Apelul unei proceduri la distanță

-Protocolul RPC;
-Microprotocoalele care compun RPC;
-Studii de caz:SunRPC; DCE-RPC

Protocolul RPC

- Protocolul nivelului Transport, numit protocolul RPC (**R**emote **P**rocedure **C**all) a fost proiectat în acest scop. RPC foloseşte aceeaşi semantică a apelului local de procedură, în sensul că dacă un proces face un apel într-o procedură el nu trebuie să-şi pună problema dacă este global sau local.
- Față de apelul local, apelul la distanță trebuie să rezolve, în plus două probleme principale:

-Spre deosebire de apelul local, care se realizează la nivelul unui calculator, apelul la distanță se face la nivelul unei rețele, care limitează dimensiunea mesajelor şi are tendința de a pierde şi reordona mesajele.

-Calculatoarele pe care se execută procesele apelant, respectiv apelat pot avea arhitecturi şi formate de reprezentare a datelor diferite.

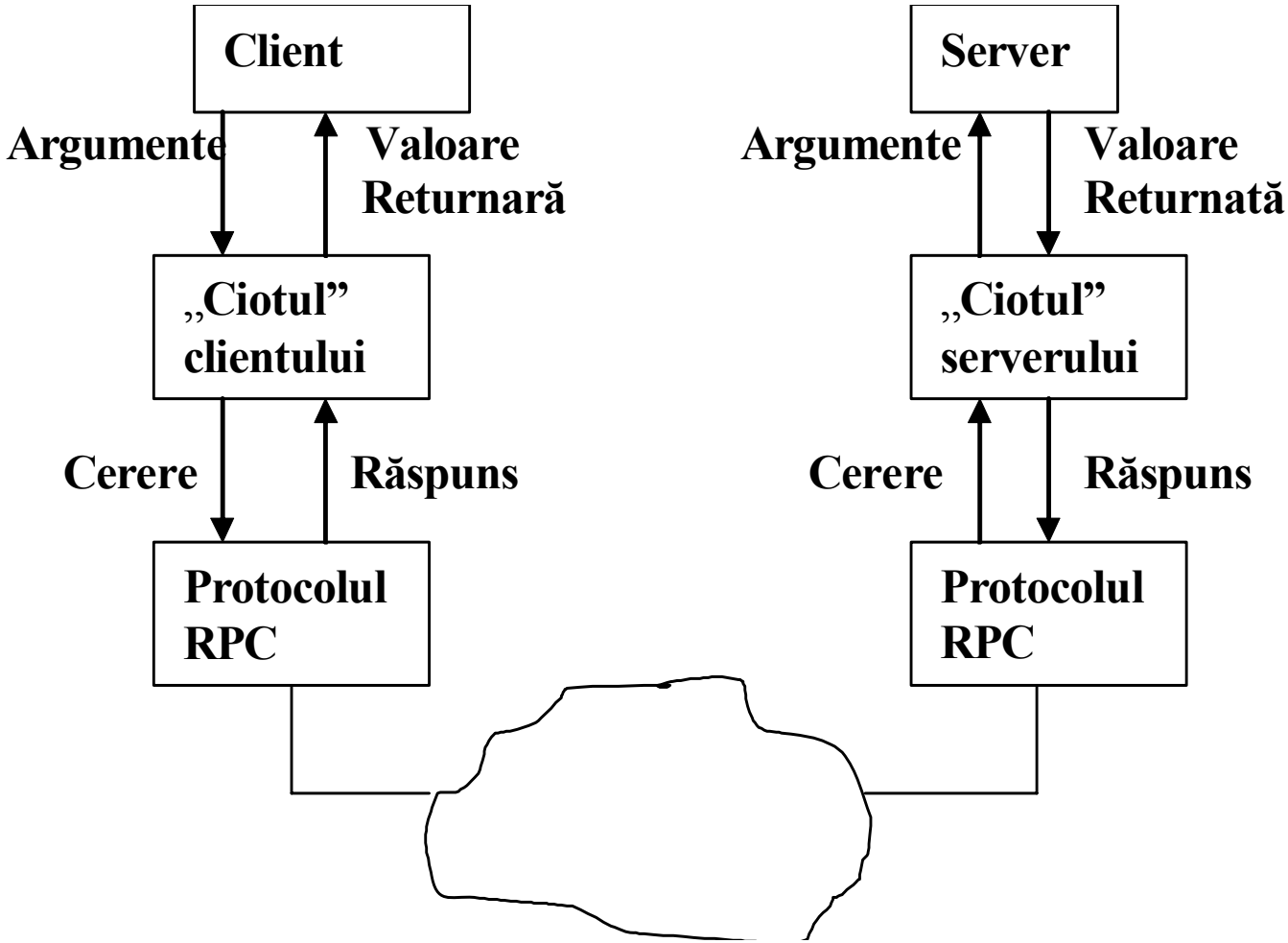
Componentele principale ale RPC

- Astfel, un mecanism RPC implică două componente principale:

- Un protocol care administrează mesajele transmise între procesele client, respectiv server şi care rezolvă problemele legate de structura particulară a rețelelor la care sunt conectate gazdele pe care sunt executate procesele respective.

- La nivelul gazdei client, limbajele de programare şi compilatoarele să împacheteze argumentele într-un mesaj cerere, care apoi să poată fi translatat în argumente pe gazda server şi, asemănător să se procedeze în sens invers cu valorile returnate. O astfel de componentă a mecanismului RPC se numeşte compilator ciot („stub”).

- Figura urm. descrie ce se întâmplă când un client invocă o procedură la distanță. În primul rând, clientul apelează un „ciot” local, căruia îi transmite parametrii procedurii. Acest „ciot” ascunde faptul că procedura este la distanță, translatând argumentele într-un mesaj cerere, după care va cere un protocol RPC pentru a transmite mesajul cerere către gazda server. La nivelul serverului, protocolul RPC furnizează mesajul cerere componentei „ciot”, care translatează mesajul primit în argumente ai unei anumite proceduri, după care realizează apelul local la nivelul serverului. După ce se termină execuția procedurii, aceasta transmite datele de retur către componentei „ciot” a serverului, care împachetează valoarea returnată într-un mesaj răspuns care este prezentat protocolului RPC pentru a fi transmis clientului.


Client
Argumente
Valoare
Returnară
„Ciotul”
clientului
Cerere
Răspuns
Protocolul
RPC
Server
Argumente
Valoare
Returnată
„Ciotul”
serverului
Cerere
Răspuns
Protocolul
RPC

Structura RPC

- Protocolul RPC execută o mulțime de funcții, deci este normal să-l considerăm ca o stivă formată din trei protocoale mai mici(microprotocoale): BLAST, CHAN şi SELECT. La baza stivei stau protocoalele care implementează rețeaua locală precum şi protocolul IP.

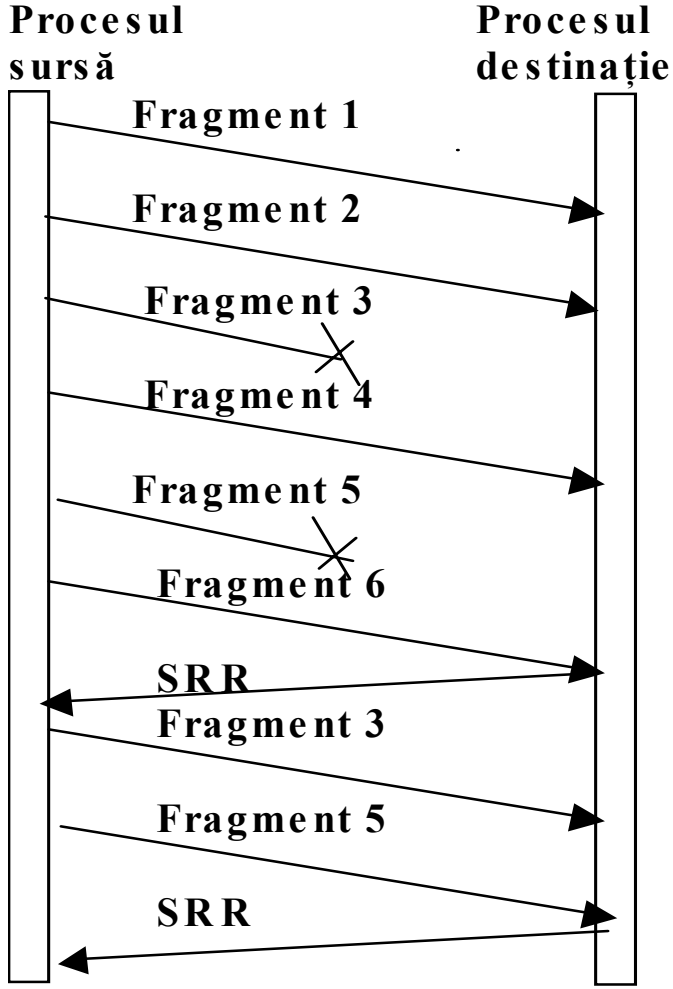
- BLAST fragmentează şi reasamblează mesajele de dimensiune mai mare. Astfel se rezolvă problema transmiterii unui mesaj care traversează rețele locale, pentru care dimensiuni maxime permise ale mesajului sunt diferite. Deşi şi IP poate face fragmentarea/reasamblarea mesajelor, algoritmii corespunzători protocolului BLAST sunt superiori.
- CHAN implementează algoritmul cerere/răspuns, pe baza mecanismelor de confirmare şi de expirare a timpului, realizând astefel furnizarea fiabilă a datelor şi sincronizarea între mesajele cerere şi răspuns.
- SELECT distribuie mesajele cerere către procesele cărora le sunt destinate. În acest sens, defineşte spațiul de adrese pentru identificarea procedurile la distanță. De asemenea SELECT administrează concurența, în cazul când o procedură este cerută în acelaşi timp de mai mulți clienți.

Algoritmul BLAST

- Deasupra lui IP se află BLAST, care are rolul de a multiplexa mesajele de dimensiune mai mică care vin de pe rețeaua locală, în mesaje de dimensiune mai mare(maxim 32 KB).
- Ideea de bază a algoritmului BLAST este, în cazul procesului sursă, să împartă un mesaj de dimensiune mai mare primit de la nivelurile superioare, într-o mulțime de fragmente mai mici, şi apoi să transmită pe rețea aceste fragmente unul după altul pe rețea. Procesul destinație transmite un **mesaj de retransmisie selectivă**(SRR-**S**elective **R**etransmission **R**equest) procesului sursă, indicând care fragmente au sosit şi care nu. Mesajul SRR mai este denumit şi o **confirmare parțială sau selectivă**. În final, aplicația sursă retransmite fragmentele pierdute. În cazul în care toate fragmentele au sosit, mesajul SRR este folosit pentru

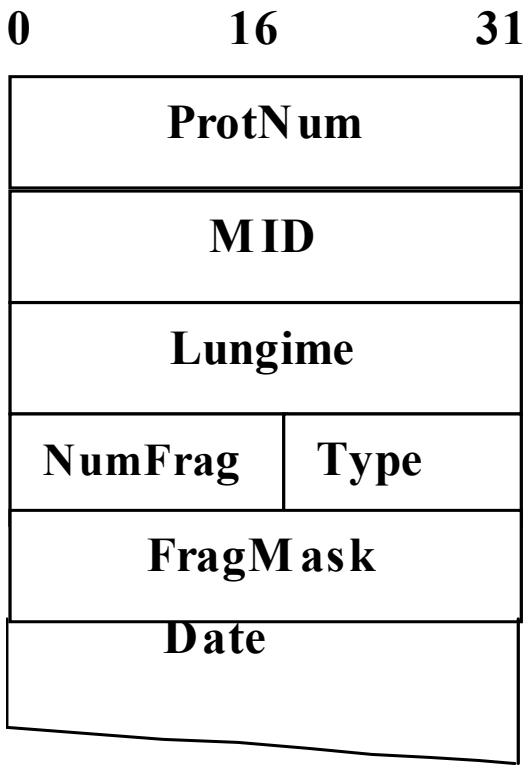
  confirmarea primirii întregului mesaj.

- **Exemplu.** Figura urm prezintă o evoluție în timp a protocolului BLAST. Fragmentele 4, 6 au fost pierdute şi acest lucru este semnalat printr-un mesaj SRR. Se efectuează retransmiterea lor, după care tot printr-un mesaj SRR se confirmă primirea lor.



- La nivelul aplicației sursă, după fragmentarea mesajului şi transmiterea fiecăruia dintre fragmente, se setează un ceas numit DONE. Oricând soseşte un SRR, aplicația sursă retransmite fragmentele cerute şi resetează ceasul DONE. Atunci când SRR va indica sosirea tuturor fragmentelor, aplicația sursă distruge copiile fragmentelor mesajului şi ceasul DONE. Dacă ceasul DONE dep. un anumit prag consideră leg. pierdută.
- La nivelul părții destinatar, atunci când soseşte primul fragment se inițializează o structură de date, în care se păstrează fiecare fragment primit şi se setează un ceas LAST_FRAG, care măsoară intervalul de timp care a trecut de la sosirea ultimului fragment. De fiecare dată când este primit câte un fragment, acesta este depus în structura de date amintită şi în momentul când toate fragmentele au fost primite, acestea sunt reasamblate, rezultând mesajul în forma lui inițială(de la aplicația sursă) şi apoi transmite mesajul protocolului de nivel superior.
- Dacă ultimul fragment soseşte(ultimul fragment are un marcaj special) dar mesajul nu este complet primit, atunci aplicația destinație determină care dintre fragmente nu au fost primite şi transmite un SRR aplicației sursă. De asemenea, este inițializat un ceas RETRY.
- Dacă ceasul LAST_FRAG depăşeşte o valoare dată, procesul destinatar stabileşte care dintre fragmente nu au fost primite şi transmite un SRR procesului expeditor. De asemenea, el setează un ceas RETRY.
- Dacă ceasul RETRY depăşeşte o valoare dată, o dată sau de două ori, atunci procesul destinatar determină care fragmente nu au fost încă primite şi retransmite un mesaj SRR.
- Dacă ceasul RETRY depăşeşte o valoare dată pentru a treia oară, atunci procesul destinatar renunță la fragmentele care au fost primite şi anulează ceasul LAST_FRAG, adică renunță la conexiune.

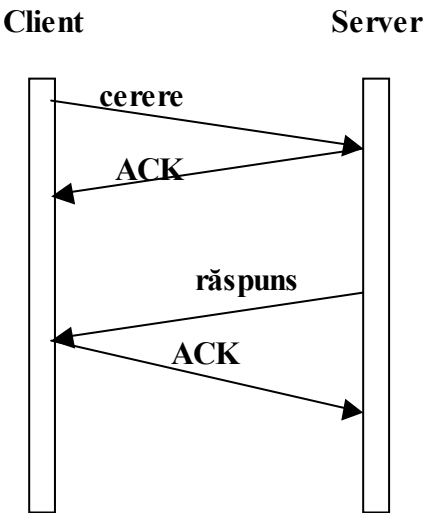
Formatul unui mesaj BLAST



- Antetul mesajului BLAST conține câmpurile:
- **Identificatorul mesajului (MID)** Toate fragmentele care aparțin aceluiaşi mesaj au aceeaşi valoare în câmpul MID.
- **Tipul** indică dacă fragmentul respectiv conține date sau este un SRR.
- **Număr protocol(ProtNum)** indică protocolul care se află deasupra lui BLAST.
- **Lungime** conține numărul octeților de date care se găsesc în acest fragment.
- **Număr de fragmente(NumFrags)** conține numărul total de fragmente ale mesajului.
- **Mască de fragment** este folosit pentru a face deosebirea între fragmente. Pentru un mesaj care conține date şi al *i*-lea bit este 1, restul fiind 0, va indica faptul că acesta este al *i*-lea fragment. Pentru mesajele care conțin un SRR, faptul că al *i*-lea bit este 1 indică sosirea celui de-al *i*-lea fragment; dacă este 0 înseamnă că fragmentul al *i*-lea a fost pierdut.

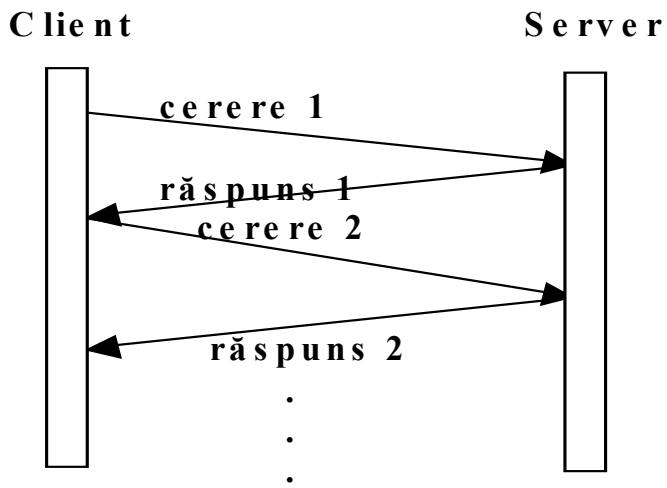
Microprotocolul CHAN

- Implementează algoritmul cerere/răspuns ce reprezintă componenta centrală a RPC. CHAN garantează furnizarea integrală a mesajelor, asigură că este furnizată numai o copie a fiecărui mesaj şi permite sincronizarea proceselor care comunică între ele. Numele de CHAN provine de la faptul că protocolul implementează un canal logic de tip cerere/răspuns, între o pereche de participanți. La fiecare moment, pe fiecare canal nu poate exista decât o singură tranzacție activă de mesaje. Dacă aplicațiile vor să execute mai multe tranzacții de tipul cerere/răspuns, la acelaşi moment, ele trebuie să deschidă mai multe canale. Pe fiecare canal, la un moment dat poate exista cel mult un mesaj cerere transmis de la o aplicație client către o aplicație server.
- **Algoritmul CHAN.** În figura urm. este ilustrată o evoluție în timp a algoritmului CHAN.



Confirmare implicită

- Clientul transmite un mesaj cerere şi serverul îl confirmă(ACK), după care serverul transmite un mesaj de răspuns şi clientul confirmă primirea lui. Deoarece mesajul de răspuns este recepționat cu o anumită întârziere şi se poate întâmpla ca în acest interval de timp mesajul cerere sa fie primit, metoda prezentată poate fi îmbunătățită, folosind o tehnică numită **confirmare implicită**.
- Aşa cum este ilustrat în figura urm, mesajul de răspuns, este folosit pentru a confirma mesajul cerere şi el trebuie să urmeze imediat cereri corespuzătoare.



Probleme de rezolvat

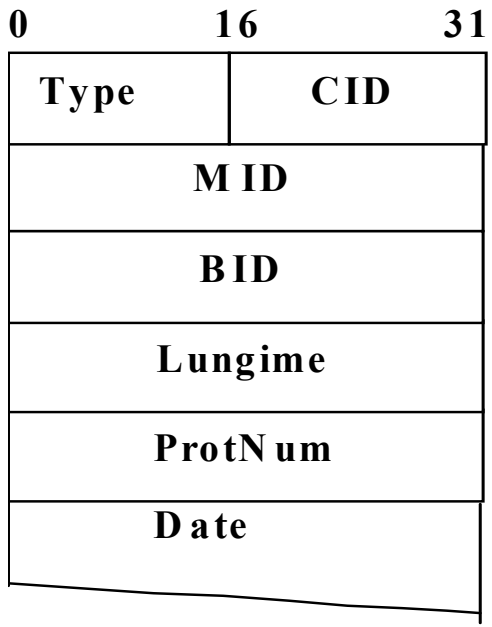
- Pierderea mesajului(informații sau o confirmare)

- Clientul şi serverul păstrează o copie a fiecărui mesaj pe care l-au transmis, până când soseşte un mesaj de confirmare.
- Fiecare participant setează un ceas, denumit generic RETRANSMIT şi retransmite mesajul pentru care nu s-a primit o confirmare după un interval de timp mai mare decât valoarea ceasului. Ambele părți resetează valoarea ceasului şi efectuează o nouă transmisie a mesajului ş.a.m.d.
- Această strategie bazată pe confirmare şi retransmisie după expirarea timpului, poate avea ca efect duplicarea mesajelor. De exemplu, mesajul original poate sosi la destinatar, mesajul de confirmare poate fi pierdut, şi în consecință să fie efectuată o retransmitere a acestuia. Astfel, gazda destinatar trebuie să facă distincție între primul mesaj şi eventualele copii, pe care trebuie să le distrugă. Acest lucru este realizat prin folosirea câmpului MID din antet. Oricare mesaj al cărui câmp MID nu se potriveşte cu cel al următorului mesaj aşteptat este distrus.

- A doua problemă este că timpul de lucru necesar serverului pentru a genera un mesaj de răspuns, poate fi destul de mare sau procesul destinatar poate fi distrus sau gazda respectivă se poate defecta(sau reseta) înainte de a se genera răspunsul. Pentru a vedea dacă serverul este un calculator mai lent sau unul care nu mai funcționează, clientul transmite periodic mesaje prin care testează dacă serverul „mai este în viață”, la care acesta răspunde cu mesaje de confirmare. Pentru aceasta, la nivelul clientului este folosit un ceas denumit PROBE.

Formatul unui mesaj CHAN

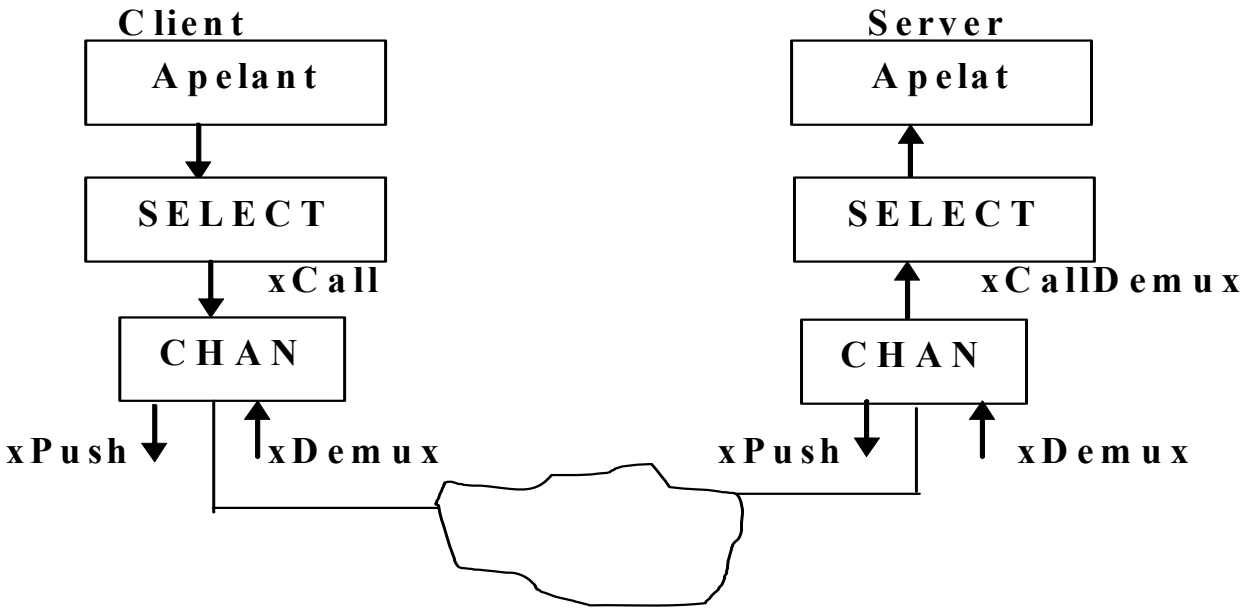
- Câmpul **type** specifică tipul mesajului, care poate fi REQ(cerere), REP(răspuns), ACK(confirmare) şi PROBE(serverul mai este „în viață” sau nu).
- Câmpul **ProtNum** identifică protocolul imediat superior lui CHAN.



- Câmpul **CID** identifică în mod unic canalul logic la care aparține mesajul respectiv. Observăm că acesta are o lungime de 16 biți, ceea ce înseamnă că CHAN permite cel mult 64K tranzacții concurente de tipul cerere/răspuns, între oricare perechi de gazde, în acelaşi timp.
- Câmpul **MID** identifică în mod unic fiecare pereche cerere/răspuns. Mesajul răspuns are acelaşi MID ca şi cererea.
- Câmpul **BID** conține identificatorul de „bootare” al gazdei. Acesta este un număr care este incrementat de fiecare dată când gazda respectivă reîncarcă sistemul de operare. Acest număr este citit de pe disk, este incrementat, apoi este salvat pe disc în timpul procedurii de „start-up”. Fiecare mesaj transmis de gazda respectivă va conține acest număr. Prin introducerea acestui câmp se încearcă să se evite apariția de mesaje duplicat datorită repornirii unei gazde.

Microprotocolul SELECT

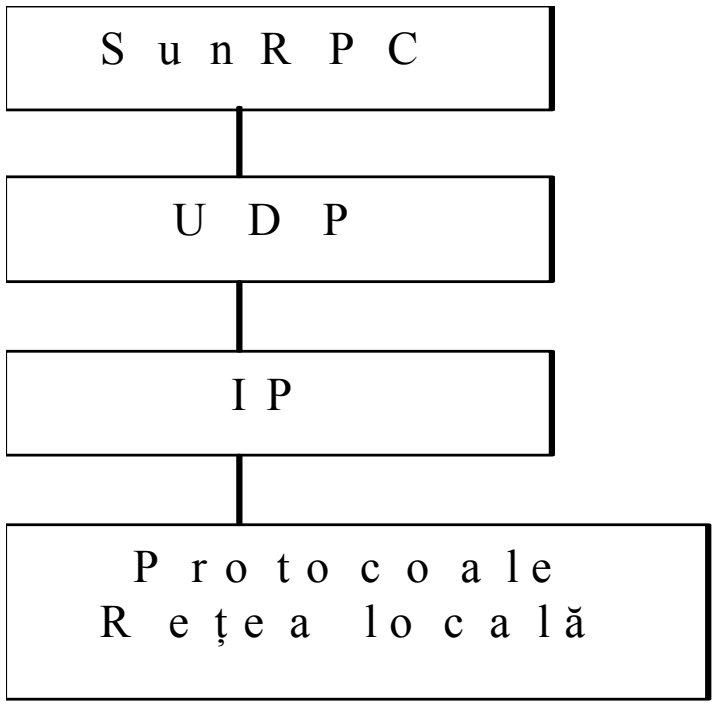
- Repartizează mesajele cerere către procedura corespunzătoare. La nivelul clientului, SELECT primeşte un număr de procedură pe care clientul doreşte să o ceară, acest număr este înregistrat în antet, după care este apelată o operație „call” care aparține lui CHAN. La nivelul serverului, SELECT foloseşte numărul de procedură pe care îl preia din antet pentru a selecta procedura de la nivel local cerută. Când această procedură întoarce rezultatele calculate de ea, SELECT le transmite protocolului de nivel mai înalt care a cerut apelul(figura urm).



- Observăm funcțiile diferite ale celor dpuă protocoale la nivel de client, respectiv server. La nivel de client, SELECT transmite apelul indicând procedura apelată, iar CHAN realizează multiplexarea mai multor mesaje, atunci când se transmite, respectiv demultiplexarea atunci când se returnează rezultate. La nivelul serverului, acțiunile se petrec dual.
- La prima vedere, nu rezultă clar de ce SELECT trebuie să existe ca un protocol şi funcțiile sale să nu fie incluse în protocolul CHAN. Există două motive pentru care a fost conceput protocolul SELECT. Primul constă în necesitatea de a identifica procedurile printr-o adresă pe două niveluri. Astfel, primul câmp conține numărul programului, iar al doilea câmp reprezintă numărul procedurii. De exemplu, la nivelul unei aplicații de administrare a unui server de fişiere, procedurile „read”, respectiv „write” şi „seek” pot avea numerele 1, respectiv 2 şi 3. De asemenea, în cazul unei aplicații de administrare a numelor procedura de „inserare” poate avea numărul 1 iar cea de „căutare” numărul 2. Observăm că există proceduri diferite care au acelaşi număr, dar ele pot fi identificate în mod unic prin numărul aplicației din care fac parte.
- Al doilea motiv, este legat de implementarea şi administrarea concurenței. Protocolul CHAN deschide câte un canal logic pentru fiecare apel al unei proceduri. Dacă la un moment dat, există mai multe apeluri ale aceleaşi proceduri, deci mai multe canale active, SELECT va administra concurent aceste apeluri.

