Lucrarea de laborator Nr. 05

Procese. Mecanisme de comunicare între procese: pipe anonim, redirectare I/O, mecanisme pipe anonim

Aplicații demonstrative:

1. Utilizarea apelului *system*() pentru a starta comenzi shell cu transmiterea codului de retur.
2. Utilizarea apelului *dup2()* pentru a duplica un descriptor de fișier peste *stdout* și revenirea la situația inițială.
3. Crearea prin apel *fork()* a două procese și conectarea lor printr-un PIPE anonim (bufer de conectare) cu redirectarea fișierelor standard de intrare respectiv ieșire.
4. Crearea prin apel *system()* a două procese și conectarea lor printr-un PIPE anonim (bufer de conectare) cu redirectarea fișierelor standard de intrare respectiv ieșire.

Canale de comunicaţie. (Pipes)

Una dintre modalitățile de comunicare între *procese* (*proces* = instanța unui program) în Unix este cea prin intermediul **canalelor de comunicaţie** (*pipes*, în limba engleză). Practic este vorba despre o "conductă" (un bufer) prin care pe la un capat se scriu mesajele, iar pe la celălalt capat se citesc - deci este vorba despre o structură de tip coadă, adică o lista FIFO (First-In,First-Out).

Aceste canale de comunicație sunt de două categorii:

- *pipe*-uri anonime (interne): aceste "conducte" sunt create în memoria internă a sistemului Unix sub forma unor bufere de comunicare (aceste bufere se mai numesc şi pipe-uri interne);
- *pipe*-uri cu nume (externe): aceste "conducte" sunt fişiere de un tip special, numit *fifo*, deci sunt păstrate în sistemul de fişiere (aceste fişiere *fifo* se mai numesc şi pipe-uri externe).

În fapt un *pipe* este un mecanism de comunicare interproces; datele scrise într-un *pipe* de către un proces pot fi citire de către un alt proces

Utilizând facilităţile de **pipe** un programator poate să creeze transformări complicate prin direcţionarea fişierului (standard) de ieşire a unui proces către fişierul (standard) de intrare a altui proces şi aşa mai departe.

Canale de comunicaţie anonime (pipe-uri anonime).

Un *pipe anonim* este un bufer de comunicare Unix în care un proces asincron poate să scrie respectiv altul, sau acelaşi proces, poate să citească. *Pipe*-ul reflectă natura asincronă a comunicării între procese.
Un *pipe anonim* este temporar creat și el dispare în momentul în care procesul care l-a creat a dispărut.

Apeluri utilizate în aplicațiile demonstrative:

- **Funcții apelate: (detalii complete pe site)** `pipe(), system(), dup(), dup2()`

Funcţia *pipe()* - crează un bufer de comunicare/ canal anonim

Apelul sistem `pipe()` utilizat în programele C pentru a realiza o conectare prin *pipe anonim* are următorul prototip:

```
#include<unistd.h>

int pipe(int fd[2]);
```

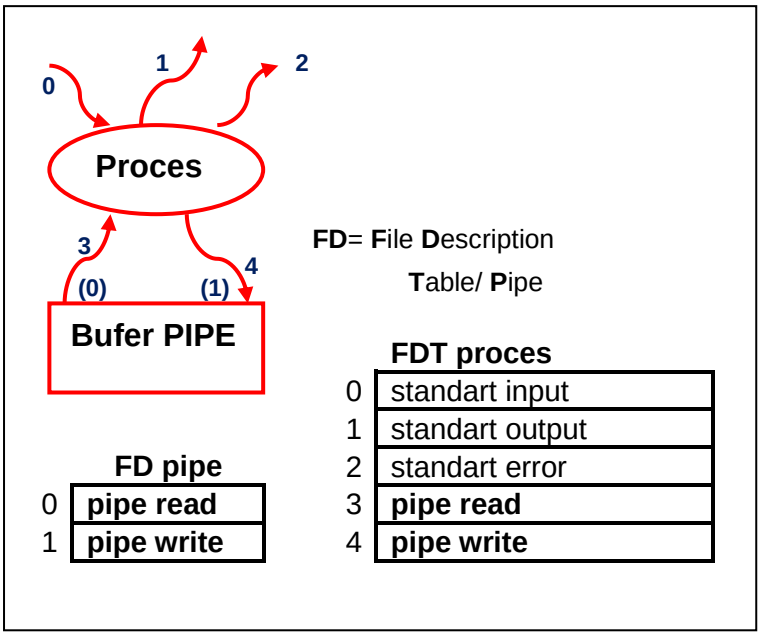
Fiecare proces Unix (mai puţin *daemonii*, similari serviciilor din Windows) are atașat cinci (3 + 2) *descriptori de fișier*, corespunzători celor trei *fişiere standard*, plus două, în urma unui apel *pipe(0)*, de *pipe*:

File descriptor-ii pentru acces *Low-level* care gestionează *standard input*, *standard output* şi *standard error* sunt `STDIN_FILENO`, `STDOUT_FILENO` şi `STDERR_FILENO` şi sunt definiţi în `unistd.h`.

File description table

Valoare întreagă	Nume	<unistd.h> constantă simbolică	<stdio.h> File pointer (stream) pt. High-Level
0	*Standart input*	*STDIN_FILENO*	*stdin*
1	*Standart output*	*STDOUT_FILENO*	*stdout*
2	*Standart error*	*STDERR_FILENO*	*stderr*
3	*Pipe read*		
4	*Pipe write*		

Apelul `pipe()` crează un bufer de comunicare- numit *Bufer pipe* în figura de mai jos, la care apelantul are acces prin descriptorii de fişier (file descriptors) `fd[0]` şi `fd[1]`.



Exemplu:

```
#define LMSG 16        //  - lungime mesaj
        pipe (int fd(2);
//- scrie date in  pipe
        write (fd[1], m1, LMSG);
sau
        write (4, m1, LMSG);

respectiv

//  - citeste date din  pipe
        read (fd[0], buffer, LMSG);
sau
        read (3, buffer, LMSG);
```

Datele trimise (scrise) către `fd[1]` sunt accesate (citite) prin `fd[0]` pe baza primul-sosit-primul-servit (`FIFO`). Un *pipe anonim* nu are un nume extern sau permanet, deci un proces poate accesa un *pipe anonim* numai prin cei doi descriptori de fişier. Din aceasta cauză un *pipe anonim* poate fi utilizat numai de către *procesul* apelant (care a creat pipe-ul) şi de către descendenţii acestuia, care la apelul funcției `fork()` moştenesc descriptorii de fişier din *FD pipe* respectiv *FDT* (funcția `fork ()` crează un proces).

Funcţia *dup()* - crează o copie a unui descriptor de fișier utilizând un număr de descriptor furnizat de utilizator (oldfd). Apelul `dup()` crează o copie a descriptorului *oldfd*. Descriptorul *newfd* rezultat în urma copierii va fi cel mai mic număr liber de descriptor de fișier.

Apelul sistem `dup()` utilizat în programele C realizează o redirectare și are următorul prototip:

```
#include<fcntl.h>

int newfd= dup(oldfd);
```

unde

oldfd -este vechiul descriptor de fișier
newfd -este copia rezultată în urma apelului *dup()*

În caz de succes, dupa apelul *dup()*, *oldfd* și *newfd* pot fi utilizați alternativ

Funcţia *dup2()* - crează o copie a unui descriptor de fișier utilizând un număr de descriptor (newfd) furnizat de utilizator.

Apelul sistem `dup2()` utilizat în programele C realizează o redirectare și are următorul prototip:

```
#include<fcntl.h>

int dup2(oldfd, newfd);
```

unde

oldfd -este vechiul descriptor de fișier
newfd -este noul descriptor de fișier utilizat de dup2() pentru a crea copia

În caz de succes, după apelul *dup2(), oldfd* și *newfd* pot fi utilzați alternativ

