9 Teoria generală a sistemelor de operare

9.1 Procese

9.1.1 Conceptul de proces

Facilitatea de a a rula mai multe programe simultan în cadrul unui sistem de operare este considerată astăzi normală de către toţi utilizatorii. Rularea unui navigator web (Mozilla Firefox, Internet Explorer, Safari, Konqueror etc.) simultan cu rularea unui program pentru citirea poştei electronice (Mozilla Thunderbird, Outlook Express, Eudora, Pine etc.) este o practică de zi cu zi a majorităţii utilizatorilor. Din punctul de vedere al unui proces de operare toate aceste programe rulând pe un sistem de operare sunt considerate procese sau task-uri.

Formal, un *proces* sau *task,* este un calcul care poate fi executat concurent (în paralel) cu alte calcule. Este o abstractizare a activităţii procesorului care pur şi simplu execută instrucţiunile care i se transmit fără a face diferenţiere între procesele cărora le aparţin. Revenind la exemplul iniţial, un procesor nu "ştie" dacă instrucţiunea executată aparţine de Mozilla Firefox sau Outlook Express.

In terminologia actuală există o tendinţă de confuzie între noţiunea de proces şi cea de program. Deşi folosirea noţiunii de program pentru a ne referi la un proces nu creează în general confuzii, considerăm totuşi necesară clarificarea situaţiei. *Procesul are un caracter dinamic*, el precizează o secvenţă de activităţi în curs de execuţie, iar *programul are un caracter static*, el numai descrie această secvenţă de activităţi. Cu alte cuvinte procesul este un program în execuţie.

Existenţa unui proces este condiţionată de existenţa a trei factori:

1. *procedură* (un set de instrucţiuni) care trebuie executată;
2. un *procesor* care să poată executa aceste instrucţiuni;
3. un *mediu* (memorie, periferice) asupra căruia să acţioneze procesorul conform celor precizate în procedură.

Execuţia unui program (şi implict transformarea acestuia în proces) implică execuţia de către procesor a fiecărei instrucţiuni din programul (procedura) respectiv(ă). In mod inerent, aceste instrucţiuni vor avea efect asupra mediului în care procesul rulează, cum ar fi: alocarea de memorie, folosirea perifericelor, tipărirea unui mesaj pe terminal, consumul de timp etc.

Trebuie menţionat că în arhitecturile actuale monoprocesor (vezi [52]), mai multe procese care rulează concomitent sunt de fapt deservite alternativ de către procesor. El (procesorul) execută alternativ grupuri de instrucţiuni din fiecare program de-a lungul unei cuante de timp, după care comută la alt grup de la alt program. Dimensiunea cuantei de timp este astfel aleasă (şi aşa de mică) încât momentele de staţionare ale unui program nu sunt sesizabile de către utilizator. Astfel se creează impresia de simultaneitate în execuţie.

Paralelismul este efectiv doar în cadrul sistemelor multiprocesor. Practic, două procese pot fi executate simultant în sens propriu doar dacă rulează pe o maşină bi-procesor.

9.1.2 Concurenţa între procese

Existenţa simultană a mai multor procese într-un sistem de operare ridică probleme în ceea ce priveşte accesul la resursele sistemului. Prin resursă înţelegem orice este necesar unui proces pentru a-şi continua activitatea. Câteva resurse mai populare sunt:

- memoria (cereri de alocare de memorie)
- discul magnetic (acces la fişiere)
- terminalul (pentru interacţiunea cu utilizatorul)
- interfaţa de reţea (pentru interacţiunea cu alte maşini legate prin reţea)

9.1.2.1 Secţiune critică; resursă critică; excludere mutuală.

O problemă legată de accesul la resurse este asigurarea corectitudinii operaţiilor executate in regim de concurenţă. Pentru a vedea cum operaţii corecte pot da rezultate greşite în regim de concurenţă, vom analiza un exemplu clasic. Să considerăm un fişier care stochează numărul de locuri libere într-un sistem de vânzare de bilete. Vânzarea unui bilet va avea ca şi consecinţă decrementarea numarului de locuri libere. De asemenea, să considerăm că există două procese care efectuează simultan operaţii asupra acestui fişier. Fragmentul de cod necesar pentru decrementarea numărului de locuri este prezentat in fig. 9.1.

```
int n;
int fd = open(”locuri.db”, ”ORDWR”);
read(fd, &n, sizeof(int));
lseek(fd, 0, SEEK_SET);
n--;
write(fd, &n, sizeof(int));
close(f);
```

Figura 9.1 O secvenţă de decrementare

In situaţia în care codul din fig. 9.1 este rulat simultan de două sau mai multe procese, şi considerând faptul că procesorul execută alternativ grupuri de instrucţiuni din procese diferite, este posibil ca instrucţiunile de mai sus să fie executate în procese separate în ordinea din figura 9.2.

Procesul A	**n = (în A)**	**Procesul B**	**n = (în B)**
`read(fd,&n,sizeof(int))`	10		
	10	`read(fd,&n,sizeof(int))`	10
`n--`	9		10
`lseek(fd,0,SEEK_SET)`	9		10
`write(fd,&n,sizeof(int))`	9		10
		`n--`	9
		`lseek(fd,0,SEEK_SET))`	9
		`write(fd,&n,sizeof(int))`	9

Figura 9.2 Programul din fig. 9.1 rulat "defavorabil" în două procese

In mod evident, ambele procese citesc *aceeaşi valoare* din fişier, o decrementează şi apoi o scriu la loc. Ca urmare, în final, în loc ca numărul de locuri să scadă cu două poziţii, va scădea doar cu una şi ca urmare operaţiile executate asupra fişierului deşi sunt greşite. In viaţa de zi cu zi o asemenea eroare ar conduce la vânzarea aceluiaşi loc în sală de două ori (o “bucurie” cu care mulţi dintre citiori au avut proabil de-a face).

Trebuie remarcat că eroarea poate să apară doar la o execuţie concurentă, altfel secvenţa de cod fiind perfect corectă. Eroarea apare din lipsa de sincronizare între accesele proceselor la fişier. Procesul B, în loc să opereze asupra numărului de locuri decrementat de procesul A, reuşeşte să citească numarul de locuri prea devreme şi va opera asupra *aceluiaşi număr de locuri* ca şi procesul A. Pentru a obţine rezultate corecte, secvenţa de execuţie dorită este cea din fig. 9.3. Adică secvenţa de cod care decrementează numărul de locuri să fie executată la un moment dat doar de un signur proces.

Procesul A	**n = (în A)**	**Procesul B**	**n = (în B)**
`read(fd,&n,sizeof(int))`	10		
`n--`	9		
`lseek(fd,0,SEEK_SET)`	9		
`write(fd,&n,sizeof(int))`	9		
		`read(fd,&n,sizeof(int))`	9
		`n--`	8
		`lseek(fd,0,SEEK_SET)`	8
		`write(fd,&n,sizeof(int))`	8

Figura 9.3 Programul din fig. 9.1 rulat ca secţiune critică

Acelaşi gen de situaţie se creează şi în momentul în care procesele scriu date pe disc. Dacă sistemul de operare ar servi aceste cereri în stilul din figura 9.2 două procese ar putea primi acelaşi bloc pe disc pentru a scrie în el date aparţinând la fişiere diferite. Este uşor de imaginat ce s-ar întâmpla dacă într-un script shell editat cu un editor de text ar apărea părţi din conţinutul unui fişier `".o"` generat de `gcc` la compilare …

Revenind la problema noastră, vom spune că porţiunea de program din fig. 9.1 începând cu `read` şi terminând cu `write` este o *secţiune critică*, deoarece nu este permis ca ea să fie executată simultan de către două procese. Analog, vom spune că poziţia pe care se stochează numărul de locuri în fişier este o *resursă critică*, deoarece nu poate fi accesată simultan de mai multe procese. In sfârşit, pentru a evita situaţia eronată de mai sus vom spune că procesele A şi B trebuie să se *excludă reciproc*, deoarece trebuie să aibă acces exclusiv la secţiunea şi la resursa critică [22].

Problema secţiunii critice a suscitat un interes deosebit în istoria **SO**. Vom expune şi noi câteva aspecte ale ei, pentru a se vedea câteva dintre "subtilităţile" programării concurente. Să vedem mai exact cerinţele problemei:

- la un moment dat, numai un singur proces este în secţiunea critică; orice alt proces care solicită accesul la ea, îl va primi după ce procesul care o ocupă a terminat de executat instrucţiunile secţiunii critice;
- vitezele relative ale proceselor sunt necunoscute;
- oprirea oricărui proces are loc numai în afara secţiunii critice;
- nici un proces nu va aştepta indefinit pentru a intra în secţiunea critică.

Pentru a fixa ideile, presupunem că fiecare proces execută în mod repetat, de un număr nedefinit de ori, întâi secţiunea critică şi apoi restul programului. Descrierea problemei astfel puse este dată în limbaj pseudo-cod în fig. 9.4.

```
do {
  <secţiune critică>
  <rest program>
} while(false);
```

Figura 9.4 Un program cu secţiune critică

Au existat multe încercări de rezolvare a problemei secţiunii critice. In [4] sunt date şi comentate trei încercări de descriere, fiecare dintre ele încălcând unele dintre condiţiile puse mai sus. Matematicianul Dekker a fost primul (1965) care a dat o soluţie software (rezonabilă, care însă este destul de complicată şi încâlcită). In 1981 Peterson a dat o soluţie ceva mai simplă. Aceste soluţii se referă la rezolvarea secţiunilor critice apărute la concurenţa în cadrul aceluiaşi proces (un subiect pe care nu l-am abordat încă). Totuşi, pentru a evidenţia complexitatea problemei, prezentăm soluţia lui Peterson în fig 9.5 şi lăsăm cititorului "plăcerea" de a demonstra că soluţia este corectă.

