

# Definiție. Scurt istoric

## Bibliografie curs (principalele titluri):

1. Andrew S. Tanenbaum – *Modern Operating Systems*, Prentice Hall, 2001
2. William Stallings – *Operating Systems. Internals and Design Principles*, Prentice hall 2005;
3. Abraham Silberschatz, Peter Galvin, Greg Gagne – *Operating System Concepts*, John Wiley & Sons, 2005;
4. Daniel Bovet, M. Cesati – *Understanding the Linux Kernel*, O'Reilly, 2003

## 1 Definiții

Există mai multe posibilități pentru definirea unui sistem de operare. Tanenbaum propune definirea sistemelor de operare din două puncte de vedere: cel al sistemului de operare ca administrator de resurse respectiv cel de extensie funcțională a calculatorului.

Silberschatz propune o definire a sistemelor de operare din punct de vedere al utilizatorului respectiv al calculatorului. Astfel, din punctul de vedere al utilizatorului, un sistem de operare constituie o unealtă cât mai simplă de utilizat pornind de la niște performanțe minimale în cazul calculatoarelor personale în timp ce în cazul stațiilor de tip mainframe, accesate de mai mulți utilizatori la un moment dat, un sistem de operare este văzut în principal ca un maximizator și gestionar al utilizării resurselor. Din punct de vedere al calculatorului, dacă putem privi lucrurile și din această perspectivă, un sistem de operare este văzut ca un gestionar de resurse.

Burgess definește un sistem de operare ca fiind un nivel al software-ului care tratează aspectele tehnice în timpul unei operări a unui calculator. Sistemul de operare funcționează în acest caz ca o protecție a utilizatorului vizavi de operațiunile calculatorului de pe nivelul de jos.

Stallings definește sistemul de operare ca fiind o aplicație care controlează execuția celorlalte programe și acționează ca o interfață între utilizator și resursele hardware ale calculatorului. Tot el definește și obiectivele unui sistem de operare ca fiind:

- ușurință în utilizare sau “convenabilitatea” - se referă la faptul că facilitează o utilizare mai ușoară a calculatorului;
- eficiență – permite o utilizare mai eficientă a resurselor calculatorului;
- abilitate de a evolu – un sistem de operare trebuie să fie astfel construit încât să permită o dezvoltare, testare și introducere de noi funcții sistem;

In figura 1. am reprezentat structura ierarhizată a unui sistem de calcul:

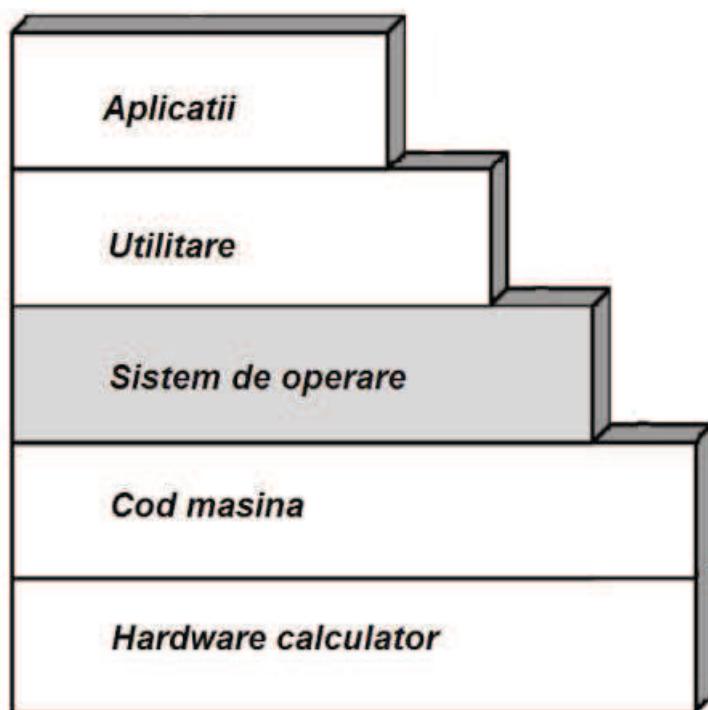


Figura 1. Structura ierarhizată a unui sistem de calcul

Practic, utilizatorul nu are legătura cu arhitectura calculatorului decât prin intermediul anumitor straturi. De fapt vedem calculatorul prin intermediul unei aplicații. Aplicația este dezvoltată de un programator într-un anumit limbaj de programare. Pentru dezvoltarea acestor programe trebuie să avem la dispoziție un set de utilitare: compilatoare, editoare de text, interpretoare. Ultimele 2 nivele constituie de fapt nivelul hardware, următoarele 2 îl constituie nivelul sistemului de operare. Acestea include și un set de utilitare. Pe lângă acestea pot fi instalate și alte utilitare pe sistem.

Sistemul de operare acționează ca un mediator între programator respectiv aplicație și resursele

Sistemele de operare asigură următoarele servicii:

- **dezvoltarea programelor:** sunt oferite o varietate de facilități legate de editoare sau depanatoare de programe pentru asistarea programatorului în munca sa. În mod normal acestea apar ca programe utilitare, deci nu fac parte din sistemul de operare dar sunt accesibile prin intermediul acestuia;

- **execuția programelor:** pentru execuția programelor trebuie să fie realizate o serie de taskuri: datele și instrucțiunile trebuie încărcate în memorie, dispozitivele de I/O trebuie initializate, trebuie pregătite o serie de resurse. Sistemul de operare le îndeplinește pe toate acestea;

- **asigură acces la dispozitivele de I/O:** sistemele de operare fac astfel ca programatorul să poată lucra direct cu dispozitivele de I/O, prin intermediul funcțiilor de *read* sau *write*, fără a fi nevoie să le programeze în detaliu;

- **acces controlat la fișiere:** în cazul mai multor utilizatori poate asigura un acces separat în funcție de drepturile pe care le au asupra fișierelor; pe lângă faptul că reprezintă dispozitive de I/O trebuie avut în vedere și formatul acestora;

- **accesul la sistem:** în cazul unui sistem partajat, sistemul de operare trebuie să asigure accesul diferențiat la resurse, protejarea acestora de un acces neautorizat;

O mulțime de erori pot apărea în timpul rulării unui program pe un calculator. Acestea includ erorile de hardware cum ar fi cea de memorie sau de cădere a unui dispozitiv sau cele de software și aici putem face referire la eroarea de overflow sau la imposibilitatea sistemului de operare de a satisface o cerere a unei aplicații. Răspunsul sistemului poate varia de la a închide programul care produce eroarea la o simplă notificare a acesteia.

Un calculator reprezintă un set de resurse pentru manevrarea, depozitarea și procesarea datelor precum și pentru controlul acestor funcții. Sistemul de operare este responsabil pentru managementul resurselor. Sistemul de operare se ocupă și de controlul acestor operații cu 2 observații:

funcționează ca și un software obișnuit, un program executat de procesor;

de multe ori pierde controlul și depinde de procesor pentru a-l reprimi.

Ca și alte programe de calculator și sistemul de operare furnizează instrucțiuni pentru procesor cu deosebirea că direcționează procesorul pentru utilizarea altor resurse și executarea altor programe.

In figura 2 sunt reprezentate principalele resurse conduse de către un sistem de operare. Astfel, o parte a sistemului de operare se află în memoria principală. Aceasta include kernelul sau nucleul care conține funcțiile cele mai utilizate. Restul memoriei principale conține date și programe. Aceasta este controlată de către sistemul de operare și hardware-ul aferent managementului memoriei (paginare, segmentare etc). Procesorul în sine este o resursă și sistemul de operare decide cât din timpul acestuia îi este alocat unui anume program. În cazul sistemelor multiprocesor decizia trebuie luată cu privire la toate procesoarele. Nu în

ultimul rând, sistemul de operare decide când un program aflat în execuție poate avea control asupra unui dispozitiv de I/O.

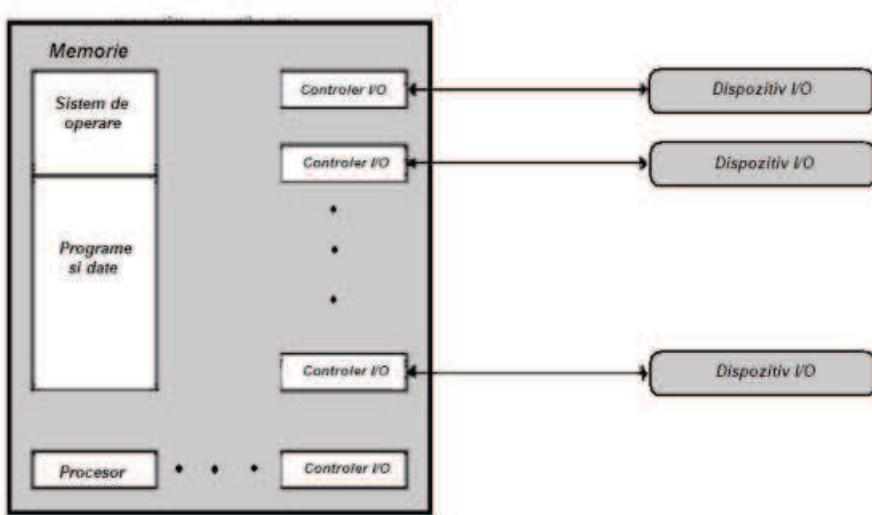


Figura 2. Principalele resurse conduse de către un sistem de operare

Unul dintre dispozitivele de I/O este hard discul. În acest caz putem avea o partiționare a acestuia ca și în cazul memoriei principale: sistemul de operare deține o partiție mai mică iar aplicațiile și datele pot acoperi restul mediului de stocare.

Un sistem de operare este necesar să evolueze în timp din următoarele motive:

- noi tipuri de hardware sau upgrade la cele existente: în cazul UNIX sau OS/2 nu era folosit mecanismul paginării deoarece hardware-ul nu avea nevoie de acest lucru. Un alt exemplu este cel dat de către folosirea mai multor ferestre de lucru (în sisteme gen Windows), lucru care a dus la un suport mai sofisticat pe partea de sistem de operare;
- noi servicii care trebuie asigurate: datorită cererii formulate de către utilizatori sau de către administratorii de sistem, sistemul de operare va trebui să ofere tot timpul servicii noi;
- variante îmbunătățite funcțional: de-a lungul timpului apar o serie de probleme în funcționare, probleme care nu au fost eliminate la testarea unui nou sistem (sau a unei variante îmbunătățite a acestuia). Evident că acestea vor fi eliminate pe măsură ce apar.

## 2 Istorici

Pentru a înțelege mai bine cum funcționează sistemele de operare este utilă o prezentare a evoluției acestora în timp.

Momentul 0 poate fi numit "motorul analitic" construit de Charles Babbage în sec. XIX. Având în vedere că era o construcție pur mecanică, nu avea cum să funcționeze. Până la mijlocul secolului XX nu mai putem consemna alte încercări în ceea ce privește construirea unui sistem de calcul. În perioada celui de al doilea război mondial, diverse colective de cercetători din Germania dar și din SUA au construit mașini de calcul pornind de la relee mecanice

Înlocuite apoi de tuburi cu vid, odată cu apariția acestora. Persoanele care se ocupau de aceste mașini de calcul făceau toate joburile: designer de sistem, programator, service. Programarea constă în realizarea unei plăci de conexiuni. Apariția cartelelor perforate după anii '50 a schimbat doar modul de "programare", acestea înlocuind plăcile de conexiune. Primele sisteme de operare au avut o procesare serială. Programatorul interacționa direct cu hardware-ul neexistând practic un sistem de operare. Aceste calculatoare funcționau prin intermediul unei console, informația era introdusă cu ajutorul unui dispozitiv (de genul celor pentru cartele de ex.) și era afișată la o imprimantă. Problemele care apăreau erau cele legate de împărțirea timpului de acces deoarece un utilizator avea la dispoziție o anumită durată pentru rezolvarea unei probleme, dacă o rezolva mai repede se pierdea restul de timp iar dacă depășea timpul alocat era forțat să se opreasă înainte de a rezolva problema, și respectiv de timpul irosit cu pregătirea rulării, timp dat de încărcarea programului sursă, a compilatorului, compilarea acestuia și salvarea în memorie a programului compilat, reîncărcarea acestuia, după care era executat. Toate aceste operațiuni însemnau montarea sau demontarea unor dispozitive iar dacă apărea o eroare totul era reluat de la capăt.

Introducerea tranzistorului în anii '55 a contribuit la dezvoltarea sistemelor de calcul. Apar și specializările în ceea ce privește personalul care se ocupa de calculatoare. Apar și primele limbi de programare cum ar fi cel de asamblare sau Fortran.

Pentru a fi evitați timpii de așteptare foarte mari datorați lucrului cu dispozitivele periferice, perioade în care procesoarele nu aveau de lucru, au fost introduse sistemele cu prelucrare în loturi. Un sistem cu prelucrare în loturi pornea de la existența în spatele său a unui program de tip monitor. Prin utilizarea acestuia, utilizatorul nu mai intra în contact direct cu sistemul. Utilizatorii introduceau joburile prin carduri sau bandă iar acestea erau puse împreună la dispozitivul de intrare al monitorului. Fiecare program redă controlul monitorului la terminarea sa astfel ca monitorul să poată procesa un nou job. Pentru a vedea mai bine cum funcționa totul ne putem încăpui 2 puncte de vedere: cel al monitorului și cel al procesorului. O parte a monitorului trebuia să se afle în memoria principală, gata de execuție. Aveam astfel un program de tip monitor rezident în memoria sistemului. Restul de utilitate și alte funcții erau încărcate în memoria principală odată cu programele. Monitorul citea una câte una joburile de la cititorul de cartele sau de bandă. Pe măsură ce un job era accesat de către programul monitor, acesta trecea la realizarea lui după care se pregătea pentru preluarea jobului următor. Rezultatele erau afișate pentru informarea utilizatorului. Din punct de vedere al procesorului, la un anumit moment de timp, acesta executa instrucțiuni din zona de memorie corespunzătoare programului monitor. Acestea cauzează citirea unui nou job în altă porțiune din memoria principală. O dată ce a fost citit noul job, procesorul va întâlni în monitor o instrucțiune care îi spune să proceseze acest nou job. Procesorul va executa instrucțiunile din programul utilizator până la terminarea acestuia sau până la întâlnirea unei erori. Limbajul de control al joburilor era utilizat pentru furnizarea de instrucțiuni pentru programul monitor. Era vorba de exemplu despre ce compilator trebuie utilizat respectiv despre ce tip de date va fi manipulat.

Practic am observat că monitorul este un simplu program de calculator. Este abilitatea procesorului de a aduce instrucțiuni din diferite zone ale memoriei. Sunt dezirabile alte caracteristici hardware cum ar fi:

- protejarea memoriei: în timpul execuției programului monitor trebuie evitată alterarea locațiilor de memorie care conțin programul monitor; în caz contrar, hardware-ul trebuie să detecteze acest lucru și să transfere controlul programului monitor;

- timer: un timer este folosit pentru a evita monopolizarea procesorului de către un singur job;
- instrucțiuni privilegiate: câteva instrucțiuni sunt considerate privilegiate și trebuie luate în considerare doar de către monitor. Dacă procesorul întâlnește astfel de instrucțiuni la executarea unui program utilizator trebuie să-și întrerupă execuția și să genereze o întrerupere. În cazul în care un program utilizator vrea să execute o operație de citire sau scriere la un dispozitiv de I/O trebuie să se adreseze procesorului;
- întreruperi: modelele mai vechi nu aveau prevăzute întreruperi dar duc la creșterea flexibilității monitorului.

Următorul pas în evoluția calculatoarelor și implicit al sistemelor de operare l-a constituit apariția circuitelor integrate. Odată cu evoluția sistemelor de operare au apărut concepțele de multiprogramare sau multitasking. Primul sistem de operare care a încercat să trateze problemele atât în cazul sistemelor mari de calcul cât și a calculatoarelor mai puțin performante a fost 360/OS.

Cu toate că multiprogramarea este eficientă, este nevoie de obținerea unui mod de lucru pentru o interacțiune mai bună a utilizatorului cu calculatorul. Din acest motiv a fost necesară apariția sistemelor cu time-sharing (timp distribuit). În aceste cazuri timpul procesorului este distribuit între mai mulți utilizatori. În astfel de sisteme mai mulți utilizatori accesază sistemul prin intermediul mai multor terminale.

Diferențele între cele 2 sisteme prezentate constau în faptul că obiectivul principal îl constă maximizarea timpului de utilizare al procesorului pentru sistemele de multiprogramare în loturi și minimizarea timpului de răspuns pentru cele cu time-sharing. În ceea ce privește modul de obținere al instrucțiunilor de către sistemul de operare, în primul caz apare limbajul de control al joburilor iar în cel de al doilea sunt introduse comenzi de la terminal.

În partea a doua a anilor '80 microcalculatoarele cunosc o dezvoltare extraordinară. Se disting 2 evoluții principale în ceea ce privește sistemele de operare. Pornind de la sistemul de operare DOS, pe care îl achiziționează de la firma care l-a dezvoltat, Bill Gates dezvoltă MS-DOS, sistem de operare distribuit împreună cu calculatoarele IBM. Inspirându-se probabil din interfața Apple Macintosh, Microsoft scoate o interfață Windows 3.1 pentru sistemul de operare DOS iar apoi Windows 95 care voia să fie un sistem de operare chiar dacă pornea din MS-DOS. Modificări ușoare au condus la Windows 98. Un sistem de operare care lucra pe 32 de biți a fost Windows NT, cu versiunea ulterioară denumită comercial Windows 2000. Evoluțiile ulterioare au însemnat Windows XP respectiv recent, Vista.

Cealaltă abordare are ca punct de pornire sistemul de operare Multics și are ca rezultat UNIX. Datorită faptului că era disponibil codul acestuia, unele companii și-au dezvoltat propriile versiuni fiind necesar astfel apariția unui standard, și anume POSIX. Linus Torvalds va porni de la o variantă redusă de UNIX și va scrie ceea ce va deveni Linux. Diferite distribuții de Linux au evoluat și sunt prezente pe piață ca o alternativă la produsele oferite de Microsoft.

# Shell scripting (1)

## 1. Structura de fișiere

Una dintre cele mai importante caracteristici Linux este modul în care tratează fișierele. În Linux totul este organizat sub formă de, adică programele pot utiliza fișiere de pe discuri, porturile seriale, imprimantele și alte dispozitive în același mod în care utilizează fișierele. Acest fapt facilitează programarea, deoarece nu mai trebuie avut grija dacă se lucrează cu un terminal, fișier pe disc sau cu o imprimantă.

Sistemul de operare Linux utilizează trei tipuri de fișiere:

- Fișiere obișnuite – sunt doar colecții de octeți asupra cărora sistemul nu impune nici o structură particulară. Acestea sunt utilizate pentru fișierele text, fișierele de date pentru programe și chiar pentru fișierele executabile ale programelor.
- Cataloge – un catalog este un mecanism care permite seturilor de fișiere să fie grupate împreună. Conceptual reprezintă un container în care pot fi plasate alte fișiere și cataloge. De fapt catalogul conține doar numele fișierelor și informații despre locația acestora. Deoarece un catalog poate conține sub cataloge avem de-a face cu o structură ierarhică.
- Dispozitive – această categorie este alcătuită din anumite tipuri de fișiere, cum ar fi cele asociate comunicației între procese sau cele asociate comunicației cu diverse dispozitive hardware ca terminale, CD-ROM, modem, etc.

Toate aceste fișiere sunt organizate într-o structură ierarhică arborescentă, având ca și vârf un singur catalog care poartă numele de "rădăcină", el fiind reprezentat prin următorul simbol "/". Acest catalog "rădăcină" conține o serie de alte cataloge ce au destinații speciale. Aceste destinații sunt doar convenții și nu sunt obligatorii.

Semnificația acestor cataloge:

/bin	executabilele comenziilor
/dev	fișierele speciale
/etc	fișierele de configurare
/home	catalogele utilizatorilor
/lib	bibliotecile standard
/mnt	se vor mount-a alte sisteme de fișiere(partiții)
/sbin	comenzi de administrare
/tmp	fișiere temporare
/usr	date disponibile utilizatorilor
/var	fișiere mari în cazuri speciale

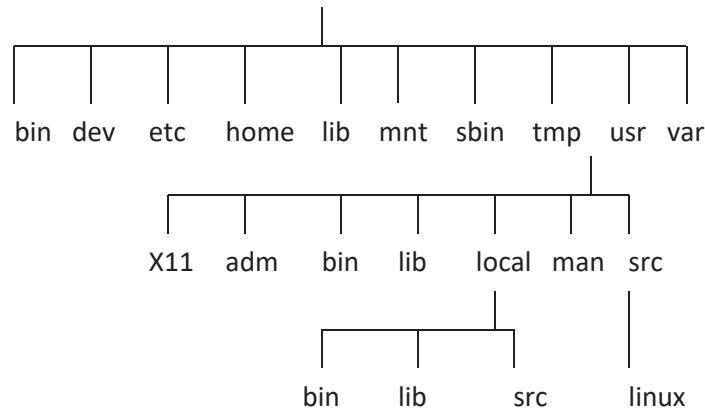


Fig. 1. Ierarhia de catalogage sistem

Este posibil ca două fișiere să aibă același nume atât timp cât sunt în catalogage diferite, referirea acestor fișiere făcându-se prin intermediul numelui de cale. Acesta poate fi:

- absolut ca punct de referință se ia catalogul "rădăcină", ex:

```
/home/user/a  
/home/dev/a
```

- relativ punctul de referință este reprezentat de un catalog arbitrar:

```
/user/a  
/user/b
```

La fiecare schimbare a catalogului noul catalog devine catalogul curent de lucru. Acest lucru este foarte important deoarece pentru fiecare program executat în Linux va rezulta un proces care va avea ca parte a stării lui interne și catalogul curent de lucru. Acesta fiind catalogul utilizat ulterior pentru referiri bazate pe calea relativă. Dacă procesul nu face aranjamente specifice pentru a utiliza un alt catalog de lucru, atunci implicit va utiliza catalogul în care a fost utilizatorul când programul a fost pornit.

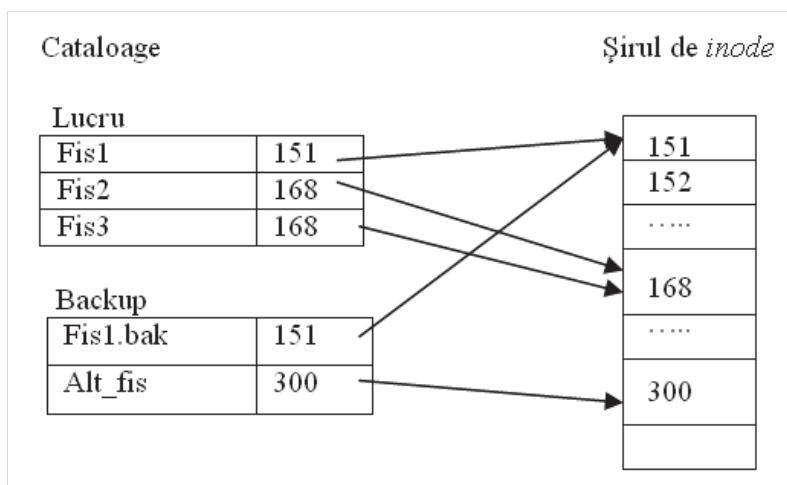


Fig. 2 Fișiere – organizare fizică

Fiecare fișier, stocat pe disc, indiferent de tipul său îi este asociat un număr numit inode care reprezintă de fapt un index al unei intrări într-un sir stocat pe disc. Fiecare element al sirului este un inode care stochează în formății specifice unui anumit fișier (când a fost creat, cine este proprietarul

și unde sunt stocate blocurile de date pentru fișierul respectiv). Un catalog conține pe lângă numele fișierului și numărul de inode asociat acestuia. Fiecare pereche nume – inode dintr-un catalog poartă numele de legătură sau link. În acest fel este posibil ca același nume de inode să apară în unul sau mai multe link-uri. Aceasta ar însemna că un fișier poate avea unul sau mai multe nume de cale valide.

## 2. Comenzi uzuale

Imediat după intrarea în sistem, implicit utilizatorul va fi plasat într-un catalog asociat cu numele de login numit home. Acesta va fi catalogul curent de lucru inițial.

- pwd - present working directory(catalog de lucru prezent). Afisează calea absolută pentru catalogul curent.
- cd – change directory.
- ls – Lista cu intrările din catalogul curent. În funcție de opțiuni lista afișată se poate modifica.

Comenzi pentru lucrul cu directoare:

- mkdir – creează un nou director
- rmdir – șterge directorul/directoarele primite ca și argument.

Comezi pentru lucrul cu fișiere:

- cp – comanda de copiere.
- touch – actualizează timpul de acces al fișierului primit ca și argument. Dacă fișierul nu există atunci va fi creat.
- rm – comanda de ștergere.
- cat – poate avea mai multe obiective: afisează, creează, concatenează fișierele primite ca și argument.
- Grep - permite căutarea unui nume sau a unui format de nume în fișierul primit ca și argument. Va deschide fișierul respectiv și va afișa fiecare linie din fișier în care se găsește cuvântul căutat.

Interpretorul de comenzi

## 3. Prezentare generală interpreter de comenzi

Interpretorul de comenzi este un program care realizează o interfață între utilizator și sistemul de operare, interfață care preia comenziile utilizatorului și le transmite sistemului de operare spre execuție. Pentru UNIX și clonele acestuia (Linux) interpretoarele de comenzi sunt independente față de sistemul de operare ceea ce a dus la dezvoltarea unui număr mare de astfel de interpretoare.

Primul interpretor de comenzi important, numit Bourne shell, a fost dezvoltat de Steven Bourne și a fost inclus într-o distribuție de UNIX lansată în 1979. Acest interpretor de comenzi este cunoscut în cadrul sistemului de operare sub numele de sh. Un alt interpretor de comenzi utilizat pe scară largă este C shell cunoscut și sub numele de csh. Csh a fost scris de Billy Joy ca parte integrantă a unei distribuții de UNIX numită BSD (Berkeley System Distribution), și a apărut la câțiva ani după sh. Numele de Csh l-a primit datorită asemănării sintaxei comenziilor sale cu cele de instrucțiuni limbajului C, fapt ce îl face mult mai ușor în utilizare celor familiarizați cu limbajul C. Toate aceste interpretoare au fost inițial dezvoltate pentru Unix dar s-au dezvoltat versiuni ale acestora și pentru Linux.

Unul dintre cele mai cunoscute interpretoare de comenzi pentru Linux este bash (bourne again shell). La intrarea în sistem fiecare utilizator primește o copie a interpretorului de comenzi. Acesta păstrează un mediu distinct pentru fiecare utilizator din sistem. În multe sisteme Linux comanda /bin/sh este de multe ori un link către interpretorul de comenzi utilizat de către sistemul de operare. Interpretorul de comenzi din Linux deși are unele similitudini cu procesorul de comenzi DOS este un instrument mult mai puternic oferind utilizatorului o multitudine de facilități.

#### 4. Caracteristici

*Expandarea numelor de cale* – În exemplele de până acum s-au folosit numai nume de cale complet precizate. Există mai multe metode prin care numele de cale să fie abreviate, iar interpretorul de comenzi va expanda aceste nume înainte de a le transmite comenziilor. Un astfel de caracter de abreviere este \*. De exemplu

```
$ cp text/* backup
```

Aici prin \* se înțelege – toate fișierele (cu excepția celor ce încep cu .) din catalogul text. Sensul comenzi este – toate fișierele din catalogul text vor fi copiate în catalogul backup. Un alt caracter special utilizat în numele fișierelor pe care interpretorul de comenzi le va expanda este semnul întrebării ?. Rolul acestuia este de a substitui un singur caracter :

```
$ ls /dev/tty?  
/dev/tty0 /dev/tty2 /dev/tty4 /dev/tty6 /dev/tty8  
/dev/tty1 /dev/tty3 /dev/tty5 /dev/tty7 /dev/tty9
```

De asemenea este posibilă precizarea unei liste de caractere, și numai caracterele conținute în această listă să poată fi expandate. Această listă este delimitată de paranteze drepte ([]). Ex:

```
$ ls /dev/tty[pq][235]  
/dev/ttyp2 /dev/ttyp3 /dev/ttyp5 /dev/ttyq2  
/dev/ttyq3 /dev/ttyq5
```

O ultimă formă de expandare pentru numele de cale este prin precizarea unei liste de cuvinte, cuvintele vor fi separate prin virgulă și cuprinse între paranteze {}. Ex:

```
$ mkdir /usr/tmp/{bin,doc}
```

Toate numele de cale sunt expandate de interpretorul de comenzi înainte a fi transmise către comenzi. În cazul în care se dorește utilizarea caracterelor speciale cu sensul lor inițial se poate utiliza unul din următoarele trei mecanisme de marcare:

- caractere escape(\) – caracterul backslash indică interpretorului de comenzi să ignore sensul special al caracterelor de după marcaj. Ex: \\*.
- ghilimele ‘ ’ – orice sir de caractere cuprins între aceste marcaje își pierde sensul special al tuturor caracterelor cuprinse în el. Ex: ‘a?’
- ghilimele duble “ ” – un sir de caractere pierde sensul special al caracterelor cuprinse în el (este valabil numai pentru caracterele prezentate până acum), cu excepția caracterelor \$ și \. Ex:

```
`cmd`
```

Executa comanda cmd. De exemplu: \$ cale=`pwd` atribuie variabilei cale rezultatul comenzi pwd.

*Redirectarea ieșirii și a intrării* – Filosofia de bază Linux este aceea de ține lucrurile cât mai simple posibil, prin punerea la dispoziția utilizatorului a unui număr mare de funcții simple. Iar prin gruparea acestor funcții împreună se pot obține comenzi mai complexe. Implicit, majoritatea

comenzilor Linux sunt configurate astfel încât intrarea acestora să o reprezinte tastatura iar datele de ieșire sunt trimise către monitor. În unele situații ar fi util dacă intrarea , respectiv ieșirea unei comenzi ar fi altele decât cele implicate. Unele comenzi au această facilitate, dar dacă pentru fiecare în parte să ar prevedea acest lucru dimensiunea executabilelor ar crește, de aceea această sarcină și-a asumat-o sistemul de operare prin intermediul acestui mecanism de redirectare.

Un exemplu clasic este cel de creare a unui nou fișier prin care ieșirea standard a comenzi cat este redirectată către fișierul care urmează a fi creat:

```
$ cat > text  
Un text de exemplu  
Ctrl-d  
$
```

Se observă că în această situație se folosește caracterul '>'. Acesta are rolul de a indica redirectarea ieșirii standard către o altă comandă. Dacă este urmat de un nume de fișier atunci acest fișier este creat dacă nu există, iar dacă există atunci va rescris. Un alt simbol utilizat de acest mecanism de redirectare este '>>'. În cazul în care ieșirea este reprezentată de un fișier datele redirectate vor fi concatenate la sfârșitul fișierului. Ex:

```
$ ls -l /usr/bin >>text
```

Rezultat: toate numele fișierelor din /usr/bin, împreună cu informațiile despre acestea sunt adăugate la fișierul text. Tot la fel cu redirectarea ieșirii se poate redirecta și ieșirea de eroare:

```
$ ls /usr/bin 2>text.err
```

La fel cum se poate realiza redirectarea ieșirii standard, la fel se poate redirecta și intrarea standard:

```
$ wc -l < /etc/passwd  
12
```

*Pipes sau conectarea proceselor* – prin redirectarea ieșirii standard a unei comenzi într-un fișier, iar apoi redirectarea aceluiași fișier ca și intrare standard pentru altă comandă se pun practic în legătură două procese prin intermediul unui fișier temporar. Acest lucru se poate face implicit de către sistemul de operare prin intermediul unui pipe. Pentru aceasta se folosește operatorul |.

