Comunicarea între procese Unix: pipe, FIFO, popen, dup2

Contents

1. Principalele apeluri sistem de comunicare între procese

Tabelul următor prezintă sintaxele principalelor apeluri sistem Unix pentru comunicare între procese:

Funcţii specifice comunicării între procese

`pipe(f)`
`mkfifo(nume, drepturi)`
`FILE *popen(c, "r|w")`
`pclose(FILE *)`
`dup2(fo,fn)`

Prototipurile lor sunt descrise, de regula, in `<unistd.h>` Parametrii sunt:

- `c` este o comandă Unix;
- `f` este un tablou de doi intregi – descriptori de citire / scriere din / in pipe;
- `nume` este numele (de pe disc) al fişierului FIFO, iar `drepturi` sunt drepturile de acces la acesta;
- `fo` si `fn` descriptori de fisiere: `fo` deschis in program cu `open`, `fn` poziţia în care e duplicat `fo`.

In caz de eşec, functiile întorc -1 (NULL la `popen`) si poziţionează `errno` se depisteaza ce eroare a aparut.

Funcţia `popen` are comanda completă (un string), interpretabilă de shell şi apoi executată

2. Analizaţi textul sursă

Considerând că toate instrucțiunile din fragmentul de cod de mai jos se execută cu succes, răspundeți la următoarele întrebări:

a) Ce va tipări rularea codului așa cum este?
b) Câte procese se creează, incluzând procesul inițial, dacă lipsește linia 8? Specificați relația părinte fiu dintre aceste procese.
c) Câte procese se creează, incluzând procesul inițial, dacă mutăm instrucțiunea de pe linia 8 pe linia 11 (pornind de la codul dat)? Specificați relația părinte fiu dintre aceste procese.

d) Ce va tipări rularea codului, dacă liniile 16 și 17 se mută în interiorul ramurii else, începând cu linia 11 a codului inițial? Justificați răspunsul.

```
1   int main() {
2       int pfd[2], i, n;
3       pipe(pfd);
4       for(i=0; i<3; i++) {
5           if(fork() == 0) {
6               write(pfd[1], &i, sizeof(int));
7               close(pfd[0]); close(pfd[1]);
8               exit(0);
9           }
10          else {
11              // a se vedea punctele c) si d)
12          }
13      }
14      for(i=0; i<3; i++) {
15          wait(0);
16          read(pfd[0], &n, sizeof(int));
17          printf("%d\n", n);
18      }
19      close(pfd[0]); close(pfd[1]);
20      return 0;
21  }
```

Răspuns:

a) 0, 1, 2 pe linii separate în orice ordine.
b) 8 procese, arbore cu 8 procese.
c) 4 procese, arbore cu 4 procese.
d) 0, 1, 2 pe linii separate întodeauna în această ordine.

3. Utilizări simple pipe şi FIFO

Pentru a ilustra modul de lucru cu pipe şi cu FIFO, vom pleca de la exemplul cunoscut de **adunare paralelă rea a patru numere**, exemplu care reclamă necesitatea comunicării între procese. Sursa programului **`add4Rau.c`** este:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main () {
    int a[] = {1,2,3,4};
    if (fork()==0) { // Procesul fiu
       a[0]+=a[1];
       exit(0);
    }
    a[2]+=a[3]; // Procesul parinte
    wait(NULL);
    a[0]+=a[2];
    printf("Suma este %d\n", a[0]);
}
```

Se ştie, suma tipărită va fi 8, nu 10, deoarece informaţia din procesul fiu nu ajunge în părinte. Vom da trei soluţii corecte pentru această problemă, toate vor tipări "**`Suma este 10`**".

Soluţia 1: comunicarea prin pipe este dată în programul **`add4p.c`**:

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

