

Marin Vlada, Universitatea din Bucureşti, Facultatea de Matematică și Informatică

OLD 2018- <http://prof.unibuc.ro/2018/02/info/>

NEW 2019- <https://unibuc.ro/user/marin.vlada/>

SO – documente - <https://unibuc.ro/user/Marin.Vlada/?profiletab=documents>

# Sisteme de operare (Operating Systems)

## CUPRINS

- Programa analitică – pag. 2
- Bibliografie & Notes – pag. 4
- Evaluare. Teme și subiecte pentru proiecte (subiecte tip A și B) – pag. 5
- Elemente introductive – pag. 7; Aspecte generale, SEISO - Software Educațional pentru învățarea Sistemelor de Operare - <http://www.c3.cniv.ro/so-mv/index.htm>
- Apariția și evoluția sistemelor de operare – pag. 12
- Curs general - sisteme de operare – pag. 30
- Anexe de studiu și analiză– sisteme de operare - pag 262

Bine ați venit!

## Welcome to Your Classroom!

[Universitatea din Bucureşti, Facultatea de  
Matematică și Informatică](#)

Programul de studii: INFORMATICĂ

Sisteme de operare

Anul II, semestrul I

Contact: Conf. univ. dr. MARIN

VLADA

Web:

<https://unibuc.ro/user/marin.vlada/>

[http://old.unibuc.ro/prof/vlada\\_m/](http://old.unibuc.ro/prof/vlada_m/)

Blog: <http://mvlada.blogspot.ro/>

Proiecte: [c3.cniv.ro](http://c3.cniv.ro), [c3.icvl.eu](http://c3.icvl.eu)

MOTTO: *I'm a bit like an operating system really... „multitasking” between these two OHPs, „interrupting” you, ... „sending you to sleep”... (John Bates).*

*“It would appear that we have reached the limits of what is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years.” John von Neumann, 1949 (Wobbe Vegter,*



Sursa image: Pleiades Suoercomputer (NASA),  
[www.nas.nasa.gov/hecc/resources/pleiades.html](http://www.nas.nasa.gov/hecc/resources/pleiades.html)

*Do not forget to be creative! This is an option and a way to happiness.* (M. Vlada 2010)

**Predicting the Future of Computing:** „Readers are invited to make predictions and collaboratively edit this timeline, which is divided into three sections: a sampling of past advances, future predictions that you can push forward or backward in time (but not, of course, into the past), and a form for making and voting on predictions.” The New York Times.

## Programa analitică

### Descriptori

**Course:** OPERATING SYSTEMS | Bachelor of Science (Computer Science) | **Software:** Windows, Unix, Linux, Java Platform

**Definition.** *Operating System:* the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.

**Word keys:** Memoria fizică, memoria virtuală, alocarea dinamică a memoriei, algoritmi de paginare, algoritmi de înlocuire a paginii; gestiunea memoriei în Unix și Windows, sincronizarea proceselor, Apeluri Sistem (System Calls), Mașini Virtuale.

### Comentariu introductiv

- Complexitatea aplicațiilor de utilizare a calculatorului în diverse domenii de activitate (inclusiv în educație), a determinat perfecționarea, atât a sistemelor de operare și limbajelor de programare, cât și a tehnologiilor și platformelor. Au fost concepute și elaborate noi sisteme de operare, noi limbi de programare, noi tehnologii. Dacă în anii '70 inventarea și utilizarea microprocesorului au însemnat o revoluție în domeniul arhitecturii calculatoarelor, în anii '90 a fost o adeverată revoluție, atât în domeniul rețelelor de calculatoare, cât și în domeniile limbajelor de programare (**Java** și **JavaScript**) și sistemelor de operare (**Linux**, **Windows**). Astfel, au apărut tehnologiile Web. Trebuie menționate dezvoltarea și evoluția limbajului **C++** care în anii '80 a implementat și dezvoltat modelul orientat spre obiecte (*modelul programării obiectuale* are rădăcini în limbajele **SmallTalk**, **Lisp** etc.) și programarea orientată spre obiecte (*OOP-Object Oriented Programming*).
- Dacă în deceniul 70 (secolul XX) la nivel universitar, erau câteva discipline proprii informaticii, astăzi există domenii complexe ale Informaticii: **Sisteme de operare**, Arhitectura calculatoarelor, Programare și Software Engineering, Rețele de calculatoare și Computing, Baze de date și sisteme informaticice, Programare și dezvoltare Web, Grafică pe calculator și realitate virtuală, Geometria computațională, Modelare și simulare, Calcul paralel și distribuit, Inteligență artificială și sisteme expert, Ingineria cunoașterii.

## OBIECTIVE

- Însusirea conceptelor teoretice ale sistemelor de operare. Cunoasterea și utilizarea sistemelor de operare Windows, Unix, Linux;
- Competente privind dezvoltarea deprinderile practice pentru înțelegerea modului de proiectare a aplicațiilor de sistem ce folosesc intensiv serviciile oferite de sistemele de operare studiate.
- Înțelegerea și asimilarea principalelor cunoștințe privind conceperea și funcțiile unui SO. Operare și comenzi în Windows, Unix, Linux
- Laborator – Consolidarea cunoștințelor de la curs prin aplicații, testări; elaborarea de aplicații ce folosesc comenzi ale unui SO. Aplicații și proiecte folosind programare C++, Java

## CONTINUT

1. STRUCTURA SISTEMELOR DE OPERARE: Evoluția sistemelor de operare; Structura și funcțiile generale ale sistemului de operare; Serviciile Sistemului de Operare; Apeluri Sistem (System Calls); Mașini Virtuale;
2. ADMINISTRAREA UNITATILOR: Organizarea sistemului de I/O, Administrarea directă a I/O cu testare periodică (polling), Administrarea operațiilor de I/O orientată pe intreruperi, Proiecția în memorie a I/O, Accesul direct la memorie, Utilizarea zonelor tampon (“buffering”)
3. GESTIUNEA MEMORIEI: Memoria fizică, memoria virtuală, alocarea statică a memoriei, alocarea dinamică a memoriei, alocarea paginată a memoriei, alocarea segmentată a memoriei, algoritmi de paginare; algoritmi de înlocuire a paginii; gestiunea memoriei în Unix și Windows;
4. SISTEMUL DE FISIERE: Conceptul de fișier, Organizarea fișierelor, conceptual de director, alocarea spațiului pentru fișiere pe disc, evidența spațiului liber de pe disc, operații cu fișiere, implementarea sistemului de fișiere, performanțele sistemului de fișiere, fiabilitatea sistemului de fișiere, gestiunea blocurilor libere, Protectia și securitatea datelor (sistem de autentificare, sistem de autorizare, protectia resurselor, protectia memoriei);
5. GESTIUNEA PROCESELOR: Stările unui proces, paralelismul proceselor, sincronizarea proceselor, comunicarea între procese, probleme clasice de coordonare a proceselor, Planificarea proceselor, interblocarea proceselor, fire de execuție, schema generală de planificare; criterii de performanță (algoritmi de planificare, algoritmul Round Robin), diagrama stărilor unui proces (procese și threaduri, comutarea proceselor, procese Unix și Windows).

## LABORATOR/SEMINAR ÎN SISTEM TUTORIAL

1. STRUCTURA SISTEMELOR DE OPERARE: Evoluția sistemelor de operare; Structura și funcțiile generale ale sistemului de operare; Serviciile Sistemului de Operare; Apeluri Sistem (System Calls); Mașini;
2. GESTIUNEA MEMORIEI: Memoria fizică, memoria virtuală, alocarea statică a memoriei, alocarea dinamică a memoriei, alocarea paginată a memoriei, alocarea segmentată a

memoriei, algoritmi de paginare; algoritmi de inlocuire a paginii; gestiunea memoriei in Unix si Windows;

3. SISTEMUL DE FISIERE: Organizarea fișierelor, conceptual de director, alocarea spațiului pentru fișiere pe disc, evidența spațiului liber de pe disc;

4. PROTECTIA DATELOR: Protectia si securitatea datelor (sistem de autentificare, sistem de autorizare, protectia resurselor, protectia memoriei);

5. GESTIUNEA PROCESELOR: Stările unui proces, paralelismul proceselor, sincronizarea proceselor, comunicarea intre procese, probleme clasice de coordonare a proceselor;

6. GESTIUNEA PROCESELOR: Planificarea proceselor, interblocarea proceselor, fire de executie, schema generală de planificare; criterii de performanță (algoritmi de planificare, algoritmul Round Robin)

## Bibliografie

1. Silberschatz A., Galvin P.B. and Gagne G., Operating Systems Concepts, 8th edn. John Wiley & Sons, 2009
2. A. Tanenbaum, Sisteme de operare moderne, Ed. Teora, 2004
3. A. Tanenbaum, Goodman, J. R., Organizarea structurata a calculatoarelor, Ed. Byblos, 2004
4. Deitel H., Operating Systems 3/e, Ed. Prentice Hall, 2004
5. Silberschatz A. Operating Systems Concepts. Seventh Edition, Wesley Publishing Company, 2006
6. Tanenbaum A. Modern Operating Systems, Prentice Hall, 2002
7. Gh. Dodescu, A. Vasilescu, B. Oancea, Sisteme de operare, Editura Economică, 2003
8. Sorin Adrian Ciureanu - Sisteme de Operare, Editura Printech, 2004
9. D. Acostachioae, Administrarea si configurarea sistemelor Linux, Ed. Polirom, 2003
10. S. Buraga, G. Ciobanu, Atelier de programare in retele de calculatoare, Ed. Polirom, 2001
11. L. Peterson, B. Davie, Retele de calculatoare: o abordare sistematică, ALL/Teora, Ed. Morgan Kaufmann, 2001/2004
12. M. Vlada, Apariția și evoluția sistemelor de operare în M. Vlada (coord.), Istoria informaticii românești. Apariție, dezvoltare și impact, vol. I Contextul international, Ed. MATRIXRO, 2019, pag. 307-326 (sect. 1.9)
13. M. Vlada, Sisteme de operare, Universitatea din Bucuresti, <http://prof.unibuc.ro/2018/02/info/>, tutorial, curs Online – <http://ebooks.unibuc.ro/informatica/Seiso/>
14. M. Vlada, Informatica, Ed. Ars Docendi, 1999
15. List of Operating Systems: <http://www.operating-system.org/>
16. Liste of Operating Systems: <http://en.wikipedia.org/>
17. GNU Operating Systems: <http://www.gnu.org/>
18. The von Neumann Architecture of Computer Systems, [http://www.brown.edu/Research/Istrail\\_Lab/von\\_neumann.php](http://www.brown.edu/Research/Istrail_Lab/von_neumann.php), Istrail și Solomon Marcus: [http://www.brown.edu/Research/Istrail\\_Lab/papers/Istrail-Marcus012912FINAL.pdf](http://www.brown.edu/Research/Istrail_Lab/papers/Istrail-Marcus012912FINAL.pdf)
19. Von Neumann, J. 1981. First draft of a report on the EDVAC.” In Stern, N. From ENIAC to Univac: An Appraisal of the Eckert-Mauchly Computers. Digital Press, Bedford, Massachusetts, <http://www.csupomona.edu/~hnriley/www/VonN.html>
20. John von Neumann’s EDVAC Report 1945 John von Neumann’s 1945 on June 30 by Hungarian mathematician John von Neumann (1903-1957), <http://www.velocityguide.com/computer-history/john-von-neumann.html>, <http://www.wps.com/projects/EDVAC/>
21. The Virtual von Neumann Architecture and the global computer, <http://meta-artificial.blogspot.com/2005/07/virtual-von-neumann-architecture.html>

## Notes

1. List of Operating Systems: <http://www.operating-system.org/>
2. Liste of Operating Systems: <http://en.wikipedia.org/>
3. GNU Operating Systems: <http://www.gnu.org/>

4. Operating Systems: <http://www.google.com/>
5. Operating Systems Articles: <http://www.articlesbase.com/>
6. Memory Virtual: [Peter J. Denning](#) is best known for pioneering work in virtual memory. He was a pioneer in the development of principles for operating systems and contributed the memory management methods used in all operating systems.. Read more:  
<http://mvlada.blogspot.com/2011>
7. The von Neumann Architecture of Computer Systems  
([http://www.brown.edu/Research/Istrail\\_Lab/von\\_neumann.php](http://www.brown.edu/Research/Istrail_Lab/von_neumann.php), Istrail si Solomon Marcus: [http://www.brown.edu/Research/Istrail\\_Lab/papers/Istrail-Marcus012912FINAL.pdf](http://www.brown.edu/Research/Istrail_Lab/papers/Istrail-Marcus012912FINAL.pdf))
8. Von Neumann, J. 1981. First draft of a report on the EDVAC." In Stern, N. *From ENIAC to Univac: An Appraisal of the Eckert-Mauchly Computers*. Digital Press, Bedford, Massachusetts, <http://www.csupomona.edu/~hnriley/www/VonN.html>
9. John von Neumann's EDVAC Report 1945 John von Neumann's 1945 on June 30 by Hungarian mathematician [John von Neumann](#) (1903-1957) , <http://www.velocityguide.com/computer-history/john-von-neumann.html>
10. <http://www.wps.com/projects/EDVAC/>
11. The Virtual Von Neumann Architecture and the global computer
12. <http://meta-artificial.blogspot.com/2005/07/virtual-von-neumann-architecture.html>

## EVALUARE. Teme și subiecte pentru PROIECTE

**NOTĂ.** Evaluarea cunoștințelor/competențelor: prezentarea (face-to-face) unui proiect (teme A și B, întrebări/subiecte). Proiectul este un fisier format .doc ce descrie o temă privind studiul, analiza, testarea facilităților, comenziilor, etc., folosind explicații, comparații, scheme, capturi de imagini, etc. și este reprezentat pe suport CD și suport hartie. Relevante sunt: argumentele demonstrative, analizele și testările, aplicațiile executate. Nu se acceptă suport Stick/Flash Memory.

### Teme și subiecte pentru PROIECTE

#### Subiecte tip A. Conceptie și utilizare SO

1. Funcțiile și caracteristicile unui SO (Gestiunea proceselor și procesoarelor, Gestiunea memoriei, Gestiunea perifericelor, Gestiunea fișierelor, Tratarea erorilor, Modul de utilizare a resurselor, Gradul de comunicare a proceselor în multiprogramare, SO pentru arhitecturi paralele)
- 2 Structura și componente SO (UNIX/Linux/Windows) (partea de control, partea de servicii, administrare și coordonare, planificare și execuție)
3. Organizarea fișierelor și directoarelor în sistemul UNIX/Linux/Windows (Clasificarea fișierelor după structură, Clasificarea fișierelor după tip, structura, localizare, comenzi, operații generale, operații asupra continutului: sort, head și tail, Organizarea fișierelor ce folosesc FAT, Organizarea fișierelor în HPFS, Organizarea fișierelor în NTFS)

4. Gestiunea fișierelor (Atribute si operații cu fișiere, Alocarea fișierelor pe disc: Alocarea contiguă, Alocarea înlanțuită, Alocarea indexată, Fiabilitatea sistemelor de fișiere)
5. Gestiunea sistemului I/O (Rutine de tratare a intreruperilor, Drivere, Programe-sistem independente de dispozitive, Primitive de nivel utilizator, Cache-ul de hard disc)
6. SO pentru retea si calculatoare paralele (SO cu microprocesoare, Programarea paralela: Memoria partajată între procese, Exemple de programare paralela, SO distribuit: Comunicare sistem client / server, Apeluri de proceduri la distanță, Comunicare în grup, Sisteme de operare distribuite: AMOEBA, GLOBE)
7. Securitatea SO (Depășirea zonei de memorie tampon (Buffer Overflow), Ghicirea parolelor (Password guessing), Interceptarea rețelei, Atacul de refuz al serviciului (Denial Of Service), Falsificarea adresei expeditorului (e-mail spoofing), Cai troieni (Trojan Horses), Uși ascunse (Back doors), Viruși, Viermi)
8. Securitatea în Linux (Open Source, Programe ce depistează și corectează vulnerabilități, Auditarea sistemului)
9. Aplicații în Linux (Comenzi Linux, Crearea proceselor, Comunicare între procese, Comunicarea între procese prin PIPE și FIFO, Comunicarea între procese prin semnale, Comunicarea între procese prin sistem V IPC: cozi de mesaje, semafoare, Interfață SOCKET, Modelul client / server TCP, Modelul client / server UDP)
10. Distribuții Linux (Particularități și caracteristici: REDHAT, DEBIAN, SUSE, LYCORIS, SLACWARE, MANDRAKE)
11. Sisteme de operare Windows (Versiuni, tipuri, caracteristici, instalare și configurare, performante)
12. Testare, experimente și comentarii asupra unei distribuții Linux (Prezentarea componentelor specifice și a modului de instalare, configurare, etc.).
13. Limbajul Shell scripting Unix/Linux (Instructiuni, structuri de control, exemple și aplicații)
14. Limbajul de lucrări (Batch) sub DOS (Instructiuni, structuri de control, exemple și aplicații)

## **Subiecte tip B. Concepție și dezvoltare SO**

1. Planificarea procesoarelor (UC) (schema generală, criterii de performanță, algoritmi de planificare UC: Algoritmul FCFS-First Come First Served, Algoritmul SJF-Shortest Job First, Algoritmul Round-Robin, alți algoritmi de planificare)
2. Gestiunea proceselor (Stare, Comutare, Crearea și terminarea proceselor, Procese și Thread-uri în UNIX/Linux, Procese și Thread-uri în Windows)
3. Comunicații și sincronizare între procese (Invalidarea/validarea intreruperilor, Instrucțiunea Test and Set (TS), Protocole de așteptare în excluderea mutuală, Mecanisme de sincronizare între

procese (obiecte de sincronizare), apariția interblocării (deadlock), Graful de alocare a resurselor, Rezolvarea problemei interblocării, procese cooperante)

4. Probleme clasice de coordonare și sincronizare procese (Problema producător-consumator: metoda semafoarelor, metoda transmiterii de mesaje, Problema bărbierului somnoroș, Problema cinei filozofilor chinezi, Problema rezervării biletelor, Problema grădinii ornamentale, Problema emițător-receptor)

5. Gestiuinea memoriei (Ierarhii de memorie, încarcarea și executia unui program: Încărcarea dinamică, Overlay-uri, Legarea dinamică, Alocarea memoriei în limbaje de programare, Scheme de alocare a memoriei: Alocare unică, Alocare cu partitii fixe (alocare statică), locare cu partitii variabile, Alocare prin swapping)

6. Paginarea memoriei (Implementarea tabelei de pagini, Segmentarea memoriei, Segmentarea paginată, Memorie virtuală, Algoritmi de înlocuire a paginii: Algoritmul FIFO, Algoritmul LRU (Least Recently Used), Algoritmul LFU (Least Frequently Used), Algoritmul RealPaged Daemon)

7. Gestiuinea memoriei în Linux și Windows - Alocarea spațiului liber de memorie (Alocatorul cu hărți de resurse, Alocatorul cu puteri ale lui doi (metoda camarazilor), Alocatorul Fibonacci, Alocatorul Karel-Mckusick, Alocatorul slab)

8. Mecanisme și tehnici de protecție SO (Criptografie: Criptografia cu chei secrete (Criptografia simetrică), Criptografia cu chei publice (Criptografia asimetrică), Sisteme de încredere: Monitorul de referință, Modelul Biba)

9. Nucleul Unix/Linux (Rol, componente, administrare sistem de fisiere, administrare memorie, planificare și execuție job-uri, Comenzi interne shell, Redirectări și conducte)

Conf. Dr. M. Vlada

<http://www.unibuc.ro/user/marin.vlada/>

## Elemente introductive

**Definition. Assembler:** Assembly language is the uncontested speed champion among programming languages. An expert assembly language programmer will almost always produce a faster program than an expert C programmer.

*,Machines have so much memory today, saving space using assembly is not important. If you give someone an inch, they'll take a mile. Nowhere in programming does this saying have more application than in program memory use. For the longest time, programmers were quite happy with 4 Kbytes. Later, machines had 32 or even 64 Kilobytes. The programs filled up memory accordingly. Today, many machines have 32 or 64 megabytes of memory installed and some applications use it all. There are lots of technical reasons why programmers should strive to write shorter programs, though now is not the time to go into that. Let's just say that*

*space is important and programmers should strive to write programs as short as possible regardless of how much main memory they have in their machine.”*

**Source:** The Art of Assembly Language Programming, Spring 2008, Yale University,  
<http://flint.cs.yale.edu/cs422/doc/art-of-asm/pdf/> (pdf)

**Ex.: prog1.asm**

```
=====
.model small
.stack
.data
    a db 00110000b
.code
start:
mov ax,@data
mov ds,ax
    mov bl,01010000b
    mov al,a
    not al      ;AL=11001111b
    mov al,a
    and al,bl ;AL=00010000b
    mov al,a
    or al,bl ;AL=01110000b
    mov al,a
    xor al,bl ;AL=01100000b
mov ah,4ch
int 21h
end start
=====
```

### TextBook vs. Tablet PC

The „Smart education” Project: 2015. S. Korea to digitise school books | Coreea de Sud schimbă manualele cu Tablete PC

The South Korean government has said it plans to digitise all textbooks for elementary, middle and high school students by 2015. This plan for „smart education” is aimed at helping students create their own study pattern, and lighten their backpacks. Source: <http://www.channelnewsasia.com/>

„Ministerul Educației, Științei și Tehnologiei din Coreea de Sud a anunțat un plan pentru migrarea de la manualele școlare clasice, pe suport de hârtie, la manuale în format electronic. Ministerul a alocat proiectului suma de aproximativ două miliarde de dolari, iar tranzitia urmează să se finalizeze în 2015, când se estimează că toți elevii din învățământul primar vor beneficia de noile manuale, precum și de dispozitive hardware corespunzătoare, tablet PC-uri sau telefoane smart. În următorii ani autoritățile trebuie să se asigure că toate materialele didactice vor fi digitalizate. Toți elevii sud coreeni vor primi gratuit tabletele digitale, iar manualele vor fi disponibile pentru descărcare de pe serverele ministerului, prin intermediul rețelelor WiFi locale.” Sursa: <http://portal.edu.ro/index.php/articles/11512>

**Tablet PC** – calculator portabil al cărui ecran (de regulă cu o diagonală de 12 inchi) îndeplinește o funcție dublă: cea de afișare a informației și interfață de manipulare a

calculatorului (de obicei prin intermediul unui stylus -unealtă de scris sub forma unei mini-baghetă din material plastic sau metal având la vârf o bobîță din plastic, folosită pentru interacțiunea cu ecranele tactile rezistive, sau folosind degetele – touch). Primul tablet a fost lansat în 2001 de către Microsoft și folosea Windows XP Tablet PC Edition.

**Tipuri de Tablet PC:** a) tip broșură ; b) tip placă; c) tip decapotabil; d) tip hibrid.



### Mobile Technology and Communications

**Topics:** Mobile Learning (m-Learning), Mobile Network Technology, Mobile software, Mobile Internet, Mobile dating, Health impact, Mobile phone features

*„M-learning” is the follow up of E-learning and which originates from D-learning (distance learning). M-learning is the delivery of education to the students who are not having fixed location or who prefer to use mobile phone technology for learning. The rapid growth in the mobile and communication sector make it possible to develop new forms of education. M-learning means delivery of education by means of the mobile phone devices, PDAs and audio players. M-learners seek the lessons in the small format.” Ref.: <http://www.networktutorials.info/>*

### Mobile Technologies

HARDWARE:iPad, iPhone, Smartphone, Tablet PC. SOFTWARE: Operating Systems – iOS, Android, Windows Phone 7, BlackBerry, Chrome OS tablet, WebOS, MeeGo

Companies (Manufacturers): Apple, Google, Microsoft, Research in Motion (RIM) Mobile Services and Apps (Apple, Google, Microsoft): iOS, Android, BlackBerry, Windows Phone7. Ref.: [www.agora.ro/conferinta/programatica-2011-mobile-services-and-apps](http://www.agora.ro/conferinta/programatica-2011-mobile-services-and-apps)

### Tehnologia secolului 21

**IBM PureSystems** – sistemele de calcul noi de tip „sisteme expert integrate”.

- Design „Scale-In”: Integrează servere, sisteme de stocare și rețele de calculatoare.
- Modele de expertiză: IBM încorporează expertiză tehnologică și industrială prin intermediul aplicațiilor software.
- Integrare cu mediul „Cloud”: Sistemele din familia PureSystems sunt create pentru mediul „Cloud Computing”.

**Detalii:**

---

<http://www.agora.ro/stire/ibm-lanseaza-o-noua-era-a-sistemelor-de-calcul>  
<http://www-03.ibm.com/press/us/en/presskit/37378.wss>

---

## Cloud computing Microsoft Azure Platform

*The Windows Azure Platform* is a Microsoft cloud platform used to build, host and scale web applications through Microsoft datacenters. Windows Azure Platform is thus classified as platform as a service and forms part of Microsoft's cloud computing strategy, along with their software as a service offering, Microsoft Online Services.

The Windows Azure Platform provides an API built on REST, HTTP and XML that allows a developer to interact with the services provided by Windows Azure. Microsoft also provides a client-side managed class library which encapsulates the functions of interacting with the services. It also integrates with Microsoft Visual Studio so that it can be used as the IDE to develop and publish Azure-hosted applications. Windows Azure became commercially available on 1 Feb 2010.

*„Running applications on machines in an Internet-accessible data center can bring plenty of advantages. Yet wherever they run, applications are built on some kind of platform. For on-premises applications, this platform usually includes an operating system, some way to store data, and perhaps more. Applications running in the cloud need a similar foundation. The goal of Microsoft's Windows Azure is to provide this.”*

## Sharding with SQL Azure

Database sharding is a technique of horizontal partitioning data across multiple physical servers to provide application scale-out. SQL Azure is a cloud database service from Microsoft that provides database functionality as a utility service, offering many benefits including rapid provisioning, cost-effective scalability, high availability and reduced management overhead. SQL Azure combined with database sharding techniques provides for virtually unlimited scalability of data for an application.

Source: <http://www.microsoft.com/windowsazure/whitepapers/>

Case Studies: <http://www.microsoft.com/windowsazure/evidence/>

## Premieră în 2011 pentru România:

Modulul de publicare a informațiilor pe Internet (Sistemul informatic pentru examenele naționale – realizat de Siveco Romania) a folosit platforma de **Cloud computing Microsoft Azure**, ceea ce asigură disponibilitate permanentă, capacitate de calcul practic infinită, redundanță și securitate a informațiilor. Acest modul a fost conceput special pentru a face față atât unui număr foarte mare de accesări simultane, cât și posibilelor atacuri menite să îintrerupă disponibilitatea serviciilor furnizate.

Conform [www.trafic.ro](http://www.trafic.ro), pe 12 iulie 2011 s-a înregistrat un nou record de vizitatori unici pentru <http://portal.edu.ro>, cu 828.935 într-o singură zi și peste 24 de milioane de afișări.

Cele mai accesate site-uri ale portalului au fost <http://admitere.edu.ro>, <http://bacalaureat.edu.ro> și <http://titularizare.edu.ro>. Sursa: <http://portal.edu.ro/index.php/articles/news/11492>.

## Despre primele calculatoare electronice

### Primele in lume

- **1946:** primul calculator electronic pe scara larga, de uz general, complet operational, [ENIAC](#) (Electronic Numerical Integrator and Calculator), finantat de armata SUA, utilizat la calculul tabelelor balistice de artilerie, proiectarea bombei cu hidrogen etc.; La 30 iunie 1945 se publica celebrul raport al lui John von Neumann intitulat First Draft of a Report on the EDVAC (EDVAC – Electronic Discrete Variable Automatic Computer), Moore School of Electrical Engineering, care contineea 43 de pagini. John von Neumann – stralucit matematician – este atras inca din anul 1944 la proiectul ENIAC.

**John von Neumann Page at Brown University**  
Ref.: [http://www.brown.edu/Research/Istrail\\_Lab/von\\_neumann.php](http://www.brown.edu/Research/Istrail_Lab/von_neumann.php)

John Von Neumann: The Scientific Genius Who Pioneered the Modern Computer, Game Theory, Nuclear Deterrence, and Much More.

John von Neumann is widely regarded as the greatest scientist of the 20th century after Einstein. Born in Budapest in 1903, John von Neumann grew up in one of the most extraordinary of scientific communities.

- **1949:** calculatorul EDSAC (Electronic Delay Storage Automatic Computer) primul calculator electronic complet echipat, operational, cu programe memorate
- **1951:** calculatorul UNIVAC I, primul calculator electronic comercial de mare succes, derivat din BINAC. Costa 250 000 \$, s-au construit 48 de sisteme!
- **1952:** primul calculator comercial [IBM 701](#) Electronic Data Processing Machines

### Primele in Romania

- **1957:** [CIFA 1 primul calculator romanesc](#), de la Bucuresti, realizat la Institutul de Fizica al Academiei, Magurele (ing. Victor Toma); Romania este a 8-a tara din lume care construieste un asemenea calculator si a doua dintre fostele tari socialiste, dupa fosta URSS [Draganescu]. Au urmat: CIFA-2 cu 800 de tuburi electronice (1959), CIFA-3 pentru Centrul de calcul al Universitatii din Bucuresti (1961), CIFA-4 (1962).
- **1961:** calculatorul MECIPT – Masina Electronica de Calcul a Institutului Politehnic Timisoara; Aplicatii (calcule pentru rezistenta) realizeate pe MECIPT-1 [Farcas] in perioada 1961-1964: proiectarea cupolei pavilionului expozitional Bucuresti, actual Romexpo (acad. D. Mateescu, programator ing. V. Baltac); proiectarea barajului Vidraru de pe Arges (18 zile in loc de 9 luni manual)

- 1963: [DASICC](#) calculator realizat de Institutul de Calcul al Academiei, Filiala Cluj-Napoca

Ref.: [MAESTRI AI INGINERIEI CALCULATOARELOR](#) (Lucian N. Vintan ), Universul ingineresc nr. 16/2007 – <http://www.agir.ro>

## Apariția și evoluția sistemelor de operare

**Text apărut în Marin Vlada (coord.), Istoria informaticii românești. Apariție, dezvoltare și impact, vol. I Contexul international, Ed. MATRIXRO, 2019, pag. 307-326**

**MOTTO:** - „I'm a bit like an operating system really... „multitasking” between these two OHPs, „interrupting” you, ... „sending you to sleep ...”

John Bates

- „It would appear that we have reached the limits of what is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years.”

John von Neumann, 1949, Wobbe Vegter,  
<http://wvegter.hivemind.net/abacus/CyberHeroes/Neumann.htm>.

Astăzi, un curs universitar care tratează domeniul *sistemelor de operare* poate avea următorul conținut (M. Vlada, *Sisteme de operare*, Universitatea din Bucuresti, <http://prof.unibuc.ro/2018/02/info/>):

1. STRUCTURA SISTEMELOR DE OPERARE: Evoluția sistemelor de operare; Structura și funcțiile generale ale sistemului de operare; Serviciile Sistemului de Operare; Apeluri Sistem (*System Calls*); Mașini Virtuale;
2. ADMINISTRAREA UNITĂȚILOR: Organizarea sistemului de I/O, Administrarea directă a I/O cu testare periodică (*polling*), Administrarea operațiilor de I/O orientată pe întreruperi, Proiecția în memorie a I/O, Accesul direct la memorie, Utilizarea zonelor tampon (*buffering*)
3. GESTIUNEA MEMORIEI: Memoria fizică, memoria virtuală, alocarea statică a memoriei, alocarea dinamică a memoriei, alocarea paginată a memoriei, alocarea segmentată a memoriei, algoritmi de paginare; algoritmi de înlocuire a paginii; gestiunea memoriei în sistemele Unix și Windows;
4. SISTEMUL DE FIȘIERE: Conceptul de fișier, Organizarea fișierelor, conceptual de director, alocarea spațiului pentru fișiere pe disc, evidența spațiului liber de pe disc, operații cu fișiere, implementarea sistemului de fișiere, performanțele sistemului de fișiere, fiabilitatea sistemului de fișiere, gestiunea blocurilor libere, Protecția și securitatea datelor (sistem de autentificare, sistem de autorizare, protecția resurselor, protecția memoriei);
5. GESTIUNEA PROCESELOR: Stările unui proces, paralelismul proceselor, sincronizarea proceselor, comunicarea între procese, probleme clasice de coordonare a proceselor, Planificarea

proceselor, interbloarea proceselor, fire de execuție, schema generală de planificare; criterii de performanță (algoritmi de planificare, algoritmul *Round Robin*), diagrama stărilor unui proces (procese și thread-uri, comutarea proceselor, procese Unix și Windows).

**Definition.** *Operating System (OS): the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.*

**List of Operating Systems** ACM<sup>1</sup>: Page created: 2004-04-03) – Operating Systems (611) , Linux Distributions (661) – Access date: 2019-05-20

Evidențiem numărul mare de sisteme de operare ce s-au elaborat până astăzi, de la începutul apariției calculatoarelor electronice. Lista cuprinde toate tipurile și derivatele *sistemelor de operare* (SO): *comerciale, gratuite, open source și surse închise*. Această listă nu distinge între etapa recentă de dezvoltare, scopul aplicației, distribuția sau platforma hardware. *Sistemele de operare* cu diferite numere de versiuni sunt luate în considerare doar dacă sunt distincte din punct de vedere tehnic unul față de celălalt.

Această listă (*Operating Systems-611, Linux Distributions-661*) oferă sute de nume de produse de produs și de proiecte ale sistemului de operare, multe dintre ele fiind bazate pe același sistem de operare cu mai multe sau mai puține diferențe în codul sursă. Cele mai multe sisteme de operare pot fi urmările până la câteva linii de dezvoltare și provin dintr-o cantitate mică de software de sistem. Acest lucru se aplică în special distribuțiilor *Linux* cu un număr redus de distribuții principale și un număr mare de instrumente derivate. Alte familii de sisteme de operare sunt *Windows* și *Unix*.

### Principalele etape în dezvoltarea sistemelor de operare

Principalele etape în dezvoltarea *sistemelor de operare* ([https://ro.wikipedia.org/wiki/Sistem\\_de\\_operare](https://ro.wikipedia.org/wiki/Sistem_de_operare)):

- **Anii 1950** – În această perioadă nu existau *limbaje de programare*, iar echipamentele electronice nu aveau *sisteme de operare*, fiind capabile să ruleze doar un singur program, utilizatorul fiind responsabil pentru interacțiunea directă cu *hardware-ul*. Până în anii '50, se dezvoltă prima generație de calculatoare, având la bază *tehnologia tuburilor vidate*. O inovație majoră este introducerea *cartelelor perforate*, ca modalitate de stocare a informației. Dezvoltatorul ideii de *sistem de operare*, care să sintetizeze operațiile unui computer, și rularea simultană a mai multor funcții ale unui program, a fost matematicianul englez *Alan Turing*, a cărui „*Turing machine*” este considerată precursorul computerelor.
- **Anii 1960** – În această perioadă, diferite companii producătoare de mașini de calcul au dezvoltat mai multe sisteme de operare: *Control Data Corporation, Burroughs Corporation, IBM, UNIVAC, General Electric, Digital Equipment Corporation*. Acestea au revoluționat conceptul de sistem de operare, introducând fiecare noiuni și caracteristici noi. Sistemele de operare au evoluat în multiprogramare, în care mai multe programe puteau rula în memorie, capacitatea de a comunica în rețea sau de a distribui mai multor utilizatori accesul la un program sau chiar nivele inelare de securitate. În anul 1961, *Burroughs Corp.* a introdus seria de calculatoare B5000, cu sistemul de operare *MCP (Master Control Program)*, primul sistem de operare care a fost scris exclusiv în *ESPOL*, un limbaj de nivel înalt, dialect al limbajului ALGOL. MCP a introdus de

<sup>1</sup> [https://www.operating-system.org/betriebssystem/\\_english/os-liste.htm](https://www.operating-system.org/betriebssystem/_english/os-liste.htm)

asemenea și prima implementare comercială de memorie virtuală. Sistemul de operare MCP este încă în utilizare și astăzi în linia de calculatoare Unisys ClearPath/MCP. *Atlas Supervisor* care a funcționat pe calculatorul *Atlas*, construit la *Universitatea Manchester* în 1962, a fost sistemul de operare care a reușit să aloce resursele de procesare ale calculatorului astfel încât acesta să poată acționa simultan pe mai multe sarcini și programe de utilizator. Diferitele sale funcții includ administrarea memoriei virtuale a computerului *Atlas*. Este considerat ca fiind primul sistem de operare modern recunoscut. UNIVAC, primul producător comercial de computere, a produs în anul 1962 sistemul de operare EXEC, pentru seria de calculatoare UNIVAC 1100/2200. Acesta a fost un sistem orientat pe loturi care putea opera cu discuri, cititoare de carduri și linii de imprimante. *General Electric* și *MIT* au dezvoltat GECOS (*General Electric Comprehensive Operating System*), care a introdus conceptul de nivele de privilegii securizate pe inele. După achiziționarea acesteia de către Honeywell, a fost redenumit în *General Comprehensive Operating System* (GCOS). *Control Data Corporation* (CDC), introduce sistemul de operare SCOPE pentru seria de calculatoare CDC 3000, CDC 6000, și CDC 7600. *Digital Equipment Corporation* a dezvoltat mai multe sisteme de operare de partajare pe timpi pentru liniile sale de calculatoare PDP-10 pe 36 biți, inclusiv TOPS-10 și TOPS-20, lansate în 1962 și 1969. TOPS-10 a fost un sistem deosebit de popular în universități și în comunitățile ARPANET.

În anul 1964, firma IBM a dezvoltat *System/360*, o familie de computere disponibile pe scară largă, cu diferite capacitați. Pentru acestea a fost dezvoltat *OS/360* (în perioada 1967-1968 România a achiziționat câteva astfel de sisteme de calcul-de la filiala IBM de la Viena, cu care a dotat universități, institute și uzine mari), primul sistem de operare capabil să lucreze la nivelul mai multor echipamente, centralizând astfel comanda unei întregi linii de procesare. De asemenea, sistemul *OS/360* a deservit computerele NASA în misiunea Apollo 11. Principalele sisteme de operare folosite astăzi de IBM sunt descendente ale OS/360. În România, primul sistem de operare a fost elaborat în 1968, pentru calculatorul electronic DACICC-200, construit la *Institutul de Calcul „Tiberiu Popoviciu”* (<https://ictp.acad.ro/>).

- **Anii 1970** – În această perioadă, microprocesoarele, cipurile și celealte componente hardware, au devenit suficient de mici pentru a încăpea într-un calculator de birou, aşa-numitul *desktop*. Cele mai folosite sisteme de operare erau *Microsoft MS-DOS*, ce putea fi instalat pe IBM PC și pe calculatoarele cu procesorul Intel 8088, și *UNIX* care putea rula pe procesoarele *Motorola 6899*. În cooperare cu *Universitatea din Minnesota*, pe parcursul anilor 1970, *Control Data Corporation* a dezvoltat sistemele de operare *CDC Kronos* și apoi *NOS (Network Operating System)*, care suportau prelucrări simultane pe loturi și partajarea pe timp. Ca multe sisteme de operare comerciale ce foloseau partajarea pe timp, interfețele acestor sisteme de operare erau o extindere a sistemelor de operare *Dartmouth BASIC*.

La sfârșitul anilor 1970, *CDC* și *Universitatea din Illinois* au dezvoltat sistemul de operare *PLATO*, ce folosea monitoare cu plasmă și partajarea pe timp în rețelele de distanță lungă. Multe concepte moderne în computerele multi-utilizator au fost inițial dezvoltate pe *PLATO*, inclusiv *forum-uri*, *panouri de mesaje*, *e-mail*, *mesagerie instantanee*, *partajare de ecran de la distanță și jocuri video multiplayer*. La mijlocul anilor 1970, sistemul de operare MVS (*Multiple Virtual Storage*) folosit pe calculatoarele IBM, *System/370* și *System/390*, oferea prima punere în aplicare a folosirii RAM-ului ca o *memorie cache transparentă* pentru date. *CP/M (Control Program/Monitor apoi redenumit Control Program for Microcomputers)*, un sistem de operare creat de *Digital Research*, în anul 1974, pentru microcompterele *Intel 8080*. Sistemul de operare *CP/M* a fost un sistem de operare compatibil cu foarte multe echipamente microcomputer, care a stat la baza *Microsoft MS-DOS* și *IBM PC DOS*. Spre sfârșitul anilor 1970, compania *Apple* a lansat sistemul de operare *Apple DOS* pentru seria de computere *Apple II*. *Apple DOS* a avut trei versiuni importante: DOS 3.1, DOS 3.2 și DOS 3.3.

- **Perioada modernă de după anul 1980** – Una dintre cel mai notabile inovații aduse în domeniul sistemelor de operare a fost introducerea de către *Apple* în 1984 a *Interfeței grafice*, conceptul de cartografiere a pixelilor ecranului și convertirea acestor date în informații digitale capabile să comunice direct cu tastatura și cu *mouse-ul*. În aceeași perioadă, *Microsoft* a introdus noțiunea de ferestre (*Windows*), iar *Linus Torvalds* aduce *kernel-ul*, care a oferit posibilitatea dezvoltării de tip sursă deschisă a ceea ce acum poartă numele de distribuții *Linux*.

În anul 1996 *Palm, Inc.* a lansat primul sistem de operare pentru dispozitivele mobile, numit *Palm OS*. Tot în același an, *Windows* prezintă platforma *Windows Mobile*, disponibilă pe majoritatea dispozitivelor de tip *PDA* din acea vreme. *Nokia* prezintă în anul 1999 primul sistem de operare *Symbian S40*, pe un *Nokia 7110*. Un an mai tarziu, *Symbian OS* a devenit cel mai popular sistem de operare mobil, instalat pe majoritatea dispozitivelor *Nokia*, cât și pe telefoanele *Ericsson*.

Începând cu anul 2007 lumea dispozitivelor mobile s-a schimbat radical odată cu apariția *iOS* și *iPhone*. Acum, utilizatorii telefoanelor mobile puteau naviga ușor printr-un meniu prietenos pe ecranul primului *iPhone*. La 5 noiembrie 2007 a fost înființată OHA (*Open Handset Alliance*), o alianță comercială de 84 de companii condusă de *Google*, dedicată dezvoltării standardelor deschise pentru dispozitive mobile. În anul 2008, această alianță a lansat *Android*, care peste câțiva ani va deveni cel mai răspândit sistem de operare mobil.

În prezent, *Windows* a reușit să ajungă cel mai popular sistem de operare. Studiile recente arată că 91% dintre posesorii de calculatoare dețin un sistem de operare de la *Microsoft*, pe când 5% un sistem de la *Apple* (Mac), iar mai puțin de 2%, mai exact 1,74% folosesc *Linux*.

### **Pionierate în domeniul elaborării sistemelor de operare**

Pionierate în domeniul elaborării *sistemelor de operare*:

1. **Compania americană IBM** – IBM Corporation a fost fondată cu numele *Computing Tabulating Recording Co.* (CTR) din New York, la 15 iunie 1911. După unirea companiilor *Tabulating Machine Corp.*, *Computing Scale Corp.* și *Compania Internațională de Înregistrare a Timpului*, CTR a fost apoi redenumit în *International Business Machines*, în anul 1924. IBM a dezvoltat *Sequek* în anul 1974, iar 2 ani mai târziu a fost redenumit în *SQL (Structured Query Language)*. Publicațiile despre modelul de relaționare al lui *E. F. Codd* au stat la baza acestei dezvoltări (Detalii: [https://www.operating-system.org/betriebssystem/\\_english/fa-ibm.htm](https://www.operating-system.org/betriebssystem/_english/fa-ibm.htm)). Costul pentru dezvoltarea sistemului de operare OS/360 a fost de 50 de milioane de dolari și are mai mult de 220.000 de linii de cod. Sistemul de operare OS/400 a fost redenumit de la lansarea V5R3 în i5/OS. De exemplu, acest sistem de operare este folosit la *eServer i5* cu procesoare *Power PC*. Noul sistem IBM System/360 este un sistem computerizat mai eficient ca și computerele anterioare IBM, care sunt construite în mod diferit. Componentele hardware (cum ar fi cititorul de cartele/carduri) erau interschimbabile, iar software-ul poate funcționa pe toate modelele familiei de produse. Dezvoltarea sistemului IBM/360 a costat 5 miliarde de dolari și a angajat în timp record peste 50.000 de angajați. Acest computer a fost fabricat pentru prima dată cu o linie de asamblare.

*Software products* – WebSphere application server; DB2 universal database; WebSphere personalization for Multiplatforms; content manager for Multiplatforms; WebSphere home page builder; Small business suite for Linux; e-collaboration; e-learning; e-Knowledge management; ViaVoice speech recognition; VisualAge for Java; Lotus ASP Solution Pack; MVS/370 - OS/360, OS/390, OS/400; z/OS (formerly OS/390 ); i5/OS (formerly OS/400); OS/2 operating system (Operating system 2).

Cele mai importante *sisteme de operare* dezvoltate:

- **MVS, OS/390, z/OS operating system**

MVS (*Multiple Virtual Storage*) este sistemul de operare dezvoltat de IBM în anii șaizeci și sprijină POSIX. MVS a fost dezvoltat mai departe de sistemul de operare MVT și a fost extins de multe funcții, cum ar fi utilizarea memoriei virtuale. Este folosit în mainframe IBM ca sistemul/370 și system/390, are nevoie de cerințe minime de hardware și convinge prinț-o stabilitate ridicată în modul de funcționare pe termen lung. Aceasta a fost utilizat în principal pentru furnizarea de resurse pentru conexiunile terminale. JCL (Job Control Language) este utilizat pentru prelucrarea lotului, în cadrul TSO/E (opțiunea de împărțire a timpului/Extended) servesc la primirea comenziilor. Ca o aplicație de dialog ISPF (*Interactive System Productivity Facility*) ajută la dezvoltarea propriei programări și documentare a aplicațiilor de dialog și a loturilor. La început, a fost un sistem de operare pe 16 biți și a fost dezvoltat în continuare pentru procesarea pe 32 de biți. Cu versiunea MVS/ESA, adresabilitatea pentru memoria RAM și hard disk a fost extinsă. După redenumirea MVS în OS / 390, suportul TCP / IP a fost adăugat în versiune. Acest sistem de operare se numește acum z/OS, iar caracterul său special „z” de la începutul „z/OS” reprezintă buna aderevare a serverelor IBM ale zSerie precum z900. Acum, acesta oferă și adresarea pe 64 de biți a memoriei.

*Sursa:* [https://www.operating-system.org/betriebssystem/\\_english/bs-mvs.htm](https://www.operating-system.org/betriebssystem/_english/bs-mvs.htm)

- **Versions Date** – Version: 1964 April – IBM System /360 (OS/360); 1967 – IBM System /360 (OS/MFT); 1968 – IBM System /360 (OS/MVT); 1972 – IBM System /370 (SVS1 und SVS2); 1974 – IBM System /370 (MVS); 1981 – IBM System /370 (MVS/XA); 1985 – IBM System /370 (MVS/ESA); 1990 – IBM System /390 (MVS/ESA); 1996 – IBM System /390 (OS/390); 2001 March – z/OS 1.0; 2003 Feb. – z/OS 1.4; 2004 March – z/OS 1.5; 2004 Sept. – z/OS 1.6; 2006 Jan. – z/OS 1.7

#### **- Sistemul de operare OS/2 Warp**

Principalele caracteristici ale OS/2 (Sistemul de operare 2) sunt, în primul rând, interfața cu utilizatorul WPS (Workplace Shell), stabilitatea și tehnologia care conduc în anii anteriori ai OS/2. Interfața cu utilizatorul este construită complet orientată spre obiect. Sistemul OS/2 al IBM nu trebuie să fie comparat cu extensiile DOS sau Windows încă de la versiunea 2.0. Această nouă versiune corespunde unei noi generații de sisteme de operare, care are potențialul de a utiliza performanța completă a unui procesor pe 32 de biți dezvoltat numai de IBM. Până la versiunea 1.3, IBM a colaborat la dezvoltarea cu Microsoft. Versiunea beta a fost testată cu 30.000 voluntari. A fost făcută sub slogan pentru a crea un succes "mai bun decât DOS". Multe programe pentru DOS și Windows (3.x, Win32s) sunt executate mai repede în OS/2 decât în mediul de operare inițial.

*Sursa:* [https://www.operating-system.org/betriebssystem/\\_english/bs-os2.htm](https://www.operating-system.org/betriebssystem/_english/bs-os2.htm)

- **Date** – Version: 1987 – OS/2 1.0 16-Bit; 1988 – OS/2 1.1 16-Bit, with graphical user interface Presentation Manager (PM); 1989 – OS/2 1.2 16-bit for the first time HPFS in, improved Presentation manager; 1991 – OS/2 1.3 16-bit; 992 March – OS/2 2.0 new design, optimized Presentation Manager, protected memory area, up to 240 simultaneous DOS Sessions possible, 32-bit, high compatibility to OS/2 1.x applications, DOS 2.x – 5.x, Windows 2.x – 3.x, "Cut and Paste", new Workplace Shell (WPS), supports LAN Server 2.0, extended Services 2.0; 1994 Jan. – OS/2 2.1 Graphic subsystem optimized, dual DOS-sessions, APM and PCMCIA support; 1994 Oct. – OS/2 Warp 3.0 networkable, P2P; 1996 – OS/2 Warp Server Combination of Warp 3.0 aIBM's LAN Server 4.0, server services, remote login; 1996 Oct. – OS/2 Warp 4.0 (merlin) Integration of the pentium III instructions, improved Plug and Play and multimedia, includes Java and VoiceType technology, universally network client; 1999 – OS/2 Warp 4.5, bootable cd-rom, new 32-bit TCP/IP stack, JFS; 2000 – Version 4.51 version which is completely improved and revised of version 4, "Convenience Pak 1"; 2002 – Version 4.52, "Convenience Pak 2", new functions and bug fixes; 1999 – Warp Server 4.5 (Aurora) for e-Business, Netfinity Manager, Java

1.1.6 JDK, new 32-bit kernel, SMP for up to 64 CPUs, JFS (64 bit Journaling File System), optimized TCP/IP protocol.

- **Sistemul de operare AIX 4.3**

AIX is the first 64-bit UNIX that becomes from the NSA in the USA the TCSEC C2 certificate, with modifications it corresponds also the TCSEC B1. AIX 4.3 can run on 64-Bit CPUs binarily 32-Bit programs and 64-Bit programs. The TCP/IP stack and the I/O system were continued to optimize on high efficiency. Up to 128 non removable disks can be combined into a logical group. OpenGL GLX 1.3 and graPHICS extensions make an increased application performance and better handling of large graphic models possible. NIS+, Java support and numerous system management Tools supplement this AIX release.

- *Versions Date – Version:* 1986 – AIX 1; 1987 – AIX 2; 1989 – AIX 3; 1990 – AIX 3.1; 1993 Sept. – AIX 3.2.5; 1993 – AIX 4.0; 1994 July – AIX 4.1; 1994 Oct. – AIX 4.1.1, first use of the CDE desktop as AIX window desktop; 1994 – AIX 4.1.2; 1995, June – AIX 4.1.3, contains in parts of CDE 1.0; 1995, Oct. – AIX 4.1.4, max. file size up to 2 GByte and 2 GByte RAM, file system up to 64 Gbyte; 1996 – AIX 4.1.5, max. file size up to 2 GByte and 2 GByte RAM, file system up to 64 Gbyte; 1996, Oct. – AIX 4.2, full support of CDE 1.0, max. file size up to 64 GByte and 4 GByte RAM, file system up to 128 Gbyte; 1994 – AIX 4.2.5, BSI E3/F-C2 Security Certification; 1997, Oct. – AIX 4.3, BSI E3/F-C2 Security Certification; 1998, April – AIX 4.3.1, Certification B1/EST-X Version 2.0.1, max. file size up to 64 GByte and 16 GByte RAM, file system up to 1 Tbyte; 1998, Oct. – AIX 4.3.2, max. file size up to 64 GByte and 32 GByte RAM; 1999, Sept. – AIX 4.3.3, max. file size up to 64 GByte and 96 GByte RAM, file system up to 1 Tbyte; 2000 – AIX 5L; 2001 May – AIX 5L 5.1, up to 32 processors and 512 GByte RAM; 2002 Oct. – AIX 5L 5.2, up to 32 processors and 1024 GByte RAM; 2004 Aug. – AIX 5L 5.3, up to 64 processors and 2048 GByte RAM; 2007 July – AIX 6 OpenBeta; 2007 Nov – AIX 6.1; 2010 Oct. – AIX 7.1 released; 2015 Dec. – AIX 7.2 released

2. **Compania americană AT&T – American Telephone & Telegraph Corporation (AT&T)** are o lungă istorie și cu numeroase inovații. Compania s-a dezvoltat pornind de la fondatorul *Alexander Graham Bell*, inventatorul telefonului, în anul 1875. *Gardiner Hubbard* și *Thomas Sander* au finanțat compania. Ei au înființat compania *Bell Telephone* în anul 1879. Din istoria completă a companiei AT & T se evidențiază domeniile de activitate în industria de comunicare și electronică. Linia de afaceri pentru calculatoare apare mai târziu. De când a ajuns la un monopol, compania a trebuit să deschidă piața, alături de alți furnizori. Numărul companiilor de telefonia a urcat la aproximativ 6.000, între anii 1894 și 1904. Numărul de telefoane fabricate a crescut de la 285.000 la 3.317.000. În anul 1984 compania s-a divizat și a devenit grupul AT & T de la Departamentul de Justiție al SUA și alte două companii *Bell Labs* și *AT&T International*. Compania formată a scăzut cu 1/3 față de cea inițială. Detalii: [https://www.operating-system.org/betriebssystem/\\_english/fa-att.htm](https://www.operating-system.org/betriebssystem/_english/fa-att.htm).

**Sistemul de operare UNIX**

În anul 1965, existau calculatoare din a III-a generație a mașinilor de calcul cu circuite integrate. Pentru executia programelor erau utilizate *sisteme de operare offline* (de exemplu *IBM System/360*). Această generație a fost ulterior înlocuită de *sisteme de operare de partajare* (time-sharing) *online* îmbunătățite. Sarcini importante au fost prelucrarea automată a loturilor de job-uri în *spooler*, care a înlocuit sistemele bazate pe un lot (*job*) începând cu anul 1969. Unul dintre primii reprezentanți ai sistemelor de calcul cu distribuție de timp a fost CTTS de la MIT, reprogramat împreună cu *Bell Labs* și *General Electric* pentru sistemul de operare MULTICS. Sistemul de operare MULTICS a fost construit complex și programat în limbajul de programare PL/1, astfel încât a fost abandonat mai târziu. *Ken Thompson* a dezvoltat în anul 1971 (de la *Bell Labs*) o versiune de asamblare a lui MULTICS, care a fost numită UNICS (1969) și ulterior sistemul de operare UNIX. *Dennis Ritchie* nu era mulțumit de ce elaborare pana atunci, de aceea era nevoie de un limbaj de programare, care să ajute la

elaborarea sistemului de operare UNIX. *Thompson* a dezvoltat BCPL (o simplificare a CPL) și limbajul de programare B, care a fost reprogramat de *Ritchie* și s-a obținut limbajul C. În anul 1974 *Thompson* și *Ritchie* au elaborat și finalizat împreună sistemul de operare UNIX în noul limbaj de programare C – scris special pentru elaborarea de sisteme de operare. Detalii: [https://www.operating-system.org/betriebssystem/\\_english/bs-unix.htm](https://www.operating-system.org/betriebssystem/_english/bs-unix.htm).

- Date – Version: 1969 Sept. – UNICS (Firma AT&T); 1971 Nov. – Time-Sharing System v1; 1972 Dec. – Time-Sharing System v2; 1973 Feb. – Time-Sharing System v3; 1973 Nov. – Time-Sharing System v4; 1974 June – Time-Sharing System v5; 1975 May – Time-Sharing System v6; 1979 Jan. – Time-Sharing System v7; 1985 Feb. – Time-Sharing System v8; 1986 Sept. – Time-Sharing System v9; 1989 Oct. – Time-Sharing System v10; 1978 – CB UNIX 1; 1978 – CB UNIX 2; 1979 – CB UNIX 3; 1981 – UNIX System III; 1982 – UNIX System IV; 1983 – UNIX System V Rel. 0; 1984 – UNIX System V Rel. 2; 1986 – UNIX System V Rel. 3.0 (SVR3); 1987 – UNIX System V Rel. 3.2; 1988 – UNIX System V Rel. 4 (SVR4), coop. with Sun Microsystems; 1992 – UNIX System V Rel. 4.2 (SVR4.2); 1993 – UNIX System V Rel. 4.2 MP (SVR4.2MP); 1985 – Mach (Carnegie-Mellon University); 1986 – Mach 2.0; 1988 – Mach 2.5; 1990 – Mach 3 (base for Darwin); 1995 – Mach 4

### 3. Compania americană Digital Research (DR Corporation)

#### Sistemele de operare CP/M, DR-DOS

*Gary A. Kildall* a dezvoltat pentru compania *Intel* limbajul de programare PL/M pentru microprocesor *Intel 8008*, derivat din PL/I (anul 1973). În același an a dezvoltat sistemul de operare CP/M – *Control Program for Microprocessors* (*Programul de control pentru microprocesoare*) în PL/M. Acesta a fost primul sistem de operare pentru computerele bazate pe *microprocesoarele Intel*. Punctul forte al sistemului de operare CP/M este *portabilitatea ridicată* pe toate tipurile de configurații hardware. Funcțiile de bază au fost programate independent de hardware, în funcție de platforma vizată, caracteristica specială fiind programată suplimentar.

În anul 1976, împreună cu soția sa *Dorothy McEwen*, *Kildall* a înființat compania *DR Inc. (Digital Research Incorporation)*. La început, sistemul de operare CP/M a fost proiectat de compania DR numai ca un program de gestionare a fișierelor pure pentru computerul x86 pe 8 biți și vândut de *Intel*. În anul 1976 a existat un sistem de operare *CP/M Bios* pentru calculatoarele *Intel 8080*. În acel moment, CP/M era sistemul de operare dominat de pe piață și utilizat de majoritatea producătorilor de calculatoare. În anul 1981, zeci de tipuri de calculatoare au concurat sub diferite sisteme de operare, cum ar fi CP/M în numeroase variante. În plus, existau sisteme de operare cu mulți producători și multe versiuni UNIX. În acel an, sistemul de operare a fost utilizat aproximativ pe 200.000 de microcomputere în mai mult de 3.000 de configurații diferite. În anul 1985 sistemul de operare CP/M a fost utilizat în întreaga lume de aproximativ 4 milioane de ori în diferite versiuni. În anul 1988 sistemul de operare CP/M a fost redenumit în DR DOS după câteva lansări. Detalii:

[https://www.operating-system.org/betriebssystem/\\_english/bs-cpm.htm](https://www.operating-system.org/betriebssystem/_english/bs-cpm.htm),

[https://www.operating-system.org/betriebssystem/\\_english/fa-dr.htm](https://www.operating-system.org/betriebssystem/_english/fa-dr.htm).

- Date – Version: 1973 – CP/M 1.0 --- – CP/M 1.4; 1979 – CP/M 2.0 – CP/M 2.2, served as the base for MS/DOS; 1982 – CP/M 3.0, 8-bit operating system 1982 – CP/M 3.1, 16-bit operating system, DR DOS Basis; 1985 – DOS-Plus 1.0 was introduced; 1987 – DOS-Plus 1.2; 1986 – DOS-Plus 2.01; 1988, May – DR DOS 3.31 was introduced, first MS-DOS compatible version; 1988 – DR DOS 3.32; 1988 – DR DOS 3.40; 1989, Jan. – DR DOS 3.41 OEM Release; 1990, May – DR DOS 5.0 (leopard) ready, with Viewmax; 1991 – DR DOS 5.11; 1991, Sept. – DR DOS 6.0 ready, Superstor, Delwatch, NetWare Lite support, DOS prompt task switcher (Taskmax).

### 4. Compania americană Microsoft – *Paul Allen* și *Bill Gates* au mers la același școală și împreună, în anul 1971 au fondat compania *Traf-O-Data*. Obiectivul era producția de computere mici (tip PC) cu procesor *Intel* pentru măsurarea traficului auto. Au primit un contract de la MITS (*Micro Instrumentation und Telemetry Systems*) pentru a elabora limbajul BASIC pentru calculatorul *Altair*. În anul 1974 au proiectat limbajul de programare BASIC

pentru *Altair 8080*, echipa numindu-se *Programator Lakeside Group*. De asemenea, limbajele de programare FORTRAN și COBOL au fost scrise pentru *Altair*, iar interpretorul BASIC a fost portat pentru TRS 80 de la Tandy. În vara anului 1975 s-a fondat *Compania Micro-Soft* pentru a dezvolta software cu care să funcționeze IBM PC și, s-a pornit cu limbajul de programare BASIC. Mai târziu, a urmat începutul anului 1977, Fortran și Assembler, COBOL în 1978 și Pascal în anul 1980. În anul 1978, *Microsoft* a primit de la *AT&T* licență de dezvoltare pentru sistemul de operare UNIX. Deoarece AT&T a protejat numele sistemului de operare UNIX, Microsoft a numit varianta sa Unix prin denumirea Xenix.

În anul 1980, compania se mută (cu 38 de angajați) în Seattle și atinge volumul de vânzări de 8 milioane USD. În acea vreme compania IBM căuta un sistem de operare care să se potrivească cu microcompterele pentru piață. La început, IBM a consultat compania Digital Research, dar fără succes și apoi, compania Microsoft. Între timp, Microsoft a fost întărită de *Steve Ballmer* prin organizare și prin finanțare. Compania *Microsoft* nu avea nici un sistem de operare propriu și, prin urmare, a cumpărat sistemul Q-DOS de la *Seattle Computer Products* cu 50.000 de dolari SUA. Compania IBM a licențiat varianta sub numele de MS-DOS. PC-ul IBM a avut un mare succes. Volumul vânzărilor și profitul Microsoft au crescut în continuare și, prin urmare, *Bill Gates* a primit un contract cu SCO pentru a scoate o varianta UNIX pentru PC-urile IBM. În primul rând, cu apariția procesorului 80286, și folosind sistemul de operare Xenix s-a dovedit a fi foarte bine. În primul rând, Microsoft a avut grija de afacerea OEM, iar SCO a fost responsabilă pentru personalizarea și îmbunătățirea derivatului UNIX. În iulie 1987, Microsoft a cumpărat compania *Forethought* și a integrat software-ul sub numele de *Powerpoint* în aplicațiile de birou. Din anul 1994, sloganul „*Unde vrei să mergi astăzi?*” a fost răspândită prin publicitate. Detalii:[https://www.operating-system.org/betriebssystem/\\_english/fa-microsoft.htm](https://www.operating-system.org/betriebssystem/_english/fa-microsoft.htm).

Cele mai importante sisteme de operare dezvoltate:

#### - Sistemul de operare Microsoft DOS (MS-DOS)

În iulie 1980, compania IBM a comandat companiei Microsoft să dezvolte un sistem de operare pe 16 biți pentru calculatorul personal (PC) pentru un contract de 186.000 de dolari USA. Deși, compania *Digital Research* a lui *Gary Kildall* avea deja, prin sistemul CP/M 86 o astfel de versiune de 16 biți, dar prin anumite circumstanțe nu a fost stabilit niciun contract cu compania IBM. *Microsoft* nu avea încă niciun sistem de operare. *Microsoft* avea licență pentru CP/M de la *Digital Research*, din noiembrie 1977 pentru 50.000 de dolari. Întrucât, *Microsoft* nu a putut să vândă licență a fost realizat un acord corespunzător cu compania *Seattle Computer Products* pentru sistemul de operare QDOS. Sistemul QDOS era o clonă pe 16 biți a sistemului CP/M și a fost terminată de *Tim Paterson* în aprilie 1980. La început, *Microsoft* a acordat licență QDOS pentru 25.000 de dolari. După ce a fost semnat un acord de licență cu IBM, *Bill Gates* a cumpărat QDOS pentru 50.000 de dolari, în iulie 1981. Cum s-a dovedit a fi o afacere foarte profitabilă, compania IBM a livrat-o pe toate computerele IBM ca PC DOS. Sistemul de operare QDOS s-a utilizat pentru prima dată pe PC-ul IBM 5150, iar pentru toate celelalte, sub numele de MS-DOS -pentru partenerul OEM. Sistemul de operare MS DOS 1.0-versiunea 1, constă din codul de asamblare de aproximativ 4.000 de linii. Detalii:

[https://www.operating-system.org/betriebssystem/\\_english/bs-msdos.htm](https://www.operating-system.org/betriebssystem/_english/bs-msdos.htm).

- Date – Version: 1981 Aug. – MS-DOS 1.0, formerly QDOS/86-DOS, can use at maximum 128 kbyte RAM, FAT established; 1981 Juni – MS-DOS 1.10.; 1982 Aug. – MS-DOS 1.25, support for double-density floppy disks; 1983 March – MS-DOS 2.0, support for IBM 10 mbyte harddisk, directorys and DD 5.25" floppy disk drives with up to 360 kbyte; 1983 Dec. – MS-DOS 2.11, extended character sets; 1984 Aug. – MS-DOS 3.0, support for high density floppy disk drives with 1.2 mbyte and harddisk devices with 32 mbyte capacity; 1984 – MS-DOS 3.1, first time with network support; 1985 – MS-DOS 3.2; 1985 – MS-DOS 3.21; 1987 April – MS-DOS 3.3, extended for IBM's PS/2 computer, supports now bigger 3.5" harddisk drives and floppy disk drives, multiple partitions, character sets for different languages; 1988 Juli – MS-DOS 4.0, XMS support, partitions with up to 2 gbyte, graphical shell,

bug fixes; 1988 Nov. – MS-DOS 4.01, supports multiple partitions bigger than 32 mbyte, bug fixes; 1991 June – MS-DOS 5.0, Major Release; 1992 – MS-DOS 5.0a, bug fixes for Undelete and Chkdsk; 1993 Aug. – MS-DOS 6.0, Competition to Novell's DR-DOS 6, DoubleSpace, Anti-Virus program, Defrag, Move command, improved MSBACKUP and several boot configurations, memory optimizer MEMMAKER, DOS Shell is delivered separately on floppy disks; 1993 Nov. – MS-DOS 6.2, DoubleSpace becomes incompatible to the previous version, Scandisk, improved of DISKCOPY and SmartDrive; 1994 March – MS-DOS 6.21, because of law conflict with Stac Electronics DoubleSpace is removed from MS DOS; 1994 May – MS-DOS 6.22, Microsoft licences double disk of Vertisoft Systems and designates it in DriveSpace, last official standalone version; 1995 Aug. – MS-DOS 7.0, MS-DOS component for Windows 95, LFS support through VFAT, more DOS programs are delivered on the Setup CD-ROM in the "oldmsdos" directory; 1996 Aug.

MS-DOS 7.10, MS-DOS component for Windows 95 B and higher, supports the first time FAT 32 harddisks; 2000 – MS-DOS 8.0, MS-DOS component for Windows ME, last MS-DOS version; 2001 Dec. – MS-DOS Support discontinued

- **Sistemul de operare Windows® Family**

În anul 1981, compania *Microsoft* a început să dezvolte sisteme de operare pentru computere (PC), primul fiind MS-DOS 1.0. Cu un an înainte, *Microsoft* a lucrat cu sistemul de operare derivat Unix XENIX OS pentru diferite platforme informatiche, dar acest domeniu a fost transferat la SCO în anul 1984. Cu sistemul de operare Windows 1.0 au fost adăugate, în anul 1985, alături de DOS, o a doua linie OS, pentru job-uri unice pentru *Consumer* (ediția *Home*) și, mai târziu cu suport de rețea adăugat. A treia linie de produse a fost lansată cu MS OS / 2 1.0 în anul 1987. Ediția profesională a fost concepută pentru aplicații de servere și clienți de rețea. În februarie 1989 a început dezvoltarea *Windows NT* (NT = Noua tehnologie), prima versiune a fost publicată cu Windows NT 3.1 în iulie 1993. Până la 200 dezvoltatori au programat în același timp, la aprox. 6 milioane de linii de cod. În timp ce MS-DOS a fost programat aproape complet în limbajul de asamblare, sistemul de operare *Windows NT* constă, de asemenea, din codul sursă al *limbajului de programare C*. Până la 450 de dezvoltatori au fost implicați în sistemul de operare Windows NT 3.51, care a fost lansat în mai 1995. Pentru lansarea din iulie 1996 până la 800 dezvoltatori au lucrat la succesorul *Windows NT 4.0*. *Windows 2000* a fost proiectul ambițios de sistem de operare care a urmat în această privință. Până la 1.400 dezvoltatori au lucrat la cele 29 de milioane de linii de cod. Costurile de dezvoltare s-au ridicat la aproximativ 1 miliard de dolari. În total, 5.000 de dezvoltatori au lucrat la cele 50 de milioane de linii de coduri de asamblare C și C ++ pentru sistemul de operare *Windows Server 2003*, lansat în aprilie 2003. Dezvoltarea versiunilor sistemului de operare pentru arhitectura MIPS, *Power PC* și *alpha* a fost anulată treptat până la lansarea pe piata a sistemului de operare *Windows 2000*. Acest lucru a fost, de asemenea, implicat în lipsa driverului și a suportului software pentru aceste platforme. Cu *Windows CE 1.0* a fost creată o nouă linie de produse pentru dispozitive mici (PDA) în anul 1996. Detalii:

[https://www.operating-system.org/betriebssystem/\\_english/bs-windows.htm](https://www.operating-system.org/betriebssystem/_english/bs-windows.htm)

- *Date – Version:* 1981 Sept. – Interface Manager; 1983 – Windows was announced; 1985 Nov. – Windows 1.0 for 8088 CPUs released; 1987 Dec. – Windows 2.0 for 386 CPUs, up to 16 mbyte RAM adressable; 1988/89 – Windows 2.1 for 286/386; 1990 – Windows 3.0; 1992 April – 3.1 (janus) includes MS-DOS 5.0; 1993 Nov. – Windows for Workgroups 3.11 (snowball) rudimentarily networkable; 1995 Aug. – Windows 95 4.0 Codename "Chicago" was released; 1996 Feb. – Windows 95 Version A (OSR1); 1996 Aug. – Windows 95 (detroit) Version B (OSR2) first time with FAT-32 support; 1997 April – Windows 95 Version B (OSR2.1); 1997 Nov. – Windows 95 Version C (OSR2.5); 1998 June – Windows 98 (memphis) about 5,000 bugs fixed; 1999 May – Windows 98 SE (Second Edition); 2000 Sept. – Windows Me (Millenium); 1980 Aug. – XENIX OS cooperation with SCO; 1982 Feb. – XENIX 2.3 cooperation with SCO; 1983 April – XENIX 3.0 cooperation with SCO; 1987 – MS OS/2 1.0 cooperation with IBM; 1988 – OS/2 1.1 cooperation with IBM; 1991 – OS/2 1.3 cooperation with IBM; 1992 – OS/2 2.0 cooperation with IBM

1993 July – Windows NT 3.1 & Advanced Server; 3.1 million lines of source code, supports HPFS; 1994 Sept. – Windows NT 3.5 (daytona), Workstation and Server, with opengl and Netware client; 9 million lines of source code; 1995 May – Windows NT 3.51, updae, transparent compression with NTFS file system, PCMCIA support, for Power-PCs available too; 1996 Aug. – Windows NT 4.0 (cairo) 16 million lines of source code, no support for HPFS anymore; 1996 – Windows NT Terminal Server Edition (hydra); 1997 – Windows NT Server 4.0 Enterprise Edition; 1998 – Windows NT Server 4.0 Terminal Server Edition; 2000 Feb. – Windows 2000, Windows version 5.0; 30 million lines of source code, about 10,000 bugs fixed; 2000 Sept. – 2000 Data center Server; 2001 Oct. – Windows XP (whistler), Windows version 5.1; 2002 – Windows XP Media Center Edition, Windows XP Tablet PC Edition; 2003 April – Windows Server 2003 version 5.2 (whistler server) 2006 – Windows Vista, Windows version 6.0, Codename Longhorn (Client); 2006 June – Windows Compute Cluster Server 2003 (Windows CCS 2003); 2008 Feb. – Windows Server 2008 (Codename Longhorn); 2007 July – Windows Home Server 1.0' 2009 Oct. – Windows 7 (Codename Vienna); 2012 Oct. 26 – Windows 8; 2012 Sept. – Windows Server 2012 released; 2013 Oct. – Windows Server 2012 R2 released; 2013 Oct. 18 – Windows 8.1 (Codename Blue); 2015 July 29 – Windows 10 final release, version 10.0.10240; 2016 Oct. – Windows Server 2016, final release. Detalii:

[https://www.operating-system.org/betriebssystem/\\_english/bs-windows.htm](https://www.operating-system.org/betriebssystem/_english/bs-windows.htm)

5.

**Compania americană Apple** – Computerul *Apple* a fost construit de *Steve Jobs*, de 21 de ani, și *Stephen G. Wozniak*, de 26 de ani, în Palo Alto/California din SUA, în anul 1976. Ca și multe alte legende ale companiei din SUA, totul a început într-un garaj în care tehnicienii au dezvoltat un produs – primul *Apple I* placă de baza pentru un calculator. Primul cumpărător pentru *Apple I* (cu 50 de circuite) a fost *Paul Terrel*, proprietar al magazinului *Byte* pentru sisteme informatiche. *Steve Wozniak* era un inginer, iar *Steve Jobs* era un om de afaceri. De aceea, *Steve Wozniak* s-a preocupat de construirea de calculatoare, câștigând experiență la compania *Hewlett-Packard* cu construcția de calculatoare de buzunar și un calculator cu tastatură și ecran color de monitorizare. În schimb, *Steve Jobs* s-a concentrat pe partea de afaceri.

Calculatorul *Apple* a fost vândut cu 666,66 de dolari în magazinele *Byte*, din mai 1976. Pentru „*Byte into an Apple*” (1 Byte = 8 biți), publicitatea de vânzare era „*primul sistem microcomputer ieftin cu port de afișare și 8 kilobyte de memorie RAM pe o singură placă PC*”. Cu această publicitate de vânzări și o astfel de declarație s-a născut logo-ul de astăzi al lui *Apple* – curcubeul colorat și mărul cu mușcătură. După dezvoltarea ulterioară, *Apple II* a apărut în anul 1977, iar apoi succesorul *Apple III*, în anul 1980. *Apple III* a fost prezentat la *Conferința Națională a Calculatoarelor* (NCC – National Computer Conference) din California în 1980-05-19. *Sistemul de operare* sofisticat era dezvoltat ca sursă închisă și cu un nucleu monolitic. *Hardware-ul* este alcătuit dintr-un procesor MOS cu 8 biți de 8 MHz, 2 kbyte ROM și 128 kbyte memorie principală la placa de bază. În anul 1983, calculatorul *Apple* a fost unul dintre primii pionieri comerciali care au oferit împreună cu *Lisa Computer* și *Lisa OS* un *sistem de operare* cu o interfață grafică pentru utilizator, împreună cu simboluri și meniuri. *Lisa Computer* a stabilit în acel moment caracteristicile care sunt standard astăzi, o interfață grafică cu *mouse-ul* pentru operații. Ca aplicații standard sunt *LisaCalc* (program de calcul), *LisaGraph* pentru grafică de prezentare, *LisaDraw* (program de desen), *LisaWrite* pentru procesarea de text, *LisaProject* pentru metoda planului net, *LisaList* ca sistem de management de fișiere și *LisaTerminal* incluse. Detalii:

[https://www.operating-system.org/betriebssystem/\\_english/fa-apple.htm](https://www.operating-system.org/betriebssystem/_english/fa-apple.htm).

#### - **Sistemul de operare Apple ProDOS 1.0**

Sistemul de operare *Apple ProDOS* 1.0 bazat pe SOS a fost publicat în octombrie 1983. Sistemul *ProDOS* a fost împărțit în *ProDOS 8* și *ProDOS 16*, pentru procesoare de 8 biți și 16 biți, mai târziu. În 24 ianuarie 1984, succesul calculatorului *Macintosh* și al sistemului de operare *Mac OS* a început cu anunțul de piață al lui *Steve Jobs*. Producția de voce sintetică a unui text a avut un răsunet mare la public pentru caracteristicile de performanță. În anul

1984, poziția de lider a companiei IBM este zdruncinată. În termen de trei luni au fost vândute peste 70.000 de sisteme. Îmbunătățirile continue ale sistemului de operare de la *Apple* au sporit superioritatea în domeniul interfeței grafice și a utilizatorului simplu. *Calculatorul MAC* (computer activat cu *mouse*) a fost introdus în anul 1986, cu GS/OS (grafică și sound OS), ce a însemnat *ProDOS 16*, noul sistem grafic de 16 biți ce a sporit considerabil eficiența. Procedura de pornire, accesul la discul de stocare și timpul de pornire al programelor erau mult mai rapide.

- **Sistemul de operare Mac OS**

La început, *Mac OS Logo Mac OS* a fost numit de către compania *Apple* ca *Mac System Software*, un sistem de operare special conceput doar pentru procesoarele *Motorola* de 68K. Cu propriul hardware *Macintosh*, *Mac OS* ocupa un rol special în lumea *sistemelor desktop*. Prima versiune a fost „*System 1*” și a apărut împreună cu sistemul de operare *Mac*, în anul 1984. *Desktop-ul* clasic este conceput ca un singur sistem de operare pentru utilizatori și ascunde aproape complet calea completă către fișiere și directoare. Reprezentarea grafică este redusă la esență. În general, interfața este foarte ușor de utilizat și nu are nevoie de butonul drept al mouse-ului pentru interacțiunea cu utilizatorul. Sistemul *Mac OS* nu include o interfață de linie de comandă

- *Date – Version:* 1984 – System 0.85, 216 kbyte in size, first time with MFS file system; 1985 – System 2.0, updates and improvements like the Finder and menș 1986 – System 3.0, contains optimizations, first time with HFS file system; 1987 – System 4.0, bug fixes; 1987 – System 4.1, improved Finder (supports HDD >32 mbyte); 1988 – System 6, 32-bit color, Quickdraw support, serial port driver support, Truetype fonts; 1990 – System 7, 32-bit memory addressing on supported hardware, first time with virtual memory, updated GUI; 1994 – System 7.5; 1997 – Mac OS 7.6; 1997 – Mac OS 8.0; 1998 – Mac OS 8.1, 32-Bit, file system HFS+, only limited memory protection, USB and Firewire on supported hardware; 1998 – Mac OS 8.5.1, Sherlock search files on hard disks and on the internet, intranet updates; 1999 – Mac OS 8.6, multi-processor capable; 1999 – Mac OS 9 (sonata), Sherlock 2 can handle files with size up to 2 tbyte, user profiles (multiple users); 2001 – Mac OS 9.1; 1999 – Mac OS X Server, Mach- Kernel 2.5 (Unix-derivated microkernel) better performance and stability; 1999 March – Mac OS X Server 1.0; 2000 – Mac OS X Server 1.2.3; 2001 March – Mac OS X 10.0 (Cheetah); 2001 Sept. – Mac OS X 10.1 (Puma), improved performance, improved GUI; 2002 Aug. – Mac OS X 10.2 (Jaguar), new applications and technology; 2003 Oct. – Mac OS X 10.3 (Panther), new features for improved productivity and security; Mac OS X Server 10.3; 2005 April – Mac OS X 10.4 (Tiger), new technology; Mac OS X Server 10.4; 2006 Jan. – Mac OS X 10.4.4, first also available vor intel based syystems; 2007 Oct. – Mac OS X 10.5 (Leopard); 2009 Sept. – Mac OS X 10.6 (Snow Leopard), only for x86-64 Bit Intel dualcore; 2016 Sept. – macOS 10.12 (Sierra) available, introducing Apple File System (APFS). Detalii:

[https://www.operating-system.org/betriebssystem/\\_english/bs-macos.htm](https://www.operating-system.org/betriebssystem/_english/bs-macos.htm)

## 6. Sistemul de operare Linux, sistem de operare elaborat de o comunitate

Etape din dezvoltarea sistemului de operare *Linux*. În anul 1986, *Linus (Benedict) Torvalds* (este creatorul și principalul dezvoltator al kernel-ului *Linux*, care a devenit kernel-ul pentru multe distribuții *Linux* și sisteme de operare precum *Android* și *Chrome OS*; *Torvalds* este născut în Helsinki, Finland, în anul 1969, [https://en.wikipedia.org/wiki/Linus\\_Torvalds](https://en.wikipedia.org/wiki/Linus_Torvalds)) și-a programat propriul driver pentru controlerul *floppy disk*. A învățat programarea intensivă a hardware-ului și a devenit mai bine informat despre computerul său *Sinclair* cu sistemul de operare *Q-DOS*. În plus, el a furnizat propriile instrumente de programator. În anul 1991, când PC-ul 386-Intel a devenit modern, a primit un PC pentru a învăța despre programarea CPU-ului de la microprocesorul 386-Intel. A fost folosit un sistem de operare *MINIX*, derivat al lui *Unix*. A învățat și a cunoscut sistemul de operare *Unix* încă din anul 1990, când era student la universitate. Sistemul de operare *Minix* a fost dezvoltat de *Andrew Tanenbaum* ca sistem de învățare și a fost folosit în special în universități. Cartea scrisă a lui A. Tanenbaum „*Operating Systems: Design and Implementation*” (*Sisteme de operare: Proiectare și implementare*) se referă la conceptele sistemului de operare și la *Minix*. Această carte a devenit cartea preferată

a lui *Torvalds*. Codul sursă al *Minix* este *Open Source*, iar toate modificările erau legate de condițiile licenței.

Deoarece nu a găsit programul emulator pentru terminale furnizat în *Minix* acceptabil, a început proiectul său de a codifica propriul emulator pentru terminale, mai bun, cu mai multe funcții la nivel hardware. În plus, și-a programat propriile drivere pentru accesul la *mediul de date* și *sistemul de fișiere*, iar altele în *limbajul de asamblare*. Cu aceste funcții, software-ul avea abilitatea de a încărca și descărca fișiere de pe Internet. În linia programului de dezvoltare a terminalelor a dezvoltat tot mai multe funcții, astfel că a luat decizia de a-l îmbunătăți într-un *sistem de operare*. Sistemul său de operare a fost derivat din concepte de la *Minix*, dar complet scris de la zero începând de la Kernel. După serile lungi de programare, la 17 septembrie 1991, sistemul de operare *Freax* versiunea 0.01 a fost terminat, deoarece mediul de dezvoltare era folosit încă *MINIX* pentru CPU-uri 386. Acesta conținea deja bash-ul GNU Shell și GNC C-compilatorul GCC de la *Richard Stallman*, care se număra printre programele standard, între timp numit sistemul de operare *Linux*. Deoarece sistemul *Linux* profită, în special de *software pool* GNU, se numește, în general *GNU/Linux*.

- *Date – Version:* 1991 Sept. – Freax 0.01 – still needs Minix and special gcc compiler, 230 kbyte source code, incl. scripts and header files, minimal version, with floppy, keyboard and serial driver software, ext file system, 386 CPU support, UNIX-Shell bash; 1991 Sept. – Freax 0.02 – bash and gcc were ported by MINIX, needs 4 mbyte RAM for compiling software; 1991 Oct. – Freax 0.03 – small user group, gcc can compile himself on Linux, only needs 2 mbyte RAM for compiling software; 1991 Nov. – Freax 0.11 – international development team, first fixed disk driver software, mkfs/fsck/fdisk program, Hercules/MDA/CGA/EGA/VGA graphic, US/German/French/Finnish Keyboard, console can beep, Linux now has his own development environment; 1992 Jan. – Freax 0.12 – for the first time page-to-disk function built-in, Linux is put under the GPL, virtual memory, harddisk caching, POSIX job-control, more persons programming linux, multi-threading file system; 1992 April – Linux 0.96 – programmer and user group raised up, X Window system from the MIT is used for the first time; 1994 March – Linux 1.0 – 4,500 kbytes source code, incl. scripts and header files, more than 170,000 lines of source code, approx. 100 developers, approx. 100,000 users, first SCSI and sound driver software, for the first time networkable, ext2 file system; 1995 March – Linux 1.2 – 250,000 lines source code, about 50% are hardware driver, porting to alpha, MIPS, and SPARC CPUs, extended network functions like IP-Forwarding and NFS, IPX, AppleTalk; 1996 Juni – Linux 2.0 – 20,300 kbytes source code, incl. scripts and header files, approx. 800,000 lines of source code, porting to m68k and PowerPC CPUs, multi-processor capable up to 16 CPUs (experimental), symbol figure "Tux the penguin" was born; 1997 April – Linux 2.1.32 – after a trademark right dispute Torvalds lets register Linux as a trademark; 1999 Jan. – Linux 2.2.0 – 269 developers works on linux, approx. 10 million users, improved SMP support, IPv6 support as first operating system, extended software support by companies like StarOffice, Netscape; 2000 June – Linux 2.2.16 – ; 2001 Jan. – Linux Kernel 2.0.39 Release, contains bug fixes for security holes; 2001 Jan. – Linux 2.4.0 – 375 developers works on linux, approx. 15 million users, runs on altogether 13 hardware platforms, improved network support, improved performance for memory transactions, extended hardware support; 2002 Jan. – Linux Kernel 2.5.2; 2002 April – Linux Kernel 2.5.10; 2002 June – Linux Kernel 2.5.20; 2002 Aug. – Linux Kernel 2.5.30; 2002 Oct. – Linux Kernel 2.5.40; 2002 Nov. – Linux Kernel 2.5.50; 2003 Feb. – Linux Kernel 2.5.60; 2003 March – Linux Kernel 2.2.25; 2003 May – Linux Kernel 2.5.70; 2003 June – Linux 2.4.21 – Kernel 2.4.20 to 2.4.21 : 1738 code changes; 2003 Dec. – Linux Kernel 2.6.0 Release, optimized for big file storage devices and high data transfer rate, TCP/IP optimized, improved memory access and process scheduler, improvement in the threadings, improved Advanced Linux Sound Architecture (ALSA), contains Security-Enhanced Linux (SELinux); 2004 Feb. – Linux Kernel 2.0.40 Release; 2004 Feb. – Linux Kernel 2.2.26 Release, contains bug fixes for security holes, last release of the 2.2 branch; 2004 Dec. – Linux Kernel 2.6.10; 2007 Feb. – Linux Kernel 2.6.20; 2009 June – Linux Kernel 2.6.30; 2009 Dec. – Linux Kernel 2.6.32 (Longterm release); 2010 May – Linux Kernel 2.6.34 (Longterm release); 2011 May – Linux Kernel 2.6.39; 2011 July – Linux Kernel 3.0; 2012 Jan. – Linux Kernel 3.2 (Longterm release); 2012 May – Linux Kernel 3.4 (Longterm release); 2013 June – Linux Kernel 3.10 (Longterm release); 2013 Nov. – Linux Kernel 3.12 (Longterm release); 2014 March – Linux Kernel 3.14 (Longterm release); 2014 Aug. – Linux Kernel 3.16 (Longterm release); 2014 Dec. – Linux Kernel 3.18 (Longterm release); 2015 Feb. – Linux Kernel 3.19; 2015

April – Linux Kernel 4.0; 2015 June – Linux Kernel 4.1 (Longterm release); 2016 Jan. – Linux Kernel 4.4 (Longterm release); 2016 May – Linux Kernel 4.6; 2016 Aug. – Linux Kernel 4.7.1. Detalii:

[https://www.operating-system.org/betriebssystem/\\_english/bs-linux.htm](https://www.operating-system.org/betriebssystem/_english/bs-linux.htm)

Distribuitori Linux (*Distributors*): *Debian*, *Gentoo*, *Linspire*, *Mandriva*, *Red Flag*, *Red Hat*, *SuSE Linux*, *Slackware*. Un distribuitor este o echipă de dezvoltatori care avansează de la nucleul sistemului *Linux* (Kernel) pentru a oferi un pachet software de instalare. Pe lângă sistemul de adaptare individual, sunt incluse numeroase aplicații suplimentare, inclusiv driverul și asistentul, care pot fi instalate și configurate confortabil cu instalarea și configurarea proprie. Aceste distribuții sunt puse la dispozitie pe Internet ca imagine ISO sau pot fi cumpărate ieftin pe suporturi CD-ROM sau DVD. Banii obținuți sunt necesari pentru dezvoltare și suport.

Definiția din DEX.

**OPERÁRE** (< *opera*) s. f. Acțiunea de a opera. ♦ (INFORM.) *Sistem de o.* = program sau colecție de programe care asigură funcționarea de bază a calculatorului precum și intermedierea dintre utilizator și calculator. Inițiază și derulează legăturile logice dintre componente fizice (hardware) și informaționale (software) a ceea ce, în limbaj curent, utilizatorii numesc calculator. Exemple de sisteme de o.: Microsoft Windows 2000, UNIX, Linux, MS-DOS, MAC OS. (Sursa: DEX, <https://dexonline.ro/definitie/operare>)

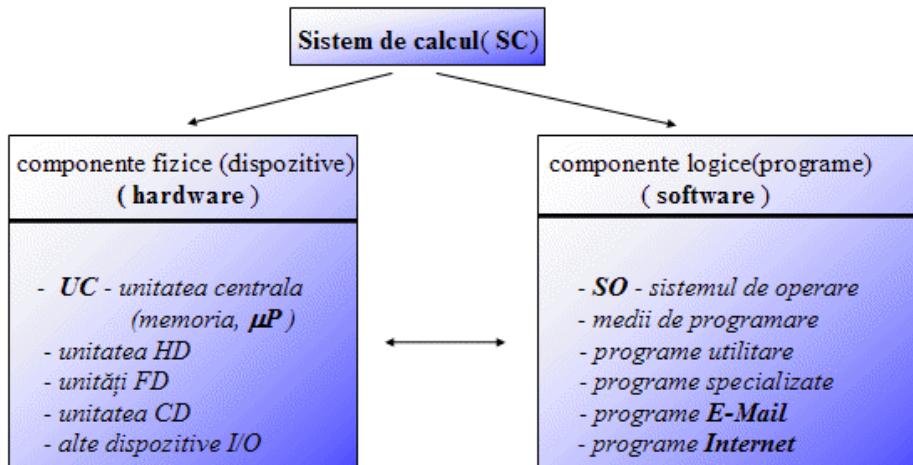
**Definiție.** Un sistem de operare este un sistem de programe care coordonează toate activitățile de calcul ce se desfășoară într-un SC pentru execuția programelor utilizatorilor folosind eficient resursele SC (memoria, micropresorul, dispozitivele I/O).

*Funcțiile generale ale unui sistem de calcul (SC):*

- comandă și controlează execuția programelor utilizatorilor;
- inițializează sistemul de calcul și verifică starea resurselor SC;
- comandă și controlează activitatea dispozitivelor I/O;
- formatează HD (hard disk) și FD (floppy disk);
- prelucrează și modifică starea fișierelor și programelor;
- asigură un sistem de protecție a datelor și programelor;
- definește structura sistemului de fișiere;
- gestionează (asigură partajarea) resurselor SC.

**Definiție rezumat:** SO – sistemul de operare este o colecție de programe (sistem de programe) ce are un nucleu și care se încarcă de pe HD în memoria internă la pornirea SC și realizează interfața dintre utilizator și dispozitivele I/O, definește structura sistemului de fișiere și gestionează resursele SC în scopul executării programelor aplicative ale utilizatorilor.

Pentru a înțelege rolul important al *sistemului de operare*, prezentăm schema următoare:



Componentele unui SO:

- *Nucleu* – realizează *servicii* pentru legătura cu rutinele BIOS și legătura cu cerințele programelor utilizatorilor (această componentă se încarcă de pe HD în memoria internă odată cu inițializarea SC);
- *Comenzi/Programe utilitare* – realizează servicii pentru utilizarea eficientă a resurselor SC (aceste *servicii* sunt oferite utilizatorului în orice moment în funcție de scopurile sale).

### Tipuri de sisteme de operare

Perfecționarea continuă a componentelor hardware ale unui **SC** implică perfecționarea și modificarea atât a *sistemelor de operare*, cât și a *componentelor software* instalate pe **SC**. În timp, aceste perfecționări și modificări au creat dificultăți în privința utilizării unor *programe* de pe un *sistem de calcul* pe altul, sau sub diverse *sisteme de operare*. Din aceste motive, pentru un **SO** și în general pentru *produsele software*, sunt importante următoarele atrbute:

- *compatibilitatea* – posibilitatea recunoașterii acestora de alte **SO** sau produse software și invers;
- *portabilitatea* – instalarea și execuția acestora pe diverse **SC**;

Sistemele de operare și, în general, *produsele software* înglobează *inteligenta și eforturile* unor *colective de cercetători, specialiști și experți* pe o anumită perioadă de timp, aceasta variind pâna la nivelul zecilor de ani, uneori perfecționările și modificările necesare fiind realizate de alte colective. De aici, așa-numitele „*versiuni*” ale produselor software obținute prin perfecționări și modificări.

În special, pentru *sistemele de operare* sunt importante următoarele atrbute:

- *monouser* – serviciile SO sunt oferite *la un moment dat* doar unui *singur utilizator*;
- *multiuser* – serviciile SO sunt accesate *simultan* de aplicații ale *mai multor utilizatori*;
- *monotasking* – **SO** execută *la un moment dat* – o singură sarcină (*task, proces*);
- *multitasking* – **SO** execută *simultan* mai multe *programe* (*task-uri, procese*).

