

Analiza situațiilor de blocare

- Un sistem care este format din sisteme de sarcini combinate paralel $\mathbf{C} = \mathbf{C}_1 \mid \mathbf{C}_2 \mid \dots \mid \mathbf{C}_n$

este *blocat* dacă evoluția 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 \mathbf{C} .

- O resursă deținută de o sarcină **neinterruptibilă** poate fi eliberată numai de sistemul de sarcini care o utilizează la un moment dat.

- Pentru studiul situațiilor de blocare vom privi sistemul fizic ca o colecție de resurse R de tip:

$R = R_1 R_2 \dots R_m$, fiecare tip conținând:

$W = W_1 W_2 \dots W_m$ exemplare

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

Exemplu de resurse R_i :

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

- Să considerăm un exemplu simplu de blocare:

Fie două sisteme \mathbf{C}_1 și \mathbf{C}_2 și câte o resursă de tip R_1 și R_2 . Evoluția întregului sistem poate fi reprezentată în spațiul cu două dimensiuni definit **spațiul de evoluție**.

Evoluția sistemului $\mathbf{C} = \mathbf{C}_1 \mid \mathbf{C}_2$ este reprezentată de succesiunea de puncte $(x_1, y_1) (x_2, y_2) \dots (x_k, y_k) \dots$ în acest spațiu

Presupunem că originea este în punctul $(0,0)$.

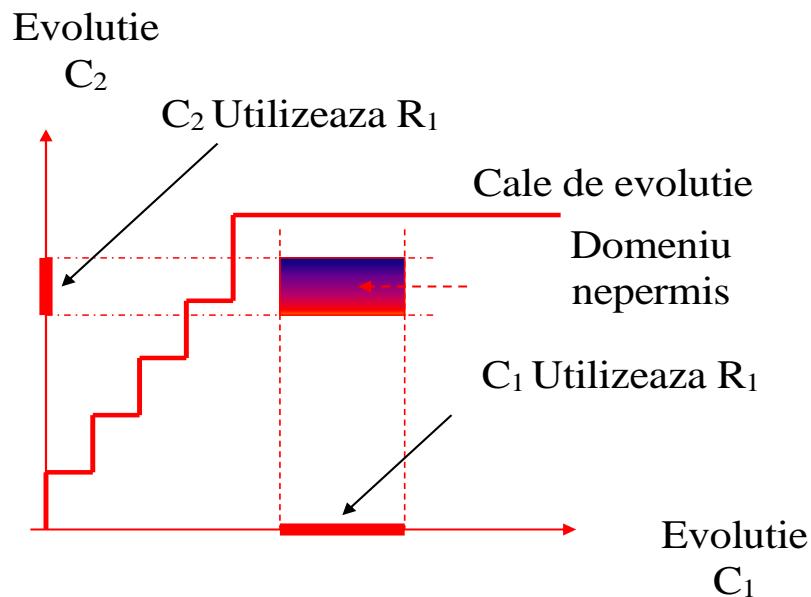
Evoluția este ireversibilă deci

$x_k \leq x_{k+1}$, $y_k \leq y_{k+1}$ și

Nu orice cale de evoluție, este realizabilă.

Evoluția în care \mathbf{C}_1 și \mathbf{C}_2 astfel să utilizeze simultan aceeași resursă R_1 este nepermisă

