

9 Teoria generală a sistemelor de operare

9.1 Procese

9.1.1 Conceptul de proces

Facilitatea de a rula mai multe programe simultan în cadrul unui sistem de operare este considerată astăzi normală de către toți utilizatorii. Rularea unui navigator web (Mozilla Firefox, Internet Explorer, Safari, Konqueror etc.) simultan cu rularea unui program pentru citirea poștei electronice (Mozilla Thunderbird, Outlook Express, Eudora, Pine etc.) este o practică de zi cu zi a majorității 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 activității procesorului care pur și simplu execută instrucțiunile care i se transmit fără a face diferențiere între procesele cărora le aparțin. Revenind la exemplul inițial, un procesor nu “știe” dacă instrucțiunea executată aparține de Mozilla Firefox sau Outlook Express.

În terminologia actuală există o tendință de confuzie între noțiunea de proces și cea de program. Deși folosirea noțiunii de program pentru a ne referi la un proces nu creează în general confuzii, considerăm totuși necesară clarificarea situației. *Procesul are un caracter dinamic*, el precizează o secvență de activități în curs de execuție, iar *programul are un caracter static*, el numai descrie această secvență de activități. Cu alte cuvinte procesul este un program în execuție.

Existența unui proces este condiționată de existența a trei factori:

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

Execuția unui program (și implicit transformarea acestuia în proces) implică execuția de către procesor a fiecărei instrucțiuni din programul (procedura) respectiv(ă). În mod inerent, aceste instrucțiuni 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 menționat că în arhitecturile actuale monoprocessor (vezi [52]), mai multe procese care rulează concomitent sunt de fapt deservite alternativ de către procesor. El (procesorul) execută alternativ grupuri de instrucțiuni 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 (și așa de mică) încât momentele de staționare ale unui program nu sunt sesizabile de către utilizator. Astfel se creează impresia de simultaneitate în execuție.

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

9.1.2 Concurența între procese

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

- memoria (cereri de alocare de memorie)
- discul magnetic (acces la fișiere)
- terminalul (pentru interacțiunea cu utilizatorul)
- interfața de rețea (pentru interacțiunea cu alte mașini legate prin rețea)

9.1.2.1 Secțiune critică; resursă critică; excludere mutuală.

O problemă legată de accesul la resurse este asigurarea corectitudinii operațiilor executate în regim de concurență. Pentru a vedea cum operații corecte pot da rezultate greșite în regim de concurență, vom analiza un exemplu clasic. Să considerăm un fișier care stochează numărul de locuri libere într-un sistem de vânzare de bilete. Vânzarea unui bilet va avea ca și consecință decrementarea numărului de locuri libere. De asemenea, să considerăm că există două procese care efectuează simultan operații asupra acestui fișier. Fragmentul de cod necesar pentru decrementarea numărului de locuri este prezentat în 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 secvență de decrementare

În situația în care codul din fig. 9.1 este rulat simultan de două sau mai multe procese, și considerând faptul că procesorul execută alternativ grupuri de instrucțiuni din procese diferite, este posibil ca instrucțiunile 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

În mod evident, ambele procese citesc *aceeași valoare* din fișier, o decrementează și apoi o scriu la loc. Ca urmare, în final, în loc ca numărul de locuri să scadă cu două poziții, va scădea doar cu una și ca urmare operațiile executate asupra fișierului deși sunt greșite. În viața de zi cu zi o asemenea eroare ar conduce la vânzarea aceluiași loc în sală de două ori (o "bucurie" cu care mulți dintre cititori au avut probabil de-a face).

Trebuie remarcat că eroarea poate să apară doar la o execuție concurentă, altfel secvența de cod fiind perfect corectă. Eroarea apare din lipsa de sincronizare între accesele proceselor la fișier. Procesul B, în loc să opereze asupra numărului de locuri decrementat de procesul A, reușește să citească numărul de locuri prea devreme și va opera asupra *aceluiași număr de locuri* ca și procesul A. Pentru a obține rezultate corecte, secvența de execuție dorită este cea din fig. 9.3. Adică secvența de cod care decrementează numărul de locuri să fie executată la un moment dat doar de un singur 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 secțiune critică

Același gen de situație se creează și 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 același bloc pe disc pentru a scrie în el date aparținând la fișiere diferite. Este ușor de imaginat ce s-ar întâmpla dacă într-un script shell editat cu un editor de text ar apărea părți din conținutul unui fișier ".o" generat de gcc la compilare ...

Revenind la problema noastră, vom spune că porțiunea de program din fig. 9.1 începând cu read și terminând cu write este o **secțiune critică**, deoarece nu este permis ca ea să fie executată simultan de către două procese. Analog, vom spune că poziția pe care se stochează numărul de locuri în fișier este o **resursă critică**, deoarece nu poate fi accesată simultan de mai multe procese. În sfârșit, pentru a evita situația eronată de mai sus vom spune că procesele A și B trebuie să se *excludă reciproc*, deoarece trebuie să aibă acces exclusiv la secțiunea și la resursa critică [22].

Problema secțiunii critice a suscitat un interes deosebit în istoria SO. Vom expune și noi câteva aspecte ale ei, pentru a se vedea câteva dintre "subtilitățile" programării concurente. Să vedem mai exact cerințele problemei:

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

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

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

Figura 9.4 Un program cu secțiune critică

Au existat multe încercări de rezolvare a problemei secțiunii critice. În [4] sunt date și comentate trei încercări de descriere, fiecare dintre ele încălcând unele dintre condițiile puse mai sus. Matematicianul Dekker a fost primul (1965) care a dat o soluție software (rezonabilă, care însă este destul de complicată și încălciată). În 1981 Peterson a dat o soluție ceva mai simplă. Aceste soluții se referă la rezolvarea secțiunilor critice apărute la concurența în cadrul aceluiași proces (un subiect pe care nu l-am abordat încă). Totuși, pentru a evidenția complexitatea problemei, prezentăm soluția lui Peterson în fig 9.5 și lăsăm cititorului "plăcerea" de a demonstra că soluția este corectă.

