Sisteme de Operare 1 - Curs 8

Curs tinut in 2012-2013 *de catre* lector dr. Sanda-Maria Dragos

Comunicare intre procese Unix

slide 2

slide 3

Una dintre modalitatile de comunicare intre procese in Unix este cea prin intermediul canalelor de comunicatie (numite *pipes*, in limba engleza). Practic este vorba despre o "conducta" (un *buffer*) prin care pe la un capat se scriu mesajele, iar pe la celalalt capat se citesc - deci este vorba despre o structura de tip coada, adica lista FIFO (First-In,First-Out).

slide 4

Aceste canale sunt de doua categorii:

Comunicarea intre procese folosind pipes. Canale interne

O metoda foarte des utilizata in UNIX pentru comunicarea intre procese este folosirea primitivei numita pipe (conducta). "Conducta" este o cale de legatura care poate fi stabilita intre doua procese inrudite (au un stramos comun sau sunt in relatia stramos-urmas). Ea are doua capete, unul prin care se pot scrie date si altul prin care datele pot fi citite, permitand o comunicare intr-o singura directie. In general, sistemul de operare permite conectarea a unuia sau mai multor

procese la fiecare din capetele unui pipe, astfel incat, la un moment dat este posibil sa existe mai multe procese care scriu, respectiv mai multe procese care citesc din pipe. Se realizeaza, astfel, comunicarea unidirectionala intre procesele care scriu si procesele care citesc.

Apelul sistem pipe( )

slide 5

Crearea conductelor de date de face in UNIX folosind apelul sistem pipe( ):

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

Functia creeaza un pipe, precum si o pereche de descriptori de fisier care refera cele doua capete ale acestuia. Descriptorii sunt returnati catre programul apelant completandu-se cele doua pozitii ale tabloului filedes trimis ca parametru apelului sistem. Pe prima pozitie (i.e., filedes[0]) va fi memorat descriptorul care indica extremitatea prin care se pot citi date (capatul de citire), iar pe a doua pozitie (i.e., filedes[1]) va fi memorat descriptorul capatului de scriere in pipe.

Cei doi descriptori sunt descriptori de fisier obisnuiti, asemanatori celor returnati de apelul sistem *open( )*. Mai mult, *pipe*-ul poate fi folosit in mod similar folosirii fisierelor, adica in el pot fi scrise date folosind functia *write( )* (aplicata capatului de scriere) si pot fi citite date prin functia *read( )* (aplicata capatului de citire). MENTIUNE: Canalul intern functioneaza ca o coada, adica o lista FIFO (= First-In,First-Out), deci citirea din pipe se face cu distrugerea (consumul) din canal a informatiei citite.

Fiind implicati descriptori de fisier obisnuiti, daca un *pipe* este creat intr-un proces parinte, fiii acestuia vor mosteni cei doi descriptori (asa cum, in general, ei mostenesc orice descriptor de fisier deschis de parinte). Prin urmare, atat parintele cat si fiii vor putea scrie sau citi din *pipe*. In acest mod se justifica afirmatia facuta la inceputul acestui document prin care se spunea ca *pipe*-urile sunt folosite la comunicarea intre procese *inrudite*. Pentru ca legatura dintre procese sa se faca corect, fiecare proces trebuie sa declare daca va folosi *pipe*-ul pentru a scrie in el (transmitand informatii altor procese) sau il va folosi doar pentru citire. In acest scop, fiecare proces trebuie sa inchida capatul *pipe*-ului pe care nu il foloseste: procesele care scriu in *pipe* vor inchide capatul de citire, iar procesele care citesc vor inchide capatul de scriere, folosind functia *close( )*.

Un posibil scenariu pentru crearea unui sistem format din doua procese care comunica prin pipe este urmatorul:

» procesul parinte creeaza un *pipe*
» parintele apeleaza *fork( )* pentru a crea fiul
» fiul inchide unul din capete (ex: capatul de citire)
» parintele inchide celalalt capat al *pipe*-ului (cel de scriere)
» fiul scrie date in *pipe* folosind descriptorul ramas deschis (capatul de scriere)
» parintele citeste date din *pipe* prin capatul de citire.

slide 6

Primitiva *pipe* se comporta in mod asemanator cu o structura de date coada: scrierea introduce elemente in coada, iar citirea le extrage pe la capatul opus.

