

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. ANALIZAȚI TEXTUL SURSĂ	1
4.3. UTILIZĂRI SIMPLE PIPE ȘI FIFO	2
4.4. JOUL NIM; EXEMPLU CU F JUCĂTORI	4
4.5. SIMULARE SH PENTRU WHO SORT ȘI 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	15

4.1. Principalele apeluri sistem de comunicare între procese

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

Funcții specifice comunicării	
<code>pipe(f)</code>	Crează doi descriptori de fișier pentru citire / scriere în pipe
<code>\$ mkfifo nume</code> sau <code>mkfifo(nume, drepturi);</code>	Crează un FIFO
<code>dup2(fo, fn)</code>	Crează o copie a descriptorului de fișier fo în poziția fn
<code>FILE *popen(lcs, "r w")</code> <code>pclose(FILE *)</code>	Lansează un sh într-un fiu, execută lcs și captează stdout / stdin Inchide popen
<code>\$ ipcs, \$ ipcmk, \$ ipcrm</code> <code>shmget shmat shmdt shmctl</code>	Comenzi Unix de lucru cu IPC (particular memorie partajată) Funcții de lucru cu memorie partajată

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

- `lcs` este o linie de comandă **interpretabilă de către shell**, deci se vor trata: `{ } ` * ? < > << >>`
- `f` este un tablou de (cel puțin) doi intregi – descriptori de citire (poziția 0) / scriere din / in pipe (poziția 1);
- `nume` este numele (de pe disc) al fișierului FIFO, iar `drepturi` sunt drepturile de acces la acesta;
- `fo` și `fn` descriptori de fișiere: `fo` deschis în program cu `open`, `fn` poziția în care e duplicat `fo`.

În caz de eșec, funcțiile întorc -1 (NULL la `popen`) și poziționează `errno` se depistează ce eroare a apărut. Se poate vedea sursa `popen` <https://github.com/lattera/freebsd/blob/master/lib/libc/gen/popen.c>

4.2. Analizați textul sursă

Considerând că toate instrucțiunile din fragmentul de cod (`pipe1.c`) de mai jos se execută cu succes, răspundeți la următoarele întrebări:

- Ce va tipări rularea codului așa cum este?
- Câte procese se creează, incluzând procesul inițial, dacă lipsește linia 13? Specificați relația părinte fiu dintre aceste procese.
- 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.
- 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:

- 0, 1, 2 pe linii separate în orice ordine.
- 8 procese, arbore cu 8 procese.
- 4 procese, arbore cu 4 procese.
- 0, 1, 2 pe linii separate întodeauna în această ordine.

4.3. Utilizări simple pipe și FIFO

Pentru a ilustra modul de lucru cu pipe și 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 știe, suma tipărită va fi 8, nu 10, deoarece informația din procesul fiu nu ajunge în părinte. Vom da trei soluții corecte pentru această problemă, toate vor tipări "**Suma este 10**".

