

Capitolul 7

Comunicarea între procese

Fiecare proces operează în propriul său spațiu de adrese virtuale, iar de evitarea interferențelor dintre procese se ocupă sistemul de operare. Implicit un proces nu poate comunica cu un alt proces numai dacă face uz de diverse mecanisme de comunicare gestionate de *kernel* (nucleul sistemului de operare). Există totuși diverse situații în care procesele trebuie să coopereze, să împartă resurse comune sau să își sincronizeze acțiunile. Există mai multe metode de comunicare între procese:

- fișiere – este cel mai simplu mecanism de comunicare interproces. Un proces scrie într-un fișier iar celălalt proces citește din fișier.
- semnale – sunt utilizate pentru a semnala asincron evenimente între procese
- *pipes* – un *pipe* conectează ieșirea standard a unui proces cu intrarea standard a altuia.
- cozi de mesaje – este o listă legată de spațiul de adresare a *kernel*-ului
- semafoare – sunt contoare utilizate pentru a gestiona accesul la resursele utilizat în comun de mai multe procese. Cel mai des sunt folosite ca și mecanism de blocare a unei resurse pentru a preveni accesul simultan a mai multor procese la o anumită resursă.
- *Sockets*- permit realizarea de conexiuni între procese locale sau în rețea.

7.1. Semnale

Una dintre cele mai vechi metode de comunicare între procese în sistemele de tip Unix(Linux) se bazează pe semnale. O caracteristică importantă a acestor semnale este că ele sunt asincrone, adică un proces poate primi un semnal la orice moment și trebuie să fie pregătit să îi răspundă. Dacă un semnal survine pe perioada execuției unui apel sistem, acesta se încheie prin returnarea unui cod de eroare(EINTR) și este sarcina procesului să refacă apelul întrerupt. Există mai multe tipuri de semnale și fiecare putând fi accesat utilizând pe baza unui nume

simbolic. Fiecare nume unic reprezintă o abreviere ce începe cu *SIG* (de ex. *SIGINT*). O listă completă a setului de semnale din sistem se poate obține prin intermediul comenzii *kill -l*. În continuare se prezintă o listă cu numărul și numele simbolic a semnalelor folosite în mod uzual:

- 1) *SIGHUP* – terminal închis (oprește procesul).
- 2) *SIGINT* – întrerupere de la tastatură (oprește procesul).
- 3) *SIGQUIT* – oprește procesul (oprește procesul și creează un fișier core).
- 4) *SIGILL* – acțiune ilegală (oprește procesul și creează un fișier core).
- 5) *SIGTRAP* 6) *SIGIOT* 7) *SIGBUS* 8) *SIGFPE*
- 9) *SIGKILL* – termină procesul.
- 10) *SIGUSR1* 11) *SIGSEGV* 12) *SIGUSR2*
- 13) *SIGPIPE* 14) *SIGALRM* 15) *SIGTERM*
- 17) *SIGCHLD* – procesul fiu oprit sau terminat
- 18) *SIGCONT* 19) *SIGSTOP* 20) *SIGTSTP* 21) *SIGTTIN*
- 22) *SIGTTOU* 23) *SIGURG* 24) *SIGXCPU* 25) *SIGXFSZ*
- 26) *SIGVTALRM* 27) *SIGPROF* 28) *SIGWINCH* 29) *SIGIO*
- 30) *SIGPWR*

De studiat! Consultați pagina de manual pentru *signal* din secțiunea 7.

Fiecare tip de semnal are asociată o acțiune ce va fi executată de nucleul sistemului de operare (*kernel*) asupra procesului când procesul primește semnalul respectiv. Implicit un proces poate trata un semnal în următoarele moduri:

- Își încheie execuția
- Ignoră semnalul – două semnale nu pot fi ignorate: *SIGKILL* – care termină execuția procesului și *SIGSTOP* care îi oprește execuția.
- Reface acțiunea implicită a semnalului.

