Laborator 6 – PIPE – uri

Comunicarea intre procese

Pana acum am studiat procesele ca entitati independente. Sistemul de operare este cel care ii ofera fiecarui proces aceasta perspectiva prin mecanisme precum **memoria virtuala.** Aceasta facilitate confera un model de programare simplificat deoarece un proces are impresia ca detine intreaga memorie principala (**RAM-ul**) si tot timpul procesor, ca si cum ar fi singurul proces ce ruleaza pe acel calculator.

In realitate, **toate** sistemele de operare moderne sunt **multitasking.** Acest lucru inseamna ca mai multe programe pot sa fie incarcate simultan in memorie. Sistemul de operare va executa aceste programe intr-un mod echitabil pentru fiecare proces si va gestiona alocarea si dezalocarea resurselor de care are nevoie procesul pentru a isi indeplini scopurile.

Totusi, nu toate procesele isi duc executia intr-un mod paralel fata de celelalte. Uneori, un anumit numar de procese trebuie sa lucreze impreuna pentru indeplinirea unui scop comun. Din acest motiv, cum si in echipele de oameni este nevoie de comunicare pentru a face un lucru bine, si procesele trebuie sa “comunice” intre ele pentru a fi eficiente.

Sistemul de operare este cel care intermediaza aceasta comunicare deoarece el este cel ce se ocupa si de executia si bunul mers al proceselor. Acesta pune la dispozitie anumtie obiecte de tip **IPC (Inter Process Communication)** ce permite comunicarea dintre doua sau mai multe procese.

Comunicarea folosind PIPE-uri

PIPE-urile reprezinta cel mai vechi mecanism de comunicare intre procese din UNIX. Ele reprezinta canale de comunicare **unidirectionala** intre doua parti. Daca folosim **google translate** sa traducem cuvantul **“PIPE”**, printre rezultatele obtinute regasim “teava”, “tub”, “conducta”. Astfel, la fel ca si la bloc unde teava de scurgere este diferita de tevile pe care vine apa calda si apa rece, si PIPE-urile definesc o “conducta” prin care informatia circula intr-o singura directie.

Printr-un PIPE, doua sau mai multe procese isi pot transmite informatii cu conditia sa existe macar un proces destinatar si un proces expeditor. In unele cazuri, acestia pot sa fie unul si acelasi proces.

PIPE-ul este practic un buffer de octeti pe care **kernel-ul** il aloca in vederea comunicarii intre doua procese. In momentul in care nu mai exista macar un destinatar si macar un expeditor, acesta il inlatura din memorie.

Intern, un proces se refera la un PIPE prin **descriptori de fisier.** Acesti descriptori pot sa refere catre capul de scriere sau catre capul de citire a unui PIPE. In program, ei pot sa fie manipulati ca orice descriptori normali de fisiere, folosind apeluri la **read()**, **write(), fcntl().**

Pentru a crea un nou pipe folosim apelul sistem:

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

Aceasta functie returneaza 0 in caz de succes si -1 in caz de esec. In vectorul “**descriptori**”, apelul sistem stocheaza descriptorii asociati tubului nou creat. Astfel ca descriptori[0] va fi un descriptor deschis pentru citire si descriptori [1] va fi un descriptor deschis pentru scriere.

Exista anumite reguli pentru citirea si scrierea dintr-un PIPE pe care le vom enumera mai jos.