Soluția 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]);
}

```

Soluția 2: comunicarea prin FIFO cu procesele în aceeași sursă. FIFO permite comunicarea între două procese care nu sunt, neapărat, înrudite. Din această cauză, se obișnuiește ca fișierul 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 soluție 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 șterge cu comanda:

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

Soluția 3: comunicarea prin FIFO între două procese create din surse diferite. Tabelul următor prezintă fișierele **add4fTata.c** și **add4fFiu.c** care vor comunica între ele:

add4fTata.c	add4fFiu.c
<pre>#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]); }</pre>	<pre>#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); }</pre>

Înainte de a lansa procesele, trebuie creat FIFO. Procesele se pot lansa în orice ordine, deoarece se așteaptă 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 puțin una). Joacă, de regulă, doi jucători, dar pot juca și mai mulți, eventual grupați 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 dacă sunt mai mult de doi jucători). **Jucătorul care ia ultimele pietre (lasă grămezile goale) pierde.**

Indiferent de numărul de jucători și de dimensiunile grămezilor, strategia câștigătoare 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 urmărește să extragă dintr-o grămadă, dacă poate, atâtea pietre încât să îi dea următorului jucător un **xor** nenul.*

Dacă sunt doar doi jucători și ambii respectă strategia, cel ce primește **xor nenul va pierde**, iar cel ce primește **xor nul va câștiga**.

Dacă sunt mai mult de doi jucători, cel care primește xor nul are șanse să piardă.

Sursa următoare implementează un joc **nim** la care participă **F** jucători și 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--) { // Se poate scoate maxim posibil
        //for (scot = 1; scot <= pietre[gm]; scot++) { // Se poate scotoate minim posibil
        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; // Sunt sanse de castig
                printf("\tJucator %d: din pietre[%d] scoate %d\n", f, gs, scot);
                return scot;
            }
        }
    }
    pietre[gm]--; // Scos o piatra la intamplare; sunt sanse de pierdere
    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();
        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);
        }
        list("scos  ");
        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** și **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** și **MAX**, pe care le folosim pentru comoditatea implementării.

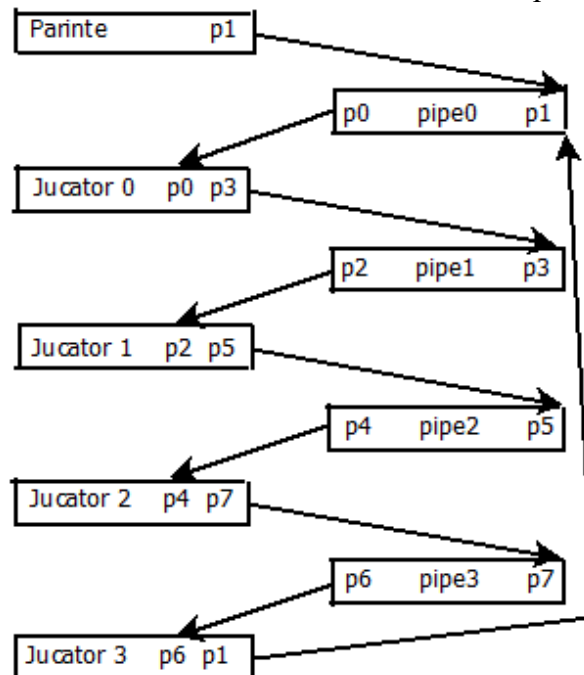
Funcția **list** afișează pentru un jucător **f** numerele de pietre din fiecare dintre cele **G** grămezi.

Funcția **scoate** caută cel mai mare număr de pietre care se poate scoate dintr-o grămadă așa încât cantitățile 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 funcția nu primește pietre în nici o grămadă.

Funcția **main** obține valorile **F** și **G**. Apoi generează aleator dimensiunea fiecărei grămezi.

Apoi se crează **F** canale pipe și **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 desfășoară după cum urmează: părintele scrie vectorul cu dimensiunile grămezilor spre primul jucător (primul proces fiu) prin **p[1]**. Apoi părintele închide toate pipe-urile și așteaptă terminarea fiilor.

Să notăm **f** unul dintre cei **F** jucători (proces fiu). Jucătorul **f** folosește 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 excepție capetele spuse mai sus. Jucătorul **f** citește dimensiunile trimise de precedentul jucător, testează posibilele terminări ale jocului, iar dacă acesta încă nu se termină, acrie noua configurație de grămezi către următorul jucător. Figura următoare ilustrează comunicările între procese pentru cazul în care **F** = 4.