Cod inițial	<pre>int c1 = 1; int c2 = 1; int schimb;</pre>	
Cod executat în paralel	<pre>do { c1 = 0; schimb = 1; while((c2 == 0) && (schimb == 1)){ <așteaptă>; } <secțiune critică>; c1 = 1; <rest program 1> } while(false)</pre>	<pre>do { c2 = 0; schimb = 2; while((c1 == 0) && (schimb == 2)){ <așteaptă>; } <secțiune critică>; c2 = 1; <rest program 2> } while(false)</pre>

Figura 9.5 Soluția Peterson pentru secțiune critică

Evident, rezolvarea nu este simplă. Dacă se cere ca secțiunea critică să fie accesibilă la mai multe procese, soluția din fig. 9.5 nu poate fi generalizată prea ușor. Mai mult, soluțiile software presupun o *așteptare activă*, adică toate procesoarele, în faza de așteptare execută în mod repetat o aceeași instrucțiune (ciclează încontinuu). Ar fi de dorit ca pe timpul cât un proces așteaptă să primească dreptul de execuție a secțiunii critice, procesorul lui să devină disponibil spre a servi alte procese. O asemenea soluție poate fi realizată folosind conceptul de *semafor* descris în secțiunea următoare.

9.1.2.2 Conceptul de semafor.

Pentru a evita situația de mai sus și pentru a permite o serie de operații 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 așteptare. Valoarea $v(s)$ este un număr întreg, care primește o valoare inițială $v_0(s)$. Coada de așteptare conține (pointeri la) procesele care așteaptă la semaforul s . Inițial coada este vidă, iar disciplina cozii depinde de sistemul de operare (FIFO, LIFO, priorități etc.).

Pentru gestiunea semafoarelor se definesc două operații indivizibile $P(s)$ și $V(s)$ ale căror roluri, exprimate laconic, sunt "a trece de resursă" și "a anunța eliberarea resursei". (Denumirile operațiilor au fost date de Dijkstra după primele litere din limba olandeză: P de la *prolaag* prescurtare de la *probeer te verlagen* – încercare de a descrește, respectiv V de la *verhoog* – a crește). În unele lucrări [4], operația **P** se mai numește **WAIT**, iar operația **V** se

mai numește **SIGNAL**. Indivizibilitatea operațiilor înseamnă că ele nu pot fi întrerupte în cursul execuției lor, deci nu pot exista două procese care să execute simultan $P(s)$ sau $V(s)$, sau simultan două operații P , sau simultan două operații V .

Definițiile exacte ale operațiilor P și V , apelate de un proces A și aplicate asupra unui semafor sunt date în fig. 9.6.

<pre>// 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 așteptare <Trece controlul la DISPECER>; } else { <Trece controlul la procesul A>; }</pre>	<pre>// 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>; }</pre>
---	--

Figura 9.6 Operațiile $P(s)$ și $V(s)$ apelate de procesul A

Pentru a sugera funcționarea (și denumirea) semaforului vom considera un exemplu concret. Fie G1 și 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 inițială: $v_0(s) = n$

Procesul de trecere:

$P(s); \quad \text{<Trenul trece pe una din cele } n \text{ linii>;} \quad V(s);$

Să notăm cu $np(s)$, $nv(s)$, numărul de primitive $P(s)$, respectiv $V(s)$ efectuate până la un moment dat, $v_0(s)$ valoarea inițială a semaforului s , iar $nt(s)$ numărul proceselor care au trecut de semaforul s . Următoarele proprietăți au loc:

1. $v(s) = v_0(s) - np(s) + nv(s)$;
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. $nt(s) = \min(v_0(s) + nv(s), np(s))$

Demonstrația acestor proprietăți este simplă, prin inducție după numărul operațiilor P și V efectuate asupra semaforului. Aspectul critic al semaforului este că operațiile asupra lui (P și V) sunt atomice. Astfel două procese nu le pot executa în același timp asupra aceluiași semafor s .

Folosirea semafoarelor rezolvă complet și elegant problema secțiunii critice prezentată în secțiunile anterioare. Rezolvarea presupune folosirea unui singur semafor s și este corectă indiferent de numărul de procese care folosesc în comun secțiunea critică. Descrierea soluției este dată în fig. 9.7.

Cheia este tocmai semaforul s . Dacă $v_0(s) = 1$ și toate procesele care folosesc secțiunea critică sunt de forma:

$...; P(s); \text{ <secțiune critică>; } V(s); ...$

atunci se demonstrează ușor că $v(s) < 1$, indiferent de numărul proceselor care operează asupra lui s cu operații **P** sau **V**. Drept urmare, în baza proprietăților menționate mai sus, un singur proces poate trece de acest semafor, deci sînt îndeplinite condițiile cerute pentru a proteja o secțiune critică. Un astfel de semafor mai poartă numele de *semafor de excludere mutuală*. Sincronizarea este o operație fundamentală în programarea concurentă. În secțiunile care urmează vom întâlni mai multe aplicații ale ei.

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

Figura 9.7 Soluția cu semafor pentru secțiune critică

9.1.2.3 Problema producătorului și a consumatorului.

Să presupunem că există unul sau mai multe procese numite *producătoare*, și unul sau mai multe procese numite *consumatoare*. De exemplu, conceptele de pipe și FIFO sunt de această natură (vezi 5.3.2 și 5.3.4). Transmiterea informațiilor 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 aștepte când bufferul este gol;
- producătorii să aștepte când bufferul este plin.

