Universitatea Politehnica Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei

Proiect Sisteme de operare avansate

Nucleul sistemului de operare UNIX (KERNEL)

Coordonator:
Conf. dr. ing. Ştefan Stăncescu

Masterand IISC:
Simona-Ioana Barbu

2013

Cuprins

Capitolul 1: Introducere

Sistemul de operare Unix este cel mai vechi sistem de operare, acesta rezistând şi impunându-se până în zilele noastre.

Unix este un sistem de operare time-sharing universal, a cărui caracteristică principală o constituie portabilitatea, mai exact disponibilitatea să pentru diverse tipuri de calculatoare (microcalculatoare, minicalculatoare, supercalculatoare) şi pentru reţele puternice de calculatoare.

Ceea ce a dus la apariţia acestui sistem de operare se numără printre urătoarele:

- necesitatea standardizării şi unificării sistemelor de operare, în special a interfeţei cu utilizatorul;
- păstrarea structurii volumelor şi fişierelor în condiţiile transferului pe alte sisteme de calcul;
- asigurarea unor niveluri superioare de portabilitate a produselor - program;
- posibilitatea interconectării de sisteme cu arhitecturi, tipuri şi puteri diferite, sub acelaşi sistem de operare.
- independenţa software-ului de aplicaţii, faţă de evoluţia hardware.

1.1. Evoluţie

În anul 1969 a apărut prima versiune experimentală scrisă în limbaj de asamblare pentru minicalculatoarele PDP-11, al firmei DEC (Digital Equipment Corporation) pentru un singur utilizator (monouser); odată cu primul compilator pentru limbajul C (1972), UNIX a fost rescris în acest limbaj cu scopul asigurării portabilităţii.

În 1978 s-a realizat prima implementare comercială ÎS/1 (Interactive System One), urmată la scurt timp de versiunea XENIX a firmei Microsoft. În 1980, UNIX s-a impus că principală soluţie de standardizare în domeniul sistemelor de operare, reprezentând o modalitate de realizare a sistemelor deschise pentru toate categoriile de sisteme de calcul.

După 1981, firma AT&T elaborează versiunile UNIX System III şi V, iar firma Berkeley Software Distribuitors (BSD) realizează standardele BSD 1,2,3 urmate de BSD 4.2, 4.3 şi din 1993 BSD 4.4 Companiile SUN şi AT&T au dezvoltat versiunea System V.4 în 1988, iar IBM, DEC şi Hewlett

Packard au format Open Software Foundation (OSF) independent de AT&T (fig. 1.1.1).

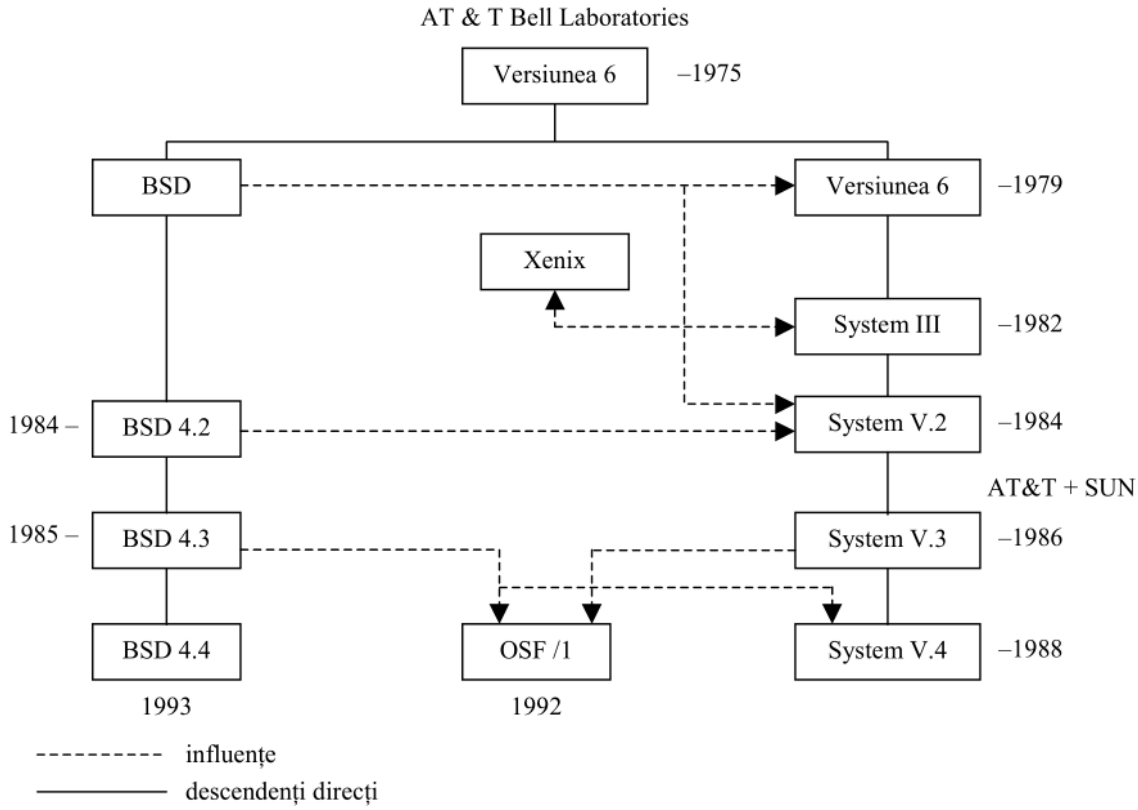

Fig. 1.1.1. Evolutia sistemelor de operare UNIX

1.2. Caracteristici

- Sistem de operare time-sharing;
- Protecţia fişierelor (prin intermediul parolelor şi a drepturilor de acces);
- Intrări/ieşiri generalizate prin care se face integrarea operaţiilor de intrare/ieşire în sistemul de fişiere;
- sincronizarea proceselor prin intermediul unui sistem de întreruperi logice;
- transferul de pagini între memoria RAM şi cea externă care permite managementul spaţiului afectat execuţiei proceselor şi controlul timpului de acces la procesele în aşteptare;
- interfeţe simple şi interactive (SHELL) prin care se asigură dialogul utilizatorului cu sistemul de operare;
- portabilitatea sistemului de operare cât şi a software-ului de aplicaţie;

- gama vastă de aplicaţii: compilatoare, sisteme de gestiune a bazelor de date, reţele de calculatoare, inteligenţă artificială, simulare, gestiune, statistică, instruire asistată de calculator, etc.;
- permite execuţia aplicaţiilor în MS-DOS, în paralel cu execuţia de procese sub UNIX (submeniu pentru MS-DOS);
- întreţinere şi dezvoltare facilă.

1.3. Arhitectura sistemului UNIX

Arhitectura unui sistem de calcul se prezintă din punct de vedere al funcţionalităţii pe trei niveluri (fig. 1.3.1.)