4.5. Simulare sh pentru `who | sort și who | sort | cat (dup2)`

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

Pentru simulare, programul principal va crea două procese fii în care va lansa, prin `exec`, comenzile `who` și `sort`. Înainte de crearea acestor fii, va crea un pipe pe care îl va da celor doi fii ca să comunice între ei: `who` își va redirecta ieșirea standard în 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 ușor de făcut). Sursele celor două programe sunt date în tabelul următor:

who sort	who sort cat
<pre>#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); } }</pre>	<pre>#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); } }</pre>

Principiul este simplu: dacă avem n comenzi în pipeline, atunci trebuie construite $n-1$ pipe-uri. Procesul ce execută prima comandă își va redirecta ieșirea standard în primul pipe. Procesul ce execută ultima comandă își va redirecta intrarea standard în ultimul pipe. Procesele ce execută comenzile intermediare, să zicem procesul i cu $i > 1$ și $i < n-1$, va avea ca intrare standard pipe-ul $i-1$ și ca ieșire 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 primește în FIFO-ul `/tmp/CERERE` un string de **exact** 20 caractere:

numar întreg pozitiv fără semn, maximum 10 cifre aliniat la dreapta și completat cu spații la stânga	nume fără spații albe, maximum 10 caractere, aliniat la stânga, completat cu spații la dreapta.
---	--

Un program client **clie.c** preia de la linia de comandă **număr** de descompus și **PID**-ul procesului server (eventual spre a-l putea trezi pe acesta!). Apoi construiește **nume** de fișier care să reprezinte unic procesul client (nume putând să fie, de exemplu, PID-ul procesului client completat la dreapta cu spații). Clientul formează cererea ca mai sus și o scrie în FIFO-ul `/tmp/CERERE`. Apoi crează FIFO-ul `/tmp/nume`

după(întâi lungimea, apoi descompunerea) și tipărește răspunsul primit pe ieșirea lui standard, după care șterge fișierul.

Serverul **serv.c** la pornire își crează FIFO-ul **/tmp/CERERE**. Apoi intră în bucla de așteptare a cererilor de la clienți. La primirea unei cereri descompune **numar** în factori primi și scrie în FIFO-ul **/tmp/nume** mai întâi lungimea răspunsului urmată de descompunerea numărului. **Serverul trebuie pornit înaintea clienților!**

Serverul poate lucra în diverse variante, în funcție de anularea sau nu a unora dintre comentariile din sursele **clie.c** și **serv.c**: liniile 18,19 de la **clie.c** și liniile 9, 15, 17, 44, 45 din **serv.c**, astfel:

- Un **server iterativ**, care așteaptă deschiderea **/tmp/CERERE** citește o cerere, o tratează și răspunde clientului. Asta dacă toate liniile precizate mai sus comentate.
- Un **server iterativ**, care așteaptă doar între cereri și este trezit de fiecare client care vrea să îi trimită o cerere. Așteaptă deschiderea **/tmp/CERERE** citește cererea, o tratează și răspunde clientului. Asta dacă sunt decommentate liniile 18,19 din **clie.c** și linia 15 din **serv.c**.
- Un **server concurent**, care crează un proces fiu care așteaptă deschiderea **/tmp/CERERE** citește o cerere, o tratează și răspunde clientului. Asta dacă se decommentează liniile 9, 17, 44, 45 din **serv.c**.
- Un **server concurent**, care doar între cereri și este trezit de fiecare client care vrea să îi trimită o cerere. La trezire crează un proces fiu care așteaptă deschiderea **/tmp/CERERE** citește cererea, o tratează și răspunde clientului. Asta dacă sunt decommentate liniile 18, 19 din **clie.c** și liniile 9, 15, 17, 44, 45 din **serv.c**.

Atragem atenția asupra celor două apeluri sistem open care deschid FIFO de cerere și de răspuns. Sarcina creării FIFO-urilor revine serverului. Ordinea celor două open trebuie să fie aceeași și la server, altfel se ajunge la deadlock.

Intrebări:

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

Sursa **clie.c** este:

```

1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <fcntl.h>
5  #include <sys/stat.h>
6  #include <string.h>
7  #include <signal.h>
8  int main (int argc, char* argv[]) {
9      char cerere[21] = "                ";
10     char nume[16] = "/tmp/                ";
11     char raspuns[1000];
12     int i, k;
13     for(i=0,k=10-strlen(argv[1]);i<strlen(argv[1]);cerere[k]=argv[1][i],i++,k++);
14     sprintf(&nume[5], "%d", getpid());
15     unlink(nume);
16     mkfifo(nume, 0666);
17     for (i = 5, k = 10; i < strlen(nume); cerere[k] =nume[i], i++, k++);
18     //int pids = atoi(argv[2]);
19     //kill (pids, SIGCONT);
20     int fc = open("/tmp/CERERE", O_WRONLY);
21     write(fc, cerere, 20);
22     close(fc);
23     int fr = open(nume, O_RDONLY);
24     read(fr, &k, sizeof(int));
25     read(fr, raspuns, k);
26     printf("Cerere: %s Raspuns: %s\n", cerere, raspuns);

```

```

27     close(fr);
28     unlink(ume);
29 }

```

Sursa **serv.c** este:

```

1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <fcntl.h>
5  #include <sys/stat.h>
6  #include <string.h>
7  # include <signal.h>
8  int main (int argc, char* argv[]) {
9      //signal(SIGCHLD, SIG_IGN);
10     char cerere[21];
11     unlink("/tmp/CERERE");
12     mkfifo("/tmp/CERERE", 0666);
13     int fc = open("/tmp/CERERE", O_RDONLY);
14     for ( ; ; ) {
15         //kill(getpid(), SIGSTOP);
16         read(fc, cerere, 20);
17         //if (fork() == 0) {
18             char nume[17] = "/tmp/          ";
19             char raspuns[1000];
20             char factor[11];
21             int i, k;
22             for(i=10,k=5; i<20 && cerere[i]!=' '; nume[k] = cerere[i], i++, k++);
23             nume[k] = '\0';
24             cerere[10] = '\0';
25             long n = atol(cerere);
26             sprintf(raspuns, "%ld = ", n);
27             long d, x;
28             for (d = 2; n >= d; d++) {
29                 for (x = 2, k = 0; x <= d / 2; x++) if (d % x == 0) k++;
30                 if (k > 0 || n % d != 0) continue;
31                 for (k = 0; n >= d && n % d == 0; n /= d, k++);
32                 if (k == 1)
33                     sprintf(factor, "%ld * ", d);
34                 else
35                     sprintf(factor, "%ld^%d * ", d, k);
36                 strcat(raspuns, factor);
37             }
38             raspuns[strlen(raspuns) - 3] = '\0';
39             k = strlen(raspuns) + 1;
40             int fr = open(nume, O_WRONLY);
41             write(fr, &k, sizeof(int));
42             write(fr, raspuns, k);
43             close(fr);
44             //exit(0);
45         //}
46     }
47 }
48

```

4.7. Exemple de utilizare popen

Utilizare **popen** cu scriere in intrarea standard pentru comenda lansată: de exemplu, scrierea in ordine alfabetică a argumentelor și 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 verifice 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 primește la linia de comandă mai multe nume de fisiere de intrare, care să fie filtrate în procese paralele în fisiere de ieșire. 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

Un prim exemplu folosește memoria partajată pentru a scrie linii formate din același caracter, de diverse lungimi. aceasta se realizează lansând programul **acc_lit** simultan de mai multe ori, cu diverse argumente. Furnizăm sursele header, creare, șterge și acces la memoria partajată prin programe. Aici toate acțiunile asupra memoriei partajate se realizează prin funcțiile sistem shmget, shmat, shmctl

lit.h

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
typedef struct segment_shm {int n; char c;} SEG;
#define KEY      (key_t) (12345L)
#define LUNG      sizeof(SEG)
```

cre_lit.c

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

del_lit.c

```
#include "lit.h"
int main () { shmctl(shmget (KEY, 0, 0), IPC_RMID, NULL); }
```

acc_lit.c

```
#include "lit.h"
int main (int argc, char *argv[]) {
    int i, k;
    SEG *p;
    int shmid = shmget(KEY, 0, 0);
    p = (SEG *)shmat(shmid, 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);
    }
}
```

Se fac compilările:

```
gcc -o c cre_lit.c
gcc -o d del_lit.c
gcc -o a acc_lit.c
./c
./a 60 X >>out & ./a 40 m >>out
./d
```

Fișierul **out** va conține linii cu **X**-uri, cu **m**-uri, eventual intercalate.

