

Laboratorul 7

Comunicare Inter-Proces – Pipes

1 Unnamed pipes – pipe-uri anonime

În sistemele UNIX, comunicația între procese înrudite (părinte–copil) se poate realiza folosind *unnamed pipes*, mecanism prin care datele pot fi trimise unidirecțional de la un proces la altul (de obicei părinte → copil).

Spre deosebire de *unnamed pipes*, *named pipes* pot fi folosite și între procese neînrudite, dar despre ele vom discuta mai târziu în laborator.

Funcția de sistem pentru crearea unui pipe este:

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

unde:

- `fildes[0]` — capăt de citire;
- `fildes[1]` — capăt de scriere.

În caz de succes, `pipe` întoarce 0, altfel -1 și setează `errno`.

Moștenirea pipe-urilor

Un pipe este moștenit automat de procesul copil creat cu `fork(2)`.

Pentru redirectionarea output-ului părintelui în input-ul copilului:

1. Părintele scrie în `fildes[1]`;
2. Copilul citește din `fildes[0]`;
3. Se închid capetele nefolosite în ambele procese.

Exemplu: părinte → copil

Program în care părintele trimite un mesaj copilului prin pipe:

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <string.h>
4 #include <errno.h>
5
6 int main() {
7     int fd[2];
8     pid_t pid;
```

```

9     char buf[100];
10
11     if (pipe(fd) < 0) {
12         perror("pipe");
13         return errno;
14     }
15
16     pid = fork();
17     if (pid < 0) return errno;
18
19     if (pid == 0) {
20         // Proces copil
21         close(fd[1]); // inchidem capatul de scriere
22         read(fd[0], buf, sizeof(buf));
23         printf("Child received: %s\n", buf);
24         close(fd[0]);
25     } else {
26         // Proces parinte
27         close(fd[0]);
28         char msg[] = "Hello from parent";
29         write(fd[1], msg, strlen(msg) + 1);
30         close(fd[1]);
31     }
32
33     return 0;
34 }

```

2 Redirectare output părinte în input-ul copilului folosind pipes

În exemplul de mai jos, codul redirecționează output-ul procesului părinte în input-ul procesului copil.

Ca și în exemplul anterior, creăm un pipe prin care să putem comunica din părinte în copil, dar folosim și funcția `dup2`:

- în părinte duplicăm capătul de scriere al pipe-ului peste `STDOUT_FILENO` (standard output);
- în copil duplicăm capătul de citire al pipe-ului peste `STDIN_FILENO` (standard input).

Astfel, orice scrie părintele la `stdout` ajunge în pipe, iar copilul citește de la `stdin` ceea ce a fost scris de părinte.

Acest comportament este folosit la implementarea operatorului `|` în consolă. De exemplu:

```
cat nume_fisier | grep string
```

crează procesul `cat` cu `stdout` redirecționat în `stdin`-ul procesului `grep`, astfel încât tot ce afișează `cat` (conținutul fișierului) devine input pentru `grep`.

Cod

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <errno.h>
4
5 int main() {
6     int fd[2];
7     pid_t pid;

```

```

8
9  if (pipe(fd) < 0) {
10     perror("pipe");
11     return errno;
12 }
13
14 pid = fork();
15 if (pid < 0) {
16     perror("fork");
17     return errno;
18 }
19
20 if (pid == 0) {
21     /* Proces copil */
22     close(fd[1]); // copilul nu scrie in pipe
23
24     dup2(fd[0], STDIN_FILENO); // redirectam stdin -> citirea din pipe
25     close(fd[0]); // nu mai avem nevoie de vechiul descriptor
26
27     char buf[100];
28     scanf("%s", buf); // citim mesajul de la parinte
29     printf("Copilul a primit: %s\n", buf);
30 } else {
31     /* Proces parinte */
32     close(fd[0]); // parintele nu citește din pipe
33
34     dup2(fd[1], STDOUT_FILENO); // redirectam stdout -> scriere in pipe
35     close(fd[1]);
36
37     printf("Salut"); // mesajul va ajunge la copil prin pipe
38 }
39
40 return 0;
41 }

```

3 Named Pipes (FIFO)

Pe lângă pipe-urile anonime (*unnamed*), UNIX oferă pipe-uri cu nume (*FIFO*), folosite pentru comunicare între procese independente.

Acestea se creează folosind:

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

- `pathname` reprezintă numele dat pipe-ului, folosit de toate procesele care vor să îl deschidă;
- `mode` reprezintă drepturile cu care creăm pipe-ul, având aceleași valori ca și drepturile pentru fișiere.

Accesul se face cu `open(2)`, `read(2)`, `write(2)`, ca pe fișiere.

Exemplu: FIFO cu dialog în ambele sensuri

Primul proces trimite date spre al doilea, iar al doilea răspunde pe același FIFO.

Observație: comunicarea în ambele direcții pe același FIFO poate crea probleme fără sincronizare. Dacă vrem să trimitem date simultan din ambele părți, se folosesc două FIFO-uri. Totuși, putem folosi unul singur dacă trimitem date pe rând.

Creare FIFO

Din consolă:

```
mkfifo myfifo
```

`mkfifo` poate fi apelat și programatic dintr-un proces. Este folosit doar în primul proces care creează pipe-ul; toate procesele ulterioare îl accesează folosind `open`, cu numele FIFO-ului.

Programul 1 — trimite și așteaptă răspuns (writer.c)

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4 #include <string.h>
5
6 int main() {
7     int fd = open("myfifo", O_RDWR);
8     char msg[] = "Ping from process 1";
9     char buf[100];
10
11     write(fd, msg, strlen(msg) + 1);
12     read(fd, buf, sizeof(buf));
13
14     printf("Process1 received: %s\n", buf);
15     close(fd);
16 }
```

Programul 2 — primește și răspunde (reader.c)

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4 #include <string.h>
5
6 int main() {
7     int fd = open("myfifo", O_RDWR);
8     char buf[100];
9
10     read(fd, buf, sizeof(buf));
11     printf("Process2 received: %s\n", buf);
12
13     char reply[] = "Pong from process 2";
14     write(fd, reply, strlen(reply) + 1);
15
16     close(fd);
17 }
```

Execuție

Terminal 1:

```
gcc writer.c -o writer
./writer
```

Terminal 2:

```
gcc reader.c -o reader
./reader
```

4 Sarcini de laborator

1. **Părinte → Copil:** creați un program în care părintele trimite o listă de numere, copilul calculează suma și o afișează (folosind pipe anonim).
2. **Chat cu FIFO:** folosiți două terminale pentru a implementa un mic program tip chat cu două FIFO-uri:
 - `fifo1` — $A \rightarrow B$
 - `fifo2` — $B \rightarrow A$