Cod iniţial	int c1 = 1;
	int c2 = 1;
	int schimb;	

Cod executat în paralel	do {
	  c1 = 0;
	  schimb = 1;
	  while((c2 == 0) &&
	  (schimb == 1)){
	  <aşteaptă>;
	  }
	  <secţiune critică>;
	  c1 = 1;
	  <rest program 1>
	} while(false)
		do {
		  c2 = 0;
		  schimb = 2;
		  while((c1 == 0) &&
		  (schimb == 2)){
		  <aşteaptă>;
		  }
		  <secţiune critică>;
		  c2 = 1;
		  <rest program 2>
		} while(false)

Figura 9.5 Soluţia Peterson pentru secţiune critică

Evident, rezolvarea nu este simplă. Dacă se cere ca secţiunea critică să fie accesibilă la mai multe procese, soluţia din fig. 9.5 nu poate fi generalizată prea uşor. Mai mult, soluţiile software presupun o *aşteptare activă*, adică toate procesoarele, în faza de aşteptare execută în mod repetat o aceeaşi instrucţiune (ciclează încontinuu). Ar fi de dorit ca pe timpul cât un proces aşteaptă să primească dreptul de execuţie a secţiunii critice, procesorul lui să devină disponibil spre a servi alte procese. O asemenea soluţie poate fi realizată folosind conceptul de *semafor* descris în secţiunea următoare.

9.1.2.2 Conceptul de semafor.

Pentru a evita situaţia de mai sus şi pentru a permite o serie de operaţii cu procese (pe care le vom prezenta mai târziu), Dijkstra a introdus conceptul de semafor.

Un *semafor s* este o pereche

`(v(s), c(s))`

unde **`v(s)`** este valoarea semaforului, iar **`c(s)`** o coadă de aşteptare. Valoarea **`v(s)`** este un număr întreg, care primeşte o valoare iniţială **`v0(s)`.** Coada de aşteptare conţine (pointeri la) procesele care aşteaptă la semaforul **`s`**. Iniţial coada este vidă, iar disciplina cozii depinde de sistemul de operare (FIFO, LIFO, priorităţi etc.).

Pentru gestiunea semafoarelor se definesc două operaţii indivizibile **`P(s)`** şi **`V(s)`** ale căror roluri, exprimate laconic, sunt "a trece de resursă" şi "a anunţa eliberarea resursei". (Denumirile operaţiilor au fost date de Dijkstra după primele litere din limba olandeză: P de la *prolaag* prescurtare de la *probeer te verlagen* – încercare de a descreşte, respectiv V de la *verhoog* – a creşte). In unele lucrări [4], operaţia **`P`** se mai numeşte **WAIT**, iar operaţia **`V`** se

mai numeşte **SIGNAL**. Indivizibilitatea operaţiilor înseamnă că ele nu pot fi întrerupte în cursul execuţiei lor, deci nu pot exista două procese care să execute simultan **P**(s) sau **V**(s), sau simultan două operaţii **P**, sau simultan două operaţii **V**.

Definiţiile exacte ale operaţiilor **P** şi **V**, apelate de un proces A şi aplicate asupra unui semafor sunt date în fig. 9.6.

```
// P(s) apelat de procesul A:

v(s) = v(s) - 1;
if(v(s) < 0) {
  STARE(A) = WAIT;
  c(s) ← A;
    //Procesul A intra în aşteptare
  <Trece controlul la DISPECER>;
} else {
  <Trece controlul la procesul A>;
}
```

```
// V(s) apelat de procesul A

v(s) = v(s) + 1;
if(v(s) <= 0) {
  c(s) → B;
    // Extrage din coadă alt proces B
  STARE(B) = READY;
  <Trece controlul la DISPECER>;
} else {
  <Trece controlul la procesul A>;
}
```

Figura 9.6 Operaţiile P(s) şi V(s) apelate de procesul A

Pentru a sugera funcţionarea (şi denumirea) semaforului vom considera un exemplu concret. Fie G1 şi G2 două gări legate prin *n* linii paralele, pe care se circulă numai de la G1 spre G2. Presupunem că în G1 intră mai mult de *n* linii. Procesul trecerii trenurilor va fi dirijat astfel:

Valoarea iniţială: `v0(s) = n`

Procesul de trecere:

```
P(s);     <Trenul trece pe una din cele n linii>;    V(s);
```

Să notăm cu *np*(*s*), *nv*(*s*), numărul de primitive **P**(s), respectiv **V**(*s*) efectuate până la un moment dat, *v0*(*s*) valoarea iniţială a semaforului *s*, iar *nt*(*s*) numărul proceselor care au trecut de semaforul *s*. Următoarele proprietăţi au loc:

1. `v(s) = v0(s) - np(s) + nv(s) ;`
2. `dacă v(s)<0` atunci `în c(s)` există `-v(s)` procese;
3. `dacă v(s)>0` atunci `v(s)` procese pot trece succesiv de semaforul s fără să fie blocate ;
4. `nt(s) = min(v0(s) + nv(s), np(s))`

Demonstraţia acestor proprietăţi este simplă, prin inducţie după numărul operaţiilor **P** şi **V** efectuate asupra semaforului. Aspectul critic al semaforului este că operaţiile asupra lui (**P** şi **V**) sunt atomice. Astfel două procese nu le pot executa în acelaşi timp asupra aceluiaşi semafor *s*.

Folosirea semafoarelor rezolvă complet şi elegant problema secţiunii critice prezentată în secţiunile anterioare. Rezolvarea presupune folosirea unui singur semafor *s* şi este corectă indiferent de numărul de procese care folosesc în comun secţiunea critică. Descrierea soluţiei este dată în fig. 9.7.

Cheia este tocmai semaforul *s*. Dacă `v0(s)=1` şi toate procesele care folosesc secţiunea critică sunt de forma:

```
...; P(s);  <secţiune critică>; V(s); ...
```

atunci se demonstrează uşor că $v(s)<1$, indiferent de numărul proceselor care operează asupra lui *s* cu operaţii **P** sau **V**. Drept urmare, în baza proprietăţilor menţîonate mai sus, un singur proces poate trece de acest semafor, deci sînt îndeplinite condiţiile cerute pentru a proteja o secţiune critică. Un astfel de semafor mai poartă numele de *semafor de excludere mutuală.* Sincronizarea este o operaţie fundamentală în programarea concurentă. In secţiunile care urmează vom întâlni mai multe aplicaţii ale ei.

Cod iniţial	sempahore s; v0(s) = 1;	
Cod executat în paralel	do { P(s); <secţiune critică>; V(s); <rest program 1> } while(false)	do { P(s); <secţiune critică>; V(s); <rest program 2> } while(false)

Figura 9.7 Soluţia cu semafor pentru secţiune critică

9.1.2.3 Problema producătorului şi a consumatorului.

Să presupunem că există unul sau mai multe procese numite *producătoare*, şi unul sau mai multe procese numite *consumatoare*. De exemplu, conceptele de pipe şi FIFO sunt de această natură (vezi 5.3.2 şi 5.3.4). Transmiterea informaţiilor de la producători spre consumatori se realizează prin intermediul unui buffer cu *n* intrări pentru *n* articole. Fiecare producător depune câte un articol în buffer, iar fiecare consumator scoate câte un articol din buffer. Problema constă în a dirija cele două tipuri de procese astfel încât:

- să existe acces exclusiv la buffer;
- consumatorii să aştepte când bufferul este gol;
- producătorii să aştepte când bufferul este plin.

Rezolvarea prin semafoare este foarte simplă. Se vor folosi trei semafoare: *gol, plin* şi *exclus.* Rezolvarea este descrisă în fig. 9.8. Semaforul *exclus* asigură accesul exclusiv la buffer în momentul modificării lui, pentru a nu permite depuneri şi extrageri de date simultane. Semaforul *plin* indică numărul de poziţii ocupate din buffer, iar semaforul *gol* numărul de poziţii libere din buffer. Depunerea unui articol în buffer necesită decrementarea (**P**) semaforului *gol* şi incrementarea (**V**) semaforului *plin* (practic o poziţie este "mutată" din semaforul *gol* in semaforul *plin*). Extragerea unui articol din buffer este operaţia inversă depunerii.

Producător	**Consumator**

semaphore plin, gol, exclus;
v0(plin) = 0;
v0(gol) = n;
v(exclus) = 1;	

do { <produce articol>; P(gol); P(exclus); <depune articol în buffer>; V(exclus); V(plin); } while(false);	do { P(plin); P(exclus); <extrage articol din buffer>; V(exclus); V(gol) <consuma articol>; } while(false);

Figura 9.8 Problema producătorului şi a consumatorului rezolvată cu semafoare

9.1.2.4 Regiuni critice condiţionate.

Prin intermediul operaţiilor de excludere şi sincronizare pot fi rezolvate o serie de probleme clasice de concurenţă. Dintre acestea au fost prezentate problema secţiunii critice şi problema producătorului şi a consumatorului. Am văzut că descrierea lor cu ajutorul semafoarelor este destul de comodă. Din păcate, operaţiile **P** şi **V**, lăsate libere la îndemâna programatorului, pot provoca necazuri mari. Incercaţi să vedeţi ce se întâmplă dacă la descrierea din fig. 9.8 se inversează între ele operaţiile **P**(*gol*) şi **P**(*exclus*) din procesul producător. Acesta este motivul pentru care au fost introduse construcţii structurate de concurenţă. Prin intermediul lor se poate exercita un control riguros asupra încălcării regulilor de concurenţă.