Studii de caz

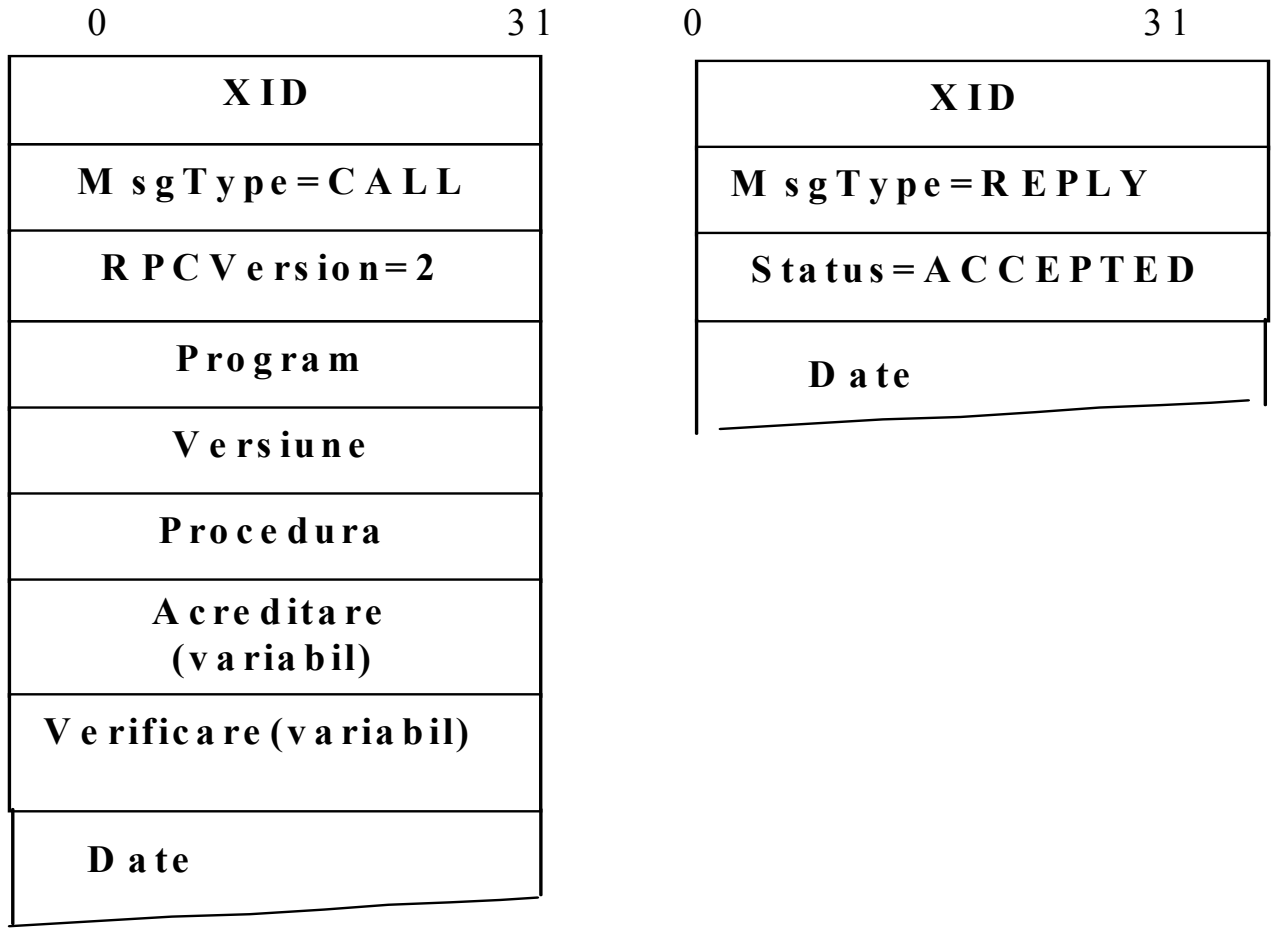
- **SunRPC** a fost implementat pe calculatoarele Sun. SunRPC implementează funcțiile microprotocoalelor SELECT/ CHAN/ BLAST fără a le considera entități distincte. Structura lui SunRPC este ilustrată în figura urm.



- În ceea ce priveşte implementare algoritmului cerere/răspuns, SunRPC diferă de CHAN prin faptul că nu garantează prezența a cel mult un mesaj pe un canal logic la un moment dat. Rolul lui SELECT este împărțit între UDP şi SunRPC; UDP distribuie mesajul aplicației, iar SunRPC procedurii din programul respectiv, căreia îi este destinat. Implementarea funcție de transmitere a mesajelor care au o dimensiune mai mare decât cea permisă de rețeaua locală respectivă, este implementată de către IP.
- Pentru a identifica o procedură la distanță, SunRPC foloseşte o adresă pe două niveluri, compusă dintr-un **număr de program** şi un **număr de procedură**, ambele memorate pe 32 de biți.
- Fiecare program poate fi localizat transmițând un mesaj către un anumit port UDP. Când soseşte un mesaj cerere la acest port, SunRPC îl preia şi apelează procedura corespunzătoare. Pentru a determina corespondența dintre numărul de port şi numărul de program RPC, există o componentă a SunRPC numită **PortMapper,** care are numărul de program RPC x00100000 şi căruia îi corespunde numărul de port UDP 111. De exemplu, pentru a transmite un mesaj cerere către procedura read a serverului de fişiere, mai întâi clientul transmite mesajul cerere către **PortMapper**, la numărul de port 111; pe baza adresei este selectată componenta care face corespondența între adresa RPC şi numărul de port. După aceasta, este transmis un mesaj prin care se specifică numărul procedurii cerute, la portul UDP furnizat anterior. De asemenea, clientul memorează corespondențele număr de program-port UDP realizate anterior, pentru a nu a mai apela la **PortMapper** în cazul apelării unei proceduri care a mai fost cerută anterior în aceeaşi sesiune de lucru.

Antetul unui mesaj SunRPC

- Formatele mesajelor SunRPC:
- a)cerere ; b)răspuns

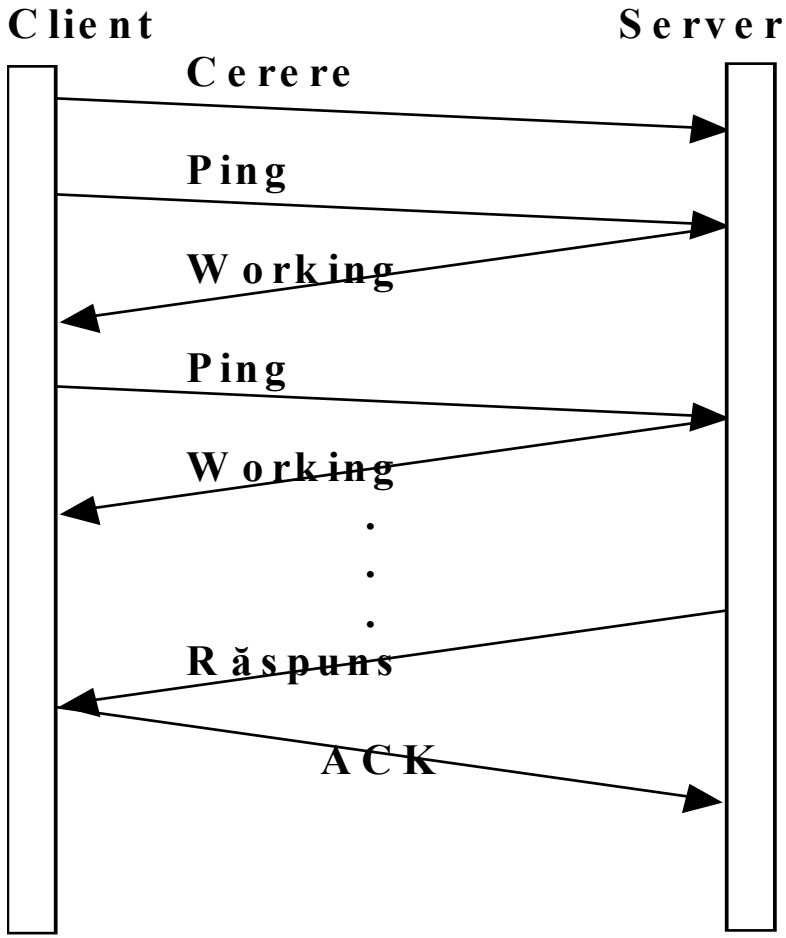


- XID reprezintă identificatorul tranzacției, având semnificația câmpului MID al antetului unui mesaj CHAN. Dacă clientul care a transmis o cerere serverului nu primeşte răspunsul până la expirarea timpului, el va retransmite mesajul. Când serverul va primi un duplicat al unui mesaj deja primit(lucru pe care îl va constata pe baza câmpului XID) îl va distruge.
- Câmpurile Acreditare şi Verificare au sunt folosite pentru autentificarea clientului, adică pentru a se verifica identitatea acestuia. Problema autentificării ține de un capitol special, al securității transmiterii datelor în rețele şi va fi discutat separat.

Protocolul DCE-RPC

- **DCE(D**istributed **C**omputing **E**nvironment**)** este un set de standarde definite de organizația OSF(**O**pen **S**oftware **F**oundation), pentru sisteme distribuite. DCE-RPC a fost proiectat să funcționeze deasupra lui UDP. Se aseamănă cu SunRPC în sensul că defineşte o schemă de adresare pe două niveluri: UDP realizează demultiplexarea şi repartizează mesajul către serverul cerut, iar DCE-RPC distribuie mesajul către o anumită procedură, pe care serverul respectiv o pune la dispoziție.
- Spre deosebire de Sun RPC, DCE-RPC implementează transmiterea a cel mult unui mesaj pe un canal logic. Acest lucru este relizat prin combinarea algoritmilor conținuți în microprotocoalele BLAST şi CHAN.
- Figura urm. prezintă o evoluție în timp a schimburilor de mesaje dintre un clent şi un server, fiecare mesaj este etichetat prin tipul său.

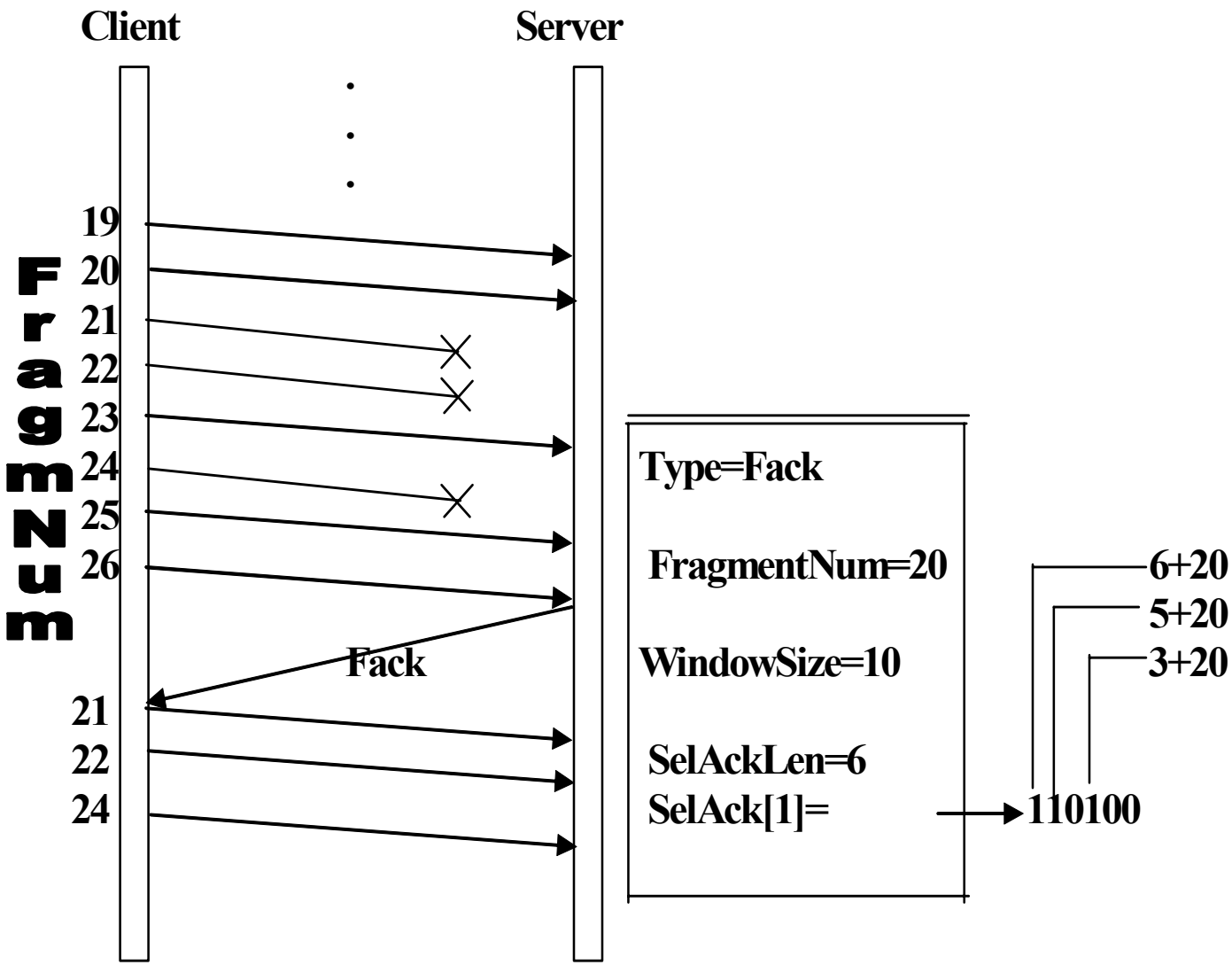
- Un schimb de mesaje DCE-RPC



- Clientul transmite un mesaj **Cerere**, după care periodic poate transmite mesaje de testare(**Ping**), la care serverul răspunde cu mesaje care indică faptul că nu s-a terminat execuția procedurii la distanță apelate (**Working**). Când primeşte mesajul de **răspuns**, clientul va confirma primirea printr-un mesaj **ACK**. Există şi alte mesaje care nu sunt prezentate în figură. De exemplu, clientul poate transmite serverului un mesaj **Quit**, prin care îi cere să abandoneze un apel transmis anterior, în curs de prelucrare; în acest caz serverul va răspunde cu un mesaj **Quack**(confirmarea unui mesaj **Quit**). De asemenea, serverul poate repinge un mesaj cerere printr-un mesaj **Reject.** La un mesaj **Ping**, serverul poate răspunde cu un mesaj **Nocall**, indicând faptul că nu a primit nici-un apel la procedura respectivă. Pe lângă tipul mesajului, atât mesajul cerere cât şi cel de răspuns, mai conțin şi alte câmpuri. Câmpul **ServerBoot** are acelaşi scop ca şi câmpul BID din cadrul protocolului CHAN. Serverul îşi înregistrează momentul său de bootare, şi această valoare este inclusă în fiecare mesaj transmis. Câmpul **ActivityId** este similar câmpului CID din cadrul protocolului CHAN, deci el identifică o conexiune logică între un client şi un server.

- Câmpul **SequenceNum** diferențiază apeluri ale aceleiaşi activități; are acelaşi scop cu câmpul MID(Message ID) al protocolului CHAN şi câmpul xid al protocolului SunRPC. Câmpul **FragmentNum** identifică fiecare fragment care face parte din acelaşi mesaj.
- Atât clientul, cât şi serverul implementează un mecanism de confirmare selectivă, care lucrează astfel: În primul rând fiecare fragment care face parte din acelaşi mesaj conține atât un număr de fragment unic, cât şi un fanion care indică dacă este un fragment obişnuit(frag) sau ultimul fragment al mesajului(last-frag). Mesajele care nu depăşesc dimensiunea unui pachet, au setat un alt fanion(no-frag). Serverul va şti când a primit mesajul complet, pe baza setării fanionului last-frag şi nu există "goluri" în secvența de numere de fragmente. În al doilea rând, ca răspuns la fiecare fragment sosit, serverul transmite un mesaj **Fack (**fragment acknowledgement**)** către client. Această confirmare identifică numărul cel mai mare de fragment pe care serverul l-a primit cu succes. De asemenea, serverul confirmă şi numerele cele mai mari de fragmente care au fost primite în afara ordinii. El realizează acest lucru printr-un vector de biți ce identifică acele fragmente pe care le-a primit în afara ordinii. În final, clientul răspunde prin retransmiterea fragmentelor pierdute.

- **Exemplu.** Figura urm ilustrează modul de lucru al acestui algoritm. Să presupunem că serverul a primit toate fragmentele până la numărul 20, şi în plus fragmentele 23, 25, şi 26. Serverul răspunde cu un mesaj **Fack** care indică fragmentul cu numărul de ordine 20 ca fiind cel cu numărul de secvență cel mai mare primit şi pentru care nu există fragmente cu numere de ordine mai mici neprimite. De asemenea, vectorul de biți **SelAck**, cu a treia componentă(23=20+3), a cincea(25=20+5) şi a şasea (26=20+6) setate este trimis clientului. Câmpul **SelAckLen** va conține dimensiunea vectorului amintit anterior.
- DCE-RPC implementează un algoritm de control al fluxului, asemănător cu cel al protocolului TCP. Pentru fiecare mesaj **Fack,** serverul confirmă clientului fragmentele primite şi îl informează cât de multe fragmente poate să transmită acum. Acest lucru este realizat prin intermediul câmpului **WindowSize** (figura urm.). Diferența față de protocolul TCP, este că aici fereastra conține numărul de fragmente cerute şi nu numărul de octeți.


Client
Server
FragNum
19
20
21
22
23
24
25
26
Fack
21
22
24
Type=Fack
FragmentNum=20
WindowSize=10
SelAckLen=6
SelAck[1]=
110100
6+20
5+20
3+20

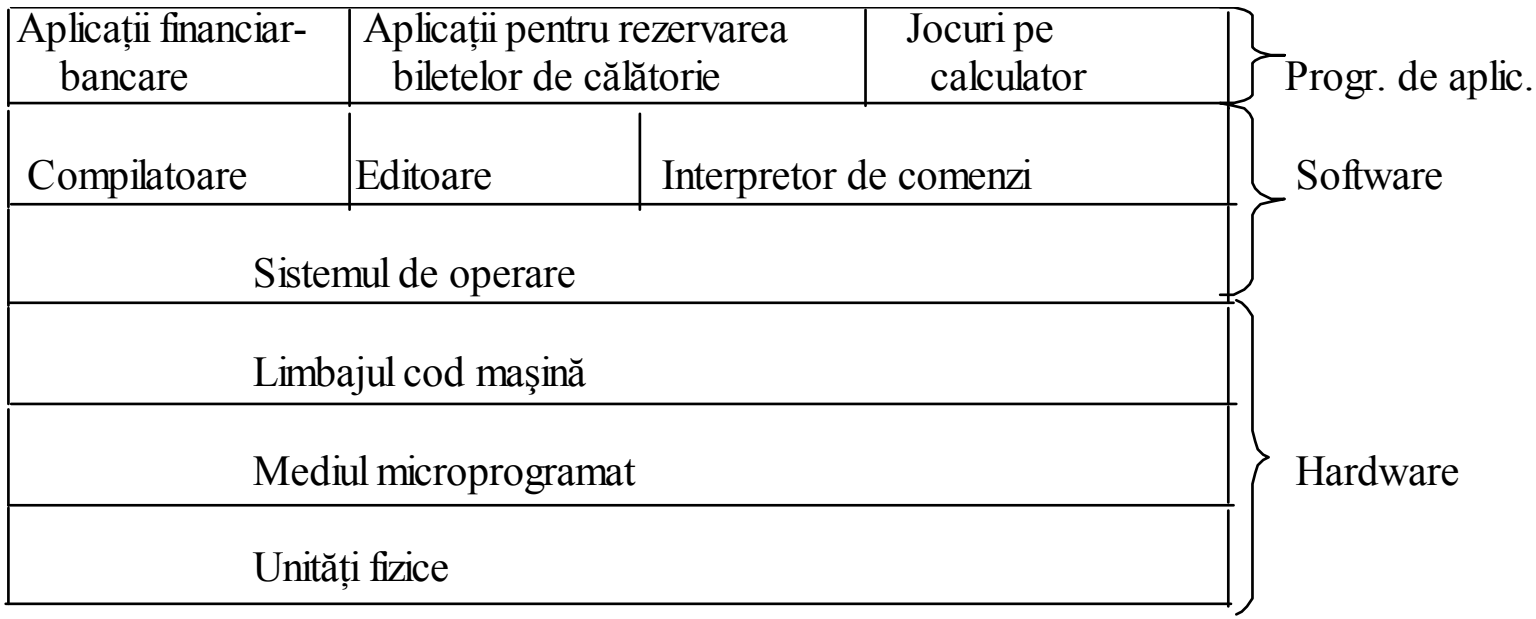
INTRODUCERE

Obiectivele cursului:

- evidențierea rolului central al sistemului de operare în cadrul componentei software a unui sistem de calcul
- prezentarea evoluției sistemelor calcul şi a celor de operare;
- prezentarea conceptelor care stau la baza sistemelor de operare, care vor fi dezvoltate în capitolele următoare;
- prezentarea structurii sistemelor de operare şi a funcțiile lor;
- prezentarea componentelor hardware care realizează interfața cu sistemul de operare.

Locul sistemului de operare în cadrul unui sistem de calcul

- Componentele principale ale unui sistem de calcul (SC) sunt cea fizică(hardware) şi cea logică(software). La rândul ei, componenta logică este formată din software-ul (programele) de aplicaţii şi software-ul de sistem, aşa cum este redat în figura următoare.



- Programele de aplicații au scopul de a rezolva cu ajutorul calculatorului o problemă specifică dintr-un anumit domeniu de activitate (financiar-bancar, ştiințific etc.). Programele de sistem oferă contextul (mediul) în care programatorii pot crea propriile programe de aplicații, care nu sunt disponibile la nivel fizic. Sistemul de operare (SO) face parte din componenta soft de sistem.
- Este dificil să se dea o definiție completă a ceea ce este un SO, în schimb, este mult mai uşor să se vadă ce face un SO. Astfel:

- SO oferă facilitățile necesare unui programator pentru a-şi construi propriile aplicații.
- SO gestionează resursele fizice(memorie, discuri, imprimante etc) şi cele logice (programe de sistem, fişiere, baze de date etc) ale sistemului de calcul, oferind posibilitatea ca utilizatorii să poată folosi în comun aceste resurse, pe baza unor anumite reguli, ceea ce conduce la scăderea cheltuielilor de prelucrare şi la creşterea performanțelor sistemului de calcul.
- SO oferă o interfață prin care aplicațiile utilizator şi cele de sistem au acces la componenta hardware.

Evoluția sistemelor de operare

- **generația I:** dispun numai de echipamentul hard.
- **generația a II-a** :

- componente hardware noi (cititorul de cartele, imprimanta, banda magnetică)
- componente software (compilator pentru limbajul FORTRAN, programe specializate destinate asamblării, încărcării şi înlănțuirii programelor, biblioteci software de funcții uzuale, rutine de interfață cu componentele fizice numite **drivere** etc.).
- conceptul de **monitor**, ce reprezintă o formă rudimentară de SO. Lucrările (job-urile) erau o succesiune a fazelor (**editarea** textului sursă al programului, **compilarea, editarea legăturilor**, **depanarea, execuția** programului).

Monitorul rezident

- Intervenția umană între faze presupune o mare pierdere de timp şi o utilizare ineficientă a echipamentului. Pentru a evita aceste neajunsuri, a apărut conceptul de **încărcare automată a job-urilor si a fazelor**. Pentru aceasta a fost creat un mic program, denumit **monitor rezident**, care realiza acest lucru. Programatorul trebuie să insereze, printre cartelele programului şi ale datelor sale, unele cartele speciale numite **cartele de comandă** care se adresează acestui monitor rezident. Prin această regulă de diferențiere, s-a definit de fapt un **limbaj de control** al job-urilor**.** Prin intermediul lui, se comandă trecerea de la o fază la alta sau de la un job la altul.

  Monitorul rezident este programul permanent activ. Pentru desfăşurarea înlănțuirilor de faze, este necesar ca operațiile de I/O să fie făcute de către monitor şi nu de către programul utilizator, pentru a depista cartelele care îi sunt adresate, adică cele de comandă. Tipul de SO prezentat se numeşte **sistem serial,** deoarece job-urile se execută unul după altul(**prelucrare pe loturi - Batch processing**). O altă caracteristică a acestui sistem este **monoprogramarea**, adică CPU nu se ocupă de alt job până când nu-l termină pe cel curent. De asemenea, programului utilizator nu-i este permis să modifice zona de memorie a monitorului rezident şi nici să comande oprirea întregului sistem.

- **Conversiile off-line** constau în transferarea conținutul cartelelor perforate pe benzi magnetice. Pentru aceasta se foloseau nişte dispozitive de conversie relativ simple. Conținutul benzilor era utilizat de către SC în locul cartelelor perforate, unitatea de bandă fiind intrarea standard. Analog, rezultatele, sub forma unor linii de imprimantă, sunt depuse mai întâi într-un fisier pe un suport magnetic, iar la terminarea lucrului, conținutul fişierului este listat la imprimantă. De asemenea, prin această metodă, se reduce timpul de lenevire al CPU datorat vitezei de lucru mai mici a unor periferice. Odată cu operarea off-line a apărut şi noțiunea de **independență față de dispozitiv** a programelor, adică aceeaşi operație de I/O să poată fi realizată de pe diferite dispozitive fizice. Acest lucru se realizează prin aşa zisele **dispozitive logice de I/O**, ce reprezintă nişte identificatori utilizați de programe, care sunt asociați prin intermediul SO dispozitivelor fizice.
- Utilizarea zonelor tampon (buffere) este un alt concept utilizat de SO în scopul utilizării eficiente a CPU. Informațiile de pe discuri care vor fi prelucrate de CPU sunt aduse în zone de memorie internă numite buffere.

Generația a III-a de calculatoare

- Conceptul de **multiprogramare** reprezintă modul de exploatare a unui SC cu un singur procesor central, care presupune existența simultană în memoria internă a mai multor programe, care se execută concurent. CPU este componenta hardware a calculatorului formată îndeosebi din componente electronce, pe când unitățile de I/O sunt formate din componente mecanice. Deci, pe de o parte CPU este mai scumpă, iar pe de altă parte ea lucrează mult mai rapid decât unitățile de I/O. Astfel, s-a pus problema utilizării cât mai eficientă a unității centrale a calculatorului. Multiprogramarea a rezolvat această probleme. Pe scurt, lucrul în multiprogramare se desfăşoară astfel: în fiecare moment CPU execută o instrucțiune a unui program(starea RUN); restul programelor, fie că aşteaptă apariția unui eveniment extern, de exemplu terminarea unui I/O etc. (starea WAIT), fie că sunt pregătite pentru a fi servite în orice moment de către CPU (starea READY). Trecerea unui program din starea RUN în starea WAIT este realizată de către program, în momentul când trebuie să execute o instrucțiune de I/O. Trecerea programelor din starea RUN în starea READY şi invers este realizată de către SO pe baza unui algoritm de planificare, concept pe care îl vom detalia într-o secțiune următoare.

