

PROGRAMARE DE SISTEM ÎN C PENTRU PLATFORMA LINUX (V)

Gestiunea proceselor, partea a II-a:

Reacoperirea proceselor – primitivele `exec()`

Cristian Vidrașcu
vidrascu@info.uaic.ro

Aprilie, 2021



Sumar

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu `exec`](#)

[Referințe bibliografice](#)

Introducere

Reacoperirea proceselor

Primitivele din familia `exec`

Caracteristicile procesului după `exec`

Demo: programe cu `exec`

Exemplul #1: Reacoperirea unui program cu alt program

Exemplul #2: Reacoperirea recursivă

Exemplul #3: Reacoperirea unui program cu fișiere deschise

Exemplul #4: Redirectarea fluxului `stdout`

Exemplul #5: Reacoperirea unui program cu un script

Alte programe demonstrative

Referințe bibliografice



Introducere

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu `exec`](#)

[Referințe bibliografice](#)

După cum am văzut în lecția anterioară, singura modalitate de a crea un nou proces în UNIX/Linux este prin apelul funcției `fork`.

Numai că în acest fel se creează o copie a procesului apelant, adică o nouă instanță de execuție a aceluiași program (*i.e.*, fișier executabil).

Și atunci, cum este posibil să executăm un alt fișier executabil decât cel care apelează primitiva `fork`?

Răspuns: prin utilizarea unui alt mecanism, acela de “**reacoperire a proceselor**”, disponibil în sistemele de operare UNIX/Linux prin intermediul primitivelor din familia `exec`.



Agenda

Introducere

Reacoperirea proceselor

Primitivele din familia `exec`
Caracteristicile procesului
după `exec`

Demo: programe cu `exec`

Referințe bibliografice

Introducere

Reacoperirea proceselor

Primitivele din familia `exec`

Caracteristicile procesului după `exec`

Demo: programe cu `exec`

Exemplul #1: Reacoperirea unui program cu alt program

Exemplul #2: Reacoperirea recursivă

Exemplul #3: Reacoperirea unui program cu fișiere deschise

Exemplul #4: Redirectarea fluxului `stdout`

Exemplul #5: Reacoperirea unui program cu un script

Alte programe demonstrative

Referințe bibliografice



Primitivele din familia `exec`

[Introducere](#)

[Reacoperirea proceselor](#)

[Primitivele din familia `exec`](#)

[Caracteristicile procesului după `exec`](#)

[Demo: programe cu `exec`](#)

[Referințe bibliografice](#)

Familia de primitive `exec` “înlocuiește” programul rulat în cadrul procesului apelant cu un alt program, specificat prin numele fișierului executabil asociat, transmis ca argument al apelului `exec`.

Spunem că noul program “*reacoperă*” *vechiul program* în procesul ce execută apelul `exec`. Simplificat spus, noul program “*reacoperă*” *procesul apelant* al funcției `exec`.

În plus, procesul “transformat” prin înlocuirea cu noul program “moștenește” caracteristicile avute de la vechiul program (cu excepția câtorva dintre acestea), inclusiv PID-ul (deoarece, d.p.d.v. al SO-ului, el este același proces).

Există 6+1 funcții în **familia de apeluri `exec`** ([5]). Aceste funcții diferă între ele prin nume și prin lista parametrilor de apel, putând fi împărțite în două categorii ce se diferențiază prin forma în care se dau parametrii de apel:

- numărul de parametri este variabil (*i.e.*, linia de comandă este dată prin enumerare)
- numărul de parametri este fix (*i.e.*, linia de comandă este specificată printr-un vector)



Primitivele din familia `exec` (cont.)

