

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, 2020



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

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

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 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 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

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

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` transformă procesul apelant într-un alt proces specificat (prin numele fișierului executabil asociat) ca argument al apelului `exec`.

Noul proces se spune că “*reacoperă*” procesul ce a executat apelul `exec`, și el moștenește caracteristicile acestuia (inclusiv PID-ul), cu excepția a câtorva dintre ele.

Există în total 6 funcții din familia `exec` ([3]). Acestea diferă între ele prin nume și prin lista parametrilor de apel, și sunt împărțite în 2 categorii, ce se diferențiază prin forma în care se dau parametrii de apel:

- numărul de parametri este variabil
- numărul de parametri este fix



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 procesului care va reacoperi procesul apelant al respectivei primitive `exec`
 - celelalte argumente pot lipsi; ele exprimă parametrii liniei de comandă pentru procesul *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`. Mai poate fi și numele unui script, care începe cu o linie de forma `#!interpreter`.
2. 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 către noul proces a unui *environment* (i.e., un set de variabile de mediu)
 - celelalte argumente sunt la fel ca la prima pereche

Observații:

1. Și în acest caz, 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, mai poate fi și numele unui script, care începe cu o linie de forma `#!interpreter`.
2. La fel ca pentru `argv[]`, ultimul element din tabloul `env[]` 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](#)

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 ce este căutat în directoarele din variabila de mediu `PATH`, în cazul în care nu este specificat împreună cu calea, relativă sau absolută, până la acel fișier. De asemenea, mai poate fi și numele unui script, care începe cu o linie de forma `#!interpreter`.
2. 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](#)

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

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

Notă: familia `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 fapt, argumentul *ref* specifică *numele real* al fișierului executabil ce se va încărca și executa, iar `argv0`, respectiv `argv[0]`, specifică *numele afișat* (de comenzi precum **`ps`**, **`pstree`**, **`w`**, ș.a.) al noului proces.



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, etc.), cu unele excepții, în condițiile specificate:

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 <code>FD_CLOEXEC</code> de închidere automată în caz de <code>exec</code> , al vreunui descriptor de fișier, a fost setat cu ajutorul primitivei <code>fcntl</code> , atunci descriptorul respectiv este închis la <code>exec</code> (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

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

Referințe bibliografice

Introducere

Reacoperirea proceselor

Primitivele din familia `exec`

Caracteristicile procesului după `exec`

Demo: programe cu `exec`

Exemplul #1: Reacoperirea unui 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

Referințe bibliografice



Exemplul #1: Reacoperirea unui program

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu exec](#)

[Exemplul #1: Reacoperirea unui 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](#)

[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: ca o dovadă a faptului că “noul” proces `after_exec` moștenește descriptorii de fișiere deschise de la procesul `before_exec`, 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 `after_exec`).



Exemplul #2: Reacoperirea recursivă

[Introducere](#)

[Reacoperirea proceselor](#)

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

[Exemplul #1: Reacoperirea
unui 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](#)

[Referințe bibliografice](#)

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-un apel recursiv la infinit.

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



Exemplul #3: Reacoperirea unui program cu fișiere deschise

[Introducere](#)

[Reacoperirea proceselor](#)

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

[Exemplul #1: Reacoperirea unui 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](#)

[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` ([3]), 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

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

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!**”; aceasta deoarece mesajul de început “**Salut...**” se pierde prin `exec`, *buffer*-ul nefiind golit pe disc.



Exemplul #4: Redirectarea fluxului stdout

[Introducere](#)

[Reacoperirea proceselor](#)

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

[Exemplul #1: Reacoperirea unui 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](#)

[Referințe bibliografice](#)

Pe lângă primitiva `dup`, mai există o primitivă, cu numele `dup2`, utilă pentru duplicarea unui descriptor de fișier ([3]). Cu ajutorul lor se poate realiza redirectarea fluxurilor standard de I/O, așa cum 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`).

* * *

Demo: exercițiile rezolvate [`Exec command #1: ls`] și [`Exec command #2: last`] prezentate în **Laboratorul #10** ilustrează două exemple de programe care apelează prin `exec` câte o comandă uzuală din UNIX/Linux, împreună cu o linie de comandă, și care, totodată, prelucrează statusul comenzii respective (*i.e.*, succes vs. eșec).



Exemplul #5: Reacoperirea unui program cu un script

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu exec](#)

[Exemplul #1: Reacoperirea unui 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](#)

[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 `execl` pentru a se “reacoperi” cu un script bash, `my_script.sh` ([2]).

Notă: mai exact, aici, efectul apelului `exec` este 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).

* * *

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

Efect: se creează un nou proces, în care se încarcă *shell*-ul implicit, ce va executa comanda specificată.



Bibliografie obligatorie

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu exec](#)

[Referințe bibliografice](#)

[1] Capitolul 4, §4.4 din cartea “Sisteme de operare – manual pentru ID”, autor C. Vidrașcu, editura UAIC, 2006. Acest manual este accesibil, 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 adresa:

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

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