Canalul de intrare – iesire

- Pentru a creşte gradul de exploatare al CPU prin eliminarea timpilor de aşteptare al acesteia, datorați diferenței între vitezele de execuție a echipamentele electronice ale CPU şi celor mecanice ale dispozitivelor de I/O, a apărut, la nivelul tehnologiei hard existente la vremea respectivă **canalul de intrare – iesire.** Acesta este un procesor specializat pe operații de I/O, care poate funcționa în paralel cu CPU. Pentru a fi lansat în execuție, canalul primeşte de la CPU o comandă de efectuare a unei operații de I/O. După lansare, cele două procesoare îşi continuă activitatea în paralel. Sincronizarea între CPU şi canalul de intrare – ieşire se poate realiza prin t**estarea periodica (pooling) a perifericelor de către CPU** sau **printr-o întrerupere lansată de către periferic**. O î**ntrerupere** este o rutină aflată la o adresă fixă de memorie lansată în urma apariției unui semnal hard, care perturbă execuția firească a instrucțiunilor programului în curs. Modul de lucru al întreruperilor va fi prezentat mai târziu. Printre altele, prin intermediul lor se realizează comunicarea dintre SC şi dispozitivele lui periferice (discuri, imprimante etc.). Rutina de întrerupere determină perifericul care a emis întreruperea şi, eventual dă perifericului o nouă comandă de I/O.8

- Relația dintre CPU şi un dispozitiv periferic se realizează astfel: fiecare periferic conține o zonă tampon proprie, capabilă să păstreze o înregistrare (o linie de imprimantă, imaginea unei cartele). CPU, printr-o rutină de I/O, acționează dual, în funcție de operația efectuată. În cazul scrierii, pune din memorie informații în această zonă tampon, iar în cazul citirii preia informațiile şi le depune în memorie. Dispozitivul periferic acționează şi el dual, în funcție de operația care îi este comandată. În cazul scrierii, ia informațiile din zona tampon proprie şi le depune pe suport. În cazul citirii, ia informațiile de pe suport şi le depune în zona tampon proprie.
- În prezent, SC folosesc două tipuri de canale: **selector** si **multiplexor**. Canalul selector este destinat să realizeze schimbul dintre memeorie şi perifericele rapide (discuri), care lucrează, la un moment dat cu un singur periferic. Canalul multiplexor este capabil să lucreze simultan cu mai multe dispozitive periferice.

SPOOLING - Simultaneous Peripherical Operation On-Line

- Au apărut discurile magnetice, care au permis operarea on-line, simultan, cu mai multe periferice (SPOOLING). El s-a obținut prin îmbinarea utilizării zonelor tampon multiple cu conversiile off-line şi cu multiprogramarea.
- Sistemul SPOOLING funcționează astfel:

- Se citesc de la un cititor, cartelele care compun un job.
- Când jobul a fost citit complet, imaginile cartelelor sunt depuse într-un buffer pe disc. Un astfel de job spunem că se află în starea HOLD.
- Când CPU este liber, el alege unul dintre joburile aflate în stare HOLD şi-l lansează în execuție.
- Liniile “tipărite” de jobul în execuție sunt depuse într-un buffer pe disc.
- Atunci cand jobul a fost executat complet, bufferul lui de ieşire pe disc devine disponibil pentru listare. Spunem că jobul se află în starea FINISH.
- Conversia de ieşire se lansează automat, listând pe o imprimanta bufferul unui job aflat în starea FINISH.

- Operațiile de conversie se fac în paralel cu execuția în regim de multiprogramare. Dacă pentru execuție sunt disponibile mai multe zone ale memoriei interne(partiții), atunci toate au acces la cozile HOLD si FINISH, executandu-se în paralel mai multe joburi. De remarcat la tehnica SPOOLING este **simultaneitatea**. La un moment dat sistemul are în lucru trei categorii de joburi:

- joburi în **curs de citire** de la unul sau mai multe cititoare; aceste joburi sunt trecute în coada HOLD.

- joburi în **curs de execuție;** numărul lor poate fi cel mult egal cu numărul de partiții disponibile sp lucreze în multiprogramare;

- joburi în curs de listare; numărul lor poate fi cel mult egal cu numărul de imprimante active conectate la sistem; aceste joburi sunt preluate din coada FINISH.

- Această tehnică se foloseşte şi la calculatoarele actuale pentru listările la imprimantă.
- Dintre sistemele de operare specifice acestei generații de calculatoare, cele mai reprezentative sunt sistemul SIRIS pentru calculatoarele IRIS 50, care s-au produs şi în țara noastră sub denumirea FELIX 256/512/1024 şi sistemul OS/360 pentru calculatoarele IBM 360.

Generației a-IV-a de calculatoare

- Sistemele interactive:Un sistem interactiv permite comunicarea on-line dintre utilizator şi sistem. De regulă, utilizatorul are la dispoziție un terminal cu tastatură şi ecran, prin care comunică cu sistemul. În astfel de sisteme, utilizatorul dă comanda, asteaptă răspunsul şi, în funcție de rezultatul furnizat de comanda precedentă, decide asupra noii comenzi. El poate, astfel să experimenteze uşor şi să vadă rezultatele imediat. SO pentru SC interactive conțin (cel puțin) câte un editor de texte pentru corectarea programelor sursă şi (cel puțin) câte un depanator interactiv care poate asista alte programe în execuție. Spre deosebire de sistemele seriale, sistemele interactive au un timp de răpuns rezonabil de ordinul secundelor, eventual al minutelor.
- **time-sharing** (timp partajat) este o variantă a multiprogramării. A apărut la generația a III-a de calculatoare (sistemul CTSS dezvoltat de MIT, sistemul MULTICS (MULTiplexed Information and Computing Service şi mai ales sistemul UNIX . A fost dezvoltat pe scară largă de generația a IV-a de calculatoare. Ele au fost proiectate pentru a permite mai multor utilizatori să fie conectați simultan la acelaşi sistem de calcul. Acest sisteme îmbină **interactivitatea** şi **multiprogramarea**.

Terminal virtual. Fiecare utilizator stabileşte o sesiune cu sistemul, prin intermediul unui **terminal virtual**, care este o simulare a hardware-ului calculatorului şi este implementat de către SO. Sistemul comută rapid de la un program la altul, înregistrând comenzile solicitate de fiecare utilizator prin terminalul său. Deoarece o tranzacție a utilizatorului cu sistemul necesită un timp de lucru mic al CPU, rezultă că într-un timp scurt, fiecare utilizator este servit cel puțin o dată. În acest fel, fiecare utilizator are impresia că lucrează singur cu sistemul. Dacă sistemele seriale încearcă să optimizeze numărul de job-uri prelucrate pe unitatea de timp, sistemele timesharing realizează o servire echitabilă a mai multor utilizatori, aflați la diverse terminale.

Diferențiere între noțiunile de job şi proces (program în execuție). În sistemele timesharing, la acelaşi moment un job poate executa două sau mai multe procese, pe când în sistemele seriale un job presupune un singur proces. Într-un sistem timesharing multiprogramat procesele se mai numesc şi **task**-uri iar un astfel de calculator se mai numeşte şi sistem **multitasking**.

Redirectarea si **legarea in pipe.** A apărut la calculatoarele din generația a III-a, dar a fost dezvoltat şi utilizat îndeosebi de calculatoarele din generația a IV-a. Aplicarea lor presupune că fiecare program lansat de la un terminal are un **fişier standard de intrare** şi un **fişier standard de ieşire.** De cele mai multe ori acestea coincid.cu tastatura, respectiv cu terminalul de la care se fac lansările. Redirectarea intrărilor standard (intrare sau/şi ieşire) permite utilizatorului să înlocuiască intrarea standard cu orice fişier, respective să se scrie rezultatele, afişate de obicei pe ecran într-un fişier oarecare, nou creat sau să fie adăugate la un fişier deja existent. Informațiile de redirectare sunt valabile din momentul lansării programului pentru care se cere acest lucru şi până la terminarea lui. Dupa terminarea programului se revine la fişierele standard implicite.

Dintre SO cele mai cunoscute ale generației a IV-a de calculatoare, amintim sistemele DOS, Windows, Unix pentru calculatoare personale şi pentru rețele de calculatoare, sistemele RSX pentru minicalculatoare (calculatoare mainframe).

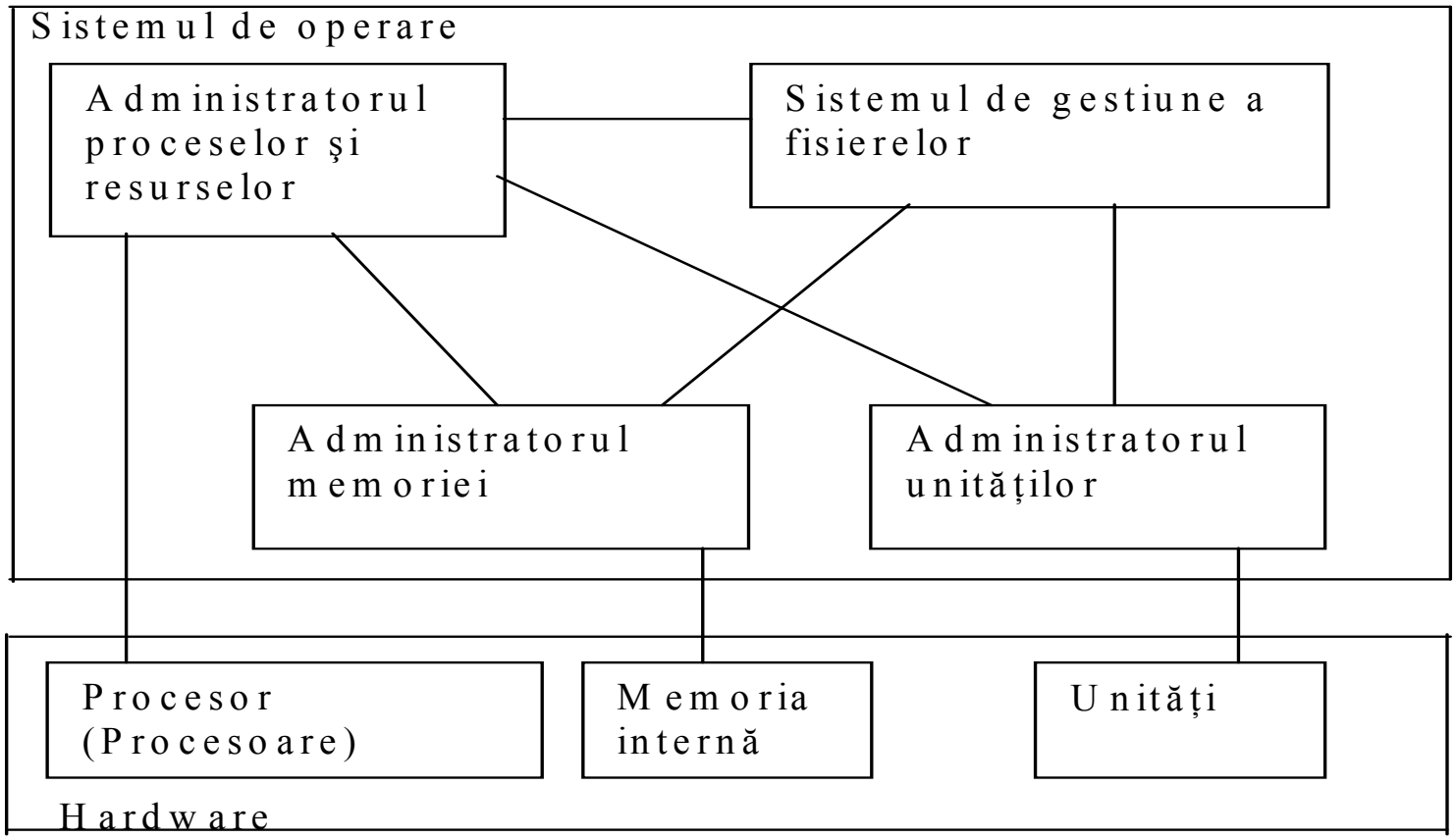
Conceptul general de proces

- Dacă sistemele de operare cu prelucrare în loturi executau lucrări(job-uri), sistemele de operare moderne bazate pe divizarea timpului execută task-uri. Aceste două concepte corespund termenului de proces. În cadrul sistemelor de operare, procesul reprezintă o entitate activă, un program în execuție ale cărui instrucțiuni sunt parcurse secvențial şi executate de către unitatea centrală a calculatorului. Dacă prin program înțelegem o entitate statică, o codificare într-un anumit limbaj de programare a unui algoritm, sub forma unui fişier stocat pe un suport extern de informație, prin proces definim o entitate dinamică, încărcat în memoria internă a sistemului de calcul. Unul sau mai multe procese pot fi asociate unui program, dar ele sunt considerate entități distincte.
- Orice proces foloseşte mai multe resurse ale sistemului de calcul: **contorul de program** (PC – **P**rogram **C**ounter) este un registru al UC care conține adresa de memorie a următoarei instrucțiuni care urmează să fie executată; **stiva de program** conține date utilizate în execuție, parametri ai subprogramelor, adrese de retur şi alte variabile locale; **secțiunea de date**, conține variabile globale; **timpul de lucru** al UC; **fişiere**; **dispozitive de I/O** etc. Aceste resurse sunt alocate procesului fie în momentul creerii lui, fie în timpul execuției lui

- În multe sisteme, procesul este unitatea de lucru. Astfel de sisteme constau dintr-o colecție de procese. Sistemul de operare, ca şi componentă soft, este format din mai multe procese, care execută codul de sistem, pe când procesele utilizator execută programe scrise de aceştia. Toate aceste procese se execută concurent la nivelul unui sistem de calcul, sau al unei rețele de calculatoare. În mod tradițional un proces conținea un singur fir de execuție; sistemele de operare moderne, privesc un proces ca fiind format din unul sau mai mute fire de execuție.
- Execuția proceselor este un mecanism combinat hard şi soft. Componenta software care este implicată în administrarea proceselor şi firelor de execuție este sistemul de operare, ale cărui sarcini sunt:
- crearea şi distrugerea proceselor şi firelor de execuție;
- planificarea proceselor;
- sincronizarea proceselor;
- comunicarea între procese;
- manipularea interblocării proceselor.

Funcțiile de bază ale sistemelor de operare

- Aceste funcții cooperează între ele pentru a satisface cerințele utilizatorilor. În figura urm. sunt prezentate interacțiunile între modulele care realizează funcțiile SO, precum şi între aceste module şi componentele hard ale sistemului de calcul.



Administrarea proceselor şi a resurselor

- Procesul reprezintă unitatea de bază a calculului, definită de un programator, iar resursele sunt elemente ale mediului de calcul necesare unui proces pentru a fi executat. Crearea, execuția şi distrugerea proceselor, comunicarea între ele, împreună cu alocarea resurselor după anumite politici, sunt aspecte deosebit de importante care vor fi discutate în detaliu în cursurile următoare.

Gestiunea memoriei.

- SO alocă necesarul de memorie internă solicitat de procese şi asigură protecția memoriei între procese. O parte este realizată prin hard, iar o parte prin soft. Această problemă va fi obiectul cursurilor următoare.

Gestiunea fişerelor

- **SO conține o colecție de module(SGF) prin intermediul cărora se asigură deschiderea, închiderea şi accesul utilizatorului la fişierele rezidente pe diferite suporturi de informații. Componentă de bază a SO, este cel mai des invocată de către utilizator şi de către operator.**

Administrarea unităților

- **Se referă la modul în care discurile, terminalele, imprimantele, procesoarele, memoria şi alte componente hardware sunt alocate, protejate în timpul alocării şi partajate după anumite reguli. Accesarea acestor componente se realizează prin intermediul driverelor**.

Arhitectura Von Neumann

- Sistemele de calcul se bazează pe conceptul de arhitectură Von Neumann, conform căruia partea de hardware este formată din:

►**Unitatea centrală de calcul** (CPU - Central Processing Unit), compusă din **unitatea aritmetică** şi **logică** (ALU-Arithmetical-Logical Unit) şi **unitatea de control**.

►**Unitatea de memorie primară** sau **executabilă** sau **internă**.

►**Unități de I/O**.

- Toate unitățile sunt conectate folosind o **magistrală** (bus), care se împarte într-o **magistrală de date** şi una **de adrese.**
- Pentru memorie şi alte unități se poate adăuga o magistrală de I/O, care nu este folosită de CPU. O astfel de organizare, permite ca o unitate să poată citi/scrie informații din memorie, fără alocarea CPU. Fiecare magistrală poate fi gândită ca fiind formată din mai multe linii (fire) paralele, care pot păstra o cifră binară.

Unitatea aritmetică şi logică

- Conține, pe lângă unitatatea funcțională un număr de **regiştri generali** şi de **stare**. Regiştrii generali sunt folosiți în efectuarea operațiilor aritmetice şi logice, atât pentru memorarea operanzilor încărcați din memorie, cât şi a rezultatului operației, care apoi va fi salvat într-o locație de memorie.
- CPU extrage şi execută instrucțiunile cod maşină ale procesului încărcat în memoria internă. În acest sens, CPU conține:
- ►o componentă care extrage o instrucțiune memorată într-o locație de memorie;
- ►o componentă care decodifică instrucțiunea;
- ►o componentă care se ocupă de execuția instrucțiunii, împreună cu alte componente ale SC.
- **Regiştrii** contor de program **PC (Program Counter),** respectiv registrul instrucțiune **IR (Instrucțion Register),** conțin adresa de memorie, respectiv o copie a instrucțiunii în curs de prelucrare**.**

Unitățile de I/O

- Unitățile de I/O sunt folosite pentru a plasa date în memoria primară şi pentru a stoca cantități mari de date pentru o perioadă lungă de timp. Astfel, ele pot fi unități de stocare (unități bloc), cum ar fi discurile, respectiv unități caracter cum ar fi tastatura, mouse-ul, display-ul terminalului precum şi unități de comunicație, cum ar fi portul serial conectat la un modem sau o interfață la rețea. Fiecare unitate foloseşte un controller de unitate pentru a o conecta la adresele calculatorului şi la magistrala de date. Controller-ul oferă un set de componente fizice pe care instrucțiunile CPU le pot manipula pentru a efectua operații de I/O. Ca şi construcție, controller-ele diferă, dar fiecare oferă aceeaşi interfață de bază. SO ascunde aceste detalii de funcționare ale controller-lor, oferind programatorilor funcții abstracte pentru accesul la o unitate, scrierea/citirea de informații etc.
- Controller-ul de unitate furnizează o interfață folosită de către mediul microprogramat de la cel mai înalt nivel. Componenta SO care manipulează dispozitivele de I/O este formată din driverele de unitate.

Moduri de lucru ale procesorului

- **Setarea modului de lucru**. Procesoarele contemporane conțin un bit care defineşte modul de lucru al procesorului. Acest bit poate fi setat în modul **utilizator** sau **supervizor.** În modul supervizor, procesorul poate executa orice instrucțiune cod maşină, pe când în modul utilizator el poate executa numai o parte dintre aceste instrucțiuni. Instrucțiunile care pot fi executate numai în modul supervizor se numesc instrucțiuni privilegiate. De exemplu, astfel de instrucțiuni sunt cele de I/O. Un proces, dacă este executat în mod utilizator, el nu poate să-şi execute propriile instrucțiuni de I/O. De aceea, aceste instrucțiuni sunt executate prin intermediul SO. Când un program de aplicație face o cerere către sistem, o instrucțiune cod maşină specială este apelată pentru a comuta procesorul în modul supervizor şi începe să execute driverul unității respectiv.
- Corespunzător celor două moduri de lucru, memoria internă este împărțită în zona de memorie utilizator şi zona de memorie supervizor. Dacă bitul mod de lucru este setat pe utilizator, atunci procesul respectiv are acces numai la zona cu acelaşi nume. Astfel, se realizează şi protecția zonei de memorie supervizor.

Nucleul **SO**

- acea parte a SO care este executată în modul supervizor. Alte procese, legate de diverse aplicații ale utilizatorilor sau chiar aplicații soft de sistem sunt executate în mod utilizator. Execuția acestor instrucțiuni nu afectează securitatea sistemului. Când un proces doreşte să execute anumite operații în mod supervizor, atunci se va face o **comutare** din modul utilizator, în cel supervizor. Acest lucru se realizează prin intermediul unei instrucțiuni **trap,** numită instrucțiune de apel al supervizorului, care setează bitul de mod de lucru şi face un salt la o locație de memorie, care se află în spațiul de memorie protejat, locație care conține începutul unei proceduri sistem care va rezolva cererea procesului. Când execuția acestei rutine supervizor s-a terminat, SO va reseta bitul de mod de lucru din supervizor în utilizator.

Apelurile de sistem

- furnizează o interfață între un proces şi sistemul de operare. Prin intermediul acestora, un program utilizator comunică cu SO şi cere anumite servicii de la acesta.
- Apelurile de sistem pot fi împărțite în cinci categorii importante:

►**controlul proceselor**(încărcarea, execuția, sfârşitul, abandonarea, setarea şi obținerea atributelor, alocarea şi eliberarea de memorie etc);

►**manipularea fişierelor**(creere, ştergere, deschidere, închidere, citire, scriere, repoziționare, setarea şi obținerea atributelor);

► **manipularea unităților**(cerere şi eliberare unitate, citire scriere şi repoziționare, obținerea şi setarea atributelor, ataşarea/detaşarea logică);

► **întreținerea informațiilor(**obținerea şi setarea timpilui sau datei calendaristice sau a sistemului, obținerea de informații despre componentele fizice şi logice ale sistemului şi posibilitatea modificării lor**);**

► **comunicații**(crearea şi anularea unei conexiuni, transmiterea şi primirea de mesaje, transferul informațiilor de stare, ataşarea/detaşarea logică a unităților la distanță).

Limbajele de asamblare şi limbajele evoluate moderne conțin instrucțiuni (comenzi) prin care sunt lansate apeluri de sistem. După transmiterea parametrilor, este declanşată o instrucțiune trap, pentru oferi controlul SO. Când a terminat de executat rutina respectivă, SO returnează un cod de stare într-un registru, care specifică terminarea normală sau anormală şi execută o instrucțiune de revenire din instrucțiunea trap

CURS 2

GESTIUNEA PROCESELOR

Obiectivele cursului:

- prezentarea starilor si tranzitiilor unui proces si a modului cum sunt gestionate de SO;

- sincronizarea proceselor;

-gestionarea resurselor critice;

- probleme clasice de coordonare;

- planificarea proceselor.

1. Stările unui proces. În timpul execuției, procesul își schimbă **starea**, definită ca mulțime a activităților executate de proces. Stările unui proces sunt: **new** (crearea procesului), **run**(procesul este în execuție), **wait**(procesul aşteaptă apariția unui anumit eveniment, cum ar fi terminarea unei operații de I/O sau primirea unui semnal), **ready**(procesul este gata de execuție aşteptînd să-i fie alocat un procesor), **finish**(terminarea execuției procesului). În sistemele cu multiprogramare, procesele trec dintr-o stare în alta, în funcție de specificul fiecăruia sau de strategia de planificare adoptată. În figura 1 este prezentată diagrama tranzițiilor unui proces în timpul execuției.

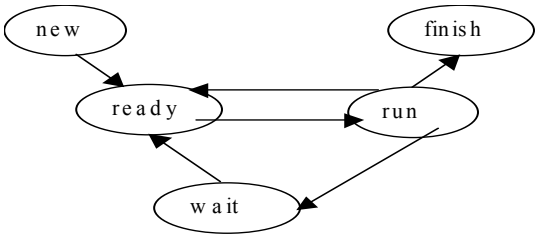


Figura 1 Diagrama stărilor şi tranzițiilor unui proces

Trecerea din starea **new** în starea **ready** înseamnă că procesul este luat în considerare pentru a fi executat; trecerea din starea **ready** în starea **run** se produce atunci când procesului îi este alocat un procesor; trecere inversă se produce atunci când, conform unei politici de planificare, procesorul trebuie alocat unui alt proces; trecerea procesului din starea **run** în starea **wait** se produce atunci când procesorul întâlneşte o cerere de executatre a unei operații de intrare/ieşire; trecerea din starea **wait** în starea **ready** se produce atunci când operația de I/O s-a terminat; trecerea în starea **finish**, înseamnă terminarea execuției procesului.

Fiecare proces este reprezentat în sistem prin **blocul de control al procesului**(PCB – **P**rocess **C**ontrol **Bl**ock), care este o zonă de memorie, ce conține următoarele informții: **starea procesului**, ale cărei valori le-am prezentat înainte; valoarea **contorului de program**, care indică adresa următoarei instrucțiuni care urmează să fie executată pentru acest proces; valorile **regiştrilor unității centrale**; când apare o întrerupere şi CPU trece la execuția altui process, valorile acestor regiştri şi ale contorului de program trebuie salvate, pentru a permite reluarea corectă a execuției procesului; când procesul primeşte din nou serviciile CPU, valorile salvate în PCB-ul procesului respectiv sunt atribuite regiştrilor acesteia. informații de **contabilizare**, referitoare la valoarea timpului CPU utilizat de proces, valori limită ale timpului CPU, între care este executat procesul respective, conturi utilizator, numere ale altor procese etc; informații de I/O (lista unităților de I/O alocate procesului, lista fişierelor deschise de process etc.); informații

de **planificare** a CPU(prioritatea procesului, pointer către coada de planificare a execuției procesului etc.); informații cu privire la modalitatea de **alocare** a memoriei interne; un pointer către un alt PCB.

2. Sincronizarea proceselor. Despre un proces se spune că este **independent** dacă execuția lui nu afectează sau nu poate fi afectată de execuția altor procese din sistem. Un astfel de proces poate fi **oprit sau repornit** fără a genera efecte nedorite, este **determinist**(ieşirile depind numai de starea de intrare), **reproductibil**(rezultatele sunt totdeauna aceleaşi, pentru aceleaşi condiții de intrare), **nu partajează date** cu alte procese din sistem. Dacă execuția unui proces poate fi afectată de execuția altor procese din sistem sau poate influența stările unor procese, atunci spunem că procesul este **cooperant**. În acest caz, procesul partajează date împreună cu alte procese din sistem, evoluția lui nu este deterministă, fiind influențată de stările acestora, nu este reproductibil etc.

Sincronizarea proceselor reprezintă un mecanism prin care un proces activ este capabil să blocheze execuția altui proces(trecerea din starea run în ready şi invers), sau să se autoblocheze(să treacă în starea wait), sau să activeze un alt proces.

Problema secțiunii critice. Considerăm un sistem în care există n procese cooperante p_0, p_1,.., p_{n-1}. De exemplu, să presupunem că două dintre procese, p_0 şi p_1 au acces la o variabilă v, primul proces scade o valoare c_0, iar al doilea adaugă o valoare c_1 la v. Dacă secvența de operații executate de cele două procese asupra variabilei v se derulează necontrolat, atunci rezultatele sunt total imprevizibile. Putem extinde problema enunțată asupra fişierelor, bazelor de date, tabele din memorie etc.

O astfel de resursă logică partajată de către două procese se numeşte **resursă critică**. Partea de program(segmentul de cod) în care un proces accesează o resursă critică se numeşte **secțiune critică**. Problema care se pune este stabilirea unor reguli după care un proces poate să intre în propria sa secțiune critică şi să comunice celorlate procese cooperante când a părăsit-o. Astfel, structura unui proces este cea descrisă în figura 2.

```
do{
    <secțiune de intrare>
    <secțiune critică>
    <secțiune de ieşire>
    <secțiune rămasă>
 }while(1);
```

Figura 2 Structura unui proces

O soluție corectă a problemei secțiunii critice trebuie să satisfacă următoarele condiții:

- **excludere mutuală**: la un anumit moment, un singur proces îşi execută propria lui secțiune critică;
- **evoluție(progres)**: un proces care nu este în secțiunea sa critică, nu poate să blocheze intrarea altor procese în propriile lor secțiuni critice, atunci când acestea doresc acest lucru;
- **aşteptare limitată**: între momentul formulării unei cereri de acces în propria secțiune critică de către un proces şi momentul obținerii accesului, trebuie acordat un număr limitat de accese celorlalte procese în propriile lor secțiuni critice.