1. In cazul citirii dintr-un tub nevid care contine n caractere folosind apelul "read(p[0], buf, m)", se citesc min(n, m) deci, daca m > n, procesul nu adoarme pana cand tubul va contine m caractere;

2. In cazul incercarii de a citi dintr-un tub vid care nu are scriitori, se considera ca s-a intalnit sfarsitul de fisier si nu se citeste nimic ("read" returneaza 0); iarasi, procesul nu adoarme asteptand ca tubul sa devina nevid sau sa capete scriitori;

3. In cazul incercarii de a citi dintr-un tub vid care are scriitori, procesul adoarme pana cand tubul nu mai e vid sau pana cand nu mai are scriitori; daca primul eveniment care se intampla este ca tubul devine nevid (cineva a scris in el), in continuare se intampla ca la punctul 1 , daca primul eveniment care se intampla este disparitia tututor scriitorilor, in continuare se intampla ca la punctul 2;

4. Exista un numar maxim de caractere care se pot scrie **atomic** intr-un tub (adica respectivele caractere vor aparea in tub consecutive sau nu vor aparea deloc), desemnat de constanta simbolica PIPE_BUF, pentru care putem include "limits.h" (in principiu este 4096);

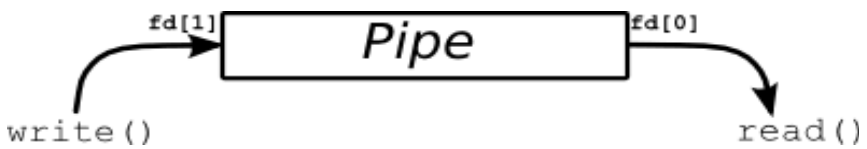
5. In cazul incercarii de a scrie intr-un tub fara cititori, procesul primeste semnalul **SIGPIPE ,** efectul implicit fiind terminarea .

6. In cazul incercarii de a scrie n caractere intr-un tub cu cititori:

a) daca n > diferenta dintre capacitatea maxima a tubului si numarul de caractere care deja exista in tub, procesul poate sa adoarma pana cand cititorii consuma suficiente caractere din tub a.i. sa poata fi scrise toate cele n caractere;

b) in cazul ca se face scrierea (respectand deci ce am zis la punctul a)), daca n <= PIPE_BUF atunci scrierea este atomica, altfel nu este neaparat atomica (deci caracterele ajung in tub in mai multe transe si daca si alte procese scriu in tub in acest timp, caracterele respective nu vor aparea consecutive, intre transe putand aparea intercalate alte caractere);

!!! ATENTIE !!! Citirea dintr-un tub este destructica, adica odata ce am citit informatia, aceasta nu mai este disponibila pentru o citire ulterioara.



Exercitii

1. Folositi apelul sistem **pipe()** sa creeati un tub. Incercati sa cititi ceva din tub, fara sa scrieti ceva inainte, ce observati ?

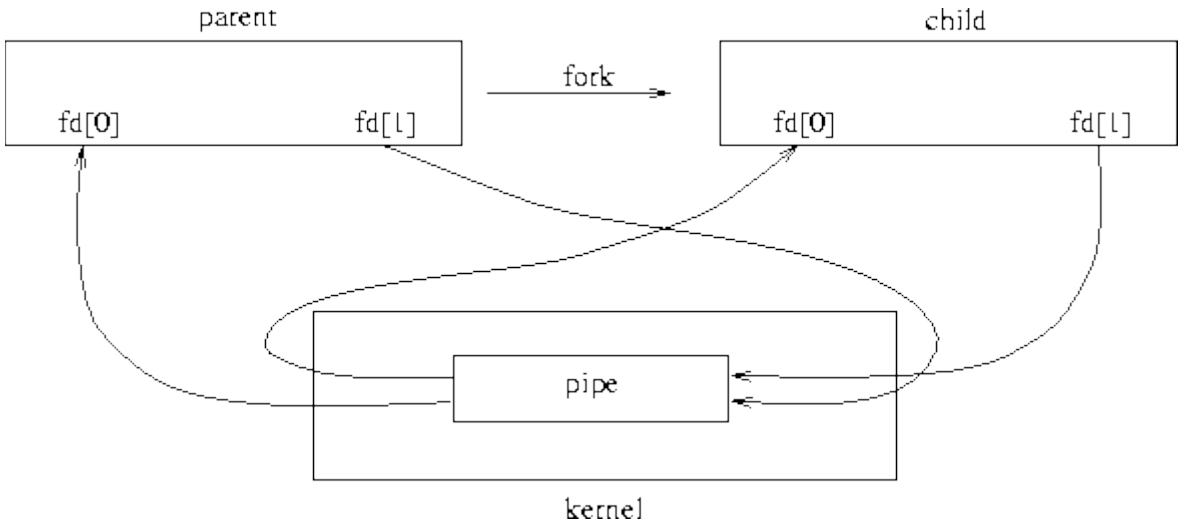
2. Folositi apelul sistem **pipe()** sa creeati un tub. Scrieti un mesaj in tub dupa care cititi-l si afisati-l la STDOUT_FILENO.