Mai jos este o implementare C în care *newfd* este descriptorul fișierului de ieșire standard (STDOUT_FILENO) care are valoarea 1. Se realizează o redirectare a fișierului standard de iesire către fișierul fisier.txt

```
#include...
...
int main()
{
  int oldfd = open("fisier.txt",O_WRONLY | O_APPEND, );                    //  open low-level
  printf("scriu in standard output \n");                  // Toate instructiunile printf
  printf("scriu in standard output \n");                  //          vor scrie in ecran (fișier standard de ieșire

/* crearea unei copii de descriptor */
  dup2(oldfd, 1) ;                    //  newfd este descriptorul de fisier al lui STDOUT_FILENO (= 1)
                                      //  oldfd este descriptorul de fisier pentru fisier.txt
// echivalent cu:
// dup2(oldfd, STDOUT_FILENO)
```

```
  printf("scriu in fisier.txt \n");      // Toate instructiunile printf
  printf("scriu in fisier.txt \n");      //          vor scrie in fisier.txt
 ...
  return 0;
}
```

Funcţia *system()* – execută comanda shell specificată.

Apelul sistem **`system()`** crează un proces copil în care se va executa comanda shell specificată. Mecanismul este similar cu apelul *execl()* de mai jos:

```
execl("/bin/sh", "sh", "-c", shell-command, (char *) NULL);
```

Apelul sistem **`system()`** are următorul prototip:

```
#include <stdlib.h>

int system(const char *command);
```

Apelul sistem **`system()`** în caz de succes returnează *codul de retur* al procesului copil (comanda shell) startat. În caz de insucces se returnează -1.
Procesul apelant rămâne automat în *wait* până când comanda shell startată se termină. Cu alte cuvinte după apelul sistem **`system()`** nu este nevoie de un apel *wait()/waitpid()* pentru a aștepta terminarea comenzii shell startată.

```
#include...
...
int main()
{
int status;

status=system("pwd"); // pwd - cu revenire dupa executare - nu e nevoie de wait()/waitpid()
system("cd /etc;ls -l|head -n5"); // cd + ls,head  cu conectare pipe
system\                    /*  cd + ls,head , awk cu conectare pipe */
(\
"cd /etc;ls -l|head -n5|awk 'BEGIN {printf (\"permisiuni de acces\tnume aditionale\tmarime\"); printf(\" \\n\")} NR>1
{printf (\"%s\t\t%s\t\t%s\\n\", substr($1,2,9), $2, $5)}'"\
);
  return 0;
        }
```

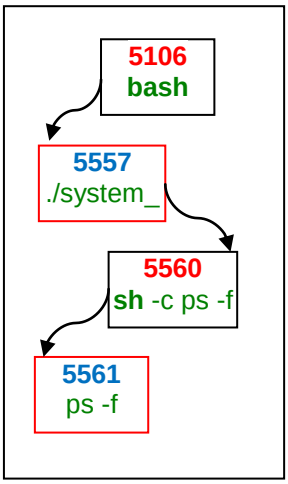
Aplicații demonstrative

1. Utilizarea apelului *system()* pentru a starta comenzi shell și transmiterea codului de retur.

- **Program system_.c**

```
/* system_.c */
/** Startare: system_      **/
/**
   Analizeaza direct modul de functionare al apelului system()
  - studiul proceselor create
  - codul de retur transmis de procesul copil
**/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
  int status;      // memoreaza starea/codul de retur al unui proces copil;
                   //          (se utilizeaza si la fork()/wait()/waitpid())
```

```
/* startare a 3 procese copil */
 // copil 1
  status=system("test -f /etc/nu_exista");// test daca fisierul nu_exista exista,
                                          // cu revenire si pozitionare cod retur pe eroare (>0, =256))
  /* nu e nevoie de wait(), se asteapta implicit terminarea copilului  */
  printf("\n\tcod retur(eroare)= %d", status);

 // copil 2
  status=system("test -f /etc/passwd");   // test daca fisierul passwd exista
                                          // cu revenire si pozitionare cod retur pe OK (=0_)
  /* nu e nevoie de wait(), se asteapta implicit terminarea copilului  */
  printf("\n\tcod retur OK= %d\n", status);

 // copil 3
  system("ps -f");                        // comanda ps creaza un proces copil- vezi output
                                          // ps -f afiseaza procesele active ale utilizatorului
  /* nu e nevoie de wait(), se asteapta implicit terminarea copilului  */

  exit(123);                  //cod de retur  proces=123


}
/* output
$ gcc -o system_ system_.c

$ ./system_

        cod retur(eroare)= 256
        cod retur OK= 0
UID        PID  PPID  C STIME TTY          TIME CMD
streian   5106  5096  0 08:57 pts/0    00:00:00 bash
streian   5557  5106  0 08:58 pts/0    00:00:00 ./system_
streian   5560  5557  0 08:58 pts/0    00:00:00 sh -c ps -f
streian   5561  5560  0 08:58 pts/0    00:00:00 ps -f

$ echo $?
123
$

*/
```