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

Producător	Consumator
semaphore plin, gol, exclus; v0(plin) = 0; v0(gol) = n; v(exclus) = 1;	
do { <produce articol>; P(gol); P(exclus); <depone 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 și a consumatorului rezolvată cu semafoare

9.1.2.4 Regiuni critice condiționate.

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

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

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

```
resource  $r$  ::  $v_1, v_2, \dots, v_n$ 
```

unde r este numele resursei, iar v_1, \dots, v_n sunt numele variabilelor de protejat.

O *regiune critică condiționată* se definește astfel:

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

unde r este numele unei resurse declarate ca mai sus, B este o expresie booleană, iar S este secvența de instrucțiuni corespunzătoare regiunii critice. Dacă este prezentă opțiunea *when*, atunci S este executată numai dacă B este adevărată. Variabilele din resursa r pot fi folosite numai de către instrucțiunile din secvența de instrucțiuni S .

Descrierea prin semafoare a unei regiuni critice condiționate este dată în fig. 9.9. Sunt necesare două semafoare și un număr întreg. Semaforul *sir* reține în coada lui toate procesele care solicită acces la regiune. Semaforul *exclus* asigură accesul exclusiv la anumite secțiuni ale implementării (în special la modificarea numărului întreg *nr*). Întregul *nr* reține câte procese au cerut acces la regiune, la un moment dat. În secțiunea următoare vom da o aplicație interesantă a regiunii critice condiționate.

În fig 9.9 algoritmul propriu-zis începe de la linia 9. Dacă regiunea critică este folosită fără o condiție asociată, se va trece direct la execuția lui S . Altfel, incrementând variabila *nr*, se marchează faptul că încă un proces accesează regiunea critică cu o condiție asociată. Cum mai multe procese pot face acest lucru simultan, incrementarea trebuie protejată. Deși la prima vedere este doar o singură operație, în realitate incrementarea se traduce într-o secvență de instrucțiuni binare care pot fi executate într-o ordine similară cu cea din fig 9.2 și poate conduce la erori. Ca urmare, înainte ca incrementarea să aibă loc, se execută **P(exclus)** care dacă condiția B este satisfăcută va proteja și execuția lui S . Altfel, dacă condiția B nu este îndeplinită procesul trebuie să intre în așteptare. 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. În acest fel concurența pentru

evaluarea condiției B ar fi. Scopul nostru însă nu este serializarea evaluării condiției B și a secvenței de instrucțiuni S , ci serializarea accesului la S în momentul în care B este satisfăcută. Ca urmare trebuie să lăsăm fiecare proces să evalueze B fără restricții. Dacă condiția B este îndeplinită, se decrementează nr (marcând astfel intrarea în regiunea critică) și se execută instrucțiunile din S . Apoi (liniile 20-21), semaforul sir este incrementat până la zero în așa fel încât controlorul va alege unul din procesele din coadă pentru a fi executat. În final semaforul $exclus$ este eliberat la linia 22. Procesul proaspăt reactivat va reevalua condiția B și dacă este îndeplinită va continua și va executa instrucțiunile 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 condiționate

Până în acest moment lucrurile ar părea clare, dar există o situație în care implementarea de mai jos nu funcționează! Este vorba de cazul în care toate procesele accesează regiunea critică condiționată în momentul în care B nu este satisfăcută și ca urmare vor intra în așteptare în coada semaforului sir . Întrebarea este: cum vor ieși aceste procese din coada semaforului? Este necesar ca cineva (un proces) să modifice starea condiției B și apoi să anunțe schimbarea proceselor care așteaptă. Aceasta poate avea loc în două moduri:

1. După câțva timp un alt proces schimbă starea condiției B , intră în regiunea critică și constată că B este satisfăcută. Ca urmare, nu va intra în coada semaforului sir și după execuție lui S îi va incrementa valoarea până la zero (liniile 20-21). În acest moment unul dintre procesele blocate în coada semaforului va fi reactivat și va trece prin secțiunea critică. La rândul lui, acesta va reincrementa semaforul până la zero astfel reactivând un nou proces.
2. Unul dintre procesele care folosesc regiunea critică nu o asociază cu condiția B și ca urmare va omite așteptarea la semaforul sir . La fel ca și în cazul anterior, după execuția 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 întotdeauna va exista un proces care va găsi condiția B satisfăcută, altfel totul se blochează. Varianta a doua este mai puțin riscantă pentru că un proces apelând regiunea critică fără condiție nu va rămâne blocat. Pentru a cuprinde ambele într-un singur program, s-a adăugat condiția din linia 10 cu închiderea ei din linia 18.

9.1.2.5 Problema citirilor și a scrierilor.

Problema a fost formulată în 1971 de către Courtois, Heymans și Parnas [4] [50]. Se presupune că există două tipuri de procese: *cititor* și *scriitor*. Ele partajează împreună o aceeași resursă, să zicem un fișier. Un proces scriitor modifică conținutul fișierului, iar unul cititor consultă informațiile din el. Spre deosebire de problema producătorului și 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.*

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

În secțiunile precedente am văzut rezolvarea diferitelor probleme de programare concurentă, folosind cel mai des operațiile **P** și **V**. Ele sunt suficiente pentru descrierea acestei probleme și invităm cititorul să o facă. În secțiunea precedentă am definit o construcție mai elegantă și mai ușor de urmărit în programe, regiunea critică condiționată.

O posibilă rezolvare, folosind regiuni critice condiționate, este cea din fig. 9.10. Soluția aparține lui P. B. Hansen, 1973, și ea dă prioritate proceselor scriitor față de cele cititor. Un proces scriitor intră în lucru imediat după ce procesele cititor active și-au încheiat citirile curente. Resursa *f* este zona la care doresc accesul cele două tipuri de procese. Presupunem că ea este un fișier, iar operațiile de bază asupra lui sunt *read* și *write*. Resursa *c* conține două contoare care indică câte procese de fiecare tip au cerut acces la resursa *f*. Este de remarcat folosirea regiunii *c* cu și fără o condiție atașată.

