Laboratorul 5

Comunicarea inter-proces
- Semnale -

1 Introducere

În mediile de dezvoltare UNIX există multiple mecanisme prin care se pot transmite informații între procese. Una dintre aceste metode este transmiterea de semnale. Semnalele marchează apariția unui eveniment, în urma căruia o acțiune specifică trebuie îndeplinită.

Există un set standard de semnale predefinite, printre care:

- `SIGINT` duce la terminarea procesului din prim-plan. Este transmis de către Kernel, în urma apăsării tastelor `Ctrl-C`;
- `SIGSTP`, este transmis de către Kernel în urma apăsării tastelor `Ctrl-Z`. Duce la întreruperea execuției procesului din prim-plan până la un primirea unui semnal ulterior de revenire (`SIGCONT`);
- `SIGSEGV` este transmis de către Kernel în urma accesării unei zone invalide de memorie. Rezultă în terminarea procesului;
- `SIGCHLD` este transmis de către un proces copil către procesul părinte, în urma schimbării stării (finalizarea, întreruperea sau continuarea execuției);
- `SIGUSR1`/`SIGUSR2` sunt semnale fără nicio însemnătate implicită, și pot fi definite de utilizator;

Fiecare semnal standard are corespunde unui număr întreg. Lista completă, precum și alte detalii pot fi regăsite în pagina de manual `signal(7)`.

2 Transmiterea Semnalelor

Semnalele pot fi trimise de către Kernel, sau de către un proces, fie către alt proces, folosind apelul de sistem `kill(2)`, sau către el însuși folosind funcția

`raise(3)`. O utilizare tipică a funcției `kill(2)` în C este:

```
int pid = fork();
if (pid == 0) {
    // ălogic copil
} else if (pid > 0) {
    // ălogic ăprinte
    ...
    if (kill(pid, sig)) < 0) {
        // eroare
    }
}
```

Întrebare: În ce fel putem apela `kill(2)`, astfel încât efectul să fie echivalent cu un apel către `raise(3)`?

3 Recepționarea Semnalelor

Fiecare semnal are o acțiune predefinită (*dispostion*), care dictează comportamentul unui proces în momentul în care primește semnalul respectiv. Procesele pot modifica acțiunea corespunzătoare unui semnal folosind apelurile de sistem `signal(2)`, sau `sigaction(2)` (din motive de portabilitate, se recomandă folosirea `sigaction(2)` în defavoarea `signal(2)`). Lista completa de acțiuni este:

- Term: Procesul își încheie execuția;
- Ign: Semnalul este ignorat și nu se ia nicio acțiune;
- Core: Procesul își încheie execuție și un fișier `core(5)` asociat este generat;
- Stop: Execuția procesului este întreruptă temporar;
- Cont: Execuția procesului (anterior întrerupt) este reluată;

În urma unui apel către `signal(2)` sau `sigaction(2)`, un proces desemnează ca una dintre următoarele acțiuni să fie declanșată când un anume semnal este recepționat:

- realizarea acțiunii predefinite a semnalului;
- ignorarea semnalului;
- înregistrarea unei funcții definite de utilizator, care se va executa în urma recepționării semnalului (*signal handler*)

Acțiunile sunt atribute per-proces: într-un proces cu mai multe fire de execuție, un semnal va declanșa aceeași acțiune în oricare dintre firele sale de execuție.

Un proces copil creat cu `fork(2)` va moșteni modificările făcute de procesul părinte asupra acțiunilor atașate semnalelor. În urma unui apel de tip `execve(2)`, acțiunile semnalelor sunt resetate la valorile implicite.

Comportamentul proceselor la recepționarea semnalelor `SIGKILL` și `SIGSTOP` va fi mereu cel implicit și nu poate fi modificat.

Atentie! Dacă multiple semnale de același tip sunt transmise concomitent către un proces, numai **o singură instanță** va fi eventual procesată.

Putem înregistra o nouă rutină în felul următor:

```
void handler(const int signum) {
    printf("semnal %d receptionat\n", signum);
}

int main(void) {
    ...
    [1]
    struct sigaction sa = { 0 };
    sa.sa_handler = handler,
    sigemptyset(&sa.sa_mask);

    [2]
    sigaction(SIGUSR1, &sa, NULL);
    ...
}
```

La `[1]` inițializăm o structură de tip `sigaction`, specificând rutina care să ruleze la primirea semnalului, setul standard de flag-uri și o mască goală pentru a nu bloca niciun semnal în timpul rutinei. În urma apelului de la `[2]`, semnalele de tip `SIGUSR1` recepționate de procesul nostru vor declanșa rutina `handler`. Consultați `sigaction(2)` pentru mai multe detalii.

Putem alege să ignorăm semnale de tip `SIGINT`, printr-o secvență în C precum:

```
    struct sigaction sa = { 0 };
    sa.sa_handler = SIG_IGN;
    sigaction(SIGINT, &sa, NULL);
```

4 Sarcini de laborator

1. Pe lângă apelul de sistem `kill(2)` menționat anterior, există și utilitarul pentru linia de comandă `kill(1)`. Consultați pagina de manual, sau meniul help (`kill -h`), pentru detalii de utilizare. Creați un proces care își afișează pid-ul și intră într-o buclă infinită. Experimentați cu transmiterea a diverse semnale către procesul creat anterior, folosind `kill(1)`.

2. Modificați programul anterior. **Păstrați** bucla infinită și înregistrați o nouă rutină de gestionare a semnalului `SIGINT` care afișează un mesaj,

dar nu întrerupe execuția procesului. Ce observați când apăsați `Ctrl-C`? Cum puteți opri execuția programului în acest caz? Găsiți mai multe soluții!

3. Creați un proces părinte și mai multe procese copil care realizează niște sarcini arbitrare pentru o perioada de timp (vezi `sleep(3)`), după care își finalizează execuția. Înregistrați o rutina de gestionare a semnalului `SIGCHLD`, care să elibereze resursele proceselor copil care și-au finalizat execuția, prin apeluri către `waitpid(2)`.

   - Care este diferența între `wait(2)` și `waitpid(2)`? Consultați manualul.
   - Ce se poate întâmpla în cazul în care mai multe procese copil își finalizează simultan execuția? Testați cu un număr mai mare de procese copil (e.g. 30).
   - Cum puteți rezolva problema? Citiți manualul `waitpid(2)` cu atenție!