De exemplu seria de comenzi:

```
$ ls /usr/bin > /tmp/temp.txt  
$ wc -w </tmp/temp.txt  
500  
$ rm /tmp/temp.txt
```

poate fi înlocuită prin:

```
$ ls /usr/bin | wc -w  
500
```

*Controlul proceselor* – Permite întreruperea unui proces și de asemenea reluarea execuției acelui proces la un moment de timp ulterior. Întreruperea execuției unui proces se poate face direct de la tastatură prin secvența Ctrl-z, după care procesul va fi întrerupt iar utilizatorul reprimește controlul putând executa alte comenzi.

```
$ cat > file.txt  
Ctrl-z  
[1]+ Stopped  
$
```

Pentru gestionarea proceselor interpretorul de comenzi are implementate o serie de comenzi:

- fg – este comanda prin care este reluată execuția unui proces, aceasta poate primi ca și parametru numărul de ordine al procesului întrerupt. Pentru a relua execuția ultimului proces întrerupt avem:

```
$ fg %1  
cat >file.txt
```

- bg – se folosește pentru suspendarea unui proces, echivalent cu Ctrl-z
- jobs – afișează procesele suspendate.

Alte facilități ale interpretorului de comenzi Linux:

- history: posibilitatea de rechemare rapidă a ultimilor comenzi executate.
- command completion: afișarea posibilelor comenzi pornind de la primele caractere ale numelor acestora.

## 5. Sintaxa

Programarea în cazul interpretorului de comenzi se face în același mod ca și în cazul altor limbaje de programare. Și în acest caz vom avea variabile care în mod normal nu sunt declarate decât atunci când este nevoie de ele. Implicit variabilele sunt memorate și considerate drept siruri de caractere, chiar și atunci când primesc valori numerice. Variabilele pot fi de mai multe tipuri: variabile predefinite sau speciale, parametrii poziționali sau variabile utilizator.

*Variabile predefinite sau speciale* – la pornirea sistemului o serie de variabile sunt inițializate cu valorile implicate mediului de operare. Cele mai importante sunt următoarele:

\$HOME – catalogul gazdă a utilizatorului curent  
\$PATH – o listă de cataloage în care interpretorul caută pentru fișierul executabil asociat unei comenzi.  
\$SHELL – numele interpretorului de comenzi curent.  
\$PS1 – definește prompterul interpretorului, implicit acesta este \$.  
\$IFS – Internal Field Separator utilizat pentru separarea cuvintelor și a liniilor .  
\$0 – numele fișierului de comenzi.  
\$# – numărul de parametri transmiși fișierului de comenzi.  
\$\$ – numărul de identificare prin care sistemul identifică scriptul în momentul execuției.

*Parametrii pozitionali* – Dacă la apelarea unui fișier de comenzi (script) sunt folosite parametrii atunci sunt create câteva variabile suplimentare. Chiar dacă nu se folosesc parametrii tot se creează variabila \$# dar va avea valoarea 0. Variabile de acest tip sunt:

\$1, \$2, .. – parametrii transmiși scriptului.  
\$\* - lista cu parametrii transmiși scriptului.

Accesarea conținutului variabilei se face prin prefixarea acesteia cu caracterul \$. Ex:

```
$ salut=Hello  
$ echo $salut  
Hello
```

Pentru ca o variabilă să fie cunoscută unui subshell această comandă trebuie să fie exportată. Aceasta se realizează cu ajutorul comenzi `export`.

*Structuri de control* – și în acest caz se întâlnesc structuri de control la fel ca și în cazul altor limbaje de programare.

*if* – testează rezultatul unui comenzi și în funcție de rezultatul acestuia se execută una dintre ramurile de declarații. Format:

```
if condiție
then declarații
[ elif condiție
then declarații]
[ else declarații ]
fi
```

Se execută condiție; dacă codul returnat este zero se execută declarațiile ce urmează primului `then`. În caz contrar, se execută condiția de după `elif` și dacă codul returnat este zero se execută declarațiile de după al doilea `then`. Ex:

```
if test -f $1
then echo $1 este un fișier obișnuit
elif test -d $1
then echo $1 este un catalog
else echo $1 este altceva
fi
```

*for* – folosit pentru trecerea printr-un anumit număr de valori (pot fi și siruri de caractere). Sintaxă:

```
for nume [in valori]
do declarații
done
```

Variabila `nume` ia pe rând valorile din lista ce urmează lui `in`. Pentru fiecare valoare se execută ciclul `for`. Dacă `in valori` este omis, ciclul se execută pentru fiecare parametru pozițional. Condiția poate fi și `in *`, caz în care variabila `nume` ia pe rând ca valoare numele intrărilor din directorul curent.

*while* – în cazul în care se dorește trecerea printr-un număr mai mare de valori este utilă folosirea acestei structuri de control. Sintaxă:

```
while condiție
do declarații
done
```

*until* – ciclu identic cu `for` dar în acest caz testarea condițiilor se face la sfârșit. Sintaxă:

```
until lista_1
do lista
done
```

*case* – această structură are sintaxa:

```
case condiție in
șablon_1) declarații;;
```

```
şablon_2) declaraţii;;  
...  
esac
```

Se compara condiție cu fiecare din şabloanele prezente si se executa lista de comenzi unde se constata potrivirea.

*Proceduri shell* – reprezintă o modalitate de a grupa mai multe comenzi shell la un singur loc în vederea execuției lor ulterioare. Forma generală a unei astfel de proceduri este:

```
nume () { lista; }
```

Unde nume este numele procedurii (sau funcției), iar lista este o listă de comenzi ce urmează să fie executate. Ex:

```
$ lsl() { ls -l; }
```

Câteva comenzi specifice interprotoarelor de comenzi:

*Test* – este folosită pentru a testa o expresie dacă este adevărată atunci test va returna 0, în caz contrar va returna 1. Test poate fi utilizat pentru operații matematice, cu siruri de caractere sau fișiere:

Operații matematice:

- eq – egalitate
- ne – diferență
- lt – mai mic
- le – mai mic sau egal
- gt – mai mare
- ge – mai mare sau egal

Operații cu siruri de caractere:

- şir1 = şir2 – egalitate
- şir1 != şir2 – diferență
- n şir1 – diferit de NULL
- z şir1 – este NULL

Operații cu fișiere:

- s test -f fișier – adevărat dacă fișierul nu este gol
- f test -f fișier – adevărat dacă fișierul este unul obișnuit
- d test -d fișier – adevărat dacă fișierul este catalog
- w test -w fișier – adevărat dacă fișierul poate fi deschis pentru scriere
- r test -r fișier – adevărat dacă fișierul poate fi deschis doar pentru citire
- x test -x fișier – adevărat dacă fișierul este executabil
- e test -e fișier – adevărat dacă fișierul există

Operatori logici

- ! expresie – negare
- expresie1 -a expresie2 – logic ŞI
- expresie1 -o expresie2 – logic SAU

*Expr* – identificator pentru operații matematice. Ex: \$ expr 1 + 3

*Export* – face disponibile variabilele primite ca și argument în subshell-urile viitoare. Implicit variabilele create într-un shell nu sunt disponibile într-un shell creat ulterior.

*Echo* – afișează argumentele la ieșirea standard

*Eval* – realizează evaluarea și apoi execuția argumentelor

*Exit* – forțează întreruperea procesului curent care va returna ca și cod de ieșire valoarea primită ca și argument. Ex: Exit n

*Read* – citește o linie din fișierul standard de intrare.

*Break* – Comanda de părăsire a celei mai interioare bucle for, while sau until ce conține break. Dacă n este specificat se ieșe din n bucle.

*Continue* – comanda permite trecerea la o nouă iterație a buclei for, while sau until.

*Return* – revenirea dintr-o funcție cu valoarea n, unde n este valoarea transmisă ca și argument. Dacă această valoare nu este precizată atunci codul returnat este cel al ultimei comenzi executate.

*Set* – folosit pentru activarea sau dezactivarea unor opțiuni sau pentru poziționarea unor parametrii poziționali.

*Sleep* – suspendă execuția pentru un număr n de secunde primit ca și argument.

# Shell scripting (2)

## 1. Interfața cu utilizatorul

În cadrul unui program shell script interfața cu utilizatorul se poate realiza prin două mecanisme:

- Prin argumente transmise în linia de comandă. Această metodă este de altfel preferată pentru că permite gruparea într-o secvență de comenzi înlănțuite, ieșirea comenzi precedente devenind intrare pentru comanda următoare. La baza acestui mecanism sunt comenzile pipe, și de redirectare a intrării/ieșirii

```
ls -lR /home/student | grep ".tar.gz" | uniq | wc -l
```

- Prin comenzi ce permit interacțiunea cu terminalul, respectiv cu utilizatorul. Reprezentative pentru această categorie sunt *echo* și *read*.

**Echo** – afișează la ieșirea standard mesajul primit ca și argument.

**Read** – citește valoarea primită de la intrarea standard și o salvează în variabila primită ca și argument.

```
echo „Nume:”
read nume
echo „Varsta:”
read varsta
calc=`expr $varsta + 1`
echo „Anul viitor $nume va avea $calc ani”
```

Mecanismele de interfațare cu utilizatorul, includ, de obicei, metode mai avansate de interfațare, ce permite crearea de interfețe elaborate. În cazul shell script, nu avem la dispoziție astfel de mecanisme, sigură excepție e comanda *dialog*. Comandă ce poate fi găsită doar pe anumite distribuții Linux. Mai jos se poate vedea un exemplu simplu de meniu ce permite interacțiunea cu utilizatorul.

```
while :
do
    clear
    echo "-----"
    echo " Main Menu "
    echo "-----"
    echo "[1] Data"
    echo "[2] Fisierele din directorul curent"
    echo "[3] Calendar"
    echo "[4] Start editor"
    echo "[5] Exit"
    echo "====="
    echo -n "Optiuni [1-5]: "
```

```

read opt
case $opt in
    1) echo "Astazi este `date` , apasa o tasta. . ." ; read ;;
    2) echo "Fisiere in `pwd`" ; ls -l ; echo "Apasa o tasta. . ." ; read ;;
    3) cal ; echo "Apasa o tasta. . ." ; read ;;
    4) nano ;;
    5) exit 0 ;;
*) echo "Ups!!! Alege 1,2,3,4, sau 5"; echo "Apasa o tasta. . ." ; read ;;
esac
done

```

## 2. Liste

Aceste tipuri de construcții oferă posibilitatea de rula secvențe de comenzi, putând înlocui structuri complicate de tipul *if/then*. Aceste construcții sunt de două tipuri:

- Liste de tip SI: *comanda1 && comanda2 && comanda3 && ... && comanda*. Secvența de comenzi se execută pe rând dacă rezultatul comenzi anterioare întoarce valoarea *true* (zero). La prima valoare de *false* execuția lanțului de comenzi se întrerupe.

```

if [ ! -z "$1" ] && echo "Argument #1 = $1" && [ ! -z
"$2" ] && \
echo "Argument #2 = $2"
then
    echo "Toate argumentele sunt definite"
else
    echo "Mai putin de 2 argumente au fost transmise
scriptului"
fi

```

- Liste de tip SAU: *comanda1 || comanda2 || comanda3 || ... || comanda*. Spre deosebire de lista și, comenziile din secvență vor fi executate atât timp cât valoarea întoarsă este *false*. La prima valoare de *true*, execuția este suspendată.

```
[ ! -f "$1" ] && echo "File \"\$1\" not found."
[ ! -f "$1" ] || (rm -f $1; echo "File \"\$1\" deleted.")
```

## 3. Siruri

O facilitate recent introdusă sunt sirurile unei dimensiuni. Elementele sirurilor pot fi initializate astfel: *variabila[index]*. Pentru a accesa conținutul unui element al sirului se folosește următoarea notație: \${variabila[index]}

```
data[10]=1
data[11]=string
echo ${data[10]} #1
echo ${data[5]} #empty
echo ${data[11]} #string
```

De asemenea se poate inițializa întregul sir astfel: *sir=( element1 element2 ... elementN )*. Pentru a realiza operații asupra tipului de date sir au fost adaptăți o serie de operatori specifici Bash

```
sir=( zero unu doi trei patru cinci )
#accesare continut
echo ${sir[0]} #zero
echo ${sir:0} #zero
echo ${sir[1]:1} #nu

#lungime element
echo ${#sir[0]} #4
echo ${#sir[1]} #3

#lungime sir
echo ${#sir[*]} #6
echo ${#sir[@]} #6

echo ${sir[@]} #zero unu doi trei patru cinci
```

Așa cum se poate vedea mai sus expresiile  *\${sir[@]}*  sau  *\${sir[\*]}*  se referă la toate elementele din sir. Pentru a determina lungimea, atât pentru sir cât și pentru elemente se folosește operatorul  *# \${#sir[@]}* , respectiv  *\${#sir[index]}*

Spre deosebire de variabilele normale, un sir poate fi declarat prin  *declare -a* , interpretorul de comenzi considerând variabila astfel definită ca fiind sir

```
declare -a culori
echo "Introduceti o lista de culori separate prin spatiu:"
read -a culori
echo "Count ${#culori[@]}"
required=rosu
present=0
for i in ${culori[*]}
do
    echo $i
    if [ $i = $required ]; then
        present=1
    fi
```

```

done
if [ $present -eq 0 ]; then
    echo "$required lipseste ..."
    culori[$[#culori[@]]]=$required
fi
echo "${culori[*]}"

```

#### 4. Expresii regulate

Sunt seturi de caractere ce permit identificarea unor modele în cadrul șirurilor de caractere. O expresie regulată conține următoarele:

- Un set de caractere – definesc secvența căutată. Cea mai simplă formă a unei expresii regulate constă dintr-un singur caracter.
- O ancore – definește poziția în cadrul liniei de text, unde expresia regulată caută modelul, Ex: ^ - începutul șirului, \$ - finalul șirului.
- Modificatori – extind sau limitează aria textului în care expresia regulată caută modelul. Din această categorie fac parte: \*, [], \.

Expresiile regulate sunt destinate operațiilor cu șiruri de caractere. Ele operând pe șiruri de caractere de la un caracter până la parte din șir sau șirul complet. Expresiile regulate nu pot fi folosite direct în operațiile cu șiruri de caractere din cadrul bash, ci doar prin intermediul unor comenzi de tipul grep sau awk.

Operatori:

- \* - indică zero sau mai multe repetări ale expresiei precedate. Ex „aab\*” e corectă pentru aab,aabb,aabb ...
- . – indică un singur caracter. Ex „ab.” Secvențele text pentru care expresia e corectă o să înceapa cu ab și trebuie să conțină cel puțin încă un caracter (abb), dar nu e corectă pentru ab simplu.
- ^ - în funcție de context poate avea două semnificații: indică începutul liniei sau neagă semnificația unui set de caractere în cadrul expresiei regulate

```
grep '^student' /etc/password
```

- \$ - indică finalul liniei:
  - o „xx\$” – secvența căutată trebuie să apară la finalul liniei
  - o „^\$” – caută o linie goală

```
grep '^$' /etc/password
grep '/bin/bash$' /etc/password
```

- [] – definesc lista de caractere ce pot apărea în secvența căutată:
  - o „[xyz]” – oricare din caracterele x,y,z
  - o „[a-e]” – oricare caracter de la a la e
  - o „[A-Ea-e]” – lista include și literele mari
  - o „[0-9]” – doar cifre
  - o „[^b-d]” – toate caracterele cu excepția celor de la b la d

```
grep "[Ss]tudent" /etc/password
```

- \ - are rol de caracter *escape* – caracterele prefixate de \ își pierd sensul special și sunt utilizate ca simple caractere. Ex \\$
- <> - secvența căutată este un cuvânt, delimitat de spații albe.

```
cat > text
cuvantul e delimitat de spatii albe
cu <> se poate cauta un cuvant
Ctrl-d
```

```
grep '\<cu\>' text
```

## 5. Awk

Este o comandă externă bash, ce aduce funcționalitate suplimentară în partea de manipulare de date. Sintaxa generală este:

```
awk -f {program awk} fisier
```

Un program awk va avea următoare structură generală:

Model {

    Acțiune 1

    Acțiune 2

...

} END

Awk va citi linie cu linie de la fișierul primit ca și argument (sau de la intrarea standard), va compara respectiva linie cu modelul (dacă un model este definit). Dacă modelul potrivește, atunci va aplica asupra liniei respective acțiunile definite.

Model poate fi:

- Expresie regulată – folosită pentru filtrarea înregistrărilor
- BEGIN – bloc ce definește acțiunile ce au loc înainte de procesarea înregistrărilor

END – definește acțiunile ce au loc după procesarea înregistrărilor

```
cat > lista
pop 10
marin 9
dorin 10
dorel 8
ionel 7
Ctrl-d
```

```
awk /do*/ '{print $1":"$2}' lista
dorin:10
dorel:8
```

În exemplul de mai sus fiecare linie reprezintă o înregistrare, iar fiecare cuvânt din linie reprezintă un câmp. Un câmp este delimitat print-un spațiu, iar identificarea câmpului se face prin variabile poziționale speciale \$1,\$2 ...

Awk are un set de variabile predefinite:

- FILENAME – numele fișierului de intrare
- RS – separatorul liniei de intrare (Implicit caracterul linie nouă)
- OFS – separatorul pentru câmpul de ieșire
- ORS – separatorul liniei de ieșire
- NF – numărul de câmpuri pe linia curentă
- NR – numărul înregistrării
- FS – separatorul de câmp (implicit caracterul spațiu sau tab)

```
cat      /etc/passwd      |      grep      'bash$' |      awk
'BEGIN{FS=":" } {print NR": "$1" "$NF}'
1: root /bin/bash
2: mysql /bin/bash
3: b624 /bin/bash
4: student /bin/bash
```

Awk fiind bazat pe limbajul C oferă posibilitatea de a realiza operații matematice:

```
$ls -l fisiere
-rw-rw-r--. 1 student student 40 Mar 3 20:25 lista
-rw-rw-r--. 1 student student 305 Mar 3 20:25 sir
-rw-rw-r--. 1 student student 68 Mar 3 20:25 text
$ls -l fisiere | awk 'BEGIN
{total=0}{total+=$5}END{print "Total:"total}'
Total:413
```

Totodată permite definirea de structuri de control de tipul if, for, while asemănătoare ca sintaxă limbajului C.

## 6. Subshells și shell-uri restricționate

La lansarea în execuție a unui shell script, interpretorul de comenzi creează nu proces nou numit subshell. În aceeași manieră în care o comandă este lansată în execuție în cadrul unui proces separat, un shell script va rula o secvență de comenzi dar într-un proces copil (subshell) al interpretorului de comenzi. Un shell script poate lansa la rândul lui procese copil, ceea ce permite o execuție paralelă a scriptului.

Definirea explicită a unui subshell se realizează prin (). Ex: (comanda1; comanda2; comanda3; ...). O listă de astfel de comenzi va rula într-un subshell.

```
var=old
(var=new)
```

```
echo $var
```

Variabilele definite într-un subshell nu sunt vizibile în afara blocului de cod definit de paranteze. Ele nu sunt accesibile procesului părinte, fiind de fapt variabile locale procesului fiu. De asemenea schimbările de directoare nu sunt vizibile în cadrul procesului părinte.

O posibilă utilizare a acestui mecanism constă în definirea unui mediu de rulare particularizat pentru anumite comenzi:

```
COMANDA1
COMANDA2
COMANDA3
(
    IFS=:
    PATH=/bin
    COMANDA4
    COMANDA5
    exit 3
)
# Parintele nu a fost modificat.
COMANDA6
COMANDA7
```

O secvență de cod definită în paranteze {} nu lansează un nou subshell.

Shellurile restricționate reprezinta o caracteristică specială, ce poate fi utilizată în cazul în care scriptul urmează să fie rulat în medii în care anumite comenzi e necesar să fie dezactivate. Această măsură limitează privilegiile cu care este rulat scriptul, pentru a minimiza posibilele daune.

Modul de rulare restricționat pentru un script poate fi activat astfel:

- Folosind `#!/bin/bash -r` în antetul scriptului – întreg scriptul rulează în modul restricționat
- Folosind comanda `set -r` – doar secvența de cod ce urmează va fi restricționată

Următoarele comenzi și acțiuni sunt dezactivate:

- Cd posibilitatea de a schimba directorul curent
- Modificarea variabilelor `$PATH`, `$SHELL`, `$BASH_ENV` sau `$ENV`
- Redirectarea ieșirii
- Invocarea comenziilor ce conțin `/`
- Invocarea comenii `exec`
- Anularea modului restricționat

```
set -r
echo "Director curent `pwd`"
cd ..
echo "Director curent `pwd`"
```

# Make, Fișiere

## 1. Make – Introducere

Make este una dintre uneltele care îmbunătățesc eficiența în medii de lucru de genul Linux/UNIX. În principiu este un generator de comenzi care pe baza unui fișier de descriere creează o secvență de comenzi ce trebuie executate de către interpretorul de comenzi Linux. Make este o unealtă folosită în general pentru determinarea relațiilor de dependență între fișiere. În situația în care se lucrează la un proiect ce implică utilizarea mai multor fișiere sursă, dacă unul sau mai multe fișiere sursă sunt modificate atunci procesul de recompilare trebuie reluat, dar în general nu este necesar ca acest proces să fie reluat în întregime de obicei fiind necesară recompilarea doar a acelor surse ce au fost modificate. În acest context make este o unealtă esențială putând determina dependențe ce pot apărea între diverse fișiere. Ca și concept de programare în ansamblu make este non-procedural, deși intrările individuale se execută. Comenziile ce trebuie executate sunt determinate printr-un proces iterativ de determinare a dependentelor fișierelor ce trebuie prelucrate. Forma generală de apelare a acestui utilitar este:

```
$make program
```

Această sintaxă indică faptul că se dorește executarea tuturor operațiilor de determinare a dependentelor între fișiere și de compilare a acestora necesare pentru crearea executabilului program. În această situație apare noțiunea de ţintă care este reprezentată de program, totodată acesta este construit din una sau mai multe dependențe, fiecare dintre acestea putând avea la rândul lor ţinte pentru alte dependențe. Caracteristica esențială a acestui utilitar este aceea că fiecare nouă apelare a acestuia va determina evaluarea în lanț tuturor dependentelor și prelucrarea în final doar a celor pentru care s-au constatat modificări. Funcționarea utilitarului make se bazează pe structură ierarhică de tipul sursă-obiect și obiect-executabil. Programatorul este responsabil pentru precizarea unor dependențe prin intermediul unui fișier numit makefile, restul fiind determinate pe baza timpului ultimei modificări și a unui set de reguli.

Fișierul de descriere:

Să presupunem că avem un program destinat unor prelucrări de date alcătuit din trei fișiere: main.c, interfata.c, prelucrari.c. O compilare manuală a programului arată astfel:

```
$ gcc main.c  
$ gcc interfata.c  
$ gcc prelucrare.c  
$ gcc -o exemplu main.o interfata.o prelucrari.o  
/usr/lib/my_lib.a
```

În acest caz particular întreaga compilare a programului se poate face într-o singura linie de comandă, dar în cazul proiectelor de mari dimensiuni la care lucrează mai multe persoane acest lucru este practic imposibil. Pentru a putea utiliza make trebuie creat un fișier de descriere numit makefile care în acest caz are următoare structură:

```

exemplu : main.o interfata.o prelucrari.o
/usr/lib/my_lib.a
    gcc -o exemplu main.o interfata.o prelucrari.o
/usr/lib/my_lib.a
main.o : main.c
    gcc -c main.c
interfata.o : interfata.c
    gcc -c interfata.c
prelucrari.o : prelucrari.c
    gcc -o prelucrari.c

```

În cazul acestui fișier de descriere se pot observa patru intrări, fiecare intrare constând din două elemente distințe – o linie numită linie de dependențe și una sau mai multe linii de comenzi. Pentru acest exemplu țintele sunt: exemplu, main.o, interfata.o, prelucrari.o. Liniile de comenzi trebuie să înceapă cu un caracter tab. Într-un fișier makefile comentariile sunt delimitate de caracterul # și continuă până la sfârșitul liniei.

O altă caracteristică este flexibilitatea – într-un fișier de descriere se pot preciza mai multe programe executabile care să fie realizate. Astfel la fișierul de descriere anterior se pot adăuga următoarele linii:

```

exemplu_text : main_text.o prelucrari.o
    gcc -o exemplu_text main_text.o prelucrari.o

main_text.o : main_text.c
    gcc -c main_text.c

```

Vom avea două interfețe diferite pentru același program. Pentru make exemplu vom avea un program cu o interfață grafică, pe când pentru make exemplu\_text, vom avea aceeași aplicație de prelucrări de date dar care va opera din linia de comandă. Cel mai simplu mod de apelare make este fără nici un parametru:

```
$ make
```

În acest caz se va realiza prima țintă găsită în fișierul de descriere makefile, bineînțeles împreună cu toate dependențele sale.

O ultimă observație este faptul că putem avea ținte fără dependențe, iar multe dintre acestea nu sunt nici măcar fișiere. De exemplu multe fișiere de descriere conțin ținta clean destinată eliminării fișierelor temporare obținute în urma procesului de compilare.

```

clean :
    /bin/rm -f *.o

```

## 2. Macro (variabile)

În cazul aplicațiilor de dimensiuni mai mari și nu numai la realizarea fișierelor de descriere se poate observa repetarea unui anumit număr de elemente putându-se ajunge la dimensiuni relativ mari ale acestor fișiere. Pentru a preveni această situație se vor utiliza macrouri și sufixe.

Definițiile de macrouri din cadrul fișierelor de descriere vor arăta astfel:

```
nume = sir de caractere.
```

În referirea acestora se va face prin construcții de genul:

```
$ (nume) sau ${nume}
```

Reguli de sintaxă:

O definiție macro este o linie care conține semnul egal. Make asociază numele din stânga semnului egal cu secvența de caractere care urmează în partea dreaptă. Pentru ca make să facă deosebirea între o linie de comandă și un macro, înaintea acestuia nu este permis un caracter tab. De asemenea pentru a diferenția un macro de o linie de dependențe nu este permisă apariția caracterului ; înaintea semnului egal. Alt caracter cu semnificație specială este \ care indică faptul că linia curentă se continuă pe linia următoare, acest caracter este înlocuit de către make cu un spațiu. Toate spațiile, taburile și linie nouă sunt înlocuite cu un singur spațiu.

Prin convenție numele de macrouri se scriu cu caractere mari. În acestea pot apărea orice combinație de caractere mari mici dar se recomandă evitarea caracterelor speciale în special \ și >.

Pentru referirea unui macro se utilizează de obicei notația în paranteze sau în acolade precedată de semnul \$. Cele căror nume este alcătuit dintr-un singur caracter nu cer utilizarea în mod obligatoriu a unei paranteze. De exemplu considerând următorul macro:

```
A=cpp
```

Acesta poate fi referit astfel: \$A, \$(A), \${A}.

ACEste structuri sunt flexibile permitând utilizarea de macro-uri în noi definiții:

```
ABC = cpp
```

```
file = text.$(ABC)
```

Prin referirea acestui macro \$(file) vom avea text.cpp.

Unei definiții macro, care după semnul egal nu are nici un caracter, își va atribui sirul nul.

O altă caracteristică a acestui utilitar este faptul că nu trebuie ținut cont de ordinea în care aceste macro sunt definite, ușurând definirile imbricate. Ex:

SOURCES	= \${MY_SRC} \${SHARED_SRC}
MY_SRC	= analizeaza.c prelucreaza.c
SHARED_DIR	= /home/proiect/src
SHARED_SRC	= \${SHARED_DIR}/depend.c

Indiferent de ordinea definițiilor de mai sus \${SOURCES} va avea următoarea formă:

```
analyzeaza.c prelucreaza.c /home/proiect/src/depend.c
```

Orice definire de macro trebuie să aibă loc înainte de linia de dependențe în care acesta apare. Această libertate de declarare implică totuși o limitare: un macro nu poate defini într-o parte a fișierului și apoi redefinit deoarece make utilizează ultima definiție pentru toate referirile acestuia.

Pe lângă macrourile definite de către utilizatori există o serie predefinite, cele mai cunoscute sunt \${CC} utilizat pentru a defini compilatorul de C curent și \${LD} indică linker-ul utilizat.

Utilitatea utilizării unui macro pentru a indica compilatorul de C utilizat pentru proiectul curent se poate observa în mediile de dezvoltare volatile în care compilatoarele sunt frecvent schimbate. Ex:

```
CC = cc
HelloWorld.o : HelloWorld.c
    ${CC} -c HelloWorld.c
```

va avea același efect ca și:

```
HelloWorld.o : HelloWorld.c
    cc -c HelloWorld.c
```

În general fiecare macro care definește o comandă (Ex CC) necesită încă un macro care să definească opțiunile pentru comanda respectivă. Astfel pentru a preciza diverse opțiuni pentru compilatorul de C (CC) vom avea macroul CFLAGS, iar pentru LD vom avea LDFLAGS.

Un alt mod de definire de macro este din linia de comandă:

```
$ make refext DIR=/usr/proj
```

De asemenea trebuie să se ia în considerare variabilele intrepretorului de comenzi care fac parte din mediul de lucru, în momentul apelării make, acestea vor fi considerate ca fiind macro-uri din fișierul de descriere și pot fi utilizate ca atare. Dacă considerăm următoarea secvență de comenzi:

```
$ DIR = /home/student/proiect
$ export DIR
$ make refext
```

În acest context variabila DIR poate fi utilizată în cadrul fișierului de descriere:

```
SRC = ${DIR}/src
refext :
    cd ${DIR}/src
```

După cum se poate observa un macro poate fi definit în mai multe moduri: prin intermediul variabilelor mediului de lucru, din linia de comandă, în interiorul fișierului de descriere sau pot fi macrouri predefinite. Pentru un anumit macro pot apărea conflicte între definițiile provenite din diverse surse. În această situație se pune problema unei structuri de priorități care să fie urmărită în cazul unor astfel de conflicte. În ordinea priorităților vom avea:

- Definiții introduse din linia de comandă.
- Definițiile din fișierul de descriere.
- Variabilele de mediu din aferente interpretorului de comenzi.
- Definițiile interne (implicite).

Un alt element important ce apare în cadrul definițiilor de macro este substituirea de șirurile de caractere. Să presupunem că avem următoarele definiții:

```
SRCS = def.c cauta.c calc.c
ls ${SRCS:.c=.o}
rezultatul va fi: def.o cauta.o calc.o
```

Algoritmul utilizat pentru substituție este următorul: se evaluează expresia \${SRCS} apoi se caută șirul de caractere ce urmează în definiție după caracterul ‘:’ și este înlocuit cu șirul de caractere de după semnul ‘=’. Această facilitate este aplicabilă grupurilor de fișiere care diferă doar printr-un sufix. Asupra substituirilor șirurilor de caractere se aplică o serie de restricții – se pot face doar la sfârșitul unui macro sau înaintea unui spațiu.

### 3. Reguli de adăugare a sufivelor

În sistemele Linux(UNIX) fișierele sursă C au întotdeauna extensia .c, o cerință impusă de compilatorul de cc. Similar fișierele sursă Fortran au extensia .f cele pentru limbajul de asamblare au extensia .s, iar compilatoarele de C și Fortran își plasează modulele obiect în fișiere cu extensia .o.

Acest tip de convenții fac posibilă utilizarea unor reguli de adăugare a sufivelor pentru simplificarea fișierelor de descriere utilizate de make. Primul exemplu va avea următoarea formă:

```
CC = gcc
OBJS = main.o interfata.o prelucrari.o
LIB = /usr/lib/my_lib.a

exemplu : ${OBJS} ${LIB}
          ${CC} -o $@ ${OBJS} ${LIB}
```

unde prin \$@ se specifică ținta curentă pentru make. Un alt macro des utilizat este \$?, are rolul de a determina dependențele ce au fost actualizate în urma țintei curente.