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



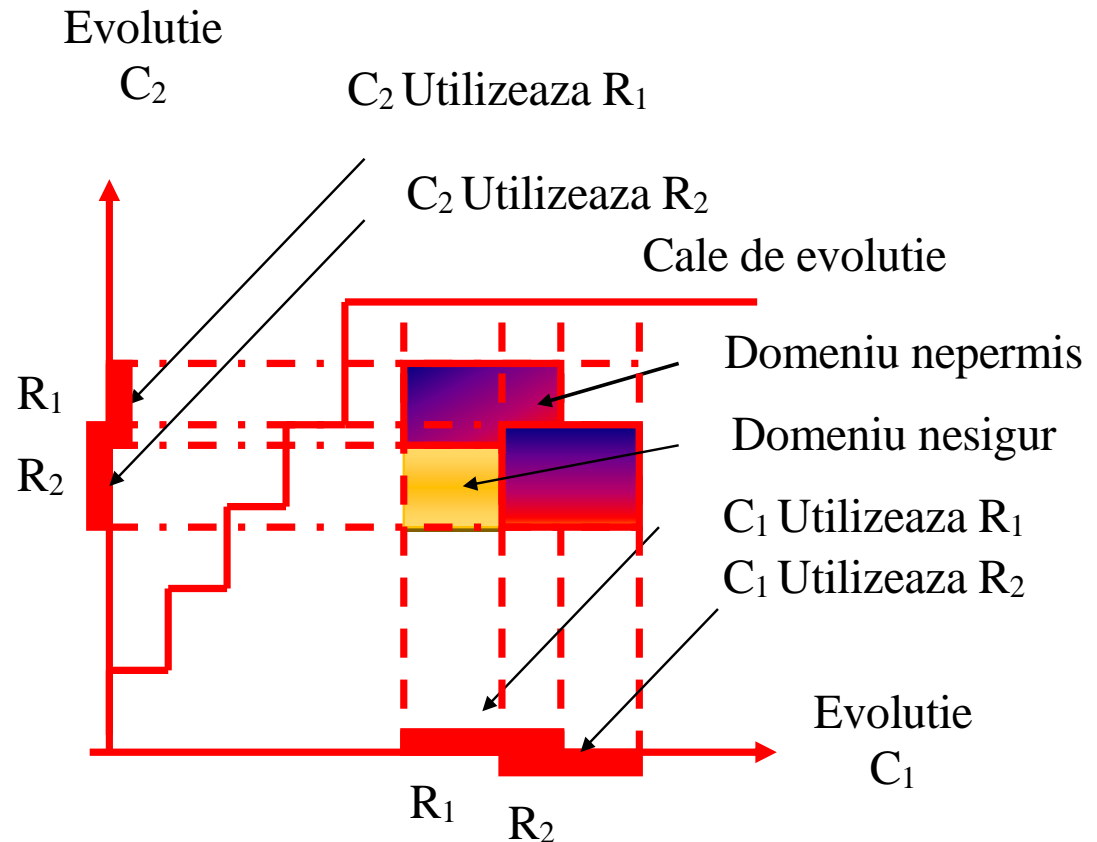
Exemplu de utilizare a doua resurse

Fie utilizarea resurselor R_1 și R_2 de către C_1 și C_2 .

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

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

Datorită ireversibilității procesului de evoluție se va ajunge în mod sigur la blocarea sistemului.



- **Blocarea** este definită ca o stare de *așteptare* circulară nerezolvabilă din cauza condițiilor de excludere mutuală și neîntreruptibilitate.
- Pentru a exista o cale de evoluție care conduce la blocare sunt necesare următoarele condiții:
 - **Utilizare exclusivă:** Fiecare sistem de sarcini dorește controlul exclusiv asupra resursei pe care o utilizează;
 - **Neîntreruptibilitate:** O resursă nu este eliberată până la terminarea completă a utilizării acesteia;
 - **Așteptare circulară:** Fiecare sistem de sarcini ține ocupate resurse în timp ce așteaptă ca altele să elibereze resurse pe care să le preia.

- Relativ la blocare se pun trei probleme:

1. Prevenirea

Una sau mai multe din condițiile care conduc la blocare sunt înlăturate printr-o proiectare adecvată sau prin restricții în utilizarea resurselor.

2. Detectarea și înlăturarea

Sistemul de operare și planificatorul supraveghează evoluția tuturor sistemelor de sarcini luând măsuri speciale de corecție a evoluției la detectarea situațiilor de blocare.

3. Evitarea

Pe baza unor informații preliminare despre cerințele de resurse, sistemul de operare / planificatorul dirijează evoluția pentru evitarea zonelor nesigure.

Considerând că fiecare tip de resursă conține 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 reprezintă cererea de resurse.

Arcele pot fi etichetate cu necesarul de resurse.

- Arcele sunt orientate și sunt definite astfel:

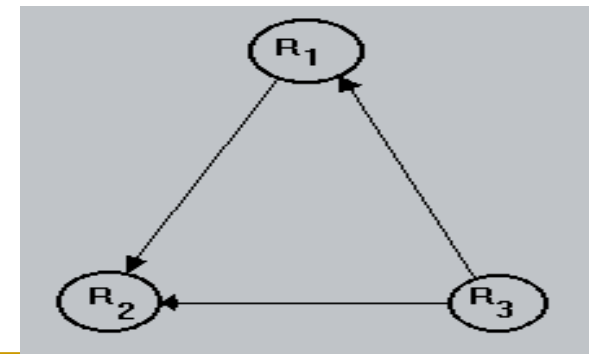
dacă la un moment dat, după evenimentul a_k , sistemul **C** ține ocupată resursa R_i în timp ce așteaptă preluarea lui R_j , graful conține un arc orientat de la nodul R_i la nodul R_j .

- În exemplul următor numărul de tipuri de resurse este $n=3$;

Un proces utilizează R_1 și o cere pe R_2 ,

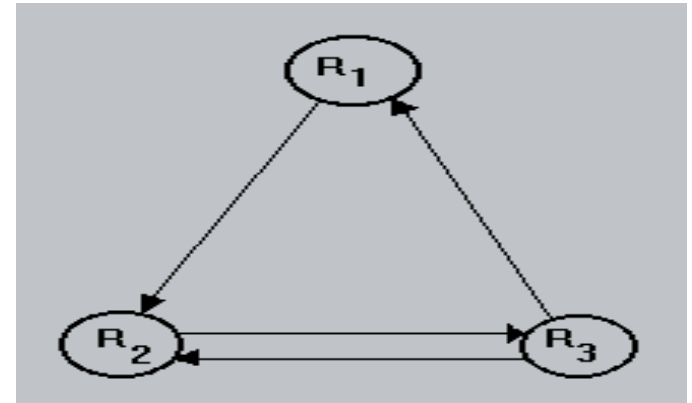
Alt proces utilizează pe R_3 și cere pe R_1 și R_2 .