2. Utilizarea apelului *dup2()* pentru a duplica un descriptor de fișier peste stdout și revenirea la situația inițială.

- **Program dup2_cu_revenire.c**

```
/** Fisier: dup2_cu_revenire.c  **/
/*
  Utilizarea dup2() pentru a schimba stdout cu descriptor de fișier
  și revenirea la situația inițială.
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
  int oldfd = open("fisier.txt",O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU|S_IRWXG);
//  O_WRONLY se deschide fisierul in scriere
// O_CREAT daca fisierul nu exista, el se creaza
```

```
// O_TRUNC daca fisierul exista, se sterg toate datele din el
// S_IRWXU|S_IRWXG drepturi rwx la nivel de user + drepturi rwx la nivel de group
// oldfd este descriptorul de fisier, este o data de tip intreg

  int salv_out = dup(STDOUT_FILENO);          // Se salvează descriptorul de fisier
                                              // al lui STDOUT_FILENO (= 1)
  printf("oldfd= %d, newfd= %d salv_out= %d \n\n", oldfd, STDOUT_FILENO, salv_out);

  /* crearea unei copii de descriptor */
  dup2(oldfd, STDOUT_FILENO) ; //  newfd este descriptorul de fisier al lui STDOUT_FILENO (= 1)
                               //  oldfd este descriptorul de fisier pentru fisier.txt
                               // din acest moment oldfd este echivalrnt cu STDOUT_FILENO
//  dup2(oldfd, 1);          //  se posate scrie si asa
  printf("1.scriu in fisier.txt \n");         // Toate instructiunile printf care scriu pe STDOUT_FILENO
  printf("2.scriu in fisier.txt \n");         //      vor scrie in fisier.txt
  write(STDOUT_FILENO,"3.scriu in fisier.txt \n", 23);  // scriu in fisier.txt in loc sa scriu pe ecran
  write(oldfd, "4.scriu in fisier.txt \n", 23);                    // scriu in fisier.txt

/* revenire la situația inițială */
  dup2(salv_out,STDOUT_FILENO);                    // revenirea la STDOUT_FILENO
  printf("5.scriu in STDOUT_FILENO \n");           // Toate instructiunile printf
  printf("6.scriu in STDOUT_FILENO \n");           //      vor scrie pe ecran
  write(STDOUT_FILENO, "7.scriu in STDOUT_FILENO \n", 26);  // scriu pe ecran
  write(oldfd, "8.scriu in fisier.txt \n", 23);    // scriu in fisier.txt
close(oldfd);
/* afisez fisierul creat */
 printf("\nAfisez fisierul creat \n");
int status =  system ("cat fisier.txt");
/* sterg fisierul creat */
   status =  system ("rm fisier.txt");

//
  return 0;
}
*/
```

Operare

```
$ gcc -o dup2_cu_revenire dup2_cu_revenire.c
$ ./dup2_cu_revenire

oldfd= 3, newfd= 1 salv_out= 4

5.scriu in STDOUT_FILENO
6.scriu in STDOUT_FILENO
7.scriu in STDOUT_FILENO

Afisez fisierul creat
1.scriu in fisier.txt
2.scriu in fisier.txt
3.scriu in fisier.txt
4.scriu in fisier.txt
8.scriu in fisier.txt
$
```

3. Crearea prin apel *fork()* a două procese și conectarea lor printr-un PIPE anonim (bufer de conectare) cu redirectarea fișierelor standard de intrare respectiv ieșire.