Cititor	Scriitor
<pre> int nw, nr; resource f; resource c :: nr, nw; nr = 0; nw = 0; </pre>	
<pre> region c when nw == 0 do nr++; <citește date>; region c do nr--; </pre>	<pre> region c do nw++; region c when nr == 0 do <așteaptă să se termine toți cititorii activi>; region f do <scrie date>; region c do nw--; </pre>

Figura 9.10 Problema citirilor și a scrierilor

9.1.3 Problema impasului

9.1.3.1 Conceptul de impas.

Noțiunea de impas nu este specifică sistemelor de operare sau informaticii, ci este un fenomen întâlnit în multe situații din viața de zi cu zi. În general, impasul poate fi definit ca o blocare a activității normale a două sau mai multe entități ca urmare a efectului lor coroborat asupra mediului.

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

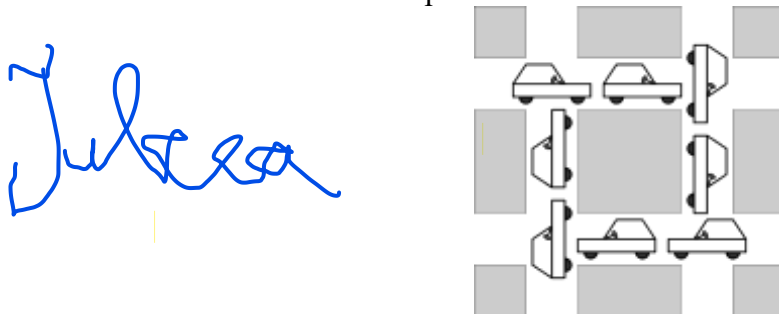


Figura 9.11 Un impas într-o intersecție

Trecând la domeniul informatic, o modalitate ușoară de a crea o situație de impas este invesarea operațiilor **P** și **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); <instrucțiuni A>; P(y); V(y); V(x);	P(y); <instrucțiuni B>; P(x); V(x); V(y);

Figura 9.12 Impas provocat de două semafoare

Dacă instrucțiunile celor două procese sunt executate întâmplător în secvența din figura 9.13, cele două procese **A** și **B** se vor afla într-o stare de impas. Cauza este oarecum similară cu situația traficului prezentată anterior. Procesul **A** va deține semaforul binar **x**, procesul **B** va deține semaforul binar **y**, apoi ambele vor încerca să obțină semaforul deținut de celălalt, fără a elibera semaforul pe care îl deține înainte de a-l obține pe cel deținut de celălalt proces. Situația este ilustrată în fig. 9.13.

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

Figura 9.13 O situație de impas

Cauza principală care a generat această situație este faptul că procesele își ocupă resursele numai atunci când au nevoie efectiv de ele și ca urmare apare situații de așteptare circulară.

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

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

Ultimele două exemple de *impas* au avut ca protagoniști câte două procese și două resurse. Impasul poate apărea însă în situații mai complexe care implică nu doar două procese, ci mai multe și nu neapărat resurse specifice, ci clase de resurse. Să presupunem că într-un anumit moment din funcționare, un sistem de operare dispune de 50MB de memorie liberă și de 10 poziții libere pentru crearea de conexiuni în rețea. În acest moment un proces **A** cere alocarea a 30MB de memorie și apoi a 7 conexiuni în rețea, iar un proces **B** cere întâi alocarea a 5 conexiuni de rețea și 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 că nu implică resurse punctuale ci clase de resurse, în cazul nostru memoria RAM și conexiunile la rețea.

Impasul rezultat din conflictul asupra câtorva resurse punctuale poate fi modelat teoretic pe baza *impasului* pe clase, considerând că fiecare clasă de elemente conține 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. În cazul în care două procese se blochează unul pe altul, doar ele vor fi în *impas* și situația se poate salva terminând forțat unul dintre ele. Dacă însă *impasul* se manifestă între un proces și sistemul de operare sau direct în interiorul sistemului de operare, parțial sistemul "îngheață" și ieșirea cea mai frecventă din asemenea situații este restartarea calculatorului!

Astfel de situații trebuie avute în vedere și prevenite de către proiectanții de sisteme de operare. Dar, după cum vom vedea, tratarea *impasului* este costisitoare și ca urmare proiectanții o tratează doar în situații izolate pentru a nu afecta performanța sistemului.

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

- ieșirea din *impas*,
- detectarea unui *impas*,
- evitarea (prevenirea apariției) *impasului*.

Cele două rezultate de mai sus dau condiții prin care este indicată eventuala apariție a unui *impas*, însă nu furnizează metode practice de rezolvare a problemelor de mai sus. În continuare vom da metode și tehnici de rezolvare a problemelor generate de *impas*.

9.1.3.2 Ieșirea din *impas*.

Soluțiile ieșirii din *impas* pot fi clasificate în manuale și automate. Soluțiile manuale sunt cele care implică acțiuni ale operatorului uman. Cele automate sunt executate de către sistemul de operare.

Ieșirea 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 existența unei

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

Indiferent dacă este automată sau manuală, soluția ieșirii din impas necesită acțiuni drastice. În cele ce urmează prezentăm trei astfel de soluții.

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

O a doua variantă este *alegerea un proces "victimă"*, care este fie cel care a provocat impasul, fie un altul de importanță mai mică, astfel încât să fie înlăturat impasul. Se distruge acest proces și odată cu el toți descendenții lui. Această alegere este o parte importantă a algoritmilor automați de rezolvare a impasului. În cazul abordării manuale, din păcate, criteriul de alegere a "victimei" este de multe ori departe de a fi obiectiv.

