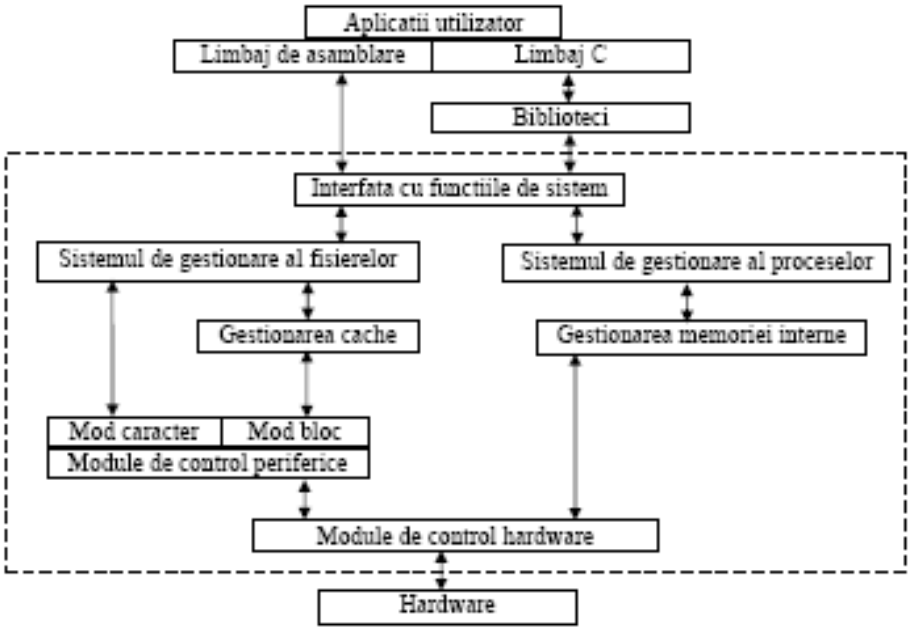
Structura ierarhizata a unui sistem de calcul



Functii din biblioteca standard de I/E

Functii ce se **incorporeaza programelor utilizator(spre deosebire de apelurile sistem)** dar care fac **uz de apelurile sistem** puse la dispozitie de sistemul de operare suport.

Scop: pun la dispozitie **metode de acces la fisiere** care sa fie: **eficiente, flexibile, portabile.**

Eficienta: **mecanism de tampoane** invizibil utilizatorului care sa reduca la minim atit numarul de accese la fisier cit si numarul de apeluri sistem.

Flexibilitatea: numarul mare de functii care permit **operatii mai complexe decit apelurile sistem de baza read si write.**

Portabilitatea: functiile din biblioteca standard de I/E nu sunt legate de particularitatile sistemului Unix.

Functii din biblioteca standard de I/E

Fluxuri si structuri FILE. Utilizarea tampoanelor

La apelurile sistem pt **fisiere** referirea se face utilizind **descriptorii de fisiere**

Pt **biblioteca I/E** conceptul similar descriptorului e **fluxul(stream)**

Cind se deschide sau se creaza un fisier prin functii din biblioteca de I/E se asociaza fisierului un flux

Functia de deschidere a unui fisier fopen returneaza un pointer la o structura FILE care reprezinta fluxul. Ac structura definita in <stdio.h> contine toate info necesare pentru gestionarea fluxului

Functii din biblioteca standard de I/E

Gestionarea fluxului prin **structura de info** continute in: **<stdio.h>**

- Descriptorul de fisier;
- Pointer la un tampon asociat fluxului;
- Dimensiunea tamponului;
- Numarul de caractere momentan prezente in tampon;
- Fanion de eroare etc;

Utilizatorii functiilor de biblioteca nu vor avea nevoie de acces direct la continutul unui **FILE**; o asemenea structura se transmite ca parametru diverselor functii de biblioteca aceste fiind implementate dependent de forma concreta a structurii FILE.

OBS Pt multe aplicatii, utilizarea **functiilor din biblioteca** standard de I/E e de preferat utilizarii **directe** a **apelurilor sistem**.

Tipuri de utilizare a tampoanelor:

1. Utilizare completa

- O op fizica de transfer are loc numai cind un tampon de I/E e plin sau gol.