3. Folositi apelul sistem **pipe()** sa creeati un tub. Inchideti tubul destinat citirii (adica **fd[0]**). Apoi incercati sa scrieti un mesaj in tub.

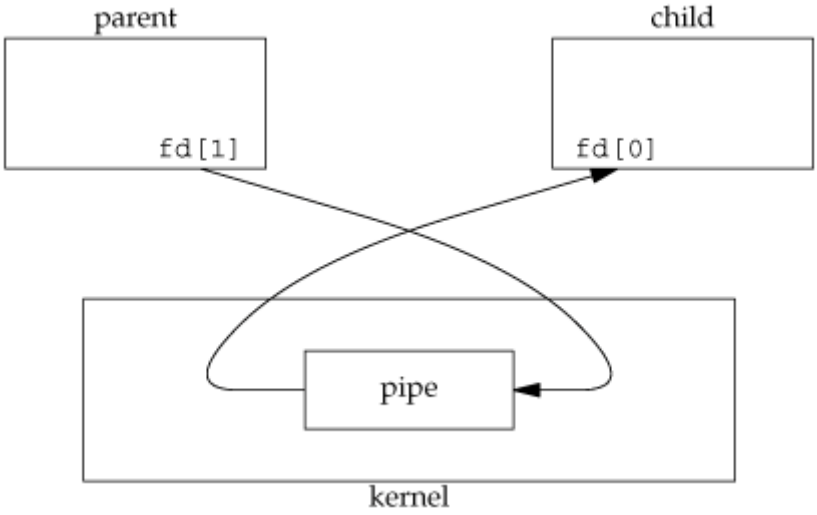
4. Folositi apelul sistem pipe() sa creeati un tub. Inchideti tubul destinat citirii (adica **fd[0]**). Apoi incercati sa scrieti un mesaj in tub. Setati un handle pentru semnalul **SIGPIPE** in care sa afisati un mesaj demonstrativ ce arata ca aceasta functie a fost apelata.

Este clar ca un pipe nu are multe intrebuintari in cadrul unui singur proces. Ramane intrebarea cum folosim un pipe creat de un proces pentru a comunica cu alte procese !?

Dupa cum am vazut in capitolul cu **procese**, printre atributele pe care un proces fiu le mosteneste de la procesul tata il reprezinta si tabela de descriptori deschisi. Astfel, dupa un apel reusit la **fork()** procesul fiu va avea de asemenea doi descriptori deschisi catre cele doua capetele ale tubului (asta in cazul in care tatal nu le-a inchis inainte de a face **fork()**). Deci accesul la tub va arata ceva de genul :



Pentru o comunicare unidirectionala intre tata si fiu este inutil sa tinem descriptorul de citire din tub al tatalui si descriptorul de scriere in tub al fiului deschisi. Si daca am vrea sa avem o comunicare bidirectionala, un singur tub ar fi greu de folosit, deoarece ar implica utilizarea unor mecanisme de sincronizare inutile, in acest caz am avea nevoie de doua tuburi. Este important sa inchidem descriptorii mentionati mai sus imediat dupa apel la **fork()**, obtinand o schema de forma:



Exercitii

1. Scrieti un program care creeaza un tub si scrie un mesaj in el dupa care face **fork()**, atat fiul cat si tatal incearca dupa aceea sa citeasca mesajul din tub si sa il afiseze la STDOUT_FILENO. Rulati programul de cateva ori dupa care executati comanda **ps.** Ce observati ?