Iata in continuare o portiune de program scris conform scenariului de mai sus:

Observatii:

1. Cantitatea de date care poate fi scrisa la un moment dat intr-un *pipe* este limitata. Numarul de octeti pe care un *pipe* ii poate pastra fara ca ei sa fie extrasi prin citire de catre un proces este dependenta de sistem (de implementare). Standardul POSIX specifica limita minima a capacitatii unui *pipe*: 512 octeti. Atunci cind un *pipe* este "plin", operatia *write()* se va bloca pina cind un alt proces nu citeste suficienti octeti din *pipe*.

2. Un proces care citeste din pipe va primi valoarea **0** ca valoare returnata de *read( )* in momentul in care toate procesele care scriau in *pipe* au inchis capatul de scriere si nu mai exista date in *pipe*.

3. Daca pentru un pipe sunt conectate procese doar la capatul de scriere (cele de la capatul opus au inchis toate conexiunea) operatiile *write* efectuate de procesele ramase vor returna eroare. Intern, in aceasta situatie va fi generat semnalul SIG_PIPE care va intrerupe apelul sistem *write* respectiv. Codul de eroare (setat in variabila globala *errno*) rezultat este cel corespunzator mesajului de eroare "Broken pipe".

4. Operatia de scriere in *pipe* este atomica doar in cazul in care numarul de octeti scrisi este mai mic decit constanta PIPE_BUF. Altfel, in sirul de octeti scrisi pot fi intercalate datele scrise de un alt proces in *pipe*. Pentru detalii, consultati manualul *pipe(7)*.

Redirectarea descriptorilor de fisier

Se stie ca functia *open( )* returneaza un descriptor de fisier. Acest descriptor va indica fisierul deschis cu *open( )* pana la terminarea programului sau pana la inchiderea fisierului. Sistemul de operare UNIX ofera, insa, posibilitatea ca un descriptor oarecare sa indice un alt fisier decat cel obisnuit. Operatia se numeste *redirectare* si se foloseste cel mai des in cazul descriptorilor de fisier cu valorile 0, 1 si 2 care reprezinta intrarea standard, iesirea standard si, respectiv, iesirea standard de eroare. De asemenea, este folosita si operatia de *duplicare* a descriptorilor de fisier, care determina existenta a mai mult de un descriptor pentru acelasi fisier. De fapt, redirectarea poate fi vazuta ca un caz particular de duplicare.

slide 7

Duplicarea si redirectarea se fac, in functie de cerinte, folosind una din urmatoarele apeluri sistem:

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Functia *dup( )* realizeaza duplicarea descriptorului *oldfd*, returnand noul descriptor. Aceasta inseamna ca descriptorul returnat va indica acelasi fisier ca si *oldfd*, atat noul cat si vechiul descriptor folosind in comun pointerul de pozitie in fisier, flag-urile fisierului etc. Daca pozitia in fisier e modificata prin intermediul functiei *lseek( )* folosind unul dintre descriptori, efectul va fi observat si pentru operatiile facute folosind celalalt descriptor. Descriptorul nou alocat de *dup( )* este cel mai mic descriptor liber (inchis) disponibil.

Functia *dup2( )* se comporta in mod asemanator cu *dup( )*, cu deosebirea ca poate fi indicat explicit care sa fie noul descriptor. Dupa apelul *dup2( )*, descriptorul *newfd* va indica acelasi fisier ca si *oldfd*. Daca inainte de operatie descriptorul *newfd* era deschis, fisierul indicat este mai intai inchis, dupa care se face duplicarea.

Ambele functii returneaza descriptorul nou creat (in cazul lui *dup2( )*, egal cu *newfd*) sau -1 in caz de eroare.

Urmatoarea secventa de cod realizeaza redirectarea iesirii standard spre un fisier deschis, cu descriptorul corespunzator *fd*:

```
...
fd=open("Fisier.txt", O_WRONLY);
...
if((newfd=dup2(fd,1))<0)
  {
    printf("Eroare la dup2\n");
    exit(1);
  }
...
printf("ABCD");
...
```

slide 8

In urma redirectarii, textul "ABCD" tiparit cu *printf( )* nu va fi scris pe ecran, ci in fisierul cu numele "Fisier.txt".