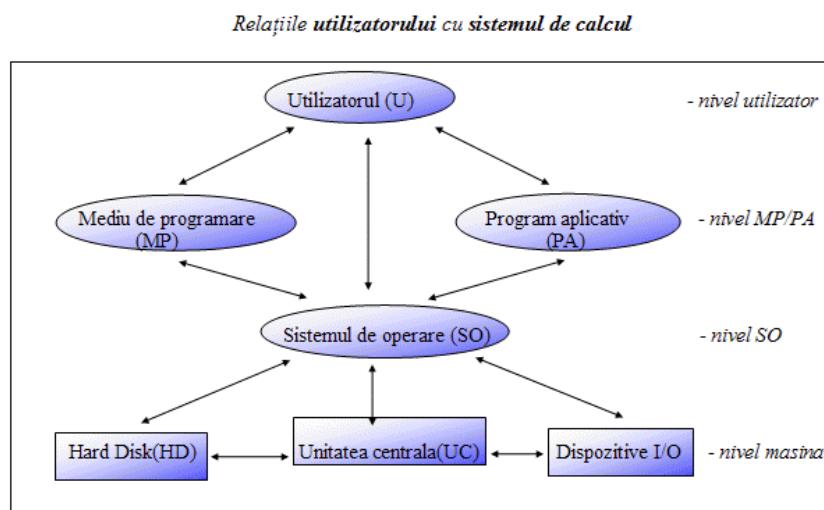
## Limbajul unui SO

Orice *limbaj de operare* este astfel *conceput și elaborat*, încât să realizeze în mod optim *funcțiile (serviciile)* pe care trebuie să le execute, dar în același timp să ofere o *interfață* comodă și eficientă între *utilizator* și *resursele SC*. Tocmai aceste deziderate au contribuit până în prezent la apariția diverselor *sisteme de operare*. Evident, unele **SO** au apărut și au dispărut, iar altele s-au perfecționat ținând seama de *tehnologia hardware*, dar și de *dezvoltarea software*. Același lucru se poate afirma și despre domeniul *limbajelor de programare*. Atât *sistemele de operare*, cât și *limbajele de programare* s-au perfecționat pentru a oferi utilizatorului *metode și tehnici moderne* în procesarea informației, în ceea ce se astăzi numește *tehnologia informației* (IT – Information Technologies). Din acest punct de vedere, un SO trebuie să execute următoarele *operații de bază*:

- *localizarea* informației;
- *memorarea/stocarea* informației;
- *procesarea/reprezentarea* informației;
- *comunicarea/vizualizarea* informației.

Limbajul sistemului de operare trebuie să ofere utilizatorului un mod eficient pentru ca acesta să poată să acceseze aceste operații de bază din domeniul *tehnologiei informației*. Aceste cerințe de bază au determinat ca sistemele de operare care sunt folosite în prezent să aibă atât asemănări, cât și deosebiri privind *limbajul, utilizarea resurselor SC, comenziile, serviciile oferite*. Evident, *structura sistemelor de calcul, performanța și tehnologia hardware* au influențat considerabil concepția, structura și performanța sistemelor de operare actuale.

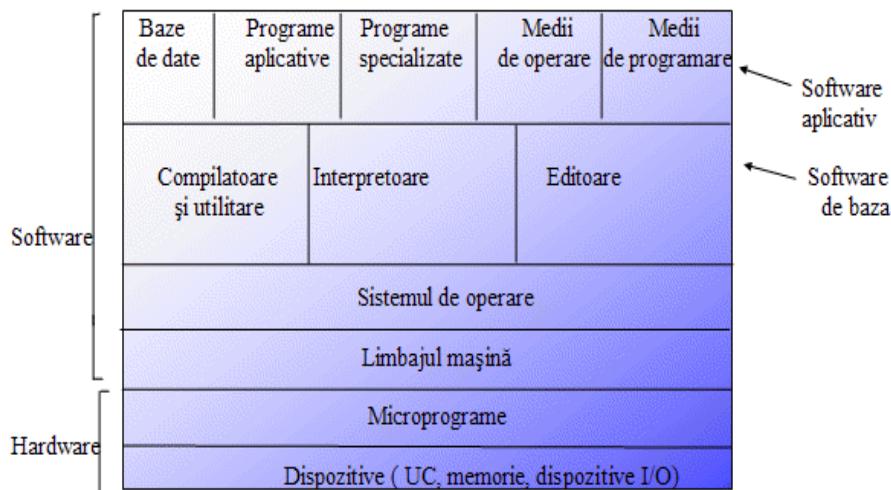
Pentru a scoate în evidență rolul SO în utilizarea unui SC prezentăm schema următoare (nivele ale diverselor operații în *relația Utilizator – SC*):



Din schema de mai sus se poate deduce rolul important al *sistemului de operare* în întreaga activitate a *sistemului de calcul*. Resursele gestionate de SO sunt:

- *fizice* ( microprocesoare, memorie internă, dispozitive I/O);
- *logice* ( programe utilitare, programe specializate, programe utilizator);
- *informationale* ( fișiere, baze de date, date de intrare/ieșire).

Realizarea funcțiilor de bază ale sistemului de calcul se face prin intermediul SO care face legatura *Hardware-Software-Utilizator*, prin urmare se pot evidenția diverse *nivele* ale interacțiunii *Hardware-Software*:



- *Microprogramele* sunt considerate ca un *software prima* localizate într-o memorie de tip ROM și reprezintă un interpretor al instrucțiunilor în *limbaj mașină* ce le traduce în *operații binare*.
- *Limbajul mașină* este un *ansamblu de instrucțiuni* pentru realizarea operațiilor *aritmetice, logice, de comparație, de conversie și de intrare/ieșire*, instrucțiuni interpretate de *microprograme* pentru a fi traduse în *operații binare* executate de unitatea centrală (UC).
- Un *interpretor* este un *program* pentru *interpretarea* instrucțiunilor scrise într-un limbaj de programare în vederea *codificării* lor direct în instrucțiuni din *limbajul mașină*. De asemenea, există *interpretoare de comenzi* ale sistemelor de operare.
- Un *compilator* este un *program* corespunzător unui limbaj de programare pentru analiza sintactică și semantică a instrucțiunilor scrise (*programul sursă*) în vederea rezolvării unei probleme, instrucțiuni ce vor fi traduse în aşa-numita *formă obiect* (*program object*), formă ce mai departe va fi tradusă în *instrucțiuni* ale *limbajului mașină*, și care se numește *forma executabilă* a programului sursă inițial. De menționat că pentru *execuția* unui *program*, este nevoie ca programul să fie sub formă *executabilă*, formă ce va fi stocată în memoria internă în vederea execuției programului respectiv.
- Un *utilitar* este un *program de interfață* între utilizator și sistemul de operare în vederea accesării unui serviciu oferit de **SO**.
- Un *editor* este un *program* pentru prelucrarea de *instrucțiuni, texte, imagini, formule* etc.

În domeniul *utilizării* calculatoarelor, *limbajele* se clasifică în *nivele*, în funcție de raportul față de *limbajul mașină* (*nivel 0*):

1. *limbaje de asamblare* (*nivel 1*) – instrucțiuni/comenzi exprimate prin *mnemonice* (prescurtări ale numelor unor operații) ce acționează asupra unor *adrese* ce nu sunt absolute, ci *simbolice*; un program traductor numit *asamblor* traduce (codifică) instrucțiunile în instrucțiuni elementare din *limbajul mașină*; unele componente ale SO sunt scrise în *limbaj de asamblare*;
2. *limbaje de macro-asamblare* (*nivel 2*) – instrucțiuni formate din *macro-instrucțiuni* ce reprezintă o comprimare a unor instrucțiuni specifice unui limbaj de asamblare;
3. *limbaje evolute* (de nivel înalt; *nivel 3*) – *limbaje simbolice* ce au structura instrucțiunilor apropiată aspectului algoritmic bazat pe *structuri de control* (*instrucțiuni structurate*), cunoscute sub denumirea de *limbaje de programare*; aceste limbaje sunt utilizate prin *compilatoare* sau *medii de dezvoltare* (editare, compilare, execuție, depanare etc.); de-a lungul vremii au fost

concepute și elaborate foarte multe (de ordinul *sutelor*) limbi de programare, dar în prezent sunt utilizate doar câteva (cele ce au rezistat perfecționărilor sistemelor de calcul): C++, Objective C, Pascal, Modula, Ada95, Java, CLOS, Fortran, Basic etc.

4. *limbaje specializate* (de nivel înalt; *nivel 4*) – *limbaje simbolice* bazate pe structuri de control având instrucțiuni ce operează asupra unor *entități* de bază (*obiecte*), de exemplu: *înregistrare*, *obiect grafic*, *eveniment*, *cunoștință* etc., aceste limbi sunt specifice fiecărui domeniu în care se realizează prelucrări: *Baze de date* (limbajele Dbase, FoxPro, Paradox, Clipper), *Grafică pe calculator* (limbajul MIRA), *Modelare și simulare* (limbajele Simula, Smalltalk), *Inteligentă artificială* (limbajele Prolog, Lisp);
5. *limbaje pseudo-cod* ( *nivel 5* ) – *limbaje algoritmice* ce au instrucțiuni apropiate de limbajele de programare, dar și de *limbajele științifice și cele naturale*;

*Definition.* Assembler – Assembly language is the uncontested speed champion among programming languages. An expert assembly language programmer will almost always produce a faster program than an expert C programmer.

*„Machines have so much memory today, saving space using assembly is not important. If you give someone an inch, they'll take a mile. Nowhere in programming does this saying have more application than in program memory use. For the longest time, programmers were quite happy with 4 Kbytes. Later, machines had 32 or even 64 Kilobytes. The programs filled up memory accordingly. Today, many machines have 32 or 64 megabytes of memory installed and some applications use it all. There are lots of technical reasons why programmers should strive to write shorter programs, though now is not the time to go into that. Let's just say that space is important and programmers should strive to write programs as short as possible regardless of how much main memory they have in their machine.”* Source: The Art of Assembly Language Programming, Spring 2008, Yale University, <http://flint.cs.yale.edu/cs422/doc/art-of-asm/pdf/> (pdf)

## SISTEME DE OPERARE.

List of Operating Systems: <http://www.operating-system.org/>

- List of Operating Systems: <http://en.wikipedia.org/>
- GNU Operating Systems: <http://www.gnu.org/>
- Operating Systems: <http://www.google.com/>
- Operating Systems Articles: <http://www.articlesbase.com/>

## Tehnologia Cloud computing

*Cloud computing* este utilizarea de diverse servicii, cum ar fi platforme de dezvoltare software, servere, stocare și software, prin intermediul Internetului, adesea denumit „nor”. În general, există trei caracteristici de *cloud computing* care sunt comune printre toți furnizorii de *cloud-computing*:

1. Back-end-ul aplicației (în special hardware-ul) este gestionat complet de un furnizor de cloud.
2. Un utilizator plătește numai pentru serviciile utilizate (memorie, timp de procesare și lățime de bandă etc.).
3. Serviciile sunt scalabile.

Multe progrese în domeniul *cloud computing* sunt strâns legate de virtualizare. Abilitatea de a plăti la cerere și la scară rapidă este în mare măsură un rezultat al furnizorilor de servicii de *cloud*

*computing*, care pot să împartă resurse împărțite între mai mulți clienți. Este comună clasificarea serviciilor de *cloud computing* ca infrastructură cu serviciu (IaaS), platformă ca serviciu (PaaS) sau software ca serviciu (SaaS).

### **Cloud computing Microsoft Azure Platform**

The Windows Azure Platform is a Microsoft cloud platform used to build, host and scale web applications through Microsoft datacenters. Windows Azure Platform is thus classified as platform as a service and forms part of Microsoft's cloud computing strategy, along with their software as a service offering, Microsoft Online Services.

The Windows Azure Platform provides an API built on REST, HTTP and XML that allows a developer to interact with the services provided by Windows Azure. Microsoft also provides a client-side managed class library which encapsulates the functions of interacting with the services. It also integrates with Microsoft Visual Studio so that it can be used as the IDE to develop and publish Azure-hosted applications. Windows Azure became commercially available on 1 Feb 2010.

*„Running applications on machines in an Internet-accessible data center can bring plenty of advantages. Yet wherever they run, applications are built on some kind of platform. For on-premises applications, this platform usually includes an operating system, some way to store data, and perhaps more. Applications running in the cloud need a similar foundation. The goal of Microsoft's Windows Azure is to provide this.”*

Premieră pentru România, anul 2011:

- Modulul de publicare a informațiilor pe Internet (*Sistemul informatic pentru examenele naționale*) a folosit platforma de *Cloud computing Microsoft Azure*, ceea ce asigură disponibilitate permanentă, capacitate de calcul practic infinită, redundanță și securitate a informațiilor. Acest modul a fost conceput special pentru a face față atât unui număr foarte mare de accesări simultane, cât și posibilelor atacuri menite să întrerupă disponibilitatea serviciilor furnizate.
- Conform [www.trafic.ro](http://www.trafic.ro), pe 12 iulie 2011 s-a înregistrat un nou record de vizitatori unici pentru <http://portal.edu.ro>, cu 828.935 într-o singură zi și peste 24 de milioane de afișări. Cele mai accesate site-uri ale portalului au fost <http://admitere.edu.ro>, <http://bacalaureat.edu.ro> și <http://titularizare.edu.ro>. Sursa: <http://portal.edu.ro/index.php/articles/news/11492>

### **Bibliografie**

1. Silberschatz A., Galvin P.B. and Gagne G., Operating Systems Concepts, 8th edn. John Wiley & Sons, 2009
2. Tanenbaum, Sisteme de operare moderne, Ed. Teora, 2004
3. Tanenbaum, Goodman, J. R., Organizarea structurată a calculatoarelor, Ed. Byblos, 2004
4. Deitel H., Operating Systems 3/e, Ed. Prentice Hall, 2004
5. Silberschatz A. Operating Systems Concepts. Seventh Edition, Wesley Publishing Company, 2006
6. Tanenbaum A. Modern Operating Systems, Prentice Hall, 2002
7. Gh. Dodescu, A. Vasilescu, B. Oancea, Sisteme de operare, Editura Economica, 2003
8. Sorin Adrian Ciureanu - Sisteme de Operare, Editura Printech, 2004
9. D. Acostachioia, Administrarea și configurarea sistemelor Linux, Ed. Polirom, 2003
10. S. Buraga, G. Ciobanu, Atelier de programare în retele de calculatoare, Ed. Polirom, 2001
11. L. Peterson, B. Davie, Retele de calculatoare: o abordare sistemică, ALL/Teora, Ed. Morgan Kaufmann, 2001/2004
12. M. Vlada, Sisteme de operare, Universitatea din Bucuresti, <http://prof.unibuc.ro/2018/02/info/>, tutorial, curs On line – <http://ebooks.unibuc.ro/informatica/Seiso/>
13. M. Vlada, Informatica, Ed. Ars Docendi, 1999

14. List of Operating Systems: <http://www.operating-system.org/>
  15. Liste of Operating Systems: <http://en.wikipedia.org/>
  16. GNU Operating Systems: <http://www.gnu.org/>
  17. The von Neumann Architecture of Computer Systems, [http://www.brown.edu/Research/Istrail\\_Lab/von\\_neumann.php](http://www.brown.edu/Research/Istrail_Lab/von_neumann.php), Istrail și Solomon Marcus: [http://www.brown.edu/Research/Istrail\\_Lab/papers/Istrail-Marcus012912FINAL.pdf](http://www.brown.edu/Research/Istrail_Lab/papers/Istrail-Marcus012912FINAL.pdf)
  18. Von Neumann, J. 1981. First draft of a report on the EDVAC.” In Stern, N. From ENIAC to Univac: An Appraisal of the Eckert-Mauchly Computers. Digital Press, Bedford, Massachusetts, <http://www.csupomona.edu/~hnriley/www/VonN.html>
  19. John von Neumann’s EDVAC Report 1945 John von Neumann’s 1945 on June 30 by Hungarian mathematician John von Neumann (1903-1957), <http://www.velocityguide.com/computer-history/john-von-neumann.html>, <http://www.wps.com/projects/EDVAC/>
  20. The Virtual von Neumann Architecture and the global computer, <http://meta-artificial.blogspot.com/2005/07/virtual-von-neumann-architecture.html>
- 

## Curs general – sisteme de operare

SISTEME DE OPERARE de SORIN ADRIAN CIUREANU, PrinTech, 2005

ONLINE <https://ro.scribd.com/doc/189341013/Sisteme-de-Operare>

### 1. INTRODUCERE

Informatica este o știință recentă și nu a avut încă timp să se structureze pe capitulo strict delimitate și bine definite. Dezvoltarea explozivă din ultimele două decenii a făcut ca ordonarea materialului să urmărească cu greu abundența de noi informații atât în domeniul tehnicii de calcul cât și în privința numeroaselor probleme în rezolvarea cărora aceasta poate fi utilizată.

Nu există o teorie unică a informaticii ci multe teorii care se suprapun parțial: arhitectura ordinatoarelor și evaluarea performanțelor lor, conceperea și verificarea circuitelor, algoritmică și analiza algoritmilor, concepția și semantica limbajelor de programare, structuri și baze de date, principiile sistemelor de operare, limbaje formale și compilare, calcul formal, coduri și criptografie, demonstrație automată, verificarea și validarea programelor, timp real și logici temporale, tratarea imaginilor, sinteza imaginilor, robotica etc. Fiecare dintre aceste domenii are problemele sale deschise, unele celebre, de exemplu găsirea unei semantici pentru limbajele de programare obiectuală.

Pe plan didactic, însă, s-au conturat anumite discipline care să asigure studenților posibilitatea de a accede la problematica vastă a informaticii. Printre altele se studiază și SISTEMELE DE OPERARE care intervin într-un sistem de calcul.

## 1.1 SISTEME DE OPERARE. DEFINIȚIE

Un sistem de operare (SO) este un set de programe care are două roluri primordiale:

-asigură o **interfață** între utilizator și sistemul de calcul, extinzând dar și simplificând setul de operații disponibile;

-asigură **gestionarea resurselor** fizice (procesor, memorie internă, echipamente periferice) și logice (procese, fișiere, proceduri, semafoare), implementând algoritmi destinați să optimizeze performanțele.

De exemplu, cele două componente ale definiției pot fi: publicitatea (interfață) și valoarea produsului (gestionarea resurselor). Exemple de SO sunt sistemele MS-DOS, WINDOWS și UNIX.

## 1.2. LOCUL SISTEMULUI DE OPERARE ÎNTR-UN SISTEM DE CALCUL

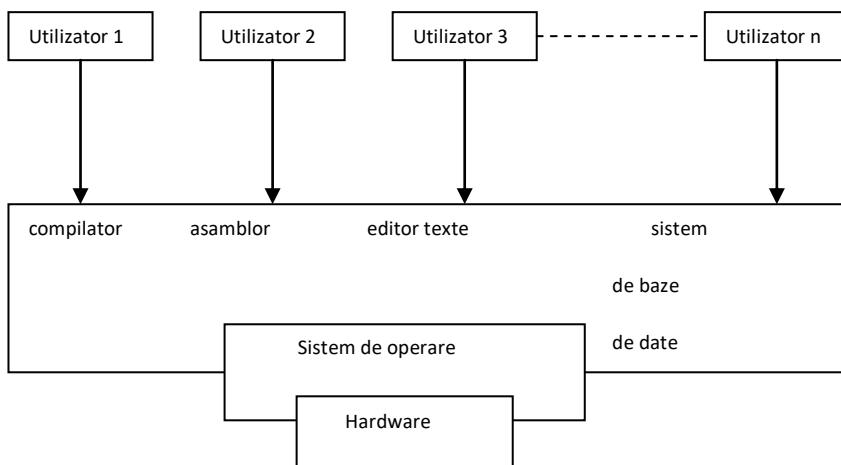
Componentele unui sistem de calcul sunt:

1.-**Hardware** - care furnizează resursele de bază (UC, memorie, dispozitive I/O).

2.-**Sistem de operare** - care controlează și coordonează utilizarea hardware-ului pentru diferite programe de aplicații și diferiți utilizatori.

3.-**Programe de aplicatie** - care definesc căile prin care resursele sistemului sunt utilizate pentru a rezolva problemele de calcul ale utilizatorilor (compilare, sisteme de baze de date, jocuri video, programe business etc.).

4.-**Utilizatori** – care pot fi persoane, mașini, alte calculatoare etc.



Acum zece ani, un sistem de operare era doar o piesă de bază a softului care rula pe o mașină și permitea manipularea fișierelor, conversa cu orice periferic și lansa programe. Acum sistemele de operare au devenit mai complexe, funcționând ca intermediari între utilizator și hardware, realizând executarea programelor utilizator cu mai mare ușurință și utilizând eficient hardware-l calculatorului.

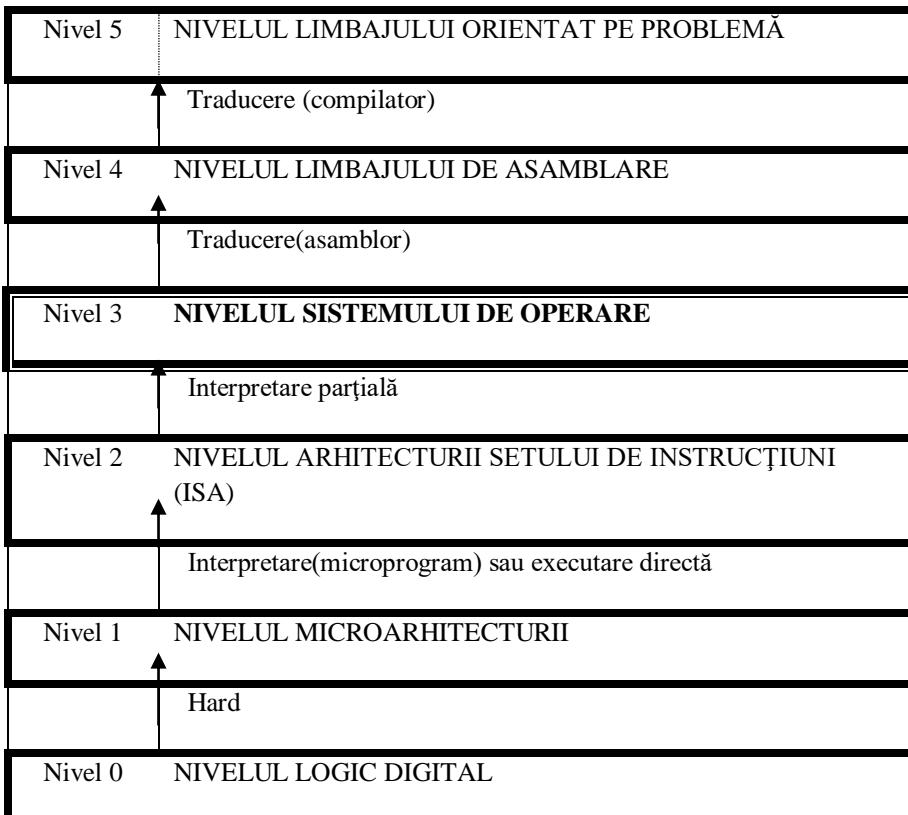
În concepția lui A. Tanenbaum, un calculator este organizat pe mai multe niveluri. Trecerea de pe un nivel pe altul se poate face prin *interpretare* sau prin *traducere*.

Dacă avem două limbi de programare  $L_i$  și  $L_{i+1}$ , se poate trece din  $L_{i+1}$  în  $L_i$  prin *interpretare* sau prin *traducere*.

**Interpretare.** Programul scris în  $L_{i+1}$  este executat pas cu pas, în sensul că instrucțiunile din  $L_{i+1}$  se execută pe rând, fiecare instrucțiune din  $L_{i+1}$  fiind o dată de intrare pentru  $L_i$ . Instrucțiunea din  $L_{i+1}$  are o secvență echivalentă de instrucțiuni care se execută în  $L_i$ . Nu se generează un nou program în  $L_i$ .

Traducere. Întreg programul din  $L_{i+1}$  se înlocuiește cu un nou program în  $L_i$ , generat prin înlocuirea fiecarei instrucțiuni din  $L_{i+1}$  cu o secvență echivalentă în  $L_i$ .

Considerând un sistem de calcul cu 6 niveluri, SO se plasează ca în următoarea schemă:



După Tanenbaum, calculatorul este foarte flexibil iar hardul și softul sunt echivalente sau, aşa cum spunea Lenz, „hardul este soft pietrificat”. În schema de mai sus, numai nivelul 0 (nivelul circuitelor) este hard pur. Celelalte niveluri pot fi implementate hard sau soft în funcție de optimul planificat.

Observăm că, în schemă, SO este pe nivelul 3. SO conține toate *instrucțiunile ISA* (Instructions Set Architecture), de pe nivelul 2, plus *apelurile sistem* (*system calls*). Un apel sistem invocă un serviciu predefinit al SO, de fapt una din instrucțiunile acestuia. Noile facilități (apelurile sistem) adăugate pe nivelul 3 sunt realizate de un interpretor care le execută pe nivelul 2 și care, din motive istorice, se numește SO. Instrucțiunile de pe nivelul 3 care sunt identice cu cele de pe nivelul 2 sunt executate direct de microprogram. Spunem că nivelul 3 este un nivel hibrid deoarece unele instrucțiuni de pe nivelul 3 sunt interpretate de SO (apelurile sistem) iar altele sunt interpretate direct de microprogram.

### 1.3. SARCINILE SISTEMULUI DE OPERARE

Pornind de la definiția unui SO, se pot identifica responsabilitățile sale, prezentate în continuare.

#### 1.3.1. Asigurarea interfeței cu utilizatorul

Trebuie precizat că interfața SO ↔ utilizator este constituită din orice instrument care permite comunicarea între un SO și un operator (utilizator), indiferent dacă este hard sau soft. În evasitotalitate, interfețele în SO sunt soft și pot lua următoarele forme.

##### 1.3.1.1. Monitoare

Unele calculatoare conțin, stocat într-o memorie ROM internă, un program numit *monitor* care se lansează automat la pornirea calculatorului și îi permite operatorului să efectueze operații simple asupra

sistemului de calcul, cum ar fi: inspectarea și modificarea registrelor procesorului, vizualizarea conținutului memoriei etc. De obicei, monitorul este un complementar al SO, în sensul că pornește când nu a putut fi încărcat SO. Este specific sistemelor de calcul mai vechi, actualmente existând puține sisteme de calcul care înglobează un monitor.

### 1.3.1.2. Interfețe în linie de comandă (interfețe text)

Interfețele în linie de comandă sunt reprezentate, în general, de un program numit *interpreter de comenzi*, care afișează pe ecran un prompter, primește comanda introdusă de operator și o execută. Comenzile se scriu folosind tastatura și pot fi însoțite de parametri. Aproape toate sistemele de operare includ o interfață în linie de comandă, unele foarte bine puse la punct (UNIX) iar altele destul de primitive (MSDOS, WINDOWS).

### 1.3.1.3 Interfețe grafice

Interfețele grafice sunt cele mai populare. Se prezintă sub forma unui set de obiecte grafice (de regulă suprafete rectangulare) prin intermediul cărora operatorul poate comunica cu SO, lansând operații, setând diferite opțiuni contextuale etc.

Vom prezenta, comparativ, caracteristicile celor două tipuri de interfețe cu calculatorul.

*Interfață în linie de comandă.*

Avantaje:

- Permite scrierea clară și explicită a comenziilor, cu toți parametrii bine definiți.
- Oferă flexibilitate în utilizare.
- Comunicarea cu SO se face mai rapid și eficient.

Dezavantaje:

- Operatorul trebuie să cunoască bine comenzile și efectele lor.
- Este mai greu de utilizat de către neprofesioniști.

*Interfață grafică*

Avantaje:

- Este intuitivă și ușor de folosit.
- Poate fi utilizată de neprofesioniști.
- Creează un mediu de lucru ordonat.
- Permite crearea și utilizarea unor aplicații complexe, precum și integrarea acestora în medii de lucru unitare.

Dezavantaje:

- Anumite operații legate, de exemplu, de configurația sistemului pot să nu fie accesibile din menurile și ferestrele interfeței.
- Interfața ascunde anumite detalii legate de preluarea și execuția comenziilor.
- Folosește mai multe resurse și este mai puțin flexibilă.

In general, există două componente principale:

-*comenzi*, care sunt introduse de utilizator și prelucrate de interpretorul de comenzi înainte de a ajunge la SO;

-*apeluri sistem*, care sunt folosite direct de către programatorii de sistem și indirect de către programatorii de aplicații, apar în programe și se declanșează în timpul execuției acestor programe, fiind asemănătoare cu procedurile.

Apelurile sistem definesc propriu zis *arhitectura* unui SO, adică aparența sa către exterior, aspectul funcțional al sistemului. Se poate considera că ele îndeplinește același rol pentru SO ca și lista de instrucțiuni pentru procesor.

Un aspect care devine tot mai important este acela al standardizării interfețelor unui SO. Asigurarea unei interfețe uniforme (atât la nivel de comenzi cât și la apeluri sistem) ar permite *portabilitatea* aplicațiilor, adică posibilitatea de a trece un program de aplicații de sub un SO sub altul, fără modificări.

### 1.3.2. Gestiunea proceselor și a procesoarelor

Un program în execuție sub controlul unui SO este un *proces*. Pentru a-și îndeplini sarcinile, procesele au nevoie de resurse, cum ar fi: timpul de lucru al UC (unitatea centrală), memorii, fișiere, dispozitive I/O. Apare evidentă necesitatea ca resursele să fie utilizate în comun. O serie de mecanisme speciale au fost introduse în SO pentru a servi la gestionarea proceselor la diferite niveluri de detaliu. În practică există o legătură strânsă și cu nivelul de întrerupere al calculatorului. Gestiunea procesorului este privită ca o parte componentă a gestionării proceselor.

### 1.3.3. Gestionarea memoriei

Acest modul controlează, în primul rând, utilizarea memoriei interne. Întotdeauna o porțiune a acestei memorii este necesară pentru însuși SO, iar restul este necesar pentru programele utilizator. Frecvent, mai multe programe utilizator se află simultan în memorie, ceea ce presupune rezolvarea unor probleme de protecție a lor, de cooperare între aceste programe sau între programe și SO, precum și a unor probleme de împărțire a memoriei disponibile între programele solicitante.

Memoria externă este implicată în activitatea modulului de gestionare a memoriei. Mecanismele *swapping* și de *memorie virtuală* folosesc memorie externă pentru a extinde capacitatea memoriei interne.

### 1.3.4. Gestionarea perifericelor

Modul de gestionare a perifericelor cuprinde toate aspectele operațiilor de introducere și extragere a informației: pregătirea operației, lansarea cererilor de transfer de informație, controlul transferului propriu zis, tratarea erorilor.

La majoritatea echipamentelor periferice actuale transferurile de intrare/ieșire (**I/O=input/output**) se desfășoară indiferent de procesor, rolul SO fiind acela de a asigura aceste transferuri cu ajutorul sistemului de întreruperi.

### 1.3.5. Gestionarea fișierelor

Fișierul este entitatea de bază a unui SO, cea care păstrează informația. Toate operațiile legate de fișiere (creare, ștergere, atribuire, copiere, colecție de fișiere, securitatea informației) sunt coordonate de SO căruia îi revine un rol important în acest caz.

### 1.3.6. Tratarea erorilor

SO este acela care tratează erorile apărute atât la nivel hard cât și la nivel soft. Tratarea unei erori înseamnă mai întâi detectarea ei, apoi modul de revenire din eroare pentru continuarea lucrului, mod care

deinde de cauza erorii și de complexitatea SO. Unele erori sunt transparente utilizatorului, altele trebuie să neapărat semnalate. De asemenea unele erori sunt fatale și duc la oprirea SO. Mecanismul principal de tratare a erorilor este ca SO să repete de un număr definit de ori operația eşuată până la desfășurarea ei cu succes sau până la apariția erorii fatale. La ora actuală SO moderne acordă un rol foarte important conceptului de siguranță în funcționare și de protecție a informației.

#### 1.4. CARACTERISTICILE SISTEMELOR DE OPERARE

Formulăm în continuare cele mai importante caracteristici ale sistemelor de operare.

##### 1.4.1. Modul de introducere a programelor în sistem

Din acest punct de vedere sistemele de operare pot fi:

- SO *seriale*, în care se acceptă introducerea lucrărilor de la un singur dispozitiv de intrare;
- SO *paralele*, în care introducerea lucrărilor se face de la mai multe dispozitive de intrare;
- SO *cu introducerea lucrărilor la distanță*.

De exemplu, sistemele UNIX și WINDOWS sunt paralele și cu introducere la distanță, pe când sistemul MS-DOS este serial.

##### 1.4.2. Modul de planificare a lucrărilor pentru execuție

Există:

- SO *orientate pe lucrări*, care admit ca unitate de planificare lucrarea, alcătuită din unul sau mai multe programe succesive ale același utilizator;
- SO *orientate pe proces*, care admit ca unitate de planificare procesul.

SO moderne sunt orientate pe proces.

##### 1.4.3. Numărul de programe prezente simultan în memorie

Sistemele pot fi:

- SO cu *monoprogramare* (cu un singur program în memoria principală);
- SO cu *multiprogramare* (cu mai multe programe existente, la un moment dat, în memoria principală).

De exemplu, sistemele UNIX și WINDOWS sunt cu multiprogramare. Sistemul MS-DOS este ceva între monoprogramare și multiprogramare.

##### 1.4.4. Gradul de comunicare a proceselor în multiprogramare

Sistemele de operare cu multiprogramare pot fi:

- SO *monotasking*, în care programele existente în memorie nu au un obiectiv comun, nu comunică și nu-și pot sincroniza activitățile;

SO *multitasking*, în care programele existente în memorie au un obiectiv comun și își sincronizează activitățile.

UNIX și WINDOWS sunt multitasking. MS-DOS este un hibrid; prin operațiile de redirectare și indirectare MS-DOS nu este monotasking pur (redirectare - sort<date.txt ; indirectare - dir|sort|more ).

##### 1.4.5. Numărul de utilizatori simultani ai SO

- SO *monouser* (cu un singur utilizator) ;

-SO *multiuser* (cu mai mulți utilizatori).

UNIX și WINDOWS sunt multiuser, MS-DOS este monouser.

#### **1.4.6. Modul de utilizare a resurselor**

După modul se utilizare a resurselor, sistemele de operare pot fi :

-SO *cu resurse alocate* (resursele alocate proceselor sunt alocate acestora pe toată desfășurarea execuției) ;

-SO *în timp real* (permite controlul executării proceselor în interiorul unui interval de timp specificat);

-SO *cu resurse partajate* (resursele necesare proceselor sunt afectate acestora periodic, pe durata unor cuante de timp).

Dacă resursa partajată este timpul unității centrale, SO devine partajat.

SO în timp real sunt utilizate pentru conducerea directă, interactivă, a unui proces tehnologic sau a altel aplicații. Procesul va transmite către SO în timp real parametrii procesului iar SO va transmite către proces deciziile luate. Informațiile despre proces sunt luate în considerare în momentul comunicării lor iar răspunsul sistemului trebuie să fie extrem de rapid, deci timpii de execuție a programelor trebuie să fie mici.

Accesul la resursele sistemului poate fi :

-*direct* (caz particular SO în timp real, când se cere o valoare partajabilă maximă a timpului de răspuns) ;

-*multiplu* (acces la resursele sistemului pentru un mare număr de utilizatori) ;

-*time sharing* (alocarea timpului se face pe o cantă de timp) ;

-*la distanță* (prelucrarea se face asupra unor date distribuite și dispersate geografic).

#### **1.4.7. SO pentru arhitecturi paralele**

**SO paralele**, folosite în multe procesoare

**SO distribuite**, folosite în multe calculatoare.

### **1.5. COMPOENȚELE SISTEMELOR DE OPERARE**

Majoritatea sistemelor de operare, pentru a răspunde rolului de interfață cu utilizatorii, sunt organizate pe două niveluri:

-*nivelul fizic*, care este mai apropiat de partea hardware a sistemului de calcul, interferând cu aceasta printr-un sistem de intreruperi;

-*nivelul logic*, care este mai apropiat de utilizator, interferând cu acesta prin intermediul unor comenzi, limbaje de programare, utilitare etc.

Potrivit acestor două niveluri, sistemele de operare cuprind în principal două categorii de programe:

-programe de control și comandă, cu rolul de coordonare și control al tuturor funcțiilor sistemelor de operare, cum ar fi procese de intrare ieșire, execuția intreruperilor, comunicația hardware-utilizator etc.;

-programe de servicii (prelucrări), care sunt executate sub supravegherea programelor de comandă și control, fiind utilizate de programator pentru dezvoltarea programelor sale de aplicație.

In general, putem considera că un SO este format din două părți: *partea de control* și *partea de serviciu*.

### 1.5.1. Partea de control

Partea de control realizează interfață directă cu hardul. Ea conține proceduri pentru:

- gestiunea întreruperilor;
- gestiunea proceselor;
- gestiunea memoriei;
- gestiunea operațiilor de intrare/ieșire;
- gestiunea fișierelor;
- planificarea lucrărilor și alocarea resurselor.

### 1.5.2. Partea de serviciu

Partea de serviciu conține instrumente de lucru aflate la dispoziția utilizatorului. Ea exploatează partea de control, dar transparent pentru utilizator.

Partea de serviciu cuprinde soft aplicativ. După destinația lor, serviciile pot fi de :

- birotică ;
- baze de date ;
- dezvoltare de aplicații/programe ;
- rețele de calculatoare ;
- produse soft pentru prelucrarea informațiilor din diverse domenii.

## 1.6. STRUCTURA SISTEMELOR DE OPERARE

În sistemele de operare apar, în general, două aspecte structurale:

-*kernel* (nucleu);

și

-*user* (utilizator).

Nucleul (kernel) are următoarele principale funcții:

- asigurarea unui mecanism pentru crearea și distrugerea proceselor;
- realizarea gestionării proceselor, procesoarelor, memoriei și perifericelor;
- furnizarea unor instrumente pentru mecanisme de sincronizare a proceselor;
- furnizarea unor instrumente de comunicație care să permită proceselor să își transmită informații.

Trebuie făcută diferență între *procesele sistem*, care au privilegii mari, și *procesele utilizator*, cu un grad de privilegii mult mai mic.

După structura lor, sistemele de operare se clasifică în :

-SO *modulare*, formate din entități cu roluri bine definite ;

-SO *ierarhizate*, în care o entitate poate folosi componente de nivel inferior (de exemplu, partea de serviciu poate folosi partea de control);

-SO *portabile*, pentru care efortul de a trece SO de pe un calculator pe altul este mic, mai mic decât cel de a-l rescrie .

Sistemele UNIX și WINDOWS sunt portabile. Cele mai vechi, de exemplu RSX, nu erau portabile.

### 1.7. DEZVOLTAREA ISTORICĂ A SISTEMELOR DE OPERARE

În dezvoltarea istorică a sistemelor de operare, după apariția lor, s-au adus multe îmbunătățiri, atât în strategia SO cât și în implementarea SO. (De exemplu; multiprogramare, multitasking, timesharing etc.). Un lucru este acum evident: puternica interdependență hard↔soft, adică între arhitectura sistemului de calcul și arhitectura SO.

Au existat perioade în care sistemul de calcul avea performanțe mai scăzute iar SO venea să suplimească aceste neajunsuri (ex. procesoare 8biți, 8080, și sisteme de operare CP/M); în alte perioade situația este inversă, adică un sistem de calcul are performanțe puternice iar sistemul de operare nu reușește să utilizeze la maxim toate facilitățile hard (de exemplu situația actuală, când sistemele de operare, atât WINDOWS cât și UNIX, nu reușesc să utilizeze în modul cel mai eficient toate facilitățile procesoarelor PENTIUM).

Interdependența între arhitectura sistemelor de calcul și structura sistemelor de operare constă și în faptul că, cel mai adesea, s-a implementat în hard o parte a funcțiilor SO. Exemple:

a) Actuala componentă BIOS (Basic I/Osystem) dintr-un PC era o componentă a sistemului de operare CP/M.

b) Managementul de memorie este actualmente implementat ca modul hard în procesor. Pentru generațiile trecute era o componentă a sistemului de gestiune a memoriei dintr-un SO.

c) Memoria CACHE, foarte utilizată astăzi și implementată exclusiv în memoria internă, în hard discuri etc, era o componentă a sistemului de operare UNIX în administrarea fișierelor.

În noile sisteme de operare, a apărut o nouă noțiune, aceea de micronucleu (microkernel). În sistemele clasice cu kernel, acesta era o componentă rezidentă permanent în memorie, în sensul că era o parte indivizibilă a SO. Însă un astfel de nucleu este foarte mare și greu de controlat. De aceea, constructorii de SO au înlocuit nucleul cu un micronucleu, o componentă de dimensiuni mai mici, rezidentă în memorie și indivizibilă. În micronucleu au rămas serviciile esențiale, cum ar fi : repornirea unui proces, facilitățile de comunicare între procese prin mesaje, rudimente de management de memorie, câteva componente de intrare- ieșire de nivel foarte jos. Restul serviciilor, inclusiv driverele, au fost mutate în afara micronucleului, în zona de « aplicații utilizator ».

SO cu micronucleu funcționează pe principiul client/server prin intermediul mesajelor. De exemplu, dacă un proces vrea să citească un fișier, el va trimite un mesaj cu ajutorul micronucleului către o altă aplicație care rulează în spațiul utilizator și care acționează ca un server de fișiere. Această aplicație va efectua comanda iar rezultatul ei va ajunge înapoi la procesul apelant, tot prin intermediul unor mesaje. În acest fel se poate crea o interfață bine definită pentru fiecare serviciu, adică un set de mesaje pe care serverul le înțelege. Se pot instala ușor noi servicii în sistem, servicii care pot fi pornite fără a fi necesară reformarea nucleului. Se pot crea în mod simplu servicii proprii de către utilizatori.

În concluzie, SO cu micronucleu au avantajul unei viteze mai mari, obținute prin eliberarea unei părți de memorie, și dezavantajul unui timp pierdut cu schimbul de mesaje. Se pare, însă, că viteza mare va duce la înlocuirea sistemelor clasice cu sisteme cu micronucleu.

În SO UNIX, un exemplu de micronucleu este ONX al firmei Quantum Software. ONX respectă standardele POSIX 1003.1, care se referă la interfața dintre aplicațiile C și serviciile nucleului, POSIX 1003.2, care se referă la interfața la nivel de SHELL și POSIX 1003.4, care se referă la SO în timp real.

ONX conține un micronucleu de 8KO în care sunt implementate planificarea și inter-schimbarea proceselor, tratarea intreruperilor, servicii de rețea de nivel scăzut. Un sistem minimal ONX adaugă la acest micronucleu un controlor de procese care creează și controlează procesele în memorie. Micronucleul conține 14 apeluri sistem.

## 2. PLANIFICAREA PROCESOARELOR (UC)

Multiprogramarea reprezintă cel mai important concept folosit în cadrul sistemelor de operare moderne. Existența în memorie a mai multor procese face posibil ca, printr-un mecanism de planificare a unității centrale (UC), să se îmbunătățească eficiența globală a sistemului de calcul, realizându-se o cantitate mai mare de lucru într-un timp mai scurt.

Există procese:

- limitate UC, când procesul are componente cu timp majoritar de desfășurare în I/O;
- limitate I/O, când procesul are componente cu timp majoritar de desfășurare în UC.

### 2.1. SCHEMA GENERALĂ DE PLANIFICARE A PROCESOARELOR

Procesele stau în memorie grupate într-un șir de așteptare în vederea alocării în UC. Implementarea acestui șir, numit coadă de așteptare, se realizează de obicei sub forma unei liste înlățuite.

*Planificatorul pe termen lung* (planificator de joburi) stabilește care sunt procesele ce vor fi încărcate în memorie. El controlează gradul de multiprogramare. Frecvența sa este mică.

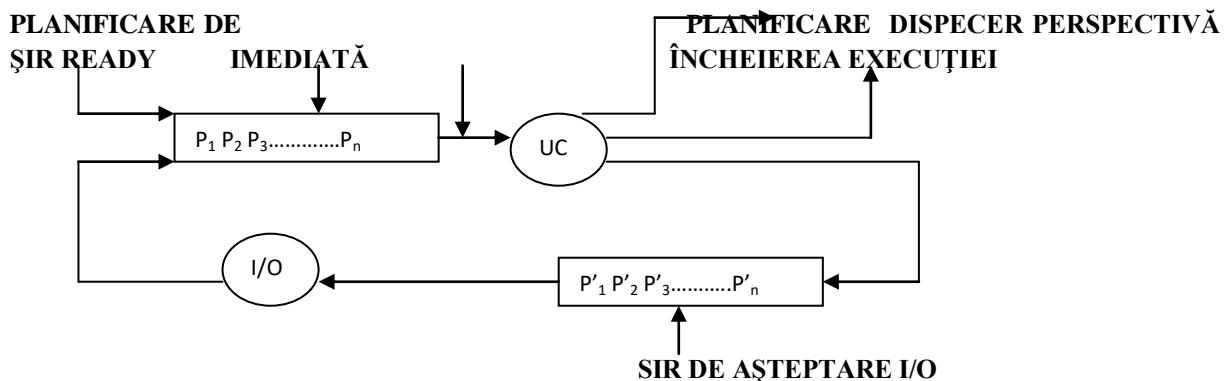


Fig. 2.1. Schema generală de planificare a proceselor.

*Dispecerul* realizează efectiv transferarea controlului UC asupra procesului selectat de planificatorul UC. Trebuie să fie rapid pentru a putea realiza eficient operațiile necesare: încărcarea registrelor, comutarea în mod utilizator etc.

*Planificatorul pe termen scurt* (planificator UC) selectează unul din procesele gata de execuție, aflate deja în memoria internă, și îl alocă UC. Are o frecvență de execuție foarte mare și trebuie proiectat foarte rapid.

## 2.2. CRITERII DE PERFORMANȚĂ A PLANIFICĂRII UC

Când se dorește un algoritm de planificare, se pot lua în considerare mai multe criterii :

-*gradul de utilizare UC* (aproximativ 40% pentru un sistem cu grad de încărcare redusă, 90% pentru un sistem cu grad mare de încărcare) ;

-*throughput* (numărul de procese executate într-un interval de timp precizat);

-*turnaround time* (durata totală a execuției unui proces) reprezintă timpul scurs între momentul introducerii procesului în memorie și momentul încheierii execuției sale; se exprimă ca suma perioadelor de timp de așteptare pentru a intra în memorie, de așteptare în sirul READY, de execuție (în UC) și de realizare a operațiilor I/O ;

-*durata de așteptare* (algoritmul de așteptare influențează numai durata de așteptare în sirul READY și nu afectează durata de execuție a procesului sau timpul destinat operațiilor I/O) ;

-*durata de răspuns* (timpul scurs între formularea unei cereri și inițierea răspunsului corespunzător).

Prin alegerea unui algoritm de planificare se urmărește optimizarea criteriului luat în considerație și anume maximizare pentru primele două și minimizare pentru ultimele trei.

## 2.3. ALGORITMI DE PLANIFICARE UC

În prezentarea algoritmilor de planificare UC performanțele sunt apreciate cu ajutorul mărimii DMA (durata medie de așteptare).

### 2.3.1. Algoritmul FCFS (First Come First Served)

Șirul READY este de tip FIFO (First Input First Output= Primul Intrat Primul Ieșit).

Exemplu:

Proces	Durata UC		Durata de așteptare
1	10		0
2	29		10
3	3		39
4	7		42
5	12		49

⇒

Durata medie de așteptare va fi :

$$\text{DMA} = (0+10+39+42+47)/5 = 140/5 = 28$$

Dezavantaje : DMA nu este minimală și poate varia în limite foarte largi în funcție de caracteristicile procesului. În plus DMA depinde de ordinea proceselor.

### 2.3.2. Algoritmul SJF (Shortest Job First)

Așa cum arată denumirea, se execută mai întâi cel mai scurt job. La egalitate, se aplică regula FCFS (First Come First Served).

Exemplu:

Proces	Durata UC		Proces	Durata UC	Durata de așteptare
1	10	⇒	3	3	0
2	29		4	7	3
3	3		1	10	10
4	7		5	12	20
5	12		2	29	32

Durata medie de așteptare va fi :

$$\text{DMA} = (3+10+20+32)/5 = 65/5 = 13$$

Dacă se cunosc cu precizie ciclurile UC (ca timp), SJF este optimal. Problema principală este cunoașterea duratei ciclului UC.

### 2.3.3. Algoritmi bazați pe prioritate

În cadrul unui astfel de algoritm, fiecărui proces i se asociază o prioritate, UC fiind alocată procesului cu cea mai mare prioritate din șirul READY. Se poate defini o *prioritate internă* și o *prioritate externă*.

Prioritatea internă se calculează pe baza unei entități măsurabile :

- limita de timp ;
- necesarul de memorie ;
- numărul fișierelor deschise ;
- raportul dintre numărul de cicluri rafală I/O și numărul de cicluri rafală UC.

Pentru prioritatea externă, criteriile folosite sunt din afara sistemului de operare :

- departamentul care sponsorizează lucrările ;
- factori politici ;
- factori financiari.

Principala problemă a algoritmilor bazați pe priorități este posibilitatea blocării la infinit (a infometării) proceselor care sunt gata de execuție, dar deoarece au prioritate redusă, nu reușesc să obțină accesul la UC. O astfel de situație poate să apară într-un sistem cu încărcare mare, în care se execută un număr considerabil de

procese cu prioritate ridicată ; acestea vor obține accesul la UC în detrimentul proceselor cu prioritate redusă care pot să nu fie executate niciodată. O soluție a acestei probleme este *îmbătrânirea proceselor*, o tehnică prin care se mărește treptat prioritatea proceselor remanente timp îndelungat în sistem.

#### 2.3.4. Algoritmi preemptivi

Un algoritm *preemptiv* permite întreruperea execuției unui proces în momentul când în sirul READY apare un alt proces cu drept prioritar de execuție.

Dintre algoritmii prezenți anterior :

-FCFS este prin definiție nepreemptiv ;

-SJF poate fi realizat preemptiv; dacă în sirul READY sosește un proces al cărui ciclu rafală UC următor este mai scurt decât ce a mai rămas de executat din ciclul curent, se întrerupe execuția ciclului curent și se alocă UC nou lui proces;

-algoritmii bazați pe prioritate, de asemenea, pot fi realizati preemptiv ; la fel ca la SJF, timpul rămas poate fi înlocuit cu mărimea priorității.

Exemplu:

Proces	Momentul sosirii în sirul READY	Durata ciclului rafală
1	0	12
2	3	3
3	4	6

În SJF fără preemptie :

Ordine	Așteptare
P1(12)	0
P2(3)	9
P3(6)	8+3

$$\text{DMA} = (9+8+3)/3=6,67$$

În SJF cu preemptie :

Ordine	Așteptare
P1 3	P1 așteaptă 3+6=9

P2 3	P2 așteaptă 0
P3 6	P3 așteaptă 3-1
P1 9	

$$\text{DMA} = (9+0+2)/3 = 3,67$$

### 2.3.5. Algoritmul Round-Robin

Este un algoritm de tip time-sharing. Fiecare proces i se alocă numai o cantă de timp (10ms – 100ms) iar sirul READY se tratează ca FIFO circular.

Exemplu:

Proces	Durata ciclului rafală	
1	10	P1 așteaptă
2	29	
3	3	
4	7	
5	12	

Dacă cuanta de timp este de 10 ms, atunci ordinea în UC este următoarea:

P1(0) P2(19) P3(0) P4(0) P5(2) P2(9) P5(0) P2(0)

În paranteză este dat timpul care a mai rămas.

Observații :

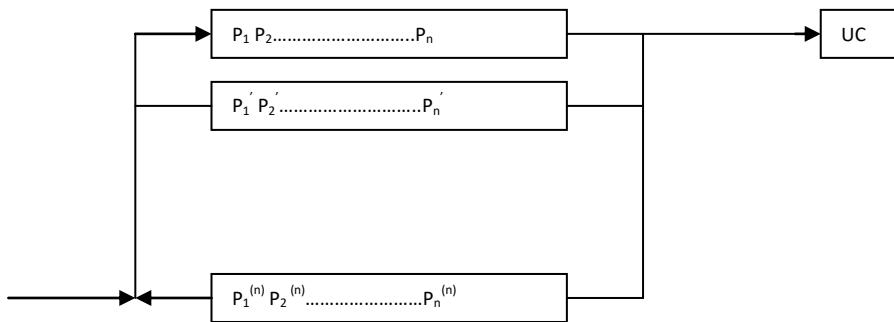
-planificatorul alocă UC fiecarui proces pe durata a cel mult o cantă ; dacă durata procesului este mai mică decât această cantă, procesul eliberează UC prin comunicarea încheierii execuției ;

-mărimea cuantei afectează performanțele algoritmului Round-Robin ; dacă cuanta este foarte mare, comportarea este asemănătoare FCFS ; dacă cuanta este foarte mică, frecvența comutării se mărește foarte mult și performanțele scad deoarece se consumă mult timp pentru salvare/restaurare registre ;

-se poate spune că algoritmul Round-Robin este un algoritm preemptiv care asigură un timp aproape egal de așteptare pentru toate procesele din sistem.

### 2.3.6. Alți algoritmi de planificare

Există unii algoritmi cu siruri de procese multinivel. Sirul READY este format din mai multe sub-siruri. Fiecare sub-sir are propriul algoritm de planificare. În schema de planificare apar siruri READY multiple :



### 3. GESTIUNEA PROCESELOR

#### 3.1. NOȚIUNI GENERALE DE PROCESE ȘI THREADURI

##### 3.1.1. Definiția procesului

Într-un sistem de calcul, un **proces** este un program în execuție; este deci o entitate activă (dinamică) a sistemului de operare și constituie unitatea de lucru a sistemului.

Înțelegerea diferenței între un program și un proces este importantă. Andrew Tannenbaum, al cărui tratat este baza documentării pentru specialiștii în SO, folosește o analogie pentru sesizarea acestei diferențe. Să considerăm un savant care coace un tort pentru aniversarea fricei sale. Are rețeta tortului și o bucătărie utilată și aprovisionată cu tot ce trebuie să intre în tort: ouă, zahăr, vanilie etc. În această analogie, savantul este procesorul (CPU=Central Processing Unit), rețeta este programul (ex. un algoritm exprimat într-o notație potrivită), ingredientele sunt datele de intrare. Procesul este activitatea savantului de a citi rețeta, de a introduce ingrediente, de a coace tortul.

Să ne închipuim acum că fiul savantului vine plângând că l-a înțepat o albină. Savantul întrește coacerea tortului înregistrând repede unde a ajuns în rețetă (starea procesului curent este salvată), caută o carte de prim ajutor (trece la un proces priorității cu alt program), și aplică instrucțiunile găsite în ea. Când primul ajutor a fost dat, savantul se întoarce la coacerea tortului și o continuă de unde a întrește-o.

Ideea este că un proces este o activitate de un anumit fel, cu un program, intrare, ieșire, stare etc. Este posibil ca un singur procesor să fie împărțit la mai multe proceze, cu ajutorul unui algoritm care să determine când să fie oprit procesul curent și să fie derulat altul.

Un *proces* este instanța de execuție a unui cod. Se utilizează și denumirea engleză de *task* (sarcină). Spațiul de adresă a unui proces cuprinde:

-*segmentul de cod* care conține imaginea executabilă a programului și este de tip RO (Read Only) și partajat de mai multe proceze;

-*segmentul de date* care conține date alocate dinamic sau static de către proces; nu este partajat și nici accesibil altor proceze; constă din:

- zona de date
- zona de rezervări (date neinitializate)
- zona de alocare dinamică;

-*segmentul de stivă* care nu este partajat și crește anumită momentul epuizării cantității de memorie pe care o are la dispoziție.

**Threadul, numit și fir de execuție**, este o subunitate a procesului, utilizat în unele SO.

##### 3.1.2. Starea procesului

Într-un sistem de calcul, un proces poate fi în diferite stări: pregătit, rulare, blocat, terminat

Când procesul este introdus în calculator, este în SISTEM (de obicei fișier sau un grup de fișiere pe un disc). Apoi planificatorul pe termen lung îl ia și-l introduce în coada de așteptare READY și atunci procesul este PREGĂTIT. Planificatorul pe termen scurt UC, conform unui algoritm de planificare, îl introduce în UC. Procesul este în RULARE. De aici, există trei posibilități:

-procesul s-a sfârșit și, după rularea în UC, trece în starea TERMINAT;

-dacă algoritmul folosit este preemptiv (de exemplu time-sharing , cu timp partajat), dacă după o cuantă de timp el mai are de rulat, este trecut din nou în coada de aşteptare READY în starea PREGĂTIT;

-dacă în timpul rulării are nevoie de o resursă externă (de obicei I/O), procesul este trecut în starea BLOCAT. După ce i s-a alocat resursa, trece din nou în starea PREGĂTIT.

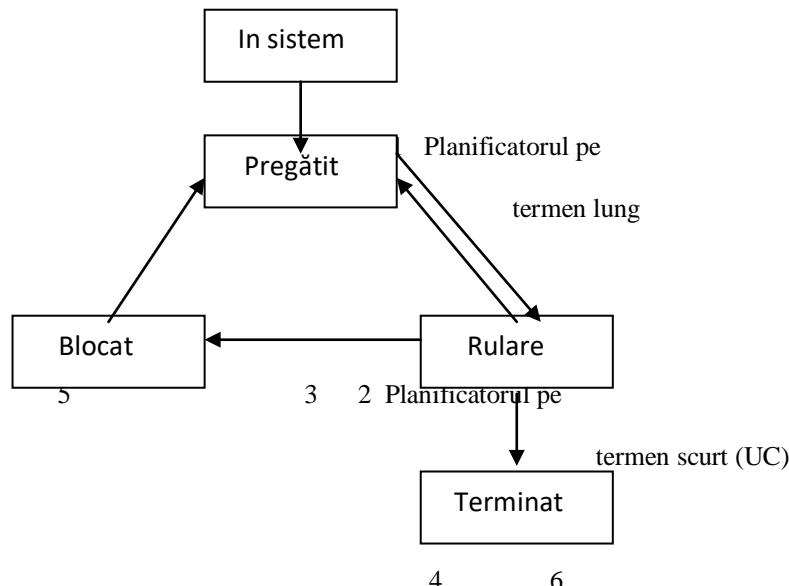


Fig. 3.1. Diagrama stărilor unui proces.

Dacă un proces este blocat temporar, el va trebui repornit mai târziu din exact aceeași stare în care se găsea când a fost oprit. În acest scop, toate informațiile despre proces trebuie salvate.

Unui proces îi se asociază de obicei o structură numită BCP (Bloc Control Proces), un descriptor de proces. În mod frecvent, descriptorii tuturor proceselor aflate în aceeași stare sunt înălțuiți în câte o listă. Vom avea o listă de procese în starea PREGĂTIT, una în starea BLOCAT etc.

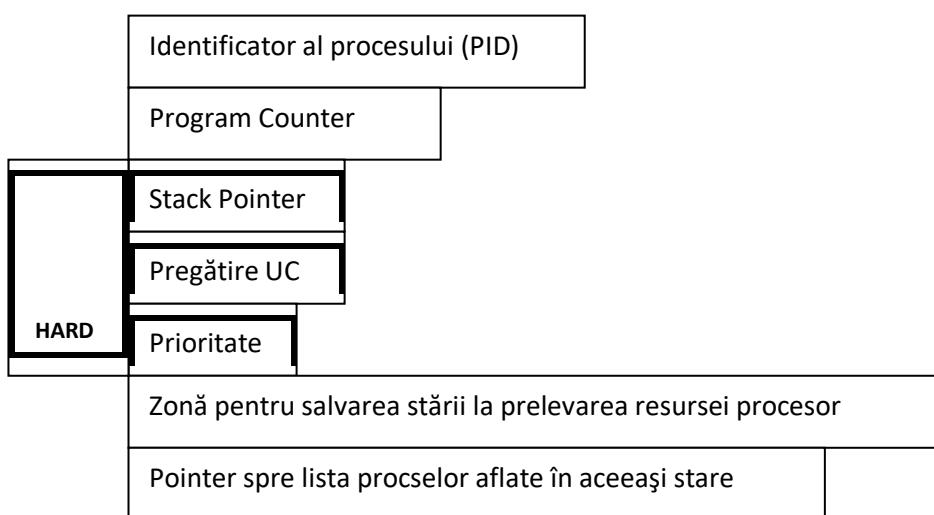


Fig 3.2. Structura BCP a unui proces.

### 3.1.3. Comutarea proceselor

Tranzitia intre două procese active într-un SO multitasking se numește comutarea proceselor (*process switch*) și are loc ca răspuns la un eveniment din sistem. Comutarea proceselor implică un cost (*overhead*) important datorită frecvenței cu care are loc în sistem și poate influența performanțele acestuia.

Eficiența operației de comutare a proceselor poate fi crescută prin prevederea unor facilități hard (seturi de registre multiple) sau printr-o modalitate de structurare a procesului (thread). Un thread (fir de execuție) este o subunitate a procesului, o diviziune a sa. Fiecare thread reprezintă un flux separat de execuție și este caracterizat prin propria sa stivă și prin stări hard (registre, flaguri).

Scopul principal al creării threadului este reducerea costului de comutare a proceselor. De vreme ce toate celelalte resurse, cu excepția procesorului, sunt gestionate de procesul care le înglobează, comutarea între threadurile care aparțin aceluiași proces implică doar salvarea stării hard și restaurarea stivei. Bineînțeles, comutarea între threadurile care aparțin unor procese diferite implică același cost de comutare.

Threadurile sunt un mecanism eficient de exploatare a concurenței programelor. Un program poate fi împărțit în mai multe părți.

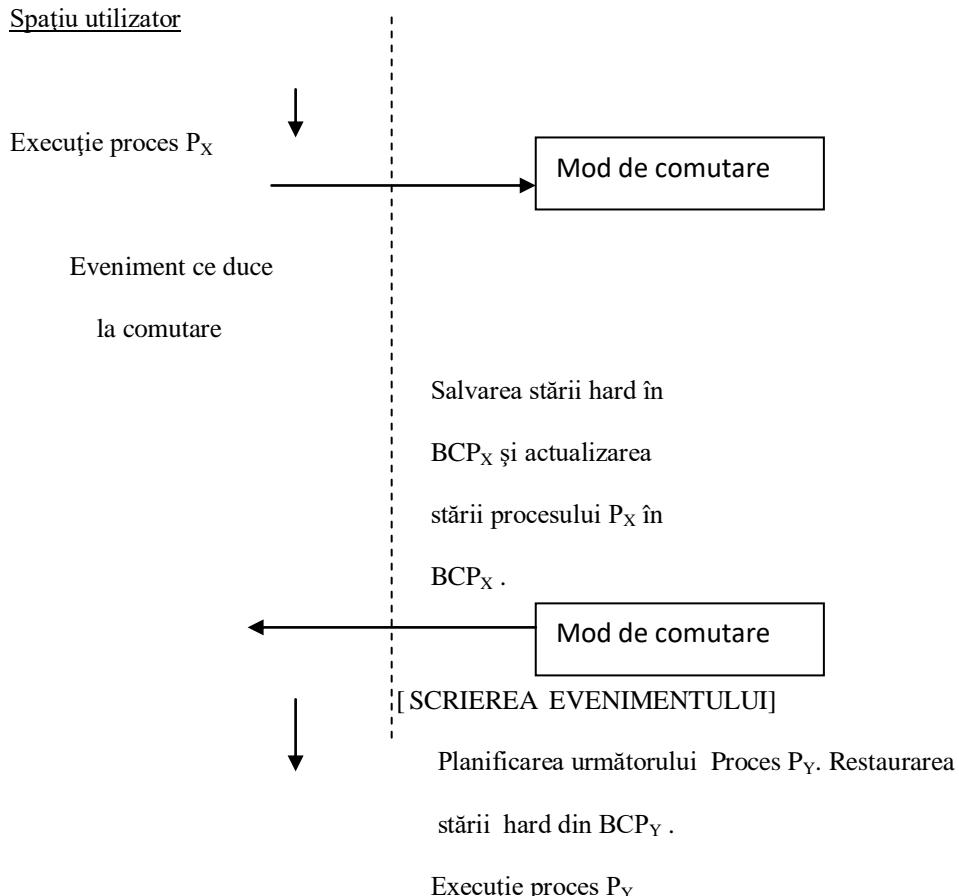


Fig.3.3 Schemă de comutare a proceselor.

### 3.1.4. Crearea și terminarea proceselor

În marea majoritate a sistemelor de operare un proces poate fi creat, în mod dinamic, de către alt proces. De obicei un proces *părinte* creează un proces *fiu*. În această modalitate de creare, există mai multe posibilități în dualitatea părinte-fiu:

- cele două procese execută, în mod independent, nesincronizat, același cod, având aceeași stivă și același segment de date;

- fiul execută alt segment de cod decât cel al părintelui, nesincronizat;

- părintele și fiul își sincronizează activitatea în sensul ei ori se execută întâi părintele și apoi fiul sau invers.

## 3.2. PROCESE ȘI THREADURI ÎN UNIX

### 3.2.1. Procese în UNIX

În sistemul de operare UNIX fiecare proces are un identificator numeric, numit identificator de proces PID. Acest identificator este folosit atunci când se face referire la procesul respectiv din interiorul programelor sau prin intermediul interpretorului de comenzi.

Două apele simple permit aflarea PID-ului procesului curent și al părintelui acestuia:

`getpid(void)` – pentru procesul curent;

`getppid(void)` – pentru procesul părinte.

Crearea unui proces se face prin apelul sistem:

`fork()`

Prin această funcție sistem, procesul apelant, numit părinte, creează un nou proces, numit fiu, care va fi o copie fidelă a părintelui. Procesul fiu va avea:

- propria lui zonă de date,

- propria lui stivă,

- propriul său cod executabil,

toate fiind copiate de la părinte, în cele mai mici detalii.

O modalitate de utilizare a apelului `fork()` este de a împărți codul unui proces în două sau mai multe procese care se vor executa în paralel. Acest lucru este utilizat în proiectarea și rularea proceselor și programelor parallele.

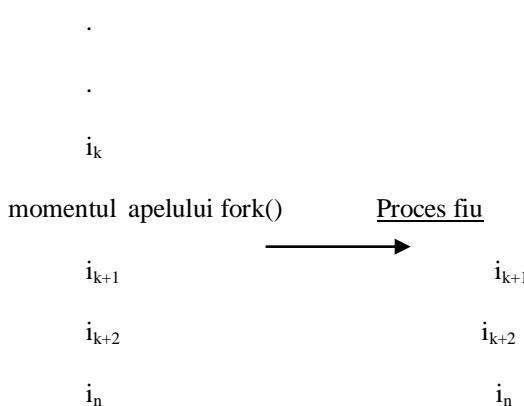
Exemplul 1.

#### Proces părinte

$i_1$

$i_2$

$i_3$



Exemplul 2. Se dă următoarea secvență de program:

```
fork();printf("A\n");
fork();printf("B\n");
fork();printf("C\n");
```

Presupunând că toate apelurile funcției fork() se execută cu succes, se cere:

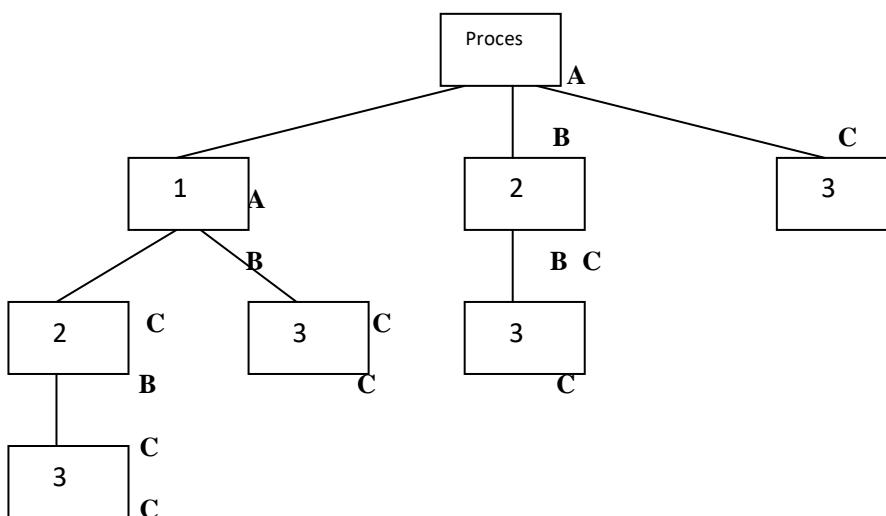
-Câte procese sunt create ?

-Să se traseze arborele părinte - fiu al proceselor create.

-Câte linii de text vor fi afișate la ieșirea standard ?

-Este posibil ca linia cu textul "B" să fie afișată înaintea liniei cu textul "A" ?

Sunt create 7 procese, plus procesul inițial, vor fi 8 procese. Cifrele arată după al cătelea fork() au fost create. După primul fork() se creează procesul cu cifra 1. Se va scrie A atât în procesul inițial cât și în cel final. După al doilea fork(), se creează procesul fiu notat cu cifra 2, adică fiecare proces creat până acum va avea câte un fiu, notat cu 2. După al treilea fork(), fiecare proces creat până acum va crea câte un nou fiu, notat cu 3. În general după  $n$  fork(), se produc  $2^{n-1}$  fii.



**Fig.3.4. Structura arborescentă a proceselor.**

Fiecare proces afișează liniile de text care urmează momentului creării sale:

linia care scrie „A” este afișată de  $2^1$  ori;

linia care scrie „B” este afișată de  $2^2$  ori;

linia care scrie „C” este afișată de  $2^3$  ori.

Numărul total de linii afișate este  $2^1 + 2^2 + 2^3 = 14$

În ceea ce privește ordinea de afișare a celor 14 linii, intervine nedeterminarea provenită din faptul că nu se știe cine va fi executat primul, părintele sau noul fiu creat. Valoarea returnată de fork() este:

-1, eroare, operația nu s-a putut execuția; 0, în codul fiului; pidf, în codul părintelui, unde pidf este identificatorul de proces al fiului nou creat.

Până acum am văzut că prin simplul apel al funcției fork() se creează un proces identic cu procesul părinte. Pentru a crea un nou proces care să ruleze un program diferit de cel al părintelui, se vor folosi funcțiile de tipul exec(), execl(), execlp(), execv(), execvp(), execll(), execvl(). Toate aceste funcții primesc ca parametru un nume de fișier care reprezintă un program executabil și reciclează lansarea în execuție a programului. Programul va fi lansat astfel încât se va suprascrie codul, datele și stiva procesului care apelează exec(), așa ca, imediat după acest apel, programul inițial să nu mai existe în memorie. Procesul va rămâne, însă, identificat prin același PID și va moșteni toate eventualele redirectări făcute în prealabil asupra descriptorilor de fișier.

În concluzie, lansarea într-un proces separat a unui program se face apelând fork() pentru crearea noului proces, după care, în porțiunea de cod executată de fiu, se va apela una din funcțiile exec().

În UNIX un proces se poate termina în mod normal sau anormal.

Terminarea normală poate fi realizată prin:

-revenire naturală;

-apelul sistem de tip exit().

Terminarea anormală se produce când:

-se apelează abort();

-procesul primește un semnal.

În marea majoritate a versiunilor UNIX, există două apeluri de terminare normală: exit(), \_exit(). Principalul rol al apelului exit() este să asigure la terminarea procesului tratarea corespunzătoare a operațiilor de introducere /extragere, golirea tampoanelor utilizate pentru acest proces și închiderea tuturor fișierelor deschise. \_exit() produce direct revenirea în nucleu, fără operațiile de la funcția exit().

Din cele prezentate până acum, nu se poate spune nimic despre sincronizarea *părinte-fiu*, adică despre cine termină primul sau cine se execuțiază primul. Apelurile wait() și waitpid() vin în întâmpinarea acestui neajuns. wait() este folosită pentru așteptarea de către părinte a fiului, deci se execuțiază mai întâi fiul și apoi părintele. waitpid() este folosită pentru așteptarea unui proces oarecare.

În UNIX procesele au un caracter dinamic. Ele se nasc, evoluează în sistem putând să naștere altor sisteme, și dispar. În felul acesta se creează o ierarhie dinamică de procese în sistem, care începe cu procesul 0 (*swapper*), continuă cu procesul 1 (*init*), proces ce dă naștere unor procese fiu. Procesul cu identificatorul 2, proces sistem, apare la unele implementări sub denumirea de *pagedaemon* și este responsabil de suportul pentru memorie virtuală. Terminarea forțată a execuției unui proces se realizează cu comanda *kill(option)(pid)*.

În general, în execuția părinte –fiu, există două situații:

- a) fiul se termină înaintea părintelui;
- b) părintele se termină înaintea fiului.

În primul caz, între momentul în care se termină fiul și momentul în care el este distrus, procesul fiu este în starea *zombie*. De obicei, după terminarea execuției fiului, părintele execută un wait() și scoate fiul din starea *zombie*.

În al doilea caz, în momentul terminării părintelui, nucleul SO este acela care examinează dacă părintele a avut copii. Dacă da, părintele nou al acestor fiu va fi procesul init(), nelăsând fiilor să fie în starea zombie.

Revenind la stările unui proces, sistemul de operare UNIX distrugе următoarele stări de bază:

- execuție în mod utilizator (USER);
- execuție în mod nucleu (KERNEL);
- gata de execuție (READY);
- în aşteptare (BLOCAT)
- zombie.

### 3.2.2. Threaduri în UNIX

În UNIX-ul tradițional nu sunt definite threadurile. Ele au apărut odată cu standardul POSIX care asigură portabilitatea între platforme hard diferite. Iată o comparație între procesele UNIX și threadurile POSIX (pthread):

Caracteristica	Procese UNIX	Pthreaduri POSIX
portabilitate	fork() standard	interfață standard portabilitate
Cost de creare	Mare	Relativ mic
Cost de comutare	Mare	Foarte mic
Spațiu de achesă	Separat	Partajat
Memorie partajată	Partajare explicită	Partajare implicită
Obiecte de excludere mutuală	Semafoare, mutexuri UNIX	Semafoare, mutexuri POSIX
Fișiere și stringuri de I/O	Tabele de fișiere separate	Tabele de fișiere unice

Un pthread se creează cu ajutorul funcției int pthread\_create (listă de parametri).

### 3.3. PROCESE ȘI THREADURI ÎN WINDOWS

În sistemele WINDOWS, entitatea de alocare a timpului procesoarelor este threadul. Fiecare proces conține cel puțin un thread de execuție, numit și threadul principal, și poate crea threaduri noi. Un proces WINDOWS are o dualitate de identificare și anume are două entități de identificare:

- handle*, care este o intrare în tabelul de resurse al sistemului;
- identifier (id)*, un număr unic atribuit unui proces (așemănător PID-ului din UNIX).

Această dualitate de identificare îngreunează lucrul cu procesele WINDOWS, în sensul că unele funcții cer ca parametru handle-ul procesului, altele identifierul acestuia.

#### 3.3.1. Procese în WINDOWS

Accesul programatorului la funcțiile SO este posibil prin intermediul unei interfețe, numită API (*Application Programming Interface*) care conține definiția tipurilor de date și funcțiile de apel ale sistemului.

Un proces tată creează un proces fiu prin intermediul apelului funcției `bool create process(lista parametri)`. Se poate considera că, în mare, această funcție are funcționalitatea combinației de apeluri `fork – exec` din UNIX. Se creează un proces nou, împreună cu threadul primar, care execută un segment de cod specificat prin numele fișierului ce conține acest segment. Valoarea returnată de funcția `createprocess` este de tip boolean și înseamnă TRUE (succes) și FALSE (eroare).

**Crearea threadurilor WINDOWS.** Crearea unui thread nou într-un proces necesită definirea unei funcții pe care threadul să o execute, urmată de apelul unei funcții `createthread` cu sintaxa `handle create thread (lista de parametri)`. Această funcție returnează handle-ul threadului nou creat.

## 4. COMUNICATIA ȘI SINCRONIZAREA ÎNTRE PROCESE

Procesele dintr-un SO pot fi:

- a) procese independente;
- b) procese cooperante.

a) Dacă execuția unui proces nu poate afecta sau nu este afectată de către execuția altor procese din sistem, atunci procesul este independent. Procesele independente au următoarele caracteristici:

- pot fi opriți și reporni fără a genera efecte nedorite;
  - sunt deterministe, adică rezultatele depind numai de starea de intrare;
  - nu se află niciodată în același stare ca și alte procese din sistem;
  - nu folosesc date, variabile, în comun cu alte procese;
  - sunt reproductibile, rezultatele sunt aceleași pentru aceleași condiții de intrare.
- b) În caz contrar, procesele sunt cooperante; au toate caracteristicile de mai sus negate.

### 4.1. PROBLEMA SECTIUNII CRITICE ȘI A EXCLUDERII MUTUALE

Cazul cel mai frecvent în care mai multe procese comunică între ele este acela prin intermediul variabilelor partajate. Dar accesul nerestriționat a două sau mai multe procese la resurse partajate poate produce erori de execuție.

Exemplu:

Un proces execută o operație de incrementare a variabilei partajate x după secvența cod:

LOAD R x /încărcarea lui x în registrul R;

INC R /incrementarea lui R;

STORE x R /memorarea valorii în X.

Presupunem că arhitectura calculatorului este cea de load/store, specific procesoarelor de mare viteză cum sunt și cele din arhitectura RISC.

Variabila locală x, în arhitecturi parallele, se numește *temporar inconsistentă*, între variabila locală și variabila globală corespunzătoare. Acest tip de inconsistenta temporară este frecvent întâlnită în execuția programelor la arhitecturi secvențiale (Neumann), dar nu prezintă nici un pericol pentru execuția programelor secvențiale normale ale căror variabile nu sunt partajate între procese concurente.

Este important de observat că inconsistenta temporară între o variabilă globală și copia ei locală reprezintă una din principalele cauze care produc erori atunci când o astfel de variabilă este accesată concurrent de procese concurente.

Din acest exemplu reies două condiții necesare pentru eliminarea erorilor datorate execuției concurente a mai multor procese:

1) Secvența de instrucțiuni din care este alcătuită operația de actualizare a unei variabile partajate trebuie să fie executată de un proces ca o *operație atomică*, neintreruptibilă de către alte procese sau chiar de SO.

2) Operația de actualizare a unei variabile partajate executată de un proces trebuie să inhibe execuția unei alte operații asupra aceleiași variabile executate de alt proces. Este deci necesară serializarea operațiilor asupra variabilelor partajate, serializare care se obține prin *excludere mutuală* a acceselor la variabila partajată.

În acest sens actualizarea unei variabile partajate poate fi privită ca o *secțiune critică*. O secțiune critică este o secvență de instrucțiuni care actualizează în siguranță una sau mai multe variabile partajate. Când un proces intră într-o secțiune critică, el trebuie să execute complet toate instrucțiunile secțiunii critice, înainte ca alt proces să o poată accesa. Numai procesului care execută o secțiune critică îi este permis accesul la variabila partajată, în timp ce tuturor celorlalte procese le este interzis accesul. Acest mecanism este denumit *excludere mutuală*, deoarece un proces exclude temporar accesul altor procese la o variabilă partajată.

Soluționarea problemei excluderii mutuale trebuie să îndeplinească următoarele cerințe:

-să nu presupună nici o condiție privind viteza de execuție sau prioritatea proceselor care accesează resursa partajată;

-să asigure că terminarea sau blocarea unui proces în afara secțiunii critice nu afectează în nici un fel toate celelalte procese care accesează resursa partajată corespunzătoare secțiunii critice respective;

-atunci când unul sau mai multe procese doresc să intre într-o secțiune critică, unul din ele trebuie să obțină accesul în timp finit.

Mecanismul de bază pe care-l urmărește un proces este:

- protocol de negociere / învingătorul continuă execuția;
- secțiune critică / utilizarea exclusivă a resursei;
- protocol de cedare / proprietarul se deconectează.

Soluționarea corectă a excluderii mutuale nu este o problemă trivială. Primul care a soluționat această problemă a fost matematicianul olandez Decker. Soluția lui, însă, nu este valabilă decât pentru două procese. Soluții de excludere mutuală au mai dat Peterson și Lampert, soluții pur soft, dar a căror eficiență este discutabilă căci nu rezolvă în întregime toate cerințele. Implementarea eficientă a excluderii mutuale se poate realiza prin suport hard (validare/invalidare de intreruperi sau instrucțiuni *Test and Set*). Aceste facilități hard sunt utilizate pentru implementarea unor mecanisme de sincronizare (obiecte de sincronizare) ca semafoare, mutexuri, bariere, monitoare etc.

#### **4.1.1. Suporțul hardware pentru implementarea excluderii mutuale**

##### **4.1.1.1. Invalidarea / validarea intreruperilor**

Instrucțiunile de invalidare/validare a intreruperilor (DI/EI) sunt disponibile în toate procesoarele. Secvența care asigură accesul exclusiv al unui singur proces la o resursă este:

- DI / invalidare intreruperi;
- Secțiune critică;
- EI / validare intreruperi.

Prin invalidarea intreruperilor se ajunge la blocarea celorlalte procese. Este un mecanism prea dur, un model pesimist de control al concurenței, cu următoarele dezavantaje:

-se blochează și activitatea altor procese (numite victime inocente) care nu au nici o legătură cu secțiunea critică respectivă;

-dacă se execută de către useri, se poate bloca sistemul iar dacă sunt executate din kernel, apare un cost de timp suplimentar (overhead) de comutare a modului de execuție kernel/user.

##### **4.1.1.2. Instrucțiunea Test and Set (TS)**

În principiu, instrucțiunea TS are ca operand adresa unei variabile de control și execută următorii pași:

-compară valoarea operandului cu o valoare constantă (de exemplu 0 pentru BUSY) și setează flagurile de condiție ale procesorului;

-setează operandul la valoarea BUSY .

Acești pași sunt execuția ca o operațiune unică, indivizibilă, numită operație atomică.

Subinstrucțiunea cod, TS, poate fi scrisă astfel:

LOAD R, operand

CMP R, USY  $\Rightarrow$  flag ZERO=1 dacă R=BUSSY

STORE operand,BUSY

#### 4.1.1.3. Protocole de aşteptare în excluderea mutuală

Avem două tipuri de protocole:

- a) *(busy-wait)*
- b) *(sleep-wait)*
- a) **BUSY-WAIT** (aşteptare ocupată)

Folosind instrucțiunea TS acest model are următoarea implementare:

ADRL TS (acces)

JZ ADRL/aşteptare cât variabila acces=BUSY

BUSY-WAIT este simplu de implementat dar are dezavantajul unei eficiențe scăzute, datorită faptului că se consumă timp de execuție al procesorului de către un proces care nu avansează.

- b) **SLEEP-WAIT** (aşteptare dormantă)

În acest tip de protocol, procesul care nu are resursă disponibilă este suspendat (adormit) și introdus într-o coadă de aşteptare. Apoi este trezit, când resursa devine disponibilă.

În sistemul uniprocesor este folosit totdeauna SLEEP-WAIT, deoarece nu se consumă timp procesor inutil.

În sistemul multiprocesor se folosesc amândouă protocolele și chiar o combinație între ele.

#### 4.1.1.4. Mecanisme de sincronizare între procese (obiecte de sincronizare)

##### a) Semafoare.

Semafoarele sunt obiectele de sincronizare cele mai des și mai mult folosite. Ele au fost introduse de Dijkstra în 1968.

Un semafor este de fapt o variabilă partajată care poate primi numai valori nenegative. Asupra unui semafor se pot executa două operații de bază:

- decrementarea variabilei semafor cu 1 (DOWN);
- incrementarea variabilei semafor cu 1 (UP).

Dacă o operație DOWN este efectuată asupra unui semafor care este mai mare decât zero, semaforul este decrementat cu 1 și procesul care l-a apelat poate continua. Dacă, din contra, semaforul este zero, operația DOWN nu poate fi efectuată și spunem că procesul așteaptă la semafor, într-o coadă de aşteptare, până când un alt proces efectuează operația UP asupra semaforului, incrementându-l.

Exemplu. Avem un complex cu terenuri de handbal și 20 echipe care joacă 10 meciuri. 1meci = 1proces. Într-un coș mare (semafor) există 7 mingi. Semaforul ia valori între 0-7. Fiecare 2 echipe iau câte o minge din coș, efectuând 7 operații DOWN. Semaforul este 0 (nu mai sunt mingi în coș) și celelalte 6 echipe așteaptă o minge în coș, deci o operație UP.

Pentru implementare, operațiile DOWN și UP vor fi înlocuite cu două primitive wait(s) și signal(s), în felul următor:

wait(s) - încearcă să decrementeze valoarea variabilei semafor s cu 1 și dacă variabila este 0 procesul rămâne în aşteptare până când s ≠ 0.

signal(s) – incrementă cu 1 semaforul s.

Implementarea semafoarelor în protocolul BUSY-WAIT

```
wait(s)
```

```
{while (s==0);
```

```
s--;}
```

```
signal(s++);}
```

Operația signal(s) este atomică (indivizibilă).

Operația wait(s) are două regimuri:

atomică, dacă  $s \neq 0$ ;

neatomică, dacă  $s=0$ .

Este normal ca, dacă semaforul este ocupat ( $s=0$ ), operația wait(s) să nu fie atomică; în caz contrar ar împiedica alte operații, inclusiv signal(s), executate de un alt proces care să acceseze variabila s pentru a o elibera.

Implementarea cu BUSY-WAIT a semafoarelor are dezavantajul unui consum de timp procesor de către procesele aflate în aşteptare. Un alt dezavantaj este posibilitatea ca un proces să fie amânat un timp nedefinit (indefinit, postponement) în obținerea unui semafor, datorită faptului că nu este nici o ordonare între procesele care așteaptă la semafor. Astfel există posibilitatea ca un proces să fie împiedicat un timp nedefinit de a obține un semafor datorită aglomerării provocate de alte procese. Un astfel de fenomen de blocare se numește *livelock* iar procesul afectat este numit *strivit (starved)*.

Implementarea semafoarelor în protocolul SLEEP-WAIT

Procesele care așteaptă la semafor stau într-o coadă de aşteptare (FIFO).

```
wait(s)
```

```
{if(s==0) procesul trece în stare suspendat;
```

```
else s--;}
```

```
signal(s)
```

```
{if(coada nu este goală) planifică un proces din coadă ;
```

```
else s++ ;
```

În loc să intre în aşteptare ocupată, atunci când semaforul este ocupat, procesul apelant este suspendat și trecut într-o coadă de aşteptare asociată semaforului apelat. Trebuie însă ca și procesul dormant (suspendat) să afle când semaforul a devenit liber. Acest lucru se realizează prin intermediul operației signal. Această operație încearcă mai întâi să trezească un proces din coada de aşteptare și să-l treacă în stare de execuție (*suspendat→run*). Numai dacă coada de aşteptare este goală, incrementază semaforul.

**b) Mutex-uri**

Denumirea MUTEX vine de la *mutual exclusion*. Mutex-ul este un caz particular de semafor și este utilizat pentru accesul mai multor procese la o singură resursă partajată. Operațiile care se execută asupra unui mutex sunt :

- operația de acces și obținere a mutex-ului, notată *cu ocupare (mutex)* sau *lock(mutex)* ;
- operația de eliberare a mutex-ului, notată *cu eliberare (mutex)* sau *unlock(mutex)*.

### c) Evenimente (semnale)

Unul dintre primele mecanisme de comunicare între procese este reprezentat de *semnale* sau *evenimente*. În UNIX se numesc semnale iar în WINDOWS se numesc evenimente. Ele anunță apariția unui eveniment și pot fi trimise de un proces altui proces, de un proces același proces sau pot fi trimise de kernel. Momentul apariției unui semnal este neprecizat, el apărând asincron.

#### Semnale UNIX

În sistemul de operare UNIX, semnalele pot proveni de la nucleu sistemului, prin care se notifică evenimente hardware, sau pot fi generate soft, de unele procese, pentru a notifica un eveniment asincron. Un semnal este reprezentat printr-un număr și un nume. În UNIX semnalele dintre procese au denumiri simbolice, SIGUSR 1 și SIGUSR 2.

Un proces care primește un semnal poate să acționeze în mai multe moduri:

- să ignore semnalul;
- să blocheze semnalul; în acest caz semnalul este trecut într-o coadă de așteptare până când este deblocat;
- să recepționeze efectiv semnalul.

Procesul stabilește ce semnale sunt blocate, folosind o masă de semnale, în care fiecare bit corespunde unui număr de semnal. În standardul POSIX există numere între 33 și 64, deci 31 semnale. Un semnal poate să apară în următoarele cazuri:

- la acționarea unei taste a terminalului;
- la apariția unei intreruperi hardware (împărțire prin zero, adresă inexistentă etc);
- atunci când un proces trimit un semnal altui proces sau thread, folosind funcțiile:
  - a) `sigqueue()` – se transmite un semnal unui proces;
  - b) `kill()` – se transmite un semnal de terminare a unui proces;
  - c) `pthread` – se transmite semnal de terminare a unui thread.

#### Evenimente WINDOWS

În SO WINDOWS, un eveniment este un obiect de sincronizare care poate fi *semnalat* sau *nesemnalat*.

Există două tipuri de evenimente:

- evenimente cu resetare manuală (evenimente care trecute în stare *semnalat* rămân în această stare până când sunt trecute în mod explicit în starea *nesemnalat*);
- evenimente cu autoresetare.

În WINDOWS un proces poate crea un eveniment folosind funcția CREATE EVENT.

Un proces în aşteptarea unui eveniment poate executa una din funcțiile de aşteptare WAIT FOR SINGLE OBJECT sau WAIT FOR MULTIPLE OBJECT.

Evenimentele sunt utilizate pentru notificarea unui proces sau thread despre apariția unui alt eveniment.

Evenimentele creează puncte de sincronizare între proceze sau threaduri.

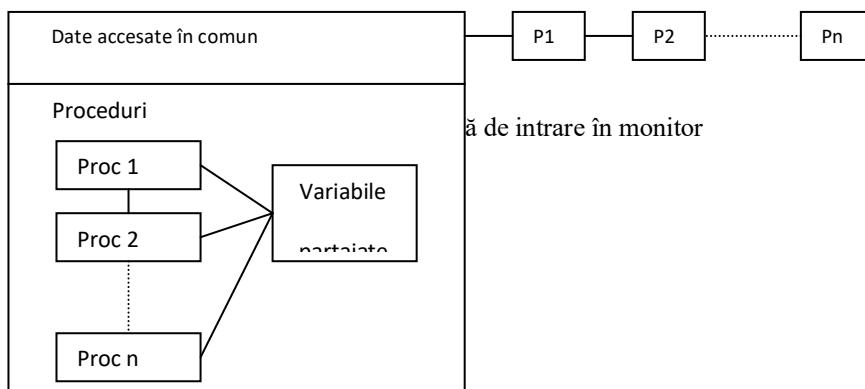
**d) Monitoare.** Monitoarele sunt mecanisme de sincronizare între proceze concurente care pot suporta abstractizarea datelor.

Prin apariția monitoarelor (HOARE, 1976), se reușește să se acopere unul din neajunsurile principale ale celor mai utilizate mecanisme de sincronizare, semafoarele, care nu suportă abstractizarea datelor.

Sintaxa monitorului este asemănătoare cu cea a unei clase, cuvântul CLASS fiind înlocuit cu MONITOR.

Este de remarcat faptul că într-un monitor se asigură excluderea mutuală, astfel că la un moment dat un singur proces poate fi activ.

Un monitor este bazat pe modularitate și încapsulare.



**Fig. 4.1. Structura unui monitor.**

Structura unui monitor constă în trei părți:

- în prima parte se declară numele monitorului și variabilele locale;
- în a doua parte se declară toate procedurile;
- în cea de a treia parte se inițializează toate variabilele acestui monitor.

În general, procedurile unui monitor sunt constituite din principalele funcții care se întâlnesc într-un sistem de operare. De exemplu, o procedură tipică este intrarea într-o secțiune critică.

#### 4.2. INTERBLOCAREA (DEADLOCK)

Alături de problema secțiunii critice, interblocarea reprezintă una din principalele probleme ce trebuie să fie rezolvate în funcționarea unui SO. Mecanismul principal al interblocării constă în faptul că o resursă cerută de un proces este deținută de alt proces. Până când procesul care deține resursa o va elibera, apare o situație tipică de interblocare.

Deoarece interblocarea este strâns legată de resursele sistemului de operare, vom prezenta, mai întâi, câteva noțiuni referitoare la aceste resurse.

#### 4.2.1. Resurse

##### 4.2.1.1. Clasificarea resurselor din punct de vedere al interblocării

Cea mai importantă clasificare din punct de vedere al interblocării este:

- a) resurse partajabile ;
- b) resurse nepartajabile.

O resursă partajabilă poate fi utilizată în comun de mai mulți utilizatori. Un exemplu clasic în acest sens este citirea unui fișier de către mai mulți utilizatori.

O resursă nepartajabilă nu poate fi folosită în același timp de către mai mulți utilizatori. Un exemplu este imprimanta. Evident, mai mulți utilizatori nu pot tipări în același timp pe aceeași imprimantă. Numai resursele nepartajabile conduc la situații de interblocare; cele partajabile nu pun probleme din acest punct de vedere. Sistemul de operare, în procesul de control al interblocării, trebuie să aibă o gestiune doar a resurselor nepartajabile.

O altă clasificare a resurselor, importantă pentru interblocare, este:

- a) resurse cu un singur element;
- b) resurse cu mai multe elemente.

Pentru situații diferite de interblocare, această clasificare este foarte importantă, ea ducând la decizii diferite. Un exemplu tipic unde intervine această clasificare este graful de alocare a resurselor.

##### 4.2.1.2. Etapele parcurse de un proces pentru utilizarea unei resurse

În vederea utilizării unei resurse, un proces trebuie să execute următoarele etape:

a)Cererea de acces la resursă. Procesul formulează o cerere de acces la resursa respectivă. Dacă nu îi este repartizată imediat, intră într-o coadă de așteptare și va aștepta până când poate dobândi resursa.

b)Utilizarea resursei. Este etapa în care procesul a primit permisiunea de utilizare a resursei, ieșe din coada de așteptare și utilizează efectiv resursa.

c)Eliberarea resursei. Se eliberează resursa și se încheie utilizarea resursei de către proces.

Pentru implementarea de către SO a acestei operații, există tabele ale SO în care fiecarei resurse îi este asociat procesul ce o utilizează și, de asemenea, fiecare resursă are asociată o coadă de așteptare cu toate procesele care au făcut cerere de utilizare a resursei. De obicei, sistemele de operare implementează, pentru fiecare din etapele de mai sus, apeluri sistem sau semafoare.

#### 4.2.2. Condiții necesare pentru apariția interblocării

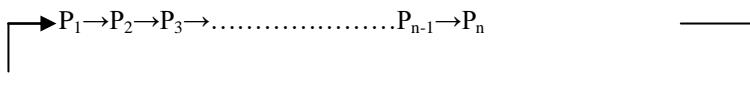
În anul 1971, Cofman a identificat patru condiții necesare care, dacă sunt îndeplinite simultan, pot conduce la apariția interblocării. Aceste condiții sunt:

a)Excluderea mutuală. Existența excluderii mutuale presupune că un proces a obținut resursa și există o coadă de așteptare a altor procese la aceeași resursă.

b)Ocupare și așteptare. Există cel puțin un proces care a obținut resursa dar care așteaptă și alte resurse suplimentare care, la rândul lor, sunt ocupate de alte resurse.

c) Imposibilitatea achizitionării forțate. Un proces nu poate achiziționa forțat o resursă aferentă altui proces decât după eliberarea resursei de către acel proces.

d) Așteptare circulară. Există un sir de procese  $P_1, P_2, \dots, P_n$ , toate în așteptare, în așa fel încât  $P_1$  așteaptă eliberarea resurse de către  $P_2$ ,  $P_2$  așteaptă eliberarea resursei de către  $P_3$ ..... $P_n$  așteaptă eliberarea resursei de către  $P_1$ .



