

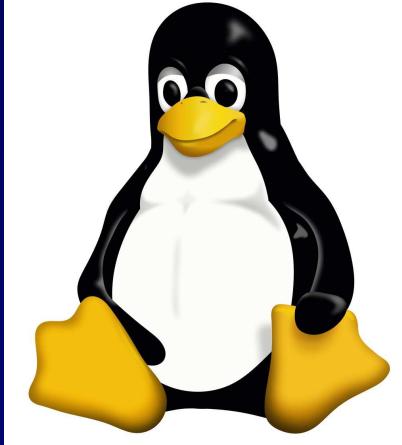
# **PROGRAMARE DE SISTEM ÎN C PENTRU PLATFORMA LINUX (I)**

**Gestiunea fișierelor, partea I-a:**

**Primitivele I/O pentru lucrul cu fișiere**

Cristian Vidrăscu  
vidrascu@info.uaic.ro

Martie, 2021



# Sumar

Introducere

[API-ul POSIX: funcții pentru operații I/O cu fișiere](#)

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

[Referințe bibliografice](#)

Introducere

## **API-ul POSIX: funcții pentru operații I/O cu fișiere**

Principalele categorii de primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

*Demo:* Un exemplu de *sesiune de lucru* cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre *file-system cache*-ul gestionat de nucleul Linux

## **Biblioteca standard de C: funcții pentru operații I/O cu fișiere**

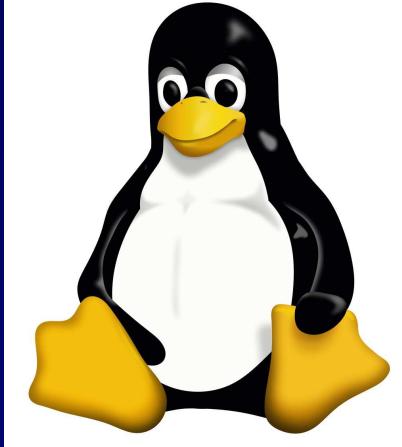
Despre biblioteca standard de C

Functiile I/O din biblioteca standard de C

Functiile de bibliotecă pentru I/O formatat

*Demo:* Un exemplu de *sesiune de lucru* cu fișiere

## **Referințe bibliografice**



# Introducere

Introducere

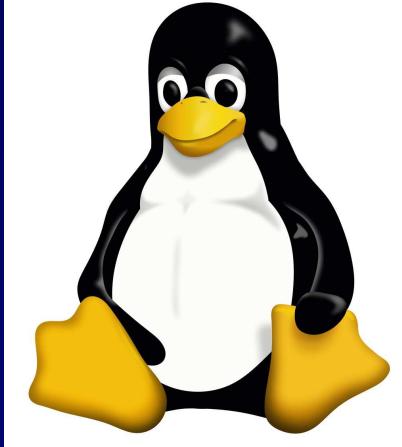
API-ul POSIX: funcții pentru operații I/O cu fișiere

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Referințe bibliografice

Funcțiile pe care le puteți apela în programele C pe care le scrieți, pentru a accesa și prelucra fișiere (atât fișiere obișnuite, cât și directoare sau alte tipuri de fișiere), se împart în două categorii:

- API-ul POSIX, ce oferă funcții *wrapper* pentru **apelurile de sistem** furnizate de nucleul Linux ; aceste funcții pot fi apelate din programe C ce vor fi compilate pentru platforma Linux și, mai general, pentru orice sistem de operare din familia UNIX ce implementează standardul POSIX.
  - Avantaj: funcțiile din acest API oferă, practic, acces la toate funcționalitățile din nucleul Linux “exportate” către *user-mode*.
  - Dezavantaj: programele care folosesc aceste funcții nu sunt portabile, e.g. nu pot fi compilate pentru platforma Windows (cel puțin nu direct, ci doar în mediul **WINDOWS SUBSYSTEM FOR LINUX**, introdus în Windows 10).
- STANDARD C LIBRARY (biblioteca standard de C), ce oferă o serie de funcții de nivel mai înalt, inclusiv pentru lucrul cu fișiere; aceste funcții pot fi apelate din programe C ce vor fi compilate pentru orice platformă ce oferă un compilator de C, plus o implementare a bibliotecii standard de C. Spre exemplu, pentru platforma Linux cel mai folosit este compilatorul **GCC** (*the GNU Compiler Collection*) și implementarea **GLIBC** (*the GNU libc*) a bibliotecii standard de C.
  - Avantaj: permite scrierea de programe portabile, între diverse platforme (e.g., Windows, UNIX/Linux, etc.).
  - Dezavantaj: conține funcții cu capacitate limitată de a gestiona resursele sistemului de operare (e.g., fișiere), fiind din acest motiv adecvată pentru scrierea unor programe simple.



# Agenda

Introducere

API-ul POSIX: funcții pentru operații I/O cu fișiere

Principalele categorii de primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

*Demo: Un exemplu de sesiune de lucru cu fișiere*

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre *file-system cache*-ul gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Referințe bibliografice

Introducere

**API-ul POSIX: funcții pentru operații I/O cu fișiere**

Principalele categorii de primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

*Demo: Un exemplu de sesiune de lucru cu fișiere*

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre *file-system cache*-ul gestionat de nucleul Linux

**Biblioteca standard de C: funcții pentru operații I/O cu fișiere**

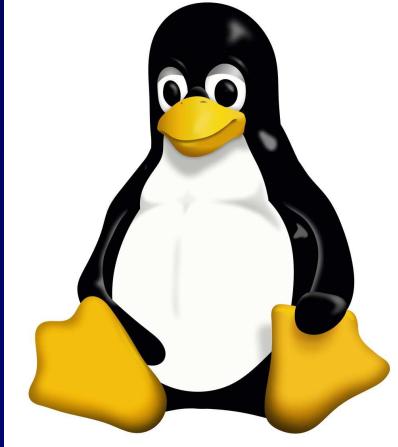
Despre biblioteca standard de C

Functiile I/O din biblioteca standard de C

Functiile de bibliotecă pentru I/O formatat

*Demo: Un exemplu de sesiune de lucru cu fișiere*

**Referințe bibliografice**



# Principalele categorii de primitive I/O

Introducere

API-ul POSIX: funcții pentru  
operări I/O cu fișiere

Principalele categorii de  
primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

Demo: Un exemplu de sesiune  
de lucru cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

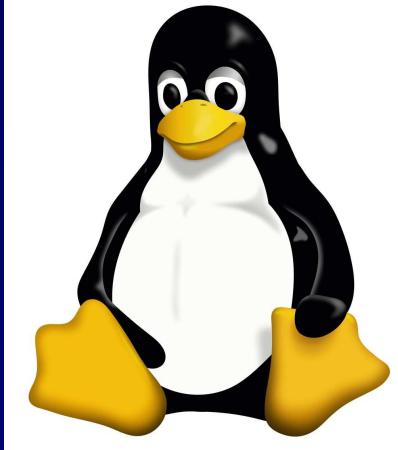
Despre file-system cache-ul  
gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Referințe bibliografice

Sistemul de gestiune a fișierelor în UNIX/Linux furnizează următoarele categorii de **apeluri sistem**, în conformitate cu standardul POSIX:

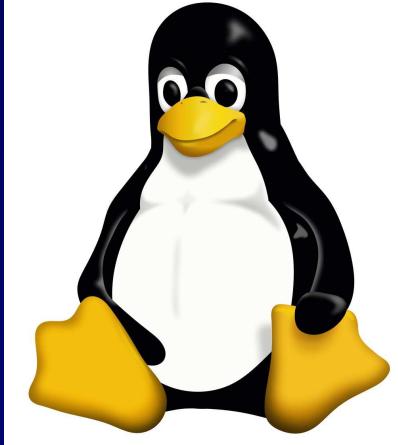
- primitive de creare de noi fișiere, de diverse tipuri: `mknod`, `mkfifo`, `mkdir`, `link`, `symlink`, `creat`, `socket`
- primitive de ștergere a unor fișiere: `rmdir` (pentru directoare), `unlink` (pentru toate celelalte tipuri)
- primitiva de redenumire a unui fișier, de orice tip: `rename`
- primitive de consultare a *i*-nodului unui fișier: `stat`/`fstat`/`lstat`, `access`
- primitive de manipulare a *i*-nodului unui fișier: `chmod`/`fchmod`, `chown`/`fchown`/`lchown`
- primitive de extindere a sistemului de fișiere: `mount`, `umount`
- primitive de accesare și manipulare a conținutului unui fișier, printr-o *sesiune de lucru*: `open`/`creat`, `read`, `write`, `lseek`, `close`, `fcntl`, și-a.
- primitive de duplicare a unei *sesiuni de lucru* cu un fișier: `dup`, `dup2`



## Principalele categorii de primitive I/O (cont.)

- primitive pentru consultarea “stării” unor *sesiuni de lucru cu fișiere* (operații I/O sincrone multiplexate): `select`, `poll`
- primitiva de “trunchiere” a conținutului unui fișier: `truncate`/`ftruncate`
- primitive de modificare a unor atribută dintr-un proces:
  - `chdir`: modifică directorul curent de lucru
  - `umask`: modifică “masca” permisiunilor implicite la crearea unui fișier
  - `chroot`: modifică rădăcina sistemului de fișiere accesibil procesului
- primitive pentru acces exclusiv la fișiere: `flock`, `fcntl`
- primitiva de “mapare” a unui fișier în memoria unui proces: `mmap`
- primitiva de creare, într-un proces, a unui canal de comunicație anonim: `pipe`
- s.a.

*Observație:* în caz de eroare, toate aceste primitive returnează valoarea `-1`, precum și un număr de eroare ce este stocat în variabila globală `errno` (definită în fișierul header `<errno.h>`), eroare ce poate fi diagnosticată cu funcția `perror()`.



# Primitiva access

Introducere

API-ul POSIX: funcții pentru  
operații I/O cu fișiere

Principalele categorii de  
primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

Demo: Un exemplu de sesiune  
de lucru cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre file-system cache-ul  
gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Referințe bibliografice

## ■ Verificarea drepturilor de acces la un fișier: primitiva access.

Interfața funcției access:

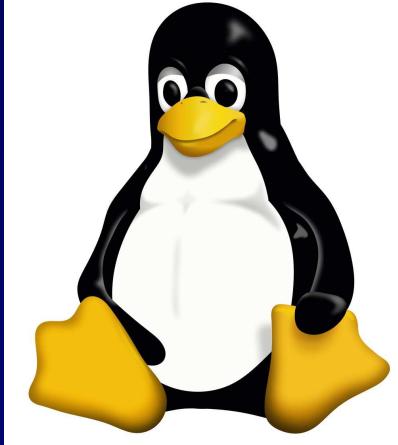
```
int access(char* nume_cale, int drept)
```

- *nume\_cale* = numele fișierului
- *drept* = dreptul de acces ce se verifică, ce poate fi o combinație (i.e., disjuncție logică pe biți) a următoarelor constante simbolice:

- ▲ X\_OK (=1) : procesul apelant are drept de execuție a fișierului ?
- ▲ W\_OK (=2) : procesul apelant are drept de scriere a fișierului ?
- ▲ R\_OK (=4) : procesul apelant are drept de citire a fișierului ?

*Notă:* pentru *drept=F\_OK* (=0) se verifică doar existența fișierului.

- valoarea returnată este 0, dacă accesul(e) verificat(e) este/sunt permis(e), respectiv -1 în caz de eroare.



## Primitiva creat

Introducere

API-ul POSIX: funcții pentru  
operații I/O cu fișiere

Principalele categorii de  
primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

Demo: Un exemplu de sesiune  
de lucru cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre file-system cache-ul  
gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Referințe bibliografice

### ■ *Crearea de fișiere de tip obișnuit: primitiva creat.*

Interfața funcției creat:

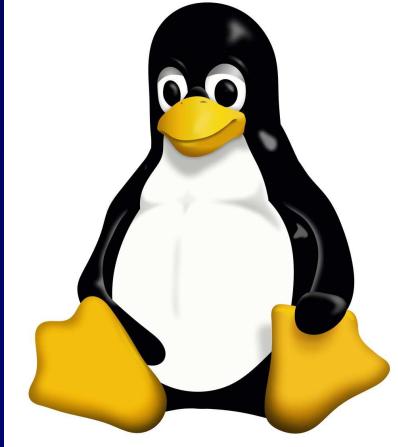
```
int creat(char* nume_cale, int perm_acces)
```

- *nume\_cale* = numele fișierului ce se creează
- *perm\_acces* = drepturile de acces pentru noul fișier creat
- valoarea returnată este descriptorul de fișier deschis, sau -1 în caz de eroare.

Efect: în urma execuției funcției creat se creează fișierul specificat și este “deschis” în scriere (!), valoarea returnată având aceeași semnificație ca la open.

*Observație:* în cazul când acel fișier deja există, el este trunchiat la zero, păstrându-i-se drepturile de acces pe care le avea.

*Notă:* practic, un apel `creat(nume_cale, perm_acces)` ; este echivalent cu apelul următor:  
`open(nume_cale, O_WRONLY | O_CREAT | O_TRUNC, perm_acces);`



## Primitiva open

Introducere

API-ul POSIX: funcții pentru  
operații I/O cu fișiere

Principalele categorii de  
primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

Demo: Un exemplu de sesiune  
de lucru cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre file-system cache-ul  
gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

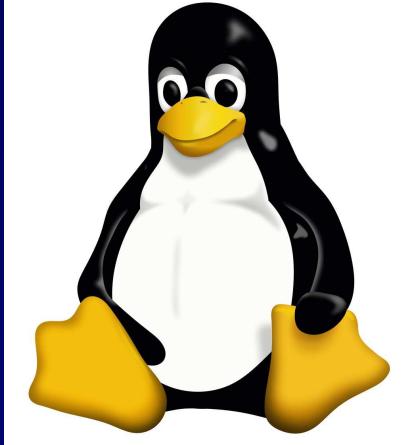
Referințe bibliografice

- “Deschiderea” unui fișier, i.e. inițializarea unei sesiuni de lucru: primitiva `open`.

Interfața funcției `open` ([3]):

```
int open(char* nume_cale, int tip_desch, int perm_acces)
```

- *nume\_cale* = numele fișierului ce se deschide
- *perm\_acces* = drepturile de acces pentru fișier (utilizat numai în cazul în care apelul va avea ca efect crearea acelui fișier)
- *tip\_desch* = specifică tipul deschiderii, putând fi exact una singură dintre valorile `O_RDONLY` ori `O_WRONLY` ori `O_RDWR`, și, eventual, combinată cu o combinație (i.e., disjuncție logică pe biți) a următoarelor constante simbolice: `O_APPEND`, `O_CREAT`, `O_TRUNC`, `O_EXCL`, `O_CLOEXEC`, `O_NONBLOCK`, și.a.
- valoarea returnată este descriptorul de fișier deschis (i.e., **indexul în tabela locală de fișiere deschise**), sau -1 în caz de eroare.