- Utilizata de fisierele care au ca suport discul. -Tamponul folosit e obtinut de functiile de I/E la momentul primei op de I/E prin apelul **malloc**.

- Scrierea pe suport extern a continutului unui tampon se numeste **golire (flushing)** si poate avea loc automat: la **umplerea tamponului sau explicit** prin **apelul functiei fflush**.

2. **Tampon la nivel de linie.** In acest caz, operatia de I/E e c-data de caracterul **return.** Se foloseste pt comunicarea cu terminalele.

   Cererea de info de intrare de la un flux cu tampon la nivel de linie **va cauza golirea tampoanelor** tuturor fluxurilor de iesire cu tampon la nivel de linie.

3. **Fara tampoane**. In acest caz functiile din biblioteca de I/E executa direct operatiile de I/E, indiferent de numarul de octeti implicati. Fluxul standard de eroare lucreaza in acest mod **=>mesajele de eroare se afiseaza cit mai rapid posibil.**

CONCLUZIE:

-Fluxul standard de eroare nu foloseste tampoane;

-Toate celelalte fluxuri folosesc tampoane la nivel de linie daca se refera la un terminal, respectiv tampoane complete in alte cazuri.

Schimbarea modului de lucru cu tampoanele: **setbuf, vsetbuf.**

Obs: se apeleaza dupa ce s-a deschis fluxul, pt ca folosesc structura FILE dar inainte de efectuarea operatiei de transfer asupra fluxului.

#include <stdio.h>

void setbuf(FILE *_fp_, char *buf);

int vsetbuf(FILE *_fp_, char *_buf_, int _mode_, size_t _size_);

setbuf: specifica doar daca se utilizeaza sau nu tampoane.

Arg ***buf*** indica spre un tampon de lungime **BUFSIZE** (const definita in <stdio.h>)

Dezautorizarea utilizarii tampoanelor se face daca argumentul ***buf*** primeste valoarea **NULL**.

vsetbuf: se specifica **modul de lucru dorit** prin argumentul ***mode***:

_IOFBF tampoane complete;

_IOLBF tampoane la nivel de linie;

_IONBF fara tampoane.

In cazul lucrului fara tampoane ***buf*** si ***size*** sunt ignorate.

In celelalte cazuri aceste arg pot specifica optional un tampon si dimensiunea sa.

Daca ***buf*** este **NULL** biblioteca isi va aloca automat tampon de dimensiune corespunzatoare fluxului.

Deschiderea si inchiderea fluxurilor

#include <stdio.h>

FILE *fopen(const char *_pathname_, const char *_type_);

FILE *freopen(const char *_pathname_, const char *_type_, file *_fp);

FILE *fdopen(int _filedes_, const char *_type_);

1. **fopen** deschide un fisier dat prin nume de cale.

2. **freopen** deschide un fisier dat asociat cu un flux dat, inchizind mai intai acel flux daca el este deja deschis. Se foloseste de obicei pt a deschide un fisier dat pt unul din fluxurile predefinite(intrare standard, iesire standard sau eroare standard).

3. **fdopen** utilizeaza un descriptor de fisier deja existent in program (obtinut prin apelurile sistem open, dup, dup2, fcnt1, sau pipe) si asociaza un flux cu acel descriptor.

   Se foloseste frecvent cu descriptorii pt **pipe-uri** sau pt comunicarea in retea **(sockets).** Cum aceste tipuri de fisiere nu pot fi deschise cu **fopen, pt ca nu au nume de cale, trebuie pornit de la descriptorii de fisier.**

Valori pentru argumentul ***type*** conform standardului ANSI C

type	**Descriere**
r sau rb	deschidere pt citire
w sau wb	trunchiaza la lung 0 sau creaza pt scriere
a sau ab	adauga;
r+ sau **r+b** sau **rb+**	deschidere pentru citire si scriere;
w+ sau **w+b** sau **wb+**	trunchiaza la zero sau creaza pt citire sau scriere;
a+ sau **a+b** sau **ab+**	deschide sau creaza pt citire si scriere la sfirsitul fisierului.