```
#include <stdlib.h>
#include <sys/wait.h>
int main () {
    int a[] = {1,2,3,4}, f[2];
    pipe(f);
    if (fork()==0) { // Procesul fiu
        close(f[0]); // Nu trebuie
        a[0]+=a[1];
        write(f[1], &a[0], sizeof(int)); // Scrie suma partiala
        close(f[1]);
        exit(0);
    }
    close(f[1]); // Nu trebuie in procesul parinte
    a[2]+=a[3]; // Procesul parinte
    read(f[0], &a[0], sizeof(int));
    close(f[0]);
    wait(NULL);
    a[0]+=a[2];
    printf("Suma este %d\n", a[0]);
}
```

Soluţia 2: comunicarea prin FIFO cu procesele în aceeaşi sursă. FIFO permite comunicarea între două procese care nu sunt, neapărat, înrudite. Din această cauză, se obişnuieşte ca fişierul FIFO să se creeze în directorul `/tmp`. Această creare se poate face, de exemplu, prin comanda:

```
$ mkfifo /tmp/fifo1
```

Evident, crearea se face înainte de a lansa procesele care o utilizează. Natural, atunci când nu mai avem nevoie de acest FIFO, el se şterge cu comanda:

```
$ rm /tmp/fifo1
```

Sursa pentru această soluţie este `add4f.c`:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/types.h>
int main () {
    int a[] = {1,2,3,4}, f;
    if (fork()==0) { // Procesul fiu
        f = open("/tmp/fifo1", O_WRONLY);
        a[0]+=a[1];
        write(f, &a[0], sizeof(int)); // Scrie suma partiala
        close(f);
        exit(0);
    }
    f = open("/tmp/fifo1", O_RDONLY);
    a[2]+=a[3]; // Procesul parinte
    read(f, &a[0], sizeof(int));
    close(f);
    wait(NULL);
    a[0]+=a[2];
    printf("Suma este %d\n", a[0]);
}
```

Soluţia 3: comunicarea prin FIFO între două procese create din surse diferite. Tabelul următor prezintă fişierele add4fTata.c şi add4fFiu.c care vor comunica între ele:

add4fTata.c	**add4fFiu.c**
`#include <stdio.h>`	`#include <stdio.h>`

```c
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
int main () {
    int a[] = {1,2,3,4}, f;
    f=open("/tmp/fifo1",O_RDONLY);
    a[2]+=a[3];
    read(f, &a[0], sizeof(int));
    close(f);
    a[0]+=a[2];
    printf("Suma este %d\n", a[0]);
}
```

```c
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/types.h>
int main () {
    int a[] = {1,2,3,4}, f;
    f = open("/tmp/fifo1", O_WRONLY);
    a[0]+=a[1];
    write(f, &a[0], sizeof(int));
    close(f);
}
```

Inainte de a lansa procesele, trebuie creat FIFO. procesele se pot lansa în orice ordine, deoarece se aşteaptă unul după celălalt. Eventual fiul poate fi lansat în background.

4. Simulare sh pentru who | sort şi who | sort | cat (dup2)

Problema 4: Simularea unui shell care executa comanda: `$ who | sort`.

Pentru simulare, programul principal va crea doua procese fii in care va lansa, prin `exec`, comenzile `who` si `sort`. Inainte de crearea acestor fii, va crea un pipe pe care il va da celor doi fii ca sa comunice intre ei: `who` isi va redirecta iesirea standard in acest pipe cu ajutorul apelului `dup2`, iar `sort` va avea ca intrare standard acest pipe, redirectat de asemenea cu `dup2`.