Un pipe nu este un nume extern sau permanent, deci un process poate sa-l acceseze numai prin cei doi descriptori de fișier. Din această cauză un *pipe* poate fi utilizat doar de procesul care l-a creat și de descendenții acestuia care moștenesc descriptorii de fișier. Comenzile *cat* și *sort*

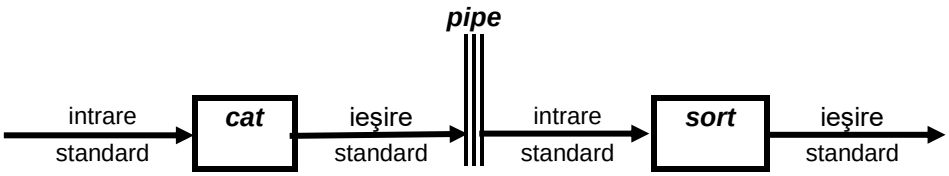
startate din procesul *main* moștenesc FD- File Descriptor-ul acestuia și pe această cale cele 3 procese (*main, cat, sort*) pot comunica.

- **Program pipe_.c**

Programul C (cu numele **pipe_.c**) tratat în această aplicație implementează startarea comenzilor din *shell* `cat /etc/passwd|sort -d` prin prin mecanismul fork-exec și utilizarea buferelor de comunicare (pipe) cu redirectarea fișierelor standard de intrare/ieșiere.
Iată mai jos ceea ce urmează a fi implement

```
$cat /etc/passwd | sort –d
      admin:x:65535:0:admin:/export/home/admin:/bin/bash
      adm:x:4:4:Admin:/var/adm:
      bin:x:2:2::/usr/bin:
      cl2-2011:x:65547:98:Chindris Luian:/export/home/cl2-
      2011:/bin/bash
      daemon:x:1:1::/:
      dladm:x:15:65:Datalink Admin:/:
      dm2-2011:x:65546:98:Dima Maria:/export/home/dm2-2011:/bin/bash
      . . . etc . . .
```

În exemplul de mai sus ieşirea standard a comenzii (*procesului*) ***cat*** (afişează fişierul */etc/passwd*) este conectată la intrarea comenzii (*procesului*) ***sort*** (afișează sortat alfabetic crescător- opţiune *–d* fișierul de intrare) printr-un bufer de comunicare ***pipe***. Efectul va fi că se afişează lista rezultată în urma comenzii **cat** sortată alfabetic.



Notă
Se vor da următoarele comenzi și se vor completa corespunzător liniile [4] și [5] din sursa programului.

```
$which  cat↵                ⇒afişează calea absolută a comenzii cat
/usr/gnu/bin/cat            ⇒[4]  se va pune în programul pipe_.c în funcția execl()

$which  sort↵               ⇒afişează calea absolută a comenzii sort
/usr/gnu/bin/sort           ⇒ [5] se va pune în programul pipe_.c în funcția execl()
```