Obs. Caracterul **b** face diferenta intre **fisiere binare** si **celelalte**;

Daca un fisier are semnul **+** in type(deschis pt scriere si citire) se aplica urmatoarele restrictii:

- o scriere (iesire) nu poate fi urmata direct de citire (intrare) ci trebuie sa intervina mai intai una din operatiile: **fflush**, **fseek**, **fsetpos**, sau **rewind**.
- o citire nu poate fi urmata de scriere decit dupa o operatie **fseek**, **fsepos** sau **rewind**.

Inchiderea unui flux se face cu functia:

```
#include <stdio.h>
int  fclose (FILE *fp);
```

Datele de iesire in cazul utilizarii tampoanelor vor fi automat golite inainte de inchiderea fluxului, dar daca mai exista date de intrare in tampon acestea se pierd. Daca tamponul fluxului a fost alocat automat el va fi eliberat, tot ca parte a tratarii **fclose**.

La terminarea normala a unui proces se produce golirea tampoanelor fluxurilor standard si inchiderea acestor fluxuri.

Operatii de transfer cu fluxurile de I/E

Operatii de transfer la 3 niveluri: **de caracter**, **de linie**, sau **directe**.

Operatii la nivel de caracter:

Functiile de citire:

```
#include <stdio.h>
int  getc (FILE *fp);  - poate fi implementata ca macroinstructiune.
int  fgetc (FILE *fp); - e functie, are adresa si adresa sa poate fi
                         argument  pt alta functie
int  getchar (void);   - echiv cu getc din fluxul standard de intrare
```

Valoarea returnata de functii este definita drept **int** nu **char**, cum ar parea normal. **=>**Se poate returna orice valoare de caracter si in plus si o indicatie de eroare sau ajungerea la sfirsitul fisierului (**EOF** definita in **<stdio.h>** ca **-1**). Pt a vedea daca e vorba de o eroare pr-zisa sau de sfirsit de fisier, mai trebuie folosite functiile ferror sau feof:

#include <stdio.h>

int ferror (FILE *fp);

int feof (FILE *fp);

Obs Ambele functii returneaza o valoare diferita de zero (**true**), daca e adevarata conditia testata, respectiv 0(**false**) in caz contrar.

In structura FILE exista 2 fanioane pentru flux. Acestea pot fi sterse prin functia:

#include <stdio.h>

int clearerr (FILE *fp);

Pt scrierea la nivel de caracter:

#include <stdio.h>

int putc (int *c*, FILE *fp);

int fputc (int *c,* FILE *fp);

int putchar (int *c*); echivalent cu putc(c, stdout)

Operatii la nivel de linie

Functiile: **fgets, gets, fputs** si **puts.**

Operatii pentru citire:

#include <stdio.h>

char *fgets(char **buf*, int *n*, FILE **fp*); - citeste din fluxul dat de al-II-lea argument; trebuie specificata si dimensiunea tamponului. Citeste pina la urmatorul caracter newlinie dar nu mai mult de n-1 caractere.

Ultimul caracter din tampon va fi octetul nul.

char *gets(char **buf*); -citeste din fluxul standard de intrare; permite depasirea tamponului, scriind restul in locatiile urmatoare de memorie; Risc de securitate.

OBS Ambele specifica adresa unui tampon in care se va depune linia citita

Operatii pentru scriere:

#include <stdio.h>

int fputs(const char **str*, FILE **fp*); - scrie un flux terminat cu octetul nul in fluxul specificat(acesta nu e scris)

int puts(const char **str*);- scrie sirul in fluxul de iesire standard si forteaza un caracter newline la sfirsit

Operatii binare

Conventional se numesc **I/E binare operatiile de transfer cu perifericele prin functii din biblioteca standard de I/E care permit transferuri de orice lungime.**

#include <stdio.h>
size_t fread(void *_ptr_, size_t _size,_ size_t _nobj_ , FILE *_fp_);
size_t fwrite(const void *_ptr_, size_t _size,_ size_t _nobj_ , FILE *_fp_);

Utilizari ale acestor functii

1.Citirea sau scrierea elementelor unui tablou binar

2.Se pot folosi numai in cadrul aceluiasi sistem

Exemplu 1.

```
#include<stdio.h>
int data=6;
size_t ret;
FILE* flux;
flux=fopen("/geta/laboratoare/ex1.txt", r+);
ret=fwrite(data, sizeof(int), 1, flux);
if (ret != 1)          printf("scrierea nu a avut succes")
fclose(flux);
```

FILE *fopen(const char *_pathname_, const char *_type_);

I/E cu format

Tiparirea cu format.