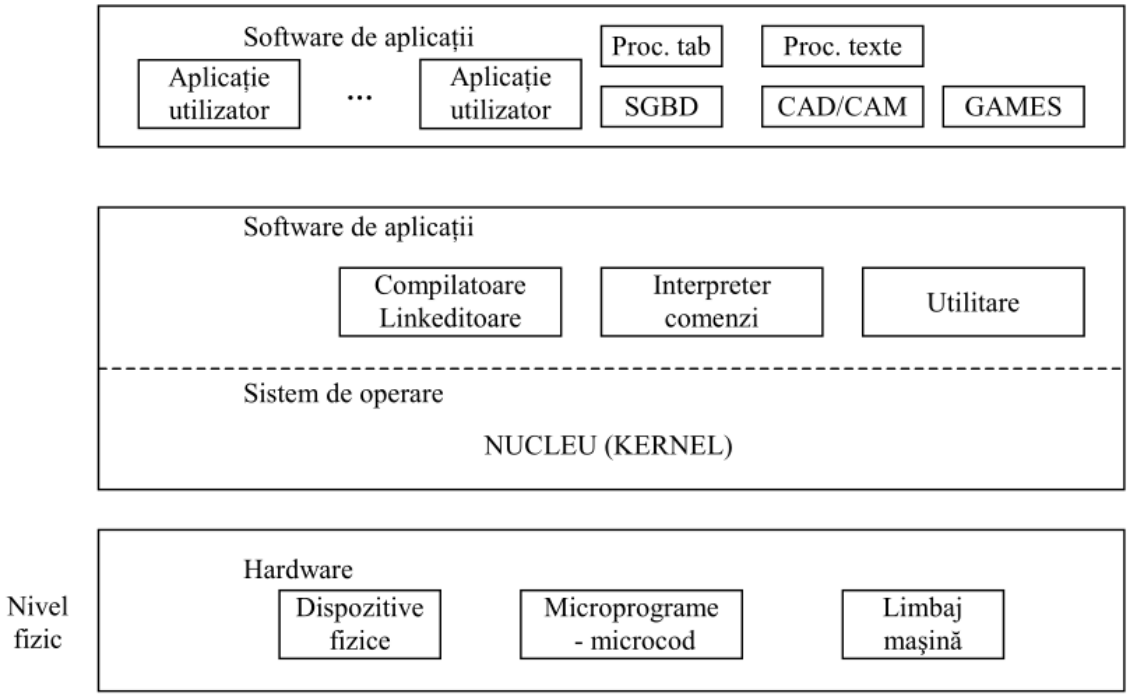

Fig 1.3.1. Arhitectura sistemelor ce lucreaza sub UNIX

Nivelul fizic este nivelul inferior. La acest nivel operează hardware-ul, care include dispozitivele fizice, microcodul şi limbajul de asamblare.

Următorul nivel - software-ul de bază sau software-ul de sistem are că principală componentă sistemul de operare având la bază nucleul (Kernel) alături de editoare de texte, compilatoare, interpretere de comenzi şi utilitare.

Nivelul superior este constituit din software-ul aplicativ şi cuprinde practic, o gamă infinită de aplicaţii.

Se poate observa faptul că orice nivel comunică numai cu nivelul imediat inferior sau superior, furnizând servicii pentru nivelul imediat superior.

Scopul principal al sistemului de operare este acela de a controla resursele hardware ale sistemului, acesta acţionând ca o interfaţă între utilizator (programele de aplicaţie pe care acesta le lansează în execuţie) şi componenta hardware (fig. 1.3.2.).

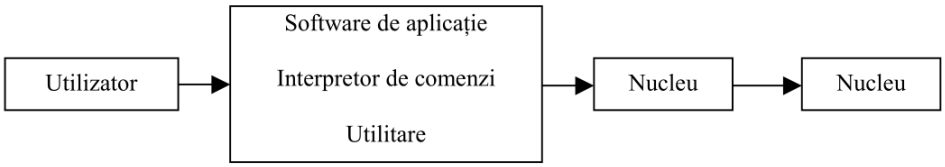

Fig. 1.3.2. Sistemul de operare UNIX- interfata intre hardware si utilizator

Sistemul UNIX lucrează în time-sharing, mai exact este constituit dintr-un nucleu (Kernel) şi un număr foarte mare de utilitare accesibile prin intermediul interpretorului de comenzi Shell, interpretor ce reprezintă interfaţa dintre sistemul de operare şi utilizator:[1]

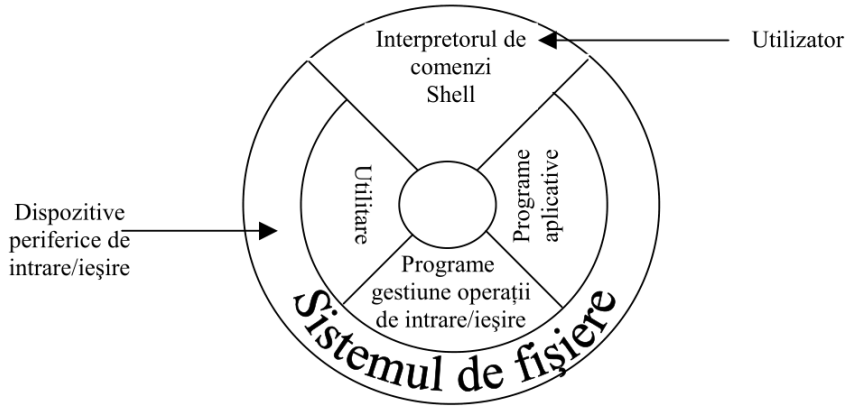

Fig 1.3.3. Componentele sistemului de operare UNIX

- Nucleu (Kernel)
  - partea rezidentă;
  - asigură servicii de sistem pentru programele de aplicaţii
  - asigură gestiunea proceselor, a memoriei, a intrărilor/ieşirilor şi a timpului.

[1] Structura UNIX, unibuc.ro

- Shell
    - interpretor de comenzi
    - mecanismul prin care sistemul de operare realizează interfaţa dintre utilizator şi sistemul de calcul

- Sistemul de fişiere cuprinde programe:
    - utilitare
    - aplicative
    - de gestiune a operaţiilor de intrare/ieşire

- Utilitare - servicii accesibile prin intermediul interpretorului de comenzi.

Interfeţele disponibile utilizatorului sunt organizate pe trei niveluri:

- nivelul exterior nucleului, care poate fi accesat de către utilizator prin intermediul utilitarelor;
- nivelul intermediar, accesat prin funcţii din biblioteca limbajului C;
- nivelul inferior, a cărui accesare se realizează prin funcţiile de sistem (System Calls).