[Introducere](#)

[Reacoperirea proceselor](#)

[Primitivele din familia `exec`](#)

[Caracteristicile procesului după `exec`](#)

[Demo: programe cu `exec`](#)

[Referințe bibliografice](#)

1) Prima pereche de primitive `exec` este reprezentată de apelurile `execl` și `execv`, ce au interfețele următoare:

- `int execl(char* ref, char* argv0, ..., char* argvN)`
- `int execv(char* ref, char* argv[])`
 - *ref* = argument obligatoriu, fiind numele programului ce va reacoperi procesul apelant al respectivei primitive `exec`
 - $N \geq 1$, adică celelalte argumente (cu excepția *argv0* și *argvN*) pot lipsi, ele exprimând parametrii efectivi ai liniei de comandă pentru programul *ref*

Observații:

1. Argumentul *ref* trebuie să fie un nume de fișier executabil care să se afle în directorul curent (sau să se specifice și directorul în care se află, prin cale absolută sau relativă), deoarece nu este căutat în directoarele din variabila de mediu `PATH`. De asemenea, *ref* mai poate fi și numele unui script, care începe cu o linie de forma `#!/interpreter`.
2. Argumentul *argv0*, respectiv *argv[0]*, specifică *numele procesului “transformat”* (i.e., numele procesului după reacoperirea cu noul program), afișat de comenzi precum `ps`, `pstree`, `w`, ș.a.
3. Ultimul argument *argvN*, respectiv ultimul element din tabloul *argv*[], trebuie să fie pointerul `NULL`.



Primitivele din familia `exec` (cont.)

[Introducere](#)

[Reacoperirea proceselor](#)

[Primitivele din familia `exec`](#)

[Caracteristicile procesului după `exec`](#)

[Demo: programe cu `exec`](#)

[Referințe bibliografice](#)

2) A doua pereche de primitive `exec` este reprezentată de apelurile `execle` și `execve`, ce au interfețele următoare:

- `int execle(char* ref, char* argv0, ..., char* argvN, char* env[])`
- `int execve(char* ref, char* argv[], char* env[])`
 - `env` = parametru ce permite transmiterea unui *environment* (i.e., un set de variabile de mediu) către noul program ce va reacoperi procesul apelant
 - celelalte argumente sunt la fel ca la prima pereche

Observații:

1. Și în acest caz au loc restricțiile din observațiile 1., 2. și 3. specificate la prima pereche.
2. La fel ca pentru `argv[]`, ultimul element din tabloul `env[]` trebuie să fie pointerul `NULL`.

* * *

Notă: funcția `execve` este apelul de sistem pentru reacoperire (a se consulta `man 2 execve`), iar celelalte funcții sunt de fapt niște *wrappere* definite în `STANDARD C LIBRARY`, ce apelează la rândul lor funcția `execve` (a se consulta `man 3 exec`).



Primitivele din familia `exec` (cont.)

[Introducere](#)

[Reacoperirea proceselor](#)

[Primitivele din familia `exec`](#)

[Caracteristicile procesului după `exec`](#)

[Demo: programe cu `exec`](#)

[Referințe bibliografice](#)

3) A treia pereche de primitive `exec` este reprezentată de apelurile `exec1p` și `execvp`, ce au interfețele următoare:

- `int exec1p(char* ref, char* argv0, ..., char* argvN)`
- `int execvp(char* ref, char* argv[])`
 - argumentele sunt la fel ca la prima pereche

Observații:

1. Argumentul *ref* indică un nume de fișier executabil care, dacă nu este specificat împreună cu calea absolută sau relativă până la acel fișier (*i.e.*, *ref* nu conține caracterul `'/'`), atunci el va fi căutat în directoarele din variabila de mediu `PATH`. De asemenea, *ref* mai poate fi și numele unui script, care începe cu o linie de forma `#!interpreter`.
2. Și în acest caz au loc restricțiile din observațiile 1., 2. și 3. specificate la prima pereche.

* * *

Notă: a 7-a funcție, numită `execvpe`, este o extensie GNU și are interfața următoare:

- `int execvpe(char* ref, char* argv[], char* env[])`
 - argumentele sunt la fel ca la apelul `execvp` și mai avem în plus și un *environment*



Primitivele din familia `exec` (cont.)