O extindere naturală este conectarea în pipe a trei programe, de exemplu `who | sort | cat`. (De aici, generalizarea la un pipeline între n comenzi este uşor de făcut). Sursele celor două programe sunt date în tabelul următor:

who | sort

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main () {
    int p[2];
    pipe (p);
    if (fork() == 0) {
        close (p[0]);
        dup2 (p[1], 1);
        execlp ("who", "who", NULL);
    } else if (fork() == 0) {
        close (p[1]);
        dup2 (p[0], 0);
        execlp ("sort", "sort", NULL);
    } else {
        close (p[0]);
        close (p[1]);
        wait (NULL);
        wait (NULL);
    }
}
```

who | sort | cat

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main () {
    int p[2], q[2];
    pipe (p);
    pipe (q);
    if (fork() == 0) {
        close (p[0]);
        close (q[0]);
        close (q[1]);
        dup2 (p[1], 1);
        execlp ("who", "who", NULL);
    } else if (fork() == 0) {
        close (p[1]);
        close (q[0]);
        dup2 (p[0], 0);
        dup2 (q[1], 1);
        execlp ("sort", "sort", NULL);
    } else if (fork() == 0) {
        close (p[0]);
        close (p[1]);
        close (q[1]);
        dup2 (q[0], 0);
        execlp ("cat", "cat", NULL);
    } else {
        close (p[0]);
```

```
        close (p[1]);
        close (q[0]);
        close (q[1]);
        wait (NULL);
        wait (NULL);
        wait (NULL);
    }
}
```

Principiul este simplu: dacă avem n comenzi în pipeline, atunci trebuie construite n-1 pipe-uri. Procesul ce execută prima comandă îşi va redirecta ieşirea standard în primul pipe. Procesul ce execută ultima comandă îşi va redirecta intrarea standard in ultimul pipe. Procesele ce execută comenzile intermediare, să zicem procesul i cu i>1 şi i < n-1, va avea ca intrare standard pipe-ul i-1 şi ca ieşire standard pipe-ul i.

Evident, în locul comenzilor `who, sort, cat` pot să apară orice comenzi, cu orice argumente.

5. Paradigma client / server; exemple

Problema pe care o vom rezolva folosind paradigma client / server este următoarea:

Să se scrie un program **server** care primeşte în FIFO-ul `/tmp/CERERE` un string de 20 caractere:

`numar` întreg pozitiv fără semn > 1, exact 10 caractere, cu completare de zerouri la stânga	**`nume`** din exact 10 caractere, fără spaţii albe (blank, tab, \n, \r etc.).

Serverul descompune **`numar`** în factori primi şi scrie această descompunere în FIFO-ul **`/tmp/nume`**

Să se scrie un program **client** care construie un string de 10 caractere **`nume`** care să reprezinte unic procesul client (nume putând să fie, de exemplu, PID-ul procesului completat la stânga cu zerouri). Clientul primeşte la linia de comandă un număr întreg de maximum 10 cifre, formează cererea ca mai sus şi o scrie într-un FIFO **`/tmp/CERERE`**. Apoi citeşte răspunsul de la server pe FIFO-ul `/tmp/nume`, mai întâi lungimea stringului, apoi stringul răspuns. In final tipăreşte răspunsul primit pe ieşirea lui standard, după care şterge FIFO-ul **`/tmp/nume`**.

Vom da două soluţii pentru **server**:

- Un **server iterativ**, care citeşte o cerere, o execută, trimite răspunsul, apoi iarăşi revine la citirea unei noi cereri ş.a.m.d.
- Un **server concurent**, care citeşte o cerere, crează un proces fiu căruia îi trimite cererea, după care revine la citirea unei noi cereri ş.a.m.d. Toată sarcina de tratare a cererii revine procesului fiu, care evoluează în acelaşi timp cu preluarea de către server a unor noi cereri.

Intrebare: de ce este nevoie ca cererea să aibă această structură rigidă?