Prin urmare, există o ierarhizare a straturilor ce se interpun între utilizator şi nucleul sistemului de operare redată în fig. 1.3.4.

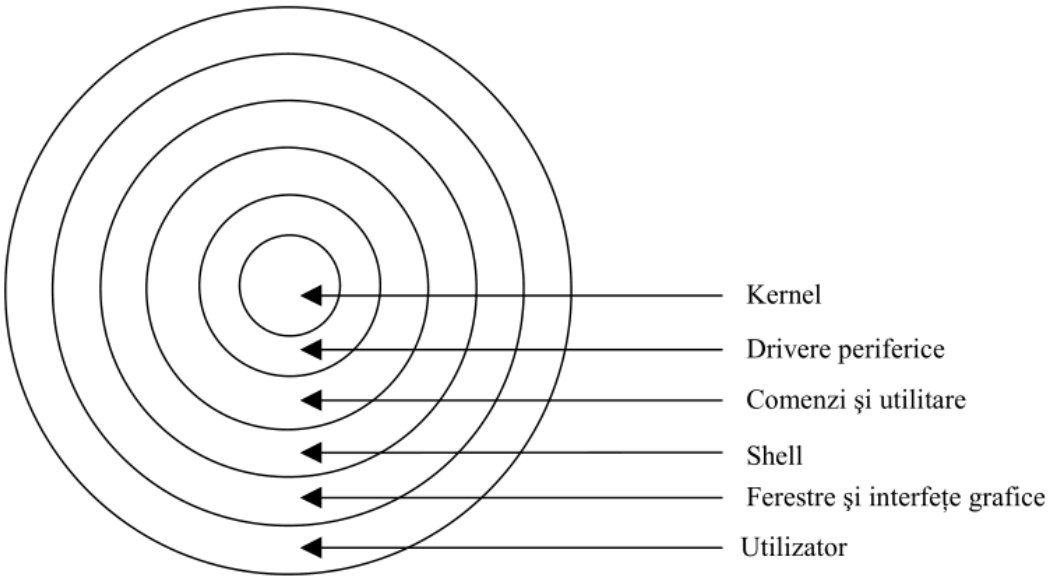

Fig. 1.3.4. Straturile interpuse intre utilizator si nucleul sistemului de operare UNIX

- Nucleul (Kernel) UNIX este constituit din două componente principale:

- sistemul de gestiune a fişierelor;
- sistemul de gestiune a proceselor.

Capitolul 2: Structură şi funcţii

Nucleul sistemului de operare este alcătuit din următoarele componente:

- programul supervizor central;
- rutine de serviciu pentru o serie de activităţi cum ar fi de exemplu, scrierea în memorie, gestiunea ceasului sistem, etc.

În mod normal, utilizatorul foloseşte comenzile Shell care dirijează execuţia programelor dorite sub supervizorul Kernel, acesta fiind astfel invizibil utilizatorului. Pentru utilizator, sistemul de operare UNIX apare ca fiind alcătuit din:

- un set de programe, fiecare corespunzând unei anumite comenzi;
- Shell-ul care dirijează comenzile şi coordonează execuţia programelor utilizatorului.

Practic însă, sistemul de operare UNIX este compus din:

- un set de programe de servicii care execută funcţiile legate de sarcinile hardware şi software;
- Kernel-ul care coordonează execuţia programelor de servicii, specificate sub forma unor comenzi adresate către Shell.

Programele de servicii sunt apelate atunci când sunt necesare (zona de tranzienţe), în timp ce supervizorul este rezident în RAM, constituind mediul software de bază pentru orice evenimente din sistem.

Principalele funcţii îndeplinite de nucleul sistemului de operare:

- planificarea, coordonarea şi gestionarea execuţiei proceselor;
- furnizarea de servicii de sistem cum sunt: tratarea operaţiilor de intrare/ieşire şi gestiunea fişierelor;
- manipularea operaţiilor dependente de hardware, întreruperile şi funcţiile de sistem;
- gestiunea memoriei.

Nucleul UNIX este alcătuit din aproximativ 10000 de linii ce constituie codul programului care, în funcţie de sistem, se transformă într-un număr mai mare sau mai mic de cuvinte maşină (sau bytes); dintre acestea, 5 - 10% din totalul codului programelor (Shell, utilitare, KERNEL şi celelalte) este variabil funcţie de sistemul de calcul şi de setul de utilitare (fig. 2.1.).

Aşa cum rezultă din fig. 2.1, o mare parte din aceste programe este destinată gestiunii memoriei şi gestiunii proceselor, această parte evidenţiind:

- conţinutul stivei;
- conţinutul registrelor sistemului;
- detalii de mediu, când procesele sunt introduse în memorie şi răspund la întreruperile procesorului.

Această parte conţine 7 - 8000 linii codul programelor scrise în limbajul C (deci portabilă pe orice sistem de calcul). Includerea acestei părţi în nucleu se justifică prin necesitatea unui răspuns foarte rapid.

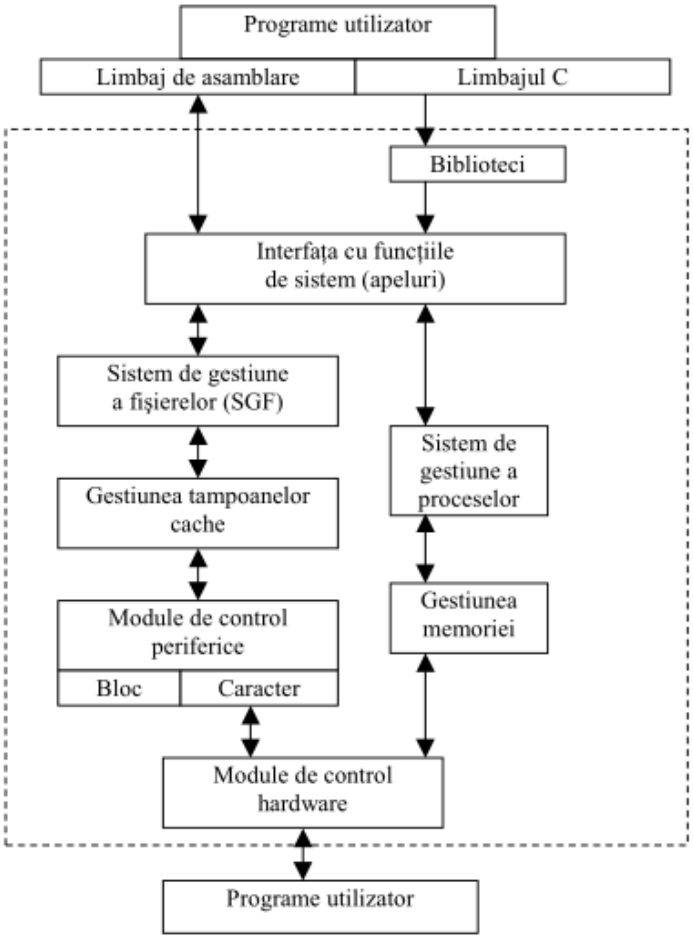

Fig. 2.1.1. Structura Kernel

- Altă parte a Kernel-ului conţine driverele dispozitivelor periferice de intrare/ieşire constituite din programe ce realizează:
- controlul adreselor de citire/scriere, adresarea registrelor de date ale perifericelor de
- intrare/ieşire;
- manipularea întreruperilor generate de aceste dispozitive;
- efectuarea recuperării erorilor.