[Introducere](#)

[Reacoperirea proceselor](#)

[Primitivele din familia `exec`](#)

[Caracteristicile procesului după `exec`](#)

[Demo: programe cu `exec`](#)

[Referințe bibliografice](#)

Valoarea returnată: în caz de eșec (e.g., datorită memoriei insuficiente, sau altor cauze posibile), toate primitivele `exec` returnează valoarea `-1`.

Altfel, în caz de succes, **apelurile `exec` nu returnează** (!), deoarece programul apelant nu mai există (fiind “reacoperit” de noul program).

Notă: familia de primitive `exec` este singurul exemplu de funcții (cu excepția primitivelor `exit` și `abort`) al căror apeluri nu returnează înapoi în programul apelant.

Observație: prin convenție `argv0`, respectiv `argv[0]`, trebuie să coincidă cu *ref* (deci cu numele fișierului executabil). Aceasta este însă doar o *convenție*, nu se produce eroare în caz că este încălcată.

De reținut: argumentul *ref* specifică *numele real* al fișierului executabil (sau scriptului) ce se va încărca și executa, iar `argv0`, respectiv `argv[0]`, specifică *numele procesului “transformat”* (i.e., numele procesului după reacoperirea programului apelant cu noul program), afișat de comenzi precum `ps`, `pstree`, `w`, ș.a.



Caracteristicile procesului după exec

Prin “reacoperirea” unui proces, “noul program” moștenește caracteristicile “vechiului program” (*i.e.*, are același PID, aceeași prioritate, același proces părinte, aceeași descriptori de fișiere deschise, ș.a.), cu unele excepții, în condițiile precizate în tabel:

Caracteristica	Condiția în care nu se conservă
Proprietarul efectiv	Dacă este setat bitul <i>setuid</i> al fișierului încărcat, proprietarul acestui fișier devine proprietarul efectiv al procesului.
Grupul proprietar efectiv	Dacă este setat bitul <i>setgid</i> al fișierului încărcat, grupul proprietar al acestui fișier devine grupul proprietar efectiv al procesului.
<i>Handler</i> -ele de semnale	Sunt reinstalate <i>handler</i> -ele implicite pentru acele semnale ce erau “corupte” (<i>i.e.</i> , interceptate).
Descriptorii de fișiere	Dacă bitul FD_CLOEXEC de închidere automată în caz de exec, al vreunui descriptor de fișier, a fost setat cu ajutorul primitivei <code>fcntl</code> , atunci descriptorul respectiv este închis la exec (ceilalți descriptori de fișiere rămân deschiși).

- Introducere
- Reacoperirea proceselor
- Primitivele din familia exec
- Caracteristicile procesului după exec
- Demo: programe cu exec
- Referințe bibliografice



Agenda

Introducere

Reacoperirea proceselor

Demo: programe cu `exec`

Exemplul #1: Reacoperirea unui program cu alt program

Exemplul #2: Reacoperirea recursivă

Exemplul #3: Reacoperirea unui program cu fișiere deschise

Exemplul #4: Redirectarea fluxului `stdout`

Exemplul #5: Reacoperirea unui program cu un script

Alte programe demonstrative

Referințe bibliografice

Introducere

Reacoperirea proceselor

Primitivele din familia `exec`

Caracteristicile procesului după `exec`

Demo: programe cu `exec`

Exemplul #1: Reacoperirea unui program cu alt program

Exemplul #2: Reacoperirea recursivă

Exemplul #3: Reacoperirea unui program cu fișiere deschise

Exemplul #4: Redirectarea fluxului `stdout`

Exemplul #5: Reacoperirea unui program cu un script

Alte programe demonstrative

Referințe bibliografice



Exemplul #1: Reacoperirea unui program cu alt program

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu `exec`](#)

[Exemplul #1: Reacoperirea
unui program cu alt program](#)

[Exemplul #2: Reacoperirea
recursivă](#)

[Exemplul #3: Reacoperirea
unui program cu fișiere
deschise](#)

[Exemplul #4: Redirectarea
fluxului `stdout`](#)

[Exemplul #5: Reacoperirea
unui program cu un script](#)

[Alte programe demonstrative](#)

[Referințe bibliografice](#)

Un exemplu ce ilustrează folosirea unui apel din familia `exec`, precum și conservarea câtorva dintre proprietățile procesului după execuția apelului `exec`, ar fi următorul:

A se vedea programul `before_exec.c`, ce apelează `exec1` pentru a se “reacoperi” cu un al doilea program, `after_exec.c` ([2]).