2. Scrieti un program care creeaza un tub dupa care face **fork().** Tatal ii va trimite un mesaj fiului prin acest tub iar fiul il va afisa la standard output.

Descriptorii standard si redirectari

Asa cum am vorbit in capitolele anterioare, fiecare proces are un numar de 3 descriptori standard pe care le mosteneste de la **shell-ul** a carui fiu este. Intr-un progam C, ne referim la ei prin cele 3 constante simbolice: **STDIN_FILENO (=0), STDOUT_FILENO (=1), STDERR_FILENO (=2).**

Putem deschide noi descriptori prin apeluri sistem precum **open(), pipe()**, etc.

Aceste apeluri intorc, deobicei, urmatorii descriptori liberi din tabela de descriptori (4,5,6 etc). Cand facem un apel la **close(),** descriptorul transmis ca parametru este marcat ca eliberat de catre sistem astfel ca un viitor apel la **open()** poate returna descriptorul eliberat anterior. Un descriptor odata inchis nu mai poate sa fie eliberat !

Prin **redirectare** intelegem actiunea de a modifica un descriptor astfel incat el sa pointeze catre o alta intrare in **Tabela de Fisiere Deschise (TFD).**

De exemplu, daca intr-un program modificam cei 3 descriptori standard, atunci eventuale apeluri la functii de biblioteca precum **printf(), puts()**, **scanf()** vor avea un comportament diferit decat cel obisnuit.

In cele ce urmeaza vom vedea cum putem redirecta descriptori.

In primul rand, pentru a **redirecta** un descriptor ne trebuie un descriptor ce pointeaza deja catre o sursa dorita de noi, descriptor ce il putem obtine prin apeluri sistem de tip **open(), pipe().**

Pentru a duplica un descriptor avem la dispozitie urmatoara functie :

#include <unistd.h>

int dup(int oldfd);

Ce returneaza -1 in caz de esec iar in caz de succes un descriptor ce va pointa catre aceeasi intrare in **TFD** ca descriptorul transmis ca parametru. Noul descriptor va avea aceleasi flag-uri ca descriptorul transmis ca parametru, dar acestea pot sa fie schimbate prin apeluri ulterioare la **fcntl().** Daca apelul reuseste, este garantat ca va intoarce cea mai mica intrare disponibila din tabela de descriptori.

Daca, spre exemplu, dorim sa **redirectam** iesirea standard intr-un fisier ales de noi atunci conform celor spuse mai sus, un algoritm posibil ar fi de forma:

1. Inchidem descriptorul **STDOUT_FILENO** prin apel **close().**
2. Facem apel la **dup()** cu un descriptor creeat de noi in prealabil ce indica catre fisierul ales de noi, acesta va cauta cel mai mic descriptor disponibil, ce in cazul nostru va fi **STDOUT_FILENO**. Asta in cazul in care **STDIN_FILENO** nu fusese inchis si el.
3. Inchidem descriptorul original prin apel la **close()** deoarece nu mai are o utilitate.

Urmatorul algoritm poate fi implementat in limbajul C astfel:

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

int main()
{
  int desc = open("fisier",O_WRONLY | O_TRUNC | O_CREAT, S_IRWXU );

  close(STDOUT_FILENO);
  dup(desc);
  close(desc);

  puts("Acest mesaj o sa fie scris in fisierul \"fisier\" ");
  return 0;

}
```

Acesti descriptori raman redirectati si in urma unui apel la primitive din familia **exec().** Astfel, daca dorim sa afisam rezultatul comenzii **ls -l** intr-un fisier pentru a putea sa parcurgem rezultatul mai usor, programul in C ar arata ceva de genul :

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

int main()
{
  int desc = open("fisier",O_WRONLY | O_TRUNC | O_CREAT, S_IRWXU );
```

```
  close(STDOUT_FILENO);
  dup(desc);
  close(desc);

  execlp("ls","ls","-l",NULL);

  return 0;
}
```