Graf de cereri fara bucle



Dacă cele trei condiții de blocare sunt posibile (operative) condiția necesară și suficientă pentru blocare este existența unei bucle (circuit, ciclu) în graful de cereri.

Exemplu , de graf de cereri în care se realizează condiția de așteptare 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 depășească, ca număr, w_i .
- Un arc de la R_i la R_j arată că există cel puțin un proces care cere una sau mai multe resurse de tip R_j și deține una sau mai multe resurse de tip R_i .
- În cazul acesta o buclă în graful de cereri reprezintă o condiție necesară dar nu și suficientă pentru existența situației 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 puțin una din condițiile necesare nu este îndeplinită.
- Prima din cele trei condiții (excluderea mutuală) este uneori impusă de restricții fizice privind accesul la resurse nepartajabile și deci nu poate fi eliminată pentru orice fel de resurse.

Pentru celelalte condiții vom enumera câteva posibilități:

- Dacă o sarcină care utilizează o resursă, cere o altă resursă *trebuie* să o elibereze pe prima și dacă este necesară în continuare să o ceară din nou împreună cu resursele adiționale de care are nevoie (**interruptibilitate**).
- Fiecare proces să ceară toate resursele de care are nevoie, putând continua numai după ce le are pe toate (**așteptare cu resurse**).
- **Ordonarea liniară** a tipurilor de resurse pentru toate sarcinile. Dacă un proces utilizează resurse de tip R_i , la un moment dat, el 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.

*Observație:

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

Detectarea și 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 și
- o *procedură* de detectare a circuitelor (buclelor) în acest graf.
- Pentru cazul general, la fiecare moment de timp t , corespunzător secvenței de execuție $\alpha = a_1 a_2 \dots a_k$, se definesc
- $x_{ij}[k]$ reprezintă numărul de resurse de tip R_j alocate, respectiv și
- $y_{ij}[k]$ cerute de C_i , după evenimentul a_k din α , unde:
$$C = C_1 | C_2 | \dots | C_n, \quad 1 \leq i \leq n \quad \text{iar} \quad 1 \leq j \leq m,$$

 m fiind numărul de tipuri de resurse

Se obțin astfel:

$$X[k] = \begin{bmatrix} \mathbf{X}_1(k) \\ \mathbf{X}_2(k) \\ \cdot \\ \cdot \\ \mathbf{X}_n(k) \end{bmatrix} \text{matrice de alocare}$$

$$Y[k] = \begin{bmatrix} \mathbf{y}_1(k) \\ \mathbf{y}_2(k) \\ \cdot \\ \cdot \\ \mathbf{y}_n(k) \end{bmatrix} \text{matrice de cereri}$$

cu

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

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

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

Să notăm cu:

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

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

$$v_j(k) = w_j - \sum_{i=1}^n x_{ij}[k], \quad 1 \leq j \leq m$$

- Pentru doi vectori X și Y , relația $X \leq Y$ este adevărată, **dacă și numai dacă** relația este adevărată pentru fiecare pereche de elemente corespondente din X și Y .
- Utilizând informația conținută în starea s_k dată de perechea de matrici $[X[k], Y[k]]$ vom elabora un algoritm de detectare a blocării.

Definiție:

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

$$\mathbf{C}_i = S_i[1] S_i[2] \dots S_i[l_i], \quad l_i \geq 1.$$

Fie

$\alpha = a_1 a_2 \dots a_k$ o secvență de execuție parțială a lui \mathbf{C} iar

$\sigma = s_0 s_1 \dots s_k$ secvența parțială de stări, corespunzătoare.

Dacă există o mulțime nevidă D de indici ai lui \mathbf{C} astfel că pentru orice i din D **nu este** adevărată relația:

$Y_i[k] \leq V[k] + \sum_{j \notin D} X_j[k]$		
<i>vector de cereri</i>	<i>vector de resurse</i>	<i>vector de resurse deținute de sarcini ce nu aparțin lui D</i>

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 $i \in D$ este blocat.

Exemplu:

Fie C_1 și C_2 două sisteme de sarcini:

$$C_1 = S_1[1], S_1[2], S_1[3] \text{ și } C_2 = S_2[1], S_2[2]$$

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)			

$\alpha = SI_1[1] SF_1[1] SI_2[1] SF_2[1] SI_1[2] SF_1[2] SI_2[2] \dots$ secvența de execuție.