Conţine aproximativ 1000 linii codul programelor scrise tot în limbajul C, dar acest număr este variabil în funcţie de numărul perifericelor de intrare/ieşire.

Primitivele de sistem sunt specifice fiecărui sistem de calcul (scrise în limbaj de asamblare, aproximativ 1000 linii codul programului), ele conţinând:

- operaţii de intrare/ieşire de bază;
- comutarea execuţiei între procese;
- permiterea sau inhibarea întreruperilor hardware;
- resetarea priorităţilor întreruperilor;
- alte operaţii.

Accesarea primitivelor de sistem se realizează prin apeluri (directive) de sistem (system calls) din programe în C sau în limbaj de asamblare.

Capitolul 3: Procese şi stări

Un sistem de calcul poate să lucreze la un moment dat în două moduri:

- utilizator, când execută un program sau proces;
- sistem (Kernel), când execută un cod sistem.

Comutarea între modul utilizator şi KERNEL se realizează prin următoarele mecanisme:

- ceasul - care întrerupe orice alt program cu frecvenţa de 60 Hz, rutina de ceas permiţând reevaluarea priorităţilor proceselor şi implicit schimbarea procesului; în absenţa altor întreruperi, ceasul realizează divizarea timpului, ceea ce permite ca sistemul să fie împărţit între mai mulţi utilizatori;

- apeluri de sistem prin care utilizatorul solicită diverse servicii oferite de sistemul de operare; cele care realizează operaţii de intrare/ieşire conducând la suspendarea procesului apelator pe durata transmiterii datelor;
- cereri de serviciu ale perifericelor de intrare/ieşire.

Procesul este conceput fundamental de organizare a sistemului de operare UNIX. În esenţa, un proces reprezintă un program în execuţie. Pentru un program activ, pot exista mai multe procese active - numite instanţe:

- din punct de vedere al procesului, operaţiile nucleului sunt prioritare;
- din punct de vedere al nucleului, procesele sunt structuri de date catalogate.

Informaţiile necesare procesului sunt memorate în:

a) tabela proceselor constituită în memorie, ce conţine o intrare pentru fiecare proces detaliind starea acestuia:
   - localizarea procesului;
   - adresa de memorie şi adresa de evacuare;
   - mărimea procesului;
   - numărul de identificare;
   - identificatorul utilizat;
b) tabela utilizator, care este ataşată fiecărui proces activ la rutinele nucleului lui (fig. 3.1.).

Crearea unui proces implică astfel iniţializarea unei intrări în tabela procesului, care iniţializează o tabelă a utilizatorului creând totodată textul real şi datele pentru proces.

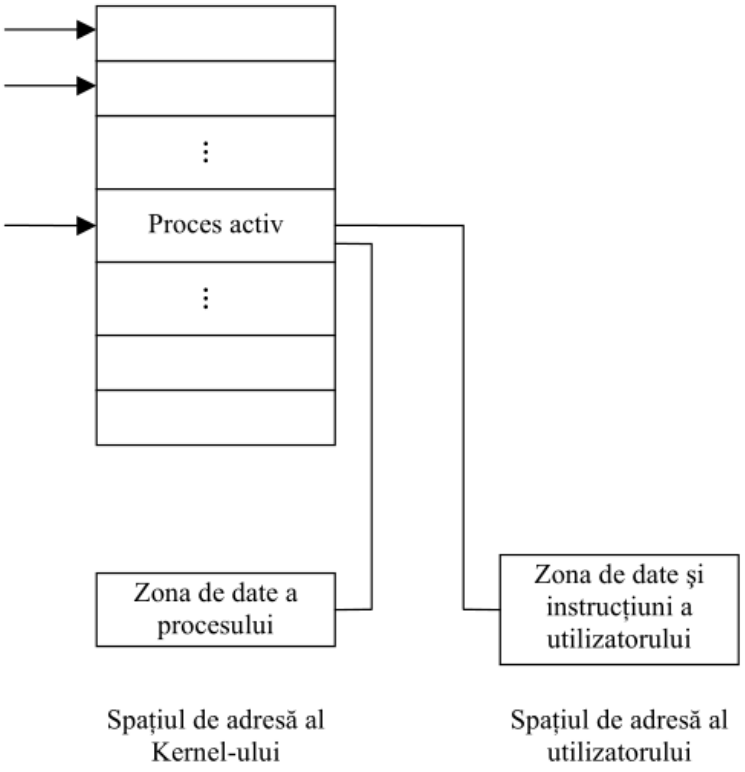

Fig. 3.1. Iniţializarea tabelei utilizator

Schimbarea stării unui proces (se execută, aşteaptă, este evacuat, este încărcat, primeşte un semnal) este o activitate ce se focalizează pe tabela procesului (fig. 3.2.):

