secțiunea critică până când li se va acorda din nou prioritate (bucla interioară while).
- Sarcinile cu prioritate vor intra în secțiunea lor critică.

```
S0:
wants_to_enter[0] ← true
while wants_to_enter[1] {
  if turn ≠ 0 {
     wants_to_enter[0] ← false
     while turn ≠ 0 {
       // busy wait
     }
     wants_to_enter[0] ← true
   }
}

// critical section
...
turn ← 1
wants_to_enter[0] ← false
// remainder section
```

Comentariu

- Un avantaj al acestui algoritm este că nu necesită instrucțiuni speciale de testare și setare (citire / modificare / scriere atomică) și, prin urmare, este extrem de portabil pe arhitecturi de mașini.
- Un dezavantaj este că este limitat la două sarcini și folosește așteptarea ocupată în loc de suspendarea procesului. (Utilizarea așteptării ocupate sugerează că sarcinile ar trebui să petreacă o cantitate minimă de timp în secțiunea critică.)
- Sistemele de operare moderne oferă primitive de excludere reciprocă, care sunt mai generale și mai flexibile decât algoritmul lui Dekker.
- Cu toate acestea, în absența unei dispute reale între cele două sarcini, intrarea și ieșirea din secțiunea critică sunt extrem de eficiente atunci când este utilizat algoritmul lui Dekker.
- Multe procesoare moderne își execută instrucțiunile ”out of order” Acest algoritm nu va funcționa pe mașinile cu procesoare ”out of order”

Algoritmul lui Peterson

- Algoritmul lui Peterson oferă o descriere algoritmică bună a rezolvării problemei secțiunilor critice și ilustrează unele dintre complexitățile implicate în proiectarea software-ului care răspunde cerințelor de excludere reciprocă, progres și așteptare limitată.
- Intr-un articol cu titlu destul de sugestiv pentru ce o să urmeze, "Miths about the Mutual Exclusion Problem" Peterson a elaborat un algoritm extrem de simplu şi elegant.
- Sunt utilizate doua variabile globale,

  flag - care arată poziţia fiecărui sarcini faţă de secţiunea critică şi turn - care rezolvă situaţiile de simultaneitate.

Algoritmul lui Peterson este:

```
bool flag[2] = {false, false};
int turn;
```

```
S0:      flag[0] = true;
S0_gate: turn = 1;
    while (flag[1] == true && turn == 1)
        {
            // busy wait
        }
        // critical section
        ...
        // end of critical section
        flag[0] = false;
```

```
S1:      flag[1] = true;
S1_gate: turn = 0;
while (flag[0] == true && turn == 0)
        {
            // busy wait
        }
        // critical section
        ...
        // end of critical section
        flag[1] = false;
```

https://en.wikipedia.org/wiki/Peterson%27s_algorithm

Comentariu

- Algoritmul îndeplinește cele trei criterii esențiale pentru a rezolva problema secțiunii critice, cu condiția ca modificările variabilelor turn, flag [0] și flag [1] **să se propage imediat și atomic.**
- Cele trei criterii sunt excluderea reciprocă, progresul și așteptarea limitată.

Excludere mutuala

- S0 și S1 nu pot fi niciodată în secțiunea critică în același timp:
- Dacă S0 se află în secțiunea critică, atunci flag [0] este adevărat.
  - În plus, fie flag [1] este fals (adică S1 și-a părăsit secțiunea critică),

    fie turn=0 (adică S1 încearcă acum să intre în secțiunea critică, dar așteaptă),
- Dacă ambele sarcini sunt în secțiunile lor critice, atunci starea trebuie să îndeplinească flag [0] și flag [1] și turn = 0 și turn = 1.
  - Nici o stare nu poate satisface atât turn = 0, cât și turn = 1, deci nu poate exista o stare în care ambele sarcini se află în secțiunile lor critice.
- Deoarece turn poate lua una din cele două valori, poate fi înlocuit cu un singur bit, ceea ce înseamnă că algoritmul necesită doar trei biți de memorie pentru implementare sicronizare.