#### 4.2.3. Graful de alocare a resurselor

Pentru descrierea stării de interblocaj este folosit un graf orientat, numit graful de alocare a resurselor. Nodurile grafului sunt alcătuite din procese și resurse. Vom nota procesele cu  $P$  și resursele cu  $R$ .

Resursa cu indice i :

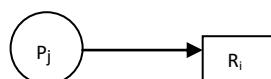


Procesul cu indice j :

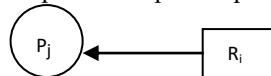


Arcele grafului sunt orientate și sunt de două feluri:

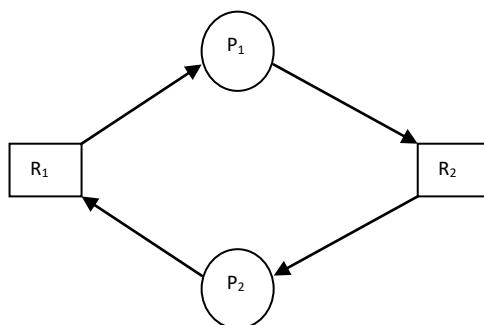
a) *arce cerere*, care reprezintă faptul că procesul  $P_j$  face o cerere de alocare a resursei  $R_i$  și așteaptă dobândirea ei;



c) *arce alocare*, care reprezintă faptul că procesului  $P_j$  i-a fost alocată resursa  $R_i$ .



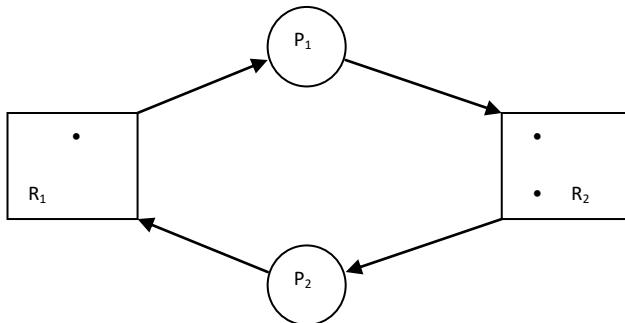
Cea mai importantă aplicație a acestui graf este legată de detecția stării de interblocaj. Pentru un graf alcătuit numai din resurse simple, existența unei bucle în graf înseamnă că în sistem a apărut o interblocaj. De exemplu, în graful următor.



**Fig.4.2. Graf de resurse.**

În graful de mai sus, procesului  $P_1$  îi este alocată resursa  $R_1$  și procesului  $P_2$  resursa  $R_2$ . Procesul  $P_1$  a făcut o cerere de alocare a resursei  $R_2$  deținută de  $P_2$ , iar  $P_2$  a făcut o cerere de alocare a resursei  $R_1$  deținută de  $P_1$ . Este o situație clară de interblocare. În graful din figura 4.2. această interblocare este vizibilă prin existența buclei.

De menționat că pentru grafurile cu resurse multiple existența buclei nu înseamnă o situație de interblocare. De exemplu, în cazul din figura 4.3., existența buclei nu înseamnă interblocare.

**Fig. 4.3. Graf fără situație de interblocare.**

#### 4.2.4. Rezolvarea problemei interblocării

Există două tipuri de metode pentru rezolvarea interblocării:

- metodă în care nu i se permite niciodată sistemului de operare să intre în interblocare;
- metodă în care se permite sistemului de operare să intre în interblocare și apoi se încearcă scoaterea sa din această stare

În cadrul primei metode se face *prevenirea* sau *evitarea* interblocării.

În cadrul celei de-a doua metode se utilizează un mecanism de *detectie* a interblocării și *revenire* din această stare.

##### 4.2.4.1. Prevenirea interblocării

Pentru a putea preveni interblocarea este suficient ca una din condițiile de apariție a acesteia să nu fie îndeplinită. Să vedem cum poate fi împiedecată îndeplinirea fiecărei din cele patru condiții.

- Excluderea mutuală

Pentru o resursă nepartajabilă nu este posibilă prevenirea interblocării prin neîndeplinirea condiției de excludere mutuală.

- Ocupare și așteptare

Pentru neîndeplinirea condiției de ocupare și așteptare sunt posibile două protocoale:

-un protocol în care fiecare proces își poate începe execuția numai după ce și-a achiziționat toate resursele necesare;

-un protocol în care unui proces nu i se permite să achiziționeze decât o resursă, achiziționarea resurselor suplimentare făcându-se cu eliberarea resurselor deja angajate.

Deși prin ambele protocoale se asigură neîndeplinirea condiției de așteptare și așteptare, totuși ele prezintă două mari dezavantaje:

-utilizarea resurselor este destul de redusă, în sensul că timpul cât o resursă este alocată unui proces și neutilizată este foarte mare;

-apare un proces de infometare, deoarece un proces nu poate să aștepte la infinit alocarea unei resurse.

c) Imposibilitatea achiziționării forțate

Neîndeplinirea acestei condiții înseamnă de fapt ca un proces să poată lua o resursă alocată altui proces în orice moment. Desigur, această achiziționare forțată a resurselor unui alt proces nu trebuie făcută haotic, ci în cadrul unui protocol de achiziționare forțată. Un exemplu de astfel de protocol este următorul : un proces care își achiziționează resurse în vederea execuției va putea lua forțat resurse de la alt proces numai dacă procesul respectiv este în așteptare. Acest protocol se aplică frecvent în cazul resurselor a căror stare poate fi ușor salvată și refăcută (ex. registrele procesorului, spațiul de memorie).

d) Așteptare circulară

Un algoritm simplu pentru eliminarea așteptării circulare este dat în continuare.

Se creează o corespondență biunivocă între toate resursele nepartajabile ale sistemului și mulțimea numerelor naturale, astfel încât fiecare resursă este identificată printr-un număr natural. De exemplu:

hard disc .....1

CD ROM.....2

imprimantă.....3

MODEM.....4

scanner.....5

Apoi, un proces poate cere resursa cu număr de ordine k , cu condiția ca să elibereze toate resursele cu indice mai mari decât k, adică k+1, k+2,.....n. În felul acesta se elimină posibilitatea de așteptare circulară.

În concluzie, pentru prevenirea interblocării se utilizează algoritmi care impun ca cel puțin una din condițiile necesare să nu fie îndeplinite. Acești algoritmi acționează prin stabilirea unor *restrictii* asupra modului în care se pot formula cererile de acces. Principalele dezavantaje ale acestei metode sunt gradul redus de utilizare a resurselor și timpul mare de așteptare a unui proces pentru o resursă.

#### **4.2.4.2. Evitarea interblocării**

Dacă pentru prevenirea interblocării se folosesc restricții asupra modurilor de formulare a cererilor pentru resurse, în evitarea interblocării se utilizează informații suplimentare referitoare la modul în care se face cererea de acces. Algoritmii de evitare diferă prin tipul și cantitatea acestor informații.

Se definesc două noțiuni: *stare sigură* și *secvență sigură*.

În cazul acestor algoritmi, fiecare proces trebuie să declare numărul maxim de resurse de fiecare tip de care ar putea avea nevoie. Algoritmul examinează mereu starea alocării resurselor pentru a avea certitudinea că nu va exista așteptare circulară.

#### Secvență sigură

Fie un sir de procese  $P_1, P_2, \dots, P_n$  exact în această ordine. Spunem că această secvență este sigură dacă pentru orice proces  $P_i$  cu  $(1 \leq i \leq n)$  s-ar cere numărul maxim de resurse, declarat inițial, atunci diferența între numărul maxim de resurse și numărul de resurse al procesului în acel moment nu depășește numărul resurselor obținute din însumarea resurselor disponibile cu resursele eliberate de procesele  $P_j$  cu  $j < i$ . Dacă nu se îndeplinește această condiție, atunci secvența este nesigură.

Stare sigură

Sistemul este într-o stare sigură dacă conține cel puțin o secvență sigură. De exemplu, fie secvența de procese:

$P_1 P_2 P_3 P_4 P_5$

	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$
Maxim resurse cerute	10	15	20	25	30
Cerere inițială	5	5	10	10	20

Total resurse = 60

Resurse disponibile =  $60 - 5 - 5 - 10 - 10 - 20 = 10$

Să analizăm secvența  $P_1 P_2 P_3 P_4 P_5$ .

$P_1$  la cerere maximă de resurse ar avea nevoie de

$10 - 5 = 5$  resurse < 10 resurse disponibile

$P_2$   $15 - 5 = 10$  resurse < 5(eliberate de  $P_1$ ) + 10(disponibile)

$P_3$   $20 - 10 = 10$  resurse <  $5(P_1) + 5(P_2) + 10$ (disponibile)

$P_4$   $25 - 10 = 15$  resurse <  $5(P_1) + 5(P_2) + 10(P_3) + 10$  (disponibile)

$P_5$   $30 - 20 = 10$  resurse <  $5(P_1) + 5(P_2) + 10(P_3) + 10(P_4) + 10$ (dispon.)

Deci această secvență este sigură.

Să analizăm secvența  $P_4 P_5 P_1 P_2 P_3$ .

$P_4$   $25 - 10 = 15 > 10$  (resurse disponibile)  $\Rightarrow$  secvență nesigură.

### a) Algoritmul bancherului

Un algoritm clasic de evitare a interblocării, bazat pe noțiunea de secvență sigură, este algoritmul bancherului. Se numește așa deoarece poate fi folosit în sistemul bancar la plata unor sume către diferiți clienți ai băncii, plată care trebuie să lase mereu banii într-o stare sigură.

Pentru a putea aplica acest algoritm trebuie să se cunoască de la început numărul maxim de resurse cerute de fiecare proces. Apoi, la fiecare cerere a unor resurse noi, se aplică algoritmul pentru a vedea dacă această cerere duce la o stare sigură sau nesigură. Dacă e sigură, cererea este acceptată, dacă nu e sigură, cererea nu este acceptată și procesul rămâne în aşteptare.

Structurile de date folosite de algoritm sunt:

n - numărul de procese din sistem

m - numărul de tipuri resursă

-disponibil[m] – un vector care indică numărul de resurse disponibile pentru fiecare tip în parte;

-maxim [n][m] – o matrice care arată numărul maxim de cereri ce pot fi formulate de către fiecare proces;

-alocare[n][m]- o matrice care arată numărul de resurse din fiecare tip de resurse care este alocat fiecărui proces;

-necesar[n][m] – o matrice care arată numărul de resurse care ar mai putea fi necesare fiecărui proces.

Dacă  $necesar[i][j] = t$ , atunci procesul  $P_i$  ar mai avea nevoie de  $t$  elemente din resursa  $r_j$ .

Avem relația:

$$necesar[i][j] = maxim[i][j] - alocare[i][j]$$

-cerere[n][m]–matricea cererilor formulate de un proces

Dacă  $cerere[i][j]=t$ , atunci procesul  $P_i$  dorește  $t$  elemente din resursa  $r_j$ .

Algoritmul banchetului are următorii pași:

#### Pas 1

Procesul  $P_i$  formulează o cerere de resurse. Dacă linia  $i$  din matricea cerere este mai mare decât linia  $i$  din matricea necesar, atunci este eroare, deci nu se trece la pasul 2.

Precizăm că  $V_1[n] < V_2[n]$ , dacă  $V_1[i] < V_2[i]$  pentru oricare  $i=1....n$

if cerere[i][x]<=necesar[i][x]  $\forall x=\overline{1.....m}$ ,  $\rightarrow$  Pas2

else eroare

#### Pas 2

if cerere[i][x]<=disponibil[x] se trece la pas 3

else wait (resursele nu sunt disponibile)

#### Pas 3

Se simulează alocarea resurselor cerute de procesul  $P_i$ , stările modificându-se astfel:

disponibil[x]=disponibil[x]-cerere[i][x];

alocare[i][x]=alocare[i][x]+cerere[i][x];  $\forall x=1....m$

necesar[i][x]=necesar[i][x]-cerere[i][x];

#### Pas 4

În acest moment se testează dacă noua stare este sigură sau nu. În acest scop se mai utilizează doi vectori:

lucru[m]

terminat[n]

Subpasul 1

Se inițializează acești vectori astfel:

`lucru[i]=disponibil[i];`

`terminat[i]=fals`

`pentru i=1,2,3.....n`

#### Subpasul 2

Se caută o valoare  $i$  astfel încât:

`terminat[i]=fals;`

`necesar[i][x]<=lucru[x];`

Dacă nu există, se trece la subpasul 4.

#### Subpasul 3

Se simulează încheierea execuției procesului, deci se execută:

`lucru[x]=lucru[x]+alocare[i][x]`

`terminat = true;`

Se trece la subpasul 2.

#### Subpasul 4

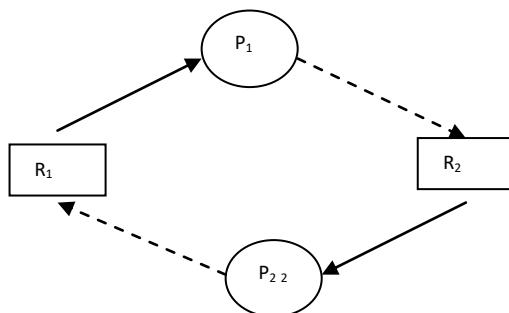
Dacă:

`terminat[i]=true pentru  $\forall i=1.....n$ ,`

Atunci sistemul este într-o stare sigură. Algoritmul bancherului poate fi utilizat în orice sistem de alocare a resurselor, pentru determinarea stărilor sigure, având un mare grad de generalitate. Principalul său dezavantaj este numărul mare de operații pe care îl cere.

#### **b) Folosirea grafului de alocare a resurselor**

O altă metodă pentru determinarea unei stări sigure este folosirea grafului de alocare a resurselor. Față de graful prezentat anterior, elementul nou este *arcul revendicare*. Acest arc arată că este posibil ca procesul  $P_i$  să revindecă în viitor resursa  $R_j$ . Ca mod grafic de reprezentare, el este trasat cu linie întreruptă. Când procesul  $P_i$  cere resursa  $R_j$ , arcul revendicare ( $P_iR_j$ ) se transformă în arc cerere ( $P_iR_j$ ). Când resursa  $R_j$  este eliberată de procesul  $P_j$ , arcul alocare ( $P_jR_i$ ) este transformat în arc revendicare ( $P_jR_i$ ). Pentru a determina stările nesigure, se caută în graf bucle în care intră arcuri revendicare. O astfel de buclă reprezintă o stare nesigură. Exemplu:



**Fig. 4.4. Graf de alocare cu arcuri revendicative.**

În graful de mai sus, procesului  $P_1$  îi este alocată resursa  $R_1$ , iar procesului  $P_2$  îi este alocată resursa  $R_2$ . În același timp procesul  $P_1$  poate să ceară în viitor resursa  $R_2$  ceea ce în graf se concretizează prin arcul revendicare ( $P_1R_2$ ). La fel, Procesul  $P_2$  poate revendica resursa  $R_1$  prin arcul ( $P_2R_1$ ). Se observă că există o buclă ( $P_1R_2P_2R_1$ ), ceea ce face ca aceste revendicări să conducă la o stare nesigură.

#### 4.2.4.3. Detectarea interblocării și revenirea din ea

Atunci când, din diverse motive, nu se pot aplica algoritmii de prevenire și evitare a interblocării, se intră în această stare. În acest caz trebuie să se execute alte două tipuri de algoritmi:

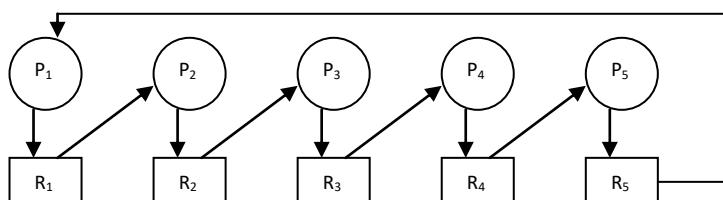
- algoritmi de detecție care să informeze sistemul asupra momentului în care s-a ajuns la această stare;
- algoritmi de revenire din starea de interblocare.

#### c) Detectia interblocării

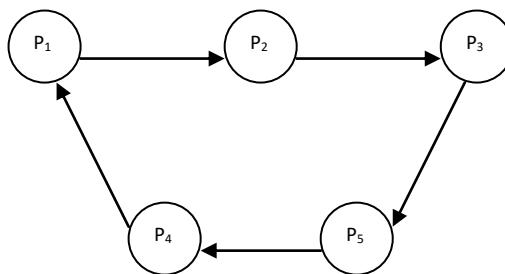
În acest scop se folosesc doi algoritmi:

- un algoritm foarte asemănător cu algoritmul bancherului;
- un graf de alocare a resurselor de tip WAIT FOR.

Deoarece algoritmul bancherului a fost discutat în amănunt, ne vom ocupa doar de graful WAIT FOR. Acest graf se obține din graful de alocare a resurselor prin eliminarea nodurilor de tip resursă ( $R_j$ ) și contopirea arcelor corespunzătoare. În acest caz, un arc ( $P_iP_j$ ) arată că  $P_i$  așteaptă ca  $P_j$  să elibereze resursa care îi este necesară.

**Fig. 4.5.Graf de alocare a resurselor.**

În acest caz, existența buclei nu înseamnă interblocare.

**Fig.4.6.Graf WAIT FOR.**

Algoritmul WAIT FOR se poate aplica numai pentru resurse simple.

Se observă că bucla existentă în graful de alocare a resurselor este mai bine pusă în evidență în graful WAIT FOR. Dar avantajul principal al utilizării grafului WAIT FOR constă în costurile mai mici pentru detecția unei bucle, deci a interblocării.

Indiferent de tipul algoritmului de detecție a interblocării, se pune problema cât de des trebuie să fie acesta apelat. Există, în general, două modalități de apelare:

- a)apelarea algoritmului ori de câte ori se formulează o cerere de resurse;
- b)apelarea algoritmului la intervale regulate de timp.

Prima modalitate detectează rapid interblocarea dar presupune un substanțial consum suplimentar de timp de calcul.

În a doua modalitate, se alege un interval de timp la care se apelează algoritmul de detectare. Desigur, un interval scurt va introduce costuri suplimentare iar la un interval lung este posibil ca starea de interblocare să fie detectată foarte târziu, cu consecințe nedorite în funcționarea sistemului.

#### d) Revenirea din interblocare

Revenirea din interblocare se poate face în două moduri:

- manual*, făcută de către operatorul sistemului;
- automat*, executată de anumite programe ale sistemului de operare.

În general, există două metode de revenire din starea de interblocare:

- 1)-Prin *dispariția așteptării circulare*; în acest caz se forțează terminarea unor procese.
- 2)-Prin *negarea achiziției forțate*; în acest caz procesele pot să achiziționeze resurse de la alte procese.

1)Folosirea metodei de dispariție a așteptării circulare are două forme:

-Încheierea forțată a tuturor proceselor interblocați; în acest fel se revine sigur din starea de interblocare dar se pierd toate rezultatele fiecărui proces implicat în interblocare.

-Încheierea forțată a căte unui singur proces implicat în interblocare; în acest caz se încheie forțat un proces și apoi se apelează algoritmul de detecție a interblocării pentru a vedea dacă mai persistă starea de interblocare; dacă da, se încheie forțat alt proces și se apelează din nou algoritmul de detecție, operația continuând până la dispariția stării de interblocare. Avantajul față de prima metodă este că nu se pierd rezultatele de la toate procesele. Dezavantajul constă în timpul suplimentar, pentru că, după terminarea forțată a unui proces, trebuie apelat algoritmul de detecție.

O altă problemă a acestei metode este determinarea procesului sau proceselor care trebuie terminate forțat. Pot intra în discuție mai mulți factori:

- prioritatea procesului;
- numărul resurselor care ar mai fi necesare procesului pentru a-și încheia normal execuția;
- numărul resurselor care au fost deja folosite de un proces;

-timpul necesar până la terminarea normală a procesului.

Din enumerarea acestor factori, se observă că este necesar un algoritm pentru alegerea procesului sau proceselor care vor fi terminate forțat. De obicei se alege factorul ce necesită un timp minim. Cel mai frecvent se utilizează alegerea după prioritate, cu atât mai mult cu cât prioritatea proceselor este deja calculată din algoritmii de planificare a procesorului.

2) Permiterea achiziționării forțate a resurselor de la proceze este a doua metodă de revenire din interblocare. Problemele care apar în acest caz sunt:

- alegerea „victimelor”, ceea ce înseamnă și selectarea resurselor care vor fi achiziționate forțat;
- continuarea procesului căruia i s-au preluat forțat resursele;
- prevenirea „înfometării”, adică evitarea faptului ca un același proces să fie ales mereu victimă.

#### **4.2.4.4. Rezolvarea interblocării în practică**

De obicei, în mod practic, interblocarea poate fi tratată în două moduri:

- a)-prin ignorarea ei;
- b)-printr-o metodă mixtă de tratare.

Ignorarea se aplică, de exemplu, sistemelor de operare instalate pe PC-uri. Atât WINDOWS-ul cât și UNIX-ul ignoră interblocarea, neavând programe pentru rezolvarea ei.

Există sisteme de operare în care interblocarea ar duce la perturbații grave în funcționare, de exemplu, în unele sisteme de operare funcționând în timp real. Acestor sisteme li se aplică metode mixte de tratare a interblocării.

O metodă mixtă clasicală are la bază împărțirea resurselor în clase de resurse ordonate ierarhic, aşa cum am prezentat la prevenirea interblocării pentru a împiedica așteptarea circulară. În acest mod, o eventuală interblocare ar putea apărea doar în interiorul unei clase.

În interiorul clasei se pot aplica metodele prezentate de prevenire, evitare, detecție și revenire din interblocare.

### **4.3. COMUNICAREA ÎNTRE PROCESE COOPERANTE**

Între două sau mai multe proceze pot exista două tipuri de comunicare:

- a) prin memorie partajată;
- b) prin sistem de mesaje.

a) În sistemele cu memorie partajată există o memorie comună pentru toți utilizatorii iar procezele pot comunica între ele prin intermediul variabilelor partajate din memoria comună. Mecanismul de comunicare este simplu: un proces actualizează o variabilă partajată iar alt proces va citi această variabilă. Această metodă este tipică pentru sistemele multicalculatoare, cu memorie partajată. Sistemul de operare nu este responsabil pentru acest tip de comunicare. Întreaga răspundere revine programatorului de aplicație. De aceea un sistem de operare pentru multicalculatoare nu diferă foarte mult de un sistem de operare pentru un monoprocesor.

b) În acest caz, al comunicării prin mesaje, procezele pot comunica între ele doar prin două operații:

- send (mesaj);
- receive (mesaj)

Deci, prin aceste două primitive de transmisie, un proces poate comunica cu altul doar transmitând sau recepționând un mesaj.

Acest tip de comunicare este specific multicalculatoarelor. Întreaga responsabilitate a comunicării prin mesaje îi revine sistemului de operare. De aceea, aceste sisteme de operare sunt mult mai complicate și greu de proiectat și realizat. Ele mai poartă numele de *sisteme de operare distribuite*.

În acest capitol vom analiza modul de transmisie prin mesaje, acest lucru fiind indiferent dacă folosim mono sau multiprocesoare.

Pentru a exista o comunicare între procese trebuie să existe o linie de comunicatie. Problemele legate de implementarea acestei linii sunt:

- modul în care se stabilesc liniile de comunicare între procese;
- numărul de procese asociate unei linii de comunicare;
- numărul de legături care pot exista între o pereche de procese;
- capacitatea unei linii de comunicare;
- tipul liniei de comunicare, adică dacă linia este unidirecțională sau bidirecțională;
- dimensiunea mesajelor care poate fi fixă sau variabilă.

Modalitățile folosite pentru implementarea logică a liniei de comunicare și a operațiilor send și receive sunt:

- comunicare directă sau indirectă;
- comunicare simetrică sau asimetrică;
- buffering implicit sau explicit (prin buffering se înțelege stocarea mesajului într-o zonă *tampon* de unde este preluat ulterior de către destinatar);
- trimitera mesajului prin copie sau referință.

#### **4.3.1. Comunicare directă și indirectă**

##### **4.3.1.1. Comunicare directă**

În comunicare directă, procesele care trimit sau recepționează mesaje trebuie să menționeze numele procesului care trimite, respectiv care recepționează mesajul.

Primitivele send și receive au următoarea formă simetrică:

send(proces1,mesaj)

(se trimit un mesaj către procesul 1)

receive(proces2,mesaj)

(se recepționează un mesaj de la procesul 2)

Pot avea și o formă asimetrică. În acest caz cele două primitive se definesc astfel:

send(proces,mesaj)

(se trimitе un mesaj către proces)

receive(identifier,mesaj)

( se recepționează un mesaj de la un proces)

Linia de comunicație are următoarele caracteristici:

- linia de comunicație între două mesaje este bidirectională;
- între procesul care, fie vrea să transmită fie vrea să recepționeze un mesaj, și celălalt proces partener se stabilește o singură legătură de comunicație.

#### **4.3.1.2. Comunicație indirectă**

În acest mod de comunicație mesajele sunt trimise și recepționate prin intermediul *cutiilor poștale* (*mail boxes*) care se mai numesc și *porturi*.

Primitivele de comunicare au următoarea formă:

send(port1,mesaj)

(se transmite un mesaj portului 1)

receive(port2,mesaj)

(se recepționează un mesaj de la portul 2)

Cutia poștală sau portul poate avea doi proprietari:

a) procesul;

b) sistemul de operare.

a) La un moment dat, o cutie poștală are un singur proprietar și astfel se cunoaște precis care este numele procesului ce va primi mesajele trimise.

Pentru ca un proces să devină proprietarul unei cutii poștale, se poate utiliza una din următoarele două metode :

-procesul poate declara o variabilă de tip cutie poștală;

-se definește mai întâi o cutie poștală și apoi se declară cine este procesul care o are în proprietate.

Când procesul proprietar al cutiei poștale își încheie execuția, trebuie să execute două lucruri: distrugerea cutiei poștale și anunțarea celorlalte procese despre distrugerea cutiei poștale.

b) În cazul în care sistemul de operare este proprietarul cutiei poștale, atunci aceasta are o existență de sine stătătoare și nu depinde de proces. Mecanismul creat de sistemul de operare cuprinde următoarele operații:

-crearea unei cutii poștale noi;

-trimiterea și recepționarea mesajelor prin cutia poștală;

-distrugerea cutiei poștale.

#### **4.3.2. Linii de comunicații și tipuri de mesaje**

#### 4.3.2.1. Linii de comunicații

Cea mai importantă proprietate a liniei de comunicație este *capacitatea* care arată dacă și în ce fel pot fi stocate mesajele. Există capacitate zero, capacitate limitată și capacitate nelimitată.

-Capacitate zero. În acest caz nu există modalitate de stocare a mesajului.

Procesul emitent va rămâne în aşteptare până când destinatarul va primi mesajul transmis. Trebuie să existe o sincronizare a proceselor care se mai numește și *rendezvous*.

-Capacitate limitată. În acest tip de transmisie există un buffer care poate stoca n mesaje.

-Capacitate nelimitată. În acest tip de transmisie există un buffer de capacitate infinită, teoretic, care nu duce niciodată la situația ca procesul emitor să aștepte.

La fel ca în transmisia de date este necesar și aici, în unele cazuri, ca procesul emitent să știe dacă mesajul emis a ajuns la destinație.

Dacă în transmisia de date există un pachet de tip ACK, aici există ceva asemănător, un mesaj „confirmare”.

De exemplu, când procesul P<sub>1</sub> transmite un mesaj procesului P<sub>2</sub>, atunci următoarea secvență de mesaje face ca P<sub>1</sub> să știe că mesajul său a fost transmis:

send(p2, mesaj)    procesul P1

receive(p1,mesaj)    procesul P2

send(p1,confirmare)    procesul P2

receive(p2,mesaj)    procesul P1

#### 4.3.2.2. Tipuri de mesaje

Din punct de vedere al dimensiunii, mesajele pot fi:

-cu dimensiuni fixe;

-cu dimensiuni variabile;

-mesaje tip.

Mesajele cu dimensiune fixă necesită o implementare simplă dar o programare mai dificilă.

Mesajele cu dimensiune variabilă necesită o implementare fizică dificilă dar o programare mai simplă.

Mesajele tip se folosesc numai în comunicația indirectă.

#### 4.3.3. Excepții în comunicarea interprocese

Dacă în sistemele cu memorie partajată apariția unei erori duce la întreruperea funcționării sistemului, în sistemele cu transmisie de mesaje apariția unei erori nu este așa de gravă. Erorile posibile trebuie să cunoască pentru a le putea trata corespunzător. Cele mai frecvente erori sunt: terminarea unui proces înainte de primirea mesajelor, pierderea mesajelor, alterarea mesajelor și amestecarea lor.

a) Terminarea unui proces înainte de primirea mesajelor. În unele cazuri, un proces, emitent sau destinatar, își poate încheia execuția înainte ca mesajul să fi fost prelucrat. Astfel, pot apărea situații în care mesaje nu vor ajunge niciodată la destinație sau situații în care un proces va aștepta un mesaj ce nu va mai ajunge niciodată.

Dacă un proces  $P_1$ , destinatar, așteaptă un mesaj de la un proces emitent  $P_2$ , care și-a terminat execuția, atunci  $P_1$  va rămâne în starea *blockat*. Pentru ca  $P_1$  să nu se blocheze există două posibilități:

- sistemul de operare termină forțat procesul  $P_1$ ;
- sistemul de operare comunică procesului  $P_1$  că procesul  $P_2$  s-a încheiat.

Dacă un proces  $P_1$  este emitent trimițând un mesaj unui proces  $P_2$  care și-a încheiat execuția, există următoarele posibilități:

- dacă linia de comunicație este limitată sau nelimitată, nu se întâmplă nimic; procesul  $P_1$  își continuă execuția;
- dacă linia de comunicație este de capacitate zero, atunci  $P_1$  se blochează; pentru ca  $P_1$  să nu se blocheze se procedează ca în cazul anterior.

b) Pierderea mesajelor. Pierderea unui mesaj se poate produce atunci când este o defecțiune în linia de comunicație. Pentru remedierea acestui lucru se pot folosi următoarele metode:

- detectarea evenimentului și retransmiterea mesajului de către sistemul de operare;
- detectarea evenimentului și retransmiterea mesajului de către procesul emitent;
- detectarea evenimentului de către sistemul de operare care comunică procesului emitent pierderea mesajului; procesul emitent decide dacă retransmite sau nu mesajul.

c) Alterarea și amestecarea (scrambling) mesajelor. Este situația în care un mesaj ajunge alterat la receptor, în sensul unei alterări a informației din conținutul său. Pentru rezolvarea acestei situații se folosesc metode clasice, din teoria transmiterii informației, de detectare și corectare a erorilor:

- folosirea polinomului de detecție și eventual de corecție a erorilor (așa numitele CRC-uri sau LRC-uri, folosite, de exemplu, la hard disc);
- folosirea *checksums*-urilor, care sunt sume ale bițiilor mesajelor; există două checksums-uri, unul calculat când se transmite mesajul și altul care se calculează din biții mesajului recepționat; La neegalitatea celor două checksums-uri, se consideră eroare;
- folosirea parității în transmiterea și recepționarea mesajului.

#### 4.3.4. Aplicații ale IPC-urilor (Intercomunicare Între Procese)

În sistemul de operare UNIX, ale cărui aplicații vor fi prezentate în capitolele următoare, există următoarele aplicații ale IPC-urilor:

- pipe-uri;
- cozi de mesaje (în SystemV);
- semafoare;
- zone de memorie partajată (în System V);

-semnale.

De asemenea, în subcapitolul următor, de procese clasice, se prezintă procesul producător – consumator rezolvat și prin metoda sistemelor de mesaje (Message Passing).

#### 4.4. PROBLEME CLASICE DE COORDONARE ȘI SINCRONIZARE A PROCESELOR

Există o serie de exemple clasice de coordonare și sincronizare a proceselor în care se regăsesc principalele probleme ce apar în astfel de situații. Multe din aceste probleme se află în structura oricărui sistem de operare. Totodată aceste probleme clasice se regăsesc și în programarea concurrentă. Le vom aborda încercând să le soluționăm cu mijloacele specifice prezentate anterior.

##### 4.4.1. Problema producător-consumator

Fie o serie de procese concurente care produc date (procese PRODUCĂTOR). Aceste date sunt consumate de alte procese (procese CONSUMATOR). Datele sunt consumate în ordinea în care au fost produse. Este posibil ca viteza de producere să difere mult de viteza de consum.

Această problemă s-ar rezolva ușor dacă ar exista un buffer de dimensiuni foarte mari, teoretic infinit, care ar permite operarea la viteze diferite ale producătorilor și consumatorilor. Cum o astfel de soluție este practic imposibilă, vom considera cazul practic al unui buffer finit. Principalele probleme care apar în acest caz sunt:

- buffer gol (consumatorii nu pot consuma date și trebuie să aștepte);
- buffer plin (producătorii nu pot înscrie date în buffer și trebuie să aștepte).

Indiferent de soluțiile alese, vor trebui rezolvate situațiile de citire din buffer-ul gol și de înscriere în buffer-ul plin.

##### 4.4.1.1. Rezolvarea problemei producător – consumator cu ajutorul semafoarelor.

Fie un buffer de dimensiune  $n$  organizat după structura coadă circulară. Bufferul are  $n$  locații pe care le-am notat cu tampon[n].

Vom considera următoarele variabile, semafoare și mutexuri:

Variabilele cu care se scrie și se citește în buffer au fost notate cu scriere și citire. Ele asigură accesul proceselor la poziția unde se dorește operația de scriere sau citire, în ordinea în care au venit.

Semafoarele, semscriere și semcitire, au rolul de a asigura excluderea mutuală între procesele producător și consumator. Semaforul semscriere conține numărul de poziții libere din buffer iar semaforul semcitire conține numărul de poziții pline. Semscriere se inițializează cu  $n$  și semcitire cu 0. Când semscriere = 0 sau semcitire =  $n$ , se va semnaliza situația de buffer plin respectiv buffer gol și procesele vor fi blocate. Se intră într-o excludere mutuală cu protocolul bussy-wait și procesele vor fi deblocate atunci când semscriere  $\neq n$  sau semcitire  $\neq 0$ .

Mutexurile mutexscriere și mutexcitire folosesc pentru excluderea mutuală între două procese de același tip. mutexscriere pentru procesele de tip producător și mutexcitire pentru procesele de tip consumator.

Procesele producător vor citi numere întregi la tastatură iar procesele consumator vor „consuma” aceste numere.

Iată o implementare a problemei producător/consumator, scrisă în limbajul C:

//declarații de variabile, semafoare, mutexuri și inițializatori

```
typedef int semafor; typedef int mutex;  
#define n 1000;  
  
int tampon[n];  
  
int scriere=0, citire=0;  
  
semafor semscreiere=n, semcitire=0;  
  
mutex mutexscriere, mutexcitire;  
  
//Procese producător  
  
int valoare, tastatură;  
  
while(1)  
  
{ valoare=scanf("%d",&tastatura);  
  
wait(semscreiere) ;  
  
lock(mutexscriere) ;  
  
tampon[scriere]=valoare ;  
  
scriere=(scriere+1)%n ;  
  
unlock(mutexscriere) ;  
  
signal(semcitire) ;}  
  
//Procese Consumator  
  
int valoare;  
  
while(1)  
  
{ wait(semcitire);  
  
lock(mutexcitire);  
  
valoare=tampon[citire];  
  
citire=(citire+1)%n;  
  
unlock(mutexcitire);  
  
signal(semscreiere);}
```

Procesele producător funcționează în felul următor:

Se consideră o buclă while din care practic nu se ieșe. Se citește un număr întreg de la tastatură în variabila valoare. Prin wait(semescriere) se asigură excluderea mutuală a procesului respectiv producător față de alte eventuale procese consumator. Prin lock(mutexscriere) se asigură excluderea mutuală a procesului respectiv producător față de alte procese producătoare. Prin tampon[scriere]=valoare se scrie efectiv valoarea în buffer. Prin scriere=(scriere+1)%n se actualizează poziția de scriere în buffer. Prin unlock(mutexscriere) se eliberează mutexul de scriere, permitând altor producători să folosească bufferul. Prin signal(semcitire) se contorizează semaforul de citire cu 1, semnalând că, după ce procesul producător a înscris o valoare în buffer, numărul de poziții din buffer pentru procesele consumatoare s-a mărit cu 1.

Procesele consumator funcționează în mod asemănător.

#### 4.4.1.2. Rezolvarea problemei producător/consumator prin transmitere de mesaje

Am studiat în subcapitolul precedent tehnica de transmitere prin mesaje. Prezentăm acum o aplicație a acestei tehnici la problema producător/consumator.

Pentru aceasta să considerăm o linie de transmisie cu capacitate limitată care folosește un buffer cu  $n$  poziții. Modul de comunicație ales pentru implementare este cel direct, deci fără mailboxuri.

Algoritmul este simplu. Consumatorul trimite mai întâi mesaje goale producătorului. Ori de câte ori producătorul are de dat un produs consumatorului, va lua un mesaj gol și va transmite consumatorului un mesaj plin. Prin aceasta numărul de mesaje din sistem rămâne constant în timp, nedepășind capacitatea limitată a bufferului de comunicație.

Bufferul de comunicație este plin atunci când producătorul lucrează mai repede decât consumatorul și toate mesajele sunt pline. În acest moment producătorul se blochează, așteptând ca un mesaj gol să se întoarcă. Bufferul de comunicație este gol atunci când consumatorul lucrează mai repede decât producătorul. Toate mesajele vor fi golite așteptând ca producătorul să le umple. Consumatorul este blocat așteptând pentru deblocare un mesaj plin.

Iată mai jos o implementare a problemei producător/consumator prin transfer de mesaje.

```
# define n 10000

{int val;

void producător()
message m; /*este mesajul transmis de producător*/
while(1)
{val=produce element(); /*o funcție care produce mesajul transmis de producător*/
receive(consumator,&m); /*așteaptă un mesaj gol*/
construieste mesaj(&m,val); /*o funcție care construiește mesajul transmis*/
send(consumator,&m);} /*se transmite efectiv mesajul consumatorului*/
void consumator()
```

```

{int i,val;
message m;
for(i=1;i<=n;i++)
/*se transmit spre producător cele n mesaje
goale*/
send(producător,&m);
while(1){
receive(producător,&m); /*se primește mesajul de la producător*/
val=extrageremesaj(&m); /*se extrage mesajul pentru a putea fi
prelucrat*/
send(producător,&m); /*se trimit o replică la mesajul gol*/
consum element(val);}} /*o funcție care are rolul de a utiliza mesajul
transmis de producător*/

```

Se observă în implementarea aleasă că parametrul mesaj este un parametru referință.

#### 4.4.2. Problema bărbierului somnoros

##### Enunt

Prăvălia unui bărbier este formată din două camere, una la stradă, folosită ca sală de așteptare, și una în spate, în care se găsește scaunul pe care se așeză clienții pentru a fi serviti. Dacă nu are clienți, bărbierul somnoros se culcă. Să se simuleze activitățile care se desfășoară în prăvălia bărbierului.

##### Rezolvare

Această problemă este o reformulare a problemei producător/consumator, în care locul bufferului de obiecte este luat de scaunul bărbierului iar consumatorul este bărbierul care își servește (consumă) clienții.

În sala de așteptare sunt n scaune pe care se așeză clienții; fiecare scaun este pentru un client. Dacă nu sunt clienți, bărbierul doarme în scaunul de frizerie. Când vine primul client îl trezește pe bărbier și bărbierul îl servește pe client, așezându-l în scaunul de frizerie. Dacă în acest timp sosesc și alți clienți, ei vor aștepta pe cele n scaune. Când toate scaunele sunt ocupate și mai vine încă un client, acesta părăsește prăvălia.

Problema constă în a programa aceste activități în aşa fel încât să nu se ajungă la aşa numitele condiții de cursă. Este o problemă clasică cu multe aplicații, mai ales în cele de help desk.

Pentru implementarea soluției vom utiliza două semafoare și un mutex:

clienți – un semafor ce contorizează clienții ce așteaptă;

bărbier – un semafor care arată dacă bărbierul este ocupat sau nu; el are două valori, 0 dacă bărbierul este ocupat și 1 dacă este liber;

mutexc – un mutex folosit pentru excludere mutuală; arată dacă scaunul de frizerie este ocupat sau nu.

De asemenea mai folosim o variabilă:

cliențînașteptare – care, aşa cum arată și numele, numără clienții care așteaptă. Această variabilă trebuie introdusă deoarece nu există o cale de a citi valoarea curentă a semafoarelor și de aceea un client

care intră în prăvălie trebuie să numere clienții care așteaptă. Dacă sunt mai puțini decât scaunele, se așează și el și așteaptă; dacă nu, părăsește frizeria.

Să descriem algoritmul. Când bărbierul intră dimineața în prăvălie, el execută funcția bărbier(), blocând semaforul clienți care este inițial pe zero. Apoi se culcă și doarme până vine primul client. Când acesta sosește, el execută funcția clienti() și ocupă mutexul care arată că scaunul de frizerie este ocupat. Dacă intră un alt client în acest timp, el nu va putea fi servit deoarece mutexul este ocupat. Va număra clienții care așteaptă și, dacă numărul lor e mai mic decât numărul scaunelor, va rămâne, dacă nu, va părăsi prăvălia. Rămânând, va incrementa variabila cliențiinasteptare. Când clientul care este servit a fost bărbierit, el eliberează mutexul, trezind clienții care așteaptă și unul din ei va ocupa mutexul, fiind servit la rândul său.

Iată mai jos implementarea acestui algoritm.

```
#define scaune 20 /*se definește numărul de scaune*/
type def int semafor;
type def int mutex ;
semafor clienti=0; /*declarații și inițializări*/
semafor bărbier=0;
mutexc=1;
int clientiinasteptare=0;

void bărbier()
{while(1{
    wait(clienti);
    wait(mutexc);
    clientiinasteptare--;
    signal(bărbier);
    signal(mutexc);
    tunde();
}

void clienti()
{wait(mutexc);
if(clientiinasteptare<scaune)
```

```

{clientiinasteptare++;
signal(clienti);
signal(mutexc);
clienttuns();
}

else
signal(mutexc);}}
```

#### **4.4.3. Problema cititori/scriitori**

Problema a fost enunțată de Coutois, Heymans și Parnas în 1971.

Un obiect (care poate fi o resursă, de exemplu un fișier sau o zonă de memorie) este partajat de mai multe procese concurente. Dintre aceste procese, unele doar vor citi conținutul obiectului partajat și aceste procese poartă numele de *cititori* iar celelalte vor scrie în conținutul obiectului partajat, purtând numele de *scriitori*.

Cerința este ca scriitorii să aibă acces exclusiv la obiectul partajat, în timp ce cititorii să poată accesa obiectul în mod concurrent (neexclusiv).

Există mai multe posibilități de a soluționa această problemă. Vom aminti două variante.

#### **Varianta 1**

Nici un cititor nu va fi ținut în așteptare, decât dacă un scriitor a obținut deja permisiunea de acces la obiectul partajat.

La un acces simultan la obiectul partajat, atât al scriitorilor cât și al cititorilor, cititorii au prioritate.

#### **Varianta 2**

Când un scriitor este gata de scriere, el va executa scrierea cât mai curând posibil.

La un acces simultan, scriitorii sunt prioritari.

Oricum, în ambele cazuri, problema principală ce trebuie rezolvată este *infometarea*, adică așteptarea la infinit a obținerii dreptului de acces.

Să implementăm un program pentru prima variantă, folosind următoarele semafoare, mutexuri și variabile:

scrie – un semafor cu mai multe roluri; el asigură excluderea mutuală a scriitorilor; este folosit de către primul cititor care intră în propria secțiune critică; de remarcat că acest semafor nu este utilizat de cititorii care intră sau ieș din secțiunea critică în timp ce alții cititori se află în propria secțiune critică;

contorcitire – o variabilă care are rolul de a ține evidența numărului de procese existente în cursul citirii;

semcontor – un semafor care asigură excluderea mutuală când este actualizată variabila contorcitire.

Dacă un scriitor este în secțiunea critică și n cititori așteaptă, atunci un cititor așteaptă la semaforul scriere iar ceilalți n-1 așteaptă la semcontor.

La signal(scrie), se poate relua fie execuția unui singur scriitor, fie a cititorilor aflați în așteptare, decizia fiind luată de planificator.

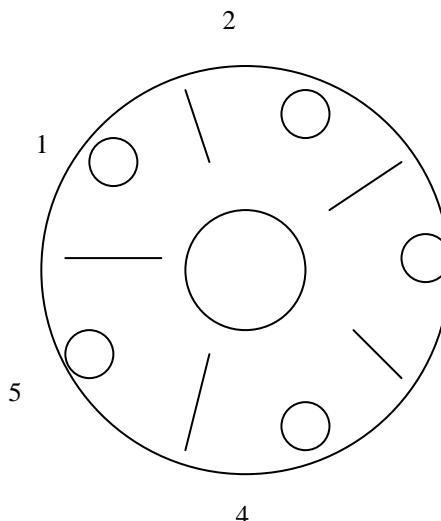
Iată implementarea programului pentru prima variantă:

```
typedef int semafor; /*declarații și initializări*/  
  
int contorcitire=0;  
  
semafor scrie=1,semcontor=1 ;  
  
void scriitor()  
  
{wait(scrie) ;  
  
scriereobiect() ;  
  
signal(scrie) ;}  
  
void cititor()  
  
{wait(semcontor) ;  
  
contor citire++;  
  
if(contorcitire==1)wait(scrie);/*primul cititor*/  
  
signal(semcontor);  
  
citireobiect();  
  
wait(semcontor);  
  
contor citire--;  
  
if(contorcitire==0)signal(scrie);/*ultimul cititor*/  
  
signal(semcontor);}
```

#### 4.4.4. Problema cinei filozofilor chinezi

Cinci filozofi chinezi își petrec viața gândind și mâncând în jurul unei mese circulare înconjurată de cinci scaune, fiecare filozof ocupând un scaun. În centrul mesei este un platou cu orez și în dreptul fiecărui filozof se află o farfurie. În stânga și în dreapta farfuriei câte un bețișor. Deci, în total, cinci farfurii și cinci bețișoare. Un filozof poate efectua două operații: gândește sau mânâncă. Pentru a putea mânca, un filozof are

nevoie de două bețișoare, unul din dreapta și unul din stânga. Dar un filozof poate ridica un singur bețișor odată. Problema cere o soluție pentru această cină.



**Fig. 4.7. Problema filozofilor chinezi.**

Trebuie rezolvate două probleme majore:

-Interblocarea care poate să apară. De exemplu, dacă fiecare filozof ridică bețișorul din dreapta sa, nimeni nu mai poate să-l ridice și pe cel din stânga și apare o situație clară de aşteptare circulară, deci de interblocare.

-Problema înfometării unui filozof care nu apucă să ridice niciodată cele două bețișoare.

Această problemă a fost enunțată și rezolvată de către Dijkstra în 1965.

Există multe soluții ale acestei probleme, marea majoritate utilizând excluderea mutuală.

Pentru a nu apărea interblocarea se folosesc, în general, soluții de prevenire a acesteia adică se impun unele restricții în ceea ce privește acțiunile filozofilor, cum ar fi:

-unui filozof i se permite să ia un bețișor numai atunci când ambele bețișoare, din dreapta și din stânga sa, sunt disponibile;

-se creează o corespondență biunivocă între mulțimea numerelor naturale și filozofii, fiecare filozof având un număr natural; o soluție asimetrică impune filozofilor cu număr impar să apuce mai întâi bețișorul din stânga și apoi pe cel din dreapta, iar filozofilor cu număr par să ia mai întâi bețișorul din dreapta și apoi pe cel din stânga.

Vom prezenta, în continuare, o soluție clasică a acestei probleme, care rezolvă și situația interblocării și pe cea a înfometării. În acest algoritm, se poate generaliza problema pentru n filozofi. Se urmărește în ce stare poate fi un filozof, existând trei stări posibile: mânâncă, gândește și este înfometat.

Unui filozof i se permite să intre în starea „mânâncă” numai dacă cel puțin unul din vecinii săi nu este în această stare. Prin această restricție se previne interblocarea.

Pentru implementare, se utilizează următoarele structuri:

stare[n] – un vector n-dimensional, în care pe poziția

i se găsește starea filozofului la un moment dat; aceasta poate fi:

- 0 pentru starea „gândește”

- 1 pentru starea „infometat”

- 2 pentru starea „măncă”

`sem[n]` – un vector n-dimensional, în care `sem[i]` este un semafor pentru filozoful i;

mutexfil – un mutex pentru excludere mutuală;

funcția filozof(i) – este funcția principală care coordonează toate celelalte funcții și care se referă la filozoful i;

funcția ridică bețișor(i) – este funcția care asigură pentru filozoful i ridicarea ambelor bețișoare;

funcția pune bețișor i – este funcția care asigură pentru fiecare filozof i punerea ambelor bețișoare pe masă;

funcția `test(i)` – este funcția care testează în ce stare este filozoful i.

Implementarea este:

```
#define n 5 /*am definit numărul de filozofi*/
```

```
#define stang(i+n-1)%n /*numărul vecinului din stânga filozofului i*/
```

```
#define drept(i+1)%n /*numărul vecinului din stânga filozofului i*/
```

```
#define gandest 0
```

```
#define infometat 1
```

```
#define mananca 2
```

```
typedef int semafor;
```

```
typedef int mutex;
```

```
int stare[n];
```

mutex mutexfil=1

semafor sem[n];

```
void filozof(int i)
```

```
while(i) {
```

gandeste()

/\*filozoful i gândește\*/

ridicabetisor(i);

/\*filozoful i ridică cele două bețișoare\*/

mananca();

/\*filozoful i mănâncă\*/

```

punebetisor(i);                                /*filozoful i pune pe masă două
bețișoare*/


---


void ridicabetisor(int i)
{
    wait(mutexfil);                            /*se intră în regiunea critică*/
    stare[i]=infometat;                        /*filozoful i este în starea înfometat*/
    test(i);                                    /*încearcă să acapareze cele două
bețișoare*/
    signal(mutexfil);                          /*se ieșe din regiunea critică*/
    wait(sem[i]);                             /*procesul se blochează dacă nu se pot
lua cele două bețișoare*/
}

void punebetisor(int i)
{
    wait(mutexfil);                            /*se intră în regiunea critică*/
    stare [i]=gandeste;                      /*filozoful i a terminat de gândit*/
    test(stang);                             /*se testează dacă vecinul din stânga
filozofului i mănâncă*/
    test(drept);                            /*se testează dacă vecinul din
dreapta filozofului i mănâncă*/
    signal(mutexfil);                        /*se ieșe din regiunea critică*/
}
void test(int i);
{
    if stare [i]==
        infometat&&stare[stang]!=
        mananca&&stare[drept]!=
        mananca)
    {
        stare[i]=mananca;
        signal(sem[i]); } }

```

#### 4.4.5. Probleme propuse pentru implementare

##### 4.4.5.1. Problema rezervării biletelor

###### Enunț

Fiecare terminal al unei rețele de calculatoare este plasat într-un punct de vânzare a biletelor pentru transportul feroviar. Se cere să se găsească o modalitate de a simula vânzarea biletelor, fără a vinde două sau mai multe bilete pentru același loc.

###### Rezolvare

Este cazul în care mai multe procese (vânzătoarele de bilete) încearcă să utilizeze în mod concurrent o resursă nepartajabilă, care este o resursă critică (locul dintren).

Problema se rezolvă utilizând excluderea mutuală iar pentru implementarea ei cea mai simplă metodă este folosirea semafoarelor.

##### 4.4.5.2. Problema grădinii ornamentale

###### Enunț

Intrarea în grădinile ornamentale ale unui oraș oriental se face prin în părți. Să se țină evidența persoanelor care au intrat în grădină.

###### Rezolvare

Fiecare poartă de intrare în grădină este o resursă care trebuie accesată exclusiv de un proces (o persoană care intră în grădină).

Dacă, la un moment dat, pe una din porți intră o persoană, atunci, în acel moment, pe nici o altă poartă nu mai intră vreo persoană în grădină.

Această problemă face parte din problema excluderii reciproce.

##### 4.4.5.3. Problema emițător-receptor

###### Enunț

Un emițător emite succesiv mesaje, fiecare dintre ele trebuind să fie recepționate de toți receptorii, înainte ca emițătorul să emită mesajul următor.

###### Solutie

Este o aplicație de tipul client-server. În acest tip de aplicații, un proces server este un proces ce oferă servicii altor procese din sistem iar un proces client este unul care solicită servicii de la server și le consumă.

Dacă procesele client și server nu sunt pe același calculator, atunci această aplicație este distribuită. Implementarea ei, cel mai adesea utilizată în sistemele multicalculator, se face prin transmisie de mesaje.



## 5. GESTIONAREA MEMORIEI

Sistemele de operare actuale folosesc multiprogramarea ceea ce înseamnă că, la un moment dat, în memorie se pot afla mai multe programe.

Problema esențială pe care trebuie să o rezolve un sistem de operare este ca un program, pentru a putea fi executat, să aibă codul executabil și datele rezidente în memorie. SO trebuie să partioneze memoria, ca să permită utilizarea ei simultană de către mai multe programe. De asemenea trebuie să asigure mecanisme de protecție pentru ca programele să poată coexista în memorie în bune condiții.

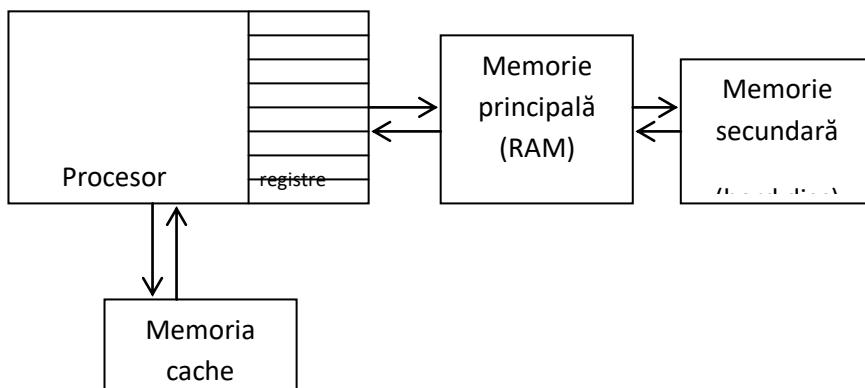
Între hardul pe care îl pune la dispoziție un sistem de calcul și sarcinile sistemului de operare există o graniță foarte flexibilă în ceea ce privește gestiunea memoriei. De-a lungul anilor, hardul a înglobat în componența sa multe din funcțiile pe care le avea sistemul de operare. În capitolul de față, vom prezenta sarcinile principale ale SO, precizând totuși ce facilități oferă hardul actual.

### 5.1. IERARHII DE MEMORIE

Principalele elemente hard care intră într-un sistem de memorie sunt:

- registrele generale ale procesorului;
- memoria CACHE;
- memoria principală;
- memoria secundară.

În *memoria internă* a unui computer intră memoria principală și memoria CACHE iar *memoria externă* este formată din memoria secundară (hard disk, floppy disk, CD-ROM etc).



**Fig.5.1. Schema principală hard a unui sistem de memorie.**

Parametrii principali ai unei memorii sunt:

- *timpul de acces la memorie* (timp necesar pentru operația de citire a memoriei);
- *capacitatea memoriei*.

La ora actuală, există următoarele categorii de memorie:

- *memorii rapide*;
- *memorii lente*;
- *memorii foarte lente*.

Memoriile rapide au un timp de acces foarte mic; din punct de vedere tehnologic sunt memorii statice și cu un cost ridicat pe unitatea de memorie. Se utilizează pentru registrele generale ale unității centrale și pentru memoria CACHE.

Memoriile lente au un timp de acces mai mare, din punct de vedere tehnologic sunt realizate ca memorii dinamice și au un cost pe unitatea de memorie mult mai mic decât al memorilor rapide. Se utilizează pentru memoria principală.

Memoriile foarte lente au timp de acces foarte mare în comparație cu celelalte tipuri, și, bineînțeles, un cost mult mai mic pe unitatea de memorie. Este cazul hard discurilor.

O ierarhie de memorie este un mecanism, transparent pentru utilizator, prin care SO acționează aşa fel încât, cu o cantitate cât mai mică de memorie rapidă și cu o cantitate cât mai mare de memorie lentă și foarte lentă, să lucreze ca și cum ar avea o cantitate cât mai mare de memorie rapidă.

Partea din SO care gestionează ierarhia de memorie are ca sarcină să urmărească ce părți de memorie sunt în uz și ce părți nu sunt folosite, să aloce memorie proceselor care au nevoie și să o dealoce când nu mai este necesară, să coordoneze schimbul între memoria principală și disc când memoria principală este prea mică pentru a conține toate procesele.

## 5.2. OPTIMIZĂRI ÎN ÎNCĂRCAREA ȘI EXECUȚIA UNUI PROGRAM ÎN MEMORIE

Există unele optimizări ale sistemelor de operare, legate de încărcarea și execuția unui program în memorie :

- încărcarea dinamică;
- suprapunerি (overlay-uri);
- legarea dinamică (.dll în WINDOWS, .so în UNIX).

### 5.2.1. Încărcarea dinamică

Încărcarea dinamică este încărcarea rutinelor în memoria principală numai atunci când este nevoie de ele. În acest mod sunt aduse în memorie numai rutinele *apelate*, rutinele neutilitate nu vor fi încărcate niciodată. Un astfel de exemplu este un program de dimensiune foarte mare care conține multe rutine de tratare a erorilor, (rutine foarte mari), erorile tratate fiind foarte rare. Desigur aceste rutine nu vor fi încărcate în memorie.

Trebuie remarcat că mecanismul de încărcare dinamică nu este implementat în SO, el fiind o sarcină a utilizatorului.

### 5.2.2. Overlay-uri

Overlay-urile furnizează un mod de scriere a programelor care necesită mai multă memorie decât memoria fizică, cu alte cuvinte a programelor de mari dimensiuni. La fel ca și în cazul încărcării dinamice, nu este o sarcină a sistemului de operare ci a utilizatorului. Aceasta trebuie să partaționeze programul în bucăți mai mici și să încarce aceste partiții în memorie aşa fel ca programul să nu aibă de suferit în execuție. Desigur o astfel de programare este complexă, dificilă. Ea se utiliza în SO mai vechi.

### 5.2.3. Legarea dinamică

Legarea dinamică este utilizată în sistemele de operare de tip WINDOWS sau OS/2 , pentru fișierele cu extensia •dll sau în UNIX , în bibliotecile cu extensia •so.

Conform acestui mecanism rutinele nu sunt incluse în programul obiect generat de computer, legarea subrutinelor fiind amânată până în momentul execuției programelor. Ca tehnică, se folosește un *stub* care, când este apelat, este înlocuit cu rutina respectivă ce se și execută. Rolul sistemului de operare este să vadă dacă rutina este în memorie și dacă nu să o încarce. În acest mod se realizează o bună partajare a codului.

Trebuie menționat că:

- programul nu funcționează dacă •dll-urile necesare nu sunt prezente în sistem;
- programul depinde de versiunea •dll-urilor.

### 5.3. ALOCAREA MEMORIEI

Alocarea memoriei este efectuată de către *allocatorul de memorie* care ține contabilitatea zonelor libere și ocupate din memorie, satisfacând cererea pentru noi zone și reutilizează zonele eliberate.

Alocarea memoriei se face ierarhic; la baza acestei ierarhii se află sistemul de operare care furnizează utilizatorilor porțiuni de memorie iar utilizatorul, la rândul său, gestionează porțiunea primită de la SO după necesitățile sale.

#### 5.3.1. Alocarea de memorie în limbaje de programare

Există o clasificare a limbajelor de programare din punct de vedere al alocării de memorie:

1)-Limbaje care nu pot aloca memorie. Este cazul limbajelor mai vechi (Fortran, Cobol). În aceste limbaje, utilizatorul nu poate aloca dinamic memorie în momentul execuției ci înaintea execuției programului.

2)-Limbaje cu alocare și delocare explicită. Aceste limbaje permit utilizatorului să ceară, pe parcursul execuției, noi zone de memorie și să returneze memoria utilizată. Este cazul funcțiilor new și free în PASCAL și new și delete sau malloc și free din C și C++.

3)-Limbaje cu colectoare de gunoaie (garbage collection). În aceste limbaje, utilizatorul nu specifică niciodată când vrea să elibereze o zonă de memorie. Compilatorul și o serie de funcții care se execută simultan cu programul deduc singure care dintre zone nu sunt necesare și le recuperează.

Avantajele acestui limbaj sunt:

-utilizatorul este scutit de pericolul de a folosi zone de memorie nealocate, prevenind astfel apariția unor *bug-uri* ;

-există siguranță că în orice moment, o zonă de memorie utilizată nu este dealocată.

Dezavantajele limbajului:

-alocarea este imprevedibilă în timp, adică nu se știe exact în care moment se va produce;

-nu se poate să se știe dacă zona de memorie utilizată va fi sau nu utilizată în viitor, deci este posibil ca un program să păstreze alocate zone de memorie care-i sunt inutile.

Această tehnică de alocare este întâlnită în limbajele Lisp și Java.

Menționăm că majoritatea alocatoarelor din nucleele sistemelor de operare comerciale sunt de tipul 1) și 2). Există și nuclee ale sistemelor de operare cu alocatoare de tipul 3), cum ar fi sistemele Mach sau Digital.

### 5.3.2. Caracteristici ale alocatoarelor

Vom prezenta câteva caracteristici ale alocatoarelor după care se poate evalua calitatea acestora.

1)-Timp de operatie. Este timpul necesar unei operații de alocare/dealocare. Acest timp depinde de tipul alocatorului, fiecare alocator trebuind să execute un număr de operații pentru fiecare funcție. Cu cât memoria disponibilă este mai mare cu atât timpul de execuție a unui apel este mai mare.

2)-Fragmentarea. O problemă cu care se confruntă alocatoarele este faptul că aproape niciodată ele nu pot folosi întreaga memorie disponibilă, pentru că mici fragmente de memorie rămân neutilizate. Aceste pierderi apar în urma împărțirii spațiului disponibil de memorie în fragmente neocupate în totalitate de programe. Fragmentarea poate fi :

- fragmentare externă;
- fragmentare internă;

Fragmentarea externă apare ori de câte ori există o partitură de memorie disponibilă, dar nici un program nu încape în ea.

Se demonstrează că atunci când avem de-a face cu alocări de blocuri de mărimi diferite, fragmentarea externă este inevitabilă. Singurul mod de a reduce fragmentarea externă este *compactarea* spațiului liber din memorie prin mutarea blocurilor dintr-o zonă în alta.

Fragmentarea internă este dată de cantitatea de memorie neutilizată într-o partitură blocată (ocupată parțial de un program).

Pentru a nu avea fragmentare internă ideal ar fi ca fiecare program să aibă exact dimensiunea partituriei de memorie în care este încărcat, lucru aproape imposibil.

3)-Concurrentă. Această caracteristică se referă la gradul de acces concurrent la memorie. Este cazul mai ales la sistemele cu multiprocesor, cu memorie partajată. Un alocator bine scris va permite un grad mai ridicat de concurrentă, pentru a exploata mai bine resursele sistemului.

4)-Grad de utilizare. Pe lângă faptul că memoria este fragmentată, alocatorul însuși menține propriile structuri de date pentru gestiune. Aceste structuri ocupă un loc în memorie, reducând utilizarea ei efectivă.

### 5.3.3. Tipuri de alocare

În sistemul de gestiune a memoriei există două tipuri de adrese:

- adrese fizice;
- adrese logice.

Adresele fizice sunt adresele efective ale memoriei fizice. Se știe că pentru a adresa o memorie fizică cu o capacitate de  $n$  octeți este necesar un număr de adrese egal cu  $\log_2 n$ .

Adresele logice, sau virtuale, sunt adresele din cadrul programului încărcat.

De obicei, în marea majoritate a alocatoarelor, în momentul încărcării unui program sau chiar al compilării lui, adresele fizice coincid cu cele logice. În momentul execuției acestea nu mai coincid.

Translatarea adreselor logice în adrese fizice este executată de către hardware-ul de mapare al memoriei.

Alocarea memoriei se poate face în două feluri:

- alocare contiguă;
- alocare necontiguă.

Alocarea contiguă înseamnă alocarea, pentru un proces, a unei singure porțiuni de memorie fizică, porțiune continuă; (elemente contigüe înseamnă elemente care se ating spațial sau temporal).

Alocarea necontiguă înseamnă alocarea, pentru un proces, a mai multor porțiuni separate din memoria fizică.

Alocarea memoriei se mai face în funcție de cum este privită memoria. Există două tipuri de memorie:

- memorie *reală*;
- memorie *virtuală*.

Memoria reală constă numai în memoria internă a sistemului și este limitată de capacitatea ei.

Memoria virtuală vede ca un tot unitar memoria internă și cea externă și se permite execuția unui proces chiar dacă acesta nu se află integral în memoria internă.

#### 5.3.4. Scheme de alocare a memoriei

Există mai multe scheme de alocare de la foarte simple la foarte complexe. În general, sunt de două categorii:

- sisteme care transportă procesele, înainte și înapoi, între memoria principală și disc (swapping și paging);
- sisteme care nu fac acest lucru ( fără swapping și paging).

a) Pentru sistemele cu alocare contiguă, există schemele:

- alocare *unică*;
- alocare *cu partiții fixe* ( alocare statică);
- alocații *cu partiții variabile* (alocare dinamică);
- alocare cu *swapping*.

b) Pentru sistemele cu alocare necontiguă:

- alocare *paginată* (simplă sau la cerere);
- alocare *segmentată* (simplă sau la cerere);
- alocare *segmentată-paginată* (simplă sau la cerere).

##### 5.3.4.1. Alocare unică

###### a) Alocare unică cu o singură partiție

Este un tip de alocare folosit în primele sisteme de operare care lucrau monouser. Este cea mai simplă schemă în care toată memoria internă este destinată sistemului de operare, fără nici o schemă de administrare a memoriei. Desigur, ea ține de domeniul istoriei.

#### b) Alocare unică cu două partiții

- În acest tip de alocare există două partiții;
- partiție pentru sistemul de operare (nucleul);
- partiție pentru utilizator.

Este cazul sistemului de operare MS-DOS. Principalul dezavantaj constă în faptul că nu se oferă soluții pentru multiprogramare.

#### 5.3.4.2. Alocare cu partiții fixe (alocare statică)

Memoria este împărțită static în mai multe partiții, nu neapărat de dimensiuni egale. În fiecare partiție poate rula cel mult un proces, gradul de multiprogramare fiind dat de numărul partiților. De obicei, împărțirea în partiții și dimensionarea acestora se face la început de către operator. Programele sunt încărcate în memorie prin niște cozi de intrare. Este posibil ca să existe o singură coadă de intrare în memorie sau diferite cozi la diferitele partiții. Din coada de intrare un program intră în cea mai mică partiție destul de mare, însă, pentru a-l primi. Spațiul neocupat de program în această partiție rămâne pierdut și în acest fapt constă dezavantajul schemei. Există atât fragmentare internă cât și fragmentare externă.

Acest tip de alocare a fost utilizat de sistemul de operare SIRIS V, în sistemele de calcul FELIX C-256/1024, sisteme care au existat și la noi în țară, în toate centrele de calcul.

În această alocare apărea pentru prima dată și un mecanism de protecție a memoriei care era asigurat de sistemul de *chei de protecție și chei de acces*.

Sistemul de memorie era împărțit în pagini de 2KO și fiecare pagină avea o cheie de protecție. Aceasta era pusă printr-o instrucțiune cod-mașină a calculatorului. Pentru ca un program să fie rulat într-o zonă a memoriei, trebuia să fie prezentate cheile de acces pentru fiecare pagină utilizată. La identitatea cheii de acces cu cea de protecție, se permitea accesul în pagina respectivă. Existau și chei de acces care deschideau orice cheie de protecție (de exemplu cheia de acces zero), precum și chei de protecție deschise de orice cheie de acces.

#### 5.3.4.3. Alocare cu partiții variabile

În această alocare numărul, locația și dimensiunea partiților variază dinamic. Ne mai fiind fixată dimensiunea partiților, care pot fi ori prea mari ori prea mici față de program, crește mult gradul de utilizare al memoriei. În schimb se complică alocarea și dealocarea memoriei și urmărirea acestor operații.

Când se încarcă un proces în memorie, i se alocă exact spațiul de memorie necesar, din memoria liberă creându-se dinamic o partiție. Când se termină un proces, partiția în care a fost procesul devine memorie liberă, ea unificându-se cu spațiul de memorie liberă existent până atunci.

Pentru gestionarea unei astfel de alocări, sistemul de operare trebuie să aibă două tabele:

- tabela partiților ocupate;
- tabela partiților libere.

Principala problemă este alegerea unui spațiu liber; această alegere trebuie făcută cu minimizarea fragmentării interne și externe. Există anumiți algoritmi de alegere a spațiului liber.

-FFA (First Fit Algoritm), prima potrivire. Se parurge lista spațiilor libere, care este ordonată crescător după adresa de început și se alege primul spațiu de dimensiune suficientă.

Acest algoritm este folosit în sistemul SO MINIX, creat de Tannenbaum.

-BFA (Best Fit Algoritm), cea mai bună potrivire. Se parurge lista spațiilor libere și se alege spațiul cu dimensiunea cea mai mică în care încape programul. În acest fel se minimizează fragmentarea internă. În cazul în care lista spațiului liber este ordonată crescător după dimensiunea spațiilor libere, se alege evident primul spațiu liber. În acest caz FFA și BFA coincid. BFA este utilizat în SO MS-DOS.

-WFA (Worst Fit Algoritm), cea mai proastă potrivire. Se parurge lista spațiilor libere ordonată crescător după dimensiune și se alege ultimul spațiu din listă.

Din punct de vedere al vitezei și al gradului de utilizare al memoriei, FFA și BFA sunt superioare strategiei WFA.

#### 5.3.4.4. Alocarea prin swapping

În acest tip de alocare, un proces, în majoritatea cazurilor în stare de așteptare, este evacuat temporar pe disc, eliberând memoria principală. Reluarea execuției procesului se face prin reîncărcarea sa de pe disc în memoria principală. *Swap* înseamnă a face schimb și, într-adevăr, este vorba de o schimbare de pe memoria principală pe una externă și înapoi. Problema principală în swapping este: ce procese sunt evacuate din memorie pe disc? Există un algoritm bazat pe priorități, numit Rollout-Rollin, conform căruia, la apariția unui proces cu prioritate ridicată, vor fi evacuate procesele sau procesul cu prioritatea cea mai scăzută.

O altă problemă este: la ce adresă din memorie va fi readus procesul evacuat. De obicei, dacă alocarea este statică, procesul va fi readus în aceeași partitie din care a plecat. În cazul alocării dinamice, procesul va fi adus în orice loc al memoriei.

Pentru ca alocarea prin swapping să aibă eficiență, este necesar ca memoria externă (harddiscul) să aibă două caracteristici: o capacitate suficient de mare și un timp de acces foarte mic.

Capacitatea mare este necesară deoarece pe disc trebuie să se evaceze toate imaginile proceselor, numărul lor putând ajunge, la un moment dat, foarte mare.

Timpul de acces trebuie să fie foarte mic. În caz contrar, costul swappingului memoria  $\Leftrightarrow$  disc poate deveni inconvenabil. O condiție esențială pentru micșorarea costului este ca timpul de execuție al procesului să fie mult mai mare decât timpul de swapping.

O altă problemă apare atunci când un proces necesită date noi în timpul execuției și, implicit, zone suplimentare de memorie. Este situația aşa numitelor procese cu dimensiune variabilă în timpul execuției. Dacă, datorită cererii suplimentare de memorie, se depășește zona de memorie afectată procesului, atunci sistemul de operare trebuie să intervină. Există următoarele posibilități:

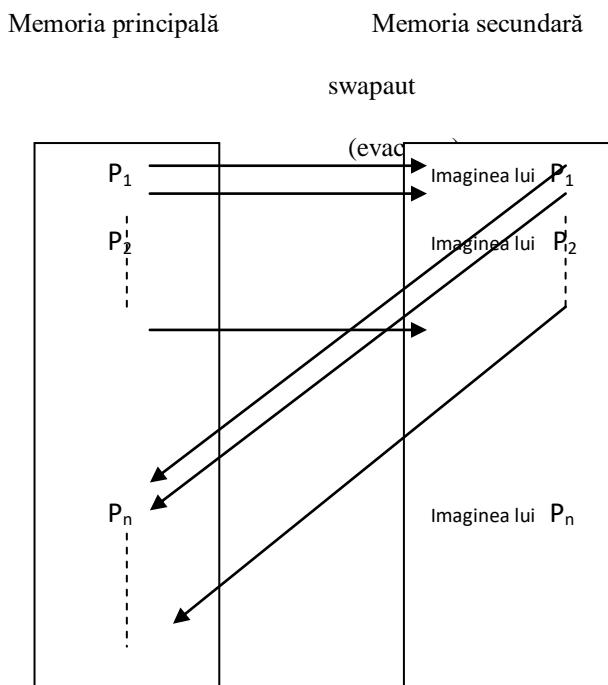
- să încheie forțat procesul care a formulat cerere de suplimentare a memoriei și să se considere această cerere ca o eroare de execuție;

- să returneze decizia procesului, în sensul că acesta va decide dacă își va încheia activitatea sau o va continua strict în zona de memorie care i-a fost impusă; în cazul alocării dinamice, să evaceze procesul pe disc, și să aștepte eliberarea unei zone de memorie satisfăcătoare pentru proces.

Trebuie menționat că încă suntem în modul de alocare contiguu, deci spațiul de adresare al unui proces nu poate depăși capacitatea memoriei interne.

În concluzie, putem spune că principalul avantaj al alocării prin swapping este faptul că se simulează o memorie mai mare decât cea fizică existentă.

Principalul dezavantaj este costul swappingului care uneori poate fi destul de mare. Mecanismul de swapping este redat în Fig.5.2.



**Fig.5.2. Mecanismul swapping.**

#### 5.4. PAGINAREA MEMORIEI

Paginarea este un tip de alocare necontiguu, aceasta însemnând că unui proces îi poate fi alocată memorie oriunde, atât în memoria internă cât și în cea externă, iar memoria alocată poate fi formată din bucăți de memorie.

##### 5.4.1. Suportul hardware

Memoria fizică este împărțită în blocuri de lungime fixă, numite *cadre de pagină (frames)* sau *pagini fizice*. Lungimea unui cadru este o putere a lui doi și este constantă pentru fiecare arhitectură de sistem în parte. Pentru Intel lungimea unui cadru este 4KO.

Memoria logică a unui proces este împărțită în pagini logice sau pagini virtuale care sunt plasate în memoria secundară, pe harddisc.

Pentru execuția unui proces, paginile sale logice trebuie încărcate în cadrele libere ale memoriei fizice, într-un mod necontiguu. Evidența cadrelor libere este ținută de sistemul de operare. Bineînțeles, dacă procesul are nevoie de n pagini logice, trebuie să se găsească n cadre libere.

Atât adresele fizice cât și cele logice sunt implementate în hard și ele conțin:

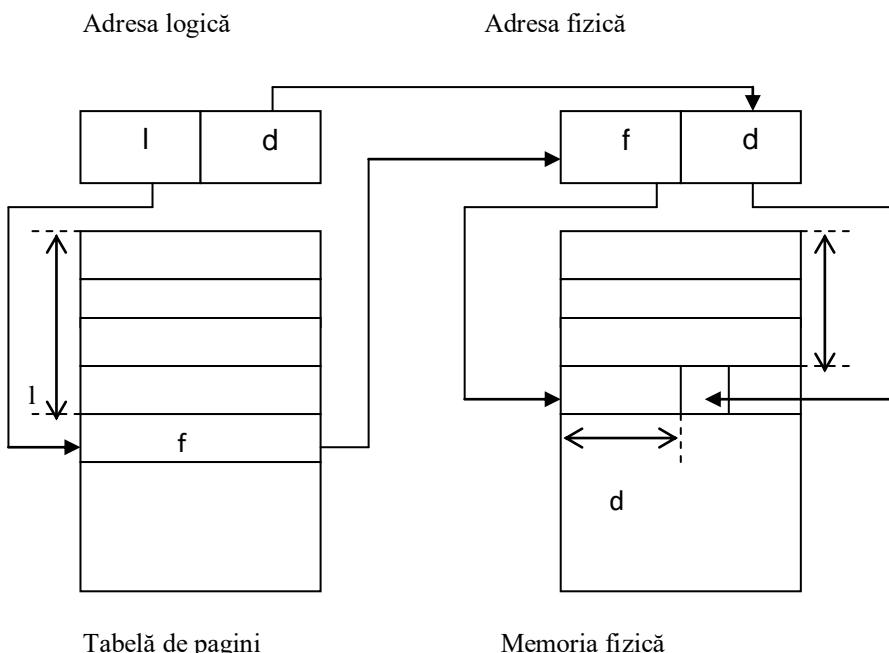
-adresa fizică=nr de cadru(f)+deplasament în cadru(d)

-adresa logică=nr de pagini logice(l)+deplasament în pagina logică

Prin *mapare* se înțelege translatarea adreselor logice în adresa fizică. Această sarcină îi revine sistemului de operare prin utilizarea *tabelei de pagini*.

Fiecare proces are o tabelă de pagini în care în care fiecare pagină logică are adresa de bază a cadrului asociat ei. Pentru translatare se folosește numărul de pagină drept index în tabela de pagini.

În schema din figura 5.3. se vede corespondența între adresa logică și cea fizică prin intermediul tabelei de pagini.



**Fig.5.3. Corespondența dintre adresa logică și cea fizică.**

#### 5.4.2. Implementarea tabelei de pagini

Păstrarea tabelelor de pagini se face în :

- a)-registrele hard;
- b)-memoria principală;
- c)-memoria hard specială, de tip asociativ.

a) Soluția de implementare a tabelelor de pagini în registrele unității centrale este , desigur , foarte rapidă dar și foarte scumpă, mai ales pentru un număr foarte mare de tabele. În plus accesul la registre se face în mod privilegiat ceea ce, la un moment dat, poate constitui un impediment.

b) Soluția de implementare a tabelei de pagini în memoria principală presupune un cost mult mai scăzut și de aceea este soluția cea mai des întâlnită. Ea impune accese multiple la memorie; mai întâi trebuie accesată tabela de pagini pentru aflarea adresei fizice asociată adresei logice dorite; apoi se accesează memoria fizică la adresa aflată pe baza translatării. În acest tip de translatare se folosește un *Registru de Bază al Tabelei de Pagină* (RBTP). Atunci când se dorește să se lucreze cu altă tabelă de pagină decât cea curentă, se încarcă RBTP cu noua valoare de pagină, reducându-să în acest fel timpul de comutare.

c) O altă soluție de implementare este aceea în care se utilizează o memorie asociativă hardware, de mică dimensiune. Aceasta folosește un set de registre associative. Fiecare registru are două componente:

-cheie, în care se memorează numărul paginii logice;

-valoare, în care se memorează numărul cadrului asociat.

Căutarea într-o astfel de memorie asociativă se face în felul următor: un element care trebuie găsit este comparat simultan cu toate cheile și unde se găsește coincidență se extrage câmpul valoare corespunzător.

În scopul utilizării unei astfel de memorii associative pentru tabela de pagină, se face următoarea corespondență:

-în *cheie* se memorează numărul paginii logice;

-în *valoare* se memorează numărul cadrului asociat.

Atunci când procesorul generează o adresă logică, dacă numărul de pagină logică coincide cu una din chei, numărul de cadrus devine imediat disponibil și este utilizat pentru a accesa memoria. Dacă numărul de pagină nu coincide cu nici una dintre chei, atunci, pentru aflarea numărului cadrului asociat, se face un acces la tabela de pagini din memoria internă. Informația astfel obținută este utilizată pentru accesarea memoriei utilizator, cât și pentru a fi adăugată în cadrul registrelor associative, împreună cu numărul de pagini asociat, ca să poată fi regăsită rapid în cadrul unei referiri ulterioare.

Alte îmbunătățiri ale implementării tăbelei de pagini folosesc:

a)-tabele de pagini pe nivele multiple;

b)-tabele de pagini inverse.

a) Pentru un spațiu de adresare foarte mare, tabelele de pagini pot avea dimensiuni mari. De exemplu, pentru o memorie principală de 4 GO ( $2^{32}$  octeți), dacă pagina are 4KO, atunci o tabelă de pagini are 1 milion de intrări. Una din tehniciile de reducere a dimensiunilor tăbelei de pagini este utilizarea unei tăbele de pagini pe nivele multiple. Aceasta echivalează cu împărțirea tăbelei de pagină în altele care să aibă dimensiuni mai mici și unde căutarea să se facă într-un timp mai scurt.

Pornind de la exemplul precedent, cu memoria principală de  $2^{32}$  octeți, o adresă logică arată astfel:

numărul de pagină	deplasament
-------------------	-------------

Fiecare tabelă de pagină are un milion de intrări. Dacă partilionăm tabelul de pagină în 4 secțiuni, fiecare secțiune are 256 K intrări iar o adresă logică pentru o secțiune arată astfel:

numărul de secțiune	numărul de pagină	deplasament
---------------------	-------------------	-------------

b) O altă modalitate de reducere a dimensiunii tăbelelor de pagini este folosirea *tăbelei de pagini inversă*. În loc de a avea o intrare în tabelă pentru fiecare pagină virtuală, avem câte o intrare pentru fiecare cadrus fizic. Deci în loc să se facă corespondență:

-pagină virtuală → cadrus fizic

se face o corespondență inversă:

-cadru fizic → pagină virtuală.

Când se translatează o adresă logică, se caută în tabela de pagini inversă numărul paginii logice și se returnează cadrul fizic corespunzător. În acest mod se face o reducere a numărului de intrări în pagină dar căutarea are o viteză mai mică, deoarece trebuie căutată întreaga tabelă.

#### 5.4.3. Concluzii privind paginarea

Principalul avantaj al paginării este eliminarea completă a fragmentării externe. Nu dispare însă și fragmentarea internă, deoarece poate rămâne un spațiu nefolosit dar alocat proceselor, fiindcă dimensiunea proceselor nu este un multiplu exact al lungimii paginilor.

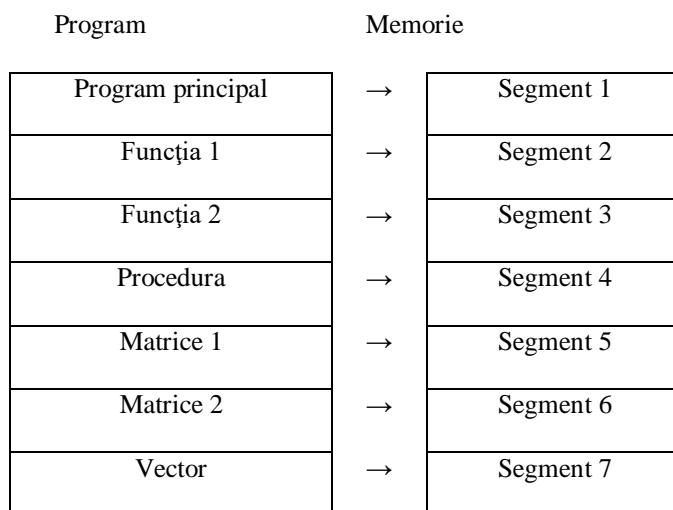
Un alt avantaj al paginării este posibilitatea de partajare a memoriei. Două sau mai multe pagini pot vedea aceeași zonă de memorie încărcând paginile logice în același cadrus fizic. Singura soluție este ca în acel cadrus fizic să fie memorat *cod reentrant*, adică un cod care nu se poate automodifica în timpul execuției. Datorită proprietății de reentrantă, este posibil ca două sau mai multe procese să execute simultan același cod, fiecare proces păstrând o copie a registrilor și a datelor proprii. În memoria fizică este necesar să se păstreze o singură copie a codului comun, fiecare tabelă de pagină indică spre același cadrus, în timp ce paginile corespunzătoare datelor proceselor sunt memorate în cadre diferite.

Un dezavantaj al paginării este faptul că fiecare acces la memorie presupune un acces suplimentar la tabela de pagini pentru calculul de adresă.

#### 5.4.4. Segmentarea memoriei

Segmentarea reprezintă o vedere a memoriei din punctul de vedere al utilizatorului care percepememoria nu ca pe o succesiune de cuvinte, așa cum este în realitate, ci ca pe o mulțime de bucăți de memorie de diverse dimensiuni. Aceste segmente pot cuprinde: programul principal, proceduri, funcții, stive, vectori, matrici etc.

Segmentarea este o schemă de administrare a memoriei în care programul este divizat în mai multe părți funcționale. Spațiul logic de adresare al programului este și el împărțit în segmente. Fiecarui segment de memorie îi corespunde o unitate funcțională a programului.



**Fig. 5.4. Principiul segmentării.**

Fiecare segment are un nume și o dimensiune, deci: -un nume -un deplasament. Programatorul vede spațiul virtual de adresare al programului ca un spațiu bidimensional, nu un spațiu unidimensional ca la programare.

#### 5.4.5. Segmentare paginată

A fost introdusă de sistemul de operare MULTICS al lui Tannenbaum și încearcă să îmbine avantajele celor două metode, de paginare și de segmentare. Fiecare segment este împărțit în pagini. Fiecare proces are o tablă de segmente, iar fiecare segment are o tabelă de mapare a paginilor. Adresa virtuală se formează din: segment, pagină și deplasament. Adresa fizică se formează din cadru de pagină și deplasament. În segmentarea paginată se elimină două dezavantaje ale segmentării pure: alocarea contiguă a segmentului și fragmentarea externă.

#### 5.4.6. Memorie virtuală

Alocarea prin memorie virtuală are capacitatea de a aloca un spațiu de memorie mai mare decât memoria internă disponibilă. Pentru aceasta se utilizează paginarea sau segmentarea combinate cu swappingul. Memoria virtuală este o tehnică ce permite execuția proceselor chiar dacă acestea nu se află integral în memorie. Metoda funcționează datorită „localității” referințelor la memorie. Numai un subset din codul, respectiv datele, unui program sunt necesare la un moment arbitrar de timp. Problema constă în faptul că sistemul de operare trebuie să prevadă care este subsetul dintr-un moment următor. Pentru aceasta se apelează la principiul localității (vecinătății), enunțat de J.P. Denning în 1968. Acest principiu are două componente:

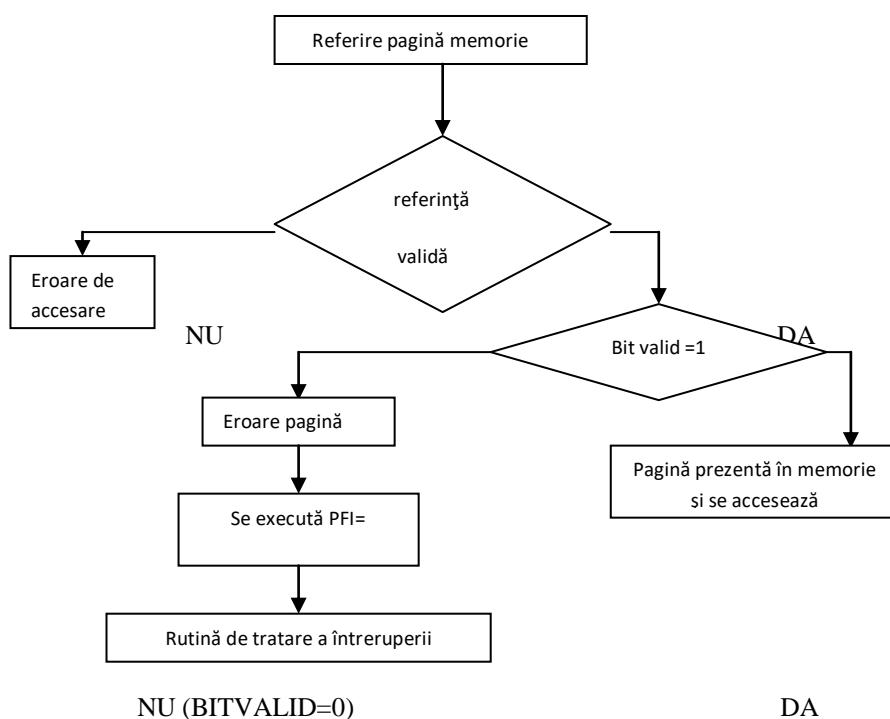
-localitate temporară – tendința de a accesa în viitor locații accesate deja în timp;

-localitate spațială – tendința de a accesa în viitor locații cu adrese apropiate de cele accesate deja.

Alocările cele mai des utilizate în memoria virtuală sunt: -alocarea de tip paginare la cerere; -alocarea de tip segmentare la cerere.

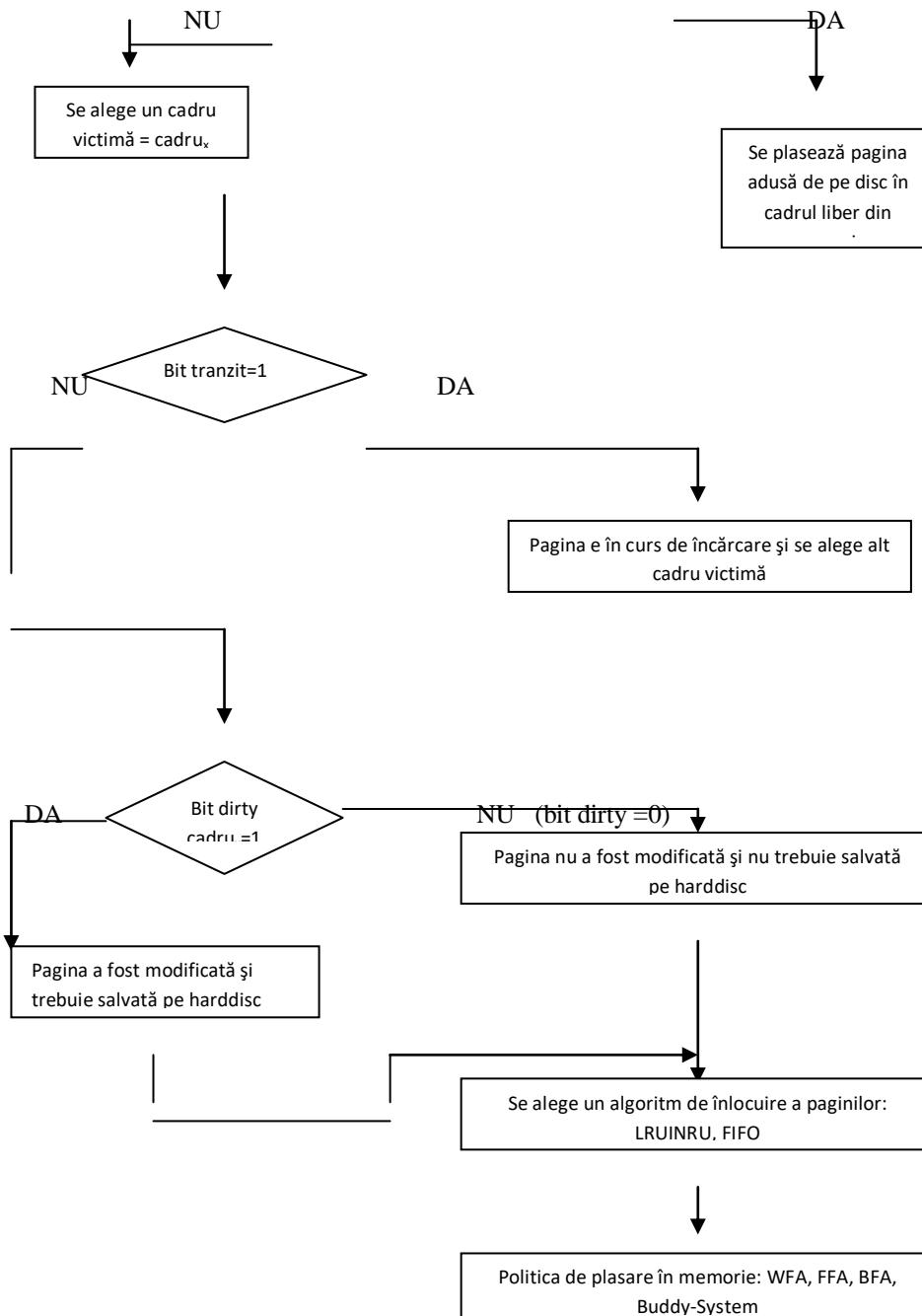
##### 5.4.6.1. Paginare la cerere

Paginarea la cerere îmbină tehnica de paginare cu tehnica swapping. În acest fel, paginile de pe harddisc sunt aduse în memorie numai când sunt *referite*, când este nevoie de ele. În acest mod se elimină restricția ca programul să fie în întregime în memorie. În fig. 5.5. se dă o organigramă a paginării la cerere.



**Fig. 5.6. Organigrama paginării la cerere.**

Așa cum se observă din organigramă, dacă referirea unei pagini este validă, adică dacă adresa ei este corectă, atunci primul lucru care se face este să se verifice bitul valid/nevalid. Acesta este implementat în tabelul de mapare a paginilor și dacă există un cadru liber în memorie, dacă este 0 pagina este pe harddisc și trebuie adusă în memorie. Dacă este zero, se lansează eroare de pagină, se generează o întrerupere de pagină (PFI = Page Fault Interrupt), de tip sincron, transparentă pentru utilizator, cu o prioritate superioară. În acest moment se va întrerupe execuția programului în curs iar sistemul de operare va lansa rutina de tratare a întreruperii de pagină. Aceasta va căuta un spațiu liber și, dacă există, va plasa pagina în el. Dacă nu, va trebui să aleagă o rutină, adică un cadru ce va fi înlocuit.

**Fig.5.7. Rutina de tratare a întreruperii de pagină.**



Rata erorilor de pagină este, pentru folosirea a trei cadre, de:  $r_3 = \frac{9}{12} \% = 75\%$

iar pentru folosirea a patru cadre :  $r_4 = \frac{10}{12} \% = 83\%$

În aceste două cazuri se observă că pentru utilizarea a 3 cadre rata erorilor de pagină este 75% iar când sunt 4 cadre rata erorilor crește, fiind 83%. Ne-am fi așteptat ca, odată cu creșterea numărului de cadre rata erorilor de pagină să scadă nu să crească. Acest fapt este cunoscut ca anomalia lui Belady și constituie unul din dezavantajele algoritmului FIFO.