Functii

#include <stdio.h>
int printf(const char *_format_)
int fprint(FILE *_fp_, const char *_format_,…);
int sprintf(char *_buf,_ const char* _format_,…);

Primele 2 functii returneaza numarul de caractere scrise la executia cu succes, respectiv o valoare negativa in caz de eroare.

Functia **sprintf** returneaza numarul de caractere memorate in tamponul *buf*.

a)printf- scrie in fisierul standard de iesire; fprintf scrie in fluxul specificat de primul parametru.

b)sprintf- scrie in memorie in tamponul *buf* dar respectind formatul precizat prin sirul *format*.

Obs exista un alt set de functii pt tiparire, numite **vprintf, vfprintf** si **vsprintf** in care lista de argumente variabile e inlocuita cu un argument de tip va_list.

Tiparirea cu format

Functii

Pentru citire se folosesc functiile:

#include <stdio.h>

int scanf(const char *format)

int fscanf(FILE *_fp,_ const char *_format_,…);

int sscanf(const char* format,…);

Obs –primele 2 opereaza asupra fisierelor, iar a treia transfera date in memorie.

Deosebiri esentiale fata de functiile similare pt scriere:

a) Argumentele din lista de arg variabile trebuie sa fie pointeri, pt ca reprezinta locatii la care se depun datele citite;

b) Valoarea returnata de aceste functii nu e nr de octeti transferati ci numarul de valori atribuite prin citire.

Apeluri sistem pentru gestionarea proceselor

- Sistemul de operare UNIX suport pt:
  - Specificarea unor conditii la executia unui program (proces)
  - Crearea si gestionarea prceselor
  - Facilitati pentru realizarea comunicarii intre procese
- In ac capitol:
  - Executia unui singur proces
  - Gestionarea proceselor

Apeluri sistem pentru gestionarea proceselor

Lansarea si terminarea unui program C sub UNIX

Apelul sistem exec- pt a incepe executia unui program

Tratarea apelului include:

- **cautarea fisierului** care contine cod executabil
- **incarcarea in memorie a continutului** (partial sau total) al fisierului
- **cedarea controlului** catre acest program

Compilatorul C si **editorul de legaturi** adauga programului o **functie speciala** numita ***rutina de start C,*** care:

- va primi controlul la pornirea programului din care va fi apoi apelata functia **main** a programului;
- preia de la nucleul UNIX **argumentele din linia de c-da** prin care este lansat programul, precum si **variabilele de ambianta**.

O schita a modului in care decurge executia programului:

Main - apeleaza alte functii definite de utilizator;

- solicita apeluri sistem;

Terminarea executiei se poate face normal sau anormal.

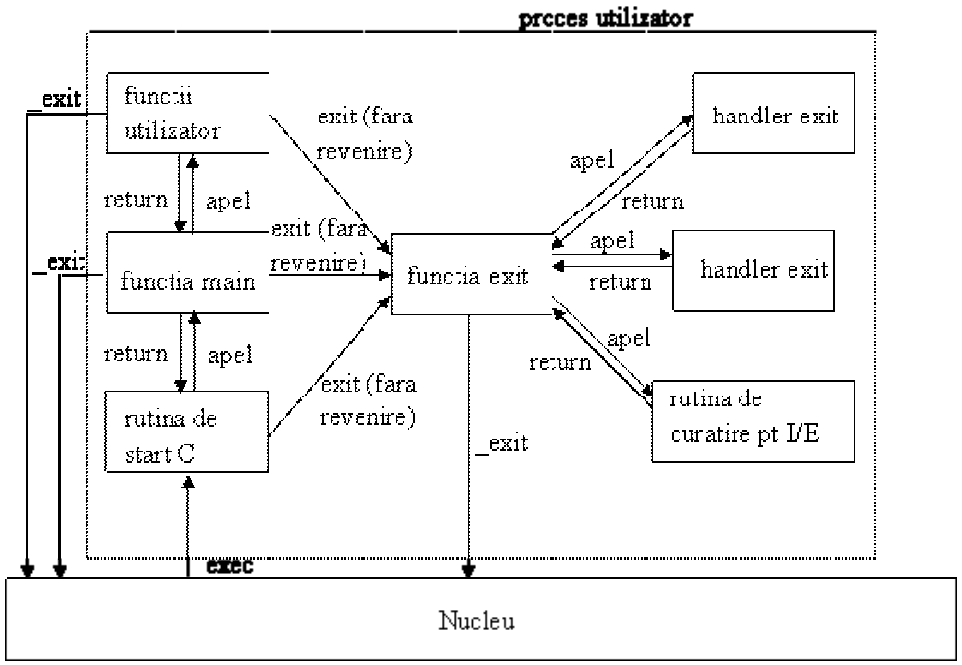
Terminarea normala poate fi realizata prin:

- **revenire din functia main**;
- apelul sistem **exit**;
- apelul sistem **_exit**;

Terminarea anormala se produce cind:

- se apeleaza **abort;**
- **procesul primeste un semnal;**

Lansarea in executie si terminarea unui program C



Apelurile sistem **exit** si **_exit**

#include <stdio.h>

void exit(int status);

#include <unistd.h>

void _exit (int status);

Argumentul: - un intreg prin care programul transmite starea de terminare a procesului;

- 0(sau alte valori … 127) la exit ↔ terminare normala;

exit: asigura la terminarea programului tratarea corespunzatoare a operatiilor de introducere/extragere inca in curs, golirea tampoanelor, inchiderea fisierelor inca deschise;

_exit: reda controlul nucleului;

Obs: se pot utiliza functii speciale care sa fie apelate de catre exit pt efectuarea unor operatii specifice fiecarui program.

#include <stdio.h>

int atexit(void (*func) (void));

Fiecare apel la **atexit** inregistreaza cite o functie al carui nume figureaza ca parametru la apel.

Transmiterea argumentelor si a ambiantei

Functia **main**

- **preia argumente din linia de c-da**. Numarul acestora (inclusiv numele fisierului cu codul executabil al programului fiind dat de argumentul **argc** al lui main, iar valorile argumentelor sub forma unor siruri de caractere se gasesc in tabloul de caractere **argv**.
- pt variabilele de mediu se foloseste o variabila globala standard, **environ**. Valoarea acestuia este un pointer care indica inceputul unui tablou de pointeri spre siruri de caractere Acestea sunt de forma *nume*=*valoare* si reprezinta variabilele de mediu.

prg1.c:

```
#include <stdlib.h>
int main ( int argc, char *argv[ ])
        { /*int argc si char…sunt arg functiei*/
        int i;
        for (i=0; i<argc; i++)
        printf ("argv[%d] : %s\n", i , argv[i]);
        }
```

gcc –o prg1.c prg1- transformarea in executabil !!!!!

$./prg1 miu1 miu2 - lansare in executie:

argv[0]: prg1

argv[1]: miu1

argv[2]: miu 2

Accesul la variabilele de mediu: getenv si **putenv**

```
#include <stdlib.h>
char *getenv(const char *name);
int putenv(const char *str);
```

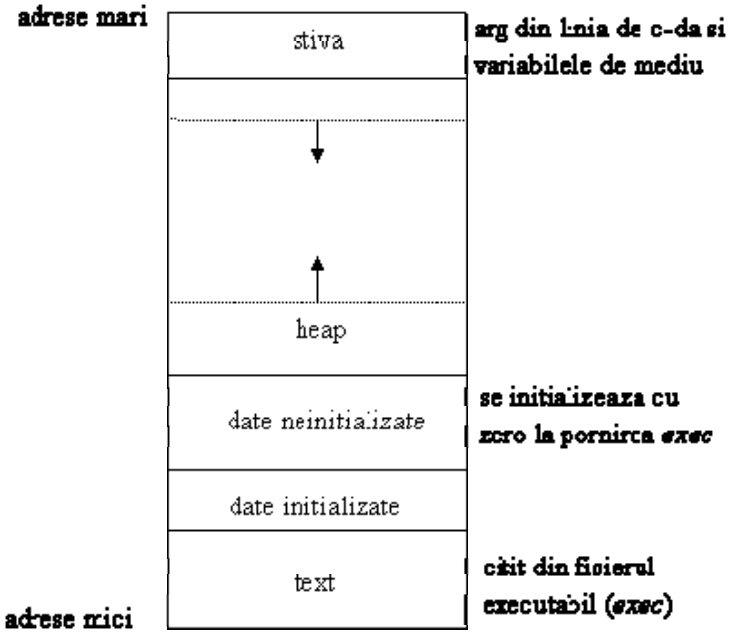
Obs

Prima functie primeste ca argument numele unei variabile de mediu si returneaza valoarea acesteia sau un pointer nul daca variabila cu acel nume nu exista.