La primirea unui semnal un proces poate alege modul de tratare a acestuia, o alternativă este să rămână la acțiunea implicită, iar tratarea va fi făcută de *kernel*. De exemplu semnalul *SIGSTOP* va modifica starea curentă a procesului care îl primește în *STOPPED* și apoi va solicita *kernelului*(*schedulerului*) sau ruleze un alt proces. Alternativ procesul poate specifica pentru un anumit semnal propria rutină de tratare. Această rutină va fi apelată de fiecare dată când este generat semnalul respectiv, iar adresa acestei este reținută în structura *sigaction*. Apelul acestei rutine va fi făcut de către *kernel*, deci procesul va rula în mod *kernel* pe parcursul tratării semnalului.

Nu orice proces din sistem poate trimite semnal oricărui alt proces, acest lucru este valabil numai pentru procesele ce aparțin *kernelului* sau superutilizatorului (*root*). Procesele normale pot trimite semnale numai proceselor cu același UID și GID, sau proceselor din același grup.

Generarea unui semnal se realizează prin setarea bit-ului corespunzător din structura *task_struct*. Dacă procesul nu a blocat semnalul și se află în așteptare dar este întreruptibil atunci starea acestuia este modificată în *Running*, iar procesul este trecut în coada de execuție. În acest fel procesul va fi luat în calcul între procesele care vor primi timp de procesor.

Semnalele nu sunt transmise proceselor imediat ce sunt generate, ele trebuie să aștepte până în momentul în care procesul rulează din nou. De fiecare dată când un proces părăsește un apel sistem sunt verificate câmpurile sale *signal* și *blocked*, iar dacă pentru respectivul proces sunt semnale neblockate ele pot fi acum transmise procesului.

La nivel de interpretor de comenzi o modalitate de trimitere a semnalelor către procese este prin intermediul comenzii *kill*. Un exemplu clasic de terminare a unui proces este următorul:

```
ps | grep processname
kill -9 PID
sau
kill -KILL PID
```

În cele ce urmează se va face o scurtă prezentare a celor mai importante apeluri sistem folosite pentru lucrul cu semnale.

Apelul sistem *signal*:

În Linux sunt disponibile mai multe apeluri sistem ce pot fi utilizate pentru generarea semnalelor. Apelul sistem *signal* poate fi întâlnit și la celelalte versiuni de Unix. Prototipul acestui apel sistem este următorul:

```
#include <signal.h>
void (*signal(int signum, void (*sig_handler)(int)))(int);
```

La prima vedere acest prototip pare complicat dar la o analiză mai atentă se poate observa că sunt necesari doar doi parametri:

- *signum* – un număr întreg care indică semnalul care va fi generat.
- *handler* – o referință către o funcție care va fi folosită ca rutină de tratare pentru semnalul respectiv.

Valoarea returnată de *signal()* este o referință către o funcție care primește ca și parametru un singur întreg și nu returnează nimic (*void*). Un exemplu de utilizare a acestui apel sistem

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

void handler(int sig){
    printf("CTRL-C\n");
    (void) signal(SIGINT,handler);
}

void main(void){
    (void) signal(SIGINT,handler);
    while(1){
        printf("Hello World!\n");
        sleep(1);
    }
}
```

Acest segment de cod afișează la terminal, în buclă infinită, mesajul *Hello World!*. La orice tentativă de a opri programul de la tastatură prin secvența *CTRL-C* semnalul *SIGINT* va fi prins de program, ca rutină de tratare a semnalului va fi folosită funcția *handler*.

Legătura între semnal (*SIGINT*) și rutina de tratare (*handler*) se realizează la apelul din funcția *main*. După realizarea acestei legături orice semnal *SIGINT* primit de proces va determina execuția rutinei de tratare (*handler*). Această funcție doar afișează mesajul *CTRL-C*, totodată reface legătura între semnal și funcție. Această refacere a legăturii între semnal și rutina de tratare este necesară datorită faptului că după primirea unui semnal se revine la tratarea implicită (terminarea execuției programului), ceea ce nu este de dorit.

Pentru acest tip de apel sistem apare o problemă legată de un scurt interval de timp în care tratarea semnalului se face în mod implicit, și dacă pe parcursul acestui interval de timp survine un semnal acesta va fi tratat în mod implicit, adică va determina încheierea execuției programului.

Apelul sistem *sigaction*

Acest apel sistem este conform standardelor POSIX, este mai complex, dar înlătură problema apărută anterior.

Conform acestor specificații fiecare proces are asociată câte o mască, care precizează setul de semnale care nu pot fi livrate procesului la momentul curent. Dacă totuși procesul primește un semnal ce aparține acestei liste el va fi pus într-o coadă de așteptare și va fi transmis procesului atunci când blocajul este înlăturat.

Când un semnal este generat, el este automat adăugat la masca de semnale a procesului ținută, astfel încât orice instanță viitoare a acestui semnal va fi blocată până la servirea semnalului curent.

Prototipul pentru acest apel sistem este următorul:

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
struct sigaction *oldact);
```