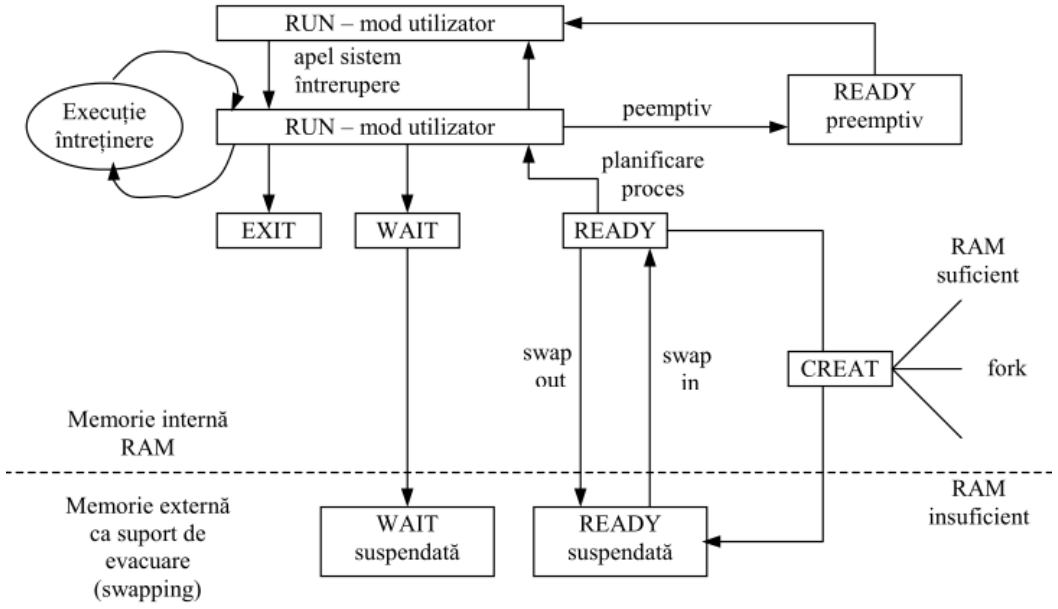

Fig. 3.2. Tranzitia starilor unui proces

- terminarea unui proces implică eliberarea intrării sale în tabela proceselor, putând fi folosită pentru alte procese;
- când procesul este activ, au loc alte evenimente, de exemplu aşteptarea terminării unei operaţii de intrare/ieşire;
- pe durata suspendării unui proces, tabela utilizatorilor nu este accesată sau modificată;
- dacă procesul este evacuat pe disc, este evacuată şi tabela utilizator în cadrul imaginii procesului;

Tabela utilizator (structură a utilizatorului) conţine în acel moment:

- numele de identificare al utilizatorului şi al grupului din care face parte, pentru stabilirea drepturilor de acces;
- pointerii din tabela de fişiere existente în sistem, pentru toate fişierele deschise;
- un pointer către inod-ul directorului curent în tabela de inod-uri;
- o listă a răspunsurilor pentru diverse semnale.

Informaţia curentă despre un proces se schimbă prin execuţia directivei *chdir* când este schimbată valoarea pointerului către inod-ul directorului curent.

Capitolul 4: Coordonarea şi gestiunea proceselor

Într-un sistem de operare multiutilizator, se pot executa mai multe programe simultan. Existând o singură CPU, în realitate un singur program se execută efectiv la un moment dat, iar celelalte se vor executa atunci când CPU le va acorda cuante de timp. În timpul execuţiei unui program, celelalte programe sunt rezidente în RAM, cu condiţia ca nici unui să nu solicite toată memoria. De aici decurg 2 sarcini ale nucleului:

- planificarea proceselor - împărţirea de cuante de timp între procese;
- gestiunea memoriei - atribuirea de zone de memorie liberă la procese şi eventuala evacuare au încărcare a unui proces în/din disc.

4.1. Planificarea proceselor

Iniţierea unui proces poate fi efectuată numai de către un alt proces activ. Când este conectat primul utilizator, Kernel-ul asigură o copie a Shell-ului ce se rulează doar pentru el, ivindu-se existenţa unei alte structuri ierarhice de procese create printr-un mecanism numit bifurcaţie (fork) prin care Kernel înlocuieşte un proces existent, prin două procese:

- procesul iniţial (părinte);
- procesul iniţiat de procesul iniţial (fiu), care împarte (partajează) toate fişierele cu procesul părinte.

După bifurcaţie, ambele procese se execută independent; excepţie: când se solicită explicit că procesul părinte să aştepte terminarea procesului fiu prin directiva WAIT; în continuare, procesul fiu poate genera la râul lui, o nouă bifurcaţie (fig.4.1.1.).

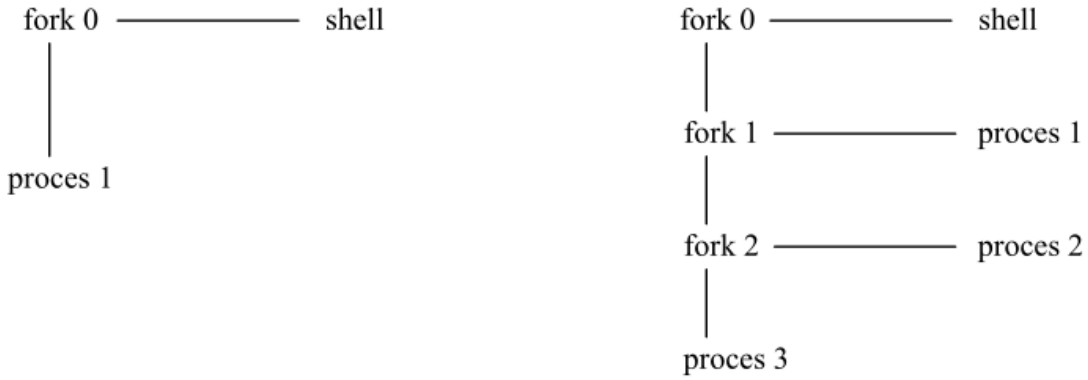

Fig. 4.1.1. Mecanismul *fork*

- De remarcat că *proces3* are acces la toate fişierele deschise de procesele anterioare, în timp ce invers nu se permite accesul. Prin urmare, fişierele sunt accesibile de pe nivelurile situate la periferia ierarhiei.

Când Shell iniţiază un proces, bifurcaţia este aranjată astfel încât Shell să aştepte terminarea procesului (procesul este executat în foreground). Dacă nu aşteaptă, înseamnă că Shell a dat naştere altor procese mai puţin prioritare (procese executate în background) adăugând după comanda corespunzătoare semnul **&**.

4.2 Alocarea memoriei

Fiind un sistem de operare multitasking, UNIX ţine evidenţa taskurilor concurente şi menţine controlul diverselor programe care sunt rezidente în RAM la un moment dat, furnizând şi informaţii privind suficienţa spaţiului de memorie:

- în mod normal, fiecare program este încărca în diverse zone de memorie RAM;
- în modul de operare cu divizarea timpului (time-sharing), poate opera fără evacuarea conţinutului memoriei pe disc; fiecare program se execută în cuanta de timp alocată rămânând în memorie numai programele a căror execuţie nu s-a încheiat (practic, numai registrele sistemului sunt divizate între programe, astfel că evacuarea programelor implică de fapt, numai evacuarea conţinutului registrelor);
- pe durata execuţiei proceselor, UNIX alocă porţiuni distincte de RAM pentru:
    - segmentul de cod (instrucţiunile programului) protejat la scriere;
    - segmentul de date ce conţine toate datele definite de utilizator: valori, variabile, etc.;
    - segmentul de stivă care conţine toate informaţiile de sistem necesare menţinerii intacte a unui proces, atunci când este evacuat din RAM pe disc sau când este încărcat în RAM de pe disc.

Această divizare este realizată din două motive:

- evacuarea segmentului de cod implică reducerea cantităţii de informaţii necesară a fi evacuate; fiind de tip read-only, acesta este identic cu imaginea sa din disc, deci nu trebuie rescris când se evacuează;
- Shell este încărcat pentru fiecare utilizator, deci există atâtea copii Shell, câţi utilizatori sunt conectaţi; dacă mai mulţi utilizatori solicită un program (segment de cod), nu este necesară copierea acestuia deoarece el poate fi folosit simultan; este suficient să se creeze şi când este necesar, să se evacueze un segment de date şi un segment de stivă pentru fiecare utilizator.