# Primitiva read

Introducere

API-ul POSIX: funcții pentru  
operații I/O cu fișiere

Principalele categorii de  
primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

Demo: Un exemplu de sesiune  
de lucru cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre file-system cache-ul  
gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Referințe bibliografice

## ■ Citirea dintr-un fișier: primitiva `read`.

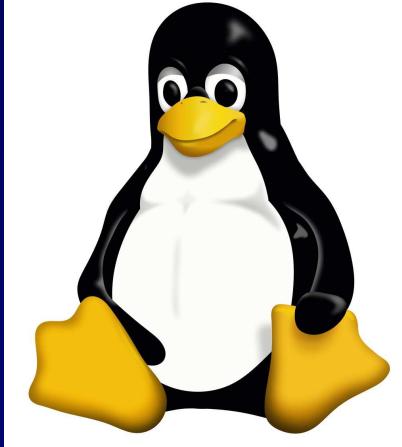
Interfața funcției `read` ([3]):

```
int read(int df, char* buffer, unsigned nr_oct)
```

- *df* = descriptorul fișierului din care se citește
- *buffer* = adresa de memorie la care se depun octetii cititi
- *nr\_oct* = numărul de octeți de citit din fișier
- valoarea returnată este numărul de octeți efectiv cititi, dacă citirea a reusit (chiar și parțial), sau -1 în caz de eroare.

*Observații:*

1. La sfârșitul citirii cursorul va fi poziționat pe următorul octet după ultimul octet efectiv citit.
2. Numărul de octeți efectiv cititi poate fi mai mic decât s-a specificat (e.g., dacă la începutul citirii cursorul în fișier este prea apropiat de sfârșitul fișierului); în particular, acesta poate fi chiar 0, dacă la începutul citirii cursorul în fișier este chiar pe pozitia EOF (i.e., *end-of-file*).



# Primitiva write

Introducere

API-ul POSIX: funcții pentru  
operații I/O cu fișiere

Principalele categorii de  
primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

Demo: Un exemplu de sesiune  
de lucru cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre file-system cache-ul  
gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Referințe bibliografice

## ■ *Scrierea într-un fișier: primitiva write.*

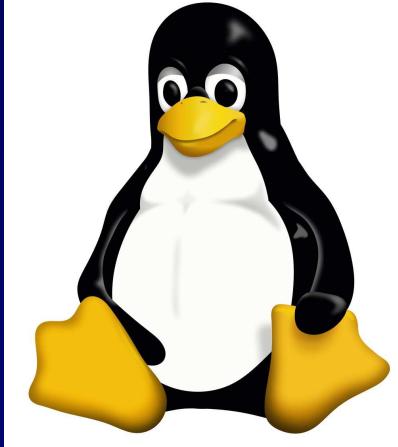
Interfața funcției write ([3]):

```
int write(int df, char* buffer, unsigned nr_oct)
```

- *df* = descriptorul fișierului în care se scrie
- *buffer* = adresa de memorie al cărei conținut se scrie în fișier
- *nr\_oct* = numărul de octeți de scris în fișier
- valoarea returnată este numărul de octeți efectiv scriși, dacă scrierea a reușit (chiar și parțial), sau -1 în caz de eroare.

*Observații:*

1. La sfârșitul scrierii cursorul va fi poziționat pe următorul octet după ultimul octet efectiv scris.
2. Numărul de octeți efectiv scriși poate fi mai mic decât s-a specificat (e.g., dacă acea scriere ar provoca mărirea spațiului alocat fișierului, iar aceasta nu se poate face din diverse motive – lipsă de spațiu liber sau depășire quota).



## Primitiva lseek

Introducere

API-ul POSIX: funcții pentru  
operări I/O cu fișiere

Principalele categorii de  
primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

Demo: Un exemplu de sesiune  
de lucru cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre file-system cache-ul  
gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

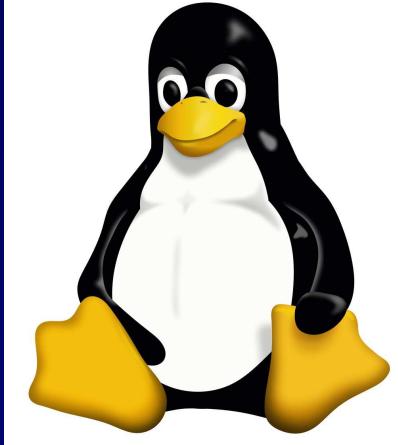
Referințe bibliografice

- *Pozitionarea cursorului într-un fisier* (i.e. ajustarea deplasamentului curent în fisier): primitiva lseek.

Interfața funcției lseek:

```
long lseek(int df, long val_ajust, int mod_ajust)
```

- *df* = descriptorul fișierului ce se poziționează
- *val\_ajust* = valoarea de ajustare a deplasamentului
- *mod\_ajust* = modul de ajustare, indicat după cum urmează:
  - ▲ SEEK\_SET (=0) : ajustare în raport cu începutul fișierului
  - ▲ SEEK\_CUR (=1) : ajustare în raport cu deplasamentul curent
  - ▲ SEEK\_END (=2) : ajustare în raport cu sfârșitul fișierului
- valoarea returnată este noul deplasament în fisier (întotdeauna, în raport cu începutul fișierului), sau -1 în caz de eroare.



## Primitiva close

Introducere  
API-ul POSIX: funcții pentru operări I/O cu fișiere  
Principalele categorii de primitive I/O  
Primitiva access  
Primitiva creat  
Primitiva open  
Primitiva read  
Primitiva write  
Primitiva lseek  
Primitiva close  
*Demo:* Un exemplu de sesiune de lucru cu fișiere  
Alte primitive I/O pentru fișiere  
Primitive I/O pentru directoare  
Şablonul de lucru cu directoare  
Despre file-system cache-ul gestionat de nucleul Linux  
Biblioteca standard de C:  
funcții pentru operații I/O cu fișiere  
Referințe bibliografice

- “Închiderea” unui fișier, i.e. finalizarea unei sesiuni de lucru: primitiva **close**.

Interfața funcției **close**:

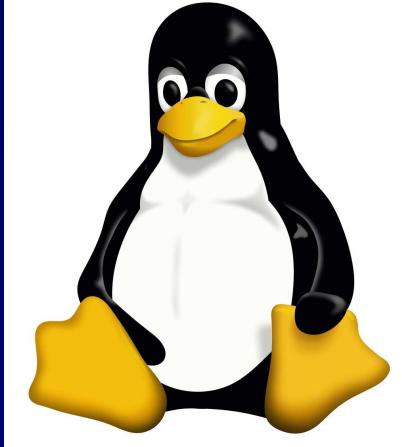
`int close(int df)`

- *df* = descriptorul de fișier deschis
- valoarea returnată este 0, dacă închiderea a reușit, respectiv -1 în caz de eroare.

**Observație:** maniera uzuală de prelucrare a unui fișier, i.e. o *sesiune de lucru*, constă în următoarele: “deschiderea fișierului”, urmată de o buclă de parcurgere a acestuia cu operații de citire și/sau de scriere, și eventual cu schimbări ale poziției curente în fișier, iar în final “închiderea” acestuia.

Exemplu: a se vedea cele două programe filtru `dos2unix.c` și `unix2dos.c` ([2]).

*Demo:* exercițiile rezolvate [AsciiStatistics] și [MyCp] prezентate în Laboratorul #6 ilustrează alte exemple de programe care apelează funcții I/O din API-ul POSIX pentru procesarea unor fișiere.



# Demo: Un exemplu de sesiune de lucru cu fișiere

Introducere

API-ul POSIX: funcții pentru operații I/O cu fișiere

Principalele categorii de primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

*Demo: Un exemplu de sesiune de lucru cu fișiere*

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre *file-system cache*-ul gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operații I/O cu fișiere

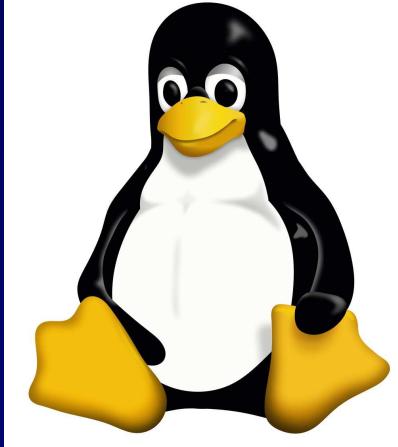
Referințe bibliografice

Iată un exemplu de program ce efectuează două sesiuni de lucru cu fișiere, mai exact realizează o copiere secvențială a unui fișier dat:

```
/* Basic cp file copy program. POSIX implementation. */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define BUF_SIZE 4096 // This is exactly the page size, for disk I/O efficiency!

int main (int argc, char *argv []) {
    int input_fd, output_fd;
    ssize_t bytes_in, bytes_out;
    char buffer[BUF_SIZE];
    if (argc != 3) { printf("Usage: cp file-src file-dest\n"); return 1; }
    input_fd = open(argv[1], O_RDONLY);
    if (input_fd == -1) { perror(argv[1]); return 2; }
    output_fd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, 0600);
    if (output_fd == -1) { perror(argv[2]); return 3; }

    /* Process the input file a record at atime. */
    while ((bytes_in = read(input_fd, buffer, BUF_SIZE)) > 0) {
        bytes_out = write(output_fd, buffer, bytes_in);
        if (bytes_out != bytes_in) {
            perror("Fatal write error."); return 4;
        }
    }
    close(input_fd); close(output_fd); return 0;
}
```



## Alte primitive I/O pentru fișiere

Introducere

API-ul POSIX: funcții pentru  
operații I/O cu fișiere

Principalele categorii de  
primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

Demo: Un exemplu de sesiune  
de lucru cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre file-system cache-ul  
gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Referințe bibliografice

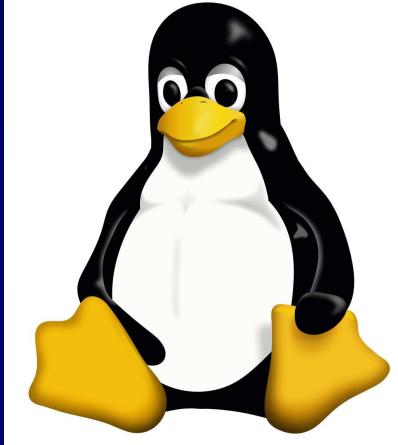
- “Duplicarea” unui descriptor de fișier: primitivele `dup` și `dup2`.
- Controlul operațiilor I/O: primitivele `fcntl` și `ioctl`.
- Obținerea de informații conținute de i-nodul unui fișier: primitivele `stat`, `lstat` sau `fstat`.
- Crearea/stergerea unei legături pentru un fișier: primitiva `link`, respectiv `unlink`.
- Schimbarea drepturilor de acces la un fișier: primitiva `chmod`.
- Schimbarea proprietarului unui fișier: primitivele `chown` și `chgrp`.
- Configurarea măștii drepturilor de acces la crearea unui fișier: primitiva `umask`.
- Montarea/demontarea unui sistem de fișiere: primitiva `mount`, respectiv `umount`.
- Crearea pipe-urilor (i.e. canale de comunicație anonime): primitiva `pipe`.
- Crearea fișierelor de tip *fifo* (i.e. canale de comunicație cu nume): primitiva `mkfifo`.

Interfața funcției `mkfifo`:

```
int mkfifo(char* nume_cale, int perm_acces);
```

- `nume_cale` = numele fișierului *fifo* ce se creează
- `perm_acces` = drepturile de acces pentru acesta
- valoarea returnată este 0 în caz de succes, sau -1 în caz de eroare.

- S.a.



# Primitive I/O pentru directoare

Introducere

API-ul POSIX: funcții pentru  
operații I/O cu fișiere

Principalele categorii de  
primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

Demo: Un exemplu de sesiune  
de lucru cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre file-system cache-ul  
gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Referințe bibliografice

- *Crearea/ștergerea unui director*: primitiva `mkdir`, respectiv `rmdir`.

Interfața funcției `mkdir`:

```
int mkdir(char* nume_cale, int perm_acces);
```

- *nume\_cale* = numele directorului ce se creează
- *perm\_acces* = drepturile de acces pentru acesta
- valoarea returnată este 0 în caz de succes, sau -1 în caz de eroare.

- *Aflarea directorului curent de lucru, al unui proces*: primitiva `getcwd`.

- *Schimbarea directorului curent, al unui proces*: primitiva `chdir`.

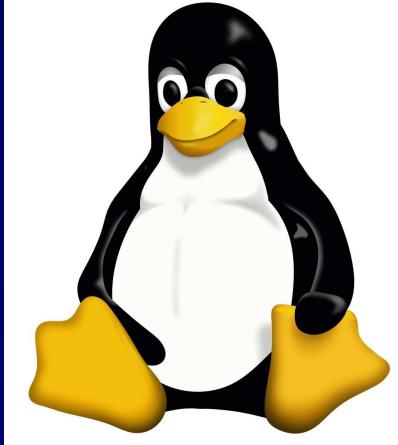
Interfața funcției `chdir`:

```
int chdir(char* nume_cale);
```

- *nume\_cale* = numele noului director curent de lucru, al procesului apelant
- valoarea returnată este 0 în caz de succes, sau -1 în caz de eroare.

- *"Prelucrarea" fișierelor dintr-un director*: primitivele `opendir`, `readdir` și `closedir`. Alte funcții utile: `rewinddir`, `seekdir`, `telldir` și `scandir`.

O sesiune de lucru cu directoare se implementează asemănător ca una cu fișiere, i.e. este o secvență de forma: "deschidere director", o buclă cu operații de citire, "Închidere director".



# Şablonul de lucru cu directoare

Introducere

API-ul POSIX: funcții pentru  
operări I/O cu fișiere

Principalele categorii de  
primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

Demo: Un exemplu de sesiune  
de lucru cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre *file-system cache*-ul  
gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Referințe bibliografice

Se folosesc tipurile de date DIR și struct dirent, împreună cu funcțiile enumerate, astfel:

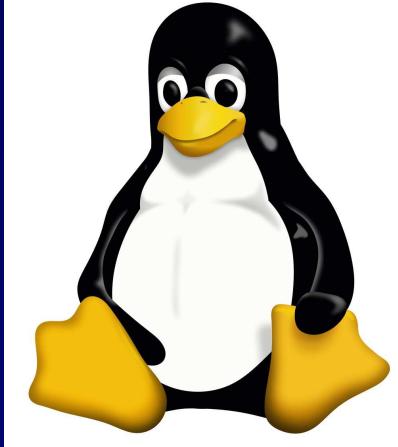
```
DIR          *dd; // descriptor de director deschis
struct dirent *de; // intrare in director

/* deschiderea directorului */
if( (dd = opendir(nume_director)) == NULL)
{
    ... // trateaza eroarea la deschidere
}

/* prelucrarea sequentiala a tuturor intrarilor din director */
while( (de = readdir(dd)) != NULL)
{
    ... // prelucreaza intrarea curenta, ce are numele: de->d_name
}

/* inchiderea directorului */
closedir(dd);
```

Demo: un exemplu de program ce utilizează acest şablon – a se vedea exercițiul rezolvat [MyFind #1] prezentat în Laboratorul #6 (de asemenea, el ilustrează și folosirea apelului stat(), pentru aflarea proprietăților unui fișier).