În locul referinței către o funcție în acest caz se transmite ca și parametru o referință către o structură numită *struct sigaction*. Aceasta are următoarea structură:

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

După cum se poate observa referința către rutina de tratare a fost introdusă în această structură. Totodată structura mai conține și un câmp *sa_mask* acesta funcționează ca și o mască de semnale suplimentară, și este valabilă pe parcursul execuției rutinei de tratare a semnalului. Câmpul *sa_flags* se interpretează la nivel de bit. Dintre aceste flag-uri cele mai semnificative sunt:

- *SA_ONESHOT* – semnalul va fi tratat în mod implicit.
- *SA_NOMASK* - ignoră câmpul *sa_mask* din structura *sigaction*.

Ambele flag-uri vor fi active dacă pentru se folosește apelul *signal* în loc de *sigaction*. Pentru obținerea de informații adiționale s-a introdus un parametru suplimentar *siginfo_t*.

Spre deosebire de *signal*, valoarea returnată de *sigaction* nu include o referință către vechea rutină de tratare. În schimb primește un nou parametru. Acesta este o altă referință către o structură de tipul *sigaction*, iar completarea acestei structuri o va face apelul *sigaction()* bazându-se pe vechea structură *sigaction*.

Apelul sistem *kill*.

În marea majoritate a cazurilor mai importante sunt semnalele generate de apariția unor evenimente: erori hardware, modificarea stării unui proces sau intervenția utilizatorului de la consolă. În afară de aceste situații există posibilitatea ca un proces să trimită semnale în mod deliberat unui alt proces, utilizând apelul sistem *kill*, dacă are permisiunile corespunzătoare. Prototipul acestui apel sistem este următorul:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Unde prin *sig* se precizează semnalul care va fi trimis, iar *pid* are următoarele sensuri:

- *pid* > 0 – semnalul *sig* va fi trimis procesului pentru care *PID = pid*.
- *pid* = 0 – semnalul *sig* va fi trimis tuturor proceselor cu același *PID* ca și procesul ce a făcut apelul.
- *pid* = -1 – semnalul *sig* va fi trimis tuturor proceselor din sistem cu excepția procesului *init* și a procesului ce a făcut apelul.
- *pid* < -1 – trimite semnalul *sig* tuturor proceselor pentru care *GID=pid*.

Apelul sistem *alarm*.

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Fiecare proces are asociat un ceas pe care îl poate folosi pentru a-și trimite semnale de tipul *SIGALRM* după depășirea unui anumit interval de timp. Apelul sistem *alarm()* primește un singur parametru care este intervalul de timp, în secunde, după care se va genera semnalul.

De studiat!

Să se realizeze un program care are următoarele caracteristici:

- va avea două procese: unul fiu și unul părinte.

- procesul părinte va afișa litera "A" pentru o secundă.
- procesul fiu va afișa litera "b" pentru o secundă.
- procesul părinte va aștepta terminarea execuției procesului fiu și va afișa - pid-ul său și al fiului precum și starea procesului fiu la terminarea execuției.