O a treia soluție este *creerea unui "punct de reluare"*, care este o fotografie a memoriei pentru procesul victimă și pentru procesele cu care el colaborează. Apoi se distruge procesul victimă și subordonații lui. Procesul victimă se reia ulterior. Crearea punctelor de reluare reclamă consum de timp și complicații, 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 reuși detecția este necesar ca **SO** să aibă o evidență clară, pentru fiecare proces, privind resursele ocupate, precum și cele solicitate dar neprimite. Pentru aceasta, cel mai potrivit model este *graful alocării resurselor*.

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

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

Dacă graful (X, U) definit mai sus este ciclic, atunci sistemul se află în stare de impas. Demonstrația este un exercițiu simplu de inducție. Detectarea ciclicității unui graf se face folosind algoritmi clasici din teoria grafelor.

În fig 9.14 prezentăm un posibil graf de alocare. Nodurile pătrate reprezintă procesele iar nodurile rotunde reprezintă resursele. Pentru o mai ușoară diferențiere arcele de așteptare pentru o resursă au fost desenate cu linie întreruptă.

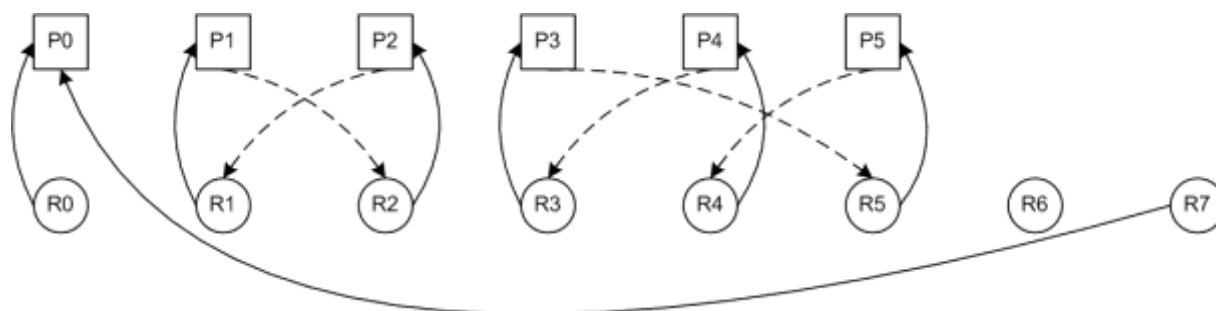


Figura 9.14 Un graf de alocare a resurselor

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

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

9.1.3.4 Evitarea (prevenirea apariției) impasului.

În 1971, Coffman, Elphic și Shoshani [4], [19] au indicat patru condiții *necesare* pentru apariția impasului:

1. procesele solicită controlul exclusiv asupra resurselor pe care le cer (condiția de *excludere mutuală*);
2. procesele păstrează resursele deja ocupate atunci când așteaptă alocarea altor resurse (condiția de *ocupă și așteaptă*);
3. resursele nu pot fi șterse din procesele care le țin ocupate, până când ele nu sunt utilizate complet (condiția de *nepreempție*);
4. există un lanț de procese în care fiecare dintre ele așteaptă după o resursă ocupată de altul din lanț (condiția de *așteptare circulară*).

Evitarea impasului presupune împiedicarea apariției uneia dintre aceste patru condiții. Vom lua fiecare condiție pe rând și vom analiza modalitățile în care ea poate fi împiedicată și efectele asupra funcționării sistemului

Condiția 1 – excluderea mutuală. În absența excluderii mutuale între procese, dispare noțiunea de așteptare după o resursă critică blocată de alt proces. Dispărând noțiunea de așteptare, evident dispare și noțiunea de impas din moment ce nici un proces nu se va opri din execuție. Eliminarea acestei condiții însă nu este o alegere fericită. După cum am demonstrat anterior, excluderea mutuală este esențială pentru pastrarea corectitudinii datelor. Practic impasul a apărut ca și un efect secundar al aplicării acestei condiții indiscutabil necesare.

Condiția 2 – ocupă și așteaptă. Această condiție spune că un proces poate aștepta după o resursă în timp ce deține (blochează) alte resurse. Eliminarea acestei condiții 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 forțată 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 posibilitățile de utilizare a sistemului. Să luăm spre exemplu cazul comun al unui editor de text care suportă deschiderea simultană a mai multor fișiere. 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 fișierelor editate. Efectul acestui comportament are două tăișuri:

- Dacă editorul deschide un singur fișier mic, practic majoritatea memoriei solicitate va fi ținută 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 trebuie să deschidă multe fișiere de dimensiuni mari, va ajunge să ocupe toată memoria alocată și să fie nevoit să refuze deschiderea altor fișiere.

Condiția 3 – nepreemptie. Eliminarea condiției de nepreemptie permite sistemului de operare să aleagă procesele care ar trebui oprite pentru a ieși din impas și chiar să le oprească. Evident simpla terminare a unui proces nu este acceptabilă și ca urmare sistemul trebuie să și repornească acel proces. Deși această abordare rezolvă situația de impas, nu garantează că procesele vor funcționa conform așteptărilor. Teoretic este posibil ca după restartarea unui proces dintr-un grup implicat într-un impas, procesele rămase să nu progreseze suficient pentru a ieși din regiunea critică înainte ca programul repornit să reajungă acolo. În acel moment impasul va reapare și din nou unul dintre procese va fi repornit. Această secvență se poate repeta la nesfârșit și deși nici unul dintre procese nu este blocat, totuși niciunul nu prograsează cu execuția. Acest fenomen este cunoscut sub numele de *impas activ* (livelock).