	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 $Y1[k] \leq V[k]$ cât și $Y2[k] \leq V[k]$ astfel că $P1[3]$ și $P2[2]$ nu pot avea loc cu valorile obținute 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 deținute.

Algoritm pentru detectarea blocării

Etapele algoritmului sunt:

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

- Algoritmul se bazează direct pe definiția de mai sus și constă din **simularea** execuției sarcinilor până când rămâne un set de sisteme de sarcini C_i pentru care, în final, resursele disponibile nu sunt suficiente sau mulțimea de indici D devine vidă.
- În cel mai defavorabil caz algoritmul dat analizează la pasul i , întreaga mulțime de sisteme de sarcini din D , în număr de $(n-i+1)$.

Timpul de execuție al algoritmului este în acest caz proporțional cu

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

- Printr-o aranjare a structurilor de date și o reprezentare mai elaborată a stărilor resurselor se poate obține un algoritm cu timpul de execuție proporțional cu $O(m.n)$.
 - Complexitatea acestui algoritm face ca implementarea lui să fie justificată numai în sisteme în care apariția unei blocări este intolerabilă.
 - În multe sisteme se utilizează o soluție 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 situație de blocare și sistemul este scos din blocare și trecut într-o anumită stare, predeterminată din care continuă evoluția.
 - Implementarea metodelor de detectare a situațiilor de "time out" se face atât la nivelul mașinii de bază cât și la nivelul sistemului de operare.
-

Evitarea blocării

- Pentru a analiza problema evitării blocării se presupune că o sarcină S constă dintr-o succesiune de pași.
- Pe parcursul execuției unui pas utilizarea resurselor rămâne constantă și aceasta implică achiziționarea resurselor necesare pentru acest pas în plus față de cele primite la pasul precedent.
- Urmează apoi o perioadă de execuție în care cerințele de resurse nu se schimbă.
- În final când se termină execuția, toate resursele care nu sunt cerute la pasul următor sunt eliberate și trecute într-o colecție de resurse disponibile.

- Fie $\sigma = s_0 s_1 \dots s_k$ secvența de stări corespunzătoare secvenței de execuție $\alpha = a_1 \dots a_k$.
- Dacă există cel puțin o secvența de execuție 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 evoluția următoare.
- Cu alte cuvinte s_k este *sigură* dacă utilizând resursele disponibile $V[k]$ și cele care vor fi eliberate de pașii care se execută la momentul respectiv, este posibil să găsim o secvență validă de pași încă neinițiați, pentru sarcinile inițiate dar neterminate astfel încât toate sarcinile din sistem să poată continua execuția spre finalizare.
- **Starea inițială** în care încă nu s-au alocat resurse, toate fiind disponibile, este întotdeauna 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 găsește în starea s_k sigură și că există cererile de resurse date de $Q[k]$.
- **Supervizorul** trebuie să determine dacă orice cerere din $Q[k]$ poate fi satisfăcută, $y_{ij}[k] \leq v_j[k]$.
- În general pot fi mai multe astfel de cereri și deci se pot prevedea diferite *criterii de alocare* pentru a stabili care cerere să fie satisfăcută mai întâi, 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 rotație;

Priorități atribuite sarcinilor;

etc.

- Pentru a determina dacă o stare s_k este sigură ca urmare a satisfacerii unor cereri de resurse trebuie căutată o secvență adecvată de pași.
- Pentru aceasta se presupune că resursele disponibile sunt completate cu cele alocate sarcinilor în curs de execuție.
- Aceasta este echivalentă cu presupunerea că primul pas într-o secvență analizată nu începe până când toți pașii în curs de execuție au toate condițiile pentru a se finaliza.

Excluderea mutuală

Definiții. Concepte de bază.

- Într-un sistem de sarcini concurente, acestea pot interacționa în două moduri:
 - indirect, prin disputa pentru aceleași resurse;
 - direct, prin transmitere / recepție de mesaje sau informații de stare.
 - Când se execută sarcini independente evoluția acestora poate fi controlată de către supervisor așa cum am văzut la analiza situațiilor de blocare.
 - Totuși când sarcinile interacționează în mod direct mecanismul de control al evoluției lor trebuie să apară în mod explicit în programele care le definesc.
 - Interacțiunea 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 și 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 inițializată 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 execuție corectă a sarcinilor.
 - Pe de altă parte, deoarece o sarcină inițializează starea unei resurse înainte de a o utiliza, ordinea în care mai multe sarcini o utilizează este indiferentă.
 - Astfel orice soluție pentru excluderea mutuală care implică a priori constrângeri privind ordinea de execuție a sarcinilor este neadecvată.
 - O soluție 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 țină seama numai de convențiile prin care se comunică sistemului de operare cererile și eliberările de resurse.
-