Un alt dezavantaj al acestui algoritm este faptul că o pagină frecvent utilizată va fi foarte des evacuată pe disc și reîncărcată în memorie.

#### 5.4.7.2. Algoritmul LRU (Least Recently Used)

Algoritmul LRU alege victimă dintre paginile cele mai puțin utilizate în ultimul timp. Algoritmul se bazează pe presupunerea că pagina care a fost accesată mai puțin într-un interval de timp va fi la fel de accesată și în continuare. Ideea este de a folosi localitatea temporară a programului.

Pentru implementare este necesar să se țină evidența utilizatorilor paginilor și să se ordoneze paginile după timpul celei mai recente referințe la ele.

Teoretic, implementarea s-ar putea face cu o coadă FIFO în care o pagină accesată este scoasă din coadă și mutată la începutul ei. Totuși această implementare este destul de costisitoare.

Principalul avantaj al algoritmului LRU este faptul că el nu mai prezintă anomalia lui Belady.

Să luăm exemplul anterior, folosit la algoritmul FIFO.

Secvența	5	4	3	2	5	4	1	5	4	3	2	1	
3 cadre	5	4	3	2	5	4	1	5	4	3	2	1	
	-	5	4	3	2	5	4	1	5	4	3	2	
	-	-	5	4	3	2	5	4	1	5	4	3	$r_3 = \frac{10}{12} = 83\%$
Situații de neînlocuire a paginilor							*	*					
4 cadre	5	4	3	2	5	4	1	5	3	3	2	1	
	-	5	4	3	2	5	4	1	5	4	3	2	
	-	-	5	4	3	2	5	4	1	1	5	3	
	-	-	-	5	4	3	2	2	4	4	1	5	
Situații de neînlocuire a paginilor					*	*		*		*		*	

Se observă că odată cu creșterea numărului de cadre scade rata erorilor de pagină, deci anomalia Belady nu mai apare. Se asemenea este corectat și celălalt dezavantaj al algoritmului FIFO și anume la LRU sunt avantajate paginile frecvent utilizate, care sunt păstrate în memorie, ne mai fiind necesară evacuarea pe disc.

Există mai multe feluri de implementare ale algoritmului LRU:

- a) LRU cu contor de accese
- b) LRU cu stivă
- c) LRU cu matrice de referințe.

a) LRU cu contor de accese se implementează hard. Se utilizează un registru general al unității centrale pe post de contor. La fiecare acces la memorie, contorul va fi incrementat. La fiecare acces la o pagină, contorul este memorat în spațiul corespunzător acelei pagini în tabela de pagini. Alegerea „victimei” constă în căutarea în tabela de pagini a pagină cu cea mai mică valoare a contorului.

b) LRU cu stivă utilizează o stivă în care sunt păstrate numerele paginilor virtuale. Când este referită o pagină, este trecută în vârful stivei. În felul acesta vom găsi „victima” la baza stivei.

c) LRU cu matrice de referințe utilizează o matrice pătratică n-dimensională, binară (cu elemente 0 și 1), unde n este numărul de pagini fizice. Inițial matricea are toate elementele 0. În momentul în care se face o referință la pagina k, se pune 1 peste tot în linia k, apoi 0 peste tot în coloana k. Numărul de unități (de 1) de pe o linie arată ordinea de referire a paginii. Alegerea „victimei” se face în matrice linia cu cele mai puține cifre de 1, indicele acestei linii fiind numărul paginii fizice aleasă ca „victimă”.

#### **5.4.7.3. Algoritmul LFU ( Least Frequently Used)**

Victima va fi pagina cel mai puțin utilizată.

Ca mod de implementare se folosește un contor de accese care se incrementează la fiecare acces de pagină dar care nu este resetat periodic ca la LRU. „Victima” va fi pagina cu cel mai mic contor.

Principalul dezavantaj al acestei metode apare în situația în care o pagină este utilizată des în faza inițială și apoi nu mai este utilizată de loc; ea rămâne în memorie deoarece are un contor foarte mare.

#### **5.4.7.4. Algoritmul real Paged Daemon**

De obicei, sistemele de operare desemnează un proces sistem responsabil cu implementarea și realizarea politicii de înlocuire a paginilor pentru memoria virtuală. Un astfel de proces autonom care stă în fundal și execută periodic o anumită sarcină se numește **demon (daemon)**. Demonul de paginare poartă numele de **paged daemon**. Acesta pregătește sistemul pentru evacuarea de pagini înainte ca evacuarea să fie necesară. Obișnuit el este într-o stare dormtantă, fiind trezit de sistemul de operare atunci când numărul cadrelor libere devine foarte mic. Dintre principalele sale sarcini amintim:

-salvează pe disc paginile cu bitul dirty pe 1, efectuând astă numita operație de „curățire” a paginilor;

-utilizând un algoritm sau o combinație de algoritmi de înlocuire a paginilor, alcătuiește o listă ordonată pentru paginile ce vor fi „victime”;

-decide câtă memorie să aloce pentru memoria virtuală.

#### 5.4.7.5. Fenomenul de trashing

Atunci când procesele folosesc mai mult timp pentru activitatea de paginare decât pentru execuția propriu zisă se spune că are loc fenomenul de trashing.

Un exemplu tipic este acela când un proces are alocat un număr mai mic de cadre decât îi este necesar. Aceasta este o condiție necesară pentru apariția trashingului. Deoarece toate paginile sunt necesare pentru rularea programului, procesul va încerca să aducă din memoria externă și restul paginilor necesare. Dacă nu sunt cadre libere, este posibil ca victimele să fie chiar pagini ale procesului. Pentru acestea se vor genera din nou cereri de aduceri în memorie și astfel se poate intra la un moment dat într-un cerc vicios, când procesul va face numai cereri de paginare, și nu va mai executa nimic din programul său. Acesta este, de fapt, trashingul.

Există mai multe modalități de a evita trashingul. Se alege un algoritm de paginare care poate fi *global* sau *local*.

Algoritmii globali permit procesului să aleagă pentru înlocuire orice cadru, chiar dacă acesta este alocat altui proces.

Algoritmii locali impun fiecărui proces să folosească pentru selecție numai cadre din propriul set, numărul cadrelor asociate procesului rămânând același.

Dacă se utilizează un algoritm local, fenomenul de trashing dispare, deoarece setul de pagini asociat unui proces în memorie este influențat numai de activitatea de paginare a procesului respectiv.

#### 5.4.7.6. Concluzii privind paginarea la cerere

Principalele avantaje ale paginării la cerere sunt:

- programul este prezent doar parțial în memorie;
- se executa mai puține operații de intrare ieșire;
- la un moment dat, este necesară o cantitate mai mică de memorie;
- crește mult gradul de multiprogramare;

-în programarea la cerere nu mai este nevoie ca programatorul să scrie overlayurile, sarcina aceasta revenind sistemului de operare.

Principalul dezavantaj este că mecanismul de gestiune a memoriei, atât hard cât și soft, are o complexitate deosebită.

### 5.5.ALOCAREA SPAȚIULUI LIBER. TIPURI DE ALOCATOARE

#### 5.5.1. Alocatorul cu hărți de resurse

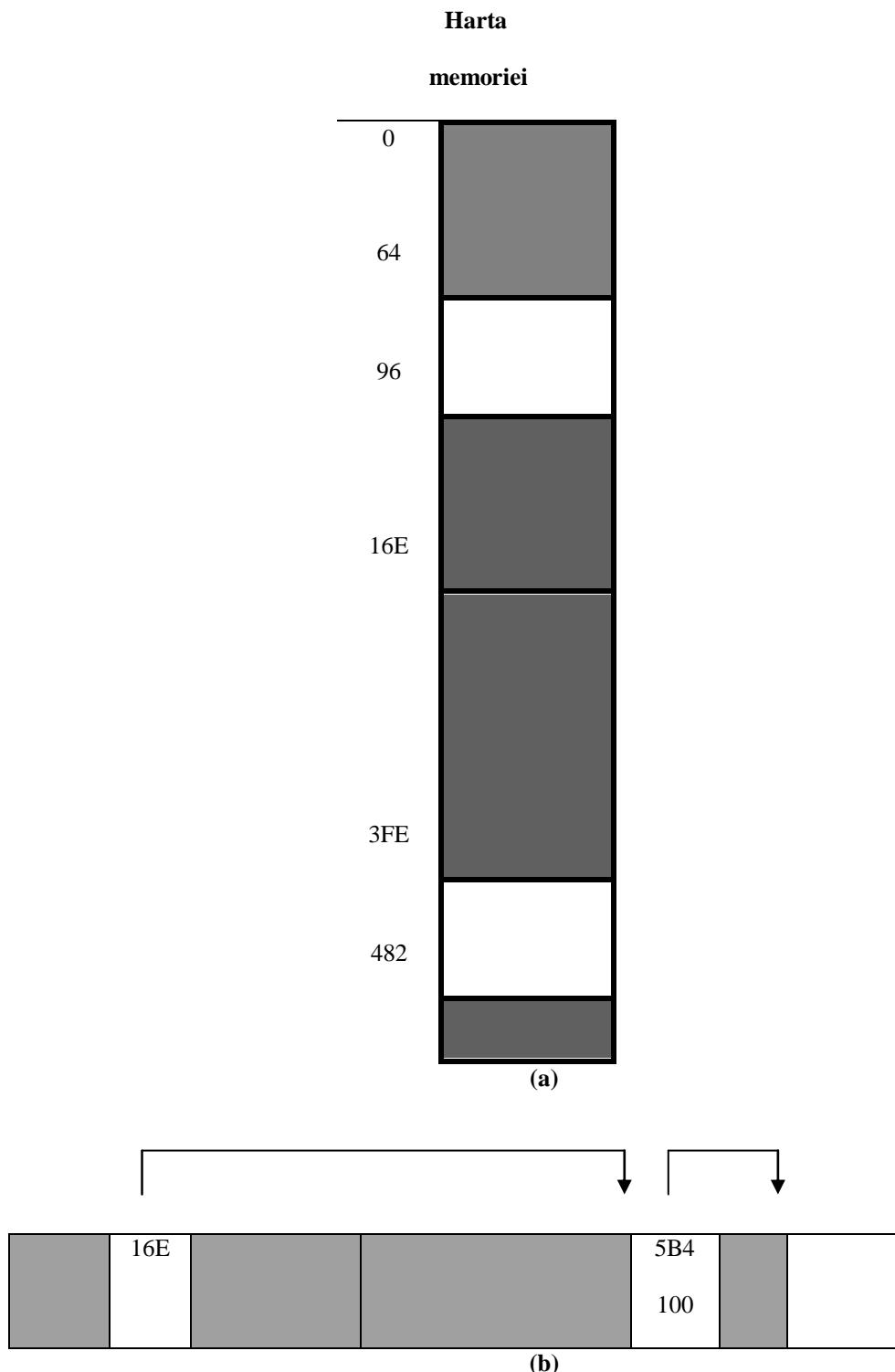
Acest alocator folosește un vector de structuri care descriu fiecare bloc liber. Iată un exemplu ce utilizează o hartă a resurselor.

**Harta resurselor**

Adresa de început a blocului (hexa)	Lungime (baiți)	Situarea blocului

0	100	Ocupat
64	50	Liber
96	200	Ocupat
16 E	400	Ocupat
3FE	100	Liber
482	50	Ocupat

**Fig. 5. 8. Harta resurselor în alocatorul cu hartă de resurse.**



**Fig. 5.9. Hărți de memorie în alocatorul cu hărți de resurse.**

Într-un alt mod de reprezentare, fiecare bloc liber conține lungimea blocului liber și adresa următorului bloc liber. Astfel, blocurile libere sunt ținute într-o structură de listă simplu înălțuită. Când alocatorul vrea să găsească un bloc liber, pleacă de la adresa primului bloc liber și parcurge lista de blocuri libere până găsește unul de dimensiune corespunzătoare. Pentru exemplul anterior, harta memoriei arată ca în fig. 5.9.(b).

Acest tip de alocare, cu hărți de resurse, este simplu dar destul de neficient. Din cauza fragmentării, complexitatea operațiilor este mare. Este posibil ca, după un anumit timp, lista de blocuri să conțină foarte multe blocuri mici și căror traversare să fie inutilă și foarte costisitoare.

### 5.5.2. Alocatorul cu puteri ale lui 2 (metoda camarazilor)

La alocatorul prezentat anterior, cel cu hărți de resurse, principalul dezavantaj este dat de căutarea unui bloc de dimensiune potrivită printre blocurile libere. Pentru a contracara acest lucru, o soluție este de a crea blocuri de dimensiuni diferite, crescătoare ca lungime, care să prezinte o ofertă mai bună în căutare. În acest sens se poate impune ca dimensiunea unui bloc să fie o putere a lui 2, deci ca un bloc să aibă  $2^k$  octeți. Tehnica de alocare este următoarea: dacă dimensiunea unei cereri să de alocare nu este o putere a lui 2, atunci se alocă o zonă a cărei dimensiune este puterea imediat superioară a lui 2, cu alte cuvinte, dacă cererea de alocare are dimensiunea cuprinsă între  $2^k$  și  $2^{k+1}$ , se alege zona cu dimensiunea  $2^{k+1}$ . Metoda se mai numește și metoda înjumătățirii, pentru că, practic, dacă există o cerere de alocare de dimensiunea  $2^k$  și aceasta nu există, atunci se alege o zonă liberă de dimensiune  $2^{k+1}$ , mai mare (dublă), care este împărțită în două părți egale. După un număr finit de astfel de operații se obține o zonă cu dimensiunea dorită și alocarea este satisfăcută.

Implementarea acestei metode este asemănătoare cu cea precedentă cu mențiunea că pentru alocatorul cu puteri ale lui 2 există liste separate pentru fiecare dimensiune  $2^k$  pentru care există cel puțin o zonă liberă. Mai trebuie menționat faptul că, atunci când două zone învecinate de dimensiune  $2^k$  devin libere, ele sunt regrupate pentru a forma o singură zonă liberă de dimensiune  $2^{k+1}$ . De aici și numele de metoda camarazilor.

### 5.5.3. Alocatorul Fibonacci

Acest alocator este asemănător cu alocatorul cu puteri ale lui 2, dar în loc să divizeze o zonă liberă în două subzone egale, o împarte în alte două de dimensiuni diferite. La fel ca în sirul lui Fibonacci, o zonă  $a_i$  este:

$$a_i = a_{i-1} + a_{i-2}$$

Când un proces își termină execuția într-o zonă ocupată, aceasta devine liberă și pot apărea următoarele situații:

-zona eliberată se află între două zone libere și atunci cele trei zone se regrupează într-o singură zonă liberă;

-zona eliberată se află între o zonă liberă și una ocupată și atunci se unesc cele două zone libere;

-zona eliberată se află între două zone ocupate și atunci zona eliberată este adăugată listelor zonelor disponibile.

### 5.5.4. Alocatorul Karel -McKusick

Acest alocator a fost construit în 1988 și este o variantă îmbunătățită a alocatorului cu puteri ale lui 2. Metoda are avantajul că elimină risipa pentru cazul blocurilor care au dimensiuni exact puteri ale lui 2. Există două îmbunătățiri majore.

a) Blocurile ocupate își reprezintă lungimea într-un vector mare de numere  $v[k]=t$ , ceea ce înseamnă că pagina k are blocuri de dimensiunea t, unde t este o putere a lui 2. De exemplu :

vectorul v

16	1024	512	32	16	64
----	------	-----	----	----	----

În acest exemplu, pagina 3 are blocuri de 512 octeți, pagina 6 are blocuri de 64 octeți etc.

b) O altă îmbunătățire este modul de calcul al rotunjirii unei puteri a lui 2. Se utilizează operatorul condițional din C (expresie 1? expresie 2: expresie 3;). Nu se folosesc instrucțiuni ci numai operatori și în felul acesta crește viteza de execuție. Acest alocator a fost utilizat pentru BDS UNIX.

#### 5.5.5. Alocatorul „slab”

Alocatorul „slab” este inspirat din limbajele orientate pe obiecte. Are zone de memorie diferite pentru obiecte diferite, formând un fel de mozaic, de unde și numele „slab” care în engleză înseamnă lespede. Iată câteva particularități ale acestui alocator:

- alocatorul încearcă, când caută zone noi, să nu acceseze prea multe adrese de memorie pentru a nu umple cache-ul microprocesorului cu date inutile; spunem că alocatorul este de tip **small foot print** (urmă mică).

- alocatorul încearcă să aloce obiecte în memorie astfel încât două obiecte să nu fie în aceeași linie în cache-ul de date;

- alocatorul încearcă să reducă numărul de operații de inițializare asupra noilor obiecte alocate.

Alocatorul „slab” constă dintr-o rutină centrală care creează alocatoare pentru fiecare obiect. Rutina primește ca parametri numele obiectului, mărimea obiectului, constrângerile de aliniere și pointere pentru o funcție construită și o funcție destinatar. Fiecare alocator are propria lui zonă de memorie în care există numai obiecte de același tip. Astfel există o zonă de pagini numai cu fișiere, o zonă de pagini numai cu date, etc. Toate obiectele dintr-o zonă au aceeași dimensiune. Fiecare alocator are o listă de obiecte care au fost de curând dealocate și le refolosește atunci când i se cer noi obiecte. Deoarece obiectele au fost dealocate, nu mai trebuie inițialiate din nou. Atunci când un alocator nu mai are memorie la dispoziție, el cere o nouă pagină în care scrie obiecte noi. Pentru că obiectele nu au fost niciodată inițializate, alocatorul cheamă constructorul pentru a inițializa un nou obiect.

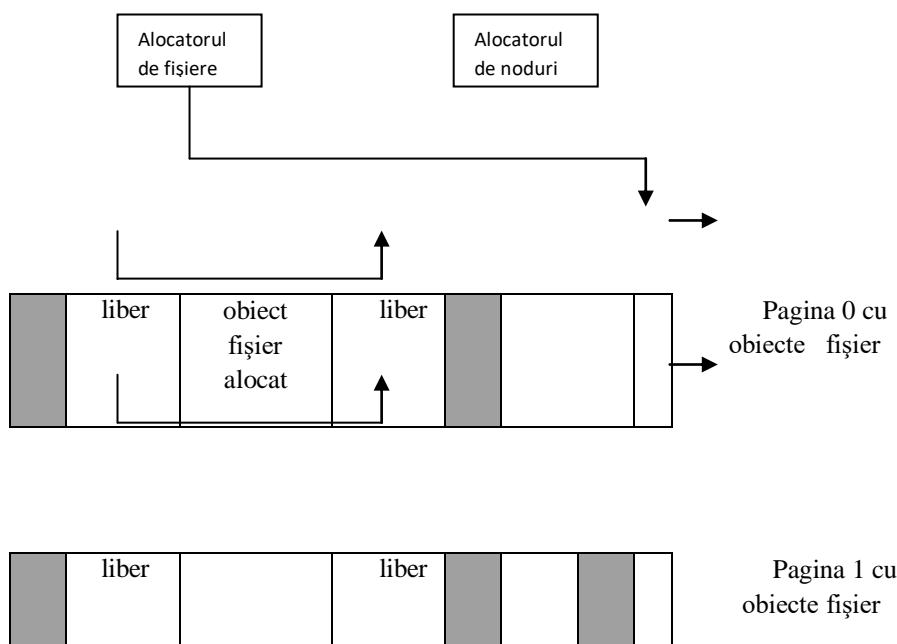


Fig 5.10. Alocatorul „slab”.

Fiecare obiect are propriul lui alocator care are în paginile sale obiecte alocate și libere. În fiecare pagină, primul obiect alocat începe la altă adresă, pentru a încărca uniform liniile din cache-ul microprocesorului. Fiecare pagină mai posedă anumite structuri de date, folosite în acest scop, ca ,de exemplu, lista dublu înlănită a tuturor paginilor pentru un anumit obiect.

Fiecare alocator folosește propriile lui pagini la fel ca alocatorul cu puteri ale lui 2. La sfârșitul unei pagini alocatorul rezervă o zonă pentru o structură de date care descrie cum este acea pagină ocupată. Primul obiect din pagină este plasat la o distanță aleatoare de marginea paginii; acest plasament are efectul de a pune obiecte din pagini diferite la adrese diferite.

Alocatorul „slab” poate returna sistemului de paginare paginile total nefolosite. El are o urmă mică deoarece majoritatea cererilor acceseză o singură pagină. Acest tip de alocator risipește ceva resurse datorită modului de plasare în pagină și pentru că are zone diferite pentru fiecare tip de obiect.

Alocatorul slab este utilizat în sistemul de operare Solaris 2.4.

## 5.6. GESTIUNEA MEMORIEI ÎN UNELE SISTEME DE OPERARE

### 5.6.1. Gestiunea memoriei în Linux

In Linux se alocă și se elibereză pagini fizice, grupuri de pagini, blocuri mici de memorie.

În ceea ce privește administrarea memoriei fizice, alocatorul de pagini poate aloca la cerere intervale contigüe de pagini fizice. Politica de alocare este cea bazată pe puteri ale lui 2, metoda camarazilor.

Pentru administrarea memoriei virtuale, nucleul Linux rezervă o zonă de lungime constantă din spațiul de adresare al fiecărui proces pentru propriul său uz intern. Această zonă conține două secțiuni:

- o secțiune statică care conține tabela de pagini cu referișe la fiecare pagină fiică disponibilă în sistem, astfel încât să existe o translație simplă de la adresele fizice la adresele virtuale atunci când se rulează codul nucleului;

- o secțiune care nu este rezervată pentru ceva anume.

### 5.6.2. Gestiunea memoriei în Windows NT

Administratorul de memorie din sistemul de operație Windows lucrează cu procese nu cu threaduri. Se utilizează paginarea la cerere, cu pagini de dimensiune fixă, maximum de 64 Kb ( la Pentium se utilizează o pagină de 4Kb). O pagină poate fi în următoarele stări:

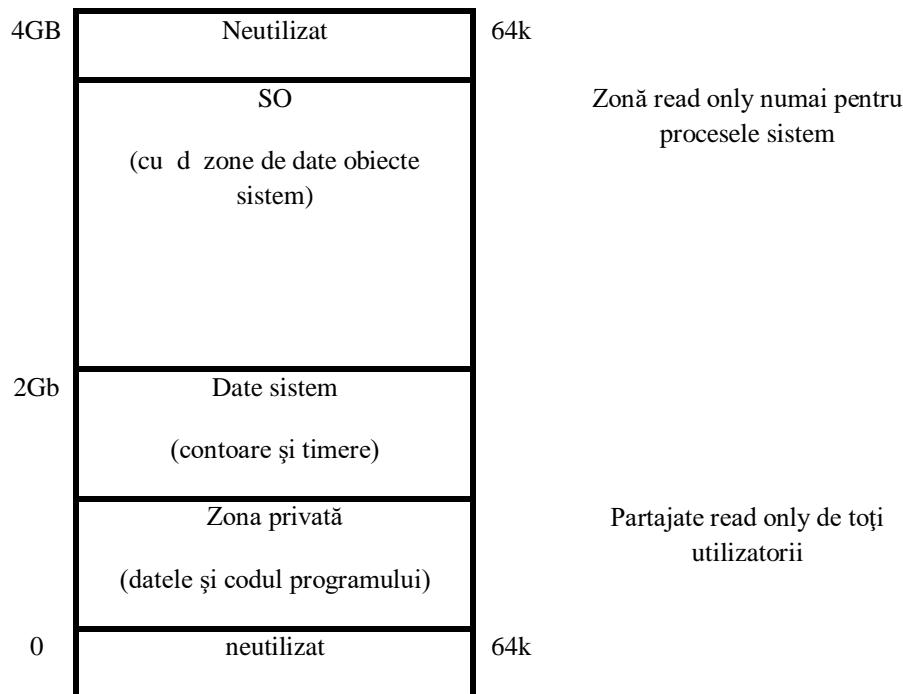
- liberă

- rezervată

- angajată (committed).

Paginile libere și cele rezervate au pagini shadow pe disc, iar accesul la ele provoacă întotdeauna eroare de pagină.

SO WINDOWS poate utiliza maximum 16 fișiere de swap. De menționat că segmentarea nu este utilizată.



**Fig. 5.11. Schema generală de administrare a memoriei WINDOWS.**

## 6. GESTIUNEA SISTEMULUI DE INTRARE/IEŞIRE

### 6.1. DEFINIREA SISTEMULUI DE INTRARE/IEŞIRE

Toate computerele au dispozitive fizice pentru acceptarea intrărilor și producerea ieșirilor. Există multe astfel de dispozitive: monitoare, tastaturi, mausuri, imprimante, scanere, hard discuri, compact discuri, floppy discuri, benzi magnetice, modemuri, ceasuri de timp real etc. Ele se numesc *dispozitive periferice*, deoarece sunt exterioare unității centrale.

O funcție importantă a sistemului de operare este gestionarea dispozitivelor periferice ale unui calculator. Fiecare SO are un subsistem pentru gestionarea lor, sistemul intrare/ieșire (**I/O, input/output**). Parte din softul I/O este independent de construcția dispozitivelor, adică se aplică multora dintre ele. Altă parte, cum ar fi driverele dispozitivelor, sunt specifice fiecărui.

Principalele funcții pe care trebuie să le genereze SO în acest scop sunt:

- generarea comenziilor către dispozitivelor periferice;
- tratarea întreruperilor specifice de intrare/ieșire;
- tratarea eventualelor erori de intrare/ieșire;
- furnizarea unei interfețe utilizator cât mai standardizată și  
cât mai flexibilă.

Gestiunea sistemului de intrare/ieșire este o sarcină destul de dificilă iar generalizările sunt greu de făcut, din diferite motive printre care:

- viteze de acces foarte diferite pentru diferitele periferice;
- unitatea de transfer diferă de la un periferic la altul ea putând fi: octet, caracter, cuvânt, bloc, înregistrare;
- datele de transfer pot fi codificate în diverse feluri, depinzând de mediul de înregistrare al dispozitivului de intrare/ieșire;
- tratarea erorilor se face în mod diferit, în funcție de periferic, deoarece și cauzele erorilor sunt foarte diferite;
- este greu de realizat operații comune pentru mai multe periferice dat fiind tipul lor diferit dar și datorită operațiilor diferite, specifice fiecărui dispozitiv.

Atunci când se proiectează un SO, pentru gestiunea sistemului de intrare/ieșire se pot avea în vedere următoarele obiective:

- a) Independentă față de codul de caractere. În acest sens, sistemul de I/O ar trebui să furnizeze utilizatorului date într-un format standard și să recunoască diversele coduri utilizate de periferice.
- b) Independentă față de periferice. Se urmărește ca programele pentru periferice să fie comune pentru o gamă cât mai largă din aceste dispozitive.

De exemplu, o operație de scriere într-un dispozitiv periferic ar trebui să utilizeze un program comun pentru cât mai multe tipuri de periferice.

Desigur, apelarea perifericelor într-un mod uniform face parte din acest context. Astfel în UNIX și WINDOWS această problemă este rezolvată prin asocierea la fiecare dispozitiv a unui fișier, dispozitivele fiind apelate prin intermediul *numelui* fișierului.

c) Eficiența operațiilor. Pot apărea deseori aşa numitele erori de ritm, datorate vitezei diferite de prelucrare a datelor de către unitatea centrală, pe de o parte, și de către periferic, pe de altă parte. Revine ca sarcină sistemului de operare să „fluidizeze” traficul.

Un exemplu de eficientizarea unei operații de intrare /ieșire este sistemul cache oferit de SO. Prin utilizarea mai multor tampoane cache, SO reușește să îmbunătățească, de multe ori, transferul de date, să-i mărească viteza.

## 6.2. CLASIFICAREA DISPOZITIVELOR PERIFERICE

Există mai multe criterii după care se pot clasifica perifericele.

a) Din punct de vedere funcțional

-*Periferice de intrare/ieșire*, utilizate pentru schimbul de informații cu mediul extern, cum ar fi: imprimanta , tastatura , monitorul.

-*Periferice de stocare*, utilizate pentru păstrarea nevolatilă a informației, cum ar fi hard discul, compact discul, floppy discul, banda magnetică.

Perifericele de stocare, la rândul lor, pot fi clasificate după variația timpului de acces, astfel

-*Periferice cu acces secvențial*, la care timpul de

acces are variații foarte mari, cazul tipic fiind banda  
magnetică.

-*Perifericele cu acces complet direct*, la care timpul  
de acces este constant, exemplul tipic fiind o  
memorie RAM.

-*Periferice cu acces direct*, la care timpul de acces  
are variații foarte mici, exemplul tipic fiind hard  
discul.

b) Din punctul de vedere al modului de operare, de servire a cererilor

-*Periferice dedicate*, care pot deservi un singur proces la un moment dat, de exemplu imprimanta.

-*Periferice distribuite*, care pot deservi mai multe procese la un moment dat, cum ar fi hard discul.

c) Din punctul de vedere al modului de transfer și de memorare a informației

-*Periferice bloc*, care memorează informația în blocuri de lungime fixă. Blocul este unitatea de transfer între periferic și memorie, fiecare bloc putând fi citit sau scris independent de celealte blocuri. Structura unui bloc este formată din partea de date propriu zisă și din informațiile de control al corectitudinii datelor ( paritate, checksum, polynom etc). Din această categorie fac parte hard discul, compact discul, banda magnetică.

-*Periferice caracter*, care utilizează siruri de caractere cărora nu le conferă structură de blocuri. Octeții din aceste siruri nu sunt adresabili și deci nu pot fi accesati prin operația de căutare. Fiecare octet este disponibil

ca un caracter curent, până la apariția următorului caracter. Din această categorie fac parte imprimanta, monitorul, tastatura.

-*Periferice care nu sunt nici bloc nici caracter*. Există și periferice care nu pot fi încadrate în nici una din aceste două categorii. Un exemplu este ceasul de timp real, cel care are rolul de a genera întreruperi la intervale de timp bine determinate.

### 6.3. STRUCTURA HARD A SISTEMELOR DE INTRARE/IEȘIRE

Părțile hard componente ale unui sistem de intrare ieșire, a cărui schemă este dată în fig. 6.1., sunt:

-*controllerul*;

-*perifericul propriu zis*.

Controllerul este alcătuit dintr-o serie de registre de comenzi și de date. Pentru un periferic simplu, ca tastatura sau imprimanta, există un singur registru de comenzi și unul de date. Pentru hard disc există mai multe registre de comenzi și unul de date. Lungimea regisrelor este funcție de arhitectura calculatorului, obișnuit sunt pe 16 biți.

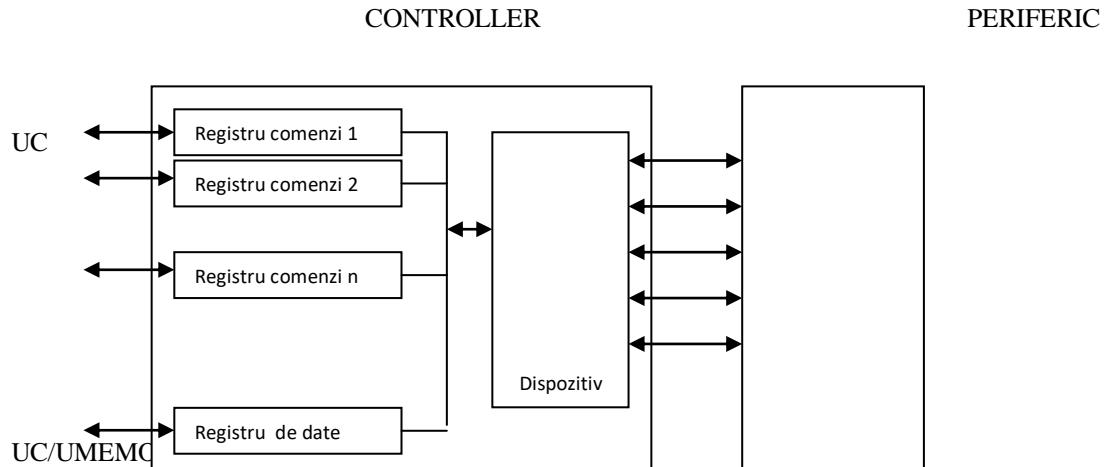


Fig. 6.1. Schema bloc a unui sistem intrare/iesire.

Secvența de lucru pentru o operație de intrare/iesire este următoarea: Când UC detectează o operație de intrare/iesire, transmite principalii parametri ai operației pe care îi depune prin intermediul bus-ului în regisrelle de comenzi și date. În continuare, controllerul, pe baza datelor din regisrelle, va sintetiza comenzi pentru echipamentul periferic, comenzi pe care le pune în interfața controller/periferic. Dacă este o operație de scriere, va pune și informațiile pe liniile de date. Perifericul, pe baza acestor comenzi, execută operația și returnează pe interfață cu controlerul răspunsurile la această operație și eventualele erori. Controlerul le va prelua și le va pune în regisrelle de comenzi și date și le va transmite la UC.

Cel mai simplu regisru de comenzi arată astfel:

1                          6                          7

GO		IE	RDY	
----	--	----	-----	--

Bitul 1, de obicei bitul de GO, are rolul de a porni efectiv operația.

Bitul 6, IE (interruption enable), are rolul de a masca sau nu întreruperea de intrare/iesire.

Bitul 7, RDY, este bitul de READY, care arată dacă perifericul este în stare liberă de a primi alte comenzi.

#### 6.4. STRUCTURA SOFT A SISTEMELOR DE INTRARE/IEŞIRE

Pe suportul hard prezentat, sistemul de operare furnizează și controlează programe care să citească și să scrie date și comenzi în registrele controllerului și sincronizează aceste operații. Sincronizarea se efectuează, de obicei, cu ajutorul întreruperilor de intrare ieșire.

Pentru a realiza aceste sarcini, softul sistemului de intrare/ieșire are patru componente:

- rutinile de tratare a întreruperilor;
- driverele, asociate dispozitivelor periferice;
- programe independente de dispozitivele periferice;
- primitivele utilizator.

##### 6.4.1.-Rutina de tratare a întreruperilor

Un proces, care are încorporată o operație intrare/ieșire, este blocat în urma unei operații de *wait* la semafor. Se inițiază operația de intrare/ieșire iar când operația este terminată, procesul este trezit printr-o operație *signal*, cu ajutorul rutinei de tratare a întreruperilor. În acest fel, procesul va fi deblocat și va trece în starea *ready*.

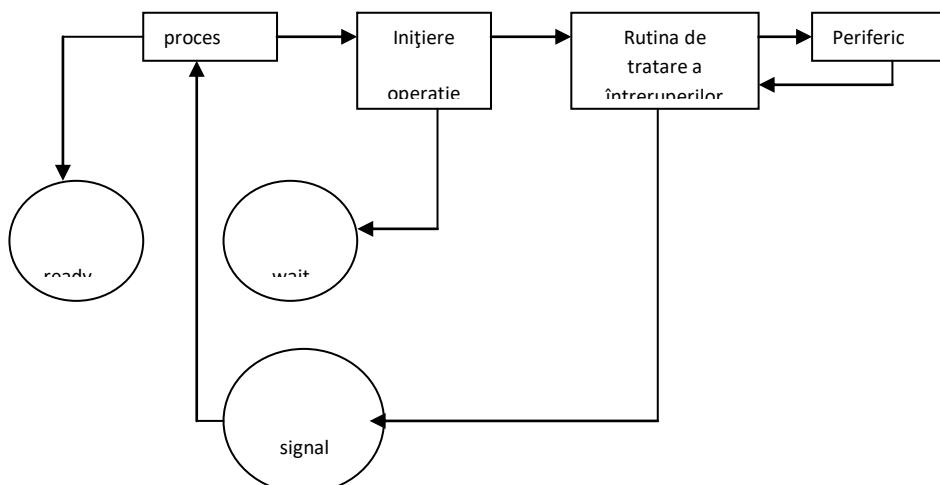
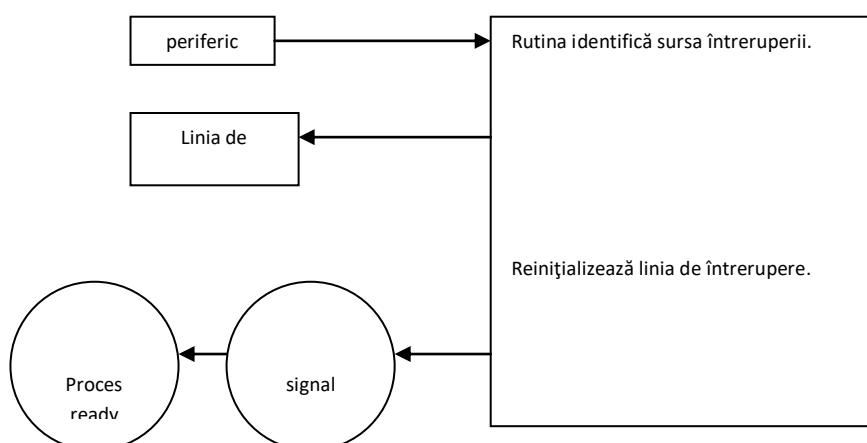


Fig.6.2. Rutina de tratare a întreruperilor.



### Fig.6.3. Reprezentarea rutinei de întrerupere.

Se observă că sarcinile principale ale rutinei de tratare a întreruperilor sunt:

- rutina identifică dispozitivul periferic care a generat o întrerupere;
- rutina reinițializează linia de întrerupere pe care a utilizat-o perifericul;
- memorează într-o stivă starea dispozitivului periferic;
- are rolul de a „trezi” procesul care a inițiat operația de I/E printr-o operație *signal*.

În acest mod procesul va fi deblocat și trecut în starea *ready*.

În realitate lucrurile nu sunt aşa de simple, SO având o mare cantitate de lucru în acest scop. Etapele parcuse în acest proces în care intervin întreruperile sunt:

- salvarea tuturor registrelor (și PSW);
- pornirea procedurii de întreruperi;
- inițializarea stivei;
- pornirea controlorului de întreruperi;
- copierea din registrele unde au fost salvate întreruperile în tabelul procesului;
- rularea procedurii de întrerupere care va citi informația din registre;
- alegerea procesului care urmează; dacă întreruperea a fost cauzată de un proces de înaltă prioritate, acesta va trebui să ruleze din nou;
- încărcarea registrelor noului proces (inclusiv PSW);
- pornirea rulării noului proces.

#### 6.4.2. Drivere

Driverul este partea componentă a SO care depinde de dispozitivul periferic căruia îi este asociat. Rolul său este de a transpune în comenzi la nivelul registrelor controllerului ceea ce primește de la nivelul soft superior. Din punct de vedere structural, driverul este un fișier care conține diferite comenzi specifice unui dispozitiv periferic.

O primă problemă care se pune este: unde ar trebui instalate aceste drivere, în nucleu (kernel) sau în spațiul utilizator (user) ? Prezența driverelor în user ar avea avantajul unei degrevări a kernelului de anumite solicitări și al unei mai bune izolări a driverelor între ele. De asemenea, la o eroare necontrolată provenită din driver, nu ar mai ceda sistemul de operare aşa ușor.

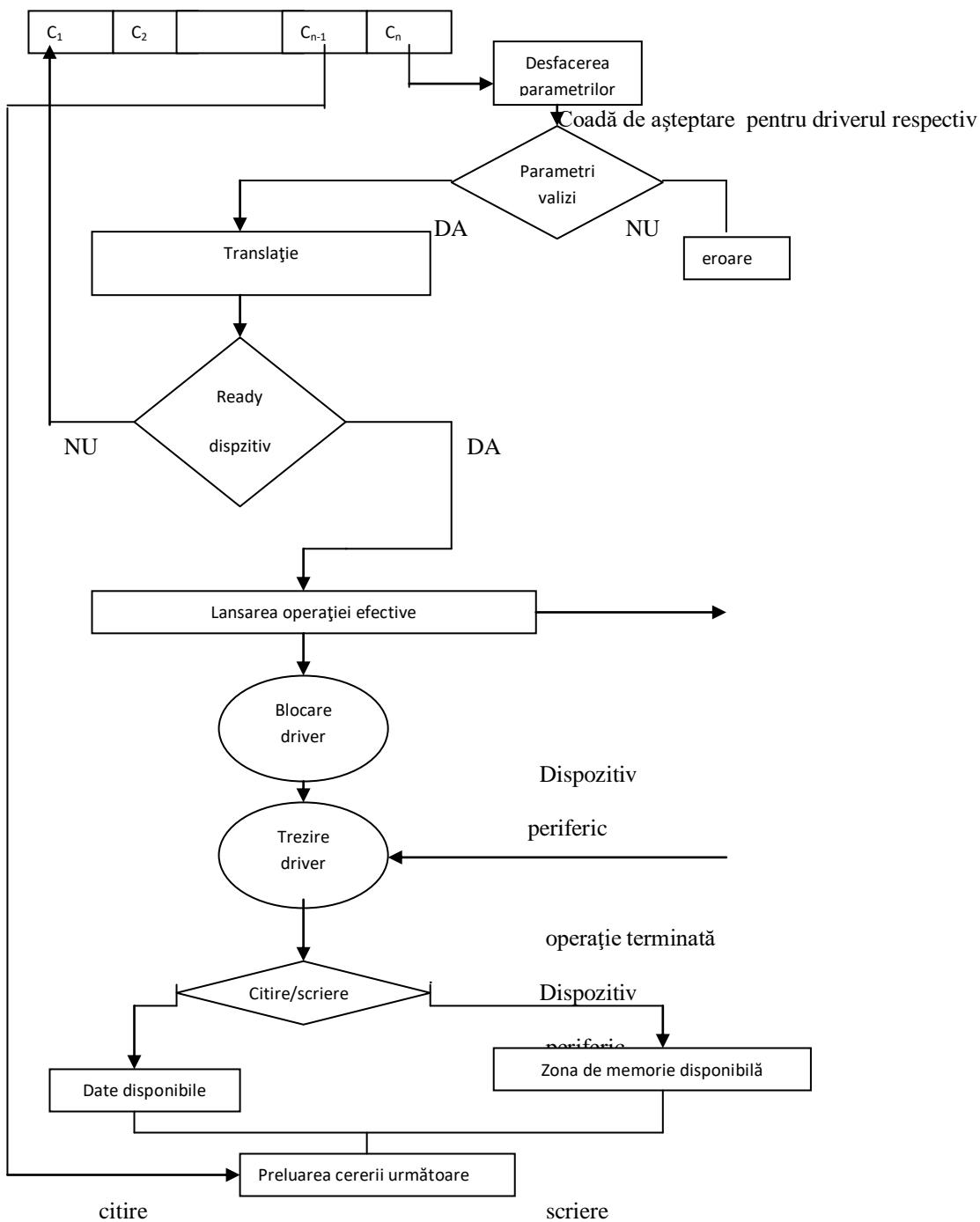
Totuși, majoritatea sistemelor de operare încarcă driverele în kernel și asta pentru o mai mare flexibilitate și o mai bună comunicare, în special pentru driverele nou instalate.

Operațiile efectuate de un driver, a cărui schemă este dată în fig. 6.4, sunt:

- verificarea parametrilor de intrare dacă sunt valizi, în caz contrar semnalarea erorii;
- translația de la termeni abstracti la termeni concreți, aceasta însemnând, de exemplu, la un hard disc;

număr bloc → adresa  $\left\{ \begin{array}{l} cap \\ cilindru \\ sector \end{array} \right.$

- verificarea stării dispozitivului ( ready sau nu);
- lansarea operației efective pentru dispozitivul periferic, de exemplu, o citire a unui sector al hard discului;
- blocarea driverului până când operația lansată s-a terminat (printr-un wait la semafor);
- „trezirea” driverului (prin operația signal);
- semnalarea către nivelul soft superior a disponibilității datelor (în cazul unei operații de citire de la periferice) sau a posibilității de reutilizare a zonei de memorie implicate în transferul de date (pentru o operație de scriere într-un periferic);
- preluarea unei noi cereri aflate în coada de așteptare.



**Fig. 6.4. Schema de funcționare a unui driver.**

Schema de funcționare a unui driver, pentru o comandă de citire a unui bloc logic de pe hard discul C, este dată în figura 6.5.

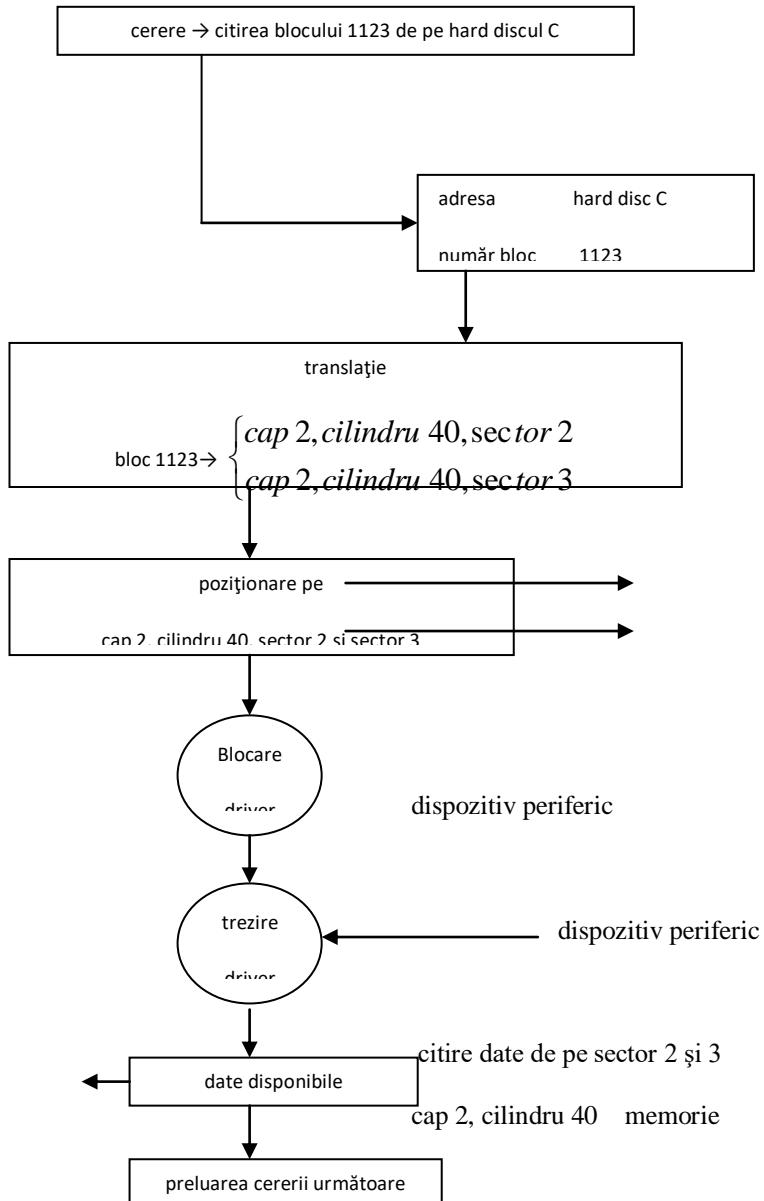


Fig. 6.5. Schema funcționării unui driver pentru o comandă de citire a unui bloc logic.

#### 6.4.3. Programme-sistem independente de dispozitive

Sarcinile ce revin acestui nivel soft sunt:

- asigurarea unei interfețe uniforme pentru toate driverele;
- realizarea corespondenței dintre numele simbolice

ale perifericelor și driverele respective; în UNIX, de exemplu, un fișier este asociat unui dispozitiv periferic;

- utilizarea unor tampoane sistem, puse la dispoziție de SO, pentru a preîntâmpina vitezele diferite ale dispozitivelor periferice;

- raportarea erorilor de intrare/ieșire în conjuncție cu aceste operații de intrare/ieșire; de obicei erorile sunt reperate și generate de drivere, rolul acestui nivel soft fiind acela de a raporta eroare ce nu a fost raportată.

#### **6.4.4. Primitive de nivel utilizator**

Cea mai mare parte a sistemului de intrare/ieșire este înglobată în sistemul de operare dar mai există niște proceduri, numite primitive de nivel utilizator, ce sunt incluse în anumite biblioteci. Aceste proceduri au rolul de a transfera parametrii apelurilor sistem pe care le inițiază. Aceste primitive pot lucra în două moduri: *sincron* și *asincron*.

O primitivă sincronă returnează parametrii numai după ce operația de intrare/ieșire a fost realizată efectiv. Se utilizează pentru operații de intrare/ieșire cu o durată ce trebuie estimată sau cu o durată foarte mică.

O primitivă asincronă are rolul numai de inițiere a operației de intrare/ieșire, procesul putând continua în paralel cu operația. Procesul poate testa mereu evoluția operației iar momentul terminării ei este marcat de o procedură, definită de utilizator, numită *notificare*. Problemele primitivelor asincrone sunt legate de utilizarea bufferului de date. Procesul inițiator trebuie să evite citirea/scrierea în buffer atât timp cât operația nu este terminată, sarcina aceasta revenindu-i programatorului. Primitivele asincrone sunt utilizate în cazul operațiilor cu o durată mare sau greu de estimat.

### **6.5. ÎMBUNĂTĂȚIREA OPERAȚIILOR DE INTRARE/IEȘIRE**

Problema esențială, la ora actuală, în ceea ce privește o operație de intrare/ieșire, este timpul de acces la hard disc, principalul dispozitiv periferic de memorare a informației. Încă există o diferență foarte mare între timpii în care se desfășoară o operație în unitatea centrală și o operație pe hard disc. Desigur, odată cu noile tehnologii, acești timpi s-au îmbunătățit și există premize de micșorare a lor în viitor, totuși decalajul de timp între operațiile din UC și cele de intrare/ieșire este încă foarte mare. Rolul SO este de încerca, prin diferiți algoritmi, să îmbunătățească, adică să micșoreze, timpul unei operații de intrare/ieșire. Multe din soluțiile adoptate au fost apoi implementate în hard, în soluțiile constructive ale hard discului. Două exemple sunt edificate în acest sens:

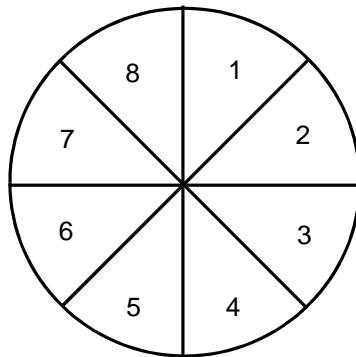
-factorul de întrețesere;

-cache-ul de hard disc.

#### **6.5.1. Factorul de întrețesere**

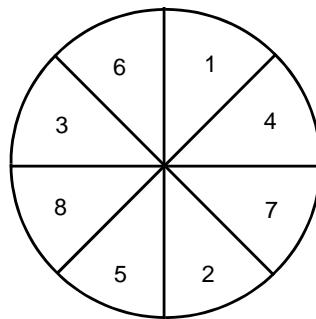
La apariția discurilor magnetice, se consuma foarte mult timp pentru citirea continuă a sectoarelor, unul după altul. Astfel pentru un disc cu 80 sectoare pe pistă, pentru citirea unei piste erau necesare 80 rotații ale platanului. Aceasta deoarece, după citirea unui sector, era necesar un timp pentru transferul datelor spre memorie iar în acest timp capul respectiv se poziționa pe alt sector. Deci, pentru citirea sector cu sector, era necesară o rotație pentru citirea unui sector. O idee simplă, venită din partea de SO, a fost de a calcula timpul de transmitere a datelor spre memorie, de a vedea pe ce sector a ajuns capul după această transmisie și de a renumerota sectoarele. Exemplu:

Fie un disc cu 8 sectoare, numerotate în ordine, ca în figura 6.6.



**Fig. 6.6. Hard disc cu 8 sectoare.**

Dacă acest disc are un factor de întrețesere egal cu 2, aceasta înseamnă că, după ce capul a mai parcurs două sectoare, datele au fost transmise în memorie. La o rotație se citesc trei sectoare. La prima rotație, sectoarele 1,2,3, la a doua rotație, sectoarele 4,5,6, la a treia rotație sectoarele 7 și 8. Pentru un factor de întrețesere 2 noua numerotare a sectoarelor va fi ca în figura 6.7.



**Fig. 6.7. Hard discul după trei rotații.**

Se observă că, prin introducerea factorului de întrețesere, nu mai sunt necesare 8 rotații pentru citirea unei piste ci numai trei.

Această soluție a fost introdusă întâi în SO, mai apoi fiind preluată în hardware de către disc, unde factorul de întrețesere era un parametru *lowlevel* al hard discului.

Datorită progresului tehnologic, de la factori de întrețesere de 3 sau 4 s-a ajuns la un factor de valoare 1, deci la o rotație se citește întreaga pistă.

### 6.5.2. Cache-ul de hard disc

Folosirea tampoanelor de date, pentru transferul de date între hard disc și memoria sistemului, a fost preluată hardware de către disc. Astfel s-a implementat o memorie cache la nivel de hard disc, memorie alcătuită pe același principiu ca și memoria cache RAM.

### 6.5.3. Crearea unui hard disc cu performanțe superioare de către SO

În general, timpul total de acces pentru un hard disc este dat de următoarele componente:

- timpul de căutare (seek time), timpul necesar pentru mișcarea mecanică a capului de scriere/citire până la pista specificată;

- latența de rotație (rotational latency) este timpul de așteptare necesar pentru ca sectorul specificat să ajungă sub capul de scriere/citire;

- timpul de transfer ( transfer time) este timpul necesar pentru a citi informația de pe sectorul specificat;

Planificarea accesului la disc are în vedere două lucruri:

- reorganizarea cererilor la disc pentru a minimiza timpul de căutare;
- un mod de plasare a informațiilor pe disc care să minimizeze latența de rotație.

Există algoritmi de plasare în care ideea principală este de a schimba ordinea de servire a cererilor venite de la procese, astfel încât să se păstreze ordinea cererilor fiecărui proces.

Dintre algoritmii care minimizează timpii de acces amintim:

- a) FCFS (First Come First Served) ;
- b) SSTF ( Shortest Seek Time Firs);
- c) SCAN, C SCAN ;
- d) LOOK, C LOOK.

a) FCFS ( First Come First Served =primul venit , primul servit). Cererile sunt servite în ordinea sosirii.

b) SSTF (Shortest Seek Time First=cel cu cel mai scurt timp de căutare, primul) . Conform acestui algoritm, capul de scriere/citire se va deplasa de la cilindrul unde este poziționat spre cilindrul cel mai apropiat. Este mai eficient decât FCFS, dar poate duce la întârzierea unor cereri și la infometare.

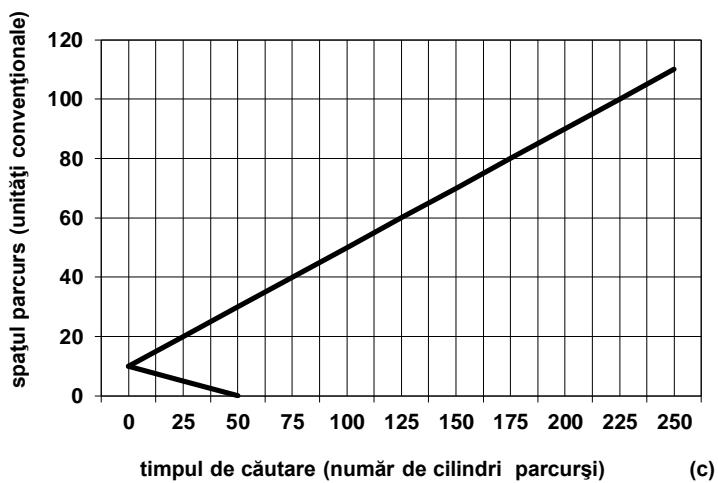
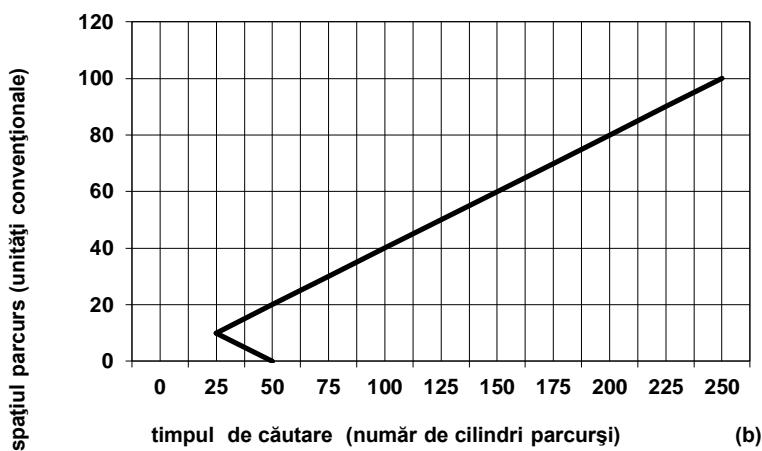
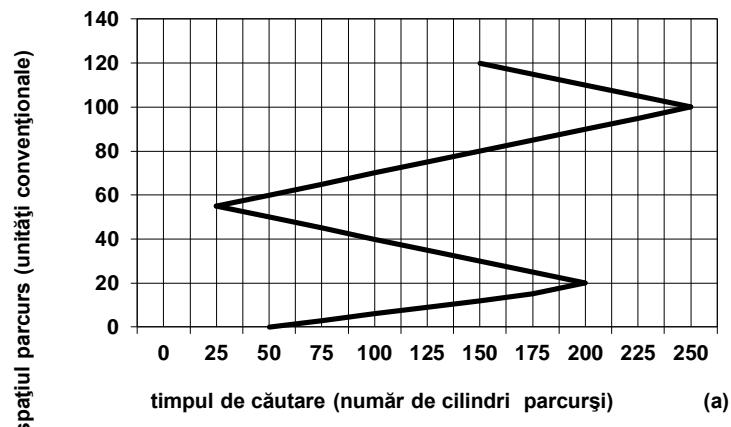
c) SCAN și C SCAN. În SCAN, capul de scriere /citire se deplasează de pe primul cilindru spre unul din capete și apoi baleiază discul până la capăt și înapoi, deservind cererile. În C SCAN , se începe de la cilindrul 0 iar când se ajunge la sfârșit, se reia baleierea tot de la cilindrul 0. Practic, în SCAN baleierea se face în ambele sensuri, în timp ce în C SCAN într-un singur sens, de la cilindrul 0 la ultimul. Prin analogie, algoritmul SCAN poate fi reprezentat de o listă dublu înăntuită, iar C SCAN de o listă simplu înăntuită.

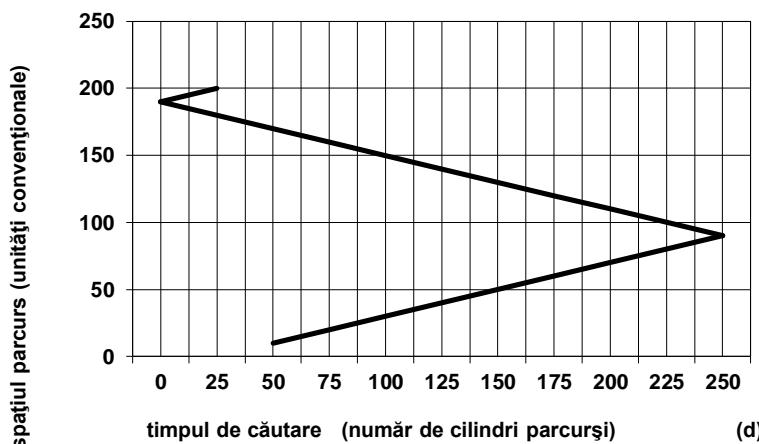
d) LOOK, C LOOK. În acest algoritm, capul de scriere/citire se deplasează până la ultima cerere, după care inversează sensul de mișcare, imediat, fără să meargă până la capătul discului.

În continuare, să luăm un exemplu și să vedem care este timpul de căutare pentru fiecare algoritm.

Fie un hard disc cu 250 cilindri, cu brațul cu capete poziționat inițial pe cilindrul 50. Fie următoarea coadă a cererilor se acces:

50, 200, 100, 25, 250, 150, 144, 143





(a)FCFS=în ordinea sosirii ; (b) SSTF=spre cel mai apropiat cilindru ; (c) SCAN ; (d) C SCAN

Să calculăm, pentru exemplul dat, timpii de căutare pentru fiecare algoritm, timpi exprimați în numărul cilindrilor parcurși.

**FCFS**-Ordinea deservirii (în ordinea cererii de acces):

50,200,100,25,250,150,144,143.

-Numărul cilindrilor parcurși:

$$(200-50)+(200-100)+(100-25)+(250-25)+(250-150)+(150-44)+(144-143)=150+100+75+225+100+6+1=\underline{\underline{657}}$$

**SSTF**-Ordinea deservirii (către cel mai apropiat cilindru):

50,25,100,143,144,150,200,250.

-Numărul cilindrilor parcurși =(50-25)+(100-25)+(143-100)+(144-

143)+(150-144)+(200-150)+(250-200)=

$$25+75+43+1+6+50+50=\underline{\underline{250}}$$

**SCAN**-Ordinea deservirii: 50,0,25,100,143,144,150,200,250

-Numărul cilindrilor parcurși = 50+25+75+43+1+6+50+50=\underline{\underline{300}}

**C SCAN**-Ordinea deservirii: 50,100,143,144,150,200,250,0,25.

- Numărul cilindrilor parcurși = 50+43+1+56+50+50+250+25=\underline{\underline{475}}

Observații:

- Cel mai timp de căutare oferă algoritmul SSTF, unul din cei mai utilizați, în ciuda faptului că poate duce la înfometare.

-Algoritmii SCAN și C SCAN se comportă mai bine pentru sistemele care au încărcare mare a discului, adică multe operații de intrare/ieșire.

- La algoritmul SSTF, cei mai favorizați din punct de vedere al timpului de acces, sunt cilindrii aflați la mijloc. De aceea, în acești cilindri sunt alocate fișiere cu frecvență ridicată de utilizare, cum ar fi structura de directoare.

-Există și algoritmi mai pretențioși în care se urmărește și minimizarea latenței de rotație.

#### 6.5.4. Construirea unui hard disc prin tehnica RAID (Redundant Arrays of Independent Disks)

Ideea RAID este de a folosi mai multe discuri simultan la aceeași magistrală și de a scrie informațiile în felii (stripes) care acoperă toate discurile. Aceste felii sunt de fapt blocuri. Discul este văzut ca un întreg dar blocurile alternează în toate discurile. Pentru un ansamblu de trei discuri, blocul 1 va fi pe discul 1, blocul 2 pe discul 2, blocul 3 pe discul 3 și apoi blocul 4 pe discul 1, blocul 5 pe discul 2 s.a.m.d. Pentru un subansamblu de **n** discuri, blocul **m** va fi pe discul **m modulo n**.

În Raid, această tehnică *a stripingului* se combină cu ideea de *redundanță*, de unde și denumirea *Redundant Arrays of Independent Disks* adică o mulțime redundantă de discuri independente. Redundanța constă în alocarea, în caz de defect, a unui disc pentru refacerea informației pe celelalte discuri.

Tehnica de refacere cea mai des utilizată este paritatea dar se pot utiliza și checksum-uri sau, mai rar, polinoame de control.

Cea mai simplă schemă RAID este cea cu două discuri, care mențin informații identice, tehnică care se mai numește *mirrored disks*.

În tehnica actuală RAID, există 7 nivele:

-nivelul 0: striping fără redundanță;

-nivelul 1: discuri oglindite (mirrored disks);

-nivelul 2: coduri Hamming detectoare de erori;

-nivelul 3: un disc de paritate la fiecare grup,

bit-interleaved;

-nivelul 4: scrieri/citiri independente, block-interleaved;

-nivelul 5: informația de paritate este împărtășiată pe toate discurile;

-nivelul 6: nivelul în care de detectează și se corectează mai mult de o eroare pe disc.

Principalul dezavantaj al discurilor RAID constă în aşa numitele „scrieri mici”. Acest lucru se produce atunci când se face o scriere într-un singur disc și toate celelalte discuri trebuie accesate pentru a se recalcule paritatea, generându-se astfel un trafic destul de mare. Acest dezavantaj poate fi contracarat prin implementarea unei cache de disc, cu o capacitate foarte mare, ceea ce face ca şoul scrisorilor mici să fie absorbit de cache, utilizatorii netrebuind să aștepte calculul parității.

## 7. GESTIUNEA RESURSELOR LOGICE. SISTEMUL DE FIŞIERE

### 7.1. NOȚIUNI INTRODUCTIVE

Pentru aplicațiile soft sunt necesare două condiții esențiale:

- stocarea informației;
- regăsirea informației.

Până acum, singura entitate pe care am studiat-o a fost procesul. Acesta, însă, nu beneficiază de aceste două condiții, un proces neputând fi stocat într-un mediu de memorare extern și, implicit, nu se poate apela la informațiile deținute de el după o anumită perioadă. De aceea se impunea introducerea unei noi entități care să satisfacă cele două condiții. Această entitate a fost denumită *fișier* iar sistemul ce gestionează fișierele se numește *sistemul de fișiere*.

Fișierul este o resursă logică a sistemului de operare și este memorat într-un mediu extern de memorare. Prin definiție, fișierul este:

- unitatea logică de stocare a informației;
- o colecție de informații definită de creatorul ei;
- o colecție de informații mapate pe periferice fizice.

Pentru a furniza un mod de păstrare a fișierelor, multe sisteme de operare au și conceptul de *director* ca o cale de a grupa mai multe fișiere.

Sistemul de fișiere dintr-un SO trebuie să realizeze următoarele operații:

-Crearea și stergerea fișierelor/directoarelor.

-Denumirea și manipularea fișierelor; fiecare fișier ar trebui să aibă un nume care să fie reprezentativ pentru conținutul său; manipularea fișierelor constă în operațiile ce se pot efectua cu ele, ca scriere , citire, căutare.

-Asigurarea persistenței datelor. Acest lucru înseamnă satisfacerea condiției de regăsire a datelor stocate, chiar și după perioade de timp foarte mari. De asemenea trebuie să existe posibilitatea de refacere a conținutului unui fișier în caz de accident, posibilitate pe care sistemul de fișiere trebuie să o pună la dispoziție.

-Asigurarea mecanismelor de acces concurrent al proceselor din sistem la informațiile stocate în fișier.

### 7.2. CLASIFICAREA FIŞIERELOR

#### 7.2.1. Clasificarea fișierelor după structură

##### 7.2.1.1. Secvență de octeți

Fișierul este alcătuit dintr-un sir de octeți. Interpretarea lor este făcută de către programele utilizator, conferindu-se astfel acestui tip de structură o maximă flexibilitate. Marea majoritatea a sistemelor de operare actuale, printre care UNIX și WINDOWS, prezintă acest tip de structură.

##### 7.2.1.2. Secvență de înregistrări

Un fișier este alcătuit dintr-o secvență de înregistrări, de obicei specifică unui anumit dispozitiv periferic. Astfel, pentru imprimantă, era linia de tipărit formată din 132 de caractere, pentru lectorul de cartele era o coloană ș.a.m.d. Acest tip de structură ține deja de domeniul istoriei.

##### 7.2.1.3. Structură arborescentă

Structura internă a unui fișier este organizată ca un arbore de căutare. Un arbore de căutare este un arbore binar ale cărui noduri au o cheie de identificare. Cheia asociată unui anumit nod este mai mare decât cheia asociată unui nod din sub-arborele drept. Cheia asociată unui anumit nod este mai mică decât cheia asociată unui nod din sub-arborele stâng. O astfel de structură este utilizată în calculatoarele de mare capacitate.

### 7.2.2. Clasificarea fișierelor după tip

#### 7.2.2.1. Fișiere normale

Aceste fișiere conțin informație utilizator și pot fi de două feluri:

- fișiere text;
- fișiere binare.

Fișierul text este format din linii de text; fiecare linie este terminată cu caracterele CR (carriage return) sau LF (line feed).

Fișierul binar este organizat ca secvențe de octeți dar sistemul de operare îl poate asocia o anumită structură internă, cazul tipic fiind al fișierelor executabile.

#### 7.2.2.2. Directoare

Sunt niște fișiere sistem destinate gestionării structurii sistemului de fișiere.

#### 7.2.2.3. Fișiere speciale de tip caracter/bloc

Sunt destinate utilizării în conjuncție cu dispozitivele periferice.

### 7.2.3. Clasificarea fișierelor după suportul pe care sunt rezidente

Din acest punct de vedere fișierele pot fi:

- fișiere pe disc magnetic;
- fișiere pe bandă magnetică;
- fișiere pe imprimantă;
- fișiere pe ecran;
- fișiere pe tastatură.

### 7.2.4. Clasificarea fișierelor după acces

#### 7.2.4.1. Fișiere cu acces secvențial

În aceste fișiere, pentru a ajunge la informație, trebuie mai întâi să se parcurgă niște structuri. Este cazul tipic al benzii magnetice. În accesul secvențial nu se pot face citiri și scrieri în același timp.

#### 7.2.4.2. Fișiere cu acces direct

Accesul se face direct într-o structură dată, existând posibilități de a se face citiri și scrieri în același timp. Este cazul tipic al hard discului.

#### 7.2.4.3. Fișiere cu acces indexat

Se face acces direct prin conținut, folosind o serie de tehnici ca fișiere de index, fișiere multi listă, B-arbore, hashing etc.

### 7.3. ATRIBUTE ȘI OPERAȚII CU FIȘIERE

#### 7.3.1. Atribute

Nume - numele fișierului este, pentru toate sistemele de operații, păstrat în formă uman inteligibilă.

Tip - acest atribut este absolut necesar deoarece SO cuprinde mai multe tipuri de fișiere(binar, text..).

Locație – este un pointer la locația fișierului pe perifericul de stocare.

Protecție – în legătură cu protecția fișierului, cele mai multe atrbute se referă la posesorul său și la drepturile de acces ale utilizatorilor săi.

Timp, data și identificatorul de utilizator – informații pentru protecția, securitatea și monitorizarea utilizării.

#### 7.3.2. Operații cu fișiere

Prezentăm, în continuare, principalele operații cu fișiere .

-Crearea unui fișier, creat().

-Citirea unui fișier; read(); se citesc din fișier un număr specificat de octeți de la o poziție curentă.

-Scrierea într-un fișier, write(); se scrie într-un fișier un număr specificat de octeți, începând de la poziția curentă.

-Deschiderea unui fișier, open(); căutarea în structura de directoare de pe disc a intrării fișierului și copierea conținutului intrării în memorie.

-Descriptorul de fișier, fd; în urma operației de deschidere a unui fișier, se returnează un întreg, numit descriptor de fișier, care va fi utilizat de procesul care a inițiat operația open() pentru toate operațiile ulterioare asupra fișierului respectiv. Descriptorul de fișier este folosit de SO ca un index într-o tabelă de descriptori asociată fiecarui proces, în care se regăsesc și alți descriptori asociați altor fișiere folosite de procesul respectiv.

-Ștergerea unui fișier, delete(), unlink().

-Trunchierea unui fișier, trunc(), înseamnă ștergerea unei zone contighe, de obicei la începutul sau sfârșitul fișierului.

-Închiderea unui fișier, close(), este mutarea conținutului unui fișier din memorie în structura de directoare de pe disc.

-Adăugarea de date la sfârșitul unui fișier, append().

-Setarea poziției în fișier, seek().

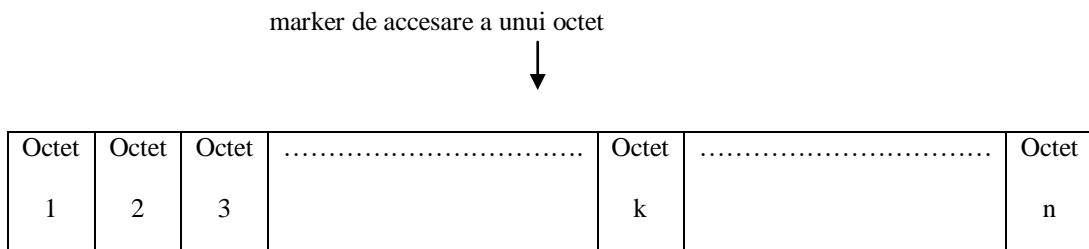
-Repoziționarea în fișier, lseek().

-Redenumirea unui fișier, rename().

-Operația de setare și citire a atributelor unui fișier, `setattributes()`.

Aceste operații prezentate au echivalent în funcțiile bibliotecă pe care utilizatorul le poate apela din program. La rândul lor, aceste funcții inițiază apele la sistem.

În sistemele de operații actuale accesarea unui fișier se face aleatoriu și nu secvențial ca în sistemele mai vechi. Aceasta înseamnă că se pot face operații de scriere/citire în orice poziție din cadrul fișierului, adică orice octet din fișier este adresabil.



**Fig. 7.1. Fișier cu secvență de octeți.**

## 7.4. IMPLEMENTAREA SISTEMULUI DE FIȘIERE

### 7.4.1. Alocarea fișierelor pe disc

Un SO lucrează la nivel logic cu blocuri. Un bloc este format din două sau mai multe sectoare fizice. Una din problemele esențiale ale implementării sistemului de fișiere este modul în care blocurile din disc sunt alocate pentru fișiere. Există, la ora actuală, trei posibilități de alocare:

- alocare contiguă;
- alocare înlanțuită;
- alocare secvențială.

#### 7.4.1.1. Alocare contiguă

În acest mod de alocare fiecare fișier ocupă o mulțime contiguă de blocuri pe disc, adică o mulțime în care elementele se ating. Avantajele acestei alocări sunt:

-este un tip de alocare foarte simplu, fiind necesare doar numărul blocului de început și numărul de blocuri ale fișierului; de exemplu, un fișier cu 8 blocuri, cu adresa blocului de început 1234, va fi reprezentat pe disc astfel:

Bloc 1234	1235	1236	1237	1238	1239	1240	1241
-----------	------	------	------	------	------	------	------

- sunt permise accese directe.

Dezavantaje:

-există o risipă de spațiu foarte mare, adică o mare fragmentare a discului datorată condiției de contiguitate a blocurilor din fișier;

-fișierele nu pot crește în dimensiune ceea ce constituie un impediment important.

Alocarea contiguă a fost folosită în primele sisteme de operare, în prezent aproape nu mai este utilizată.

#### 7.4.1.2. Alocarea înlănțuită

Fiecare fișier este o listă liniară înlănțuită de blocuri pe disc.

Fiecare bloc conține partea de informație, care este blocul propriu zis, și o legătură spre blocul următor, de tip pointer.

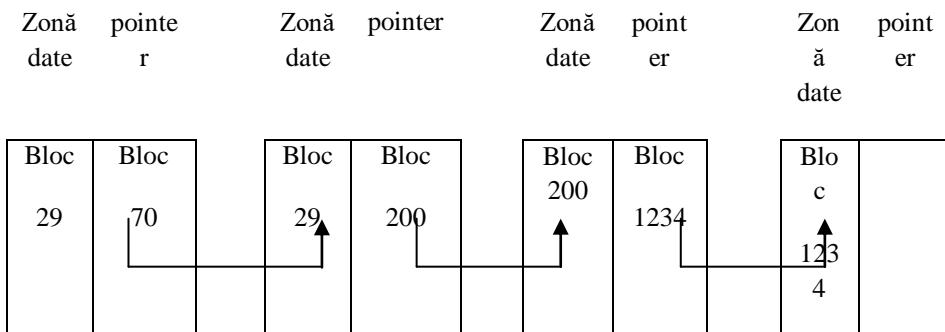
În figura 7.3. este dată o astfel de reprezentare pentru un fișier format din 4 blocuri.

Avantaje:

- blocurile unui fișier pot fi împrăștiate oriunde în disc căci nu mai există condiția de contiguitate;
- fișierul poate crește în dimensiuni pentru că oricând i se pot aloca blocuri libere de pe disc;
- accesul secvențial se face relativ ușor.

Dezavantaje:

- accesul direct se face destul de greu, datorită numeroaselor legături ce pot să apară la un moment dat;
- o corupere a lanțului de pointere, duce la distrugerea fișierului sau chiar a întregului sistem de fișiere.



**Fig. 7.3. Reprezentarea unui fișier cu 4 blocuri. Alocare înlănțuită.**

Acest tip de alocare este specific sistemelor de operare ce folosesc tabela FAT (File Allocation Table). În tabela FAT un fișier este reprezentat ca în exemplul din figura 7.4.

Nume fișier	Nr. bloc început	Nr. bloc sfârșit
test	29	1234

**Fig. 7.4. Reprezentarea unui fișier în tabele FAT.**

Sisteme de operare ce utilizează alocarea înlanțuită, deci tabela FAT, sunt: MS DOS, WINDOWS 95, WINDOWS 98, OS/2(versiunea 1 și 2).

#### 7.4.1.3. Alocarea indexată

În acest tip de alocare se pun la un loc toți pointerii, într-un bloc de index.

Alocarea indexată grupează referințele și le asociază cu un fișier particular.

Avantaje:

- se realizează un acces direct;
- aproape dispără fragmentarea externă;
- este o situație avantajoasă în cazul în care sistemul de operare are multe fișiere de dimensiune mică.

Dezavantaje:

- implică un cost suplimentar prin accesul la blocul de index.

Alocarea indexată este utilizată în sistemele UNIX, în sistemul cu i-noduri.

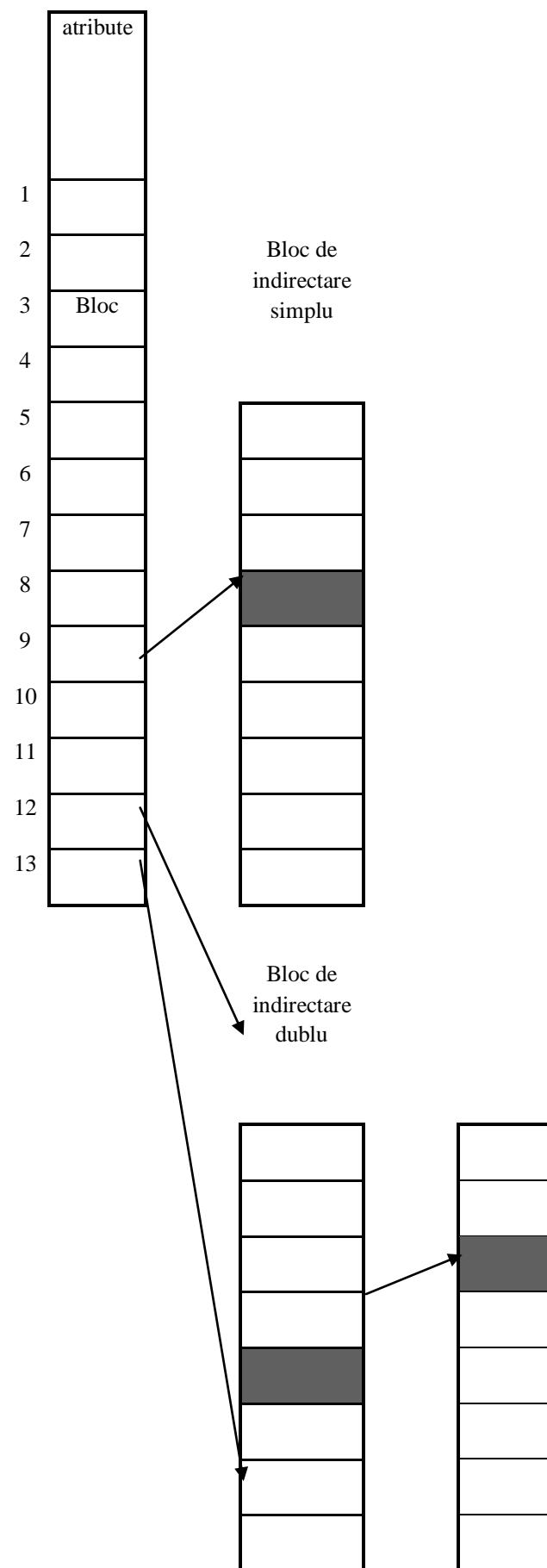
Un i-nod (index nod) conține un câmp de date în care sunt stocate atributele fișierului respectiv și o legătură spre alt bloc. Fiecărui fișier îi corespunde un i-nod.

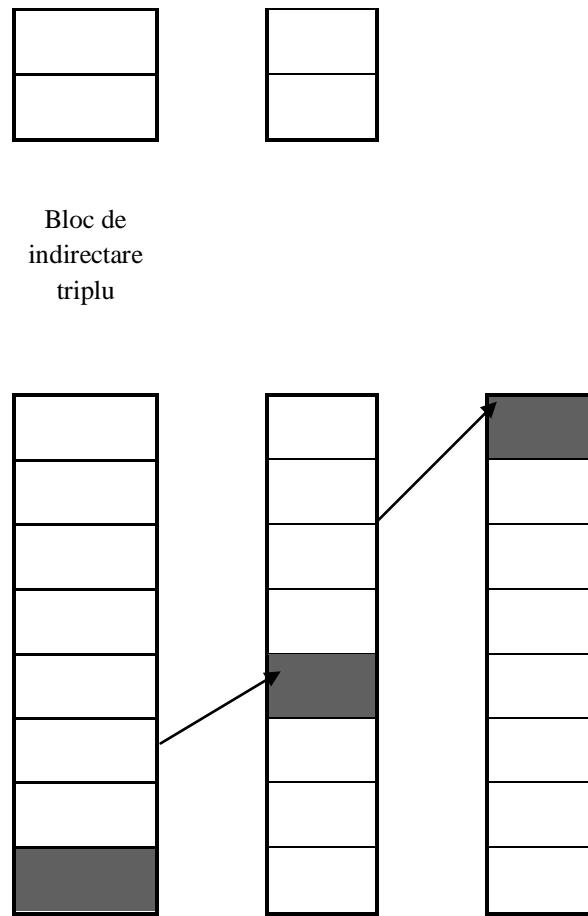
Structura unui nod de indexare este dată în figura 7.4.

Această implementare permite o indirectare până la nivel de trei blocuri.

Adresa indexată unde se găsesc informațiile din fișier poate fi:

- blocul inițial, dacă adresa este mai mică decât 10 blocuri;
- blocul următor indexat, pentru bloc de indirectare simplă;
- al doilea bloc, dat de blocul următor, pentru bloc de indirectare dublă;
- al treilea bloc, după indirectarea dată de primele 2 blocuri, pentru bloc de indirectare triplă.

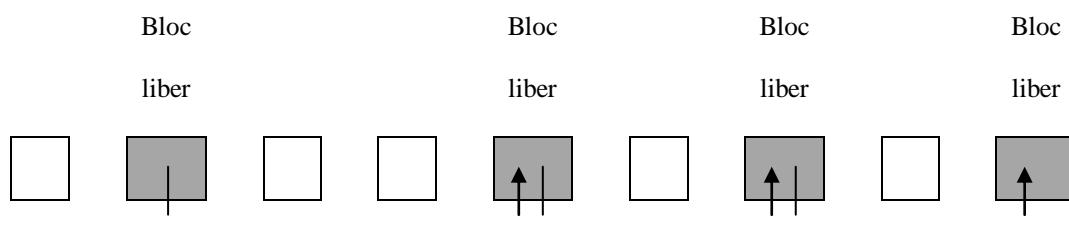




**Fig. 7.4. Structura unui i-nod ( nod de indexare)**

#### 7.4.2. Evidența blocurilor libere

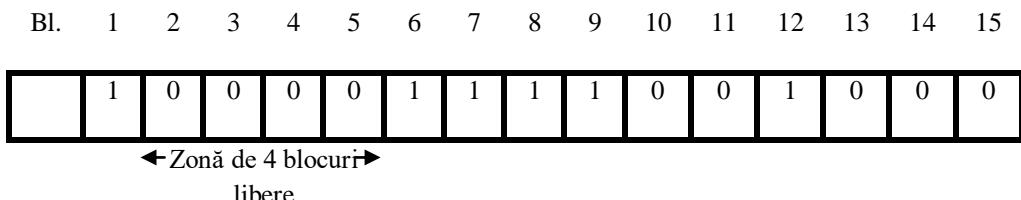
Operațiile de creare, ștergere, scriere , efectuate asupra fișierelor, sunt operații dinamice, ceea ce necesită, în orice moment, alocare/dealocare de blocuri pentru fișiere. Cum se aloca, firesc, blocuri libere în acel moment, este necesar ca sistemul de fișiere să aibă o evidență a blocurilor libere în orice moment. Cele mai utilizate metode de evidență sunt:-metoda listei înlănțuite;-metoda hărții de biți;-gruparea. În prima metodă, blocurile libere din disc sunt organizate într-o structură de listă simplu înlănțuită. Operațiile efectuate asupra acestei liste sunt: inserarea de noi noduri, pentru blocurile libere nou apărute, și ștergerea de noduri pentru blocurile ce au fost ocupate. Această listă este implementată la o adresă dintr-un bloc dinainte stabilită.



**Fig. 7.5. Lista simplu înlănțuită a blocurilor libere din disc.**

În metoda hărților de biți, se utilizează o structură de hartă de biți în care fiecare bit corespunde unui bloc de pe disc. Un bit 0 înseamnă că blocul este liber iar un bit 1 înseamnă că blocul este ocupat.

Această metodă are avantajul, față de prima, că harta de biți are o dimensiune constantă, relativ mică. În plus, localizarea unei zone libere de blocuri cu dimensiune prestabilită se face ușor, prin căutarea în hartă a unei secvențe de biți 0.



**Fig. 7.6.Hartă de biți pentru evidența blocurilor libere pe disc.**

În metoda grupării se folosește o structură care memorează zonele libere din disc, fiecare zonă fiind caracterizată prin adresa primului bloc din zona liberă și prin numărul de blocuri.

#### 7.4.3. Eficiența și performanța sistemului de fișiere

Eficiența unui sistem de fișiere depinde în primul rând de doi factori:

- de algoritmii de alocare a discului, discutați anterior;
- de tipurile de date păstrate în intrarea fișierului din director.

Performanța unui sistem de fișiere poate fi îmbunătățită prin trei factori:

-*disc cache*; aceasta înseamnă o secțiune bine determinată a memoriei principale, utilizată ca memorie cache a blocurilor de disc cele mai des utilizate;

-*free-behind și read ahead* care sunt tehnici de optimizare a accesului secvențial;

-*RAM disc* care înseamnă declararea unei porțiuni de memorie principală ca un disc virtual.

#### 7.4.4. Fiabilitatea sistemelor de fișiere

Este necesar ca sistemele de fișiere să ofere instrumente care să permită următoarele lucruri:

- a)-Evitarea distrugerii informației pe disc.
- b)-Recuperarea informațiilor în urma unor erori hard sau soft.
- c)-Asigurarea consistenței sistemului de fișiere.

##### 7.4.4.1. Evitarea distrugerii informației pe disc

În acest context, una dintre cele mai importante probleme este evidența blocurilor defecte (bad blocks). Această problemă este rezolvată atât la nivel hard cât și la nivel soft.

La nivel hard, până în anul 1993, era permis unui producător de hard discuri ca acestea să aibă un număr limitat de sectoare defecte, sectoare ce erau stanțate pe hard disc. După 1993, standardele internaționale nu au mai permis producătorilor să scoată pe poarta fabricii discuri cu sectoare defecte.

La nivel soft, sistemul de operare, prin intermediul sistemului de fișiere, are încorporat un fișier *bad bloc* în care sunt trecute toate blocatele defecte ale hard discului. Un bloc situat în acest fișier nu va fi utilizat de sistemul de fișiere, fiind ignorat. Există două tipuri de metode pentru a determina dacă un bloc este defect și pentru a-l trece în fișierul de bad-bloc:

-metode *offline*;

-metode *online*.

Metodele *offline* constau în a rula programe fără ca sistemul de operare să fie încărcat, deci a efectua teste independent de sistemul de operare. La sistemele mai noi acest lucru se face, de obicei, prin formatarea *lowlevel*, când se determină blocatele defecte, se trec într-o memorie locală a discului și apoi, la încărcarea SO, ele vor fi trecute în fișierul *bad block*. La sistemele mai vechi, când nu există noțiunea de formatare a discului, se treceau teste, se determinau sectoarele defecte și se deruttau aceste sectoare, în sensul că se treceau pe pista de rezervă a discului și nu mai erau considerate blocate defecte. Metodele *online* se desfășoară atât timp cât sistemul de operare este activ. De obicei, sistemul de fișiere are înglobat un program care testează blocatele unui disc și le declară defecte dacă este cazul.

#### 7.4.4.2. Recuperarea informației în urma unor erori soft sau hard

Recuperarea unui fișier poate fi făcută prin următoarele metode:

-prin mecanisme de *back-up*;

-prin sisteme jurnalizate.

Un mecanism de back-up înseamnă copierea întregului sistem de fișiere pe alt periferic decât discul, adică pe bandă magnetică sau, mai adesea, pe streamer. Această operație de salvare se face periodic, perioada fiind hotărâtă de inginerul de sistem, și se numește *massive dump* sau *periodic dump*. Principalul ei dezavantaj constă în timpul lung ce necesită. Există și o variantă ce necesită timp mai scurt, numită *incremental dump*, care constă în salvarea pe bandă sau pe streamer numai a fișierelor ce au fost modificate de la ultima operație de salvare.

Sistemele jurnalizate înregistrează fiecare actualizare a sistemului de fișiere ca pe o tranzacție. Toate tranzacțiile sunt scrise într-un jurnal. O tranzacție este considerată comisă atunci când este scrisă în jurnal. Tranzacțiile din jurnal sunt operate asincron în sistemul de fișiere. După ce sistemul de fișiere este modificat, tranzacție este ștersă din jurnal. În momentul în care sistemul de fișiere a căzut dintr-o eroare de hard sau soft, se repornește sistemul de operare și tranzacțiile din jurnal vor fi, toate, actualizate.

#### 7.4.4.3. Asigurarea consistenței sistemului de fișiere

Probleme de consistență a sistemului de fișiere apar în cazul în care unele modificări făcute sistemului de fișiere nu au fost actualizate pe disc. Verificarea consistenței sistemului de fișiere se face, atât la nivel de bloc cât și la nivel de fișiere, cu ajutorul unor programe utilitare.

Un exemplu de program care verifică consistența la nivel de bloc, din sistemul UNIX, este cel dat în continuare.

C1- indică numărul de apariții ale blocului respectiv în toate i- nodurile.

C2- indică numărul aparițiilor blocului în lista de blocuri libere.

În mod normal trebuie să avem  $C1+C2=1$

**Tabelul 7.1. Situațiile anormale care pot să apară în asigurarea consistenței sistemului de fișiere.**

C1	C2	Situație apărută	Corecție
0	0	Se irosește un bloc ce nu va putea fi alocat	Se face C2 = 1
1	1	Există riscul ca blocul să fie alocat mai multor fișiere; pot apărea supraîncărcări de date	Se face C2=0
0	>1	Bloc alocat mai multor fișiere	Se face C2=1
>1	0	Blocul este șters într-unul din fișiere; el va fi trecut în lista de noduri libere și, eventual, alocat altui fișier. Celelalte C1-1 referințe la blocul respectiv vor adresa o informație invalidă întrucât e posibil ca el să fie alocat unui fișier nou creat.	Se alocă C1-1 blocuri libere în care se copiază conținutul blocului incorect adresat.C1-1 dintre referințele către blocul respectiv vor fi înlocuite către copile acestuia.

Pentru verificarea consistenței la nivel de fișier se folosesc două contoare, CF1 și CF2. CF1 contorizează aparițiile fișierului în directorul de sistem și CF2 contorizează numărul de legături stocat în i-nodul asociat fișierului respectiv. În mod normal CF1=CF2. Situațiile anormale sunt:

CF1<CF2, fișierul va figura ca fiind adresat chiar dacă el a fost șters din toate directoarele unde figura și irosește un i-nod;

CF1>CF2, se alocă același i-nod pentru două fișiere diferite.

#### 7.4.5. Protecția fișierelor

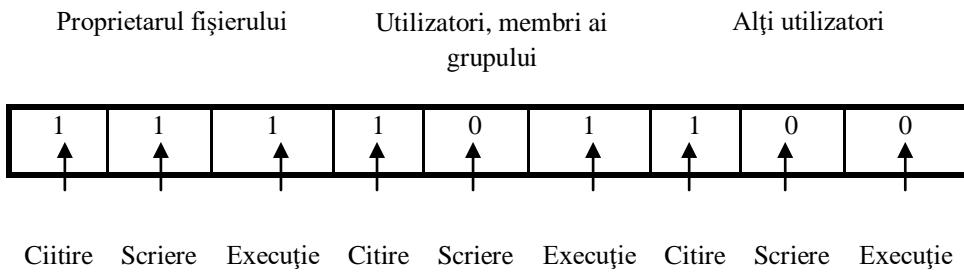
Este foarte important ca fiecare utilizator să-și definească drepturile de acces asupra propriilor fișiere, precizând cine le poate accesa și ce operație poate efectua asupra lor.

Sistemul UNIX are unul din cele mai eficiente sisteme de protecție a fișierelor. În acest sistem sunt trei tipuri de utilizatori și proprietari :

- proprietarul fișierului;
- utilizatori, membri ai grupului din care face parte posesorul fișierului;
- alți utilizatori.

și trei tipuri de operații: citire, scriere și execuție.

Implementarea se face cu un grup de 9 biți, 3 biți de fiecare operație, pentru fiecare tip de utilizator. Acest grup de 9 biți este conținut în unul dintre atributele care se găsesc în i-nodul asociat.

**Fig. 7.7. Implementarea sistemului de protecție în UNIX.**

În acest caz proprietarul are toate drepturile asupra sistemului, utilizatorii membri ai grupului au drept de citire și execuție iar ceilalți utilizatori au doar dreptul de citire.

Metoda utilizată în sistemul WINDOWS este ce a listelor de acces. Fiecare fișier îi este asociată o listă de control de acces (*acces control list = ACL*). Lista conține mai multe intrări de control al accesului (*acces control entries = ACE*). Fiecare ACE specifică drepturile de acces pentru un utilizator sau grup de utilizatori, conținând un identificator al acestora, o descriere a drepturilor și drepturile pe care le conferă asupra fișierului, precum și alte opțiuni cum ar fi cea legată de posibilitatea ca un fișier creat în cadrul unui director să moștenească drepturile de acces ale acestuia.

În momentul în care un utilizator încearcă să acceseze fișierul, se verifică automat în cadrul listei de control asociate (ACL) dacă este vorba de un acces permis sau nepermis.

#### 7.4.6. Organizarea fișierelor pe disc

##### 7.4.6.1. Organizarea în sistemul de operare UNIX

În UNIX există următoarea organizare a sistemului de fișiere:

Blocul de BOOT	Superblocul	Lista de i-noduri	Blocuri de date
-------------------	-------------	-------------------	-----------------

Blocul de boot conține proceduri și funcții pentru inițializarea sistemului.

Superblocul conține informații despre starea sistemului de fișiere: dimensiunea, numărul de fișiere ce pot fi create, adresa spațiului liber de pe disc, numărul de i-noduri, numărul de blocuri.