Prin putenv se poate defini o variabila de mediu, argumentul str trebuind sa fie : nume=valoare

Elemente de gestionarea memoriei

Segmentele unui program C



Segmentul text - instructiunile programului – in limbaj masina
- acces numai pt citire
- poate fi partajat

Segmentul de date initializate

- contine variabilele **initializate explicit in textul sursa al programului**, in afara oricarei functii(variabilele initializate in functii apar in stiva)

Segmentul de date neinitializate

- rezervat pt variabilele declarate in afara functiilor, fara a fi initializate.

Stiva - in aceasta zona se memoreaza variabilele declarate in interiorul functiilor precum si info necesare pt revenirea din apelurile la functii

Zona heap – utilizata pt variabilele alocate dinamic.

OBS - in fisierul executabil sunt memorate numai segmentul de text si cel al datelor initializate. Pt segmentul datelor neinitializate este suficient sa se memoreze **dimensiunea necesara**, pt a putea face rezervarea la incarcarea programului.

Biblioteci partajate

<=> Nu mai e necesar sa se includa in fiecare program la editarea de legaturi functiile de biblioteca apelate, ci e suficient ca in memorie sa se incarce o copie a acestora la care fac acces toate programele care au specificata optiunea de utilizare a bibliotecii partajate.

UTILIZAREA partajarii se indica prin optiuni la compilare (actioneaza in realitate la editarea de legaturi): pt compilatorul **gcc** optiunea este **-share**.

Apelurile sistem **setjmp** si l**ongjmp**

Posibilitatea ca **dintr-o functie sa fie efectuat un salt in alta functie.**

#include

int setjmp(jmp_buf *env*); (1)

void longjmp(jmp_buf *env*, int *val*); (2)

(1) – returneaza 0 daca este efectuat corect, respectiv ceva dif de zero daca revine dintr-un apel la longjmp;

- apelul se face din locatia in care urmeaza sa se faca revenirea;

- argumentul *env* este un fel de tablou in care se pastreaza toate info necesare pentru a restabili starea stivei de apeluri la logjmp.

Cind se intilneste o eroare se face apel la **longjmp.**

(2) - 1arg-setjmp

- al-II-lea arg permite ca pt acelasi setjmp sa se utilizeze mai multe longjmp. La revenirea din setjmp se va putea verifica de unde a fost efectuat apelul longjmp si proceda in consecinta.

CREAREA SI TERMINAREA PROCESELOR

Unix ofera suport la nivelul apelurilor sistem atit pentru specificarea unor conditii la executia unui program (proces) cit si pentru **crearea si gestionarea proceselor** ca si pentru realizarea comunucarii intre procese.

Tabela de procese: gestionarea proceselor

- O INTRARE PT FIECARE PROCES
- Accesibila doar nucleului
- dimensiunea impusa la crearea nucleului → nr limitat de procese

O intrare in tabela de procese contine urmatoarele cimpuri:

- starea procesului
- localizarea procesului in memoria interna sau cea utilizata pentru swaping
- identificatori ai utilizatorului si ai grupului
- identificatorul procesului
- parametrii de planificare pentru obtinerea procesului
- semnale transmise procesului dar netratate

Mai exista:

- zona de memorie atasata fiecarui proces continand informatii referitoare la executia procesului
- 2 tabele(tabela regiunilor per sistem si tabela regiunilor per proces) care contin adrese, dimensiuni ale zonelor de memorie implicate si sunt utilizate pt a gestiona memoria.

UNIX- PROCES- PID

Procese speciale(procese sistem sau procese utilizator)- incep la conectarea sistemului si se termina la deconectarea lui =>procese permanente:

- Proc cu identificatorul 0- swapper responsabil cu planificarea in executie a proceselor (proces sistem). Este creat in momentul incarcarii nucleului.