Condiția 4 – așteptarea circulară. O soluție simplă pentru împiedicarea așteptării circulare este impunerea unei ordini de blocare a resurselor. Sistemul de operare decide o ordine absolută a tuturor resurselor și 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 așezați opt filozofi. Pe masă, în loc să fie șaisprezece tacâmuri (furculițe și cuțite) sunt doar opt, așezate în așa fel încât lângă fiecare farfurie este un cuțit și o furculiță (vezi fig 9.15). Fiecare filozof are nevoie de două tacâmuri pentru a mânca, și odată ce a luat un tacâm de pe masă nu-l va elibera până când nu termină de mâncat. Dacă toți filozofii iau de pe masă tacâmul din stânga, ne aflăm se creează o situație 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. În consecință filozoful G va avea la dispoziție două tacâmuri pentru a mânca. În momentul în care G va termina, va elibera tacâmul 6 și filozoful F va putea mânca și el. Procesul se va continua astfel până când toți filozofii vor fi mâncat.

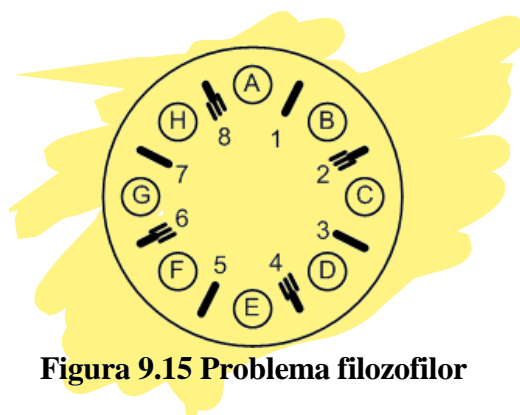


Figura 9.15 Problema filozofilor

În realitate însă, această soluție 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. În 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 complecși sau mai puțin performanți.

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

Evitarea impasului se poate face și impunând o modalitate restrictivă de alocare de resurse. În esență, înainte de a servi o cerere de resurse, sistemul de operare analizează alocările de resurse. Dacă servirea cererii conduce la o stare cu potențial de a genera un impas, cererea de resurse este amânată și reanalizată în momentul în care are loc o modificare în starea alocării de resurse în sistem. Soluțiile 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 cunoașterea pentru fiecare proces a necesităților maxime de resurse din fiecare clasă. Astfel, la încărcare, fiecare proces va transmite sistemului de operare cantitățile de resurse din fiecare clasă de care va avea nevoie în cel mai rău caz.

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

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

Soluția este ineficientă pentru că este foarte puțin probabil ca un proces să folosească imediat după încărcare maximum declarat de resurse. În general resursele sunt folosite treptat și rareori se va atinge maximum. Ca urmare, în majoritatea covârșitoare a cazurilor un proces poate să-și înceapă lucrul fără ca sistemul de operare să poată acoperi întreaga plajă de resurse pe care o poate solicita. Algoritmul bancherului evită impasul fără a impune condiții atât de restrictive.

Algoritmul Bancherului [4], [19] datorat lui Dijkstra este probabil cel mai renumit pentru alocarea controlată. Spre diferență de soluția precedentă algoritmul bancherului permite tuturor proceselor să se încarce și să înceapă execuția. În 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 rămâne blocat la nesfârșit) cu condiția să aștepte până când se mai eliberează resurse.

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

1. Bancherul dispune de o sumă limitată de împrumut, mai mică decât suma creditelor maxime ale clienților
2. Bancherul nu percepe nici dobândă și nici comision
3. Toți clienții returnează banii în cele din urmă

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

- `resurseExistente`
 - Cantitățile de resurse existente în sistem pentru fiecare clasă de resurse.
 - Tabel unidimensional. Fiecărei clase de resurse îi corespunde o poziție
- `maxResursePerProces`
 - Cantitățile maxime de resurse pe care fiecare proces le poate solicita
 - Tabel bi-dimensional. Fiecare linie corespunde unui proces. Fiecare celulă dintr-o linie conține cantitatea corespunzătoare unei clase de resurse.
- `resurseAlocatePerProces`
 - Cantitățile de resurse alocate fiecărui proces
 - Tabel bi-dimensional. Fiecare linie corespunde unui proces. Fiecare celulă dintr-o linie conține 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 poziție
- `resurseDisponibile`
 - Resursele disponibile în sistem
 - Tabel unidimensional. Fiecărei clase de resurse îi corespunde o poziție
- `proceseImplicate`
 - Tablou pentru marcarea proceselor implicate în algoritm
 - Fiecărui proces îi corespunde o poziție

Folosind notațiile de mai sus, algoritmul bancherului este prezentat în limbaj pseudocod în 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. Sfârșit algoritm>;

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

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

<Dacă nu există nici un astfel de proces, înseamnă că cererea inițială
```



```

    nu poate fi servită și procesul solicitant este pus în
    așteptare. Sfârșit algoritm>;