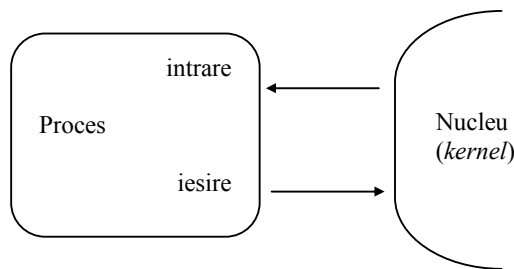
7.2. Pipes

În Linux, majoritatea interpretoarelor de comenzi acceptă redirectarea. Un exemplu clasic în acest sens este secvența de comenzi:

```
$cat /etc/passwd | wc -l
```

Rezultatul acestei secvențe de comenzi este că ieșirea standard a comenzii *cat* devine intrare standard pentru comanda *wc*. Deci un *pipe* este o metodă prin care se poate realiza conectarea ieșirii standard a unui proces cu intrarea standard altuia. *Pipe*-urile sunt una dintre cele mai vechi metode utilizate la comunicarea între procese. Comunicația realizată între două procese se poate realiza într-o singură direcție, de unde și denumirea de unidirecționale (*half-duplex*). Nici unul dintre procesele implicate nu sunt conștiente de aceste redirectări și este sarcina interpretorului de comenzi să gestioneze aceste *pipe*-uri.

Când un proces creează un *pipe*, nucleul (*kernel*) va genera doi descriptori de fișiere care urmeză să fie utilizați împreună cu *pipe*-ul. Unul dintre descriptori este folosit pentru a crea o cale de intrare (scriere) în *pipe*, iar celălalt pentru a crea o cale de ieșire. În acest punct utilizarea unui *pipe* nu are o utilitate practică prea mare atâta vreme cât procesul poate comunica numai cu el însuși. O reprezentare a procesului și a *kernelului* după ce a fost creat un *pipe*:



Din diagrama anterioară se poate observa modul de conectare a descriptorilor de fișiere. Un proces poate trimite date (scrie) prin *fd0*, și le poate obține prin *fd1*. La nivel de *pipe* transferul datelor se face în interiorul *kernelui*, iar *pipe*-ul este

reprezentat prin intermediul unui *inode*, care la rândul lui aparține tot *kernel*-ului și nu este legat de nici un sistem fizic de fișiere. Din ceea ce s-a prezentat până acum utilizarea *pipe*-urilor nu ar avea nici un sens deoarece în această situație un *pipe* ar fi folosit doar pentru comunicarea în cadrul unui și același proces. Dar situația se schimbă dacă după crearea *pipe*-ului are loc și crearea unui proces prin intermediul apelului sistem *fork*. În această situație procesul părinte va moșteni toți descriptorii de fișiere deschiși pentru procesul părinte, și bazându-ne pe pe acest mecanism avem bazele comunicării întreprocese (mai precis între procese aflate în relația părinte-fiu).

7.2.1 Apeluri sistem

Ceea ce s-a prezentat până acum a fost doar o prezentare de ansamblu, iar exemplul a fost la nivelul interpretorului de comenzi. Pentru a utiliza facilitățile oferite de *pipe*-uri în cadrul limbajului de programare C.

Apelul sistem pipe

Este utilizat pentru crearea unui fișier de tip *pipe*.

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

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

Returnează 0 în caz de succes și -1 în caz de eroare.

Argumentul *pfd* este un tablou cu două elemente care după execuția funcției va conține:

- Descriptorul de fișier pentru citire – *pfd[0]*;
- Descriptorul de fișier pentru scriere – *pfd[1]*;

Rolul acestui apel sistem este de a crea o cale de comunicație, de citire și scriere, între două procese. Această cale de comunicație se bazează pe cei doi descriptori de fișier care sunt returnați, de apelul sistem *pipe*, în tabloul *pfd*. Astfel din *pfd[0]* se citesc datele, iar în *pfd[1]* se scriu datele.

Așa cum s-a mai precizat comunicarea prin intermediul *pipe*-urilor este specifică comunicării între procese, deci în conjuncție cu apelul sistem *fork*. Un exemplu clasic este situația în care se dorește ca procesul părinte să transmită un mesaj procesului fiu. Programul C corespunzător va realiza următorii pași:

- Apelul sistem *pipe* – creează descriptorii de citire/scriere.
- Apelul sistem *fork* – se creează procesele părinte și fiu, cei doi descriptori se vor regăsi în ambele procese.

- Procesul părinte va închide *pfid[0]* descriptorul de citire.
- Procesul fiu va închide *pfid[1]* descriptorul de scriere.
- Se realizează transferul mesajului prin intermediul descriptorilor de fișier rămași deschiși.