Vom descrie o astfel de construcţie structurată, numită *regiune critică condiţionată*. Ea înglobează într-un mod elegant conceptele de secţiune critică şi resursă critică despre care am vorbit mai sus, cu posibilitatea (eventuală) de a executa regiunea numai dacă este îndeplinită o anumită condiţie.

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

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

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

O *regiune critică condiţionată* se defineşte astfel:

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

unde `r` este numele unei resurse declarate ca mai sus, `B` este o expresie booleană, iar `S` este secvenţa de instrucţiuni corespunzătoare regiunii critice. Dacă este prezentă opţiunea `when`, atunci `S` este executată numai dacă `B` este adevărată. Variabilele din resursa `r` pot fi folosite numai de către instrucţiunile din secvenţa de instrucţiuni `S`.

Descrierea prin semafoare a unei regiuni critice condiţionate este dată în fig. 9.9. Sunt necesare două semafoare şi un număr întreg. Semaforul `sir` reţine în coada lui toate procesele care solicită acces la regiune. Semaforul `exclus` asigură accesul exclusiv la anumite secţiuni ale implementării (în special la modificarea numărului întreg `nr`). Intregul `nr` reţine câte procese au cerut acces la regiune, la un moment dat. In secţiunea următoare vom da o aplicaţie interesantă a regiunii critice condiţionate.

In fig 9.9 algoritmul propriu-zis începe de la linia 9. Dacă regiunea critică este folosită fără o condiţie asociată, se va trece direct la execuţia lui `S`. Altfel, incrementând variabila `nr`, se marchează faptul că încă un proces accesează regiunea critică cu o condiţie asociată. Cum mai multe procese pot face acest lucru simultan, incrementarea trebuie protejată. Deşi la prima vedere este doar o singură operaţie, în realitate incrementarea se traduce într-o secvenţă de instrucţiuni binare care pot fi executate într-o ordine similară cu cea din fig 9.2 şi poate conduce la erori. Ca urmare, înainte ca incrementarea să aibă loc, se execută **P**(`exclus`) care dacă condiţia `B` este satisfăcută va proteja şi execuţia lui `S`. Altfel, dacă condiţia `B` nu este îndeplinită procesul trebuie să intre în aşteptare. Aceasta se face prin **P**(`sir`), dar nu înainte de a elibera semaforul `exclus` prin **V** (`exclus`). Dacă **V** (`exclus`) este omis, practic orice alt proces care ar accesa regiunea critică se va opri la linia 9. In acest fel concurenţa pentru

evaluarea condiţiei *B* ar fi. Scopul nostru însă nu este serializarea evaluării condiţiei *B* şi a secvenţei de instrucţiuni *S*, ci serializarea accesului la *S* în momentul în care *B* este satisfăcută. Ca urmare trebuie să lăsăm fiecare proces să evalueze *B* fără restricţii. Dacă condiţia *B* este îndeplinită, se decrementează *nr* (marcând astfel intrarea în regiunea critică) şi se execută instrucţiunile din *S*. Apoi (liniile 20-21), semaforul *sir* este incrementat până la zero în aşa fel încât controlorul va alege unul din procesele din coadă pentru a fi executat. In final semaforul *exclus* este eliberat la linia 22. Procesul proaspăt reactivat va reevalua condiţia B şi dacă este îndeplinită va continua şi va executa instrucţiunile din *S*.

```
 1 // Valorile de pornire
 2 semaphore sir, exclus;
 3 int nr, i, cuConditie;
 4 v0(sir) = 0;
 5 v0(exclus) = 1;
 6 nr = 0;
 7
 8 //Codul corespunzător regiunii
 9 P(exclus);
10 if(cuConditie) {
11   nr++;
12   while(!B) {
13     V(exclus);
14     P(sir);
15     P(exclus);
16   }
17   nr--;
18 }
19 S;
20 for(i=0; i<nr; i++)
21   V(sir);
22 V(exclus);
```

Figura 9.9 Implementarea unei regiuni critice condiţionate

Până în acest moment lucrurile ar părea clare, dar există o situaţie în care implementarea de mai jos nu funcţionează! Este vorba de cazul în care toate procesele accesează regiunea critică condiţionată în momentul în care *B* nu este satisfacută şi ca urmare vor intra în aşteptare în coada semaforului *sir*. Intrebarea este: cum vor ieşi aceste procese din caoda semaforului? Este necesar ca cineva (un proces) să modifice starea condiţiei B şi apoi să anunţe schimbarea proceseleor care aşteaptă. Aceasta poate avea loc în două moduri:

1. După câtva timp un alt proces schimbă starea condiţiei *B*, intră în regiunea critică şi constată că *B* este satisfăcută. Ca urmare, nu va intra în coada semaforului *sir* şi după execuţie lui *S* îi va incrementa valoarea până la zero (liniile 20-21). In acest moment unul dintre procesele blocate în coada semaforului va fi rectivat şi va trece prin scţiunea critică. La rândul lui, acesta va reincrementa semaforul până la zero astfel reactivand un nou proces.
2. Unul dintre procesele care folosesc regiunea critică nu o asociază cu condiţia *B* şi ca urmare va omite aşteptarea la semaforul *sir*. La fel ca şi în cazul anterior, după execuţia lui S, acest proces va incrementa semaforul activând procesele din coada lui.

Un program care se bazează pe varianta 1 trebuie să se asigure ca întotdeuna va exista un proces care va găsi condiţia *B* satisfăcută, altfel totul se blochează. Varianta a doua este mai puţin riscantă pentru că un proces apelând regiunea critică fără condiţie nu va rămâne blocat. Pentru a cuprinde ambele într-un singur program, s-a adăugat condiţia din linia 10 cu închiderea ei din linia 18.

9.1.2.5 Problema citirilor şi a scrierilor.

Problema a fost formulată în 1971 de către Courtois, Heymans şi Parnas [4] [50]. Se presupune că există două tipuri de procese: *cititor* şi *scriitor*. Ele partajează împreună o aceeaşi resursă, să zicem un fişier. Un proces scriitor modifică conţinutul fişierului, iar unul cititor consultă informaţiile din el. Spre deosebire de problema producătorului şi a consumatorului (unde toate procesele aveau acces exclusiv la resursă), la acest tip de problemă *orice proces scriitor are acces exclusiv la resursă, în timp ce mai multe procese cititor pot avea acces simultan la ea.*

Intre cerinţele problemei trebuie să se aibă în vedere ca nici un proces să nu fie nevoit să aştepte indefinit. De asemenea, trebuie să se precizeze, pentru cazurile de acces simultan, care tip de proces este mai prioritar. In prima versiune era stabilit ca procesele cititor să fie mai prioritare. Aceasta putea conduce la situaţia ca procesele scriitor să aştepte indefinit dacă o infinitate de procese cititoare cer accesul. Versiunile ulterioare au dat prioritate proceselor scriitor. Drept urmare, este posibil ca procesele cititor să aştepte indefinit.

In secţiunile precedente am văzut rezolvarea diferitelor probleme de programare concurentă, folosind cel mai des operaţiile **P** şi **V**. Ele sunt suficiente pentru descrierea acestei probleme şi invităm cititorul să o facă. In secţiunea precedentă am definit o construcţie mai elegantă şi mai uşor de urmărit în programe, regiunea critică condiţionată.

O posibilă rezolvare, folosind regiuni critice condiţionate, este cea din fig. 9.10. Soluţia aparţine lui P. B. Hansen, 1973, şi ea dă prioritate proceselor scriitor faţă de cele cititor. Un proces scriitor intră în lucru imediat după ce procesele cititor active şi-au încheiat citirile curente. Resursa `f` este zona la care doresc accesul cele două tipuri de procese. Presupunem că ea este un fişier, iar operaţiile de bază asupra lui sunt `read` şi `write`. Resursa `c` conţine două contoare care indică câte procese de fiecare tip au cerut acces la resursa `f`. Este de remarcat folosirea regiunii `c` cu şi fără o condiţie ataşată.

Cititor	Scriitor

`int nw, nr;`
`resource f;`
`resource c :: nr, nw;`
`nr = 0;`
`nw = 0;`	

`region c`
`  when nw == 0`
`  do nr++;`
`<citeşte date>;`
`region c`
`  do nr--;`
	`region c`
	`  do nw++;`
	`region c`
	`  when nr == 0`
	`  do <aşteaptă să se termine toţi cititorii activi>;`
	`region f`
	`  do <scrie date>;`
	`region c`
	`  do nw--;`

Figura 9.10 Problema citirilor şi a scrierilor

9.1.3 Problema impasului

9.1.3.1 Conceptul de impas.

Noţiunea de impas nu este specifică sistemelor de operare sau informaticii, ci este un fenomen întâlnit în multe situaţii din viaţa de zi cu zi. In general, impasul poate fi definit ca o blocare a actvităţii normale a două sau mai multe entităţi ca urmare a efectului lor coroborat asupra mediului.

Un exemplu de impas uşor de vizualizat este cel al traficului în jurul unui cvartal. După cum se vede în fig 9.11, cele 8 maşini sunt blocate în mod circular, şi traficul nu poate continua fără ca una dintre ele să dea înapoi.

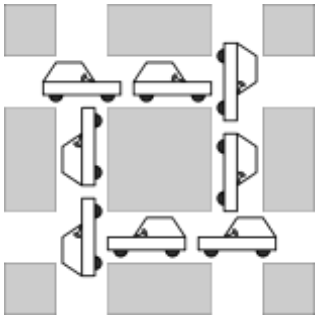

Figura 9.11 Un impas într-o intersecţie

Trecând la domeniul informatic, o modalitate uşoară de a crea o situaţie de impas este invesarea operaţiilor **P** şi **V** în procese concurente. Fig 9.12 prezintă codul celor două procese.

