4. Comunicarea între procese Unix: pipe, FIFO, popen, dup2, shm

Contents

4. COMUNICAREA ÎNTRE PROCESE UNIX: PIPE, FIFO, POPEN, DUP2, SHM 1
4.1. PRINCIPALELE APELURI SISTEM DE COMUNICARE ÎNTRE PROCESE 1
4.2. ANALIZAŢI TEXTUL SURSĂ 1
4.3. UTILIZĂRI SIMPLE PIPE ŞI FIFO 2
4.4. JOCUL NIM; EXEMPLU CU F JUCĂTORI 4
4.5. SIMULARE SH PENTRU WHO | SORT ŞI WHO | SORT | CAT (DUP2) 7
4.6. PARADIGMA CLIENT / SERVER; EXEMPLE 8
4.7. EXEMPLE DE UTILIZARE POPEN 10
4.8. UTILIZARE SHM 11
4.9. PROBLEME PROPUSE 12

4.1. Principalele apeluri sistem de comunicare între procese

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

Funcţii specifice comunicării	
`pipe(f)`	Crează doi descriptori de fişier pentru citire / scriere in **pipe**
`mkfifo(nume, drepturi)`	Crează un FIFO
`FILE *popen(lcs, "r|w")`	Lanseaza un sh într-un fiu, executa **lcs** şi captează **stdout** / **stdin**
`pclose(FILE *)`	Inchide **popen**
`dup2(fo,fn)`	Crează o copie a descriptorului de fişier **fo** în poziţia **fn**

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

- `lcs` este o linie de comandă interpretabilă de către shell, deci se vor trata: ${ } ` ` * ? < > << >>
- `f` este un tablou de (cel puţin) doi intregi – descriptori de citire (poziţia 0) / scriere din / in pipe (poziţia 1);
- `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 apărut.

Pentru amatori, vezi sursa `popen` https://github.com/lattera/freebsd/blob/master/lib/libc/gen/popen.c

4.2. Analizaţi textul sursă

Considerând că toate instrucțiunile din fragmentul de cod (`pipe1.c`) 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 13? Specificați relația părinte fiu dintre aceste procese.

c) Câte procese se creează, incluzând procesul inițial, dacă mutăm instrucțiunea de pe linia 13 pe linia 16 (pornind de la codul dat)? Specificați relația părinte fiu dintre aceste procese.
d) Ce va tipări rularea codului, dacă liniile 22 și 23 se mută în interiorul ramurii else, începând cu linia 16 a codului inițial? Justificați răspunsul.

```
1  #include <stdio.h>
2  #include <sys/wait.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  int main() {
6      int pfd[2], i, n;
7      pipe(pfd);
8      for(i=0; i<3; i++) {
9          if(fork() == 0) {
10             close(pfd[0]);
11             write(pfd[1], &i, sizeof(int));
12             close(pfd[1]);
13             exit(0);
14         }
15         else {
16             // a se vedea punctele c) si d)
17         }
18     }
19     close(pfd[1]);
20     for(i=0; i<3; i++) {
21         wait(0);
22         read(pfd[0], &n, sizeof(int));
23         printf("%d\n", n);
24     }
25     close(pfd[0]);
26     return 0;
27 }
```

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.

4.3. 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);
    }
    // Procesul parinte
    a[2]+=a[3];
    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]);
        a[0]+=a[1];
        write(f[1], &a[0], sizeof(int));
        close(f[1]);
        exit(0);
    }
    // Procesul parinte
    close(f[1]);
    a[2]+=a[3];
    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 face înainte de a lansa procesele care o utilizează, de exemplu, prin comanda:

```
$ mkfifo /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>
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));
        close(f);
        exit(0);
    }
    // Procesul parinte
    a[2]+=a[3];
    f = open("/tmp/fifo1", O_RDONLY);
    read(f, &a[0], sizeof(int));
    close(f);
    wait(NULL);
    a[0]+=a[2];
    printf("Suma este %d\n", a[0]);
}
```

Natural, atunci când nu mai avem nevoie de acest FIFO, el se şterge cu comanda:

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

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

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.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]);
}
```

add4fFiu.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.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 unul dintre ele (sau ambele), poate fi lansat în background.

4.4. Jocul nim; exemplu cu F jucători

Este vorba de un joc al marinarilor: se considera **G** grămezi, fiecare grămadă având un număr de pietre, cel puţin una). Joacă, de regulă, doi jucători, dar pot juca şi mai mulţi, eventual grupaţi în două echipe. Notăm cu **F** numărul de jucători.

La fiecare pas, jucătorul aflat la mutare elimină un număr nenul de pietre (eventual toate) doar dintr-o singură grămadă. Jucătorii mută alternativ (sau într-o ordine prestabilită daca sunt mai mult de doi jucători). **Jucătorul care ia ultimele pietre** (lasă grămezile goale) **pierde**.

Indiferent de numărul de jucători şi de dimensiunile grămezilor, strategia câştigătoare este unică: se face suma modulo 2 (sau exclusiv, operatorul ^ din C) a totalului pietrelor din grămezi. Să notăm cu **xor** această sumă. Fiecare *jucător urmăreşte să extragă dintr-o grămadă, dacă poate, atâtea pietre încât să îi dea următorului jucător un **xor** nenul*. Dacă un jucător primeşte **xor** cu valoare nenulă, atunci **el este posibil să piardă**, iar dacă sunt numai doi jucători şi fiecare din ei respectă regula, **el va pierde sigur**. Evident, dacă nu este posibil să se obţină **xor** nenul, atunci se poate extrage la întâmplare, de exemplu o singură piatră din grămada cea mai mare, caz în care următorul jucător are şanse de câştig.

Sursa următoare implementează un joc **nim** la care participă **F** jucători şi sunt **G** grămezi de pietre:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <time.h>

#define MAX(a,b) ((a > b) ? a : b)
#define MIN(a,b) ((a < b) ? a : b)

int G, F, g, f, i, r;    // nr gramezi (max 10), nr fii (max 10)
int p[20];               // canalele pipe intre fii
unsigned int pietre[10]; // gramezile de pietre

void list(char *s) {
    printf("Jucator: %d %s [ ", f, s);
    for (int i = 0; i < G; i++) printf("%03u ",pietre[i]);
    printf("]\n");
    fflush(stdout);
```

```
}

int scoate() {
    int gm, gs, k; // gm unde sunt cele mai multe pietre, gs de unde scot
    unsigned int scot, xor; // cate scot din gs, valoarea sumei nim
    for (gm = 0, k = 0; k < G; k++)
        if (pietre[gm] < pietre[k])
            gm = k;
    if (pietre[gm] == 0) return 0;
    for (scot = pietre[gm]; scot > 0; scot--) {
        for ( gs = 0; gs < G; gs++) {
            if (pietre[gs] < scot) continue;
            for (k = 0, xor = 0; k < G; k++) {
                if (k == gs) xor ^= (pietre[k] - scot);
                else         xor ^= pietre[k];
            }
            if (xor != 0) {
                pietre[gs] -= scot;
                printf("\tJucator %d: din pietre[%d] scoate %d\n", f, gs, scot);
                return scot;
            }
        }
    }
    pietre[gm]--;
    printf("\tJucator %d: din pietre[%d] scoate 1 si poate pierde\n", f, gm);
    return 1;
}

int main (int argc, char **argv) {
    srand(time(0));
    G = MAX(2, MIN((argc > 1)? atoi(argv[1]): 2, 10));
    F = MAX(2, MIN((argc > 2)? atoi(argv[2]): 2, 10));
    for (i = 0; i < G; i++) pietre[i] = rand() % 100 + 1;
    //G = 3;//test
    //pietre[0] = 7; pietre[1] = 5; pietre[2] = 3;//test
    //pietre[0] = 1; pietre[1] = 1;//test
    //pietre[0] = 0; pietre[1] = 0;//test
    for (f = 0; f < 2*F; f += 2) pipe(&p[f]);
    for (f = 0; f < F; f++) {
        if (fork() == 0) {
            if (f < F - 1) {
                for (i = 0; i < 2*F; i++)
                    if (i != 2*f && i != 2*f + 3)
                        close(p[i]);
            } else { // Ultimul scrie la primul
                for (i = 0; i < 2*F; i++)
                    if (i != 2*f && i != 1)
                        close(p[i]);
            }
            for ( ; ; ) {
                r = read(p[2*f], pietre, G*sizeof(unsigned int));
                if (r <= 0) { // p[2*f] inchis
                    printf("Jucatorul %d nu mai are intrarea p[%d]\n", f, 2*f);
                    fflush(stdout);
                    close(p[2*f]);
                    if (f < F - 1) close(p[2*f + 3]); else close(p[1]);
                    exit(0);
                }
                list("intrare");
                i = scoate();
                list("scos   ");
                if (i == 0) { // gramezi fara pietre
                    printf("Jucatorul %d a nu a mai primit pietre\n", f);
                    fflush(stdout);
                    close(p[2*f]);
                    if (f < F - 1) close(p[2*f + 3]); else close(p[1]);
                    exit(0);
                }
```

```
                    for (i = 0, r = 0; i < G; i++) r += pietre[i];
                    if (r == 0) { // f a pierdut
                        printf("Jucatorul %d a PIREDUT!\n", f);
                        fflush(stdout);
                        close(p[2*f]);
                        if (f < F - 1) close(p[2*f + 3]); else close(p[1]);
                        exit(0);
                    }
                    if (f < F - 1)
                        write(p[2*f + 3], pietre, G*sizeof(unsigned int));
                    else
                        write(p[1], pietre, G*sizeof(unsigned int));
                }
            }
        }
        // Parinte
        f = -1;
        list("start ");
        write(p[1], pietre, G*sizeof(unsigned int)); // Scrie primului fiu
        for (f = 0; f < 2*F; f++) close(p[f]);       // Inchide pipe-urile
        for (f = 0; f < F; f++) wait(NULL);          // Asteapta dupa fii
    }
```