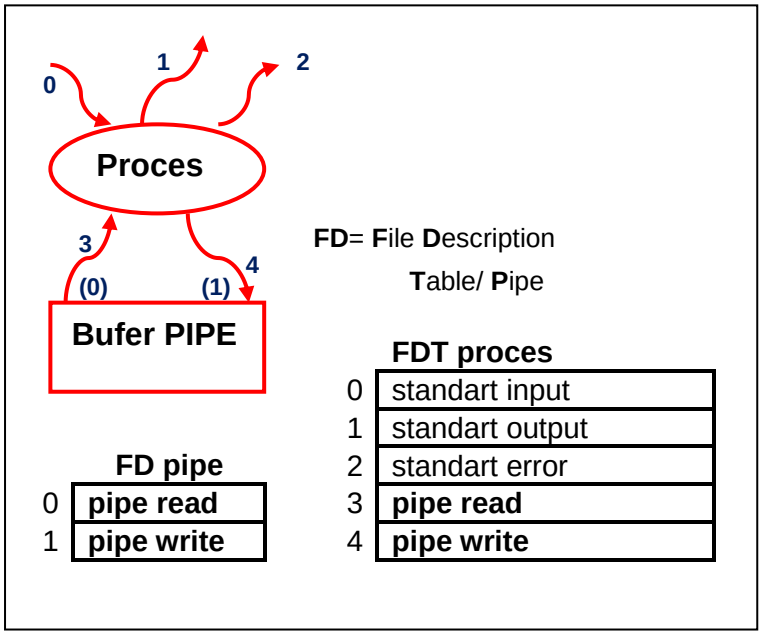
- **Program pipe_.c**

```
/* pipe_.c*/
/*
    [4] în copil, se execută /usr/gnu/bin/cat  /etc/passwd

    [5] în părinte, se execută /usr/gnu/bin/sort  -d
*/

#include<stdio.h>
#include <stdlib.h>                 // pentru functia exit()
#include<unistd.h>
#include<fcntl.h>
```

```
int main(void)
{
        int fd[2];
        pid_t childpid;

        pipe(fd);                                       //      [1]- cresare bufer de comunicare (pipe)
        if (( childpid=fork())==0)                      //      [2]     - creare proces copil
                {
                 /*cod copil: comanda cat este startata din copil*/
        // sleep(10); // test pt. a forta ca c-da sort sa astepte ca lista sa fie disponibila in pipe- NU E NRCESAR
                fprintf(stderr,"COPIL:: startez comanda cat /etc/passwd \n");
//       [3.1]
                dup2(fd[1],STDOUT_FILENO);              //  fd[1] și  stdout sunt acum echivalente
                close(fd[0]);
                close(fd[1]);
//      --3.1
                execl("/usr/gnu/bin/cat","se ignora","/etc/passwd",NULL);       //  [4] se execută cat /etc/passwd
                perror("execl: comanda cat esuata");
        } else
                {
                /*cod parinte: comanda sort este startata din parinte*/
                /* Comanda sort sorteaza alphabetic sau numeric o lista.
                        -d  -- in ordine alfabetica
                */
                // sleep(10); // test pt. a forta ca c-da cat sa livreze lista in pipe- NU E  NECESAR
                fprintf(stderr,"PARINTE:: startez comanda sort -d\n");
//       [3.2]
                        dup2(fd[0], STDIN_FILENO);      // fd[0] și  stdin sunt acum echivalente
                        close(fd[0]);
                        close(fd[1]);
                        execl("/usr/gnu/bin/sort", "se ignora","-d",NULL);      //  [5] se execută sort -d
//      --3.2
                        perror("execl: comanda sort esuata");
                }
                /* secventa comuna parinte / copil  */
         exit(0);
}
```