Pentru a obține executabilul exemplu utilitarul make verifică mai întâi dacă main.o este “up to date”(actual). Realizează acest lucru verificând dacă sunt actuale toate fișierele care conform fișierului de descriere sunt folosite pentru compilarea lui main.o. Dacă descoperă de exemplu că fișierul main.c a fost modificat va apela compilatorul de C pentru acel fișier, dacă descoperă un fișier main.f mai recent decât main.o atunci va apela compilatorul de Fortran. Dacă regulile de adăugare a sufivelor nu sunt precizate, atunci make va utiliza regulile de adăugare implicite. Pentru exemplu precizat acestea sunt:

```
.SUFFIXES : .o .c .s
.C.O :
        $(CC) $(CFLAGS) -c $<
.S.O :
        $(AS) $(ASFLAGS) -c $@ $<
```

Linia ce conține .SUFFIXES are forma unei linii de precizare a dependențelor, dar apare o diferență. Sirurile conținute în această linie sunt sufixe care vor fi considerate de make ca fiind semnificative. Dacă o dependență se termină cu un astfel de sufix, și nu are nici o comandă definită în mod explicit de către utilizator, make va căuta o regulă ce poate fi aplicată dependenței respective. Să considerăm exemplul anterior. La construirea executabilului, make va urma pașii:

- Va căuta fiecare fișier dependență dat de expresiile \${OBJS} și \${LIB}, care va deveni la rândul lui țintă. Aceste operații sunt realizate pentru a se stabili care fișier trebuie recompilat înainte de a realiza executabilul final.
- Executabilul exemplu depinde printre alte fișiere și de prelucrari.o. Când make va căuta dependențele pentru prelucrari.o va verifica prima dată sunt definiții date de utilizator care prelucrari.o ca și țintă. Negăsind niciuna va nota ca sufixul semnificativ este .o și va căuta un alt fișier din directorul curent care să poată fi folosit pentru a construi fișierul prelucrari.o. Acest fișier trebuie:
  - să aibă același nume (în afară de sufix) cu prelucrari.o.
  - să aibă un sufix semnificant.
  - să poată fi folosit la construcția lui prelucrari.o pe baza unei reguli de adăugare a sufixelor existentă.

În cazul exemplului de mai sus toate regulele sunt satisfăcute de fișierul prelucrari.c. Regula care va fi folosită este:

```
.C.O :
$ (CC) $ (CFLAGS) -c $<
```

Unde \$< are un sens înrudit cu \$? dar este valabil doar în cazul regulelor de generare a sufixelor. Evaluează care dependență a determinat utilizarea acelei regule – și este fișierul prelucrari.c. Fișierul cu extensia .o poate fi considerat ținta acestei regule, iar fișierul cu extensia .o dependența.

- În continuare make va aplica regula de creare a fișierului prelucrari.o numai dacă:
  - numai sunt alte dependențe care trebuie să fie verificate înainte.
  - prelucrari.o este anterior lui prelucrari.c
- În final după ce s-a realizat acest proces pentru fiecare fișier din \${OBJS} și \${LIB}, make va executa comanda ld pentru a crea fișierul exemplu dacă în ierarhia de dependențe există cel puțin un fișier cu o dată de modificare mai recentă decât exemplu.

În lista de sufixe ce urmează după .SUFFIXES trebuie să fie câte un spațiu și trebuie să lipsească dintre cele două sufixe care încep o regulă de adăugare de sufixe.

CC	=	gcc
CFLAGS	=	-g
LD	=	\$ (CC)
LDFLAGS	=	
RM	=	rm
EXE	=	exemplu
SRCS	=	main.c interfata.c prelucrari.c
OBJS	=	\${SRCS:.c=.o}

```
# clear
.SUFFIXES:
# define
.SUFFIXES: .o .c

# regula pentru .c -> .o
.c.o :
    $(CC) $(CFLAGS) -c $<

# tinta implicita "all"
all : $(EXE)

$(EXE) : $(OBJS)
    $(LD) -o $@ $(OBJS)

$(OBJS) : exemplu.h

clean :
    -$(RM) -f $(EXE) $(OBJS)
```

Într-un fișier de descriere se pot preciza și comenzi ale sistemului de operare, iar make le va trata identic cu interpretorul de comenzi. Astfel fișierului de descriere va avea acces la toate caracteristicile interpretorului de comenzi ca de exemplu redirectarea ieșirii sau expandarea numelor de cale. Singura restricție este ca toată secvența de comenzi să fie într-o singură linie. Astfel secvența de comenzi:

```
cd tmp
rm *
```

nu va funcționa, dar puse pe o singură linie vom obține rezultatul dorit:

```
cd tmp; rm *
```

pentru secvențe mai mari de comenzi se poate utiliza caracterul '\' care permite comenzilor să aparțină teoretic unei singure linii.

Make, pe lângă țintele obișnuite(fișiere) mai poate avea și ținte simple, nume de conveniență, pentru diverse activități, numite ținte vide. Putem avea astfel de ținte în momentul în care se dorește utilizarea lui make ca și unealtă de instalare a unor programe:

```
INSTALLDIR = /usr/local/exemplu

install      : disp trac plot
              cp -f disp ${INSTALLDIR}
              cp -f trac ${INSTALLDIR}
              cp -f plot ${INSTALLDIR}
              cd ${INSTALLDIR}; \
              chmod 755 disp trac plot
```

Practic aceste ţinte vide sunt în afara lanțului normal de dependențe între fișiere și forțează într-un fel realizarea unei acțiuni. Dacă pentru fișierele reale verifică timpul ultimei modificări pentru aceste tipuri de ţinte se utilizează alte reguli:

- O ţintă vidă este tot timpul “out of date”, astfel comenziile asociate ei se execută tot timpul.
- O dependență vidă este tot timpul mai actuală decât ţinta ei provocând tot timpul reconstruirea ţintei.

Dacă se utilizează ca și dependență un nume vid, acesta trebuie să apară ca și ţintă altfel make se va opri cu eroare.

# Sistemul de fișiere - Management

Managementul fișierelor este considerat o parte a sistemului de operare. În majoritatea aplicațiilor, fișierul este elementul central. Cu excepția aplicațiilor de timp real și a altor aplicații specializate, intrarea acestora este un fișier iar ieșirea este salvată tot într-un fișier pentru o utilizare ulterioară.

Termenii utilizați în mod curent în discuțiile referitoare la fișiere sunt:

- câmp – dată elementară, conține o singură valoare fiind caracterizată prin lungime și tip;
- înregistrare – este o colecție de câmpuri înrudite ce pot fi tratate ca și un întreg;
- fișier – o colecție de înregistrări similare, este tratat ca o singură entitate de utilizatori și aplicații și este referit prin nume.

Este necesar ca aplicațiile și utilizatorii să poată să facă uz de fișiere. Operațiile tipice ce trebuie să poată fi efectuate asupra acestora sunt:

- obține toate înregistrările unui fișier;
- determină obținerea unei singure înregistrări din fișier;
- determină obținerea înregistrării care este "următoarea" sau "anterioară" într-o secvență logică oarecare după cea mai recentă înregistrare obținută;
- inserarea unei noi înregistrări într-un fișier, poate fi necesar ca înregistrarea să fie făcută pe o poziție particulară;
- ștergerea unei înregistrări existente;
- obținerea unei înregistrări, actualizarea acesteia și rescrierea ei înapoi în fișier;
- obține un număr de înregistrări;

Un sistem de gestionare a fișierelor este un set de rutine ce oferă servicii utilizatorilor și aplicațiilor. Uzual, singurul mod în care un utilizator sau o aplicație poate obține accesul la un fișier este prin intermediul sistemului de gestionare a fișierelor, ceea ce îl degrevează pe programator de sarcina de a dezvolta pentru fiecare aplicație în parte rutine specializate de acces la fișiere.

Un sistem de gestionare a fișierelor trebuie să urmărească anumite obiective:

- să respecte constrângerile de gestionare a datelor și cerințele utilizatorilor;
- să garanteze validitatea datelor din fișiere;
- să optimizeze performanța din ambele puncte de vedere: sistem și utilizator;
- să ofere suport de I/E pentru o gamă cât mai largă de dispozitive de stocare;
- să minimizeze sau să eliminate potențialele pierderi sau distrugeri de date;
- să ofere o interfață standardizată pentru rutinele de I/E;
- să ofere suport de I/E pentru mai mulți utilizatori.

Realizarea obiectivelor anterior propuse depinde de diferențele aplicații ce trebuie rulate și de mediul în care va rula sistemul de calcul. Un sistem interactiv de uz general trebuie să respecte un set minimal de constrângeri:

- fiecare utilizator trebuie să fie capabil să creeze, să șteargă și să modifice fișiere;
- fiecare utilizator trebuie să poată avea acces controlat la fișierele altui utilizator;
- fiecare utilizator trebuie să poată controla tipurile de accese permise la fișierele sale;
- fiecare utilizator trebuie să fie capabil să își restructureze fișierele conform nevoilor sale;
- fiecare utilizator trebuie să poată să își mute datele între fișiere;
- fiecare utilizator trebuie să poată să își protejeze și recupereze datele în caz de distrugere.

- fiecare utilizator trebuie să își poată accesa fișierele printr-un nume simbolic.

Organizarea tipică a unui astfel de sistem este următoarea:

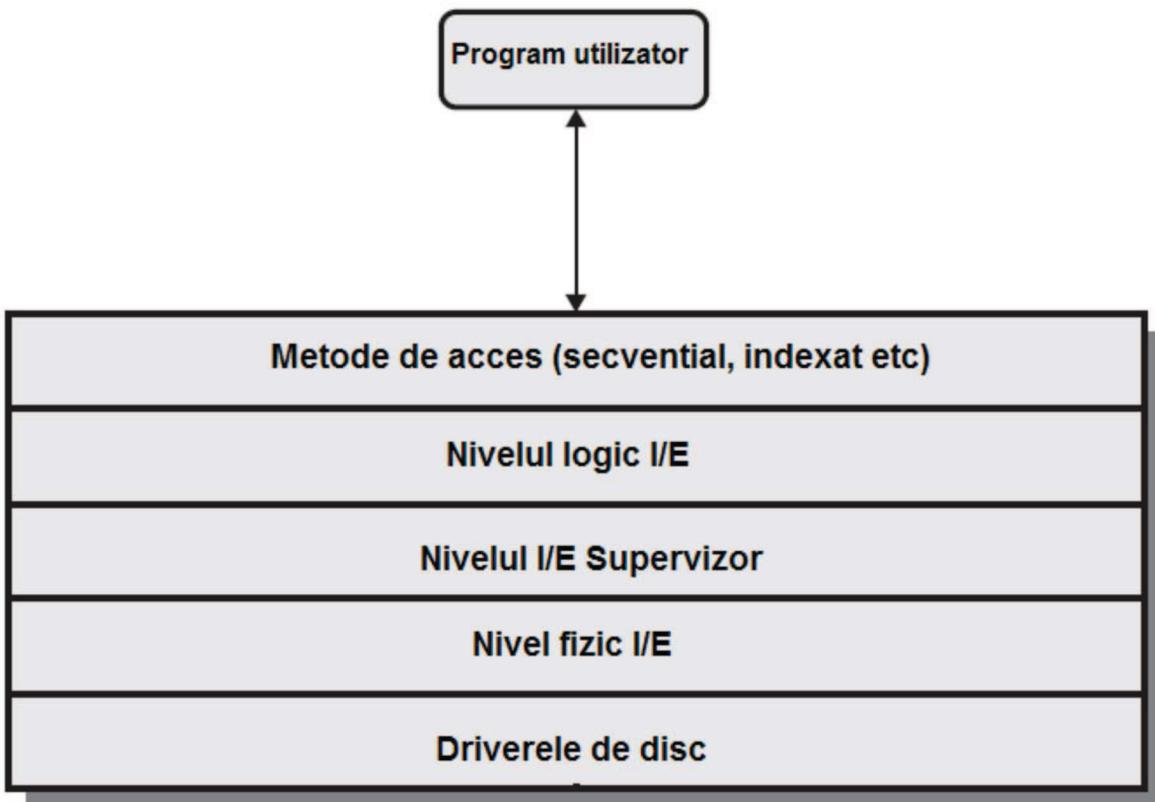


Figura 1 Arhitectura sistemului de fișiere

Pe cel mai de jos nivel într-un sistem de gestionare a fișierelor se află drivele de dispozitiv. Acestea comunică direct cu perifericele și sunt responsabile cu pornirea operațiilor de I/E și de încheierea acestora. Deseori, drivele de dispozitiv sunt considerate ca făcând parte din sistemul de operare.

Prima interfață între mediul înconjurător și sistemul de calcul este reprezentată de sistemul de fișiere elementar sau nivelul de I/E fizic. Acest nivel este responsabil pentru schimburile de date ce au loc cu discurile sau alte medii de stocare, în consecință tot în sarcina acestui nivel cade și plasarea blocurilor de date pe al doilea mediu de stocare și totodată aducerea blocurilor de date în memorie în vederea utilizării lor ulterioare. Nivelul fizic de I/E este deseori considerat ca făcând parte și el din sistemul de operare.

Nivelul de I/E elementar supervisor este responsabil pentru inițierea și terminarea tuturor operațiilor de I/E pentru fișiere. La acest nivel structurile de control au rolul de a gestiona dispozitive de I/E, programarea operațiilor și starea fișierelor. Acest nivel mai este responsabil și cu selecția dispozitivului cu care va fi realizată o operație de I/E. Si acest nivel este considerat ca făcând parte din sistemul de operare.

Nivelul logic de I/E permite utilizatorilor și aplicațiilor să acceseze înregistrări. După cum s-a văzut, la nivelul de jos se află sistemul de fișiere elementar care gestionează blocuri de date, iar nivelul logic al operațiilor de I/E gestionează înregistrările fișierelor. Acest nivel oferă înregistrărilor de scop general capabilități de I/E și gestionează informațiile elementare despre fișiere.

Nivelul sistemului de fișiere cel mai apropiat de utilizator în mod curent este numit metodă de acces. Aceasta oferă o interfață standard între aplicații și sistemul de fișiere precum și dispozitivele ce

conțin datele. Metodele de acces diferite reflectă structuri de fișiere diferite și moduri diferite de accesare și procesare a datelor.

Sistemul de gestionare a fișierelor este responsabil cu îndeplinirea unei serii de funcții ce vor fi discutate în continuare. Utilizatorii și aplicațiile interacționează cu sistemul de fișiere prin intermediul comenziilor pentru crearea și ștergerea fișierelor și pentru realizarea de operații asupra acestora. Înainte de realizarea oricărei operații, sistemul de fișiere trebuie să identifice și să localizeze fișierele selectate. Pentru aceasta este nevoie de utilizarea unui catalog (director) care să servească la descrierea locațiilor tuturor fișierelor și a atributelor acestora. În plus, în cazul sistemelor cu mai mulți utilizatori este necesar să fie îmbunătățit controlul acceselor utilizatorilor la fișiere. Acolo unde utilizatorii și aplicațiile lucrează cu înregistrări, nivelul de I/E lucrează pe bază de blocuri. Astfel, înregistrările unui fișier trebuie "blocate" pentru ieșire și "deblocate" pentru intrare. Pentru a putea efectua această "blocare" de I/E sunt necesare mai multe funcții. Pentru aceasta este necesar ca mediul de stocare să fie gestionat implicând alocarea de blocuri libere fișierelor și gestionarea spațiului liber pentru a cunoaște blocurile disponibile a fi alocate noilor fișiere sau pentru mărirea celor existente. În plus cererile de I/E pentru blocuri individuale trebuie programate.

In continuare vom folosi termenul de organizarea fișierelor care se referă la structura logică a înregistrărilor și este determinată de organizarea acestora. În alegerea modului de organizare a fișierelor se urmăresc câteva criterii:

- acces rapid;
- actualizare rapidă;
- economie de spațiu;
- întreținere simplă;
- siguranță;

Prioritatea acestor criterii este relativă și depinde de aplicațiile la care vor fi utilizate fișierele. Numărul de alternative de organizare a fișierelor care au fost implementate sau doar propuse este extrem de mare. Dintre acestea se vor aminti doar cinci mai reprezentative:

- aglomerare;
- fișiere secvențiale;
- fișiere secvențiale indexate;
- fișiere indexate;
- fișiere cu accesare directă;

Cea mai simplă formă de organizare a fișierelor este aşa-numita aglomerare. Datele sunt stocate în ordinea în care sosesc. Scopul acestei forme de organizare este să acumuleze mase mari de date și să le stocheze. Înregistrările pot avea câmpuri diferite sau similare, în ordine diferite. Astfel, fiecare câmp trebuie să se autodescrie inclusiv numele lui și valoarea. Lungimea lui trebuie indicată în mod implicit de către delimitator, inclusiv în mod explicit ca și subcâmp sau cunoscută implicit pentru fiecare câmp de date. Sistemul de fișiere nu este structurat, accesarea unei înregistrări se face prin căutare exhaustivă. Acest tip de structurare se poate folosi în situația în care datele sunt colectate și stocate în mod aleator sau când datele nu sunt ușor de organizat. În această situație spațiul este bine utilizat când datele înregistrate variază în dimensiune și structură și este ideal pentru căutări exhaustive.

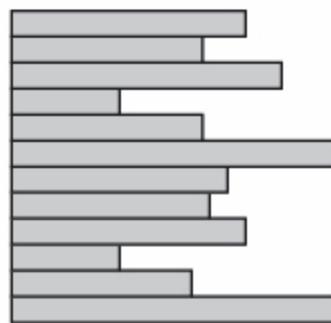


Figura 2 Organizarea de tip aglomerare a fișierelor

Cea mai întâlnită formă de organizare a fișierelor este cea secvențială. În cazul acestor tipuri de fișiere se utilizează pentru înregistrări un format fix. Toate înregistrările au aceeași lungime constând din același număr de câmpuri de lungime fixă într-o anumită ordine. Din cauză că lungimea și poziția câmpului sunt cunoscute, numai valorile înregistrărilor trebuie înregisterate. Numele câmpului și lungimea sunt atributele structurii de fișiere. Un câmp particular, de obicei primul câmp al înregistrării, este referit ca și câmpul cheie. Câmpul cheie identifică în mod unic înregistrarea astfel încât valori cheie pentru înregistrări diferite sunt tot timpul diferite. În continuare înregistrările sunt stocate într-o anumita ordine: alfabetică pentru cheile text și numerică pentru cele numerice. Fișierele secvențiale sunt de obicei utilizate de mai multe aplicații și sunt în general optimizate pentru astfel de aplicații dacă acestea trebuie să proceseze toate înregistrările. Pentru aplicațiile care implică căutarea numai a unor anumite înregistrări, performanța lasă mult de dorit. Tipic, un fișier secvențial este stocat într-o ordonare secvențială simplă a înregistrărilor pe blocuri, astfel încât organizarea fizică pe disc este identică cu organizarea logică a fișierului. În acest caz procedura standard este de a plasa noile înregistrări într-un nou fișier numit fișier de tranzacții. Periodic se realizează o actualizare a fișierului original de pe disc cu fișierul de tranzacție pentru a produce un nou fișier cu secvența corectă de chei.

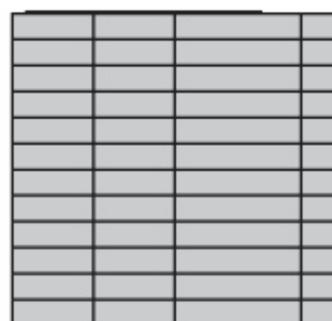


Figura 3. Organizarea secvențială a fișierelor

O abordare care să înlăture dezavantajele abordării secvențiale este cea a fișierelor secvențiale indexate. În acest caz se menține caracteristica cheie a fișierelor organizate secvențial: înregistrările sunt organizate într-o secvență bazată pe câmpuri cheie. Au fost adăugate două caracteristici noi: un index la fișiere pentru a se putea face căutări cu acces aleator, și un fișier de depășire. În cea mai simplă structură secvențială indexată se utilizează un singur nivel de indexare. Fiecare înregistrare în fișierul indexat constă din două câmpuri:

- un câmp cheie care este identic cu câmpul cheie din fișierul principal;
- o referință în fișierul principal;

Pentru a găsi un anumit câmp se caută în index pentru a găsi cea mai mare cheie a cărei valoare este egală sau precede cheia cu valoarea dorită. Căutarea este continuată în fișierul principal la locația indicată de referință.

Pentru a putea observa diferențele dintre cele două abordări se prezintă în continuare un scurt exemplu. Se consideră un fișier secvențial cu un milion de înregistrări. O căutare pentru o anumită cheie necesită în medie o jumătate de milion de înregistrări accesate. Să presupunem că este construit un index care conține 1000 de intrări. În această situație este nevoie în medie de 500 de accese la fișierul de index urmate de alte 500 de accese în fișierul principal. Astfel, căutarea medie a fost redusă de la 500.000 la 1000 de căutări.

O altă problemă ce trebuie menționată în cazul sistemului de fișiere secvențial indexate este adăugarea de noi înregistrări. Această operație se realizează în următoarea manieră: fiecare înregistrare din fișierul principal conține un câmp suplimentar care nu este vizibil aplicațiilor. Când o nouă înregistrare este adăugată unui fișier, ea este adăugată fișierului de depășire. Înregistrarea din fișierul principal care este imediat precedentă noii înregistrări este modificată să conțină o referință către noua înregistrare din fișierul de depășire. La fel ca și în cazul fișierelor secvențiale, fișierele secvențiale indexate sunt ocazional actualizate cu fișierul de depășire pentru a obține fișierul actualizat.

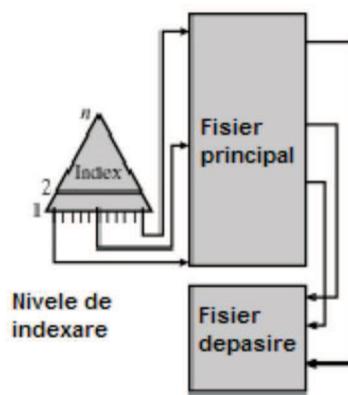


Figura 4. Organizarea secvențial indexată a fișierelor

Pentru a crește eficiența acceselor se pot utiliza mai multe nivele de indexare. Astfel, cel mai de jos nivel de indexare este tratat ca și un fișier secvențial, iar nivelul superior ca și un index pentru acest fișier.

Fișierele indexate secvențiale păstrează o limitare a fișierelor secvențiale: procesarea efectivă este limitată pentru că se bazează doar pe un singur câmp al fișierului. Când este necesară căutarea unei înregistrări pe baza altor attribute decât câmpul cheie, atunci amândouă formele sunt neadecvate. Pentru a obține flexibilitate este necesară utilizarea unor indexări multiple, adică pentru fiecare tip de câmp care poate face obiectul unei căutări. În general la fișierele indexate conceptul de secvențialitate este abandonat, iar înregistrările sunt accesate pe baza indexărilor. Rezultatul este că nu mai există restricții la plasarea înregistrărilor atâtă vreme cât înregistrarea respectivă are o referință la ea, deci înregistrările pot avea o lungime variabilă. Se utilizează două tipuri de indexări:

- o indexare exhaustivă ce conține câte o intrare pentru fiecare înregistrare din fișierul principal;
- o indexare parțială care conține intrări numai pentru înregistrările pentru care câmpul de interes există.

În figuri 12-13.5 sunt prezentate cele 2 situații.

Fișierele cu accesare directă utilizează capacitatea de accesare directă a oricărui bloc cu adresa cunoscută de pe disc. În cazurile anterioare era nevoie de o cheie pentru fiecare înregistrare în parte. Fișierele cu acces direct fac uz de așa-numitul "hashing" a valorilor cheie. Fișierele cu acces direct sunt utilizate atunci când este nevoie de o accesare rapidă a înregistrărilor.

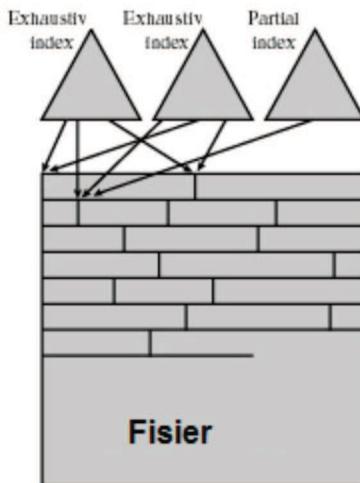


Figura 5 Organizarea indexată a fișierelor

Directoarele sunt asociate cu orice sistem de management a fișierelor și cu colecțiile de fișiere. Directoarele conțin informații despre fișiere incluzând atributele, locația și proprietarul. Directorul este el însuși un fișier al cărui proprietar este sistemul de operare și este accesibil diverselor rutine ale sistemului de gestionare a fișierelor. Deși unele informații din directoare sunt disponibile utilizatorilor, aceasta se realizează indirect prin intermediul rutinelor sistem, astfel încât utilizatorii nu pot accesa direct directoarele nici numai pentru citire. Din punctul de vedere al utilizatorului, directorul realizează o mapare între numele fișierelor, cunoscute utilizatorilor și aplicațiilor, și fișierele propriu-zise. Astfel, fiecare intrare fișier din director conține și numele fișierului. O categorie importantă de informații referitoare la un fișier este spațiul ocupat cu caracteristicile sale: locație și dimensiune. În sistemele multiutilizator se pot preciza informații suplimentare referitoare la controlul accesului la fișier. Uzual un utilizator este proprietarul fișierului și poate da altor utilizatori anumite privilegii.

Unele informații pot fi stocate în prima înregistrare asociată cu fișierul: aceasta reduce spațiul necesar pentru director, restul elementelor trebuie să fie în director: numele, adresa, dimensiunea și organizarea. Cea mai simplă formă a unui director este o listă de intrări, câte una pentru fiecare fișier. Această structură poate fi reprezentată printr-un director secvențial cu numele fișierului servind ca și cheie. Această tehnică a fost utilizată în primele sisteme de gestiune a fișierelor. Pentru a înțelege cerințele pentru structura unui director este bine să fie prezentate tipurile de operații care trebuie să poată fi executate asupra sa:

- căutare: când un utilizator sau o aplicație referă un fișier, intrarea fișierului din directorul respectiv trebuie căutată.
- creare de fișier: când un nou fișier este creat trebuie adăugată o nouă intrare directorului.
- ștergere de fișier: când un fișier este șters trebuie înălțată intrarea respectivă din director.
- listarea directorului.

Această structură de organizare a directorului nu oferă suport utilizatorului în organizarea fișierelor și îl forțează să nu folosească același nume pentru două fișiere diferite. Un prim pas în rezolvarea problemelor prezentate anterior ar fi utilizarea unei scheme pe două nivele. În acest caz există un director pentru fiecare utilizator și un director principal care le cuprinde pe toate. Directorul principal are câte o intrare pentru fiecare director utilizator oferind de asemenea și adresa și informațiile de acces pentru respectivele directoare. Fiecare director utilizator este o simplă listă cu fișierele utilizatorului respectiv. Datorită acestui aranjament, numele de fișier trebuie să fie unice pentru un utilizator, deci în continuare nu se oferă suport pentru structurarea colecțiilor mari de fișiere. O soluție mult mai flexibilă și general adoptată este structura ierarhică sau structura arborescentă. La fel ca și la modelul anterior prezentat avem un director principal numit și director

rădăcina care conține un număr de directoare utilizator. Fiecare director utilizator poate conține mai multe subdirectoare și fișiere ca și intrări. O reprezentare schematică a acestui model este următoarea:

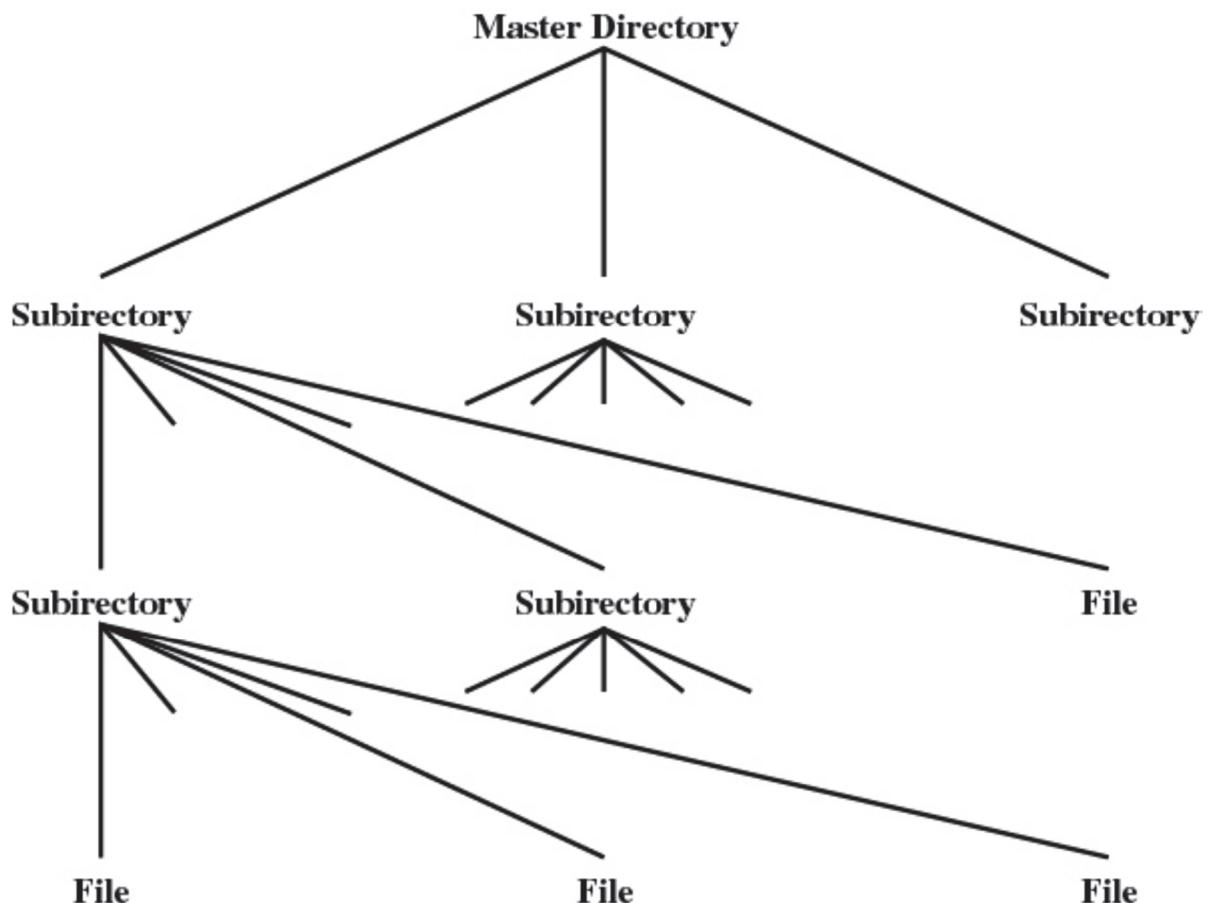


Figura 6 Structura arborescentă a directoarelor

Utilizatorii trebuie să poată referi fișierele prin intermediul unor nume simbolice. Deci fiecare fișier din sistem trebuie să aibă un nume unic pentru a nu apărea ambiguități la referirea fișierelor. Datorită structurii arborescente orice fișier din sistem poate fi localizat utilizând o cale de la rădăcină sau directorul principal, mergând pe diferitele ramuri până când se atinge fișierul dorit. Seriile de nume de directoare acumulate pe parcurs culminând cu numele directorului propriu-zis poartă numele de nume de cale (pathname). Din cauză că numele de cale sunt lungi lucrul cu ele este destul de dificil. De obicei un utilizator sau un proces este asociat cu directorul curent care poartă numele de director de lucru. Fișierele pot fi referite relativ la directorul curent sau de lucru. Când un utilizator intră în sistem directorul de lucru este directorul asupra căruia are drepturi de proprietate.