Vom începe cu **prezentarea `client.c`**. Sursa acestuia este:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
int main (int argc, char* argv[]) {
    char cerere[21] = "01234567890123456789",
    fiforasp[16] = "/tmp/0123456789", rasp[500];
    int lrasp, fc, fr;
    sprintf(rasp, "%010lu", atol(argv[1]));
```

```
        memcpy(cerere, rasp, 10);
        sprintf(rasp, "%010d", getpid());
        memcpy(&cerere[10], rasp, 10);
        memcpy(&fiforasp[5], rasp, 10);
        while ((fc = open("/tmp/CERERE", O_WRONLY)) < 0) sleep(1); // !!
        printf("Cerere: %s\n", cerere);
        write(fc, cerere, 20);
        close(fc);
        while ((fr = open(fiforasp, O_RDONLY)) < 0) sleep(1); // !!
        read(fr, &lrasp, sizeof(int));
        read(fr, rasp, lrasp);
        printf("Raspuns: %s\n", rasp);
        close(fr);
        unlink(fiforasp);
    }
```

Acţiunea clientului este cea descrisă mai sus. Trebuie să atragem atenţia asupra celor două apeluri sistem `open` care deschid FIFO de cerere şi de răspuns. Sarcina creării FIFO-urilor revine serverului. Ordinea celor două `open` trebuie să fie aceeaşi şi la server, altfel se ajunge la deadlock.

Se observă că `open` se reia până când se reuşeşte deschiderea. Aici reluarea se face din secundă în secundă (se poate şi mai des sau chiar reluare fără pauză). Motivul acestei reluări este acela că dacă serverul ocupă procesorul după client, este posibil ca unul din FIFO-uri să nu fie creat la momentul `open` al clientului. Dacă se întâmplă asta, se va trece de `open`, în ciuda regulilor FIFO, şi se semnalează "`No such file or directory`".

Cele două servere, serveriter,c şi serverconc.c sunt foarte asemănătoare. Le prezentăm simultan în tabelul de mai jos.

serveriter.c

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

int main (int argc, char* argv[]) {

char cerere[21] = "01234567890123456789";
char fiforasp[16]="/tmp/0123456789";
    char rasp[500], factor[50];
    int lrasp, fc, fr;
    unsigned long n, p, d;
    unlink("/tmp/CERERE");
    mkfifo("/tmp/CERERE", 0777);
    fc = open("/tmp/CERERE", O_RDONLY);
    for ( ; ; ) {
        if (read(fc, cerere, 20) != 20)
            continue;

  memcpy(&fiforasp[5],&cerere[10],10);
        cerere[10] = '\0';
        n = atol(cerere);
        sprintf(rasp, "%lu = ", n);
        d = 2;
        while(n > 1) {
            p = 0;
            while(n % d == 0) {
                p = p + 1;
                n = n / d;
            }
```

serverconc.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
# include <signal.h>
int main (int argc, char* argv[]) {
    signal(SIGCHLD, SIG_IGN);
char cerere[21]= "01234567890123456789";
char fiforasp[16]= "/tmp/0123456789";
    char rasp[500], factor[50];
    int lrasp, fc, fr;
    unsigned long n, p, d;
    unlink("/tmp/CERERE");
    mkfifo("/tmp/CERERE", 0777);
    fc = open("/tmp/CERERE", O_RDONLY);
    for ( ; ; ) {
        if (read(fc, cerere, 20) != 20)
            continue;
        if (fork() == 0) {
      memcpy(&fiforasp[5],&cerere[10],10);
            cerere[10] = '\0';
            n = atol(cerere);
            sprintf(rasp, "%lu = ", n);
            d = 2;
            while(n > 1) {
                p = 0;
                while(n % d == 0) {
                    p = p + 1;
                    n = n / d;
                }
```

```
            if (p > 0) {
                sprintf(factor,
                    "%lu^%lu * ",d,p);
                strcat(rasp, factor);
            }
            d = d + 1;
        }
        rasp[strlen(rasp) - 3] = '\0';
        lrasp = strlen(rasp) + 1;
        unlink(fiforasp);
        fr = mkfifo(fiforasp, 0777);
        fr = open(fiforasp, O_WRONLY);
        write(fr, &lrasp, sizeof(int));
        write(fr, rasp, lrasp);
        close(fr);