Exemplul de mai sus ilustrează că multiaccesul la segmentul de memorie partajată crează probleme. De regulă, o proiectare judicioasă face ca unele procese să scrie numai unele variabile din zonă, alte procese altele etc.

În acest sens propunem **un al doilea exemplu** de programare a unui computerului de bord al unui turism. Dorim să tratăm semnalarea unor evenimente. Un proces **ph** informează computerul la evenimentele de oprire din rulaș a mașinii (la un semafor, risc de accident etc.). Un alt proces, **ps**, informează computerul supra evenimentelor de viraje la stânga. În fine, un proces **po** raportează, din când în când, asupra ultimelor evenimente.

Vom simula evoluția acestui mecanism prin trei procese care în mod aleator intră în acțiune. Procesele **ph** și **ps** vor înregistra în locațiile **mh** și **ms** dintr-un segment de memorie partajată momentele producerii evenimentelor. Procesul **po** va informa asupra ultimelor evenimente produse: momentele la care au acționat ultima dată procesele **po**, **ph** și **ps**, intervalul de timp în care au apărut ultimele evenimente. Mesajele lui **po** sunt de forma:

```

po: 1      Nici un eveniment pana in prezent
po: 1      Nici un eveniment pana in prezent
po: 2      Evenimentul ms: 1   in ultimele 1 sec.
po: 3      Evenimentul ms: 1   in ultimele 2 sec.
po: 4      Evenimentul ms: 1   in ultimele 3 sec.
po: 5      Evenimentul ms: 1   in ultimele 4 sec.
po: 5      Evenimentul ms: 5   in ultimele 0 sec.
po: 5      Evenimentul ms: 5   in ultimele 0 sec.
po: 6      Evenimentul ms: 5   in ultimele 1 sec.
po: 7      Evenimentele ms: 7 si mh: 7 in ultimele 0 sec.
po: 7      Evenimentele ms: 7 si mh: 7 in ultimele 0 sec.
po: 8      Evenimentele ms: 7 si mh: 7 in ultimele 1 sec.
po: 8      Evenimentele ms: 7 si mh: 7 in ultimele 1 sec.
po: 9      Evenimentele ms: 7 si mh: 7 in ultimele 2 sec.
po: 10     Evenimentele ms: 7 si mh: 7 in ultimele 3 sec.
po: 10     Evenimentele ms: 7 si mh: 7 in ultimele 3 sec.
po: 10     Evenimentele ms: 7 si mh: 7 in ultimele 3 sec.
po: 10     Evenimentele ms: 7 si mh: 7 in ultimele 3 sec.
po: 10     Evenimentele ms: 7 si mh: 7 in ultimele 3 sec.
po: 11     Evenimentele ms: 7 si mh: 7 in ultimele 4 sec.
po: 11     Evenimentele ms: 7 si mh: 7 in ultimele 4 sec.
po: 12     Evenimentele ms: 12      si mh: 7 in ultimele 5 sec.
po: 13     Evenimentele ms: 13      si mh: 13 in ultimele 0 sec.
po: 13     Evenimentele ms: 13      si mh: 13 in ultimele 0 sec.
po: 13     Evenimentele ms: 13      si mh: 13 in ultimele 0 sec.
po: 13     Evenimentele ms: 13      si mh: 13 in ultimele 0 sec.
po: 14     Evenimentele ms: 13      si mh: 14 in ultimele 1 sec.