Daca se lansează **`./a.out 8 3`** (**G**=8 şi **F**=3) atunci un posibil rezultat ar putea fi:

```
Jucator: -1 start  [ 001 022 060 033 017 061 041 030 ]
Jucator: 0 intrare [ 001 022 060 033 017 061 041 030 ]
     Jucator 0: din pietre[5] scoate 61
Jucator: 0 scos    [ 001 022 060 033 017 000 041 030 ]
Jucator: 1 intrare [ 001 022 060 033 017 000 041 030 ]
     Jucator 1: din pietre[2] scoate 60
Jucator: 1 scos    [ 001 022 000 033 017 000 041 030 ]
Jucator: 2 intrare [ 001 022 000 033 017 000 041 030 ]
     Jucator 2: din pietre[6] scoate 41
Jucator: 2 scos    [ 001 022 000 033 017 000 000 030 ]
Jucator: 0 intrare [ 001 022 000 033 017 000 000 030 ]
     Jucator 0: din pietre[3] scoate 33
Jucator: 0 scos    [ 001 022 000 000 017 000 000 030 ]
Jucator: 1 intrare [ 001 022 000 000 017 000 000 030 ]
     Jucator 1: din pietre[7] scoate 30
Jucator: 1 scos    [ 001 022 000 000 017 000 000 000 ]
Jucator: 2 intrare [ 001 022 000 000 017 000 000 000 ]
     Jucator 2: din pietre[1] scoate 22
Jucator: 2 scos    [ 001 000 000 000 017 000 000 000 ]
Jucator: 0 intrare [ 001 000 000 000 017 000 000 000 ]
     Jucator 0: din pietre[4] scoate 17
Jucator: 0 scos    [ 001 000 000 000 000 000 000 000 ]
Jucator: 1 intrare [ 001 000 000 000 000 000 000 000 ]
     Jucator 1: din pietre[0] scoate 1 si poate pierde
Jucator: 1 scos    [ 000 000 000 000 000 000 000 000 ]
Jucatorul 1 a PIREDUT!
Jucatorul 2 nu mai are intrarea p[4]
Jucatorul 0 nu mai are intrarea p[0]
```

Câteva cuvinte despre implementare.

Mai întâi am definit două macrouri **MIN** şi **MAX**, pe care le folosim pentru comoditatea implementării.

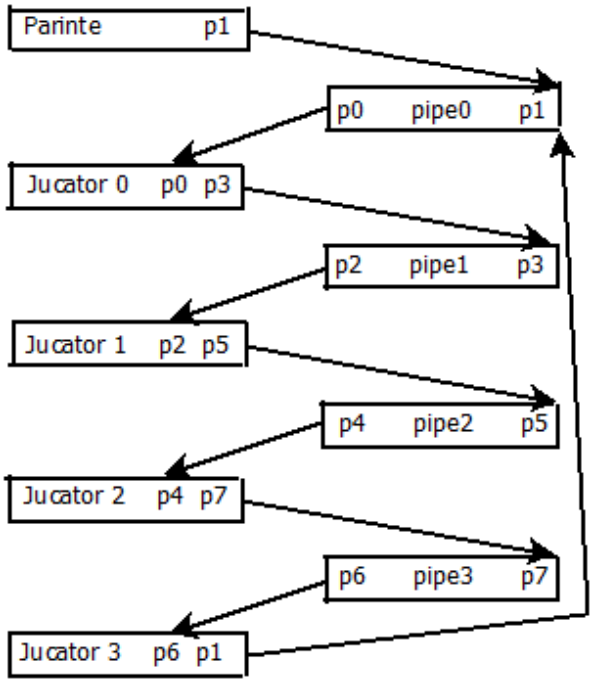
Funcţia **list** afişează pentru un jucător **f** numerele de pietre din fiecare dintre cele **G** grămezi.