Progresul: o sarcină nu poate intra imediat din nou în secțiunea critică dacă celălaltă sarcină și-a setat flag-ul pentru a spune că ar dori să intre în secțiunea sa critică.

Așteptarea limitată : în algoritmul lui Peterson, o sarcină nu va aștepta niciodată mai mult de o tură pentru intrarea în secțiunea critică.

Algoritmul lui Peterson pentru mai multe sarcini

- Algoritmul de filtrare generalizează algoritmul lui Peterson la N> 2 procese.
- În loc de un flag boolean, necesită o variabilă întreagă pe sarcină, stocată într-un singur registru atomic care este scrisă de o singura sarcină și citită de mai multe sarcini (SWMR) și N-1 variabile suplimentare în registre similare.
- Registrele pot fi reprezentate în pseudocod sub formă de tablouri:
- level : array of N integers
- last_to_enter : array of N−1 integers
- Variabilele de nivel iau valori de până la N − 1, fiecare reprezentând o „variabilă de așteptare” distinctă înainte de secțiunea critică.
- Sarcinile avansează de la o variabilă la alta, terminând cu variabila în celula N − 1, care este secțiunea critică. Mai exact, pentru a obține o blocare, sarcina Si va executa

```
i ← Si
for ℓ from 0 to N−1 exclusive
    level[i] ← ℓ
    last_to_enter[ℓ] ← i
    while last_to_enter[ℓ] = i and there exists k ≠ i, such that level[k] ≥ ℓ
        wait
```

Pentru a elibera blocarea la ieșirea din secțiunea critică, sarcina Si stabilește nivelul [i] la -1.

Soluţii hardware de excludere mutuală în mediu centralizat pentru sisteme monoprocesor

- În vederea realizării excluderii mutuale se pot prevedea diferite instrucţiuni sau primitive implementate la nivelul arhitecturii maşinii.
- Există o mare diversitate de astfel de soluţii, iar în continuare vor fi considerate doar câteva exemple mai sugestive.
- Într-un sistem monoprocesor pot exista sarcini concurente care îsi dispută accesul la resurse.
- Apare şi un paralelism efectiv între Unitatea Centrală de prelucrare şi subsistemul de I/E care poate antrena şi alte subansamble, cum ar fi:
  - DMA, canale de I/E, module de memorie,
- Relaţia între UCP şi celelalte subansamble este de tip "master-salve".

- În special datorită sistemului de întreruperi, într-un sistem monoprocesor apare o întreţesere a execuţiei sarcinilor, iar anumite secţiuni ale acestora pot fi regiuni critice care trebuie executate în mod exclusiv, fără a fi întrerupte.
- Pentru a realiza aceasta trebuie ca sistemul de intrerupere SIntr să poată fi dezactivat / activat sau anumite nivele să fie mascate / demascate.

  Astfel avem:

<început sarcina>;	<început sarcina>;
dezactivare_SIntr;	*mascare_într;*
<secţiune critică>;	<sectiune critică>;
activare_SIntr;	*demascare_într;*
<rest sarcina>;	<rest sarcina>;

- Desigur că ar exista o soluţie pentru excluderea mutuală prin evitarea acesteia, prin eliminarea concurenţei, dar aceasta nu este acceptabilă datorită degradării performanţelor.

Soluţii hardware de excludere mutuală în mediu centralizat pentru sisteme multiprocesor

Pentru sistemele multiprocesor realizarea excluderii mutuale depinde şi mai mult de arhitectura sistemului. În principiu, trebuie realizată execuţia indivizibilă a unor secvenţe de acces la memoria comună.

Dintre soluţiile posibile amintim:

- instrucţiuni atomice de tip XCHG care schimbă între ele două locaţii de memorie sau un registru general şi o locaţie de memorie;
- instrucţiuni atomice de tip TAS (Test And Set) sau RMW (Read Modify Write) care citesc o celulă de memorie, o modifică şi actualizează noua valoare;
- instrucţiuni atomice de tip Lock, Unlock;
- instrucţiuni atomice de incrementare, decrementare;
- instrucţiuni atomice de înlocuire şi adunare.