- Proc cu identificatorul 1- creat la sfirsitul incarcarii nucleului pt a executa programul /etc/init sau /sbin/init. Este un proces utilizator dar ruleaza cu privilegii de root si are rolul de a aduce sistemul intr-o stare initiala cunoscuta.

Procesul init - deteremina configuratia sistemului si face prin /bin/login sa se afiseze promperul pt fiecare terminal.

- devine parintele tuturor proceselor orfane

- Proc cu identificatorul 2 - proces sistem (pagedaemon- la unele implementari) si este responsabil de suportul pentru memorie virtuala.

Aflarea **pid-ului** procesului curent si a **pid-ului** parinte a acestuia.

```
#include <sys/type.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Crearea unui nou proces- apelul sistem fork ()- mai putin swapper

```
#include <sys/type.h>
#include <unistd.h>
pid_t fork(void);
```

Efect: - returneaza **-1** in caz de eroare;

- la un apel reusit dupa punctul de revenire exista un proces nou-proces fiu al procesului care a facut apelul la **fork**;
- **functia returneaza un tip specific UNIX**, definit in **types.h**
- returneaza procesului parinte **pid-ul fiului**, iar in fiu returneaza **0**.
- este o copie a parintelui;

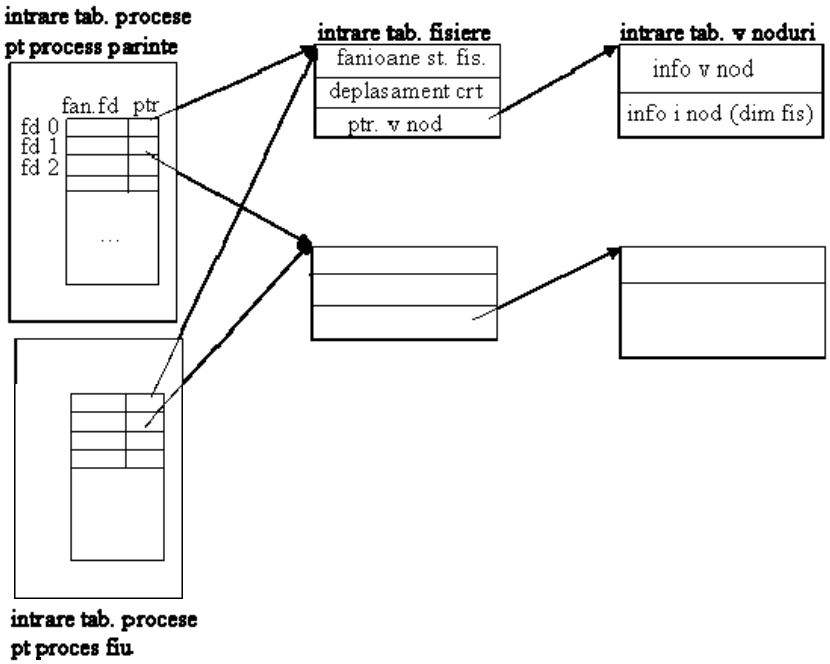
Obs

Procesul fiu este o copie a procesului parinte (cu unele optimizari de copiere). Segmentul text e partajat de procesele parinte si fiu iar copierea celorlalte segmente se va face numai atunci cind unul din procese va solicita o operatie.

Nu este posibil sa se cunoasca daca dupa fork **procesul fiu** ajunge sa fie executat inaintea procesului parinte sau situatia va fi inversa. Acest lucru este dependent de algoritmul de planificare a proceselor folosit in fiecare implementare UNIX.

Daca e necesar sa se garanteze o ordine intre executie trebuie folosite diverse mijloace de sincronizare.

Dupa **fork** toti descriptorii de fisiere ai parintelui sunt duplicati in fiu. Parintele si fiul partajeaza aceeasi intrare in tabela de fisiere deschise!!!!!



OBS. Dupa fork toti descriptorii de fisiere ai parintelui sunt duplicati in fiu.
Parintele si fiul vor partaja aceeasi intrare in tabela de fisiere deschise.
Daca atit parintele cit si fiul scriu la acelasi descriptor fara sincronizare, rezultatele vor fi intercalate.

Tratarea descriptorilor fisierelor deschise de parinte dupa fork:

a) Parintele asteapta ca procesul fiu sa se termine cu ajutorul apelului sistem **wait.** Chiar daca fiul va actualiza indicatorii de prelucrare in fisiere, operatiile nu vor fi intercalate.

b) Fiecare proces stabileste cu ce fisiere va lucra si inchide descriptorii fisierelor pe care nu le va utiliza. Se elimina interferentele actiunilor celor 2 procese asupra fisierelor.

Procesul fiu mai mosteneste de la parinte o serie de alte resurse:

- identificatorii reali si efectivi pt utilizator si grup;
- catalogul curent;
- masca pentru crearea fisierelor;
- masca pt semnale si modul de tratare al acestora;
- variabilele de mediu;
- terminalul de control;

Proprietati care nu se mostenesc:

- lacatele de fisiere puse de parinte;
- alarmele in asteptare pentru parinte;
- semnalele in asteptare la parinte.

Situatii posibile de esec la un apel **fork**:

a) daca in sist sunt prea multe procese create de o stare de eroare: un ciclu infinit in care se creaza procese;

b) daca se depaseste nr maxim de procese acceptat in sistem de la un anumit utilizator.

Situatii care reclama utilizarea lui **fork**:

a) **cind un proces trebuie sa se duplice** astfel incit parintele si fiul sa poata executa simultan portiuni de cod diferite ale aceluiasi program: situatie tipica pt serverele concurente din aplicatiile de retea dezvoltate dupa modelul client-server: dupa acceptarea unei cereri, parintele creaza un fiu care trateaza cererea, iar parintele se reintoarce la punctul unde accepta noi cereri.

b) cind un proces doreste sa lanseze in executie un alt program, simultan cu continuarea propriei sale executii; situatie tipica pentru interpretoarele de c-zi, care vor crea cite un proces pt executia fiecarei c-zi.

Structura de program posibila pentru a utiliza **fork()**