- Vom considera că excluderea mutuală trebuie rezolvată de însuși sistemul de sarcini, utilizând *variabile globale* sau variabile de stare și mesaje pentru comunicația între sarcini în funcție de context: centralizat sau distribuit.
- Sarcinile care trebuie să se execute mutual exclusiv constituie secțiuni critice.
- O soluție corectă pentru excluderea mutuală trebuie să elimine apariția unor rezultate diferite sau blocari în tranzițiile sistemului în condițiile unor timpi de execuție a sarcinilor diferiți, pentru diferite procesoare.
- De exemplu: Să considerăm posibilitatea ca două sarcini S_1 și S_2 care se execută pe procesoare diferite să utilizeze în comun o resursă R.
- S_1 și S_2 utilizează o variabilă globală care specifică disponibilitatea resursei R.

- S_1 testează variabila, găsește R disponibilă și înainte ca S_1 să schimbe valoarea variabilei pentru a arăta că R este ocupată, S_2 testează și ea variabilă și găsește R disponibilă.
- Mai concret S_1 și S_2 care evoluează într-un context centralizat au primit aceeași interpretare, citirea variabilei Semafor, incrementarea valorii citite și scrierea noii valori.

Fie, de exemplu, secvența $SI_1 SI_2 SF_1 SF_2$, deci

- 1°. S_1 citește variabila Semafor;
- 2°. S_2 citește variabila Semafor;
- 3°. S_1 incrementează și scrie noua valoare;
- 4°. S_2 incrementează și scrie noua valoare.

În urma acestei secvențe de execuție, Semafor primește valoarea Semafor + 1 în loc de Semafor + 2.

- Rezolvarea cererilor de acces la memorie pentru UCP și DMA reprezintă un exemplu de excludere mutuală la nivel hardware.
 - Scrierea unor texte la o imprimantă comună într-o rețea locală de calculatoare constituie un exemplu de excludere mutuală, la nivel de aplicații.
 - Accesul la o înregistrare dintr-un fișier pentru a fi citită și modificată de mai multe sarcini constituie de asemenea un exemplu de excludere mutuală 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.
-

Definiție:

Fie $\alpha = a_1, a_2, \dots, a_k$ o secvență de execuție parțială a sistemului de sarcini $C=(S, <)$. Sarcinile S și S' sunt **mutual exclusive** în α dacă și numai dacă cel mult una din ele este activă după orice prefix a lui α .

Pentru a implementa excluderea mutuală vom considera că o soluție este corectă dacă cel mult o sarcină este în secțiunea critică la un moment dat și sunt satisfacute următoarele condiții adiționale:

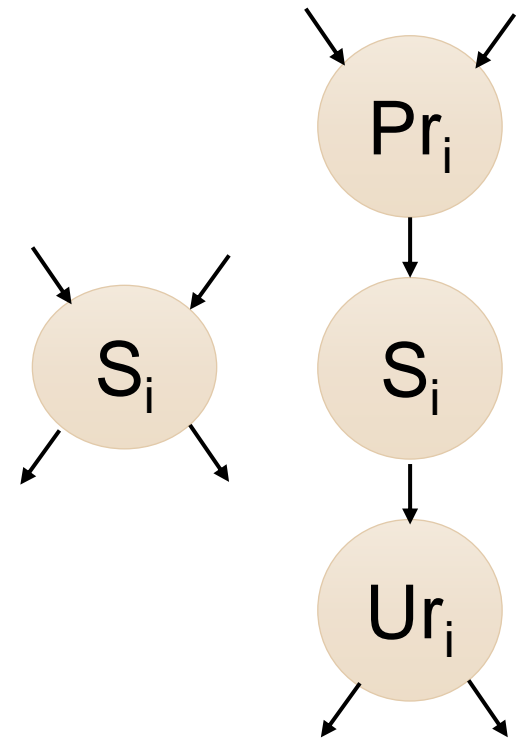
- Nu se face nici o presupunere relativă la activitatea și numărul de procesoare pe care se execută sistemul de sarcini.
Desigur, așa cum s-a arătat și mai sus, se consideră ca citirea și scrierea unei variabile de memorie sunt operații atomice, indivizibile. Lansarea simultană a unor astfel de operații va avea ca efect execuția lor secvențială, într-o ordine oarecare, necunoscută apriori.
- Soluția 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 secțiunii critice nu poate bloca altă sarcină de la intrarea în secțiunea critică.
- Două sarcini care doresc să intre în secțiunea critică nu pot fi într-o buclă de așteptare reciprocă.
Deci dacă se eliberează secțiunea critică și există sarcini în așteptare, este exclusă posibilitatea ca nici un proces să nu intre în secțiunea critică. În plus, o sarcină care dorește să intre în secțiunea critică nu poate fi în stare de așteptare un timp nedefinit.
- Toate sarcinile trebuie tratate la fel, fără priorități sau alte privilegii.