Excludere mutuală cu instrucțiuni XCHG

- Instrucţiunea XCHG(r,m) interschimbă, ca o operaţie atomică, conţinutul registrului r cu celula de memorie m (semafor).
  - deci r_i este o variabilă locală a sarcinii S_i şi este iniţializată cu 0, iar
  - m este o variabilă globală, comună tuturor sarcinilor care este iniţializată cu 1.
- Ambele variabile pot lua valorile 0, 1.

Algoritmul pentru S_i va fi:

<început sarcina>;

repetă

XCHG(r_i,m);

până_când r_i;

<secţiune critică>;

XCHG(r_i,m);

<rest sarcina>;

- Numai procesul care găseşte m=1 va intra în secţiunea critică şi va trece în zero. La ieşire din secţiunea critică va returna pe m la valoarea 1 pentru a permite celorlalte sarcini să intre în secţiunea critică. (m=1 este o conventie, in unele sisteme se considera m=0)
- Când o sarcina Si intră în secţiunea critică, r_i=1.
- Deci pentru a realiza excluderea mutuală trebuie ca în orice moment să se verifice relaţia:

$$m + \sum_{i=0}^{n-1} r_i = 1,$$

- Microprocesorul x86 include în setul de instrucţiuni o astfel de instrucţiune care prefixată cu LOCK realizează funcţia necesară pentru implementarea excluderii mutuale.

LOCK_Si

```
AST:    MOV     AL,0
        LOCK
        XCHG    AL,m
        TEST    AL,AL
        JZ      AST
```

se intra in executia sarcinii , zona critica

UNLOCK_Si

```
        MOV     m,1
```

LOCK_Sj

```
AST:    MOV     AL,0
        LOCK
        XCHG    AL,m
        TEST    AL,AL
        JZ      AST
```

se intra in executia sarcinii , zona critica

UNLOCK_Sj

```
        MOV     m,1
```

Excluderea mutuală cu instrucțiuni TAS

- Instrucţiunea TAS(m) testează conţinutul lui m şi
  - dacă este 0, m ← 1 şi reîntoarce rezultat adevărat.
  - dacă m este 1, lasă conţinutul nemodificat şi întoarce rezultat fals.
- Variabila globală m este iniţializată cu 0.

Pentru S_i, algoritmul este:

<început sarcina>;

repetă

până_când TAS(m);

<secţiune critică>;

m ← 0;

<rest sarcina>

- Având în vedere că instrucţiunea TAS(m) se execută strict serial, numai sarcina care găseşte m=0 va intra în secţiunea critică, şi va face atribuirea m ← 1.
- La ieşirea din secţiunea critică va face m ← 0 pentru a permite celorlalte sarcini să intre în secţiunea critică.

- Microprocesorul x68000 include în setul de instrucţiuni o instrucţiune TAS indivizibilă, pentru excludere mutuală în structuri multiprocesor.

Excludere mutuală cu instrucțiuni LOCK, UNLOCK

Primitiva LOCK(m)

este echivalentă cu:

{

repetă nimic *cât_timp* m=0;

m ← 0;

}

Primitiva UNLOCK(m)

este echivalentă cu:

{

m ← 1;

}

Pentru familia x86

LOCK_Si

```
AST:    MOV     AL,0
        LOCK
        XCHG    AL,m
        TEST    AL,AL
        JZ      AST
```

se intra in executia sarcinii , zona critica

UNLOCK_Si

```
        MOV     m,1
```

- Iniţializând variabila m cu 1, algoritmul de excludere mutuală este:

<început sarcina>

LOCK(m);

<secţiune critică>;

UNLOCK(m);

<rest sarcini>;

```
LOCK_Si
AST:    MOV     AL,0
        LOCK
        XCHG    AL,m
        TEST    AL,AL
        JZ      AST
```

se intra in executia sarcinii , zona critica

```
UNLOCK_Si
        MOV     m,1
```

- La unele procesoare, această funcţie este realizată de instrucţiuni denumite TSB (Test and Switch Branch).