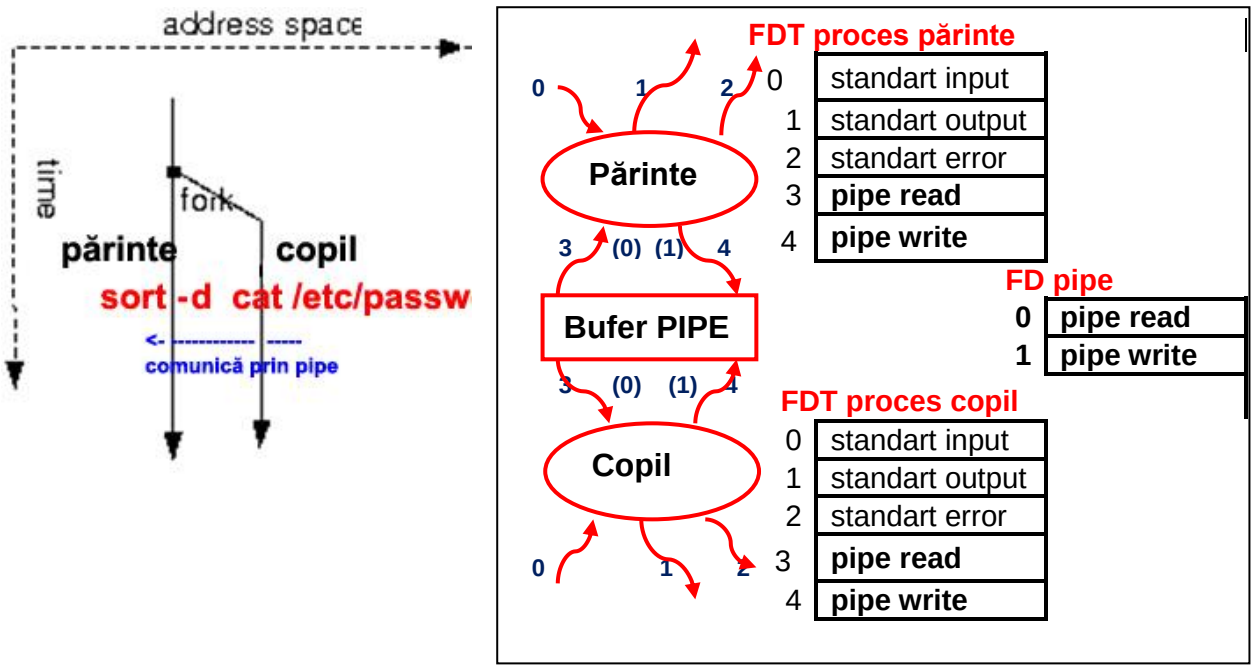
- **Explicații funcționare program**

[1] Apelul funcției *pipe()* crează un bufer de comunicare la care apelantul are acces prin descriptorii fd[0] și fd[1]. Datele scrise în fd[1] sunt citite din fd[0] dupa principiul de baza first-in-first-out (FIFO).



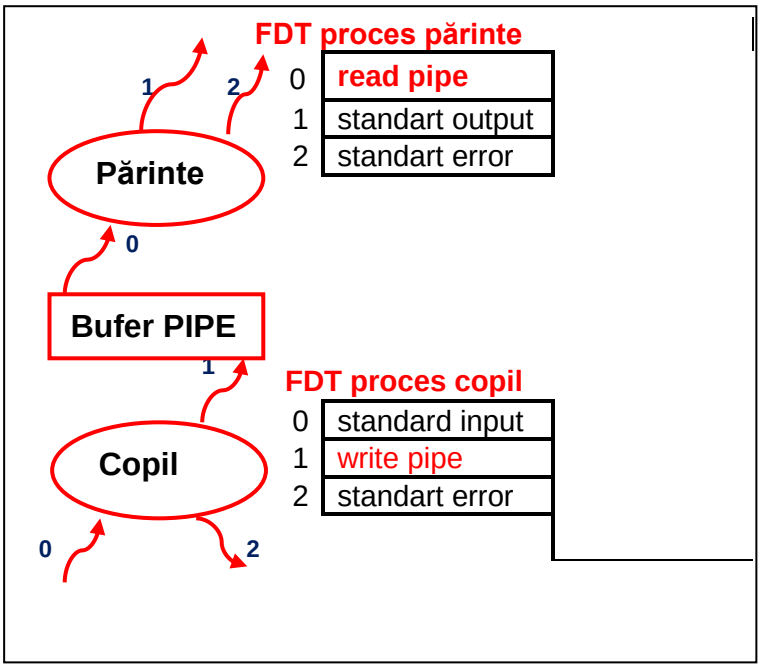
Un *pipe* nu este un nume extern sau permanent, deci un process poate sa-l acceseze numai prin cei doi descriptori de fișier. Din această cauză un *pipe* poate fi utilizat doar de procesul care l-a creat și de descendenții acestuia care moștenesc descriptorii de fișier la apelul funcției *fork()* de creare proces.

[2] Apelul funcției *fork()* crează un *proces copil* conform figurii de mai jos, realizându-se apoi datorită moștenirii comunicarea *părinte copil* prin *pipe*:



[3.1/.2] în *copil(.1)* / *părinte(.2)*, realizează conectarea celor două procese printr-un PIPE (bufer de comunicare) cu redirectarea fişierelor standard de intrare respectiv ieşire către PIPE.

Starea FDT în momentul apelului `execl`



[4] în *copil*, se execută */usr/gnu/bin/cat /etc/passwd*

[5] în *părinte*, se execută */usr/gnu/bin/sort -d*

- **Compilare linkeditare şi rulare program *pipe_.c***
  gcc –o executabil sursa.c

  $*gcc -o pipe_ pipe_.c*
  $*./pipe_*

- **Mesaje afişate în urma rulării**

PARINTE:: startez comanda sort -d
COPIL:: startez comanda cat /etc/passwd
admin:x:65535:0:admin:/export/home/admin:/bin/bash
adm:x:4:4:Admin:/var/adm:
bin:x:2:2::/usr/bin:
cl2-2011:x:65547:98:Chindris Lucian:/export/home/cl2-2011:/bin/bash
daemon:x:1:1::/:
dladm:x:15:65:Datalink Admin:/:
dm2-2011:x:65546:98:Dima Maria:/export/home/dm2-2011:/bin/bash
. . . etc . . .