Proces A	Proces B

semaphore x, y;
v0(x) = 1;
v0(y) = 1;	

P(x); <instrucţiuni A>; P(y); V(y); V(x);	P(y); <instrucţiuni B>; P(x); V(x); V(y);

Figura 9.12 Impas provocat de două semafoare

Dacă instrucţiunile celor două procese sunt executate întâmplător în secvenţa din figura 9.13, cele două procese ***A*** şi ***B*** se vor afla într-o stare de impas. Cauza este oarecum similară cu situaţia traficului prezentată anterior. Procesul ***A*** va deţine semaforul binar ***x***, procesul ***B*** va deţine semaforul binar ***y***, apoi amble vor încerca să obţină semaforul deţinut de celălalt, fără a elibera semforul pe care îl deţine inainte de a-l obţine pe cel deţinut de celălat proces. Situaţia este ilustrată în fig. 9.13.

Proces A	Proces B
P(x);	
	P(y);
<instrucţiuni A>;	
	<instrucţiuni B>;
P(y);	
wait ...	P(x) wait...

Figura 9.13 O situaţie de impas

Cauza principală care a generat această situaţie este faptul că procesele îşi ocupă resursele numai atunci când au nevoie efectiv de ele şi ca urmare apare situaţii de aşteptare circulară.

Apropiindu-ne mai mult de funcţionarea unui sistem de operare, să presupunem că în sistem sunt numai două benzi magnetice ***MM0*** şi ***MM1***. Dacă un proces cere mai întâi ***MM0*** şi apoi

MM1 fără a elibera ***MM0***, iar al doilea cere ***MM1*** şi apoi ***MM0*** fără a elibera ***MM0***, poate apare o astfel de aşteptare circulară. Acest fenomen este cunoscut în literatură sub mai multe denumiri, şi anume: *impas, interblocare, deadlock, deadly embrace* etc. [4], [10], [51].

Ultimele două exemple de *impas* au avut ca protagonişti câte două procese şi două resurse. Impasul poate apărea însă în situaţii mai complexe care implică nu doar două procese, ci mai multe şi nu neaparat resurse specifice, ci clase de resurse. Să presupunem că într-un anume moment din funcţionare, un sistem de operare dispune de 50MB de memorie liberă şi de 10 poziţii libere pnetru crearea de conexiuni în reţea. In acest moment un proces ***A*** cere alocarea a 30MB de memorie şi apoi a 7 conexiuni în reţea, iar un proces ***B*** cere întâi alocarea a 5 conexiuni de reţea şi apoi alocarea a 40MB de memorie. Similar cu exemplul anterior, este posibil ca ambelor procese să li se satisfacă primele cereri, însă după aceea din lipsă de resurse va apare impas. Acest tip de impas este mai greu de detectat pentru ca nu implică resurse punctuale ci clase de resurse, în cazul nostru memoria RAM şi conexiunile la reţea.

Impasul rezultat din conflictul asupra câtorva resurse punctuale poate fi modelat teoretic pe baza impasului pe clase, considerând că fiecare clasă de elemente conţine un singur element. Altfel spus, impasul pe clase este o generalizare a impasului pe resurse punctuale.

Impasul este o stare gravă care poate duce la un blocaj al sistemului de operare sau la distrugerea unor procese. In cazul în care două procese se blochează unul pe altul, doar ele vor fi în impas şi situaţia se poate salva terminând forţat unul dintre ele. Dacă însă impasul se manifestă între un proces şi sistemul de operare sau direct în interiorul sistemului de operare, parctic sistemul "îngheaţă" şi işirea cea mai frecventă din asemenea situaţii este restartarea calculatorului!

Astfel de situaţii trebuie avute în vedere şi prevenite de către proiectanţii de sisteme de operare. Dar, după cum vom vedea, tratarea impasului este costisitoare şi ca urmare proiectanţii o tratează doar în situaţii izolate pentru a nu afecta performanţa sistemului.

In legătură cu impasul, fiecare sistem de operare trebuie să dea rezolvări măcar la una dintre următoarele probleme:

- ieşirea din impas,
- detectarea unui impas,
- evitarea (prevenirea apariţiei) impasului.

Cele două rezultate de mai sus dau condiţii prin care este indicată eventuala apariţie a unui impas, însă nu furnizează metode practice de rezolvare a problemelor de mai sus. In continuare vom da metode şi tenhici de rezolvare a problemelor generate de impas.

9.1.3.2 Ieşirea din impas.

Soluţiile ieşirii din impas pot fi clasificate în manuale şi automate. Soluţiile manuale sunt cele care implică acţiuni ale operatorului uman. Cele automate sunt executate de către sistemul de operare.

Ieşirea din starea de impas automată (de către sistemul de operare) presupune în primul rând detectarea acestei stări. La rândul ei, detectarea stării de impas implică existenţa unei

componente a sistemului de operare care să întreţină şi să analizeze starea alocărilor de resurse din sistem (vom discuta acest aspect mai jos). Pe baza acestei analize, sistemul de operare recunoaşte starea de impas şi trebuie să aleagă o soluţie pentru a-l rezolva.

Indiferent dacă este automată sau manuală, soluţia ieşirii din impas necesită acţiuni drastice. In cele ce urmează prezentăm trei astfel de soluţii.

O primă variantă este *reîncărcarea sistemului de operare*. Soluţia nu prea este de dorit, dar este cea mai la îndemâna operatorului. Dacă pierderile nu sunt prea mari se poate adopta şi o astfel de strategie.

O a doua varintă este *alegerea un proces "victimă"*, care este fie cel care a provocat impasul, fie un altul de importanţă mai mică, astfel încât să fie înlăturat impasul. Se distruge acest proces şi odată cu el toţi descendenţii lui. Această alegere este o parte importantă a algortimilor automaţi de rezolvarea a impasului. In cazul abordării manuale, din păcate, criteriul de alegere a "victimei" este de multe ori departe de a fi obiectiv.

O a treia soluţie este *cre*erea *unui "punct de reluare", care este o fotografie a memoriei pentru procesul victimă* şi pentru procesele cu care el colaborează. Apoi se distruge procesul victimă şi subordonaţii lui. Procesul victimă se reia ulterior. Crearea punctelor de reluare reclamă consum de timp şi complicaţii, motiv pentru care de multe ori nu se execută.

9.1.3.3 Detectarea impasului

Detectarea unui impas se face atunci când **SO** nu are un mecanism de prevenire a impasului. Pentru a se reuşi detecţia este necesar ca **SO** să aibă o evidenţă clară, pentru fiecare proces, privind resursele ocupate, precum şi cele solicitate dar neprimite. Pentru aceasta, cel mai potrivit model este *graful alocării resurselor*.

Facem ipoteza, că fiecare resursă este într-un singur exemplar şi vom nota `R1, R2, ... , Rm` aceste resurse. Fie acum `P1, P2, ... , Pn` procesele din sistem. Se construieşte un graf bipartit (**X**, **U**) astfel:

- **X** = {`P1, P2, ..., Pn, R1, R2, ..., Rm`}
- `(Rj, Pi)` este arc în **U** dacă procesul `Pi` a ocupat resursa `Rj`.
- `(Pi, Rj)` este arc în **U** dacă procesul `Pi` aşteaptă să ocupe resursa `Rj`.

Dacă graful (**X**, **U**) definit mai sus este ciclic, atunci sistemul se află în stare de impas. Demonstraţia este un exerciţiu simplu de inducţie. Detectarea ciclicităţii unui graf se face folosind algoritmii clasici din teoria grafelor.

In fig 9.14 prezentăm un posibil graf de alocare. Nodurile pătrate reprezintă procesele iar nodurile rotunde reprezintă resursele. Pentru o mai uşoară diferenţiere arcele de aşteptare pentru o resursă au fost desenate cu linie întreruptă.

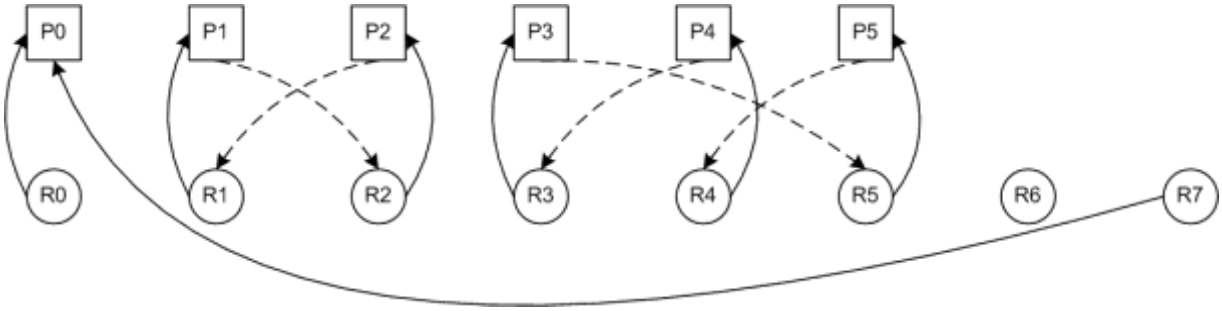

Figura 9.14 Un graf de alocare a resurselor

Dacă selectăm numai subgraful cu nodurile {P1, P2, R1, R2}, iar pentru resursele R1 şi R2 considerăm unităţile de bandă MM0 şi MM1, atunci se obţine graful de alocare pentru unul dintre exemplele prezentate în 9.1.3.1.