Lista de i-noduri este lista tuturor nodurilor index conținute în sistem, o listă lineară în care un nod este identificat printr-un indice. Dintre i-noduri, primul din listă este destinat gestionii blocurilor diferite iar al doilea are directorul de rădăcini.

Blocurile de date conțin atât date utilizator cât și date folosite de sistemul de fișiere

##### 7.4.6.2. Organizarea fișierelor în SO ce folosesc FAT

În SO care utilizează mecanisme de tip FAT, structura unei intrări de director este:

Nume fișier	Extensie	Atribute	Rezervat	Timp	Data	Numărul primului bloc

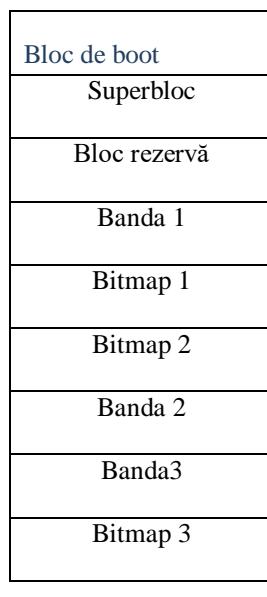
Principalul dezavantaj al mecanismelor FAT este absența mecanismului de crearea legăturilor la fișiere. Pentru realizarea unor legături ar fi necesar ca o aceeași intrare să figureze în două sau mai multe directoare, având drept consecință duplicarea unor date legate de poziționarea pe disc, de timpul și dimensiunea fișierului respectiv, ceea ce poate genera inconsistențe.

#### 7.4.6.3. Organizarea în HPFS (High Performance File System)

Această organizare utilizează pentru descrierea localizării pe disc a fișierelor structuri de date de tip arbori B care au principala proprietate de a avea performanțe ridicate pentru operațiile de căutare. În această organizare discul arată astfel:

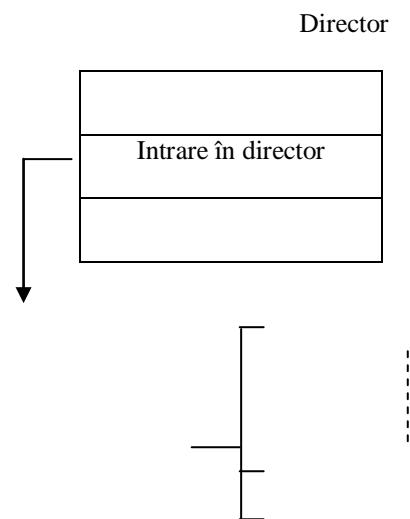
-Primele 18 sectoare conțin blocul de boot, superblocul, blocul de rezervă. Aceste blocuri conțin informații despre inițializarea sistemului de fișiere, gestiunea sistemului și refacerea sistemului după producerea de erori.

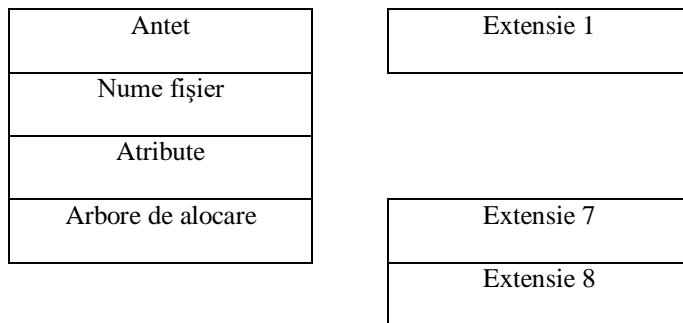
-Benzi de 8MB, fiecare bandă având asociată o hartă de biți de dimensiunea 2kB în care un bit corespunde unui bloc de 4KB din cadrul benzii; dacă bitul e zero bitul este liber, dacă bitul e 1 blocul este alocat. Hărțile de biți sunt situate în mod alternativ la sfârșitul și la începutul benzilor aşa încât să se poată face alocări contigue de până la 16MB. Se lasă spațiu între fișierele existente și cele noi create în ideea obținerii unor alocări de blocuri contigue.



**Fig. 7.8. Organizarea benzilor cu hărți de biți în sistemul HPFS.**

Pentru descrierea fișierelor se utilizează F-noduri.





**Fig. 7.9. Descrierea fișierelor cu F-noduri.**

F-nodurile sunt structuri de date care conțin atributele și informațiile legate de localizarea pe disc a fișierelor. Fiecare F-nod are cel mult 8 extensi. Extensiile sunt entități distincte, fiecare din ele putând adresa blocuri de maxim 16MB

#### 7.5.6.4. Organizarea în NTFS (New Technology File System)

Fiecare volum NTFS conține un MFT (Master File System) care este un fișier cu informații despre fișierele și directoarele de pe volumul respectiv. MFT este organizat ca o succesiune de înregistrări dintre care primele 16 sunt utilizate pentru descrierea MFT-ului însuși și pentru furnizarea informației necesare după situații de avarie. Următoarele înregistrări din MFT descriu fișierele și directoarele.

## 8. SISTEME DE OPERARE PENTRU CALCULATOARE PARALELE

### 8.1. NOTIUNI INTRODUCTIVE

Platformele hardware pentru programare paralelă sunt:

- tablouri de procesoare;
- multiprocesoare;
- multicalculatoare;
- rețele de calculatoare (stație de lucru în LAN).

În funcție de platformele hardware există următoarele tipuri de sisteme de operare:

- sisteme de operare în rețea*, pentru platforme hardware de tip multicalculatoare;
- sisteme de operare cu multiprocesoare*, pentru platforme hardware de tip multiprocesoare;
- sisteme de operare distribuite*, pentru platforme hardware de tip multicalculatoare.

Denumirea de *sisteme de operare distribuite* se găsește în literatura de specialitate, în unele cazuri, pentru toate calculatoarele paralele. În ultimul timp, însă, s-a impus termenul de *distribuit* numai pentru multicalculatoare, astfel încât termenul *sisteme de operare distribuite* înseamnă sisteme de operare numai pentru multicalculatoare.

O altă problemă esențială într-un calculator paralel este distribuția sarcinilor între sisteme de operare, programe, compilatoare; nici în ziua de astăzi nu lucrurile nu sunt prea clare în ceea ce privește rolul fiecărui.

**Limbajele de programare** în calculatoarele paralele pot fi la rândul lor clasificate în:

- limbaje de *programare paralelă*, pentru programarea prin variabile partajate, pe multiprocesoare;
- limbaje de *programare distribuită*, pentru programarea prin transfer de mesaje, folosită în multicalculatoare.

În prezent există un mare număr de limbaje de programare paralelă și distribuită. Aceste limbaje sunt de două categorii:

-limbaje paralele și distribuite dezvoltate din limbaje secvențiale uzuale ca: Fortran, Pascal, C; astfel de limbaje sunt limbajele Power Fortran, Power C, C\*.

-limbaje paralele și distribuite noi, scrise special pentru multiprocesoare și multicalculatoare; astfel de limbaje sunt limbajele Occam, Orca, Ada, Parlog, SR, Emerald.

O altă clasificare a limbajelor de programare paralele, legată de tipul compilatoarelor, este:

- limbaje de programare paralelă *explicite*;
- limbaje de programare paralelă *implicite*.

În limbajele de programare paralelă explicite, programatorul este acela căruia îi revine sarcina de a descrie algoritmul paralel în mod explicit, specificând activitățile paralele precum și modul de comunicare și sincronizare a proceselor sau threadurilor.

În limbajele de programare paralelă implicate, compilatorul este acela care detectează paralelismul algoritmului prezentat ca un program secvențial și generează codul corespunzător pentru execuția activităților paralele pe procesoarele componente ale calculatorului. Astfel de compilatoare se mai numesc *compilatoare cu paralelizare* căci au sarcina de a paraleliza programul. Dezvoltarea acestor compilatoare este limitată de dificultatea de a detecta și analiza dependențele în programele complexe. Paralelizarea automată a programelor este mai ușor de executat pentru paralelismul la nivel de instrucțiune, în care se distribuie instrucțiunile unei bucle, și este mai greu de făcut pentru programele cu o structură neregulată, care prezintă apeluri multiple de funcții, proceduri, ramificații sau bucle imperfect imbricate. Un exemplu de astfel de compilatoare este cel al multiprocesoarelor Silicon Graphics în care se rulează Power C și care includ un analizor de sursă ce realizează paralelizarea automată a programelor.

Din punct de vedere al sistemelor de operare interesează aspectele de cuplare ale hardului și softului.

Din punct de vedere al hardului, multiprocesoarele sunt *puternic cuplate* iar multicalculatoarele sunt *slab cuplate*.

Din punct de vedere al softului, un sistem slab cuplat permite calculatoarelor și utilizatorilor unui sistem de operare o oarecare independentă și, în același timp, o interacționare de grad limitat. Într-un sistem software slab cuplat, într-un grup de calculatoare fiecare are memorie proprie, hardware propriu și un sistem de operare propriu. Deci calculatoarele sunt aproape independente în funcționarea lor. Un defect în rețea de interconectare nu va afecta mult funcționarea calculatoarelor, deși unele funcționalități vor fi pierdute.

Un sistem soft puternic cuplat are în componență programe de aplicații care interacționează mult între ele dar și cu sistemul de operare.

Corespunzător celor două categorii de hardware (slab și puternic cuplat) și celor două categorii de software (slab și puternic cuplat), există patru categorii de sisteme de operare, dintre care trei au corespondent în implementările reale.

## 8.2. SISTEME DE OPERARE ÎN REȚEA

Sistemele de operare din această categorie au:

- hardware *slab cuplat*;
- software *slab cuplat*.

Fiecare utilizator are propriul său *workstation*, cu un sistem de operare propriu care execută comenzi locale. Interacțiunile care apar nu sunt puternice. Cel mai des apar următoarele interacțiuni:

-conectarea ca user într-o altă stație, ceea ce are ca efect transformarea stației proprii într-un *terminal la distanță* al stației la care s-a făcut conectarea;

-utilizarea unui sistem de fișiere global, accesat de toate stațiile din rețea, plasat într-una din stații, denumită *server de fișiere*.

Sistemele de operare din această categorie au sarcina de a administra stațiile individuale și serverele de fișiere și de a asigura comunicațiile dintre acestea. Nu este necesar ca pe toate stațiile să ruleze același sistem de operare. Când pe stații rulează sisteme de operații diferite, stațiile trebuie să accepte toate același format al mesajelor de comunicație.

În sistemele de operare de tip rețea nu există o coordonare în execuția proceselor din rețea, singura coordonare fiind data de faptul că accesul la fișierele nelocale trebuie să respecte protocoalele de comunicație ale sistemului.

### 8.3. SISTEME DE OPERARE CU MULTIPROCESOARE

Acstea sisteme de operare au:

- hardware *puternic cuplat*;
- software *puternic cuplat*.

În astfel de sisteme hardul este mai dificil de realizat decât sistemul de operare. Un sistem de operare pentru sisteme cu multiprocesoare nu diferă mult de față de un sistem de operare uniprocesor.

În ceea ce privește gestiunea proceselor, faptul că există mai multe procesoare hard nu schimbă structura sistemului de operare. Va exista o coadă de așteptare unică a proceselor pentru a fi rulate, organizată exact după aceleași principii ca la sisteme uniprocesor. La fel, gestiunea sistemului de intrare/ieșire și gestiunea fișierelor rămân practic neschimbate.

Câteva modificări sunt aduse gestiunii memoriei. Accesul la memoria partajată a sistemului trebuie făcută într-o secțiune critică, pentru prevenirea situației în care două procesoare ar putea să aleagă același proces pe care să-l planifice în execuție simultană. Totuși, de cele mai multe ori, accesul la memoria partajată trebuie să fie făcută de programator, adică de cel care proiectează aplicația. Sistemul de operare pune la dispoziție mijloacele standard (semafoare, mutexuri, monitoare) pentru implementarea realizării secțiunii critice, mijloace care sunt aceleași ca la sistemele uniprocesor. De aceea, sistemele de operare pentru multiprocesoare nu trebuie fundamental tratate altfel decât sistemele de operare pentru uniprocesoare.

#### 8.3.1. Programarea paralelă

În programarea paralelă, mai multe procese sau threaduri sunt executate concurrent, pe procesoare diferite, și comunică între ele prin intermediul variabilelor partajate memorate în memoria comună. Avantajele programării paralele sunt:

- o viteză de comunicare între procese relativ ridicată;
- o distribuție dinamică a datelor;
- o simplitate de partaționare.

##### 8.3.1.1. Memoria partajată între procese

Deoarece în programarea paralelă memoria partajată joacă un rol important, să prezintăm câteva caracteristici ale acesteia.

Am văzut, în capitolele anterioare, că fiecare proces are un spațiu virtual de procese alcătuit din:

- cod;
- date;
- stivă.

Un proces nu poate adresa decât în acest spațiu iar alte procese nu au acces la spațiul de memorie al procesului respectiv.

În memoria partajată, modalitatea prin care două sau mai multe procese pot accesa o zonă de memorie comună este crearea unui *segment de memorie partajată*. Un proces creează un segment de memorie partajată definindu-i dimensiunea și drepturile de acces. Apoi amplasează acest segment în propriul spațiu de adrese.

După creare, alte procese pot să-l atașeze și să-l amplaseze în spațiul lor de adrese. Esențial este ca segmentul de memorie să fie creat în memoria principală comună și nu într-o din memoriile locale.

### 8.3.1.2. Exemple de programare paralelă

#### Modelul PRAM

Modelul **PRAM** (Parallel Random Acces Machines) a fost dezvoltat de Fortan și Willie pentru modelarea unui calculator paralel cu cost suplimentar de sincronizare și acces la memorie nul. El conține procesoare care accesează o memorie partajată. Operațiile procesoarelor la memoria partajată sunt:

- citirea exclusivă (Exclusive Read ,ER) ; se permite ca un singur procesor să citească dintr-o locație de memorie, la un moment dat;
- scriere exclusivă ( Exclusive Write, EW); un singur procesor va scrie într-o locație de memorie, la un moment dat;
- citire concurentă (Concurrent Read, CR); mai multe procesoare pot să citească o locație de memorie la un moment dat;
- scriere concurentă (Concurrent Write, CW); mai multe procesoare pot să scrie într-o locație de memorie la un moment dat.

Combinând aceste operații, rezultă următoarele modele:

**EREW**, cu citire și scriere concurentă; este modelul cel mai restrictiv în care numai un singur procesor poate să scrie sau să citească o locație de memorie la un moment dat;

**CREW**, cu citire concurentă și scriere exclusivă; sunt permise accese concurente doar în citire, scrierea rămânând exclusivă;

**ERCW**, cu citire exclusivă și scriere concurentă; sunt permise accese concurente în scriere, citirea rămânând exclusivă;

**CRCW**, cu citire și scriere exclusivă; sunt permise accese concurente de scriere și citire la aceeași locație de memorie.

Scrierea concurentă (CW) se rezolvă în următoarele moduri:

**COMMON PRAM**, operațiile de scriere memorează aceiași valoare la locația accesată simultan.

**MINIMUM PRAM**, se memorează valoarea scrisă de procesorul cu indicele cel mai mic.

**ARBITRARY PRAM**, se memorează numai una din valori, aleasă arbitrar.

**PRIORITY PRAM**, se memorează o valoare obținută prin aplicarea unei funcții asociative ( cel mai adesea însumarea) tuturor valorilor cu care se accesează locația de memorie.

#### Algoritmi PRAM

Pentru algoritmii paraleli se folosesc un limbaj de nivel inalt (de exemplu **C**) la care se adaugă două instrucțiuni paralele

1) `forall<lista de procesare>do`

`<lista instrucțiuni>`

endfor

Se execută, în paralel, de către mai multe procesoare (specificate în lista de procesoare) unele operații (specificate în lista de instrucțiuni).

2) parbegin

<lista de instrucțiuni>

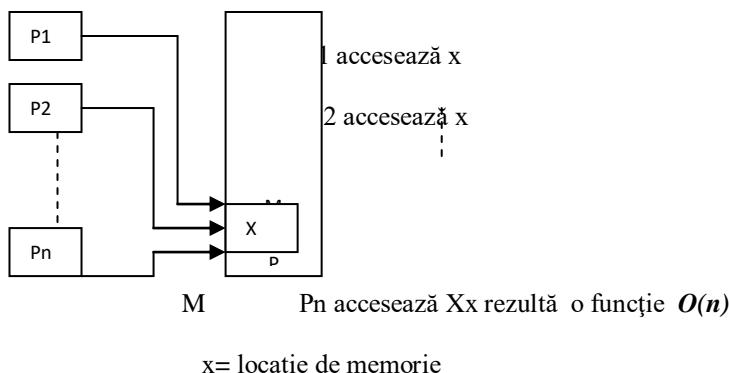
parend

Se execută, în paralel, mai multe instrucțiuni (specificate în lista de instrucțiuni).

### Program paralel de difuziune

Operația de difuziune (broadcast) constă din transmiterea unei date către toate procesele din sistem. Dacă există o dată într-o locație din memoria partajată, ea va trebui transmisă către celelalte n procesoare. Modelul ales este EREW.

Dacă procesoarele ar citi fiecare valoare din locație, pentru n procesoare, ar rezulta o funcție de tip  $O(n)$ , deci un timp liniar care este mare și nu exploatează posibilitatea de execuție concurrentă a mai multor procesoare.



**Fig. 8.1. Program paralel de difuziune (model EREW).**

Soluția de difuziune presupune următorii pași.

- Se alege un vector  $a[n]$ ,  $n$  fiind numărul de procesoare în memorie partajată.
- Date de difuzat este  $a[0]$  și în memoria locală a procesorului  $P_0$ .
- În primul pas, la iterată  $j=0$ , procesorul  $P_1$  citește data din  $a[0]$ , o înscrive în memoria locală și în vectorul partajat în poziția  $a[1]$ .
- În al doilea pas, la iterată  $j=1$ , procesoarele  $P_2$  și  $P_3$  citesc data din  $a[0]$  și  $a[1]$ , o înscrui în memoria locală și în locațiile  $a[2]$  și  $a[3]$ .
- În general, în iterată  $j$ , procesoarele  $i$ , cu condiția  $2^j \leq i < 2^{j+1}$ , citesc data de difuzat de la locațiile  $a[i-2^j]$ , o memorizează în memoria locală și o înscrui în vector în locațiile  $a[i]$ .

Iată un exemplu pentru  $n=8$  procesoare. Data de difuzat inițială este 100.

Initial

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]

100							
-----	--	--	--	--	--	--	--

100							
P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>	P <sub>5</sub>	P <sub>6</sub>	P <sub>7</sub>

Primul pas

(iterația j=0)

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]

100	100						
-----	-----	--	--	--	--	--	--

100	100						
P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>	P <sub>5</sub>	P <sub>6</sub>	P <sub>7</sub>

Al doilea pas

(iterația j=1)

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]

100	100	100	100				
-----	-----	-----	-----	--	--	--	--

100	100	100	100				
P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>	P <sub>5</sub>	P <sub>6</sub>	P <sub>7</sub>

Al treilea pas

(iterația j=2)

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]

100	100	100	100	100			
-----	-----	-----	-----	-----	--	--	--

100	100	100	100	100	100	100	100
P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>	P <sub>5</sub>	P <sub>6</sub>	P <sub>7</sub>

Deoarece la fiecare iterăție se dublează numărul de procesoare care au recepționat datea de difuzat, sunt necesare  $\log_2 n$  iterății. Într-o iterăție  $j$ , sunt active  $2^j$  procesoare, cele cu indice i având  $2^j \leq i < 2^{j+1}$ .

Timpul de execuție este estimat de  $O(\log n)$ , deci un timp sublinear.

Iată o implementare a difuziunii într-un limbaj formal PRAM, în EREW:

DIFUZIUNEA:

```
int n, a[n];
forall(0≤i<n)do
    for(j=0; j<log n; j++)
        if(2j≤i<2j+1)
            {citește valoarea din a[i-2j];
             memorează valoarea în memoria locală a lui Pi
             și în a[i];}
    endfor
```

### Program paralel de reducere

Fiind date  $n$  valori ca elemente ale unui vector și operația de adunare  $,$ , reducerea este operația:

$$a[1]+a[2]+a[3]+\dots+a[n]$$

Implementarea secvențială a reducerii este:

```
for(i=1;i<n;i++)
    a[0]+=a[i]
```

Se observă că rezultatul reducerii se găsește în prima componentă a vectorului  $a[0]$ . La fel ca și la difuziune, implementarea secvențială duce la un timp  $O(n)$ .

Algoritmul paralel de reducere este unul de tip EREW, utilizând  $m/2$  procesoare. Se consideră inițial datele într-un vector  $a[n]$ , pentru simplificare luându-se  $n$  ca multiplu de doi. Pașii sunt următorii:

- în primul pas, iterația  $j=0$ , toate procesoarele  $P_i (0 \leq i < n/2)$  sunt active și fiecare adună două valori  $a[2i]$  și  $a[2i+1]$ , rezultatul fiind  $a[2i]$ ;

- în al doilea pas, iterația  $j=1$ , sunt active procesoarele  $P_i$ , cu  $i=1$  și  $i=2$ , care adună  $a[2i]$  cu  $a[2i+2]$ , rezultatul fiind  $a[2i]$ ;

- în general, în iterația  $j$ , sunt active procesoarele cu indice  $i$ , multiplu de 2; fiecare actualizează locația  $a[2i]$ , deci execută  $a[2i]=a[2i]+a[2i+2]$ .

Se observă că numărul de iterații  $j$  este  $\log_2 n$ , deci avem un timp de execuție de tip  $O(\log n)$ , un algoritm sublinear.

Dăm mai jos un exemplu de implementare pentru  $n=8$ .

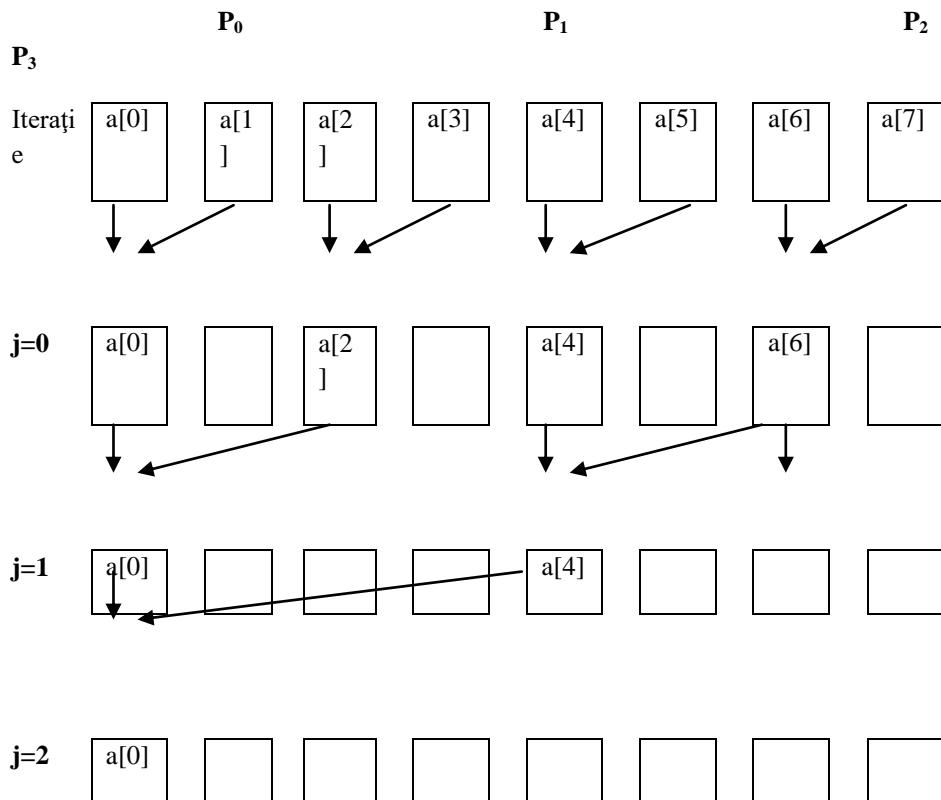


Fig. 8.2. Program paralel reducere (model EREW)

#### REDUCERE

```

int n, a[n];
forall(0≤i<n/2)do
    for(j=0;j<log n;j++)
        if(i modulo 2j+1==0)
            a[2i]=a[2i+2j];
endfor

```

#### 8.4. SISTEME DE OPERARE DISTRIBUITE

Aceste sisteme fac parte din categoria

- hardware slab cuplat (multicalculatoare)
- software puternic cuplat.

Din punct de vedere hardware, un astfel de sistem, tip multicalculator, este ușor de realizat. Un multicalculator este format din mai multe procesoare, fiecare procesor având propria unitate de control și o memorie locală. Procesoarele sunt legate între ele printr-o rețea de comutație foarte rapidă. O astfel de arhitectură hardware prezintă o serie de avantaje:

- adăugarea de noi procesoare nu conduce la degradarea performanțelor;

- un astfel de sistem multicalculator permite folosirea mai eficientă a procesoarelor; în general, un utilizator solicită egal în timp un calculator, existând perioade în care unitatea centrală nu este încărcată și perioade în care este foarte solicitată (la programe cu calcule foarte multe, de exemplu la un sistem liniar cu multe ecuații și necunoscute); într-un sistem distribuit încărcarea procesoarelor se face mult mai bine, mărinindu-se mult viteza de calcul;

- sistemul poate fi modificat dinamic, în sensul că se pot înlocui procesoare vechi cu unele noi și se pot adăuga unele noi, cu investiții mici, fără să fie nevoie de reinstalare de soft sau reconfigurare de resurse.

Din punct de vedere al sistemului de operare, însă, lucrurile se complică. Este foarte dificil de implementat un sistem de operare distribuit. Sunt de rezolvat multe probleme legate de comunicație, sincronizare, consistență informației etc.

Scopul acestor sisteme de operare distribuite este de a crea impresia utilizatorului că tot multicalculatorul este un singur sistem multitasking și nu o colecție de calculatoare distințe. De fapt, un multicalculator ar trebui să aibă o comportare de *uniprocesor virtual*. Ideea este ca un utilizator să nu fie preocupat de existența mai multor calculatoare în sistem.

Un sistem de operare distribuit trebuie să prezinte anumite caracteristici:

- să existe un singur mecanism de comunicare între procese, astfel încât fiecare proces să poată comunica cu altul, indiferent dacă acesta este local sau nelocal; mulțimea apelurilor sistem trebuie să fie aceeași și tratarea lor identică;

- sistemul de fișiere trebuie să fie același pentru toate stațiile componente, fără restricții de lungime a numelor fișierelor și cu aceleași mecanisme de protecție și securitate;

- pe fiecare procesor trebuie să ruleze același kernel.

Problema esențială care se pune acum în sistemele distribuite este împărțirea sarcinilor între sistemul de operare, programatorul de operații și compilator. La ora actuală nu există sarcini precise pentru fiecare dintre ele, lucrurile în acest domeniu fiind noi sau în curs de formare.

În ceea ce privește siguranța în funcționare a unui sistem distribuit, se poate spune că acesta este mai sigur decât un sistem de operare centralizat. Într-adevăr, căderea unui procesor dintr-un sistem centralizat duce la căderea întregului sistem, pe când într-un sistem distribuit procesoarele care dispar pot fi suplinite de celelalte procesoare. Punctul critic al siguranței de funcționare în sistemele distribuite este siguranța rețelei de interconectare. Se pot pierde mesaje și, în cazul supraîncărcării rețelei de interconectare, performanțele scad foarte mult.

#### **8.4.1. Structura unui SO distribuit**

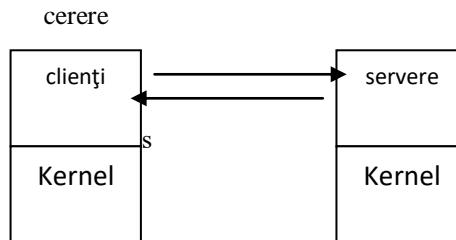
În sistemele de operare distribuite se folosesc transferuri de mesaje prin intermediul celor două primitive de transfer de mesaje send și receive, primitive care au fost studiate în capitolul „Comunicare între procese”. Un sistem de operare distribuit (SOD) trebuie să folosească numai aceste primitive. Revine ca sarcina principală găsirea unor modele de comunicare cu ajutorul cărora să se poată implementa un SOD. Principalele modele de comunicare utilizate în prezent sunt:

- comunicarea client server;
- apelul de procedură la distanță, RPC(Remote Procedure Call);
- comunicația de grup.

##### **8.4.1.1. Comunicarea client-server**

Este un model foarte des aplicat în multe domenii. În sistemele de operare distribuite există două tipuri de grupuri de procese cooperante:

- servere*, grup de procese ce oferă servicii unor utilizatori;
- clienți*, grup de procese care consumă serviciile oferite de servere.



**Fig. 8.3. Modelul de comunicare client/server.**

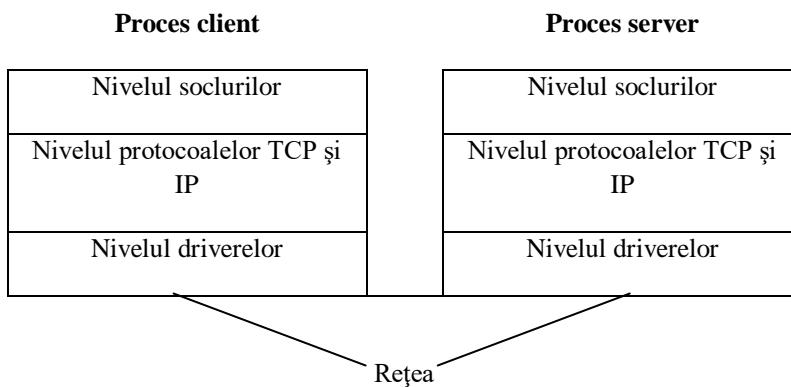
Atât procesele *server* cât și cele *client* execută același kernel al sistemului de operare, în timp ce procesele *server* și *client* sunt executate în spațiul utilizator. Există procese servere specializate în diferite servicii de : lucru cu fișiere, compilatoare, printare etc. Un proces care face o cerere către un server pentru a solicita un anumit serviciu devine un proces client. De remarcat că același proces sau grup de procese pot să fie, în diferite momente, atât server cât și client.

Avantajul principal al acestui tip de comunicație este dat de eficiență și simplitatea execuției. Comunicația are loc fără să se stabilească mai înainte o conexiune între client și server, clientul trimînd o cerere iar serverul răspunzând cu datele solicitate.

Principalul dezavantaj al acestui model este dat de dificultatea de programare, deoarece programatorul trebuie să apeleze explicit funcțiile de transfer de mesaje.

#### Comunicarea prin socluri (sokets) în rețelele UNIX ce funcționează pe sistemul client-server

Soclurile (Sockets) sunt un mecanism de comunicație între procese în rețelele UNIX introdus de 4.3 BSD ca o extensie sau generalizare a comunicației prin „pipes-uri”. Pipes-urile facilitează comunicațiile între procesele de pe același sistem în timp ce socketurile facilitează comunicarea între procese de pe sisteme diferite sau de pe același sistem.



**Fig. 8.4. Implementarea comunicării prin socluri.**

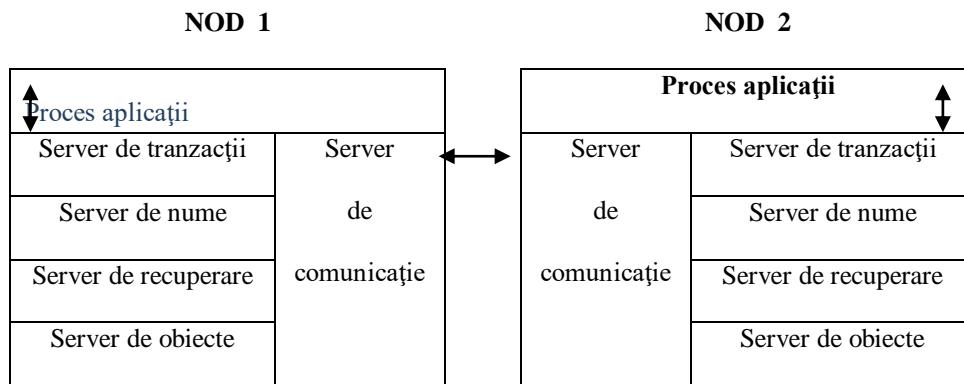
Acet subsistem conține trei părți:

- nivelul soclurilor, care furnizează interfață dintre apelurile sistem și nivelurile inferioare;
- nivelul protocoalelor, care conține modulele utilizate pentru comunicație (TCP și IP);
- nivelul driverelor, care conține dispozitivele pentru conectare în rețea.

Comunicarea între procese prin socluri utilizează modelul client server. Un proces server („listener”) ascultă pe un soclu (un capăt al liniei de comunicație între procese) iar un proces client comunică cu serverul la celălalt capăt al liniei de comunicație, pe un alt soclu care poate fi situat și pe alt nod. Nucleul sistemului de operare memorează date despre linia de comunicație deschisă între cele două procese.

#### **Exemplu de sistem distribuit bazat pe modelul client-server**

Sistemul din acest exemplu (fig. 8.5.) are un client, care este procesul aplicație, și câteva servere care oferă diferite servicii.



**Fig. 8.5. Exemplu de structură a unui sistem de operare distribuit.**

Acest sistem distribuit bazat pe modelul client-server are șase componente dintre care două sunt programabile de către utilizator.

1) Serverul de comunicatie este un singur proces în fiecare nod, rolul său fiind de a asigura comunicarea aplicațiilor de pe acel nod cu restul sistemului. El asigură independența aplicațiilor față de localizarea serverelor de obiecte. Pentru a determina localizarea unui obiect în rețea, serverul de comunicație interoghează serverul de nume local.

2) Serverul de nume are rolul de a memora și determina plasarea managerilor de obiecte din rețea. Există un singur server de nume în fiecare nod.

3) Serverul tranzacțiilor are rolul de a recepta și trata toate apelurile de *Inceputtranzacție*, *Sfârșittranzacție* și *Terminareanormalătranzacție*. Acest server coordonează protocolul pentru tranzacțiile distribuite. Există un singur server de acest fel în fiecare nod.

4) Serverul de obiecte conține module programabile de utilizator, pe baza unor funcții de bază livrate de biblioteca de funcții. Are rolul de a implementa operațiile asupra obiectelor partajate și de a rezolva concurența la nivelul obiectelor. Pot exista oricâte servere de obiecte în fiecare nod.

5) Serverele de recuperare asigură, recuperarea tranzacțiilor cu *Terminareanormală*, *Sabotate Voluntar* de procesul de aplicație sau involuntar la căderea unui nod. Există câte un server pentru fiecare obiect.

6) Procesele aplicații sunt scrise de utilizator.

#### 8.4.1.2. Apelul de proceduri la distanță RPC (Remote Procedure Call)

Ideea acestui mecanism de comunicație este ca un program să apeleze proceduri care își au locul în alte calculatoare.

Se încearcă ca apelul unei proceduri la distanță să semene foarte mult cu apelul unei proceduri locale, la fel ca în limbajele de nivel înalt studiate.

În momentul în care un proces aflat în procesorul 1 apelează o procedură din procesorul 2, procesul apelant este suspendat din procesorul 1 și execuția sa continuă în procesorul 2. Informația este transferată de la procesul apelant la procedura apelată prin intermediul parametrilor de apel iar procedura va returna procesului apelant rezultatul execuției, la fel ca în cazul unei proceduri locale. În tot acest mecanism, transferurile de mesaje nu sunt vizibile pentru programator. Într-un astfel de mecanism nu pot fi folosiți decât parametri de valoare, parametrii de referință neputând fi utilizati datorită acheselor diferite ale memoriei locale, aici neexistând o memorie comună.

RPC este o tehnică destul de des utilizată în sistemele de operare distribuite, chiar dacă mai există probleme atunci când un calculator se blochează sau când există modalități diferite de reprezentare a datelor pe diferite calculatoare.

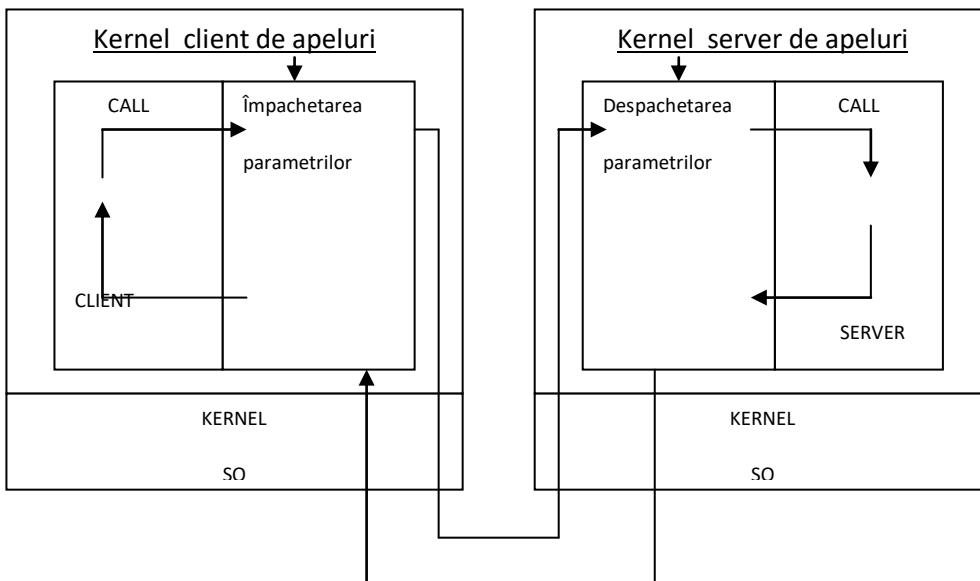


Fig. 8.6. Schema de principiu a funcționării RPC.

Pașii execuția la apelul unei proceduri la distanță pot fi prezentati astfel:

- Procedura client apelează local kernelul client de apeluri, transferându-i parametrii.
- Kernelul client de apeluri împachetează parametrii primiți, construiește un mesaj pe care îl predă kernelului SO client.
- Kernelul SO client transmite mesajul către kernelul SO server și de aici spre kernelul server de apeluri.
- Kernelul server de apeluri despachetează parametrii și apelează procedura server.

e) Serverul execută funcția și returnează rezultatul kernelului server de apeluri.

f) Kernelul server de apeluri împachetează rezultatul, construiește mesajul pe care îl predă kernelului SO server.

g) Kernelul SO server transmite mesajul kernelului client de apeluri.

h) Kernelul client de apeluri despachetează rezultatul și-l returnează procedurii client apelante.

In comunicarea RPC, aşa cum am prezentat-o mai sus, se leagă un singur client de un singur server. Această comunicare este de tip sincron.

În ultima vreme se utilizează protocoale de tip RPC asincron în care se poate ca un client să comunice cu mai multe servere simultan.

Mecanismele RPC au fost standardizate de OSF (Open Software Foundation). Există două mecanisme RPC mai cunoscute:

NCS, dezvoltat de firma Hewlett Packard;

ONC(Open Network Computing) dezvoltat de firma Scan.

Consorțiul DEC (Distributed Environment Corporation) a lansat de asemenea un RPC pe platforma Microsoft.

RMI (Remote Method Invocation) este un apel la distanță de tip RPC dar folosind obiectele în sistemele Java.

OSF DCE este un pachet de specificații (Application Environment Specification, AES) ale serviciilor utilizabile de către o aplicație client-server într-un mediu distribuit și eterogen. Odată cu specificațiile DCE, OSF a realizat și un set de produse program care implementează aceste servicii, facilitează administrarea lor și simplifică elaborarea, testarea și instalarea aplicațiilor care le utilizează. Fiecare producător are libertatea de a utiliza și adapta produse OSF sau de a ține la propria sa implementare.

#### **8.4.1.3. Comunicația de grup**

Principalul dezavantaj al mecanismului de comunicare RPC este că implică o comunicare unu la unu, adică un singur client cu un singur server. Pentru ca mai mulți clienți să comunice cu un server, s-a introdus un nou tip de mecanism de comunicare numit comunicație de grup.

Un grup de comunicatie înseamnă o mulțime de procese care se execută într-un sistem distribuit, care au proprietatea ca, atunci când se transmite un mesaj grupului, toate procesele din grup le recepționează. Este un tip de comunicare unu la mulți.

Grupurile pot fi clasificate după mai multe criterii .

a) În funcție de procesele care au dreptul de a transmite un mesaj în interiorul grupului, pot fi:

-*grupuri închise*, în care numai procesele membre ale grupului pot transmite un mesaj;

-*grupuri deschise*, în care orice proces poate transmite un mesaj membrilor grupului.

b) În funcție de drepturile de coordonare a operațiilor în grup, sunt:

-*grupuri omogene*, în care toate procesele au drepturi egale, nici unul dintre ele nu are drepturi suplimentare și toate deciziile se iau în comun;

-grupuri ierarhice, în care unul sau mai multe procese au rolul de coordonator, iar celelalte sunt procese executante.

c) În funcție de admiterea de noi membri în grup, există:

-*grupuri statice*, care își păstrează neschimbate dimensiunea și componenții de la crearea și până la dispariție; după crearea unui grup static nici un proces membru nu poate să părăsească grupul;

-*grupuri dinamice*, la care se pot atașa procese membre noi sau altele pot părăsi grupul.

d) În funcție de modul de gestionare, pot exista:

-*grupuri centralizate*, în care un proces, numit server, execută toate operațiile referitoare la crearea grupurilor, atașarea de noi procese, părăsirea grupului de unele procese etc.; serverul de grup menține o bază de date a grupurilor, cu membrii fiecărui grup; dezavantajul este că dacă serverul se distrugе, toate grupurile își încetează activitatele;

-*grupuri distribuite*, în care fiecare proces este răspunzător pentru atașarea sau părăsirea unui grup; când un proces se distrugе, el nu mai poate anunța grupul de părăsirea sa și trebuie ca celelalte procese să descopere acest lucru.

Pentru implementarea comunicației între grupuri, sistemele de operare pun la dispoziție primitive de gestionare a grupurilor.

O posibilitate mult mai des folosită este aceea de a utiliza biblioteci de comunicație care suportă comunicația de grup. Două dintre cele mai cunoscute astfel de biblioteci sunt:

-PVM(Parallel Virtual Machine);

-MPI(Message Passing Interface).

### **Biblioteca PVM**

PVM (Parallel Virtual Machine) este un pachet de programe dezvoltat de Oak Ridge National Laboratory, Universitatea Statului Tennessee și de Universitatea Emory. Proiectul PMV a fost conceput de Vaidy Sunderam și Al. Geert de la Oak Ridge National Laboratory.

PVM asigură un mediu de lucru unitar în care programele paralele pot fi dezvoltate eficient utilizând un mediu hardware deja existent. Se asigură o transparență în rutarea mesajelor prin rețea, în conversia datelor și în planificarea taskurilor.

În PVM, utilizatorul va scrie aplicațiile ca o colecție de taskuri care cooperează. Aceste taskuri vor accesa resursele PVM cu ajutorul unor biblioteci de rutine de interfață. Rutinele din cadrul bibliotecilor asigură inițierea și terminarea taskurilor, comunicarea și sincronizarea lor. În orice punct al execuției unei aplicații concurente, orice task în execuție poate iniția sau termina alte taskuri, poate adăuga sau elimina calculatoare din mașina virtuală. Utilizatorii pot scrie programe în Fortran sau C, folosind rutine din PVM. Modelul de programe utilizat este cel cu transfer de mesaje. Sunt incluse facilități de asigurare a toleranței la defecte.

### **Descrierea sistemului PVM**

Sistemul PVM are două componente:

-demonul *pvmd*;

-biblioteca de *rutine PVM*.

Demonul pvm trebuie să existe pe toate mașinile care alcătuiesc mașina virtuală. Acest demon a fost proiectat astfel încât orice utilizator să-l poată instala pe orice mașină dacă dispune de un login valid. Când un utilizator dorește să ruleze o aplicație PVM, va trebui să creeze mașina și apoi să o pornească. O aplicație PVM poate fi pornită de pe orice calculator. Mai mulți utilizatori pot configura mașini virtuale care se pot suprapune și fiecare utilizator poate executa câteva aplicații PVM simultan.

Biblioteca de rutine PVM conține un set complet de primitive care sunt necesare pentru cooperare între tastaturi. Există următoarele tipuri de rutine:

- rutine de trimitere și recepționare a mesajelor;
- rutine de inițiere a tastaturilor;
- rutine de coordonare a tastaturilor;
- rutine de coordonare a mașinii virtuale.

Modelul de calcul utilizat se bazează pe faptul că o aplicație este alcătuită din mai multe taskuri, fiecare task fiind responsabil pentru calculul unei părți a problemei.

O aplicație poate accesa resursele de calcul în trei moduri diferite:

- modul *transparent*, în care taskurile sunt plasate în sistem prin mașina cea mai potrivită;
- modul *dependent de arhitectură*, în care utilizatorul poate indica o arhitectură specifică pe care un task poate fi executat;
- modul *cu specificare a mașinii*, în care utilizatorul poate indica o anume mașină pe care să se execute un task.

Toate taskurile sunt identificate cu un întreg numit *task identifier* (TID), echivalent PID-ului din sistemele de operare. Aceste TID-uri trebuie să fie unice în cadrul mașinii virtuale și sunt asigurate de demonul pvm local. PVM conține rutine care returnează valoarea TID astfel încât aplicațiile pot identifica taskurile din sistem. Pentru a programa o aplicație, un programator va scrie unul sau mai multe programe secvențiale în C, C++, Fortran++, cu apeluri la rutinile din bibliotecile PVM.

Pentru a executa o aplicație, un utilizator inițiază o copie a unui task, numit *task master* iar acesta va iniția taskuri PVM care pot fi rulate pe alte mașini sau pe aceeași mașină cu task masterul. Acesta este cazul cel mai des întâlnit, dar există situații în care mai multe taskuri sunt inițiate de utilizator și ele pot iniția la rândul lor alte taskuri.

### Consola PVM

Consola PVM este un task de sine stătător care permite utilizatorului să pornească, să interogheze și să modifice mașina virtuală. Consola poate fi pornită și oprită de mai multe ori pe orice gazdă din PVM, fără a afecta rularea PVM sau a altor aplicații. La pornire, consola PVM determină dacă aceasta rulează ; dacă nu, se execută pvm pe această gazdă. Prompterul consolei este `pvm>`.

Comenzile ce se pot aplica pe acest prompter sunt:

- |                                |                                           |
|--------------------------------|-------------------------------------------|
| <code>add</code>               | , adaugă gazde la mașina virtuală;        |
| <code>conf</code>              | , afișează configurația mașinii virtuale; |
| (numele gazdei, pvm tid, tipul |                                           |

arhitecturii și viteza relativă);

delete , elimină gazde din mașina virtuală;

halt , termină toate procesele PVM, inclusiv consola și oprește mașina virtuală;

id , afișează identificatorul de task al consolei;

jobs , afișează lista taskurilor în execuție;

kill , termină un task PVM;

mstat , afișează starea gazdelor specificate;

pstat , afișează starea unui task specificat;

quit , părăsește consola lăsând demonii și taskurile în execuție;

reset , termină toate procesele PVM exceptând consola;

spawn , pornește o aplicație PVM.

PVM permite utilizarea mai multor console. Este posibil să se ruleze o consolă pe orice gazdă din mașina virtuală și chiar mai multe console în cadrul aceleiași mașini.

### Implementarea PVM

Pentru implementarea PVM s-a ținut cont de trei obiective:

- mașina virtuală să cuprindă sute de gazde și mii de taskuri;
- sistemul să fie portabil pe orice mașini UNIX;
- sistemul să permită construirea aplicațiilor tolerante aplicațiilor tolerante la defecte.

S-a presupus că sunt disponibile socluri pentru comunicație între procese și că fiecare gazdă din mașina virtuală se poate conecta direct la celealte gazde, utilizând protocoale IP(TCP UDP). Un pachet trimis de un *pvm* ajunge într-un singur pas la alt *pvm*. Fiecare task al mașinii virtuale este marcat printr-un identificator de task (TID) unic. Demonul *pvm* asigură un punct de contact cu exteriorul pentru fiecare gazdă. *Pvm* este de fapt un router care asigură controlul proceselor și detecția erorilor. Primul *pvm*, pornit de utilizator, este denumit *pvm master*, ceilalți, creați de master, sunt denumiți *slave*. În timpul operării normale, toți *pvm* sunt considerați egali.

Toleranța la defecte este asigurată în felul următor:

- dacă masterul a pierdut contactul cu un slave, îl va marca și îl va elimina din mașina virtuală;
- dacă un slave pierde contactul cu masterul, atunci el se elimină singur din mașină.

Structurile de date importante pentru *pvm* sunt tabele de gazde care descriu configurația mașinii virtuale și taskurile care rulează în cadrul acesteia.

Există următoarele biblioteci:

-*biblioteca PVM (lib pvm);*

-*comunicațiile PVM.*

Biblioteca PVM conține o colecție de funcții care asigură interfațarea taskurilor cu *pvm* și cu alte taskuri, funcții pentru împachetarea și despachetarea mesajelor și unele funcții PVM de sistem. Această bibliotecă este scrisă în C și este dependenă de sistemul de operare și de mașină.

Comunicațiile PVM se bazează pe protocolele de Internet IP (TCP și UDP) pentru a se asigura de portabilitatea sistemului.

Protocolele IP facilitează rutarea prin mașini tip poartă intermediară. Rolul său este de a permite transmiterea datelor la mașini ce nu sunt conectate direct la același rețea fizică. Unitatea de transport se numește *datagramă IP*. La acest nivel se folosesc adrese IP care sunt formate din două părți:

- prima parte identifică o rețea și este folosită pentru rutarea datagramei;
- a doua parte identifică o conexiune la un anumit dat în interiorul rețelei respective.

Serviciul de livrare asigurat de IP este de tipul *fără conexiune*, fiecare datagramă fiind rutată prin rețea independent de celelalte datagrame.

UNIX folosește două protocole de transport:

-UDP (User Datagram Protocol), un protocol fără conexiune;

-TCP (Transmission Control Protocol), orientat pe conexiune; caracteristicile TCP-ului constau în transmiterea adresei destinației o singură dată la stabilirea conexiunii, garantarea odinei datelor transmise și bidirectionalitatea.

În sistemele PVM sunt posibile trei tipuri de comunicații:

- între demonii *pvm*;
- între *pvm* și taskurile asociate;
- între taskuri.

a) Comunicația *pvm*-*pvm*

Demonii *pvm* comunică între ei prin socluri UDP. Pachetele vor avea nevoie de mecanisme de confirmare și directare UDP. Se impun, de asemenea, limitări în ceea ce privește lungimea pachetelor, ajungându-se la fragmentarea mesajelor lungi. Nu s-a utilizat TCP din trei motive:

- o mașină virtuală compusă din n mașini necesită  $n(n-1)/2$  conexiuni, un număr care ar fi greu de stabilit;

-protocolul TCP nu poate sesiza căderea unei gazde *pvm*;

-protocolul TCP limitează numărul fișierelor deschise.

b) Comunicația *pvm*-task

Un task comunică cu un pvmid căruia îi este asociat prin intermediul conexiunilor. Taskul și pvmid mențin o structură FIFO de pachete, comutând între citire și scriere pe conexiuni TCP. Un dezavantaj al acestui tip de conectare este numărul crescut de apeluri sistem necesare pentru a transfera un pachet între un task și un pvmid.

#### c) Comunicație task-task

Comunicațiile între taskurile locale se realizează ca și comunicațiile pvmid-task. În mod normal, un pvmid nu comunică cu taskurile de pe alte gazde. Astfel, comunicațiile ce trebuie realizate între taskuri de pe gazde diferite vor folosi ca suport comunicațiile pvmid.

#### Limitări ale resurselor

Limitările resurselor impuse de sistemul de operare și de hardul disponibil se vor reflecta în aplicațiile PVM. Câteva din limitări sunt afectate dinamic datorită competiției dintre utilizatorii de pe aceeași gazdă sau din rețea. Numărul de taskuri pe care un pvmid le poate deservi este limitat de doi factori:

- numărul permis unui utilizator de către sistemul de operare;
- numărul de descriptori de fișiere disponibili pentru pvmid.

Mărimea maximă a mesajelor PVM este limitată de mărimea memoriei disponibile pentru un task.

Aceste probleme vor fi evitate prin revizuirea codului aplicației, de exemplu prin utilizarea mesajelor cât mai scurte, eliminarea strangulațiilor și procesarea mesajelor în ordinea în care au fost generate.

#### Programarea în PVM

Lansarea mașinii virtuale PVM se face prin lansarea în execuție a procesului server master, folosind funcția

pvm-start-pvmid .

Apoi se lansează funcția

pvm-setopt

în care se setează diferitele opțiuni de comunicație.

Pe procesorul server master trebuie să fie un fișier de descriere a configurației mașinii și anume fișierul h.stfile .

Acesta conține numele stațiilor pe care se configurează mașina virtuală, căile unde se află fișierele executabile, căile unde se află procesele server (demonii) de pe fiecare stație, codul utilizatorului și parola.

Configurația mașinii virtuale este dinamică; stații noi în rețea pot fi adăugate prin apelul funcției pvm-addh.sts

Alte stații pot fi excluse prin funcția  
pvm-delh.sts

Funcționarea mașinii se oprește prin  
pvm-halt

#### Controlul proceselor.

Pentru atașarea unui proces la mașina virtuală PVM se apelează funcția  
pvm-mytid

Prin această funcție se înrolează procesul și se returnează identificatorul de task TID.

Crearea dinamică de procese noi în mașina virtuală se face prin apelul funcției

pvm-spawn

La apelul funcției de mai sus se specifică adresa stației unde vor fi create procesele noi, chiar PVM-ul putând selecta stațiile unde vor fi create și lansate procesele noi. De exemplu:

numt=pvm-spawn(„my-task”,NULL,pvmtaskdefault,0,n-task,tids);

Se solicită crearea a n-task procese care să execute programul my-task pe stațiile din PVM. Numărul real de procese create este returnat de funcția numt. Identifierul fiecărui task creat este depus într-un element al vectorului tids.

Un proces membru al mașinii PVM poate părăsi configurația prin apelul

pvm-exit

sau poate fi terminat de către un alt proces care apelează funcția

pvm-kill(tid)

Interfața PVM suportă două modele de programare diferite și pentru fiecare din acestea se stabilesc condițiile de mediu de execuție. Aceste modele sunt SPMD(Single Program Multiple Data) și MPMD (Multiple Program Multiple Data).

În modelul SPMD, n instanțe ale aceluiași program sunt lansate cu n taskuri ale unei aplicații paralel, folosind comanda spwan de la consola PVM, sau, normal, în cele n stații simultan. Nici un alt task nu este creat dinamic de către taskurile aflate în execuție, adică nu se apelează funcția pvm-spwan. În acest model inițializarea mediului de programare constă în specificarea stațiilor pe care se execută cele n taskuri.

În modelul MPMD, unul sau mai multe taskuri distincte sunt lansate în diferitele stații și acestea creează dinamic alte taskuri.

#### Comunicațiile în mașina virtuală PVM

În mașina virtuală PVM există două tipuri de mesaje între procesele componente :

- comunicație punct la punct, adică transferul de mesaje între două procese;

-comunicație colectivă, adică difuziunea sau acumularea de mesaje într-un grup de procese.

Comunicația punct la punct.

Pentru transmiterea unui mesaj de către un proces A la un mesaj B, procesul transmițător A inițializează mai întâi bufferul de transmisie prin apelul funcției de inițializare:

int pvm-init send(int encode);

Argumentul (encode) stabilește tipul de codare al datelor în buffer. Valoarea returnată este un identificator al bufferului curent de transmisie în care se depun datele împachetate, folosind funcția

pvm-pack

După ce datele au fost depuse în bufferul de transmisie, mesajul este transmis prin apelul funcției

int pvm-send(int tid,int msgtag);

Funcția de transmisie este blocantă. Ea returnează controlul procesului apelant numai după ce a terminat de transmis mesajul. Argumentul tid este identifierul procesului receptor iar flagul msgtag este folosit pentru a comunica receptorului o informație de tip mesaj pe baza căreia receptorul ia decizia acceptării mesajului și a tipului de prelucrări pe care trebuie să le facă asupra datelor din mesaj. Valoarea returnată de funcția pvm-send este

0, transmisie corectă

\_\_\_\_\_ -1, în cazul apariției unei erori.

Pentru recepția unui mesaj, procesul receptor apelează funcția

```
int pvmrecv(int tid,int msgtag);
```

Această funcție este blocantă. Ea returnează controlul procesului apelant numai după terminarea execuției, deci după recepția mesajului. Argumentul tid specifică identificatorul procesului de la care se așteaptă mesajul iar argumentul msgtag specifică ce tip de mesaj este așteptat. Recepția se efectuează numai pentru mesajele care corespund celor două argumente sau pentru orice transmițător, dacă tid=-1, și orice tip de mesaj, dacă msgtag=-1.

Valoarea returnată este identificatorul bufferului de recepție în care au fost depuse datele receptionate. Din acest buffer datele sunt extrase folosind funcția de despachetare a datelor

`pvm-unpack`

În biblioteca PVM există și o funcție de recepție neblocantă

`pvm-nrecv`

Aceasta returnează controlul procesului apelant imediat, fără ca acesta să aștepte dacă mesajul a fost receptionat.

#### Comunicația colectivă în PVM

În sistemul PVM se pot crea grupuri dinamice de procese care execută sarcini corelate, comunică și se sincronizează intre ele. Un proces se poate ataşa unui grup, definit printr-un nume unic în mașina virtuală PVM, prin apelul funcției

```
int pvmjoingrp(cher*group-name);
```

Dacă grupul cu numele group-name nu există, el este creat. Valoarea returnată este un identificator al instanței procesului în grupul respectiv, care se adaugă identificatorului procesului. Acest identificator este 0 pentru procesul care creează grupul și are cea mai mică valoare disponibilă în grup pentru fiecare proces care apelează funcția pvmjoingroup.

Un proces poate aparține simultan unuia sau mai multor grupuri din mașina virtuală. El poate părăsi un grup prin apelul funcției

```
int pvmleavegroup(cher*group.name);
```

Dacă un proces părăsește un grup, fără ca un altul să-l înlocuiască, pot apărea goluri în identificatorii de instanță de grup ai proceselor rămase.

Un proces poate obține diferite informații de grup. Funcțiile pvmgetinst, pvmgettid, pvmgsizes returnează identificatorul procesului în grup.

Comunicația colectivă în mașina virtuală PVM se desfășoară între procesele membre ale unui grup. Funcțiile de comunicație colectivă sunt:

#### -Funcții de difuziune

```
int pvmicast(cher*group-name, int msgtag);
```

Se difuzează asincron mesajul aflat în bufferul de transmisie curent al procesului apelant cu flagul de identificare msgtag, către toate procesele membre ale grupului cu nume group-name. Procesul apelant poate fi sau nu membru al grupului. Dacă este membru, nu se mai transmite msg lui însuși.

#### -Funcția de distribuire

```
int pvm-scatter(void*my array,void*s-array,
                int dim, int type,int msgtag,
                cher*group-name,int root);
```

Un vector de date de tipul type, cu multe s-array și de dimensiune dim, aflat în spațiul de adresă al procesului rădăcină, este distribuit uniform tuturor proceselor din grupul cu numele group-name. Fiecare proces din grup trebuie să apeleze funcția pvm-scatter și fiecare recepționează o parte a datelor din vectorul s-array din procesul rădăcină (cu identificatorul root), în vectorul local s-array.

#### Funcția de colectare

```
int pvm-gather(void*g-array,void*myarray,
               int dim,int type,cher*group-name,
               int root);
```

Se colectează toate mesajele cu flagul de identificare msgtag de la toate procesele membre ale grupului cu nume group-name în procesul rădăcină cu identificatorul root. Colectarea are loc în vectorul de date g-array, a datelor aflate în fiecare proces în vectorii cu numele group-name în procesul rădăcină (definit de utilizator) cu identificatorul root.

#### Funcția de reducere

```
int pvm reduce(int operation,void*myrols,int dim,
               int type,int msgtag,cher*groupe-name,
               int root);
```

Se efectuează operația de reducere paralelă între toate procesele membre ale unui grup. Argumentul operation definește operația de reducere. Fiecare proces efectuează mai întâi operația de reducere a datelor din vectorul local de date, de tipul type, cu numele myrols, de dimensiune dim. Valoarea rezultată este transferată procesului rădăcină root în care se va afla valoarea de reducere finală.

#### Funcția de sincronizare într procesele membre

```
int pvm-barrier(cher*group name, int ntasks);
```

La apelul acestei funcții procesul este blocat până când un număr ntasks procese din grupul cu numele group-name au apelat funcția pvm-barrier.

### **Exemple de programare în PVM**

#### Program de reducere paralelă

Am văzut în programarea paralelă (pe multiprocesoare) cum se realizează operația de reducere când avem la dispoziție o memorie comună. Să vedem acum cum se face această operație prin transfer de mesaje în pvm.

```
#include"pvm3.h"
```

```
#define NTASKS 4

int main()
{int mytid,tids(NTASKS-1),groupid,sum,info;
/*se crează grupul de comunicație*/
mytid=pvm-mytid();
groupid=pvm-joingroup(„summex”);
sum=groupid;
/*Primul proces crează celelalte NTASKS-1 procese*/
if(groupid==0)
{info=pvm-spawn(„summex”,NULL,pvmtasksdefault,
*&,NTASKS-1,tids);
printf(„groupid=%d spawned%d tasks\n”,
groupid,info();}
/*se introduce o barieră până ce NTASKS procese s-au alăturat grupului*/
pvm-freeze group(„summex”,NTASKS);
/*Apelul funcției de comunicație colectivă pentru calculul sumei unui grup*/
pvm-reduce(pvm sum,&sum,1,
pvm-INT,1,„summex”,0);
/*Procesul 0 tipărește rezultatul operației de reducere*/
if(groupid==0) print(„sum=%d\n”,sum”);
/*Sincronizare pentru ca toate procesele să execute operația înainte de părăsirea grupului și terminarea
programului*/
pvm-barrier(„summex”,NTASKS)
pvm-evgroup(„summex”);
pvm –exit();
```

Reducerea paralelă se execută pe un număr de procese (taskuri) definit prin constanta NTASKS care se alerge în funcție de numărul de stații din rețea. Fiecare proces începe execuția prin obținerea propriului identificator de task PVM (mytid). Apoi procesul se atașează grupului cu numele „summex” și obține la atașare identificatorul de instanță de grup (groupid). Procesul master are groupid=0 și creează NTASKS-1 procese. Deoarece operația de reducere paralelă nu poate începe până când grupul nu este complet creat, se introduce un punct de sincronizare prin funcția pvm-freeze group. Această funcție oprește creșterea numărului de procese din grupul cu numele group-name la valoarea size, transformând grupul într-un grup static. Ea are și rolul de sincronizarea și trebuie să fie apelată înainte de execuția unor operații colective într-un grup., deoarece o astfel

de operație, odată începută, poate fi perturbată de apariția unui nou membru în grup. În fiecare proces se execută reducerea intr-un vector cu lungimea 1. În variabila sum fiecare proces memorează identificatorul său (mytid). După execuția reducerii paralele, suma tuturor identificatorilor taskurilor din grup este afișată la consola de către procesul master. Pentru ca încă un proces să nu poată părăsi grupul înainte ca toate procesele grupului să fie terminat operațiile, se introduce un nou punct de sincronizare, prin apelul funcției pvm-barrier, după care programul poate fi terminat prin desființarea grupului. Toate procesele sunt detașate din grup prin funcția pvm-lv group și părăsesc mașina virtuală prin pvm-exit.

### Biblioteca MPI (Message Passing Interface)

MPI este o bibliotecă standard pentru construirea programelor paralele, portabile în aplicații C și Fortran++, care poate fi utilizată în situații în care programul poate fi parționat static într-un număr fix de procese.

Cea mai mare diferență între biblioteca MPI și cea PVM este faptul că grupurile de comunicație MPI sunt grupuri statice. Dimensiunea grupului este statică și este stabilită la cererea grupului. În acest fel proiectarea și implementarea algoritmilor paraleli se face mai simplu și cu un cost redus.

Funcțiile din MPI sunt asemănătoare cu cele din PVM. Deși MPI are peste 190 de funcții, cele mai des folosite sunt:

```

MPI-Init() /*initializează biblioteca MPI*/
MPI-Finalize /*închide biblioteca MPI*/
MPI-Comm-SIZE /*determină numărul proceselor în comunicator*/
MPI-Comm-rank() /*determină rangul procesului în cadrul grupului*/
MPI-send() /*trimit un mesaj*/
MPI-recv() /*primește un mesaj*/

```

Iată mai jos programul de reducere, pe care l-am prezentat în PVM, implementat în MPI.

```

#include<mpi.h>
#include<sys/types.h>
#include<stdio.h>
int main()
{
    int mytid,ntasks,sum;
    MPI_Init;
    /*Crearea contextului de comunicație implicit și obținerea rangului procesului în acest context*/
    MPI_Comm_rank(MPI_COMM_WORLD,&mytid=;
    /*Aflare dimensiunii grupului de procese*/
    MPI_Comm_SIZE(MPI_COMM_WORLD,&ntasks);

```

```

if(mytid==0)

printf(mytid=%d,ntasks=%d\n",mytid,ntasks);

/*Calcului local al sumei parțiale;suma parțială este egală cu identificatorul procesului*/

sum=mytid;

/*Sincronizarea cu toate procesele să fie lansată*/

MPI-barrier(MPI-COMM-WORLD);

MPI-reduce(MPI-SUM,&sum,1,MPI-INT,1,MPI-
WORLD,0);

/*Rezultatul reducerii se află în procesorul =*/

if(mytid==0)printf(,sum=%d\n",sum);

MPI-Finalize();

exit(); }

```

Definirea mediului de programare paralelă în sistemul MPI se face prin execuția unui program de inițializare care stabilește procesoarele ce vor rula în MPI. În MPI toate procesele unei aplicații se creează la inițializarea acestora. În cursul execuției unei aplicații nu pot fi create procese dinamice și nu poate fi modificată configurația hardware utilizată.

#### Comunicații, grupuri și contexte în MPI

*Comunicatorul* specifică un grup de procese care vor coordona operații de comunicare, fără să afecteze sau să fie afectat de operațiile din alte grupuri de comunicare.

*Grupul* reprezintă o colecție de procese. Fiecare proces are un rang în grup, rangul luând valori de la 0 la n-1. Un proces poate aparține mai multor grupuri, caz în care rangul dintr-un grup poate fi total diferit de rangul în alt grup.

*Contextul* reprezintă un mecanism intern prin care comunicatorul garantează grupului un spațiu sigur de comunicare.

La pornirea unui program, sunt definiți doi comunicatori implicați:

- MPI-COMM-WORLD , care are ca grup de procese toate procesele din job;
- MPI-COMM-SELF, care se creează pentru fiecare proces, fiecare având rangul 0 în propriul comunicator.

Comunicatorii sunt de două feluri:

- intracomunicatori, care coordonează operații în interiorul unui grup;
- extracomunicatori, care coordonează operații între două grupuri de procese.

*Lansarea* în execuție se poate face în două moduri:

CRE (Cluster Tools Runtime Environment)

LSF (Load Sharing Facility)

În mediul CRE există patru comenzi care realizează funcții de bază.

```
mprun /*execută programe MPI*/
mpkill /*termină programele*/
mpps /*afișează informații despre
      programele lansate*/
mpinfo /*afișează informații despre noduri*/
```

#### 8.4.2. Exemple de sisteme de operare distribuite

##### 8.4.2.1. Sistemul de operare AMOEBA

Sistemul de operare AMOEBA a fost creat de profesorul Andrew Tanenbaum la Vrije Universiteit Amsterdam. S-au urmărit două scopuri:

-crearea unui sistem de operare distribuit care să gestioneze mai multe calculatoare interconectate într-un mod transparent pentru utilizator, astfel încât să existe iluzia utilizării unui singur calculator;

-crearea unei platforme pentru dezvoltarea limbajului de programare distribuit ORCA.

AMOEBA oferă programatorilor două mecanisme de comunicație:

- comunicație RPC (Remote Procedure Call);
- comunicație de grup.

Comunicația de tip RPC utilizează trei primitive: trans /\*un client transmite o cerere spre server\*/

get-request /\*severul își anunță disponibilitatea\*/

put-reply /\*serverul comunică rezultatul unei cereri\*/

Comunicația de grup asigură ca toate procesele din grup să primească aceleași mesaje și în aceeași ordine. În AMOEBA există un proces *secvențiator* care are două roluri:

- acordă numere de ordine mesajelor care circulă în grup;
- păstrează o istorie a mesajelor și realizează, atunci când este cazul, o retransmisie a mesajelor care nu au fost transmise.

Transmiterea unui mesaj către grup se face în două moduri, funcție de lungimea mesajului:

-se transmite mesajul către secvențiator, acesta atașează mesajului un număr de secvențe după care secvențiatorul difuzează mesajul către toate procesele; fiecare mesaj trece de două ori prin rețea;

-se anunță prin difuzare că se dorește transmiterea unui mesaj iar secvențiatorul răspunde prin acordarea unui număr de secvențe după care procesul care a dorit să transmită mesajul face difuzarea acestuia; în acest caz se difuzează mai multe mesaje, fiecare procesor fiind întrerupt odată pentru mesajul care solicită numărul de secvențe și a două oară pentru mesajul propriu zis.

Procesul care transmite mesajul se blochează până primește și el mesajul, ca orice proces din grup. Secvențiatorul păstrează o istorie a mesajelor transmise. Procesul anunță secvențiatorul numărul de secvență al ultimului mesaj recepționat. Secvențiatorul poate, la rândul său, să ceară situația mesajelor de la un proces care nu a mai transmis de multă vreme nimic. Utilizând aceste informații, secvențiatorul poate să își gestioneze în mod corespunzător istoria.

#### Structura sistemului de operare AMOEBA

AMOEBA este bazat pe un microkernel care rulează pe fiecare procesor în parte, deasupra căruia rulează servere ce furnizează servicii. Microkernelul asigură gestiunea principalelor resurse ale sistemului ce pot fi grupate în patru categorii.

- 1)Gestiunea proceselor și threadurilor.
- 2)Gestiunea de nivel jos a memoriei.
- 3)Gestiunea pentru comunicație.
- 4)Gestiunea operațiilor de intrare/ieșire de nivel jos.

1) Procesele reprezintă mecanismul care asigură execuția în AMOEBA. Un proces conține un singur spațiu de adrese. Threadurile (firele de execuție) sunt interne unui proces și au acces la spațiul de adrese al procesului.

În AMOEBA un procesor nu are proprietar și de aceea utilizatorul nu are nici un fel de control asupra procesoarelor pe care rulează aplicațiile. Sistemul de operare ia deciziile plasării unui proces pe un anumit procesor, în funcție de diferiți factori ca: încărcarea procesorului, memoria disponibilă, puterea de calcul etc.

Anumite procesoare pot fi dedicate rulării unor servere care cer multe resurse, de exemplu serverul de fișiere.

2)Gestiunea memoriei se face prin segmente de lungime variabilă. Acestea sunt păstrate în totalitate în memorie și sunt rezidente în memorie, adică sunt păstrate tot timpul în memorie. Nu se face swapping cu aceste segmente. Din această cauză dimensiunea memoriei trebuie să fie foarte mare, Tannenbaum considerând resursa memorie ca una ieftină.

Segmentele de memorie pot să fie mapate în spațiul de adresă al mai multor procese care se execută pe același procesor. În felul acesta se creează memorie partajată.

- 3)Gestiunea comunicației se face, aşa cum a fost prezentat, prin RPC și comunicație de grup.

4)Operațiile de intrare/ieșire se efectuează prin intermediul driverelor. Pentru fiecare dispozitiv există un driver. Acestea aparțin microkernelului și nu pot fi adăugate sau șterse dinamic. Comunicația driverelor cu restul sistemului se face prin RPC. Deoarece s-a ales o soluție cu microkernel, serverele sunt aceleia care au preluat sarcina vechiului kernel. Exemple de servere sunt:

- servere de fișiere;
- serverul de boot;
- serverul de execuție;
- serverul de TCP/IP;
- serverul de generare a numerelor aleatorii;

- serverul de erori;
- serverul de mail.

Pentru principalele servere există apeluri de biblioteci cu ajutorul cărora un utilizator poate accesa obiecte. Există un compilator special cu ajutorul căruia se pot crea funcții de bibliotecă pentru serverele noi create. În AMOEBA toate resursele sunt văzute ca niște *obiecte*. Acestea sunt entități ce apar într-o nouă paradigmă de programare. Prin definiție, un obiect este o colecție de variabile care sunt legate împreună de un set de proceduri de acces numite *metode*. Proceselor nu li se permite accesul direct la aceste variabile ci li se cere să invoke metode. Un obiect constă dintr-un număr de cuvinte consecutive în memorie (de exemplu în spațiul de adresă virtuală din kernel) și este deci o structură de date în RAM.

Sistemele de operare care utilizează obiectele, le construiesc astă fel ca să furnizeze o interfață uniformă și consistentă cu toate resursele și structurile de date ca procese, threaduri, smafoare etc. Uniformitatea constă în numirea și accesarea în același mod al obiectelor, gestionarea uniformă a partajării obiectelor între procese, punerea în comun a controlorilor de securitate, gestionarea corectă a cotelor de resurse etc.

Obiectele au o structură și pot fi tipizate. Structurarea se face prin două părți principale:

- o parte (*header*) care conține o anumită informație comună tuturor obiectelor de toate tipurile ;
- o parte cu date specifice obiectului.

Obiectele sunt gestionate de servere prin intermediul *capabilităților*. La crearea unui obiect, serverul construiește o capacitate protejată criptografic pe care o asociază obiectului. Clientul primește această capacitate prin care va accesa obiectul respectiv.

Concluzii.

Principala deficiență a sistemului de operare AMOEBA este că nu s-a creat o variantă comercială a lui, din mai multe motive:

-fiind un produs al unui mediu universitar, nu a avut forță economică să se impună pe piața IT;

-deși a fost unul dintre primele sisteme de operare distribuite (proiectarea a început în anii optzeci) și deși a introdus multe concepte moderne, preluate astăzi în majoritatea sistemelor de operare distribuite (microkernelul, obiecte pentru abstractizarea resurselor, RPC, comunicații de grup), alte firme au introdus rapid alte sisteme de operare.

#### **8.4.2.2. Sistemul de operare GLOBE**

În prezent, profesorul Tanenbaum lucrează la un nou sistem de operare distribuit, denumit GLOBE, ale cărui prime versiuni au apărut deja. Este un sistem de operare, creat pentru aplicații distribuite pe o arie largă, care utilizează obiecte locale sau distribuite. În GLOBE, un obiect este o entitate formată din:

- o colecție de valori ce definesc starea obiectului;
- o colecție de metode ce permit inspectarea și modificarea stării obiectului;
- o colecție de interfețe.

Un obiect local este conținut în întregime (stare, metode, interfață) într-un singur spațiu de adresă.

Un obiect distribuit este o colecție de obiecte locale care aparțin unor spații de adrese diferite.

Obiectele locale comunică între ele pentru a menține o stare globală consistentă.

## 9. SECURITATEA SISTEMELOR DE OPERARE

### 9.1. NOȚIUNI INTRODUCTIVE

Termenul de securitate, într-un sistem de operare, implică noțiuni multiple și complexe legate de foarte multe aspecte. Este greu de limitat partea din acțiunea de securitate ce revine sistemului de operare, deoarece aceasta este o chintesnă a hardului, programării, tehnicilor de programare, structurilor de date, rețelelor de calculatoare etc. De aceea vom încerca să tratăm în general problemele de securitate, insistând asupra celor legate strict de sistemul de operare.

Din punctul de vedere al sistemului de calcul și, implicit, al sistemului de operare, există trei concepte fundamentale de securitate care reprezintă în același timp și obiective generale de securitate:

- confidențialitatea;
- integritatea;
- disponibilitatea.

Confidențialitatea se referă la accesul datelor, adică la faptul că anumite date, considerate secrete, nu trebuie să fie accesate de utilizatori neautorizați. Proprietarii datelor au dreptul să specifice cine are acces la ele iar sistemul de operare trebuie să impună aceste specificații. Principala amenințare este expunerea datelor iar în momentul în care datele sunt accesate de persoane neautorizate are loc o pierdere a confidențialității.

Integritatea se referă la faptul că datele pot fi modificate numai de utilizatori autorizați; în caz contrar, adică atunci când un utilizator neautorizat modifică niște date, are loc o pierdere de integritate. Principala amenințare, în acest caz, este coruperea datelor.

Disponibilitatea se referă la faptul că datele sunt accesibile pentru utilizatorii autorizați la un moment dat. Atunci când datele nu sunt disponibile, este vorba de refuzul serviciilor (denial of service).

Alte două concepte din securitatea sistemelor de calcul sunt legate de dreptul unor utilizatori la accesul de date:

- autentificarea;
- autorizarea.

Autentificarea înseamnă operațiunea de demonstrare că un utilizator are identitatea declarată de acesta, operațiune ce presupune solicitarea unor informații suplimentare de la utilizatorul respectiv.

Autorizarea reprezintă constatarea dreptului unui utilizator de a efectua anumite operații.

Cele mai răspândite forme de autentificare sunt:

- autentificarea prin parole;
- autentificarea provocare-răspuns;
- autentificarea ce folosește un obiect fizic;
- autentificarea ce folosește date biometrice.

Autentificarea prin parole este una dintre cele mai răspândite forme de autentificare în care utilizatorul trebuie să tasteze un nume de conectare și o parolă. Este o formă de autentificare ușor de implementat. Cea mai

simplă implementare constă dintr-o listă în care sunt stocate perechi de nume-parolă. Această listă este păstrată de sistemul de operare într-un fișier de parole stocat pe disc. Stocarea poate fi:

- necriptată;
- criptată.

Soluția necriptată este din ce în ce mai puțin utilizată, deoarece oferă un grad de securitate redus, prea mulți utilizatori având acces la acest fișier.

Soluția criptată utilizează parola drept o cheie pentru criptarea unui bloc fix de date. Apoi programul de conectare citește fișierul cu parole, care este de fapt o sumă de linii scrise în alfabetul ASCII, fiecare linie corespunzând unui utilizator. Dacă parola criptată conținută în acea linie se potrivește cu parola introdusă de utilizator, criptată și ea, atunci este permis accesul.

O îmbunătățire a acestei soluții este utilizarea parolelor de unică folosință, când utilizatorul primește un caiet ce conține o listă de parole. La fiecare conectare se folosește următoarea parolă din listă. Dacă un intrus descoperă vreodată o parolă, o poate folosi doar o singură dată, la următoarea conectare trebuind altă parolă.

O schemă elegantă de generare a unor parole de unică folosință este schema lui Lamport, elaborată în 1981. Dacă  $n$  este numărul de parole de unică folosință, atunci se alege o funcție unidirecțională  $f(x)$ , cu  $y=f(x)$  și proprietatea că fiind dat  $x$  este ușor de găsit  $y$  dar fiind dat  $y$  este foarte greu de găsit  $x$ . Dacă prima parolă secretă este  $p$ , atunci prima parolă de unică folosință este dată de legea:

$$\text{Parola}_1 = P_1 = f(f(f(\dots\dots\dots f(s)))) \dots\dots\dots$$

$\leftarrow$  n ori  $\rightarrow$        $\leftarrow$  n ori  $\rightarrow$

$$\text{Parola}_2 = P_2 = f(f(f(\dots\dots\dots f(s)))) \dots\dots\dots$$

$\leftarrow$  n-1 ori  $\rightarrow$        $\leftarrow$  n-1 ori  $\rightarrow$

$$\text{Parola}_n = P_n = f(s)$$

Dacă  $P_i$  (cu  $1 \leq i \leq n$ ) este o parolă de unică folosință la „ $i$ ”-a alegere, atunci:

$$P_{i-1} = f(P_i)$$

Cu alte cuvinte, se poate calcula ușor parola anterioară dar nu este nici o posibilitate de a calcula parola următoare.

Autentificarea provocare-răspuns constă într-o serie de întrebări puse utilizatorului, fiecare întrebare având un răspuns. Toate răspunsurile la întrebări sunt stocate în sistemul de operare. După compararea răspunsurilor se permite sau nu conectarea utilizatorilor.

O altă variantă este folosirea unui algoritm care să stea la baza acestui răspuns. Utilizatorul alege o cheie secretă,  $c$ , pe care o instalează pe server. În momentul conectării, serverul trimite un număr aleatoriu,  $a$ , spre utilizator. Aceasta calculează o funcție  $f(a, c)$  unde  $f$  este o funcție cunoscută, pe care o retrimite serverului. Serverul verifică dacă rezultatul primit înapoia se potrivește cu cel calculat. Avantajul acestei metode este acela că, chiar dacă tot traficul dintre server și utilizator este interceptat de un intrus, acest lucru nu îi va permite intrusului să se conecteze data viitoare. Funcția  $f$  trebuie să fie suficient se complexă, astfel încât să nu poată fi dedus, având un număr foarte mare de eșanțioane de trafic.

Autentificarea ce folosește un obiect fizic. Dacă în autentificarea provocare-răspuns se verifică ceea ce utilizatorii și-au, în autentificarea ce folosește obiecte fizice se verifică niște caracteristici fizice ale utilizatorilor.

Obiectele fizice cele mai folosite sunt cartelele care pot fi de mai multe feluri:

- cartele magnetice;
- cartele electronice;
- cartele inteligente.

Cartelele magnetice stochează informațiile sub formă magnetică, existând senzori magnetici care au rolul de citire/scriere. Senzorii magnetici sunt foarte diferenți: magnetostrictivi, cu efect Hall, cu magnetoimpedanță etc. De obicei, în aceste cartele sunt stocate parole. Din punct de vedere al securității, cartelele magnetice sunt destul de riscante deoarece senzorii magnetici utilizati sunt ieftini și foarte răspândiți.

Cartelele electronice au la bază senzori electronici care înseamnă toți senzorii electrici afară de cei magnetici.

Cartelele inteligente au la bază un microprocesor, de obicei simplu, pe 8 biți. Ele utilizează un protocol criptografic, bazat pe principii criptografice pe care le vom studia în capitolul următor.

Autentificarea ce folosește date biometrice se bazează pe folosirea unor caracteristici fizice ale utilizatorilor, numite date biometrice, și care, de obicei, sunt unice pentru fiecare persoană. Astfel de date biometrice pot fi: amprente digitale, amprente vocale, tiparul retinei etc. Un astfel de sistem de autentificare are două părți:

- înrolarea;
- identificarea.

Înrolarea constă din măsurarea caracteristicilor utilizatorului, din digitizarea rezultatelor și din stocarea lor prin înregistrare într-o bază de date asociată utilizatorului și aflată în sistemul de operare.

Identificarea constă din măsurarea, încă odată, a caracteristicilor utilizatorului care vrea să se conecteze și din compararea lor cu rezultatele culese la înrolare.

Problema esențială în acest tip de autentificare este alegerea caracteristicilor biometrice, caracteristici care trebuie să aibă suficientă variabilitate, încât sistemul să poată distinge fără eroare dintre mai multe persoane.

O caracteristică nu trebuie să varieze mult în timp. Vocea unei persoane poate să se schimbe în timp, mai ales la persoanele instabile psihic, deci această caracteristică nu oferă o stabilitate temporală.

O caracteristică biometrică din ce în ce mai mult utilizată în ultimul timp, tocmai datorită unei bune stabilități temporale, este tiparul retinei. Fiecare persoană are un diferit tipar de vase de sânge retinale, chiar și gemenii. Aceste tipare pot fi fotografiate cu acuratețe.

## 9.2. ATACURI ASUPRA SISTEMULUI DE OPERARE ȘI MĂSURI DE PROTECȚIE ÎMPOTRIVA LOR

Există numeroase tipuri de atacuri asupra unui sistem de operare și, implicit, mai multe clasificări. O clasificare a atacurilor constă în:

- atacuri din interiorul sistemului;
- atacuri din exteriorul sistemului.

Atacurile din interior sunt săvârșite de utilizatori deja autorizați iar atacurile din exterior sunt executate, de cele mai multe ori, prin intermediul unei rețele de calculatoare.

Vom prezenta, în continuare, principalele atacuri ce se pot executa asupra unui sistem de operare.

### **9.2.1. Depășirea zonei de memorie tampon (Buffer Overflow)**

De multe ori, spațiul de memorie alocat unui program se dovedește a fi insuficient și se depășește acest spațiu, informațiile fiind stocate la o altă adresă. Acest tip de atac este un atac din interiorul sistemului și poate fi intenționat sau nu.

Un caz tipic de atac neintenționat este cel al programatorului în limbajul C, care lucrează cu vectori în memorie și care nu face, prin program, verificarea limitelor de vectori. Limbajul C este unul flexibil, chiar prea flexibil, iar compilatorul de C nu face verificarea limitelor vectorilor, lăsând acest lucru în seama programatorilor.

Modul de combatere a acestui atac se face prin utilizarea de tehnici de programare corecte care să verifice eventualele depășiri ale dimensiunilor zonelor de memorie alocate dar și prin instalarea de versiuni actualizate ale pachetelor de programe.

### **9.2.2. Ghicirea parolelor (Password guessing)**

Acest atac înseamnă încercarea de aflare a unor parole. De obicei se utilizează un program creat de către CRACKERI (spărgători de parole), program care, printr-o analiză comparativă, poate determina o corespondență între variantele presupuse criptate. Cel mai simplu program de spargere a parolelor este generarea de cuvinte până se găsește unul care să se potrivească. Cuvintele sunt generate fie prin permutări de componente fie prin utilizarea cuvintelor unui dicționar. Dacă parolele sunt criptate atunci mecanismele de decriptare sunt mai diferite.

Modalitățile de protecție împotriva atacului de ghicire a parolelor sunt:

- utilizarea sistemului *shadow*, pentru ca fișierul de parole să nu poată fi accesat de utilizatori;
- impunerea pentru utilizatori a unor reguli stricte la schimbarea parolelor;
- educarea utilizatorilor, în sensul că aceștia trebuie să respecte niște reguli fixe de stabilirea parolelor;
- folosirea periodică a unui program spărgător de parole, pentru a verifica complexitatea acestora, și atenționarea utilizatorilor respectivi.

### **9.2.3. Interceptarea rețelei (IP sniffing)**

Acest atac constă în monitorizarea informațiilor care circulă printr-o interfață de rețea, pentru detectarea eventualelor parole necriptate. Programele care efectuează interceptarea traficului din rețea se numesc *sniffere*. Se utilizează un interceptor de rețea și apoi se face captarea traficului într-un fișier. Deoarece viteza rețelelor a crescut mult în ultimul timp, fișierul în care se intercepteză rețeaua devine foarte mare în scurt timp, putând umple întreg hard discul. Din această cauză se obișnuiește să se capteze primele sute de octeți ai pachetelor, unde, cu mare probabilitate, se va afla numele și parola utilizatorului.

Este un atac din interior și se efectuează asupra parolei. De aceea mijlocul de combatere cel mai obișnuit trebuie să fie criptarea parolelor. O altă metodă de protecție este segmentarea rețelei în mai multe subrețele, utilizarea switch-urilor fiind indicată.

### **9.2.4. Atacul de refuz al serviciului (Denial Of Service)**

Prin aceste atacuri se degradează sau se dezafectează anumite servicii ale sistemului de operare. În rețelele de calculatoare, de exemplu, există bombardamentul cu pachete, cunoscut ca PACKET FLOOD care constă în transmiterea către un calculator țintă un număr foarte mare de pachete de date, având ca rezultat încărcarea traficului în rețea. Uneori se poate ajunge chiar la blocarea rețelei. Atacul poate proveni de la singură

sursă (DOS= Denial Of Service) sau de la mai multe surse (DDOS=Distributed Denial Of Service), caz mai rar întâlnit. Există trei tipuri de bombardamente cu pachete:

a) TCP - Bombardamentul se face în protocolul TCP (Transmission Control Protocol), iar un flux de pachete TCP sunt trimise spre țintă.

b) ICMP - Acest atac se mai numește PING FLOOD și utilizează pachete ICMP.

d) UDP – Bombardamentul se realizează, cu un flux de pachete UDP (User Datagram Protocol) trimise spre țintă.

Pentru a deruta filtrele de pachete existente în interiorul fiecărei rețele, în bombardarea cu pachete există programe speciale care modifică atributele pachetelor trimise. Exemple:

-se modifică adresa IP-sursă, (IP-Spoofing = falsificarea adresei IP), pentru a ascunde identitatea reală a pachetelor;

-se modifică portul sursei sau destinației;

-se modifică alte valori ale atributelor din antetul pachetelor IP.

Exemple de bombardamente cu pachete, pentru încărcarea traficului.

-Atacul **SYN-flood**. Se trimit pachete care au numai bitul de SYN setat. În felul acesta se deschid multe conexiuni care sunt incomplete. Deoarece fiecare conexiune trebuie prelucrată până la starea finală, se va depăși timpul admis, se va declara *time-out* și sistemul se va bloca.

- Atacul **smarf**. Este un atac de tipul ICMF împotriva unei ținte care este adresa broadcast a rețelei. Atacul se face cu adresă sursă IP modificată și va duce la generare de trafic suplimentar.

- Atacul **fraggle**. Este un atac cu pachete UDP având ca țintă portul 7 al adresei broadcast al rețelei. În felul acesta un singur pachet va fi transmis întregului segment al rețelei.

Exemple de bombardamente cu pachete, în vederea unor vulnerabilități ale serviciilor retelei.

- Atacul **tear-drop**. Acest atac exploatează protocolul TCP pentru fragmentele IP suprapuse ce nu sunt gestionate corect, adică cele care nu verifică corectitudinea lungimii fragmentelor.

- Atacuri **land craft**. Se trimit pachete SYN ce au adresa sursei identică cu adresa destinație, deschizând astfel o conexiune vidă.

- Atacuri **ping of death**. Se trimit pachete ICMP de dimensiuni foarte mari, ce depășesc lungimea standard de 64 kB, cât permite ICMP, depășirea acestei valori ducând la disfuncții ale stivei de comunicație.

- Atacuri **naptha**. Acest atac constă în deschiderea unui număr mare de conexiuni și abandonarea lor în diferite stări. La un moment dat se ajunge la refuzul serviciilor de rețea a calculatorului țintă.

Modalitățile de prevenire a atacurilor de tip DOS sunt:

- utilizarea unor versiuni cât mai recente ale sistemului de operare;

-implementarea mecanismului SYNCOOKIES care constă în alegerea particulară ale numerelor inițiale de secvență TCP, în aşa fel ca numărul inițial de secvență din server să crească puțin mai repede decât numărul inițial de secvență de pe client;

-separarea serviciilor publice de cele private, utilizate în interiorul rețelei;

- utilizarea de IP separate pentru fiecare serviciu în parte (HTTP, SMTP,DNS.....);
- instalarea unei conexiuni de siguranță care să preia traficul extern în cazul unui atac PACKET FLOOD;
- instalarea de *firewall-uri* la nivel de pachet, pentru serviciile care nu se utilizează în mod curent;
- dezactivarea serviciilor ce nu sunt necesare;
- separarea Intranetului de Internet.

#### **9.2.5. Atacuri cu bomba e-mail**

Acest atac constă din trimiterea repetată a unui mesaj către aceeași țintă.

Principala metodă de protecție este refuzul mesajului primit de la utilizatorul respectiv.

Un alt atac de tip e-mail este atacul **SPAM (e-mail spamming)**. Acest atac este un atac mai nou în care se trimit mesaje nesolicitante, de cele mai multe ori de tip reclamă, de către un expeditor care utilizează o adresă falsă.

#### **9.2.6. Falsificarea adresei expeditorului (e-mail spoofing)**

Este un atac care constă din recepționarea de către utilizator a unui e-mail care are adresa expeditorului diferită de cea originală. Este utilizat, în general, pentru a ascunde adresa atacatorului. Această modificare a adresei expeditorului este favorizată de faptul că protocolul de transport al mesajelor, utilizat în rețele, (SMTP=Simple Mail Transfer Protocol), nu prevede nici un sistem de autentificare.

Prevenirea acestui tip de atac poate fi făcută prin diferite metode:

- utilizarea criptografiei pentru autentificare;
- configurarea serverului de e-mail pentru a refuza conectarea directă la portul SMTP sau limitarea accesului la el;
- stabilirea unui singur punct de intrare pentru e-mail-ul primit de rețea, permitându-se astfel concentrarea securității într-un singur punct precum și instalarea unui firewall.

#### **9.2.7. Cai troieni (Trojan horses)**

Caii troieni informatici sunt programe care se ascund sub forma unor fișiere executabile obișnuite. Odată pătrunși într-un fișier, gazda poate efectua orice operație.

Exemple:

- aplicațiile denumite ”văduva neagră” (black widow) de pe www care acționează asupra browserelor web, blocându-le sau deteriorându-le;
- caii troieni instalati în scriptul CGI , care deteriorează scriptul.

Ca mijloace de luptă împotriva cailor troieni se recomandă realizarea periodică de copii de siguranță a sistemelor de fișiere, pentru a putea restaura fișierele executabile originale în cazul alterării acestora.

#### **9.2.8. Uși ascunse (Back doors and traps)**

Sunt cazuri particulare de cai troieni. Se creează o ”Ușă” care de fapt este un utilizator nou și care permite acordare de privilegii speciale unui anumit utilizator.

### 9.2.9. Viruși

Un virus este o secvență de cod care se autoinserează într-o gazdă, inclusiv în sistemul de operare, pentru a se propaga. Această secvență de cod, neputând rula independent, apelează la execuția programului gazdă pentru a se putea activa.

Crearea virusului a pornit de la o idee a profesorului Cohen, la începutul anilor 80, care, într-un articol, explica că s-ar putea crea secvențe de cod-program care să provoace anumite daune.

Primii viruși sunt consemnați în istoria informaticii în 1987, în Pakistan. De atunci și până în prezent virușii au cunoscut o dezvoltare spectaculoasă, fiind principalii actori ai atacurilor din afara sistemului.

Dacă în anii 80 o eroare a unui sistem de calcul avea cauza principală în redusa fiabilitate hard, astăzi majoritatea erorilor sunt cauzate de viruși.

Trebuie remarcat că, la ora actuală, există o largă răspândire a virușilor în sistemul de operare WINDOWS și o răspândire foarte mică, chiar nulă, în sistemele de operare de tip UNIX.

Faptul că sistemele de operare UNIX nu sunt vulnerabile la viruși se datorează gestiunii stricte a memoriei și a proceselor ce se execută. Chiar dacă un virus reușește să pătrundă în sistemul UNIX, posibilitatea lui de replicare este extrem de redusă.