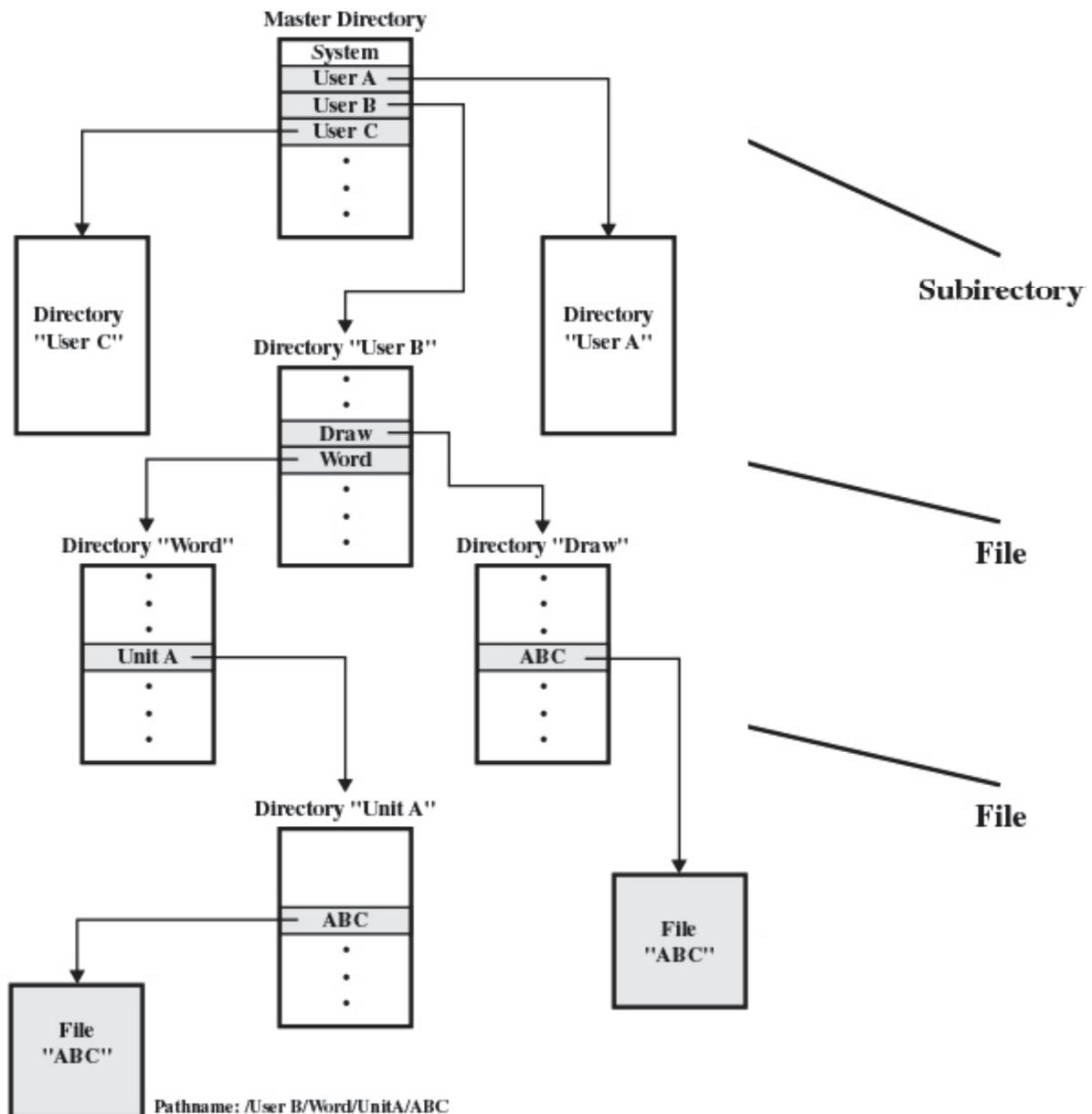


Figura 7 Exemplu de structură arborescentă

In sistemele cu mai mulți utilizatori există necesitatea ca anumite fișiere să poată fi făcute disponibile mai multor utilizatori. In aceasta situație se pot ridica două probleme:

- drepturile de acces;
- gestionarea acceselor simultane;

Sistemul de fișiere are sarcina să ofere unelte flexibile care să permită distribuirea fișierelor între utilizatori, de asemenea sistemul de fișiere trebuie să ofere utilizatorilor un număr de opțiuni prin care să controleze accesul altor utilizatori la fișiere. Uzual utilizatorii sau grupurile de utilizatori primesc anumite drepturi de acces la fișier dintre care precizăm:

- fără nici un drept – utilizatorii nu pot nici măcar afla de existența fișierului, cu atât mai puțin să îl acceseze. Pentru a înăspri restricția utilizatorilor nu le este permis accesul în directorul care conține fișierul;
- utilizatorul poate vedea că fișierul există și poate determina cine este proprietarul;
- execuție – utilizatorul poate încărca și executa un program dar nu îl poate copia;

- citire – utilizatorul poate citi fișierul în orice scop, inclusiv copierea și execuția;
- adăugare – utilizatorul poate citi fișierul, poate adăuga date, deseori numai la sfârșit, dar nu poate modifica sau șterge date din fișier.
- actualizare – utilizatorul poate modifica, șterge și adăuga date fișierului. Aceasta implică în mod normal inițial scrierea fișierului, rescrierea lui completă sau parțială și ștergerea totală sau în parte a acestuia;
- modificarea protecției – utilizatorul poate schimba drepturile de acces oferite celorlalți utilizatori. Uzual acest lucru îl poate face doar proprietarul fișierului;
- ștergere – utilizatorul poate șterge fișierul din sistemul de fișiere;

Se poate considera că aceste drepturi constituie o hierarhie, fiecare dintre aceste drepturi implicându-le pe toate cele care îl preced. Un utilizator este desemnat ca fiind proprietarul fișierului, de obicei este utilizatorul care a creat fișierul inițial. Proprietarul are toate drepturile prezentate anterior și poate oferi și altor utilizatori diferite drepturi de acces la respectivul fișier. Accesul se poate realiza pe diverse clase de utilizatori:

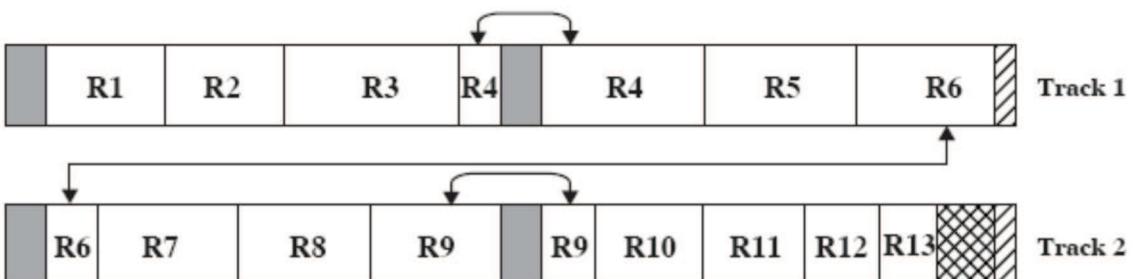
- utilizatori specifici: utilizatori individuali care sunt desemnați prin ID-ul lor;
- grupuri de utilizatori: set de utilizatori care nu sunt definiți individual;
- tuturor: tuturor utilizatorilor care au acces la respectivul sistem. Aceste fișiere sunt publice.

Când accesul este oferit pentru adăugare sau actualizare, atunci sistemul de operare sau sistemul de gestionare a fișierelor trebuie să asigure disciplina. O primă abordare a problemei ar fi să se permită utilizatorului să blocheze accesul altor utilizatori la fișier pe parcursul actualizării lui. O abordare mai elegantă ar fi să se blocheze doar înregistrările din fișier care sunt modificate. Tot în acest context apar și problemele de excludere mutuală sau de blocare în aşteptare.

Pentru realizarea operațiilor de I/E este necesar ca înregistrările să fie organizate pe blocuri. Se pune problema modului de organizare a informației pe blocuri. Se pot considera mai multe situații:

- blocuri cu înregistrări de lungime fixă;
- blocuri cu dimensiune variabilă extinse;
- blocuri cu dimensiune variabilă neextinse.

Blocuri de dimensiune fixă sunt alcătuite dintr-un număr fix de înregistrări care la rândul lor au lungimea fixă. În acest caz poate rămâne spațiu neutilizat la sfârșitul fiecărui fișier. Aceste blocuri sunt de obicei utilizate pentru fișierele secvențiale, la fel cum este ilustrat și în figura următoare:

**Blocuri fixe****Blocuri de lungime variabila extinse****Blocuri de lungime variabila**

Data	Pierderi datorate potrivirii inregistrarii la marimea blocului
Pierderi datorate hardware-ului	-
Pierderi datorate potrivirii blocului la track	Pierderi datorate dimensiunii blocului in raport cu inregistrarea fixa

Figura 8 Organizarea pe blocuri

In situația blocurilor cu dimensiune variabilă extinse nu vom avea spațiu nefolosit ca și în situația anterioară. Astfel, unele înregistrări trebuie să fie peste două blocuri, iar al doilea bloc va fi indicat printr-un pointer. Acest tip de organizare este extrem de eficient din punctul de vedere al ocupării spațiului, dar apar probleme în ceea ce privește gestionarea fișierelor. În această situație se folosesc blocuri de dimensiune variabilă dar nu este permisă extinderea înregistrărilor peste mai multe blocuri, ceea ce poate conduce la apariția de spațiu neocupat la sfârșitul blocurilor.

Pe mediul de stocare secundar, un fișier este constituit dintr-o colecție de blocuri. Sistemul de operare sau sistemul de gestionare al fișierelor este responsabil cu alocarea blocurilor fișierelor. În această situație, se pot ridica două probleme de gestionare a spațiului de stocare:

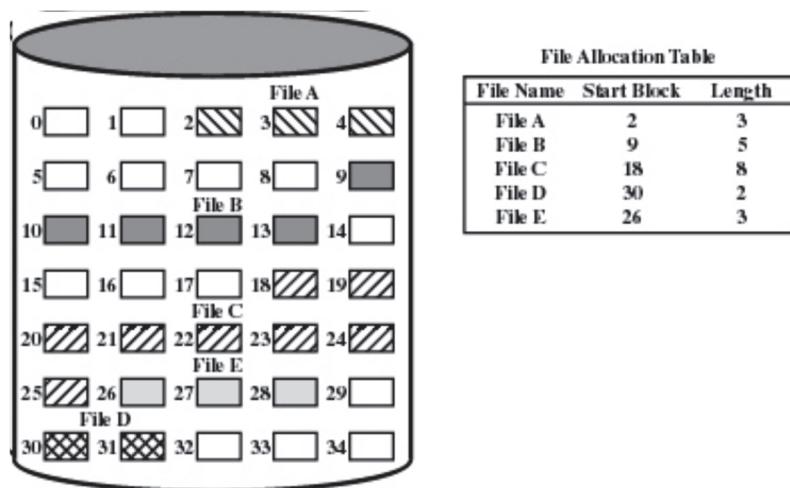
- spațiul de pe mediul de stocare secundar trebuie alocat fișierelor;
- este necesar să se țină evidența spațiului ce poate fi utilizat pentru alocare.

Alocarea fișierelor implică mai multe probleme. O primă problemă este cât spațiu să se aloce la crearea fișierului. O politică de alocare a spațiului cere ca spațiul maxim ce va fi ocupat de fișier să fie declarat la crearea fișierului. Dar pentru multe aplicații este dificil, dacă nu imposibil de estimat dimensiunea maximă a fișierului. În aceste situații, utilizatorii și aplicațiile vor tinde să supraestimeze dimensiunea necesară pentru fișier ceea ce va conduce în final la o risipă de spațiu. De aceea ar fi avantajos să se utilizeze o alocare dinamică a spațiului în care fișierul primește numai când are nevoie.

O altă problemă ce trebuie luată în considerare este dimensiunea cuantei cu care se face alocarea de spațiu. În principiu sunt două alternative:

- porțiuni contigue mari și variabile: aceasta va conduce la creșterea performanței. Dimensiunea variabilă elimină risipa, iar tabelele de alocare a fișierelor sunt mici.
- blocuri: porțiuni mici de dimensiune fixă ce oferă o mai mare flexibilitate. Acestea pot solicita tabele largi sau structuri complexe pentru alocarea lor. Contiguitatea a fost abandonată, iar blocurile sunt alocate după nevoie.

În primul caz, prealocarea fișierului se face în grupuri de blocuri contigue. În această situație este nevoie numai de o referință către primul bloc și de numărul de blocuri alocate, iar tabela de alocare are nevoie doar de o singură intrare pentru fiecare fișier. Acest tip de alocare este ideală pentru fișierele secvențiale. Singura problemă este că poate apărea o fragmentare externă făcând dificilă obținerea spațiului pentru alocarea blocurilor de dimensiuni mari. De aceea este necesar să se efectueze periodic o operațiune de compactare pentru a elibera spațiu suplimentar.



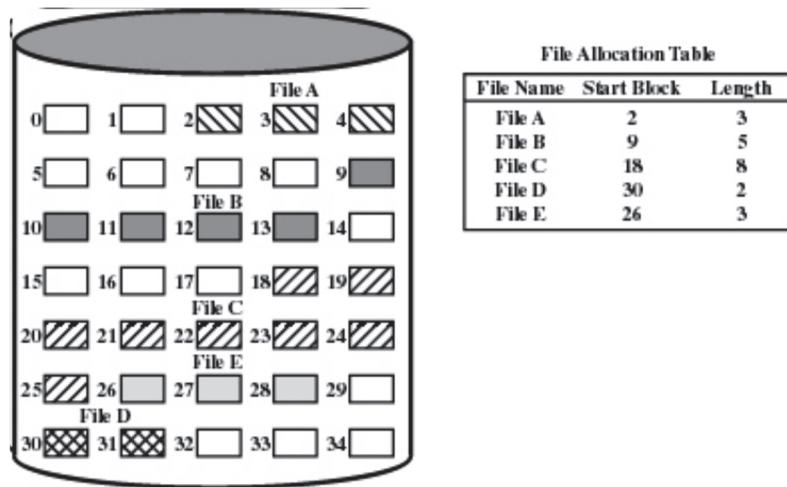
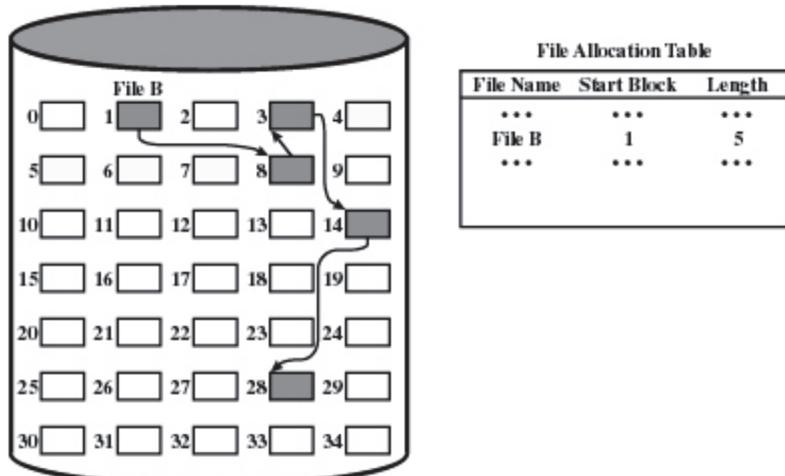


Figura 9 Alocarea contiguă a fișierelor înainte de compactare și după

La cealaltă extremă se află alocarea înlanțuită. În acest caz alocarea se face la nivel de blocuri individuale. Fiecare bloc conține o referință către următorul bloc din lanț. În această situație este nevoie doar de o intrare pentru fiecare fișier în tabela de alocare a fișierelor. În acest caz nu se pune problema fragmentării. Acest tip de organizare este ideal pentru fișierele secvențiale ce se vor procesa secvențial. Problema ce poate apărea este necesitatea de a face accese în diferite zone ale discului. Pentru a corecta această problemă este necesar să se facă periodic o compactare a fișierului.



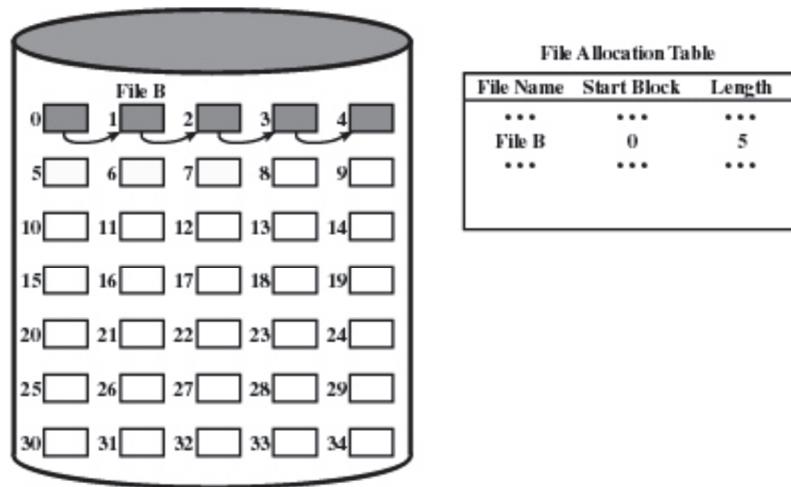
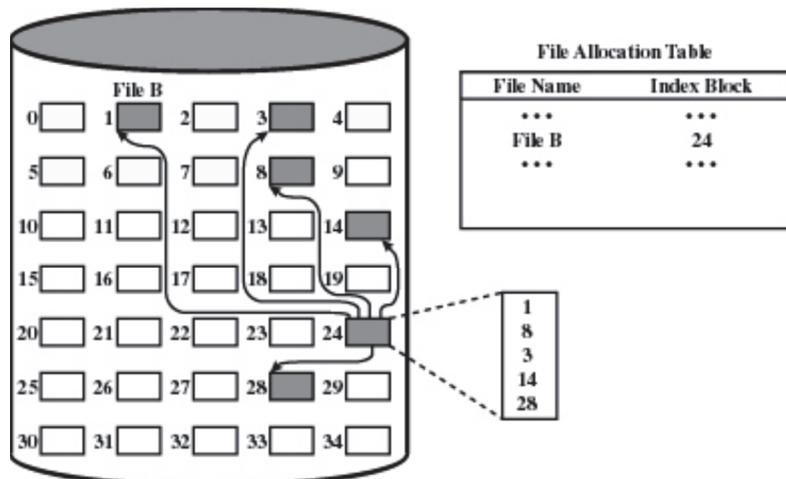


Figura 10 Alocarea înlanțuită a fișierelor înainte de compactare și după

In cazul alocării indexate, tabela de alocare a fișierelor conține un nivel de indexare separat pentru fiecare fișier: indexul are câte o intrare pentru fiecare porțiune de fișier alocată. Uzual indexările din fișier nu sunt stocate ca și părți ale tabelei de alocare, ci sunt memorate în blocuri separate și tabela de alocare conține o referință către acel bloc. Alocarea se poate face pe bază de blocuri cu dimensiune fixă sau pe porțiuni de dimensiune variabilă. In primul caz se elimină fragmentarea iar în al doilea îmbunătățește localizarea. In ambele situații este nevoie să se efectueze compactarea la anumite intervale de timp.



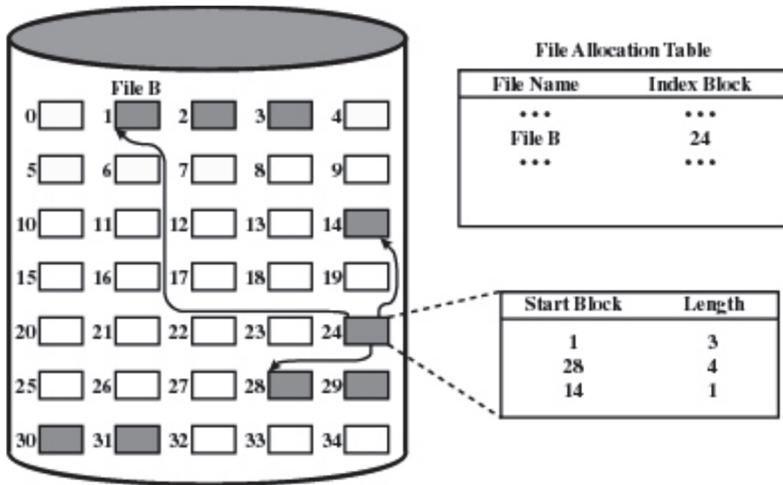


Figura 11 Alocarea indexată a fișierelor folosind blocuri fixe sau variabile

In cazul sistemului de operare UNIX toate fișierele sunt tratate de nucleu ca și "streams". Orice organizare logică a structurii este dependentă de aplicație. Sistemul de operare gestionează doar structura fizică a fișierelor. Se pot identifica patru tipuri de fișiere diferite:

- obișnuite – fișiere ce conțin date introduse de utilizator, de diverse aplicații sau de sistemul de operare.
- directoare – conțin o listă cu numele de fișiere și câte o referință pentru fiecare fișier în parte. Directoarele sunt organizate ierarhic.
- speciale – utilizate pentru accesarea perifericelor (terminale, imprimante). Fiecare dispozitiv de I/E are asociat câte un fișier special.
- pipe – fișiere speciale folosite la comunicația între fișiere.

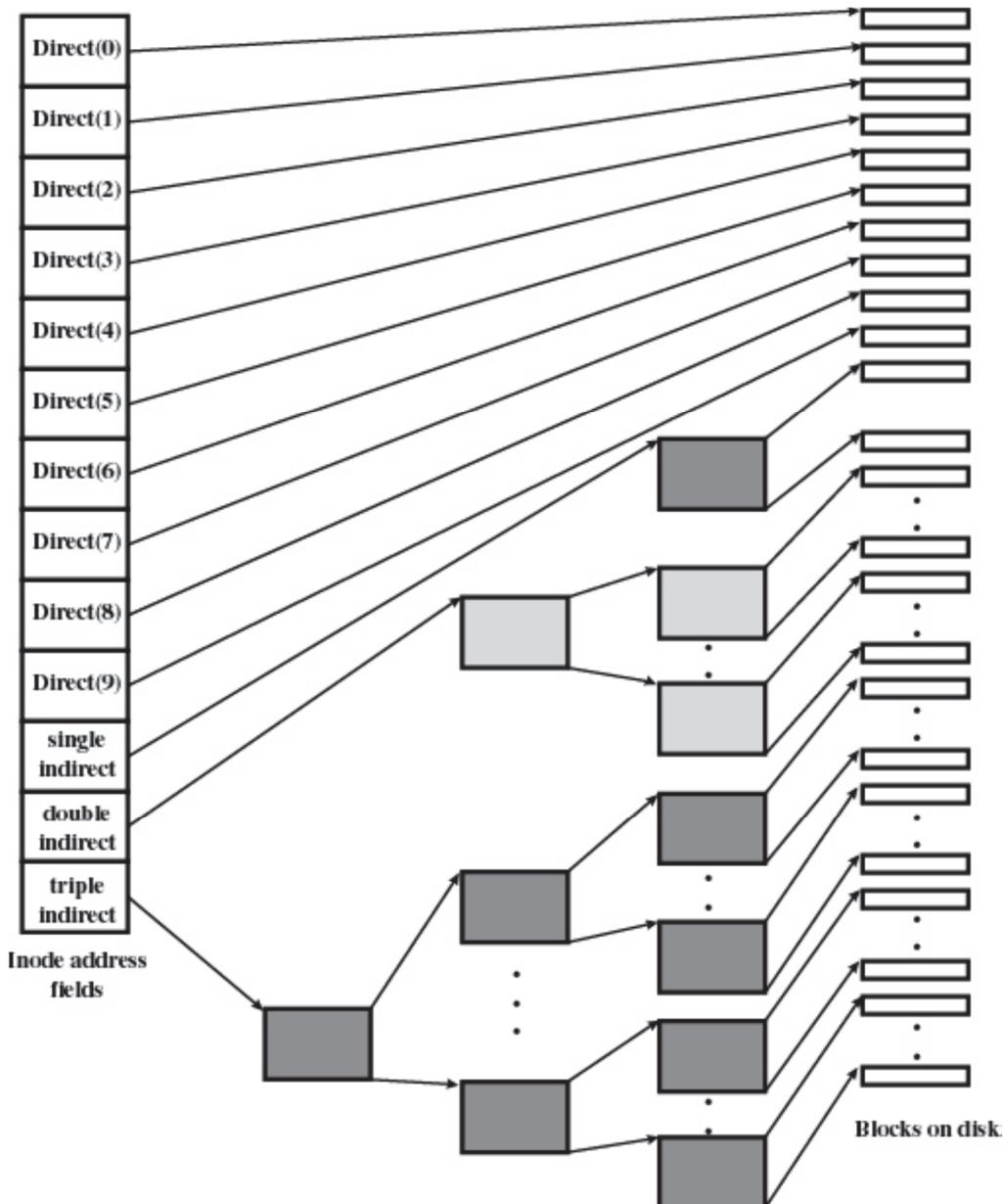


Figura 12 Sistemul de directoare UNIX

Sistemele de operare Windows suportă mai multe sisteme de fișiere, inclusiv cele de tip tabelă de alocare a fișierelor (FAT) folosite la sistemele de operare Windows 95, MS-DOS și OS/2, dar folosește și un sistem special – sistem de fișiere NT (NTFS), care este proiectat să satisfacă cerințele ale stațiilor de lucru și ale serverelor. Elementele cheie ale acestui sistem de fișiere sunt:

- discuri mari, fișiere mari: suportă discuri și fișiere de dimensiuni mari, mult mai eficient decât alte sisteme de operare;
- fluxuri de date multiple: ușor conținutul fișierelor este tratat ca și flux de octeți. În NTFS există posibilitatea de a defini mai multe fluxuri de date pentru un singur fișier;
- facilitate de indexare generală: NTFS associază o colecție de atribută cu fiecare fișier.
- Fișierele sunt descrise în cadrul sistemului de gestiune a fișierelor cu ajutorul unei baze de date relaționale, astfel fișierele pot fi indexate după fiecare atribut.

# Fișiere – apeluri sistem

## 1. Sistemul de fișiere Linux – considerente generale

Sistemul de fișiere din UNIX este caracterizat prin:

- o structură ierarhică
- tratare consistentă a fișierelor de date
- protejarea fișierelor

În cazul sistemului de fișiere din Linux se urmăresc aceleași principii de bază ca și în cazul UNIX, adică se încearcă ca intrarea și ieșirea pentru diverse dispozitive ca discuri, terminale, imprimante să se realizeze la fel ca și lucrul cu fișiere obișnuite.

Fiecare fișier este reprezentat de o structură numită inode (Fig. 1). Fiecare inode conține descrierea fișierului: tipul fișierului, drepturile de acces, proprietarul, informații referitoare la data, dimensiune, referințe către blocurile de date. Adresele blocurilor de date alocate fișierului sunt de asemenea stocate în cadrul inode-ului. Când un utilizator solicită o operație de intrare/ieșire asupra fișierului, kernel-ul sistemului de operare convertește indexul curent într-un număr de bloc și utilizează acest număr ca și index în tabela adreselor de bloc și scrie sau citește un bloc.

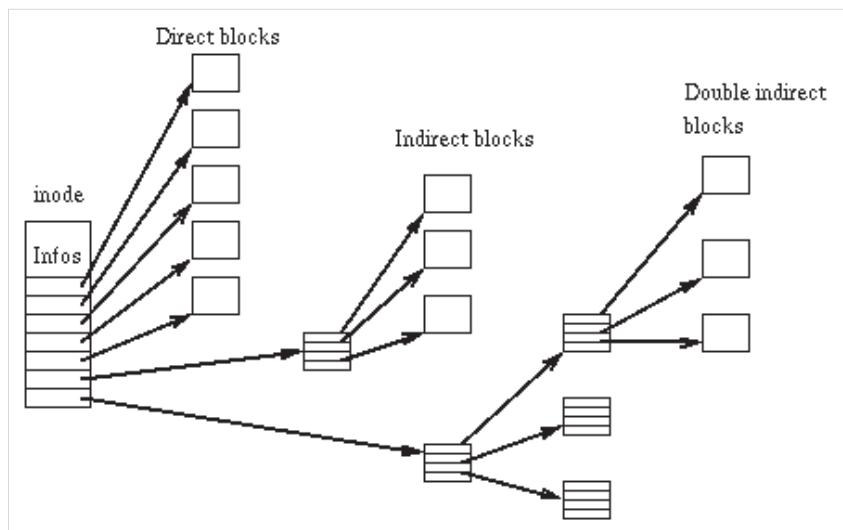


Fig. 1 Sistemul de fișiere

## 2. Accesul la fișiere

În Linux fișierele sunt accesate prin intermediul unor descriptori de fișiere. Fiecare proces putând avea deschise un anumit număr maxim de fișiere la același moment de timp, implicit acest număr este 256. Prin convenție descriptorii 0, 1 și 2 sunt alocați de către sistem la pornire:

- descriptorul 0 este utilizat ca și intrare standard
- descriptorul 1 este ieșire standard
- descriptorul 2 este ieșirea standard de eroare.

Fiecare descriptor de fișiere alocat este asociat cu un descriptor de fișiere deschis. Un descriptor de fișiere conține informații referitoare la un anumit fișier, un index care precizează unde va avea loc următorul acces în fișier, modul de acces la fișier, și alte informații legate de acesta.

Este posibil ca mai mulți descriptori de fișiere, chiar și aparținând unor procese separate să indice în același timp spre același descriptor de fișier deschis, adică informația dintr-o structură de descriptor deschis va fi utilizată simultan de mai mulți descriptori de fișiere.

Există cinci apeluri care generează descriptori de fișiere: creat, open, fcntl, dup și pipe. În continuare vor fi descrise primele două iar restul vor fi prezentate în capitolele următoare.

Apelul sistem open:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Rezultat: descriptorul de fișier sau -1 în caz de eroare.

Numărul de argumente al acestei funcții poate fi doi sau trei. Argumentul al treilea este folosit doar la crearea de fișiere noi. Apelul cu două argumente este folosit pentru citirea și scrierea fișierelor existente. Funcția returnează cel mai mic descriptor de fișier disponibil. Acesta poate fi utilizat în apelurile sistem: read, write, lseek și close. IDU efectiv sau GIDU efectiv al procesului care execută apelul trebuie să aibă drepturi de citire/scriere, funcție de valoarea argumentului flags.

Pointerul din fișier este poziționat pe primul octet din fișier.

Argumentul flags se formează printr-un SAU pe biți între constantele:

- O\_RDONLY - Fișierul este deschis în citire.
- O\_WRONLY - Fișierul este deschis în scriere.
- O\_RDWR - Fișierul este deschis în adăugare (citire+scriere).

Acestea sunt definite în fișierul antet fcntl.h.

Următoarele constante simbolice sunt optionale:

- O\_APPEND - Scriserile succese se produc la sfârșitul fișierului.
- O\_CREAT - Dacă fișierul nu există el este creat.
- O\_EXCL - Dacă fișierul există și O\_CREAT este poziționat, apelul open eșuează.
- O\_NDELAY - La fișiere pipe și cele speciale pe bloc sau caracter cauzează trecerea în modul fără blocare atât pentru apelul open cât și pentru operațiile viitoare de I/E. În versiunile noi System V înlocuit cu O\_NONBLOCK.
- O\_TRUNC Dacă fișierul există îl sterge conținutul.
- O\_SYNC Forțează scrierea efectivă pe disc prin write. Întârzie mult întregul sistem, dar este eficace în cazuri critice.

Argumentul al treilea (mode) poate fi o combinație de SAU pe biți între constantele simbolice:

- S\_IRUSR, S\_IWUSR, S\_IXUSR User: read, write, execute
- S\_IRGRP, S\_IWGRP, S\_IXGRP Group: read, write, execute
- S\_IROTH, S\_IWOTH, S\_IXOTH Other: read, write, execute

Acestea definesc drepturile de acces asupra unui fișier și sunt definite în fișierul antet sys/stat.h.