# Despre *file-system cache*-ul gestionat de nucleul Linux

Introducere

API-ul POSIX: funcții pentru operațiile I/O cu fișiere

Principalele categorii de primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

Demo: Un exemplu de sesiune de lucru cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre *file-system cache*-ul gestionat de nucleul Linux

Biblioteca standard de C:  
funcții pentru operațiile I/O cu fișiere

Referințe bibliografice

La nivelul componentei de gestiune a sistemelor de fișiere din cadrul nucleului unui SO, se folosește o zonă de memorie internă din *kernel-space* ce implementează un *cache* pentru operațiile cu discul (*i.e.*, se păstrează în memoria RAM conținutul celor mai recent accesate blocuri de disc).

Acest *cache* este denumit ***file-system cache*** (sau *disk cache*) în literatura de specialitate, iar el funcționează după aceleași **reguli generale ale *cache*-urilor** de orice fel:

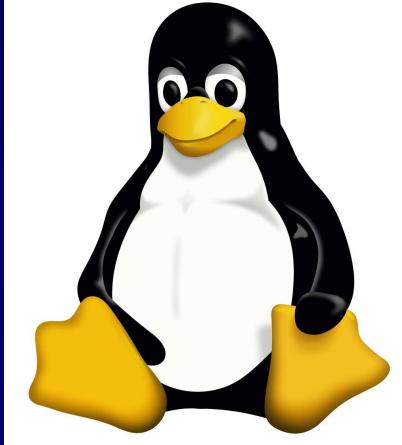
- i) citiri repetitive ale aceluiași bloc de disc, la intervale de timp foarte scurte, vor regăsi informația direct din *cache*-ul din memorie;
- ii) scrieri repetitive ale aceluiași bloc de disc, la intervale de timp foarte scurte, vor actualiza informația direct în *cache*-ul din memorie, iar pe disc informația va fi actualizată o singură dată, la momentul operației de ***cache-flushing***;
- iii) operațiile de invalidare/actualizare a informației din *cache*: ... ; ș.a.

Granularitatea acestui *cache* (*i.e.*, **unitatea de alocare** în *cache*) este pagina, care are o dimensiune dependentă de arhitectura hardware (*e.g.*, pentru arhitectura x86/x64 dimensiunea paginii este de 4096 octetii). Cu alte cuvinte, operațiile efective de I/O prin DMA între memorie și disc transferă blocuri de informație cu această dimensiune!

Acest *file-system cache* este unic per sistem, *i.e.* există o singură instanță a sa, gestionată de SO și utilizată simultan (ca și “resursă partajată”) de toate procesele ce se execută în sistem.

*Notă:* mai multe detalii despre aceste lucruri veți afla într-un curs teoretic ulterior.

Despre implicațiile existenței acestui *file-system cache* pentru programarea aplicațiilor folosind funcțiile **read** și **write** din API-ul POSIX puteți citi [aici](#).



# Agenda

Introducere

API-ul POSIX: funcții pentru operații I/O cu fișiere

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Despre biblioteca standard de  
C  
Funcțiile I/O din biblioteca  
standard de C  
Funcțiile de bibliotecă pentru  
I/O formatat  
*Demo:* Un exemplu de *sesiune de lucru* cu fișiere

Referințe bibliografice

Introducere

## **API-ul POSIX: funcții pentru operații I/O cu fișiere**

Principalele categorii de primitive I/O

Primitiva access

Primitiva creat

Primitiva open

Primitiva read

Primitiva write

Primitiva lseek

Primitiva close

*Demo:* Un exemplu de *sesiune de lucru* cu fișiere

Alte primitive I/O pentru fișiere

Primitive I/O pentru directoare

Şablonul de lucru cu directoare

Despre *file-system cache*-ul gestionat de nucleul Linux

## **Biblioteca standard de C: funcții pentru operații I/O cu fișiere**

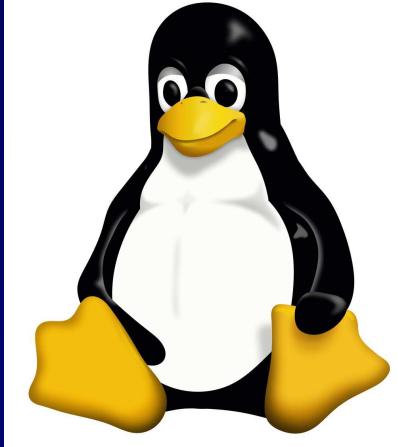
Despre biblioteca standard de C

Funcțiile I/O din biblioteca standard de C

Funcțiile de bibliotecă pentru I/O formatat

*Demo:* Un exemplu de *sesiune de lucru* cu fișiere

## **Referințe bibliografice**



# Despre biblioteca standard de C

Introducere

API-ul POSIX: funcții pentru  
operații I/O cu fișiere

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Despre biblioteca standard de  
C

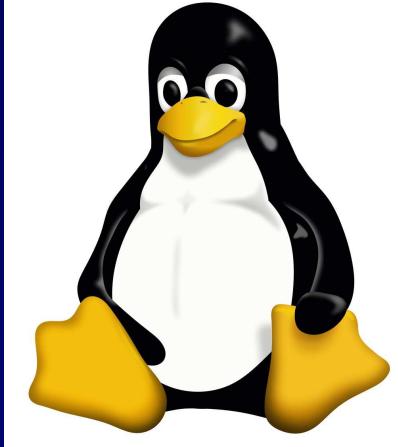
Funcțiile I/O din biblioteca  
standard de C

Funcțiile de bibliotecă pentru  
I/O formatat

Demo: Un exemplu de *sesiune*  
de lucru cu fișiere

Referințe bibliografice

- Biblioteca standard de C conține funcții cu capacitate limitată de a gestiona resursele sistemului de operare (e.g., fișiere)
- Este adeseori adecvată pentru scrierea unor programe simple
- Permite scrierea de programe portabile, între diverse platforme (e.g., Windows, UNIX/Linux, etc.)
- Include fișierele: `<stdlib.h>`, `<stdio.h>` și `<string.h>` ([4])
- Performanță competitivă
- Este restricționată doar la operații I/O sincrone
- Nu avem control al securității fișierelor prin biblioteca standard de C
  
- Apelul `fopen()` specifică dacă fișierul este text sau binar
- *Sesiunile de lucru cu fișiere* sunt identificate prin pointeri către structuri FILE
  - NULL semnifică valoare invalidă
  - Pointerii sunt “handles” pentru obiecte de tipul *sesiune de lucru cu un fișier*
- Erorile sunt diagnosticate cu funcțiile `perror()` sau `ferror()`



# Funcțiile I/O din biblioteca standard de C

Introducere

API-ul POSIX: funcții pentru  
operări I/O cu fișiere

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Despre biblioteca standard de  
C

Funcțiile I/O din biblioteca  
standard de C

Funcțiile de bibliotecă pentru  
I/O formatat

Demo: Un exemplu de sesiune  
de lucru cu fișiere

Referințe bibliografice

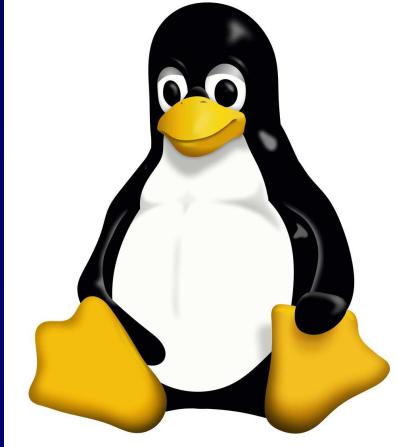
Biblioteca standard de C conține un set de funcții I/O (cele din header-ul `<stdio.h>` ([4])), care permit și ele prelucrarea unui fișier în maniera uzuală:

- `fopen` = pentru “deschiderea” fișierului
- `fread`, `fwrite` = pentru citire, respectiv scriere binară
- `fscanf`, `fprintf` = pentru citire, respectiv scriere formatată
- `fclose` = pentru “închiderea” fișierului

*Observație:* acestea sunt funcții de bibliotecă (nu sunt apeluri sistem) și lucrează *buffer*-izat, cu *stream*-uri I/O, iar descriptorii de fișiere utilizati de ele nu sunt de tip int, ci de tip FILE\*.

*Notă:* implementările acestor funcții de bibliotecă utilizează totuși apelurile de sistem corespunzătoare fiecărei platforme în parte (*i.e.*, Windows vs. Linux/UNIX).

*Observație:* sunt mult mai multe funcții I/O în biblioteca `<stdio.h>`; pentru a vedea lista lor și descrierea bibliotecii standard de I/O, inclusiv detalii despre cele 3 fluxuri I/O standard (*i.e.*, `stdin`, `stdout` și `stderr`), vă recomand consultarea paginii de manual `man 3 stdio`.



## Funcțiile I/O din biblioteca standard de C (cont.)

Introducere

[API-ul POSIX: funcții pentru operații I/O cu fișiere](#)

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Despre biblioteca standard de C

[Funcțiile I/O din biblioteca standard de C](#)

Funcțiile de bibliotecă pentru I/O formatat

*Demo:* Un exemplu de *sesiune de lucru* cu fișiere

[Referințe bibliografice](#)

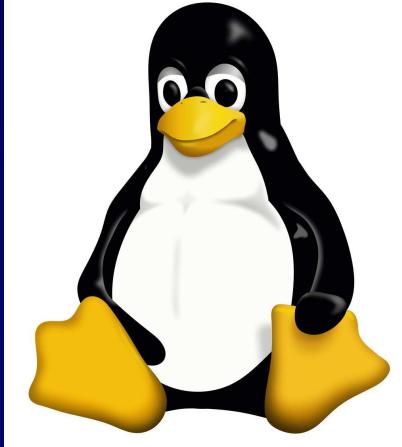
Ce înseamnă că aceste funcții de bibliotecă lucrează *buffer-izat* ?

*Răspuns:* înseamnă că folosesc un *cache* pentru disc implementat la nivelul bibliotecii standard de C (<stdio.h>), adică “deasupra” *file-system cache*-ului gestionat la nivelul nucleului SO-ului, despre care vă voi vorbi la cursurile teoretice.

Cu alte cuvinte, acesta este un *cache* al informațiilor din *file-system cache*, care la rândul său este un *cache* al informațiilor de pe disc.

În plus, acest *cache* gestionat de biblioteca <stdio.h> este implementat în *user-space* (la fel ca și toate funcțiile bibliotecii), ceea ce înseamnă că este *unic per proces* și nu per sistem, adică nu există un singur *cache* al bibliotecii care să fie partajat de toate procesele ce utilizează apele la bibliotecii.

*Concluzie:* rețineți faptul că acest *cache* gestionat de biblioteca stdio nu este unic per sistem, ca în cazul *file-system cache*-ului gestionat de SO, ci este “local” procesului.



# Funcțiile de bibliotecă pentru I/O formatat

Introducere

API-ul POSIX: funcții pentru  
operări I/O cu fișiere

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Despre biblioteca standard de  
C  
Funcțiile I/O din biblioteca  
standard de C  
Funcțiile de bibliotecă pentru  
I/O formatat

Demo: Un exemplu de *sesiune*  
de lucru cu fișiere

Referințe bibliografice

Biblioteca conține o serie de funcții care fac citiri/scrieri “formatate”, adică efectuează conversia între cele două reprezentări, *binară* vs. *textuală*, ale fiecărui tip de dată, pe baza unui argument *format* ce descrie conversiile de făcut prin niște “specificatori de format”. Funcțiile respective sunt:

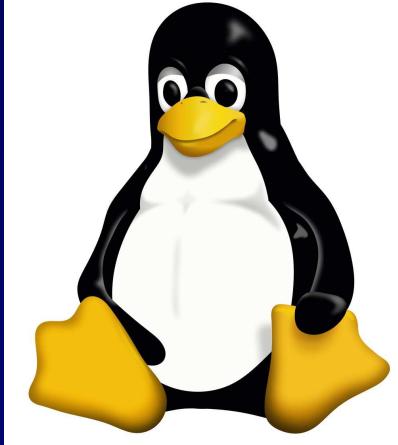
- perechea **scanf / printf** : citire de la `stdin`/scriere pe `stdout` ;
- perechea **fscanf / fprintf** : citire dintr-un fișier de pe disc/scriere într-un fișier de pe disc ;
- perechea **sscanf / sprintf** : citire dintr-un *string* în memorie / scriere într-un *string* în memorie .

Argumentul *format* folosește “specificatori de format”, de forma ’%literă’, pentru a descrie diferite tipuri de date și, astfel, determină ce fel de conversie se va face între cele două reprezentări, *binară* vs. *textuală*, ale tipului respectiv de dată.

Spre exemplu, iată câțiva specificatori de format și tipul de dată asociat fiecărui:

- %c : un caracter
- %s : un string (*null-terminated*)
- %d : un `int` (un întreg cu semn), reprezentarea *textuală* fiind cea corespunzătoare scrierii numărului în baza 10
- %u : un `unsigned int` (un întreg fără semn), reprezentarea *textuală* fiind cea corespunzătoare scrierii numărului în baza 10
- %o : un `unsigned int` (un întreg fără semn), reprezentarea *textuală* fiind cea corespunzătoare scrierii numărului în baza 8
- %x sau %X : un `unsigned int` (un întreg fără semn), reprezentarea *textuală* fiind cea corespunzătoare scrierii numărului în baza 16
- %f : un `double` (un număr “real” cu semn), reprezentarea *textuală* fiind cea corespunzătoare scrierii numărului în notația cu punct zecimal
- %e : un `double` (un număr “real” cu semn), reprezentarea *textuală* fiind cea corespunzătoare scrierii numărului în notația cu mantisă E
- s.a.

Pentru detalii suplimentare despre aceste perechi de funcții și despre argumentul *format* utilizat de ele, consultați documentația: `man 3 scanf` și `man 3 printf`. Suplimentar, puteți consulta și materialul disponibil [aici](#).