Deşi nu sunt evidente la prima vedere, graful din figura de mai sus conţine două cicluri. Primul ciclu implică procesele P1 şi P2, şi resursele R1 şi R2. Al doilea ciclu implică procesele P3, P4 şi P5, şi resursele R3, R4, şi R5. In ambele cazuri aveam de-a face cu stări de impas.

9.1.3.4 Evitarea (prevenirea apariţiei) impasului.

In 1971, Coffman, Elphic şi Shoshani [4], [19] au indicat patru condiţii *necesare* pentru apariţia impasului:

1. procesele solicită controlul exclusiv asupra resurselor pe care le cer (condiţia de *excludere mutuală*);
2. procesele păstrează resursele deja ocupate atunci când aşteaptă alocarea altor resurse (condiţia de *ocupă şi aşteaptă*);
3. resursele nu pot fi şterse din procesele care le ţin ocupate, până când ele nu sunt utilizate complet (condiţia de *nepreempţie*);
4. există un lanţ de procese în care fiecare dintre ele aşteaptă după o resursă ocupată de altul din lanţ (condiţia de *aşteptare circulară*).

Evitarea impasului presupune împiedicarea apariţiei uneia dintre aceste patru condiţii. Vom lua fiecare condiţie pe rând şi vom analiza modalităţile în care ea poate fi împiedicată şi efectele asupra funcţionării sistemului

Condiţia 1 – excluderea mutuală. In absenţa excluderii mutuale între procese, dispare noţiunea de aşteptare după o resursă critică blocată de alt proces. Dispărând noţiunea de aşteptare, evident dispare şi noţiunea de impas din moment ce nici un proces nu se va opri din execuţie. Eliminarea acestei condiţii însă nu este o alegere fericită. După cum am demonstrat anterior, excluderea mutuală este esenţială pentru pastrarea corectidudinii datelor. Practic imasul a apărut ca şi un efect secundar al aplicării acestei condiţii indiscutabil necesare.

Condiţia 2 – ocupă şi aşteaptă. Această condiţie spune că un proces poate aştepta după o resursă în timp ce deţine (blochează) alte resurse. Eliminarea acestei condiţii poate fi făcută în două moduri:

- Procesul trebuie să elibereze toate resursele blocate înainte de a solicita o nouă resursă.
- Procesul trebuie să blocheze toate resursele de care are nevoie la pornire.

Prima variantă este destul de forţată pentru că cere procesului să concureze pentru toate resursele ori de câte ori ar avea nevoie de una nouă.

A doua variantă nu este aplicabilă sistemelor de operare interactive (Windows, Unix etc.) pentru că ar restrânge posibilităţile de uitilizare a sistemului. Să luăm spre exemplu cazul comun al unui editor de text care suportă deschiderea simultană a mai multor fişiere. Dacă sistemul de operare ar impune blocarea tuturor resurselor la încărcarea procesului, editorul nostru ar fi obligat să solicite o cantitate finită de memorie pe care să o folosească la încărcarea fişierelor editate. Efectul acestui comportament are două tăişuri:

- Dacă editorul deschide un singur fişier mic, practic majoritatea memoriei solicitate va fi ţinută blocată fără motiv. Evident ar fi mult mai util ca sistemul de operare să dispună de această memorie pentru a servi alte procese.
- Dacă editorul trebie să dechidă multe fişiere de dimensiuni mari, va ajunge să ocupe toată memoria alocată şi să fie nevoit să refuze deschiderea altor fişiere.

Condiţia 3 – nepreempţie. Eliminarea condiţiei de nepreempţie permite sistemului de operare să aleagă procesele care ar trebui oprite pentru a ieşi din impas şi chiar să le oprească. Evident simpla terminare a unui proces nu este acceptabilă şi ca urmare sistemul trebuie să şi repornească acel proces. Deşi această abordare rezolvă situaţia de impas, nu garantează că procesele vor funcţiona conform aşteptărilor. Teoretic este posibil ca după restartarea unui proces dintr-un grup implicat într-un impas, procesele rămase să nu progreseze suficient pentru a ieşi din regiunea critică înainte ca programul repornit să reajungă acolo. In acel moment impasul va reapare şi din nou unul dintre procese va fi repornit. Această secvenţa se poate repeta la nesfârşit şi deşi nici unul dintre procese nu este blocat, totuşi niciunul nu progrseză cu execuţia. Acest fenomen este cunoscut sub numele de *impas activ* (livelock).

Condiţia 4 – aşteptarea circulară. O soluţie simplă pentru împiedicarea aşteptării circulare este impunerea unei ordini de blocare a resurselor. Sistemul de operare decide o ordine absolută a tuturor resurselor şi nici un proces nu are voie să blocheze o resursă cu număr de ordine mai mare înaintea uneia cu un număr de ordine mai mic. Pentru a vedea efectiv efectul blocării ordonate, vom prezenta o *problema filozofilor*.

Să considerăm o masă circulară la care sunt aşezaţi opt filozofi. Pe masă, în loc să fie şaisprezece tacâmuri (furculiţe şi cuţite) sunt doar opt, aşezate în aşa fel încât lângă fiecare farfurie este un cuţit şi o furculiţă (vezi fig 9.15). Fiecare filozof are nevoie de souă tacâmuri pentru a mânca, şi odată ca a luat un tacâm de pe masă nu-l va elibera până când nu termină de mâncat. Dacă toţi filozofii iau de pe masă tacâmul din stânga, ne aflăm se creează o situaţie de impas, pentru că nici un filozof nu va avea două tacâmuri pentru a putea mânca.

Impunerea blocării resurselor (luării tacâmurilor) în ordine, va preveni impasul pentru că filozoful H nu va putea ridica tacâmul 8 înaintea tacâmului 7. In consecinţă filozoful G va avea la dispoziţie două tacâmuri pentru a mânca. In momentul în care G va termina, va elibera tacâmul 6 şi filozoful F va putea mânca şi el. Procesul se va continua astfel până când toţi filozofii vor fi mâncat.

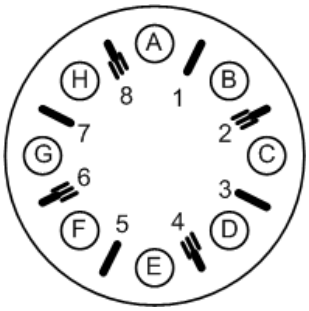

Figura 9.15 Problema filozofilor

In realitate însă, acestă soluţie este dificil de implementat pentru programatorii unui sistem de operare. Având în vedere cantitatea mare de resurse dintr-un sistem (de ordinul miilor) respectarea unei ordini prestabilite devine practic imposibilă. In plus, ordinea resurselor va fi de cele mai multe ori diferită de ordinea logică în care un program are nevoie de resurse, cauzând astfel scrierea de algoritmi mult mai complecşi sau mai puţin performanţi.

9.1.3.5 Alocarea controlată (conservativă) de resurse; algoritmul bancherului.

Evitarea impasului se poate face şi impunând o modalitate restrictivă de alocare de resurse. In esenţă, înainte de a servi o cerere de resurse, sistemul de operare analizează alocările de resurse. Dacă servirea cererii conduce la o stare cu potenţial de a genera un impas, cererea de resurse este amânată şi reanalizată în momentul în care are loc a modificarea în starea alocării de resurse în sistem. Soluţiile pe care le vom discuta vor adresa cazul general al impasului în care lucrăm cu clase de resurse.

Pentru a evalua dacă o stare a alocărilor poate conduce la impas este necesară cunoaşterea pentru fiecare proces a necesităţilor maxime de resurse din fiecare clasă. Astfel, la încărcare, fiecare proces va transmite sistemului de operare cantităţile de resurse din fiecare clasă de care va avea nevoie în cel mai rău caz.

Soluţie ineficientă. Folosind aceste maxime declarate de fiecare proces la încărcare, este foarte simplu să evităm impasul. Dacă adunăm maximul declarat de un nou proces pentru o resursă la cantitatea din acea resursă deja alocată şi suma depăşeşte maximul disponibil în sistem, pur şi simplu punem procesul în aşteptare până când se vor elibera suficiente resurse.

Aspectul pozitiv al acestei soluţii este că garantează servirea imediată a tuturor cererilor de resurse venite din partea proceselor încărcate. Acest lucru este posibil pentru că algoritmul menţine întotdeauna un număr redus de procese încărcate pentru a avea resurse suficiente pentru ele.

Soluţia este ineficientă pentru că este foarte puţin probabil ca un proces să folosească imediat după încărcare maximul declarat de resurse. In general resursele sunt folosite treptat şi rareori se va atinge maximul. Ca urmare, în majoritatea covârşitoarea a cazurilor un proces poate să-şi înceapă lucrul fără ca sistemul de operare să poată acoperi întreaga plaja de resurse pe care o poate solicita. Algortimul bancherului evită impasul fără a impune condiţii atât de restrictive.

Algoritmul Bancherului [4], [19] datorat lui Dijkstra este probabil cel mai renumit pentru alocarea controlată. Spre diferenţă de soluţia precedentă algoritmul bancherului permite tuturor proceselor să se încarce şi să înceapă execuţia. In schimb, algoritmul nu garantează servirea imediată a tuturor cererilor proceselor încărcate (un lucru de altfel rezonabil). Tot ce

garantează algoritmul bancherului este că orice proces încărcat poate fi servit (nu va rămane blocat la nesfârşit) cu condiţia să aştepte până când se mai eliberează resurse.