Redirectarile de fisiere se pastreaza chiar si dupa apelarea unei functii de tip *exec( )* (care suprascrie procesul curent cu programul luat de pe disc). Folosind aceasta facilitate, este posibila, de exemplu, conectarea prin *pipe* a doua procese, unul din ele ruland un program executabil citit de pe disc. Secventa de cod care realizeaza acest lucru este data mai jos. Se considera ca parintele deschide un pipe din care va citi date, iar fiul este un proces care executa un program de pe disc. Tot ce afiseaza la iesirea standard procesul luat de pe disc, va fi redirectat spre capatul de scriere al *pipe*-ului, astfel incat parintele poate citi datele produse de acesta.

slide 9

Functia *fdopen( )* a fost folosita pentru a putea folosi avantajele functiilor de biblioteca pentru lucrul cu fisiere in cazul unui fisier (capatul de citire din pipe) indicat de un descriptor intreg (in speta, s-a dorit sa se efectueze o citire formatata, cu *fscanf( )*, din *pipe*).

popen si pclose

Unix ofera functiile de biblioteca *popen* si *pclose*. Biblioteca standard <stdio.h> descrie aceste apeluri:

```c
#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

slide 10

command este un string ce contine o comanda shell, iar *type* este un string ce [oate avea voaloarea "r" sau "w".

Efectul *popen* este urmatorul: dechide un pipe, apoi executa un fork. Procesu; fiu executa prin exec comnda. Procesele tata si fiu comunica prin pipe astfel:

» daca type="r", atunci procesul tata citeste din pipe, folosind pointerul la fisier intors de popen, iesirea standard data de comanda.
» daca type="w", atunci procesul tata scrie in pipe, folosind pointerul la fisierul intors de popen, iar ceea ce scrie se constituie in intrarea standard pentru comanda

Daca comanda a fost lansata cu intrarea redirectata dinspre procesul apelant, atunci in continuare putem trimite date comenzii lansate prin *fwrite()* sau *fprintf()*. Daca comanda a fost lansata cu iesirea redirectata spre procesul apelant, atunci iesirea comenzii se poate citii din procesul apelant cu functiile *fread()* sau *fscanf()*. In ambele cazuri, incheierea se face prin apelul functiei *pclose()*. Apelul functiei *pclose()* inchide pipe-ul si intoarce codul de retur al comezii. La esec intoarce -1.

Comunicarea intre procese folosind FIFO. Canale externe

Deci un canal extern este un canal prin care pot comunica doua sau mai multe procese, comunicatia facindu-se in acest caz printr-un fisier de tip *fifo*.

Comunicatia intre procese prin intermediul unui canal *fifo* poate avea loc daca acele procese cunosc numele fisierului *fifo* respectiv.

Modul de comunicare prin intermediul unui fisier *fifo* - la fel ca la fisiere obisnuite: mai intii se deschide fisierul, apoi se scrie in el si/sau se citeste din el, iar la sfirsit se inchide fisierul.

Crearea unui fisier *fifo* se face cu ajutorul functiilor `mknod` sau `mkfifo`. Urmatorul program exemplifica modul de creare a unui fisier fifo.

slide 12

slide 11

```c
/* mkf.c : creare fisier fifo */

#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<errno.h>
extern int errno;

main(int argc, char** argv)
{
  if(argc != 2)
  {
    fprintf(stderr,"Sintaxa apel: mkf nume_fifo\n");
    exit(1);
  }