```
if ((pid=fork())<0)// testeaza daca functia fork() se executa corect
        {
        perror("Eroare");// afisaza mesajul de eroare
        exit(1);
        }
if(pid==0)
        {
        /*codul fiului*/
        exit(0);// terminare corecta a procesului fiu
        }
        /*codul parintelui*/
```

```
#include <stdio.h>
#include <sys/types.h>
int main(void)
{
        pid_t pid ;
        pid=fork() ;
        printf("Test de apel fork:%d\n", pid);// Instructiune executata atat de fiu cat si de
                                                //parinte
}
```

Actiuni realizate la apelul functiei **fork**

-nucleul verifica daca exista resurse necesare pt crearea unui proces fiu

-se parcurge tabela proceselor si se gaseste o pozitie libera care se atribuie procesului fiu, obtinindu-se PID-ul ce va fi returnat

-se verifica numarul proceselor create de utilizator. Daca nu se depaseste nr maxim, procesul fiu va aparea ca fiind creat. Altfel se returneza eroare. Variabila errno poate avea valorile EAGAIN sau ENOMEM

-se completeaza intrarea in intrarea in tabela corespunzatoare fiului unele date preluate de la parinte: id-ul grupului de procese, si o val utilizata pt a calcula prioritatea sistemului Se mai adauga: id-ul parintelui, prioritatea initiala, timpul de utilizare CPU.

-se returneaza 0 in procesul fiu si id-ul fiului in procesul parinte.

Principalele diferente intre procesele parinte si fiu

- valoarea returnata de fork
- id-ul procesului si al procesului parinte
- pt fiu se reseteaza valorile ceasurilor care caracterizeaza existenta sa
- blocarile unor fisiere realizate de parinte nu sunt mostenite.

Se face diferenta intre codul executat in procesul fiu si cel executat in procesul parinte. La compilare trebuie utilizata erro.o – fisierul obiect pt prelucrarea erorilor

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

int gvar=4;

int main(void)
{
        pid_t pid;
        int var=7;
        printf("Inainte de fork\n");
        if((pid=fork()==-1)
                err_sys("Eroare fork");
        else if(pid==0)
        { /*procesul fiu*/
                gvar++;
                var+=2;
        }
        else
                sleep(2);
        printf("Proces (pid)=%d, gvar=%d var=%d\n", getpid(), gvar,var);
        exit(0);
}
```