In **bash** pentru a redirecta rezultatul unui comenzi se folosesc sintaxele:

$ comand < fisier_input (pentru a redirecta intrarea standard in fisier_input)
$ comanda > fisier_output (pentru a redirecta iesirea standard in fisier_output)

$ comanda 2> fisier_err (pentru a redirecta iesirea standard de erori in fisier_err)

Exercitii

1. Scrieti un program care redirecteaza intrarea standard intr-un fisier. Dupa care foloseste functia de biblioteca **scanf()** sa citeasca doi intregi din acest fisier si ii afiseaza la standard output.

Asa cum am redirectat acesti descriptori standard catre un fisier, ei pot sa fie redirectati si catre capatul unui pipe. Asta implica si ca la capatul celalalt al acestui pipe sa existe un descriptor deschis.

In **bash** sintaxa pentru a redirecta standard output-ul unei comenzi intr-un pipe ce va avea celalalt capat in standard input-ul unei alte comenzi este :

$ comanda1 | comanda2

Putem sa generalizam si sa obtinem un sir de comenzi ce o sa fie legate prin pipe-uri adica:

$ comanda1 | comanda2 | comanda3 | comanda4 | ….. | comandan

Aceasta facilitate este importanta deoarece permite filtrarea rezultatului unei comenzi pentru a obtine rezultatele cautate.

Exemplu frecvent de folosire:

Utilitarul **grep** este folosit pentru a cauta un anumit **pattern** intr-un fisier cu date. Aceasta comanda se poate lansa sub forma:

$ grep pattern fisier1 fisier2 … fisiern

Ce va cauta pattern-ul in fiecare dintre fisierele transmise ca parametru. In lipsa listei de fisiere acesta va cauta patten-ul in standard input.

Deci il putem utiliza in urmatoarele situatii:

$ ls -l | grep fisier

Pentru a verifica daca un fisier cu un anumit nume exista in directorul curent

$ ps -e | grep executabil

Pentru a verifica daca exista un proces lansat dupa un anumit executabil.

$ nm libmylib.so | grep simbol

Pentru a verifica daca exista un anumit simbol intr-o biblioteca.

La pagina 100, din cartea **Dezvoltarea aplicatilor in C/C++ sub sistemul de operare UNIX,** exista un exemplu cum o comanda de tipul: **ls -l | wc -l** poate sa fie implementata intr-un program C.
Dupa cum am vazut, un apel la **read()** pe un tub vid dar cu descriptori deschisi pentru scriere va atrage dupa sine blocarea procesului apelant. Aceasta dispozitie poate sa fie schimbata prin functia:

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

int fcntl(int fd, int cmd, ... /* arg */ );
```

Ce este folosita pentru manipularea descriptorilor de fisier. O descriere explicita a acestei functii se gaseste la pagina 46 din carte sau in pagina de manual Linux prin comanda:

$ man fcntl

Pentru a face citirea dintr-un tub neblocanta este nevoie sa se seteze flag-ul **O_NONBLOCK** pentru descriptorul de citire din tub folosind aceasta functie. Daca acest flag a fost setat, un apel la **read()** ar intoarce codul de eroare -1 dar nu se va bloca, iar **errno** va fi setat la valoarea **EAGAIN.**

Exercitiu

1. Folositi **fcntl()** sa modificati comportamentul descriptorului de citire dintr-un pipe creat anterior. Incercati sa faceti un apel la functia **read()** dupa care verificati codul de retur al functiei si daca **errno** este setat la **EAGAIN.**

Un apel la **write()** care intentioneaza sa scrie mai mult decat capacitatea tubului, va atrage dupa sine blocarea procesului apelant. Cu ajutorul functiei **fcntl()** putem modica si descriptorul pentru scriere in tub, la care sa adaugam flag-ul **O_NONBLOCK,** pentru a evita aceasta situatie. In tub se vor scrie doar cati octeti se poate iar **write()** va returna acest numar.

Tuburi cu nume

Dupa cum am spus in capitolul dedicat fisierelor, exista un tip special de fisiere denumite fisiere de tip **PIPE.** Aceste fisiere indeplinesc aceleasi propietati ca tuburile temporare deschise prin apel la **pipe()** numai ca ele au legatura pe disc. Adica sunt persistente si rezista in timp, chiar daca inchidem calculatorul, spre deosebire de pipe-urile temporare ce sunt tinute de catre kernel in **RAM** si sunt eliminate atunci cand nu mai exista nici un descriptor asociat vreunui cap al tubului.

Pentru a creea un fisier de tip pipe folosim comanda:

$ mkfifo nume_pipe

Daca dorim anumite drepturi specifice de access pentru fisierul nostru decat cele implicite vom folosi argumentul **-m** al comenzii astfel:

$ mkfifo -m masca nume_pipe

Unde **masca** are acelasi format ca argumentul dat comenzii **chmod.** Adica daca dorim ca asupra fisierului creat de noi sa aiba drepturi de scriere si de citire proprietarul, grupul proprietar si ceilalti vom folosi comanda astfel:

$ mkfifo -m 666 fisierul_meu_pipe

Intr-un program C, putem sa creea un fisier de tip pipe prin apelul sistem:

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

int mkfifo(const char *pathname, mode_t mode);

Ce returneaza 0 in caz de succes si -1 in caz de eroare. Acest apel va creea un fisier de tip pipe cu numele si calea indicata prin **pathname** si drepturilde acces indicate prin **mode.** Un program corespunzator comenzii de mai sus ar putea arata asa:

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

int main(int argc,char argv[])
{
   if( mkfifo("fisierul_meu_pipe",S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH |
                    S_IWOTH ) == 0 )
  {
      puts("A reusit!");
      return EXIT_SUCCESS;
   }
   else
   {
     puts("Nu a reusit!");
     return EXIT_FAILURE;
   }

}
```

Pentru a obtine descriptori catre un fisier special de tip **pipe** vom face un apel la **open().** Si le vom folosi normal, cum le foloseam si inainte, dar avand in minte regulile de citire si scriere din pipe enumerate mai sus. Daca vrem sa facem citirea sau scrierea neblocanta vom putea pune flag-ul **O_NONBLOCK** inca de la deschiderea fisierului.

Exercitiu

1. Scrieti un program care va creea un fisier special de tip pipe dupa care deschide un descriptor in

scriere pentru acel fisier. Acesta va face **fork()** dupa care va scrie un mesaj in fisierul cu pricina. Fiul va deschide un descriptor in citire pentru acel fisier, va citi mesajul si il va afisa la iesirea standard.

Tema

1. Scrieti un program care foloseste doua tuburi pentru a facilita o comunicare bidirectionala intre un proces tata si un proces fiu. Procesul parinte va intra intr-un ciclu infinit in care la fiecare iteratie va citi o bucata de text de la standard input si o va trimite fiului prin tub. Fiul va converti textul in uppercase (adica va face din litere mici litere mari) si il va retrimite tatalui prin celalalt tub. Tatal va citi datele din partea fiului si le va afisa la iesirea standard. Fiul va fi creeat inainte de intrarea in ciclul infinit, la fel si pipe-urile.

2. Scrieti un program "nrc" care se lanseaza sub forma:

$ nrc com

unde "com" este o comanda externa (adica asociata unui fisier executabil de pe disc) avand eventual si argumente in linia de comanda (deci com este un sir de cuvinte) si care lanseaza comanda "com", numarand cuvintele scrise de ea pe standard output. In acest scop procesul "nrc" creaza un tub fara nume, apoi se bifurca (cu "fork") a.i. intrarea standard a tatalui si iesirea standard a fiului sa fie in acest tub, apoi fiul se inlocuieste (cu o functie din familia "exec") cu un proces ce executa "com".