  if( mknod(argv[1], S_IFIFO|0666, 0) == -1 )
  /* sau: if( mkfifo(argv[1], 0666) == -1 ) */
  {
    if(errno == 17)   // 17 = errno for "File exists"
    {
       fprintf(stdout,"Note: fifo %s exista deja !\n",argv[1]);
       exit(0);
    }
    else
    {
       fprintf(stderr,"Error: creare fifo imposibila, errno=%d\n",errno);
       perror(0);
       exit(2);
    }
  }
}
```

Alternativ, un fisier *fifo* poate fi creat si direct de la prompterul *shell*-ului, folosind comenzile `mknod` sau `mkfifo`.

Restul operatiilor asupra canalelor *fifo* se fac la fel ca la fisiere obisnuite, fie cu functiile I/O de nivel scazut (`open`, `read`, `write`, `close`), fie cu functiile I/O de nivel inalt (`fopen`, `fread`/`fscanf`, `fwrite`/`fprintf`, `fclose`).

Prin urmare, deschiderea unui fisier *fifo* se face cu apelul functiei `open` sau `fopen`, intr-unul din urmatoarele trei moduri posibile:

1. *read* & *write* (deschiderea ambelor capete ale canalului),

2. *read-only* (deschiderea doar a capatului de citire),

3. sau *write-only* (deschiderea doar a capatului de scriere),
   mod ce este specificat prin parametrul transmis functiei de deschidere.

slide 13

Observatie importanta:
In mod implicit, deschiderea se face in mod **blocant**: o deschidere *read-only* trebuie sa se "sincronizeze" cu una *write-only*. Cu alte cuvinte, daca un proces incearca o deschidere a unui capat al canalului extern, apelul functiei de deschidere ramine blocat (i.e., functia nu returneaza) pina cind un alt proces va deschide celalalt capat al canalului extern.

Si in cazul canalelor externe apar restrictiile si problemele de la canale interne si anume:

Caracteristici si restrictii ale canalelor externe:

1. Canalul extern este un canal unidirectional, adica pe la un capat se scrie, iar pe la capatul opus se citeste.
   Insa toate procesele pot scrie la capatul de scriere, si/sau sa citeasca la capatul de citire.

2. Unitatea de informatie pentru canalul extern este octetul. Adica, cantitatea minima de informatie ce poate fi scrisa in canal, respectiv citita din canal, este de 1 octet.

3. Canalul extern functioneaza ca o coada, adica o lista FIFO (= First-In,First-Out), deci citirea din canal se face cu distrugerea (consumul) din canal a informatiei citite.
   Asadar, citirea dintr-un canal extern (i.e., fisier *fifo*) difera de citirea din fisiere obisnuite, pentru care citirea se face fara consumul informatiei din fisier.

4. Dimensiunea (i.e. capacitatea) canalului extern este limitata la o anumita dimensiune maxima (4ko, 16ko, etc.), ce difera de la un sistem Unix la altul.

5. Citirea dintr-un canal extern (cu primitiva `read`):
   » Apelul `read` va citi din canal si va returna imediat, fara sa se blocheze, numai daca

mai **este suficienta informatie** in canal, iar in acest caz valoarea returnata reprezinta numarul de octeti cititi din canal.

» Altfel, daca canalul este gol, sau nu contine suficienta informatie, apelul de citire `read` va **ramine blocat** pina cind va avea suficienta informatie in canal pentru a putea citi cantitatea de informatie specificata, ceea ce se va intimpla in momentul cind alt proces va scrie in canal.

» Alt caz de exceptie la citire, pe linga cazul golirii canalului:
daca un proces incearca sa citeasca din canal si nici un proces nu mai este capabil sa scrie in canal vreodata (deoarece toate procesele si-au inchis deja capatul de scriere), atunci apelul `read` returneaza imediat valoarea `0` corespunzatoare faptului ca a citit `EOF` din canal.
In concluzie, pentru a se putea citi `EOF` din canalul *fifo*, trebuie ca mai intai toate procesele sa inchida canalul in scriere (adica sa inchida descriptorul corespunzator capatului de scriere).

Observatie: La fel se comporta si celelalte functii de citire (`fread`, `fscanf`, etc.) la citirea din canale externe.

6. Scrierea intr-un canal intern (cu primitiva `write`):
   » Apelul `write` va scrie in canal si va returna imediat, fara sa se blocheze, numai daca mai **este suficient spatiu liber** in canal, iar in acest caz valoarea returnata reprezinta numarul de octeti efectiv scrisi in canal (care poate sa nu coincida intotdeauna cu numarul de octeti ce se doreau a se scrie, caci pot apare erori I/O).

   » Altfel, daca canalul este plin, sau nu contine suficient spatiu liber, apelul de scriere `write` va **ramine blocat** pina cind va avea suficient spatiu liber in canal pentru a putea scrie informatia specificata ca argument, ceea ce se va intimpla in momentul cind alt proces va citi din canal.

   » Alt caz de exceptie la scriere, pe linga cazul umplerii canalului:
   daca un proces incearca sa scrie in canal si nici un proces nu mai este capabil sa citeasca din canal vreodata (deoarece toate procesele si-au inchis deja capatul de citire), atunci sistemul va trimite acelui proces semnalul `SIGPIPE`, ce cauzeaza intreruperea sa si afisarea pe ecran a mesajului "**Broken pipe**".

Observatie: La fel se comporta si celelalte functii de scriere (`fwrite`, `fprintf`, etc.) la scrierea in canale externe.

Observatie:
La fel ca la canale interne, cele afirmate mai sus, despre blocarea apelurilor de citire sau de scriere in cazul canalului gol, respectiv plin, corespund comportamentului implicit, **blocant**, al canalelor externe.
Insa, exista posibilitatea modificarii acestui comportament implicit, intr-un comportament **neblocant**, situatie in care apelurile de citire sau de scriere nu mai ramin blocate in cazul canalului gol, respectiv plin, ci returneaza imediat valoarea -1, si seteaza corespunzator variabila `errno`.

Modificarea comportamentului implicit in comportament **neblocant** se realizeaza prin setarea atributului `O_NONBLOCK` pentru descriptorul corespunzator acelui capat al canalului intern pentru care se doreste modificarea comportamentului, ceea ce se poate face fie direct la deschiderea canalului, fie dupa deschidere, prin intermediul functiei `fcntl`. Spre exemplu, apelul:

```
    fd_out = open("canal_fifo", O_WRONLY | O_NONBLOCK );
```

va seta la deschidere atributul `O_NONBLOCK` pentru capatul de scriere al canalului extern cu numele `canal_fifo` din directorul curent de lucru. Iar apelul:

```
    fd_out = open("canal_fifo",O_WRONLY);
    ...
    fcntl(fd_out,F_SETFL,O_NONBLOCK);
```

va seta, dupa deschidere, atributul `O_NONBLOCK` pentru capatul de scriere al canalului extern cu numele `canal_fifo` din directorul curent de lucru.

Atentie: functiile de nivel inalt (`fread`/`fwrite`, `fscanf`/`fprintf`, etc.) lucreaza *buffer*-izat. Ca atare, la scrierea intr-un canal folosind functiile `fwrite`, `fprintf`, etc., informatia nu ajunge imediat in canal in urma apelului, ci doar in *buffer*-ul asociat acelui descriptor, iar golirea *buffer*-ului in canal se va intimpla abia in momentul umplerii *buffer*-ului, sau la intilnirea caracterului '\n' in specificatorul de format al functiei de scriere, sau la apelul functiei `fflush` pentru acel descriptor.
Prin urmare, daca se doreste garantarea faptului ca informatia ajunge in canal imediat in urma apelulului de scriere cu functii de nivel inalt, trebuie sa se apeleze, imediat dupa apelul de scriere, si functia `fflush` pentru descriptorul asociat capatului de scriere al acelui canal.

Deosebiri fata de canalele interne:

1. Functia de creare a unui canal extern nu produce si deschiderea automata a celor doua capete, acestea trebuie dupa creare sa fie deschise explicit prin apelul functiei de deschidere.

2. Un canal extern poate fi deschis, la oricare din capete, de orice proces, indiferent daca acel proces are sau nu vreo legatura de rudenie (prin `fork` / `exec`) cu procesul care a creat canalul extern.
   Aceasta este posibil deoarece un proces trebuie doar sa cunoasca numele fisierului *fifo* pe care doreste sa-l deschida, pentru a-l putea deschide. Bineinteles, procesul respectiv mai trebuie sa aiba si drepturi de acces pentru acel fisier *fifo*.

slide 14

3. Dupa ce un proces inchide un capat al unui canal *fifo*, acel proces poate redeschide din nou acel capat pentru a face alte operatii I/O asupra sa.

Pentru mai multe detalii despre canalele *fifo*, va recomand citirea paginii de *help* a comenzii de creare a unui canal extern:

```
UNIX> man 3 mkfifo
```

si a paginii generale despre canale externe:

```
UNIX> man fifo
```