Semafoare. Conceptul de semafor a fost introdus de Dijkstra. Un semafor s este o pereche (v,q); v este un întreg ce reprezintă valoarea semaforului, fiind inițializat la crearea semaforului cu o valoare v_0 iar q este o coadă, inițial vidă în care sunt introduse procesele care nu reuşesc să treacă de semaforul s.

Asupra semaforului pot acționa două operații indivizibile, W(wait) şi S(signal), executate asupra unui anumit proces. Operația W poate fi considerată ca o încercare de trecere a semaforului iar operația S ca o permisiune de trecere. Efectul celor două primitive este descris în figura 3; p este un proces care încearcă să treacă de semaforul s.

Observații. 1. Dacă nw şi ns reprezintă numărul primitivelor W, respectiv S executate pe semaforul s, atunci

$$v = v_0 - nw + ns \qquad (2.1)$$

2. Valoarea inițială v_0 este un întreg pozitiv.

3. La un moment dat, dacă valoarea semaforului v este negativă, respectiv pozitivă, valoarea ei absolută reprezintă numărul proceselor blocate în coadă, respectiv numărul proceselor care pot trece de semaforul s.

4. Dacă presupunem că la un moment dat numărul proceselor care au trecut cu succes de semaforul s este nt, atunci evident că nt ≥ nw. De asemenea:

$$nt = \min\{v_0+ns, nw\} \qquad (2.2)$$

5. Modificarea valorilor întregi ale semafoarelor trebuie să fie executate indivizibil, adică atunci când un process modifică valoarea semaforului nici un alt proces nu poate face acest lucru.

```
do {  v--:
      If (v < 0) // procesul p  se introduce în coadă
            {Stare(p)  = wait;
            q← p}
            } while(1):
```

a) Efectul primitivei W

```
do {  v:=v+1:
      If  (v <= 0)  // procesul p este scos din coadă şi activat
            {       q→ p;
                    Stare(p) = ready}
                    } while(1):
```

b) Efectul primitivei S

Figura 3. Descrierea primitivelor unui semafor.

Utilizarea semafoarelor în gestionarea resurselor critice. Putem folosi semafoarele pentru a rezolva problema unei secțiuni critice, partajată de către *n* procese. Procesoarele folosesc în comun un semafor mutex (**mut**ual **ex**clusion), a cărui valoare inițială este 1. Fiecare proces p_i este organizat ca în figura 4

Observăm că secțiunea critică este precedată de o primitivă W, care aplicată semaforului mutex va permite intrarea în secțiunea critică numai în cazul în care resursa critică cerută este liberă. După execuția resursei critice, urmează execuția primitivei S asupra aceluiaşi semafor, al cărei efect este eliberarea resursei şi eventual deblocarea unui proces din coada semaforului, dacă aceasta este nevidă, care să treacă în secțiunea sa critică. De asemenea, operațiile W şi S trebuie să fie indivizibile.

```
do {    W(mutex);
        <secțiune critică>
        S(mutex);
        <secțiune rămasă>
 } while(1);
```

Figura 4. Structura unui proces care utilizează semafoare pentru sincronizare

Pentru a demonstra corectitutinea soluției, mai întâi vom proba proprietatea de excludere mutuală. Numărul proceselor care se se află la un moment dat în secțiunea critică este nt-ns. Deoarece valoarea inițială a semaforului este 1, conform relației (2.1), avem inegalitatea nt-ns ≤ 1, care are semnificația că cel mult un proces se poate afla la un moment dat în secțiunea sa critică. Mai trebuie să arătăm că dacă nici un proces nu se află în secțiunea sa critică, atunci intrarea unui proces în secțiunea sa critică va avea loc după un interval finit de timp, care este echivalent cu a arăta că dacă nici un proces nu se află în secțiunea sa critică, atunci nu există procese blocate în coada de aşteptare a semaforului. Dacă nici un proces nu se află în secțiunea sa critică, atunci nt = ns. Dacă ar exista procese blocate în coada semaforului mutex, atunci nt > nw şi deci în conformitate cu (2.2) nt=1+ns, ceea ce reprezintă o contradicție.

3. Probleme clasice de coordonare a proceselor

Problema producător/consumator. Procesele de tip producător/ consumator apar destul de frecvent în cadrul sistemelor de operare, de exemplu atunci când două procese comunică între ele. Procesul de tip producător **generează informațiile**(mesajele), care vor fi **folosite** de către procesul de tip consumator. Această problemă poate fi rezolvată prin utilizarea unei zone de memorie accesibilă tuturor proceselor care cooperează. Ea este structurată în funcție de natura informațiilor schimbate şi de regulile care guvernează comunicația propriu-zisă. Această zonă comună este o resursă critică, ce trebuie protejată.

Fie procesele p_i şi p_j care comunică între ele. Cele două procese partajează o zonă comună ZC, care este divizată în n celule-mesaj, în care se depun(producătorul), respectiv se extrag(consumatorul) mesaje de dimensiune fixă. Procesele p_i şi p_j au o evoluție ciclică ce cuprinde operațiile de producere şi de depunere, respectiv de extragere şi de consumare a unui mesaj din ZC, aşa cum este redat în figura 5. Depunerea, respectiv extragerea unui mesaj din zona comună, trebuie să satisfacă următoarele restricții:

a) Consumatorul nu poate să extragă un mesaj, pe care producătorul este în curs să-l depună, adică operațiile de depunere şi extragere trebuie să se execute în excludere mutuală la nivelul mesajului.
b) Producătorul nu poate să depună un mesaj atunci când ZC este plină, iar dacă ZC este vidă, consumatorul nu poate extrage nici un mesaj.

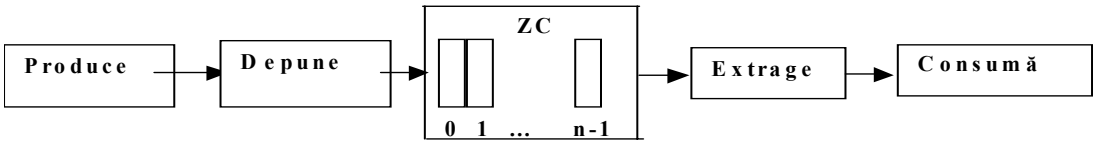


Figura 5 Structura zonei comune ZC

Aceste situații limită trebuie detectate şi semnalate proceselor care depind de ele. Condiția a) poate fi satisfăcută prin introducerea unui semafor de excludere mutuală, asociat unei zone comune ZC, mutex(ZC), inițializat cu valoarea 1. Restricția b) se poate respecta, utilizând semafoarele s_0 şi s_1 pentru a înregistra, în orice moment, starea de ocupare a zonei ZC; s_0, inițializat cu 0 va indica numărul de celule-mesaj depuse în ZC, iar s_1, inițializat cu n, va indica numărul celulelor-mesaj libere în ZC. Structura proceselor p_i, respectiv p_j este prezentată în figura 6.

Problema cititori/scriitori. Atunci când o structură de date, de exemplu un fişier sau o bază de date este accesată în comun de către mai multe procese concurente, dintre care unele doresc doar **să citească** informații, iar altele doresc **să actualizeze**(citească, scrie sau modifice) structura respectivă, cele două tipuri de procese se împart generic în **cititori** şi respectiv **scriitori.** Se impun următoarele restricții:

a) Doi cititori pot accesa simultan obiectul partajat;
b) Un cititor şi un scriitor nu pot accesa în acelaşi timp obiectul disputat.
c)

do { **construieşte** mesaj: W(s_1); W(mutex(ZC)); ZC←mesaj; S(mutex(ZC)); S(s_1); } while(1);	do { W(s_0); W(mutex(ZC)); ZC→mesaj; S(mutex(ZC)); S(s_0); **Consumă mesaj** } while(1);
p_i: proces producător de mesaje	p_j: proces consumator de mesaje

Figura 6. Descrierea proceselor producător/consumator.

Pentru a respecta cele două condiții, accesul la resursă va fi protejat; problema de sincronizare astfel apărută se numeşte **cititori/scriitori.** Problema cititori/scriitori poate fi formulată în mai multe variante, toate implicând lucrul cu priorități. Cea mai simplă dintre acestea, impune ca nici un cititor să nu fie obligat să aştepte decât în cazul în care un scriitor a obținut deja permisiunea de utilizare a obiectului disputat. O altă variantă cere ca, de îndată ce un scriitor este gata de execuție, el să îşi realizeze scrierea cât mai repede posibil, adică dacă un scriitor aşteaptă pentru a obține accesul la obiectul disputat, nici un cititor nu va mai primi permisiunea de a accesa la resursă.

O soluție a primei variante a problemei cititori/scriitori permite proceselor să partajeze semafoarele A("Acces"), Sc("Scrie") şi variabila cu valori întregi C("contor"). Sc, inițializat cu 1 este folosit de ambele tipuri de procese. El asigură excluderea mutuală a scriitorilor, este folosit de către primul/ultimul cititor care intră/iese din propria secțiune critică, dar nu şi de către cititorii care intră/ies în/din aceasta în timp ce alți cititori se află în propriile secțiuni critice.

Semaforul A, inițializat cu 1 este folosit pentru protejarea variabilei C, care este inițializată cu 0 şi are rolul de a memora numărul proceselor care citesc din obiectul partajat la momentul considerat. Primul cititor care accesează resursa critică(C=1) blochează accesul scriitorilor, iar ultimul cititor (C=0), deblochează accesul la obiectul

partajat.Structura generală a unui proces cititor este prezentată în figura 7.a, iar a unui proces scriitor în figura 7.b.

do {
W(A);
C++;
if (C ==1) W(Sc);
S(A);
<citirea>
W(A);
C--;
if (C == 0) S(Sc);
} while(1);
	do {
	W(Sc);
	<scrirea>
	S(Sc);
	} while(1);

a) Procesul cititor	b) Procesul scriitor

Figura 7. Descrierea structurii proceselor de tip scriitor/scriitor

Dacă un scriitor se află în propria secțiune critică în timp ce N cititori se află în stare de aşteptare, atunci, în şirul de aşteptare, un cititor este asignat prin semaforul Sc iar ceilalți N-1 cititori sunt asignați prin semaforul A. De asemenea, atunci când un scriitor execută S(Sc), se poate relua fie execuția unui singur scriitor, fie a cititorilor aflați în aşteptare, selecția fiind făcută de către planificator.

Mecanisme de planificare

Aşa cum am menționat în cursul I, un sistem de operare care utilizează multiprogramarea, permite ca mai multe procese să fie încărcate în memoria executabilă la un moment dat şi să folosească în comun CPU, utilizând **multiplexarea timpului**. Unul din motivele pentru care este utilizată multiplexarea, este că SO însuşi este implementat ca unul sau mai multe procese, deci trebuie să existe o modalitate prin care procesele utilizator şi cele ale SO să partajeze CPU. De asemenea, procesele care execută operații de I/O, trebuie să cedeze CPU altor procese. Mecanismul de planificare este o componentă a administratorului de procese, care manipulează eliberarea CPU de către un proces în execuție şi selectarea unui alt proces care urmează să fie servit de către CPU, pe baza unei anumite strategii.

Organizarea planificării proceselor. Planificatorul este responsabil de multiplexarea proceselor în vederea utilizării CPU. Când este momentul ca un proces să elibereze CPU şi să treacă în starea READY sau WAIT, din mulțimea proceselor aflate în starea READY este selectat un alt proces care să utilizeze CPU. **Politica de planificare** determină când este momentul ca un proces să elibereze CPU şi care dintre procesele aflate în starea READY să treacă în starea RUN. **Mecanismul de planificare** determină modul cum administratorul de procese multiplexează CPU de către mai multe procese şi cum este alocată CPU unuia dintre proces. Mecanismul de planificare este compus din mai multe părți, care diferă în funcție de modul de implementare al SO respectiv. Figura 8 prezintă cele trei componente.

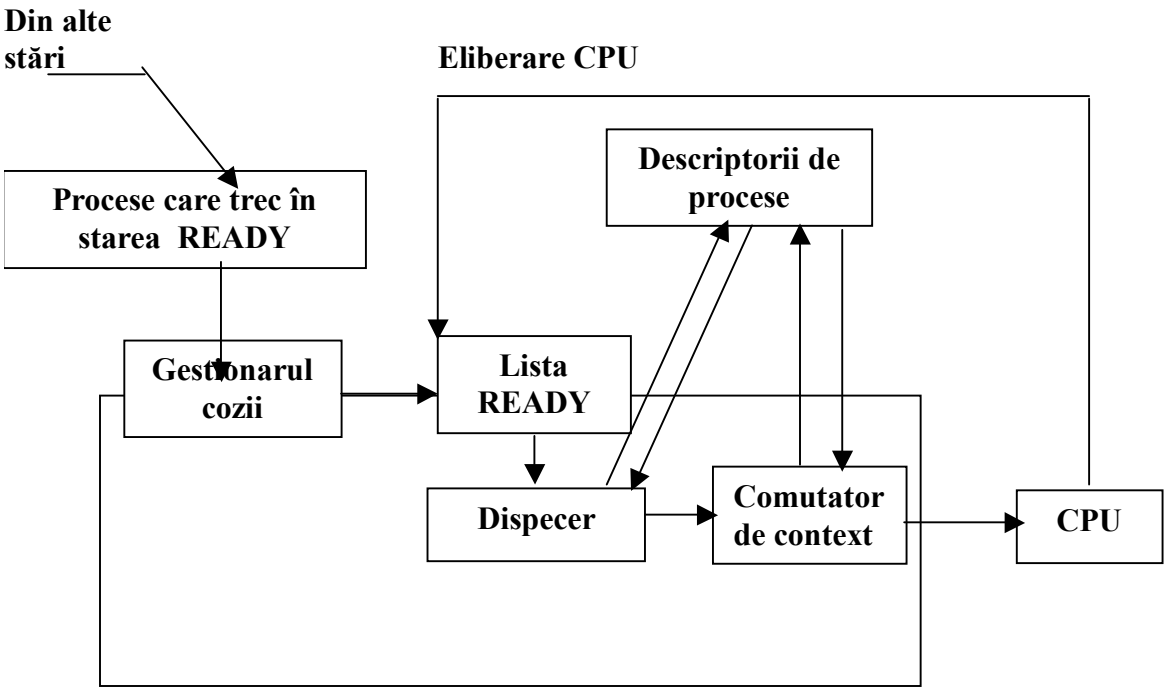


Figura 8. Structura planificatorului

Gestionarul cozii proceselor care aşteptă să primească serviciile CPU. Când un proces trece în starea READY, descriptorul său este prelucrat şi este introdus într-o coadă înlănţuită numită **lista READY**; această coadă este organizată pe priorităţi, care sunt calculate de către gestionarul cozii, ori la introducerea procesului în coadă, sau în momentul selectării unui proces pentru a fi servit de către CPU.

Comutatorul de context(CS-Context **S**witcher**)** salvează conţinuturile tuturor regiştrilor CPU(PC, IR, starea procesului, starea ALU etc.) relative la procesul în execuţie, în momentul când planificatorul comută CPU de la execuţia procesului respectiv. la un alt proces. Există două modalităţi prin care se execută această comutare de context:

• **voluntară**, care presupune că procesul care deţine CPU, cere el însuşi eliberarea acesteia pentru a fi acordată unui alt proces. Această metodă implică existenţa unei instrucţiuni cod maşină care acţionează asemănător cu o instrucţiune de apel de procedură, în sensul că salvează adresa următoarei instrucţiuni de executat a procesului servit de CPU, după care face un salt la o adresă. Acea adresă conţine instrucţiunea cu care va începe (sau relua) execuţia procesului căruia îi este cedată CPU. El diferă de apelul de procedură prin faptul că adresa următoarei instrucţiuni nu este salvată în stiva procesului, ci undeva la o adresă specială de memorie. O astfel de instrucţiune este asemănătoare unei întreruperi.

• **involuntară**, care presupune că eliberarea CPU se face prin intermediul unei întreruperi. Această metodă presupune existenţa unui **dispozitiv de măsurare a intervalelor de timp**, care să genereze o întrerupere oricând expiră intervalul de timp alocat procesului. O astfel de metodă de planificare se mai numeşte şi planificare cu evacuare sau cu forţare.

Dispecerul este cerut pentru a selecta un nou proces aflat în lista READY, pentru a fi servit de către CPU, indiferent de modalitatea de eliberare a CPU. Acest lucru se poate realiza în două moduri:

• Procesul eliberează în mod voluntar CPU, când face o cere de resursă, trecând astfel controlul administratorului resurselor; CPU fiind eliberată, alte procese o pot utiliza. După terminarea utilizării resursei cerute, procesul revine imediat în starea READY. Dacă resursa cerută nu este disponibilă, procesul este blocat(starea WAIT), până când resursa devine liberă, iar după alocarea şi utilizarea ei trece în starea READY.

• Procesul poate fi evacuat în mod involuntar din utilizarea procesorului.

Salvarea contextului procesului. Când CPU este multiplexată, apar două tipuri de comutare a contextului. În primul caz, se realizează salvarea contextului procesului care este în curs de execuție de către SO şi este încărcat cel al dispecerului. A doua situație constă în cedarea CPU de către dispecer şi încărcarea procesului care urmează să fie servit de către CPU. Salvarea contextului este o operație destul de costisitoare, care poate afecta performanțele. Dacă notăm cu n, numărul regiştrilor generali şi cu m numărul regiştrilor de stare şi dacă presupunem că pentru a salva un singur registru sunt necesare b operații, fiecare instrucțiune de stocare necesitând K unități de timp, atunci pentru salvarea stării procesului sunt necesare (n+m)bK unități de timp.

Performanțe. Planificatorul are o influență majoră asupra performanțelor unui calculator, care lucrează în regim de multiprogramare, deoarece el decide când un proces este servit de către CPU. Dacă momentul când un proces este selectat să fie servit, tinde către momentul când este introdus în lista READY, atunci putem spune că el va fi servit aprope când doreşte. În schimb, dacă un proces este neglijat de către dispecer(fenomenul de „înfometare"), atunci va sta destul de mult în starea READY, deci timpul său de execuție va fi destul de mare. De asemenea, performanțele planificatorului sunt influențate de timpul necesar pentru comutarea contextului. Toate aceste aspecte sunt influențate atât de componentele hardware, cât şi de cele software, reprezentate de componenta de planificare a SO.

Selectarea strategiei. În cele ce urmează vom nota cu $P = \{p_0,..,p_{n-1}\}$ mulțimea celor n procese din sistem, $S(p_i)$ este starea procesului $p_i, i = 0,..,n\text{-}1, S(p_i) \in \{RUN, READY, WAIT\}$. De asemenea $\tau(p_i)$ este timpul efectiv de servire, adică intervalul de timp cât procesul p_i va fi în starea RUN; $W(p_i)$ este intervalul de timp petrecut de proces în starea READY pâna la prima tranziție în starea RUN. De asemenea, $T(p_i)$ reprezintă timpul total petrecut în sistem de proces, din momentul când procesul intră pentru prima dată în starea READY, până când este executat.

În funcție de tipul prelucrării, una dintre aceste mărimi este considerată factor de eficiență. Pentru sistemele cu prelucare în loturi, $T(p_i)$ este considerat factor de eficiență. Bineînțeles că aici p_i este un „job", nu un proces. În cazul sistemelor cu partajarea timpului, factorul de eficiență este **timpul de răspuns**, care se compune din timpul cât procesul(comanda) stă în starea READY şi timpul cât este servit efectiv de CPU.Cererile de servicii adresate unității centrale se intercalează cu cereri de alocări de resurse. Modelul de planificare al proceselor poate fi simplificat, efectele datorate

competiției pentru alocarea resurselor, exceptând CPU(figura 9). În cadrul acestui model, procesul poate fi numai într-una din stările RUN sau READY.

Algoritmii de planificare se împart în două categorii: **cu evacuare** şi **fără evacuare.** Algoritmii de planificare fără evacuare presupun că odată ce unui proces i s-a alocat CPU, acesta va primi serviciile acesteia pâna la terminarea execuției sale. Algoritmii cu evacuare se bazează pe calculul priorității fiecărui proces. Unul dintre procesele cu prioritatea cea mai înaltă va fi servit de CPU. Când în lista READY este introdus un proces cu o prioritate mai mare decât cea a celui servit de către CPU, acesta va fi evacuat şi introdus în lista READY, execuția lui fiind reluată când prioritatea lui va fi cea mai înaltă, iar CPU va servi noul proces. Algoritmii cu evacuare sunt folosiți de către sistemele care partajează CPU în mod involuntar, pe când cei fără evacuare sunt utilizați în cazul eliberării voluntare a CPU.

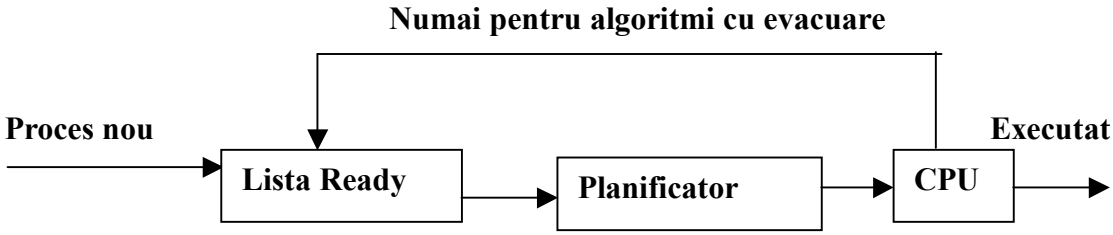


Figura 9 Un model simplificat de planificare a proceselor

Partiționarea unui proces în procese mai mici. Presupunem că un proces intercalează efectuarea de calcule, cu cereri de efectuare a unor operații de I/O, astfel încât el solicită CPU pentru k periode de timp distincte şi, deci va executa k cereri de efectuare a unor operații de I/O. Atunci, timpul total de servire cât îi este alocată CPU va fi

$$\tau(p_i) = \tau_1 + ... + \tau_k$$

De asemenea, dacă $d_1,..,d_k$ sunt timpii necesari efectuării operațiilor de I/O, atunci timpul cât procesul va sta în sistem va fi

$$\tau_1 + d_1 + ... + \tau_k + d_k$$

Presupunem că procesul p_i este descompus în k procese $p_{i1},...,p_{ik}$, în care τ_{ij} este timpul de servire pentru p_{ij}. De asemenea, fiecare proces p_{ij} este executat printr-o prelucrare care nu este întreruptă de alte procese. Reamintesc că orice proces poate fi orientat către efectuarea de calcule sau de operații de I/O. O astfel de descompunere urmăreşte ca procesele mai mici să fie orientate spre calcule sau spre operații de I/O, în scopul de a se evita cât mai mult operațiile costisitoare de comutare.

Cursul 3
Strategii de planificare a executiei proceselor
Fire de executie

Strategiile fără evacuare permit unui proces ca atunci când îi este alocată CPU să fie executat complet, deci nu mai există procesul de comutare şi nici cel de introducere şi extragere din lista READY. Aceste metode se bazează pe modelele de aşteptare.

Algoritmul FCFS(First Come First Served) sau **FIFO (First In First Output)**. Conform acestui algoritm, lucrările sunt servite în ordinea lor cronologică în care cer procesorul. Priorităţile job-urilor sunt calculate de administratorul cozii, pe baza valorii timpului când intră în lista READY. Dispecerul va selecta lucrarea pentru care timpul de aşteptare este cel mai lung. Lista READY este organizată pe principiul FIFO, administratorul cozii adăugând lucrări la sfârşitul cozii, iar dispecerul scoate lucrările pentru a fi servite de CPU din capul cozii. Este un algoritm simplu, dar nu foarte eficient.

Exemplul 1: Să presupunem că există 5 joburi în sistem, p_0, p_1, p_2, p_3, p_4 cu timpii de execuţie 350, 125, 475, 250 respectiv 75. Dacă acestea sosesc în ordinea prezentată, ordinea temporală a execuţiei lor, este reliefată prin diagrama Gantt din figura 1. Avem:

$$T(p_0) = \tau(p_0) = 350$$
$$T(p_1) = \tau(p_1) + T(p_0) = 125 + 350 = 475$$
$$T(p_2) = \tau(p_2) + T(p_1) = 475 + 475 = 950$$
$$T(p_3) = \tau(p_3) + T(p_2) = 250 + 950 = 1200$$
$$T(p_4) = \tau(p_4) + T(p_3) = 75 + 1200 = 1275$$

Media timpilor petrecuţi în sistem va fi deci

$$T = (350 + 475 + 950 + 1200)/5 = 850$$

De asemenea timpii de aşteptare vor fi: $W(p_0) = 0$, $W(p_1) = T(p_0) = 350$, $W(p_2) = T(p_1) = 475$, $W(p_3) = T(p_2) = 950$, $W(p_4) = T(p_3) = 1200$

Deci media timpului de aşteptare până la servire, va fi

$W = (0 + 350 + 475 + 950 + 1200)/5 = 595$.

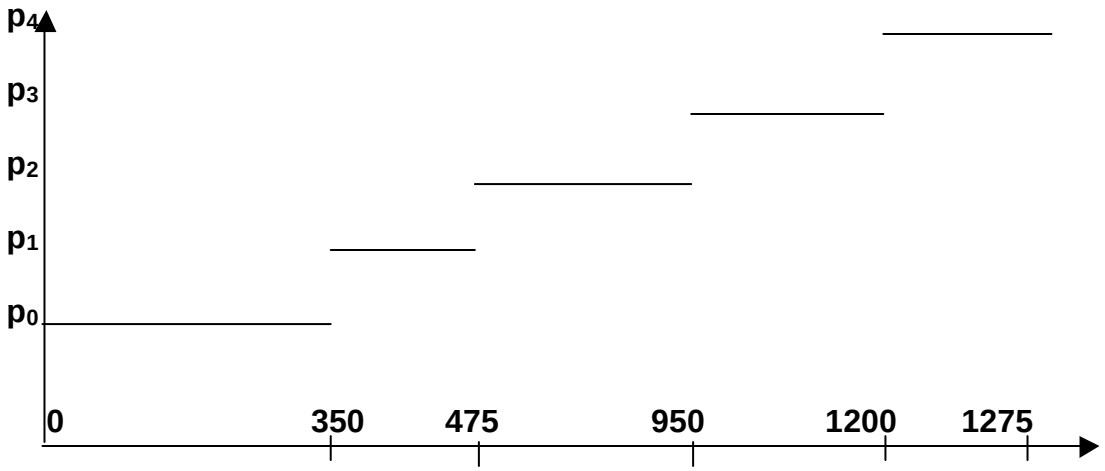


Figura 1. Ordinea execuției proceselor folosind strategia FCFS

Dacă ordinea de sosire este p_4, p_1, p_2, p_3, p_0 vom avea

$$T(p_4) = \tau(p_4) = 75$$
$$T(p_1) = \tau(p_1) + T(p_4) = 125 + 75 = 200$$
$$T(p_2) = \tau(p_2) + T(p_1) = 200 + 475 = 675$$
$$T(p_3) = \tau(p_3) + T(p_2) = 250 + 675 = 925$$
$$T(p_0) = \tau(p_0) + T(p_3) = 75 + 925 = 1000$$

Media timpilor petrecuți în sistem va fi deci

$T = (75 + 200 + 675 + 925 + 1000)/5 = 2875/5 = 575$

De asemenea timpii de aşteptare vor fi: $W(p_4) = 0$, $W(p_1) = T(p_4) = 75$, $W(p_2) = T(p_1) = 200$, $W(p_3) = T(p_2) = 675$, $W(p_1) = T(p_3) = 925$

Deci media timpului de aşteptare până la servire, va fi

$W = (0 + 75 + 200 + 675 + 925)/5 = 1875/5 = 355$.