Închiderea descriptorilor neutilizați este necesară din cauză că descriptorii de fișiere sunt aceeași atât în procesul părinte cât și în procesul fiu, situație datorată apelului sistem *fork*. Programul C corespunzător este prezentat în cele ce urmează.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void){
    int    pfd[2], nr_bytes;
    pid_t  pid_fiu;
    char   mesaj[20] = "Hello world!";
    char   readbuffer[80];

    pipe(pfd);

    if((pid_fiu = fork()) == -1)
    {
        perror("fork");
        exit(1);
    }

    if(pid_fiu == 0)
    {
        //procesul fiu închide descriptorul de scriere
        close(pfd[1]);

        //citește mesajul din pipe
        nr_bytes = read(pfd[0], readbuffer,
            sizeof(readbuffer));

        //și îl afișează
        printf("Received string: %s", readbuffer);
    }
    else
    {
        //procesul părinte închide descriptorul de
        //citire
        close(pfd[0]);

        //trimite mesajul prin descriptorul de scriere
```

```
        write(pfd[1], mesaj, (strlen(mesaj)+1));
        exit(0);
    }
    return(0);
}
```

Apelul sistem *dup* și *dup2*

Este utilizat pentru a duplica un descriptor de fișier.

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

```
int dup(int fd);
```

```
int dup2(int fd, int nfd);
```

Returnează noul descriptor în caz de succes, sau -1 în caz de eroare.

Atenție! Cu *dup2* vechiul descriptor este închis.

Acest apel sistem creează un nou descriptor de fișier pe lângă cel existent, descriptorul (numărul) returnat este cel mai mic disponibil. O utilizare imediată a acestui apel sistem ar fi posibilitatea redirectării intrării/ieșirii standard.

Apelul *dup* – deși vechiul și noul descriptor pot fi utilizate pe rând, de obicei una dintre căile de comunicare standard sunt închise.

Dacă vom considera:

```
.
.
pid_fiu = fork();

if(pid_fiu == 0)
{
    // închide intrarea standard a procesului fiu
    close(0);

    // duplică partea de intrare a pipe-ului cu
    // intrarea standard
    dup(pfd[0]);
    execlp("sort", "sort", NULL);
    .
    .
}
```

Deoarece descriptorul de fișier 0 (intrarea standard) a fost închis, apelul *dup* forțează descriptorul de intrare a *pipe*-ului ca intrare standard, apoi se realizează apelul sistem *execlp()* care suprascrie segmentul de cod a procesului fiu cu programul de sortare. Pentru că noul program executat moștenește căile de comunicare ale procesului din care a fost creat el moștenește de fapt ca și cale de

intrare standard descriptorul de citire din *pipe*. Astfel tot ce trimite procesul părinte inițial prin *pipe* ajunge la programul de sortare.

Apelul *dup2* – acest apel include, atât apelul de duplicare a căii de comunicare standard cât și operația de închidere a căii de comunicare standard, într-un singur apel sistem. În plus acest apel sistem este garantat a fi *atomic*, adică nu va fi acceptată nici o cerere de întrerupere (semnal) din partea sistemului pe parcursul execuției acestui apel. În cazul apelului sistem *dup()* trebuia realizată operația *close* înainte. Adică aveam două apeluri sistem, ceea ce creaa un anumit grad de vulnerabilitate, dat de intervalul dintre cele două apeluri. Dacă sosește un semnal între cele două apeluri operația de duplicare nu mai are loc.

În acest caz vom avea:

```
.
.
pid_fiu = fork();

if(pid_fiu == 0)
{
    //închide și duplică intarea standard
    dup2(0, pfd[0]);
    execlp("sort", "sort", NULL);
    .
}
.
```

De studiat

Să se realizeze un program cu următoarele caracteristici:

- Va avea trei procese: un proces părinte și două procese fiu.
- Procesul părinte va produce și afișa caractere “A” ce vor fi trimise printr-un pipe primului fiu, timp de 5 secunde.
- Primul fiul va număra caracterele primite de la procesul părinte și pentru fiecare 10 caractere de tip “A” va afișa un caracter “B”, după 5 secunde va trimite celui de al doilea fiu o statistică cu numărul de caractere afișate.
- Al doilea fiu va afișa statistica primită de la primul fiu.