## Demo: Un exemplu de sesiune de lucru cu fișiere

Introducere

API-ul POSIX: funcții pentru operații I/O cu fișiere

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Despre biblioteca standard de  
C

Funcțiile I/O din biblioteca  
standard de C

Funcțiile de bibliotecă pentru  
I/O formatat

*Demo: Un exemplu de sesiune  
de lucru cu fișiere*

Referințe bibliografice

lată un exemplu de program ce efectuează două *sesiuni de lucru cu fișiere*, mai exact realizează o copiere secvențială a unui fisier dat:

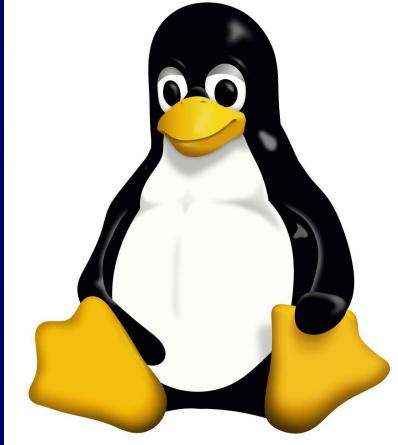
```
/* Basic cp file copy program. C library implementation. */
#include <stdio.h>
#define BUF_SIZE 4096 // This is exactly the page size, for disk I/O efficiency!

int main (int argc, char *argv []) {
    FILE *input_file, *output_file;
    ssize_t bytes_in, bytes_out;
    char buffer[BUF_SIZE];
    if (argc != 3) { printf("Usage: cp file-src file-dest\n"); return 1; }
    input_file = fopen(argv[1], "rb");
    if (input_file == NULL) { perror(argv[1]); return 2; }
    output_file = fopen(argv[2], "wb");
    if (output_file == NULL) { perror(argv[2]); return 3; }

    /* Process the input file a record at a time. */
    while ((bytes_in = fread(buffer, 1, BUF_SIZE, input_file)) > 0) {
        bytes_out = fwrite(buffer, 1, bytes_in, output_file);
        if (bytes_out != bytes_in) {
            perror("Fatal write error."); return 4;
        }
    }
    fclose(input_file); fclose(output_file);
    return 0;
}
```

*Notă:* acest exemplu este disponibil pentru descărcare de aici: [cp\\_stdio.c \(\[2\]\)](#).

*Demo: exercițiile rezolvate [ArithmeticMean], [MyExpr] și [MyWc] prezentate în Laboratorul #6 ilustrează alte exemple de programe care apelează funcții I/O din biblioteca `<stdio.h>`.*



## Bibliografie obligatorie

Introducere

API-ul POSIX: funcții pentru operații I/O cu fișiere

Biblioteca standard de C:  
funcții pentru operații I/O cu  
fișiere

Referințe bibliografice

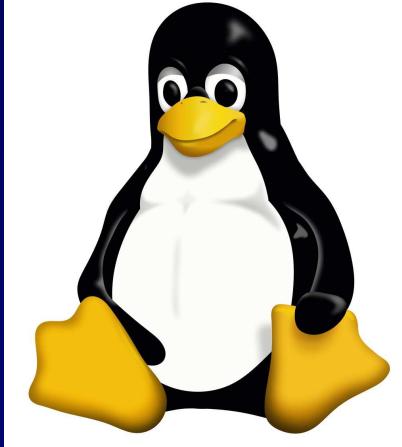
- [1] Capitolul 3, §3.1 din cartea “Sisteme de operare – manual pentru ID”, autor C. Vidrașcu, editura UAIC, 2006. Acest manual este accesibil, în format PDF, din pagina disciplinei “Sisteme de operare”:
- <https://profs.info.uaic.ro/~vidrascu/S0/books/ManualID-S0.pdf>
- [2] Programele demonstrative amintite pe parcursul acestei prezentări pot fi descărcate de la adresa:
- <https://profs.info.uaic.ro/~vidrascu/S0/cursuri/C-programs/file/>
- [3] POSIX API: `man 2 open`, `man 2 read`, `man 2 write`, și.a.
- [4] STANDARD C LIBRARY: `man 3 stdio`, `man 3 string`, `man 0p stdlib.h`.

# **PROGRAMARE DE SISTEM ÎN C PENTRU PLATFORMA LINUX (II)**

**Gestiunea fișierelor, partea a II-a:  
Accesul concurent sau exclusiv la fișiere. Blocaje pe fișiere**

Cristian Vidrăscu  
[vidrascu@info.uaic.ro](mailto:vidrascu@info.uaic.ro)

Martie, 2021



# Sumar

Introducere

Modul de acces concurrent la fisiere

Modul de acces exclusiv la fisiere – Blocaje pe fisiere

Referințe bibliografice

Introducere

## Modul de acces concurrent la fisiere

*Demo (1): Un exemplu de acces concurrent la un fișier*

## Modul de acces exclusiv la fisiere – Blocaje pe fisiere

Structura de date flock pentru blocage

Primitiva fcntl pentru blocage

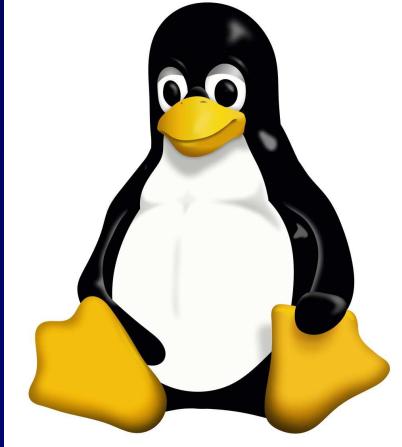
Caracteristici ale blocajelor pe fisiere

*Demo (2): Un exemplu de acces exclusiv la un fișier*

*Demo (3): Ilustrarea caracterului *advisory* al blocajelor*

*Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier*

Referințe bibliografice



## Introducere

[Introducere](#)

[Modul de acces concurrent la fisiere](#)

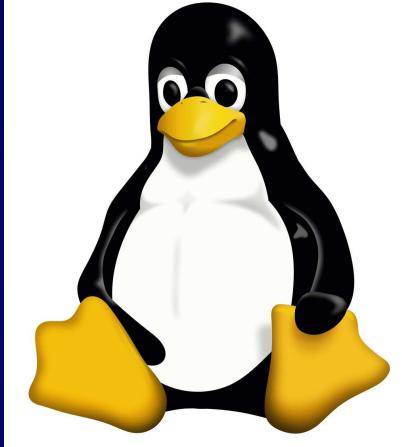
[Modul de acces exclusiv la fisiere – Blocaje pe fisiere](#)

[Referințe bibliografice](#)

Deoarece sistemele de operare din familia UNIX (în particular, și Linux-ul) sunt sisteme *multi-tasking* (*i.e.*, sisteme care suportă execuția “simultană” a mai multor programe), în mod ușual este permis *accesul concurrent* la fisiere, adică mai multe procese pot accesa “simultan” în citire și/sau în scriere un același fișier, sau chiar o aceeași înregistrare dintr-un fișier.

Acest mod de acces concurrent (“simultan”) la un fișier de către procese diferite poate avea însă uneori și efecte nedorite (ca de exemplu, distrugerea integrității datelor din fișier).

Din acest motiv, în sistemele din familia UNIX s-au implementat mecanisme care să permită și un mod de *acces exclusiv* la fisiere, adică un mod de acces în care un singur proces are, la un moment dat, permisiunea de acces la un fișier, sau chiar la o anumită înregistrare dintr-un fișier.



# Agenda

Introducere

Modul de acces concurrent la fisiere

*Demo (1): Un exemplu de acces concurrent la un fișier*

Modul de acces exclusiv la fisiere – Blocaje pe fisiere

Referințe bibliografice

Introducere

## Modul de acces concurrent la fisiere

*Demo (1): Un exemplu de acces concurrent la un fișier*

## Modul de acces exclusiv la fisiere – Blocaje pe fisiere

Structura de date flock pentru blocaje

Primitiva fcntl pentru blocaje

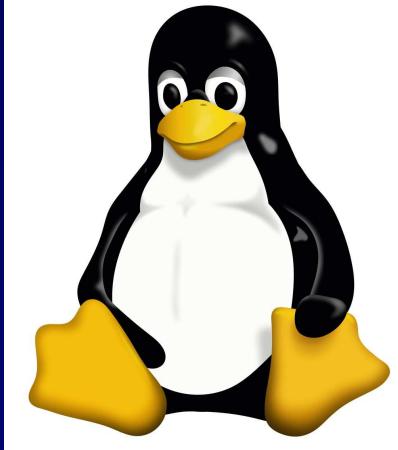
Caracteristici ale blocajelor pe fisiere

*Demo (2): Un exemplu de acces exclusiv la un fișier*

*Demo (3): Ilustrarea caracterului *advisory* al blocajelor*

*Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier*

Referințe bibliografice



## Demo (1): Un exemplu de acces concurrent la un fișier

Introducere

Modul de acces concurrent la fișiere

*Demo (1): Un exemplu de acces concurrent la un fișier*

Modul de acces exclusiv la fișiere – Blocaje pe fișiere

Referințe bibliografice

*Observație:* d.p.d.v. al programatorului, acesta nu trebuie să utilizeze nicio tehnică suplimentară celor discutate în lectia precedentă despre accesul la fisiere, pentru a “beneficia” de accesul în mod concurrent (“simultan”) la un fișier. Totul se petrece la momentul execuției: dacă utilizatorul rulează în același timp două sau mai multe instanțe de programe ce accesează în mod ușual un același fișier, atunci accesele la fișier se vor petrece “simultan” (i.e., aprox. în același timp).

Iată un exemplu de program ce poate fi utilizat pentru a ilustra efectele accesului concurrent la un fișier: a se vedea programul `access_v1.c` ([2]).

Mai întâi, un demo de execuție ce ilustrează *accesul secvențial la fișier*, i.e. un singur proces dorește să acceseze fișierul într-un anumit interval de timp.

Creăm un fișier `fis.dat` ce conține următoarea linie de text: `aaaa#bbbb#cccc#dddd#eeee`

Apoi lansăm în execuție secvențială mai multe instanțe ale acestui program, e.g. prin comanda:

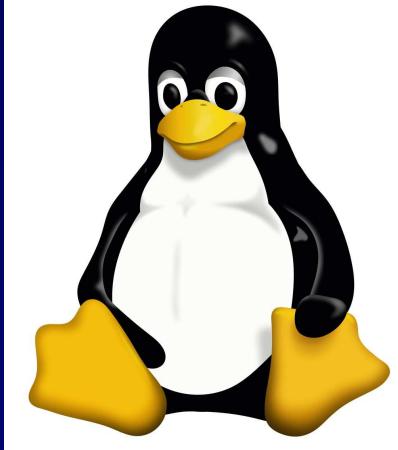
UNIX> `./access_v1 1 ; ./access_v1 2 ; ./access_v1 3`

Care va fi conținutul fișierului după terminarea execuției acestei comenzi ?

După execuția primei instanțe, fișierul va arăta astfel: `aaaa1bbbb#cccc#dddd#eeee`

După execuția instanței a doua, fișierul va arăta astfel: `aaaa1bbbb2cccc#dddd#eeee`

După execuția instanței a treia, rezultatul final va arăta astfel: `aaaa1bbbb2cccc3dddd#eeee`



## Demo (1): Un exemplu de acces concurrent la un fișier (cont.)

Introducere

Modul de acces concurrent la fișiere

*Demo (1): Un exemplu de acces concurrent la un fișier*

Modul de acces exclusiv la fișiere – Blocaje pe fișiere

Referințe bibliografice

În acum, un demo de execuție ce ilustrează *accesul concurrent la fișier*: mai multe procese (*i.e.*, instanțe ale programului) ce doresc să acceseze fișierul în același interval de timp.

“Reinițializăm” fișierul `fis.dat` cu următoarea linie de text:

`aaaa#bbbb#cccc#ddd#eeee`

Apoi lansăm în execuție paralelă (“simultană”) două instanțe ale acestui program, prin comanda:

UNIX> `./access_v1 1 & ./access_v1 2 &`

Care va fi conținutul fișierului după terminarea execuției acestei comenzi ?

Probabil vă așteptați ca după execuție fișierul să arate astfel:

`aaaa1bbbb2cccc#ddd#eeee`

sau

`aaaa2bbbb1cccc#ddd#eeee`

(în funcție de care dintre cele două procese a reușit mai întâi să suprascrie primul caracter '#' din acest fișier, celuilalt proces rămânându-i al doilea caracter '#' pentru a-l suprascrie.)

În realitate, repetând de oricâte ori execuția acestei comenzi, întotdeauna se va obține:

`aaaa1bbbb#cccc#ddd#eeee`

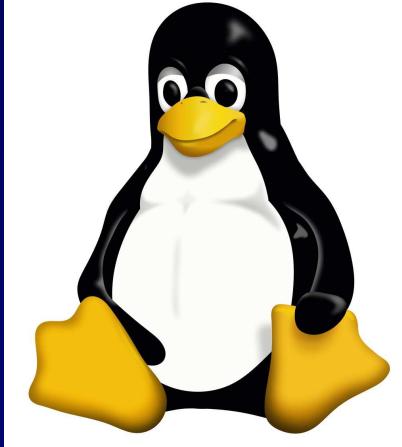
sau

`aaaa2bbbb#cccc#ddd#eeee`

Motivul: datorită apelului `sleep(5)` care provoacă o așteptare de 5 secunde între momentul depistării primei înregistrări din fișier care este '#' și momentul suprascrierii acestei înregistrări cu alt caracter.

*Observație:* prin eliminarea apelului `sleep(5)` din program, repetând execuția acestei comenzi de un număr suficient de mare de ori, se pot obține toate cele 4 rezultate de mai sus, cu frecvențe diferite de observare.

*Demo:* pentru explicații mai detaliate, a se vedea [FirstDemo] prezentat în suportul de laborator #7.