Observație: executând programul `before_exec` veți putea constata faptul că variabila `nrBytesRead` va avea valoarea `-1` în mesajul afișat de programul `after_exec` (motivul fiind că intrarea standard `stdin` este moștenită ca fiind închisă în procesul “reacoperit” cu `after_exec`).

Aceasta constituie o dovadă a faptului că “noul program” de după reacoperire, `after_exec`, moștenește descriptorii de fișiere deschise de la “vechiul program” de dinainte de reacoperire, `before_exec`.



Exemplul #2: Reacoperirea recursivă

Un al doilea exemplu: un program care se “reacoperă” cu el însuși, dar la al doilea apel își modifică parametrii de apel pentru a-și putea da seama că este la al doilea apel și astfel să nu intre într-o “bucă infinită” de apeluri recursive.

A se vedea programul `exec_rec.c` ([2]).

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu `exec`](#)

Exemplul #1: Reacoperirea
unui program cu alt program

**Exemplul #2: Reacoperirea
recursivă**

Exemplul #3: Reacoperirea
unui program cu fișiere
deschise

Exemplul #4: Redirectarea
fluxului `stdout`

Exemplul #5: Reacoperirea
unui program cu un script

Alte programe demonstrative

[Referințe bibliografice](#)



Exemplul #3: Reacoperirea unui program cu fișiere deschise

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu `exec`](#)

[Exemplul #1: Reacoperirea](#)

[unui program cu alt program](#)

[Exemplul #2: Reacoperirea](#)

[recursivă](#)

[Exemplul #3: Reacoperirea](#)

[unui program cu fișiere](#)

[deschise](#)

[Exemplul #4: Redirectarea](#)

[fluxului `stdout`](#)

[Exemplul #5: Reacoperirea](#)

[unui program cu un script](#)

[Alte programe demonstrative](#)

[Referințe bibliografice](#)

A se vedea programul `com-0.c`, care se “reacoperă” cu programul `com-2.c` ([2]).

Observație: programul `com-0.c` redirectează fluxul `stdout` în fișierul `fis.txt`, folosind primitiva `dup` ([5]), și ca atare programul `com-2.c` moștenește această redirectare. Astfel, veți observa că mesajele scrise vor apare în acel fișier și nu pe ecran.



Exemplul #3: Reacoperirea unui program cu fișiere deschise (cont.)

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu `exec`](#)

[Exemplul #1: Reacoperirea unui program cu alt program](#)

[Exemplul #2: Reacoperirea recursivă](#)

[Exemplul #3: Reacoperirea unui program cu fișiere deschise](#)

[Exemplul #4: Redirectarea fluxului `stdout`](#)

[Exemplul #5: Reacoperirea unui program cu un script](#)

[Alte programe demonstrative](#)

[Referințe bibliografice](#)

Comportamentul în cazul fișierelor deschise în momentul apelului primitivelor `exec`: dacă s-au folosit instrucțiuni de scriere *buffer*-izate (ca de exemplu funcțiile `fprintf`, `fwrite` ș.a. din biblioteca standard I/O de C), atunci *buffer*-ele nu sunt scrise automat în fișier pe disc în momentul apelului `exec`, deci informația din ele se pierde (!).

Notă: în mod normal *buffer*-ul este scris în fișier abia în momentul când s-a umplut, sau la întâlnirea caracterului `'\n'`. Dar se poate forța scrierea *buffer*-ului în fișier cu ajutorul funcției `fflush` din biblioteca standard I/O de C.

A se vedea programul `com-1.c`, care se “reacoperă” cu programul `com-2.c` ([2]).