```

Textele sursă sunt:

auto.h

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <time.h>
typedef struct segment_shm {time_t init; time_t mh; time_t ms;} SEG;
#define LUNG    sizeof(SEG)
#define HTIME  8
#define STIME   7
#define OTIME   2

```

po.c

```

#include "auto.h"
int main (int argc, char *argv[]) {
    SEG *p;
    p = (SEG *)shmat(atoi(argv[1]), NULL, 0);
    p->init = time(NULL); p->mh = 0L; p->ms = 0L;
    printf("\n");
    for ( ; ; ) {
        sleep(random() % OTIME);
        long init = p->init;
        long mh = p->mh;
        long ms = p->ms;
        long acum = time(NULL);
        int evhs = (int)(acum - ms);
        if (evhs < (int)(acum - mh)) evhs = (int)(acum - mh);
        int po = (int)(acum - init);
        int ph = (int)(mh - init);
        int ps = (int)(ms - init);
        if (mh == 0 && ms == 0)
            printf("po: %d\tNici un eveniment pana in prezent\n", po);
    }
}

```

```

        else if (mh == 0)
            printf("po: %d\tEvenimentul ms: %d\tin ultimele %d sec.\n", po, ps,
po-ps);
        else if (ms == 0)
            printf("po: %d\tEvenimentul mh: %d\tin ultimele %d sec.\n", po, ph,
po-ph);
        else if (mh <= ms)
            printf("po: %d\tEvenimentele ms: %d\tși mh: %d\tin ultimele %d
sec.\n", po, ps, ph, po-ph);
        else
            printf("po: %d\tEvenimentele ms: %d\tși mh: %d\tin ultimele %d
sec.\n", po, ps, ph, po-ps);
    }
}

```

ps.c

```

#include "auto.h"
int main (int argc, char *argv[]) {
    SEG *p;
    p = (SEG *)shmat(atoi(argv[1]), NULL, 0);
    for ( ; ; ) {
        sleep(random() % STIME);
        p->ms = time(NULL);
    }
}

```

ph.c

```

#include "auto.h"
int main (int argc, char *argv[]) {
    SEG *p;
    p = (SEG *)shmat(atoi(argv[1]), NULL, 0);
    for ( ; ; ) {
        sleep(random() % HTIME);
        p->mh = time(NULL);
    }
}

```

Succesiunea comenzilor de lansare este:

```

florin@ubuntu:~/shm$ ipcmk -M 24
Shared memory id: 19
florin@ubuntu:~/shm$ gcc -o o po.c
florin@ubuntu:~/shm$ gcc -o s ps.c
florin@ubuntu:~/shm$ gcc -o h ph.c
florin@ubuntu:~/shm$ ./o 19 & ./h 19 & ./s 19&

```

Segmentele de memorie partajată pot fi manevrate nu numai prin programe C, ci și prin comenzi Unix. În exemplul de mai sus am creat segmentul de memorie partajată cu comanda **ipcmk -M 24**, care a creat un segment de 24 octeți (în care intră variabilele `init`, `ms` și `mh`). În urma acestei comenzi, sistemul a furnizat id-ul segmentului de memorie partajată, care a fost transmis prin linia de comandă celor trei procese.

Comanda **ipcs** dă informații despre ipc-urile din sistem, în particular despre segmentele de memorie partajată, sub forma:

```

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

```

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

Comanda **ipcrm -M cheie** permite ștergerea unui segment de memorie partajată furnizând cheia.

Comanda **iperm -m id** permite ștergerea unui segment de memorie partajată furnizând id-ul. În exemplul nostru, ștergerea se face prin comanda **iperm -m 19**.

4.9. Probleme propuse

1. Se dă fișierul **grep.c** care conține fragmentul de cod de mai jos și care se compilează în directorul personal al utilizatorului sub numele **grep**. Răspundeți la următoarele întrebări, considerând că toate instrucțiunile se execută cu succes.

- Enumerați și explicați valorile posibile ale variabilei **n**.
- Ce vor afișa pe ecran rulările, dacă directorul curent nu se află în variabila de mediu PATH

- grep grep grep.c
- ./grep grep grep.c
- ./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:

- Desenați ierarhia proceselor create, incluzând și procesul părinte.
- Dați fiecare linie afișată de program, împreună cu procesul care o tipărește.
- Câte caractere sunt citite din pipe?
- Cum este afectată terminarea proceselor dacă lipsește linia 20?
- Cum este afectată terminarea proceselor dacă lipsesc liniile 20 și 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.