Lupta contra virușilor este astăzi una dintre cele mai importante probleme. Ca și în cazul virusului biologic, ideal ar fi ca virusul informatic să fie evitat. Pentru aceasta ar trebui respectate anumite reguli importante cum ar fi:

- alegera unui sistem de operare cu un înalt grad de securitate;
- instalarea de aplicații sigure și evitarea copiilor a căror proveniență este dubioasă;
- achiziționarea unui program antivirus bun și upgradarea sa cât mai des posibil;
- evitarea atașamentelor de pe e-mail;
- crearea frecventă a copiilor de siguranță pentru fișierele cele mai importante și salvarea lor pe medii de stocare externe (CD-uri, streamere etc.).

Programele antivirus create până în prezent folosesc diferite tehnici antivirus cu ar fi:

- scanarea de viruși;
- verificarea de integritate în care programul antivirus utilizează tehnica checksum-ului;
- verificarea de comportament în care programul antivirus stă tot timpul în memorie și captează el însuși toate apelurile sistem.

### 9.2.10. Vierme (Worms)

Viermii sunt niște viruși care nu se reproduc local ci pe alte calculatoare, de obicei prin Internet.

Un vierme este un program care poate rula independent, consumând resursele gazdei pentru a se executa și care poate propaga o versiune funcțională proprie către alte calculatoare. Viermele funcționează după principiul "caută și distrugă". Un vierme se răspândește în mod automat, instalându-se în calculatoarele ce prezintă vulnerabilități. Din păcate, la ora actuală factorul de multiplicare al viermilor este exponențial. Pe lângă acțiunile distructive, un vierme creează un trafic uriaș în rețea ducând la un refuz al serviciilor.

Exemple de viermi:

- viermele MORRIS este primul vierme din istorie, creat de un student de la Universitate Cornell;
- viermele CODE RED care exploatează un bug din WEB numit IIS.

### 9.3. MECANISME DE PROTECȚIE

În acest subcapitol vom prezenta principalele mecanisme de protecție folosite în sistemele de operare. Ne vom ocupa de două mecanisme de protecție curente, *criptografia și sistemele firewall*, precum și de două concepte de securitate, *monitorul de referință și sistemele de încredere*.

#### 9.3.1. Criptografie

Criptografia are ca scop transformarea unui mesaj sau a unui fișier, denumit *text în clar (plaintext)*, într-un *text cifrat*, denumit *ciphertext*.

Acest lucru se poate realiza în două moduri:

- criptarea cu cheie secretă (criptografia simetrică);
- criptarea cu chei publice (criptografia asimetrică).

##### 9.3.1.1. Criptografia cu chei secrete (criptografia simetrică)

În acest sistem este folosită o singură cheie, atât pentru criptarea cât și pentru decriptarea informației.

Între expeditor și destinatar se negociază un protocol comun, de maximă siguranță, care are rolul de a transmite de la expeditor la destinatar o cheie de criptare secretă.

În cadrul criptografiei cu chei secrete există mai multe tehnici:

- a)-cifrurile bloc (block ciphers);
- b)-cifrurile flux (stream ciphers);
- c)-codurile de autentificare a mesajelor(MAC)).

a)-Cifrul bloc transformă un bloc de text de lungime fixă într-un bloc de text criptat, de aceeași lungime, cu ajutorul unei chei secrete. Putem spune că în acest tip de criptare, deoarece nu se modifică numărul de caractere al textului inițial, are loc o permutare a caracterelor din setul inițial. Există mai multe tehnici de criptare:

- cifrul bloc iterativ;
- modul carte de coduri (ECB=Electronic Code Block);
- modul cu înlănțuire (CBC=Cipher Block Chaining);
- modul cu reacție (CFB=Cipher Feed Back);
- modul cu reacție la ieșire (OFB=Output Feed Back).

-Cifrul bloc iterativ. Se aplică la fiecare iterare o aceeași transformare, utilizând o subcheie. Setul de subchei este derivat din cheia secretă de criptare, prin intermediul unei funcții speciale. Numărul de cicluri dintr-un cifru iterativ depinde de nivelul de securitate dorit. În general, un număr ridicat de cicluri va îmbunătăți performanța, totuși, în unele cazuri, numărul de iterații poate fi foarte mare. Cifrurile Feistel reprezintă o clasă

specială de cifruri bloc iterative în care textul criptat este generat prin aplicarea repetată a aceleiași transformări sau a funcției iterative. Se mai numesc și cifruri DES (Data Encryption Standard). Într-un cifru Feistel, textul original este despărțit în două părți, funcția iterativă fiind aplicată unei jumătăți folosind o subcheie iar ieșirea acestei funcții este calculată SAU-EXCLUSIV cu cealaltă jumătate. Cele două jumătăți sunt apoi interschimbate.

**-Modul carte de coduri.** Fiecare text original (de fapt bloc de text) este criptat independent, cu alte cuvinte fiecărui bloc de text original îi corespunde un bloc de text cifrat.

**-Modul cu înlătuire.** Fiecare bloc de text original este calculat SAU-EXCLUSIV cu blocul criptat precedent și apoi este criptat. Este utilizat un vector de inițializare, de preferință pseudo aleatoriu.

**-Modul cu reacție.** Blocul cifrat precedent este criptat iar ieșirea este combinată cu blocul original printr-o operație SAU-EXCLUSIV.

b)-Cifrurile flux seamănă cu cifrurile bloc dar au avantajul că sunt mult mai rapide. Dacă cifrurile bloc lucrează cu blocuri mari de informație, cifrurile flux lucrează cu bucăți mici de text, de cele mai multe ori la nivel de bit.

c)-Coduri de autentificare a mesajelor (MAC=Message Authentication Code). Un asemenea cod este o etichetă de autentificare numită și sumă de control (checksum) și derivă din aplicarea unei scheme de autentificare, împreună cu o cheie secretă, unui mesaj. Spre deosebire de semnăturile digitale, Mac-urile sunt calculate și verificate utilizând aceeași cheie, astfel încât ele pot fi verificate doar de către destinatar. Există patru tipuri de MAC-uri:

-Sigure necondiționat. Sunt bazate pe criptarea unui drum unic. Textul cifrat al mesajului se autentifică pe sine însuși și nimici altcineva la drumul unic. Un MAC sigur condiționat poate fi obținut prin utilizarea unei chei secrete folosite doar odată.

-Bazate pe funcția de dispersie (HMAC). O funcție de dispersie H reprezintă o transformare ce primește la intrare valoarea  $m$  și returnează un sir de lungime fixă,  $h$ . Se utilizează una sau mai multe chei împreună cu o funcție de dispersie, pentru a produce o sumă de control care este adăugată mesajului.

-Bazate pe cifruri flux. Un cifru flux sigur este utilizat pentru a descompune un mesaj în mai multe fluxuri.

-Bazate pe cifruri bloc. Se criptează blocuri de mesaj utilizând DES și CBC, furnizându-se la ieșire blocul final al textului cifrat ca sumă de control.

Sistemul DES (Data Encryption Standard) este o aplicație a cheilor secrete. Se utilizează în acest sistem chei de 56 biți. Sistemul DES este destul de vulnerabil și de aceea el se utilizează împreună cu un sistem sigur de gestionare a cheilor de criptare. Variante mai performante ale DES-ului sunt:

-triple DES, unde se criptează datele de trei ori consecutiv;

-DESX, unde se utilizează o cheie de criptare de 64 biți, de tip SAU-EXCLUSIV, înainte de criptarea cu DES iar după DES se mai utilizează încă o dată o cheie de criptare.

### 9.3.1.2. Criptarea cu chei publice (Criptografia asimetrică)

Fiecare persoană deține câte două perechi de chei, una publică –ce poate fi chiar disponibilă pe Internet– și una privată. Avantajul acestui sistem de criptare este că nu este necesară asigurarea securității transmisiei informației. Oricine poate transmite o informație utilizând cheia publică dar informația nu poate fi decriptată decât prin intermediul cheii private, deținută doar de destinatar. Cheia privată este într-o legătură matematică cu cea publică. Protejarea împotriva unor atacuri care se fac prin derivarea cheii private din cheia publică se realizează făcând această derivare cât mai dificilă, aproape imposibilă.

Criptografia cu chei secrete este utilizată de sistemele tradiționale. Este un mod mult mai rapid dar are dezavantajul că modul de transmisie a cheilor trebuie să fie foarte sigur.

**Sistemul RSA (Rivest Shamir Adleman).** Acest sistem utilizează chei publice și oferă mecanisme de criptare a datelor și semnături digitale.

Algoritmul de funcționare al sistemului RSA este următorul:

- se aleg două numere prime mari  $p$  și  $q$
- se calculează  $n=pq$
- se alege un număr  $e$ ,  $e < n$ ,  $e$  fiind prim cu  $(p-1)(q-1)$
- se calculează un număr  $d$ , astfel încât  $(ed-1)$  să fie divizibil prin  $(p-1)(q-1)$
- cheia publică este  $(n,e)$
- cheia privată este  $(n,d)$

Obținerea cheii private  $d$  pornind de la cheia publică  $(n,e)$  este dificilă. Se poate determina cheia privată  $d$  prin favorizarea lui  $n$  în  $p$  și  $q$ . Securitatea sistemului RSA se bazează pe faptul că această determinare este foarte dificilă.

Criptarea prin intermediul RSA se realizează astfel:

- expeditorul mesajului  $m$  creează textul cifrat  $c=m^e \text{ mod } n$ , unde  $(e,n)$  reprezintă cheia publică a destinatarului;
- la decriptare, destinatarul calculează  $m=c^d \text{ mod } n$ .

Relația dintre  $e$  și  $d$  asigură faptul că destinatarul decriptează corect mesajul. Deoarece numai destinatarul cunoaște valoarea lui  $d$ , doar el poate decripta mesajul.

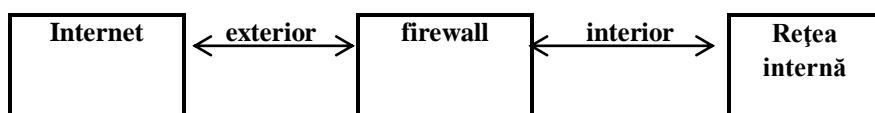
Siguranța RSA se bazează în primul rând pe gestionarea cheilor private, mecanism dependent de implementarea algoritmului RSA. De asemenea, este importantă alegerea unei perechi puternice de numere prime  $p$  și  $q$ . Numerele prime puternice au diferite proprietăți care le fac greu de fabricat.

Dimensiunea cheii utilizate într-un algoritm RSA se referă la dimensiunea lui  $n$ . Cu cât  $n$  este mai mare cu atât securitatea algoritmului este mai mare dar și funcționarea acestuia este mai lentă. Mărimea uzuală a cheii este de 1024 biți.

Sistemul RSA este utilizat împreună cu un sistem criptografic cu chei scurte, cum ar fi DES.

### 9.3.2. Dispozitive firewall

Un firewall este un sistem care separă o rețea protejată de una neprotejată, rețea neprotejată fiind în majoritatea cazurilor INTERNET. Un astfel de sistem monitorizează și filtrează traficul dintre cele două rețele, conform unei politici predefinite de control al accesului.



### Fig. 9.1. Locul unui firewall

Termenul firewall înseamnă ”perete de foc” și arată capacitatea de a segmenta o rețea mai mare în subrețele.

Un firewall are două interfețe:

- una către exterior, de cele mai multe ori către Internet;
- una direcționată către rețeaua internă pe care o protejează.

Filtrarea traficului dintre aceste două rețele se face după anumite criterii și vizează:

- adresele IP sursă și destinație ale pachetelor de informație vehiculate (address filtering);
- anumite porturi și protocoale (HTTP, FTP, TELNET) (protocol filtering).

Un firewall de rețea nu poate administra transferul de date efectuat de către un utilizator care folosește o legătura la Internet de tip deal-up, ocolind procedurile de securitate și implicit firewall-ul în sine.

#### **9.3.2.1. Tipuri de firewall**

Putem considera patru tipuri :

1) Firewall-uri cu filtrare de pachete (Packet Filtering Firewalls). Funcționează la nivelul IP al modelului OSI și respectiv la nivelul IP al modelului TCP/IP. Se analizează sursa de proveniență și destinație a fiecărui pachet în parte, acceptând sau blocând traficul derulat de acestea. De obicei, acest tip de firewall este implementat la nivel de router, ceea ce implică un cost minim. Principalul său dezavantaj constă în incapacitatea de a furniza o securitate, prin reguli complexe de identificare și validare a IP-urilor, motiv pentru care este indicată utilizarea împreună cu un alt firewall extern care să ofere protecție suplimentară.

2) Poarte de circuit (Circuit Level Gateways). Rulează la nivelul 5 al modelului OSI și respectiv nivelul 4 al modelului TCP/IP. Se monitorizează sesiunile TCP dintre rețeaua internă și rețeaua Internet. Se utilizează un server intermediar care maschează datele de pe calculatoarele rețelei private. Punctul slab al porților de curent este faptul că nu se verifică pachetele ce constituie obiectul traficului cu rețeaua publică ci doar filtrează în funcție titlu.

3) Proxi-uri de aplicatie (Proxies). Sunt cele mai complexe soluții de tip firewall dar și cele mai scumpe. Funcționează la nivel de aplicație al modelului OSI. Se verifică pachetele de date și se blochează accesul pachetului care nu respectă regulile stabilite de proxy. Pentru un proxy de web, de exemplu, acesta nu va permite niciodată accesul unui trafic de protocol FTP sau TELNET. Serverul local vede această soluție ca pe un simplu client, în timp ce rețeaua publică îl va recepta ca fiind însuși serverul. Proxi-urile de aplicație pot crea fișiere de tip log cu activitatea utilizatorilor din rețea sau pot monitoriza autentificările acestora, oferind și o verificare de bază a pachetelor transferate cu ajutorul antivirusului încorporat.

4) Firewall-uri cu inspecții multistrat (Stateful Multilayer Inspections). Este o combinație între cele trei tipuri descrise anterior. Acest tip se bazează pe algoritmi proprii de recunoaștere și aplicare a politicilor de securitate, spre deosebire de o aplicație proxy standard. Iinspecția multinivel oferă un înalt grad de securitate, performanțe foarte bune și o transparență oferită end-user-urilor. Este cea mai scumpă dar, fiind foarte complexă, se poate transforma într-o armă împotriva rețelei pe care o protejează dacă nu este administrată de personal competent.

#### **9.3.2.2. Funcțiile unui firewall**

Un firewall are următoarele posibilități:

- monitorizează căile de acces în rețeaua privată, permitând în acest fel o mai bună monitorizare a traficului și o mai ușoară detectare a încercărilor de infiltrare;
- blochează traficul înspre și dinspre Internet;
- selectează accesul în spațiul privat pe baza informațiilor conținute în pachete;
- permite sau interzice accesul la rețeaua publică de pe anumite spații specificate;
- poate izola spațiul privat de cel public și realiza interfață dintre cele două.

Un firewall nu poate să execute următoarele:

- să interzică importul/exportul de informație dăunătoare vehiculată ca urmare a acțiuni răutăcioase a unor utilizatori aparținând spațiului privat, cum ar fi căsuța poștală și atașamentele;
- să interzică scurgerea de informație de pe alte căi care ocolește firewall-ul (acces prin deal-up ce nu trece prin router);
- să apere rețeaua privată de userii ce folosesc sisteme fizice mobile de introducere a datelor în rețea (USB Stick, CD, dischete);
- să prevină manifestarea erorilor de proiectare ale aplicațiilor ce realizează diverse servicii, precum și punctele slabe ce decurg din exploatarea acestor greșeli.

### **9.3.2.3. Firewall-uri în sistemele de operare Windows**

Vom exemplifica instalarea și configurarea unui firewall în Windows. Am ales versiunea **trial** a programului **Zone Alarm Pro**.

Pentru început, programul se instalează în mod standard, urmând ca setările să se facă ulterior. Programul oferă, pe lângă protecția firewall, protejarea programelor aflate pe hard disc, protecția e-mail și posibilitatea blocării cookies-urilor, pop-urilor și barierelor nedorite.

Setările se pot face pe trei nivele diferite:

- High, nivel ce oferă o protejare cvasitotală, strict monitorizată, dar care împiedică opțiunile de sharing;
- Medium, nivel ce oferă o setare cu posibilitatea vizionării resurselor proprii din exterior dar fără posibilitatea modificării acestora;
- Low, nivel la care firewall-ul este inactiv iar resursele sunt expuse la atacuri.

Pentru poșta electronică programul se comportă ca o adeverată barieră în fața unor eventuale supraaglomerări a căsuței poștale, având posibilitatea blocării e-mail-urilor ce vin consecutiv. În default există opțiunea de acceptare a maximum 5 e-mail-uri ce vin într-un interval de 2 secunde. Există și opțiunea refuzării e-mail-urilor însoțite de diverse atașamente sau a acceptării lor doar pentru cele ce nu au un număr de recipiente atașate care depășește totalul admis de utilizator.

### **9.3.2.4. Firewall-uri în sistemul de operare Linux**

În sistemul de operare Linux a fost implementat la nivel de kernel un firewall numit **iptables**. Acest sistem oferă posibilitatea de a filtra sau redirecționa pachetele de date, precum și de a modifica informațiile despre sursa și destinația pachetelor, procedură numită NAT (Network Address Translation). Una dintre aplicațiile sistemului NAT este posibilitatea deghizării pachetelor (macquerading). Deghizarea înseamnă că pachetele trimise de către sistemele aflate în rețea, care au stabilite ca gateway o anumită mașină, să pară transmisă de mașina respectivă și nu de cea originară. Mașina, configurată ca firewall, retrimit pachetele venite, dinspre rețea spre exterior, făcând să pară că provin tot de la ea. Acest mecanism este foarte util atunci când există o mașină care realizează legătura la Internet, o singură adresă IP alocată și mai multe calculatoare în rețea care au definită mașina respectivă ca gateway. Situația este des întâlnită în cadrul companiilor mici și mijlocii dotate cu o legătură Internet permanentă, mai ales ca urmare a crizei de adrese IP manifestată în ultimii ani.

Nucleul Linux definește două tabele de reguli:

- filter**, utilizată pentru filtrul de pachete;
- nat**, utilizată pentru sistemul NAT.

Există 5 lanțuri predefinite care înseamnă o succesiune de reguli utilizate pentru verificarea pachetelor de date ce tranzitează sistemul. Acestea sunt:

- lanțul INPUT, disponibil pentru tabela filter;
- lanțul FORWARD, disponibil pentru tabela filter;
- lanțul PREROUTING, disponibil pentru tabela nat;
- lanțul POSROUTING, disponibil pentru tabela nat;
- lanțul OUTPUT, disponibil pentru ambele tabele.

Atunci când pachetul intră în sistem printr-o interfață de rețea, nucleul decide dacă el este destinat mașinii locale (lanțul INPUT) sau altui calculator (lanțul FORWARD). În mod similar, pachetele care ies din mașina locală trec prin lanțul OUTPUT. Fiecare lanț conține mai multe reguli ce vor fi aplicate pachetelor de date care le tranzitează. În general, regulile identifică adresele sursă și destinație a pachetelor, însă de porturile sursă și destinație, precum și protocolul asociat. Când un pachet corespunde unei reguli, adică parametrii menționați mai sus coincid, asupra pachetelor se va aplica o anumită acțiune care înseamnă direcționarea către o anumită țintă. Dacă o regulă specifică acțiunea accept, pachetul nu mai este verificat folosind celelalte reguli ci este direcționat către destinație. Dacă regula specifică acțiunea DROP, pachetul este "aruncat", adică nu i se permite să ajungă la destinație. Dacă regula specifică acțiunea REJECT, pachetului tot nu i se permite să ajungă la destinație dar este trimis un mesaj de eroare expeditorului. Este important ca fiecărui lanț să-i fie atribuită o acțiune implicită care va reprezenta destinația pachetului dacă nici una din regulile stabilite nu corespunde. Astfel lanțul INPUT trebuie să aibă DROP sau REJECT ca acțiune implicită pentru orice pachet care se dorește a fi filtrat.

Dacă dorim să implementăm un firewall într-o mașină care realizează conexiunea Internet într-o companie privată, vom permite accesul din exterior la serviciile web, precum și la serviciile de e-mail. Conexiunea la rețeaua locală este realizată prin interfața **eth0**, iar cea pe Internet prin interfața **pppo**.

Există mai multe interfețe grafice pentru generarea de firewall-uri pentru sistemul **iptables**, cum ar fi : Firewall Bulder, Firestarter etc.

La sfârșitul implementării se verifică funcționarea firewall-ului, testarea fiind mai greu de realizat dacă acesta conține multe reguli. Testarea se face din afara rețelei, utilizând un scanner (SATAN sau NMAP) și din

interior. După ce s-au stabilit regulile de filtrare și s-a testat funcționarea firewall-ului este recomandabil să se salveze configurația curentă a sistemului iptables.

### 9.3.3. Sisteme de încredere

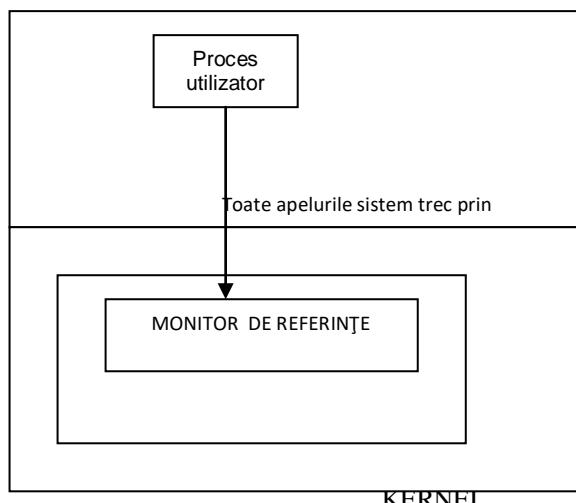
Sistemele de încredere sunt un concept mai nou și sunt capabile să ofere securitate la nivel de sistem. Pentru aceste sisteme s-au formulat clar cerințe de securitate care s-au și îndeplinit. Fiecare sistem de încredere conține o Bază de Calcul de Încredere (TCB= Trusted Computing Base) care constă în hardul și softul ce conțin toate regulile de securitate. Dacă această TCB funcționează conform specificațiilor, sistemul de securitate nu poate fi compromis.

TCB-ul constă dintr-o parte a kernelului sistemului de operare și din programele utilitare ce au putere de superutilizator. Funcțiile sistemului de operare care fac parte din TCB sunt separate de restul SO, pentru a le minimaliza dimensiunea și a le verifica corectitudinea.

Vom prezenta în continuare câteva modele de sisteme de încredere.

#### 9.3.3.1. Monitorul de referințe

Monitorul de referințe este o parte importantă a TCB-ului care acceptă toate apelurile sistem ce implică securitatea, cum ar fi deschiderea de fișiere, și decide dacă ar trebui să fie procesate sau nu. Astfel, monitorul de referință permite amplasarea tuturor deciziilor de securitate într-un singur loc, fără posibilitatea de a-l colăsi.



**Fig. 9.2. Schema unui monitor de referințe**

#### 9.3.3.2. Modelul cu Liste de Control al Accesului (ACL)

Un sistem de calcul conține multe obiecte care necesită a fi protejate. Aceste obiecte pot fi componente hard (unități de disc, segmente de memorie, imprimante etc.) sau componente soft (procese, fișiere, semafoare etc.). Fiecare obiect are un nume unic prin care este referit și un set finit de operații permise, de exemplu scriere/citire pentru fișiere, up/down pentru semafoare etc. Este necesară o modalitate de a interzice proceselor de a accesa obiecte pe care nu sunt autorizate să le acceseze. Trebuie creat un mecanism pentru restricționarea proceselor la un subset al operațiilor legale, când este necesar. Se introduce noțiunea de *domeniu de protecție* care înseamnă un set de perechi (obiecte, drepturi). Fiecare pereche specifică un obiect și un subset al operațiilor permise asupra lui. Un drept, în acest context, înseamnă permisiunea de a executa una din aceste operații. O implementare a domeniilor de protecție este matricea de protecție în care coloanele sunt obiecte iar liniile sunt domenii sau drepturi. Un exemplu de matrice de protecție este cel de mai jos:

<b>obiecte domenii</b>	<b>Fișier 1</b>	<b>Fișier 2</b>	<b>Fișier 3</b>	<b>Imprimanta 1</b>	<b>Imprimanta 2</b>
<b>1</b>	<b>citire</b>	<b>citire scriere</b>			
<b>2</b>			<b>citire scriere</b>		<b>scriere</b>
<b>3</b>				<b>scriere</b>	

**Fig.9.3. Matrice de protecție.**

Practic, o astfel de matrice are dimensiuni foarte mari iar împrăștierarea în cadrul matricei este și ea foarte mare. De aceea matricea se păstrează pe rânduri sau pe coloane, cu elemente nevide. Dacă matricea se păstrează pe coloane, structura de date este Lista de Control al Accesului (ACL = Acces Control List). Dacă se păstrează pe linii, se numește lista de capabilități (Capability List) sau lista C (C-List) iar intrările individuale se numesc capabilități.

Liste de Control al Accesului. Fiecare obiect are asociat o ACL în care sunt trecute drepturile respective. Pentru exemplul dat, ACL-urile obiectelor sunt:

**Fișier 1** →1;c;2;3;

**Fișier 2** →1;cs;2;3;

**Fișier 3** → 1;2;cse;3;

**Imprimantă 1**→1;2;3;s;

**Imprimantă 2**→1;2;s;3;

Proprietarul fișierului poate să modifice ACL-urile.

Capabilități. Se reține matricea pe linii. Listele 1,2,3 de capabilități ale obiectelor sunt, pentru exemplul de mai sus, următoarele:

1                  2                  3

**Fișier 1**              **Fișier 1:c**      **Fișier 3:cse**      **Imprimantă 1:s**

**Fișier 2**              **Fișier 2:cs**      **Imprimantă2:s**

**Fișier 3**

**Imprimantă 1**

**Imprimantă 2**

### 9.3.3.3. Modelul Bell-La Padula

Cele mai multe sisteme de operare permit utilizatorilor individuali să decidă cine poate să citească și să scrie fișierele sau alte obiecte proprii. Această politică se numește control discreționar al accesului. În unele medii modelul funcționează bine dar există și medii în care se cere o securitate superioară, de exemplu armata, spitalele etc. Pentru aceste medii sunt necesare o serie de reguli mult mai stricte. Se folosesc, în aceste cazuri, un control obligatoriu al accesului. Se regularizează fluxul informației pentru a se asigura că nu sunt scăpare în mod nebănuitor. Un astfel de model este și Bell-La Padula, folosit des și în armată. În acest model oamenilor din armată le sunt atribuite nivele în funcție de natura documentelor permise lor spre vizualizare. Un proces ce rulează din partea unui utilizator primește nivelul de securitate al utilizatorului. Din moment ce există mai multe nivele de securitate, această schemă se numește sistem de securitate multinivel. Modelul Bell-Padula aparține acestui sistem, el având reguli despre cum poate circula informația.

-Proprietatea simplă de securitate: un proces rulând la nivelul de securitate k poate citi doar obiecte la nivelul său sau mai jos.

-Proprietatea asterisc (\*): un proces rulând la nivelul de securitate k poate scrie doar obiecte la nivelul său sau mai sus.

Deci, conform acestui model, procesele pot citi în jos și scrie în sus. Dacă sistemul impune cu strictețe aceste două proprietăți, se poate arăta că nu se poate scurge nici o informație de la un nivel mai înalt la un nivel mai coborât. În acest model, ilustrat grafic în fig.9.4., procesele citesc și scriu obiecte dar nu comunică între ele.

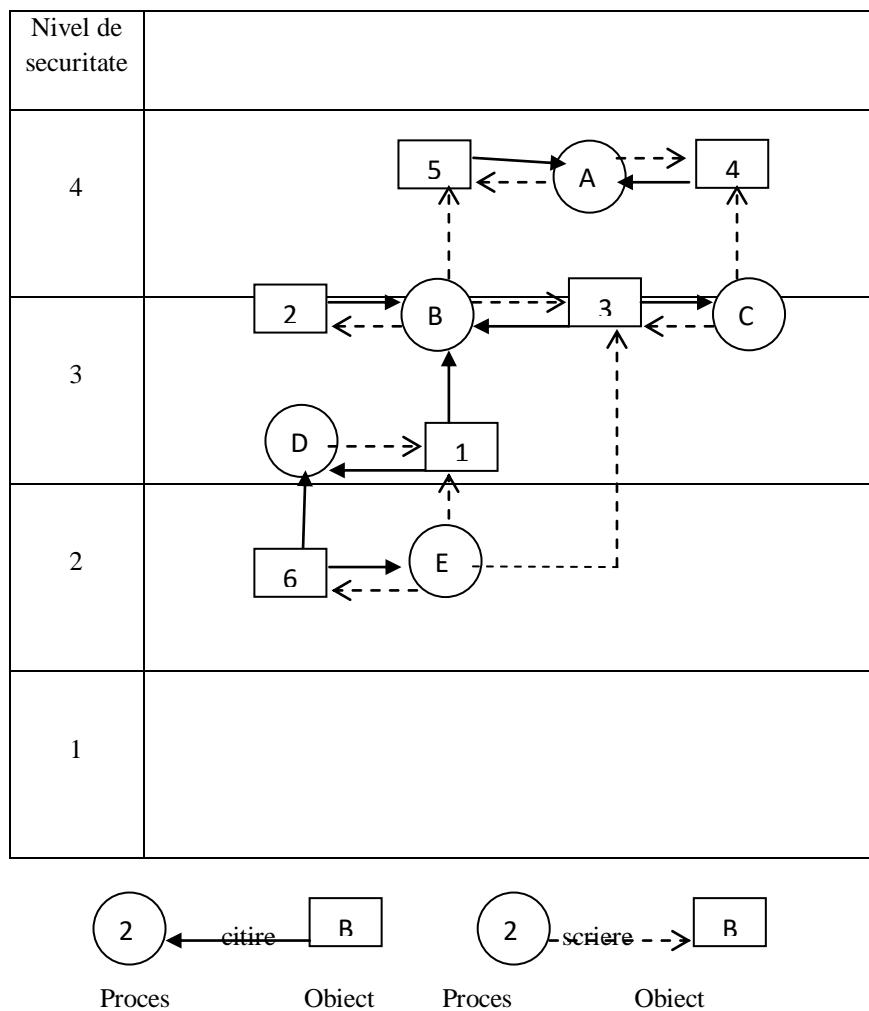


Fig. 9.4. Schemă de securitate multinivel.

Proprietatea simplă de securitate afirmă că toate săgețile continue (de citire) merg în lateral sau în sus iar săgețile întrerupte (de scriere) merg în lateral sau în sus. Din moment ce toată informația circulă doar orizontal sau în sus, orice informație ce pornește de la nivelul k nu poate să apară la un nivel inferior. Deci nu există nici o cale care să miște informația în jos, garantându-se astfel securitatea modelului. Dacă transpunem aceste lucruri în termeni militari, un locotenent poate să ceară unui soldat să-i dezvăluie tot ce știe ca apoi să copieze această informație în fișierul unui general, fără a încălca securitatea.

În concluzie, modelul Bell-La Padula a fost conceput să păstreze secrete dar nu să garanteze integritatea datelor.

#### **9.3.3.4. Modelul Biba**

Pentru a se garanta integritatea datelor ar trebui inversate proprietățile din modelul Bell-La Padula, adică:

- 1) Un proces rulând la nivelul de securitate k poate scrie doar obiecte la nivelul său sau mai jos.
- 2) Un proces rulând la nivelul de securitate k poate citi doar obiecte la nivelul său sau mai sus.

Aceste proprietăți stau la baza modelului Biba care asigură integritatea datelor dar nu și secretizarea lor. Bineînțeles că modelul Biba și modelul Bell-La Padula nu pot fi realizate simultan.

#### **9.3.3.5. Modelul securității Cărții Portocalii**

În 1985 Departamentul Apărării Statelor Unite ale Americii a publicat undокумент cunoscut sub denumirea de *Cartea Portocalie*, un document care împarte sistemele de operare în câteva categorii bazate pe criteriul de securitate. Iată mai jos criteriile de securitate ale Cărții Portocalii. Nivelele de securitate sunt notate cu D, C1, C2, B1, B2, B3, A1, în ordinea în care securitatea crește spre dreapta.

CRITERIU	D	C1	C2	B1	B2	B3	A1
<b>Politica de securitate</b>							
Control discretabil al accesului		x	x	→	→	x	→
Refolosirea obiectului			x	→	→	→	→
Etichete				x	x	→	→
Integritatea catalogării				x	→	→	→
Exportare a informației catalogate				x	→	→	→
Catalogarea ieșirii lizibile				x	→	→	→
Control obligatoriu al accesului				x	x	→	→
Catalogări ale sensibilității subiectului					x	→	→
Etichete ale dispozitivelor					x	→	→
<b>Posibilitatea de contabilizare</b>							
Identificare și autentificare		x	x	x	x	x	→
Audit			x	x	x	x	→

Cale de încredere					x	x	→
<b>Asigurare</b>							
Arhitectura sistemului		x	x	x	x	x	→
Integritatea sistemului		x	→	→	→	→	→
Testarea securității		x	x	x	x	x	x
Specificările și verificarea modelului				x	x	x	x
Analiza canalului camuflat					x	x	x
Administrarea caracterului de încredere					x	x	→
Administrarea configurației					x	→	x
Recuperarea de încredere						x	→
Distribuire de încredere							x
<b>Documentare</b>							
Ghid pentru aspecte de securitate		x	→	→	→	→	→
Manual al caracteristicilor de încredere		x	x	x	x	x	→
Documentație de testare		x	→	→	x	→	x
Documentație de proiectare		x	→	x	x	x	x

Legendă : x - sunt noi cerințe;

→ - cerințele pentru următoarea categorie

inferioară se aplică și aici.

Să analizăm categoriile de securitate și să subliniem câteva dintre aspectele importante.

Nivelul D de conformitate este ușor de atins, neavând nici un fel de cerințe de securitate. Adună toate sisteme ce au eșuat să treacă chiar și teste de securitate minimă. MS-DOS și Windows 98 sunt de nivel D.

Nivelul C este intenționat pentru medii cu utilizatori cooperativi. C1 necesită un mod protejat al sistemului de operare, conectarea utilizatorilor prin autentificare și capacitatea pentru utilizatori de a specifica ce fișiere pot fi făcute disponibile altor utilizatori și cum. Se cere o minimă testare de securitate și documentare. Nivelul C2 adaugă cerință ca și controlul discretizabil al accesului să fie coborât la nivelul utilizatorului individual. Trebuie ca obiectele date utilizatorului să fie inițializate și este necesară o minimă cantitate de auditare. Protecția rwh din Unix este de tipul C1 dar nu de tipul C2.

Nivelele B și A necesită ca tuturor utilizatorilor și obiectelor controlate să le fie asociată o catalogare de securitate de genul: neclasificat, secret, strict secret. Sistemul trebuie să fie capabil a impune modelul Bell-La Padula. B2 adaugă la această cerință ca sistemul să fi fost modelat de sus în jos într-o manieră modulară. B3 conține toate caracteristicile lui B2, în plus trebuie să fie ACL-uri cu utilizatori și grupuri, trebuie să existe un TCB formal, trebuie să fie prezentă o auditare adecvată de securitate și trebuie să fie incusă o recuperare sigură

după blocare. A1 cere un model formal al sistemului de protecție și o dovedă că sistemul este corect. Mai cere și o demonstrație că implementarea este conformă cu modelul.

#### 9.3.4. **Securitatea în sistemele de operare Windows**

Windows NT satisface cerințele de securitate C2 în timp ce Windows 98 nu satisface aceste cerințe.

Cerințele de securitate, de tip C2, ale versiunii Windows NT sunt:

- înregistrarea sigură , cu măsuri de protecție împotriva atacurilor de tip spoofing;
- mijloace de control al accesului neîngrădit;
- mijloace de control ale acceselor privilegiate;
- protecția spațiului de adrese al unui proces;
- paginile noi trebuie umplute cu zerouri;
- înregistrări de securitate.

Înregistrare sigură înseamnă că administratorul de sistem poate cere utilizatorilor o parolă pentru a se înregistra. Atacul de spoofing se manifestă atunci când un utilizator scrie un program răuvoitor care afișează caseta de înregistrare și apoi dispără. Când un alt utilizator va introduce un nume și o parolă, acestea sunt scrise pe disc iar utilizatorului i se va răspunde că procesul înregistrare a eşuat. Acest tip de atac este prevenit de Windows NT prin indicația că utilizatorul să tasteze combinația CTRL+ALT+DEL, înainte de a se înregistra utilizatorul. Această sevență este capturată de driverul de tastaturi care apelează un program al sistemului de operare ce afișează caseta de înregistrare reală. Procedeul funcționează deoarece procesele utilizator nu pot dezactiva procesarea combinației de taste CTRL+ALT+DEL din driverul de tastatură.

Mijloacele de control al accesului permit proprietarului unui fișier sau obiect să spună cine îl poate utiliza și în ce mod.

Mijloacele de control al accesului privilegiat permit administratorului să treacă de aceste restricții când este necesar.

Protecția spațiului de adrese înseamnă că fiecare proces utilizator are propriul său spațiu virtual de adrese protejat și inaccesibil pentru alte procese neautorizate.

Paginile noi trebuie umplute cu zerouri pentru ca procesul curent să nu poată găsi informații vechi puse acolo de proprietarul anterior.

Înregistrările de securitate permit administratorului de sistem să producă un jurnal cu anumite evenimente legate de securitate.

##### 9.3.4.1. Concepte fundamentale de securitate în sistemele Windows

Fiecare utilizator este identificat de un SID (Security ID). Acestea sunt numere binare cu antet scurt urmat de o componentă aleatoare mare. Când un utilizator lansează un proces în execuție, procesul și firele sale de execuție rulează cu SID-ul utilizatorului. Cea mai mare parte a securității sistemului este proiectată pentru a se asigura că fiecare obiect este accesat doar de firele de execuție cu SID-urile autorizate.

Fiecare proces are un jeton de acces care specifică SID-ul propriu și alte caracteristici. Este valid și asigurat în momentul înregistrării de *winlogon*.

antet	Timp de expirare	Grupuri	DAGL implicit	SID utilizator	SID grup	SID-uri restricționate	Privilegii

**Fig. 9.5. Structura unui jeton de acces Windows.**

Timpul de expirare indică momentul în care jetonul nu mai este valid dar actualmente nu este utilizat.

Grupuri specifică grupurile de care aparține procesul (Discretionary Acces Control List).

DAGL implicit este lista de control al accesului, discretă, asigurată, pentru obiectele create de proces.

SID utilizator indică proprietarul procesului.

SID-uri restrictionate este un câmp care permite proceselor care nu prezintă încredere să participe la lucrări cu alte procese demne de încredere, dar mai puțin distructive.

Lista de privilegii oferă procesului împunericiri speciale cum ar fi dreptul de a opri calculatorul sau de a accesa fișiere pe care în mod normal nu ar putea să le acceseze.

În concluzie jetonul de acces indică proprietarul procesului și ce setări implicate și împunericiri sunt asociate cu el.

Când un utilizator se înregistrează, winlogon oferă procesului inițial un jeton de acces. De obicei, procesele ulterioare moștenesc acest jeton de-a lungul ierarhiei. Jetonul de acces al unui proces se aplică inițial tuturor firelor de execuție ale procesului. Totuși un fir de execuție poate obține un jeton de acces de-a lungul execuției. Un fir de execuție client poate să-și paseze jetonul de acces unui fir de execuție server pentru a-i permite serverului să-i acceseze fișierele protejate și alte obiecte. Un astfel de mecanism se numește impersonalizare.

Descriptorul de securitate este un alt concept utilizat de Windows. Fiecare obiect are un descriptor de securitate care indică cine poate efectua operații asupra lui.

Un descriptor de securitate este format din:

-antet;

-DACL (Discretionary Acces Control List)

-unul sau mai multe ACE (Acces Control Elements).

Cele două tipuri principale de elemente sunt *Permite* și *Refuză*. Un element Permite specifică un SID și o hartă de biți care specifică ce operații pot face procesele cu respectivul SID asupra obiectului. Un element Refuză funcționează după același principiu, doar că o potrivire înseamnă că apelantul nu poate efectua operația respectivă.

În structura unui descriptor de securitate mai există și SACL (System Acces Control List) care este asemănătoare cu DACL numai că nu specifică cine poate utiliza obiectul ci specifică operațiile asupra obiectului păstrate în jurnalul de securitate al întregului sistem. În exemplul nostru fiecare operație executată de George asupra fișierului va fi înregistrată.

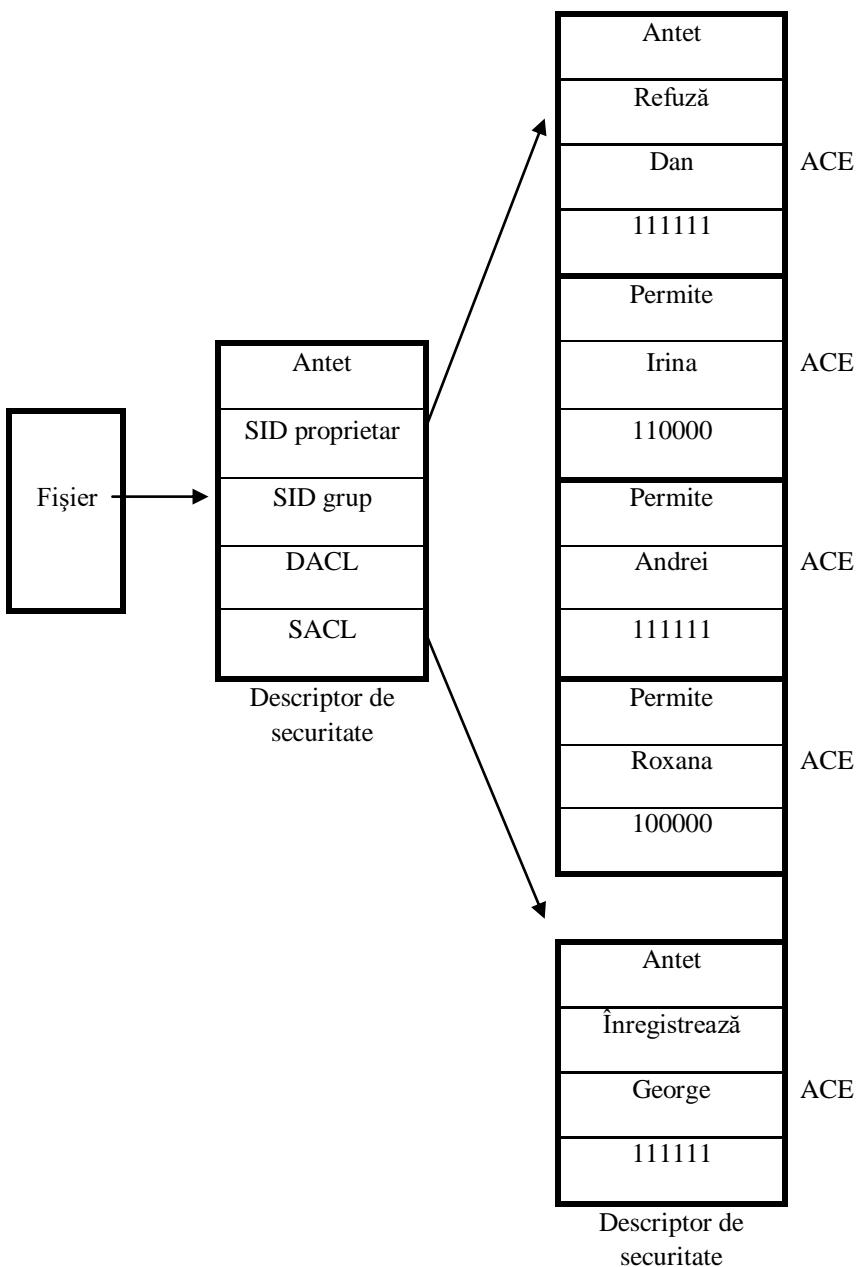


Fig. 9.6. Descriptoare de securitate Windows

#### 9.3.4.2. Implementarea securității

Pentru a implementa descriptorul de securitate prezentat anterior, se folosesc multe din funcțiile pe care Windows NT le are din Win32API, legate de securitate, marea majoritate gestionând descriptorii. Iată etapele utilizate în gestionarea acestor descriptori:

- se alocă spațiul descriptorului;
- se inițializează descriptorul cu funcția Initialize Security Descriptor, funcție care scrie și antetul;
- se caută apoi proprietarul SID-ului și al grupului și dacă ei nu există se utilizează funcția Look Account Sid ;

-se initializează DACL-ul sau SACL-ul descriptorului de securitate cu ajutorul funcției Add Access Denied Ace; aceste funcții se execută de atâtea ori câte funcții trebuie să încarcate;

-se atașează ACL-ul creat la descriptorul de securitate;

-după crearea descriptorului de securitate el poate fi pasat ca parametru pentru a fi adăugat la obiect

### 9.3.5. Securitatea în Linux

#### 9.3.5.1. Open source

Există două modalități în ceea ce privește optica producătorilor de produse soft:

-una, închisă, în care codul sursă este secret și, în general, orice informație este deținută numai de proprietar;

-una, deschisă, în care codul sursă este disponibil pentru toți utilizatorii și în care informația circulă deschis.

Din cea de-a doua categorie face parte și comunitatea **Open source**. Linux este un sistem de operare free care face parte din comunitatea **Open source**.

Să vedem ce avantaje, din punct de vedere al securității, ar oferi apartenența la această comunitate. Să încercăm să argumentăm acest lucru prin prizma bug-urilor care există în cadrul programelor și provin din erori de programare sau proiectare, introduse intenționat sau nu. Aceste buguri pot fi:

-depășirea capacitatilor tampon, care pot duce la obținerea de privilegii neautorizate de către utilizator;

-gestiunea necorespunzătoare a fișierelor, care poate permite eventualelor persoane răuvoitoare să modifice sau să deterioreze fișiere importante;

-algoritmi de autentificare sau de criptare prost proiectați, care pot conduce la accesul neautorizat al unor persoane sau decriptarea cu ușurință a unor date confidențiale, cum ar fi parolele utilizatorilor;

-caii troieni.

Securitatea sporită a programelor open source provine chiar din disponibilitatea codului sursă. Deoarece codul sursă este la dispoziția multor utilizatori se ajunge la depistarea bug-urilor cu o probabilitate mult mai mare decât la sistemele închise. Iată câteva din aceste avantaje:

-oricine poate corecta codul sursă, aceste corecții fiind făcute publice în cel mai scurt timp și incluse în următoarea sesiune;

-atunci când sunt descoperite, bug-urile sunt reparate foarte rapid; de exemplu, când a apărut bug-ul Ping O Death, în Unix crearea unui patch de corecție a durat doar câteva ore, în timp ce în Windows răspunsul a venit doar după o săptămână;

-sistemele solide de comunicație dintre membrii comunității, precum liste de discuții și resursele localizate on-line, ajută la eficientizarea procesului de dezvoltare și depanare, reducând și distribuind efortul.

#### 9.3.5.2 Programe ce depistează și corectează vulnerabilități

**Programul NESSUS.** Este un program de scanare evoluat, în sensul că, după scanarea propriu-zisă stabilește versiunile serviciilor care rulează pe mașina întă, putând determina vulnerabilitatea acesteia. NESSUS constă din două părți:

-partea de server, care efectuează practic scanările;

-partea de client care reprezintă o interfață grafică xwindows cu utilizatorul.

Serverul Nessus folosește protocolul TCP pentru a aștepta cererile de la clienți, clienți autentificați printr-un sistem de chei publice El Gamal. Traficul este criptat cu ajutorul unui cifru flux.

**Programul Titan** poate depista și corecta în mod automat vulnerabilitățile unui sistem Unix; este scris în BASH. Există trei moduri de rulare a acestui utilitar:

-modul verificare, recomandat a se folosi prima dată, cu rol de a testa sistemul fără să facă nici o modificare asupra sa;

-modul informal, care testează sistemul și afișează modificările pe care le-ar efectua în mod normal;

-modul reparare care va modifica fișierele de configurare a sistemului în vederea securizării lui.

Titan jurnalizează problemele descoperite și acțiunile produse într-un director. Iată câteva vulnerabilități pe care Titan le poate depista și corecta:

-bug-urile din protocolul TCP, având protecție împotriva atacurilor de tip SYN flood, PING flood și NFS bind;

-serviciile considerate periculoase, de exemplu automount, sunt dezactivate;

-drepturile de acces ale fișierelor și catalogelor;

-conturile utilizatorilor speciali care sunt șterse sau dezactivate.

#### 9.3.5.3. Auditarea sistemului

Auditarea unui sistem constă în operațiunile de certificare că acel sistem nu a fost compromis și că se află într-o stare sigură.

Un program destinat certificării sistemului de fișiere este

#### **Programul Tripwire.**

Acest program este un sistem de monitorizare a fișierelor care poate fi utilizat pentru a asigura integritatea acestora. Sistemul este realizat de compania Tripwire, Inc.

Prima etapă a programului constă din analiza sistemului de fișiere și crearea unei baze de date a fișierelor importante, într-un moment în care acestea sunt considerate sigure. Se stabilesc acum fișierele și directoarele ce vor fi monitorizate.

Odată creată baza de date, integritatea fișierelor importante poate fi oricând verificată. Se detectează fișierele noi, cele șterse și cele modificate. Dacă aceste modificări sunt valide, baza de date va fi actualizată; în caz contrar, ele vor fi raportate administratorului sistemului.

Fișierele importante, cum ar fi baza de date și fișierele de configurație, sunt criptate cu ajutorul algoritmului El Gamal. Se utilizează două fișiere de chei care memorează câte o pereche de chei, una publică și una privată. Fișierul global este utilizat pentru a proteja fișierele de configurare iar fișierul local de chei este folosit pentru protecția bazelor de date și fișierelor raport.

## 10. SISTEMUL DE OPERARE LINUX. APLICAȚII

### 10.1. SCURT ISTORIC

Anul 1965 poate fi considerat ca punct de pornire pentru sistemul de operare UNIX. Atunci a fost lansat la MIT (Massachusetts Institute of Technology) un proiect de cercetare pentru crearea unui sistem de operare multiutilizator, interactiv, punându-se la dispoziție putere de calcul și spațiu de memorie în cantitate mare. Acest sistem s-a numit MULTICS. Din el, Ken Thompson a creat în 1971 prima versiune UNIX. De atunci UNIX-ul a cunoscut o dezvoltare puternică în noi versiuni.

In 1987, profesorul Andrew Tanenbaum creează, la Vrije Universiteit din Amsterdam, un sistem de operare , asemănător cu UNIX-ul, cu scop didactic, pentru dezvoltare de aplicații, numit MINIX.

Pornind de la MINIX, un tânăr student finlandez, Linus Torvalds, a realizat în anul 1991 un sistem de operare mai complex, numit LINUX. Datorită faptului că a fost încă de la început „free”, LINUX-ul a cunoscut o puternică dezvoltare, fiind sprijinită de majoritatea firmelor mari, cu excepția, bineînțeles, a MICROSOFT-ului.

În anul 1998 s-a format Free Standards Group în cadrul căruia există inițiative de standardizare pentru a încuraja dezvoltarea de programe și aplicații în LINUX. Ceea ce a lipsit multă vreme LINUX-ului a fost un standard pentru ierarhia din sistemul de fișiere al programelor și aplicațiilor dar mai ales al bibliotecilor.

Lucrul cel mai important în LINUX este faptul că întregul sistem de operare și o serie întreagă de operații sunt puse la dispoziție, inclusiv sursele. Multe universități și facultăți de specialitate își dezvoltă propriile aplicații în LINUX.

### 10.2. DISTRIBUȚII ÎN LINUX

LINUX-ul este un sistem distribuit. Dintre cele trei specii de sisteme cu procesoare multiple și anume: multiprocesoare, multicompunere și sisteme distribuite, sistemele distribuite sunt sisteme slab cuplate, fiecare din noduri fiind un computer complet, cu un set complet de periferice și propriul sistem de operare.

Există la ora actuală peste 300 de distribuții în LINUX. Vom trece în revistă cele mai importante și mai utilizate distribuții.

#### 10.2.1. Distribuția SLACWARE

Creatorul acestei distribuții este Patrick Volkerding, prima versiune fiind lansată de acesta în aprilie 1993. Această distribuție are două priorități de bază: ușurința folosirii și stabilitatea.

Instalarea se face în mod text, existând posibilitatea alegerii unui anumit KERNEL la bootare, deși cel implicat este suficient, în general, pentru o instalare normală.

Formatul standard al pachetelor SLACKWARE este .tgz și există o serie de programe de instalarea, upgradarea și ștergerea pachetelor, cum ar fi *install pkg*, *upgrade pkgrd*, care nu pot fi folosite decât direct din linia de comandă.

#### 10.2.2. Distribuția REDHAT

Este una din cele mai răspândite distribuții. Conține un sistem de pachete RPM, sistem ce întreține o bază de date cu fișierele și pachetele din care provin.

Versiunea REDHAT conține codul sursă complet al sistemului de operare și al tuturor utilitarelor, cea mai mare parte a acestuia fiind scrisă în limbajul C.

La instalarea sistemului nu se permit decât partii tip ext 2 și ext 3. Există module de kernel și sisteme jurnalizate, ca JFS sau Reiser FS, care pot fi montate și după instalare. La instalare se poate configura un sistem FireWall cu trei variante de securitate, High, Medium și N.FireWall.

Există o sută de aplicații destinate configurării în mod grafic a sistemului, aplicații specifice atât interfeței GNOME cât și KDE.

#### **10.2.3. Distribuția DEBIAN**

Este considerată cea mai sigură dintre toate distribuțiile existente. De această distribuție se ocupă o comunitate restrânsă de dezvoltatori, pachetele fiind foarte bine studiate și testate înainte de a fi lansate ca *release-uri* într-o distribuție. De obicei, aplicațiile și programele sunt cu câteva versiuni în urma celorlalte distribuții. Nu există, în general, o activitate comandă de promovare a acestei distribuții, singurul suport fiind asigurat de o listă de discuții.

Procedura de instalare este una din cele mai dificile. Există două posibilități de instalare:

- a) simplă;
- b) avansată.

În varianta simplă se aleg taskuri de instalare, fiecare task presupunând instalarea mai multor pachete reunite de acesta.

În cazul unei variante avansate, există posibilitatea alegerii efective a fiecărui pachet în parte.

Meritul de bază al acestei distribuții este acela că este cea mai apropiată de comunitatea Open Source. Pachetele sunt de tipul *deb* și sunt administrate cu ajutorul unui utilitar numit ART(Advanced Package Tool).

#### **10.2.4. Distribuția MANDRAKE**

Este o distribuție care pune bază pe o mare internaționalizare, fiind disponibile versiuni de instalare în 40 de limbi. Există și o distribuție în limba română, în două variante de manipulare a tastaturii, qwerty și qwertz.

MANDRAKE are un sistem de gestiune dual:

- a) Linux Conf;
- b) Drak Conf.

Linus Conf reprezintă moștenirea de la RedHat și permite modificarea setărilor obișnuite ale sistemului: gestiunea utilizatorilor, a serviciilor (dns, mail, rtp), configurarea plăcii de rețea.

Drak Conf este un utilitar care se dorește a fi pentru LINUX ceea ce Control Panel reprezintă pentru WINDOWS. El permite configurarea sistemului de la partea hardware ce folosește Hard Drake (plăci de rețea, video, sunet, tuner) până la cele mai utilizate servicii (web rtp, dns, samba, NIS, firewall). Au fost introduse wizard-uri pentru toate serviciile ce pot fi setate grafic.

#### **10.2.5. Distribuția LYCORIS**

Este o companie foarte Tânără care și-a propus drept scop crearea unei versiuni Linux foarte ușor manevrabilă. Instalarea acestei distribuții este cea mai simplă instalare dintre toate existente până în prezent. Nu este permisă nici un fel de selecție a pachetelor, individuale sau în grup. Procesul de instalare începe imediat după alegerea partii, rulând în fundal, iar configurațiile sistemului se realizează în paralel.

O aplicație utilă este Network Browser, realizată de Lycoris, care permite interconectarea simplă cu alte sisteme Linux sau Windows. Această versiune de Linux se adresează utilizatorilor obișnuiți cu mediile de lucru Microsoft Windows. În acest sens, desktopul conține MyLinux System și Network Browser. Configurarea sistemului se realizează prin intermediul unui Control Panel care este de fapt KDE Control Center. Managerul sistemului de fișiere, Konqueror, reunește aplicațiile executabile de către Microsoft Windows și le rulează cu ajutorul programului Wine, cu care se pot chiar instala programe Windows.

#### **10.2.6. Distribuția SUSE**

SUSE este distribuția europeană (de fapt germană) cea mai cunoscută și de succes. Se axează pe personalizarea sistemului în cât mai multe țări europene. Din păcate limba română lipsește din această distribuție.

### **10.3. APLICAȚII LINUX**

Vom prezenta în aceste aplicații o serie de programe pe baza unor apeluri sistem sau a unor funcții din LINUX. Mediul de programare este limbajul C.

Aplicațiile sunt structurate pe modelul unui laborator la disciplina Sisteme de operare, fiecare lucrare de laborator prezentând trei părți:

*1) Considerații teoretice asupra lucrării.*

*2) Desfășurarea lucrării, cu exemple de programe.*

*3) Tema pentru lucru individual, care, de obicei, propune crearea de diferite programe în contextul lucrării respective.*

#### **10.3.1. Comenzi LINUX**

*1) Considerații teoretice*

Vom prezenta aici câteva din comenzi principale ale sistemului de operare LINUX, date pe linia de comandă. Aceste comenzi reprezintă interfețe între utilizator și sistemul de operare și ele sunt de fapt programe ce se lansează în execuție cu ajutorul unui program numit *interpreter de comenzi* sau, în terminologia UNIX, numit *shell*.

*2) Desfășurarea lucrării*

a) Comenzi pentru operații asupra proceselor.

Listarea proceselor active în sistem.

Comanda ps (process status) furnizează informații detaliate, în funcție de opțiunile afișate, despre procesele care aparțin utilizatorului.

Comanda \$ps, cu următorul exemplu de răspuns:

<u>pid</u>	<u>TTY</u>	<u>STAT</u>	<u>TIME</u>	<u>COMMAND</u>
3260	p3	R	0:00	bash
3452	p4	W	1:21	ps
4509	p3	Z	5:35	ps
5120	p9	S	8:55	bash

Prima coloană (pid) reprezintă identificatorul procesului.

A doua coloană (TTY) reprezintă terminalul de control la care este conectat procesul. Pot fi și adrese de ferestre sau terminale virtuale, cum este și în exemplul de față ( valorile p3, p4, p9 sunt adrese de terminale virtuale).

A treia coloană reprezintă starea procesului.

R(Running)	=	în execuție
S(Sleeping)	=	adormit pentru mai puțin de 20 secunde
I(Idle)	=	inactiv, adormit pentru mai mult de 20 secunde
W(Swapped out)	=	scos afară din memoria principală și trecut pe hard disc
Z(Zombie)	=	terminat și așteaptă ca părintele să se termine
N(Nice)	=	proces cu prioritate redusă

A patra coloană (TIME) indică timpul de procesor folosit de proces până în prezent.

A cincia coloană (COMMAND) listează numele programului executat de fiecare proces.

Dintre opțiunile acestei comenzi amintim:

Opțiunea \$ps -u, cu următorul răspuns:

USER	pid	%CPU	MEM	VSZ	RSS	TTY	STA T	START	CMD.
500	3565	0.0	1.4	4848	1336	pts/0	S	19:15	bash
500	3597	0.0	0.7	3824	688	pts/0	R	19:37	ps-u

unde câmpurile noi reprezintă:

USER	=	Numele proprietarului fișierului
%CPU	=	Utilizarea procesorului de către proces
%MEM	=	Procente de memorie reală folosite de proces
START	=	Ora la care procesul a fost creat
RSS	=	Dimensiunea reală în memoria procesului(kB)

Opțiunea \$ps -l , cu următorul răspuns:

F	S	UID	PID	PPID	C	PPI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	500	3565	3563	0	76	0	-	1212	-	pts/0	00:00:00	bash
0	R	500	3599	3565	0	78	0	-	1196	-	pts/0	00:00:00	ps

unde câmpurile noi reprezintă:

UID	=	Identifierul numeri al proprietarului procesului
F	=	Fanioane care indică tipul de operații executate de proces
PPID	=	Identifierul procesului părinte
NI	=	Incrementul de planificare al proceului
SZ	=	Dimensiunea segmentelor de date al stivei
WCHAN	=	Evenimentul pe care procesul îl așteaptă

Opțiunea \$ps -e\_determină ca la fiecare comandă să se afișeze atât argumentele cât și ambianța de execuție.

Opțiunea \$ps -a\_determină afișarea de informații și despre procesele altor utilizatori momentanii conectați în sistem.

#### Listarea activităților diversilor utilizatori

Comanda \$W, cu următorul răspuns:

<u>USER</u>	<u>TTY</u>	<u>FROM</u>	<u>LOGING</u>	<u>IDLE</u>	<u>JLPU</u>	<u>PCPU</u>	<u>WHAT</u>
Ion	0	-	7:13pm	?	0,00s	0,32s	jusr/bin/gnome
Ion	pts/0	0:0	7:15pm	0,00s	0,06s	0,01s	W

#### Listarea dinamică a proceselor din sistem

Comanda \$top cu care se poate avea o imagine dinamică a proceselor din sistem și nu o imagine statică ca la comanda W. Perioada de actualizare este implicit de 5 secunde.

b) Comenzi pentru operații generale asupra fișierelor și cataloagelor:

Comanda \$pwd , pentru afișarea numelui catalogului curent.

Comanda \$ls , pentru afișarea conținutului unui catalog ; este echivalentă cu comanda DIR din MS-DOS. Cele mai des utilizate opțiuni sunt \$ls -l , \$ls -al, \$ls -li bin.

Comanda \$cd , pentru schimbarea catalogului curent; nume catalog.

Comanda \$rm , pentru ștergerea unei intrări în catalog; nume catalog.

Comanda \$cat , pentru listarea conținutului unui fișier; nume fișier, cu opțiunile cele mai frecvente \$cat -n , (afișează numărul de ordine la fiecare linie din text) și \$cat -v (afișează și caracterele netipăribile).

Comanda \$cp , pentru copierea unui fișier; nume1, nume2.

Comanda mv, redenumirea unui fișier; sursă destinație.

#### 3) Temă

Să se realizeze toate comenziile prezentate în această lucrare, folosind un utilitar al LINUX-ului, de preferință mc.

### 10.3.2. Crearea proceselor

#### 1) Considerații teoretice

Pentru crearea proceselor LINUX folosește apelul sistem fork(). Ca urmare a acestui apel un proces părinte creează un proces fiu. Funcția fork() returnează o valoare după cum urmează:

- 1, dacă operația nu s-a putut efectua, deci eroare;
- 0, în codul FIULUI;
- PID FIU, în codul părintelui.

În urma unui fork procesul fiu, nou creat, va moșteni de la părinte atât codul cât și segmentele de date și stiva. În ceea ce privește sincronizarea părintelui cu fiul, nu se poate spune care se va executa mai întâi, fiul sau părintele. Se impun două probleme majore:

- a) sincronizarea părintelui cu fiul;
- b) posibilitatea ca fiul să execute alt cod decât părintele.

Pentru a rezolva aceste două situații se folosesc două apeluri: wait() și exec().

Funcția wait() rezolvă sincronizarea fiului cu părintele. Este utilizată pentru așteptarea terminării fiului de către părinte. Dacă punem un wait() în cadrul părintelui, se execută întâi fiul și apoi părintele. Există două apeluri:

```
pid_t wait(int*status)
pid_t waitpid(pid_t pid, int*status, int flags)
```

Prima formă wait() este folosită pentru așteptarea terminării fiului și preluarea valorii returnate de acesta. Parametrul status este utilizat pentru evaluarea valorii returnate, cu ajutorul câtorva macro-uri definite special.

Funcția waitpid() folosită într-un proces va aștepta un alt proces cu un pid dat.

Funcția exec() are rolul de a face ca fiul să execute alt cod decât părintele. Există mai multe forme ale acestei funcții : execvp, execle, execvl, execlp, execvp. De exemplu, pentru execvp avem sintaxa:

```
int execvp(const char*filename,const* char arg)
```

Prin această funcție fiul va executa, chiar de la creare fișierul cu nume filename.

Deoarece este prima aplicație cu programe scrise și executate în limbajul C, prezentăm mai jos etapele lansării unui program în C, în sistemul de operare LINUX.

-Se editează un fișier sursă în limbajul C, utilizând un editor de texte, de exemplu vi, kate sau mc. Se numește fișierul, de exemplu nume fișier.C

-Se compilează fișierul editat în C cu comanda:

```
$gcc numefișier.c
```

-Se execută fișierul rezultat în urma compilării.

```
$ ./a.out
```

#### 2) Desfășurarea lucrării

Se vor executa următoarele programe:

-Programul a

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
#include<sys/types.h>

main()
{
    int pid,status,i;

    if((pid=fork())<0) {printf("EROARE");
                           exit(0);}

    if(pid==0) {printf("am pornit copilul\n");
                 for(i=1;i<=25;i++)
                 {
                     fflush(stdout);
                     printf("0");
                 }
                 exit(1);}

    else {wait(&status);
          printf("am pornit părintele\n");
          for(i=1;i<=25;i++)
          {
              fflush(stdout);
              printf("1");
          }}
}
```

Aceasta este o schemă posibilă de apelare a funcției fork().

Dacă în urma execuției s-a terminat cu eroare, se va afișa „EROARE”. Dacă suntem în procesul fiu (pid ==0), atunci codul fiului înseamnă scrierea a 25 cifre de 0, iar dacă suntem în procesul părinte, se vor scrie 25 cifre de 1. Deoarece avem apelul wait în cadrul părintelui, întâi se va executa fiul și apoi părintele. Pe ecran se va afișa:

am pornit copilul 00000000000000000000000000000000

am pornit părintele 11111111111111111111111111111111

Rolul fflush(stdout) este de a scrie pe ecran, imediat ce bufferul stdout are un caracter în el. Fără fflush(stdout), momentul scrierii pe ecran este atunci când stdout este plin. În acest caz trebuie să stim când se afișează pe ecran ceva. Dacă acest program nu ar conține wait , atunci nu s-ar ști cine se execută primul și cine al doilea, fiul și părintele lucrând în paralel. În programul nostru, când fiul tipărește numai 0 iar părintele numai 1, ar trebui ca, la o rulare, să avem la printare o secvență de 0 ,1 amestecată.( Ex: 0011101010110...). Dar, dacă rulăm de multe ori acest program, constatăm că de fiecare dată el va tipări mai întâi 25 de 0 și apoi 25 de 1, ca și cum a existat wait-ul. Care este explicația? Ea trebuie căutată în modul de rulare, în *time sharing*, în funcție de cuanta de timp alocată fiecărui proces. Oricum ea este foarte mare în raport cu duratele proceselor părinte și fiu

din exemplul nostru. De aceea, procesele fiu și părinte nu vor fi întrerupte din rularea în procesor, deoarece ele se termină amândouă într-o singură cantă.

-Programul b

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
#include<sys/types.h>

main()
{
    int pid,status,i;
    if(pid=fork())<0) {printf("EROARE");
        exit(0)}
    if(pid==0) {printf("am pornit copilul\n");
        execl("./sorin1","./sorin1",NULL);
        exit(1);}
    else {wait(&status);
        printf("am pornit părintele\n");
        for (i=1;i<=25;i++)
            {fflush(stdout);
            printf("1");}}}
```

unde sorin1 este un fisier executabil ce se obtine prin compilarea fisierului sursă sorin1c astfel:

```
$gcc -o sorin1 sorin1c
```

Fisierul sorin 1c are continutul:

```
#include<stdio.h>
```

```
main()
{int i;
```

```
for(i=1;i<=25;1++)  
    printf("0");{
```

Se observă că în acest program în codul fiului avem funcția execclp care va înlocui codul părintelui prin codul dat de fișierul executabil sorin 1 care, de fapt, realizează același lucru ca la programul anterior. Rezultatul rulării va fi:

000000000000000000000000000000

am pornit părintele | | | | | | | | | | | | | | | | | | | |

Față de rezultatul de la programul anterior, nu se mai execută printf("am pornit copilul"). Din ce cauză? Pentru că atunci când se execută fiul, printarea "am pornit copilul" este trecută în bufferul stdout, însă când se execută execvp, bufferul se golește și se execută strict numai fișierul sorin1.

### Programul c

```
#unclude<stdlib.h>
#include<sys/types.h>
#include<unistd.h>

main()
{
pid_t copil_pid;
copil_pid=fork();
if(copil_pid>0)
/*suntem în părinte și vom dormi 120 secunde*/
sleep(120);
if(copil_pid== -1) {printf("EROARE");
exit(0);}
else {printf("pid copil%d\n",
getpid());
exit(1)
/*suntem în copil și ieșim imediat*/}}
}
```

În acest program fiul își termină execuția imediat iar părintele este întârziat cu 120 secunde. În acest timp fiul intră în starea zombie, până când procesul init(), care preia copiii orfani, va prelua rolul de părinte al fiului și îl va scoate din starea zombie, eliberând tabela proceselor. Programul se compilează cu:

```
$gcc -o zombie numefișier.c
```

și se execută astfel :

```
$ ./zombie &
```

Dacă se lansează comanda :

```
$ps -a
```

se poate observa cât timp stă fiul în starea zombie, până îl preia procesul init(),

3) Temă

Să se scrie un program care să creeze 10 procese. Fiecare proces va scrie o linie numai cu cifre; procesul 0 va scrie o linie de 0, procesul 1 va scrie o linie numai de 1..... procesul 9 va scrie o linie numai de 9. Liniile vor trebui scrise în ordinea cifrelor, deci prima linie de 0 și ultima de 9.

### 10.3.3. Comunicare între procese

#### 10.3.3.1. Comunicarea între procese prin PIPE-uri și FIFO

##### 1) Considerații teoretice

a) Pipe-ul este un pseudofișier care servește la comunicarea unidirecțională între două procese. Faptul că este unidirecțional a fost considerat ulterior ca una dintre limitele mecanismului și de aceea unele versiuni actuale au înlocuit pipe-ul unidirecțional prin cel bidirecțional. Astfel la SOLARIS pipe-urile sunt bidirecționale dar în Linux sunt unidirecționale. Aici, deci, vom considera pipe-urile unidirecționale.

O altă caracteristică a pipe-ului este faptul că procesele care comunică între ele trebuie să aibă un grad de rudenie, de exemplu tată-fiu.

Un pipe este creat cu apelul:

```
int pipe (int file des[2]);
```

care creează în kernel un pipe accesibil în procesul apelant prin doi descriptori:

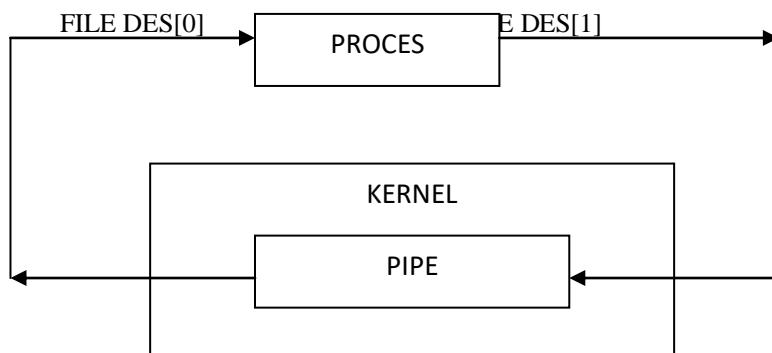
file des[0] deschis în citire

file des [1] deschis în scriere

În urma apelului pipe există două returnări:

0 , în caz de succes,

-1, în caz de eroare.



**Fig. 10.1. Schema unui pipe unidirecțional.**

Marea majoritate a aplicațiilor care utilizează pipe-urile închid, în fiecare dintre procese, capătul de pipe neutilizat în comunicarea unidirecțională. Astfel, dacă pipe-ul este utilizat pentru comunicația părinte-fiu, atunci procesul părinte scrie în pipe iar procesul fiu citește din pipe.

b) FIFO (pipe-uri cu nume)

Restricția ca procesele care comunică să fie înrudite este eliminată la pipe-urile cu nume care sunt fișiere de tip FIFO.

Apelul sistem pentru crearea unui FIFO este:

```
#include<sys/types.h>
#include<sys/stat.h>
int mknod(const char* pathname, mode_t mode);
```

Cele două argumente ale apelului sunt:

```
char* pathname, ( numele FIFO-ului)
mode_t mode, (apare numai la deschidere, are loc și
crearea fișierului).
```

După ce FIFO a fost creat, î se pot ataşa toate operațiile tipice cu fișiere (open, read, write, unlink etc.).

## *2) Desfăşurarea lucrării*

### Programul a

Acesta este un program care transmite un text ("text prin pipe") prin pipe de la procesul părinte la procesul fiu.

```
#include<unistd.h>
#include<sys/types.h>
#define MAXLINE 500
main()
{int n, fd[2];
pid_t pid;
char line [MAXLINE];
if(pipe(fd)<0) {printf("eroare pipe");
exit(0);}
if((pid=fork())<0) { printf("eroare fork")
exit(1);}
else
if(pid>0) /*părinte*/
close(fd[0]);
```

```
    write(fd[1],"text prin pipe",14);  
  
eose    {close(fd[1]);/*fiu*/  
  
n=read(fd[0],line, MAXLINE);  
  
write(1,line,n);}  
  
exit(0);}
```

### Program b

Programul folosește două pipe-uri, pipe1 și pipe2, încercând să simuleze un pipe bidirectional. Se va observa că, de fiecare dată când se execută o operație la un capăt al pipe-ului, celălalt capăt este închis.

```
#include<stdio.h>  
  
main()  
  
{int child pid,pipe1[2],pipe2[2];  
  
if(pipe(pipe1)<0!! pipe(pipe2)<0  
  
    perror("nu pot crea pipe");  
  
if((childpid=fork())<0  
  
    perror("nu pot crea fork");  
  
else  
  
if(childpid>0) /*părinte*/  
  
{close(pipe1[0]);  
  
close(pipe2[1]);  
  
client(pipe2[0],pipe[1]);  
  
while(wait((int*)0)!=childpid  
  
    close(pipe1[1]);  
  
    close(pipe2[0]); exit(0);}  
  
else  
  
{/*copil*/  
  
close(pipe1[1]);close(pipe2[0]);  
  
server(pipe1[0],pipe2[1]);  
  
close(pipe1[0]);close(pipe2[1]); exit(0);}}
```

Program c

Acest program crează un proces copil care citește dintr-un pipe un set de caractere trimis de procesul părinte, convertind orice literă mică într-o literă mare. Procesul părinte citește sirul de caractere de la intrarea standard.

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

int piped[2]; /*pipe-ul*/
int pid; /*pid-ul fiului*/
int c; /*caracterele citite*/

main()
{
if(pipe(piped)<0) /*se crează pipe-ul 1*/
{perror("eroare pipe");
exit(0);}

if((pid=fork())<0) /*creare proces fiu*/
{perror("eroare fork"); exit(1);}

if (pid) /*sunt în procesul
părinte*/
{
close(piped[0]); /*închiderea descriptorului
de citire*/
printf("proces părinte:introduceți sirul de
caractere");
/*se citește de la intrarea
standard*/
while(read(0,&c,1))/*se scriu datele în
pipe*/
if(write(piped[1],&c,1)<0)
}
}
```

```

{perror("eroare scriere");exit(2);}

/*se închide descriptorul

de scriere în pipe*/

close(piped[1]);

if (wait(NULL)<0)

{perror("eroare wait");exit(3);}

exit(4);

else

{

/*sunt în fiu*/

close(piped[1]); /*se închide descriptorul

de scriere*/

printf("proces datele\n");

while(read(piped[0],&c,1))

{if(islower(c) /*este litera mică?*/

printf("%c",toupper(c));

else

printf("%c",c);}

close(piped[0]); /*se închide descriptorul

de citire*/

exit(0); }

```

### Program d

În acest program se citesc date dintr-un fișier existent, specificat ca prim argument linie de text, și se trimit prin pipe comenzi sort ce are ieșire redirectată într-un fișier specificat ca argumentul al doilea.

```

#include<stdio.h>

#include<unistd.h>

int main(int argc,char*argv[])

{char buf[81];

char command[80];

FILE*ip;

FILE*pp;

```

```

Spr int f(command,"sort>%s",argv[2]);
ip=fopen(argv[1],"r");
pp=open(command,"w");
while(fgets(buf,80,ip))fputs(bif,pp);
pclose(pp);
fclose(ip);
exit(0);

```

Programul se compilează cu

```

$gcc -o pope pope.c
şi se lansează
$./ pope fişierintrare fişierieseire

```

### Program e

Acest program este asemănător cu programul **d** numai că de data aceasta nu mai comunică părintele cu fiul prin pipe ci două procese prin FIFO.

```

#include<stdio.h>
#include<stdlib.h>
#define FIFO-FILE"MY FILO"
int main(int argc,char*argv[])
{FILE*fp;
if(argc!=2){printf("utilizare:fifoc[sir]\n");
exit(1);}
if((fp=fopen(FIFO-FILE,"w"))==NULL)
{perror("fopen"); exit(1);}
fputs(argv[1],fp)
felse(fp);
return 0;}

```

Programul se compilează cu

```

$gcc/o fifoc
şi se lansează în execuție serverul cu

```

\$ ./fifos

Apoi, dacă s-a lansat serverul în background, se dă umătoarea comandă de la același terminal. Dacă s-a lansat serverul în foreground, se dă comanda de la alt terminal.

\$ ./fifoc ”sir de curățire afișat la server”

3) Temă

a) Să se creează un pipe prin care procesul părinte va trimite procesului fiu numai ”numerele rotunde” dintre cele citite de părinte de la tastatură. (Un număr rotund este numărul care, în baza 2, are numărul de cifre 0 egal cu numărul de cifre 1.

b) Să se creeze un FIFO prin care un proces va trimite altui proces numai numere multiple de 11 dintre cele citite de la tastatură de primul proces.

#### **10.3.3.2. Comunicarea între proceze prin semnale**

I) Considerații teoretice

Semnalele reprezintă unul dintre primele mecanisme de comunicare între proceze. Ele anunță apariția unui eveniment. Semnalele pot fi trimise de către un proces altui proces sau pot fi trimise de către kernel. Momentul apariției unui semnal este neprecizat el apărând asincron.

Un semnal este reprezentat printr-un număr și un nume care se poate vedea prin lansarea comenzii

\$kill -1

Se pot trimite semnale:

-cu comanda kill,

-în program cu apelul sistem kill(),

-cu anumite combinații de chei de la tastatură,

-când se îndeplinesc anumite condiții: de exemplu eroare de virgulă mobilă (SIGFPE) sau referirea unei adrese din afara spațiului unui proces (SIGSEGV)

-sau prin kernel care poate semnaliza, de exemplu, prin SIGURG apariția aut-of-band pe un socket.

#### Functia kill()

Trimite un semnal unui proces sau unui grup de proceze.

int kill(pid\_t pid,int sig);

Pentru ca un proces să poată trimite un semnal procesului identificat prin pid, trebuie ca user ID-ul real sau efectiv al procesului care trimite semnalul să se potrivească cu ID-ul real sau set-user-ID salvat al procesului care recepționează semnalul.

-Dacă pid>0 semnalul se trimite tuturor procesului pid;

-dacă pid==0 semnalul se trimite tuturor proceselor care fac parte din același grup de proceze cu procesul care trimite semnalul, dacă există permisiunile necesare;

-dacă pid== -1 semnalul se trimit tuturor proceselor (cu excepția unui set nespecificat de procese sistem), dacă există permisiunile necesare;

-dacă pid<0&&pid!= -1, semnalul se trimit tuturor proceselor care fac parte din grupul de procese al cărui pgid este egal cu modulul valorii primului argument, dacă există permisiunile necesare;

-dacă al doilea argument este 0, nu se trimit nici un semnal; se testează existența procesului specificat în primul argument.

Există două moduri de lucru cu semnalele:

- A) folosind standardul inițial (stil vechi, nerecomandat);
- B) folosind noul stil.

În ambele situații, pentru procesul care recepționează un semnal, putem seta trei tipuri de acțiuni:

-acțiunea implicită, reprezentată prin pointerul la funcție SIG\_DEL;

-acțiunea de a ignora semnalul recepționat, reprezentată prin pointerul la funcție SIG\_IGN;

-acțiunea precizată printr-o funcție, numită handler, reprezentată printr-un pointer la funcție (numele funcției este adresa ei).

#### A) Standardul vechi

În vechiul stil, pentru a seta o acțiune corespunzătoare unui semnal foloseam funcția signal() al cărui prototip era:

```
void(*signal(int sig,void(*handler)(int)))(int);
```

glibc folosește pentru handlere tipul sig\_t.

Mai există extensia GNU:

```
typedef void(*sighandler_t handler)(int);
sighandler_t signal(int signum,
sighandler_t handler);
```

#### B) Standardul Posix

Cele trei cazuri rămân și aici valabile.

Putem să specificăm un handler pentru semnal când acțiunea este de tipul captare de semnal:

```
void handler(int signo);
```

Două semnale nu pot fi captate (nu putem scrie handlere pentru ele): SIGKILL și SIGSTOP.

Putem ignora un semnal prin setarea acțiunii la SIG\_IN. Pentru SIGKILL și SIGSTOP nu se poate acest lucru.

Putem seta o acțiune implicită prin folosirea lui SIG\_IN. Acțiunea implicită înseamnă, pentru majoritatea semnalelor, terminarea unui proces. Două semnale au acțiunea implicită să fie ignorate: SIGCHLD ce este trimis părintelui când un copil a terminat și SIGURG la sosirea unor date aut-of-band.

#### Funcția sigaction()

Pentru a seta acțiunea corespunzătoare unui semnal, în loc de funcția signal() vom folosi funcția sigaction(). Pentru aceasta trebuie să alocăm o structură de tipul sigaction:

```
typedef void(*sighandler_t)(int signo)
struct sigaction
{
    sighandler_t sa_handler; /*pointer la o funcție
        de captare semnal sau SIG_IGN sau SIG_DEF*/
    sigset_t sa_mask; /*setul de semnale blocate în
        timpul execuției handlerului*/
    unsiged long sa_flags; /*flaguri speciale*/
    void(*sa_restorer)(void); /*pointer la o funcție captare de semnal*/
};
```

Câteva flaguri:

**SA\_NOCLDSTOP** - un semnal SIGCHLD este trimis părintelui unui proces când un copil de-al său a terminat sau e opriit. Dacă specificăm acest flag, semnalul SIGCHLD va fi trimis numai la terminarea unui proces copil.

**SA\_ONESHOT** - imediat ce handlerul pentru acest semnal este rulat, kernelul va resetă acțiunea pentru acest semnal la SIG\_DEL.

**SA\_RESTART** – apelurile sistem ”lente” care returnau cu eroarea EINTR vor fi restartate automat fără să mai returneze.

Prototipul funcției sigaction() este:

```
int sigaction(int signum, struct sigaction act,
             struct sigaction oact);
```

Unde signum este semnalul a cărui livrare urmează să fie setată, prima structură sigaction act conține setările pe care kernelul le va utiliza cu privire la semnalul signum, iar a doua structură oact memorează vechile setări (pentru a fi setate ulterior); se poate specifica NULL pentru ultimul argument dacă nu ne interesează restaurarea.

#### Alte funcții utilizate

Tipul de date pe care se bazează funcțiile pe care le vom prezenta este **sig\_set** și reprezintă un set de semnale. Cu acest tip putem opera ușor o listă de semnale kernelului.

Un semnal poate aparține sau nu unui set de semnale.

Vom opera asupra unui obiect **sig\_set** numai cu ajutorul următoarelor funcții:

```
int sigempzyset(sigset_t*set);
int sigfillset(sigset_t*set);
int sigaddset(sigset_t*set,int signo);
```

```
int sigdelset(sigset_t*set,int signo);
int sigismember(const sigset_t*set, int signo)
```

Observăm că primul argument este un pointer la setul de semnale. sigempzys() scoate toate semnalele din set, iar sigfillset() adaugă toate semnalele setului. Trebuie neapărat să folosim una din cele două funcții pentru a inițializa setul de semnale. sigaddset() adaugă un semnal setului iar sigdelset() scoate un semnal din set.

Un concept important referitor la procese îl reprezintă masca de semnale corespunzătoare procesului. Aceasta precizează care semnale sunt blocate și nu vor fi livrate procesului respectiv; dacă un astfel de semnal este trimis, kernelul amâna livrarea lui până când procesul deblochează acel semnal. Pentru a modifica masca de semnale se utilizează:

```
int sigprocmask(int how,const sigset_t*modset,
sigset_t*oldset);
how poate fi:
```

SIG\_BLOCK – semnalele conținute în modset vor fi adăugate măștii curente și semnalele respective vor fi și ele blocate.

SIG\_UNBLOCK – semnalele conținute în modset vor fi scoase din masca curentă de semnale.

SIG\_SETMASK – masca de semnale va avea exact același conținut cu modset.

Când un semnal nu poate fi livrat deoarece este blocat, spunem că semnalul respectiv este în aşteptare.

Un proces poate afla care semnale sunt în aşteptare cu:

```
int sigpending(sigset_t*set);
```

În variabila set vom avea toate semnalele care aşteaptă să fie livrate dar nu sunt, deoarece sunt blocate.

Un proces poate să-și suspende execuția simultan cu schimbarea măștii de semnale pe timpul execuției acestui apel sistem prin utilizarea lui:

```
int sigsuspend(const segset_t*mask);
```

Este scos din această stare de oricare semnal a cărui acțiune este precizată printr-un handler sau a cărui acțiune este să termine procesul. În primul caz, după execuția handlerului se revine la masca de semnale de dinaintea lui sigsuspend() iar în al doilea caz (când acțiunea e să termine procesul) funcția sigsuspend() nu mai returnează. Dacă masca este specificată ca NULL, atunci va fi lăsată nemodificată.

Concluzii pentru semnalele Posix:

- un semnal instalat rămâne instalat-(vechiul stil dezactivă handlerul);
- în timpul execuției handlerului, semnalul respectiv rămâne blocat; în plus, și semnalele specificate în membrul sa\_mask al structurii sigaction sunt blocate;
- dacă un semnal este transmis de mai multe ori când semnalul este blocat, atunci va fi livrat numai odată, după ce semnalul va fi deblocat;
- semnalele sunt puse într-o coadă.

#### Semnale de timp real

Modelul de semnale implementat în UNIX în 1978 nu era sigur. În decursul timpului au fost aduse numeroase îmbunătățiri și în final s-a ajuns la un model Posix de timp real.

Vom începe cu definiția structurii sigval

```
union sigval {
    int sival_int;
    void*sival_ptr;
};
```

Semnalele pot fi împărțite în două categorii:

- 1) semnale realtime ale căror valori sunt cuprinse între SIGRTMIN și SIRTMAX (vezi cu \$kill\_1);
- 2) restul semnalelor.

Pentru a avea certitudinea comportării corecte a semnalelor de timp real va trebui să specificăm pentru membrul sa\_flags al structurii sigaction valoarea SA\_SIGINFO și să folosim unul din semnalele cuprinse între SIGRTMIN și SIGRTMAX.

Ce înseamnă semnale de timp real?

Putem enumera următoarele caracteristici:

- 1) FIFO – semnalele nu se pierd; dacă sunt generate de un număr de ori, de același număr de ori vor fi livrate;
- 2) PRIORITĂȚI – când avem mai multe semnale neblocate, între limitele SIGRTMIN și SIRTMAX, cele cu numere mai mici sunt livrate înaintea celor cu numere mari (SIGRTMIN are prioritate mai mare decât SIRTMIN+1);
- 3) Comunică mai multă informație – pentru semnalele obișnuite singurul argument pasat era numărul semnalului; cele de tipul real pot comunica mai multă informație.

Prototipul funcției handler este:

```
void func(int signo,siginfo_t*info,void*context);
```

unde signo este numărul semnalului iar structura siginfo\_t este definită:

```
typedef struct{
    int si_signo;//la fel ca la argumentul signo
    int si_code;//SI_USER,SI_QUEUE,SI_TIMER,
    SI_ASYNCIO,SI_MESSAGEQ
    union sigval si_value; /*valoare întreagă sau pointer de la emițător*/
}siginfo_t;
```

SI\_ASYNCIO înseamnă că semnalul a fost trimis la terminarea unei cereri I/O asincrone.  
 SI\_MESSAGEQ înseamnă că semnalul a fost trimis la plasarea unui mesaj într-o coadă de mesaje goale.  
 SI\_QUEUE înseamnă că mesajul a fost trimis cu funcția sigqueue().  
 SI\_TIMER semnal generat la expirarea unui timer.  
 SI\_USER semnalul a fost trimis cu funcția kill().

În afară de funcția kill() mai putem trimite semnale cu funcția sigqueue(), funcție care ne va permite să trimitem o union sigval împreună cu semnalul.

Pentru SI\_USER nu mai putem conta pe si\_value. Trebuie să și :

act.sa\_sigachon=func;//pointer k funcție handler

act.sa\_flags=SA-SIGINFO;//rest time

## *2) Desfășurarea lucrării*

### Program a

/\*Compilăm programul cu

\$gcc -o semnal semnal.c

și după lansare observăm de câte ori a fost chemat handlerul acțiune, în două cazuri:

-lăsăm programul să se termine fără să mai trimitem alte semnale;

-lăsăm programul în fundal și trimitem mai multe comenzi prin lansarea repetată:

\$kill -10 pid

Ar trebui să găsim valoarea:

10000+nr-de-killuri \*/

```
#include<signal.h>
#include<stdio.h>
#include<string.h>
#include<sys/types.h>
#include<unistd.h>
sig_atomic_t nsigusr1=0;
void acțiune(int signal_number{
```

```

++nsigusrl;}

int main(){
    int i;

    struct sigaction sa;

    printf("am pid-ul %d\n",getpid());

    memset(&sa,0,sizeof(sa));

    sa.sa_handler=actiune;

    sigaction(SIGUSR1,&sa,NULL);

/*aici efectuăm calcule....*/

    for(i=0;i<10000;i++)

    {kill(getpid(),SIGUSR1);}

    for(i=0;i<20;i++)sleep(3);

    printf("SIGUSR1 a apărut de %d ori\n",
    nsigusfl1);

    return 0;
}

```

**Program b**

```

#include<signal.h>

/*vom compila programul cu

$gcc-o ter1 ter1

și vom putea lansa programul în fundal cu

$./ter1 &

Vom citi pid-ul copilului și-i vom putea trimite un semnal

cu

$kill-10pid-copil

```

și vom vedea starea de ieșire (corespunzătoare semnalului trimis). La o altă lansare în foreground, lăsăm programul să se termine și observăm starea de ieșire. Să se remarce utilizarea handlerului curătare-copil care este instalat să trateze terminarea unui copil. Această abordare permite ca părintele să nu fie blocat într-un wait() în aşteptarea stării de ieșire a copilului\*/

```
#include<>
```

```
#include<>
```

```
#include<>

sig_atomic_t copil_stare_exit;

void curatare_copil(int signal_number){

int stare;

wait(&stare);

copil_stare_exit=stare;

}

int main(){

int i;

pid_t pid_copil;

int copil_stare

struct sigaction sa;

memset(&sa,0,sizeof(sa));

sa.sa_handler=curatare.copil;

sigaction(SIGCHLD,&sa,NULL);

/*aici lucrăm*/



pid_copil=fok();

if(pid_copil !=0){

/*suntem în parinte*/

printf("pid-ul copilului este %d\n",pid_copil);

sleep(30);//să ne asigurăm că nu terminăm înaintea copillui

}

else {

/*suntem în copil*/

sleep(15);

execlp("ls","ls","-1","/",NULL;

/*nu trebuie să ajungem aici*/
}
```

```

exit(2);

}

for(i=0;i<10000000;i++){

if(WIFEXISTED(copil_stare_exit)){

printf("copilul a ieșit normal cu starea

exit%d\n",WEXITSTATUS(copil_stare_exit));

printf("în handler de terminare am setat variabila globală la valoarea %d\n",

copil_stare_exit);}

else

printf("copilul a ieșit anormal cu semnalul

%d\n",WTERSIG(copil_stare_exit));

return 0;

}

```

### Program c

/\*Programul vrea să testeze comportarea real time a semnalelor din intervalul SIGRTMIN-SIGRTMAX.

După fork() copilul blochează receptia celor trei semnale. Părintele trimite apoi câte trei salve purtând informație (pentru a verifica ordinea la recepție), pentru fiecare semnal, începând cu semnalul cel mai puțin priorității. Apoi copilul deblochează cele trei semnale și vom putea vedea câte semnale și în ce ordine au fost primite.