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

<Mergi la pasul 2>;

```

Figura 9.16 Algoritmul bancherului

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

În cele ce urmează vom oferi o implementare a algoritmului bancherului. Pentru simplitate vom considera că sistemul de operare dispune de patru tipuri de resurse și că asupra acestor resurse vor acționa cinci procese. Implementarea conține și 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 alocairi[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, alocairi);

// Simulam faptul ca procesul procesAles ajunge sa se termine cu bine,
// adica resursele ii sunt dealocate si este marcat ca neimplicat in
// algoritm
proceseImplicate[procesAles] = 0;
for(j=0; j<RESURSE; j++)
    alocairi[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 secvență: P1, P0, P2, P3, P4. Dacă însă în tabelul `resurseAlocatePerProces` se modifică alocarea resursei R2 pentru procesul P2 din 1 în 2, algoritmul nu va servi cererea din cauza apariției riscului de impas. Lăsăm pe seama cititorului să ruleze programul din fig. 9.17 și să vadă rezultatele obținute.

9.2 Conceptul de multiprogramare.

Cel mai important concept, introdus mai întâi la sistemele seriale și 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 existența simultană în memoria internă a mai multor procese care se execută concurrent. Multiprogramarea duce la o mai bună utilizare a procesorului și a memoriei.

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

Pe scurt, lucrul în multiprogramare se desfășoară astfel: în fiecare moment procesorul execută o instrucțiune a unui proces. Vom spune că acest proces este în starea **RUN**. Restul proceselor, fie că așteaptă apariția unui eveniment extern (terminarea unei operații 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 așteaptă un eveniment extern spunem că sunt în starea **WAIT**, iar cele care sunt pregătite de execuție 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 semnaliza apariția evenimentelor externe;

Sistemul de operare să gestioneze, să aloce și să protejeze resursele între utilizatori. Prin resurse înțelegem: memoria, dispozitivele periferice, fișierele, 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 însuși; el (procesul) știe, în funcție de rezultatele obținute până în prezent, când trebuie să urmeze o instrucțiune de I/O.

Trecerea proceselor din starea **RUN** în starea **READY** și 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

Acceastă modalitate de cedare a procesorului se bazează în general pe programatorul care dezvoltă programul să introducă instrucțiuni de cedare în codul sursă. În alte situații sistemul de calcul în sine introduce instrucțiuni de cedare printre instrucțiunile binare ale programului. Un planificator care folosește această metodă se numește *nepreemptiv*.

Cedarea voluntară a procesorului este o practică din vremurile de început ale sistemelor de operare. Deși simplifică mult munca planificatorului, acest gen de cedare dă unui proces posibilitatea de a bloca întregul 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 și nici să nu apeleze instrucțiuni de cedare a procesorului.

9.2.1.2 Cedarea involuntară a procesorului

Cedarea involuntară a procesorului presupune intervenția planificatorului pentru a întrerupe procesul curent. Un astfel de planificator se numește *preemptiv*. Întreruperea 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 aceeași pentru toate procesele și cheamă instrucțiunea de cedare a procesorului. În urma acestei cedări, planificatorul sistemului având o prioritate absolută, preia procesorul (devine activ) și decide care dintre procesele existente va primi procesorul.

9.2.2 Funcționarea unui planificator

În fig. 9.18 este prezentat algoritmul după care funcționează un planificator. În el se consideră că sunt n procese, iar prioritățile lor sunt numerele 1, 2, ..., n , cu 1 prioritatea ca mai mică și n prioritatea ca mai mare.

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

Procesul **P3**, are nevoie să efectueze până la terminare două operații I/O și are cea mai mare prioritate dintre cele trei procese. Procesul **P2** are nevoie de trei operații I/O. Procesul **P1**, cel mai puțin prioritar, are nevoie să efectueze până la terminare o singură operație I/O. Presupunem că cele trei procese intră în sistem în același timp. Desenul ilustrează pe orizontală axa timpului, iar porțiunile în care este scris "**RUN**" indică faptul că procesorul execută succesiv instrucțiuni mașină ale procesului respectiv. În funcție de momentele în timp în care cele trei procese solicită operații 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 puțin 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 priorității mici, reușește să intre în starea **RUN** numai când **P2** și **P3** sunt în starea **WAIT** sau **FINISH** (procesul s-a terminat). Acestea sunt momentele 2, 6, 8 și 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 și 8. Acesta este de fapt scopul principal al multi programării: valorificarea procesorului de către alt proces câtă vreme cel curent așteaptă un eveniment extern.

P3	RUN	WAIT			RUN	WAIT			RUN				
P2	READY	RUN	WAIT	RUN	WAIT	RUN	WAIT			RUN			
P1	READY		RUN	READY		RUN	WAIT	RUN	READY		RUN		
	0	1	2	3	4	5	6	7	8	9	10	11	12

Figura 9.19 Exemplu de evoluție în multiprogramare