Observație: dacă eliminăm apelul `fflush` din programul `com-1.c`, atunci pe ecran se va afișa doar mesajul incomplet “... , **tuturor!**”. Acest lucru se întâmplă deoarece mesajul de început “**Salut**...” se pierde prin `exec`, conținutul *buffer*-ului nefiind scris pe disc.



Exemplul #4: Redirectarea fluxului `stdout`

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu `exec`](#)

Exemplul #1: Reacoperirea
unui program cu alt program

Exemplul #2: Reacoperirea
recursivă

Exemplul #3: Reacoperirea
unui program cu fișiere
deschise

**Exemplul #4: Redirectarea
fluxului `stdout`**

Exemplul #5: Reacoperirea
unui program cu un script

[Alte programe demonstrative](#)

[Referințe bibliografice](#)

Pe lângă primitiva `dup`, mai există și o altă primitivă, cu numele `dup2`, ambele fiind utilizate pentru duplicarea unui descriptor de fișier ([5]).

Cu ajutorul acestor apeluri se poate realiza redirectarea fluxurilor standard de I/O, precum am văzut în exemplul precedent (*i.e.*, programul `com-0.c`).

* * *

Un alt exemplu de redirectare a fluxului `stdout`: programul `redirect.c` ([2]).

În acest caz redirectarea se face către fișierul `fis.txt`, iar apoi este anulată (prin redirectarea înapoi către terminalul I/O fizic asociat sesiunii de lucru curente, referit prin numele `/dev/tty`).



Exemplul #5: Reacoperirea unui program cu un script

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu `exec`](#)

Exemplul #1: Reacoperirea
unui program cu alt program

Exemplul #2: Reacoperirea
recursivă

Exemplul #3: Reacoperirea
unui program cu fișiere
deschise

Exemplul #4: Redirectarea
fluxului `stdout`

**Exemplul #5: Reacoperirea
unui program cu un script**

[Alte programe demonstrative](#)

[Referințe bibliografice](#)

Un exemplu ce ilustrează folosirea unui apel din familia `exec` pentru a “reacoperi” programul apelant cu un script, ar fi următorul:

A se vedea programul `exec_script.c`, ce apelează `exec1` pentru a se “reacoperi” cu un script bash, `my_script.sh` ([2]).

Notă: mai exact, aici, efectul apelului `exec` este acela de a “reacoperi” programul apelant cu o instanță a interpretorului specificat pe prima linie din script, iar această instanță va interpreta scriptul linie cu linie (și-l va executa, astfel, în manieră interpretată).



Alte programe demonstrative

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu exec](#)

Exemplul #1: Reacoperirea unui program cu alt program

Exemplul #2: Reacoperirea recursivă

Exemplul #3: Reacoperirea unui program cu fișiere deschise

Exemplul #4: Redirecarea fluxului stdout

Exemplul #5: Reacoperirea unui program cu un script

[Alte programe demonstrative](#)

[Referințe bibliografice](#)

Iată un exemplu de program care *apelează prin exec o comandă* oarecare, însoțită de o listă de argumente, iar apoi prelucrează statusul execuției comenzii respective (*i.e.*, succes vs. eșec):

```
#include ...
int main (int argc, char *argv []) {
    pid_t pid;    int ret;    char* dirname = (argc < 2) ? "." : argv[1];
    /* Creez un proces fiu, care va rula comanda ls prin exec. */
    if(-1 == (pid=fork()) ) {    perror("Eroare la fork");    exit(1);    }
    /* In procesul fiu apelez exec pentru a executa comanda dorita. */
    if (pid == 0) {
        execl("/bin/ls", "ls", "-l", "-i", dirname, NULL);
        perror("Eroare la exec");
        exit(10); // Returnez un numar mare, nu 1,2,... care ar putea fi returnate si de ls!
    }
    /* (Doar in procesul parinte) Acum cercetez cum s-a terminat procesul fiu. */
    wait(&ret);
    if( WIFEXITED(ret) ) {
        switch( WEXITSTATUS(ret) ) {
            case 10: printf("Comanda ls nu a putut fi executata (eroare la exec).\n"); break;
            case 0:  printf("Comanda ls s-a executat cu succes.\n"); break;
            default: printf("Comanda ls s-a executat cu esec (cod: %d).\n", WEXITSTATUS(ret));
        }
    }
    else printf("Comanda ls a fost terminata fortat (semnal: %d).\n", WTERMSIG(ret));
    return 0;
}
```

Notă: programul complet este disponibil în exemplul [\[Exec command #1: ls\]](#), prezentat în suportul online al [laboratorului #11](#).