Observație. Mediile timpilor petrecuți în sistem precum şi a timpilor de aşteptare sunt mai mici în cazul al doilea. Mai mult, aceste medii descresc, dacă ordinea de mărime a timpilor de servire respectă ordinea de sosire.

Algoritmul SJN (Shortest Job Next) se bazează pe observația anterioară. De fiecare dată, jobul care se execută primul este cel care consumă cel mai puțin timp CPU. Probabil că acest algoritm este cel mai bun, însă are dezavantajul că ar trebui să se cunoască dinainte timpul CPU necesar al fiecărui proces.

Exemplul 2. Considerând procesele prezentate în exemplul anterior, conform acestei metode ordinea de execuție este p_4, p_1, p_3, p_0, p_2 , reliefată în diagrama Gantt din figura 2 şi atunci:

$$T(p_4) = \tau(p_4) = 75$$
$$T(p_1) = \tau(p_1) + T(p_4) = 125 + 75 = 200$$
$$T(p_3) = \tau(p_3) + T(p_1) = 250 + 200 = 450$$
$$T(p_0) = \tau(p_0) + T(p_3) = 350 + 450 = 800$$
$$T(p_2) = \tau(p_2) + T(p_0) = 475 + 800 = 1275$$

Media timpilor petrecuți în sistem va fi deci

$$T = (75 + 200 + 450 + 800 + 1275)/5 = 2450/5 = 490$$

De asemenea timpii de aşteptare vor fi: $W(p_4) = 0$, $W(p_1) = T(p_4) = 75$, $W(p_3) = T(p_1) = 200$, $W(p_0) = T(p_3) = 450$, $W(p_2) = T(p_0) = 800$

Deci media timpului de aşteptare până la servire, va fi

$$W = (0 + 75 + 200 + 450 + 800)/5 = 1525/5 = 305.$$

Se observă că valorile calculate ale factorilor de eficiență sunt mult mai mici dacă se foloseşte această metodă față de cea anterioară.

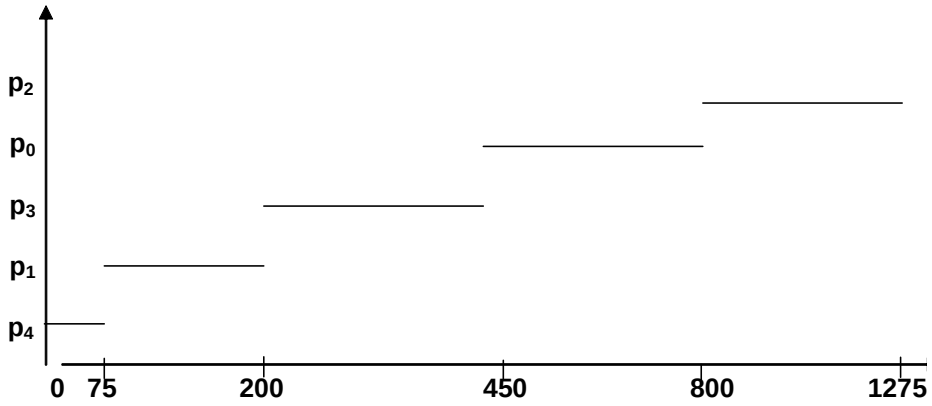


Figura 2. Ordinea temporală a execuției proceselor folosind strategia SJN

Algoritmul bazat pe priorități este cel mai des folosit. Toate celelalte strategii folosite, sunt cazuri particulare ale acestora. În planificarea bazată pe priorități, proceselor le este alocată CPU pe baza unor priorități alocate extern. Aceste priorități sunt numere naturale, un proces având prioritate față de altul dacă numărul alocat are o valoare mai mică. Joburile se ordonează după aceste priorități, apoi se execută în această ordine. Se disting două tipuri

de priorități: **interne** şi **externe**. Prioritățile interne sunt acordate pe baza unor particularități ale procesului, relative la mediului de calcul respectiv, cum ar fi de exemplu **timpul de execuție**. Prioritățile externe reflectă **importanța sarcinii** pe care procesul respectiv o execută, stabilită pe baza anumitor informații cum ar fi de, exemplu **numele utilizatorului**(cui aparține procesul respectiv), **natura sarcinii**(cât de importantă este problema pe care o rezolvă) etc.

Alegerea priorităților pentru procese este o problemă cheie, pentru ca această metodă de planificare să fie eficientă. Şi aici există pericolul apariției fenomenului de „înfometare", adică procesele care au o prioritate mai slabă să nu fie executate. Pentru a se evita acest fenomen, se poate considera drept criteriu de acordare a priorităților timpul de când jobul respectiv se află în sistem.

Exemplul 3. Presupunem că avem 5 procese p_0,p_1,p_2,p_3,p_4 ale căror priorități sunt 5, 2, 3, 1, 4 iar timpii de execuție sunt cei din exemplul anterior. Diagrama Gantt din figura 3 descrie ordinea execuției proceselor.

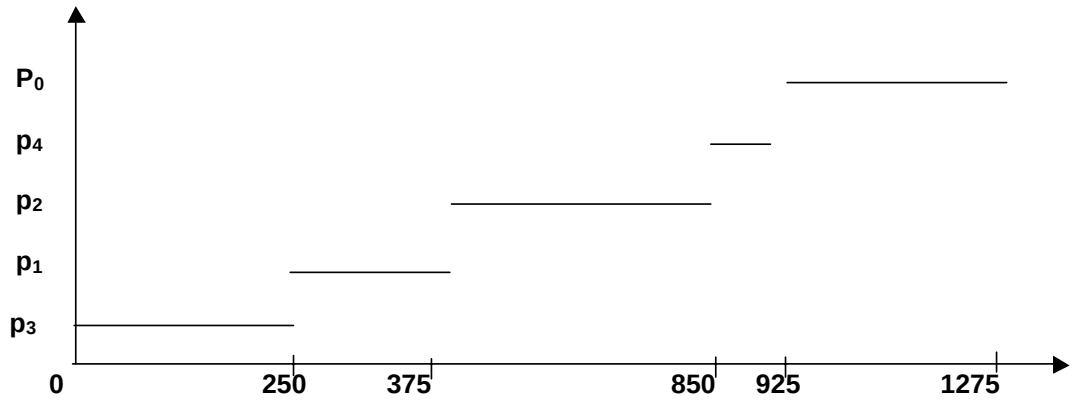


Figura 3. Ordinea temporală a execuției proceselor folosind strategia bazată pe priorități

Vom avea:

$$T(p_0) = \tau(p_0) + \tau(p_4) + \tau(p_2) + \tau(p_1) + \tau(p_3) = 350 + 75 + 475 + 125 + 250 = 1275$$

$$T(p_1) = \tau(p_1) + \tau(p_3) = 125 + 250 = 375$$

$T(p_2) = \tau(p_2) + \tau(p_1) + \tau(p_3) = 475 + 125 + 250 = 850$

$T(p_3) = \tau(p_3) = 250$

$T(p_4) = \tau(p_4) + \tau(p_2) + \tau(p_1) + \tau(p_3) = 75 + 475 + 125 + 250 = 925$

Deci media timpului de petrecut în sistem va fi $T = (1275 + 375 + 850 + 250 + 925)/5 = 735$

De asemenea, timpii de aşteptare se pot detemina într-un mod asemănător celui din exemplul anterior, aceştia fiind: $W(p_0)=925$, $W(p_1)=250$, $W(p_2)=375$, $W(p_3)=0$, $W(p_4)=850$. Deci media timpului de aşteptare va fi $W(p_0)=(925+250+375+0+850)/5=480$

Strategii cu evacuare. În cazul algoritmilor cu evacuare, CPU este alocată procesului cu prioritatea cea mai înaltă, dintre toate procesele care sunt în starea READY. Toate procesele cu prioritatea mai mică cedează CPU procesului cu prioritatea cea mai înaltă, atunci când acesta o cere. Oricând un proces intră în starea READY, el poate întrerupe imediat procesul servit de către CPU, dacă acesta are o prioritate mai slabă. Aceasta înseamnă că planificatorul este apelat de fiecare dată când un proces intră în starea READY. De asemenea, el declanşează măsurătorul de intervale de timp atunci când cuanta de timp a trecut.

Strategiile cu evacuare sunt adesea folosite pentru a se asigura un răspuns rapid proceselor cu o prioriate înaltă sau pentru a se asigura o bună partajare a CPU între toate procesele. Există versiuni cu evacuare pentru unele dintre strategiile fără evacuare prezentate anterior. În cazul strategiei SJN, procesului cu cererea de timp cea mai mică îi este alocat CPU. Dacă p_i este în curs de execuţie şi intră în starea READY un alt proces, p_j, atunci prin metoda SJN este necesară compararea lui $\tau(p_i)$ cu $\tau(p_j)$. Acest lucru este posibil deoarece se cunoşte că timpul de servire al lui p_i este cel mai mic, dintre timpii tuturor proceselor aflate în starea READY. Dacă p_j cere un timp de execuţie mai mic decât timpul rămas până la terminarea execuţiei lui p_i, atunci lui p_j i se va aloca CPU, iar p_i va fi introdus în lista READY, cu $\tau(p_i)$ fiind timpul rămas până la terminarea execuţiei lui. Într-un mod similar, se procedează şi în cazul strategiei bazată pe împărţire proceseloe gata de execuţie în clase de priorităţi.

În cazul algoritmilor fără evacuare, nu am considerat costul schimbării de context între procese, deoarece atunci am presupus că odată ce unui proces îi este alocată CPU, acesta va fi executat până la capăt. Aceată problemă capătă o importanță deosebită în cazul strategiilor cu evacuare, datorită faptului că pe durata execuției unui proces acesta poate fi evacuat de mai multe ori, iar fiecare evacuare este însoțită de o schimbare de context, al cărei cost nu poate fi neglijat.

Planificarea circulară(RR-**R**ound **R**obin**)** este poate cea mai utilizată dintre toți algoritmii de planificare, în special de SO care lucrează în time-sharing. Principala ei caracteristică, este servirea echitabilă a tuturor proceselor care cer alocarea CPU. Să presupunem că în sistem avem *n* procese p_0,..,p_{n-1}, ordonate după indicele lor. Se defineşte o cuantă de timp, dependentă de sistem şi pe durata unei cuante se alocă procesorul unui proces. Coada READY a proceselor este tratată circular; mai întâi este servit p_0, apoi p_1, ş.a.m.d p_{n-1}, după care este servit din nou p_0. Dacă procesorul termină de servit un proces înainte de expirarea cuantei de timp alocate, o nouă cuantă de timp este definită şi aceasta este alocată procesului următor. Când se termină execuția unui proces, el este scos din lista READY. Când un nou proces este introdus în lista READY, este inserat ultimul în această coadă circulară. De fiecare dată când apare un astfel de eveniment, planificatorul este apelat imediat, şi eventual se defineşte o nouă cuantă de timp şi se selectează un nou proces.

În cazul acestui algoritm, trebuie să luăm în considerație efectul schimbării de context între procese. Fie *C* timpul necesr efectuării acestei operații. Dacă fiecăruia dintre cele *n* procese îi este alocat *q* unități de timp CPU, atunci timpul total de servire al celor *n* procese va fi *n(q+C)*.

Exemplul 4. Să considerăm procesele p_0, p_1, p_2, p_3, p_4 care au timpii de execuție definiți în exemplele anterioare, iar cuanta de timp este de 50 de unități şi presupunem că se neglijează timpul necesar schimbării de context. În figura 4 este prezentată ordinea servirii proceselor şi momentele terminării execuției lor.

Din diagrama din figura 5 putem deduce:

- Timpii petrecuți în sistem de fiecare proces: $T(p_0)=1100$, $T(p_1)=550$, $T(p_2)=1275$, $T(p_3)=950$, $T(p_4)=475$ şi media timpului petrecut în sistem de un proces $T=(1100+550+1275+950+475)/5=870$.
- Timpii de aşteptare până la prima servire: $W(p_0)=0$, $W(p_1)=50$, $W(p_2)=100$, $W(p_3)=150$, $W(p_4)=200$ şi media acestor timpi $W=(0+50+100+150+200)/5=100$.
- Dacă şi timpul de comutare între procese, ca fiind de 10 unități de timp, obținem: $T(p_0)=1320$, $T(p_1)=660$, $T(p_2)=1535$, $T(p_3)=1140$, $T(p_4)=565$, $T=(1320+660+1535+1140+565)/5=1044$, $W(p_0)=0$, $W(p_1)=60$, $W(p_2)=120$, $W(p_3)=180$, $W(p_4)=220$ şi $W=(0+60+120+180+240)/5=120$.

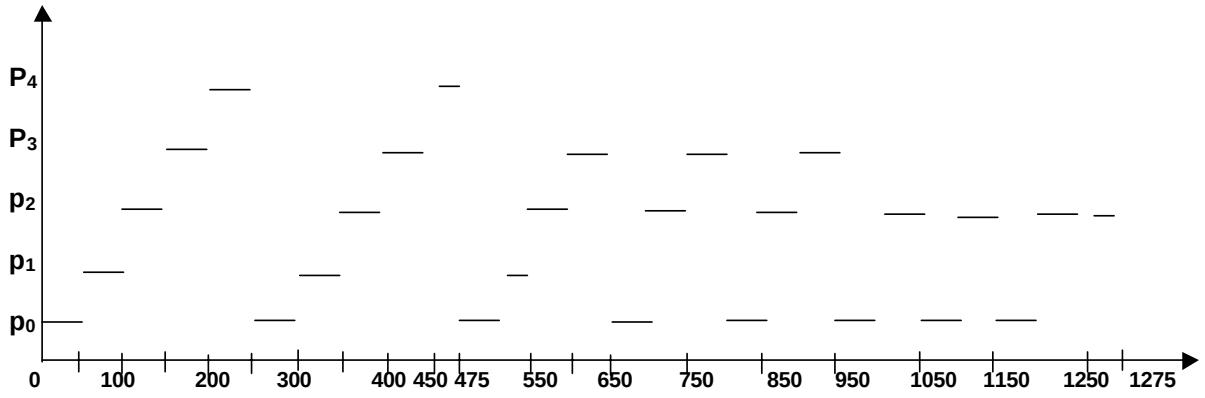


Figura 5 Planificare RR cu cuanta de timp de 50 de unități

Metoda cozilor pe mai multe niveluri este o extensie a planificării bazate pe priorități, care presupune că toate procesele care au aceeaşi prioritate sunt plasate în aceeaşi coadă. Între cozi, planificatorul alocă CPU folosind o anumită strategie, iar în interiorul cozii o altă strategie. De exemplu, la un SO de tip mixt, interactiv şi serial, pot fi create 5 cozi distincte: **taskuri sistem**, **lucrări interactive**, **lucrări în care se editează texte**, **lucrări seriale obişnuite**, **lucrări seriale ale studenților** etc.

Fire de execuție

În mod tradițional, fiecărei aplicații îi corespunde un proces. Pe calculatoarele moderne se execută multe aplicații în care, din punct de vedere logic, sunt structurate pe activități multiple, care s-ar putea desfăşura simultan. De **exemplu,** în cadrul unui „browser" de WWW, o activitate poate fi afişarea unor texte sau imagini, pe când alta poate fi regăsirea unor informații de pe anumite servere dintr-o rețea de calculatoare. Un procesor de texte poate executa simultan afişarea de imagini sau texte, citirea unor comenzi de la tastatură şi verificarea greşelilor gramaticale sau de sintaxă ale textului introdus.

Modelul proceselor, se bazează pe două considerente principale:

- procesul dispune de o colecție de resurse, între care un **spațiu de memorie**, necesare execuției sale;
- procesul are un **singur fir de control**, adică la un moment dat cel mult o singură instrucțiune a procesului poate fi în execuție pe un procesor.

Aşa cum am arătat în secțiunea anterioară, într-un sistem de operare multiproces, una dintre operațiile de bază ale SO, o constituie **comutarea proceselor,** conform unui algoritm de planificare. Comutarea proceselor implică salvarea de către nucleu a unor informații suficiente, care ulterior să fie restaurate, atunci când procesorul respectiv va primi din nou serviciile CPU, pentru ca execuția să fie reluată cu instrucțiunea următoare celei care a fost executată ultima. Aceste informații se referă la diverse categorii de resurse deținute de proces, ceea ce face ca operația de comutare a proceselor să fie relativ costisitoare din punctul de vedere al timpului procesor consumat pentru efectuarea ei. De asemenea, crearea unui nou proces în sistem este o operație de o complexitate deosebită.

Atât cerințele unor noi domenii de aplicații, cât şi dezvoltările tehnologice şi arhitecturale ale sistemelor de calcul(în special sistemele multiprocesor), au impus renunțarea la condiția de unicitate a firului de control în cadrul unui proces. A apărut astfel

conceptul de **fir de execuție** (**thread**); mai multe asemenea fire pot coexista într-un proces, utilizând în comun resursele procesului. Avantajele principale fiind:

• **eliminarea costului** creării de noi procese;

• **simplificarea comutării**, pentru că numărul de informații salvate/restaurate va fi mai redus;

• posibilitatea ca o **aplicație interactivă să-şi continue execuția,** chiar dacă o parte a ei este blocată sau execută o operație mai lungă în timp; de exemplu, un browser de WWW menține interacțiunea cu utilizatorul, chiar dacă a început încărcarea unei imagini;

• **utilizarea arhitecturilor multiprocesor**, care permit ca fiecărui fir de control să-i fie alocat un procesor.

Firele de execuție pot coexista într-un proces(în acelaşi spațiu de memorie), evident cu necesitatea de a fi sincronizate atunci când solicită acces la resurse partajate, ca şi în cazul proceselor. Resursele proprii fiecărui fir de execuție sunt, în general limitate la: stivă, variabile locale şi numărător(contor) de program. Variabilele globale ale unui program vor fi implicit accesibile tuturor firelor din acel program.

Firele de execuție se împart în;

• **fire ale utilizatorului,** implementate de către o bibliotecă ce poate fi accesată de către utilizator şi care oferă posibilități de creare, planificare şi administrare a **thred**-urilor, fără a fi necesar suportul nucleului; acestora li se asigură spațiu de memorie în vederea execuției lor în spațiul utilizator;

• **fire ale nucleului SO,** implementate de către sistemul de operare, care realizează crearea, planificarea şi administrarea lor în spațiul nucleului, ele realizând anumite operații protejate cerute de către procese.

Utilizarea firelor de execuție se poate realiza folosind mai multe modele:

• **mai multe fire utilizator care corespund la un fir al nucleului,** este caracterizat de eficiență, deoarece administrarea lor este realizată în spațiul utilizatorului, dar în cazul în care unul dintre

firele utilizator generează un apele de sistem, prin care intră în starea de blocare, întregul proces va fi blocat; de asemenea, deoarece numai un fir de execuție poate, la un moment dat să acceseze nucleul, nu este posibil ca mai multe fire de execuție să se execute în paralel, pe un sistem multiprocesor.

• **un fir utilizator care corespunde la un fir al nucleului** permite concurența execuției, deoarece atunci când un fir de execuție a inițiat un apel de sistem care generează blocarea, celelalte fire de execuție se pot executa în continuare; dezavantajul constă în numărul mare de fire nucleu care trebuie create.

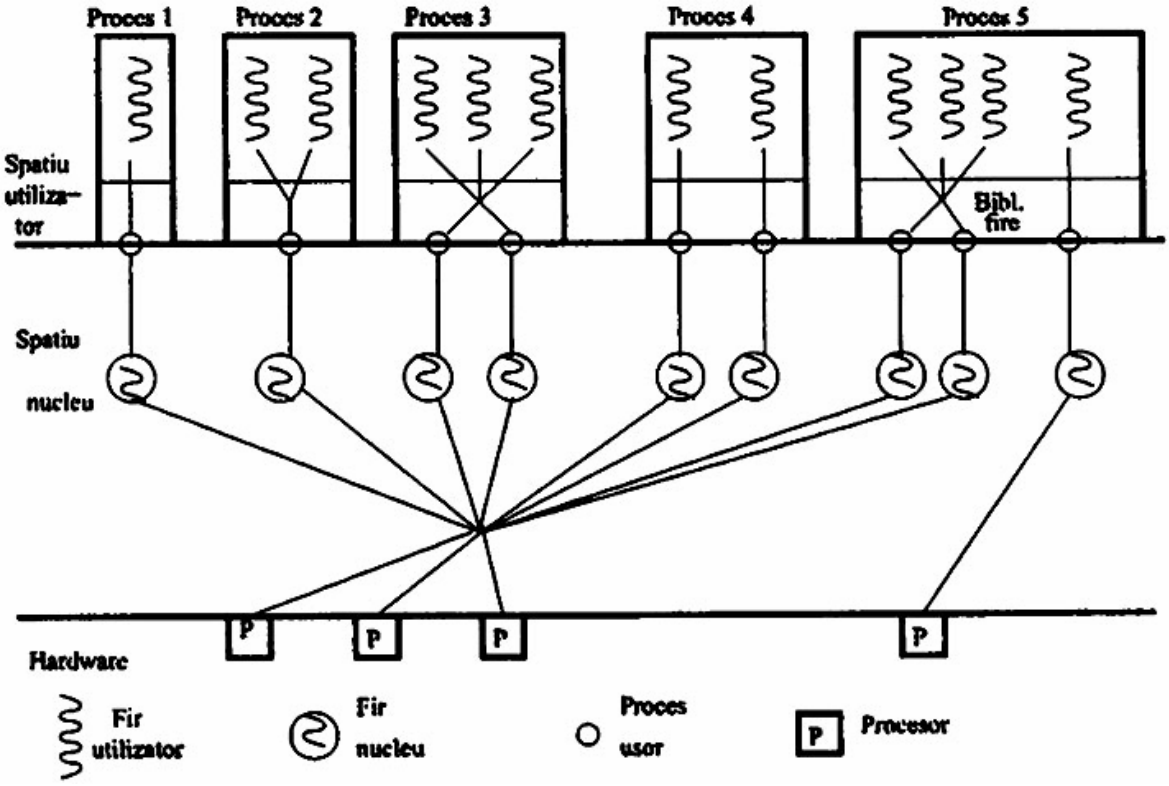
• **mai multe fire utilizator care corespund la mai multe fire ale nucleului(**în număr mai mic sau egal decât cele ale utilizatorului) este o situație specifică, sistemelor multiprocesor, deoarece este posibil, ca la un moment dat, mai multe fire ale unei aplicații, care sunt servite de procesoare diferite, să lanseze un acelaşi apel de sistem, care să fie rezolvat de fire nucleu diferite.

Există mai multe variante conceptuale şi de implementare, după cum firele de execuție se manifestă numai în spațiul utilizator sau au reprezentare şi în nucleu. Modelul cel mai complet este cel oferit în sistemul de operare Solaris, prezentat în figura 2.14. **Procesele uşoare** (**lightweight process,** prescurtat LWP) pot fi considerate ca o corespondență între firele la nivelul utilizatorului şi firele din nucleu. Fiecare proces uşor este legat, pe de o parte de una sau mai multe fire utilizator, iar pe de altă parte, de un fir nucleu. Planificarea pentru execuție a LWP se face independent, astfel că LWP separate pot fi executate simultan în sisteme multiprocesor.

Figura 6 ilustrează diversele relații care se pot stabili între cele 4 tipuri de entități. Se observă, în primul rând că un LWP este vizibil şi în spațiul utilizator, ceea ce înseamnă că vor exista structuri de date corespunzătoare LWP atât în cadrul unui proces, cât şi în cadrul nucleului.

Procesul 1 din figura 6 este un proces "clasic": el are un singur fir de execuție la nivelul utilizatorului, legat cu un LWP. **Procesul 2** este o ilustrare a situației în care firele de execuție multiple sunt

vizibile numai în spațiul utilizator: toate sunt puse în corespondență cu un singur LWP, deci la un moment dat numai unul dintre firele procesului 2 poate fi în execuție, pentru că planificarea se face la nivelul LWP. **Procesul 3** are mai multe fire de execuție, multiplexate pe un număr mai mic de LWP. Aplicațiile de acest gen conțin paralelism, al cărui grad poate fi controlat prin numărul de LWP puse la dispoziție.



Starea de interblocare

Evitarea Interblocarii

Detectarea interblocarii

Metode mixte

Conceptul de interblocare

- Resursele logice (fişiere, baze de date, semafoare etc.) sau fizice (imprimante, spațiul de memorie internă, memorii externe, cicluri UC etc.) pot fi formate din unul sau mai multe elemente. Etapele parcurse de un proces în cursul utilizării unei resurse sunt:
- • **cerere de acces**: dacă cererea nu poate fi satisfăcută imediat, procesul care a formulat-o va fi nevoit să aştepte până când poate dobândi resursa;
- • **utilizare**: procesul poate folosi resursa;
- • **eliberare**: procesul eliberează resursa.
- Cererea şi eliberarea de resurse se face prin **apeluri de sistem**. Evidența alocării resurselor se realizează prin intermediul unei **tabele de sistem.** Dacă un proces cere o resursă ale cărei elemente sunt alocate altor procese din sistem, atunci procesul care a efectuat cererea va fi pus într-o coadă de aşteptare, asociată resursei respective.
- Se spune că un set de procese se află în stare de **interblocare** atunci când orice proces din setul respectiv se află în aşteptarea unui eveniment de eliberare a unei resurse cerute, ce poate fi produs numai de către un proces aflat în mulțimea respectivă. Pentru rezolvarea problemei interblocării se folosesc, în principiu două metode. Prima constă în utilizarea unui protocol care să nu permită niciodată sistemului să intre în starea de interblocare. Acest lucru se realizează fie prin **prevenirea**, fie prin **detectarea** interblocării. Cea de a doua metodă permite sistemului intrarea în starea de interblocare şi apoi rezolvă această problemă.

Condiții necesare pentru apariția interblocării

- **excludere mutuală**: există cel puțin o resursă ocupată în mod exclusiv, adică fără a putea fi folosită în comun de către un singur proces; dacă un alt proces formulează o cerere pentru aceeaşi resursă, va fi nevoit să aştepte până în momentul eliberării ei;
- • **ocupare şi aşteptare**: există cel puțin un proces care ține ocupată cel puțin o resursă şi aşteaptă să obțină resurse suplimentare ocupate în acel moment de către alte procese;
- • **imposibilitatea achiziționării forțate**: resursele nu pot fi achiziționate forțat de către un proces de la un alt proces care le ocupă în acel moment; resursele pot fi eliberate numai de către procesele care le ocupă, decât după ce acestea şi-au îndeplinit sarcinile;
- • **aşteptare circulară**: în sistem există un set de procese aflate în starea de aşteptare, (p1,p2,..,pn), astfel încât p1 aşteaptă eliberarea unei resurse ocupate de către p2, p2 aşteaptă eliberarea unei resurse ocupate de către p3, ş.a.m.d. pn-1 aşteaptă eliberarea unei resurse ocupate de către pn, pn aşteaptă eliberarea unei resurse ocupate de către p1.
- Se observă că ultima condiție implică şi cerința de ocupare şi aşteptare, astfel încât cele patru condiții nu sunt complet independente; cu toate acestea este util ca fiecare condiție să fie discutată şi tratată separat.