# Agenda

Introducere

Modul de acces concurrent la fisiere

Modul de acces exclusiv la fisiere – Blocaje pe fisiere

Structura de date flock pentru blocaje

Primitiva fcntl pentru blocaje  
Caracteristici ale blocajelor pe fisiere

*Demo (1): Un exemplu de acces exclusiv la un fisier*

*Demo (2): Ilustrarea caracterului *advisory* al blocajelor*

*Demo (3): Un exemplu de acces exclusiv *optimizat* la un fisier*

Referințe bibliografice

## Introducere

### **Modul de acces concurrent la fisiere**

*Demo (1): Un exemplu de acces concurrent la un fisier*

### **Modul de acces exclusiv la fisiere – Blocaje pe fisiere**

**Structura de date flock pentru blocaje**

**Primitiva fcntl pentru blocaje**

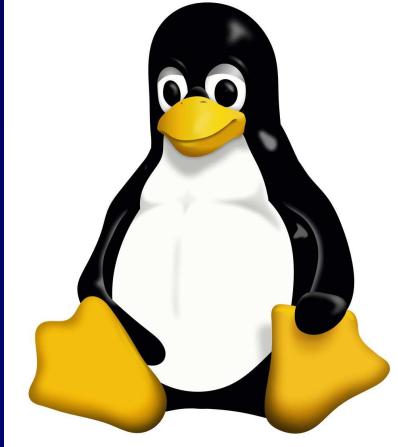
**Caracteristici ale blocajelor pe fisiere**

*Demo (2): Un exemplu de acces exclusiv la un fisier*

*Demo (3): Ilustrarea caracterului *advisory* al blocajelor*

*Demo (4): Un exemplu de acces exclusiv *optimizat* la un fisier*

## Referințe bibliografice



## Structura de date flock pentru blocaje

Introducere

Modul de acces concurrent la fisiere

Modul de acces exclusiv la fisiere – Blocaje pe fisiere

Structura de date flock pentru blocaje

Primitiva fcntl pentru blocaje

Caracteristici ale blocajelor pe fisiere

Demo (2): Un exemplu de acces exclusiv la un fisier

Demo (3): Ilustrarea caracterului *advisory* al blocajelor

Demo (4): Un exemplu de acces exclusiv *optimizat* la un fisier

Referințe bibliografice

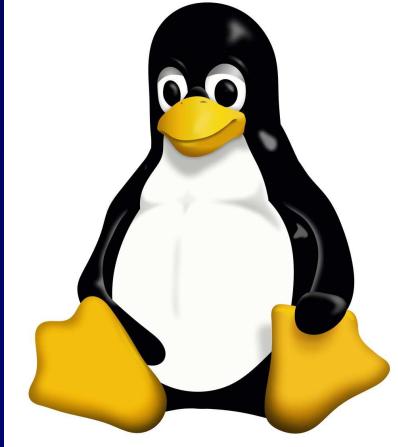
Sistemele din familia UNIX furnizează programatorilor un **mecanism de blocare** (*i.e.*, de punere de “*lacăte*”) pe **portiuni de fisier** pentru accesul în mod exclusiv.

Prin acest mecanism se definește o zonă de **acces exclusiv** în fisier. O asemenea porțiune nu va putea fi accesată în mod concurrent de mai multe procese pe toată durata de existență a blocajului.

Pentru a specifica un blocaj (*i.e.*, un “*lacăt*”) pe o porțiune dintr-un fisier (sau pe întregul fisier), se utilizează structura de date **flock**, definită în fisierul header **fcntl.h** în felul următor:

```
struct flock
{
    short l_type;    // indica tipul blocarii
    short l_whence; // indica pozitia relativă (originea)
    long l_start;   // indica pozitia de start, in raport cu originea
    long l_len;     // indica lungimea portiunii blocate
    int l_pid;
}
```

*Observație:* după ce se completează câmpurile structurii de mai sus, ulterior se va apela funcția **fcntl** pentru a pune efectiv “*lacătul*” pe porțiunea respectivă din fisier.



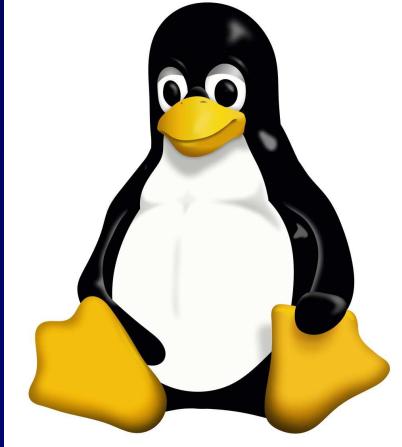
## Structura de date flock pentru blocaje (cont.)

Semnificația câmpurilor structurii flock:

- câmpul l\_type indică tipul blocării, putând avea ca valoare una dintre constantele:
  - F\_RDLCK : blocaj în citire
  - F\_WRLCK : blocaj în scriere
  - F\_UNLCK : deblocaj (*i.e.*, se înlătură lacătul)
- câmpul l\_whence indică poziția relativă (*i.e.*, originea) în raport cu care este interpretat câmpul l\_start, putând avea ca valoare una dintre următoarele constante simbolice:
  - SEEK\_SET (=0) : originea este BOF (*i.e.*, *begin of file*)
  - SEEK\_CUR (=1) : originea este CURR (*i.e.*, *current position in file*)
  - SEEK\_END (=2) : originea este EOF (*i.e.*, *end of file*)
- câmpul l\_start indică poziția (*i.e.*, offset-ul în raport cu originea l\_whence) de la care începe porțiunea blocată.

*Observație:* l\_start trebuie să fie negativ pentru l\_whence=SEEK\_END.
- câmpul l\_len indică lungimea în octeți a porțiunii blocate.
- câmpul l\_pid este gestionat de funcția fcntl care pune blocajul, fiind utilizat pentru a memora PID-ul procesului proprietar al acelui lacăt.

*Observație:* are sens consultarea acestui câmp doar atunci când funcția fcntl se apelează cu parametrul F\_GETLK.



## Primitiva fcntl pentru blocaje

Introducere

Modul de acces concurrent la fisiere

Modul de acces exclusiv la fisiere – Blocaje pe fisiere

Structura de date flock pentru blocaje

Primitiva fcntl pentru blocaje

Caracteristici ale blocajelor pe fisiere

Demo (2): Un exemplu de acces exclusiv la un fisier

Demo (3): Ilustrarea caracterului *advisory* al blocajelor

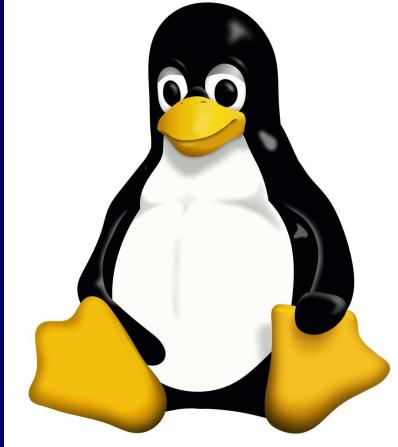
Demo (4): Un exemplu de acces exclusiv *optimizat* la un fisier

Referințe bibliografice

Interfața funcției fcntl ([3] – una dintre ele, cea pentru blocaje):

```
int fcntl(int fd, int mod, struct flock* sfl)
```

- *fd* = descriptorul de fisier deschis pe care se pune lacătul
- *sfl* = adresa structurii flock ce definește acel lacăt
- *mod* = indică modul de punere, putând lua una dintre valorile:
  - *F\_SETLK* : permite punerea unui lacăt pe fisier, în citire sau în scriere, sau scoaterea uneia deja pus (funcție de tipul specificat în structura flock).  
*Observație:* în caz de eșec datorită conflictului cu alt lacăt deja pus, se setează variabila errno la valoarea EACCES sau la EAGAIN.
  - *F\_GETLK* : permite extragerea informațiilor despre un lacăt pus pe fisier.
  - *F\_SETLKW* : permite punerea lacătelor în mod “blocant”, adică se așteaptă (*i.e.*, funcția nu returnează) până când se poate pune lacătul. Motivul posibil de așteptare: se încearcă blocarea unei zone deja blocate de un alt proces.
- valoarea returnată este 0, sau -1 în caz de eroare.



## Primitiva fcntl pentru blocaje (cont.)

### Observații:

- Pentru a putea pune un lacăt în citire, respectiv în scriere, pe un descriptor de fișier, acesta trebuie să fi fost anterior deschis în citire, respectiv în scriere.
- Blocajul este scos automat atunci când procesul care l-a pus înlătură acel fișier, sau își termină execuția.
- Scoaterea (deblocarea) unui segment dintr-o portiune mai mare anterior blocată poate produce două segmente blocate.
- Câmpul l\_pid din structura flock este actualizat de funcția fcntl.
- Lacătele nu se transmit proceselor fiii în momentul creării acestora cu funcția fork.  
Motivul: fiecare lacăt are în structura flock asociată PID-ul procesului care l-a creat (și care este deci proprietarul lui), iar procesele fii au, bineînțeles, PID-uri diferite de cel al părintelui.
- În Linux mai există alte două interfețe ce oferă lacăte pe fișiere:
  - funcția flock → pentru detalii consultați documentația: `man 2 flock`
  - funcția lockf → pentru detalii consultați documentația: `man 3 lockf`
- Există și două comenzi utile pentru lacăte: `flock` și `lsocks` ([4]).

Introducere

Modul de acces concurrent la fisiere

Modul de acces exclusiv la fisiere – Blocaje pe fisiere

Structura de date flock pentru blocaje

Primitiva fcntl pentru blocaje

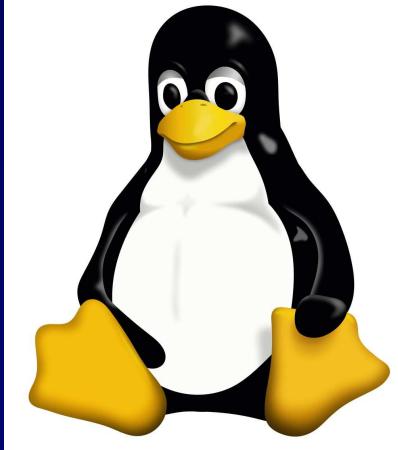
Caracteristici ale blocajelor pe fisiere

Demo (2): Un exemplu de acces exclusiv la un fișier

Demo (3): Ilustrarea caracterului *advisory* al blocajelor

Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier

Referințe bibliografice



## Caracteristici ale blocajelor pe fișiere

Introducere  
Modul de acces concurrent la fișiere  
  
Modul de acces exclusiv la fișiere – Blocaje pe fișiere  
Structura de date flock pentru blocaje  
Primitiva fcntl pentru blocaje  
Caracteristici ale blocajelor pe fișiere  
*Demo (2): Un exemplu de acces exclusiv la un fișier*  
*Demo (3): Ilustrarea caracterului *advisory* al blocajelor*  
*Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier*  
  
[Referințe bibliografice](#)

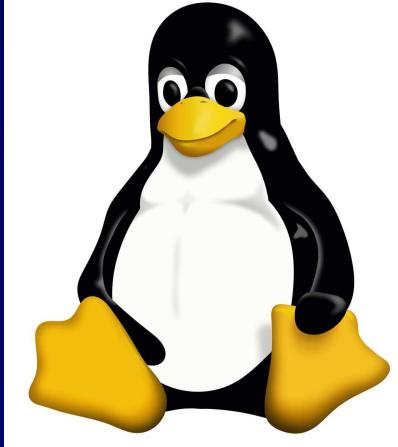
- **Important:** lacătele în scriere (*i.e.*, cele cu tipul F\_WRLCK) sunt *exclusive*, iar cele în citire (*i.e.*, cele cu tipul F\_RDLCK) sunt *partajate*, în sensul **CREW** (“*Concurrent Read or Exclusive Write*”).

Cu alte cuvinte: în orice moment, pentru orice porțiune dintr-un fișier, cel mult un proces poate deține un lacăt în scriere pe acea porțiune (și atunci nici un proces nu poate deține concomitent vreun lacăt în citire), sau este posibil ca mai multe procese să dețină lacăte în citire pe acea porțiune (și atunci nici un proces nu poate deține concomitent vreun lacăt în scriere).

- **Important:** funcționarea corectă a lacătelor se bazează pe *cooperarea* proceselor pentru asigurarea accesului exclusiv la fișiere, *i.e.* toate procesele care vor să acceseze mutual exclusiv un fișier (sau o porțiune dintr-un fișier) vor trebui să folosească lacăte pentru accesul respectiv.

Cu alte cuvinte: **blocajele puse pe fișiere sunt *advisory*, nu sunt *mandatory*!**

Altfel, spre exemplu, dacă un proces scrie direct un fișier (sau o porțiune dintr-un fișier), apelul său de scriere NU va fi împiedicat de un eventual lacăt în scriere (sau citire) pus pe acel fișier (sau acea porțiune de fișier) de către un alt proces.



## Demo (2): Un exemplu de acces exclusiv la un fișier

Introducere

Modul de acces concurrent la fișiere

Modul de acces exclusiv la fișiere – Blocaje pe fișiere

Structura de date flock pentru blocaje

Primitiva fcntl pentru blocaje

Caracteristici ale blocajelor pe fișiere

*Demo (2): Un exemplu de acces exclusiv la un fișier*

*Demo (3): Ilustrarea caracterului advisory al blocajelor*

*Demo (4): Un exemplu de acces exclusiv optimizat la un fișier*

Referințe bibliografice

Putem rescrie programul anterior, adăugând utilizarea de lacăte în scriere pentru a “inhiba” accesul concurrent la fișier: a se vedea programul `access_v2.c` ([2]).

“Reinițializăm” fișierul `fis.dat` cu următoarea linie de text: `aaaa#bbbb#cccc#dddd#eeee`

Apoi lansăm în execuție paralelă (“simultană”) două instanțe ale acestui program, prin comanda:

UNIX> `./access_v2 1 & ./access_v2 2 &`

Care va fi conținutul fișierului după terminarea execuției acestei comenzi ?

De data aceasta, oricâtă execuții s-ar face, întotdeauna se va obține rezultatul urmărit:

`aaaa1bbbb2cccc#dddd#eeee`

sau

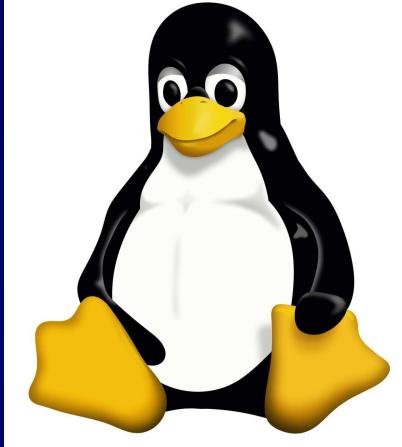
`aaaa2bbbb1cccc#dddd#eeee`

*Observație:* în programul de mai sus apelul de punere a lacătului este nebllocant (*i.e.*, cu parametrul `F_SETLK`). Se poate face și un apel blocant, *i.e.* funcția `fcntl` nu va returna imediat, ci va sta în așteptare până când reușește să pună lacătul.

A se vedea programul `access_v2w.c`

Lansând simultan în execuție două instanțe ale acestui program, se va constata că obținem același rezultat ca și în cazul variantei nebllocante.

*Demo:* pentru explicații mai detaliate, a se vedea [SecondDemo] prezentat în suportul de laborator #7.