Funcţia **scoate** caută cel mai mare număr de pietre care se poate scoate dintr-o grămadă aşa încât cantităţile rămase în grămezi să aibă xor nenul. Dacă nu se poate, atunci scoate o piatră din grămada cea mai mare. Intoarce numărul de pietre scoase, sau 0 dacă funcţia nu primeşte pietre în nici o grămadă.

Funcţia **main** obţine valorile **F** şi **G**. Apoi generează aleator dimensiunea fiecărei grămezi.

Apoi se crează **F** canale pipe şi **F** procese fii care comunică între ele prin aceste pipe-uri. Descriptorii pipe-urilor sunt memorate în vectorul **p**. Comunicarea prin aceste pipe-uri se desfăşoară după cum urmează: părintele scrie vectorul cu dimensiunile grămezilor spre primul jucător (primul proces fiu) prin **p[1]**. Apoi părintele închide toate pipe-urile şi aşteaptă terminarea fiilor.

Să notăm **f** unul dintre cei **F** jucători (procese fiu). Jucătorul **f** foloseşte capătul **p[2*f]** pentru citire, iar pentru scriere capătul **p[2*f+3]** dacă **f** nu este ultimul, respectiv scrie în **p[1]** - către primul jucător -dacă **f** este ultimul jucător. Imediat după creare se închid toate capetele pipe care nu sunt necesare, adică fac excepţie capetele spuse mai sus. Jucătorul f citeşte dimensiunile trimise de precedentul jucător, testează posibilele terminări ale jocului, iar dacă acesta încă nu se termină, acrie noua configuraţie de grămezi către următorul jucător. Figura următoare ilustrează comunicările între procese pentru cazul în care **F** = 4.



4.5. 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./t 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	**who | sort | cat**

`#include <stdio.h>`
`#include <unistd.h>`
`#include <sys/wait.h>`
`int main () {`
`  int p[2];`
`  pipe (p);`
`  if (fork() == 0) {`
`  close (p[0]);`
	`#include <stdio.h>`
	`#include <unistd.h>`
	`#include <sys/wait.h>`
	`int main () {`
	`  int p[2], q[2];`
	`  pipe (p);`
	`  pipe (q);`
	`  if (fork() == 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);
    }
}
```

```
        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.

4.6. 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 **exact** 20 caractere:

`numar` întreg pozitiv fără semn, maximum 10 cifre aliniat la dreapta si completat cu spaţii la stânga	**`nume`** este numele unui fişier, maximum 10 caractere, aliniat la stânga, cu zerou terminal şi completat cu spaţii la dreapta.

Serverul trebuie pornit înaintea clienţilor! La pornire îşi crează mai întâi FIFO-ul **`/tmp/CERERE`** după care intră în bucla de aşteptare a cererilor de la clienţi. La primirea unei cereri descompune **`numar`** în factori primi şi scrie în FIFO-ul **`/tmp/nume`** mai întâi lungimea răspunsului urmată de descompunerea numărului.

Programul **client** preia de la linia de comandă **`număr`** de descompus şi **`PID`**-ul procesului server (spre a-l putea trezi pe acesta!). Apoi construieşte **`nume`** de fişier care să reprezinte unic procesul client (nume putând să fie, de exemplu, PID-ul procesului client completat la dreapta cu spaţiii). Clientul formează cererea ca mai sus şi o scrie în FIFO-ul **`/tmp/CERERE`**. Apoi citeşte răspunsul de la server din FIFO-ul **`/tmp/nume`** (întâi lungimea, apoi descompunerea) şi tipăreşte răspunsul primit pe ieşirea lui standard, după care şterge fişierul.

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.

Intrebări:

1. De ce este nevoie ca cererea să aibă această structură rigidă?
2. De ce răspunsul se dă tot într-un FIFO şi nu într-un fişier obişnuit?
3. Este neapărată nevoie de utilizarea semnalelor pentru adormire / trezire?

