

## Capitolul 6

# PROCESE

Linux este unul din sistemele de operare cu adevărat *multi-tasking*. *Procesele* reprezintă o parte fundamentală a acestui sistem de operare, practic ele controlează toate activitățile care se desfășoară în sistem.

UNIX, sistemul inițial din care a derivat Linux, definea un proces ca și “un spațiu de adrese cu unul sau mai multe fire de execuție care se execută în cadrul acelui spațiu de adrese și care solicită resurse sistem pentru acele fire de execuție.” Firele de execuție vor fi tratate în capitolele următoare. Deocamdată vom considera procesul ca un program care rulează.

Un sistem de operare *multi-tasking* este un sistem de operare care permite execuția simultană a mai multor procese. Fiecare instanță a unui program în execuție constituie un proces.

Ca și sistem de operare *multi-user*, Linux permite accesul simultan al mai multor utilizatori. Fiecare utilizator poate rula mai multe programe sau mai multe instanțe ale aceluiași program, toate la același moment de timp. Simultan sistemul de operare rulează alte programe de gestionare a resurselor sistemul și de control a accesului utilizatorilor la sistem.

Un program, sau un proces, care rulează este constituit din următoarele elemente: codul program, date, variabile, fișiere deschise și mediul de rulare. De obicei un sistem Linux va avea biblioteci și coduri sursă împărțite între mai multe procese, deci la un orice moment de timp pentru aceste resurse utilizate în comun va exista o singură copie în memoria sistem.

### 6.1. Identificatoare de proces - PID

Fiecare proces ce rulează sub Linux are asociat un identificator de proces unic numit PID (*Process Identity Numbers*) prin care este cunoscut în sistem. Fiecare proces din sistem începând cu procesul 1 *init* are un părinte a cărui PID îi este cunoscut și îl poate accesa.

Fiecare proces aparține unui grup de procese, fiecare grup având la rândul lui un număr de identificare care este de fapt identificatorul de proces (PID-ul) procesului părinte al grupului.

Când se pune problema drepturilor de acces a proceselor la diverse fișiere se folosește un set de patru numere de identificare. Aceste sunt cunoscute sub numele de utilizatorul real și identificatorul de grup, și utilizatorul efectiv și identificatorul lui de grup. Identificatorul real al unui proces este de fapt UID (*user ID*) și GID (*group ID*) ale utilizatorului care rulează procesul. Identificatorul efectiv este de obicei același cu cel al utilizatorului real cu excepția cazului în care sunt setate opțiunile *setuid* și *setgid*. Dacă una sau amândouă opțiunile sunt activate atunci UID sau GID efectiv al procesului vor primi valorile UID sau GID asociate fișierului din care rulează procesul.

Să presupunem că avem un utilizator cu UID 200 și GID 20 care dorește să ruleze programul */usr/bin/passwd*. Acest program are UID 0 (root) și GID 1 și are de asemenea set bitul *setuid*. Când este rulat programul procesul asociat va avea ID-ul utilizatorului real 200, ID-ul grupului real la 20, ID-ul utilizatorului efectiv 0, iar ID-ul grupului efectiv 20.

ID-ul real est folosit pentru a identifica utilizatorul pentru care este rulat un proces. ID-urile efective sunt folosite pentru a putea realiza o sortare a permisiunilor și privilegiilor pe care procesele le au la accesarea unui fișier. Acest lucru se realizează pe baza următorului algoritm:

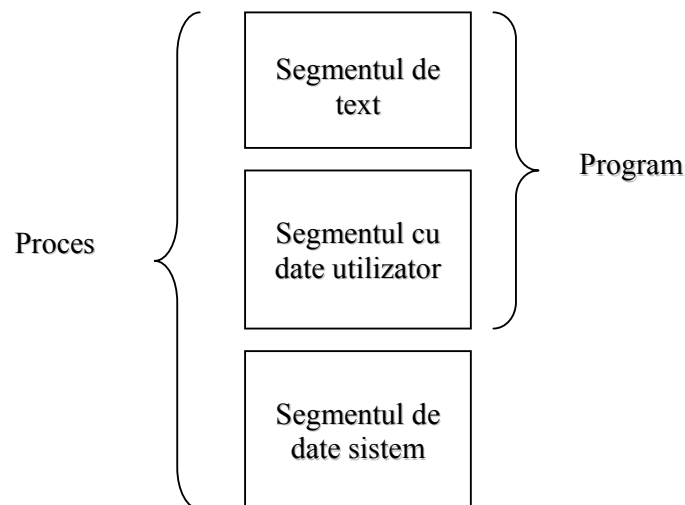
1. Dacă ID-ul utilizatorului efectiv al procesului este 0 (root) atunci procesului nu i se aplică nici o restricție, în caz contrar treci la pasul 2.
2. Dacă ID-ul utilizatorului efectiv al procesului este același cu ID-ul utilizator al fișierului atunci procesului i se acordă accesul la fișier în conformitate cu drepturile de acces ale proprietarului acelui fișier. Dacă nu treci la pasul 3.
3. Dacă ID-ul grupului efectiv al procesului este același cu ID-ul de grup al fișierului atunci procesului i se acordă accesul la fișier în conformitate cu drepturile de acces ale grupului fișierului. Dacă nu treci la pasul 4.
4. Accesul la fișier este acordat pe baza drepturilor de acces ale celorlalți utilizatori (*others*).

Toate aceste numere de identificare se pot obține cu ajutorul următoarelor funcții:

```
uid_t getuid(void) // obține ID-ul utilizatorului real
uid_t getgid(void) // obține ID-ul grupului real
uid_t geteuid(void) // obține ID-ul utilizatorului efectiv
uid_t getegid(void) // obține ID-ul grupului efectiv
uid_t getpid(void) // obține ID-ul procesului
uid_t getppid(void) // obține ID-ul procesului părinte
uid_t getpgrp(void) // obține ID-ul de grup al procesului
```

## 6.2. Crearea proceselor utilizând *fork*

Vom considera că un proces este construit din trei părți separate conform figurii următoare:



*Segmentul de text* conține instrucțiunile în cod mașină care urmează să fie executate. Acest segment poate fi accesat numai pentru citire (deci codul sursă din această parte nu poate fi modificat), ceea ce permite utilizarea acestui segment de către două sau mai multe procese din sistem care rulează același program. De exemplu dacă mai mulți utilizatori rulează aceeași versiune de interpretor de comenzi (de ex. *bash*), în memoria principală va fi o singură copie a programului pe care o folosesc în comun toți utilizatorii.

*Segmentul de date utilizator* conține toate datele cu care operează un proces pe parcursul execuției sale, aceste date includ și variabilele ce vor fi utilizate de proces. Datele din acest segment sunt modificabile și în plus fiecare proces are câte un segment de date unic.

*Segmentul de date sistem* conține elementele caracteristice mediului de lucru în care rulează un program. Acest segment realizează distincția între programe și procese. Programul are un caracter static, este localizat pe disc, și este constituit dintr-un set de instrucțiuni și date folosite pentru inițializarea segmentelor text și de

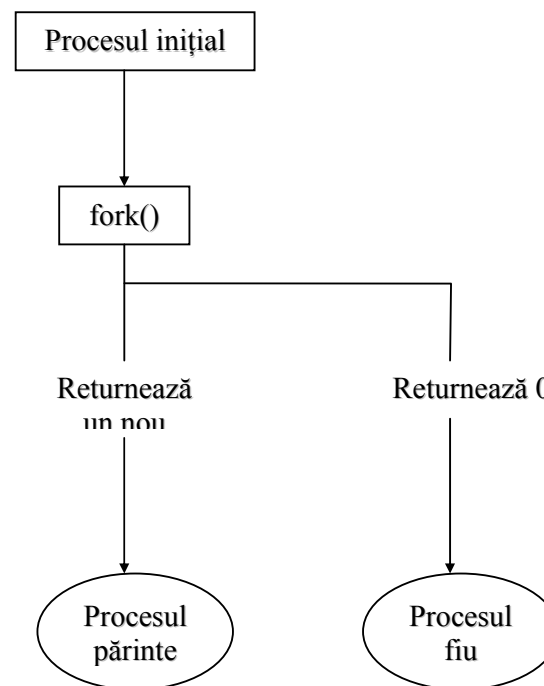
date ale proceselor. Procesul are un caracter dinamic iar execuția sa necesită interacțiunea între segmentele de date, text și sistem.