```

*/
#include<signal.h>
#include<stdio.h>
#include<string.h>
#include<sys/types.h>
#include<unistd.h>

typedef void sigfunc_rt(int,siginfo_t*,void*);

static void sig_rt(int,siginfo_t*,void*);

sigfunc_rt*signal_rt(int,sigfunc_rt*,sigset*);
```

```
int main(){
    int i,j;
    pid_t pid;
    sigset_t newset;
    union sigval val;
    printf("SIGRTMIN=%d,SIGRTMAX=%d\n",
    (int)SIGRTMIN,(int)SIGRTMAX);
    if((pid=fork())==0){//în copil
        sigemptyset(&newset);//initializează set
        semnale
        sigaddset(&newset,SIGRTMIN);//adaugă un semnal
        setului
        sigaddset(&newset,SIGRTMIN+1);
        sigaddset(&newset,SIGRTMIN+2);
        sigprocmask(SIG_BLOCK,&newset,NULL);//blocare
        receptie set semnale
        signal_rt(SIGRTMIN,sig_rt,&newset);/*folosește o funcție aparte pentru setare structura sigaction și
apoi cheamă funcția sigaction()*/
        signal_rt(SIGRTMIN+1,sig_rt,/newset);
        signal_rt(SIGRTMIN+2,sig_rt,/newset)
        sleep(6);//delay pentru a lăsa părintele
        să trimită toate semnalele
        sigprocmask(SIG_UNBLOCK,&newset,
        NULL)//deblochază receptia semnalelor
        sleep(3)//lasă timp pentru livrarea semnalelor
        exit(0)
    }
    //în părinte
    sleep(3);//lasă copilul să blocheze semnalele
```

```

for(i=SIGRTMIN+2;i>=SIGRTMIN;i--){
    for(j=0;j<=2;j++){
        val.sival_int=j;
        sigqueue(pid,i,val);/*în loc de kill() folosim
această funcție pentru trimitere semnal cu
argumentele:pid_copil,nr_semnal,extra_valoare*/
        printf("am trimis semnal %d,val=%d\n",i,j);
    }
}
exit(0);
}

static void sig_rt(int signo,siginfo_t*info,
void*context){
    ptintf("recepție semnal #%d,code=%d,
ival=%d\n",signo,info->si_code,
info->si_value.sival_int);
}

/*vom seta structura sigaction și apoi vom apela funcția sigaction() pentru a comunica kernelului
comportamentul dorit*/
sigfunc_rt*signal_rt(int signo,
sigfunc_rt*func,sigset_t*mask)
{
    struct sigaction act,oact;
    act.sa_sigaction=func;
    act.sa_mask)=mask;
    act.sa_flags=SA-SIGINFO;//foarte important
    pentru a activa realtime
    if(signo==SIGALRM){

#define SA_INTERRUPT
    act.sa_flags|=SA_INTERRUPT;
}

```

```

#endifif

}else{

#endifif SA_RESTART

act.sa_flags|=SA_RESTART;

#endifif

}

if(sigaction(signo,&act,&oact)<0)

return((sigfunc.rt*)SIG_ERR);

return(oact.sa_sigaction);}

```

### 3) Temă

Să se scrie un program care șterge în 10 minute, toate fișierele temporare, (cu extensie.bak), din directorul curent.

#### 10.3.3. Comunicație între procese prin sistem V IPC.Cozi de mesaje

Mecanismele de comunicare între procese prin sistem V IPC sunt de trei tipuri:

- cozi de mesaje;
- semafoare;
- memorie partajată.

Pentru a genera cheile necesare obținerii identificatorilor, care vor fi folosite în funcțiile de control și operare, se utilizează funcția ftok():

```
#include<sys/ipc.h>

key_t ftok(const char* pathname,int id);
```

unde tipul de dată keyt\_t este definit în <sys/types.h>

Funcția folosește informații din sistemul de fișiere pornind de la pathname, numărul i-node și din LSB-ul lui id. Pathname nu trebuie șters și recreat între folosiri deoarece se poate schimba i-node-ul. Teoretic nu este garantat că folosind două pathname-uri diferite și același id vom obține două chei de 32 biți diferite. În general, se convine asupra unui pathname unic între client și server și dacă sunt necesare mai multe calale se utilizează mai multe id-uri.

#### Structura ipc-perm

O structură însoțește fiecare obiect IPC

```
struct ipc_perm{
    uid_t uid;
```

```

    gid_tgid;
    uid_t cuid;
    gid_t cgid;
    mode_t mode;
    ulong seq;
    key_t key;
}

```

uid și gid sunt id-urile proprietar, cuid și cgid sunt id-urile creator ce vor rămâne neschimbate, mode conține permisiunile read-write, seq un număr de secvență care ne asigură că nu vor fi folosite de procese diferite din întâmplare și, în final, cheia key.

Pentru a obține identificatorul care va fi utilizat în funcția de control și operare cu ajutorul funcțiilor `_get()`, putem folosi ca prim argument o valoare întoarsă de funcția `ftok()` sau valoarea specială IPC\_PRIVATE (în acest caz avem certitudinea că un obiect IPC nou și unic a fost creat, deoarece nici o combinație de pathname și id nu va genera valoarea 0, datorită faptului că numărul de i-nod este mai mare ca zero).

Se pot specifica în oflag IPC\_CREAT, câns de va crea o nouă intrare, corespunzător cheii specificate dacă nu există sau IPC\_CREAT|IPC\_EXCL când, la fel ca mai sus, se va crea o nouă cheie dacă nu există sau va returna eroarea EEXIST dacă deja există. Dacă serverul a creat obiectul, clienții pot să nu specifice nimic.

#### Permișii IPC

În momentul creerii unui obiect IPC cu una din funcțiile `_get()`, următoarele informații sunt salvate în structura `ipc_perm`:

-permisiunile dec read sau/și write pentru user, grup și alții:

0400 read de către user

0200 write de către user

0040 read de către grup

0020 write de către grup

0004 read de către alții

0002 write de către alții

-cuid și cgid (id-urile creator) sunt setați la uid-ul efectiv și gid-ul efectiv al procesului chemător (aceștia nu se pot schimba)

-uid și gid sunt setați la fel ca mai sus; se numesc uid și gid proprietar; se pot schimba prin apelul unei funcții de control `ctl( )` cu argument `IPC_SET`.

Verificarea permisiunilor se face atât la deschiderea obiectului cu `_get( )` cât și de fiecare dată când un obiect IPC este folosit. Dacă un obiect, la deschidere, are precizat pentru membrul mode să nu aibă drept de read grupul și alții și un client, și un client folosește un oflag ce include acești biți, va obține o eroare chiar la `_get( )`. Această eroare s-ar putea ocoli prin precizarea unui flag 0, dar de aceea se vor verifica la orice operație permisiunile.

Iată ordinea testelor-la prima potrivire se acordă accesul:

-superuserului i se permite accesul;

-dacă uid-ul efectiv este egal cu uid-ul sau cuid-ul obiectului IPC și dacă bitul corespunzător din membrul mode este setat, se permite accesul;

-dacă gid-ul efectiv este egal cu cu gid-ul sau cgid-ul obiectului IPC și bitul corespunzător din membrul mode al obiectului este setat, accesul este permis;

-dacă bitul corespunzător din membrul mode al obiectului IPC este setat, se permite accesul.

### Comenzi de vizualizare IPC

\$ipcs, pentru a vedea informații pentru fiecare IPC.

Pentru a șterge IPC-uri din sistem folosim:

\$ipcrm -q msg\_id (pentru cozi)

\$ipcrm -m shm\_id (pentru memorie partajată)

\$ipcrm -s sem\_id (pentru semafoare)

Există și o sintaxă cu aceleași opțiuni dar cu litere mari unde se specifică ca ultim argument cheia.

### Cozi de mesaje

Un proces cu privilegiile corespunzătoare și folosind identificatorul cozii de mesaje poate păstra mesaje în ea, după cum un proces cu privilegiile corespunzătoare poate citi mesajele. Nu este necesar (la fel ca la POSIX) ca un proces să aștepte mesaje înainte ca să plasă mesaje în coadă.

Kernelul păstrează informațiile pentru o coadă într-o structură definită în <sys/msg.h> ce conține :

```
struct msgid_ds {
    struct ipc_perm msg_perm;//permisiuni read-write
    msgqnum_t msgqnum; //nr.mesaje prezente în coadă
    msglen_t msg_qbytes ; //nr.max de bytes permisi în coadă
    pid_t msg_lspid ; //pid-ul ultimei operații msgsnd()
    pid_t msg_lrpid ; //pid-ul ultimei operații msgrcv()
    time_t msg_stime ; //timpul ultimei operații msgsnd()
    time_t msg_rtime; //timpul ultimei operații msgrcv()
    time_t msg_ctime ; /*timpul ultimei operații msgctl() ce a modificat structura*/
    La LINUX tipurile msgqnum_t, msglen_t,pid_t sunt ushort.
```

### Funcția msgget()

Are prototipul:

```
int msgget(key_t key, int msgflg);
```

O nouă coadă de mesaje este creată sau se accesează o coadă existentă. Valoarea de return este identificatorul cozii de mesaje și aceasta va fi folosită ca prim argument pentru celelalte funcții de control și operare.

Primul argument al funcției poate fi IPC\_PRIVATE sau o valoare obținută prin apelul funcției ftok(). O nouă coadă este creată dacă specificăm IPC\_PRIVATE pentru msgflg sau dacă nu specificăm IPC\_PRIVATE dar specificăm IPC\_CREAT și nici o coadă nu este asociată cu key. Altfel va fi doar referită.

Flagul IPC\_EXCL, dacă este împreună cu IPC\_CREAT (prin folosirea lui | ), face ca funcția msgget() să returneze eroarea EEXIST dacă coada există deja.

La crearea unei noi cozi se vor inițializa următorii membri ai structurii msqid\_ds :

-msg\_perm.cuid și msg\_perm.uid sunt setați la userul uid efectiv al procesului chemător;

-msg\_perm.cgid și msg\_perm.gid sunt setați la gid-ul efectiv al procesului apelant;

-cei mai puțin semnificativi 9 biți ai lui msg\_perm.mode sunt setați la valoarea celor mai puțin semnificativi 9 biți ai lui msflg.

-msg-qnum, msg-lspid, msg-lrpid, msg-stime, msg\_rtime, sunt setați la 0;

-msg\_ctime este setat la timpul curent;

-msg\_qbytes este setat la limita sistemului de operare( la Linux MSGMNB).

Alte erori returnate:

EACCES – dacă un identificator există pentru key dar procesul apelant nu are permisiunile necesare;

EIRDM – (Linux) coada este marcată pentru ștergere;

ENOENT - coada nu există și nici nu s-a specificat IPC\_CREAT;

ENOMEM - (Linux) o coadă trebuie creată dar nu există memorie pentru structura de date.

#### Functia msgsnd()

Cu ajutorul funcției:

```
int msgsnd(int msqid, const void* msgp, size_t,
           int msgflg);
```

vom trimite un mesaj în coada specificată prin identificatorul msqid. Al doilea argument este un pointer către un buffer, definit de utilizator, care are în primul câmp o componentă de tip long și care specifică tipul mesajului, urmată apoi de porțiunea de date.

În Linux prototipul funcției este:

```
int msgsnd(int msqid, const struct msgbuf* msgp,
           size_t msgsz, int msgflg);
```

#### Functia msgrecv()

Pentru a citi din coadă folosim:

```
size_t msgrecv(int msqid, void* msgp,
               size_t msgsz, int msgflg);
```

Argumentul msgp este un pointer la o structură buffer, definită de utilizator, care conține ca prim membru un întreg de tip long ce specifică tipul mesajului urmat de zona de date:

```
struct msgbuf{
    long mtype;
    char mtext[1];
};
```

Primul membru reprezintă tipul mesajului recepționat. mtext este textul mesajului. Argumentul msgsz specifică lungimea în bytes a componentei mtext. Mesajul recepționat va fi trunchiat la lungimea msgsz dacă în cadrul flagurilor msgflg precizăm MSG\_NOERROR. Altfel funcția msgrecv() va returna eroarea E2BIG.

Argumentul msctype determină politica la recepție astfel:

- dacă == 0, primul mesaj din coadă este solicitat;
- dacă == n>0, primul mesaj de tipul n este solicitat;
- dacă == n<0, primul mesaj al cărui tip este mai mic sau egal cu valoarea absolută a lui msctype va fi solicitat.

Argumentul msgflg precizează cum să se procedeze dacă tipul dorit nu este în coadă:

-dacă msgflg conține și IPC\_NOWAIT, funcția msgrecv() va returna imediat cu eroarea ENOMSG; astfel se intră în sleep până când:

- un mesaj de tipul dorit este disponibil în coadă;
- coada căreia îi solicităm un mesaj este distrusă de altcineva, astfel că msgrecv() returnează eroarea EIDRM;
- sleepul este întrerupt de un semnal.

Dacă recepționarea s-a efectuat cu succes, structura informațională asociată cu msqid este actualizată astfel:

- msg\_qnum va fi decrementat cu 1;
- msg\_lpid va fi setat la pid-ul procesului apelant;
- msg\_rtime este setat la timpul curent.

La Linux prototipul funcției este:

```
Ssize_t msgrecv(int msqid,struct msgbuf*msgp,
Ssize_t msgsz,long msctype,\int msgflg);
```

#### Functia msgctl()

Prototipul funcției este;

```
int msgctl(int msqid,int cmd,struct msqid_ds*buf);
```

Sunt permise următoarele comenzi:

IPC\_STA va copia informațiile din structura informațională asociată cu msqid în structura indicată prin pointerul buf, dacă avem dreptul de read asupra cozii;

IPC\_SET va scrie unii membri din structura indicată prin pointerul buf în structura de date informațională a cozii; membrii care pot fi modificați sunt:

msg\_perm.uid

msg\_perm.gid

msg\_perm.mode //numai LSB 9 biți

msg\_qbytes

Această actualizare se va efectua dacă procesul apelant are privilegiile necesare: root sau user id-ul efectiv al procesului este cel al msg\_perm.cuid sau msg\_perm.uid. La Linux pentru a mări msg\_qbytes peste valoarea sistem MSGMNB trebuie să fim root. După o operație de control se va actualiza și msg\_ctime.

IPC\_RMID va distruge coada al cărei identificator a fost specificat în msgctl() împreună cu structura de date informațională msqid\_ds asociată. Această comandă va putea fi executată de un proces cu user id-ul efectiv egal cu cel msg\_perm.cuid sau msg\_perm.uid.

Erori:

EINVAL - msqid greșit sau comanda greșită;

EIDRM – coada deja distrusă;

EPERM – comanda IPC\_SET sau IPC\_RMID dar procesul apelant nu are drepturile necesare;

EACCES – comanda IPC\_STAT dar procesul apelant nu are dreptul de read;

EFAULT – comanda IPC\_SET sau IPC\_STAT dar adresa specificată de pointerul buf nu este accesibilă.

## 2) Desfășurarea lucrării

### Program a

/\*

\*scriere-q.c—scriu mesaj în coadă

se compilează cu

\$gcc –o scriere – q scriere – q.c

și se lansează în execuție

\$./scriere\_q nr-nivel

se introduc linii de text.

Se pornește citirea cu

\$gcc scriere-q nr\_nivel

Să se încerce citirea cu niveluri diferite\*/

```
#include<stdio.h>
```

```
#include<errno.h>
```

```
#include<sys/ipc.h>
```

```
#include<sys/msg.h>
```

```
#include <sys/stat.h>
```

```
struct my_buf{
```

```
    long mytype;
```

```
    char mytext[400];
```

```
};
```

```
int main(int argc,char*argv[])
```

```
{
```

```
    struct my_buf buf;
```

```
    int msqid;
```

```
    key_t key;
```

```
    if(argc!=2){printf("utilizare:$./scriere-q
```

```
nivel-numeric\n");}
```

```
    if((key=ftok("scriere-q.c",'L'))==(key_t)-1){
```

```
        perror("ftok");
```

```
        exit(1)
```

```
}
```

```
    if((msqid=msgget(key,S_IRUSR|S_IWUSR|S_IRGRP|S
```

```
_IROTH|IPC_CREAT))==-1){
```

```
        perror("msgget");
```

```
        exit(1);
```

```
}
```

```
    printf("introduceți linii de text,\r\npt.
```

```

terminare:\n");
buf.mtype=atoi(argv[1];/*nu ne interesează
acum*/
while(gets(buf.mytext),!feof(stdin)){
if(msgsnd(msqid,(struct msghdr*)&buf,
sizeof(buf),0)==-1)
perror("msgsnd");
}
if(msgctl(msqid,IPC_RMID,NULL)==-1){
perror("msgctl");
exit(1)
}
return 0;
}

```

### Program b

/\*

citire-q.c- citește coada

Se lansează scrierea:

./scriere-q nivel

se începe introducerea liniilor de test.

Se ieșe cu ^D.

Să se lanseze citirile cu niveluri aleatoare

./citire-q nivel

\*/

#include<errno.h>

#include<sys/ipc.h>

#include<sys/msg.h>

#include<sys/stat.h>

```

struct my_buf {
    long mtype;
    char mytext[400];
};

int main(int argc,char*argv[])
{
    struct my_buf buf;
    int msqid;
    key_t key;

    if(argc!=2){printf("Utilizare:./citire-q
nivel_ numeric\n");exit(2);}

    if((key=ftok("scriere-q.c",'L'))==(key_t)-1{
        /*aceeași cheie ca în scriere-coada.c*/
        perror("ftok");
        exit(1);
    }

    if((msqid=msgget(key,S_IRUSR|S_IWUSR|S_IRGRP|S
IROTH))==-1 {/*conectare la coadă*/
        perror("msgget");
        exit(1);
    }

    printf("citire mesaj:sunt gata pentru recepție mesaje.....\n");
    while(1){/*citire mesaj nu se termină
niciodată*/
        if(msgrcv(msqid, (struct msgbuf*)&buf,
sizeof(buf),atoi(argv[1],0)==-1){
            perror("msgrcv");
        }
    }
}

```

```

    exit(1);

}

printf("citire-mesaj:\'%s\'\n",buf.mytext);

}

return 0;
}

```

### 3) Temă

Să se creeze două procese numite "client" și "server" în care clientul va introduce mesaje în sir iar serverul va extrage mesajul de prioritate maximă.

#### 10.3.3.4. Comunicație între procese prin sistem V IPC. Semafoare

Așa cum am arătat în capitolele anterioare, semaforul a fost inventat de Edsger Dijkstra, ca obiect de sincronizare a proceselor. Implementarea din Linux este bazată pe acest concept dar oferă facilități mai generale.

Există o implementare în SVR4, foarte complexă, cu următoarele caracteristici:

- semafoarele nu există individual ci numai în seturi, numărul semafoarelor dintr-un set fiind definit la crearea setului;

- crearea și inițializarea sunt două operații separate și distincte; crearea se face prin apelul semget iar inițializarea prin apelul semet1;

- deoarece semafoarele rămân în sistemul de operare după terminarea proceselor care le utilizează, ca și la celelalte structuri IPC, trebuie găsită o soluție de tratare a situațiilor în care un program se termină fără a elibera semafoarele alocate.

Forma structurilor semafoarelor este:

```

struct semid_ds
{
    struct ipc_perm sem_perm;
    struct sem *sem_base; /*primul semafor*/
    ushort sem_usems; /*numărul semafoarelor din
set*/
    time_t sem_otime; /*timpul ultimei operații*/
    time_t sem_ctime; /*timpul ultimei
modificări*/}

```

La rândul său, câmpul sem\_base are următoarele structuri:

```

struct sem
{
    ushort semval; /*valoarea semaforului*/
}

```

```

pid_t sempid; /*pid-ul ultimei operații*/

ushort semcnt; /*numărul de procese care
îndeplinesc condiția semval>crtval*/
ushort semzcnt; /*numărul de procese pentru
semval=0*/
};

```

Apelul sistem de creare a semafoarelor este:

```

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/sem.h>

int semget(key_t key,int nsems, int flag);

/*unde nsems reprezintă numărul de semafoare din set; valoarea lui se pune la crearea semaforului*/

```

Apelul sistem pentru inițializare este:

```

int semcti(int semid,int semnum,int cmd,union semnum arg);
unde parametrii reprezintă:

```

semnum indică un semafor din set pentru funcțiile cmd

union semnum arg are următoarea structură:

union sem num

{int val; /\*pentru SETVAL\*/

struct semid\_ds\*buf; /\*pentru IPC\_STAT și

IPC\_SET\*/

ushort\*array; /\*pentru GETALL șiSETALL\*/

}

cmd specifică 10 funcții de executat asupra setului identificat de semid.

## 2) Desfășurarea lucrării

Program a Este un program care implementează primele semafoare binare, inventate de Dijkstra, pentru niște procese care intră în secțiunea critică.

```

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/sem.h>

#include<errno.h>

#define SEMPERM 0100

```

```

#define TRUE 1
#define FALSE 0

typedef union_semun
{
    int val;
    struct semid_ds*buf;
    ushort*array;}semun;

int init_sem(key_t semkey)
{
    int status=0,semid;
    if((semid=semget(semkey,1,SEMPERM |
IPC_CREAT | IPC_EXCL))==-1)
    {if(errno==EEXIST) semid=semget(semkey,1,0);}
    else
    {semun arg; arg.val=1;
    status=semctl(semid,0,SETVAL,arg);}
    if(semid== -1|| status== -1)
    {perror("eroare intrare"); return -1 }
    return semid;
/*implementarea operației p(s)*/
    int p(int semid)
    {struct sem buf p_buf;
    p_buf.sem_num=0;
    p_buf._op=-1;
    p_buf.sem_flg=SEM_UNDO;
    if(semop(semid,&p_buf,1)==-1)
    {perror("p(semid) failed");exit(1);}
    return 0;}
/*implementarea operației v(s)*/
    int v(int semid)
    {struct sembuf v_buf;

```

```

v_buf.sem_num=0;
v_buf.sem_op=1;
v_buf.sem_flg=SEM_UNDO;
if(semop(semid,&v_buf,1)==-1)
{perror("v(semid) failed");exit(1);}
return 0;
/*procese concurente ce folosesc semaforul*/
int semid;
pid_t pid=getpid();
if((semid=init_sem(skey))<0) exit(1);
printf("proces %d inaintea secțiunii
critice\n",pid);
p(semid);printf("procesul %d în secțiune
critică\n", pid);
sleep(5);/*se desfășoară operațiile critice*/
printf("procesul %d părăsește secțiunea
critică\n",pid);
v(semid); printf("procesul %diese\n",pid);
exit(0);
/*programul principal*/
main()
{
key_t semkey=0x200;
int i;
for(i=0;i<3;i++)if(fork()==0)procsem(semkey);
}

```

### 3) Temă

Să se implementeze problema producător-consumator folosind semafoarele.

### 10.3.3.5. Comunicația între procese prin sistem V IPC. Memorie partajată

#### 1) Considerații teoretice

Mecanismul care permite comunicarea între două sau mai multe procese folosind o zonă comună de memorie este folosit frecvent de multiprocesoare. Pentru a utiliza memoria partajată este nevoie de sincronizarea proceselor, deci de utilizarea excluderii mutuale. Implementarea excluderii mutuale se poate face cu obiecte de sincronizare, cel mai adesea cu semafoare.

Structura pentru evidența segmentelor de memorie este:

```
struct shmid_ds

{ struct ipc_perm shm_perm; /* drepturi de acces */

  struct anon_map*shm_map; /* pointer spre kernel */

  int shm_segsz; /* dimensiune segment */

  ushort shm_lkcnt; /* zăvorârea segmentului */

  pid_t shm_lpid; /* pid petru ultim shmop */

  pid_t shm_cpid; /* pid-ul procesului creator */

  ulong shm_nattch; /* număr de atașări curente */

  ulong shm_cnattch; /* shminfo */

  time_t shm_atime; /* timp ultima atașare */

  time_t shm_dtime; /* timp ultima deatașare */

  time_t shm_ctime; /* timp ultima modificare */

}
```

Pentru a obține un identificator de memorie partajată se utilizează apelul `shmget`

```
#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

int shmget(key_t key,int size,int flag);
```

#### 2) Desfășurarea lucrării

##### Program a

```
#include<stdio.h>

#include<signal.h>

#include<sys/types.h>

#include<sys/ipc>
```

```

#include<sys/sem.h>

#define SHMKEY1(key_t)0x100 /*cheia primului
segment*/
#define SHMKEY2 (key_t) 0x1AA /*cheia pentru al
doilea segment*/
#define SEMKEY (key_t) 0x100 /*cheie pentru
semafoare*/
#define SIZ 5*BUFSIZ /*dimensiunea segmentului
BUFSIZ*/
struct databuf
{
    int d_nread;
    char d_buf[SIZ];
};

typedef union_semun
{
    int val;
    struct semid_ds*buf;
    ushort*array;
} semun;
/*rutinele de initializare*/
#define IFLAGS(IPC_CREAT|IPC_EVCL)
#define ERR ((struct databuf*)-1)
static int shmid1,shmid2,semid;
/*se creează segmentele de memorie partajată*/
void getseg(struct databuf**p1,struct databuf p2)
{
    if((shmid1=shmget(SMKEY,sizeof(struct databuf),
0600|IFLAGS))==-1)
        perror("shmget error"); exit(1);
    /*atașează segmentele de memorie*/
    if((*p1=(struct databuf*)shmat(shmid1,0,0))==ERR)
        perror("shmget error"); exit(1);
}

```

```

if((*p2=(struct databuf*)shmat(shmid2,0,0))==ERR)
{perror("shmget error"); exit(1);}

    int getsem(void)
    {semun x;x,val=0;

     /*se creează un set cu două semafoare*/

     if((semid=semget(SEMKEY,2,0600|FLAGS))==-1)
     {perror("semget error"); exit(1);

      /*se initializează valorile semafoarelor*/

      if(semctl(semid,0,SETVAL,x)==-1)
      {perror("semct error"); exit(1);

       return semid;

      /*rutina pentru ștergerea identificatorilor

       memoriei partajate și semafoarelor*/

      void remobj(void)

      {if(shmctl(shmid1,IPC-RMID,NULL)==-1
      {perror("semct error"); exit(1);

      {if(shmctl(shmid2,IPC-RMID,NULL)==-1
      {perror("semct1 error"); exit(1);

      {if(semctl(semid,IPC-RMID,NULL)==-1
      {perror("shmctl1 error"); exit(1); }

      /*definiție pentru operațiile p() și v() pe

       cele două semafoare*/

      struct sembuf p1={0,-1,0},p2={1,-1,0};

      struct sembuf v1={0,1,0}, v2={1,1,0};

      /*rutina de citire*/

      void reader(int semid,struct databuf*buf1,
      struct databuf*buf2)

      {for(;)

      /*citire în tamponul buf1*/
      buf1->d_nread=read(0,buf1->d_buf,SIZ);

```

```

/*punct de sincronizare*/
semop(semid,&v1,1);
semop(semid,&p2,1);

/*test pentru procesul writer*/
if(buf1→d_nread<=0)
return;
buf2→d.nread=read(0,buf2→d-buf,SIZ);
semop(semid,&p1,1);
semop(semid,&v2,1);
if(buf2
d_nread<=0) return;
write(1,buf2→d_buf,buf2→d_nread);{}}

/*program principal*/
main()
{int semid;
pid_t pid;
struct databuf*buf1,*buf2)
semid=getsem();
switch(pid=fork())
{case -1;perror("fork error"); exit(1);
case 0; /*proces fiu*/
writer(semid,buf1,buf2);
remobj();
brech;
default:/*proces părinte*/
reader(semid,buf1,buf2);
brech;}
exit(0);}

```

### 10.3.3.6. Comunicația între fire de execuție

#### 1) Considerații teoretice

În capitolul 3 am definit firele de execuție și motivele din care au fost introduse. Firele de execuție (thread-urile) pot fi considerate ca niște subunități ale proceselor.

Crearea unui fir de execuție se face prin comanda:

```
#include<pthread.h>
```

```
int pthread_create(pthread_t*thread,const pthread_
```

```
attrt*attr,void*(start_routine)(void*),void arg);
```

Firul de execuție nou creat va executa codul din start\_routine căruia î se transmit argumentele arg.

Noul fir de execuție are atributele transmise prin attr, iar dacă ele sunt implicite se utilizează NULL. Dacă funcția se execută cu succes ea va returna 0 și în thread se va pune identificatorul nou creat.

Terminarea execuției unui fir de așteptare se specifică prin apelul funcției pthread.

```
#include<pthread.h>
```

```
void pthread_exit(void*status);
```

O altă proprietate a unui fir de execuție este detașarea. Firele de execuție detașate eliberează, în momentul terminării lor, memoria pe care au deținut-o, astfel că alte fire nu se pot sincroniza cu fire detașate. Implicit, firele sunt create cu atributul joinable, ceea ce face ca alte fire să poată specifica că așteaptă terminarea unui astfel de fir.

```
#include<pthread.h>
```

```
int pthread_join(pthread_t thread,void status);
```

#### 2) Desfășurarea lucrării

##### Program a

```
#include<stdio.h>
```

```
#include<pthread.h>
```

```
int global=5;
```

```
void*copilfuncție(void*p)
```

```
{printf("copilaici,pid=%d,global=%d\n",getpid(),
```

```
global());
```

```
global=15;
```

```
{ printf("copil,globalacum=%d\n",global);}
```

```
main()
```

```

pthread_t copil;
pthread_create(&copil,NULL,copilfuncție,NULL);
printf("părinte,pid=%d,global=%d\n",getpid(),
global);
global=10;
pthread_join(copil,NULL);
printf("nucopil,global=%d\n",global);}

```

Programul se va compila astfel

```
$gcc -D-REENTRANT -o sorin sorin.c -epthread
```

unde sorin.c este fișierul sursă iar constanta REENTRANT specifică execuția în paralel a fililor.

Un posibil răspuns ar fi:

```
copil aici,pid=3680,global=5
```

```
copil,global acum 15
```

```
părinte,pid=3680,global=15
```

```
nu copil,global=10
```

### 3) Temă

Să se creeze trei fire de execuție în care:

- primul fir calculează media aritmetică a n numere citite,
- al doilea fir calculează media geometrică a n numere citite,
- al treilea fir calculează media armonică a n numere citite.

(n și numerele se citesc de la tastatură) . Apoi să se compare rezultatele.

#### **10.3.3.7. Interfața SOCKET**

##### *1) Considerații teoretice*

Interfața SOCKET reprezintă o facilitate generală de comunicare a proceselor aflate, în general, pe mașini diferite.

Un SOCKET poate avea tipuri diferite și poate fi asociat cu unul sau mai multe procese, existând în cadrul unui domeniu de comunicație. Datele pot fi schimbate numai între SOCKET-uri aparținând aceluiași domeniu de comunicație.

Există două primitive pentru SOCKET-uri.

##### Prima primitivă

```
#include<sys/types.h>
#include<sys/socket.h>
int socket(int domain,int type,int protocol)
```

int domain este un parametru ce stabilește formatul adreselor mașinilor implicate în transferul de date.  
Uzual aceste domenii sunt:

AF-UNIX, care stabilește domeniile de comunicare locală (UNIX);

AF-INET, care folosește protocolul TCP/IP și este utilizat în INTERNET.

int type se referă la modalitățile de realizare a comunicării. Cele mai utilizate tipuri sunt:

SOCK-STREAM, în care un flux de date se transmite într-o comunicare de tip full-duplex;

SOCK-DGRAM, în care se stabilește o comunicare fără conexiune cu utilizarea datagramelor.

int protocol specifică protocolul particular utilizat pentru transmisia datelor. De obicei se utilizează valoarea 0(zero).

#### A doua primitivă este SOCKETPAIR()

Aceasta se utilizează pentru crearea unei perechi de SOCKET-uri conectate.

```
#include<sys/types.h>
#include<sys/socket>
int socketpair(int domain,int type,int protocol,
int SV[2];
```

Primele trei argumente sunt la fel ca la socket iar cel de-al patrulea argument SV[2] este la fel ca la pipe.

#### *2) Desfășurarea lucrării*

##### Program a

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<stdio.h>
#include<errno.h>

#include<unistd.h>
#include<string.h>
#include<sys/wait.h>

main()
{
    int sd[2];
    /*mesajele folosite*/
    char*p="sunt părintele";
    char*c="sunt copilul";
    char tampon[500];
    /*se creează perechea de socketuri*/
```

```
if(socketpair(AF-UNIX,SOCK-STREAM,0,fd)==-1)
{perror("eroare la crearea socket");exit(1);}

/*crearea fiului*/
switch(fork())
{case -1 : /*eroare*/
perror("eroare la creareproces");
exit(1);

brech;
case 0: /*fiu*/
/*citim din socket mesajul*/
if(read(fd[1],buf,100)<0)
{perror("eroare la citire"); exit(2);}

printf("procesul cu pid-ul%d(fiu)a primit%s\n",
getpid, tampon);

/*scriem mesajul copilului*/
if(write(fd[1],c,100)<0)
{perror("eroare la citire"); exit(2);}

/*trimitem EoF*/
else fd[1];
exit(0);

default /*parinte*/
/*trimitem mesajul parintelui*/
if(writw(fd[0],p,100)<0)
{perror("eroare scriere"); exit(3);}

/*citim mesajul pornind de la copil*/
if(read(fd[0],tampon,100)<0)
{perror("eroare citire"); exit(3);}

printf("procesul cu pid %d(parinte)
a primit%s\n",getpid(),tampon);
```

```

/*să aşteptăm terminarea copilului*/
if(wait(NULL)<0)
{perror("eroare wait"); exit(3);}

close(sd[0]);

return 0} }

```

După rulare se va afișa:

procesul cu pidul 10896(fiul) |sunt părintele|

procesul cu pidul 10897(părintele)|sunt fiul|

eroare: No child proces

### 3) Temă

Să se creeze o pereche de socket-uri în care primul socket va trimite celui de-al doilea socket un sir de caractere, iar cel de-al doilea va returna primului socket caracterele ordonate după alfabetul ASCII.

#### **10.3.3.8. Modelul client/server-TCP**

##### *1) Considerante teoretice*

În modelul client /server o mașină numită server oferă anumite servicii altor mașini numite clienți.

TCP (Transmission Control Protocol) este un protocol de comunicație care realizează legătura între client și server prin intermediu socketurilor, utilizând streamuri.

Schema generală de funcționare client/server – TCP, în Linux, este dată în fig. 10.2.

#### **Server TCP**

Se execută următoarele apeluri sistem:

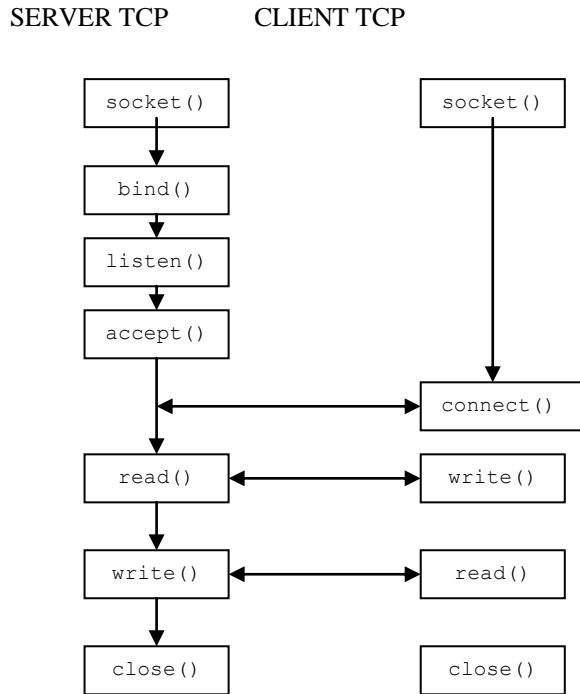
socket() – se crează un socket care va trata conexiunile cu clienții.  
bind() - se atașează socketul creat anterior la un port de comunicație.

listen() – se instalează socketul în vederea ascultării portului pentru stabilirea conexiunii cu clienții.

accept() – se așteaptă realizarea unei conexiuni cu un client și apoi acest apel blochează programul până când vine o cerere de conectare de la alt client.

read(),write() – primitive pentru schimbul de mesaje client/server.

close() – se închide conexiunea cu clientul.



**Fig. 10.2. Schema de funcționare client/server-TCP.**

### Client TCP

Se utilizează aceleași apeluri sistem ca și la server, cu excepția primitivei accept() care trebuie să conțină adresa IP și portul serverului la care se conectează clientul.

La apelul read() din server va corespunde un apel write() la client iar la write() din server va corespunde un apel read() în client.

Primitivele utilizate în acest protocol sunt:

#### a) bind()

```
#include<sys/types.h>
#include<sys/socket.h>

int bind(int sockfd,struct sockaddr*addr,
socklen_t addrlen);

int sockfd este descriptorul serverului.
```

struct sockaddr\*addr este o structură care reține informația de adresă pentru orice tip de socket-uri. Este definită astfel:

```
struct sockaddr
{
    unsignt short sa_family;
    char      sa_data[16]
}
```

În cazul INTERNET-ului structura utilizată este:

```
struct sockaddr_in
{
    short int sin_family; /*familia de adrese AF_INET*/
    unsignet short int sin_port; /*portul(0-65365)*/
    struct in_addr sin_addr; /*adresa Internet*/
    unsignet char sin_zero[8]; /*octeți neutilizați*/
}
```

Trebuie testat că sin\_zero este nul și acest lucru se realizează prin funcțiile bzero() sau manset().

Adresa Internet este stocată în structura in\_addr :

```
struct in_addr
{
    unsigned long int s_addr; /*adresa IP*/
```

#### **b) listen()**

```
#include<sys/socket.h>
int listen(int sockd,int backlog);
-backlog arată numărul de conexiuni permise în coada de așteptare a conexiunilor clienți, uzual fiind 5.
```

#### **c) accept()**

Se rulează așteptarea de către master.

```
#include<sys/types.h>
#include<sys/socket.h>
in accept(int socd,struct sockaddr*addr,
soclen_t*addrlen)
```

#### *2) Desfășurarea lucrării*

Acest program creează un server și un client; serverul primește un sir de caractere de la client și îl trimite înapoi în ecou. Clientul citește un sir de caractere de la intrarea standard, îl trimite serverului, apoi așteaptă ca serverul să îl returneze.

#### **Server C**

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
```

```
#include<errno.h>
#include<unistd.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define PORT 8081 /*se definește portul*/
extern int errno; /*codul de eroare*/
main()
{/*structurile utilizate de server și client*/
    struct sockaddr_in server;
    struct sockaddr_in from;
    char tampon[100]; /*mesaj trimis de client*/
    int sd      /*descriptorul de socket*/
    /*se crează un socket*/
    if((sd=socket(AF_INET,SOCK_STREAM,0))==-1)
    {perror("eroare socket\n");return errno;}
    /*se pregătesc structurile de date*/
    bzero(&server,sizeof(server));
    bzero(&from,sizeof(from));
    server.sin_family=AF_INET; /*familia de socketuri*/
    server.sin_addr.s_addr=htonl(INADDR_ANY);
    server.sin_port=htons(PORT);
    /*se atașează socketul*/
    if(bind(sd,(struct sockaddr*)&server,
    sizeof((struct sockaddr))==1)
    {perror("eroare bind\n");return errno;}
    /*serverul ascultă linia dacă vin clienți*/
    if(listen(sd,5)==-1)
    {perror("eroare listen\n");return errno;}
```

```

/*se servesc clienții*/

while(1)

{int client;

int length=sizeof(from);

printf("se așteaptă la portul %d\n",PORT);

fflush(stdout);

/*se acceptă un client*/

/*serverul se blochează până la realizarea conexiunii*/

client=accept(sd,(struct sockaddr*)&from,&length);

if(client<0)

{perror("eroare client\n");continue;}

bzero(tampon,100);

/*s-a realizat conexiunea,

se așteaptă mesajul*/

printf("așteptăm mesajul\n");fflush(stdout);

/*se citește mesajul*/

if(read(client,tampon,100)<=0)

{perror("eroare read\n");close(client)continue;}

/*s-a închis conexiunea cu clientul*/

printf("mesaj recepționat, trimitem mesaj înapoi");

/*se returnează mesajul clientului*/

if(write(client,tampon,100)<=0)

{perror("eroare write\n");continue;}

else printf("transmitere cu succes\n");

/*am terminat cu acest client,

se închide conexiunea*/

close(client)}}

```

### Client C

```

/*retransmite serverului mesajul primit de la acesta*/
#include<sys/types.h>

#include<sys/socket.h>

```

```

#include<etinet/in.h>
#include<errno.h>
#include<unistd.h>
#include<stdlib.h>
#include<netdb.h>
#include<stroing.h>

extern int errno; /*codul de eroare*/

int port; /*portul de conectare la server*/

int main(int argc,char*argv[])
{
    int sd; /*descriptorul de socket*/
    /*structura utilizată la conectare*/
    struct sockaddr_in server;
    char tampon[100]; /*mesaj transmis*/
    /*testarea argumentelor din linia de comandă*/
    if(argc!=3)
        {perror("eroare argumente\n");return -1;}
    port=atoi(argv[2]);
    /*se creează socketul*/
    if((sd=socket(AF_INET,SOCK_STREAM,0))==-1)
        {perror("eroare socket\n");return errno;}
    server.sin_family=AF_INET; /*familia socketului*/
    server.sin_addr.s_addr=inet_addr(argv[1]);
    /*adresa IP a serverului*/
    server.sin_port=htons(port); /*portul de
                                conectare*/
    if(connect(sd,(struct sockaddr*)&server)==-1)
        {perror("eroare connect\n");return errno;}
    /*citirea mesajului și transmiterea către server*/
    bzero(tampon,100);
}

```

```

printf("introduceți mesajul")fflush(stdout);

read(0,buffer,100);

if(writw(sd,tampon,100)<=0)

{perror("eroare scriere\n");return errno;}

/*afişarea mesajului primit*/

printf("mesajul primit este%s\n",tampon);

close(sd);}

```

Pentru compilarea clientului și a serverului vom folosi comenziile:

```
$gcc -o server server.c
```

```
$gcc -o client client.c
```

Pentru execuția programelor:

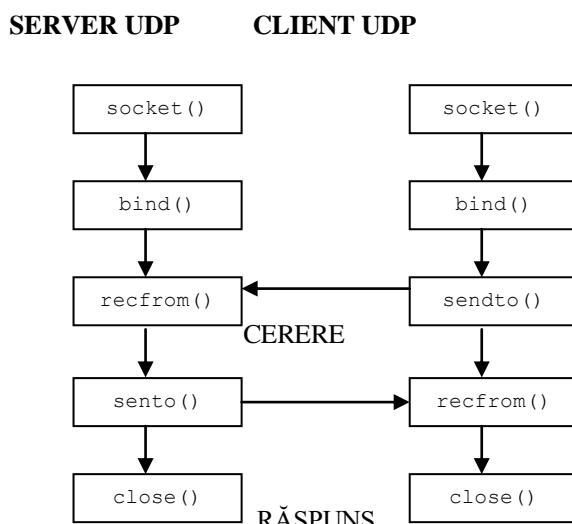
```
$ ./server
```

```
$ ./client 127.0.0.1 8081
```

3) Temă. Să se creeze două fișiere, server și client, în care clientul trimite serverului numere întregi iar serverul va returna clientului numai numerele pare dintre cele transmise (în TCP).

#### 10.3.3.9. Modelul client/server-UDP (User Datagrama Protocol)

În acest protocol de transmisie a datelor nu se realizează o conexiune între client și server pentru ca apoi să se citească și să se scrie date. În UDP transmisia este asincronă, în sensul că clientul și serverul își trimit mesaje unul altui prin intermediul primitivelor SEND și RECEIVE. Structura de date transmisă se numește datagramă. Organizarea UDP este dată în fig.10.3.



**Fig.10.3. Organigrama UDP**

Apelurile folosite sunt:

recvfrom() cu sintaxa:

```
#include<sys/types.h>
#include<sys/socket.h>

int recvfrom(int sockd,void*buf,size_t len,
int flags,struct sockaddr*from,socklen_t*fromlen);
```

sendto() cu sintaxa:

```
#include<sys/types.h>
#include<sys/socket.h>

int sendto(intsockd,const void*msg,size_t len,
int flags, const struct sockaddr*to,socklen_t
tolen)
```

*2) Desfășurarea lucrării*

**Server c**

```
/*server UDP iterativ (echo)
Așteaptă un mesaj de la clienti;

Mesajul primit este trimis înapoi.*/
#include<sys/types.h>
#include<sys/socket.h>
#include<stdio.h>
#include<netinet/in.h>
#include<errno.h>ss
#include<unistd.h>
/*portul folosit*/
#define PORT 8081
/*codul de eroare returnat d anumite apeluri*/
extern int errno;
```

```

/*programul*/

ini

main()

{

/*structurile folosite de server și client*/

struct sockaddr_in adresa;

struct sockaddr client;

char buffer[10 /*mesajul trimis de client*/]

int sd      /*descriptorul de socket*/


/*lansăm serverul în fundal...*/

switch(fork())

{

case :-1    /*eroare la fork*/

    perror("Fork error\n");

    return errno;

case 0:     /*copilul trăiește*/

    break;

default:    /*părintele moare...*/

    print("serverul a fost lansat în fundal\n");

    exit(0);

}

/*creăm un socket*/

if ((sd=socket(AF_INET,SOCK_DGRAM,0))==-1)

{perror("eroare socket().\n");return errno;}


/*să pregătim structura folosită de server*/

adresa.sin_family=AF_INET;

/*stabilirea familiei de socket-uri*/

```

```

adresa.sin_addr.s_addr=htonl(INADDR_ANY);

/*acceptăm orice adresă*/

adresa.sin_port=htons(PORT);

/*utilizăm un port utilizator*/

/*atașăm socketul*/

int (bind(sd,struct sockaddr*)&adresa,
         sizeof(struct sockaddr))== -1)

{perror("eroare la bind().\n");return errno; }

/*servim în mod iterativ clienții*/

while(1)

{

    int bytes;

    int length = sizeof(client);

    /*citim mesajul primit de la client*/

    if(bytes=recvfrom(sd,buffer,100,0,
                      &client      ,&length)<0)

    {perror("eroare la recvfrom().\n");

     return errno; }

    /*..după care îl trimitem înapoi*/

    if(sendto(sd,buffer,bytes,0,&client,length)<0)
    {perror("eroare la sendto() spre client.\n");

     return errno; }

}

/*while*/
}
/*main*/

```

### Client UDP(echo)

/\*Client UDP (echo)

Trimite un mesaj unui server;

Mesajul este recepționat de la server.\*/

```
#include<sys/types.h>
#include<sys/socket.h>
#include<stdio.h>
#include<netinet/in.h>
#include<errno.h>
#include<netdb.h>
#include<string.h>

/*codul de eroare returnat de anumite apeluri*/ extern int errno;

/*portul de conectarea la server*/
int port;

/*programul*/int
main(int argc,char*argv[])
{
    /*descriptorul de socket*/
    int sd;
    /*structura folosita pentru conectare*/
    if(arg!=3)
        {printf("sintaxa:%s<adresa-server><port>\n",
               argv[0]);return-1}
    /*stabilim portul*/
    port=atoi(argv[2]);
    /*cream socketul*/
    if((sd=socket(AF_INET,SOCK_DGRAM,0))==-1)
        {perror("eroare la socket().\n");return errno;}
    /*umplem structura folosita pentru realizarea dialogului cu serverul*/
    server.sin_family=AF_INET;
    /*familia socketului*/
    server.sin_addr.s_addr=inet_addr(argv[1]);
    /*adresa IP a serverului*/
```

```
server.sin_port=htons(port);
/*portul de conectare*/
/*citirea mesajului de la intrareastandard*/
bzero(buffer,100);
printf("introduceți mesajul:");
fflush(stdout);
read(0,buffer,100;
length=sizeof(server);
/*trimiterea mesajului către server*/
if(sendto(sd,buffer,strlen(buffer),0,
&server,length)<0)
{perror("eroare la sendto() spre server.\n");
return errno;}
printf("mesajul primit este: '%s'.\n",buffer);
/*închidem socketul, am terminat*/
close(sd)
}
```

Clientul va necesita două argumente în linia de comandă, semnificând adresa IP a serverului și portul de conectare la serverul UDP. Dacă ambele programe rulează pe aceeași mașină, atunci vom putea introduce:

```
./server -udp
./client -udp 127.0.0-18081
```

Serverul va rula automat în fundal (adoptând postura de *daemon*).

3) *Tema.*Să se scrie un program în care un client va trimite un sir de numere întregi serverului iar acesta va returna către client numerele primite în ordine inversă.

## Bibliografie

- 1.**D. Cramer**, *Interworking with TCP-IP*, vol.1, Prentice Hall, New -Jersey,1991.
- 2.**Andrew S. Tanenbaum**, *Modern Operating Systems*, Prentice Hall,1992.
- 3.**Iosif Ignat, Emil Muntean, Kalman Pustzai**, *Microinformatica*, 1992.
- 4.**B. Chapman, E.D. Zwickly**, *Building Internet Firewalls*, O'Reilly&Associates, 1995.
- 5.**Traian Ionescu, Daniela Saru, John Floroiu**, *Sisteme de operare-principii și funcționare*, Editura tehnică, București, 1997.
- 6.**R. Stevens**, *UNIX Network Programming, vol. 1, Networking*, Prentice Hall, 1998
- 7.**Felicia Ionescu**, *Principiile calculului paralel*, Editura Tehnică, București, 1999.
- 8.–*Interprocess Communications*, Prentice Hall, N.J. 1999.
- 9.**A. Silberschatz, P.B. Galvin, G.Gagne**, *Applied Operating System Concepts*, Wiley, New-York,200.
- 10.**Liviu Miclea**, *Noțiuni de sisteme de operare și rețele de calculatoare (LINUX)*, Universitatea Tehnică Cluj-Napoca, 2001.
- 11.**Dan Cosma**, *UNIX. Aplicații*, Ed. de Vest, Timișoara, 2001.
- 12.**Ioan Jurcă**, *Sisteme de operare*, Editura de Vest, Timișoara, 2001.
- 13.**Sabin Buraga, Gabriel Ciobanu**, *Atelier de programare în rețele de calculatoare*, Editura Polirom, Iași, 2001.
- 14.**Dragoș Acostăchioiaie**, *Securitatea sistemelor LINUX*, Editura Polirom, Iași, 2001.
15. **Andrew Tanenbaum**, *Sisteme de Operare Moderne*, Editura Byblos, București, 2004.
17. **Cristian Vidrașcu**, <http://www.infoiasi.ro/~vidrascu>.
18. **Mihai Budiu**, *Alocarea memoriei în nucleul sistemului de operare*, <http://www.cs.cmu.edu/mihaiib>, 1998.
17. **[http://www.oreilly.com/catalog/opensources/book/linus.html](http://www.oreilly.com/catalog/opensources/book/)**

Pg.

1. INTRODUCERE.....	3
1.1. SISTEME DE OPERARE. DEFINIȚIE.....	4
1.2. LOCUL UNUI SISTEM DE OPERARE ÎNTR-UN SISTEM DE CALCUL.....	4
1.3. FUNCȚIILE UNUI SISTEM DE OPERARE.....	7
1.3.1. Asigurarea interfeței cu utilizatorul.....	7
1.3.1.1. Monitoare.....	7
1.3.1.2. Interfețe în linie de comandă.....	7
1.3.1.3. Interfețe grafice.....	8
1.3.2. Gestionarea proceselor și procesoarelor.....	9
1.3.3. Gestionarea memoriei.....	9
1.3.4. Gestionarea perifericelor.....	10
1.3.5. Gestionarea fișierelor.....	10
1.3.6. Tratarea erorilor.....	11
1.4. CARACTERISTICILE SISTEMELOR DE OPERARE.....	11
1.4.1. Modul de introducere a programelor în sistem.....	11
1.4.2. Modul de planificare a lucrărilor pentru execuție.....	12
1.4.3. Numărul de programe prezente simultan în memorie.....	12
1.4.4. Gradul de comunicare a proceselor în multiprogramare.....	12
1.4.5. Numărul de utilizatori simultani ai SO.....	13
1.4.6. Modul de utilizare a resurselor.....	13
1.4.7. SO pentru arhitecturi paralele.....	14
1.5. COMPOENȚELE SISTEMELOR DE OPERARE.....	14
1.5.1. Partea de control.....	14
1.5.2. Partea de serviciu.....	15
1.6. Structura sistemelor de operare.....	15
1.7. DEZVOLTAREA ISTORICĂ A SISTEMELOR DE OPERARE.....	16
<b>2 PLANIFICAREA PROCESOARELOR (UC).....</b>	<b>19</b>

2.1	SCHEMA GENERALĂ DE PLANIFICARE.....	19
2.2.	CRITERII DE PERFORMANȚĂ A PLANIFICĂRII.....	20
2.3.	ALGORITMI DE PLANIFICARE UC.....	21
2.3.1.	Algoritmul FCFS (First Come First Served).....	21
2.3.2.	Algoritmul SJF (Shortest Job First).....	22
2.3.3.	Algoritmi bazați pe prioritate.....	22
2.3.4.	Algoritmi preempivi.....	23
2.3.5.	Algoritmul Round-Robin.....	25
2.3.6.	Alți algoritmi de planificare.....	26
3.	<b>GESTIUNEA PROCESELOR.....</b>	<b>27</b>
3.1.	NOȚIUNI GENERALE DE PROCESE ȘI THREAD-URI.....	27
3.1.1.	Definiția procesului.....	27
3.1.2.	Starea procesului.....	28
3.1.3.	Comutarea proceselor.....	30
3.1.4.	Crearea și terminarea proceselor.....	32
3.2.	PROCESE ȘI THREAD-URI ÎN UNIX.....	32
3.2.1.	Procese în UNIX.....	32
3.2.2.	Thread-uri în UNIX.....	37
3.3.	PROCESE ȘI THREAD-URI ÎN WINDOWS.....	37
3.3.1.	Procese în WINDOWS.....	38
4.	<b>COMUNICAȚIA ȘI SINCRONIZAREA ÎNTRE PROCESE.....</b>	<b>39</b>
4.1.	PROBLEMA SECTIUNII CRITICE ȘI A EXCLUDERII MUTUALE.....	39
4.1.1.	Suportul hardware pentru implementarea excluderii mutuale... 4.1.1.1. Invalidarea/validarea intreruperilor..... 4.1.1.2. Instrucțiunea Test and Set (TS)..... 4.1.1.3. Protocole de aşteptare în excluderea mutuală..... 4.1.1.4. Mecanisme de sincronizare între procese (obiecte de sincronizare).....	42 42 42 43 43

4.2. INTERBLOCAREA (DEADLOCK).....	49
4.2.1. Resurse.....	49
4.2.1.1. Clasificarea resurselor din punct de vedere al interblocării.....	49
4.2.1.2. Etapele parcurse de un proces pentru utilizarea unei resurse.....	50
4.2.2. Condiții necesare pentru apariția interblocării.....	50
4.2.3. Graful de alocare a resurselor.....	51
4.2.4. Rezolvarea problemei interblocării.....	53
4.2.4.1. Prevenirea interblocării.....	53
4.2.4.2. Evitarea interblocării.....	55
4.2.4.3. Detectarea interblocării și revenirea din ea.....	59
4.2.4.4. Rezolvarea interblocării în practică.....	63
4.3. COMUNICAȚIA ÎNTRE PROCESE COOPERANTE.....	63
4.3.1. Comunicație directă și indirectă.....	65
4.3.1.1. Comunicație directă.....	65
4.3.1.2. Comunicație indirectă.....	66
4.3.2. Linii de comunicații și tipuri de mesaje.....	67
4.3.2.1. Linii de comunicații.....	67
4.3.2.2. Tipuri de mesaje.....	68
4.3.3. Excepții în comunicația interprocese.....	68
4.3.4. Aplicații ale IPC-urilor.....	70
4.4. PROBLEME CLASICE DE COORDONAREA ȘI SINCRONIZAREA PROCESELOR.....	70
4.4.1. Problema producător-consumator.....	70
4.4.1.1. Rezolvarea problemei producător-consumator cu ajutorul semafoarelor.....	71
4.4.1.2. Rezolvarea problemei producător-consumator prin transmitere de mesaje.....	73

4.4.2.	Problema bărbierului somnoros.....	75
4.4.3.	Problema cititori/scriitori.....	77
4.4.4.	Problema cinei filozofilor chinezi.....	79
4.4.5.	Probleme propuse pentru implementare.....	84
4.4.5.1.	Problema rezervării bileteor.....	84
4.4.5.2.	Problema grădinii ornamentale.....	84
4.4.5.3.	Problema emițător-receptor.....	85
<b>5</b>	<b>GESTIONAREA MEMORIEI.....</b>	<b>87</b>
5.1.	IERARHII DE MEMORIE.....	87
5.2.	OPTIMIZĂRI ÎN ÎNCĂRCAREA ȘI EXECUȚIA UNUI PROGRAM ÎN MEMORIE.....	89
5.2.1.	Încărcarea dinamică.....	89
5.2.2.	Overlay-uri.....	90
5.2.3.	Legarea dinamică.....	90
5.3.	ALOCAREA MEMORIEI.....	91
5.3.1.	Alocarea memoriei în limbaje de programare.....	91
5.3.2.	Caracteristici ale alocatoarelor.....	92
5.3.3.	Tipuri de alocare a memoriei.....	93
5.3.4.	Scheme de alocare a memoriei.....	94
5.3.4.1.	Alocare unică.....	95
5.3.4.2.	Alocare cu partii fixe (alocare statică).....	95
5.3.4.3.	Alocare cu partii variabile.....	96
5.3.4.4.	Alocare prin swapping.....	97
5.4.	PAGINAREA MEMORIEI.....	99
5.4.1.	Suportul hardware.....	100
5.4.2.	Implementarea tablei de pagini.....	101
5.4.3.	Concluzii privind paginarea.....	104
5.4.4.	Segmentarea memoriei.....	104
5.4.5.	Segmentarea paginată.....	105

5.4.6.	Memorie virtuală.....	106
5.4.6.1.	Paginare la cerere.....	106
5.4.7.	Algoritmi de înlocuire a paginii.....	109
5.4.7.1.	Algoritmul FIFO.....	109
5.4.7.2.	Algoritmul LRU (Least Recently Used).....	111
5.4.7.3.	Algoritmul LFU (Least Frequently Used).....	113
5.4.7.4.	Algoritmul Real Paged Daemon.....	113
5.4.7.5.	Fenomenul thrashing.....	114
5.4.7.6..	Concluzii privind paginarea la cerere.....	114
5.5.	ALOCAREA SPAȚIULUI LIBER. TIPURI DE ALOCATOARE.....	115
5.5.1.	Alocatorul cu hărți de resurse.....	115
5.5.2.	Alocatorul cu puteri ale lui doi (metoda camarazilor).....	117
5.5.3.	Alocatorul Fibonacci.....	118
5.5.4.	Alocatorul Karels-Mckusick.....	118
5.5.5.	Alocatorul slab.....	119
5.6.	GESTIUNEA MEMORIEI ÎN UNELE SISTENE DE OPERARE.....	121
5.6.1.	Gestiunea memoriei în Linux.....	121
5.6.2.	Gesiunea memoriei în WINDOWS NT.....	121
<b>6.</b>	<b>GESTIUNEA SISTEMULUI DE INTRARE/IEȘIRE.....</b>	<b>123</b>
6.1.	DEFINIREA SISTEMULUI DE INTRARE/IEȘIRE.....	123
6.2.	CLASIFICAREA DISPOZITIVELOR PERIFERICE.....	125
6.3.	STRUCTURA HARD A UNUI SISTEM DE INTRARE/IEȘIRE.....	126
6.4.	STRUCTURA SOFT A UNUI SISTEM DE INTRARE/IEȘIRE.....	128
6.4.1.	Rutine de tratare a intreruperilor.....	128
6.4.2.	Drivere.....	130
6.4.3.	Programe-sistem independente de dispozitive.....	134
6.4.4.	Primitive de nivel utilizator.....	134
6.5.	ÎMBUNĂTĂȚIREA OPERAȚIILOR DE INTRARE/IEȘIRE.....	135

6.5.1.	Factorul de întrețesere.....	135
6.5.2.	Cache-ul de hard disc.....	137
6.5.3.	Crearea de către SO a unui hard disc cu performanțe superioare.....	137
6.5.4.	Construirea unui hard disc prin tehnica RAID (Redundant Arrays of Independent Disks).....	141
<b>7.</b>	<b>GESTIUNEA RESURSELOR LOCALE.....</b>	<b>143</b>
7.1.	NOȚIUNI INTRODUCTIVE.....	143
7.2.	CLASIFICAREA FIȘIERELOR.....	144
7.2.1.	Clasificarea fișierelor după structură.....	144
7.2.1.1.	Secvențe de octeți .....	144
7.2.1.2.	Secvențe de înregistrare.....	144
7.2.1.3.	Structura arborescentă.....	145
7.2.2.	Clasificarea fișierelor după tip.....	145
7.2.2.1.	Fișiere normale.....	145
7.2.2.2.	Directoare.....	145
7.2.2.3.	Fișiere speciale de tip caracter / bloc.....	145
7.2.3.	Clasificarea fișierelor după suportul pe care sunt rezidente....	146
7.2.4.	Clasificarea fișierelor după acces.....	146
7.2.4.1.	Fișiere cu acces secvențial.....	146
7.2.4.2.	Fișiere cu acces direct.....	146
7.2.4.3.	Fișiere cu acces indexat.....	146
7.3.	ATRIBUTE ȘI OPERAȚII CU FIȘIERE.....	146
7.3.1.	Atribute.....	146
7.3.2.	Operații cu fișiere.....	147
7.4.	IMPLEMENTAREA SISTEMULUI DE FIȘIERE.....	148
7.4.1.	Alocarea fișierelor pe disc.....	149
7.4.1.1.	Alocarea contiguă.....	149
7.4.1.2.	Alocarea înlănțuită.....	149

7.4.1.3. Alocarea indexată.....	151
7.4.2. Evidența blocurilor libere.....	153
7.4.3. Eficiența și performanța sistemului de fișiere.....	154
7.4.4. Fiabilitatea sistemelor de fișiere.....	154
7.4.4.1. Evitarea distrugerii informației.....	155
7.4.4.2. Recuperarea informației în urma unei erori hard sau soft.....	156
7.4.4.3. Asigurarea consistenței sistemului de fișiere.....	156
7.4.5. Protecția fișierelor.....	158
7.4.6. Organizarea fișierelor pe disc.....	159
7.4.6.1. Organizarea fișierelor în SO Unix.....	159
7.4.6.2. Organizarea fișierelor ce folosesc FAT.....	160
7.4.6.3. Organizarea fișierelor în HPFS.....	160
7.4.6.4. Organizarea fișierelor în NTFS.....	162
<b>8. SISTEME DE OPERARE PENTRU CALCULATOARE PARALELE....</b>	<b>163</b>
8.1. NOȚIUNI INTRODUCTIVE.....	163
8.2. SISTEME DE OPERARE ÎN REȚEA.....	165
8.3. SISTEME DE OPERARE CU MULTIPROCESOARE.....	166
8.3.1. Programarea paralelă.....	167
8.3.1.1. Memoria partajată între procese.....	167
8.3.1.2. Exemple de programare paralelă.....	168
8.4. SISTEME DE OPERARE DISTRIBUITE.....	174
8.4.1. Structura unui sistem de operare distribuit.....	175
8.4.1.1. Comunicare sistem client / server.....	176
8.4.1.2. Apeluri de proceduri la distanță.....	179
8.4.1.3. Comunicare în grup.....	182
8.4.2. Exemple de sisteme de operare distribuite.....	198
8.4.2.1. Sistemul de operare AMOEBA.....	198
8.4.2.2. Sistemul de operare GLOBE.....	201

<b>9. SECURITATEA SISTEMELOR DE OPERARE.....</b>	<b>205</b>
9.1. NOȚIUNI INTRODUCTIVE.....	205
9.2. ATACURI ASUPRA SISTEMULUI DE OPERARE ȘI MĂSURI DE PROTECȚIE ÎMPOTRIVA LOR.....	210
9.2.1. Depășirea zonei de memorie tampon (Buffer Overflow).....	210
9.2.2. Ghicirea parolelor (Password guessing).....	211
9.2.3. Interceptarea rețelei.....	211
9.2.4. Atacul de refuz al serviciului (Denial Of Service).....	212
9.2.5. Atacuri cu bomba e-mail.....	214
9.2.6. Falsificarea adresei expeditorului (e-mail spoofing).....	214
9.2.7. Cai troieni (Trojan Horses).....	215
9.2.8. Uși ascunse (Back doors an traps).....	215
9.2.9. Viruși.....	216
9.2.10. Viermi.....	217
9.3. MECANISME DE PROTECȚIE.....	218
9.3.1. Criptografie.....	218
9.3.1.1. Criptografia cu chei secrete (Criptografia simetrică) .....	218
9.3.1.2. Criptografia cu chei publice (Criptografia asimetrică) .....	221
9.3.2. Dispozitive firewall.....	222
9.3.2.1. Tipuri de firewall.....	223
9.3.2.2. Funcțiile unui firewall.....	224
9.3.2.3. Firewall-uri în sistemele de operare Windows.....	225
9.3.2.4. Firewall-uri în sistemul de operare Linux.....	226
9.3.3. Sisteme de încredere.....	228
9.3.3.1. Monitorul de referință.....	229
9.3.3.2. Modelul Liste de Control al Accesului (ACL).....	229
9.3.3.3. Modelul Bell-La Padula.....	231
9.3.3.4. Modelul Biba.....	233

9.3.3.5. Modelul securității Cărții Portocalii.....	233
9.3.4. Securitatea în sistemele de operare Windows.....	235
9.3.4.1. Concepte fundamentale de securitate în Windows....	237
9.3.4.2. Implementarea securității.....	239
9.3.5. Securitatea în Linux.....	240
9.3.5.1. Open Source.....	240
9.3.5.2. Programe ce depistează și corectează vulnerabilități...	241
9.3.5.3. Auditarea sistemului.....	242
<b>10 SISTEME DE OPERARE LINUX. APLICAȚII.....</b>	<b>245</b>
10.1 SCURT ISTORIC.....	245
10.2 DISTRIBUȚII IN LINUX.....	246
10.2. Distribuția SLACWARE.....	246
1.	
10.2. Distribuția REDHAT.....	247
2.	
10.2. Distribuția DEBIAN.....	247
3.	
10.2. Distribuția MANDRAKE.....	248
4.	
10.2. Distribuția LYCORIS.....	248
5.	
10.2. Distribuția SUSE.....	249
5.	
10.3 APLICAȚII LINUX.....	249
10.3. Comenzi LINUX.....	250
1.	
10.3. Crearea proceselor.....	252
2.	
10.3. Comunicare între procente.....	257
3.	
10.3.3.1. Comunicarea între procente prin PIPE și FIFO.....	257
10.3.3.2. Comunicarea între procente prin semnale.....	263

---

10.3.3.3. Comunicarea între procese prin sistem V IPC.	
Cozi de mesaje.....	275
10.3.3.4. Comunicarea între procese prin sistem V IPC.	
Semafoare.....	
	285
10.3.3.5. Comunicarea între procese prin sistem V IPC	
Memorie partajată.....	
	288
10.3.3.6. Comunicarea între fire de execuție.....	291
10.3.3.7. Interfață SOCKET.....	293
10.3.3.8. Modelul client / server TCP.....	296
10.3.3.9. Modelul client / server UDP.....	302
<b>BIBLOGRAFIE.....</b>	<b>309</b>

**DICȚIONAR****DE TERMENI ȘI PRESCURTĂRI**

<b>Algoritm</b>	<u>-de planificare a proceselor:</u>	
	FCFS=First Come First Served	Primul venit primul servit
	FIFO=First Input First Output	Primul intrat primul ieșit
	SJF=Shortest Job First	Cea mai scurtă sarcină mai întâi
	<u>-de alegere a spațiului de memorie liber:</u>	
	FFA=First Fit Algotithme	Algoritm de prima potrivire
	BFA=Best Fit Algorithme	Alg. de cea mai bună potrivire
	WFA=Worst Fit Algorithme)	Alg. de cea mai proastă potrivire
	<u>-de înlocuire a paginii de memorie</u>	
	FIFO=First Input First Output	Primul intrat primul ieșit
	LRU=Least Recently Used	Ultimul folosit
	LFU=Least Frequenzly Used	Cel mai puțin utilizat
	Algoritm real- Paged daemon	Pregătește sistemul pentru evacuarea de pagini
ACE	Acces Control Entries	Intrări de control al accesului
ACL	Acces Control List	Listă de control al accesului
AES	Application Environment Specification	Specificație a serviciilor utilizabile de către o aplicație client/server
APC	Application Procedure Call	Apelul unei proceduri de aplicație
API	Application Programming Interface	O interfață ce conține definiția tipurilor de date și funcțiilor apel în WINDOWS
APT	Advanced Package Toll	Utilitar pentru pachete deb
Atribute	Caracteristici ale fișierelor	Nume, tip, locație, protecție etc.
BCP	Bloc Cotrol Process	Descriptor de proces
BIOS	Basic I/O System	Un sistem de dispozitive intrare/iesire
Boot	Proces Boot	Un proces care inițializează SO
Boot Block	Bloc de Boot	Conține proceduri și funcții pentru inițializarea sistemului de fișiere

Booting	Botare	Creare de procese care să pornească SO
Broadcast	Emisie, emitere	
Bug	defect	În hard și în soft
Bussy	Ocupat	
Bussy-sleep	Așteptare dormantă	Protocol de așteptare în excluderea mutuală
Bussy-wait	Așteptare ocupată	Protocol de așteptare în excluderea mutuală
CACHE		Parte din memoria principală
CBC	Cipher Bloc Chainning	Mod de criptare cu înlátiuire
CFB	Cipher Feed Back	Mod de criptare cu reacție
Checksumm	Sumă de verificare	Suma bițiilor mesajelor
Circuit Level Gateways	Portițe de circuit	
CList	Capabilities List	Listă de capabilități
Coadă	Structură de date	Funcționează după principiul FIFO, cu comenziile POP și PUSH
CODE RED		Numele unui vierme
CORBA	Common Object Request Broker Arhitecture	Sistem bazat pe derularea obiectelor
CPU	Central Processing Unit	Unitate centrală de procesare
CRE	Cluster tools Runtime Environement	Mediu de lansare în execuție în MPI
DACL	Discretionary ACL	Listă de acces discreționară
Daemon	Demon	Un proces care stă în fundal, pentru manevrarea diferitelor activități
Deadlock	Impas	Interblocare în comunicația proceselor
DEC	Distributed Environement Corporation	Consorțiu
DES	Data Encryption Standard	Standard de incriptare a datelor bazat pe chei secrete
DESX	Data Encryption Standard X	Tip de DES
DI/EI	Invalidare/validare întreruperi	
Director	(Catalog)	Un fișier sistem care gestionează structura sistemului de fișiere
Disck cache	O secțiune a memoriei principale	Pentru blocurile de disc cele mai des utilizate

DNS	Domain Name System	Schemă de bază de date care mapează în ASCII numele gazdelor pe adresele lor IP
DOS	Disck Operating System	
Driver		Partea de SO care depinde de perifericul asociat
ECB	Electronic Code Bloc	Mod de criptare cu carte de coduri
Eveniment	(Semnal)	Mecanism de comunicare între procese
FAT	File Alocation Tabel	Tabelă de alocare a fișierului
Firestar	În UNIX	Interfață grafică pentru iptables
Firewall	Perete de foc	Sistem ce separă o rețea protejată de una neprotejată
Firewall Builder	În UNIX	Interfață grafică pentru iptables
Fișier	Entitatea de bază a unui SO	Păstrează informația
Fragile		Atac cu pachete UDP la portul 7 al adresei broadcast
Free behind	Liber în urmă	Tehnică de optimizare a accesului secvențial
FTP	File Transfer Protocol	Mod de transfer al fișierelor
Handle	O intrare în tabelul de resurse	
HMAC	H Message Authentication Code	Cod de autentificare a mesajelor bazat pe funcția de dispersie H
HPFS	High Performance File System	Sistem de fișiere de înaltă performanță
HTTP	Hiper Text Transfer Protocol	
I/O	Input/Output =	Intrare / Ieșire
ICMP		Protocol de transmitere a pachetelor de date
Id	Identifier proces	În Windows
i-nod	Nod de indexare	
IP	Internet Protocol	
IP sniffing	Atac asupra IP	Interceptează o rețea
IPC	Inter Process Communication	Comunicație între procese
IPspoofing	Falsificarea adresei IP	Tip de atac
Iptables		Firewall în UNIX
ISA	Instructions Set Architecture	Arhitectura setului de instrucțiuni
JDK	JAVA Development Kit	

Kernel	Nucleu	
Land craft	„Aterizare forțată”	Tip de atac
LIFO	Last Input First Out	Ultimul intrat, primul ieșit
Listă	Structură de date	Şir de noduri într-o relație de ordine
LST	Lood Sharing Facility	Mediu de lansare în execuție în MPI
MAC	Message Autentification Code	Cod de autentificare a mesajelor
MBR	Master Boot Record	
MFT	Master File System	Fișier de informații asupra fișierelor de pe volumul respectiv
MP	Message Passing	Transmisie de mesaje
MPI	Message Passing Inteface	Intefăță în transmisia de mesaje
MORRIS		Numele primului vierme inventat
MS-DOS	Microsoft Disck Operating System	
Mutex	Mutual exclusive	Obiect de sincronizare
NAT	Network Address Translation	Procedura de filtrare a pachetelor
NAT	Network Adress Translation	Firewall în Windows
NESSUS	În UNIX	Program de scanare care determină vulnerabilitățile unei mașini
Network Browser		Aplicație ce permite interconectarea cu alte sisteme UNIX sau WINDOWS
NFS bind		Tip de atac
NTFS	New Tehnology File System	Noua tehnologie a sistemului de fișiere-sistem de organizare a fișierelor
Obiect	În unele limbaje de programare ca C++, JAVA, CORBA	O colecție de variabile legate între ele printr-un set de proceduri numite metode; în loc de fișiere sau documente
OFB	Out Feed Back	Mod de criptare cu reacție la ieșire
Offline	În afara SO	Tip de metodă pentru determinarea blocurilor defecte
ONC	Open Network Computing	Rețea deschisă de computere
Online	În cadrul SO	Tip de metodă pentru determinarea blocurilor defecte
Open Sources	Surse deschise	Comunitatea sistemelor de operare libere
OSI	Open Szstem Interconnect	Standard de transmisie

Overhead	Costul comutării procesului	
Overlay	Suprapunere	
Packet flood	Potop de pachete	Bombardarea unei rețele cu pachete de date
Page daemon	Demon de paginare	Pregătește sistemul pentru evacuarea de pagini
Paging	Paginarea memoriei	
PVM	Parallel Virtual Machine	Biblioteca paralela
Periferic	Dispozitiv aflat în afara UC	Ex: hard disc, imprimanta modem, scanner, CD ROM
PFF	Packet Filtering Firewalls	Firewall-uri cu filtrare de pachete
PID	Process Identifier	Identifier de proces UNIX
Ping flood		Tip de atac
Ping of death		Tip de atac
Pipe	conductă	Un pseudofisier pentru comunicația unidirecțională între două procese
PRAM	Parallel Random Acces Machines	Mașini cu acces întâmplător paralel
Proces	Program în execuție	Sub controlul SO
Proces Switch	Comutarea procesului	
Program Counter	Registru special vizibil utilizatorului	Conține adresa de memorie a următoarei instrucțiuni de utilizat
Proxies	Proxi-uri de aplicații	
RAID	Redundant Array of Independent Discks	O mulțime redundantă de discuri independente
RAM-disc	O porțiune de memorie principală declarată ca disc virtual	
Read ahead	Citește înainte	Tehnică de optimizare a accesului secvențial
RPC	Remote Process Comunication	
RO	Read Only=enumai citire	Segment de cod
SACL	System Acces Control	
Scrambling	Învălmășeală	Alterarea și amestecarea mesajelor
Semafor		Obiect de sincronizare
Shel	În UNIX	Interpretor de comenzi
SID	Security identifier	Identifier de securitate

SIGUSR	În UNIX	Tip de semnal
Smarf		Atac de tipul ICMF asupra adresei broadcast a rețelei
SMI	Stateful Multilayer Inspections	Firewall-uri cu inspecții multistrat
SMTP	Simple Mail Transfer Protocol	Protocol de transfer al mesajelor
Sniffer	Adulmecare	Program de interceptarea traficului de rețea
SO	Sistem de operare	
SPAM	E-Mail spamming	
Spoofing	E-Mail spoofing	Falsificarea adresei expeditorului
Stack Pointer	Registru special vizibil utilizatorului	Pointează pe partea de sus a stivei curente de memorie
Stivă	Structură de date	Funcționează după principiul LIFO
Streamer	Nucleu în spațiul utilizator (analog pipelines-urilor)	Conectează dinamic un proces utilizator la driver
Swapping	Deplasare, mișcare	Deplasarea proceselor de pe memoria principală pe hard disc
Synchronous calls	Apeluri sincrone	Apeluri blocante
SYNCOOKIES		Un mecanism de prevenire a atacurilor DOS
SYNflood	Potop de pachete SYN	Bombardare cu pachete care au setat numai bitul SYN
SYSTEM V	Versiune de UNIX	Se bazează pe Berkeley UNIX
System calls	Apeluri sistem	
System Shadow		Împiedică accesarea fișierului de parole
Task	Proces	
TCB	Trusted Computing Base	Bază de calcul de încredere
TCP	Transmission Control Protocol	Protocol de control al transmisiei pachetelor de date
TCP/IP	Transmission Control Protocol / Internet Protocol	Protocol de control al transmisiei pachetelor de date
Tear drop		Tip de atac TCP
TELNET	Television Netwok	Rețea de televiziune
Thread	Fir de execuție	Subunitate a unui proces

Time out	Timp depășit	Rezultat al atacului SYNflood
TITAN	In UNIX	Program de scanare care determină și corectează vulnerabilitățile unei mașini
Trashing	Devalorizare	Când procesele folosesc mai mult timp pentru paginare decât pentru execuție
Tripwire	În UNIX	Program de auditare
TS	Test and Set	Suport hard pentru validare/invalidare de întreruperi sau instrucțiuni
UC	Unitatea Centrală	
UDP	User Datagram Protocol	Protocol de transmitere a pachetelor de date
URL	Uniform Resource Locator	Adresa unică a unei pagini Web
WWW	World Wide Web	Largă rețea mondială
Web	Rețea, țesătură	Un mare graf orientat de documente care pointează spre alte documente
Web page	Pagină Web	Document în rețeaua Web
Winlogon		Serviciu de securitate in Windows
Zombie	Stare a unui proces	Proces terminat, în aşteptarea terminării părintelui
Zone Allarm Pro	în WINDOWS	Program de instalare a unui firewall

## ANEXE - Sisteme de operare (M. Jalobreanu)

### 1 Interfețe cu utilizatorul

Orice sistem de operare deține o interfață prin intermediul căreia realizează comunicarea cu operatorul uman. Primele sisteme de operare aveau interfețe foarte simple, formate dintr-un set mic de comenzi de bază. Spre exemplu, **CP/M**, un sistem destinat microcalculatoarelor cu procesoare pe 8 biți (de exemplu Z80), avea aproximativ 5 comenzi. Odată cu trecerea timpului, interfețele dintre sistemele de operare și utilizator au devenit din ce în ce mai complexe, oferind mai multe facilități și ușurând munca de configurare și întreținere a calculatoarelor pe care le deservesc.

Pentru a vedea care sunt avantajele și dezavantajele diferitelor sisteme de operare în ceea ce privește interfața cu utilizatorul, să încercăm mai întâi să realizăm o clasificare a interfețelor. În primul rând, trebuie remarcat faptul că *interfață* este orice instrument care permite comunicarea între un sistem de operare și un operator, indiferent dacă acest instrument este de natură *hardware* sau *software*. Din această perspectivă, putem observa următoarele tipuri de interfețe cu utilizatorul:

- **Interfețe hardware.** De exemplu, tastatura unui mic calculator de buzunar poate fi considerată o interfață *hardware*.
- **Interfețe software.** Acestea sunt reprezentate de sisteme de programe care, sub o formă sau alta, inițiază și întrețin un dialog cu utilizatorul calculatorului, în scopul utilizării și / sau configurației acestuia. Ele formează cvasitotalitatea interfețelor cu utilizatorul incluse în sistemele de operare, deci ne vom ocupa numai de ele. Interfețele *software* pot fi:
  - **Monitoare.** Unele calculatoare conțin, stocat într-o memorie ROM internă, un program numit *monitor*, care se lansează automat la pornirea calculatorului și îi permite utilizatorului să efectueze operații simple asupra sistemului de calcul, cum ar fi: inspectarea și modificarea regiștrilor procesorului, vizualizarea și alterarea conținutului memoriei etc. De obicei, programul monitor pornește în cazul în care nu a putut fi încărcat sistemul de operare.
  - **Interfețe în linie de comandă (sau interfețe text).** Acestea sunt reprezentate, în general, de un program numit *interpretor de comenzi*, care afișează pe ecran un prompter, primește comanda introdusă de operator și o execută. Comenzile se scriu folosind tastatura și pot fi însoțite de parametri. Aproape toate sistemele de operare includ o interfață în linie de comandă, unele foarte bine puse la punct (cazul sistemelor Unix) iar altele destul de primitive (MS-DOS și MS-Windows).
  - **Interfețe grafice.** Sunt cele mai populare interfețe cu utilizatorul și se prezintă sub forma unui set de obiecte grafice (de regulă suprafete rectangulare) prin intermediul căror operatorul poate comunica cu sistemul de operare, lansând aplicații, setând diferite opțiuni contextuale etc. Dispozitivul cel mai folosit în acest caz este mouse-ul, de aceea acest tip de interfață este utilă în primul rând utilizatorilor neexperimentați și neprofesioniștilor.

#### *Avantaje și dezavantaje ale diferitelor categorii de interfețe*

Tabelul următor prezintă, comparativ, caracteristicile interfețelor cu utilizatorul.

Interfață în linie de comandă	Interfață grafică
-------------------------------	-------------------

<p><b>Avantaje:</b></p> <ul style="list-style-type: none"> <li>• Permite scrierea clară și explicită a comenziilor, cu toți parametrii bine definiți</li> <li>• Oferă flexibilitate în utilizare</li> <li>• Comunicarea cu sistemul de operare se face rapid și eficient</li> </ul> <p><b>Dezavantaje:</b></p> <ul style="list-style-type: none"> <li>• Operatorul trebuie să cunoască bine comenziile și efectele lor</li> <li>• Este mai greu de utilizat de către neprofesioniști</li> </ul>	<p><b>Avantaje:</b></p> <ul style="list-style-type: none"> <li>• Este intuitivă și ușor de folosit</li> <li>• Poate fi utilizată și de către neprofesioniști</li> <li>• Creează un mediu de lucru ordonat</li> <li>• Permite crearea și utilizarea de aplicații de complexe, precum și integrarea acestora în medii de lucru unitare</li> </ul> <p><b>Dezavantaje:</b></p> <ul style="list-style-type: none"> <li>• Anumite operații legate, de exemplu, de configurarea sistemului pot să nu fie accesibile din meniurile și ferestrele interfeței grafice</li> <li>• Interfața ascunde anumite detalii legate de preluarea și execuția comenziilor</li> <li>• Folosește mai multe resurse și este mai puțin flexibilă decât interfața în linie de comandă</li> </ul>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Interfețele cu utilizatorul în sistemele Windows și UNIX

### 1. Windows 95

În sistemul de operare Windows 95 interfața este, prin excelență, una grafică. Ea poate fi recunoscută în primul rând după bara de aplicații (*task bar*) așezată de obicei în partea de jos a ecranului și după *desktop*-ul pe care se pot plasa ușor iconuri reprezentând aplicații sau grupuri. Cu toate că există posibilitatea modificării aspectului anumitor elemente de interfață, mai ales prin instalarea de aplicații de tip *Microsoft Plus*, utilizatorul nu poate să-și aleagă singur interfața pe care o dorește, fiind obligat să o accepte aşa cum este. Trebuie remarcat, însă, faptul că ea este bine proiectată, plăcută în utilizare și funcțională.

Activitatea de configurație a sistemului de operare Windows 95 se face exclusiv folosind aplicații specializate. Majoritatea dintre ele pot fi găsite în grupul *Control Panel*, care poate fi deschis din meniul Start/Settings. Iată, în continuare, câteva dintre ele, pe categorii:

- **Configurarea afișării.** Se face cu aplicația *Display* din *Control Panel*; poate fi apelată și apăsând pe desktop butonul din dreapta al mouse-ului și selectând *Properties*. Se pot configura: rezoluția de afișare, culorile, fonturile folosite, *screen-saver*-ele, fundalul etc.
- **Opțiunile Multimedia.** Se folosește *Control Panel->Multimedia*.
- **Instalarea și dezinstalarea de aplicații.** *Control Panel->Add/Remove Programs*.
- **Configurarea interfeței de rețea.** *Control Panel->Network* sau clic dreapta + *Properties* pe iconul *Network Neighborhood*, permite instalarea placii de rețea, configurația protocoalelor utilizate etc.
- **Inspectarea caracteristicilor și configurația componentelor hardware.** Se face din *Control Panel->System* sau clic dreapta + *Properties* pe iconul *My Computer*. Este afișată versiunea de Windows, informațiile de înregistrare, precum și tipul procesorului folosit și dimensiunea memoriei RAM. Selectând *Device Manager*, se pot vedea și modifica informațiile despre dispozitivele hardware din calculator. *Hardware Profile* permite declararea de configurații

hardware diferite, în funcție de calculatorul pe care este instalat sistemul, iar *Performance* afișează un diagnostic al funcționării sistemului și permite modificarea unor setări avansate legate de sistemul de fișiere, acceleratorul grafic și memoria virtuală.

## 1.1 Interfața în linie de comandă în Windows 95

### 1.1.1. Comenzi MS-DOS

Modul "linie de comandă" al sistemului Windows 95 este foarte sărac în facilități și constă, de fapt, în setul de comenzi introdus de către sistemul de operare MS-DOS. De altfel, MS-DOS este livrat împreună cu Windows și este integrat cu acesta, pentru asigurarea compatibilității cu aplicațiile mai vechi.

Principalele comenzi MS-DOS sunt:

- **copy sursă destinație** - Copiază fișierul/fișierele indicate de *sursă* în *destinație*. Dacă *destinație* este un director, fișierele vor fi copiate în directorul respectiv. Dacă *destinație* lipsește, se copiază în directorul curent. Dacă în loc de sursă este dat fișierul special cu numele CON (reprezentând consola), se citește un text de la tastatură până când utilizatorul introduce CTRL-Z urmat de ENTER, iar textul citit se salvează ca fișier în *destinație*. Un caz particular este concatenarea de fișiere. Comanda

```
copy fis1.txt + fis2.txt + fis3.txt dest.txt
```

realizează concatenarea fișierelor *fis1.txt*, *fis2.txt*, *fis3.txt* și salvarea rezultatului într-un fișier cu numele *dest.txt*.

- **md nume** - Creează un director cu numele *nume*.
- **dir [nume]** - Dacă *nume* e director, afișează toate fișierele conținute în el; dacă e fișier, afișează informații doar despre acel fișier.
- **cd nume** - Schimbă directorul curent în *nume*.
- **rd nume** - Șterge directorul *nume*.
- **del nume** - Șterge fișierul cu numele dat.
- **ren vechi nou** - Redenumește *vechi* în *nou*.
- **type fișier** - Afișează conținutul fișierului.
- **sort [ / R ]** - sortează pe linii, în funcție de codurile ASCII ale caracterelor componente, în ordine crescătoare sau, dacă se folosește /R, în ordine descrescătoare. Datele de intrare se iau de la intrarea standard, iar rezultatul va fi scris la ieșirea standard.

**Apelați comenzile MS-DOS cunoscute cu parametrul /H, studiați parametrii pe care îi acceptă și încercați diferite variante.**

**Observație:** Numele de fișiere pot fi exprimate, în cazul tuturor comenziilor, și sub forma de *tipare* generale. Acestea sunt construcții care identifică o mulțime de fișiere și conțin caracterele '\*' și '?'. Caracterul '\*' înlocuiește apariția a zero sau mai multe caractere în numele de fișier astfel specificat, iar '?' înlocuiește *exact o apariție* a unui caracter în numele fișierului. De exemplu, construcția **\*.txt** identifică toate fișierele din directorul curent care au extensia **txt**, iar **a\*c???.doc** se potrivește cu toate fișierele ale căror nume încep cu litera **a**, după care există zero sau mai multe caractere, apoi litera **c**, urmată de încă exact două caractere și au extensia **doc** (exemplu: **arcul.doc**, **acul.doc**, **aracet.doc**, ...). Numele fișierelor MS-DOS pot avea cel mult 8 caractere, urmate de o extensie de maxim 3 litere. În Windows, această limitare a fost înălțată, numele putând avea oricâte caractere sau "extensii".

## 2. UNIX

### 2.1 Configurare. Interfețe grafice

După cum se știe, există mai multe versiuni ale sistemului de operare UNIX, atât comerciale (Sun OS, Sun Solaris, SCO Unix, HP-UX etc) cât și distribuite gratuit (Linux, BSD). Variațiile de la o versiune la alta (în ceea ce privește implementarea și funcționarea lor) sunt, în unele cazuri semnificative, însă, în general, asemănările între diferitele variante sunt mult mai mari decât deosebirile.

Configurarea unui sistem Unix se face în primul rând prin intermediu *fișierelor de configurare*. Acestea sunt întotdeauna fișiere text, cu un format care le face ușor de citit și de modificat, în care se memorează informații referitoare la proprietățile obiectului configurat. Spre exemplu, în Linux, fișierul **/etc/passwd** conține informațiile despre utilizatorii din sistem, parolele lor, directorul de lucru etc.:

```
root:xEwOwyFNfi6PM:0:0:root:/bin/bash
bin:*:1:1:bin:/bin:
daemon:*:2:2:daemon:/sbin:
adm:*:3:4:adm:/var/adm:
lp:*:4:7:lp:/var/spool/lpd:
sync:*:5:0:sync:/sbin:/bin/sync
shutdown:*:6:0:shutdown:/sbin:/sbin/shutdown
halt:*:7:0:halt:/sbin:/sbin/halt
mail:*:8:12:mail:/var/spool/mail:
news:*:9:13:news:/var/spool/news:
uucp:*:10:14:uucp:/var/spool/uucp:
operator:*:11:0:operator:/root:
games:*:12:100:games:/usr/games:
gopher:*:13:30:gopher:/usr/lib/gopher-data:
ftp:*:14:50:FTP User:/home/ftp:
nobody:*:99:99:Nobody/:
gica:aEezwzFnf56J1:500:500:Georgica,,,:/home/gica:/bin/bash
ioana:ncg3wyFNjd7dM:501:501:Ioana X.,,,,:/home/ioana:/bin/tcsh
xy9679:hszeEDw3ioS91:502:502:Student X. Y.,,,,:/home/xy9679:/bin/ksh
```

Pe lângă utilizatorii propriu-zisi, în acest fișier apar și aşa-numiții utilizatori speciali, a căror existență e cerută de sistem și de către unele aplicații. Pentru fiecare utilizator, apar, în ordine, separate prin ':', următoarele câmpuri: numele de cont, parola criptată, identificatorul numeric al utilizatorului (*user id*), identificatorul grupului din care face parte utilizatorul, numele complet al utilizatorului, directorul său de lucru (*home directory*) și interpretorul de comenzi pe care îl folosește.

**Analizați fișierele de configurare standard care se găsesc în directorul /etc. Urmăriți rolul lor și modul în care se precizează setările. (Se vor analiza doar aspectele esențiale, pentru a vă forma o idee sumară despre aceste fișiere)**

În ceea ce privește interfața cu utilizatorul, trebuie făcute câteva precizări. În primul rând, toate versiunile de Unix moderne oferă atât interfețe grafice cât și interfețe în linie de comandă. În plus, acestea sunt foarte bine integrate una cu cealaltă; practic, nu se poate spune că Unix ar fi în primul rând în linie de comandă, dar nici că este bazat pe o interfață grafică. În al doilea rând, în toate sistemele Unix, *interfața grafică poate fi aleasă de către utilizator* dintr-o ofertă bogată, în funcție de preferințele acestuia.

Arhitectura sistemului de operare este cea care conferă această flexibilitate. Dacă în Windows modul linie de comandă reprezintă, de fapt, utilizarea unui subset de funcții sistem specific unui sistem de

operare mai vechi (MS-DOS), în Unix se folosesc aceleași servicii sistem, indiferent de tipul interfeței folosite. De asemenea, spre deosebire de Windows, unde interfața grafică este strâns legată de nucleul sistemului de operare, Unix oferă o modularizare a acestei funcționalități care îi conferă independență. Pentru ca lucrurile să fie mai clare, să vedem care sunt aceste module din Unix. Există, înainte de toate, sistemul de operare propriu-zis, cu nucleul său și funcțiile aferente. Apoi există, ca o componentă separată, sistemul numit *X Window*, care implementează operațiile grafice de bază necesare unei interfețe bazată pe ferestre, lucrând direct cu adaptorul video din calculator și, în sfârșit, deasupra acestui modul, se află *gestionarul de ferestre* care realizează interfața grafică propriu-zisă. Această ultimă componentă este cea care, de obicei, poate fi aleasă de către utilizator. Există multe astfel de gestionare de ferestre, comerciale sau gratuite, prezente sub forma unor aplicații de sine stătătoare. Fiecare oferă o interfață specifică (aspectul ferestrelor și al componentelor grafice), introducând facilități proprii. În plus, există și o serie de biblioteci de elemente grafice (de exemplu *Motif*) care oferă programatorului diferite componente cu care să construiască interfața grafică a aplicațiilor pe care le implementează.

În continuare, se vor da câteva exemple de interfețe grafice și fișierele lor de configurare. Pentru fiecare dintre ele există un fișier de configurare global în sistem și câte unul (optional) pentru fiecare utilizator. Dacă un utilizator decide că dorește să-și personalizeze setările pentru interfața pe care o folosește, creează în directorul său gazdă fișierul de configurare corespunzător. Sistemul va lua în considerare acest fișier, ignorându-l pe cel global. De obicei utilizatorul nu creează de la început întregul fișier de configurare, ci îl copiază în directorul său pe cel global și îl modifică. Aspectele evidențiate mai jos sunt valabile pentru sistemul de operare *Linux* în varianta RedHat 5.0. În general, o citire atentă a fișierelor de configurare originale livrate cu managerul de ferestre respectiv este suficientă pentru a înțelege cum trebuie făcute modificările în aceste fișiere și ce facilități pot fi configurate.

- **fvwm** - Este o interfață simplă și eficientă, foarte configurabilă. Permite împărțirea spațiului de lucru (*desktop*) în mai multe pagini, fiecare din pagini ocupând întreg ecranul. Fișierul global de configurare se află în directorul **/etc/X11/fvwm** și se numește [system.fvwmrc](#). Utilizatorii își pot crea în directorul propriu un astfel de fișier, denumind-u **.fvwmrc**. Iată câteva din posibilele setări:

- **Culori:**

HiForeColor	Black
HiBackColor	SpringGreen3
#PagerBackColor	#5c54c0
#PagerForeColor	orchid
StickyForeColor	Black
StickyBackColor	SeaGreen
 # Menu colors	
MenuForeColor	Black
MenuBackColor	grey
MenuStippleColor	SlateGrey

- **Selectarea ferestrelor**

```
# Fereastra selectată se suprapune automat pe celelalte,după o perioadă
```

# de 770 ms.