## Demo (3): Ilustrarea caracterului *advisory* al blocajelor

Introducere

Modul de acces concurrent la fisiere

Modul de acces exclusiv la fisiere – Blocaje pe fisiere

Structura de date flock pentru blocaje

Primitiva fcntl pentru blocaje  
Caracteristici ale blocajelor pe fisiere

Demo (2): Un exemplu de acces exclusiv la un fișier

Demo (3): Ilustrarea caracterului *advisory* al blocajelor

Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier

Referințe bibliografice

lată o justificare a observației anterioare despre caracterul *advisory* al blocajelor:

“Reinițializăm” fișierul `fis.dat` cu linia de text: `aaaa#bbbb#cccc#dddd#eeee`

și apoi rulăm următoarea comandă:

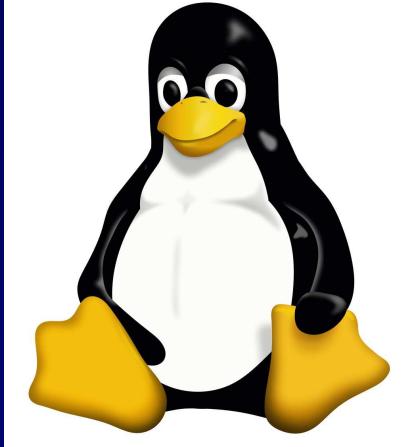
UNIX> `./access_v2 1 & sleep 2 ; echo "xyxyxy" > fis.dat`

Care va fi conținutul fișierului după terminarea execuției acestei comenzi ?

\* \* \*

Răspuns: la finalul execuției acestei comenzi, fișierul `fis.dat` va conține linia de text: `xyxy1y`, ceea ce ne demonstrează că suprascrierea executată de comanda `echo` în fișier s-a petrecut în intervalul de timp al celor 5 secunde în care instanța `access_v2` deținea blocajul pe fișier!

Demo: pentru explicații mai detaliate, a se revedea ultima parte din [SecondDemo].



## Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier

Introducere

Modul de acces concurrent la fișiere

Modul de acces exclusiv la fișiere – Blocaje pe fișiere

Structura de date flock pentru blocaje

Primitiva fcntl pentru blocaje

Caracteristici ale blocajelor pe fișiere

*Demo (2): Un exemplu de acces exclusiv la un fișier*

*Demo (3): Ilustrarea caracterului *advisory* al blocajelor*

*Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier*

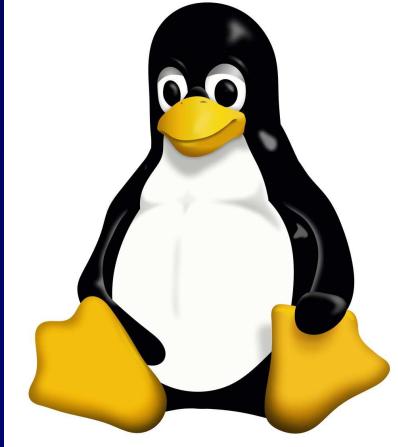
Referințe bibliografice

***Observație importantă:*** a doua versiune a programului demonstrativ (ambele variante, și cea neblockantă, și cea blocantă) nu este optimă:

Practic, cele două procese (*i.e.*, cele două instanțe ale programului executate în paralel) își fac treaba *secvențial*, unul după altul, și nu concurrent, deoarece de abia după ce se termină acel proces care a reușit primul să pună lacăt pe fișier, va putea începe și celălalt proces să-și facă treaba (*i.e.*, parcurgerea fișierului și înlocuirea primului caracter '#' întâlnit).

\* \* \*

Această observație ne sugerează că putem *îmbunătăți timpul total de execuție* permitând celor două procese să se execute într-adevăr concurrent, iar pentru aceasta trebuie să punem lacăt doar pe un singur caracter (și anume pe prima poziție din fișier la care întâlnim caracterul '#') și doar timpul minim necesar pentru a face suprascrierea, în loc să blocăm tot fișierul, încă de la început și până la finalul execuției programului.



## Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier (cont.)

Introducere  
Modul de acces concurrent la fișiere  
Modul de acces exclusiv la fișiere – Blocaje pe fișiere  
Structura de date flock pentru blocaje  
Primitiva fcntl pentru blocaje  
Caracteristici ale blocajelor pe fișiere  
*Demo (2): Un exemplu de acces exclusiv la un fișier*  
*Demo (3): Ilustrarea caracterului *advisory* al blocajelor*  
*Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier*  
Referințe bibliografice

Versiunea a treia, cu blocaj la nivel de caracter și de durată minimală:

*Implementarea acestei optimizări:* programul va trebui să facă următorul lucru – când întâlnește primul caracter '#' în fișier, pune lacăt pe el (*i.e.*, pe exact un caracter) și apoi îl rescrie: a se vedea programul (în varianta blocantă) **access\_v3.c** ([2]).

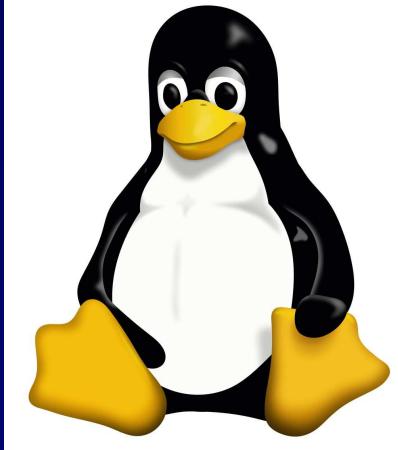
În acest caz, care credeți că va fi conținutul fișierului după terminarea execuției în paralel a două instanțe ale acestei versiuni a programului ?

\* \* \*

*Observație:* ideea de rezolvare aplicată în programul **access\_v3.c** nu este întrutoțul corectă, în sensul că nu se va obține întotdeauna rezultatul scontat, deoarece între momentul depistării primei poziții a unui caracter '#' în fișier și momentul reușitei blocajului există posibilitatea ca acel '#' să fie suprascris de cealaltă instantă executată în paralel !

*Notă:* tocmai pentru a forța apariția unei situații care cauzează producerea unui rezultat nedorit, am introdus în program acel apel **sleep(5)** între punerea blocajului pe caracterul '#' și rescrierea lui.

Cum se poate remedia acest neajuns al programului **access\_v3.c** ? → → →



## Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier (cont.)

Introducere

Modul de acces concurrent la  
fișiere

Modul de acces exclusiv la  
fișiere – Blocaje pe fișiere

Structura de date flock pentru  
blocaje

Primitiva fcntl pentru blocaje  
Caracteristici ale blocajelor pe  
fișiere

*Demo (2): Un exemplu de  
acces exclusiv la un fișier*

*Demo (3): Ilustrarea  
caracterului *advisory* al  
blocajelor*

*Demo (4): Un exemplu de  
acces exclusiv *optimizat* la un  
fișier*

Referințe bibliografice

→ → Acet neajuns al programului `access_v3.c` se poate corecta astfel:

După punerea blocajului, se verifică din nou dacă acel caracter este într-adevăr '#' (pentru că între timp s-ar putea să fi fost rescris de cealaltă instanță executată în paralel) și, dacă nu mai este '#', atunci trebuie scos blocajul și reluată bucla de căutare a primului caracter '#' întâlnit în fișier.

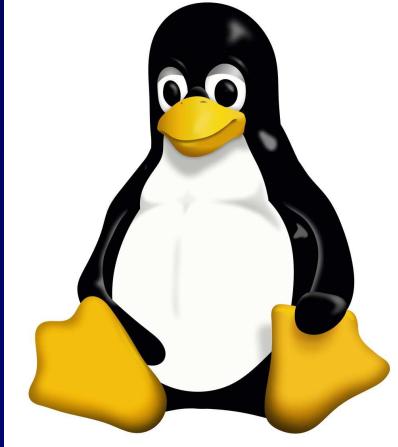
**v4** → *Temă: adăugați această corecție la programul `access_v3.c`.*

\* \* \*

*Rezolvare:* dacă nu reușiți să adăugați singuri această corecție, puteți să vă uitați aici:  
`access_v4.c`.

\* \* \*

*Demo:* pentru explicații mai detaliate despre această variantă mai eficientă a programului demonstrativ, a se vedea [ThirdDemo] prezentat în *suportul de laborator #7*.



## Bibliografie obligatorie

[Introducere](#)

[Modul de acces concurrent la fisiere](#)

[Modul de acces exclusiv la fisiere – Blocaje pe fisiere](#)

[Referințe bibliografice](#)

- [1] Capitolul 3, §3.2 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/lock/>
- [3] POSIX API: `man 2 fcntl`, `man 2 flock` și `man 3 lockf`.
- [4] Documentația comenziilor pentru lacăte: `man 1 flock` și `man 8 lslocks`.

# **PROGRAMARE DE SISTEM ÎN C PENTRU PLATFORMA LINUX (III)**

## **Gestiunea fișierelor, partea a III-a: Fișiere mapate în memorie – primitiva `mmap()`**

Cristian Vidrăscu  
`vidrascu@info.uaic.ro`

Aprilie, 2021

# Sumar

Introducere

[Primitivelor din familia mmap](#)

[Demo: programe cu mmap](#)

[Referințe bibliografice](#)

## Introducere

### Primitivelor din familia mmap

Primitiva mmap

Primitiva munmap

Caracteristici ale mapărilor create cu mmap

Primitiva msync

### Demo: programe cu mmap

Exemplul #1: O mapare “privată”, cu permisiuni *read-only*

Exemplul #2: O mapare “partajată”, cu permisiuni *read&write*

Exemplul #3: O mapare “partajată”, cu scrieri “înafara” mapării

Exemplul #4: O altă mapare “partajată”, pentru crearea unui fișier

Alte exemple de programe cu mapări

## Referințe bibliografice

# Introducere

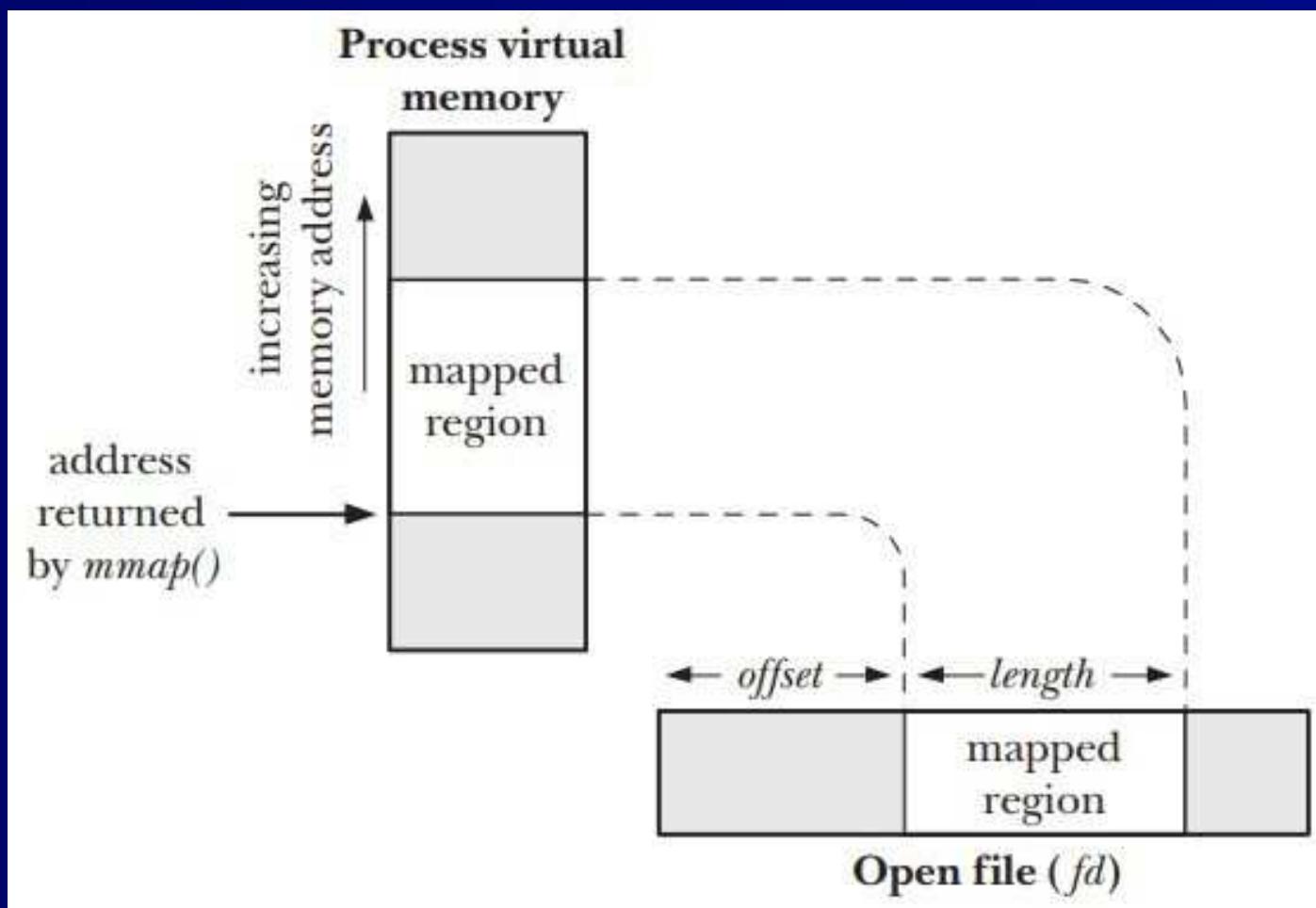
[Introducere](#)

[Primitivele din familia mmap](#)

[Demo: programe cu mmap](#)

[Referințe bibliografice](#)

*Fisier mapat în memorie* – un mecanism prin care (o parte din) conținutul unui fișier este “mapat” în memorie, în spațiul virtual de adrese al procesului ce apelează, în acest scop, primitiva mmap. Prin această “mapare” se realizează practic o corelație directă “octet-la-octet” între o porțiune din spațiul virtual de adrese al procesului și o porțiune a unui fișier de pe disc:



Cu alte cuvinte, paginilor virtuale ce formează respectiva porțiune din spațiul virtual de adrese al procesului, li se asociază drept *backing store* (i.e., “spațiul” pe disc rezervat pentru evacuarea lor din memorie), de către nucleul sistemului de operare, zona de pe disc ce stochează acea porțiune a fișierului de pe disc, în loc de a le rezerva spațiu în *fisierul de swap* al sistemului de operare. *Observație*: veți afla mai multe detalii despre administrarea memoriei virtuale prin *pentinare la cerere* în cursul teoretic #10.

## Introducere (cont.)

[Introducere](#)

[Primitivile din familia mmap](#)

[Demo: programe cu mmap](#)

[Referințe bibliografice](#)

**Atenție:** termenul *fișier mapat în memorie* (în engleză, *memory-mapped file*) se referă la acea porțiune din spațiul virtual de adrese al procesului pentru care s-a stabilit, printr-un apel `mmap`, o corelație directă “octet-la-octet” cu o porțiune a unui fișier de pe disc. Deci nu confundați semnificația acestui termen cu fișierul propriu-zis de pe disc (sau cu porțiunea acestuia de pe disc).

\* \* \*

Prinț-o mapare, putem face accese de citire și scriere direct în memorie asupra fișierului, ca și cum am citi sau scrie diverse variabile din program, fără să mai utilizămapelurile de sistem `read/write` (sau funcțiile de I/O din biblioteca standard de C).

