

Laboratorul 4

Procese

1 Crearea unui proces nou

În mediile de dezvoltare UNIX, funcția sistem cu care se creează procese noi este `fork(2)`. Odată invocată, funcția creează un proces nou (numit proces copil), acesta fiind o copie a procesului apelant, dar cu câteva diferențe, dintre care le enumerăm aici pe cele mai importante (ele pot diferi de la sistem de operare la sistem de operare):

- procesul copil are ca părinte chiar pe procesul ce a apelat `fork(2)` (în vreme ce procesul apelant va avea un alt părinte);
- procesul copil are un ID (denumit și `pid`) diferit de cel al părintelui;
- procesul copil are un singur fir de execuție (`thread`, v. Laboratorul 6);
- procesul copil pornește de la zero în ceea ce privește resursele utilizate și timpul de execuție, precum și alți indicatori similari de gestionare a proceselor.

Din momentul apelului, dacă acesta este încheiat cu succes, fiecare viitoare instrucțiune va fi executată atât de părinte, cât și de copil. Diferențierea se face în funcție de valoarea de retur a lui `fork(2)`: copilul primește valoarea 0 iar părintele `pid`-ul copilului. Astfel, o construcție tipică C este

```
pid_t pid = fork();
if (pid < 0)
    return errno;
else if (pid == 0)
    /* child instructions */
else
    /* parent instructions */
```

Oricând în timpul execuției putem afla `pid`-ul procesului curent și pe cel al procesului părinte cu ajutorul funcțiilor `getpid(2)` și, respectiv, `getppid(2)`.

```
printf("Parent %d Me %d\n", getppid(), getpid());
```

Părintele își poate suspenda activitatea pentru a aștepta finalizarea execuției unui proces copil cu ajutorul funcției `wait(2)`, care oferă ca valoare de retur `pid`-ul copilului. **Atenție**, această funcție redă control părintelui când se termină execuția **oricăruia** dintre copiii săi. Pentru cazuri complexe, în care se dorește așteptarea unuia sau mai multor procese anume, se pot folosi funcții avansate, precum `waitpid(2)` sau `wait4(2)`, dar care nu fac obiectul laboratorului.

Operația este utilă pentru a sincroniza și ordona instrucțiunile.

```
pid_t pid = fork();
if (pid < 0)
    return errno;
else if (pid == 0)
    /* child instructions */
    printf("First!");
else {
    /* parent instructions */
    wait(NULL);
    printf("Last!");
}
```

2 Executarea unui program existent

Executarea unui program se realizează cu ajutorul apelului sistem `execve(2)`.

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

Aceasta **suprascrisie** complet procesul apelant cu un nou proces, conform programului găsit la calea indicată în `path`. **Atenție**, calea trebuie să fie absolută, de pildă `/bin/pwd` și nu `pwd`. Pentru a obține această cale, puteți folosi comanda `which(1)`:

```
$ which pwd
/bin/pwd
$ which vi
/usr/bin/vi
```

Argumentele programului sunt puse în `argv` respectând convenția obișnuită din C: pe prima poziție (`argv[0]`) se află calea absolută către program urmată de argumente. Lista se încheie cu `null`. Variabilele de sistem din mediul de execuție sunt puse în ultimul argument `envp`. Aceasta este o listă de șiruri de caractere similară cu `argv` exceptând convenția primului element.

Din cauza efectului distructiv asupra procesului curent, `execve(2)` este adesea folosit împreună cu `fork(2)` astfel încât procesul nou-creat să fie cel suprascris.

```

pid_t pid = fork();
if (pid < 0)
    return errno;
else if (pid == 0)
    /* child instructions */
    char *argv[] = {"pwd", NULL};
    execve("/bin/pwd", argv, NULL);
    perror(NULL);
else
    /* parent instructions */

```

Având în vedere suprascrierea procesului curent, `execve(2)` nu mai revine în programul inițial decât în cazul în care a apărut o eroare folosindu-se `errno` pentru a determina cauza. Cele mai des întâlnite erori sunt calea greșită sau lipsa lui `argv`.

3 Sarcini de laborator

1. Creați un proces nou folosind `fork(2)` și afișați fișierele din directorul curent cu ajutorul `execve(2)`. Din procesul inițial afișați `pid`-ul propriu și `pid`-ul copilului. De exemplu:

```

$ ./forkls
My PID=41875, Child PID=62668
Makefile      collatz.c      forkls.c      so-lab-4.tex
collatz       forkls         ncollatz.c
Child 62668 finished

```

2. Ipoteza Collatz spune că plecând de la orice număr natural dacă aplicăm repetat următoarea operație

$$n \mapsto \begin{cases} n/2 & \text{mod } (n, 2) = 0 \\ 3n + 1 & \text{mod } (n, 2) \neq 0 \end{cases}$$

șirul ce rezultă va atinge valoarea 1. Implementați un program care folosește `fork(2)` și testează ipoteza generând șirul asociat unui număr dat în procesul copil. Exemplu:

```

$ ./collatz 24
24: 24 12 6 3 10 5 16 8 4 2 1.
Child 52923 finished

```

3. Implementați un program care să testeze ipoteza Collatz pentru mai multe numere date. Pornind de la un singur proces părinte, este creat câte un copil care se ocupă de un singur număr. Părintele va aștepta să termine

execuția fiecare copil. Programul va demonstra acest comportament folosind funcțiile `getpid(2)` și `getppid(2)`. Exemplu:

```
$ ./ncollatz 9 16 25 36
Starting parent 6202
9: 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4
2 1.
36: 36 18 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5
16 8 4 2 1.
Done Parent 6202 Me 40018
Done Parent 6202 Me 30735
16: 16 8 4 2 1.
25: 25 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40
20 10 5 16 8 4 2 1.
Done Parent 6202 Me 13388
Done Parent 6202 Me 98514
Done Parent 58543 Me 6202
```