Pentru a clarifica numele algoritmului, funcţionarea algoritmului este similară cu procedura prin care un bancher (sistemul de operare) decide când şi cât (resurse) să împrumute unui client (procesul) dintr-o sumă maximă cu care îl poate credita. Diferenţele faţă de lumea reală sunt:

1. Bancherul dispune de o sumă limitată de împrumut, mai mică decât suma creditelor maxime ale clienţilor
2. Bancherul nu percepe nici dobândă şi nici comision
3. Toţi clienţii returnează banii în cele din urmă

Inainte de a prezenta algoritmul propriu-zis trebuie să prezentăm structurile de date pe care le vom folosi.

- `resurseExistente`
  - Cantităţile de resurse existente în sistem pentru fiecare clasă de resurse.
  - Tabel unidimensional. Fiecărei clase de resurse îi corespunde o poziţie
- `maxResursePerProces`
  - Cantităţile maxime de resurse pe care fiecare proces le poate solicita
  - Tabel bi-dimensional. Fiecare linie corespunde unui proces. Fiecare celulă dintr-o linie conţine cantitatea corespunzătoare unei clase de resurse.
- `resurseAlocatePerProces`
  - Cantităţile de resurse alocate fiecărui proces
  - Tabel bi-dimensional. Fiecare linie corespunde unui proces. Fiecare celulă dintr-o linie conţine cantitatea corespunzătoare unei clase de resurse.
- `solicitant`
  - Procesul care solicită resurse
- `cerere`
  - Cererea de analizat emisă de procesul solicitant
  - Tabel unidimensional. Fiecărei clase de resurse îi corespunde o poziţie
- `resurseDisponibile`
  - Resursele disponibile în sistem
  - Tabel unidimensional. Fiecărei clase de resurse îi corespunde o poziţie
- `proceseImplicate`
  - Tablou pentru marcarea proceselor implicate în algoritm
  - Fiecărui proces îi corespunde o poziţie

Folosind notaţiile de mai sus, algoritmul bancherului este prezentat în limbaj pseudocod in fig 9.16.

```
<Fă o copie tabelului "resurseAlocatePerProces" în tabelul "alocari">;

<Dacă nici un proces nu mai este implicat în algoritm, cererea poate
 fi servită. Sfârşit algoritm>;

<Populează tabelul "resurseDisponibile" cu diferenţele
 dintre "resurseExistente" şi sumele resurselor alocate fiecărui
 proces în tabelul "alocari">;

<Găseşte un proces Pi pentru care sunt suficiente resurse pentru ai
 servi cererea maximă>;

<Dacă nu există nici un astfel de proces, însemnă că cererea iniţială
```

```
     nu poate fi servită şi procesul solicitant este pus în
     aşteptare. Sfârşit algoritm>;

    <Simulează terminarea procesului Pi scoţându-l din lista proceselor
     implicate şi eliberându-i resursele (populează cu 0 intrarea lui Pi
     din tabelul "alocari")>;

    <Mergi la pasul 2>;
```

Figura 9.16 Algoritmul bancherului

In esenţă, algoritmul bancherului consideră cererea servită şi apoi verifică dacă există o ieşire din cel mai rău caz posibil, adică atunci când toate procesele îşi cer maximul de resurse. Aceasta se face găsind rând pe rând câte un proces căruia i se poate servi maximul de resurse solicitat, şi apoi simulându-i terminarea. Dacă folosind acest procedeu, se ajunge la terminarea tuturor proceselor, înseamnă că servirea cererii nu implică riscul unui impas.

In cele ce urmează vom oferi o implementare a algoritmului bancherului. Pentru simplitate vom considera că sistemul de operare dispune de patru tipuri de resurse şi că asupra acestor resurse vor acţiona cinci procese. Implementarea conţine şi date de test pentru care algoritmul poate fi verificat.

```
#include<stdio.h>

#define PROCESE 5
#define RESURSE 4


// Date initiale care descriu alocarea de resurse din sistem
int resurseExistente[PROCESE] = {7, 7, 6, 5};

int maxResursePerProces[PROCESE][RESURSE] = {{3, 2, 4, 2},  // P0
                                             {3, 2, 3, 1},  // P1
                                             {4, 1, 2, 3},  // P2
                                             {3, 3, 4, 4},  // P3
                                             {6, 5, 5, 1}}; // P4

int resurseAlocatePerProces[PROCESE][RESURSE] = {{1, 1, 0, 1},  // P0
                                                 {1, 0, 1, 0},  // P1
                                                 {1, 0, 1, 1},  // P2
                                                 {0, 1, 0, 2},  // P3
                                                 {0, 1, 1, 0}}; // P4

void tiparesteStare(int procesAles,
                    int proceseImplicate[PROCESE],
                    int resurseDisponibile[RESURSE],
                    int alocari[PROCESE][RESURSE]) {
  int i, j;

  printf("\nProcese implicate: ");
  for(i=0; i<PROCESE; i++) {
    if(proceseImplicate[i] == 1) {
      printf("P%d  ", i);
    }
  }

  printf("\nProces ales: P%d\n", procesAles);

  printf("Cereri maxime de resurse ale lui P%d:", procesAles);
  for(j=0; j<RESURSE; j++) {
    printf("  %d", maxResursePerProces[procesAles][j]);
```

```
  }

  printf("\nResurse disponibile:");
  for(j=0; j<RESURSE; j++) {
    printf("  %d", resurseDisponibile[j]);
  }

  printf("\nResurse alocate lui P%d:", procesAles);
  for(j=0; j<RESURSE; j++) {
    printf("  %d", alocari[procesAles][j]);
  }
  printf("\n");
}

int verificaCerere(int solicitant, int *cerere) {
  // Variabile de lucru
  int resurseDisponibile[RESURSE];
  int proceseImplicate[PROCESE] = {1, 1, 1, 1, 1};
  int alocari[PROCESE][RESURSE];
  int i, j, procesAles, gasit, n;

  // Copiem tabela de alocariare de resurse la procese si adaugam la copie
  // noua cerere
  for(i=0; i<PROCESE; i++) {
    for(j=0; j<RESURSE; j++) {
      alocari[i][j] = resurseAlocatePerProces[i][j];
      if(i == solicitant) {
        alocari[i][j] += cerere[j];

        // Procesul cere mai multe resurse decat maximul permis
        if(alocari[i][j] > maxResursePerProces[i][j])
          return -1; //Sfarsit algoritm
      }
    }
  }

  while(1) {
    // Calculam cantitatile de resurse disponibile
    for(j=0; j<RESURSE; j++) {
      resurseDisponibile[j] = resurseExistente[j];
      for(i=0; i<PROCESE; i++) {
        resurseDisponibile[j] -= alocari[i][j];

        // Insuficiente resurse disponibile. Starea curenta conduce la impas
        if(resurseDisponibile[j] < 0)
          return 0; // Sfarsit algoritm
      }
    }

    // Calculam numarul de procese inca implicate in algoritm
    n = 0;
    for(i=0; i<PROCESE; i++) {
      n += proceseImplicate[i];
    }

    // Toate procesele au trecut cu bine prin analiza. Cererea va fi servita
    if(n == 0)
      return 1; // sfarsit algoritm

    // Cautam un proces pentru care avem suficiente resurse pentru a-i
    // servi cererea maxima
    procesAles = -1;
    for(i=0; i<PROCESE; i++) {
```

```
      if(proceseImplicate[i] != 1)
        continue;

      gasit = 1;
      for(j=0; j<RESURSE; j++) {
        if(maxResursePerProces[i][j]-alocari[i][j] > resurseDisponibile[j]) {
          gasit = 0;
          break;
        }
      }
      if(gasit == 1) {
        procesAles = i;
        break;
      }
    }

    // Nici unul dintre procesele implicate in algoritm nu poate fi
    // servit cu maximul posibil, deci exista posibilitatea unui impas
    if(procesAles == -1)
      return 0; // sfarsit algoritm

    tiparesteStare(procesAles,proceseImplicate, resurseDisponibile, alocari);

    // Simulam faptul ca procesul procesAles ajunge sa se termine cu bine,
    // adica resursele ii sunt dealocariate si este marcat ca neimplicat in
    // algoritm
    proceseImplicate[procesAles] = 0;
    for(j=0; j<RESURSE; j++)
      alocari[procesAles][j] = 0;
  }
}

int main() {
  int cerere[RESURSE] = {1, 2, 2, 1};
  int solicitant = 1;

  switch(verificaCerere(solicitant, cerere)) {
    case -1:
      printf("\n\nEROARE. Procesul cere mai mult decat maximul declarat.\n");
      break;
    case  0:
      printf("\n\nCererea nu poate fi servita fara risc de impas. "
             "Procesul va fi pus in asteptare.\n");
      break;
    case  1:
      printf("\n\nCererea poate fi servita fara risc de impas.\n");
      break;
  }

  return 0;
}
```

Figura 9.17 Implementarea algoritmului bancherului

Pentru datele de test din figura 9.17 algoritmul decide că cererea poate fi servită, eliminând toate procesele în următoarea secvenţă: P1, P0, P2, P3, P4. Dacă însă în tabelul `resurseAlocatePerProces` se modfică alocarea resursei `R2` pentru procesul `P2` din 1 in 2, algoritmul nu va servi cererea din cauza apariţiei riscului de impas. Lăsăm pe seama cititorului să ruleze programul din fig. 9.17 şi să vadă rezultatele obţinute.

9.2 Conceptul de multiprogramare.

Cel mai important concept, introdus mai întâi la sistemele seriale şi preluat apoi la toate sistemele de calcul moderne, este conceptul de *multiprogramare*. El reprezintă modul de exploatare a unui sistem de calcul cu un singur procesor central, care presupune existenţa simultană în memoria internă a mai multor procese care se execută concurent. Multiprogramarea duce la o mai bună utilizare a procesorului şi a memoriei.