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

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <signal.h>
int main (int argc, char* argv[]) {
    char cerere[21] = "                    ";
    char nume[16] = "/tmp/          ";
    char raspuns[1000];
    int i, k, fc, fr;
    int pids = atoi(argv[2]);
    for (i = 0, k = 10 - strlen(argv[1]); i < strlen(argv[1]); cerere[k] =
argv[1][i], i++, k++);
    sprintf(&nume[5], "%d", getpid());
    for (i = 0, k = 10; i < strlen(argv[2]); cerere[k] = argv[2][i], i++, k++);
    //kill (pids, SIGCONT);
    fc = open("/tmp/CERERE", O_WRONLY);
    write(fc, cerere, 20);
    close(fc);
    fr = open(nume, O_RDONLY);
    read(fr, &k, sizeof(int));
    read(fr, raspuns, k);
    printf("Cerere: %s Raspuns: %s\n", cerere, raspuns);
    close(fr);
    unlink(nume);
}
```

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.

Serverul, `serv.c` este unul iterativ. Părţile comentate conţin transformarea lui într-unul concurent.

```
#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];
    mkfifo("/tmp/CERERE", 0777);
    int fc = open("/tmp/CERERE", O_RDONLY);
    for ( ; ; ) {
        //kill(getpid(), SIGSTOP);
        read(fc, cerere, 20);
        //if (fork() == 0) {
            char nume[16] = "/tmp/          ";
```

```
            char raspuns[1000];
            char factor[11];
            int i, k, fr;
            for(i = 10, k = 5; i < 20; nume[k] = cerere[i], i++);
            cerere[10] = '\0';
            long n = atol(cerere);
            sprintf(raspuns, "%ld = ", n);
            long d, x;
            for (d = 2; d <= n / 2; d++) {
                for (x = 2, k = 0; x < d / 2; x++) if (d % x == 0) k++;
                if (k > 0) continue;
                for (k = 0; n >= d && n % d == 0; n /= d, k++);
                if (k == 1)
                    sprintf(factor, "%ld * ", d);
                else if (k > 1)
                    sprintf(factor, "%ld^%d * ", d, k);
                if (k >= 1) strcat(raspuns, factor);
            }
            raspuns[strlen(raspuns) - 3] = '\0';
            k = strlen(raspuns);
            fr = mkfifo(nume, 0777);
            fr = open(nume, O_WRONLY);
            write(fr, &k, sizeof(int));
            write(fr, raspuns, k);
            close(fr);
        //    exit(0);
        //}
    }
}
```

4.7. 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", argv[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]);
    }
}
```

4.8. Utilizare shm

Recomandări:
Comanda **ipcs** dă informaţii despre ipc-urile din sistem:

```
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 18         florin     600        524288     2          dest
0xb8d5e071 38         florin     644        500        0
0x00000000 41         florin     600        524288     2          dest

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

Comanda **ipcmk -M** *lungime* permite crearea unui segment de memorie partajată de o anumită lungime.

Comanda **ipcrm -M** *cheie* permite ştergerea unui segment de memorie partajată furnizând cheia, iar
Comanda **ipcrm -m** *id* permite ştergerea unui segment de memorie partajată furnizând id-ul.

Atenţie la muntiaccesul în segmentul de memorie partajată. De regulă, o proiectare judicioasă fece ca unele procese să scrie numei unele variabile din zonă, alte procese altele etc.

La ataşarea la un segment de memorie partajată se poate defini numai access read-only.

Exemplul de mai jos foloseşte memoria partajată pentru a scrie linii formate din acelaşi caracter, de diverse lungimi. aceasta se realizează lansând programul access simultan de mai multe ori, cu diverse argumente.

Furnizăm crearea şi ştergerea segmentului prin program.

shm_litere.h

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
//Antet: defineste structura unui segment de memorie partajata
typedef struct segment_shm {int n; char c;} SEG;
#define CHEIE    (key_t) (12345L)
#define LUNG     sizeof(SEG)
```

creare.c

```
#include "shm_litere.h"
int main () {
    shmget (CHEIE, LUNG, IPC_CREAT |0666);
}
```

delete.c

```
#include "shm_litere.h"
int main () {
    int sd;
    sd = shmget (CHEIE, 0, 0);
    shmctl(sd, IPC_RMID, NULL);
}
```

access.c

```
#include "shm_litere.h"
int main (int argc, char *argv[]) {
    int i, k, sd;
    SEG *p;
    sd = shmget(CHEIE, 0, 0);
    p = (SEG *)shmat(sd, NULL, 0);
    for (k=0; k<1000; k++) {
        p->n = atoi(argv[1]);
        p->c = argv[2][0];
        printf("\n");
        for (i = 0; i < p->n; i++)
           putchar(p->c);
    }
}
```

4.9. 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.