Algoritmi de excludere mutuală în mediu centralizat

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

- Dacă cel puțin două sarcini independente ale unui sistem **C** trebuie să utilizeze mutual exclusiv resursa R vom modifica **C** pentru a asigura această cerință.
- Pentru fiecare S_i care utilizează resursa R vom introduce sarcinile Pr_i și 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 și Ur_i vor fi complet specificate.
- Pr_i va fi astfel proiectat încât să poată fi terminat și deci să permită inițierea lui S_i dacă și numai dacă pentru $\forall j \neq i$, Pr_j nu poate fi terminat.
- Funcția 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 și Ur_i astfel ca să nu existe subsecvențe de execuție de forma:

$$PrF_i \quad PrF_j \quad UrF_i$$

vom considera ca variabile globale:

p - o variabilă de tip întreg.

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

Inițial $p=0$ și $Q[i]=0$, $0 \leq i \leq n$.

- **Funcția vectorului Q** este de a memora într-o ordine **FIFO** lista sarcinilor ce așteaptă utilizarea lui R . Acest lucru se realizează prin memorarea indicelui sarcinei care utilizează pe R în Q .
- Astfel, coada de așteptare la resursa R este organizată ca o simplă listă înlănțuită, memorată în Q .
- *Variabila p* conține indicele ultimului element din lista (ultimul element intrat în coada Q).
- Condiția de coadă vidă, considerată ca și condiție inițială este dată de $p=0$ și $Q[0]=0$.

Definirea sarcinilor auxiliare Pr și Ur

Procese auxiliare Pr_i și Ur_i pot fi specificate prin organigramele următoare

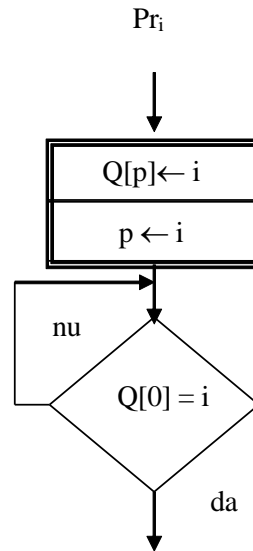
Sarcina Pr_i arată intenția lui S_i de a utiliza resursa R , introduce pe S_i în coada Q și așteaptă într-o buclă până când S_i ajunge în capul listei $Q[0]$.

Sarcina Ur_i înlătură pe S_i din coadă și trece următoarea sarcină pentru execuție în capul listei.

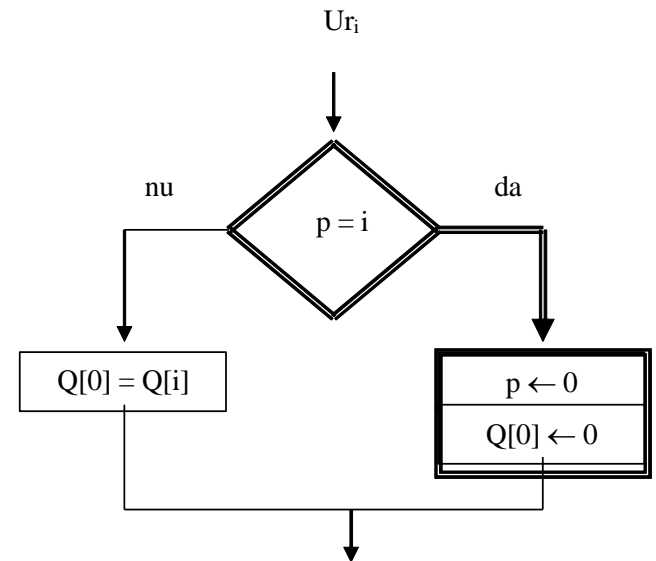
Dacă S_i a fost ultima sarcină introdusă în coadă

($p=i$ când se execută Ur_i)

atunci coada este vidă și se trec în zero p și $Q[0]$.