    }
}
```

```
                if (p > 0) {
                    sprintf(factor,
                       "%lu^%lu * ",d,p);
                    strcat(rasp, factor);
                }
                d = d + 1;
            }
            rasp[strlen(rasp) - 3] = '\0';
            lrasp = strlen(rasp) + 1;
            unlink(fiforasp);
            fr = mkfifo(fiforasp, 0777);
            fr = open(fiforasp, O_WRONLY);
            write(fr, &lrasp, sizeof(int));
            write(fr, rasp, lrasp);
            close(fr);
        }
    }
}
```

Serverul concurent, după ce primeşte cererea, crează un proces fiu, care implicit primeşte această cerere. Pentru ca fii să nu rămână în starea zombie, prima instrucţiune din server este apelul sistem `signal`.

6. Exemple de utilizare popen

Utilizare `popen` cu scriere in intrarea standard pentru comenda lansată: de exemplu, scrierea in ordine alfabetică a argumentelor şi a variabilelor de mediu:

```
#include <stdio.h>
main (int argc, char *argv[], char *envp[]) {
    FILE *f;  int i;
    f = popen("sort", "w");
    for (i=0; argv[i]; i++ )
        fprintf(f, "%s\n", envp[i]);
    for (i=0; envp[i]; i++ )
        fprintf(f, "%s\n", envp[i]);
    pclose (f);
}// Rezultatul, pe iesirea standard
```

Utilizare `popen` cu preluarea iesirii standard a comenzii lansate, de exemplu să se verificare că userul `florin` este logat:

```
#include <stdio.h>
#include <string.h>
main () {
    FILE *f;  char l[1000], *p;
    f = popen("who", "r");
    for ( ; ; ) {
        p = fgets(l, 1000, f);
        if (p == NULL) break;
        if (strstr(l, "florin")) {
            printf("DA\n");
            pclose (f);
            return;
        }
    }
    printf("NU\n");
    pclose (f);
}
```

Lansarea în paralel a mai multor programe filtru, folosind mai multe `popen`. Presupunem ca avem un program lansabil: `$ filtru intrare iesire` care transforma fisierul `intrare` in fisierul `iesire` dupa reguli stabilite de user. Se cere un program care primeste la linia de comandă mai multe nume de fisiere de intrare, care să fie filtrate în procese paralele în fisiere de ieşire. Programul este:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
main(int argc, char* argv[]) {
    int i;
    char c[50][200];
    FILE *f[50];
    for (i=1; argv[i]; i++) {
        strcpy(c[i], "./filtru ");
        strcat(c[i], argv[i]);
        strcat(c[i], " ");
        strcat(c[i], argv[i]);
        strcat(c[i], ".FILTRU");
        popen(c[i], "r");
        f[i] = popen(c[i], "r");
        pclose(f[i]);
    }
}
```

7. Probleme propuse

1. Se dă fişierul **grep.c** care conţine fragmentul de cod de mai jos şi care se compilează în directorul personal al utilizatorului sub numele **grep**. Răspundeţi la următoarele întrebări, considerând că toate instrucţiunile se execut cu succes.

a) Enumeraţi şi explicaţi valorile posibile ale variabilei **n**.

b) Ce vor afişa pe ecran următoarele rulări, considerând că directorul personal al utilizatorului nu se află în variabila de mediu PATH

```
1. grep grep grep.c
2. ./grep grep grep.c
3. ./grep grep
```

```
1  int main(int c, char** v) {
2    int p[2], n;
3    char s[10] = "ceva";
4    pipe(p);
5    n = fork();
6    if(n == 0) {
7      close(p[0]);
8      printf("înainte\n");
9      if(c > 2)
10       execlp("grep","grep",v[1],v[2],NULL);
11     strcpy(s, "dup ");
12     write(p[1], s, 6);
13     close(p[1]);
14     exit(0);
15   }
16   close(p[1]);
17   read(p[0], s, 6);
18   close(p[0]);
19   printf("%s\n", s);
20   return 0;
21 }
```