Primele sisteme de calcul ce lucrează în multiprogramare au apărut cu ceva mai înainte de anul 1965. Tehnologic, în aceeaşi perioadă au apărut plăcile cu circuite integrate. De acum se poate vorbi de generaţia a III-a de calculatoare.

Pe scurt, lucrul în multiprogramare se desfăşoară astfel: în fiecare moment procesorul execută o instrucţiune a unui proces. Vom spune că acest proces este în starea **RUN**. Restul proceselor, fie că aşteaptă apariţia unui eveniment extern (terminareaunei operaţii I/O, scurgerea unui interval de timp etc.), fie că sunt pregătite pentru a fi servite, în orice moment, de către procesor. Despre cele care aşteaptă un eveniment extern spunem că sunt în starea **WAIT**, iar cele care sunt pregătite de execuţie spunem că sunt în starea **READY**.

Pentru ca un sistem de operare să poată lucra în multiprogramare, este necesar ca:
Sistemul de calcul să dispună de un sistem de întreruperi pentru a semnala apariţia evenimentelor externe;
Sistemul de operare să gestioneze, să aloce şi să protejeze resursele între utlizatori. Prin resurse înţelegem: memoria, dispozitivele periferice, fişierele, timpul fizic etc.

9.2.1 Trecerea unui proces dintr-o stare într-alta

Trecerea unui proces din starea **RUN** în starea **WAIT** este (evident) comandată de către procesul însuşi; el (procesul) ştie, în funcţie de rezultatele obţinute până în prezent, când trebuie să urmeze o instrucţiune de I/O.

Trecerea proceselor din starea **RUN** în starea **READY** şi invers poate fi comandată sau de proces (cedare voluntară de procesor) sau poate fi comandată de către sistemul de operare (cedare involuntară de procesor).

Trecere unui proces dintr-o stare într-alta este făcută pe baza unui *algoritm de planificare* rulat de o componentă a sistemului de operare numită *planificator*. Planificatorul este o componentă centrală a oricărui sistem de operare modern.

9.2.1.1 Cedarea voluntară a procesorului

Această modalitate de cedare a procesorului se bazează în general pe programatorul care dezvoltă programul să introducă instrucţiuni de cedare în codul sursă. In alte situaţii sistemul de calcul în sine introduce instrucţiuni de cedare printre instrucţiunile binare ale programului. Un planificator care foloseşte această metode se numeşte *nepreemptiv*.

Cedarea voluntară a procesorului este o practică din vremurile de început ale sistememlor de operare. Deşi simplifică mult munca planificatorului, acest gen de cedare dă unui proces posibilitatea de a bloca întregului sistem de operare (o problemă foarte serioasă care se poate

remedia doar prin re-startarea sistemului de calcul). Pentru a bloca sistemul, un proces nu trebuie decât să ruleze un ciclu infinit în care să nu facă nici o cerere de resurse care să-l întrerupă şi nici să nu apeleze instrucţiuni de cedare a procesorului.

9.2.1.2 Cedarea involuntară a procesorului

Cedarea involuntară a procesorului presupune intervenţia planificatorului pentru a întrerupe procesul curent. Un astfel de planificator se numeşte *preemptiv*. Intreurperea procesului curent se face în general folosind o întrerupere activată de sistemul de calcul la intervale constante (în general) de timp. Procedura de servire a întreruperii este aceeaşi pentru toate procesele şi cheamă instrucţiunea de cedare a procesorului. In urma acestei cedări, planificatorul sistemului având o proritate absolută, preia procesorul (devine activ) şi decide care dintre procesele existente va primi procesorul.

9.2.2 Funcţionarea unui planificator

In fig. 9.18 este prezentat algoritmul după care funcţionează un planificator. In el se consideră că sunt n procese, iar priorităţile lor sunt numerele 1, 2, ..., ***n***, cu 1 prioritatea ca mai mică şi ***n*** prioritatea ca mai mare.

Fiecare proces este încărcat într-o *partiţie* de memorie. Fiecare partiţie are un *număr de prioritate*. In fig. 9.19 se prezintă o posibilă evoluţie a trei procese în regim de multiprogramare. Fiecare dintre cele trei procese este încărcat într-o partiţie proprie.

Procesului **P3**, are nevoie să efectueze până la terminare două operaţii I/O şi are cea mai mare prioritate dintre cele trei procese. Procesul **P2** are nevoie de trei operaţii I/O. Procesului **P1**, cel mai puţin prioritar, are nevoie să efectueze până la terminare o singură operaţie I/O. Presupunem că cele trei procese intră în sistem în acelaşi timp. Desenul ilustrează pe orizontală axa timpului, iar porţiunile în care este scris "**RUN**" indică faptul că procesorul execută succesiv instrucţiuni maşină ale procesului respectiv. In funcţie de momentele în timp în care cele trei procese solicită operaţii de I/O, fig. 9.19 ilustrează 12 momente eveniment, momente în care se schimbă starea unora dintre cele trei procese.

```
do {
  for(i=n; i>=1; i--) {
    if(<Pi este în starea RUN>) {
      EXECUTA(<urmatoarea instructiune masina din Pi>);
      continue;
    } else if(<Pi este în starea READY>) {
      // Exista, în mod sigur, cel putin un proces Pj cu i>j în starea RUN
      STARE(Pj) = READY;
      STARE(Pi) = RUN;
      EXECUTA(<urmatoarea instructiune masina din Pi>);
      continue;
    }
  }
} while(true);
```

Figura 9.18 Implementarea unui planificator

Observăm că **P1** din cauza priorităţii mici, reuşeste să intre în starea **RUN** numai când **P2** şi **P3** sunt în starea **WAIT** sau **FINISH** (procesul s-a terminat). Acestea sunt momentele 2, 6, 8 şi 11. Să mai observăm, de asemenea, că sistemul de calcul este bine exploatat, singura perioadă de timp în care procesorul este neîncărcat este între momentele 7 şi 8. Acesta este de fapt scopul principal al multi programării: valorificarea procesorului de către alt proces câtă vreme cel curent aşteaptă un eveniment extern.

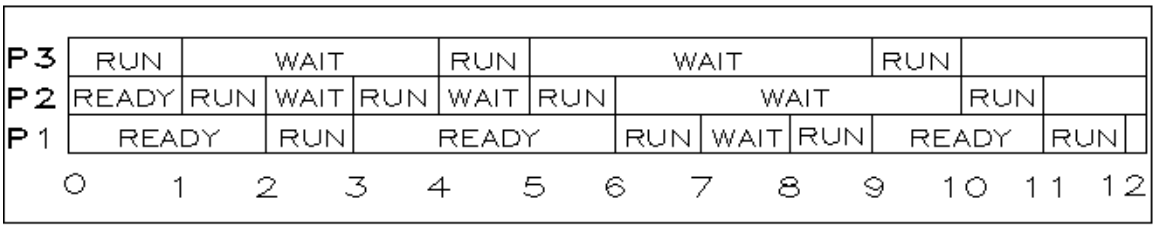

Figura 9.19 Exemplu de evoluţie în multiprogramare

Evoluţiile din fig. 9.19 descriu o situaţie particulară. Astfel, toate procesele sunt lansate în execuţie simultan, iar după terminarea lui **P**1 şi **P**2 nu se mai lansează nimic. In realitate, fiecare partiţie primeşte spre execuţie un nou proces (dacă există) după terminarea celui curent. Deci, spre exemplu, la momentul 10 în partiţia 1 se va lansa un nou proces.

9.3 Planificarea proceselor

9.3.1 Sarcinile planificatorului de procese.

Planificatorul de procese este responsabil pentru trecerea proceselor din starea **READY** în starea **RUN** şi invers. Pentru aceasta panificatorul trebuie să îndeplinească următoarele sub-sarcini:

- ţinerea evidenţei tuturor proceselor din sistem;
- alegerea procesului căruia i se va atribui procesorul şi pentru cât timp;
- alocarea procesului un procesor;
- eliberarea procesorului la ieşirea procesului din starea **RUN**.

Pentru fiecare proces din sistem planificatorul întreţine o strctură de date numită Process Control Block (**PCB**) în care trebuie să apară, printre altele, următoarele informaţii:

- starea procesului;
- pointer la următorul **PCB** în aceeaşi stare;
- numărul procesului (acordat de planificator);
- contorul de proces (adresa instrucţiunii maşină care urmează a fi executată);
- zonă pentru copia regiştrilor generali ai maşinii;
- limitele zonei (zonelor) de memorie alocate procesului;
- lista fişierelor deschise etc.

Toate structurile **PCB** sunt memorate într-o zonă de memorie proprie planificatorului de procese. Starea unui proces poate fi **RUN**, **READY**, **WAIT** sau **SWAP** (vezi 8.2.1.1). Procesele din aceeaşi stare sunt înlănţuite între ele.

Deoarece toate procesele folosesc regiştrii generali ai procesorului, este necesar ca la ieşirea din starea **RUN** conţinutul acestora să fie salvat. La o nouă intrare în **RUN** se reface conţinutul regiştrior. Spre exemplu, în fig. 9.20 este prezentat un schimb **READY** ↔ **RUN** între două procese **A** şi **B**. Evident, la trecerea din B în **A** operaţiile se inversează.