Evitarea interblocării

- **Starea alocării resurselor** este definită de numărul de resurse disponibile şi alocate şi de numărul maxim de cereri de resurse formulate de către procese. Se spune că o stare este **sigură** dacă sistemul poate aloca fiecărui proces resursele cerute (până la numărul maxim), într-o anumită ordine şi evitând apariția interblocării. Mai exact, sistemul se află într-o stare sigură numai dacă există o **secvență sigură**. Se spune că o secvență de procese (p1,p2,..,pn) este o secvență sigură pentru starea de alocare curentă dacă, pentru fiecare pi, resursele pe care acesta le-ar mai putea cere pot fi alocate dintre resursele disponibile, la care se mai adaugă resursele deținute de către toate celelalte procese pj, cu j<i. În acest caz, dacă resursele cerute de către procesul pi nu sunt disponibile imediat, acesta va trebui să aştepte până când toate procesele pj, cu j<i îşi încheie execuția. În acest moment pi poate obține toate resursele de care are nevoie, îşi termină sarcina, eliberează resuirsele şi se încheie, după care procesul pi+1 poate obține resursele pe care le doreşte ş.a.m.d. În cazul în care nu există o astfel de secvență, se spune că starea sistemului este nesigură. O stare de interblocare este o stare nesigură; nu toate stările nesigure sunt interblocări, dar o stare nesigură poate conduce la interblocare. Într-o stare nesigură, sistemul de operare nu poate împiedica procesele să formuleze în aşa fel cererile de alocare a resurselor încât acestea să ducă la interblocare.

Exemplu

- Să considerăm un sistem în care 4 procese p_1, p_2, p_3, p_4 folosesc în comun o resursă cu 20 de elemente. Pe durata întregii execuții procesele au nevoie de maximum 14, 6, 10 şi respectiv 9 elemente. Inițial, procesele formulează o cerere pentru 7, 3, 4 şi respectiv 2 elemente. Analizând situația sistemului la momentul inițial se constată că el se află într-o stare sigură, deoarece, de exemplu, secvența p_2, p_1, p_3, p_4 este o
- secvență sigură. Dacă procesul p4 cere încă 2 elemente şi acestea îi sunt alocate, se trece nu numai într-o stare nesigură dar şi într-o stare de interblocare, deoarece numărul de resurse libere, egal cu 2, este insuficient pentru oricare dintre procesele aflate în aşteptare. Algoritmul de evitare a interblocării pentru cazul resurselor **cu mai multe elemente**, care va fi descris în continuare, se numeşte **algoritmul bancherului**.

Algoritmul bancherului

- Pentru implementarea algoritmului sunt necesare câteva structuri de date care să codifice starea de alocare a resurselor sistemului. Dacă n este numărul de procese din sistem şi m este numărul de resurse, se definesc:
- • D(„Disponibil”): vector de dimensiune m care indică numărul de elemente disponibile ale resurselor: D(j) conține numărul elementelor disponibile ale resursei rj, j=1,..,m;
- • M(„Maxim”): matrice de dimensiune nxm care indică numărul maxim de cereri care pot fi formulate de către fiecare proces; M(i,j) conține numărul maxim de elemente ale resursei rj cerute de procesul pi;
- • A(„Alocare”): matrice de dimensiune nxm care indică numărul de elemente din fiecare resursă care sunt alocate în mod curent fiecărui proces; A(i,j) conține numărul de elemente ale resursei rj alocate procesului pi;
- • N(„Necesar”): matrice de dimensiune nxm care indică numărul de elemente ce ar mai putea fi necesare fiecărui proces; N(i,j) conține numărul de elemente ale resursei rj de care ar mai avea nevoie procesul pi pentru a-şi realiza sarcina; evident că N(i,j)= M(i,j)-A(i,j).

- Fie C_j vectorul cererilor formulate de către procesul p_i; $C_i(j)$ este numărul cererilor formulate de procesul p_i din resursa r_j; în momentul în care procesul p_i formulează o cerere de resurse, vor fi parcurse următoarele etape:
- **Pas 1.** Dacă $C_i \leq N_i$, se execută pasul 2, altfel, se consideră că a apărut o eroare, deoarece procesul a depăşit cererea maxim admisibilă.
- **Pas 2.** Dacă Ci≤D, se execută pasul 3, altfel, pi este nevoit să aştepte (resursele nu sunt disponibile).
- **Pas 3.** Se simulează alocarea resurselor cerute de procesul pi şi starea se modifică astfel: D:=D- Ci; Ai:= Ai+ Ci; Ni:= Ni- Ci.
- Pentru verificarea **stării de siguranță** a sistemului se foloseşte algoritmul următor. Algoritmul constă în parcurgerea secvenței de procese, simularea terminării execuției unui proces, adică adăugarea la disponibilul de resurse existent înaintea terminării procesului de la pasul respectiv a resurselor procesului care tocmai şi-a terminat execuția.
- Algoritmul foloseşte:
- vectorul de lucru L, de dimensiune m ale cărui componente vor corespunde resurselor sistemului, componenta L(i) va conține numărul resurselor ri disponibile la un moment dat;
- un vector T de dimensiune n, care corespunde proceselor şi care va marca procesele parcurse;
- un vector s de dimensiune n, componenta s(i) va conține indicele (poziția) din secvență a procesului.

Pas 1(Inițializări)**.** L:=D; pentru i=1,..,n; T(i):=0; citeşte s.

Pas 2. (Parcurgerea secvenței de procese**)**

K:=0;

do { k:=k+1;

if (N(s(k)) <=L) then

{ L:=L+A(s(k));

T(**s(k)**):=1

}

until k=n or not T(**s(k)**)

}

Pas 3. (Verificare dacă toate procesele din secvență au fost marcate cu true). Dacă T(i):=1 pentru i=1,..,n, atunci sistemul se află într-o stare sigură.

Dacă starea de alocare a resurselor rezultată este sigură, se alocă procesului pi resursele cerute. În caz contrar, procesul pi este nevoit să aştepte, iar sistemul reface starea de alocare a resurselor existentă înainte de execuția pasului 3.

Exemplu

- Să presupunem că în sistem avem 5 procese p_1, p_2, p_3, p_4, p_5 şi patru resurse r_1,r_2,r_3,r_4, care au 3, 14, 12 respectiv 13 elemente. Se consideră că starea inițială este definită de:

$$A = \begin{pmatrix} 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 0 \\ 1 & 3 & 5 & 4 \\ 0 & 6 & 3 & 2 \\ 0 & 0 & 1 & 4 \end{pmatrix} \qquad M = \begin{pmatrix} 0 & 0 & 1 & 2 \\ 1 & 7 & 5 & 0 \\ 2 & 3 & 5 & 6 \\ 0 & 6 & 5 & 2 \\ 0 & 6 & 5 & 6 \end{pmatrix}$$

$$N = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 6 & 4 & 2 \end{pmatrix} \qquad D = \begin{pmatrix} 1 & 5 & 2 & 0 \end{pmatrix}$$

- **Iterația 1**(procesul p_1): . $(0\ \ 0\ \ 0\ \ 0) \le (1\ \ 5\ \ 2\ \ 0)\ T = (1\ \ 0\ \ 0\ \ 0\ \ 0)$

$$L = (0\ \ 0\ \ 1\ \ 2) + (1\ \ 5\ \ 2\ \ 0) = (1\ \ 5\ \ 3\ \ 2)$$

- **Iterația 2**(procesul p3): $(1\ \ 0\ \ 0\ \ 2) \le (1\ \ 5\ \ 3\ \ 2)\ \ T = (1\ \ 0\ \ 1\ \ 0\ \ 0)$

$$L = (1\ \ 3\ \ 5\ \ 4) + (1\ \ 5\ \ 3\ \ 2) = (2\ \ 8\ \ 8\ \ 6)$$

- **Iterația 3**(procesul p4): $(0\ \ 0\ \ 2\ \ 0) \le (2\ \ 8\ \ 8\ \ 6)\ T = (1\ \ 0\ \ 1\ \ 1\ \ 0)$

$$L = (0\ \ 6\ \ 3\ \ 2) + (2\ \ 8\ \ 8\ \ 6) = (2\ \ 14\ \ 11\ \ 8)$$

- **Iterația 4**(procesul p5): $(0\ \ 6\ \ 4\ \ 2) \le (2\ \ 14\ \ 11\ \ 8)\ \ T = (1\ \ 0\ \ 1\ \ 1\ \ 1)$

$$L = (0\ \ 6\ \ 3\ \ 2) + (2\ \ 8\ \ 8\ \ 6) = (2\ \ 14\ \ 11\ \ 8)$$

- **Iterația 5**(procesul p2): $(0\ \ 7\ \ 5\ \ 0) \le (2\ \ 14\ \ 11\ \ 8)\ \ T = (1\ \ 1\ \ 1\ \ 1\ \ 1)$

$$L = (1\ \ 0\ \ 0\ \ 0) + (2\ \ 14\ \ 11\ \ 8) = (3\ \ 14\ \ 11\ \ 8)$$

- **Pasul 3** , $T = (1\ \ 1\ \ 1\ \ 1\ \ 1)$ deci sistemul se află într-o stare sigură

- În cazul în care procesul p1 formulează o cerere suplimentară pentru 4, repectiv 2 elemente din r2 , respectiv *r3*, trebuie să se verifice dacă această cerere poate fi satisfăcută. Conform algoritmului bancherului, se verifică îndeplinirea relațiilor:

C1≤N1, adică (0,4,2,0) ≤ (0,7,5,0)

C1≤D, adică (0,4,2,0) ≤ (1,5,2,0)

- Cum inegalitățile sunt îndeplinite, se simulează alocarea. Starea sistemului devine:

$$A=\begin{pmatrix}0&0&1&2\\1&4&2&0\\1&3&5&4\\0&6&3&2\\0&0&1&4\end{pmatrix} \quad N=\begin{pmatrix}0&0&0&0\\0&3&3&0\\1&0&0&2\\0&0&2&0\\0&6&4&2\end{pmatrix} \quad D=\begin{pmatrix}1&1&0&0\end{pmatrix}$$

- Temă. Să se arate că această stare a sistemului este sigură.

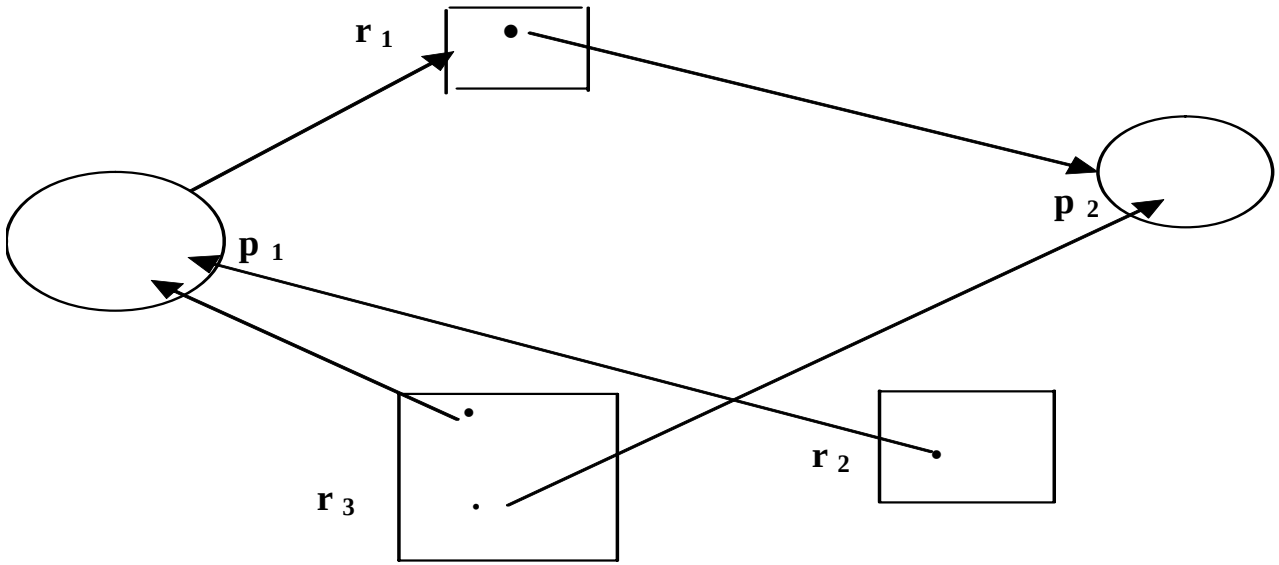
- Indicație. Se consideră secvența (p_1, p_3, p_4, p_2, p_5).

Graful de alocare a resurselor

- Algoritmul bancherului are o complexitate de ordinul m x n^2; pentru evitarea interblocării, în cazul resurselor cu un singur element se utilizează un algoritm de complexitate mult mai mică, care va fi descris în continuare.
- **Graful de alocare a resurselor** este de forma G=(N,A), în care N=PȕR, P={p_1,..,p_n} fiind mulțimea proceselor iar R={r_1,..,r_m} mulțimea resurselor. Un arc poate fi de forma (p_i,r_j) sau (r_j,p_i); arcul (p_i,r_j), numit **arc cerere,** are semnificația că procesul p_i a cerut un element al resursei r_j, iar arcul (r_j ,p_i), numit **arc alocare,** înseamnă că un element al resursei r_j a fost alocat procesului p_i. În cadrul grafului, procesele sunt reprezentate grafic prin cercuri, iar resursele prin pătrate sau dreptunghiuri. Deoarece resursele pot fi formate din mai multe elemente, fiecare element se reprezintă cu ajutorul unui punct plasat în interiorul pătratului sau dreptunghiului respectiv. Atunci când procesul pi formulează o cerere pentru un element al resursei rj, se inserează în graful alocare a resurselor un arc cerere. În momentul în care cererea este satisfăcută, arcul cerere este transformat în arc alocare. Atunci când procesul eliberează resursa, arcul alocare este şters din graf.

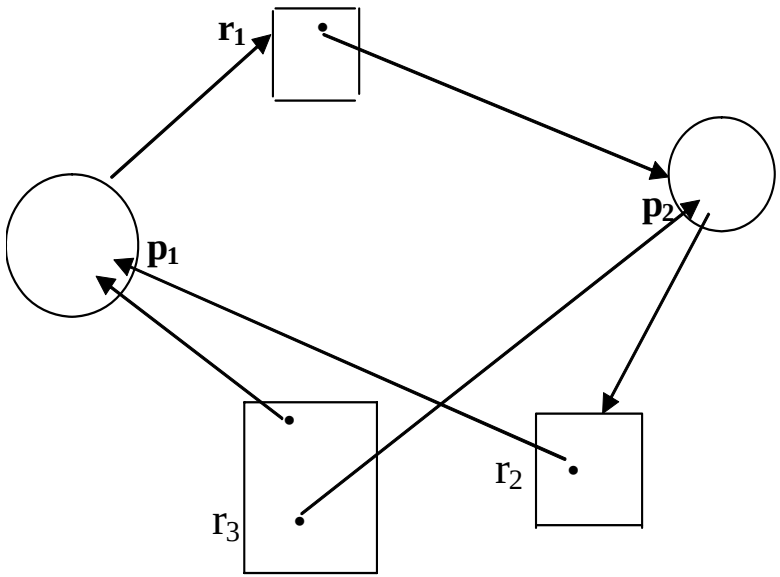
Exemplu

- Considerăm un sistem cu două resurse r_1 şi r_2 care au câte un element, şi o resursă r_3 cu două elemente în care există două procese, p_1 şi p_2. Avem deci: $P=\{p_1,p_2\}$, $R=\{r_1,r_2,r_3\}$. Dacă $A=\{(p_1,r_1), (r_1,p_2), (r_2,p_1), (r_3,p_1), (r_3,p_2)\}$, reprezentarea grafică a grafului este cea din figura urmatoare

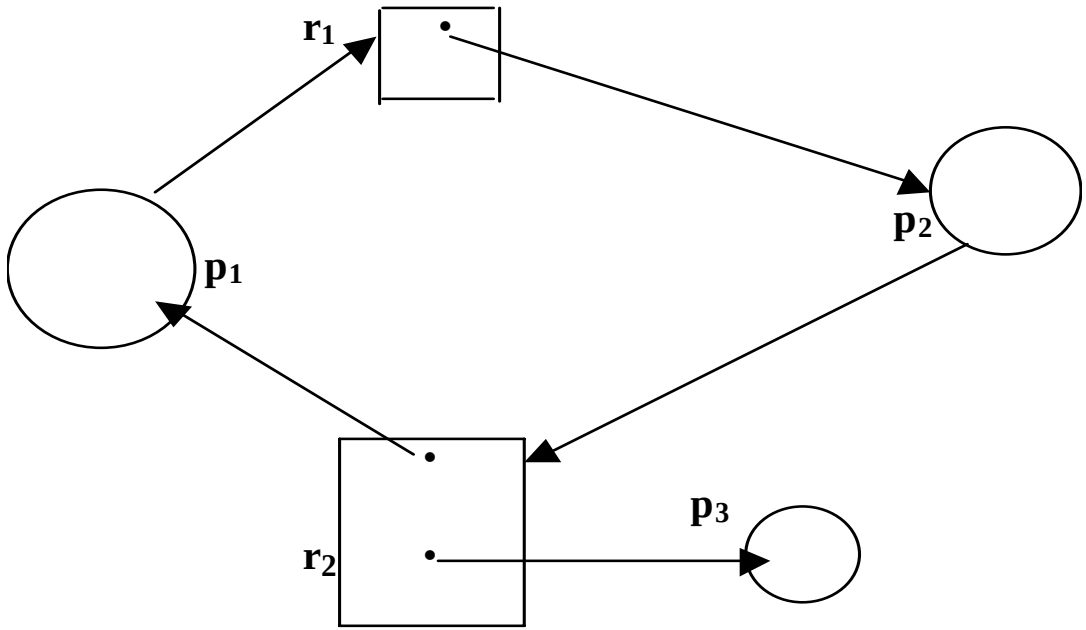


- Mulțimea de arce ilustrează starea proceselor:
- procesul p_1 are alocată câte un element al resurselor r_2 şi r_3 şi aşteaptă obținerea unui resursei r_1;
- procesul p_2 are alocată resursa r_1 şi un element al resursei r_3.

- Cunoscând definiția grafului de alocare a resurselor, se poate arăta că dacă graful nu conține circuite, în sistem nu există interblocare. Dacă există un circuit, atunci poate să apară interblocarea. Dacă există resurse cu mai multe elemente, atunci existența unui circuit nu implică în mod necesar apariția interblocării, aceasta fiind doar o condiție necesară dar nu şi suficientă. Pentru a exemplifica această afirmație, se poate folosi din nou situația prezentată anterior, dar se presupune în plus că că procesul p_2 formulează o cerere suplimentară(figura urmatoare). Deoarece resursa de tip r_2 cerută de procesul p_2 nu este disponibilă, fiind deja alocată procesului p_1, în graf se inserează arcul cerere (p_2,r_2), formându-se circuitul (p_1, r_1, p_2, r_2, p_1). În acest moment, cele două procese intră în starea de interblocare.



- În figura următoare, este prezentată o situație în care, deşi există un circuit (p_1, r_1, p_2, r_2, p_1), nu există interblocare. După ce p_3 îşi termină execuția, p_2 poate primi r_2 ,şi după terminarea execuției lui cedează r_1 care poate fi folosită de către p_1 şi care poate astfel să-şi termine execuția.
-



Detectarea interblocării

- Atunci când sistemul nu utililizează algoritmi de evitare a interblocării, deorece utilizarea lor ar mări costurile, se pot utiliza metodele de detectare şi revenire, care însă, pe lângă cheltuielile datorate utilizării unor structuri de date suplimentare, presupun şi cheltuieli datorate revenirii din starea de interblocare.
- Problema care se pune, este când să fie apelat algoritmul de detectare a interblocării. Dacă acest lucru se face ori de câte ori se formulează o cerere de resurse, efectul este creşterea substanțială a timpului de calcul.
- O variantă mai puțin costisitoare este apelarea algoritmului la anumite **intervale de timp** sau atunci când **gradul de utilizare a CPU** scade sub un anumit prag (de exemplu 40 la sută), justificarea fiind că o interblocare poate conduce la „paralizarea” funcționării sistemului şi deci la o scădere a gradului de utilizare a CPU.

Algoritmul de detectare pentru resurse cu mai multe elemente

- Algoritmul foloseşte structurile de date *D, A* şi *C* cu aceeşi semnificație ca şi în cazul algoritmului de evitare a interblocării.
- Algoritmul prezentat ia în considerație toate procesele a căror execuție nu s-a încheiat încă şi analizează toate secvențele posibile de alocare a resurselor pentru aceste procese.
- **Pas 1.** Fie L şi T vectori de dimensiune m, respectiv n. Pentru i=1,..,n, L(i):=D(i). Pentru i=1,..,n, dacă $A_i \neq 0$, atunci T(i):=false, altfel, T(i):=true.
- **Pas 2.** Se caută i astfel încât: T(i)=false şi $C_i \leq L$. Dacă nu există, goto pas 4.
- **Pas 3.** Se execută secvența: L:=L+Ai; T(i):=true; goto pas 2.
- **Pas 4.** Dacă există i, i=1,..,n, T(i)=false, sistemul se află în starea de interblocare. În plus procesul pentru care T(i)=false, provoacă interblocarea.
- La pasul 3 sunt preluate resursele deținute de procesul p_i, în cazul în care la pasul 2 se îndeplineşte condiția $C_i \leq L$, este că p_i nu este implicat în mod curent într-o interblocare; deci, se presupune că p_i nu va mai cere alte resurse pentru a-şi termina execuția şi va elibera toate resursele care i-au fost alocate. Altfel, la următoarea utilizare a algoritmului se poate sa fie depistatăo interblocare.

Exemplu

- Presupunem că avem trei procese p_1, p_2, p_3, care folosesc în comun trei resurse r_1, r_2, r_3, având câte 4, 2 şi respectiv 2 elemente.
- De asemenea, valorile structurilor de date, care dau starea inițială sunt:

$$A = \begin{pmatrix} 2 & 2 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \qquad C = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 2 & 1 & 0 \end{pmatrix} \qquad D = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$$

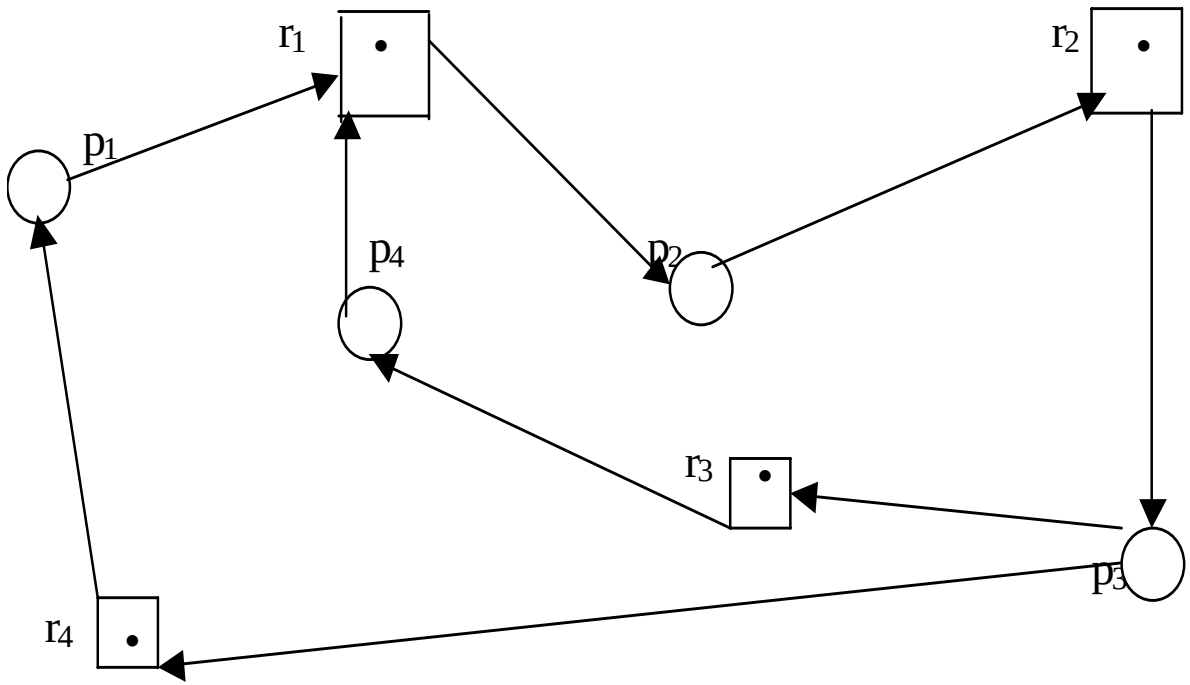
- Se poate verifica faptul că nu există interblocare;aplicând algoritmul se constată că secvența (p_2, p_1, p_3) asigură îndeplinirea condiției T(i)=true pentru i=1,2,3.
- Dacă însă procesul p_1 formulează o cerere suplimentară pentru 2 elemente ale lui r_1, modificând prima linie a matricii C , care va conține acum valorile 2,0,1, sistemul intră în starea de interblocare. Chiar dacă procesul p_1 îşi încheie execuția şi eliberează resursele care i-au fost alocate (1,0,1), numărul total de resurse disponibile nu este suficient pentru a putea acoperi necesarul formulat de oricare dintre celelalte două procese. Prin urmare, procesele p_1 şi p_3 sunt interblocate

Algoritmul de detectare a interblocării pentru resurse cu un singur element

- Algoritmul de detectare a interblocării pentru resurse cu mai multe elemente necesită un număr de operații de ordinul mxn^2.
- Dacă toate resursele au câte un singur element, se poate defini un algoritm care utilizează o variantă a grafului de alocare a resurselor, numit **graful ”aşteaptă-pentru”.**
- Acesta se obține din graful de alocare a resurselor prin eliminarea nodurilor resursă şi contopirea arcelor corespunzătoare: un arc (p_i,p_j) indică faptul că pi aşteaptă ca p_j să elibereze resursa care îi este necesară. Într-un graf ”aşteaptă-pentru”, un arc (p_i,p_j) există dacă şi numai dacă, pentru o resursă r_k oarecare, există două arce în graful alocării resurselor asociat (p_i,r_k) şi (r_k,p_j).

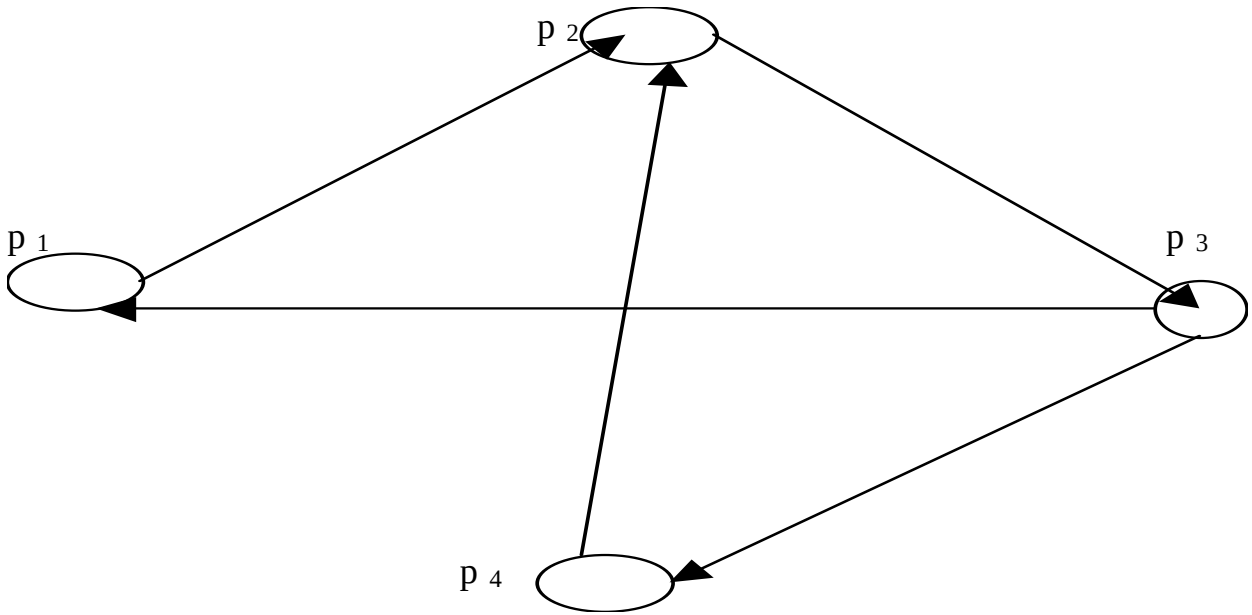
Exemplu

- În figura urmatoare este prezentat un graf de alocare a resurselor



Graful ”aşteaptă-pentru” asociat.