Apelul sistem creat:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat( const char *path, mode_t mod);
```

Întoarce: descriptorul de fișier sau -1 în caz de eroare.

Apelul este echivalent cu:

```
open( path, O_WRONLY | O_CREAT | O_TRUNC, mod);
```

Argumentul path specifică numele fișierului, iar mod drepturile de acces.

Dacă fișierul creat nu există, este alocat un nou inode și o legătură spre el este plasată în directorul unde acesta a fost creat. Proprietarul (IDU efectiv și GIDU efectiv) procesului care execută acest apel trebuie să aibă permisiunea de scriere în director. Fișierul deschis va avea drepturile de acces specificate de argumentul al doilea din apel. Apelul întoarce cel mai mic descriptor de fișier disponibil. Fișierul este deschis în scriere și dimensiunea sa inițială este 0. Timpii de acces și modificare din inode sunt actualizați.

Dacă fișierul există (este nevoie de permisiunea de căutare în director) conținutul lui este pierdut și el este deschis în scriere. Nu se modifică proprietarul sau drepturile de acces ale fișierului. Argumentul al doilea este ignorat.

Apelul sistem read:

Pentru a citi un număr de octeți dintr-un fișier, de la poziția curentă, se folosește apelul read. Sintaxa este:

```
#include <unistd.h>

ssize_t read( int fd, void *buf, size_t noct);
```

Înțoarce: numărul de octeți citiți efectiv, 0 la EOF, -1 în caz de eroare.

Citește *n* octeți din fișierul deschis referit de *fd* și îi depune în tamponul referit de *buf*. Pointerul în fișier este incrementat automat după o operație de citire cu numărul de octeți citiți. Procesul care execută o operație de citire așteaptă până când kernel-ul depune datele de pe disc în bufferele cache. În mod ușual, kernel-ul încearcă să accelereze operațiile de citire citind în avans blocuri de disc consecutive pentru a anticipa eventualele cereri viitoare.

Apelul sistem write

Pentru a scrie un număr de octeți într-un fișier, de la poziția curentă, se folosește apelul write. Sintaxa este:

```
#include <unistd.h>

ssize_t write( int fd, const void *buf, size_t noct);
```

Întoarce: numărul de octeți scriși și -1 în caz de eroare.

Apelul scrie noți octeți din tamponul buf în fișierul a cărui descriptor este fd.

Interesant de semnalat la acest apel este faptul că scrierea fizică pe disc este întârziată. Ea se efectuează la inițiativa nucleului fără ca utilizatorul să fie informat. Dacă procesul care a efectuat apelul, sau un alt proces, citește datele care încă nu au fost scrise pe disc, kernel-ul le citește înapoi din bufferele cache. Scrierea întârziată este mai rapidă, dar are trei dezavantaje:

- eroare disc sau căderea kernelului produce pierderea datelor;
- un proces care a inițiat o operație de scriere nu poate fi informat în cazul apariției unei erori de scriere;
- ordinea scriierilor fizice nu poate fi controlată.

Pentru a elimina aceste dezavantaje, în anumite cazuri, se folosește fanionul O\_SYNC. Dar cum aceasta scade viteza sistemului și având în vedere fiabilitatea sistemelor UNIX de astăzi se preferă mecanismul de lucru cu tampoane cache.

Apelul sistem close

Pentru a disponibiliza descriptorul atașat unui fișier în vederea unei noi utilizări se folosește apelul close.

```
#include <unistd.h>
```

```
int close( int fd );
```

Întoarce: 0 în caz de succes și -1 în caz de eroare.

Apelul nu golește bufferele kernel-ului și deoarece fișierul este oricum închis la terminarea procesului apelul se consideră a fi redundant.

Apelul sistem lseek

Pentru poziționarea absolută sau relativă a pointerului de fișier pentru un apel read sau write se folosește apelul lseek.

```
#include <sys/types.h>
#include <unistd.h>
```

```
off_t lseek( int fd, off_t offset, int interp );
```

Întoarce: un deplasament în fișier sau -1 în caz de eroare.

Nu se efectuează nici o operație de I/O și nu se trimite nici o comandă controlerului de disc. Dacă interp este SEEK\_SET poziționarea este față de începutul fișierului (primul octet din fișier este la deplasament zero). Dacă interp este SEEK\_CUR poziționarea este relativă la poziția curentă. Dacă interp este SEEK\_END poziționarea este față de sfârșitul fișierului.

Apelurile open, creat, write și read execută implicit lseek. Dacă un fișier este deschis folosind constanta simbolica O\_APPEND se efectuează un apel lseek la sfârșitul fișierului înaintea unei operații de scriere.

Programul următor realizează copierea unui fișier:

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char*argv[]) {
    struct stat buf;
    int fdr, fdw, rv;
    char buff[256];

    if((argv[1] == NULL) || (argv[2] == NULL)){
        printf("Utilizare: %s sursa destinatie\n",
        argv[0]);
        exit(0);
    }

    stat(argv[1],&buf);
    if(!S_ISREG(buf.st_mode)){
        perror(argv[1]);
        exit(0);
    }

    fdr = open(argv[1], O_RDONLY);
    fdw = open(argv[2], O_CREAT|O_WRONLY,
    S_IREAD|S_IWRITE);
    while(rv = read(fdr, &buff, 128)){
        write(fdw, &buff, rv);
    }

    return 0;
}
```

Pentru compilarea programului se folosește următoarea sintaxă:

```
$gcc -o copiere copiere.c
```

Exemplul de mai sus presupune că numele fișierului ce conține codul sursă este copiere.c, numele executabilului va fi copiere. Pentru a lansa în execuție programul vom scrie:

```
$/copiere fisier1 fisier2
```

### 3. Gestionaarea directoarelor

Citirea unui director se poate face de oricine are drept de citire asupra directorului respectiv. Scrierea unui director poate fi făcută numai de către kernel. Pentru obținerea de informații despre fișierele nedeschise se folosesc apelurile sistem stat, fstat și lstat:

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *path, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

Întoarce: 0 în caz de reușită și -1 în caz contrar.

Pentru fișierul specificat prin path sau filedes se completează o structură de informații care poate fi accesată prin intermediul lui buf. Cele trei funcții produc același rezultat deosebind ceea ce constă în modul de specificare a fișierului la care se referă. Astfel stat primește ca și argument calea către fișierul respectiv, fstat un descriptor de fișier, iar lstat se referă la un link.

Structura prin care se obțin informații referitoare la fișiere are următoarea formă:

```
struct stat {
    dev_t      st_dev; /* număr de dispozitiv */
    ino_t      st_ino;  /* inode */
    mode_t     st_mode; /* tip fișier-drepturi */
    nlink_t    st_nlink; /* număr de legături */
    uid_t      st_uid;  /* ID proprietar */
    gid_t      st_gid;  /* ID grup */
    dev_t      st_rdev; /* tip dispozitiv – fișiere speciale */
    off_t      st_size; /* dimensiune fișier */
    unsigned long st_blksize; /* dimensiunea blocului pentru I/E */
    unsigned long st_blocks; /* numărul de blocuri alocate */
    time_t     st_atime; /* data ultimului acces */
    time_t     st_mtime; /* data ultimei modificări */
    time_t     st_ctime; /* data ultimei modificări a stării */
};
```

Tipul de fișier codificat în câmpul st\_mode poate fi determinat folosind macroulile:

- S\_ISREG() Fișier obișnuit (REGular file).
- S\_ISDIR() Fișier director.
- S\_ISCHR() Dispozitiv special în mod caracter.
- S\_ISBLK() Dispozitiv special în mod bloc.
- S\_ISFIFO() Fișier pipe sau fifo.
- S\_ISLNK() Legătura simbolica.
- S\_ISSOCK() socket?

Apelul sistem opendir

```
#include <sys/types.h>
#include <dirent.h>
```

```
DIR *opendir( const char *pathname);
```

Întoarce: pointer dacă operația s-a încheiat cu succes, NULL în caz de eroare.

```
struct dirent *readdir( DIR *dp);
```

Întoarce: pointer dacă operația s-a încheiat cu succes, NULL în caz de eroare.

```
int closedir( DIR *dp);  
Întoarce -1 în caz de eroare.
```

Structura dirent definită în fișierul antet dirent.h este dependenta de implementare. SVR4 și 4.3+BSD definesc structura astfel încât ea conține cel puțin doi membri:

```
struct dirent {  
    ino_t d_ino;  
    char d_name[NAME_MAX +1];  
}
```

Structura DIR este o structură internă folosită de aceste patru funcții pentru a păstra informații despre directorul citit. Primul apel readdir citește prima intrare dintr-un director. Ordinea intrărilor dintr-un director este dependentă de implementare. De regulă nu este alfabetică.

Apelul sistem link

Pentru a adăuga o nouă legătură la un director se folosește apelul:

```
#include <unistd.h>  
  
int link(const char *oldpath, const char newpath);  
Întoarce: 0 în caz de reușită și -1 în caz contrar.
```

Argumentul oldpath trebuie să fie o legătură existentă, ea furnizează numărul inode-ului.

Dacă legăturile . și .. din fiecare director sunt ignorate structura sistemului de fișiere este arborescentă. Programe care parcurg structura ierarhica (de exemplu, comanda find) pot fi ușor implementate fără probleme de traversare multiplă sau bucla infinită. Pentru a respecta această cerință doar superuser-ul are dreptul să stabilească o nouă legătură la un director. Crearea celei de a doua legături la un director este utilă pentru a-l putea muta în altă parte a arborelui sau pentru a-l putea redenumi.

Apelul sistem unlink

Pentru a șterge o legătură (cale) dintr-un director se folosește apelul:

```
#include <unistd.h>  
  
int unlink( const char *path);  
Întoarce: 0 în caz de reușită și -1 în caz contrar.
```

Apelul decrementează contorul de legături din inode și șterge intrarea director. Dacă acesta devine 0 spațiul ocupat de fișierul în cauză devine disponibil pentru o altă utilizare, la fel și inode-ul. Doar superuser-ul poate să șteargă un director.

Programul următor listează conținutul unui catalog(director):

```
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/stat.h>
```

```
#include <unistd.h>
#include <dirent.h>

int main(int argc, char*argv[ ]) {
    struct stat buf;
    struct dirent *entry;
    DIR *dr;

    if(argv[1] == NULL) {
        printf("Utilizare: %s director\n" , argv[0]);
        exit(0);
    }

    stat(argv[1],&buf);
    if(!S_ISDIR(buf.st_mode)) {
        perror(argv[1]);
        exit(0);
    }
    dr = opendir(argv[1]);
    while(entry = readdir(dr)) {
        printf("%s\n",entry->d_name);
    }

    return 0;
}
```

#### 4. Atributele fișierelor

Pentru o gestionare eficientă a fișierelor este necesară nu numai cunoașterea unui număr cât mai mare de date referitoare la acestea dar și posibilitatea modificării acestor date. În paragraful următor vor fi prezentate o serie de funcții folosite pentru modificarea atributelor fișierelor.

Una dintre caracteristicile esențiale ale sistemului de operare Linux este securitatea. Implementarea acesteia la nivel de structură de fișiere se face prin precizarea pentru fiecare fișier în parte a drepturilor de citire, scriere și execuție pentru proprietarul fișierului, pentru grupul proprietarului și pentru toate celelalte clase de utilizatori. Aceste drepturi pot fi modificate cu ajutorul apelurilor sistem chmod și fchmod:

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

Întoarce: 0 în caz de succes și -1 în caz contrar.

Pentru a modifica drepturile de acces la fișiere este necesar ca procesul să aibă cel puțin UID(user id) efectiv efectiv egal cu al proprietarului fișierului, dacă nu drept de superuser.

Parametrii mode pot fi precizați atât în formă octală cât și sub forma unor constante simbolice de forma:

- S\_ISUID sud la execuție (04000)
- S\_ISGID sgid la execuție (02000)
- S\_ISVTX bitul sticky (01000)
- S\_IRUSR, S\_IWUSR, S\_IXUSR – drepturile de citire, scriere și execuție pentru proprietarul fișierului.
- S\_IRGRP, S\_IWGRP, S\_IXGRP – pentru grupul utilizatorului.
- S\_IRO, S\_IWO, S\_IXO – pentru toți ceilalți utilizatori.

Un alt element de securitate în ceea ce privește sistemul de fișiere este apelul sistem umask:

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask(mode_t mask);
```

În momentul în care se încearcă modificarea drepturilor de acces asupra unui fișier automat se ține cont și de valoarea atributului mask al funcției umask după formula mode & ~(mask). Ex.:

```
int fd;
umask(0222);
if((fd=creat("tmp", 0666)==-1)) printf("creat error");
system("ls -l tmp");
Rezultat: -r--r-- 1 student user 0 Mar 18 11:05 tmp
```

Drepturile de acces la fișierul tmp precizate prin intermediul apelului sistem creat erau 0666(r w - r w - r w - ) dar prin luarea în considerare a măștii s-a obținut ca și rezultat 0222(r -- r -- r --).

Fiecare utilizator primește la intrarea în sistem o mască, valoarea implicită a acesteia este 022, dar interpretorul de comenzi permite modificarea acesteia prin intermediul comenzii umask.

Apelul sistem chown:

```
#include <sys/types.h>
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
Întoarc: 0 în caz de succes -1 în caz contrar.
```

Efecte vizibile la aplicarea acestui apel sistem se pot obține numai dacă utilizatorul are drepturi de superuser, sau este proprietarul fișierului dar în acest caz poate modifica doar grupul cu unul din care face parte.

# Procese - Managementul memoriei

Într-un sistem mono procesor, memoria principală este împărțită în 2 părți, o parte pentru sistemul de operare (monitorul rezident, kernel) iar cealaltă pentru programul care se află în execuție.

Într-un sistem multi-program, partea din memorie alocată utilizatorilor trebuie să fie împărțită astfel încât să fie atribuită fiecărui proces corespunzător userului respectiv. Taskul de împărțire și gestionare al memoriei poartă numele de managementul memoriei. Managementul memoriei este un task vital sistemelor de operare deoarece în momentul în care procesele așteaptă evenimente de la dispozitivele de I/O sau datorită faptului că procesorul e ocupat, memoria trebuie să fie foarte bine gestionată pentru a putea încărca cât mai multe procese.

In mod normal, programatorul nu cunoaște în avans care alte programe vor fi rezidente în memoria principală în timpul execuției propriului program. In plus, ne-ar place să putem face un swap anumitor procese pentru a mări spațiul pentru cele aflate în starea de READY. După ce un proces a fost swapat el va reveni înapoi în memoria principală la o altă adresă. Aceasta reprezintă relocalizarea. In acest caz, referințele de memorie trebuie translatăte în cod la noile adrese fizice. Datorită acestui fapt apar anumite aspecte tehnice ilustrate în figura următoare. Figura descrie imaginea unui proces. Pentru simplitate vom considera că imaginea procesului ocupă o zonă continuă în memoria principală. Sistemul de operare trebuie să cunoască adresa unde găsește informația de control a procesului și a stivei ca și punct de intrare pentru execuția programului corespondent acestui proces. Deoarece sistemul de operare este responsabil pentru aducerea proceselor în memoria principală, aceste adrese sunt ușor de obținut. In plus, procesorul trebuie să folosească referințele de memorie din program.

Din această cauză, hardware-ul (dat de procesor) și software-ul (dat de sistemul de operare) trebuie să fie capabile să translateze referințele de memorie găsite în program în adrese fizice reflectând locația curentă a programului în memoria principală.

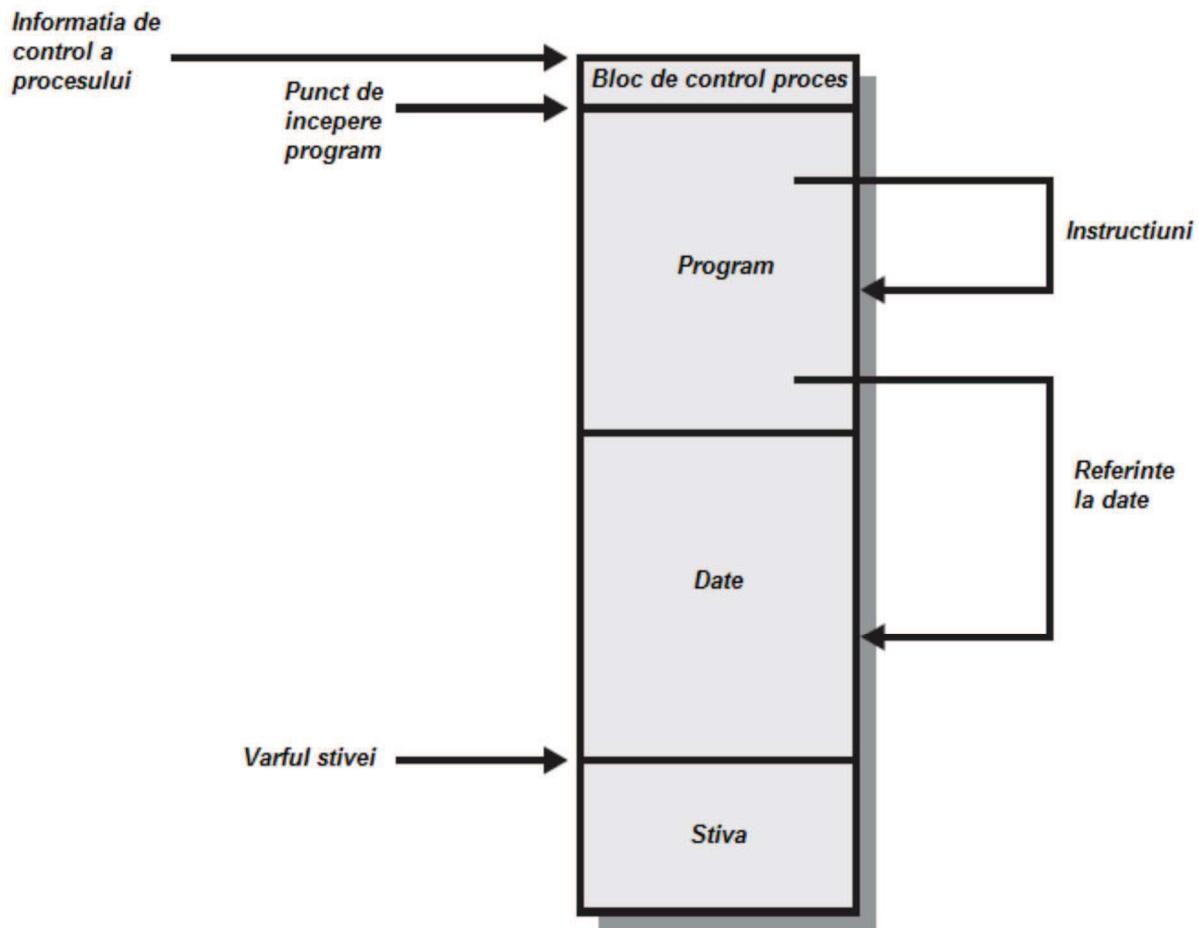


Figura 1. Imaginea unui proces în memorie

Fiecare proces trebuie protejat împotriva nedoritelor interferențe ale altor proceze, provocate accidental sau intenționat. Astfel, programele reprezentând alte proceze trebuie să nu fie capabile de a accesa referințe de memorie ale altui proces pentru citire sau scriere fără o permisiune în acest sens. Deoarece localizarea programului în memoria principală este necunoscută, e imposibilă verificarea adreselor absolute pentru asigurarea protecției. Imaginea procesului prezentată în figura anterioară ilustrează cerințele legate de protecție. Normal, un proces utilizator nu poate accesa o zonă a sistemului de operare, program sau date. Apoi un program al unui proces nu poate fi continuat cu o instrucție a altui proces în mod ușor.

Nici zona de date nu poate fi accesată între proceze fără a exista un protocol în acest sens. Fiecare mecanism de protecție trebuie să fie flexibil astfel încât să permită mai multor proceze să acceseze o aceeași zonă de memorie. Procesele care coopereză pentru îndeplinirea unui task trebuie să dețină un acces partajat asupra unei zone comune de date. Sistemul de management al memorie trebuie să permită un acces partajat la aceeași zonă de memorie fără a încălca protecția datelor.

Programele sunt organizate în module în timp ce memoria are o organizare logică dată de secvențe de biți. O serie de avantaje pot apărea dacă sistemul de operare vede programele și datele organizate sub forma unor module:

- modulele pot fi scrise și compilate independent cu toate referințele dintre ele rezolvate în momentul rulării;
- pot fi atribuite diverse grade de protecție diferitelor module destul de ușor;
- se poate realiza o distribuire ușoara a modulelor către mai mulți utilizatori.

Memoria calculatorului este organizată în cel puțin 2 nivele: memoria principală, scumpă dar foarte rapidă și memoria secundară, mai ieftină dar înceată. Un task important pentru managementul memoriei îl constituie organizarea memoriei pe cele două nivale. Una dintre principalele operații pentru sistemul de operare este aducerea programelor în memoria principală pentru a fi executate de către procesor. În sistemele moderne aceasta este cunoscută sub numele de memorie virtuală. Memoria virtuală se bazează pe una din următoarele tehnici de organizare: segmentare sau paginare. În cazul partițiilor de dimensiune egală (partiționare fixă) un proces poate fi încărcat într-o anumită parte din partitie dacă dimensiunea lui este mai mică sau egală cu dimensiunea partitiei. În cazul în care toate partițiile sunt pline, procesul poate fi swapat. În cazul în care un program nu poate intra într-o partitie, acest lucru trebuie specificat. Practic, sistemul nu este eficient deoarece un program cât de mic va ocupa o întreagă partitie duce la fragmentare internă.

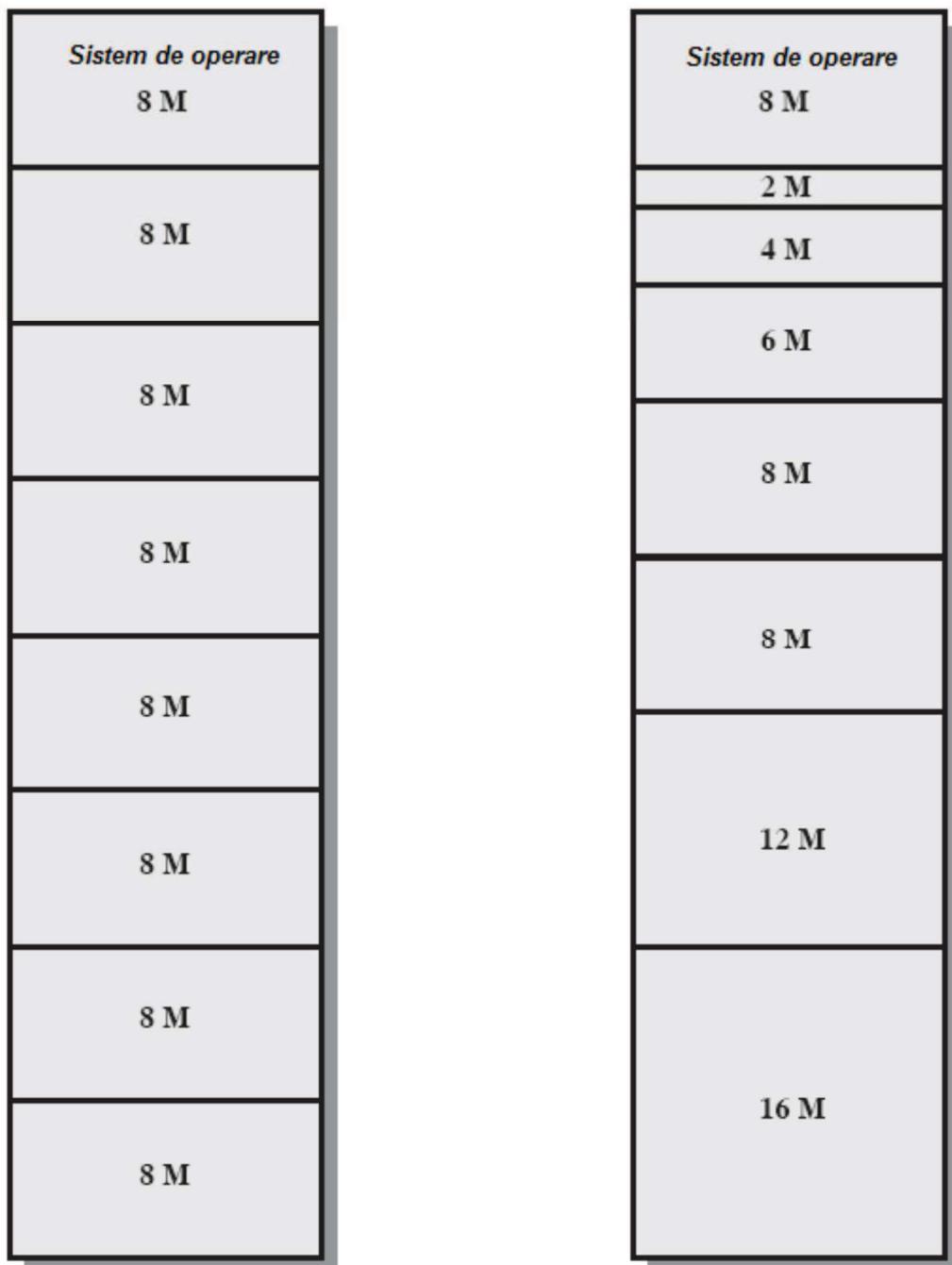


Figura 2. Exemple de partiționare fixă

In figura 2. avem 2 exemple pentru partităonare fixă. Prima figură ne prezintă partitii de dimensiuni egale în timp ce a doua partităonare are dimensiuni diferite pentru partitiiile care o compun.

Procesul de partităonare fixă are multe neajunsuri. Datorită partitiiilor fixe nu contează care dintre ele este utilizată. În cazul în care avem partitii de dimensiuni diferite există 2 cai de atribuire a proceselor. Cea mai simplă metodă este atribuirea fiecărui proces celei mai mici partitii în care acesta intră. Apare o coadă de aşteptare pentru fiecare partitie pentru că la un moment dat e posibil ca mai multe procese de dimensiuni apropiate să dorească aceeași partitie. Avantajele constau în faptul că este minimizată cantitatea de memorie irosită. Dar această modalitate este optimă din punct de vedere al unei partitii dar nu și din punctul de vedere al întregului sistem.

In figura următoare observăm cele 2 mecanisme care sunt discutate. Presupunând că la un moment dat o zonă de memorie de 8 M nu este ocupată de nici un proces dar există cozi de aşteptare pentru porțiuni mai reduse de memorie ne dăm seama că acest prim mecanism poate duce la diverse întârzieri care ar putea fi evitate dacă s-ar permite unui proces mai mic să folosească și zone mai mari de memorie. Astfel, avem o a doua abordare cu o singură coadă de aşteptare pentru toate procesele. Pentru un proces va fi aleasă cea mai mică partitie pe care acesta o poate ocupa. In cazul în care toate partitiiile sunt ocupate, atunci va fi luată o decizie de swapare. In acest caz pot fi luați în calcul diferenți factori precum prioritatea sau alte criterii.

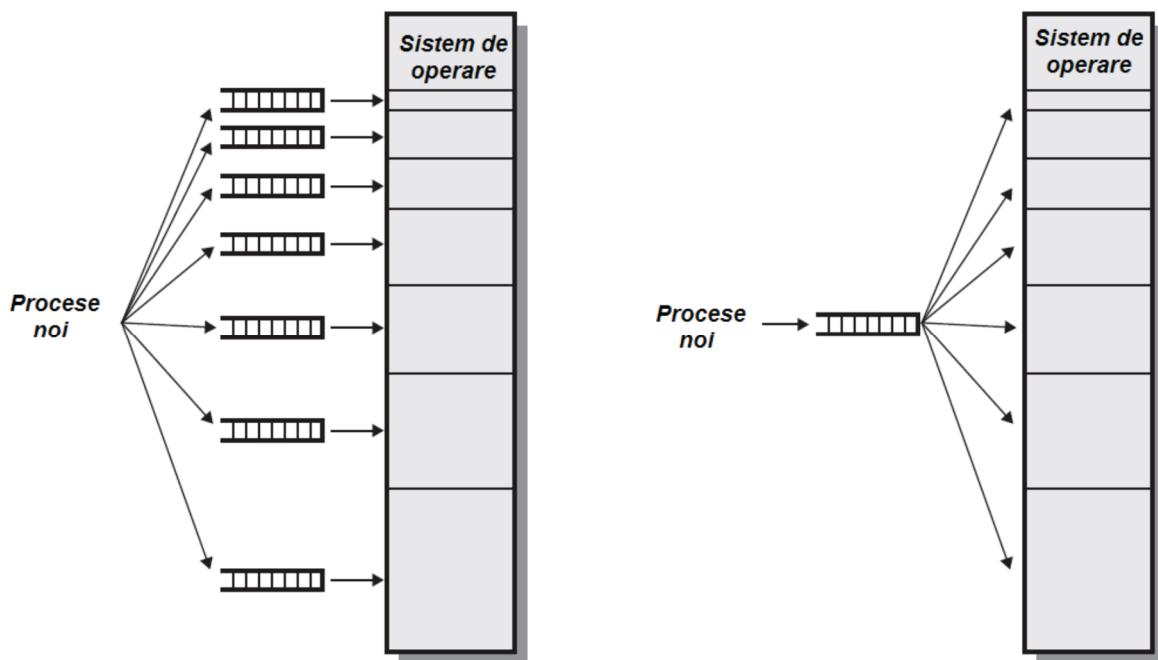


Figura 3. Metode de partităonare a memoriei

Partităonarea prin această metodă oferă o relativă flexibilitate dar mai apar o serie de dezavantaje:

- numărul de partitii limitează numărul de procese active din sistem;
- ocuparea partitiiilor nu este eficientă.

In prezent acest mod de partităonare nu mai este folosit.

In cadrul partităonării dinamice, partitiiile sunt de lungime și număr variabile. Fiecare proces are alocată exact atâtă memorie cât are nevoie. In figura 4, inițial memoria principală este goală, în afară de partea ocupată de sistemul de operare. Primele 3 procese sunt încărcate începând cu locația la care se termină sistemul de operare și ocupă doar spațiul suficient pentru fiecare proces. Apoi sistemul de operare face un swap la 2 procese lăsând suficient loc pentru altele datorită faptului că la

acel moment nici un proces nu e in stare de Ready. Deoarece procesul 4 este mai mic ca și ocupare în memorie decât procesul 2 apare un spațiu liber în memorie. Dacă se dorește aducerea din nou a procesului 2, nu mai este posibil datorită faptului că acesta ocupă prea mult loc. Pentru a putea fi adus se face un swap la procesul 1. Aceasta ilustrare ne conduce la apariția unei fragmentări externe datorată apariției spațiilor libere în memorie. O metodă de îmbunătățire a situației se numește compactare. Din timp în timp, sistemul de operare face un shift pentru procente astfel ca acestea să ocupe cât mai bine spațiul de memorie pe care îl au la dispoziție.

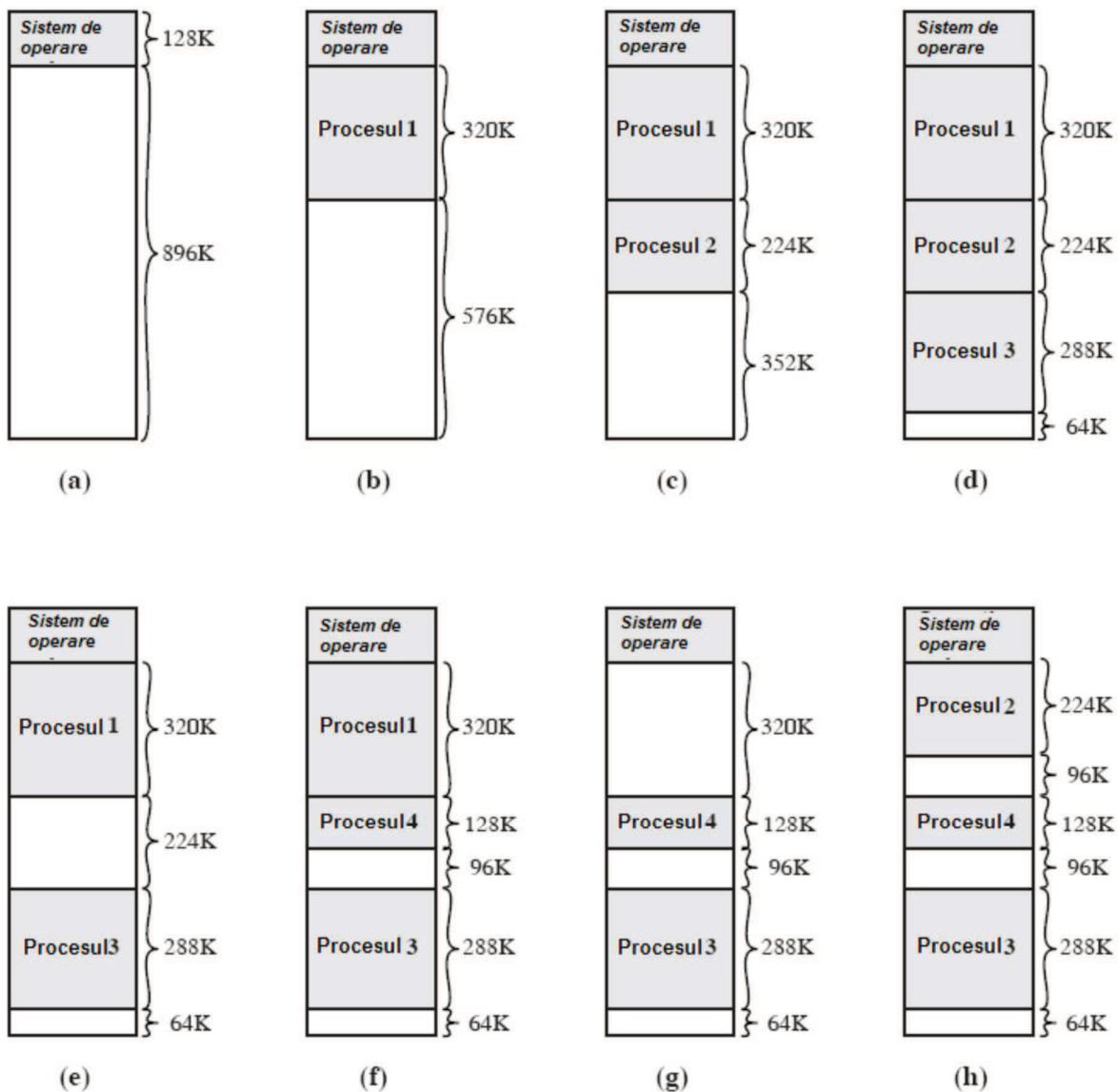


Figura 4. Exemplu de partiționare dinamică

Datorită faptului că procesul de compactare este unul consumator de timp, sistemul de operare trebuie să știe cum să atribuie procente spațiului de memorie. Trei algoritmi de plasare pot fi folosiți. Primul dintre aceștia se numește “best-fit” și alege blocul care este cel mai apropiat ca dimensiune de dimensiunea procesului. Algoritmul al doilea poartă denumirea de “first-fit” și pornește căutarea de la începutul spațiului de memorie alegând blocul de memorie care este suficient de “încăpător” pentru proces. Ultimul, “next-fit” scaneză zona de memorie începând de la ultimul bloc atribuit până găsește unul corespunzător pentru proces. În cele 2 reprezentări putem observa diferențele dintre cei trei algoritmi. Astfel, pentru first-fit este găsită prima zonă de memorie liberă în care putem pune procesul respectiv, pentru best-fit este găsită cea mai bună zonă iar pentru next-fit

zona de memorie începând de la ultima alocare. Pentru a stabili care dintre cei algoritmi este mai bun trebuie să luăm în considerare și modul de apariție sau de swapare al proceselor în ultimele două cazuri. În general cel mai utilizat este next-fit și pentru că se raportează la un spațiu de memorie care a fost ultimul ocupat. Best-fit este cel mai bun din punct de vedere al ocupării memoriei dar și cel mai încet, în timp ce first-fit este cel mai simplu dar nu cel mai eficient.

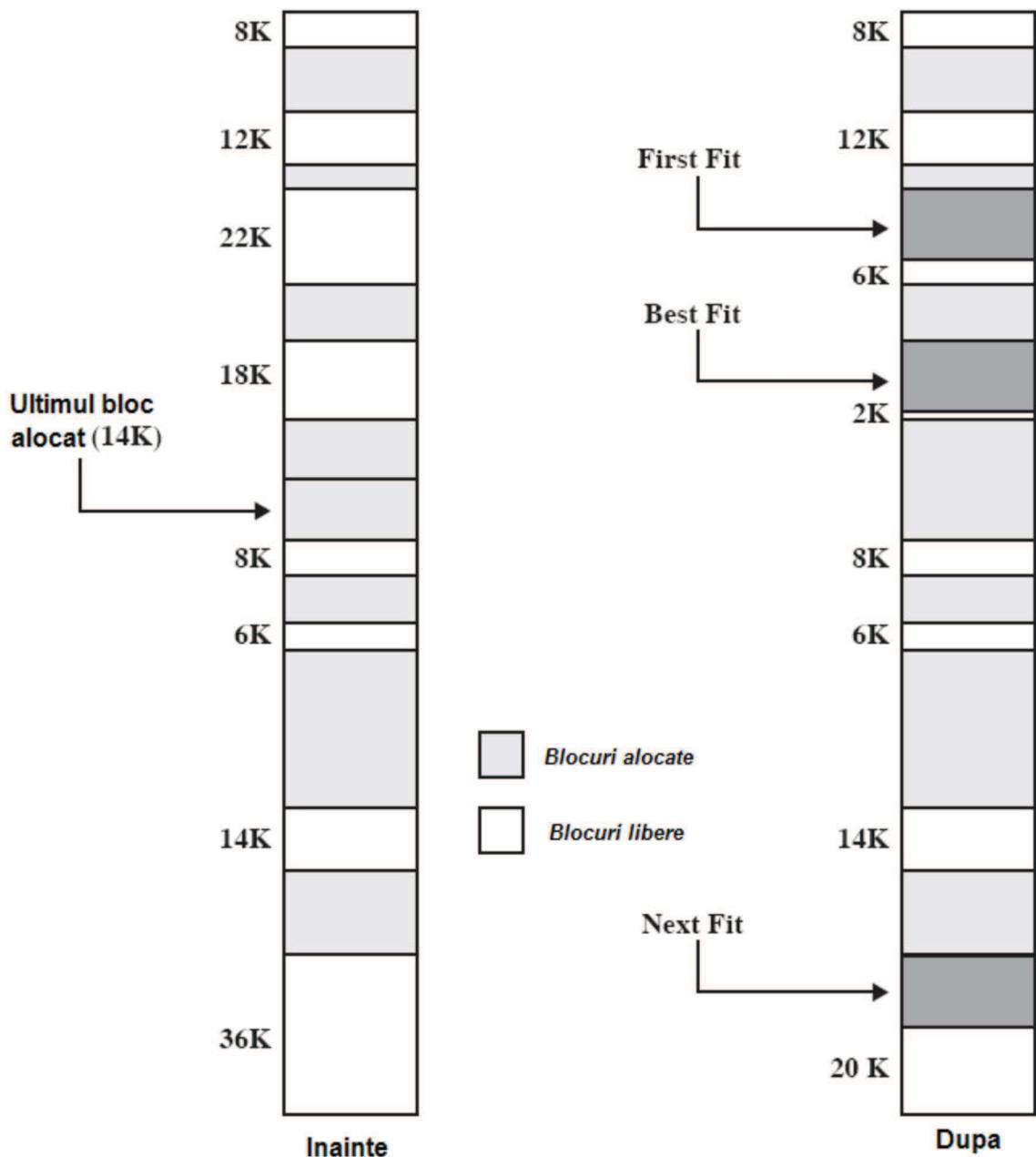


Figura 5. Comparație între cei 3 algoritmi

Într-un sistem de operare cu multiprogramare se ajunge la un moment dat în situația în care toate procesele din memoria principală sunt în starea blocat și chiar și după compactare există prea putină memorie pentru a asigura spațiul pentru alte procese. Atât partitōnarea fixă cât și cea dinamică au problemele lor. O partitōnare fixă limitează numărul proceselor și poate duce la o folosire ineficientă a spațiului de memorie în timp ce una dinamică e mai complicată de realizat și apar probleme la compactare. Un compromis interesant îl reprezintă sistemele de tip buddy. Într-un astfel de sistem,

spațiul disponibil pentru alocare este văzut ca blocuri de dimensiunea  $2k$  unde  $k$  poate lua valori între 0 și  $l$  și să obținându-se astfel blocuri de mărime maximă sau minimă.

Pentru început, întreg spațiul este tratat ca un singur bloc cu dimensiunea  $2u$ . Dacă este făcută o cerere mai mică decât această valoare, atunci întregul bloc este alocat. Altfel, întregul bloc este împărțit în două de dimensiuni  $2u-1$ . Dacă unul dintre cele neîmpărțite la doi poate conține procesul, atunci acesta este încărcat. Dacă nu, algoritmul continuă până se ajunge la dimensiunea dorită. La fiecare moment de timp sistemul menține o listă de blocuri de dimensiune inferioară care nu sunt alocate. Un spațiu liber din memorie poate fi împărțit în două și trecut între blocurile de dimensiune corespunzătoare care pot fi ocupate cu procese. În momentul în care o pereche de blocuri rămân libere, ele pot fi unite și se obține un bloc cu o dimensiune mai mare.

1 Mbyte	1 M				
Cerere 100 K	A = 128 K	128 K	256 K		512 K
Cerere 240 K	A = 128 K	128 K	B = 256 K		512 K
Cerere 64 K	A = 128 K	C = 64 K	64 K	B = 256 K	512 K
Cerere 256 K	A = 128 K	C = 64 K	64 K	B = 256 K	D = 256 K
Eliberare B	A = 128 K	C = 64 K	64 K	256 K	D = 256 K
Eliberare A	128 K	C = 64 K	64 K	256 K	D = 256 K
Cerere 75 K	E = 128 K	C = 64 K	64 K	256 K	D = 256 K
Eliberare C	E = 128 K	128 K		256 K	D = 256 K
Eliberare E			512 K		D = 256 K
Eliberare D				1 M	

Figura 6. Exemplu sistem de tip buddy

În figură considerăm un bloc inițial de 1 Mbyte. Prima cerere, A este de 100 kbytes pentru care alocăm un bloc de 128 kbytes. Blocul inițial este împărțit de 3 ori pentru a obține segmentul dorit. Următoarea cerere, B, ne solicită un bloc de 256K. Acest bloc este deja disponibil și poate fi alocat fără probleme. Procesul continuă. Observăm că în momentul în care se eliberează blocul de memorie notat cu E acesta este imediat unit cu jumătatea lui, s.a.m.d. până la eliberarea întregului spațiu.

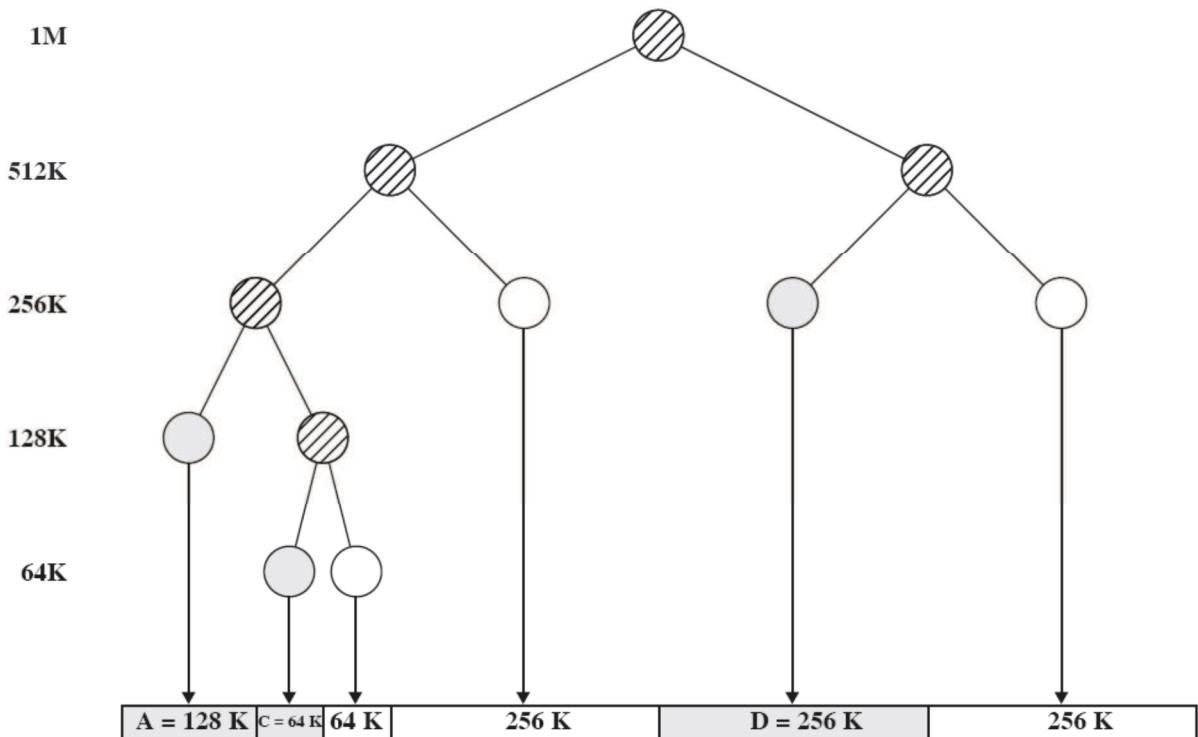


Figura 7. Structura arborescentă a sistemelor de tip buddy

Sistemul buddy are o alocare de tip arborescent prezentată în figura 7. Frunzele arborelui corespund partiționării curente a memoriei. Dacă 2 părți de memorie sunt frunze înseamnă că cel puțin una trebuie alocată. În caz contrar, ele vor fi unite într-un bloc mai mare. Cu toate că sistemul îmbunătășește partiționarea fixă sau dinamică, în sistemele moderne folosirea paginării sau segmentării la memoria virtuală dă rezultate îmbunătățite. Doar Unix-ul mai folosește o formă modificată a acestui algoritm, dar cu tendințe de schimbare.

### 1. Paginarea

O modalitate foarte răspândită pentru managementul memoriei este cea de paginare. Prin aceasta înțelegem împărțirea în bucăți mici a memoriei și împărțirea fiecărui proces într-un număr de entități de această dimensiune. În cazul procesului, părțile lui se numesc pagini iar cele de memorie se mai numesc frame-uri. Sistemul de operare întreține un tabel pentru fiecare proces. Acest tabel conține locații pentru fiecare pagină a procesului și adrese de memorie cu numărul paginii și un deplasament în pagină. Odată cu paginarea, convertirea din adrese logice în adrese fizice este realizată de către procesor.

În figura 8. am reprezentat un exemplu de încărcare a proceselor în memorie. Procesul A conține 4 pagini. Procesul B este încărcat după procesul A și are doar 3 pagini. Apoi este încărcat și procesul C cu 4 pagini. Procesul B este suspendat și swappat din memoria principală. Procesul D care este adus în memoria principală ocupă 5 pagini și va ocupa locațiile lăsate libere de procesul B și încă 2 în plus. Observăm că paginarea simplă înseamnă de fapt o împărțire fixă a memoriei cu diferența că paginile sunt entități mici și un proces poate fi împărțit în mai multe astfel de entități.

Memorie principală		Memorie principală		Memorie principală	
Numar frame		0	A.0	0	A.0
0		1	A.1	1	A.1
1		2	A.2	2	A.2
2		3	A.3	3	A.3
3		4		4	B.0
4		5		5	B.1
5		6		6	B.2
6		7		7	
7		8		8	
8		9		9	
9		10		10	
10		11		11	
11		12		12	
12		13		13	
13		14		14	

Memorie principală		Memorie principală		Memorie principală	
0	A.0	0	A.0	0	A.0
1	A.1	1	A.1	1	A.1
2	A.2	2	A.2	2	A.2
3	A.3	3	A.3	3	A.3
4	B.0	4		4	D.0
5	B.1	5		5	D.1
6	B.2	6		6	D.2
7	C.0	7	C.0	7	C.0
8	C.1	8	C.1	8	C.1
9	C.2	9	C.2	9	C.2
10	C.3	10	C.3	10	C.3
11		11		11	D.3
12		12		12	D.4
13		13		13	
14		14		14	

Figura 8. Exemplu de paginare în cazul celor 4 procese

## 2. Memoria virtuală

Memoria virtuală este o tehnică ce permite execuția proceselor fără a impune necesitatea ca întreg fișierul executabil să fie încărcat în memorie sau capacitatea de a adresa un spațiu de memorie mai mare decât este cel din memoria internă a unui calculator. Utilizarea memoriei virtuale oferă avantajul că poate fi executat un program de dimensiune oricât de mare (mai mare decât dimensiunea memoriei fizice). Spațiul de adrese al procesului este divizat în părți care pot fi încărcate în memoria internă atunci când execuția procesului necesită acest lucru și transferate înapoi în memoria secundară, când nu mai este nevoie de ele. Spațiul de adrese al unui program se împarte în partea de cod, cea de date și cea de stivă, identificate atât de compilator cât și de mecanismul hardware utilizat pentru relocare.

Partea (segmentul) de cod are evident un număr de componente mai mare, fiind determinată de fazele de execuție (logică) ale programului. De exemplu, aproape toate programele conțin o fază necesară inițializării structurilor de date utilizate în program, alta pentru citirea datelor de intrare, una (sau mai multe) pentru efectuarea unor calcule, altele pentru descoperirea erorilor și una pentru ieșiri. Analog, există partiții ale segmentului de date. Această caracteristică a programului se numește localizare a referințelor în spațiu și este foarte importantă în strategiile utilizate de către sistemele de memorie virtuală.

Când o anumită parte a programului este executată, este utilizată o anumită porțiune din spațiul său de adrese, adică este realizată o localizare a referințelor. Când se trece la o altă fază a calculului, corespunzătoare logiciei programului, este preferențiată o altă parte a spațiului de adrese și, deci se schimbă această localizare a referințelor.

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

Translatarea adreselor virtuale. Fie  $N = \{d_0, d_1, \dots, d_{n-1}\}$  mulțimea paginilor în spațiul de adrese virtuale și  $M = \{b_0, b_1, \dots, b_{m-1}\}$  mulțimea paginilor fizice din memoria primară alocate procesului. O adresă virtuală este un întreg i, unde  $0 \leq i \leq G = 2^{g+h}$ , deoarece există  $n = 2^g$  pagini, fiecare având  $2^h$  cuvinte (locații) de memorie. O adresă fizică, k, este o adresă de memorie, de forma  $k = U2^h + V (0 \leq V \leq 2^h)$ , unde U este numărul pagini fizice. Deci  $U2^h$  este adresa din memoria primară a primei locații din pagina fizică, iar V este deplasamentul din cadrul paginii fizice U. Deoarece procesului îi sunt alocate  $2^j$  pagini fizice, vom avea  $H = 2^{j+h}$  locații de memorie fizică care pot fi utilizate de proces.

Deoarece fiecare pagină are aceeași dimensiune c, adresa virtuală i poate fi convertită într-un număr de pagină și un deplasament în cadrul paginii, numărul de pagină fiind l div c, iar deplasamentul i mod c. Dacă considerăm reprezentarea binară a adresei, atunci numărul de pagină poate fi obținut prin deplasare spre dreapta a adresei cu h poziții, iar deplasamentul prin extragerea primilor g biți cei mai puțin semnificativi din adresă. Deci fiecare adresă virtuală, respectiv adresă fizică va fi de forma (p,d), respectiv de forma (f,d), unde p este numărul paginii virtuale, f este numărul paginii fizice, iar d este adresa (deplasamentul) în cadrul paginii.

Orice pagină virtuală din spațiul de adrese al procesului, poate fi încărcată în oricare din paginile fizice din memoria internă alocate acestuia. Acest lucru este realizat printr-o componentă hardware numită MMU(Memory Management Unit). Dacă este referențiată o locație dintr-o pagină care nu este încărcată în memoria internă, MMU oprește activitatea unității centrale, pentru ca sistemul de operare să poată executa următorii pași:

- Procesul care cere o pagină neîncărcată în memoria internă este suspendat.
- Administratorul memoriei localizează pagina respectivă în memoria secundară.

- Pagina este încărcată în memoria internă, eventual în locul altelui pagini, dacă în memoria internă nu mai există pagini fizice libere alocate procesului respectiv și în acest caz, tabela de pagini este modificată.
- Execuția procesului se reia din locul în care a fost suspendat.

Fiecare proces are propria lui tabelă de pagini, în care este trecută adresa fizică a paginii virtuale, dacă ea este prezentă în memoria operativă. La încărcarea unei noi pagini virtuale, aceasta se depune într-o pagină fizică liberă. Deci, în memoria operativă, paginile fizice sunt distribuite în general necontinuu, între mai multe proceze. Spunem că are loc o proiectare a spațiului virtual peste cel real.

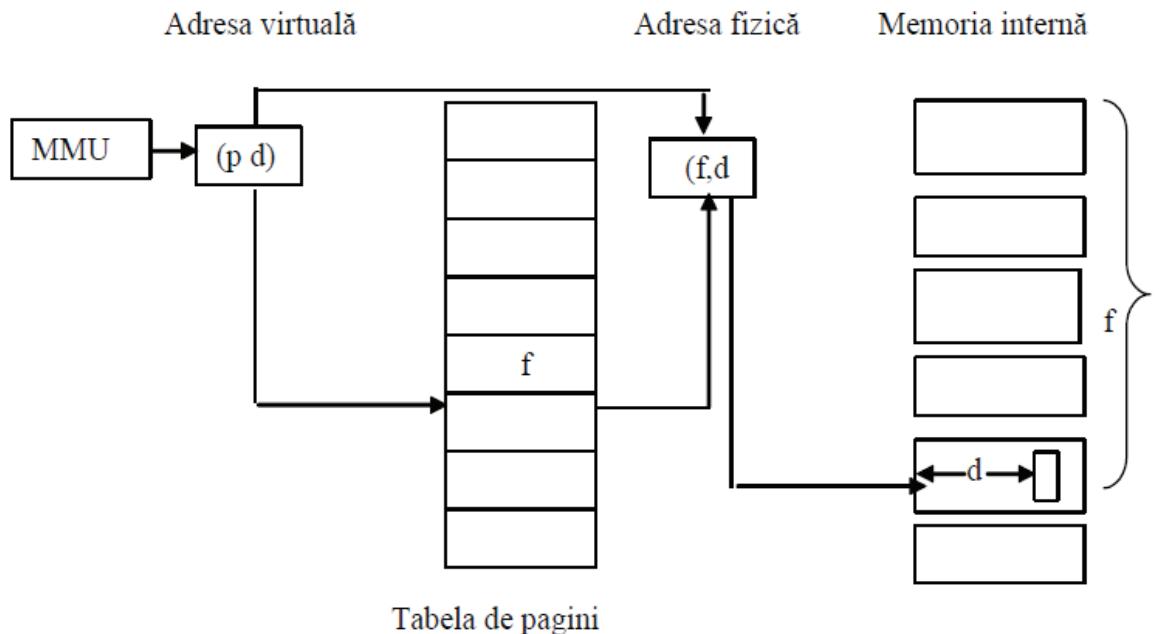


Figura 9. Translatarea unei pagini virtuale într-o fizică

Acest mecanism are avantajul că folosește mai eficient memoria operativă, fiecare program ocupând numai memoria strict necesară la un moment dat. Un alt avantaj este posibilitatea folosirii în comun, de către mai multe programe, a instrucțiunilor unor proceduri. O procedură care permite acest lucru se numește procedură reentrantă.

Evidența paginilor virtuale încărcate în pagini fizice, este realizată printr-o tabelă. Dacă prin  $M[0..m]$  notăm memoria operativă, prin j puterea lui 2 (numărul de biți) care dă lungimea unei pagini, prin TP adresa de start a tabelei de pagini, atunci pe baza adresei virtuale  $(p, d)$  se poate determina locația de memorie fizică corespunzătoare, care este  $M[TP + p] * 2^j + d$ , dacă locația virtuală respectivă este încărcată în memorie.

Formula anterioară este valabilă atunci când tabela de pagini ocupă un spațiu în memoria operativă. Tabela de pagini poate fi privită ca o funcție, care are ca argument numărul de pagină virtuală și care determină numărul de pagină fizică. Pe baza deplasamentului se determină locația din memoria fizică unde se va încărca locația virtuală preferențiată. Această metodă ridică două mari probleme:

- tabela de pagini poate avea un număr mare de intrări. Calculatoarele moderne folosesc adrese virtuale pe cel puțin 32 de biți. Dacă, de exemplu o pagină are dimensiunea de 4K, atunci numărul intrărilor în tabelă este mai mare decât un milion. În cazul adreselor pe 64 de biți numărul intrărilor în tabelă va fi, evident mult mai mare.

- corespondența dintre locația de memorie virtuală și cea din memoria fizică trebuie să se realizeze cât mai rapid posibil; la un moment dat, o instrucțiune cod mașină poate referențial una sau chiar două locații din memoria virtuală.

Pentru rezolvarea acestei probleme, s-au încercat mai multe soluții. O primă metodă constă în folosirea de registre ai unității centrale pentru memorarea intrărilor în tabelă, soluție care asigură un acces rapid dar care este foarte costisitoare. O a doua metodă propune ca tabela de index să fie încărcată în memoria primară, folosind un registru pentru a memora adresa de început a tabelei. Această metodă permite ca tabela de index să fie încărcată în zone diferite din memorie, prin modificarea conținutului registrului. O metodă mult mai eficientă, care înlocuiește căutarea secvențială cu cea arborescentă este organizarea tabelei pe mai multe niveluri. Astfel, o adresă virtuală pe 32 de biți de exemplu, este un triplet (Pt1, Pt2, d), primele două câmpuri având o lungime de 10 biți, iar ultimul de 12 biți. În figura 10 este ilustrată o astfel de tabelă de pagini.

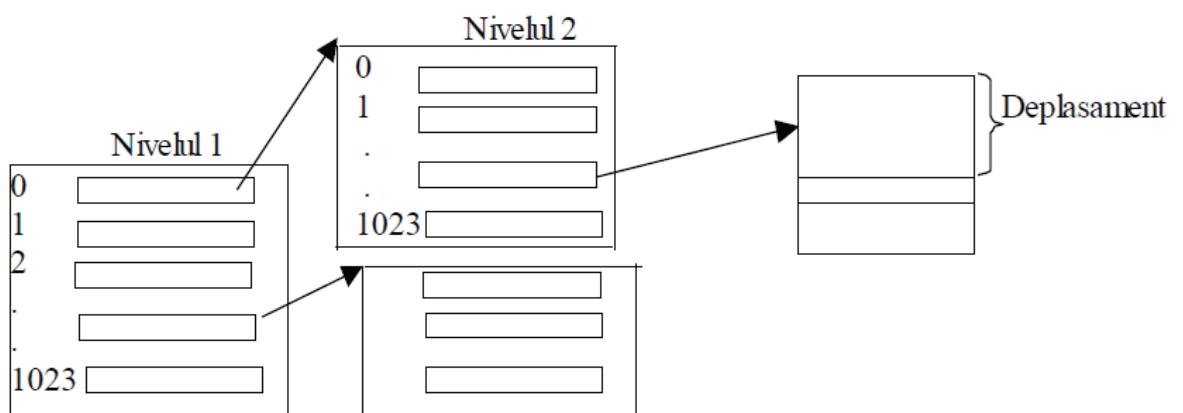


Figura 10 Tabele de pagini pe 2 niveluri

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

Pe lângă câmpul care conține numărul de pagină fizică, intrarea respectivă mai conține și alte câmpuri, dintre care cele mai semnificative sunt:

- Bitul prezent/absent indică faptul că pagina respectivă este/ sau nu (setat pe 1/setat pe 0) încărcată în memoria fizică.
- Biții de protecție specifică ce fel de tip de acces este permis. În forma cea mai simplă, acest câmp este de 1 bit, setat pe 0 pentru permisiune de citire și scriere și 1 numai pentru citire. Sistemele moderne folosesc 3 biți de protecție, unul setat pentru permisiunea de citire, unul pentru scriere și unul care specifică dreptul de execuție.
- Bitul modifikat indică dacă în pagina respectivă s-a scris sau nu.
- Bitul referit indică dacă pagina respectivă a fost preferențiată sau nu.
- Bitul "caching disabled/enabled" indică posibilitatea încărcării paginii respective în memoria cache.

Tabela de pagini se poate organiza și pe 3 sau 4 niveluri, în funcție de dimensiunea memoriei virtuale și a celei fizice. Dacă tabela are o dimensiune mare, ea poate fi păstrată(cel puțin parțial) și pe un disc rapid.

**Memoria asociativă.** Indiferent de modul de organizare a tabelei de pagini, atunci când este referențiată o locație dintr-o pagină virtuală, în funcție de modul de organizare a tablelei, precum și de instrucțiunea cod mașină respectivă, trebuie să se facă una sau mai multe referințe la diverse locații de memorie. S-a observat că majoritatea programelor au un număr mare de referințe la un număr relativ mic de pagini. Astfel, un număr mic de intrări în tabela de paginare sunt efectiv citite. O soluție pentru rezolvarea acestei probleme, este utilizarea unei componente hardware care să proiecteze spațiul de adrese virtuale, în cel de adrese fizice, fără a trece prin tabela de pagini, numită memorie asociativă. Ea face parte din MMU și constă dintr-un număr mic de intrări, 8 în exemplul din figura 11, și care de regulă nu depășește 32.

Intrare validă	Pagină virtuală	Modificat	Protecție	Pagina fizică
1	140	1	RW	31
1	20	0	RX	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	RX	50
1	21	0	RX	45
1	860	1	RW	14
1	861	1	RW	75

Figura 11 Un exemplu de memorie asociativă

Fiecare intrare conține informații despre o pagină, cum ar fi, de exemplu, numărul de pagină virtuală, bitul de modificare, codul de protecție și numărul de pagină fizică unde se află încărcată pagina virtuală respectivă. Aceste câmpuri corespund cu cele corespunzătoare din tabela de pagini. De asemenea, mai există un câmp de un bit, care indică dacă intrarea este validă, adică este în curs de utilizare sau nu.

În exemplul prezentat, procesul este într-o buclă care traversează paginile 19, 20 și 21, deci aceste intrări sunt prezente în memoria asociativă, având setat codul de protecție pentru citire și execuție. Datele utilizate reprezintă un tablou care se află în paginile 129 și 130. Pagina 140 conține indicații folosite în calculele efectuate asupra tabloului. Stiva de program este conținută în paginile 860 și 861.

Cum funcționează memoria asociativă? Când o adresă virtuală este prezentată MMU pentru translatare, componenta hardware verifică mai întâi dacă numărul său de pagină virtuală este prezent în memoria asociativă, lucru care se realizează prin compararea lui cu cele din toate intrările din tabelă. Dacă se găsește o astfel de intrare și dacă accesarea acesteia nu violează bitii de protecție, pagina fizică corespunzătoare este luată direct din memoria asociativă, fără a mai fi necesară nici o referință la tabela de pagini. Dacă pagina virtuală este prezentă în memoria asociativă dar instrucțiunea respectivă violează drepturile de protecție (de exemplu să scrie într-o pagină setată numai pentru citire), este generată o eroare de protecție, fiind același lucru cu a nu găsi pagina respectivă în memoria asociativă. Când numărul de pagină virtuală nu este găsită în memoria asociativă, atunci MMU va detecta lipsa acesteia, va face o căutare în tabela de pagini, și va aduce pagina găsită într-o intrare din memoria asociativă, după ce, eventual a evacuat pagina existentă în intrarea respectivă.

**Alocare segmentată.** Din punctual de vedere al utilizatorului, o aplicație este formată dintr-un program principal și o mulțime de subprograme(funcții sau proceduri). Acestea folosesc diferite structuri de date (tablouri, stive etc), precum și o mulțime de simboluri(variabile locale sau globale). Toate aceste sunt identificate printr-un nume.

Pornind de la această divizare logică a unui program, s-a ajuns la o metoda de alocării segmentare a memoriei. Spre deosebire de metodele de alocare a memoriei bazate pe partitōnare, unde fiecărui proces trebuie să i se asigure un spațiu contigu de memorie, mecanismul de alocare segmentată, permite ca un proces să fie plasat în zone de program distincte, fiecare dintre ele conținând o entitate de program, numit segment. Segmentele pot fi definite explicit prin directive ale limbajului de programare sau implicit prin semantica programului. De exemplu, un compilator de Pascal poate crea segmente de cod pentru fiecare procedură sau funcție, segmente pentru variabilele globale, segmente pentru variabilele locale, precum și segmente de stivă. Deosebire esențială dintre alocarea paginată și cea segmentată este aceea că segmentele sunt de lungimi diferite.

În mod analog cu alocarea paginată, o adresă virtuală este o pereche  $(s, d)$ , unde  $s$  este numărul segmentului iar  $d$  este adresa din cadrul segmentului. Adresa reală (fizică) este o adresă obișnuită. Transformarea unei adrese virtuale într-o adresă fizică, se face pe baza unei tabele de segmente. Fiecare intrare în acesta tabelă este compusă dintr-o adresă de bază (adresa fizică unde este localizat segmentul în memorie) și lungimea segmentului (limită). În figura 12 este ilustrat modul de utilizare a tabelei de segmente.

Componenta  $s$  a adresei virtuale, este un indice în tabela de segmente. Valoarea deplasamentului  $d$  trebuie să fie cuprinsă între 0 și lungimea segmentului respectiv (în caz contrar este lansată o instrucțiune trap, care generează o intrerupere). Dacă valoarea  $d$  este validă, ea este adăugată adresei de început a segmentului și astfel se obține adresa fizică a locației respective.

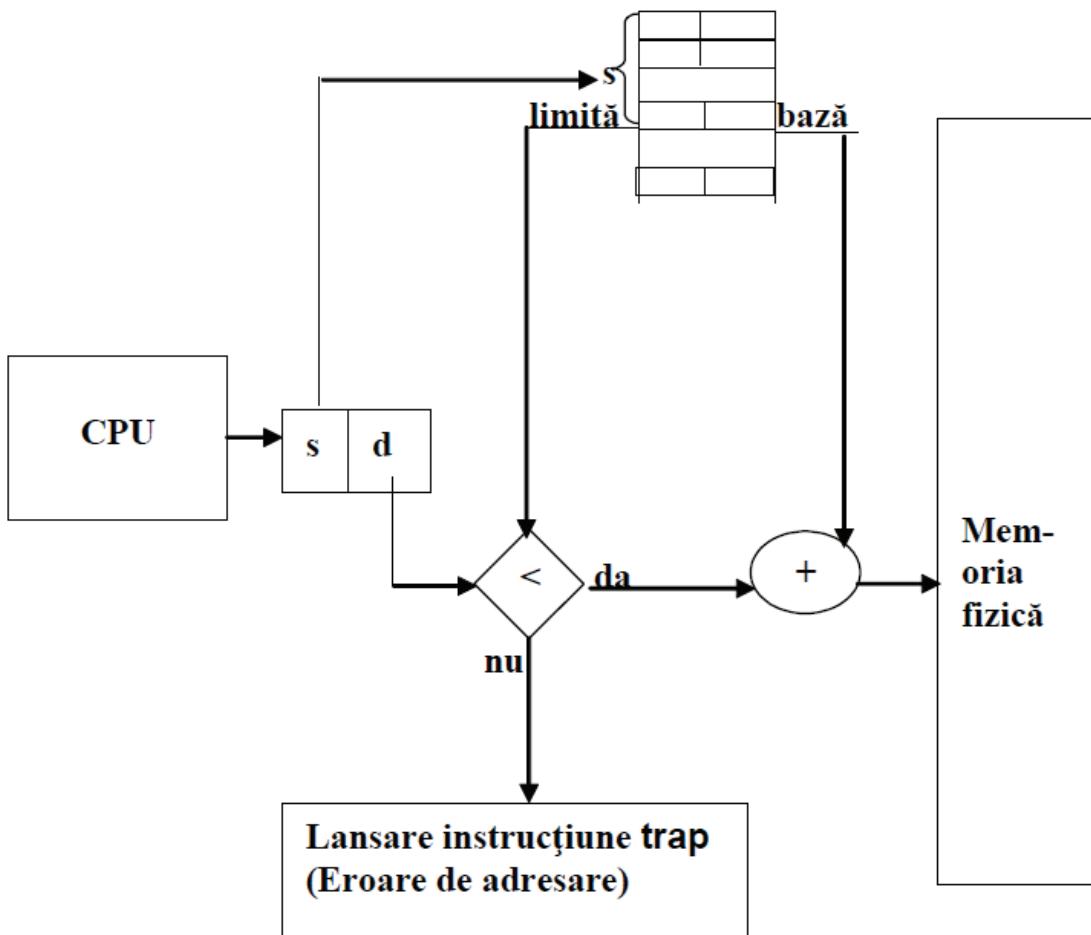


Figura 12. Translatarea adreselor segmentate

Avantajele alocării segmentate, față de cea bazată pe partitōnare sunt:

- se pot cere segmente reentrantă, care pot fi partajate de mai multe procese. Pentru aceasta este suficient ca toate procesele să aibă în tabelele lor aceeași adresă pentru segmentul respectiv.
- se poate realiza o bună protecție a memoriei. Fiecare segment în parte poate primi alte drepturi de acces, drepturi trecute în tabela de segmente. De exemplu, un segment care conține instrucțiuni cod mașină, poate fi declarat "read-only" sau executabil. La orice calcul de adresă, se verifică respectarea modului de acces al locațiilor din segmentul respectiv.

Alocarea segmentată cu paginare. Cele două metode de alocare a memoriei au avantajele și dezavantajele lor. În cazul segmentării poate să apară fenomenul de fragmentare. În cazul paginării, se efectuează o serie de operații suplimentare de adresare. De asemenea, unele procesoare folosesc alocarea paginată (Motorola 6800), iar altele folosesc segmentarea (Intel 80x86, Pentium). Ideea segmentării cu paginare, este aceea că alocarea spațiului pentru fiecare segment să se facă paginat. Această metodă este utilizată de sistemele de operare actuale.

Spațiul de adrese virtuale al fiecărui proces este împărțit în două partiții; prima partiție este utilizată numai de către procesul respectiv, pe când cealaltă partiție, este partajată împreună cu alte procese. Informațiile despre prima partiție, respectiv a doua partiție sunt păstrate în descriptorul tabelei locale (LDT – Local Descriptor Table), respectiv descriptorul tabelei globale (GDT – Global Descriptor Table). Fiecare intrare în LDT/GDT este reprezentată pe 8 octeți și conține informații despre un anumit segment, printre care adresa de început și lungimea aceluia segment.

Adresa virtuală este o pereche (selector, deplasament). Selectorul este reprezentat pe 16 biți și este un triplet (s, g, p) în care s este numărul de segment (13 biți), g specifică dacă segmentul este din LDT sau GDT și p specifică protecția. Deplasamentul este reprezentat pe 32 de biți și specifică adresa locală a unui octet în cadrul segmentului.

Calculatorul are 6 registre de segmente, ceea ce permite ca un proces să adreseze 6 segmente în orice moment. De asemenea are 6 registre pe 8 octeți, care păstrează descriptori pentru LDT sau GDT. Acest mod de organizare permite accesarea rapidă a entităților respective.

Translatarea unei adrese virtuale în adresă fizică se realizează astfel:

- registrul de segment conține un pointer către intrarea din LDT sau GDT corespunzătoare aceluia segment;
- de aici se iau adresa de început și deplasamentul;
- dacă adresa respectivă este validă, valoarea deplasamentului este adăugată la adresa de început și astfel se obține o adresă liniară, care apoi este translată într-o adresă fizică.

Așa cum am specificat mai devreme, fiecare segment este împărțit în pagini, fiecare pagină având o dimensiune de 4 Ko. Datorită dimensiunii posibile a unui segment, tabelele de pagini poate avea un număr mare de intrări (până la 1 000 000). Deoarece fiecare intrare are o lungime de 4 octeți, rezultă că o astfel de tabelă poate avea o dimensiune de 4 Mo, deci ea nu poate fi păstrată ca o listă liniară în memoria internă. Soluția este de a folosi o tabelă pe două niveluri. Mecanismul de translatare a adresei este similar cu cel utilizat în cazul alocării paginate a memoriei.

# Procese – apeluri sistem

Linux este unul din sistemele de operare multi-tasking. Procesele reprezintă o parte fundamentală a acestui sistem de operare, practic ele controlează toate activitățile care se desfășoară în sistem.

UNIX, sistemul initial din care a derivat Linux, definea un proces ca și “un spațiu de adrese cu unul sau mai multe fire de execuție care se execută în cadrul aceluiași spațiu de adrese și care solicită resurse sistem pentru acele fire de execuție.” Firele de execuție vor fi tratate în capitolele următoare.

Un sistem de operare multi-tasking este un sistem de operare care permite execuția simultană a mai multor procese. Fiecare instanță a unui program în execuție constituie un proces.

Ca și sistem de operare multi-user, Linux permite accesul simultan al mai multor utilizatori. Fiecare utilizator poate rula mai multe programe sau mai multe instanțe ale aceluiași program, toate la același moment de timp. Simultan sistemul de operare rulează alte programe de gestionare a resurselor sistemului și de control a accesului utilizatorilor la sistem.

Un program, sau un proces, care rulează este constituit din următoarele elemente: codul program, date, variabile, fișiere deschise și mediul de rulare. De obicei un sistem Linux va avea biblioteci și coduri sursă împărțite între mai multe procese, deci la un orice moment de timp pentru aceste resurse utilizate în comun va exista o singură copie în memoria sistem.

## 1. Mecanismul de identificare al procesului - PID

Fiecare proces ce rulează sub Linux are asociat un identificator de proces unic numit PID (Process Identity Numbers) prin care este cunoscut în sistem. Fiecare proces din sistem începând cu procesul 1, numit init, are un părinte a cărui PID îl este cunoscut și îl poate accesa.

Fiecare proces aparține unui grup de procese, fiecare grup având la rândul lui un număr de identificare care este de fapt identificatorul de proces(PID-ul) procesului părinte al grupului.

Când se pune problema drepturilor de acces a proceselor la diverse fișiere se folosește un set de patru numere de identificare. Aceste sunt cunoscute sub numele de utilizatorul real și identificatorul de grup, și utilizatorul efectiv și identificatorul lui de grup. Identificatorul real al unui proces este de fapt UID(user ID) și GID(group ID) ale utilizatorului care rulează procesul. Identificatorul efectiv este de obicei același cu cel al utilizatorului real cu excepția cazului în care sunt setate opțiunile setuid și setgid. Dacă una sau amândouă opțiunile sunt activate atunci UID sau GID efectiv al procesului vor primi valorile UID sau GID asociate fișierului din care rulează procesul.

Să presupunem că avem un utilizator cu UID 200 și GID 20 care dorește să ruleze programul /usr/bin/passwd. Acest program are UID 0 (root) și GID 1 și are de asemenea set bitul setuid. Când este rulat programul procesul asociat va avea ID-ul utilizatorului real 200, ID-ul grupului real la 20, ID-ul utilizatorului efectiv 0, iar ID-ul grupului efectiv 20.

ID-ul real este folosit pentru a identifica utilizatorul pentru care este rulat un proces. ID-urile efective sunt folosite pentru a putea realiza o sortare a permisiunilor și privilegiilor pe care procesele le au la accesarea unui fișier. Acest lucru se realizează pe baza următorului algoritm:

- Dacă ID-ul utilizatorului efectiv al procesului este 0 (root) atunci procesului nu i se aplică nici o restricție, în caz contrar treci la pasul următor.

- Dacă ID-ul utilizatorului efectiv al procesului este același cu ID-ul utilizator al fișierului atunci procesului i se acordă accesul la fișier în conformitate cu drepturile de acces ale proprietarului aceluia fișier. Dacă nu treci la pasul următor.
- Dacă ID-ul grupului efectiv al procesului este același cu ID-ul de grup al fișierului atunci procesului i se acordă accesul la fișier în conformitate cu drepturile de acces ale grupului fișierului. Dacă nu treci la pasul următor.
- Accesul la fișier este acordat pe baza drepturilor de acces ale celorlalți utilizatori (others).

Toate aceste numere de identificare se pot obține cu ajutorul următoarelor funcții:

```
uid_t getuid(void)      // obține ID-ul utilizatorului real
uid_t getgid(void)      // obține ID-ul grupului real
uid_t geteuid(void)      // obține ID-ul utilizatorului efectiv
uid_t getegid(void)      // obține ID-ul grupului efectiv
uid_t getpid(void)      // obține ID-ul procesului
uid_t getppid(void)      // obține ID-ul procesului părinte
uid_t getpgrp(void)// obține ID-ul de grup al procesului
```

## 2. Creearea proceselor utilizând fork

Vom considera că un proces este construit din trei părți separate conform figurii următoare(Fig. 1):

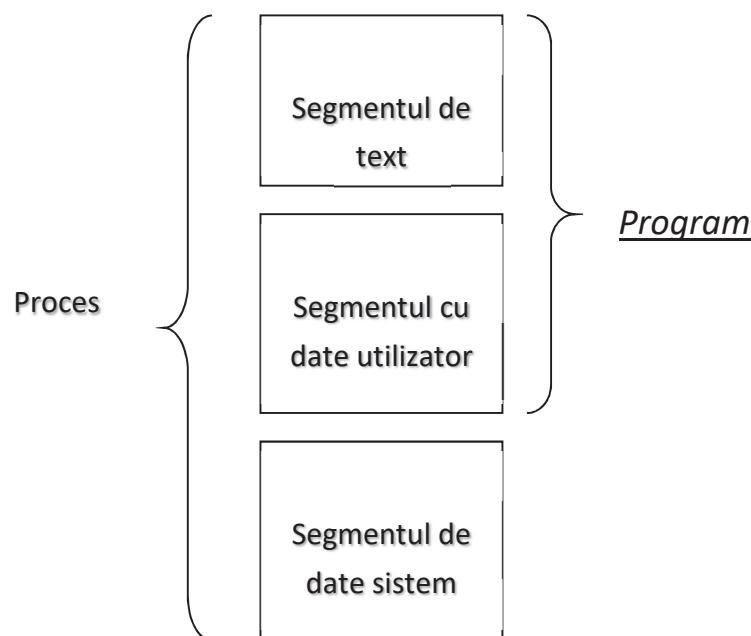


Fig.1 Structura proces

Segmentul de text conține instrucțiunile în cod mașină care urmează să fie executate. Acest segment poate fi accesat numai pentru citire(deci codul sursă din această parte nu poate fi modificat), ceea ce permite utilizarea acestui segment de către două sau mai multe proceze din sistem care

rulează același program. De exemplu dacă mai mulți utilizatori rulează aceeași versiune de interpreter de comenzi (de ex. bash), în memoria principală va fi o singură copie a programului pe care o folosesc în comun toți utilizatorii.

Segmentul de date utilizator conține toate datele cu care operează un proces pe parcursul execuției sale, aceste date includ și variabilele ce vor fi utilizate de proces. Datele din acest segment sunt modificabile și în plus fiecare proces are câte un segment de date unic.

Segmentul de date sistem conține elementele caracteristice mediului de lucru în care rulează un program. Acest segment realizează diferența între programe și procese. Programul are un caracter static, este localizat pe disc, și este constituit dintr-un set de instrucțiuni și date folosite pentru inițializarea segmentelor text și de date ale proceselor. Procesul are un caracter dinamic iar execuția sa necesită interacțiunea între segmentele de date, text și sistem.

Sub Linux, pentru crearea de noi procese se folosește apelul sistem fork:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Practic acest apel realizează un nou proces, numit proces fiu, iar procesul care a apelat fork devine proces părinte al noului proces creat. Cele două procese sunt identice în termeni de conținut al segmentelor lor de date și text și aproape identice în ceea ce privește segmentele sistem. Singurele diferențe între astfel de procese apar la nivelul unor atrbute care trebuie să fie diferite (cum ar fi PID care trebuie să fie unic pentru fiecare proces în parte). Odată ce procesul fiu a fost creat, ambele procese, părinte și fiu, își continuă execuția din interiorul apelului fork. Adică următoarea acțiune pentru fiecare proces în parte este să părăsească fork, fiecare returnând o valoare diferită (Fig.2).

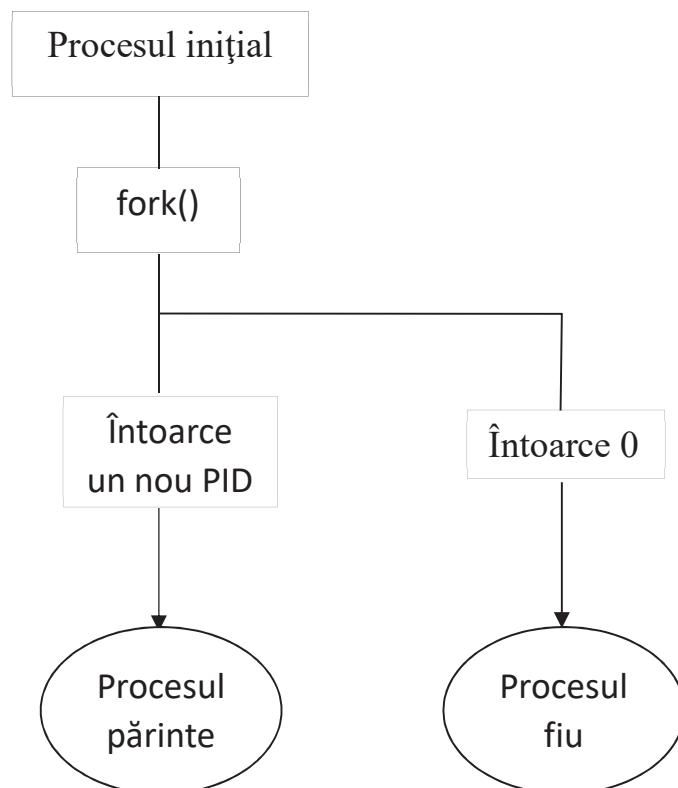


Fig. 2 Fork

Două procese virtual identice care rulează simultan se pot dovedi utile numai în situația în care acestea realizează acțiuni diferite. În această situație este nevoie de un mecanism prin care să se realizeze diferențierea între cele două procese. Acest lucru se realizează relativ simplu prin faptul că fork returnează valori diferite celor două procese. Astfel procesului părinte îi returnează PID-ul noului proces fiu creat, în timp ce procesului fiu îi returnează 0. În mod normal alocarea ID-urilor pentru procese începe cu 1 pentru procesul init. Vom avea în continuare un exemplu de program care creează un nou proces:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main(){
    pid_t val;
    printf("PID înainte de fork: %d\n", (int)getpid());
    if(val=fork())
        printf("PID părinte: %d\n", (int)getpid());
    else
        printf("PID fiu: %d\n", (int)getpid());
}
```

La rularea programului se vor genera trei linii de ieșire. Prima va afișa valoarea PID-ului procesului înainte de apelul fork, a doua valoarea PID a procesului părinte după apelul fork, iar a treia va afișa valoarea PID a procesului fiu. Un exemplu de ieșire obținut la rularea programului:

PID înainte de fork: 16953

PID părinte: 16953

PID fiu: 16954

După execuția apelului fork majoritatea atributelor procesului părinte sunt disponibile nemodificate procesului fiu. Principalele atrbute sunt:

- apartenența la grupul de procese.
- terminalul în care rulează (dacă acesta există).
- UID și GID reale și efective.
- directorul de lucru curent.
- masca pentru crearea fișierelor (umask).

În plus, toți descriptorii de fișiere, care în procesul părinte sunt asociați cu fișiere deschise vor fi duplicați în procesul fiu. Adică atât descriptorii de fișiere ai procesului fiu cât și ai procesului părinte vor indica spre aceeași descriptori de fișiere deschise.

### 3. Apelul sistem exec

Dacă se utilizează interpretorul de comenzi pentru a rula o comandă (ex. ls) pentru rularea ei acesta va apela la un anumit moment fork. Se pune problema cum se poate ca în procesul fiu în loc de o copie a interpretorului de comenzi să fie rulat ls.

Soluția în acest caz este utilizarea apelului sistem exec. De fapt sunt mai multe versiuni ale acestui apel sistem dar toate realizează în esență același lucru. Acest apel sistem reprezintă de fapt modalitatea de rulare de programe sub Linux. Acest lucru îl realizează prin înlocuirea segmentelor de date și text ale procesului care apelează exec cu cele ale programului a cărui nume a fost transmis ca și argument. O altă problemă ce ar putea apărea este faptul că majoritatea programelor ce sunt rulate în mod curent din linia de comandă primesc și parametri sau argumente, deci apelul exec trebuie să acopere și aceste aspecte. Acest lucru se poate realiza în C prin argumentele argc și argv, deci și exec trebuie să îl transmită programului într-o formă asemănătoare.

Cea mai simplă versiune de exec este execl. Prototipul acestei funcții este:

```
#include <unistd.h>
int execl( const char *path, const char *arg, ...);
```

unde path este calea completă către programul care urmează să fie executat, acesta este urmat de o listă variabilă de argumente ce trebuie transmise programului. Această listă trebuie terminată cu NULL(0). În continuare vom avea un exemplu de program care să ruleze comanda ls -l:

```
#include <stdio.h>
#include <unistd.h>

main(){
    execl("/bin/ls","ls","-l",0);
    printf("terminat cu erori.");
}
```

Primul parametru pentru execl este calea completă pentru programul ce urmează a fi rulat, acesta este fișierul care va fi rulat, de aici se obțin și drepturile de acces la fișier(dacă are sau nu drept de execuție) restul parametrilor sunt folosiți pentru lista de argumente argv. În exemplu ls va fi argv[0], iar -l argv[1].

Din apelurile exec nu se revine. Deci mesajul terminat cu erori din exemplu nu va fi afișat. Singura excepție este cazul în care a apărut o eroare la apelul exec în această situație se revine la vechiul proces și se returnează și codul de eroare (în această situație va fi afișat mesajul terminat cu erori).

În plus pentru a face toți acești parametri disponibili noului program apelurile exec transmit o valoare și pentru variabila:

```
extern char **environ;
```

Această variabilă are același format ca și variabila argv diferența fiind că prin variabila environ se transmit valori ale variabilelor aparținând mediului de lucru ale procesului original.

Acest apel sistem mai conține și alte versiuni singurele diferențe constând în modul de apelare, funcțional toate se comportă la fel:

```
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ..., 
char* const envp[]);
int execv( const char *path, char *const argv[]);
```

```
int execvp( const char *file, char *const argv[] );
int execve (const char *filename, char *const argv []
            ,char *const envp[] );
```

În cazul ultimelor trei apeluri argumentele ce trebuie transmise programelor sunt organizate sub forma unei liste.

Ca și în cazul apelului fork majoritatea atributelor sunt păstrate în urma unui apel exec. Acest lucru se datorează faptului că segmentul sistem rămâne intact în urma unui apel exec. Cel mai important aspect este acela că descriptorii de fișiere asociați cu descriptorii de fișiere deschise rămân valabile și după apelul lui exec. Singura excepție este un marcat (flag) asociat descriptorilor de fișiere (nu este asociat descriptorilor de fișiere deschise) numit close on exec. Dacă acest flag nu este activat atunci descriptorul de fișier respectiv este valabil și după apelul exec. Modificarea acestui flag se poate face prin intermediul apelului sistem fcntl.

#### 4. Apelurile sistem wait și exit

Pe parcursul rulării procesului fiu procesul părinte fie așteaptă ca acesta să își termine execuția, fie își continuă execuția. În cazul interpretorului de comenzi de exemplu, alegerea rămâne la latitudinea utilizatorului. În mod normal interpretorul de comenzi așteaptă, pentru fiecare comandă introdusă din linia de comenzi, până la terminarea execuției acesteia. Dar utilizatorul are posibilitatea indică interpretorului de comenzi să nu mai aștepte terminarea comenzi și să își reia execuția imediat prin adăugarea caracterului '&' la sfârșitul comenzi.

Pentru ca procesul părinte să aștepte terminarea execuției unui proces fiu, procesul părinte trebuie să execute apelul sistem wait(). Prototipul acestui apel sistem este următorul:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status)
```

Apelul sistem wait returnează PID-ul procesului fiu ce și-a încheiat execuția. Parametrul care îl primește apelul sistem wait este o referință către o locație care va primi valoarea stării de terminare a procesului fiu.

Când un proces și-a terminat execuția el va executa apelul sistem exit fie în mod explicit fie indirect prin intermediul unei biblioteci. Prototipul apelului sistem exit este următorul:

```
#include <stdlib.h>
void exit(int status)
```

Acest apel sistem primește ca și argument un parametru de stare care va fi transmis procesului părinte prin intermediul unei locații referită de parametrul apelului wait. Apelul sistem wait poate întoarce mai multe informații referitoare la starea procesului fiu la terminare prin intermediul valorii de stare. Pentru extragerea acestor informații se poate folosi un macro numit WEXITSTATUS. Ex:

```
#include <sys/wait.h>
int statval, exstat;
pid_t pid;
pid = wait(&statval);
```

```
exstat = WEXITSTATUS(statval);
```

Schema bloc care descrie funcționarea a două procese, unul părinte care așteaptă terminarea execuției procesului fiu și a procesului fiu care folosește apelul sistem exec și a cărui execuție se termină cu apelul exit, arată astfel (Fig. 3):

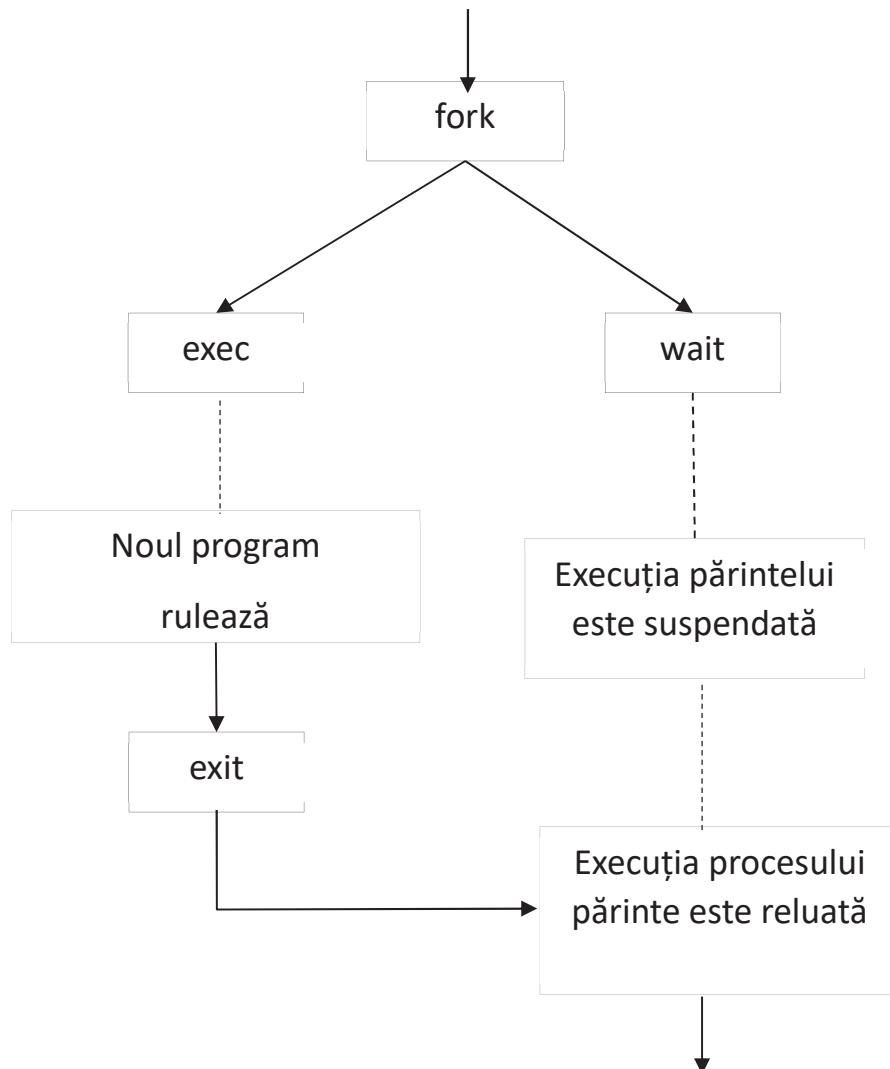


Fig. 3 Proces părinte în aşteptare

O altă versiune a apelului sistem wait este waitpid cu următorul prototip:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

Prin parametrul options oferă facilități suplimentare. De exemplu, în urma unui apel waitpid procesul părinte nu își suspendă execuția dacă nici un proces fiu nu are disponibilă starea de ieșire. Options poate fi o operație de tip SAU între următoarele constante:

- WNOHANG – apelul nu este blocant.
- WUNTRACED – preia și starea proceselor fiu trecute în stare stop.

- WCONTINUED - preia și starea proceselor fiu trecute în starea continue.

Pentru parametrul pid avem următoarele posibilități:

- < -1 așteaptă încheierea execuției oricărui proces fiu a cărui ID de grup este egal cu pid.
- -1 comportament identic cu pid.
- 0 așteaptă încheierea execuției oricărui proces fiu a cărui ID de grup este egal cu ID-ul de grup al procesului.
- > 0 așteaptă încheierea execuției procesului fiu a cărui PID este egal cu pid.

Există posibilitatea ca un proces fiu să își încheie execuția înainte ca procesul părinte să apeleze wait. În această situație procesul fiu va intra într-o stare numită zombie, în care toate resursele ocupate de procesul fiu au fost eliberate cu excepția structurii de date care conține starea sa de ieșire. Când procesul părinte efectuează apelul wait, starea de ieșire este transmisă procesului părinte iar resursele ocupate cu structura de date a procesului fiu pot fi eliberate.

# Procese – comunicare

Fiecare proces operează în propriul său spațiu de adrese virtuale, iar de evitarea interferențelor dintre procese se ocupă sistemul de operare. Implicit un proces nu poate comunica cu un alt proces numai dacă face uz de diverse mecanisme de comunicare gestionate de kernel(nucleul sistemului de operare). Există totuși diverse situații în care procesele trebuie să coopereze, să împartă resurse comune sau să își sincronizeze acțiunile. Există mai multe metode de comunicare între procese:

- fișiere – este cel mai simplu mecanism de comunicare interproces. Un proces scrie într-un fișier iar celălalt proces citește din fișier.
- semnale – sunt utilizate pentru a semnaliza asincron evenimente între procese
- pipes – un pipe conectează ieșirea standard a unui proces cu intrarea standard a altuia.
- cozi de mesaje – este o listă legată de spațiul de adresare a kernel-ului
- semafoare – sunt contoare utilizate pentru a gestiona accesul la resursele utilizat în comun de mai multe procese. Cel mai des sunt folosite ca și mecanism de blocare a unei resurse pentru a preveni accesul simultan a mai multor procese la o anumită resursă.
- Sockets – permit realizarea de conexiuni între procese locale sau în rețea.

## 1. Semnale

Una dintre cele mai vechi metode de comunicare între procese în sistemele de tip UNIX(Linux) se bazează pe semnale. O caracteristică importantă a acestor semnale este că ele sunt asincrone, adică un proces poate primi un semnal la orice moment și trebuie să fie pregătit să îi răspundă. Dacă un semnal survine pe perioada execuției unui apel sistem, acesta se încheie prin returnarea unui cod de eroare(EINTR) și este sarcina procesului să refacă apelul întrerupt. Există mai multe tipuri de semnale și fiecare putând fi accesat utilizând pe baza unui nume simbolic. Fiecare nume unic reprezintă o abreviere ce începe cu SIG (de ex. SIGINIT). O listă completă a setului de semnale din sistem se poate obține prin intermediul comenzi kill -l. În continuare se prezintă o listă cu numărul și numele simbolic a semnalelor folosite în mod ușual:

1) SIGHUP – terminal închis (oprește procesul).			
2) SIGINT – întrerupere de la tastatură (oprește procesul).			
3) SIGQUIT – oprește procesul (oprește procesul și creează un fișier core).			
4) SIGILL – acțiune ilegală (oprește procesul și creează un fișier core).			
5) SIGTRAP	6) SIGIOT	7) SIGBUS	8) SIGFPE
9) SIGKILL – termină procesul.			
10) SIGUSR1	11) SIGSEGV	12) SIGUSR2	
13) SIGPIPE	14) SIGALRM	15) SIGTERM	
17) SIGCHLD – procesul fiu opriș sau terminat			
18) SIGCONT	19) SIGSTOP	20) SIGTSTP	21) SIGTTIN
22) SIGTTOUT	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO
30) SIGPWR			

Fiecare tip de semnal are asociată o acțiune ce va fi executată de nucleul sistemului de operare (kernel) asupra procesului când procesul primește semnalul respectiv. Implicit un proces poate trata un semnal în următoarele moduri:

- Își încheie execuția
- Ignoră semnalul – două semnale nu pot fi ignorate: SIGKILL – care termină execuția procesului și SIGSTOP care îi oprește execuția.
- Reface acțiunea implicită a semnalului.

La primirea unui semnal un proces poate alege modul de tratare a acestuia, o alternativă este să rămână la acțiunea implicită, iar tratarea va fi făcută de kernel. De exemplu semnalul SIGSTOP va modifica starea curentă a procesului care îl primește în STOPPED și apoi va solicita kernelului(schedulerului) sau ruleze un alt proces. Alternativ, procesul poate specifica pentru un anumit semnal propria rutină de tratare. Această rutină va fi apelată de fiecare dată când este generat semnalul respectiv, iar adresa acestei este reținută în structura sigaction. Apelul acestei rutine va fi făcut de către kernel, deci procesul va rula în mod kernel pe parcursul tratării semnalului.

Nu orice proces din sistem poate trimite semnal oricărui alt proces, acest lucru este valabil numai pentru procesele ce aparțin kernelului sau superutilizatorului (root). Procesele normale pot trimite semnale numai proceselor cu același UID și GID, sau proceselor din același grup.

Generarea unui semnal se realizează prin setarea bit-ului corespunzător din structura task\_struct. Dacă procesul nu a blocat semnalul și se află în aşteptare dar este întreruptibil atunci starea acestuia este modificată în Running, iar procesul este trecut în coada de execuție. În acest fel procesul va fi luat în calcul între procesele care vor primi timp de procesor.

Semnalele nu sunt transmise proceselor imediat ce sunt generate, ele trebuie să aștepte până în momentul în care procesul rulează din nou. De fiecare dată când un proces părăsește un apel sistem sunt verificate câmpurile sale signal și blocked, iar dacă pentru respectivul proces sunt semnale neblocate ele pot fi acum transmise procesului.

La nivel de interpretor de comenzi o modalitate de trimitere a semnalelor către procese este prin intermediul comenzi kill. Un exemplu clasic de terminare forțată a execuției unui proces este următorul:

```
ps | grep processname  
kill -9 PID
```

În cele ce urmează se va face o scurtă prezentare a celor mai importante apeuri sistem folosite pentru lucru cu semnale.

*Apelul sistem signal:*

În Linux sunt disponibile mai multe apeuri sistem ce pot fi utilizate pentru gestionarea semnalelor. Apelul sistem signal poate fi întâlnit și la celealte versiuni de UNIX. Prototipul acestui apel sistem este următorul:

```
#include <signal.h>  
void(*signal(int signum, void (*sighandler) (int)))  
(int);
```

La prima vedere acest prototip pare complicat dar la o analiză mai atentă se poate observa că sunt necesari doar doi parametrii:

- signum – un număr întreg care indică semnalul a cărui tratare va fi rescrisă.

- handler – o referință către o funcție care va fi folosită ca rutină de tratare pentru semnalul respectiv.

Valoarea returnată de signal() este o referință către o funcție care primește ca și parametru un singur întreg și nu returnează nimic (void). Un exemplu de utilizare a acestui apel sistem:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void handler(int sig) {
    printf("CTRL-C\n");
    (void) signal(SIGINT, handler);
}

void main(void) {
    (void) signal(SIGINT, handler);
    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

Acest segment de cod afișează la terminal, în buclă infinită, mesajul Hello World!. La orice tentativă de a opri programul de la tastatură prin secvența CTRL-C semnalul SIGINT va fi prins de program, ca rutină de tratare a semnalului va fi folosită funcția handler.

Legătura între semnal (SIGINT) și rutina de tratare (handler) se realizează la apelul din funcția main. După realizarea acestei legături orice semnal SIGINT primit de proces va determina execuția rutinei de tratare (handler). Această funcție doar afișează mesajul CTRL-C, totodată refac legătura între semnal și funcție. Această refacere a legăturii între semnal și rutina de tratare este necesară datorită faptului că după primirea unui semnal se revine la tratarea implicită (terminarea execuției programului), ceea ce nu este de dorit.

Pentru acest tip de apel sistem apare o problemă legată de un scurt interval de timp în care tratarea semnalului se face în mod implicit, și dacă pe parcursul acestui interval de timp survine un semnal acesta va fi tratat în mod implicit, adică va determina încheierea execuției programului.

#### *Apelul sistem sigaction:*

Acest apel sistem este conform standardelor POSIX, este mai complex, dar înălătură problema apărută anterior. Conform acestor specificații fiecare proces are asociată câte o mască, care precizează setul de semnale care nu pot fi livrate procesului la momentul curent. Dacă totuși procesul primește un semnal ce aparține acestei liste el va fi pus într-o coadă de așteptare și va fi transmis procesului atunci când blocajul este înălăturat.

Când un semnal este generat, el este automat adăugat la masca de semnale a procesului țintă, astfel încât orice instantă viitoare a acestui semnal va fi blocată până la servirea semnalului curent.

Prototipul pentru acest apel sistem este următorul:

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

După cum se poate observa referința către rutina de tratare a fost introdusă în această structură. Totodată structura mai conține și un câmp sa\_mask acesta funcționează ca și o mască de semnale suplimentară, și este valabilă pe parcursul execuției rutinei de tratare a semnalului. Câmpul sa\_flags se interpretează la nivel de bit. Dintre aceste flag-uri cele mai semnificative sunt:

- SA\_ONESHOT – semnalul va fi tratat în mod implicit.
- SA\_NOMASK - ignoră câmpul sa\_mask din structura sigaction.

Ambele flag-uri vor fi active dacă pentru se folosește apelul signal în loc de sigaction. Pentru obținerea de informații adiționale s-a introdus un parametru suplimentar siginfo\_t.

Spre deosebire de signal, valoarea returnată de sigaction nu include o referință către vechea rutină de tratare. În schimb primește un nou parametru. Acesta este o altă referință către o structură de tipul sigaction, iar completarea acestei structuri o va face apelul sigaction() bazându-se pe vechea structură sigaction.

*Apelul sistem kill:*

În mareea majoritate a cazurilor mai importante semnalele sunt generate de apariția unor evenimente: erori hardware, modificarea stării unui proces sau intervenția utilizatorului de la consolă. În afară de aceste situații există posibilitatea ca un proces să trimită semnale în mod deliberat unui alt proces, utilizând apelul sistem kill, dacă are permisiunile corespunzătoare. Prototipul acestui apel sistem este următorul:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Unde prin sig se precizează semnalul care va fi trimis, iar pid are următoarele sensuri:

- pid > 0 – semnalul sig va fi trimis procesului pentru care PID = pid.
- pid = 0 – semnalul sig va fi trimis tuturor proceselor cu același PID ca și procesul ce a făcut apelul.
- pid = -1 – semnalul sig va fi trimis tuturor proceselor din sistem cu excepția procesului init și a procesului ce a făcut apelul.
- pid < -1 – trimite semnalul sig tuturor proceselor pentru care GID=pid.

*Apelul sistem alarm:*

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Fiecare proces are asociat un ceas pe care îl poate folosi pentru a-și trimite semnale de tipul SIGALRM după depășirea unui anumit interval de timp. Apelul sistem alarm() primește un singur parametru care este intervalul de timp, în secunde, după care se va genera semnalul.

## 2. Pipes

În Linux, majoritatea intrepretoarelor de comenzi acceptă redirectarea. Un exemplu clasic în acest sens este secvența de comenzi:

```
$ cat /etc/passwd | wc -l
```

Rezultatul acestei secvențe de comenzi este că ieșirea standard a comenzi cat devine intrare standard pentru comanda wc. Deci un pipe este o metodă prin care se poate realiza conectarea ieșirii standard a unui proces cu intrarea standard altuia. Pipe-urile sunt una dintre cele mai vechi metode utilizate la comunicarea între proceze. Comunicația realizată între două proceze se poate realiza într-o singură direcție, de unde și denumirea de unidirectionale (half-duplex). Nici unul dintre procezele implicate nu sunt conștiente de aceste redirectări și este sarcina intrepretorului de comenzi să gestioneze aceste pipe-uri.

Când un proces creează un pipe, nucleul (kernelul) va genera doi descriptori de fișiere care urmează să fie utilizati împreună cu pipe-ul. Unul dintre descriptori este folosit pentru a crea o cale de intrare (scriere) în pipe, iar celălalt pentru a crea o cale de ieșire. În acest punct utilizarea unui pipe nu are o utilitate practică prea mare atât vreme cât procesul poate comunica numai cu el însuși. O reprezentare a procesului și a kernelului după ce a fost creat un pipe:

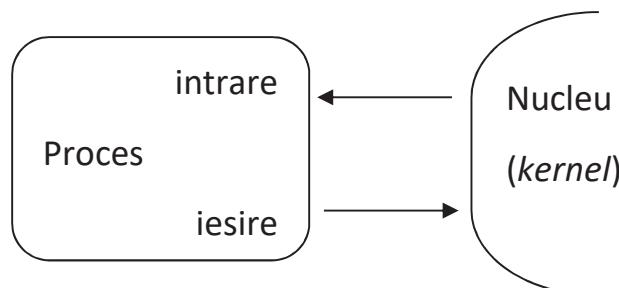


Fig. 1 Comunicare proces – nucleu

Din diagrama anterioară se poate observa modul de conectare a descriptorilor de fișiere. Un proces poate trimite date (scrie) prin fd0, și le poate obține prin fd1. La nivel de pipe transferul datelor se face în interiorul kernelui, iar pipe-ul este reprezentat prin intermediul unui inode, care la rândul lui aparține tot kernel-ului și nu este legat de nici un sistem fizic de fișiere. Din ceea ce s-a prezentat până acum utilizarea pipe-urilor nu ar avea nici un sens deoarece în această situație un pipe ar fi folosit doar pentru comunicarea în cadrul unuia și același proces. Dar situația se schimbă dacă după crearea pipe-ului are loc și crearea unui proces prin intermediul apelului sistem fork. În această situație procesul părinte va moșteni toți descriptorii de fișiere deschisi pentru procesul părinte, și bazându-ne pe acest mecanism avem bazele comunicării intreproces (mai precis între proceze aflate în relația părinte-fiu).

Ceea ce s-a prezentat până acum a fost doar o prezentare de ansamblu, iar exemplul a fost la nivelul interpretorului de comenzi. Pentru a utiliza facilitățile oferite de pipe-uri în cadrul limbajului de programare C.

*Apelul sistem pipe:*

Este utilizat pentru crearea unui fișier de tip pipe.

```
#include<unistd.h>
int pipe(int pfd[2]);
```

Întoarce 0 în caz de succes și -1 în caz de eroare.

Argumentul pfd este un tablou cu două elemente care după execuția funcției va conține:

- Descriptorul de fișier pentru citire – pfd[0];
- Descriptorul de fișier pentru scriere – pfd[1];

Rolul acestui apel sistem este de a crea o cale de comunicație, de citire și scriere, între două procese. Această cale de comunicație se bazează pe cei doi descriptori de fișier care sunt returnați, de apelul sistem pipe, în tabloul pfd. Astfel din pfd[0] se citesc datele, iar în pfd[1] se scriu datele.

Așa cum s-a mai precizat comunicarea prin intermediul pipe-urilor este specifică comunicării între procese, deci în conjuncție cu apelul sistem fork. Un exemplu clasic este situația în care se dorește ca procesul părinte să transmită un mesaj procesului fiu. Programul C corespunzător va realiza următorii pași:

- Apelul sistem pipe – creează descriptorii de citire/scriere.
- Apelul sistem fork – se creează procesele părinte și fiu, cei doi descriptori se vor regăsi în ambele procese.
- Procesul părinte va închide pfd[0] descriptorul de citire.
- Procesul fiu va închide pfd[1] descriptorul de scriere.
- Se realizează transferul mesajului prin intermediul descriptorilor de fișier rămași deschiși.

Închiderea descriptorilor neutilizați este necesară din cauză că descriptorii de fișiere sunt aceeași atât în procesul părinte cât și în procesul fiu, situație datorată apelului sistem fork. Programul C corespunzător este prezentat în cele ce urmează.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

int main(void) {

    int      pfd[2], nr_bytes, status;
    pid_t    pid_fiu;
    char    mesaj[] = "Hello world!";
    char    readbuffer[80];
```

```

pipe(pfd);

if((pid_fiu = fork()) == -1)
{
    perror("fork");
    _exit(1);
}

if(pid_fiu == 0)
{
    //procesul fiu închide descriptorul de scriere
    close(pfd[1]);

    //citește mesajul din pipe
    nr_bytes=read(pfd[0],readbuffer,sizeof(readbuffer));

    //și îl afisează
    printf("Received string: %s\n", readbuffer);

}else{

    //procesul părinte închide descriptorul de citire
    close(pfd[0]);

    //trimite mesajul prin descriptorul de scriere
    write(pfd[1], mesaj, (strlen(mesaj) + 1));
    wait(status);
    _exit(0);
}
return(0);
}

```

*Apelul sistem dup și dup2:*

Este utilizat pentru a duplica un descriptor de fișier.

```
#include <unistd.h>
int dup(int fd);
int dup2(int fd, int nfd);
```

Returnează noul descriptor în caz de succes, sau -1 în caz de eroare. Iar cu dup2 vechiul descriptor este închis.

Acet apel sistem creează un nou descriptor de fișier pe lângă cel existent, descriptorul (numărul) returnat este cel mai mic disponibil. O utilizare imediată a acestui apel sistem ar fi posibilitatea redirectării intrării/ieșirii standard.

Apelul dup – deși vechiul și noul descriptor pot fi utilizate pe rând, de obicei una dintre căile de comunicare standard sunt închise.

Dacă vom considera:

```

...
pid_fiu = fork();

if(pid_fiu == 0)
{
    //închide intrarea standard a procesului fiu
    close(0);

    //duplică partea de intrare a pipe-ului cu intrarea
    standard
    dup(pfd[0]);
    execlp("sort", "sort", NULL);
    ...
}

```

Deoarece descriptorul de fișier 0 (intrarea standard) a fost închis, apelul dup forțează descriptorul de intrare a pipe-ului ca intrare standard, apoi se realizează apelul sistem execlp() care suprascrie segmentul de cod a procesului fiu cu programul de sortare. Pentru că noul program executat moștenește căile de comunicare ale procesului din care a fost creat, el moștenește de fapt, ca și cale de intrare standard descriptorul de citire din pipe. Astfel tot ce trimite procesul părinte inițial prin pipe ajunge la programul de sortare.

Apelul dup2 – acest apel include, atât apelul de duplicare a căii de comunicare standard cât și operația de închidere a căii de comunicare standard, într-un singur apel sistem. În plus acest apel sistem este garantat a fi atomic, adică nu va fi acceptată nici o cerere de întrerupere (semnal) din partea sistemului pe parcursul execuției acestui apel. În cazul apelului sistem dup() trebuia realizată operația close înainte. Adică aveam două apeluri sistem, ceea ce crea un anumit grad de vulnerabilitate, dat de intervalul dintre cele două apeluri. Dacă sosește un semnal între cele două apeluri operația de duplicare nu mai are loc.

În acest caz vom avea:

```

...
pid_fiu = fork();

if(pid_fiu == 0)
{
    //închide și duplică intrarea standard
    dup2(0, pfd[0]);

```

```
    execlp("sort", "sort", NULL);  
    ...  
}
```

# Fire de execuție

## 1. Prezentare generală

O caracteristică esențială a unui sistem de operare de tip UNIX/Linux este execuția concurrentă a mai multor procese, la fel ca și procesele, firele de execuție par a rula concurrent, responsabil de această “concurență” este nucleul, kernel-ul, sistemului de operare care întrerupe rularea thread-urilor la anumite intervale de timp oferind și altora șansa de a fi rulate.

Conceptual, thread-ul este o “subdiviziune”, el există doar în cadrul unui proces. La execuția unui program, sistemul de operare creează un nou proces și în cadrul procesului respectiv va crea un singur thread în care programul apelat se execută secvențial, optional, din thread-ul inițial se pot crea thread-uri suplimentare, ansamblul acestor thread-uri reprezentând programul apelat, fiecare thread executând părți diferite ale programului la momente diferite de timp.

Dacă în cazul creării unui nou proces, procesul fiu execută programul procesului părinte dar cu resursele părintelui, memorie virtuală, descriptori de fișiere, etc., copiate procesul fiu putând modifica oricare resursă fără a afecta procesul părinte, și viceversa, la crearea unui thread nimic nu este copiat. Ambele thread-uri vor utiliza resursele sistem în comun, dacă un thread închide un descriptor de fișier, celelalte thread-uri nu vor mai putea efectua operații asupra aceluiași descriptor de fișier. De asemenea datorită faptului că un proces, respectiv thread-urile asociate unui proces, poate executa doar un singur program la un moment dat, dacă un thread apelează una dintre funcțiile exec execuția celorlalte thread-uri este întreruptă.

Comutarea execuției între două procese implică un număr relativ mare de operații. Aceste operații sunt necesare pentru a se asigura faptul că nu apar interferențe între diferențele resurse ale unor procese diferite, această condiție fiind necesară în cadrul unui sistem multiutilizator. În situația în care se dorește ca două sau mai multe procese să coopereze, pentru atingerea unui scop comun, acest obiectiv se poate realiza prin utilizarea mecanismelor de comunicare interproces. Totuși și în cazul utilizării acestor mecanisme sunt necesare operațiile de comunicare între procese care impun operații suplimentare. Dacă se dorește eliminarea acestor operații suplimentare atunci procesele pot fi înlocuite cu fire de execuție.

Firele de execuție mai sunt numite și “procese ușoare” (light weight processes – LWP’s). Un proces, în general, are mai multe părți componente cod, date, stivă, și de asemenea un anumit timp de execuție pe procesor. În cazul firelor de execuție vom avea mai multe procese care au alocate propriile lor intervale de timp de procesor dar folosesc în comun segmentele de cod, date, memoria și structurile da date.

În acest caz operațiile necesare pentru comutarea între două fire de execuție implică un efort semnificativ mai mic decât în situația unor procese. Deoarece firele de execuție au fost proiectate astfel încât să fie cooperative, nucleul sistemului de operare nu trebuie să ia măsuri suplimentare de protecție a resurselor acestora în cadrul unui proces. Din acest motiv firele de execuție au primit denumirea de “procese ușoare”. La polul opus se află procesele ca atare, care comparativ cu firele de execuție implică un efort de comutare mai mare și ca urmare se mai numesc și “procese grele” (heavy weight processes – HWP’s)

În general procesele pot fi implementate în două moduri:

- *În spațiul nucleului* – de obicei aceste procese sunt implementate în nucleul sistemului de operare, implementare ce se bazează pe controlul, în nucleu, a tabelelor de semnale asociate firului de execuție. În acest caz nucleul sistemului de operare este răspunzător de programarea execuției fiecărui fir, iar intervalul de timp alocat firului este scăzut din timpul alocat, global, procesului din care face parte firul. Dezavantajul acestei metode de control al timpului de execuție este dat de apariția unui efort suplimentar la comutarea utilizator-nucleu-utilizator. Avantajul major este dat de planificarea controlată a firelor de execuție (la expirarea cuantei de timp de procesor nucleul îi va întrerupe execuția și va transmite controlul firului următor), deci este exclusă blocarea procesului datorată unor operații de I/E, iar în cazul sistemelor de calcul cu mai multe procesoare performanța poate crește proporțional cu numărul de procesoare adăugate sistemului.
- *În spațiul utilizatorului* – în acest caz este evitat controlul nucleului asupra firului de execuție – pentru o astfel implementare, numită și multitasking cooperativ, se definesc un set de rutine ce sunt responsabile de comutarea execuției între fire. Tipic un fir de execuție cedează controlul în mod explicit procesorul, prin apelul explicit a unei rutine, trimiterea unui semnal. O problemă majoră ce poate apărea este acapararea de către un fir a întregului timp de procesor asociat unui proces (ex. dacă procesul execută o operație de I/E), o posibilă rezolvare este utilizarea unui semnal de timp care să determine comutarea între fire.

În paragrafele ce urmează discuția se va baza pe fire de execuție POSIX numite și pthreads (firele ce nu se conformează acestui standard se numesc cthreads).

În cadrul unui proces un thread este recunoscut pe baza unui identificator, iar la creare fiecare thread execută o funcție. Aceasta este o funcție obișnuită ce conține codul ce trebuie rulat de thread, iar la ieșirea din funcție se termină și execuția thread-ului.

## 2. Apeluri sistem

Apelul sistem de bază pentru crearea de noi thread-uri este pthread\_create.

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void * (*start_routine)(void *), void *arg);
```

Funcția pthread\_create creează noi thread-uri și trebuie să i se transmită următoarele argumente:

- O referință către o variabilă de tip pthread\_t, în care este salvat identificatorul thread-ului.
- O referință către un obiect de tip thread attribute. Prin acest argument se controlează modul în care thread-ul interacționează cu restul programului. Dacă argumentul transmis este NULL atunci se vor folosi setările implicate.
- O referință către funcția ce urmează a fi executată de thread. Referința către funcție trebuie să fie de tipul: void\* (\*) (void \*).
- Argument către thread. Acesta trebuie să fie de tip void \*, și este transmis funcției thread-ului.

La apelul funcției pthread\_create are loc crearea thread-ului, iar revenirea din funcție se face imediat. Rularea celor două thread-uri (părinte și fiu) are loc asincron, iar execuția unui program nu trebuie să se bazeze pe o anumită ordine de rulare a thread-ilor.

Un exemplu clasic de program este Hello World, de această dată realizat cu thread-uri:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

//funcția ce va fi executată de thread-uri
void* print_message( void *ptr ){
    char *message;
    message = (char *) ptr;
    printf("%s ", message);
    return NULL;
}

int main(){
    pthread_t thread1, thread2;
    char *message1 = "Hello";
    char *message2 = "World\n";

    // creearea primului thread - tipărește Hello
    if (pthread_create( &thread1, NULL, &print_message,
(void*)message1)) {
        fprintf(stderr,"first thread error\n");
        exit(1);
    }

    // "adoarme" procesul părinte pentru o secundă.
    sleep(1);

    // creearea celui de al doilea thread - tipărește World
    if(pthread_create( &thread2,  NULL,  &print_message,
(void*)message2)) {
        fprintf(stderr,"second thread error\n");
        exit(1);
    }

    sleep(1);
    exit(0);
}
```

Compilarea programului se realizează cu următoarea comandă:

```
$ gcc -o hello_thread hello_thread.cc -D_REENTRANT -lpthread
```

Opțiunea `-lpthread` trebuie specificată la compilarea programului datorită faptului că suportul pentru thread-uri este adăugat sub forma unei biblioteci.

O problemă ce suportă două moduri de tratare, în cazul thread-urilor, este cea a terminării execuției și a valorii returnate de thread. O primă modalitate este cea prezentată în exemplul anterior în care funcția tratată de thread se încheie prin apelul `return`, iar valoarea transmisă este considerată ca fiind valoare returnată de thread. O altă modalitate este prin intermediul apelului sistem `pthread_exit`.

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Această funcție poate fi apelată direct din funcția thread-ului sau din altă funcție apelată direct sau indirect din funcția thread-ului. Valoarea primită ca și argument este valoarea ce va fi returnată de thread.

Tot din exemplul anterior se poate observa că transmiterea parametrilor către thread se realizează prin ultimul parametru al apelului `pthread_create`. Dacă nu se dorește transmiterea unui parametru funcției thread-ului atunci argumentul respectiv primește valoare `NULL`. Valoarea acestui parametru poate fi o referință către orice structură de date.

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>

// definirea structurii de date ce va fi transmisă ca
// argument thread-ului
struct chars {
    char val;
    int count;
};

// funcția thread-ului
void *char_print(void *ptr) {

    struct chars* c = (struct chars*)ptr;
    for(int i=0;i<c->count;i++) putchar(c->val);
    pthread_exit(0);
}

int main() {
```

```

pthread_t thread1, thread2;
struct chars thread1_arg,thread2_arg;

// stabilirea parametrilor pentru thread-uri
thread1_arg.val='a';
thread1_arg.count=100000;

thread2_arg.val='b';
thread2_arg.count=150000;

// creearea thread-urilor

pthread_create(&thread1,NULL,&char_print,&thread1_arg
);

pthread_create(&thread2,NULL,&char_print,&thread2_arg
);

// încheierea execuției
exit(0);
}

```

Exemplul de mai sus creează două thread-uri fiecare afișând câte un caracter de către un anumit număr de ori, totodată este și un exemplu ce ilustrează modul în care se pot transmite funcției thread-ului argumente mai complexe. Totuși acest program ascunde o problemă de funcționare. Funcția thread-ului primește ca și argumente referințe către structuri de date definite ca și variabile locale în thread-ul principal al programului. Datorită faptului că nu există nici o regulă asupra ordinii de executare a thread-urilor este posibil ca thread-ul principal să își încheie execuția înaintea celor două thread-uri de afișare a caracterelor. O consecință este faptul că este dealocată memoria rezervată structurilor de date transmise ca și parametrii thread-urilor. O rezolvare a acestei probleme este apelul sistem pthread\_join:

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

Acest apel sistem suspendă execuția thread-ului apelant până la încheierea execuției thread-ului identificat prin ID-ul th.

Pthread\_join primește două argumente – primul este ID-ul thread-ului după care thread-ul apelant trebuie să aștepte să își încheie execuția, iar al doilea este o referință către o locație de memorie în care se va reține valoarea returnată de thread-ul th. Dacă valoarea returnată de thread-ul th nu interesează, atunci al doilea argument va fi NULL.

În consecință pentru ca exemplul anterior să fie funcțional trebuie completat cu următorul cod:

```
...
```

```

// crearea thread-urilor
pthread_create(&thread1,NULL,&char_print,&thread1_arg);
pthread_create(&thread2,NULL,&char_print,&thread2_arg);

// thread-ul părinte așteaptă încheierea execuției
// thread-urilor fiu.

pthread_join(thread1,NULL);
pthread_join(thread2,NULL);

// încheierea execuției
exit(0);
}

```

Trebuie avut grijă ca orice tip de date ce este transmis unui thread prin referință să nu fie dealocat, de thread-ul părinte sau de oricare alt thread.

Uneori este util de determinat care este thread-ul care execută un anumit cod. Pentru a determina ID-ul unui thread, avem la dispoziție apelul sistem pthread\_self, care are ca și rezultat ID-ul thread-ului care îl apelează. De asemenea putem compara două thread-uri pe baza ID-urilor cu apelul pthread\_equal.

```

#include <pthread.h>

pthread_t pthread_self(void);
int pthread_equal(pthread_t      thread1,      pthread_t
thread2);

```

De exemplu este o eroare ca un thread să apeleze pthread\_join pentru el însuși. Pentru a evita o astfel de eroare se poate utiliza următoarea secvență de cod:

```

if (!pthread_equal(pthread_self(), alt_thread))
    pthread_join(alt_thread, NULL);

```

Un alt aspect important în ceea ce privește crearea și utilizarea thread-urilor este dat atributelor thread-urilor ce pot fi transmise ca și argument funcției pthread\_create. De fapt, acest apel sistem primește ca și argument o referință către un obiect de tip thread attribute. Dacă referința este NULL, atunci se vor folosi atributele implicate pentru crearea thread-ului.

Dacă se dorește creare unui thread pe baza unor atrbute particolare este nevoie de crearea unui obiect de tip thread attribute. Relativ la acest tip de obiecte sunt definite mai multe apeluri sistem dintre care cele mai importante sunt: pthread\_attr\_init - initializează obiectul primit ca și argument cu valorile implicate, și pthread\_attr\_destroy – dealocă resursele ocupate de obiectul primit ca și argument.

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Pentru a defini un thread cu atribute particulare se aplică următorii pași:

- Se creează un obiect de tipul thread attribute – se declară o variabilă de acest tip.
- Obiectul creat este transmis ca și parametru funcției pthread\_attr\_init – obiectul este inițializat cu atributele implicate.
- Se modifică atributele dorite.
- Se crea un thread pe baza apelului pthread\_create căruia i se transmite ca și argument nou creatul thread attribute.
- Resursele ocupate de thread attribute trebuie eliberate – se folosește pthread\_attr\_destroy.

Un singur obiect de tip thread attribute poate fi utilizat pentru crearea mai multor thread-uri, după care prezența acestui nu mai este necesară și obiectul poate fi dealocat.

```
# include<pthread.h>

void* thread_function(void *arg) {
    ...
}

int main() {
    pthread_attr_t attr;
    pthread_t thread;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    pthread_create(&thread, &attr, &thread_function, NULL);
    pthread_attr_destroy(&attr);
    ...
    exit(0);
}
```

Pentru majoritatea programelor un singur atribut al thread-ului este important, restul atributelor se utilizează doar în cazul programelor de timp real. Acest atribut este *detach state*. Conform acestui atribut un thread poate fi creat fie joinable (implicit), fie detached. În primul caz, joinable, starea thread-ului nu este preluată de către sistemul de operare automat și rămâne în sistem, ca și în cazul proceselor zombie, până când un alt thread apelează pthread\_join. În cazul în care thread-ul creat are atributul detached, starea de ieșire a thread-ului este automat preluată de sistemul de operare. Dezavantajul în acest caz este dat de dispariția posibilității de sincronizare între thread-uri prin intermediul funcției pthread\_join.

Chiar dacă un thread a fost creat inițial ca și joinable ulterior poate fi trecut în starea detached prin intermediul apelului sistem pthread\_detach.

În condiții normale un thread își încheie execuția fie prin terminarea rulării funcției thread-ului returnând o valoare, fie prin apelul sistem pthread\_exit. Mai există și o a treia posibilitate în care un thread solicită unui alt thread încheierea execuției. În acest caz este folosit apelul sistem pthread\_cancel:

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

Deseori execuția normală a unui thread implică alocarea unor resurse, utilizarea lor și în final dealocarea lor. Dacă execuția unui thread este anulată în mijlocul unui astfel de proces, resursele alocate inițial se vor pierde.

În concluzie din acest punct de vedere un thread se poate afla în una din următoarele stări:

- Asincron anulabil – își poate încheia execuția în orice moment.
- Sincron anulabil – se poate solicita din exterior încheierea execuției, dar nu în orice moment. Cererile de încheiere a execuției vor fi puse într-o coadă de așteptare și vor fi luate în considerare doar în anumite momente ale execuției.
- Neanulabil – toate cererile de încheiere a execuției sunt ignorate.

Implicit toate thread-urile sunt sincron anulabile.

Acest mecanism de terminare a execuției unui thread pe baza unor cereri externe se bazează pe următoarele apeluri sistem:

```
#include <pthread.h>
int pthread_setcancelstate(int state, int *oldstate );
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

Așa cum am să arătat anterior unui thread asincron anulabil i se poate solicita în orice moment să își încheie execuția, în timp ce un thread sincron anulabil își poate încheia execuția doar în anumite momente numite puncte de anulare.

Pentru ca un thread să fie asincron anulabil (implicit este sincron) se folosește apelul pthread\_setcanceltype unde primul argument este PTHREAD\_CANCEL\_ASYNCHRONOUS (PTHREAD\_CANCEL\_DEFERRED pentru a reveni la cazul sincron), al doilea argument este o referință către o variabilă care conține vechea stare, dacă lipsește – NULL:

```
#include <pthread.h>
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,
NULL);
```

Pentru a testa dacă un thread este anulabil într-un anumit punct se utilizează – pthread\_testcancel. Utilizarea acestui apel sistem este valabilă în cazul thread-urilor asincrone. Se poate indica din interiorul thread-ului secțiunile neanulabile prin apelul pthread\_setcancelstate:

```
...
int old_cancel_state;
...
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_cancel_state)
...
pthread_setcancelstate(old_cancel_state, NULL);
...
```

Pe baza acestui apel se pot implementa secțiuni critice, care fie se execută în întregime, fie nu se execută de loc. Dacă un thread începe execuția unei astfel de secțiuni atunci trebuie să o continue până la sfârșitul secțiunii fără a fi anulat.

Implementarea thread-urilor POSIX este diferită de implementarea din multe alte sisteme de operare de tip UNIX. Diferența este dată de faptul că în Linux thread-urile sunt implementate ca și procese. Fiecare apel `pthread_create` creează un nou proces ce rulează thread-ul nou creat. Oricum acest proces este diferit de cel creat cu `fork`, el utilizează în comun spațiul de adrese și resursele cu procesul original.

O problemă ce se poate pune în cazul programelor implementate multithread este gestionarea semnalelor. Să presupunem că un astfel de program primește un semnal din exterior, se pune întrebarea care thread va apela funcția de tratare a semnalului (handler-ul). În cazul sistemelor de operare de tip Linux problema este rezolvată chiar de implementarea acestora de tip proces – semnalul va fi tratat de thread-ul ce rulează programul principal. De asemenea, în interiorul unui program multithread este posibil, ca un thread să poată trimite un semnal unui alt thread, pentru aceasta se folosește apelul sistem `pthread_kill`.