a) Sarcina Pr_i



b) Sarcina Ur_i

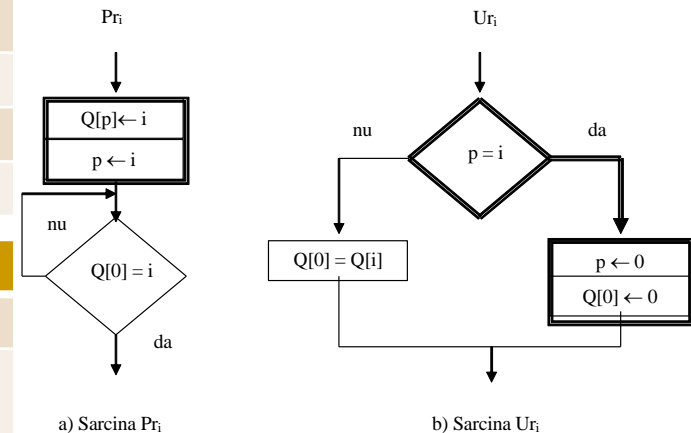
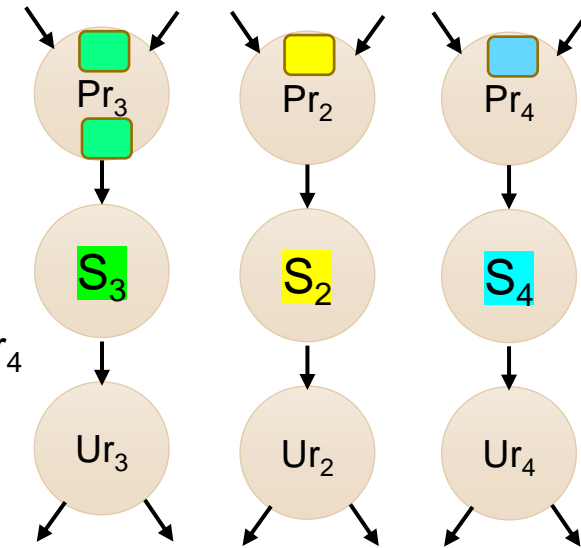
Pentru a putea elimina posibilitatea interferențelor între Pr_i, Pr_j sau Pr_i, Ur_j ce se execută concurrent trebuie ca anumite operații asupra variabilelor globale să **fie indivizibile**, cele în chenar dublu.

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

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

- Se arată modul de reprezentare în coada Q considerând solicitările de a intra în secțiunea critică în ordinea $S_3 S_2 S_4$.

$\alpha = \dots \text{PrI}_3 \text{PrI}_2 \text{PrI}_4 \text{PrF}_3 \text{SI}_3 \text{SF}_3 \text{Ur}_3 \text{PrF}_2 \text{SI}_2 \text{SF}_2 \text{Ur}_2 \text{PrF}_4 \text{SI}_4 \text{SF}_4 \text{Ur}_4$
 ..



Initial $p = 0$ și $Q[0] = 0$.

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	2		
Q[1]					
Q[2]	4				
Q[3]	2				
Q[4]					

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

Exemplu de interferență care conduce la o funcționare incorectă:

Să presupunem că $p=Q[0]=0$ și că Pr_j începe execuția.

- **Situația 1.** După ce Pr_j se execută $Q[0] \leftarrow j$ și înainte de a executa $p \leftarrow j$, o sarcină Pr_k cu viteza mai mare în execuție (pe un procesor mai rapid) execută ambele operații **$Q[p] \leftarrow k ; p \leftarrow k$**

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

Considerând *cele două operații indivizibile* **$Q[p] \leftarrow i ; p \leftarrow i$**

se elimină astfel de interferențe.

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

Înainte ca Ur_i să șteargă $Q[0]$ și p **$Q[0] \leftarrow 0 ; p \leftarrow 0$** 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 $Q[0]$ și p în zero.

Este evident că Pr_k nu se va executa niciodată, *deci operațiile implicate de Ur_i trebuie să fie indivizibile.*

- În procedura de realizare a excluderii mutuale arătate anterior **primitiva de așteptare** P_{r_i} ocupă procesorul, ceea ce duce la utilizarea ineficientă a acestuia în sisteme în care un număr mare de sarcini intră în competiție pentru resursa R.
- În același timp apare o puternică interferență la memorie prin accesul la variabilele globale; operațiile indivizibile fiind destul de lungi.
- Soluția de excludere mutuală prezentată are marele dezavantaj că necesită execuția unor operații indivizibile complicate, ceea ce de cele mai multe ori nu este posibil prin arhitectura procesoarelor sau chiar dacă este posibil, duce la scăderea performanțelor sistemului.

În continuare vom prezenta o suită de algoritmi, în ordinea dezvoltării lor, care presupun că singurele operații indivizibile sunt citirea și scrierea unei locații 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 publicați, desprinderea sarcinilor P_{r_i} și U_{r_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 Bruijn
- Algoritmul lui Eisenberg și Mac Quire
- **Algoritmul lui Peterson**
- Algoritmul lui Burns

Algoritmul lui Dekker

- Algoritmul lui Dekker se aplică pentru 2 sarcini S_0 și 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
```

Semnificația acestor variabile este următoarea:

`wants_to_enter[] ← false` S_i nu dorește să intre în secțiunea critică.

`wants_to_enter[] ← true` S_i dorește să intre în secțiunea critică.

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

- Prin variabilele `wants_to_enter` sarcinile își dispută accesul la secțiunea critică.
- Dacă ele doresc simultan să intre în secțiunea critică, vor utiliza variabila comună `turn` pentru a arbitra această situație 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
```

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 în zona critică înainte de a-și seta indicatorul (ceea ce implică cel puțin o sarcina va intra în bucla while).
- Alternativ, dacă variabila celui alt 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 (buclo interioară while).
- Sarcinile cu prioritate vor intra în secțiunea lor critică.