Efectul scrierilor în memorie va fi “propagat” pe disc cu întârziere, atunci când nucleul decide să salveze paginile *dirty* pe disc (e.g., atunci când le selectează drept victime pentru evacuare din memorie).

\* \* \*

Un alt avantaj al acestui mecanism: un anumit fișier poate fi “mapat” simultan în spațiile virtuale de adrese a două (sau mai multor) procese și astfel acestea pot coopera schimbând informații prin modelul de comunicație cu *shared memory*.

Un exemplu simplu de procese cooperante prin modelul de comunicație cu memorie partajată: revedeți şablonul producător-consumator, discutat în cursul teoretic #6.

Alte exemple de procese cooperante prin modelul de comunicație cu memorie partajată: revedeți toate problemele de sincronizare discutate în cursurile teoretice #5 și #6.

# Agenda

Introducere

Primitivele din familia mmap

Primitiva mmap

Primitiva munmap

Caracteristici ale mapărilor  
create cu mmap

Primitiva msync

Demo: programe cu mmap

Referințe bibliografice

## Introducere

### Primitivele din familia mmap

Primitiva mmap

Primitiva munmap

Caracteristici ale mapărilor create cu mmap

Primitiva msync

### Demo: programe cu mmap

Exemplul #1: O mapare “privată”, cu permisiuni *read-only*

Exemplul #2: O mapare “partajată”, cu permisiuni *read&write*

Exemplul #3: O mapare “partajată”, cu scrieri “înafara” mapării

Exemplul #4: O altă mapare “partajată”, pentru crearea unui fișier

Alte exemple de programe cu mapări

## Referințe bibliografice

# Primitiva mmap

Introducere

Primitivele din familia mmap

Primitiva mmap

Primitiva munmap

Caracteristici ale mapărilor  
create cu mmap

Primitiva msync

Demo: programe cu mmap

Referințe bibliografice

- “Maparea” unui fișier în memoria virtuală a unui proces : se realizează cu primitiva mmap.

Interfața funcției mmap ([4]) :

```
void *mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

unde:

- **Valoarea returnată**: adresa de start a mapării create cu succes (i.e., *începutul regiunii mapate în spațiul virtual al procesului apelant*), sau MAP\_FAILED (= (void \*) -1) în caz de eroare.
- *addr* = adresa de start pentru noua mapare ce se va crea în spațiul virtual al procesului apelant. Dacă *addr*=NULL, nucleul va alege în mod convenabil o adresă *page-aligned* (i.e., multiplu de dimensiunea paginii) la care va crea noua mapare. Altfel, valoarea *addr* este folosită de nucleu doar cu rol de *hint* (cu o excepție: în cazul folosirii flag-ului MAP\_FIXED).
- *length* = lungimea noii mapări ce se creează (lungimea trebuie să fie un întreg strict pozitiv).
- *fd* = identifică fișierul (sau un alt obiect, e.g. un *device*) asociat mapării ce se creează.  
*Notă*: descriptorul *fd* poate fi închis **imediat** după apelul mmap, fără invalidarea mapării create.
- *offset* = trebuie să fie un întreg pozitiv multiplu de dimensiunea paginii (!).  
*Notă*: maparea nou creată este **initializată** prin copierea de pe disc a conținutului portiunii din fișierul asociat ce începe de la poziția *offset* și de lungime *length* (cu o excepție: în cazul folosirii flag-ului MAP\_UNINITIALIZED). Iar ca “destinație pe disc” pentru acele modificări efectuate în memorie ce trebuie “propagate” pe disc este folosită aceeași portiune din fișier.

# Primitiva mmap (cont.)

Introducere

Primitivele din familia mmap

Primitiva mmap

Primitiva munmap

Caracteristici ale mapărilor create cu mmap

Primitiva msync

Demo: programe cu mmap

Referințe bibliografice

- “Maparea” unui fișier în memoria virtuală a unui proces – interfața funcției mmap (cont.):
  - *prot* = specifică tipul de protecție al tuturor paginilor de memorie ce formează noua mapare (și trebuie să nu fie în conflict cu modul de deschidere al fișierului). Poate avea ca valoare fie constanta simbolică PROT\_NONE – paginile mapării nou create NU vor putea fi accesate, fie o combinație (*i.e.*, disjuncție logică pe biți) a uneia sau a mai multora dintre constantele:
    - ▲ PROT\_READ – paginile mapării nou create vor putea fi accesate pentru citire;
    - ▲ PROT\_WRITE – paginile mapării nou create vor putea fi accesate pentru scriere;
    - ▲ PROT\_EXEC – paginile mapării nou create vor putea fi accesate pentru execuție.
  - *flags* = o serie de *flag*-uri folosite pentru a determina dacă modificările (screrile) efectuate de proces în paginile mapării vor fi “vizibile” sau nu și în celelalte procese ce mapează același fișier, precum și dacă aceste modificări efectuate în memorie vor fi “propagate” (*i.e.*, flush-uite pe disc) în fișierul propriu-zis stocat pe disc. Poate fi folosită exact una singură dintre valorile:
    - ▲ MAP\_PRIVATE – se creează o mapare “privată” (de tip *copy-on-write*);
    - ▲ MAP\_SHARED – se creează o mapare “partajată”.Aceasta poate fi însotită, eventual, de o combinație (*i.e.*, disjuncție logică pe biți) a altor valori, precum ar fi: MAP\_FIXED, MAP\_LOCKED, MAP\_ANONYMOUS, MAP\_UNINITIALIZED, și.a. Pentru a afla semnificația acestor valori, consultați documentația funcției mmap ([4]).

# Primitiva munmap

Introducere

Primitivele din familia mmap

Primitiva mmap

Primitiva munmap

Caracteristici ale mapărilor create cu mmap

Primitiva msync

Demo: programe cu mmap

Referințe bibliografice

- “*Ștergerea unei mapări din memoria virtuală a unui proces* : se realizează cu primitiva `munmap`.

Interfața funcției `munmap` ([4]):

```
int munmap (void *addr, size_t length)
```

- *addr* = adresa de start pentru maparea din spațiul virtual al procesului apelant ce se va șterge. Adresa specificată trebuie să fie multiplu de dimensiunea paginii.
- *length* = lungimea mapării ce se va șterge.
- **Valoarea returnată:** 0, în caz de succes, și respectiv -1, în caz de eroare.

*Observații:*

- 1) Parametrul *length* nu trebuie să fie neapărat multiplu de dimensiunea paginii, dar se va lua în considerare cel mai mic multiplu de dimensiunea paginii, mai mare sau egal cu *length*, deoarece unitatea de alocare/dealocare în spațiul virtual de adrese al unui proces este pagina.
- 2) Apelul `munmap` “șterge” intervalul de adrese specificat prin parametri (rotunjit la un număr întreg de pagini) din spațiul virtual de adrese al procesului apelant, ceea ce are drept efect faptul că orice acces ulterior la vreo adresă din acel interval va genera o eroare de tip “referință invalidă” (i.e., se generează semnalul SIGSEGV, având ca efect terminarea anormală a procesului, cu un mesaj de eroare “Segmentation fault”).
- 3) Nu este eroare dacă maparea ce se șterge nu reprezintă un interval de adrese corespunzătoare unor pagini mapate, la momentul apelului respectiv, în spațiul virtual de adrese al procesului apelant.
- 4) Mapările create prin `mmap` sunt “sterse” automat la terminarea execuției procesului. Pe de altă parte, închiderea descriptorului de fișier utilizat într-un apel `mmap` nu provoacă “ștergerea” mapării respective.

# Caracteristici ale mapărilor create cu mmap

Introducere

Primitivele din familia mmap

Primitiva mmap

Primitiva munmap

Caracteristici ale mapărilor  
create cu mmap

Primitiva msync

Demo: programe cu mmap

Referințe bibliografice

- 1) *Important:* modul portabil de a crea o mapare este de a specifica *addr* ca 0 (NULL) și de a omite MAP\_FIXED din *flags*. În acest caz, nucleul alege adresa pentru mapare; adresa va fi aleasă într-o manieră adecvată pentru a nu intra în conflict cu nicio mapare existentă și nu va fi 0.
- 2) Semnificația celor două tipuri de mapări (MAP\_PRIVATE vs. MAP\_SHARED) :
  - Pentru o mapare “privată” (de tip *copy-on-write*), scrierile efectuate de procesul ce a creat-o NU vor fi “vizibile” în celelalte procese ce mapează aceeași porțiune de fișier și nici NU vor fi “propagate” în fișierul propriu-zis de pe disc (ci doar, eventual, în *fișierul de swap* al sistemului).
  - Pentru o mapare “partajată”: scrierile efectuate de proces vor fi “vizibile” în celelalte procese ce mapează aceeași porțiune de fișier și vor fi “propagate” în fișierul propriu-zis de pe disc.  
**Important:** momentul “propagării” pe disc a scrierilor în memorie este, implicit, controlat de către nucleu, prin algoritmul de “gestiune” a paginilor *dirty*. Însă, putem forța explicit “propagarea” pe disc a scrierilor în memorie folosind primitiva *msync*.
- 3) Lungimea efectivă (*i.e.*, dimensiunea în octeți) a mapării nou create va fi cel mai mic multiplu de dimensiunea paginii, mai mare sau egal cu *length* (deoarece unitatea de alocare/dealocare în spațiul virtual de adrese al unui proces este pagina). Astfel, dacă parametrul *length* nu este multiplu de dimensiunea paginii, atunci la crearea mapării restul adreselor din ultima pagină a mapării vor fi inițializate cu zero, iar scrierile ulterioare la aceste adrese nu vor da eroare, dar nici NU vor fi “propagate” în fișierul de pe disc.

# Caracteristici ale mapărilor create cu mmap (cont.)

Introducere

Primitivele din familia mmap

Primitiva mmap

Primitiva munmap

Caracteristici ale mapărilor  
create cu mmap

Primitiva msync

Demo: programe cu mmap

Referințe bibliografice

- 4) În urma unui apel `fork`, procesul fiu “moștenește” memoria mapată cu primitiva `mmap` de către părinte, anterior creării fiului. Maparea respectivă va avea în procesul fiu aceleași atribută și aceeași poziune de fișier asociată ca în procesul părinte (mai multe detalii despre aceste aspecte vom vedea în lecția practică următoare, dedicată apelului `fork`).
- 5) Pe anumite arhitecturi hardware (e.g., arhitectura x86/x64) modelul de protecție a acceselor la memorie permite doar valorile *read-only* și *read&write*, dar nu și *write-only*. Cu alte cuvinte, permisiunea `PROT_WRITE` implică automat și permisiunea `PROT_READ`, chiar dacă aceasta din urmă nu este specificată explicit în apelul `mmap`.
- 6) Pe anumite arhitecturi hardware permisiunea `PROT_READ` implică automat și permisiunea `PROT_EXEC` (e.g., *CPU*-uri x86 mai vechi, fără suport pentru **bitul NX**, s.a.), iar pe alte arhitecturi nu implică acest lucru (e.g., arhitectura x64, *CPU*-uri x86 cu suport pentru **bitul NX**, s.a.). Pentru portabilitatea programelor, este recomandat să se specifice explicit permisiunea `PROT_EXEC` în apelul `mmap` ce va crea o mapare din care se intenționează să se execute cod.
- 7) Paginile fizice (din RAM) ce stochează paginile virtuale din care este format spațiul virtual de adrese al unui proces sunt gestionate de nucleu conform schemei de **administrare a memoriei virtuale la cerere** (a se vedea cursurile teoretice #9 și #10). Mai exact, pe durata de viață a procesului, fiecare pagină virtuală a sa trece prin perioade când este rezidentă în memorie (*i.e.*, se află într-o pagină fizică din RAM) și perioade când nu este rezidentă în memorie (*i.e.*, conținutul său este doar pe disc, într-un fișier mapat în memorie sau în *fișierul de swap* al sistemului). Pentru a afla care pagini sunt rezidente și care nu la un moment dat, se poate utiliza primitiva `mincore` ([4]).
- 8) De asemenea, nucleul permite “încuierea” unor pagini virtuale în memorie – astfel, ele vor rămâne rezidente în permanență (până la terminarea procesului sau până la “descuierea” lor), nemaifiind alese drept victimă de algoritmul de *page-swapping*. Pentru a “încui” anumite pagini ale procesului, sau pe toate, se utilizează primitivele `mlock` și, respectiv, `mlockall` ([4]). Iar pentru a le “descuia” se utilizează primitivele `munlock` și, respectiv, `munlockall` ([4]).

# Primitiva msync

Introducere

Primitivele din familia mmap

Primitiva mmap

Primitiva munmap

Caracteristici ale mapărilor  
create cu mmap

Primitiva msync

Demo: programe cu mmap

Referințe bibliografice

- “Sincronizarea” unui fișier cu maparea sa din memoria virtuală a unui proces : se realizează cu primitiva msync. Interfața funcției msync ([4]):