4.3. Divizarea timpului şi resurselor

În sisteme de operare multitasking, procesele individuale nu sunt executate până la terminare, având alocate cuante de timp de către procesor într-o planificare de tip Round- Robin;
Cuantele de timp nu sunt egale, lungimea depinzând de prioritatea taskului, de accesibilitatea datelor de intrare solicitate şi a perifericelor de ieşire.

Alocarea se face având în vedere maximizarea utilizării resurselor hardware, cu respectarea priorităţii taskurilor critice.

Priorităţile sunt periodic reînnoite astfel:

- taskurile cu timp de execuţie mai mare, vor avea prioritate mai scăzută;
- taskurile de tip întrebare-răspuns vor avea răspuns instantaneu;
- taskurile ce constă în schimbarea de caractere au cea mai scăzută prioritate.

După expirarea timpului alocat, taskul este evacuat într-un fişier de evacuare pe disc unde vă rămânea până i se dă din nou controlul; în acel moment, imaginea salvată anterior se reîncarcă în RAM; ea va cuprinde:

- conţinutul părţii de memorie inscriptibilă;
- conţinutul registrelor;
- numele directorului curent;
- lista fişierelor deschise de procesul considerat.

În acest sens, sunt utile prezentarea apelurilor de sistem:

I=fork ()

Care permite crearea unui nou proces fiu, având active procesele părinte şi fiu. Practic, se introduce un nou proces în tabela de procese a sistemului UNIX, dar nu se lansează în execuţie nici un program. Se crează astfel o bifurcaţie în care procesul fiu este realizat că o copie a procesului părinte, nefiind necesară evacuarea programului din memorie ci doar copierea zonelor de date.

J=execl (nume, arg1, arg2,..., argn, o)

Solicită execuţia unui alt program; nume, arg1, arg2,..., argn sunt pointeri către şiruri de caractere care specifică numele programului şi numele

argumentelor sale; o - pointer către o zonă de pointeri, fiecare din ei punctând câte un şir din mediu.

Prin acest apel, nucleul va determina ce program original să fie înlocuit cu alt program.

Secvenţa *fork-execl* crează o copie a procesului părinte, apoi înlocuieşte procesul fiu cu programul care doreşte să-l execute (procesul fiu nu este alterat de *execl*).

K=wait (status)

Determină procesul părinte să aştepte terminarea execuţiei procesului fiu;
Status este pointer la o valoare întreagă, iar ***k*** - valoarea returnată ca număr de identificare al procesului fiu terminat.

O aplicaţie frecventă a lui execl se întâlneşte la înlănţuirea programelor; de exemplu, o compilare în mai mulţi paşi implică încărcarea programului şi datelor pentru execuţie la primul pas; în continuare, se procedează asemănător şi la al doilea pas; o dată ce primul pas a fost executat complet, nu există nici un motiv de a se întoarce la el, deci a se suprascrie memoria ocupată la prima trecere cu programul şi zona de date.

Înlănţuirea ne este totdeauna practică pentru secvenţe în care părţi din program solicitate sunt dependente de date; astfel, procesul părinte crează un proces fiu (prin fork) cu cerere de aşteptare (wait); procesul fiu va exista ca proces evacuabil şi va avea acces la acelaşi spaţiu de memorie ca şi procesul părinte; prin execl, procesul fiu poate executa următorul program, putând el însuşi să fie rescris; deci, când procesul fiu modificat şi suprascris va fi încheiat, controlul va reveni procesului părinte.

Deoarece numărul de procese concurente nu este limitat, această schemă permite execuţia unui mare număr de programe cu condiţia să fie posibil a se divide în programe şi date astfel încât fiecare execuţie să încapă în RAM.

Capitolul 5: Sincronizarea proceselor

Sincronizarea proceselor poate fi sub controlul sistemului de operare prin intermediul conductelor de comunicaţie *(pipe)* sau al utilizatorului, prin intermediul evenimentelor care reprezintă modul de determinare al momentului în care un proces devine gata pentru execuţie *(READY*).

La un moment dat un proces se află în execuţie; blocarea procesului se poate realiza prin funcţia sleep, moment în care nucleul trece toate procesele care îndeplinesc condiţii stabilite, în starea gata de execuţie; dintre acestea, numai un singur proces se va lansa în execuţie, celelalte fiind trecute din nou în starea de blocare.

Mecanismele de sincronizare prin intermediul evenimentelor privesc sincronizarea prin intermediul semnalelor sau al semafoarelor, respectiv al directivelor de sistem.

5.1. Sincronizarea prin intermediul semnalelor.

Aşa cum s-a menţionat anterior, la îndeplinirea condiţiilor prestabilite pentru lansarea în execuţie, nucleul trece procesele în starea gata de execuţie prin intermediul funcţiei wakeup.

O altă modalitate de sesizare a unor evenimente apărute în sistem o constituie plasarea unor biţi de atenţie *(lock-bit)* în tabela de procese, nucleul efectuând verificarea activării bitului corespunzător care se asimilează cu emiterea unui semnal la trecerea din modul sistem în modul utilizator înainte, respectiv după blocarea fiecărui proces.

La apariţia unui semnal se pot ivi următoarele acţiuni ce se execută în continuare:

- terminarea procesului, când se afişează de regulă, şi imaginea procesului în memorie în fişierul *core*;
- acţiuni specifice care sunt desemnate prin funcţia:

Signal (nr_semnal, nr_rutină_tratare_semnal)

De remarcat că după tratarea semnalului, se revine la valorile standard ale acţiunilor ce se vor executa ulterior.

Un caz particular îl constituie utilizarea funcţiei de sistem wait, care permite blocarea unui proces până la terminarea unuia dintre fiii acestuia sau până la recepţionarea unui semnal:

P=wait (&stare)

Unde:

- *p* reprezintă valoarea returnată a funcţiei (identificatorul de proces al fiului):
    - - 1 dacă s-a recepţionat un semnal, o eroare sau s-a terminat deja execuţia fiilor procesului;
    - - numărul de identificare al fiului a cărui execuţie s-a terminat;
- *stare* adresa unde se specifică modul de terminare al execuţiei iniţiate de fiu.

Pentru sincronizarea procesului cu un anumit fiu se poate folosi:

While (wait (&stare)! =pid)

Care reia ciclic testarea terminării execuţiei acţiunii fiului cu identificatorul pid, ceea ce evident implică blocarea procesului apelant (tatăl).