- Folosind această metodă, se poate afirma că în sistem există interblocare dacă şi numai dacă graful „aşteaptă-pentru” conține un circuit. Pentru a detecta interblocările, sistemul trebuie să actualizeze graful „aşteaptă-pentru” şi să apeleze periodic un algoritm care să testeze dacă există circuite în acest graf. Acest algoritm necesită un număr de n^2 operații, deci complexitatea este mai mică decât a algoritmului general. Totuşi, există un dezavantaj, şi anume operațiile suplimentare necesare reprezentării şi actualizării grafului „aşteaptă-pentru”.



Achiziționarea forțată a resurselor

- Această metodă asigură eliminarea stării de interblocare prin achiziționarea succesivă a resurselor utilizate de anumite procese şi alocarea lor altor procese, până când se elimină interblocarea.
- Principalele probleme care trebuie rezolvate în acest caz sunt:

- alegerea proceselor cărora li se vor lua resursele, după criterii care să conducă la costuri minime(numărul resurselor ocupate, timpul de execuție consumat);

- reluarea execuției: un proces căruia i s-au luat forțat anumite resurse nu-şi mai poate continua normal execuția, ea trebuie reluată dintr-un moment anterior, când a primit prima dintre resursele luate;

- evitarea „înfometării” , adică să nu fie selectat pentru achiziționare forțată a resurselor un acelaşi proces.

Metode mixte de tratare a interblcărilor

- Practica a dovedit că nici una dintre metodele de bază prezentate anterior, nu poate acoperi toate cauzele care produc interblocarea. O modalitate de rezolvare a acestei probleme, este combinarea algoritmilor de bază prezentați. Metoda propusă are la bază ideea că resursele pot fi grupate în clase şi în fiecare clasă se utilizează metoda de gestionare a resurselor cea mai adecvată. Datorită grupării resurselor în clase, o interblocare nu poate implica mai mult decât o clasă de resurse şi, cum în interiorul unei clase se aplică una dintre metodele de bază, în sistem nu pot să apară interblocări.
- Ca **exemplu**, se poate considera un sistem format din patru clase de resurse:

- resurse interne, adică resursele utilizate de către sistem(de exemplu, PCB);

- memoria centrală;

- resursele job-ului: dispozitive (discurile) şi fişiere;

- spațiul din memoria auxiliară alocat fiecărui job utilizator.

- O metodă mixtă pentru rezolvarea interblocării în cazul acestui sistem, ordonează clasele descrise anterior, folosind în cadrul fiecărei clase următoarele abordări:

- prevenirea interblocării prin achiziționare forțată a memoriei centrale(se poate evacua oricând un job în memoria auxiliară);

- evitarea interblocării în cazul resurselor job-ului(informațiile necesare despre formularea cererilor de resurse pot fi obținute din liniile de comandă);

- alocarea prealabilă a spațiului din memoria auxiliară asociat fiecărui job utilizator(se cunoaşte necesarul maxim de memorie externă al fiecărui job).

Gestiunea memoriei

Introducere. În conformitate cu arhitectura von Neumann, memoria primară împreună cu regiştrii CPU şi memoria „cache” formează **memorie executabilă**, deoarece aceste componente sunt implicate în execuția unui proces. CPU poate încărca instrucțiunile acestuia numai din memoria primară. Unitățile de memorie externă (secundară) sunt utilizate pentru a stoca date pentru o mai lungă perioadă de timp. Fişierele executabile, pentru a deveni procese, precum şi informațiile prelucrate de acestea, trebuie să fie încărcate în memoria primară.

Aministratorului memoriei interne este responsabil de alocarea memoriei primare proceselor şi de a acorda asistență programatorului în încărcarea/salvarea informațiilor din/în memoria secundară. Astfel, partajrea memoriei interne de către mai multe procese şi minimizarea timpului de acces sunt obiective de bază ale administratorului memorie interne.

Spațiul de adrese al unui proces

- Componentele unui program sursă sunt reprezentate folosind identificatori, etichete şi variabile, ce reprezintă nişte nume simbolice şi formează **spațiul de nume** al programului sursă. **Faza de compilare** transformă un text sursă într-un modul obiect, adică fiecare nume simbolic este translatat într-o adresă relativă la modulul obiect. **Faza de editare de legături** grupează mai multe module, formând un fişier, numit modul absolut, stocat pe un suport extern până când se cere execuția lui. **Editorului de legături** transformă adresele din cadrul modulelor în aşa-zisele **adrese relocabile. Faza de translatare (relocare)** a adresei constă în transformarea adreselor relative la fişierul executabil, în adrese de memoria internă, realizându-se astfel **imaginea executabilă** a programului. Acest lucru este realizat de o componentă a SO, numită **încărcător**(loader). **Spațiul de adrese al unui proces** este mulțimea locațiilor alocate acestuia, atât din memoria primară, cât şi din cea secundară, servicii ale SO şi resurse. Cea mai mare parte a spațiului de adrese al unui proces se proiectează în locații de memorie primară.
- Altor obiecte referențiate de către un program le sunt asociate adrese de memorie internă. Spațiul de adrese defineşte toate entitățile logice folosite de către un proces şi specifică o adresă prin care ele sunt referențiate. Un program poate fi gândit ca o specificare a unei activități(algoritm) care urmează să fie realizată de către un proces. El conține un set de instrucțiuni care urmează să fie executate şi o mulțime de variabile pe care le utilizează. Când un program sub formă de fişier executabil este gata de execuție, se realizează o corespondență care defineşte unde vor fi plasate procedurile şi datele în spațiul de adrese al procesului.

Încărcarea programului

- Înainte ca un program să fie executat, trebuie să-i fie alocat un spațiu din memoria primară. Dacă se lucrează în multiprogramare, este posibil ca în memorie să fie prezente simultan mai multe programe. Fiecare program foloseşte zona(zonele) de memorie alocată(alocate) lui. De asemenea, pe durata execuției unui program, necesarul de memorie variază. Odată ce sistemul cunoaşte care locații de memorie urmează a fi folosite pentru execuția programului, poate să realizeze corespondența dintre adresele de memorie primară alocate procesului respectiv şi spațiul său de adrese. Deci programul executabil este translatat într-o formă finală accesibilă unității de control a CPU şi încărcat în memoria primară, la o anumită adresă de memorie. Când **contorul de program** este inițializat cu adresa primei instrucțiuni executabile(principalul punct de intrare din program), CPU începe să execute programul.
- În anumite SO, administratorul memoriei poate să şteargă o parte din programul executabil din memoria primară, să-l salveze în memoria secundară şi să elibereze zona de memorie ocupată, astfel încât aceasta să poată fi alocată altor procese. Astfel, chiar după ce un program a fost convertit într-o formă executabilă în memoria internă, el poate fi stocat şi în memoria externă, atâta timp cât este necesar. Totuşi, odată cu execuția procesului, imaginea sa din memoria internă se schimbă, fără a se modifica şi copia existentă pe disc. De exemplu, copia unei date din memoria secundară va fi modificată numai dacă procesul execută o comandă de scriere, prin care în mod explicit modifică zona de pe disc ocupată de data respectivă.

Funcțiile administratorului; obiective ale gestiunii memoriei

Funcțiile administratorului

- Alocarea de spațiu de memorie internă proceselor;
- Realizarea corespondenței dintre spațiul de adrese al procesului şi locații de memorie internă;
- Minimizarea timpului de acces la locațiile de memorie.

- Realizarea acestor funcții este condiționată atât de componenta hardware, cât şi de cea software, conținută în SO. Odată cu evoluția componentelor hardware ale SC, s-au schimbat şi strategiile de administrare a memoriei, pentru a se valorifica aceste îmbunătățiri. Reciproc, strategiile privind gestiunea memoriei au evoluat în timp, ceea ce a condus la evoluția componentelor hardware ale sistemelor de calcul.
- Principalele **obiective** ale gestiunii memoriei sunt:

- calculul de translatare a adresei(relocare);
- protecția memoriei;
- organizarea şi alocarea memoriei operative;
- gestiunea memoriei secundare;
- politici de schimb între procese,
- memoria operativă şi memoria secundară.

Metode clasice de alocare a memoriei

- La **sistemele monoutilizator** unui “job” îi este disponibil întreg spațiul de memorie utilizator. Gestiunea acestui spațiu cade exclusiv în sarcina utilizatorului. El are la dispoziție **tehnici de suprapunere** (overlay) pentru a-şi putea rula programele mari. Ideea de bază este de a păstra în memoria internă numai acele instrucțiuni şi date de care este nevoie permanent. Celelalte grupuri de instrucțiuni sunt încărcate în memoria internă numai atunci când este nevoie de ele, după care sunt evacuate.

Alocarea cu partiții fixe

- **Alocarea cu partiții fixe** (MFT-Memory Fix Tasks sau alocare statică). Se presupune că memoria este împărțită în N zone de lungime fixă numite **partiții**. Presupunem că o partiție i este de lungime N_i şi este alocată unui proces pe toată durata execuției lui, indiferent dacă o ocupă complet sau nu. Editorul de legături pregăteşte programele pentru a fi rulate într-o zonă de memorie prestabilită.
- De obicei, partițiile au lungimi diferite, astefel încât procesele solicită partiții în conformitate cu dimensiunile lor. Dacă un process k are nevoie de n_k unități de memorie ele poate fi încărcat în oricare dintre partițiile i, pentru care $N_i \geq n_k$. În timpul execuției procesului, un spațiu de dimensiune $N_i - n_k$ rămâne neutilizat. Acest fenomen se numeşte **fragmentare internă**. Problema care se pune este să se aleagă partiția astfel încât porțiunea de memorie nefolosită să aibă o dimensiune cât mai mică, adică să se minimizeze diferențele de forma $N_i - n_k$.
- Dacă un proces nu încape în nici una dintre partițiile existente, el nu poate fi executat. Una dintre problemele cele mai dificile este fixarea acestor dimensiuni. Alegerea unor dimensiuni mai mari scade probabilitatea ca unele procese să nu poată fi executate, dar scade şi numărul proceselor active din sistem. În cazul în care există job-uri în sistem care aşteaptă să fie executate, dar toate partițiile libere existente la momentul respectiv sunt prea mici, apare fenomenul de **fragmentare externă a memoriei**.

Selectarea job-urilor care urmează să fie executate se face de către **planificator**, în funcție de necesarul de memorie(pe baza informațiilor transmise de către utilizator sau determinate automat de către sistem) şi de partițiile disponibile existente la momentul respectiv. În general, există două moduri de legare a proceselor la partiții:

- Fiecare partiție are coadă proprie; legarea la o anumită partiție a proceselor se va face pe baza necesității diferenței minime între dimensiunea partiției şi a procesului (**best fit**-cea mai bună potrivire).

- **O singură coadă pentru toate partițiile**; SO va alege pentru procesul care urmează să intre în lucru, în ce partiție se va executa.

Selectarea lucrării se poate face prin:

- o strategie de tip FCFS(**F**irst **C**ome **F**irst **S**erved), care are dezavantajul că o anumită lucrare trebuie să aştepte în coadă, chiar dacă există o partiție disponibilă în care ar începea, iar în fața lui în coadă se află job-uri care necesită partiții mai mari;
- pe baza împărțirii job-urilor în clase de priorități, în funcție de importanța lor, care poate avea dezavantajul prezentat mai sus;
- pe baza celei mai bune potrivirii între dimensiunea job-ului cu dimensiunea partiției.

Metodele prezentate pot fi combinate. De exemplu, dacă avem mai multe job-uri în sistem care au aceeaşi prioritate, va fi ales cel care se potriveşte cel mai bine peste partiția care devine disponibilă.

Legarea prin cozi proprii partițiilor este mai simplă din punctul de vedere al SO; în schimb, legarea cu o singură coadă este mai avantajoasă din punctul de vedere al fragmentării mai reduse a memoriei.

- Deoarece în memorie există mai multe job-uri în execuție, trebuie rezolvate două probleme: **relocarea** şi **protecția memoriei.** O soluție a rezolvării ambelor probleme este ca CPU să conțină două registre speciale, **registrul de bază** şi **registrul limită**. Când lucrarea este planificată pentru execuție, în registrul de bază este încărcată adresa primei instrucțiuni din fişierul executabil, iar registrul limită va conține lungimea partiției.
- Alocarea cu partiții fixe a fost folosită la sistemele generației a III-a de calculatoare(IBM 360, Felix C256/512/1024), dar ea nu este recomandată pentru utilizarea în cadrul sistemelor unde nu se cunoaşte dinainte de ce spațiu de memorie are nevoie procesul pentru a fi executat, aspect întâlnit adesea în cadrul sistemelor de operare moderne.

- **Interschimbarea job-urilor(job-swapping)** apare în cazul sistemelor cu organizarea memoriei în partiții fixe, din necesitatea ca la anumite momente unele dintre ele să fie evacuate din memorie iar altele să fie introduse în memorie. De exemplu, dacă se execută un job şi apare un alt job de prioritate mai înaltă, jobul de prioritate mai slabă va fi evacuat pe disc. În mod normal, un job care a fost evacuat va fi readus în aceeaşi partiție, restricție impusă atât strategia de alocare, cât şi de metoda de relocare. Dacă relocarea se face în momentul asamblării sau în momentul încărcării(relocare statică), job-ul nu poate fi transferat într-o altă partiție; dacă se foloseşte relocarea dinamică(cu registru de bază şi registru limită, de exemplu) acest lucru este posibil.
- Interschimbarea joburilor necesită o memorie externă cu acces direct şi rapid, care să poată îngloba copii ale tuturor imaginilor de memorie utilizator. Toate procesele ale căror imagini de memorie se află pe disc şi care sunt gata să intre în execuție se grupează într-o coadă, în timp ce procesele existente în memorie la momentul respectiv formează altă coadă. Atunci când planificatorul doreşte să lanseze în execuție un proces, el apelează dispecerul care verifică dacă procesul se află în memorie. Dacă nu şi dacă nu există nici o partiție liberă, dispecerul evacuează din memorie unul dintre procese, introduce în locul său procesul dorit, reîncarcă registrele şi transferă controlul procesului selectat. Bineînțeles că o acțiune de acest fel presupune şi cea de salvare a contextului procesului în execuție(a conținuturilor regiştrilor utilizați de către acesta), acțiune care este destul de complexă.

Alocarea cu partiții variabile

- **Alocarea cu partiții variabile** (alocare dinamică sau alocare MVT – Memory Variable Task), reprezintă o extensie a alocării cu partiții fixe, care permite o exploatare mai eficientă a memoriei SC. În cazul multiprogramării cu partiții fixe, problema cea mai dificilă este optimizarea dimensiunii partițiilor, astfel încît să se minimizeze fragmentarea memoriei. De asemenea, se presupune că joburile au o dimensiune cunoscută, ipoteză care nu este în general adevărată.
- Aceste inconveniente pot fi rezolvate dacă se admite **modificarea dinamică a dimensiunii partițiilor**, în funcție de solicitările adresate sistemului şi de capacitatea de memorie încă disponibilă la un moment dat. Prin folosirea acestei metode, numărul şi dimensiunea partițiilor se modifică în timp.
- În momentul în care procesul intră în sistem, el este plasat în memorie într-un spațiu în care încape cea mai lungă ramură a sa. Spațiul liber în care a intrat procesul, este acum descompus în două partiții: una în care se află procesul, iar cealaltă într-un spațiu liber care poate fi alocat altui proces. De asemenea, când un proces îşi termină execuția, spațiul din memorie ocupat de el este eliberat, urmând a fi utilizat de către un alt proces. Apare, deci o alternanță a spațiilor libere cu cele ocupate.

- Pentru a se obține spații libere de dimensiune cât mai mare, SO va declanşa operația de **alipire a unor spații libere vecine sau de compactare a memoriei (relocare a adreselor),** adică de deplasare a partițiilor active către partiția ocupată de către nucleul SO, pentru a se concatena toate fragmentele de memorie neutilizate. De regulă, operația de compactare este complexă, presupunând efectuarea de operații de modificare a adreselor; în practică se aleg soluții de compromis, cum ar fi:

- Se lansează **periodic** compactarea, la un interval de timp fixat, indiferent de starea sistemului. Procesele care nu au loc în memorie aşteaptă compactarea sau terminarea altui proces.
- Se realizează o **compactare parțială** pentru a asigura loc numai procesului care asteaptă.
- Se încearcă numai mutarea unuia dintre procese, cu concatenarea spațiilor rămase libere.

- **Strategii de administrare a spațiului din memoria internă.** Aşa cu am menționat anterior, la un moment dat memoria se prezintă ca o alternanță a spațiilor libere cu cele ocupate. Cele libere vor fi alocate proceselor care cer memorie, iar cele ocupate, când sunt eliberate trebuie, eventual să fie concatenate cu alte spații libere, pentru a obține zone contigue de dimensiune cât mai mare. Deci, sunt necesare metode prin care să se țină evidența spațiilor libere şi a celor ocupate şi să se aloce spațiile de memorie solicitate.

- **Administrarea memoriei folosind liste înlănțuite**. Vom presupune că întreaga cantitate de memorie solicitată la un moment dat este formată dintr-un **şir de octeți consecutivi, care se alocă proceselor** dintr**-**un rezervor de memorie (numit **heap**), de unde se ia acestă memorie. De asemenea, presupunem că există două rutine, una pentru **a aloca** o zonă de memorie şi de a întoarce adresa ei de început şi o a doua rutină pentru **a elibera** spațiul alocat anterior, în vederea refolosirii lui.
- Fiecare zonă liberă începe cu un **cuvânt de control,** care conține un **pointer** către următoarea porțiune liberă şi un camp care conține lungimea zonei respective. La fel se întâmplă în cazul unei **zone ocupate.** O zonă ocupată(respectiv liberă) este reperată după cuvântul ei de control. În timp, eate posibil ca două zone libere să devină adiacente. Sistemul conține o **procedură de comasare** a două zone libere adiacente.
- În momentul în care un proces cere o anumită cantitate de memorie, sistemul caută o zonă liberă de unde să se ocupe o anumită porțiune.

- Pentru aceasta se folosesc următoarele strategii:

- Metoda primei potriviri (First-Fit). Esența metodei constă în aceea că partiția solicitată este alocată în prima zonă liberă în care încape.Principalul avantaj al metodei este simplitatea căutării de spațiu liber.

- Metoda celei mai bune potriviri (Best-Fit). Esența metodei constă în căutarea acelei zone libere care lasă după alocare cel mai puțin spațiu liber. **Avantajul** metodei constă în economisirea zonelor de memorie mai mari. **Dezavantajul** este legat de timpul suplimentar de căutare şi generarea blocurilor de lungime mică, adică **fragmentarea internă excesivă**.

Primul neajuns este eliminat parțial, dacă lista de spații libere se păstrează nu în ordinea crescătoare a adreselor, ci în ordinea crescătoare a lungimilor spațiilor libere; în acest caz algoritmul s-ar complica foarte mult.

- Metoda celei mai rele potriviri (Worst-fit) este dual ă metodei Best-Fit. Esența ei constă în căutarea acelei zone libere care lasă după alocare cel mai mult spațiu liber. Deşi numele metodei sugerează că este vorba despre o metodă mai slabă, în realitate nu este chiar aşa. Faptul că după alocare rămâne un spațiu liber mare, este benefic, deoarece în spațiul rămas poate fi plasată în viitor o altă partiție.

Metoda alocării prin camarazi (Buddy-system)

- **Metoda alocării prin camarazi (Buddy-system)** se bazează pe reprezentarea binară a adreselor şi faptul că dimensiunea memoriei interne este un multiplu al unei puteri a lui 2. Presupunem că se dimensiunea memoriei interne este de forma $cx2^n$, iar unitatea de alocare a memoriei este de forma 2^m.
- **Exemplul 1.** Dacă sistemul are o memorie internă de 32 Mo, atunci c=1 şi n=25. Dacă dimensiunea memoriei interne este de 192 Mo, c=3 şi n=26. De asemenea, se poate considera că unitatea de alocare este de 256 Ko, adică 28.
- Ținând cont de proprietățile operațiilor cu puteri ale lui 2, atât dimensiunile spațiilor alocate, cât şi ale celor libere sunt de forma 2^k, cu $k \leq m \leq n$. În concluzie, sistemul va păstra liste separate ale adreselor spațiilor disponibile, în funcție de dimensiunea lor exprimată ca putere a lui 2. Vom numi lista de ordin k, lista tuturor adreselor unde încep spații libere de dimensiune 2^k. Vor exista astfel n-m+1 liste de spații disponibile.
- **Exemplul 2**. Dacă considerăm că dimensiunea memoriei interne este de 192 Mo, vom avea 17 liste: lista de ordin 8, având dimensiunea unui spațiu de 256 octeți; lista de ordin 9, cu spații de dimensiune 512 ş.a.m.d.

- Presupunem că, fiecare spațiu liber(ocupat) de dimensiune 2^k, are adresa de început un multiplu de 2^k. Două spații libere se numesc **camarazi de ordinul** k, dacă adresele lor A1 şi *A2* verifică una dintre proprietățile următoare:

- A1<A2 , A2=A1+2^k şi A1 mod 2^{k+1}=0

- A2<A1, A1=A2+2^k şi A2 mod 2^{k+1}=0

- Această definiție, exprimă o proprietate fundamentală: Atunci când într-o listă de ordinul k apar doi camarazi, sistemul îi concatenează într-un spațiu de dimensiune 2^k+1 şi reciproc, un spațiu de dimensiune 2^k+1 se poate împărți în două spații de dimensiune 2^k.

- **Algoritmul de alocare de memorie**.
- **Pas 1**. Fie o numărul de octeți solicitați. Se determină *min{p/ m ≤ p ≤ n, o ≤ 2^p}.*
- **Pas 2**. Se determină *k=min{i/p ≤ i≤ n şi lista de ordin i este nevidă}.*
- **Pas 3.** Dacă *k=p*, atunci aceasta este alocată şi se şterge din lista de ordinul *p* altfel se alocă primii 2^p octeți, se şterge zona din lista de ordinul k şi se creează în schimb alte *k-p* zone libere, având dimensiunile $2^p, 2^{p+1},...,2^{k-1}$.
- **Observație.** Pasul 3 al algoritmului se bazează pe egalitatea
- $2^k-2^p=2^p+2^{p+1}+...+2^{k-1}$.
- **Exemplul 3.** Se doreşte alocarea a 1000 octeți, deci p=10. Nu s-au găsit zone libere nici de dimensiunea 2^{10}, nici 2^{11} şi nici 2^{12}. Prima zonă liberă de dimensiune 2^{13} are adresa de început $5x2^{13}$ şi o notăm cu I. Ca rezultat al alocării a fost ocupată zona A de dimensiune 2^{10} şi au fost create încă trei zone libere: B de dimensiune 2^{10}, C de dimensiune 2^{11} şi D de dimensioune 2^{12}. Zonele *B*, *C* şi D se trec respectiv în listele de ordine 10, 11 şi 12, iar zona I se şterge din lista de ordin 13.

- **Algoritmul de eliberare.**
- **Pas 1**. Fie 2^p dimensiunea zonei eliberate. Se introduce zona respectivă în lista de ordinul p.
- **Pas 2.** k=p
- **Pas 3. Verifică** dacă există camarazi de ordin k:
- Dacă **da**, efectuează **comasarea** lor; **Şterge** cei doi camarazi; **Introdu** noua zonă liberă de dimensiune 2^{k+1} în lista de ordin k+1.
- **Pas 4.** k=k+1; goto Pas 3.
- **Exemplul 4.** Să presupunem, de exemplu că la un moment dat zonele A, C şi D de dimensiuni $5x2^{13}$, respectiv $21x2^{11}$ şi $11x2^{12}$ sunt libere, iar zona B de dimensiune $41x2^{11}$ este ocupată, zonele fiind adiacente, în ordinea A, B, C, D. În conformitate cu paşii descrişi mai sus, se execută următoarele acțiuni:
- Se trece zona B în lista de ordin 10.
- Se observă că zonele A şi B sunt camarazi. Drept urmare, cele două zone sunt comasate şi formează o nouă zonă X. Zona X se trece în lista de ordin 11, iar zonele A şi B se şterg din lista de ordin 10.
- Se observă că zonele X şi C sunt camarazi; ele sunt comasate şi formează o zonă Z care se trece în lista de ordin 12, înlocuind zonele X şi C din lista de ordin 11.
- Se observă că Z şi D sunt camarazi; ele sunt şterse din lista de ordin 12, iar în lista de ordin 13 se introduce rezultatul comasării lor.