Alte programe demonstrative (cont.)

Introducere

Reacoperirea proceselor

Demo: programe cu `exec`

Exemplul #1: Reacoperirea unui program cu alt program

Exemplul #2: Reacoperirea recursivă

Exemplul #3: Reacoperirea unui program cu fișiere deschise

Exemplul #4: Redirecarea fluxului `stdout`

Exemplul #5: Reacoperirea unui program cu un script

Alte programe demonstrative

Referințe bibliografice

Demo: exemplele [Exec command #2: `last`] și [Exec command #3: `ls ...; rm ...`], ce sunt prezentate în suportul online al [laboratorului #11](#), ilustrează alte două programe care apelează prin `exec` o comandă simplă și, respectiv, o secvență de două comenzi simple, însoțite fiecare de câte o listă de argumente. Iar la final, fiecare program prelucrează statusul execuției comenzii respective (*i.e.*, succes vs. eșec).

Observație: funcția `system` permite lansarea de comenzi uzuale de UNIX/Linux dintr-un program C, printr-un apel de forma: `system(comanda)`;

Efect: se creează un nou proces, în care se încarcă *shell*-ul implicit, iar acesta va executa comanda specificată (pentru detalii suplimentare, consultați documentația `man 3 system`). Exemplificare:

```
#include ...
int main (int argc, char *argv []) {
    int ret; char cmdline[19+2*PATH_MAX]; char* dirname = (argc < 2) ? "." : argv[1];
    sprintf(cmdline, "ls -l %s ; rm -r -i %s", dirname, dirname);
    ret = system( cmdline ); /* Apelul functiei system pentru executia liniei de comanda */
    printf("Apelul system() s-a terminat, returnand valoarea: %d.\n", ret);
    return 0;
}
```

* * *

Demo: exemplul [['Supervisor-workers' pattern #1N: A coordinated distributed sum #1N \(v1, using regular files for IPC\)](#)], prezentat în suportul online al [laboratorului #11](#), ilustrează un program cu o funcționalitate mai complexă decât cele din exemplele precedente, ce utilizează de asemenea primitivele `fork`, `wait` și `exec` pentru implementarea funcționalității oferite. Acest program ilustrează o aplicare a șablonului de cooperare 'Supervisor/workers' pentru realizarea unui calcul paralel.



Bibliografie obligatorie

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu exec](#)

[Referințe bibliografice](#)

[1] Cap. 4, §4.4 din cartea “*Sisteme de operare – manual pentru ID*”, autor C. Vidrașcu, editura UAIC, 2006. *Notă*: este accesibilă, în format PDF, din pagina disciplinei “Sisteme de operare”:

- <https://profs.info.uaic.ro/~vidrascu/S0/books/ManualID-S0.pdf>

[2] Programele demonstrative amintite pe parcursul acestei prezentări pot fi descărcate de la:

- <https://profs.info.uaic.ro/~vidrascu/S0/cursuri/C-programs/exec/>

[3] Suportul online de laborator asociat acestei prezentări:

- https://profs.info.uaic.ro/~vidrascu/S0/labs/suport_lab11.html

Bibliografie suplimentară:

[4] Cap. 27 din cartea “The Linux Programming Interface : A Linux and UNIX System Programming Handbook”, autor M. Kerrisk, editura No Starch Press, 2010.

- <https://profs.info.uaic.ro/~vidrascu/S0/books/TLPI1.pdf>

[5] POSIX API: `man 2 execve`, `man 3 exec`, `man 2 dup`.