2. Considerând că toate instrucțiunile din fragmentul de cod de mai jos se execută cu succes, răspundeți la următoarele întrebări:

a) Desenaţi ierarhia proceselor create, incluzând şi procesul părinte.
b) Daţi fiecare linie afişată de program, împreună cu procesul care o tipăreşte.
c) Câte caractere sunt citite din pipe?
d) Cum este afectată terminarea proceselor dacă lipseşte linia 20?
e) Cum este afectată terminarea proceselor dacă lipsesc liniile 20 şi 21?

```
1  int main() {
2      int p[2], i=0;
3      char c, s[20];
4      pipe(p);
5      if(fork() == 0) {
6          close(p[1]);
7          while(read(p[0], &c, sizeof(char))) {
8              if (i<5 || i > 8) {
9                  printf("%c", c);
10             }
11             i++;
12         }
13         printf("\n"); close(p[0]);
14         exit(0);
15     }
16     printf("Result: \n");
17     strcpy(s, "exam not passed");
18     close(p[0]);
19     write(p[1], s, strlen(s)*sizeof(char));
20     close(p[1]);
21     wait(NULL);
22     return 0;
23 }
```

3. Clientul transmite serverului un nume de fisier iar serverul intoarce clientului continutul fisierului indicat sau un mesaj de eroare in cazul ca fisierul dorit nu exista.

4. Clientul ii transmite serverului un nume de utilizator, iar serverul ii returneaza clientului datele la care utilizatorul respectiv s-a conectat.

5. Clientul ii transmite serverului un nume de server Unix, si primeste lista tuturor utilizatorilor care lucreaza in acel moment la serverul respectiv.

6. Clientul ii transmite serverului un nume de utilizator iar serverul ii intoarce clientului numarul de procese executate de utilizatorul respective.

7. Clientul ii transmite serverului un nume de fisier si primeste de la acesta un mesaj care sa indice tipul fisierului sau un mesaj de eroare in cazul in care fisierul nu exista.

8. Clientul ii transmite serverului un nume de director si primeste de la acesta lista tuturor fisierelor text din directorul respectiv, respectiv un mesaj de eroare daca directorul respectiv nu exista.

9. Clientul ii transmite serverului un un nume de director, iar serverul ii retransmite clientului numarul total de bytes din toate fisierele din directorul respectiv.

10. Clientul ii transmite serverului un nume de fisier iar serverul intoarce clientului numarul de linii din fisierul respectiv.

11. Clientul ii transmite serverului un nume de fisier si un numar octal. Serverul va verifica daca fisierul respectiv are drepturi de acces diferite de numarul indicat. Daca drepturile coincid, va transmite mesajul "Totul e OK!" daca nu va seta drepturile conform numarului indicat si va transmite mesajul "Drepturile au fost modificate".

12. Clientul ii transmite serverului un nume de director iar serverul ii intoarce clientului continutul directorului indicat, respectiv un mesaj de eroare in cazul in care acest director nu exista.

13. Clientul ii transmite serverului un nume de utilizator, iar serverul ii intoarce clientului numele complet al utilizatorului si directorul personal.

14. Clientul ii transmite serverului un nume de fisier, iar serverul ii intoarce clientului numele tuturor directoarelor care contin fisierul indicat.

15. Clientul ii transmite serverului un nume de utilizator, iar serverul ii returneaza informatiile indicate de "finger" pentru utilizatorul respectiv, respectiv un mesaj de eroare daca numele respectiv nu indica un utilizator recunoscut de sistem.

16. Clientul ii transmite serverului o comanda Unix, iar serverul o executa si retransmite clientului rezultatul executiei. In cazul in care comanda este invalida, serverul va transmite un mesaj corespunzator.