5.2. Sincronizarea prin semafoare

Particularitatea sincronizării prin intermediul semafoarelor constă în aceea că se pot executa operaţii pe o mulţime de semafoare printr-un singur apel de funcţie. Structurile de date asociate semafoarelor sunt constituite din:

- identificatorul mulţimii de semafoare;
- descrierea mulţimii de semafoare care include:
    - utilizatorii care au acces la mulţimea de semafoare;
    - numărul de semafoare al mulţimii;
    - momentul ultimei accesări şi al ultimei actualizări al mulţimii semafoarelor;
- descrierea semaforului care cuprinde:
    - valoarea semaforului;
    - identificatorul ultimului proces care a accesat semaforul;

- numărul proceselor care aşteaptă ca valoarea semaforului să devină mai mare decât valoarea sa curentă;
- numărul proceselor care aşteaptă ca valoarea semaforului să devină zero.

Algoritmul de execuţie al unei operaţii pe semafoare se desfăşoară astfel:

a) dacă semaforul operaţiei este mai mare ca zero, atunci acesta se va aduna la valoarea semaforului;
b) dacă semaforul operaţiei este zero atunci:
   - dacă valoarea semaforului este zero, funcţia se termină imediat;
   - dacă valoarea semaforului este zero şi indicatorii de condiţie *(flags)* arată *NOWAIT*, funcţia se termină imediat; în caz contrar:
   - se incrementează numărul de procese care aşteaptă ca valoarea semaforului să fie zero;
   - se suspendă procesul curent până când valoarea semaforului devine zero (prin decrementarea numărului de procese ce aşteaptă ca valoarea semaforului să fie zero) sau se primeşte un semnal prin care se efectuează aceeaşi decrementare şi se reia procesul.

C) dacă semaforul operaţiei este mai mic decât zero şi:

- dacă valoarea semaforului este mai mare sau egală cu modulul semaforului operaţiei, atunci din valoarea semaforului se va scădea valoarea absolută a semaforului;
- dacă valoarea semaforului este mai mică decât modulul semaforului operaţiei şi indicatorii de condiţie specifică *NOWAIT*, funcţia se va termina imediat; în caz contrar:
  - se incrementează numărul proceselor care aşteaptă ca valoarea semaforului să devină mai mare ca valoarea sa curentă, suspendându-se procesul curent până când valoarea semaforului devine mai mare sau egală cu valoarea absolută a semaforului operaţiei (când numărul de procese ce aşteaptă ca valoarea să devină mai mare ca valoarea curentă se decrementează, iar din valoarea semaforului se va scădea valoarea absolută a semaforului operaţiei sau se primeşte un semnal (când numărul de procese care aşteaptă ca valoarea semaforului să devină mai mare decât valoarea curentă se decrementează, iar procesul suspendat se reia).

5.3 Sincronizarea prin directive de sistem

Sincronizarea se poate realiza şi printr-o serie de funcţii disponibile utilizatorului cum sunt:

Kill (identificator_proces, număr_semnal);

Care realizează trimiterea unui semnal pentru un proces;

Pause ();

Realizează blocarea procesului curent până la primirea unui semnal;

Allarm (număr_secunde);

Trimite un anumit semnal (întrerupere de ceas) după intervalul de timp specificat în număr de secunde.

Capitolul 6: Comunicaţia între procese

La sistemele de operare UNIX este necesară realizarea unor modalităţi practice de comunicaţie între procese; mecanismele actuale permit alternativele prezentate în continuare.

6.1. Comunicaţia prin fişiere

Comunicaţia prin intermediul fişierelor oferă avantajul simplităţii de realizare şi o modalitate directă prin care se poate efectua comunicarea. Dezavantajul constă în aceea că este sarcina programatorului să controleze accesul concurent, prin testarea ciclică a momentului eliberării zonei critice.

Exemplu

```
While ((f=creat ("fişier", 0111)) <0) sleep (2);
        < zona critică >
Unlink ("fişier");
```

Prin care primul proces care reuşeşte să creeze fişierul, va trece la execuţia instrucţiunilor din regiunea critică; până ce nu va executa funcţia *unlink*, orice alt apel al funcţiei de creare a unui fişier *(creat),* va bloca procesul prin apelul funcţiei *sleep.*

6.2. Comunicaţia prin fişiere de tip pipe

Fişierele de tip *pipe* sunt fişiere obişnuite la care însă operaţiile de citire/scriere se realizează într-o ordine prestabilită *FIFO (First Input First Output*); aceste fişiere sunt tranzitorii, datele citite într-o manieră strictă a ordinii în care au fost scrise respectând regula de sincronizare producător/consumator (o dată citită dintr-un fişier nu mai poate fi reluată, iar memorarea se face ca la orice fişier utilizând însă numai blocuri adresate direct).

La UNIX există două tipuri de fişiere pipe, fiecare generând anumite particularităţi ale modului în care se realizează comunicaţia:

1) **fişiere fără nume** care se crează cu directivă

Pipe (p)

Unde p reprezintă descriptorul de fişier astfel:

P [0] pentru citire

P [1] pentru scriere.

De remarcat că în blocurile alocate fişierelor (de regulă numai primele 10 blocuri de disc) alocarea se realizează conform algoritmului Round-Robin.

Scrierea măreşte dimensiunea fişierului iar citirea o micşorează; dacă un proces încearcă să scrie un articol ce depăşeşte capacitatea fişierului, nucleul execută transferul până la umplerea acestuia, apoi blochează procesul; când se citeşte din acest fişier se poate scrie din nou, dar s-ar putea să existe mai multe procese care aşteptau scrierea în fişier ducând la apariţia competiţiei.

Comunicaţia prin fişier pipe fără nume se poate realiza numai între un proces şi fiii săi, deoarece fişierul pipe nu se identifică decât prin descriptorul său şi intrările în tabela cu fişiere deschise în zona de procese ale utilizatorului, aceasta fiind moştenită de către fiii săi; astfel, zonele de date ale proceselor nu pot fi partajate în mod normal.

2) **Fişierele pipe cu nume** oferă câteva facilităţi comparativ cu fişierele pipe fără nume:

- se crează într-un director ca orice fişier obişnuit prin directiva mknod cu tipul fişierului 010000;
- la deschiderea să prin directiva open, se poate specifica blocarea procesului ce încearcă să deschidă pentru citire un fişier pipe nedeschis anterior pentru scriere;
- se va şterge explicit prin directiva unlink;
- datorită unui nume de identificare este permisă accesarea lui de către orice procese independente.

6.3. Comunicarea prin zone de memorie comune

Zonele de memorie comune permit proceselor să utilizeze în comun, o zonă de memorie prin transferul unei zone din spaţiul virtual al proceselor; accesul este controlat integral de programator.

O *zonă de memorie comună* este definită prin:

- identificator;
- segment de memorie;
- o structură de date ce include:
    - numărul de identificare al utilizatorului care a creat zona, al utilizatorului curent şi al grupului;
    - dimensiunea segmentului;
    - identificatorul procesului care a accesat zona de memorie;
    - numărul de procese care partajează această zonă;
    - alte informaţii.