Evoluțiile din fig. 9.19 descriu o situație particulară. Astfel, toate procesele sunt lansate în execuție simultan, iar după terminarea lui **P1** și **P2** nu se mai lansează nimic. În realitate, fiecare partiție primește spre execuție un nou proces (dacă există) după terminarea celui curent. Deci, spre exemplu, la momentul 10 în partiția 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** și invers. Pentru aceasta planificatorul trebuie să îndeplinească următoarele sub-sarcini:

- ținerea evidenței tuturor proceselor din sistem;
- alegerea procesului căruia i se va atribui procesorul și pentru cât timp;
- alocarea procesului un procesor;
- eliberarea procesorului la ieșirea procesului din starea **RUN**.

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

- starea procesului;
- pointer la următorul **PCB** în aceeași stare;
- numărul procesului (acordat de planificator);
- contorul de proces (adresa instrucțiunii mașină care urmează a fi executată);
- zonă pentru copia regiștrilor generali ai mașinii;
- limitele zonei (zonelor) de memorie alocate procesului;
- lista fișierelor 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 aceeași stare sunt înlănțuite între ele.

Deoarece toate procesele folosesc regiștrii generali ai procesorului, este necesar ca la ieșirea din starea **RUN** conținutul acestora să fie salvat. La o nouă intrare în **RUN** se reface conținutul regiștrilor. Spre exemplu, în fig. 9.20 este prezentat un schimb **READY** ↔ **RUN** între două procese **A** și **B**. Evident, la trecerea din **B** în **A** operațiile se inversează.

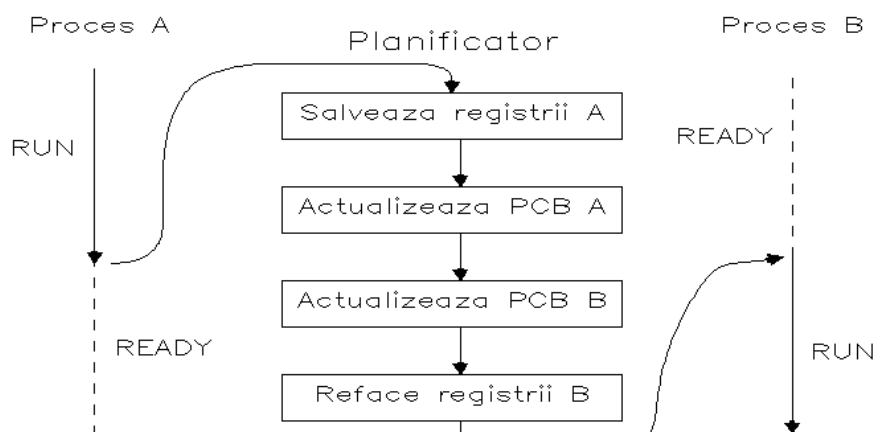


Figura 9.20 Schimbul RUN ↔ READY între două procese

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

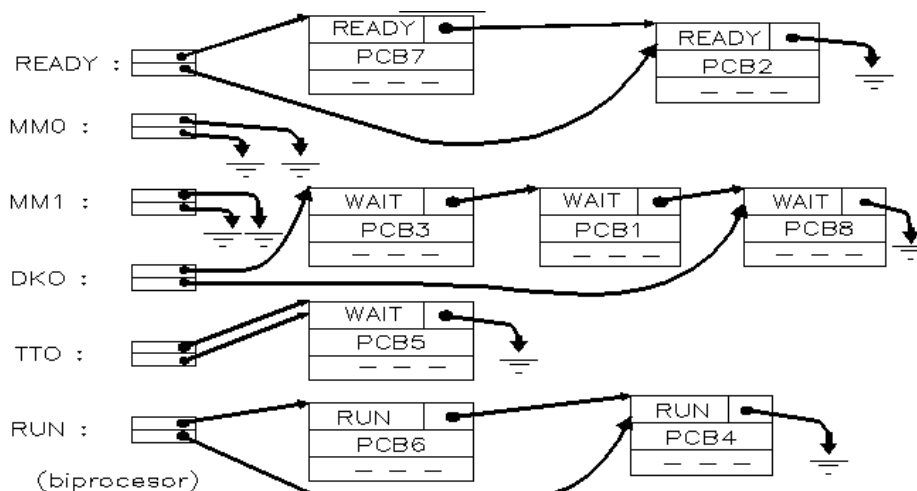


Figura 9.21 Cozi gestionate de planificatorul proceselor

9.3.2 Algoritmi de planificare.

O mare parte a literaturii a pus un accent pe algoritmi de planificare. Iată numai câteva lucrări de referință în domeniul sistemelor de operare, care tratează pe larg această problemă: [4], [10], [45], [49]. Motivele principale pentru care s-a dat importanță mare planificării sunt că performanța unui sistem de operare este influențată decisiv de funcționarea planificatorului și nu în ultimul rând pentru că planificare se pretează ușor la modelarea matematică.

Alte lucrări, cum ar fi [22], [29], enumeră foarte pe scurt câțiva 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*. În cele ce urmează, enumerăm cei mai utilizați algoritmi de planificare a proceselor.

9.3.2.1 FCFS (First Come First Served)

Numele algoritmului înseamnă în traducere directă înseamnă *primul venit - primul servit*. Este cunoscut și 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 execuție:

- 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 puțin timp procesor. Probabil că acest algoritm este cel mai bun, însă are un mare dezavantaj: presupune cunoașterea exactă a timpului procesor necesar execuției procesului.

9.3.2.3 Algoritm bazat pe priorități.

Este algoritmul cel mai des folosit. Toți ceilalți algoritmi descriși în literatură sunt cazuri particulare ale acestuia. Fiecare proces primește un număr de prioritate. Procesele se ordonează după aceste priorități, apoi se execută în această ordine.

Se disting două metode de stabilire a priorităților:

1. procesul primește prioritatea la intrarea în sistem și și-o păstrează până la sfârșit;
2. Sistemul de operare calculează prioritățile după reguli proprii și le atașează, dinamic, proceselor în execuție.

Varianta 1) este folosită destul de des dar are dezavantajul că dacă apar multe procese cu prioritate mare, atunci cele cu prioritate mică așteaptă 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 așteaptă mai mult și se readuce la valoare inițială după ce intră în execuție.

9.3.2.4 Algoritm bazat pe termene de terminare (deadline scheduling)

Acest algoritm de planificare este folosit în principal la sistemele cu cerințe de timp real foarte stricte. În asemenea sisteme este critic pentru un task sau proces să se termine la timp. Astfel fiecărui task i se atașează un termen de terminare. Un caz special sunt task-urile repetitive cărora li se asociază termene de terminare asociate fiecărei iterații. Planificatorul folosește aceste termene pentru a decide când și cărui task și pentru cât timp să-i acorde procesorul în așa fel încât să se termine la timp. Evident pentru acest gen de planificare este esențială cunoașterea exactă a duratei fiecărui task. La apariția unui nou task în sistem, planificatorul îl va accepta pentru execuție 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 secvenței de execuție 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).

Acești algoritmi sunt destinați în special sistemelor de operare care lucrează în timesharing. Pentru realizare, se definește 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 sfârșitul cozii, al doilea proces este preluat de către procesor ș.a.m.d.

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

O altă variantă urmărește echilibrarea sistemului folosind așa-zisul *Round-Robin cu reacție*. 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 Algoritm de cozi pe mai multe nivele

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

Taskuri sistem.
Lucrările interactive.
Lucrări în care se editează texte.
Lucrări seriale obișnuite.
Lucrări seriale ale studenților.
...