Paginarea si segmentarea a memoriei

-Conceptul de memorie virtuala

-Translatarea adreselor

-Alocarea paginata a memoriei

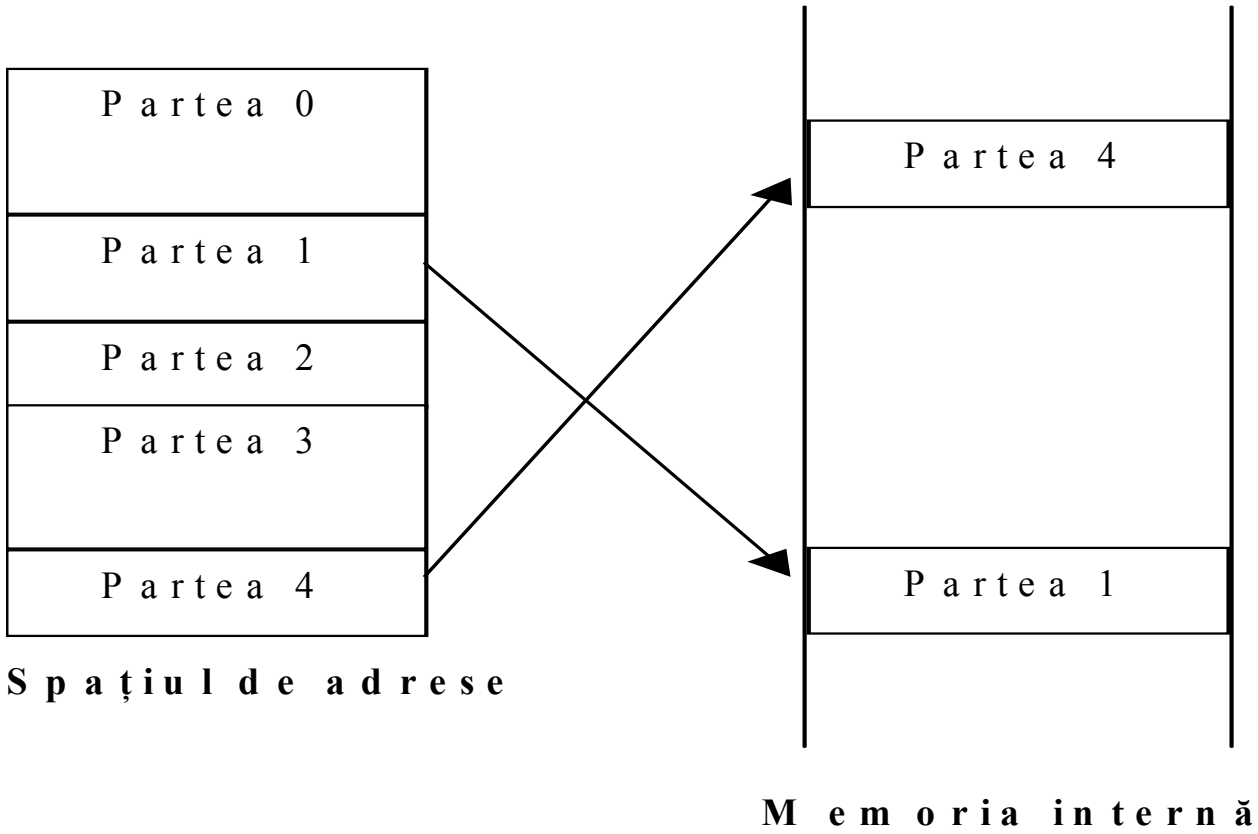
-Alocarea segmentata a memoriei

-Alocarea paginata si segmentata a memoriei

Memoria virtuală

- Dacă presupunem că *n* este dimensiunea magistralei de adrese, atunci numărul maxim de locații adresabile este 2^n . Dimensiunea memoriei interne este mult mai mică decât această valoarea. Acest fapt sugerează prelungirea spațiului de adrese pe un disc rapid. Mulțimea acestor locații de memorie formează memoria virtuală.
- Utilizarea memoriei virtuale, permite execuția proceselor fără a impune necesitatea ca întreg fişierul executabil să fie încărcat în memorie. Spațiul de adrese al procesului este divizat în părți care pot fi încărcate în memoria internă atunci când execuția procesului necesită acest lucru şi transferate înapoi în memoria secundară, când nu mai este nevoie de ele. Spațiul de adrese al unui program se împarte în **partea de cod**, cea **de date** şi cea **de stivă**, identificate atât de compilator cât şi de mecanismul hardware utilizat pentru relocare.
- Partea (segmentul) de cod are evident un număr de componente mai mare, fiind determinată de fazele de execuție (logica) ale programului. De exemplu, aproape toate programele conțin o fază necesară inițializării structurilor de date utilizate în program, alta pentru citirea datelor de intrare, una (sau mai multe) pentru efectuarea unor calcule, altele pentru descoperirea erorilor şi una pentru ieşiri. Analog, există partiții ale segmentului de date. Această caracteristică a programului se numeşte localizare a **referințelor în spațiu** şi este foarte importantă în strategiile utilizate de către sistemele de memorie virtuală.

- Când o anumită parte a programului este executată, este utilizată o anumită porțiune din spațiul său de adrese, adică este realizată o localizare a referințelor. Când se trece la o altă fază a calculului, corespunzătoare logicii programului, este referențiată o altă parte a spațiului de adrese şi, deci se schimbă această localizare a referințelor. în figura urm., spațiul de adrese este divizat în 5 părți.



- Numai părțile 1 şi 4 din spațiul de adrese corespund unor faze ale programului care se execută la momentul respectiv, deci numai acestea vor fi încărcate în memoria internă. Părți diferite ale programului vor fi încărcate în memoria primară la momente diferite, în funcție de localizarea în cadrul procesului.
- Sarcina administratorului memoriei este de a deduce localizarea programului şi de a urmări încărcarea în memorie a partițiilor din spațiul de adrese corespunzătoare, precum şi de a ține evidența acestora în memoria internă, atâta timp cât sunt utilizate de către proces.
- Administratorul memoriei virtuale alocă porțiuni din memoria internă care au aceeaşi dimensiune cu cele ale partițiilor din spațiul de adrese şi, deci încarcă imaginea executabilă a părții corespondente din spațiul de adrese într-o zonă din memoria primară. Acest lucru are ca efect utilizarea de către proces a unei cantități de memorie mult mai reduse.

Translatarea adreselor

- Funcția de translatare a adreselor virtuale, Ψ_t este o corespondență, variabilă în timp a **spațiului de adrese virtuale** ale unui proces, în **spațiul de adrese fizice**, adică mulțimea tuturor locațiilor din memoria internă alocate acelui proces.

- Se cunosc două metode de virtualizare: **alocare paginată şi alocare segmentată.** Deci:

$$\Psi_t : <Spatiul\_de\_adrese\_relative> \rightarrow <Spatiul\_de\_adrese\_fizice> \cup \{\Omega\}$$

- în care t este un număr întreg, care reprezintă timpul virtual al procesului iar Ω este un simbol care corespunde adresei nule. Când un element i, al spațiului de adrese virtuale este încărcat în memoria internă, $\Psi_t(i)$ este adresa fzică unde adresa virtuală i este încărcată.

- Dacă $\Psi_t(i) = \Omega$, la momentul virtual t şi procesul face referință la locația i, atunci sistemul întreprinde următoarele acțiuni:
- Administratorul memoriei cere oprirea execuției procesului.
- Informația referențiată este regăsită în memoria secundară şi încărcată într-o locație de memorie k.
- Administratorul memoriei schimbă valoarea funcției .
- Administratorul memoriei cere reluarea execuției programului.
- Observăm că locația referențiată din spațiul de adrese virtuale nu este încărcată în memoria primară, după ce s-a declanşat execuția instrucțiunii cod maşină respective. Acest lucru va declanşa **reexecutarea** instrucțiunii după ce locația respectivă a fost încărcată din memoria virtuală în memoria primară.
- De asemenea, dimensiunea spațiului de adrese virtuale ale unui proces, este **mai mare** decât dimensiunea spațiului de adrese fizice alocat procesului, spre deosebire de metodele când întregul fişier executabil era încărcat în memorie.

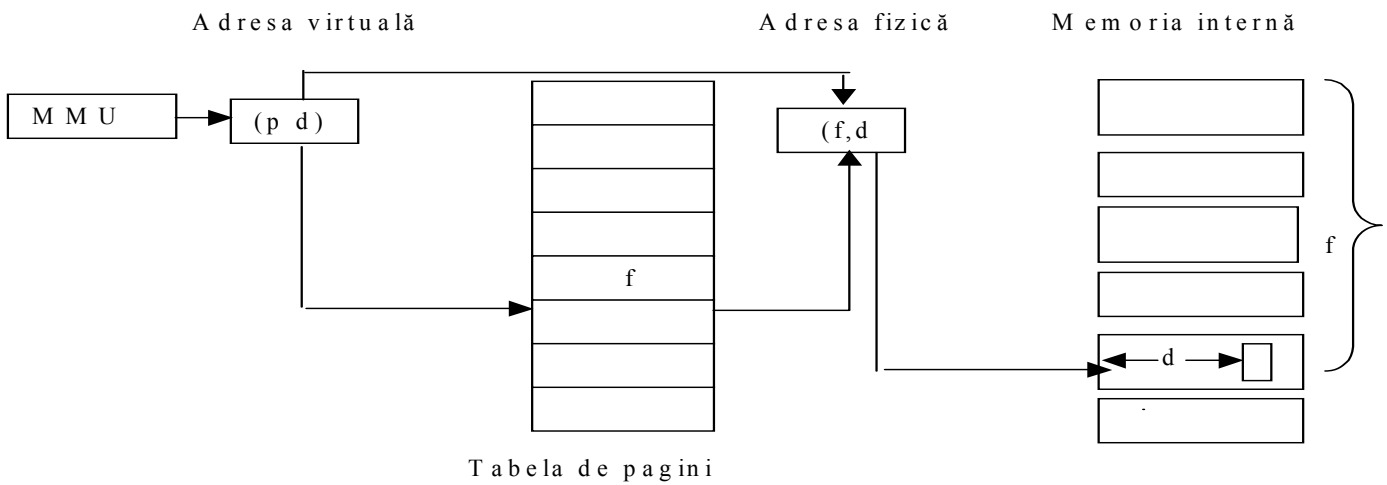
Alocarea paginată

- **Alocarea paginată** a apărut la diverse SC pentru a evita fragmentarea memoriei interne, care apare la metodele anterioare de alocare.
- Memoria virtuală este împărțită în zone de lungime fixă numite **pagini virtuale.** Paginile virtuale se păstrează în memoria secundară. Memoria operativă este împărțită în zone de lungime fixă, numite **pagini fizice.** Lungimea unei pagini fizice este fixată prin hard. Paginile virtuale şi cele reale(fizice) au aceeaşi lungime, lungime care este o putere a lui 2 şi care este o constantă a sistemului(de exemplu 1Ko, 2Ko etc). Să presupunem că G reprezintă numărul de locații de memorie virtuală ale unui fişier executabil, ale căror adrese sunt cuprinse între 0 şi G-1. De asemenea, să notăm cu H numărul locațiilor din memoria internă, necesare pentru a încărca conținutul fişierului executabil, în timpul execuției procesului (H<G). De asemenea, cele G-1 locații virtuale, corespund la n, cu $n=2^g$ pagini virtuale, fiecare pagină fiind de dimensiune c, cu $c=2^h$. Spațiul de adrese din memoria primară poate fi gândit ca o mulțime de m pagini fizice,$m=2^j$ fiecare având dimensiunea $c=2^h$, deci cantitatea de memorie internă alocată procesului va fi $H=2^{h+j}$. La un anumit moment al execuției, conform principiului localizării, numai o parte din paginile virtuale vor fi încărcate în memoria internă, în pagini fizice. Problema care se pune este translatarea fiecărei adrese virtuale într-o adresă fizică.

Translatarea adreselor virtuale

- Fie $N = \{d_0, d_1, .., d_{n-1}\}$ mulțimea paginilor în spațiul de adrese virtuale şi $M = \{b_0, b_1, .., b_{m-1}\}$ mulțimea paginilor fizice din memoria primară alocate procesului. O adresă virtuală este un întreg i, unde
- $0 \le i < G = 2^{g+h}$, deoarece există $n = 2^g$ pagini, fiecare având 2^h cuvinte(locații) de memorie. O adresă fizică, k, este o adresă de memorie, de forma , $k = U2^h + V (0 \le V < 2^h)$ unde U este numărul pagini fizice. Deci $U2^h$ este adresa din memoria primară a primei locații din pagina fizică, iar V este deplasamentul din cadrul paginii fizice U. Deorece procesului îi sunt alocate 2^J pagini fizice, vom avea $H = 2^{j+h}$ locații de memorie fizică care pot fi utilizate de proces. Corespondența dintre adresele virtuale şi cele fizice are forma
- $\Psi_t : [0,...,G-1] \rightarrow < U, V > \cup \{\Omega\}$

- Deoarece fiecare pagină are aceeaşi dimensiune c, adresa virtuală i poate fi convertită într-un număr de pagină şi un deplasament în cadrul paginii, numărul de pagină fiind i div c, iar deplasamentul i mod c. Dacă considerăm reprezentarea binară a adresei, atunci numărul de pagină poate fi obținut prin deplasare spre dreapta a adresei cu h poziții, iar deplasamentul prin extragerea primilor g biți cei mai puțin semnificativi din adresă. Deci fiecare adresă virtuală, repectiv adresă fizică va fi de forma (p,d), respectiv de forma (f,d), unde p este numărul paginii virtuale, f este numărul paginii fizice, iar d este adresa (deplasamentul) în cadrul paginii.
- Orice pagină virtuală din spațiul de adrese al procesului, poate fi încărcată în oricare din paginile fizice din memoria internă alocate acestuia. Acest lucru este realizat printr-o componentă hardware numită MMU(**M**emory **M**anagement **U**nit), care implementează funcția (figura urm).



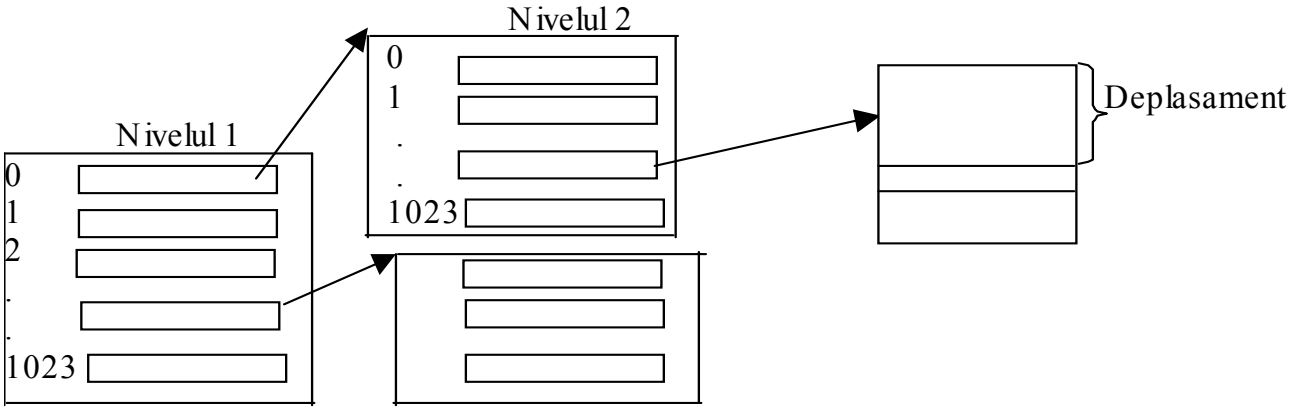
- Dacă este referențiată o locație dintr-o pagină care nu este încărcată în memoria internă, MMU opreşte activitatea CPU, pentru ca SO să poată executa următorii paşi:
- Procesul care cere o pagină neîncărcată în memoria internă este suspendat.
- Administratorul memoriei localizează pagina respectivă în memoria secundară.
- Pagina este încărcată în memoria internă, eventual în locul altei pagini, dacă în memoria internă nu mai există pagini fizice libere alocate procesului respectiv şi în acest caz, tabela de pagini este modificată.
- Execuția procesului se reia din locul în care a fost suspendat.
- Fiecare proces are propria lui tabelă de pagini, în care este trecută adresa fizică a paginii virtuale, dacă ea este prezentă în memoria operativă. La încărcarea unei noi pagini virtuale, aceasta se depune într-o pagină fizică liberă. Deci, în memoria operativă, paginile fizice sunt distribuite în general necontinuu, între mai multe procese. Spunem că are loc o **proiectare a spatiului virtual peste cel real.**
- Acest mecanism are avantajul că foloseşte mai eficient memoria operativă, fiecare program ocupând numai memoria strict necesară la un moment dat. Un alt avantaj este posibilitatea folosirii în comun, de către mai multe programe, a instrucțiunilor unor proceduri. O procedură care permite acest lucru se numeste **procedură reentrantă.**

- **Evidența paginilor virtuale** încărcate în pagini fizice se poate realiza încărcată în memorie.Tabela de pagini poate fi privită ca o funcție, care are ca argument numărul de pagină virtuală şi care determină numărul de pagină fizică. Pe baza deplasamentului se determină locația din memoria fizică unde se va încărca locația virtuală referențiată. Această metodă ridică două mari probleme:

1. Tabela de pagini poate avea un număr mare de intrări. Calculatoarele moderne folosesc adrese virtuale pe cel puțin 32 de biți. Dacă, de exemplu o pagină are dimensiunea de 4K, atunci numărul intrărilor în tabelă este mai mare decât un million. În cazul adreselor pe 64 de biți numărul intrărilor în tabelă va fi, evident mult mai mare.
2. Corespondența dintre locația de memorie virtuală şi cea din memoria fizică trebuie să se realizeze cât mai rapid posibil; la un moment dat, o instrucțiune cod maşină poate referenția una sau chiar două locații din memoria virtuală.

O metodă mult mai eficientă, care înlocuieşte căutarea secvențială cu cea arborescentă este **organizarea tabelei pe mai multe niveluri.** Astfel, o adresă virtuală pe 32 de biți de exemplu, este un triplet (Pt1, Pt2, d), primele două câmpuri având o lungime de 10 biți, iar ultimul de 12 biți.

- În figura urm. este ilustrată o astfel de tabelă de pagini.



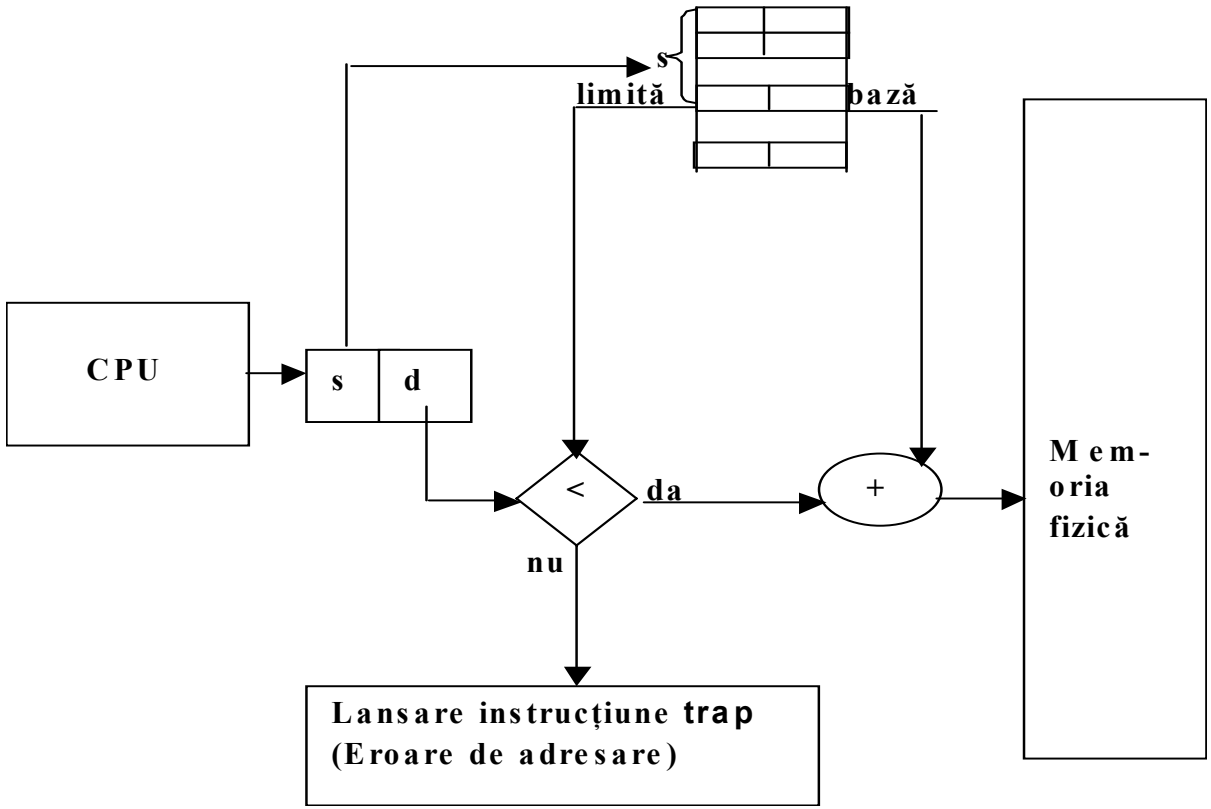
- Observăm că numărul de intrări în tabela de nivel 1 este de 2^{10}=1024, iar dimensiunea unei pagini este de 2^{12}=4x2^{10}=4K. Fiecare intrare în această tabelă conține un pointer către o tabelă a nivelului 2. Când o adresă virtuală este prezentată MMU, se extrage valoarea câmpului PT1, care este folosit ca index în tabela de la nivelul 1. Pe baza acestuia, se găseşte adresa de început a uneia dintre tabelele de la nivelul 2. Câmpul PT2 va fi un index în tabela de la nivelul 2 selectată, de unde se va lua numărul de pagină fizică corespunzător numărului de adresă virtuală conținut în adresa referențiată. Pe baza acestui număr şi a deplasamentului (d), se determină locația de memorie fizică în care va fi încărcată respectiva locație din memoria virtuală. În condițiile specificate, o tabelă de la nivelul 2 va gestiona o zonă de memorie de capacitate 1024 x 4 K=4 M.

- **Observație.** Tabela de pagini se poate organiza şi pe 3 sau 4 niveluri, în funcție de dimensiunea memoriei virtuale şi a celei fizice. Dacă tabela are o dimensiune mare, ea poate fi păstrată(cel puțin parțial) şi pe un disc rapid.
- **Exemplu.** Fie adresa virtuală Ox00403004 (4202996 în zecimal). Dacă se face transformarea în binar se obține valoarea 1 pentru câmpul PT1, 3 pentru PT2, iar pentru deplasament valoarea 4. Pe baza valorii lui PT1, MMU va găsi că componenta PT2 a adresei se află în a doua tabelă de la nivelul 2, care corespunde la adrese între 4M şi 8M . Folosind PT2 ca index în a doua tabelă de nivel 2, având valoarea 3, se determină că adresa locației este cuprinsă între adresele relative la pagina respectivă de memorie 12292 şi 16383, corespunzătoare adreselor absolute 4 202 592 şi 4 210 687. Se adugă valoarea 4 a deplasamentului şi se obține adresa locației, adică 4 202 596. Pe baza valorii bitului **present/absent** se determină dacă pagina virtuală este (sau nu) adusă în memoria fizică.

Alocare segmentată.

- Din punctual de vedere al utilizatorului, o aplicație este formată dintr-un program principal şi o mulțime de subprograme(funcții sau proceduri). Aceste folosesc diferite structuri de date (tablouri, stive etc), precum şi o mulțime de simboluri (variabile locale sau globale). Toate aceste sunt identificate printr-un nume.
- Pornind de la această divizare logică a unui program, s-a ajuns la o metoda de alocării **segmentare** a memoriei. Spre deosebire de metodele de alocare a memoriei bazate pe partiționare, unde fiecărui proces trebuie să i se asigure un spațiu contiguu de memorie, mecanismul de alocare segmentată, permite ca un proces să fie plasat în zone de program distincte, fiecare dintre ele conținând o entitate de program, numit **segment**. Segmentele pot fi definite explicit prin directive ale limbajului de programare sau implicit prin semantica programului.
- De exemplu, un compilator de Pascal poate crea segmente **de cod** pentru fiecare procedură sau funcție, segmente pentru **variabilele globale,** segmente pentru **variabilele locale,** precum şi **segmente de stivă**. Deosebire esențială dintre alocarea paginată şi cea segmentată este aceea că segmentele sunt de **lungimi diferite.**

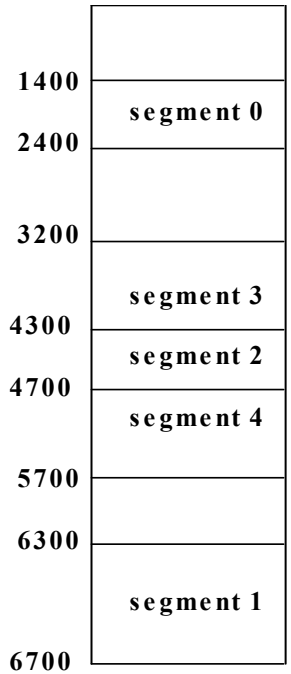
- În mod analog cu alocarea paginată, **o adresă virtuală** este o pereche (s,d), unde s este numărul segmentului iar d este adresa din cadrul segmentului. Adresa reală (fizică) este o adresă obisnuită. Transformarea unei adrese virtuale într-o adresă fizică, se face pe baza unei **tabele de segmente**. Fiecare intrare în acestă tabelă este compusă dintr-o **adresă de bază**(adresa fizică unde este localizat segmentul în memorie) şi **lungimea segmentului(limită)**. În figura urm. este ilustrat modul de utilizare a tabelei de segmente.



- Componenta s a adresei virtuale, este un indice în tabela de segmente. Valoarea deplasamentului d trebuie să fie cuprinsă între 0 şi lungimea segmentului respectiv(în caz contrar este lansată o instrucțiune trap, care generează o întrerupere). Dacă valoarea d este validă, ea este adăugată adresei de început a segmentului şi astfel se obține adresa fizică a locației respective.
- **Exemplu.** Să presupunem că avem un proces care, din punct de vedere logic este format dintr-un program principal, un subprogram şi în care se apelează o procedură de sistem. Corespunzător celor trei entități vom avea trei segmente, la care se adaugă un segment pentru stivă şi unul pentru tabele de simboluri. Figura urm. redă alocarea de spațiu în memoria internă pentru acest proces.

Număr segment	Limită	Bază
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Tabela de segmente



Memoria fizică

- Segmentul 2 are o lugime de 400 octeți şi începe la locația 4300. Astfel, unei referințe la octetul 53 al segmentului 2, îi corespunde locația din memoria internă 4500+53=4353.
- Avantajele alocării segmentate, față de cea bazată pe partiții sunt:
- se pot creea **segmente reentrante**, care pot fi partajate de mai multe procese. Pentru aceasta este suficient ca toate procesele să aibă în tabelele lor aceeaşi adresă pentru segmentul respectiv.
- se poate realiza o **bună protecție a memoriei**. Fiecare segment în parte poate primi alte drepturi de acces, drepturi trecute în tabela de segmente. De exemplu, un segment care conține instrucțiuni cod maşină, poate fi declarat “read-only” sau executabil. La orice calcul de adresă, se verifică respectarea modului de acces al locațiilor din segmental respectiv.

Alocarea segmentată cu paginare

- Cele două metode de alocare a memoriei au avantajele şi dezavantajele lor. În cazul segmentării poate să apară fenomenul de fragmentare. În cazul paginării, se efectuează o serie de operații suplimentare de adresare. De asemenea, unele procesoare folosesc alocarea paginată (Motorola 6800), iar altele folosesc segmentarea (Intel 80x86, Pentium). Ideea segmentării cu paginare, este aceea că alocarea spațiului pentru fiecare segment să se facă paginat. Această metodă este utilizată de sistemele de operare actuale.
- Spațiul de adrese virtuale al fiecărui proces este împărțit în două partiții; prima partiție este utilizată numai de către procesul respectiv, pe când cealaltă partiție, este partajată împreună cu alte procese. Informațiile despre prima partiție, respective a doua partiție sunt păstrate în **descriptorul tabelei locale (LDT – L**ocal **D**escriptor **T**able**),** respectiv **descriptorul tabelei globale (GDT – G**lobal **D**escriptor **T**able**).** Fiecare intrare în LDT/GDT este reprezentată pe 8 octeți şi conține informații despre un anumit segment, printre care adresa de început şi lungimea acelui segment.
- Adresa virtuală este o pereche (selector, deplasament). Selectorul este reprezentat pe 16 biți şi este un triplet (s, g, p) în care s este numărul de segment (13 biți), g specifică dacă segmental este din LDT sau GDT şi p specifică protecția. Deplasamentul este reprezentat pe 32 de biți şi specifică adresa locală a unui octet în cadrul segmentului.

Translatarea unei adrese virtuale în adresă fizică

- se realizează astfel:
- registrul de segment conține un pointer către intrarea din LDT sau GDT corespunzătoare acelui segment;
- de aici se iau adresa de început şi deplasamentul;
- dacă adresa respectivă este validă, valoarea deplasamentului este adăugată la adresa de început şi astfel se obține o adresă liniară, care apoi este translatată într-o adresă fizică.
- Aşa cum am specificat mai devreme, fiecare segment este împărțit în pagini, fiecare pagină având o dimensiune de 4 Ko. Datorită dimensiunii posibile a unui segment, tabelele de paginii poate avea un număr mare de intrări (până la 1 000 000). Deoarece fiecare intrare are o lungime de 4 octeți, rezultă că o astfel de tabelă poate avea o dimensiune de 4 Mo, deci ea nu poate fi păstrată ca o listă liniară în memoria internă. Soluția este de a folosi o tabelă pe două niveluri. Mecanismul de translatare a adresei este similar cu cel utilizat în cazul alocării paginate a memoriei.