4. Crearea prin apel *system()* a două procese și conectarea lor printr-un PIPE anonim (bufer de conectare) cu redirectarea fișierelor standard de intrare respectiv ieșire.

Un pipe nu este un nume extern sau permanent, deci un process poate sa-l acceseze numai prin cei doi descriptori de fișier. Din această cauză un *pipe* poate fi utilizat doar de procesul care l-a creat și de descendenții acestuia care moștenesc descriptorii de fișier. Comenzile *cat, head* și *sort* startate din procesul *main* moștenesc FD-File-Descriptor-ul acestuia și pe această cale cele 4 procese (*main, cat, head, sort*) pot comunica.
Programul C (cu numele **pipe__.c**) tratat în această aplicație implementează startarea comenzilor din *shell* `cat /etc/passwd |head -n5 |sort -d` prin mecanismul *system-exec* și utilizarea buferelor de comunicare (pipe).
Iată mai jos ceea ce urmează a fi implement

$*cat /etc/passwd |head -n5| sort –dr*

```
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

- **Program pipe__.c**

Programul redirectează prin *pipe* anonim ieșirea standard a comenzii *cat /etc/passwd* către intrarea comenzii *sort -d*. Comenzile *cat* și *sort* startate din procesul *main* prin apel *system()* moștenesc FD-File-Descriptor-ul acestuia și pe această cale cele 3 procese (*main, cat, hrad, sort*) pot comunica.

```
/** pipe__.c **/
 /** Startare: $ pipe__ **/

#include<stdio.h>
#include <stdlib.h>                                    // pentru functia exit()
#include<unistd.h>
```

```
#include<fcntl.h>

int main(void)
{
int fd[2];
pid_t childpid;
// salvari
int salv_out = dup(STDOUT_FILENO);          // salvez STDOUT_FILENO=1
int salv_in = dup(STDIN_FILENO);            // salvez STDIN_FILENO=1

pipe(fd);                   //          creare pipe

dup2(fd[1],STDOUT_FILENO);          //  fd[1] și  stdout sunt acum echivalente
close(fd[1]);                       // ramane doar stdout
fprintf(stderr,"Startez comanda \"cat /etc/passwd\" care scrie in pipe  \n");// sunt obligat sa ies pe stderr,
                                                                            //deaorece strdout este acum pipe
system ("cat /etc/passwd|head -n5");        // iesirea comenzii cat e catre pipe, intrarea e stdin
perror("system/cat: ");                     // in scop  didactic, arată că operația anterioară este OK

/* revenire stdout la situația inițială */
  dup2(salv_out,STDOUT_FILENO);             // revenirea la STDOUT_FILENO

dup2(fd[0], STDIN_FILENO);                  // fd[0] și  stdin sunt acum echivalente
close(fd[0]);                               // ramane doar stdin
fprintf(stderr,"Startez comanda \"sort -dr\" care citeste din pipe  \n");
system ("sort -dr");                        // intrarea  comenzii sort  e din pipe iesirea e pe stdout
perror("system/sort:");                     // in scop  didactic, arată că operația anterioară este OK

/* revenire stdin la situația inițială */
  dup2(salv_in,STDIN_FILENO);  // revenirea la STDOUT_FILENO

 exit(0);
}
/**  output
$ gcc -o pipe__ pipe__.c
$ ./pipe__
        Startez comanda "cat /etc/passwd" care scrie in pipe
        system/cat: : Success
        Startez comanda "sort -dr" care citeste din pipe
        sys:x:3:3:sys:/dev:/usr/sbin/nologin
        sync:x:4:65534:sync:/bin:/bin/sync
        root:x:0:0:root:/root:/bin/bash
        daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
        bin:x:2:2:bin:/bin:/usr/sbin/nologin
        system/sort:: Success
$
**/
```