```
AutoRaise 770
# Dacă această opțiune e comentată, fereastra primește focus-ul (se
selectează)
# imediat ce cursorul mouse-ului intră în interiorul său.
#ClickToFocus
```

- ***Setări referitoare la iconuri***

```
# StubbornIcons face ca iconurile să se "de-iconifice"
# în poziția lor originală
# în loc să o facă pe pagina curentă
StubbornIcons

# Cu AutoPlacement, iconurile se vor plasa sub ferestrele
# active. Optiunea următoare schimbă acest lucru.
StubbornIconPlacement

# Iconurile vor "urmări" pagina curentă:
StickyIcons
```

Există extensii ale acestei interfețe, una dintre ele, **fvwm2**, fiind configurată în RedHat 5.0 să semene cu interfața Windows 95.

- **twm** - Oferă o interfață simplă. Fișierul de configurare global este </etc/X11/twm/system.twmrc>, iar cel utilizator **.twmrc**.
- **afterstep** - Interfața este prietenoasă și are un aspect plăcut. Configurarea seamănă cu cea de la **fvwm**, iar fișierele de configurare sunt: </etc/X11/system.steprc> și, respectiv, **.steprc**. Versiunile mai noi nu mai folosesc aceste fișiere, introducând metode de configurare prin intermediul unor meniuri și opțiuni, introduse în interfață.

Cel mai simplu mod de a alege care dintre aceste interfețe va fi folosită este crearea în directorul utilizatorului a unui fișier *executabil* cu numele **.Xclients**, care va conține linia

### **exec interfață**

(de exemplu, **exec afterstep**). Pentru setarea flag-ului *executabil* pentru acest fișier, se va folosi comanda

### **chmod +x .Xclients**

**Încercați diferite interfețe grafice dintre cele instalate în sistemul pe care lucează. După modificarea fișierului **.Xclients**, trebuie să reporniți sistemul grafic (de exemplu, să ieșiți folosind meniul *Exit* și să intrați din nou).**

## **2.2 Comenzi UNIX**

În modul linie de comandă, Unix prezintă mult mai multe facilități decât MS-DOS. Interpretorul de comenzi (care pornește după introducerea numelui utilizatorului și a parolei) poate fi ales de către utilizator. Există mai multe interpretoare clasice, fiecare răspunzând anumitor cerințe. Interpretorul "standard" în Unix este **sh**, dar foarte folosite sunt și **bash**, **tcsh**, **ksh**, **csh**.

Comenzile Unix sunt de fapt programe executabile care pot fi găsite în directoarele **/bin**, **/usr/bin**. Diferențele între interprotoarele de comenzi (numite și **shell**-uri) se văd mai ales în contextul fișierelor de comenzi. Practic, aceste interprotoare permit scrierea de adevărate programe, complexe, folosind comenzile Unix și directivele speciale.

Iată, în continuare, principalele comenzi Unix (fișierele de comenzi urmând a fi tratate mai târziu):

### **man [opțiuni] [secțiune] comandă**

Este una dintre cele mai importante comenzi. Ea are ca efect afișarea paginii de manual care descrie comportarea comenzi date ca argument. Manualul este împărțit în secțiuni, numerotate de la 1 la 8. Cele mai importante secțiuni sunt:

- **Secțiunea 1** - Descrie comenzile standard
- **Secțiunea 2** - Funcțiile sistem Unix, apelabile în C
- **Secțiunea 3** - Funcțiile de bibliotecă C
- **Secțiunea 5** - Formate ale diferitelor fișiere specifice sistemului UNIX
- **Secțiunea 8** - Comenzi de administrare a sistemului, cu privilegii sporite

Pentru fiecare secțiune există câte un capitol introductiv care descrie conținutul acesteia. Pentru a vedea, de exemplu, introducerea la secțiunea 4, se scrie:

#### **man 4 intro**

Este, de asemenea, posibilă citirea unei pagini de manual referitoare chiar la comanda **man**:

#### **man man**

#### **pwd**

tipărește numele directorului curent

#### **cd director**

schimbă directorul curent

#### **ls [-adgilrst] fișier ...**

afișează conținutul unui director

#### **mkdir director**

creează directorul cu numele precizat

#### **rmdir director**

șterge directorul precizat, dacă este gol.

#### **cat <lista de fișiere>**

afiseaza la fișierul standard de ieșire conținutul fișierelor date în listă (se pot astfel concatena mai multe fișiere)

#### **lpr [optiuni] fișier**

tipărește la imprimantă fișierele date

### **wc [opțiuni] [fișiere]**

numără caractere, cuvinte și linii în fișiere

### **grep [opțiuni] [șir de caractere] fișier**

filtrează liniile unui fișier, căutând şabloane

### **find director ... condiție**

se caută în directoarele date și în subdirectoarele lor fișierele care satisfac *condiție*. De exemplu, comanda **find . -print** afișează toate fișierele din subarborele curent.

### **od [opțiuni] fișier**

afișează conținutul unui fișier în diferite formate: octal, zecimal, hexa, ASCII etc.

### **rm [opțiuni] fișier ...**

șterge fișierele date

### **mv fișier1 fișier2**

### **mv fișier director**

schimbă numele *fișier1* în *fișier2* sau mută *fișier* în directorul *director*

### **cp fișier1 fișier2**

### **cp fișier ... director**

copiază *fișier1* în *fișier2* sau *fișier* în *director*

### **In fișier1 [fișier2]**

creează o nouă intrare în director pentru fișierul precizat

### **chmod drepturi fișier**

schimbă drepturile de acces ale unui fișier

### **df [sistem de fișiere]**

afișează informații despre un sistem de fișiere (disc, partiție etc): spațiul total, spațiul liber și altele. Sistemul de fișiere poate fi indicat prin directorul în care partiția sau discul respectiv este montat, adică "rădăcina" sistemului de fișiere. Mai multe detalii despre sistemele de fișiere, structura lor și modul de montare pot fi găsite în [Lucrarea 3](#).

### **du [opțiune] [director]**

afișează informații privind spațiul ocupat de fișierele dintr-un director (inclusiv subdirectoarele sale). **du .** afișează aceste informații pentru directorul curent.

### **ps [opțiuni] [proces]**

afișează starea proceselor care rulează pe sistem la momentul curent. Un proces este, în principiu, un program aflat în execuție, împreună cu stiva sa și zona de date proprie.

#### **kill proces**

oprirea unui proces în curs de execuție

#### **test expresie**

se evaluatează *expresie*, iar dacă valoarea ei este *adevărat* se returnează **0**. Condițiile se referă la fișiere, directoare, şiruri etc.

#### **date**

afișează data și ora curente

#### **who [am I]**

afișează numele utilizatorilor conectați la sistem

#### **echo [-n] argumente**

afișează argumentele date

#### **login [utilizator]**

intrare în sistem

#### **logout**

ieșire din sistem

## 2. Fișiere de comenzi

### 1. Înlănțuirea comenziilor

În Unix, dar și, parțial, în DOS, majoritatea comenziilor folosesc așa-numitele *fișiere standard de intrare și fișiere standard de ieșire*. Acestea sunt concepte abstrakte care reprezintă sursa din care comenziile își iau datele de intrare, respectiv destinația în care ele scriu rezultatele. Deci comenziile citesc din intrarea standard și scriu în ieșirea standard. În mod normal, intrarea standard este reprezentată de tastatura calculatorului, iar ieșirea standard de către dispozitivul de afișare (monitorul).

**Exemplu:** comanda **sort** (existentă atât în DOS cât și în UNIX) funcționează după principiul enunțat. Dacă este apelată fără nici un parametru, ea va aștepta introducerea liniilor de text de la tastatură (*intrarea standard*), până la introducerea caracterului **^Z** urmat de **Enter** în MS-DOS, sau a caracterului **^D**, în Unix, după care va sorta liniile și le va afișa în ordine pe ecran (*ieșirea standard*).

Intrarea și ieșirea standard pot fi schimbată folosind operatorii de **redirectare**. Redirectarea "conectează" intrarea sau ieșirea comenziilor la un fișier dat. Pentru redirectarea intrării se folosește operatorul '**<**', iar pentru redirectarea ieșirii operatorul '**>**'.

**Exemplu:** comanda următoare preia liniile care trebuie sortate din fișierul **date.txt**, iar rezultatele vor fi afișate pe ecran. Se redirecțiază, deci, numai intrarea standard:

**sort < date.txt**

Pentru a redirecta numai ieșirea, adică liniile de text să fie citite de la tastatură, dar rezultatul să fie scris într-un fișier, se folosește următoarea formă:

**sort > ordonat.txt**

Redirectările se pot combina, astfel încât liniile să fie citite dintr-un fișier, iar rezultatul să fie scris în altul:

**sort < date.txt > ordonat.txt**

**Încercați aceste exemple.**

Comenziile se pot și *înlăntui*, în sensul că ieșirea generată de una devine intrare pentru alta. Pentru aceasta, se folosește operatorul '|', numit uneori operatorul *pipe* (conductă).

**Exemplu:** Comanda **more** realizează afișarea pagină cu pagină a datelor citite din intrarea standard. O construcție de forma:

**ls | more**

face ca ieșirea lui **dir** să fie legată la intrarea lui **more**, astfel încât, efectul va fi afișarea pagină cu pagină a fișierelor din directorul curent.

Se pot înlăntui oricâte comenzi și, prin urmare, pentru afișarea pagină cu pagină, ordonate alfabetic, a tuturor fișierelor din directorul curent, se folosește comanda:

**ls | sort | more**

**Încercați acest exemplu.**

**Observație:** Toate aceste considerente sunt valabile atât în DOS, cât și în UNIX.

## 2. Fișiere de comenzi

### 2.1 Fișiere de comenzi MS-DOS

Interpretorul de comenzi MS-DOS permite crearea și folosirea așa-numitelor *fișiere de comenzi indirecte*. Acestea sunt, de fapt, fișiere care conțin înlăntuirile de comenzi MS-DOS și directive speciale pentru executarea de către interpretor a unor sarcini ceva mai complexe. Modul de realizare a acestor fișiere seamănă cu scrierea de programe într-un limbaj de programare.

Înainte de a trece la descrierea directivelor utilizate în fișierele de comenzi, trebuie să explicăm noțiunea de *variabilă de mediu*. Interpretorul de comenzi deține o zonă de memorie specială în care pot fi păstrate diferite variabile. Aceste variabile sunt numite *variabile de mediu* și pot fi accesate prin intermediul comenzi **set**. Definirea unei variabile se face astfel:

**set nume=valoare**

atribuindu-se variabilei *nume* valoarea *valoare*. Valoarea unei variabile este întotdeauna un *șir de caractere*. Pentru accesarea valorii unei astfel de variabile din interiorul fișierelor de comenzi sau din

linie de comandă se folosește construcția `%nume%`. Iată un scurt exemplu: fie o variabilă `a` căreia dorim să-i atribuim valoarea `abcde`. Acest lucru se face astfel:

```
set a=abcde
```

Pentru a afișa valoarea acestei variabile se scrie:

```
echo %a%
```

Există un set de variabile de mediu a căror semnificație este predefinită. De exemplu, variabila **PATH** descrie căile (directoarele) în care interpreterul de comenzi caută programele executabile atunci când utilizatorul nu precizează explicit locația la care se află programul pe care dorește să îl lanseze. Un alt exemplu este variabila **DIRCMD** care stochează parametrii pe care interpreterul îi adaugă fiecărei comenzi **dir** pe care o întâlnește. Dacă scriem `set DIRCMD=/p`, atunci orice comandă **dir** viitoare se va comporta ca și cum ar fi fost introdusă sub forma `dir /p`, deci va afișa numele fișierelor pagină cu pagină.

**Apelați comanda `set` fără parametri și analizați variabilele de mediu definite în sistem.**

**Directive** utilizabile în fișiere de comenzi sunt:

**goto etichetă**

Se execută salt necondiționat la eticheta *etichetă*. O etichetă este un sir de maxim 8 caractere, precedat de ':' Exemplu:

```
:salt
rem ciclu infinit
goto salt
```

**call fișier de comenzi**

Se execută fișierul de comenzi precizat (extensia implicită a numelui fișierului este `.bat`), după care execuția revine la linia de după **call**. Se poate lansa în execuție un fișier de comenzi fără a se mai reveni în fișierul inițial; aceasta se face scriind *numele* fișierului la sfârșitul celui inițial (pe ultima linie).

**if [not] condiție comandă**

Este singura instrucțiune condițională acceptată de MS-DOS. Efectul este executarea comenzi *comanda* dacă a fost îndeplinită condiția *condiție*. Condiția poate fi:

**ERRORLEVEL nr**

Orice program, la terminare, poate să returneze un număr prin care indică modul în care și-a încheiat execuția. Acest lucru se poate realiza, de exemplu în programele scrise în C, prin apelarea funcției **exit()** cu un parametru (valoarea de returnat în sistem) sau apelând, în funcția **main()**, instrucțiunea **return** cu parametru întreg. În MS-DOS, valoarea returnată de programe poate fi testată folosind **ERRORLEVEL**. Condiția **ERRORLEVEL nr** este considerată

adevărată dacă *ERRORLEVEL* este mai mare sau egală cu nr. *ERRORLEVEL* va lua valoarea returnată de ultimul program apelat.

*şir1==şir2*

Condiția e îndeplinită dacă şirurile sunt egale. Şirurile *nu* trebuie puse între ghilimele. Iată câteva exemple utile:

**if** %2 == abcde **rem** parametrul al doilea a fost şirul abcde

**if** .%1 == . **rem** nu există parametri în linia de comandă

**EXIST** fișier

Testează dacă există un fișier.

**for %%var in (multime) do comandă**

Variabila *var* ia pe rând valorile din multime, la fiecare pas executându-se *comandă*. În comandă, fiecare apariție a lui *%%var* este înlocuită cu valoarea curentă din multime. Multimea poate conține și nume globale de fișiere, care conțin caracterele '\*' sau '?'.

**pause [comentariu]**

Așteaptă apăsarea unei taste afișând comentariul dat.

**echo on**

**echo off**

**echo text**

**echo on** face ca fiecare linie din fișierul de comenzi să fie afișată în momentul în care e executată. Acest efect poate fi inhibat prin **echo off**. Comanda **echo text** realizează afișarea mesajului *text*.

### **Observații:**

1. Orice comandă poate începe cu caracterul @, caz în care nu mai este afișată la execuție, indiferent de starea setată prin **echo**.
2. Numele fișierelor de comenzi trebuie să aibă extensia **.bat**.
3. Parametrii pe care fișierul de comenzi îi primește în linia de comandă atunci când este lansat pot fi accesăți folosind construcții de tipul **%nr. %1** reprezentă primul parametru, **%2** al doilea și.a.m.d. până la **%9**. Dacă programul din fișierul de comenzi are nevoie de mai mult de 9 parametri în linia de comandă, atunci se poate folosi directiva **shift**. După primul apel al directivei **shift**, **%1** va reprezenta al doilea parametru, **%2** al treilea și.a.m.d. Fiecare nou apel al lui **shift** va deplasa cu o poziție spre dreapta semnificația construcțiilor **%nr**. Deci un al doilea apel va face ca **%1** să fie al treilea parametru, iar **%9** al unsprezecelea.

*Exemplu:* fie programul **exmp.bat**, apelat astfel: **exmp unu doi trei**. Mai jos este prezentat programul și efectul său:

```

@echo off
rem   "Exemplu de preluare"
rem   "a argumentelor"
echo               "%1"
echo               "%2"
set      a=%3
echo %a%

```

C:\>exmp	unu	doi	trei
unu			
doi			
trei			
C:\>			

## 2.1 Fișiere de comenzi UNIX

### 2.1.1 Interpretoare de comenzi

In sistemul de operare Unix exista mai multe interpretoare de comenzi, selectable de catre utilizator. Fiecare interpretor accepta un limbaj specific, astfel ca fisierile de comenzi care pot fi scrise difera in functie de acest limbaj. Interpretorul de comenzi "standard" este **sh**, in Linux el fiind inlocuit cu interpretorul **bash**. In continuare, ne vom referi la comenzile si directivele specifice acestui interpretor, pentru detalii referitoare la celelalte variante putand fi consultate paginile de manual corespunzatoare.

Ca terminologie, in limba engleza interpretorul de comenzi mai este numit **shell**, iar un program (fisier de comenzi) scris in limbajul recunoscut de acesta se numeste **shell script**.

Lansarea in executie a unui fisier de comenzi se face fie tastand direct numele acestuia (el trebuie sa aiba dreptul de executie setat) sau apeland interpretorul de comenzi cu un parametru reprezentand numele fisierului de comenzi (exemplu: **sh fisier**). In UNIX nu exista o "extensie" dedicata care sa identifice fisierile de comenzi, asa ca numele lor pot fi alese liber.

Comenzile UNIX pot fi grupate in liste de comenzi trimise spre executie intepretorului. Ele vor fi executate pe rand, o comanda fiind lansata in executie numai dupa ce comanda anterioara s-a terminat. Listele se formeaza scriind un sir de comenzi separate prin caracterul ';'. Exemplu:

**cd exemplu ; ls -al**

Daca intr-o lista, in loc de separatorul ';' se foloseste separatorul '**&&**', atunci o comanda nu va fi executata decat in cazul in care precedenta s-a terminat cu cod de succes (codul 0). Daca se foloseste '**||**', atunci conditia este ca precedenta sa se fi terminat cu cod de eroare (cod diferit de 0).

De remarcat faptul ca, atunci cand comenziile se inlantuiesc prin caracterul '**|**' (pipe) ele vor fi executate in paralel.

O comanda poate fi lansata si in fundal (in *background*), adica executia ei se va desfasura in paralel cu cea a interpretorului de comenzi, acesta afisand promptul imediat ce a lansat-o, fara sa-i mai astepte terminarea. Acest lucru se realizeaza adaugand caracterul '**&**' la sfarsitul liniei care contine comanda respectiva. Exemplu:

**du > du.log &**

### 2.1.2 Variabile de mediu

Variabilele de mediu pot sa contin ca valoare un sir de caractere. Atribuirea de valori se face astfel:

**variabila = valoare**

De exemplu

**v=ABCD**

va asigna variabilei cu numele v sirul "ABCD". Daca sirul asignat contine si spatii, el trebuie incadrat intre ghilimele.

Referirea unei variabile se face prin numele ei, precedat de simbolul \$. De exemplu,

**echo \$v**

va determina afisarea textului ABCD.

In UNIX exista cateva variabile predefinite.

- variabile *read-only*, actualizate de interpretor:
- **\$?** - codul returnat de ultima comanda executata
- **\$\$** - identificatorul de proces al interpretorului de comenzi
- **\$!** - identificatorul ultimului proces lansat in paralel
- **\$#** - numarul de argumente cu care a fost apelat fisierul de comenzi curent
- **\$0** - contine numele comenzi executate de interpretor
- **\$1, \$2 ...** - argumentele cu care a fost apelat fisierul de comenzi care se afla in executie
- variabile initializate la intrarea in sesiune:
- **\$HOME** - numele directorului "home" afectat utilizatorului;
- **\$PATH** - caile de cautare a programelor;
- **\$PS1** - prompter-ul pe care il afiseaza interpretorul atunci cand asteapta o comanda;
- **\$PS2** - al doilea prompter;
- **\$TERM** - numele terminalului pe care se lucreaza.

**Apelați comanda `set` fără parametri și analizați variabilele de mediu definite în sistem.**

### 2.1.3 Directive de control

Directivele de control ale interpretorului **sh** sunt structurile de limbaj care pot fi utilizate in scrierea de programe. In continuare vor fi prezentate cateva din cele mai folosite structuri de control.

**Consultati pagina de manual a interpretorului de comenzi bash sau sh si analizati directivele si facilitatile pe care acesta le pune la dispozitie.**

2.1.3.1. Instructiuni de decizie• *Instructiunea if*

```
if lista1
then
else lista2
fi lista3
```

```
if lista1
then
elif lista2
then
else lista3
fi lista4
lista5
```

O comanda returneaza o valoare la terminarea ei. In general, daca o comanda s-a terminat cu succes ea va returna 0, altfel va returna un cod de eroare nenul.

In prima forma a comenzii **if**, se executa *lista1*, iar daca si ultima instructiune din lista returneaza codul 0 (succes) se executa *lista2*, altfel se executa *lista3*.

In a doua forma se pot testa mai multe conditii: daca *lista1* se termina cu succes, se va executa *lista2*, altfel se executa *lista3*. Daca aceasta se termina cu succes se executa *lista4*, altfel se executa *lista5*.

• *Instructiunea case*

```
case cuvant in
tipar1) lista1;;
tipar2) lista2;;
...
esac
```

Aceasta instructiune implementeaza decizia multipla. Sablonul *tipar* este o constructie care poate contine simbolurile ? si \*, similara celor folosite la specificarea generica a numelor de fisiere. Comanda expandeaza (evalueaza) sirul *cuvant* si incearca sa il potriveasca pe unul din tipare. Va fi executata lista de comenzi pentru care aceasta potrivire poate fi facuta.

2.1.3.2. Instructiuni de ciclare• *Instructiunea while*

```
while lista1
do lista2
done
```

Se executa comenzile din *lista2* in mod repetat, cat timp lista de comenzi *lista1* se incheie cu cod de succes.

- **Instructiunea *until***

```
until lista1
do lista2
done
```

Se executa comenzile din *lista2* in mod repetat, pana cand lista de comenzi *lista1* se incheie cu cod de succes.

- **Instructiunea *for***

```
for variabila [in val1, val2 ...]
do lista
done
```

Se executa lista de comenzi in mod repetat, variabila luand pe rand valorile *val1, val2, ...* Daca lipseste cuvantul cheie **in**, valorile pe care le va lua pe rand *variabila* vor fi parametrii din linia de comanda pe care i-a primit fisierul de comenzi atunci cand a fost lansat in executie.

<u>2.1.3.3.</u>	<i>Alte</i>	<i>comenzi</i>
-----------------	-------------	----------------

- **break** - permite iesirea din ciclu inainte de indeplinirea conditiei;
- **continue** - permite reluarea ciclului cu urmatoarea iteratie, inainte de terminarea iteratiei curente;
- **exec cmd** - comenzi specificate ca argumente sunt executate de interpretorul de comenzi in loc sa se creeze procese separate de executie; daca se doreste rularea comenziilor in procese separate ele se scriu direct, asa cum se scriu si in linia de comanda
- **shift** - realizeaza deplasarea argumentelor cu o pozitie la stanga (\$2\$1, \$3\$2 etc);
- **wait [pid]** - permite sincronizarea unui proces cu sfarsitul procesului cu pid-ul indicat sau cu sfarsitul tuturor proceselor "fii";
- **expr expresie** - permite evaluarea unei expresii.

## 2.1.4 Substitutia comenziilor

Atunci cand intr-un *shell script* o comanda este incadrata de caractere ` (accent grav), interpretorul de comenzi va executa comanda, dupa care rezultatul acesteia (textul) va substitui locul comenzi in program. De exemplu, comanda

```
director=`pwd`
```

va atribui variabilei *director* rezultatul executiei comenзii **pwd**, adica sirul de caractere ce contine numele directorului curent.

Un exemplu de utilizare a substitutiei este construirea de expresii aritmetice:

```
contor=1
contor=`expr $contor+1`
```

Aceasta sevenita initializeaza o variabila *contor* la valoarea 1 (sir de caractere !) si apoi o "incrementeaza", in sensul ca la sfarsit, ea va contine sirul de caractere "2".

### 2.1.5 Exemple

1.

```
while           test          -r          fisier
do             sleep         5
done
```

Programul testeaza daca "fisier" este accesibil la citire; in caz afirmativ programul se suspenda 5 secunde.

2.

```
until           test          -r          fisier
do             sleep         5
done
```

Programul se suspenda cate 5 secunde cat timp "fisier" nu este accesibil la citire.

3.

```
contor=$#
cmd=echo
while test $contor -gt 0
do
cmd="$cmd \$\$contor"
contor=`expr $contor - 1`
done
eval $cmd
```

Programul realizeaza tiparirea argumentelor cu care a fost apelat, in ordine inversa. Se executa cate un ciclu pentru fiecare argument, incepand cu ultimul. Argumentul prelucrat este indicat de variabila *contor*, care pleaca de la valoarea \$# (numarul de argumente) si se decrementeaza la fiecare parcurgere a ciclului. Programul construieste o comanda echo, la care adauga argumentele in ordine inversa (linia 5); sirul \\$\$contor are ca scop sa creeze un text format din \$ si valoarea curenta a lui

contor (\ semnifica faptul ca \$ care urmeaza trebuie luat ca atare si nu ca si caracterul de inceput al numelui unei variabile).

4.

```
contor=$#
cmd=echo
while true
do
cmd="$cmd \$contor"
contor=`expr $contor - 1`
if test $contor -eq 0
then break
fi
done
eval $cmd
```

5.

```
contor=$#
cmd=echo
while true
do
cmd="$cmd \$contor"
contor=`expr $contor - 1`
if test $contor -gt 0
then continue
fi
eval $cmd
exit
done
```

6.

```
if test $# -eq 0
then ls -l | grep '^d'
else for i
do
for j in $i/*
do
if test -d $j
then echo $j
fi
done
done
fi
```

### 3. Sisteme de fisiere

Fiecare sistem de operare are un mod propriu de organizare si exploatare a informatiei stocate pe suporturile de memorare fizice. Principiile, regulile si structurile care realizeaza acest lucru compun *sistemul de fisiere* caracteristic sistemului de operare respectiv.

In general, din punctul de vedere al utilizatorului, sistemele de fisiere prezinta o organizare bazata pe conceptele de *fisier* si *director* (*catalog*). Fisierele sunt entitati care incapsuleaza informatia de un anumit tip, iar directoarele grupeaza in interiorul lor fisiere si alte directoare. Orice fisier sau director poate fi identificat prin numele sau, indicat in mod absolut, ca nume de cale sau relativ, fata de directorul curent.

In cazul discurilor fixe (*hard-disk-uri*) si in cel al dischetelor, informatia se memoreaza folosind proprietatile magnetice ale acestora. Hard-disk-ul contine in interior mai multe platane ce pot memora informatie, iar discheta este formata dintr-un singur disc flexibil (cu ambele fete magnetizate). O fata a unui disc este impartita in *piste*, care sunt cercuri concentrice in care poate fi memorata informatia. Pistele sunt impartite la randul lor in *sectoare*, un sector memorand o cantitate fixa de informatie (de obicei 512 octeti). Citirea si scrierea informatiei pe un disc se face la nivel de *blocuri de date*. Un bloc (*cluster*) poate fi format dintr-un singur sector (cum se intampla la dischete) sau din mai multe (ca la hard-disk-uri).

Un hard-disk poate fi impartit de utilizator in *partitii*, fiecare partitie comportandu-se, la nivel utilizator, ca un disc de sine statator. Partitia memoreaza sistemul de fisiere, de unde rezulta ca pe acelasi disc fizic pot fi intalnite mai multe sisteme de fisiere. Pentru calculatoarele personale obisnuite (PC), informatiile referitoare la partitiile se memoreaza la inceputul discului, in asa-numita *tabela de partitii*. Aceasta contine 4 intrari in care memoreaza pozitiile, dimensiunile si tipurile partitiilor de pe disc. Partitiile memorate tabela de la inceputul discului se numesc *partitii primare*, care pot fi, evident, cel mult 4 la numar. Este posibil, insa, ca in interiorul oricarei partitii primare sa se creeze cate o noua tabela de partitii, referind partitiile care fizic se afla in interiorul partitiei curente si care se numesc *partitii extinse*.

In cele ce urmeaza, vom trece in revista principalele caracteristici ale sistemelor de fisiere caracteristice pentru doua sisteme de operare: MS-DOS (Windows) si Unix.

## 1. Sistemul de fisiere in MS-DOS

### 1.1 Organizarea discurilor in MS-DOS

Primul sector al partitiei sau discului care contine sistemul se numeste *sectorul de boot*. Acesta contine urmatoarele informatii:

Offset	Dimensiune (octeti)	Continut
+00h	3	JMP adresa. Salt la rutina de incarcare a sistemului de operare

+03h	8	Numele producatorului si versiunii
+0Bh	2	Numarul de octeti pe sector
+0Dh	1	Numarul de sectoare pe cluster
+0Eh	2	Numarul de sectoare rezervate (inaintea FAT)
+10h	1	Numarul de FAT-uri
+11h	2	Numarul maxim de intrari in directorul radacina
+13h	2	Numarul total de sectoare
+15h	1	Media descriptor
+16h	2	Numarul de sectoare dintr-un FAT
+18h	2	Numarul de sectoare pe pista
+1Ah	2	Numarul de capete de citire/scriere
+1Bh	2	Numarul de sectoare ascunse
+1Dh	...	Codul de bootare

Directoarele sunt memorate ca structuri speciale, ca tabele in care fiecare intrare reprezinta un fisier. De fapt, un director este memorat ca un fisier obisnuit, dar care contine informatii despre alte fisiere. Exista un director radacina, memorat dupa tabela de alocare a fisierelor (FAT), care are o dimensiune limitata.

Structura unei intrari in director este:

Offset	Dimensiune	Continut
+00h	8	Numele fisierului
+08h	3	Extensia numelui de fisier
+0Bh	1	Atribute
+0Ch	0Ah	Rezervat
+16H	2	Ora ultimei modificari a fisierului

+18h	2	Data ultimei modificari a fisierului
+1Ah	2	Numarul primului cluster ocupat de fisier
+1Ch	4	Dimensiunea fisierului (in octeti)

Tabela de alocare a fisierelor (File Allocation Table - FAT)

FAT este o structura care este folosita pentru localizarea datelor care apartin unui fisier. Ea este, de fapt, o structura de tip tablou care memoreaza in interiorul ei liste inlantuite care indica clusterurile ce compun fisierele. Fiecare locatie din FAT are 12 biti la dischete, 16 biti la partitiile MS-DOS obisnuite (FAT16) si 32 biti la partitiile FAT32 recunoscute de catre Windows 95 OSR2 si Windows 98. Primul octet din FAT contine un octet de identificare numit *media descriptor*. Urmatorii 5 octeti (FAT12) sau 7 octeti (FAT16) sau 15 octeti (FAT32) contin valoarea 0FFh.

Celelalte intrari din FAT corespund fiecare unui cluster de pe disc (clusterurile se numara de la spatiul imediat urmator FAT-ului). Astfel, intrarea 1 din FAT corespunde clusterului 1, intrarea 2 clusterului 2, s.a.m.d.

*Fiecare intrare in FAT memoreaza numarul urmatorului cluster din fisierul din care face parte clusterul care corespunde intrarii. Numarul primului cluster al unui fisier este memorat, dupa cum s-a vazut deja, in intrarea in director corespunzatoare fisierului respectiv. Se vede ca numarul de biti pe care este reprezentata o intrare in FAT limiteaza, astfel, numarul maxim de clustere pe disc.*

Exemplu: Fie fisierul abc.txt care incepe in clusterul 5 si fisierul xyz.exe care incepe in clusterul 4. O posibila organizare a spatiului ocupat de aceste fisiere este prezentata mai jos:

### Sistemul de fisiere in UNIX

#### 2.1 Organizarea discurilor in Unix

Spatiul fiecarei partitii Unix contine urmatoarele zone:

Bloc Incarcare	Super-bloc	Zona index	noduri	Swapping	Continut
-------------------	------------	---------------	--------	----------	----------

- **Blocul de incarcare (boot block)**contine programele care realizeaza incarcarea partii rezidente a sistemului de operare Unix.
- **Superblocul** contine informatii generale despre sistemul de fisiere de pe disc: inceputul zonelor urmatoare, inceputul zonelor libere de pe disc.
- **Zona de noduri index** are o dimensiune fixata la crearea sistemului de fisiere si contine cate o intrare pentru fiecare fisier ce poate fi creat pe acest suport
- **Zona pentru swapping** (daca exista) este rezervata pentru pastrarea imaginilor proceselor atunci cand sunt eliminate temporar din memorie pentru a face loc altor procese. De obicei, insa, pentru zona de swap se folosesc partitii distincte.

- Ultima zona contine blocurile care memoreaza fisierele propriu-zise.

*Intrarile in director* au o structura foarte simpla, continand doar doua campuri:

- numele fisierului
- numarul nodului index asociat fisierului

### 2.1.1. Structura nodurilor index

Un nod index (*i-node*) contine informatiile esentiale despre fisierul caruia ii corespunde. Exista cate un singur nod index pentru fiecare fisier. Este posibil sa intalnim mai multe intrari in director indicand acelasi nod index (sistemul de fisiere din Unix accepta crearea de legaturi multiple).

Informatia din nodul index cuprinde:

- **identificatorul utilizatorului:** *uid (user-id.)*. Identifica proprietarul fisierului
- **identificatorul de grup al utilizatorului**
- **drepturile de acces la fisier.** Drepturile sunt de trei tipuri (*r-read, w-write, x-execute*) si sunt grupate pe trei categorii:
  - *user* - drepturile proprietarului fisierului
  - *group* - drepturile utilizatorilor din grupul proprietarului
  - *others* - drepturile tuturor celorlalti utilizatori
- **timpul ultimului acces la fisier**
- **timpul ultimei actualizari a fisierului**
- **timpul ultimului acces pentru actualizarea nodului index**
- **codul fisierului (tipul fisierului)**. Fisierele pot fi: fisiere obisnuite (-), directoare (d), periferice (c) etc.
- **lungimea fisierului (in octeti)**
- **contorul de legaturi al fisierului**. Reprezinta numarul de legaturi existente spre acest nod index. Este utilizat la operatia de stergere a nodului index.
- **lista de blocuri** care contin fisierul

Lista de blocuri de pe disc care contin fisierul se realizeaza printre-un tablou cu 13 intrari. Primele 10 intrari contin direct adresele de bloc (cluster) pentru primele 10 blocuri ale fisierului. A unsprezecea intrare din aceasta lista este adresa unui bloc, rezervat fisierului, al carui continut este, insa, interpretat ca lista de adrese de blocuri. Se spune ca aceste blocuri sunt adresate prin *indirectare simpla*. Intrarea a 12-a contine un bloc al carui continut consta in adrese de blocuri, care *acestea* contin adrese de blocuri de date (*indirectare dubla*). In mod analog, intrarea cu numarul 13 determina o *indirectare tripla*.

## **2.2 Legaturi si fisiere speciale**

Sistemul de fisiere din UNIX permite crearea asa-numitelor legaturi la fisiere. O asemenea legatura (*link*) este vazuta de catre utilizator ca un fisier cu un nume propriu, dar care in realitate refera un alt fisier de pe disc. Orice operatie care se executa asupra fisierului legatura (mai putin stergerea) isi va avea efectul de fapt asupra fisierului indicat de legatura. Daca este solicitata stergerea, efectul depinde de tipul legaturii respective.

Legaturile sunt de doua tipuri:

- **fizice (hard links)**
- **simbolice (symbolic links)**

Legaturile din prima categorie se realizeaza prin introducerea de intrari in director care pointeaza spre acelasi nod index, si anume cel al fisierului indicat. Cand spre fisier este stearsa si ultima intrare in director care il indica, fisierul in sine va fi sters si el. Legaturile de acest tip au dezavantajul ca nu pot indica nume de directoare si nici fisiere din alte partitii decat cea pe care se afla.

Legaturile simbolice sunt de fapt fisiere distincte, marcate cu un cod special, care au ca si continut numele complet al fisierului indicat. Stergerea lor nu afecteaza fisierul. Pot referi directoare, precum si fisiere si directoare din alta partitie sau alt disc, dar au dezavantajul ca pentru ele (fiind fisiere) trebuie creat un nod index separat si, in plus, ocupa spatiu pe disc prin continutul lor.

Crearea legaturilor spre fisiere sau directoare se face cu ajutorul comenzii **ln**.

- **ln *fisier\_indicat nume\_legatura*** - creeaza o legatura "fizica"
- **ln -s *fisier\_indicat nume\_legatura*** - creeaza o legatura simbolica

Pe langa legaturi, in Unix exista si alte fisiere speciale. Tipul acestora poate fi observat citind primul caracter afisat de comanda **ls -l**

Astfel, avem:

1. Fisiere obisnuite
2. Directoare. Dupa cum am vazut, sunt fisiere care, avand un format special, grupeaza fisiere
3. Fisiere speciale care corespund unor dispozitive orientate pe caractere
4. Fisiere speciale care corespund unor dispozitive orientate pe blocuri
5. Fisiere FIFO
6. Legaturi simbolice

Fisierele speciale evidente la punctele 3 si 4 reprezinta metoda prin care sistemul Unix abstractizeaza dispozitivele de intrare-iesire si alte dipozitive din sistemul de calcul. Toate aceste fisiere se gasesc in directorul **/dev**.

Spre exemplu, fiecarei unitati de disc ii corespunde cate un fisier in directorul **/dev**. In Linux, primei unitati de dischete ii corespunde fisierul **/dev/fd0**, celei de-a doua **/dev/fd1**, s.a.m.d. Primului hard-disk cu interfata IDE din sistem ii corespunde fisierul special **/dev/hda**, iar primei sale partitii fisierul **/dev/hda1**. A doua partitie de pe primul disc are ca si correspondent fisierul **/dev/hda2**, al doilea hard-disk IDE se refera cu **/dev/hdb**, s.a.m.d.

### **2.3. Montarea sistemelor de fisiere**

Fisierele speciale care indica unitati de disc sau partitii sunt folosite in operatia numita *montare* a sistemelor de fisiere. Sistemul de operare Unix permite montarea intr-un director a unui sistem de fisiere aflat pe un disc sau o partitie. Aceasta inseamna ca, dupa montare, in directorul respectiv se va afla intreaga structura de fisiere si directoare de pe sistemul de fisiere respectiv. Mecanismul este

deosebit de puternic, deoarece ofera posibilitatea de a avea o structura de directoare unitara, care grupeaza fisiere de pe mai multe partitii sau discuri. Daca se adauga si sistemul de fisiere **NFS** (*Network File System*), aceasta structura de directoare va putea contine si sisteme de fisiere montate de la distanta (de pe alta masina)

Montarea unui sistem de fisiere se face cu comanda **mount**. Data fara nici un parametru, ea afiseaza sistemele de fisiere montate in momentul respectiv in sistem. O alta forma a ei este urmatoarea:

#### **mount fisier-special director**

care monteaza un disc sau o partitie intr-un director dat; sau

#### **mount -t tip fisier-special director**

cu acelasi efect, doar ca se specifica in clar tipul sistemului de fisiere care se monteaza. Diferitele variante de Unix cunosc mai multe sau mai putine tipuri de sisteme de fisiere. Spre exemplu, Linux cunoaste, printre altele, urmatoarele:

- **minix** - sistemul de fisiere al sistemului de operare MINIX
- **ext2** - *Second-Extended File System* - sistemul caracteristic Linux
- **msdos** - sistemul de fisiere DOS FAT16 sau FAT12
- **vfat** - sistemul de fisiere DOS cu extensia pentru nume lungi introdusa de Windows 95
- **iso9660** - sistem de fisiere pentru CD-ROM (cel mai raspandit) cu o serie de extensii ale sale
- **proc** - un sistem de fisiere virtual ale carui componente furnizeaza informatii despre starea sistemului

De obicei, montarea de sisteme de fisiere poate fi facuta numai de catre utilizatorul **root** (cel mai privilegiat utilizator, administratorul sistemului), dar se poate permite si utilizatorilor obisnuiti sa monteze anumite partitii sau unitati de disc.

#### **Exemplu:**

Montarea unei dischete introdusa in prima unitate de dischete, care contine si fisiere cu nume lungi create in Windows se face astfel

#### **mount /dev/fd0 diskA**

unde **diskA** este numele directorului in care se va monta discheta, aflat in directorul curent.

**Important:** Orice sistem de fisiere montat de pe o unitate de disc care permite inlaturarea discului respectiv trebuie *demontat* inainte de a scoate discul. De asemenea, inainte de inchiderea sau repornirea calculatorului, trebuie de-montate si sistemele de fisiere de pe discurile fixe (in Linux, aceasta din urma operatie se efectueaza automat la restartarea sistemului prin apasarea simultana a tastelor Ctrl+Alt+Del). De-montarea fisierelor se face cu comanda

#### **umount fisier-special**

sau

**umount director**

(unde *director* este numele directorului in care a fost montat sistemul de fisiere).

**Scrieti un fisier de comenzi Unix care poate fi apelat din linia de comanda astfel:**

**listall [-l] [-d] [-c] [-b] [director]**

**Programul afiseaza toate fisierile din directorul dat ca argument si din toate subdirectoarele sale, in functie de conditia data ca optiune, astfel:**

- **-l afiseaza fisierile de tip legatura**
- **-d afiseaza directoarele**
- **-c afiseaza dispozitivele speciale pe caracter**
- **-b afiseaza dispozitivele speciale orientate pe blocuri**

In linia de comanda pot sa apara zero sau mai multe din aceste optiuni. Daca ele lipsesc, se vor afisa toate fisierile si directoarele. Daca argumentul *director* lipseste, se va lua in considerare directorul curent.

## 4. Apeluri sistem si functii de biblioteca C pentru lucrul cu fisiere

Orice sistem de operare pune la dispozitia programatorilor o serie de *servicii* prin intermediul carora acestora li se ofera acces la resursele hardware si software gestionate de sistem: lucrul cu tastatura, cu discurile, cu dispozitivul de afisare, gestionarea fisierelor si directoarelor etc. Aceste servicii se numesc *apeluri sistem*. De cele mai multe ori, operatiile pe care ele le pot face asupra resurselor gestionate sunt operatii simple, cu destul de putine facilitati. De aceea, frecvent, se pot intalni in bibliotecile specifice limbajelor de programare colectii de functii mai complicate care gestioneaza resursele respective, dar oferind programatorului niveluri suplimentare de abstractizare a operatiilor efectuate, precum si importante facilitati in plus. Acestea sunt *functiile de biblioteca*. Trebuie subliniat faptul ca functiile de biblioteca cu ajutorul carora se poate gestiona o anumita resursa sunt implementate folosind chiar functiile sistem corespunzatoare, specifice sistemului de operare.

In acest document vor fi prezentate functiile sistem pe care le pun la dispozitie sistemele de operare MS-DOS si UNIX pentru lucrul cu fisiere. Se va presupune in continuare ca limbajul de programare utilizat este C si ca sunt cunoscute toate caracteristicile si facilitatile acestui limbaj. Vor fi trecute in revista, de asemenea, cateva functii de biblioteca C care servesc aceluiasi scop.

## 1. Apeluri sistem pentru lucru cu fisiere

Apelurile sistem care vor fi discutate aici sunt caracteristice atat sistemului de operare UNIX cat si MS-DOS. Operatiile care pot fi aplicate fisierelor folosind aceste functii se refera la deschiderea fisierelor, scrierea si citirea de blocuri de date in si din fisiere, mutarea si copierea fisierelor etc.

Pentru a putea actiona asupra unui fisier, este nevoie inainte de toate de a identifica in mod unic fisierul. In cazul functiilor discutate, identificarea fisierului se face printr-un asa-numit *descriptor de fisier (file descriptor)*. Aceasta este un numar intreg care este asociat fisierului in momentul deschiderii acestuia.

### 1.1. Functiile open si close

Deschiderea unui fisier este operatia prin care fisierul este pregatit pentru a putea fi prelucrat in continuare. Aceasta operatie se realizeaza prin intermediul functiei **open**:

```
int open(const char *pathname, int oflag, [, mode_t mode]);
```

Functia returneaza -1 in caz de eroare. In caz contrar, ea returneaza descriptorul de fisier asociat fisierului deschis.

Parametri:

- **pathname** - contine numele fisierului
- **oflag** - optiunile de deschidere a fisierului. Este, in realitate un sir de biti, in care fiecare bit sau grupa de biti are o anumita semnificatie. Pentru fiecare astfel de semnificatie exista definite in fisierul *header C fcntl.h* cate o constanta. Constantele se pot combina folosind operatorul '|'(sau *logic pe biti*) din C, pentru a seta mai multi biti (deci a alege mai multe optiuni) in parametrul intreg **oflag**. Iata cateva din aceste constante:
  - O\_RDONLY - deschidere numai pentru citire
  - O\_WRONLY - deschidere numai pentru scriere
  - O\_RDWR - deschidere pentru citire si scriere
  - O\_APPEND - deschidere pentru adaugare la sfarsitul fisierului
  - O\_CREAT - crearea fisierului, daca el nu exista deja; daca e folosita cu aceasta optiune, functia **open** trebuie sa primeasca si parametrul *mode*.
  - O\_EXCL - creare "exclusiva" a fisierului: daca s-a folosit O\_CREAT si fisierul exista deja, functia **open** va returna eroare
  - O\_TRUNC - daca fisierul exista, continutul lui este sters
- **mode** - se foloseste numai in cazul in care fisierul este creat si specifica drepturile de acces asociate fisierului. Acestea se obtin prin combinarea unor constante folosind operatorul sau ('|'), la fel ca si la optiunea precedenta. Constantele pot fi:
  - S\_IRUSR - drept de citire pentru proprietarul fisierului (*user*)
  - S\_IWUSR - drept de scriere pentru proprietarul fisierului (*user*)
  - S\_IXUSR - drept de executie pentru proprietarul fisierului (*user*)
  - S\_IRGRP - drept de citire pentru grupul proprietar al fisierului
  - S\_IWGRP - drept de scriere pentru grupul proprietar al fisierului
  - S\_IXGRP - drept de executie pentru grupul proprietar al fisierului
  - S\_IROTH - drept de citire pentru ceilalți utilizatori
  - S\_IWOTH - drept de scriere pentru ceilalți utilizatori
  - S\_IROTH - drept de executie pentru ceilalți utilizatori

Pentru crearea fisierelor poate fi folosita si functia

```
creat (const char *pathname, mode_t mode)
```

echivalenta cu specificarea optiunilor O\_WRONLY | O\_CREAT | O\_TRUNC la functia **open**.

Dupa utilizarea fisierului, acesta trebuie *inchis*, folosind functia

```
int close (int filedes)
```

in care *filedes* este descriptorul de fisier obtinut la **open**.

### 1.2. Functiile read si write

Citirea datelor dintr-un fisier deschis se face cu functia

```
ssize_t read(int fd, void *buff, size_t nbytes)
```

Functia citeste un numar de exact *nbytes* octeti de la pozitia curenta in fisierul al carui descriptor este *fd* si ii pune in zona de memorie indicata de pointerul *buff*. Este posibil ca in fisier sa fie de citit la un moment dat mai putin de *nbytes* octeti (de exemplu daca s-a ajuns spre sfarsitul fisierului), astfel ca functia *read* va pune in buffer doar atatia octeti cati poate citi. In orice caz, *functia returneaza numarul de octeti cititi din fisier*, deci acest lucru poate fi usor observat.

Daca s-a ajuns exact la sfarsitul fisierului, functia returneaza zero, iar in caz de eroare, -1.

Scrierea datelor se face cu

```
ssize_t write(int fd, void *buff, size_t nbytes)
```

Functia scrie in fisier primii *nbytes* octeti din bufferul indicat de *buff*. Returneaza -1 in caz de eroare.

### 1.3. Functia lseek

Operatiile de scriere si citire in si din fisier se fac la o anumita pozitie in fisier, considerata pozitia curenta. Fiecare operatie de citire, de exemplu, va actualiza indicatorul pozitiei curente incrementand-o cu numarul de octeti cititi. Indicatorul pozitiei curente poate fi setat si in mod explicit, cu ajutorul functiei **lseek**:

```
off_t lseek(int fd, off_t offset, int pos)
```

Functia pochiedea indicatorul la deplasamentul *offset* in fisier, astfel:

- daca parametrul *pos* ia valoarea SEEK\_SET, pozitionarea se face relativ la inceputul fisierului
- daca parametrul *pos* ia valoarea SEEK\_CUR, pozitionarea se face relativ la pozitia curenta
- daca parametrul *pos* ia valoarea SEEK\_END, pozitionarea se face relativ la sfarsitul fisierului

Parametrul *offset* poate lua si valori negative si reprezinta deplasamentul, calculat in octeti.

In caz de eroare, functia returneaza -1.

#### 1.4. Alte functii

- `int mkdir(const char *pathname, mode_t mode)` - creeaza un director
- `int rmdir(const char *pathname)` - sterge un director

***Observatie: La orice folosire a unei functii sistem este foarte important sa se testeze valoarea returnata de aceasta. Daca apelul functiei s-a incheiat cu eroare, programul trebuie sa recunoasca acest caz si sa actioneze in consecinta, de exemplu prin tiparirea unui mesaj de eroare si eventual terminarea executiei.***

## 2. Functii de biblioteca

In biblioteca standard C exista cateva functii pentru gestionarea fisierelor. Acestea folosesc pentru identificarea fisierelor un descriptor reprezentat de o structura de date, FILE.

### 2.1. Deschiderea si inchiderea unui fisier

`FILE *fopen(const char *filename, const char *mode)`

Functia deschide fisierul indicat prin `filename`, creeaza o structura FILE continand informatii despre fisier si returneaza un pointer catre aceasta. Acest pointer va fi elementul care va putea fi folosit in continuare pentru accesarea fisierului. Parametrul `mode` este un sir de caractere care indica modul de deschidere a fisierului. "r" semnifica deschidere pentru citire, "w" deschidere pentru scriere. Poate fi specificat si tipul fisierului: "t" pentru fisier text, "b" pentru fisier binar. Optiunile pot fi combinate, de exemplu sub forma "r+t".

Inchiderea fisierului se face cu

`fclose(FILE *stream)`

unde `stream` este pointerul spre structura FILE obtinut la deschiderea fisierului.

### 2.2. Operatii asupra fisierelor

- `int fprintf(FILE *stream, const char *format, ...)`; - scriere in fisier cu formatare; sirul de caractere care specifica formatul este similar celui de la instructiunea `printf`.
- `int fscanf(FILE *stream, const char *format, ...)`; - citire din fisier, asemanator cu functia `scanf`.
- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)`; - citeste din fisierul indicat de `stream` un numar de `nmemb` elemente, fiecare de dimensiunea `size`, si le pune in zona de memorie indicata de `ptr`.
- `size_t fwrite( void *ptr, size_t size, size_t nmemb, FILE *stream)`; - scrie in fisierul indicat de `stream` un numar de `nmemb` elemente, fiecare de dimensiunea `size`, pe care le ia din zona de memorie indicata de `ptr`.

Consultati paginile de manual corespunzatoare apelurilor sistem si functiilor de biblioteca discutate, precum si functiilor inrudite cu acestea. Apelurile sistem sunt tratate in

sectiunea 2 a manualului Unix (*man 2 nume\_functie*), iar functiile de biblioteca in sectiunea 3.

## 5. Apeluri sistem si functii de biblioteca C pentru lucrul cu directoare

### 1. Gestionarea directoarelor

Accesul la informatiile memorate in directoare se face cu ajutorul unor functii speciale, cu ajutorul carora se pot afla numele fisierelor continute si informatii despre acestea (dimensiune, data crearii, drepturile aferente etc).

Daca in ceea ce priveste gestionarea fisierelor puteau fi intalnite aspecte comune in cazurile sistemelor de operare DOS si UNIX, modul in care se exploateaza directoarele (din perspectiva programatorului) difera substantial.

#### 1.1. Gestionarea directoarelor in MS-DOS

##### 1.1.1. Functiile *findfirst* si *findnext*

Aceste functii utilizeaza ca intermediar o structura de date numita **struct ffbblk**, care contine informatii despre un anumit fisier:

```
struct ffbblk {
    char ff_reserved[21];
    char ff_attrib; /*atributele fisierului*/
    int ff_ftime; /*timpul crearii fisierului*/
    int ff_fdate; /*data crearii fisierului*/
    long ff_fsize; /*dimensiunea fisierului (in octeti)*/
    char ff_fname[13]; /*numele si extensia fisierului
    (cu punct)*/
}
```

- **int findfirst(const char \*pathname, struct ffbblk \*ffblk, int attrib)**

gaseste primul fisier care se potriveste tiparului dat in *pathname* (tipar care poate contine caracterele \* si ? ) si initializeaza structurile de date folosite intern de catre functiile de cautare de fisiere. Functia va completa o structura **ffbblk** cu informatii despre primul fisier gasit. Adresa acestei structuri trebuie data de catre utilizator in parametrul *ffblk*. Parametrul *attrib* specifica atributele de cautare. (Pentru mai multe detalii, consultati help-urile compilatoarelor de C pentru MS-DOS).

- **int findnext(struct ffbblk \*ffblk)**

Primul fisier care corespunde unui anumit tipar poate fi gasit cu *findfirst*. Urmatoarele fisiere care corespund aceluiasi tipar pot fi gasite apeland succesiv functia *findnext*. Aceasta primeste ca parametru structura **ffblk** folosita la *findfirst*, va gasi fisierul cautat si va completa structura **ffblk** cu informatiile specifice fisierului gasit.

## 1.2. Gestionarea directoarelor in UNIX

### 1.2.1. Aflarea atributelor fisierelor

```
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

Functia *stat* primeste ca parametru un nume de fisier (cale) si returneaza o structura **stat**, care contine informatii despre fisierul respectiv. Structura **stat** are urmatoarea forma:

```
struct stat
{
    dev_t          st_dev;      /* device */
    ino_t          st_ino;      /* inode */
    umode_t        st_mode;     /* protection */
    nlink_t        st_nlink;    /* number of hard
links */
    uid_t          st_uid;      /* user ID of
owner */
    gid_t          st_gid;      /* group ID of
owner */
    dev_t          st_rdev;     /* device type
(if inode device) */
    off_t          st_size;     /* total size, in
bytes */
    unsigned long  st_blksize;  /* blocksize for
filesystem I/O */
    unsigned long  st_blocks;   /* number of
blocks allocated */
    time_t         st_atime;    /* time of last
access */
    time_t         st_mtime;    /* time of last
modification */
    time_t         st_ctime;    /* time of last
change */
};
```

Pointerul care indica zona de memorie in care functia *stat* va returna aceste informatii trebuie dat ca al doilea parametru al functiei. Zona de memorie trebuie in prealabil rezervata (prin *malloc*) pentru a putea memora structura **stat**.

Functia *fstat* are acelasi efect, cu deosebirea ca ea primeste ca argument un descriptor de fisier, si nu numele acestuia, deci se poate aplica doar fisierelor in prealabil deschise.

Functia *lstat* este asemanatoare cu *stat*, cu diferenta ca, daca este aplicata unei legaturi simbolice, informatiile returnate se vor referi la legatura, si nu la fisierul indicat.

### 1.2.2. Functii de biblioteca pentru citirea directoarelor

Directoarele sunt, in esenta, fisiere cu un format special. Sistemul de operare UNIX pune la dispozitia programatorului un set de apeluri sistem care ofera posibilitatea de a citi continutul directoarelor, accesand astfel informatii despre fisierele si directoarele continute. Folosind aceste apeluri sistem, biblioteca standard C defineste un set de functii care se conformeaza standardului POSIX si care ofera aceleasi facilitati. Fiind recomandabil sa se utilizeze, in locul apelarii directe a functiilor sistem, functiile de biblioteca , in continuare vor fi prezentate numai acestea din urma.

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
void rewinddir(DIR *dir);
int closedir(DIR *dir);
```

Conform specificatiilor POSIX, structura **dirent** contine un camp

```
char d_name[];
```

de dimensiune nespecificata, cu cel mult **NAME\_MAX** caractere exclusiv caracterul de terminare (NULL). Folosirea altor campuri existente in aceasta structura dauneaza portabilitatii programului. Campul *d\_name* contine numele unei intrari in director (nume de fisier, director etc.).

Structura **DIR** este o structura utilizata intern de cele patru functii. Structura va fi returnata (initializata) de *opendir* si trebuie transmisa ca parametru celorlalte functii.

Inspectarea fisierelor dintr-un director se face astfel:

- se "deschide" directorul dorit cu functia *opendir*.
- se citeste, pe rand, cate o intrare in director, apeland succesiv functia *readdir*. Fiecare apel al acestei functii va returna o structura **dirent**, in care se vor gasi informatii despre intrarea in director citita. Intrarile in director vor fi parcurse, astfel, una dupa alta, pana cand se ajunge la ultima intrare. In momentul in care nu mai exista intrari in director de citit, functia *readdir* va returna 0, iar in caz de eroare -1. Dupa cum a fost aratat mai sus, singura informatie care poate fi extrasă (conform POSIX) din structura **dirent** este numele intrarii in director. Toate celelalte informatii despre intrarea citita se pot afla apeland in continuare functiile *stat*, *fstat* sau *Istat*.
- in final. directorul este inchis, folosind *closedir*.

**Consultati paginile de manual corespunzatoare acestor functii.**

#### Anexa: Alte functii pentru lucrul cu directoare si fisiere

*Consultati paginile de manual corespunzatoare urmatoarelor functii, precum si cele ale functiilor inrudite:*

- **int link(const char \*oldpath, const char \*newpath);** - creeaza legaturi fixe spre fisiere
- **int symlink(const char \*topath, const char \*frompath);** - creeaza legaturi simbolice spre fisiere sau directoare
- **int unlink(const char \*pathname);** - sterge o intrare in director (legatura, fisier sau director)
- **int rename(const char \*oldpath, const char \*newpath);** - redenumire / mutare de fisiere

- `int rmdir(const char *pathname)`; - stergere de directoare
  - `int chdir(const char *path)`; - schimbarea directorului curent
  - `char *getcwd(char *buf, size_t size)`; - determinarea directorului curent
- 

## 6. Procese

### 1. Concepte de baza

Orice sistem de calcul modern este capabil sa execute mai multe programe in acelasi timp. Cu toate acestea, in cele mai multe cazuri, unitatea centrala de prelucrare (CPU) nu poate executa la un moment dat decat un singur program. De aceea, sarcina de a rula mai multe programe in acelasi timp revine sistemului de operare, care trebuie sa introduca un model prin intermediul caruia executia programelor, privita din perspectiva utilizatorului, sa se desfasoare in paralel. Se realizeaza, de fapt, un *pseudoparalelism*, prin care procesorul este alocat pe rand programelor care trebuie rulate, cate o cuanta de timp pentru fiecare, astfel incat din exterior ele par ca ruleaza efectiv in acelasi timp.

Cel mai raspandit model care introduce paralelismul in executia programelor este modelul bazat pe **procese**. Acest model este cel adoptat de sistemul de operare Unix si va face obiectul acestei lucrari.

Un *proces* este un program secvential *in executie*, impreuna cu zona sa de date, stiva si numaratorul de instructiuni (program counter). Trebuie facuta inca de la inceput distinctia dintre *proces* si *program*. Un program este, in fond, un sir de instructiuni care trebuie executate de catre calculator, in vreme ce un proces este o abstractizare a programului, specifica sistemelor de operare. Se poate spune ca un proces executa un program si ca sistemul de operare lucreaza cu procese, iar nu cu programe. Procesul include in plus fata de program informatiile de stare legate de executia programului respectiv (stiva, valorile registrilor CPU etc.). De asemenea, este important de subliniat faptul ca un program (ca aplicatie software) poate fi format din *mai multe procese* care sa ruleze sau nu in paralel.

Orice proces este executat secvential, iar mai multe procese pot sa ruleze in paralel (intre ele). De cele mai multe ori, executia in paralel se realizeaza alocand pe rand procesorul cate unui proces. Desi la un moment dat se executa un singur proces, in decurs de o secunda, de exemplu, pot fi executate portiuni din mai multe procese. Din aceasta schema rezulta ca un proces se poate gasi, la un moment dat, in una din urmatoarele trei stari [Tanenbaum]:

- In executie
- Pregatit pentru executie
- Blocat

Procesul se gaseste *in executie* atunci cand procesorul ii executa instructiunile. *Pregatit de executie* este un proces care, desi ar fi gata sa isi continue executia, este lasat in asteptare din cauza ca un alt proces este in executie la momentul respectiv. De asemenea, un proces poate fi *blocat* din doua motive: el isi suspenda executia in mod voit sau procesul efectueaza o operatie in afara procesorului,

mare consumatoare de timp (cum e cazul operatiilor de intrare-iesire - acestea sunt mai lente si intre timp procesorul ar putea executa parti din alte procese).

## 2. Utilizarea proceselor in UNIX

### 2.1 Apelul sistem fork( )

Din perspectiva programatorului, sistemul de operare UNIX pune la dispozitie un mecanism elegant si simplu pentru crearea si utilizarea proceselor.

Orice proces trebuie creat de catre un alt proces. Procesul creator este numit *proces parinte*, iar procesul creat *proces fiu*. Exista o singura exceptie de la aceasta regula, si anume procesul *init*, care este procesul initial, creat la pornirea sistemului de operare si care este responsabil pentru crearea urmatoarelor procese. Interpretorul de comenzi, de exemplu, ruleaza si el in interiorul unui proces.

Fiecare proces are un identificator numeric, numit *identificator de proces (process identifier - PID)*. Acest identificator este folosit atunci cand se face referire la procesul respectiv, din interiorul programelor sau prin intermediul interpretorului de comenzi.

Un proces trebuie creat folosind apelul sistem

#### `pid_t fork()`

Prin aceasta functie sistem, procesul apelant (parintele) creeaza un nou proces (fiul) care va fi o *copie fidela* a parintelui. Noul proces va avea propria lui zona de date, propria lui stiva, propriul lui cod executabil, toate fiind copiate de la parinte in cele mai mici detalii. Rezulta ca variabilele fiului vor avea valorile variabilelor parintelui in momentul apelului functie *fork()*, iar executia fiului va continua cu instructiunile care urmeaza imediat acestui apel, codul fiului fiind identic cu cel al parintelui. Cu toate acestea, in sistem vor exista din acest moment doua procese independente, (desi identice), cu zone de date si stiva distincte. Orice modificare facuta, prin urmare, asupra unei variabile din procesul fiu, va ramane invizibila procesului parinte si invers.

Procesul fiu va mosteni de la parinte toti descriptorii de fisier deschisi de catre acesta, asa ca orice prelucrari ulterioare in fisiere vor fi efectuate in punctul in care le-a lasat parintele.

Deoarece codul parintelui si codul fiului sunt identice si pentru ca aceste procese vor rula in continuare in paralel, trebuie facuta clar distinctia, in interiorul programului, intre actiunile ce vor fi executate de fiu si cele ale parintelui. Cu alte cuvinte, este nevoie de o metoda care sa indice care este portiunea de cod a parintelui si care a fiului. Acest lucru se poate face simplu, folosind valoarea returnata de functia *fork()*. Ea returneaza:

- -1, daca operatia nu s-a putut efectua (eroare)
- 0, in codul fiului
- *pid*, in codul parintelui, unde *pid* este identificatorul de proces al fiului nou-creat.

Prin urmare, o posibila schema de apelare a functiei *fork()* ar fi:

```
...
if( ( pid=fork() ) < 0 )
{
    perror("Eroare");
}
```

```

        exit(1);
    }
if(pid==0)
{
    /* codul fiului */
    ...
    exit(0)
}
/* codul parintelui */
...
wait(&status)

```

### 2.2 Functiile wait() si waitpid()

`pid_t wait(int *status)`

`pid_t waitpid(pid_t pid, int *status, int flags)`

Functia `wait()` este folosita pentru asteptarea terminarii fiului si preluarea valorii returnate de acesta. Parametrul `status` este folosit pentru evaluarea valorii returnate, folosind cateva macro-uri definite special (vezi paginile de manual corespunzatoare functiilor `wait()` si `waitpid()`). Functia `waitpid()` este asemanatoare cu `wait()`, dar asteapta terminarea unui anumit proces dat, in vreme ce `wait()` asteapta terminarea oricarui fiu al procesului curent. Este obligatoriu ca starea proceselor sa fie preluata dupa terminarea acestora, astfel ca functiile din aceasta categorie nu sunt optionale.

### 2.3. Functiile de tipul exec()

Functia `fork()` creeaza un proces identic cu procesul parinte. Pentru a crea un nou proces care sa ruleze un program diferit de cel al parintelui, aceasta functie se va folosi impreuna cu unul din apelurile sistem de tipul `exec()`: `exec()`, `execvp()`, `execv()`, `execvp()`, `execle()`, `execve()`.

Toate aceste functii primesc ca parametru un nume de fisier care reprezinta un program executabil si realizeaza lansarea in executie a programului. *Programul va fi lansat atfel incat se va suprascrie codul, datele si stiva procesului care apeleaza exec(), astfel incat, imediat dupa acest apel programul initial nu va mai exista in memorie. Procesul va ramane, insa, identificat prin acelasi numar (PID) si va mosteni toate eventualele redirectari facute in prealabil asupra descriptorilor de fisiere (de exemplu intrarea si iesirea standard). De asemenea, el va pastra relatia parinte-fiu cu procesul care a apelat fork().*

Singura situatie in care procesul apelant revine din apelul functiei `exec()` este acela in care operatia nu a putut fi efectuata, caz in care functia returneaza un cod de eroare (-1).

In consecinta, lansarea intr-un proces separat a unui program de pe disc se face apeland `fork()` pentru crearea noului proces, dupa care in portiunea de cod executata de fiu se va apela una din functiile `exec()`.

*Observatie: consultati paginile de manual corespunzatoare acestor functii.*

### 2.4 Functiile system() si vfork()

`int system(const char *cmd)`

Lanseaza in executie un program de pe disc, folosind in acest scop un apel `fork()`, urmat de `exec()`, impreuna cu `waitpid()` in parinte.

### **pid\_t vfork()**

Creeaza un nou proces, la fel ca `fork()`, dar nu copiaza in intregime spatiul de adrese al parintelui in fiu. Este folosit in conjunctie cu `exec()`, si are avantajul ca nu se mai consuma timpul necesar operatiilor de copiere care oricum ar fi inutile daca imediat dupa aceea se apeleaza `exec()` (oricum, procesul fiu va fi supascris cu programul luat de pe disc).

### ***2.5 Alte functii pentru lucrul cu procese***

`pid_t getpid()` - returneaza PID-ul procesului curent  
`pid_t getppid()` - returneaza PID-ul parintelui procesului curent  
`uid_t getuid()` - returneaza identificatorul utilizatorului care a lansat procesul curent  
`gid_t getgid()` - returneaza identificatorul grupului utilizatorului care a lansat procesul curent

### ***2.6 Gestionarea proceselor din linia de comanda***

Sistemul de operare UNIX are cateva comenzi foarte utile care se refera la procese:

- **ps** - afiseaza informatii despre procesele care ruleaza in mod curent pe sistem
- **kill -semnal proces** - trimite un semnal unui proces. De exemplu,  
`kill -9 123`  
va termina procesul cu numarul 123
- **killall -semnal nume** - trimite semnal catre toate procesele cu numele *nume*

Exista si alte comenzi utile; pentru folosirea lor, este recomandat sa se consulte paginile de manual UNIX.

### **1. Explicati efectul urmatoarei secvente de cod:**

```
int i;
for(i=1; i<=10; i++)
    fork();
```

**2. Realizati un program C pentru UNIX care creeaza 5 procese (inclusiv parintele). Fiecare proces afiseaza pe ecran cate 10 linii continand tipul sau (parinte, fiu1, fiu2, fiu3 fiu4) si PID-ul propriu. Dupa aceea, procesele fiu se vor termina returnand valori diferite, iar parintele va afisa valorile returnate de catre fii.**

### **Bibliografie:**

[Tanenbaum] Andrew S. Tanenbaum: *Modern Operating Systems*, Prentice Hall, 1992, pag. 27-31, 279-284

## ***7. Comunicarea intre procese folosind semnale***

## 1. Tipuri de semnale

Procesele pot comunica intre ele in mai multe moduri, unul din acestea fiind utilizarea *semnalelor*. Semnalele sunt intreruperi software care pot sa apara asincron in sistem. Orice proces poate genera sau primi semnale. Procesul poate alege unul dintre urmatoarele moduri de tratare a semnalelor:

- sa le capteze, definind functii de tratare a semnalelor (*signal handlers*)
- sa le ignore
- sa accepte comportamentul implicit la aparitia unui semnal: terminarea procesului

Semnalele sunt de diferite tipuri, fiecare avand cate un numar. Standardul POSIX cere existenta in orice sistem UNIX cel putin a urmatoarelor semnale:

- SIGABRT - abandonarea (*abort*) procesului
- SIGALARM - semnal generat de ceasul cu alarma
- SIGFPE - eroare aritmetica in virgula mobila (ex: impartire la zero)
- SIGHUP - terminalul folosit de proces a fost inchis
- SIGILL - procesul a executat o instructiune ilegală
- SIGINT - procesul trebuie interupt
- SIGQUIT - cerere de iesire din program provenita de la utilizator
- SIGKILL - distrugerea procesului
- SIGPIPE - procesul a scris intr-un *pipe* fara cititori
- SIGSEGV - procesul a referit o adresa de memorie nevalida
- SIGTERM - utilizatorul cere terminarea normala a procesului
- SIGUSR1, SIGUSR2 - semnale definite de utilizator

Diferitele variante de UNIX implementeaza aceste semnale, adaugand si altele noi. De exemplu, in Linux, exista urmatoarele semnale POSIX:

Signal	Value	Action	Comment
<hr/>			
SIGHUP	1	A	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	A	Interrupt from keyboard
SIGQUIT	3	A	Quit from keyboard
SIGILL	4	A	Illegal Instruction
SIGABRT	6	C	Abort signal from abort(3)
SIGFPE	8	C	Floating point exception
SIGKILL	9	AEF	Kill signal
SIGSEGV	11	C	Invalid memory reference
SIGPIPE	13	A	Broken pipe: write to pipe with no readers
SIGALRM	14	A	Timer signal from alarm(2)
SIGTERM	15	A	Termination signal
SIGUSR1	30,10,16	A	User-defined signal 1
SIGUSR2	31,12,17	A	User-defined signal 2
SIGCHLD	20,17,18	B	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	DEF	Stop process
SIGTSTP	18,20,24	D	Stop typed at tty
SIGTTIN	21,21,26	D	tty input for background process

```
SIGTTOU 22,22,27      D      tty output for background process
```

**Nota:** consultati pagina de manual *signal* din sectiunea 7, in Linux.

## 2. Apelul sistem *signal*

```
void (*signal(int signum, void (*handler)(int)))(int);
```

Apelul sistem instaleaza o noua functie de tratare pentru semnalul cu numarul *signum*. Argumentul *handler* poate fi functia de tratare a aparitiei semnalului sau una din urmatoarele valori:

- SIG\_IGN - ignora semnalul
- SIG\_DFL - reseteaza semnalul la comportarea implicita (*default*)

Semnalele SIG\_KILL si SIG\_STOP nu pot fi ignorate si nici nu pot fi instalate functii de tratare pentru ele.

Functia returneaza vechia valoare a handler-ului de semnal sau SIG\_ERR in caz de eroare.

Functia de tratare a semnalului trebuie sa aiba un parametru intreg, prin care primeste numarul semnalului aparut.

Scrieti un program UNIX in C care realizeaza urmatoarele:

- sistemul este format din trei procese: parinte si doi fii
- fiii ignora semnalul SIG\_TERM
- parintele instaleaza o alarma care il va intrerupe dupa 7 secunde; intre timp va afisa pe ecran, in mod continuu, caracterul 'A'. La primirea semnalului de alarma va anunta fiii prin semnalul SIG\_USR1, dar va continua sa afiseze (la infinit) caracterul 'A'
- fiii afiseaza continuu caracterele 'B' si 'C', pana in momentul in care primesc semnalul SIG\_USR1, moment in care se termina
- cand si ultimul fiu se termina, parintele preia starea amandurora si se termina si el

Vor fi folosite functii de tratare a semnalelor (una pentru generarea SIG\_USR1 in parinte si trei pentru terminarea proceselor parinte, fiu1 si fiu2) si functiile *alarm*, *kill*. Indicatie: cand un fiu se termina, parintele primeste semnalul SIG\_CHLD.

## 8. Comunicarea intre procese folosind pipes

O metoda foarte des utilizata in UNIX pentru comunicarea intre procese este folosirea primitivei numita *pipe* (conducta). "Conducta" este o cale de legatura care poate fi stabilita intre doua procese inrudite (au un stramos comun sau sunt in relatia stramos-urmas). Ea are doua capete, unul prin care se pot scrie date si altul prin care datele pot fi citite, permitand o comunicare intr-o singura directie. In general, sistemul de operare permite conectarea a unuia sau mai multor procese la fiecare din capetele unui *pipe*, astfel incat, la un moment dat este posibil sa existe mai multe procese care scriu, respectiv mai multe procese care citesc din *pipe*. Se realizeaza, astfel, comunicarea unidirectionala intre procesele care scriu si procesele care citesc.

### 1. Apelul sistem *pipe()*

Crearea conductelor de date de face in UNIX folosind apelul sistem *pipe()*:

```
int pipe(int filedes[2]);
```

Functia creeaza un *pipe*, precum si o pereche de descriptori de fisier care refera cele doua capete ale acestuia. Descriptorii sunt returnati catre programul apelant completandu-se cele doua pozitii ale tabloului *filedes* trimis ca parametru apelului sistem. Pe prima pozitie va fi memorat descriptorul care indica extremitatea prin care se pot citi date (capatul de citire), iar pe a doua pozitie va fi memorat descriptorul capatului de scriere in *pipe*.

Cei doi descriptori sunt descriptori de fisier obisnuiti, asemanatori celor returnati de apelul sistem *open()*. Mai mult, *pipe*-ul poate fi folosit in mod similar folosirii fisierelor, adica in el pot fi scrise date folosind functia *write()* (aplicata capatului de scriere) si pot fi citite date prin functia *read()* (aplicata capatului de citire).

Fiind implicati descriptori de fisier obisnuiti, daca un *pipe* este creat intr-un proces parinte, fiii acestuia vor mosteni cei doi descriptori (asa cum, in general, ei mostenesc orice descriptor de fisier deschis de parinte). Prin urmare, atat parintele cat si fiii vor putea scrie sau citi din *pipe*. In acest mod se justifica afirmatia facuta la inceputul acestui document prin care se spunea ca *pipe*-urile sunt folosite la comunicarea intre procese *inrudite*. Pentru ca legatura dintre procese sa se faca corect, fiecare proces trebuie sa declare daca va folosi *pipe*-ul pentru a scrie in el (transmitand informatii altor procese) sau il va folosi doar pentru citire. In acest scop, fiecare proces trebuie sa inchida capatul *pipe*-ului pe care nu il foloseste: procesele care scriu in *pipe* vor inchide capatul de citire, iar procesele care citesc vor inchide capatul de scriere, folosind functia *close()*.

Functia returneaza 0 daca operatia de creare s-a efectuat cu succes si -1 in caz de eroare.

Un posibil scenariu pentru crearea unui sistem format din doua procese care comunica prin *pipe* este urmatorul:

- procesul parinte creeaza un *pipe*
- parintele apeleaza *fork()* pentru a crea fiul
- fiul inchide unul din capete (ex: capatul de citire)
- parintele inchide celalalt capat al *pipe*-ului (cel de scriere)
- fiul scrie date in *pipe* folosind descriptorul ramas deschis (capatul de scriere)
- parintele citeste date din *pipe* prin capatul de citire.

Primitiva *pipe* se comporta in mod asemanator cu o structura de date coada: scrierea introduce elemente in coada, iar citirea le extrage pe la capatul opus.

Iata in continuare o portiune de program scris conform scenariului de mai sus:

```
void main()
{
    int pfd[2];
    int pid;
```

```

...
if(pipe(pfd)<0)
{
    printf("Eroare la crearea pipe-ului\n");
    exit(1);
}

...
if((pid=fork())<0)
{
    printf("Eroare la fork\n");
    exit(1);
}
if(pid==0) /* procesul fiu */
{
    close(pfd[0]); /* inchide capatul de citire; */
                    /* procesul va scrie in pipe */
    ...
    write(pfd[1],buff,len); /* operatie de scriere in
pipe */

    ...
    close(pfd[1]); /* la sfarsit inchide si capatul
utilizat */
    exit(0);
}
else /* procesul parinte */
{
    close(pfd[1]); /* inchide capatul de scriere; */
                    /* procesul va citi din pipe */
    ...
    read(pfd[0],buff,len); /* operatie de citire din
pipe */

    ...
    close(pfd[0]); /* la sfarsit inchide si capatul
utilizat */
    exit(0);
}
}

```

Coneksiunea intre cele doua procese din care este format programul de mai sus este reflectata in figura urmatoare:

*Observatii:*

1. Cantitatea de date care poate fi scrisa la un moment dat intr-un *pipe* este limitata. Numarul de octeti pe care un *pipe* ii poate pastra fara ca ei sa fie extrasi prin citire de catre un proces este dat de constanta predefinita PIPE\_BUF.
2. Un proces care citeste din pipe va primi valoarea 0 ca valoare returnata de *read()* in momentul in care toate procesele care scriau in *pipe* au inchis capatul de scriere si nu mai exista date in *pipe*.
3. Daca pentru un pipe sunt conectate procese doar la capatul de scriere (cele de la capatul opus au inchis toate conexiunea) operatiile *write* efectuate de procesele ramase vor returna eroare. Intern, in aceasta situatie va fi generat semnalul SIG\_PIPE care va intrerupe apelul sistem *write* respectiv. Codul de eroare (setat in variabila globala *errno*) rezultat este cel corespunzator mesajului de eroare "Broken pipe".

## **2. Redirectarea descriptorilor de fisier**

Se stie ca functia *open()* returneaza un descriptor de fisier. Acest descriptor va indica fisierul deschis cu *open()* pana la terminarea programului sau pana la inchiderea fisierului. Sistemul de operare UNIX ofera, insa, posibilitatea ca un descriptor oarecare sa indice un alt fisier decat cel obisnuit. Operatia se numeste *redirectare* si se foloseste cel mai des in cazul descriptorilor de fisier cu valorile 0, 1 si 2 care reprezinta intrarea standard, iesirea standard si, respectiv, iesirea standard de eroare. De asemenea, este folosita si operatia de *duplicare* a descriptorilor de fisier, care determina existenta a mai mult de un descriptor pentru acelasi fisier. De fapt, redirectarea poate fi vazuta ca un caz particular de duplicare.

Duplicarea si redirectarea se fac, in functie de cerinte, folosind una din urmatoarele apeuri sistem:

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Functia *dup()* realizeaza duplicarea descriptorului *oldfd*, returnand noul descriptor. Aceasta inseamna ca descriptorul returnat va indica acelasi fisier ca si *oldfd*, atat noul cat si vechiul descriptor folosind in comun pointerul de pozitie in fisier, flag-urile fisierului etc. Daca pozitia in fisier e modificata prin intermediul functiei *lseek()* folosind unul dintre descriptori, efectul va fi observat si pentru operatiile facute folosind celalalt descriptor. Descriptorul nou alocat de *dup()* este cel mai mic descriptor liber (inchis) disponibil.

Functia *dup2()* se comporta in mod asemanator cu *dup()*, cu deosebirea ca poate fi indicat explicit care sa fie noul descriptor. Dupa apelul *dup2()*, descriptorul *newfd* va indica acelasi fisier ca si *oldfd*. Daca inainte de operatie descriptorul *newfd* era deschis, fisierul indicat este mai intai inchis, dupa care se face duplicarea.

Ambele functii returneaza descriptorul nou creat (in cazul lui *dup2()*, egal cu *newfd*) sau -1 in caz de eroare.

Urmatoarea secventa de cod realizeaza redirectarea iesirii standard spre un fisier deschis, cu descriptorul corespunzator *fd*:

```
...
fd=open("Fisier.txt", O_WRONLY);
...
if((newfd=dup2(fd,1))<0)
{
    printf("Eroare la dup2\n");
    exit(1);
}
...
printf("ABCD");
```

In urma redirectarii, textul "ABCD" tiparit cu `printf()` nu va fi scris pe ecran, ci in fisierul cu numele "Fisier.txt".

Redirectarile de fisiere se pastreaza chiar si dupa apelarea unei functii de tip *exec( )* (care suprascrie procesul curent cu programul luat de pe disc). Folosind aceasta facilitate, este posibila, de exemplu, conectarea prin *pipe* a doua procese, unul din ele ruland un program executabil citit de pe disc. Secventa de cod care realizeaza acest lucru este data mai jos. Se considera ca parintele deschide un pipe din care va citi date, iar fiul este un proces care executa un program de pe disc. Tot ce afiseaza la iesirea standard procesul luat de pe disc, va fi redirectat spre capatul de scriere al *pipe*-ului, astfel incat parintele poate citi datele produse de acesta.

```
void main()
{
    int pfd[2];
    int pid;
    FILE *stream;

    ...
    if(pipe(pfd)<0)
    {
        printf("Eroare la crearea pipe-ului\n");
        exit(1);
    }
    ...
    if((pid=fork())<0)
    {
        printf("Eroare la fork\n");
        exit(1);
    }
    if(pid==0) /* procesul fiu */
    {
        close(pfd[0]); /* inchide capatul de citire; */
                        /* procesul va scrie in pipe */
    }
}
```

```

...
    dup2(pfd[1],1); /* rediecteaza iesirea standard
spre pipe */
...
    execp("ls","ls","-l",NULL); /* procesul va rula
comanda ls */
    printf("Eroare la exec\n");
    /* Daca execp s-a intors, inseamna ca
programul nu a putut fi lansat in executie */
}
else /* procesul parinte */
{
    close(pfd[1]); /* inchide capatul de scriere; */
    /* procesul va citi din pipe */
...
    stream=fdopen(pfd[0],"r");
    /* deschide un stream (FILE *) pentru capatul
de citire */
    fscanf(stream,"%s",string);
    /* citire din pipe, folosind stream-ul asociat
*/
...
    close(pfd[0]); /* la sfarsit inchide si capatul
utilizat */
    exit(0);
}
}

```

Functia *fdopen()* a fost folosita pentru a putea folosi avantajele functiilor de biblioteca pentru lucrul cu fisiere in cazul unui fisier (capatul de citire din pipe) indicat de un descriptor intreg (in speta, s-a dorit sa se efectueze o citire formatata, cu *fscanf()*, din *pipe*).

1. Sa se scrie un program care creeaza trei procese, astfel:

- primul proces (parinte) citeste dintr-un fisier cu numele *date.txt* un sir de caractere, pana la sfarsitul fisierului si le trimite printre-un *pipe* primului fiu
- primul fiu primeste caracterele de la parinte si selecteaza din ele literele mici, pe care le trimite printre-un *pipe* catre cel de-al doilea fiu.
- al doilea fiu creeaza un fisier numit *statistica.txt*, in care va memora, pe cate o linie, fiecare litera distincta intalnita si numarul de aparitii a acesteia in fluxul de date primit. In final, va trimite catre parinte, printre-un *pipe* suplimentar, numarul de litere distincte intalnite.
- parintele afiseaza pe ecran rezultatul primit de la al doilea fiu.

2. Sa se realizeze un program de tip *prodicatori-consumatori* astfel:

- programul primește ca parametri în linia de comandă două numere: numarul de procese producător și numarul de procese consumator;
- resursa comună tuturor proceselor este un *pipe* creat de procesul parinte;
- parințele este la randul lui producător;
- în general, producătorii scriu în *pipe* un număr oarecare de caractere, iar consumatorii citesc din *pipe*, caracter cu caracter, căt timp acest lucru este posibil.

Producătorii vor avea urmatoarea formă:

- a) Parințele produce un număr de caractere '\*';
- b) ceilalți producători sunt procese independente (existente pe disc ca programe de sine statatoare) care generează la ieșirea standard un număr oarecare de caractere '#'. Pentru realizarea scopului propus, ieșirea standard a acestor procese va fi conectată la capatul de scriere al *pipe*-ului.
- c) Cel puțin unul dintre producători este comanda 'ls'.

Consumatorii citesc caracterele din *pipe* și își afisează identificatorul de proces urmat de caracterul citit (la un moment dat).

În sistem mai există un *pipe* prin care consumatorii vor raporta în final rezultatele către parințele tuturor, scriind în el, cu ajutorul funcției *fprintf()*, linii de forma:

*numar\_proces : numar\_octeti*

unde *numar\_octeti* este numarul total de octeti cititi de procesul respectiv din *pipe*. Parințele va prelua aceste informații și le va afisa pe ecran.

**Nota:** Aceasta lucrare de laborator se va desfasura

## 9. Fire de executie

Intr-o lucrare de laborator anterioara au fost introduse elementele de baza referitoare la *procese*. Recapitulând pe scurt, un proces era văzut ca fiind format dintr-o zonă de cod, o zonă de date, stiva și registri ai procesorului (Program Counter și alții). În consecință, fiecare proces apare ca o entitate distinctă, independentă de celelalte procese aflate în execuție la un moment dat. De asemenea, a fost remarcat faptul că, având în vedere că procesorul poate rula la un moment dat un singur proces, procesele sunt executate pe rand, după un anumit algoritm de planificare, astfel încât, la nivelul aplicațiilor, acestea par că se execută în paralel.

Se desprind, astfel, două idei importante referitoare la procese:

- rulează independent, având zone de cod, stiva și date distincte
- trebuie planificate la execuție, astfel încât ele să ruleze aparent în paralel

Execuția planificată a proceselor presupune că, la momente de timp determinate de algoritmul folosit, procesorul să fie "luat" de la procesul care tocmai se execută și să fie "dat" unui alt proces. Această comutare între procese (*process switching*) este o operație consumatoare de timp, deoarece

trebuie "comutate" toate resursele care aparțin proceselor: trebuie salvati și restaurati toti registrii procesor, trebuie (re)mapate zonele de memorie care aparțin de noul proces etc.

Un concept interesant care se regăsește în toate sistemele de operare moderne este acela de *fir de executie (thread)* în interiorul unui proces. Firele de execuție sunt uneori numite *procese usoare (lightweight processes)*, sugerându-se asemănarea lor cu procesele, dar și, într-un anume sens, deosebirile dintre ele.

Un fir de execuție trebuie văzut ca un flux de instrucțiuni care se executa *in interiorul unui proces*. Un proces poate să fie format din mai multe asemenea fire, care se executa în paralel, *având, însă, în comun toate resursele principale caracteristice procesului*. Prin urmare, în interiorul unui proces, firele de execuție sunt entități care rulează în paralel, împărțind între ele zona de date și executând portiuni distincte din același cod. Deoarece zona de date este comună, toate variabilele procesului vor fi văzute la fel de către toate firele de execuție, orice modificare facută de către un fir devenind vizibilă pentru toate celelalte. Generalizând, un proces, așa cum era el percepțut în lucrările de laborator precedente, este de fapt un proces format dintr-un singur fir de execuție.

La nivelul sistemului de operare, execuția în paralel a firelor de execuție este obținută în mod asemănător cu cea a proceselor, realizându-se o comutare între fire, conform unui algoritm de planificare. Spre deosebire de cazul proceselor, însă, aici comutarea poate fi făcută mult mai rapid, deoarece informațiile memorate de către sistem pentru fiecare fir de execuție sunt mult mai puține decât în cazul proceselor, datorită faptului că firele de execuție au foarte puține resurse proprii. Practic, un fir de execuție poate fi văzut ca un numarator de program, o stivă și un set de registri, toate celelalte resurse (zona de date, identificatori de fișier etc) aparținând procesului în care rulează și fiind exploatațate în comun.

### 9.1 Implementarea firelor de execuție în Linux

Linux implementează firele de execuție oferind, la nivel scăzut, apelul sistem *clone()*:

```
pid_t clone(void *sp, unsigned long flags)
```

Functia *clone()* este o interfață alternativă la funcția sistem *fork()*, ea având ca efect crearea unui proces fiu, oferind, însă, mai multe opțiuni la creare.

Dacă *sp* este diferit de zero, procesul fiu va folosi *sp* ca indicator al stivei sale, permitându-se astfel programatorului să aleagă stiva nouului proces.

Argumentul *flags* este un sir de biti continând diferite opțiuni pentru crearea procesului fiu. Octetul inferior din *flags* conține semnalul care va fi trimis la parinte în momentul terminării fiului nou creat. Alte opțiuni care pot fi introduse în cuvântul *flags* sunt: COPYVM și COPYFD. Dacă este setat COPYVM, paginile de memorie ale fiului vor fi copii fidele ale paginilor de memorie ale parintelui, ca la funcția *fork()*. Dacă COPYVM nu este setat, fiul va împărtăși parintele paginile de memorie ale acestuia. Cand COPYFD este setat, fiul va primi descriptorii de fișier ai parintelui ca și copii distincte, iar dacă nu este setat, fiul va împărtăși descriptorii de fișier cu parintele.

Functia returnează PID-ul fiului în parinte și zero în fiu.

Prin urmare, apelul sistem *fork()* este echivalent cu:

```
clone(0, SIGCLD | COPYVM)
```

Se observa ca functia *clone()* ofera suficiente facilitati pentru a putea crea primitive de tip fire de executie.

Pentru un exemplu "clasic" de utilizare a functiei *clone()*, consultati fisierul [\*clone.c\*](#), al carui autor este Linus Torvalds.

## 9.2 Utilizarea firelor de executie

In programe este indicat sa nu se foloseasca direct functia *clone()*, in primul rand din cauza ca ea nu este probabila (fiind specifica Linux) si apoi pentru ca utilizarea ei este intrucatva greoala.

Standardul POSIX 1003.1c, adoptat de catre IEEE ca parte a standardelor POSIX, defineste o interfata de programare pentru utilizarea firelor de executie, numita *pthread*. Interfata este implementata pe multe arhitecturi; mai mult, sistemele de operare care continuteau biblioteci proprii de fire de executie (cum este SOLARIS) introduc suport pentru acest standard.

In Linux exista o biblioteca numita *LinuxThreads*, care implementeaza versiunea finala a standardului POSIX 1003.1c si utilizeaza functia *clone()* ca instrument de creare a firelor de executie. In continuare, ne vom referi la aceasta biblioteca de functii si vom trece in revista o parte din primitivele introduse de catre ea.

### 9.2.1 Aspecte practice privind utilizarea bibliotecii LinuxThreads

O forma speciala a acestei biblioteci este inclusa in biblioteca *glibc2* disponibila in distributiile RedHat incepand cu versiunea 5.0. Biblioteca *LinuxThreads* poate fi instalata si pe sisteme care folosesc alte versiuni de Linux si poate fi obtinuta chiar prin intermediul acestei pagini, accesand fisierul [\*linuxthreads.tar.gz\*](#). (Detalii referitoare la instalare se gasesc intr-un fisier README inclus in arhiva) Distributia contine si paginile de manual corespunzatoare functiilor definite de biblioteca, utile si in cazul utilizarii RedHat 5.0, unde ele lipsesc.

Odata ce biblioteca a fost instalata cu succes, programele pot fi compilate folosind comanda:

```
gcc -D_REENTRANT -lpthread <fisier.c> -o <executabil>
```

Se observa ca este necesara definirea constantei *\_REENTRANT* (din considerente legate de executia paralela a firelor) si ca trebuie inclusa explicit biblioteca *pthread* (numele sub care se regaseste LinuxThreads, conform POSIX).

### 9.2.2 Crearea firelor de executie

Un fir de executie se creaza folosind functia

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);
```

Functia creeaza un *thread* care se va executa in paralel cu *thread*-ul creator. Noul fir de executie va fi format de functia *start\_routine* care trebuie definita in program avand un singur argument de tip **(void \*)**. Parametrul *arg* este argumentul care va fi transmis acestei functii. Parametrul *attr* este un cuvant care specifica diferite optiuni de creare a firului de executie. In mod obisnuit, acesta este dat ca NULL, acceptand optiunile implicite. Firul de executie creat va primi un identificator care va fi

returnat in variabila indicata de parametrul *thread*. Functia returneaza 0 daca creearea a avut succes si un numar diferit de zero in caz contrar.

*Thread*-ul va consta in executia functiei date ca argument, iar terminarea lui se va face ori apeland explicit functia *pthread\_exit()*, ori implicit, prin iesirea din functia *start\_routine*.

### 9.2.3 Terminarea firelor de executie

Un fir de executie se poate termina apeland:

```
void pthread_exit(void *retval);
```

Valoarea *retval* este valoarea pe care *thread*-ul o returneaza la terminare. Starea returnata de firele de executie poate fi preluata de catre *oricare* din *thread*-urile aceluiasi proces, folosind functia:

```
int pthread_join(pthread_t th, void **thread_return);
```

Aceasta intrerupe firul de executie care o apeleaza pana cand firul de executie cu identificatorul *th* se termina, moment in care starea lui va fi returnata la adresa data de parametrul *thread\_return*.

Demn de observat este faptul ca *nu pentru toate firele de executie poate fi preluata starea de iesire*. De fapt, conform standardului POSIX, firele de executie se impart in doua categorii:

- **joinable** - ale caror stari pot fi preluate de catre celelalte fire din proces
- **detached** - ale caror stari nu pot fi preluate

In cazul *thread*-urilor *joinable*, in momentul terminarii acestora, resursele lor nu sunt complet dezalocate, asteptandu-se un viitor *pthread\_join* pentru ele. Firele de executie *detached* se dezaloca in intregime, starea lor devenind nedisponibila pentru alte fire de executie.

Tipul unui fir de executie poate fi specificat la crearea acestuia, folosind optiunile din argumentul *attr* (implicit este *joinable*). De asemenea, un fir de executie *joinable* poate fi "detasat" mai tarziu, folosind functia *pthread\_detach()*.

### 9.2.4 Observatii

- Un proces, imediat ce a fost creat, este format dintr-un singur fir de executie, numit fir de executie principal (initial).
- Toate firele de executie din cadrul unui proces se vor executa in paralel.
- Datorita faptului ca impart aceeasi zona de date, firele de executie ale unui proces vor folosi in comun toate variabilele globale. De aceea, se recomanda ca in programe firele de executie sa utilizeze numai variabilele locale, definite in functiile care implementeaza firul, in afara de cazurile in care se doreste partajarea explicita a unor resurse.
- Daca un proces format din mai multe fire de executie se termina ca urmare a primirii unui semnal, toate firele de executie ale sale se vor termina.
- Daca un fir de executie apeleaza functia *exit()*, efectul va fi terminarea *intregului* proces, cu toate firele de executie din interior.
- Orice functie sau apel sistem care lucreaza cu sau afecteaza procese, va avea efect asupra intregului proces, indiferent de firul de executie in care a fost apelata functia respectiva. De exemplu, functia *sleep()* va "adormi" toate firele de executie din proces, inclusiv firul de executie principal (initial), indiferent de firul care a apelat-o.

- Standardul POSIX defineste si cateva primitive de sincronizare intre firele de executie pentru accesul la resursele comune, care nu fac obiectul lucrarii de fata.

---

Sa se scrie un program C format dintr-un singur proces in care ruleaza 6 fire de executie (inclusiv cel initial). Procesul creeaza un tablou de 5 caractere, ale carui elemente sunt initializate cu '#'. In proces vor exista 5 fire de executie "producator" care vor completa continuu cate o locatie aleatoare din tablou cu cate un caracter, astfel: primul fir cu caracterul 'A', al doilea cu 'B' s.a.m.d. pana la 'E'. Fiecare fir de executie "producator" va numara cate caractere a introdus in tablou; primul fir se va termina in momentul in care a introdus 100 000 de caractere, al doilea 200 000 s.a.m.d. Thread-ul principal va afisa continuu continutul tabloului, pana cand toate firele "producator" se incheie, moment in care va prelua starea acestora si va termina procesul.

*Indicatie: pentru anuntarea terminarii firelor de executie "producatori", fiecare din ele va seta o variabila globala distincta, firul principal testand toate aceste variabile.*