6.4. Comunicarea prin cozi de mesaje

Principiul de comunicaţie prin intermediul cozilor de mesaje constă în comunicarea prin intermediul unei cozi circulare de mesaje gestionată de nucleul sistemului de operare.

Transmiterea se realizează cu sincronizare, deci procesele ce solicită primiri de mesaje se vor bloca până ce recepţionează mesajul solicitat, iar cele ce doresc să emită, se vor bloca dacă nu există nici un proces care să recepţioneze iar coada de mesaje este plină.

Mesajele ce se transmit sunt caracterizate printr-un tip (reprezentat printr-un număr la alegere), iar recepţia se poate face prin specificarea tipului.

Capitolul 7: Gestiunea fişierelor speciale

Toate operaţiile de intrare/ieşire sunt privite ca operaţii cu fişiere, Shell nerecunoscând existenţa unităţilor periferice, deci Kernel ascunde unităţile fizice reale sub forma unor fişiere aparenţe.

Fiecare unitate fizică este înzestrată cu un program special *(driver)* care translatează acţiunile solicitate de unităţile virtuale astfel încât utilizatorul poate comunica cu orice unitate ataşată sistemului fără nici o deosebire, dacă aceasta dispune de driverul corespunzător.

Deoarece aceste fişiere sunt deosebite de fişierele utilizator, ele sunt referite ca **fişiere speciale** de tip **caracter** sau **bloc**, fiind recunoscute două tipuri de unităţi periferice (disc şi terminal) a căror inod-uri conţin:

- numărul major - indică tipul dispozitivului;
- numărul minor - indică numărul unităţii.

7.1. Fişierele disc

Fişierele disc sunt organizate fizic în blocuri de 512 bytes, fiecare fişier având alocat un număr întreg de blocuri. Blocurile nu trebuie să fie neapărat contigue pe disc, spre deosebite de alte sisteme ce impun acest lucru. Atribuirile standard pentru fişiere implicit sunt:

- 0 fişier standard de intrare - tastatură;
- 1 fişier standard de ieşire - ecranul terminalului;
- 2 ecranul de intrare al terminalului, ce serveşte gestionării mesajelor de eroare şi a informaţiilor de diagnostic.

Identificarea fişierelor se realizează prin:

- intrările din directori care identifică fişierele printr-un număr de index asociat fiecărui fişier;
- numărul de index pentru orice volum fizic, este punctator într-o altă tabelă rezidentă pe volum *(index-listă)*;
- intrările în lista de indecşi ale fiecărui volum, conţin un număr de intrări numite *inod*-uri, motiv pentru care lista de indecşi se mai numeşte tabelă de *inod*-uri;
- un *inod* conţine o serie de informaţii cu privire la fiecare fişier:
    - numărul de identificare al uşer care a creat fişierul;

- starea de protecţie a fişierului (*ReadOnly, Open*, etc);
- 13 cuvinte ce arată blocurile unităţii ocupate de fişier;
- dimensiunea fişierului;
- data la care a avut loc ultima modificare;
- de câte ori a fost referit în directori;
- biţii de identificare a directorilor, fişierelor speciale şi a fişierelor mari.

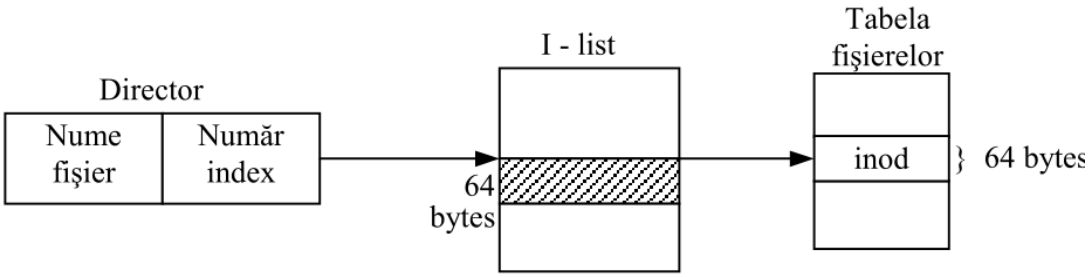

Fig 7.1.1. Identificarea unui fisier pe disc

- Corespondenţa nume fişier-index este disponibilă prin comanda **ls-i** şi se numeşte legătură *(link)*, aceasta servind drept mijloc de identificare şi gestiune a fişierului; alte comenzi ce afectează legăturile:

- *mv* - mută legătura;
- *ln* - crează o legătură;
- *rm* - şterge o legătură;

Când se lucrează cu un fişier în program, în RAM există o altă structură ce aparţine nucleului *(File Table)* unde se copiază un inod din *i-LIST*, dar zona alocată în memorie va avea 64 B ai inod-ului plus o zonă ce conţine informaţii necesare în timpul lucrului (pointeri de citire/scriere, contor pentru procesele ce folosesc fişierul, etc.).

Ca şi în MS-DOS, fişierele se identifică în program printr-un *FD (File Descriptor)*. Fiecare proces are o zonă de informaţii în care se păstrează toate *FD* - urile pentru fişierele deschise în timpul lucrului şi o zonă de cod a procesului neaparţinând nucleului ci în *UOF (User's Open File).*

Accesul la sistemul de fişiere se face la trei nivele:

1) prin utilitare - nivelul 1;
2) prin funcţii de bibliotecă <stdio. H> - nivelul 2 care apelează 3);

3) prin apeluri de sistem - nivelul 3: *open, close, read, write* prin care se gestionează fişierele la cel mai scăzut nivel.

Pentru reducerea numărului operaţiilor de intrare/ieşire la periferice ce lucrează la nivel de bloc, nucleul organizează în memoria internă o zonă de tampone cache (structuri de date software) diferite de memoria cache.

Tampoanele cache sunt organizate în două liste dublu înlănţuite (unele putând face parte din ambele); lista tampoanelor în uz este organizată conform unei funcţii de dispersie *(hashing)* ce ţine seama de numărul logic al dispozitivului periferic şi numărul blocului al cărui conţinut se încarcă
În zona de date ataşată tamponului. Funcţiile sunt apelate de nivelul inferior al sistemului de gestiune al fişierelor (SGF).

Bibliografie

1. Gheorghe D, Oancea B, Vasilescu A, ed. Editura Economica, 2003
2. McMullen J, Advanced UNIX user's handbook, Prentice Hall PTR, 2000
3. Tanenbaum A, Computer Networks, 3rd ed, PrenticeHall, 1996
4. Sistemul de operare UNIX , unibuc.ro
5. Teaching Unix, www.ee.surrey.ac.uk
6. Kernel_(computing) , wikipedia.org