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 și elegant.
- Sunt utilizate două variabile globale,
flag - care arată poziția fiecărei sarcini față de secțiunea critică și
turn - care rezolvă situațiile 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;
```

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 mutuală

- 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.

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 cu un 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 S_i va executa
 $i \leftarrow S_i$
for ℓ from 0 to $N-1$ exclusive
 $level[i] \leftarrow \ell$
 $last_to_enter[\ell] \leftarrow i$
 while $last_to_enter[\ell] = i$ and there exists $k \neq i$, such that $level[k] \geq \ell$
 wait
Pentru a elibera blocarea la ieșirea din secțiunea critică, sarcina S_i stabilește nivelul $[i]$ la -1 .

Soluții hardware de excludere mutuală în mediu centralizat pentru sisteme monoprocesor

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

- În special datorită sistemului de întreruperi, într-un sistem monoprosesor apare o întrețesere a execuției sarcinilor, iar anumite secțiuni ale acestora fiind regiuni critice care trebuie executate în mod exclusiv, fără a fi întrerupte.
- Pentru a realiza aceasta trebuie ca sistemul de întrerupere SIntr să poată fi dezactivat / activat sau anumite nivele să fie mascate / demascate.

Astfel avem:

<început sarcina>;

dezactivare_SIntr;

<secțiune critică>;

activare_SIntr;

<rest sarcina>;

<început sarcina>;

mascare_într;

<secțiune critică>;

demascare_într;

<rest sarcina>;

- Desigur că ar exista o soluție pentru excluderea mutuală prin evitarea acesteia, prin eliminarea concurenței, dar aceasta nu este acceptabilă datorită degradării performanțelor.

Soluții hardware de excludere mutuală în mediu centralizat pentru sisteme multiprocesor

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

Dintre soluțiile posibile amintim:

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

Excludere mutuală cu instrucțiuni XCHG

- Instrucțiunea $XCHG(r,m)$ interschimbă, ca o operație atomică, conținutul registrului r cu celula de memorie m (semafor).
 - deci r_i este o variabilă locală a sarcinii S_i și este inițializată cu 0, iar
 - m este o variabilă globală, comună tuturor sarcinilor care este inițializată 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 ;

<secțiune critică>;

$XCHG(r_i,m);$

<rest sarcina>;

- Numai procesul care găsește $m=1$ va intra în secțiunea critică și va trece în zero. La ieșire din secțiunea critică va returna pe m la valoarea 1 pentru a permite celorlalte sarcini să intre în secțiunea critică. ($m=1$ este o convenție, în unele sisteme se considera $m=0$)
- Când o sarcină S_i intră în secțiunea critică, $r_i=1$.
- Deci pentru a realiza excluderea mutuală trebuie ca în orice moment să se verifice relația:

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

- Microprocesorul x86 include în setul de instrucțiuni o astfel de instrucțiune care prefixată cu LOCK realizează funcția necesară pentru implementarea excluderii mutuale.

LOCK_ S_i

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

se intra in executia sarcinii , zona critica

UNLOCK_ S_i

```
MOV     m,1
```

LOCK_ S_j

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

se intra in executia sarcinii , zona critica

UNLOCK_ S_j

```
MOV     m,1
```

Excluderea mutuală cu instrucțiuni TAS

- Instrucțiunea TAS(m) testează conținutul lui m și
 - dacă este 0, $m \leftarrow 1$ și reîntoarce rezultat adevărat.
 - dacă m este 1, lasă conținutul nemodificat și întoarce rezultat fals.
- Variabila globală m este inițializată cu 0.

Pentru S_i , algoritmul este:

<început sarcina>;

repetă

până_când TAS(m);

<secțiune critică>;

$m \leftarrow 0$;

<rest sarcina>

- Având în vedere că instrucțiunea TAS(m) se execută strict serial, numai sarcina care găsește $m=0$ va intra în secțiunea critică, și va face atribuirea $m \leftarrow 1$.
 - La ieșirea din secțiunea critică va face $m \leftarrow 0$ pentru a permite celorlalte sarcini să intre în secțiunea critică.
-
- Microprocesorul x68000 include în setul de instrucțiuni o instrucțiune TAS indivizibilă, pentru excludere mutuală în structuri multiprocesor.

Excludere mutuală cu instrucțiuni LOCK, UNLOCK

Pentru familia x86

Primitiva **LOCK(m)**
este echivalentă cu:

```
{  
    repetă nimic cât timp m=0;  
    m ← 0;  
}
```

Primitiva **UNLOCK(m)**
este echivalentă cu:

```
{  
    m ← 1;  
}
```

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
-----	-----

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

<început sarcina>

LOCK(m);

<secțiune 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 funcție este realizată de instrucțiuni denumite TSB (Test and Switch Branch).