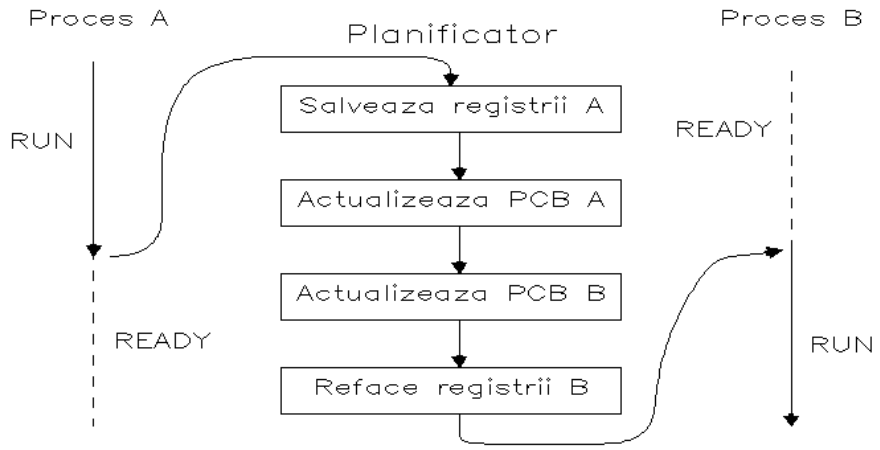

Figura 9.20 Schimbul RUN ↔ READY între două procese

In sarcina planificatorului de procese stă şi organizarea cozilor la perifericele de I/O. Deoarece mai multe procese pot solicita simultan accesul la un periferic, planificatorul întreţine cozi de aşteptare pentru toate perifericele. In fig. 9.21 este exemplificat un ansamblu de cozi proprii planificatorului de procese. Cozile **MM0**, **MM1**, **DKO** şi **TTO** sunt cozi de aşteptare asociate cu perifericele cu acelaşi nume. Coada **READY** conţine procese aflate în starea **READY** (pregătite pentru a intra în execuţie în orice moment). Coada **RUN** conţine procese aflate în exeuţie deja. Cum un singur proces poate fi în execuţie pe un procesor, se deduce că fig 9.21 se referă la un sistem de calcul bi-procesor.

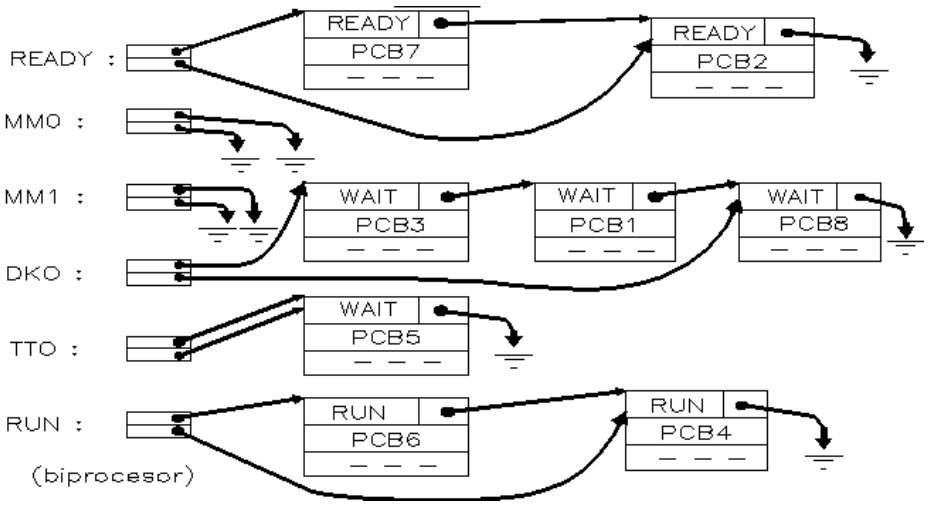

Figura 9.21 Cozi gestionate de planificatorul proceselor

9.3.2 Algoritmi de planificare.

O mare parte a literaturii a pus un accent pe algoritmii de planificare. Iată numai câteva lucrări de referinţă în domeniul sistemelor de operare, care tratează pe larg această problemă: [4], [10], [45], [49]. Motivele principale pentru care s-a dat importanţă mare planificării sunt că performanţa unui sistem de operare este influenţată decisiv de funcţionarea planificatorului şi nu în ultimul rând pentru că planificare se pretează uşor la modelarea matematică.

Alte lucrări, cum ar fi [22], [29], enumeră foarte pe scurt câţiva algoritmi, trecând repede peste acest capitol. Punctul de vedere al celor din urmă este justificat prin faptul că *ponderea planificării în sistemele de operare este destul de mică.* In cele ce urmează, enumerăm cei mai utilizaţi algoritmi de planificare a proceselor.

9.3.2.1 FCFS (First Come First Served)

Numele alogritmului însemna în traducere directă înseamnă *primul venit - primul servit*. Este cunoscut şi sub numele de *FIFO (First In First Out)*. Procesele sunt servite în ordinea lor cronologică. Este un algoritm simplu, dar nu prea eficient.

Spre exemplu, presupunem că există 3 procese cu următorii timpi de execuţie:

- procesul 1 durează 24 minute;
- procesul 2 durează 3 minute;
- procesul 3 durează 3 minute.

Dacă acestea sosesc la sistem în ordinea 1, 2, 3, atunci timpul mediu de servire este:

$$\frac{24 + 27 + 30}{3} = 27 \text{ minute}$$

Dacă sosesc în ordinea 2, 3, 1 timpul mediu de servire este:

$$\frac{3 + 6 + 30}{3} = 13 \text{ minute}$$

9.3.2.2 SJF (Shortest Job First).

Se execută primul, jobul (procesul) care consumă cel mai puţin timp procesor. Probabil că acest algoritm este cel mai bun, însă are un mare dezavantaj: presupune cunoaşterea exactă a timpului procesor necesar execuţiei procesului.

9.3.2.3 Algoritmul bazat pe priorităţi.

Este algoritmul cel mai des folosit. Toţi ceilalţi algoritmi descrişi în literatură sunt cazuri particulare ale acestuia. Fiecare proces primeşte un număr de prioritate. Procesele se ordonează după aceste priorităţi, apoi se execută în această ordine.

Se disting două metode de stabilire a priorităţilor:

1. procesul primeşte prioritatea la intrarea în sistem şi şi-o păstrează până la sfârşit;
2. Sistemul de operare calculează priorităţile după reguli proprii şi le ataşează, dinamic, proceselor în execuţie.

Varianta *1*) este folosită destul de des dar are dezavantajul că dacă apar multe procese cu prioritate mare, atunci cele cu prioritate mică aşteaptă practic indefinit. Acest fenomen este cunoscut sub numele de *starvation*. Pentru a evita acest fenomen, de obicei cele două metode sunt combinate. Astfel, unui proces cu prioritate mică i se va mări prioritatea cu cât aşteaptă mai mult şi se readuce la valoare iniţială după ce intră în execuţie.

9.3.2.4 Algoritm bazat pe termene de terminare (deadline scheduling)

Acest algoritm de planificare este folosit în principal la sistemele cu cerinţe de timp real foarte stricte. In asemenea sisteme este critic pentru un task sau proces să se termine la timp. Astfel fiecărui task i se ataşează un termen de terminare. Un caz special sunt task-urile repetitive cărora li se asociază termene de terminare asociate fiecărei iteraţii. Planificatorul foloseşte aceste termene pentru a decide când şi cărui task şi pentru cât timp să-i acorde procesorul în aşa fel încât să se temrine la timp. Evident pentru acest gen de planificare este esenţială cunoaşterea exactă a duratei fiecărui task. La apariţia unui nou task în sistem, planificatorul îl va accepta pentru execuţie doar dacă poate găsi o alocare prin care termenele noului task să fie respectate fără a le afecta pe cele ale task-urilor deja existente.

O strategie posibilă pentru calcularea secvenţei de execuţie a task-urilor este aducerea în stare **RUN** întotdeauna a task-ului cu termenul de terminare cel mai apropiat.

9.3.2.5 Round-Robin (planificare circulară).

Aceşti algoritmi sunt destinaţi în special sistemelor de operare care lucrează în timesharing. Pentru realizare, se defineşte o *cuantă de timp,* de obicei între 10-100 milisecunde. Coada READY a proceselor este tratată circular. Pe durata unei cuante se alocă procesorul unui proces. După epuizarea acestei cuante, procesul este trecut la sfârşitul cozii, al doilea proces este preluat de către procesor ş.a.m.d.

Există şi *variante de Round-Robin.* Spre exemplu, dacă un proces nu şi-a consumat în întregime cuanta (spre exemplu din cauza unei operaţii de I/O), atunci locul lui în coada **READY** este invers proporţional cu partea consumată din cuantă. De exemplu, dacă şi-a consumat toată cuanta, atunci procesul trece la sfârşitul cozii; dacă şi-a consumat numai jumătate din cuantă, atunci el trece la mijlocul cozii etc.

O altă variantă urmăreşte echilibrarea sistemului folosind aşa-zisul *Round-Robin cu reacţie*. Când un proces nou este acceptat, el se rulează mai întâi atâtea cuante câte au consumat celelalte procese, după care trece la planificarea Round-Robin simplă.

9.3.2.6 Algoritmul de cozi pe mai multe nivele

Acest algoritm se aplică atunci când lucrările pot fi clasificate uşor în grupe distincte. Spre exemplu, la un sistem de operare de tip mixt, interactiv şi serial, pot fi create 5 cozi distincte, înşirate mai jos, de la cea mai prioritară, la cea mai puţin prioritară:

Taskuri sistem.
Lucrările interactive.
Lucrări în care se editează texte.
Lucrări seriale obişnuite.
Lucrări seriale ale studenţilor.
...