Sub Linux, pentru crearea de noi procese se folosește apelul sistem *fork*:

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

```
pid_t fork(void);
```

Practic acest apel realizează un nou proces, numit proces fiu, iar procesul care a apelat *fork* devine proces părinte a noului proces creat. Cele două procese sunt identice în termeni de conținut al segmentelor lor de date și text și aproape identice în ceea ce privește segmentele sistem. Singurele diferențe între astfel de procese apar la nivelul unor atribute care trebuie să fie diferite (cum ar fi PID care trebuie să fie unic pentru fiecare proces în parte). Odată ce procesul fiu a fost creat, ambele procese, părinte și fiu, își continuă execuția din interiorul apelului *fork*. Adică următoarea acțiune pentru fiecare proces în parte este să părăsească *fork*, fiecare returnând o valoare diferită.



Două procese virtual identice care rulează simultan se pot dovedi utile numai în situația în care acestea realizează acțiuni diferite. În această situație este nevoie de un mecanism prin care să se realizeze diferențierea între cele două procese. Acest lucru se realizează relativ simplu prin faptul că *fork* returnează valori diferite celor două procese. Astfel procesului părinte îi returnează PID-ul noului proces fiu creat, în timp ce procesului fiu îi returnează 0. În mod normal alocarea ID-urilor pentru procese începe cu 1 pentru procesul *init*. Vom avea în continuare un exemplu de program care crează un nou proces:

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

main(){
    pid_t val;
    printf("PID inainte de fork: %d\n", (int) getpid());
    if(val=fork())
        printf("PID parinte: %d\n", (int) getpid());
    else
        printf("PID fiu: %d\n", (int) getpid());
}
```

La rularea programului se vor genera trei linii de ieșire. Prima va afișa valoarea PID-ului procesului înainte de apelul *fork*, a doua valoarea PID a procesului părinte după apelul *fork*, iar a treia va afișa valoarea PID a procesului fiu. Un exemplu de ieșire obținut la rularea programului:

```
PID inainte de fork: 16953
PID parinte: 16953
PID fiu: 16954
```

După execuția apelului *fork* majoritatea atributelor procesului părinte sunt disponibile nemodificate procesului fiu. Principalele atribute sunt:

- apartenența la grupul de procese.
- terminalul în care rulează (dacă acesta există).
- UID și GID reale și efective.
- directorul de lucru curent.
- masca pentru crearea fișierelor (umask).

În plus toți descriptorii de fișiere care în procesul părinte sunt asociați cu fișiere deschise vor fi duplicați în procesul fiu. Adică atât descriptorii de fișiere ai procesului fiu cât și ai procesului părinte vor indica spre aceeași descriptori de fișiere deschise.

### 6.3. Apelul sistem *exec*

Dacă se utilizează interpretorul de comenzi pentru a rula o comandă (ex. *ls*) pentru rularea ei acesta va apela la un anumit moment *fork*. Se pune problema cum se poate ca în procesul fiu în loc de o copie a interpretorului de comenzi să fie rulat *ls*?

Soluția în acest caz este utilizarea apelului sistem *exec*. De fapt sunt mai multe versiuni ale acestui apel sistem dar toate realizează în esență același lucru. Acest apel sistem reprezintă de fapt modalitatea de rulare de programe sub Linux. Acest lucru îl realizează prin înlocuirea segmentelor de date și text ale procesului care apeleză *exec* cu cele ale programului a cărui nume a fost transmis ca și argument. O altă problemă ce ar putea apărea este faptul că majoritatea programelor ce sunt rulate în mod curent din linia de comandă primesc și parametrii sau argumente, deci apelul *exec* trebuie să acopere și aceste aspecte. Acest lucru se poate realiza în C prin argumentele *argc* și *argv*, deci și *exec* trebuie să i le transmită programului într-o formă asemănătoare.

Cea mai simplă versiune de *exec* este *execl*. Prototipul acestei funcții este:

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
```

unde *path* este calea completă către programul care urmează să fie executat, acesta este urmat de o listă variabilă de argumente ce trebuie transmise programului. Această listă trebuie terminată cu NULL(0). În continuare vom avea un exemplu de program care să ruleze comanda *ls -l*:

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

main(){
    execl("/bin/ls", "ls", "-l", 0);
    printf("terminat cu erori.");
}
```

Primul parametru pentru *execl* este calea completă pentru programul ce urmează a fi rulat, acesta este fișierul care va fi rulat, de aici se obțin și drepturile de acces la fișier(dacă are sau nu drept de execuție) restul parametrilor sunt folosiți pentru lista de argumente *argv*. În exemplu *ls* va fi *argv[0]*, iar *-l argv[1]*.

**Atenție!** Din apelurile *exec* nu se revine. Deci mesajul *terminat cu erori* din exemplu nu va fi afișat. Singura excepție este cazul în care a apărut o eroare la apelul *exec* în această situație se revine la vechiul proces și se returnează și codul de eroare (în această situație va fi afișat mesajul *terminat cu erori*).

În plus pentru a face toți acești parametri disponibili noului program apelurile *exec* transmit o valoare și pentru variabila:

```
extern char **environ;
```

Această variabilă are același format ca și variabila *argv* diferența fiind că prin variabila *environ* se transmit valori ale variabilelor aparținând mediului de lucru ale procesului original.

Acest apel sistem mai conține și alte veriuni singurele diferențe constând în modul de apelare, funcțional toate se comportă la fel:

```
int execlp(const char *file, const char *arg, ...);
```

```
int execl(const char *path, const char *arg, ..., char
```

```
* const envp[]);
```

```
int execv(const char *path, char *const argv[]);
```

```
int execvp(const char *file, char *const argv[]);
```

```
int execve(const char *filename, char *const argv [],
```

```
char *const envp[]);
```

În cazul ultimelor trei apeluri argumentele ce trebuie transmise programelor sunt organizate sub forma unei liste.

Ca și în cazul apelului *fork* majoritatea atributelor sunt păstrate în urma unui apel *exec*. Acest lucru se datorează faptului că segmentul sistem rămâne intact în urma unui apel *exec*. Cel mai important aspect este acela că descriptorii de fișiere asociați cu cu descriptorii de fișiere deschise rămân valabile și după apelul lui *exec*. Singura excepție este un marcaj (*flag*) asociat descriptorilor de fișiere (nu este asociat descriptorilor de fișiere deschise) numit *close on exec*. Dacă acest *flag* nu este activat atunci descriptorul de fișier respectiv este valabil și după apelul *exec*. Modificarea acestui *flag* se poate face prin intermediul apelului sistem *fcntl*.

## 6.4. Apelurile sistem *wait* and *exit*

Se poate pune problema ce face procesul părinte în timp ce procesul fiu rulează. Pe parcursul rulării procesului fiu procesul părinte fie așteaptă ca acesta să își termine execuția, fie își continuă execuția.

În cazul interpretorului de comenzi de exemplu, alegerea rămâne la latitudinea utilizatorului. În mod normal interpretorul de comenzi așteaptă, pentru fiecare comandă introdusă din linia de comenzi, până la terminarea execuției acesteia. Dar utilizatorul are posibilitatea indica interpretorului de comenzi să nu mai aștepte terminarea comenzii și să își reia execuția imediat prin adăugarea caracterului ‘&’ la sfârșitul comenzii.

Pentru ca procesul părinte să aștepte terminarea execuției unui proces fiu, procesul părinte trebuie să execute apelul sistem *wait()*. Prototipul acestui apel sistem este următorul:

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

```
#include <sys/wait.h>
```

```
pid_t wait(int *status)
```

Apelul sistem *wait* returnează PID-ul procesului fiu ce și-a încheiat execuția. Parametrul care îl primește apelul sistem *wait* este o referință către o locație care va primi valoarea stării de terminare a procesului fiu.

Când un proces și-a terminat execuția el va executa apelul sistem *call* fie în mod explicit fie indirect prin intermediul unei biblioteci. Prototipul apelului sistem *exit* este următorul:

```
#include <stdlib.h>
```

```
void exit(int status)
```

Acest apel sistem primește ca și argument un parametru de stare care va fi transmis procesului părinte prin intermediul unei locații referită de parametrul apelului *wait*.

Apelul sistem *wait* poate mai multe informații referitoare la starea procesului fiu la terminare prin intermediul valorii de stare. Pentru extragerea acestor informații se poate folosi un macro numit *WEXITSTATUS*. Ex:

```
#include <sys/wait.h>
```

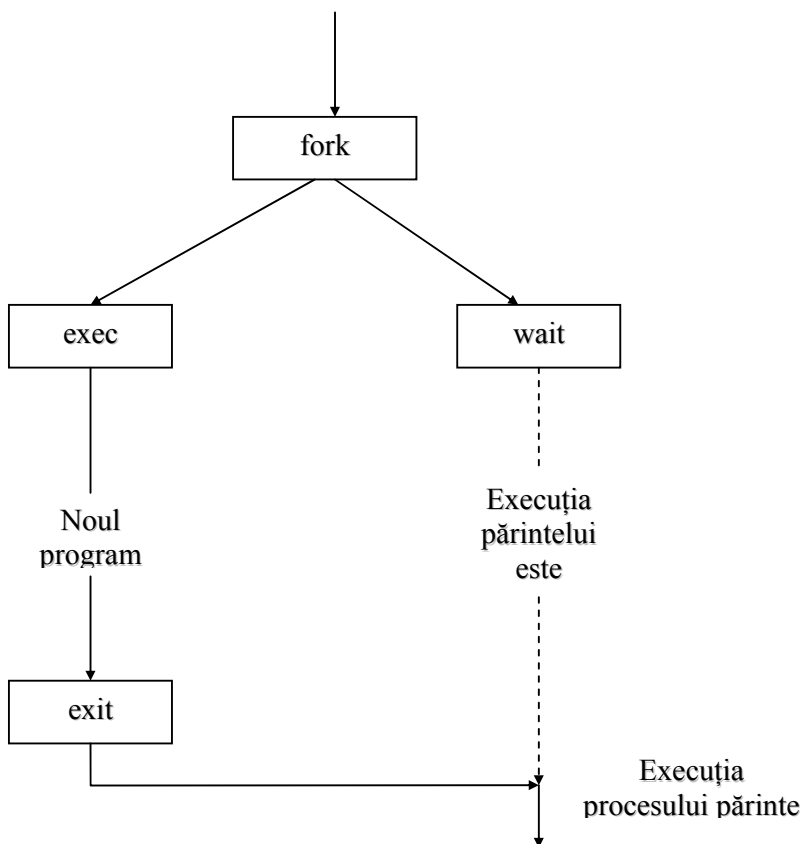
```
int statval, exstat;
```

```
pid_t pid;
```

```
pid = wait(&statval);
```

```
exstat = WEXITSTATUS(statval);
```

Schema bloc care descrie funcționarea a două procese, unul părinte care așteaptă terminarea execuției procesului fiu și a procesului fiu care folosește apelul sistem *exec* și a cărui execuție se termină cu apelul *exit*, arată astfel:



Pentru parametrul *pid* avem următoarele posibilități:

- < -1 așteaptă încheierea execuției oricărui proces fiu a cărui ID de grup este egal cu *pid*.
- -1 comportament identic cu *pid*.
- 0 așteaptă încheierea execuției oricărui proces fiu a cărui ID de grup este egal cu ID-ul de grup al procesului.
- > 0 așteaptă încheierea execuției procesului fiu a cărui PID este egal cu *pid*.

Există posibilitatea ca un proces fiu să își încheie execuția înainte ca procesul părinte să apeleze *wait*. În această situație procesul fiu va intra într-o stare numită *zombie*, în care toate resursele ocupate de procesul fiu au fost eliberate cu excepția structurii de date care conține starea sa de ieșire. Când procesul părinte efectuează apelul *wait*, starea de ieșire este transmisă procesului părinte iar resursele ocupate cu structura de date a procesului fiu pot fi eliberate.

**De studiat!** Să se realizeze un program în C care să creeze trei procese (inclusiv părintele). Fiecare proces își afișează PID-ul, iar procesul părinte numără de la 1-5000, primul proces fiu de la 5000-10000, iar al doilea proces fiu afișează litere din alfabet.

O altă versiune a apelului sistem *wait* este *waitpid* cu următorul prototip:

```

#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
  
```

Prin parametrul *options* oferă facilități suplimentare. De exemplu în urma unui apel *waitpid* procesul părinte nu își suspendă execuția dacă nici un proces fiu nu are disponibilă starea de ieșire.