```
int msync (void *addr, size_t length, int flags)
```

- *addr* = adresa de start pentru maparea (din spațiul virtual al procesului apelant) pentru care vrem să “propagăm” pe disc (în portiunea de fișier asociată mapării) scrierile deja efectuate în memorie și încă “nepropagate” (*i.e.*, paginile *dirty* ale mapării respective).
- *length* = lungimea mapării, și a portiunii de fișier de pe disc asociate ei, ce se vor sincroniza.
- *flags* = se inițiază, în mod blocant sau neblocant, un *flushing* pe disc a paginilor *dirty* din acea mapare, prin specificarea exact a uneia dintre valorile:
  - ▲ MS\_SYNC – se cere un *flushing* în mod blocant (*i.e.*, se așteaptă finalizarea scrierii efective pe disc a paginilor *dirty* din acea mapare);
  - ▲ MS\_ASYNC – se cere un *flushing* în mod neblocant (*i.e.*, fără a se aștepta finalizarea scrierii efective pe disc a paginilor *dirty* din acea mapare)

Oricare dintre cele două valori poate fi, eventual, combinată (*i.e.*, disjuncție logică pe biți) cu valoarea MS\_INVALIDATE, prin care se cere invalidarea celorlalte mapări posibil existente ale aceluiași fișier (prin invalidare, acestea se vor actualiza cu modificările survenite pe disc).

- **Valoarea returnată:** 0, în caz de succes, și respectiv -1, în caz de eroare.

# Primitiva msync (cont.)

Introducere

Primitivele din familia mmap

Primitiva mmap

Primitiva munmap

Caracteristici ale mapărilor  
create cu mmap

Primitiva msync

Demo: programe cu mmap

Referințe bibliografice

## ■ “Sincronizarea” unui fișier cu maparea sa din memoria virtuală a unui proces (cont.)

*Observații:*

- 1) Parametrul *addr* este valoarea returnată de apelul mmap ce a creat acea mapare (deci obligatoriu este multiplu de dimensiunea paginii).
- 2) Parametrul *length* este valoarea declarată în apelul mmap respectiv, nefiind obligatoriu să fie multiplu de dimensiunea paginii (a se vedea cele explicate anterior).
- 3) Reformulez o afirmație anterioară (*i.e.*, caracteristica 3) descrisă la mmap): dacă parametrul *length* nu este multiplu de dimensiunea paginii, atunci scrierile în maparea din memorie a acelei porțiuni de fișier, la adrese situate în ultima pagină alocată mapării, “dincolo” de adresa dată de restul împărțirii întregi a valorii *length* la dimensiunea paginii, vor reuși fără a da eroare, dar efectele acestor scrieri nu vor fi “propagate” și în fișierul de pe disc.
- 4) **Important:** apelul munmap nu efectuează și un apel msync implicit (*i.e.*, nu face și *flushing* pentru paginile *dirty* din acel moment).  
Cu alte cuvinte, când ștergeți explicit o mapare fără să o sincronizați mai întâi pe disc, este posibil să “pierdeți” ultimele modificări efectuate în memoria acelei mapări (*i.e.*, acestea nu se vor salva în fișierul de pe disc).

# Agenda

Introducere

Primitivele din familia mmap

Demo: programe cu mmap

Exemplul #1: O mapare “privată”, cu permisiuni *read-only*

Exemplul #2: O mapare “partajată”, cu permisiuni *read&write*

Exemplul #3: O mapare “partajată”, cu scrieri “înafara” mapării

Exemplul #4: O altă mapare “partajată”, pentru crearea unui fișier

Alte exemple de programe cu mapări

Referințe bibliografice

## Introducere

### Primitivele din familia mmap

Primitiva mmap

Primitiva munmap

Caracteristici ale mapărilor create cu mmap

Primitiva msync

### Demo: programe cu mmap

Exemplul #1: O mapare “privată”, cu permisiuni *read-only*

Exemplul #2: O mapare “partajată”, cu permisiuni *read&write*

Exemplul #3: O mapare “partajată”, cu scrieri “înafara” mapării

Exemplul #4: O altă mapare “partajată”, pentru crearea unui fișier

Alte exemple de programe cu mapări

## Referințe bibliografice

# Exemplul #1: O mapare “privată”, cu permisiuni *read-only*

Introducere

Primitivele din familia `mmap`

Demo: programe cu `mmap`

Exemplul #1: O mapare  
“privată”, cu permisiuni  
*read-only*

Exemplul #2: O mapare  
“partajată”, cu permisiuni  
*read&write*

Exemplul #3: O mapare  
“partajată”, cu scrieri “înafara”  
mapării

Exemplul #4: O altă mapare  
“partajată”, pentru crearea  
unui fișier

Alte exemple de programe cu  
mapări

Referințe bibliografice

Un exemplu ce ilustrează folosirea apelului `mmap` pentru realizarea unei mapări de tip “privată”, cu permisiuni de acces *read-only*, a unei porțiuni specificate dintr-un fișier.

A se vedea variantele de program `mmap_ex1a.c` și `mmap_ex1b.c` ([1]).

*Observație:* pentru explicații mai detaliate ale acestor programe și descrierea comportamentului lor la execuție, consultați exemplul [FirstDemo – `mmap_ex1{a,b}`] din suportul online de laborator ([3]).

Ambele variante de program demonstrează *citirea direct din memorie* a informației mapate din fișier, în locul utilizării interfeței clasice de acces I/O la disc (adică fără a folosi apelurile de sistem `read` și `write`, sau funcții de I/O din biblioteci de genul `stdio.h`).

Diferența dintre cele două variante de program constă în modul de tratare a cazului în care utilizatorul programului introduce date de intrare “invalidă” (*i.e.*, pentru acest program, aceasta înseamnă introducerea unui *offset* “ne-aliniat”):

- i) prima variantă abordează modul clasic de tratare, folosit până acum: afișarea unui mesaj de eroare și terminarea execuției programului;
- ii) a doua variantă ilustrează un nou mod de tratare: “corectarea” prin program a datelor de intrare “invalidă” introduse de utilizator și continuarea execuției programului cu aceste date “corectate”.

## Exemplul #2: O mapare “partajată”, cu permisiuni *read&write*

Introducere

Primitivele din familia mmap

Demo: programe cu mmap

Exemplul #1: O mapare “privată”, cu permisiuni *read-only*

Exemplul #2: O mapare “partajată”, cu permisiuni *read&write*

Exemplul #3: O mapare “partajată”, cu scrieri “înafara” mapării

Exemplul #4: O altă mapare “partajată”, pentru crearea unui fișier

Alte exemple de programe cu mapări

Referințe bibliografice

Aici se ilustrează folosirea apelului `mmap` pentru realizarea unei mapări “partajate”, cu permisiuni de acces *read&write*, a unei porțiuni specificate dintr-un fișier.

A se vedea programul `mmap_ex2f.c` ([1]).

*Observație:* pentru explicații mai detaliate ale acestui program și descrierea comportamentului său la execuție, consultați exemplul [SecondDemo – mmap\_ex2f] din suportul online de laborator ([3]).

Acest program demonstrează *citiri și scrieri direct în memorie* a informației mapate din fișier, în locul utilizării interfeței clasice de acces I/O la disc (adică fără a folosi apelurile de sistem `read` și `write`, sau funcții de I/O din biblioteci de genul `stdio.h`), fiind obținut prin adăugarea și de operații de “scriere” la programul din exemplul precedent, plus toate modificările necesare în acest scop.

**Important:** în acest exemplu am ilustrat activitatea iterativă de modificare a unui program (mai precis, a variantei cu “corectarea” datelor de intrare “invalide” a programului din primul exemplu demonstrativ), pentru a obține funcționalitatea dorită în acest exemplu. Cu alte cuvinte, am prezentat un *ciclu iterativ de modificare a variantei curente a programului*, pentru eliminarea *bug-urilor* introduse pe parcursul adăugării funcționalității suplimentare dorite pentru acest al doilea program demonstrativ.

Vă recomand să studiați cu atenție cele 6 versiuni successive ale programului și modul de dezvoltare al lor în manieră iterativă!

## Exemplul #3: O mapare “partajată”, cu scrieri “înafara” mapării

Introducere

Primitivele din familia mmap

Demo: programe cu mmap

Exemplul #1: O mapare “privată”, cu permisiuni *read-only*

Exemplul #2: O mapare “partajată”, cu permisiuni *read&write*

Exemplul #3: O mapare “partajată”, cu scrieri “înafara” mapării

Exemplul #4: O altă mapare “partajată”, pentru crearea unui fișier

Alte exemple de programe cu mapări

Referințe bibliografice

Un exemplu ce ilustrează folosirea apelului `mmap` pentru realizarea unei mapări de tip “partajată”, cu permisiuni de acces *read&write*, a unei porțiuni specificate dintr-un fișier, și care în plus ilustrează ce se întâmplă când scriem la adrese situate “înafara” mapării respective (i.e., la adrese de memorie situate după cea corespunzătoare sfârșitului porțiunii de fișier mapate în memorie).

A se vedea variantele de program `mmap_ex3a.c` și `mmap_ex3b.c` ([1]).

*Observație:* pentru explicații mai detaliate ale acestor programe și descrierea comportamentului lor la execuție, consultați exemplul [ThirdDemo – `mmap_ex3{a,b}`] din suportul online de laborator ([3]).

Acest program demonstrează cazul scrierilor “înafara” mapării respective, precum și efectul lor asupra fișierului de pe disc (i.e., “Are loc actualizarea modificărilor în fișierul de pe disc sau nu?”), fiind obținut prin adăugarea de noi operații de “scriere” la programul din exemplul precedent, la adrese de memorie situate după cea corespunzătoare sfârșitului porțiunii de fișier mapate în memorie.

Cele două variante de program tratează două cazuri diferite: scrieri la adrese situate “înafara” mapării respective, dar totuși în interiorul ultimei pagini alocate mapării, versus scrieri la adrese situate “dincolo de” ultima pagină alocată mapării.

## Exemplul #4: O altă mapare “partajată”, pentru crearea unui fișier

Introducere

Primitivele din familia mmap

Demo: programe cu mmap

Exemplul #1: O mapare “privată”, cu permisiuni *read-only*

Exemplul #2: O mapare “partajată”, cu permisiuni *read&write*

Exemplul #3: O mapare “partajată”, cu scrieri “înafara” mapării

Exemplul #4: O altă mapare “partajată”, pentru crearea unui fișier

Alte exemple de programe cu mapări

Referințe bibliografice

Un exemplu ce ilustrează folosirea apelului `mmap` pentru realizarea unei mapări de tip “partajată”, cu permisiuni de acces *read&write*, a unei porțiuni specificate dintr-un fișier, și în care facem doar scrieri în fișier, și nu actualizări de tipul citire+scriere.

A se vedea programul `mmap_ex4c.c` ([1]).

*Observație:* pentru explicații mai detaliate ale acestui program și descrierea comportamentului său la execuție, consultați exemplul [FourthDemo – `mmap_ex4c`] din suportul online de laborator ([3]).

Acest program demonstrează doar operații de scriere (fără citire prealabilă), direct în memorie, a conținutului nou pentru acel fișier, urmată de observarea salvării în fișierul de pe disc a informațiilor scrise în memorie. Practic, urmărim să creăm fișierul, cu un anumit conținut (nou) ; nu ne interesează conținutul vechi, în caz că acel fișier exista cumva dinainte.

**Important:** și în acest exemplu am ilustrat activitatea iterativă de modificare a unui program, pentru a obține versiunea de program cu funcționalitatea dorită în acest exemplu. Cu alte cuvinte, am prezentat iarăși un *ciclu iterativ de modificare a variantei curente a programului*, pentru eliminarea *bug-urilor* introduse pe parcursul adăugării funcționalității dorite pentru acest al 4-lea program demonstrativ. Vă recomand să studiați cu atenție cele 3 versiuni succesive ale programului și modul de dezvoltare a lor în manieră iterativă!

# Alte exemple de programe cu mapări

Introducere

Primitivele din familia mmap

Demo: programe cu mmap

Exemplul #1: O mapare “privată”, cu permisiuni *read-only*

Exemplul #2: O mapare “partajată”, cu permisiuni *read&write*

Exemplul #3: O mapare “partajată”, cu scrieri “înafara” mapării

Exemplul #4: O altă mapare “partajată”, pentru crearea unui fișier

Alte exemple de programe cu mapări

Referințe bibliografice

*Demo:* exercițiul rezolvat [txt2bin\_write-mapped-file], prezentat în suportul online de laborator ([3]), ilustrează un exemplu de program care citește de la tastatură o secvență de numere întregi, introduse prin reprezentarea lor textuală, și le scrie în memorie (deci în format binar), în maparea corespunzătoare fișierului de ieșire specificat.

(*Notă:* practic, acest program este o reimplementare, utilizând o mapare în locul funcțiilor clasice de I/O, a programului demonstrativ [txt2bin\_write-file], prezentat în suportul online al **laboratorului #7**.)

\* \* \*

*Demo:* exercițiul rezolvat [bin2txt\_read-mapped-file], prezentat în suportul online de laborator ([3]), ilustrează un exemplu de program care afișează pe ecran reprezentarea textuală a numerelor citite prin inițializarea mapării în memorie a unui fișier de date specificat de pe disc, fișier ce conține o secvență numere stocate în format binar.

(*Notă:* practic, acest program este o reimplementare, utilizând o mapare în locul funcțiilor clasice de I/O, a programului demonstrativ [bin2txt\_read-file], prezentat în suportul online al **laboratorului #7**.)

\* \* \*

*Demo:* exercițiul rezolvat [Demo 'data race' \_shmemp #1 : ...], prezentat în suportul online de laborator ([3]), ilustrează şablonul de cooperare Producător-Consumator ce a fost prezentat în cursul teoretic #6, particularizat pe un exemplu concret de informație ce este produsă de un proces și consumată de celălalt proces. Se utilizează un fișier mapat în memoria ambelor programe pentru a obține o zonă de memorie partajată prin intermediul căreia se transmite informația de la procesul producător la cel consumator și, în plus, nu se folosește niciun mecanism de sincronizare a citirilor și scrierilor în regiunea de memorie partajată, ceea ce are ca posibil efect citiri de informații “incorecte”.

Astfel, acest exemplu mai ilustrează și fenomenul de *data race* ce a fost prezentat la începutul cursului teoretic #5, având rolul de a vă atrage atenția asupra nevoii de folosire a unor tehnici specifice pentru sincronizarea execuției programelor, în scopul “reparării” programelor ca să nu (mai) “suferă” de acest fenomen nedorit.

# Bibliografie obligatorie

Introducere

Primitivele din familia mmap

Demo: programe cu mmap

Referinte bibliografice

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

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

[2] Capitolul 49 din cartea “The Linux Programming Interface : A Linux and UNIX System Programming Handbook”, autor M. Kerrisk, editura No Starch Press, 2010.

Această carte este accesibilă, în format PDF, din pagina disciplinei “Sisteme de operare”:

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

[3] Suportul online de laborator asociat acestei prezentări:

- [https://profs.info.uaic.ro/~vidrascu/S0/labs/suport\\_lab9.html](https://profs.info.uaic.ro/~vidrascu/S0/labs/suport_lab9.html)

[4] POSIX API: `man 2 mmap`, `man 2 munmap`, `man 2 msync`, `man 2 mprotect`,  
`man 2 mincore`, `man 2 mlock / mlockall`, `man 2 munlock / munlockall`.

# Programare concurentă în C (III) :

*Gestiunea proceselor, partea I-a:  
Crearea și sincronizarea proceselor – primitivele  
**fork( )** și **wait( )***

Cristian Vidrăscu

[vidrascu@info.uaic.ro](mailto:vidrascu@info.uaic.ro)

# Sumar

- Notiuni generale despre procese
- Primitive utile referitoare la procese
- Crearea proceselor – primitiva `fork()`
- Terminarea proceselor
- Sincronizarea proceselor – primitiva `wait()`
- Sabloane de cooperare și sincronizare

# Notiuni generale despre procese

*Program* = un *fisier executabil* (evident, obținut prin compilare dintr-un fisier sursă), aflat pe un suport de memorare extern (e.g. *harddisk*).

## DEFINIȚIE:

“Un *proces* este un program aflat în curs de execuție.”

Mai precis, un *proces* este o instantă de execuție a unui program, fiind caracterizat de: o durată de timp (*i.e.* perioada de timp în care se execută acel program), o zonă de memorie alocată (zona de cod + zona de date + stiva), timp procesor alocat, și.a.

# Notiuni generale despre procese (cont.)

Nucleul sistemului de operare păstrează evidența proceselor din sistem prin intermediul unei **tabele a proceselor active**. Aceasta conține câte o intrare pentru fiecare proces existent în sistem, intrare ce conține o serie de informații despre acel proces:

- PID-ul = identificatorul de proces – este un întreg pozitiv, de tipul `pid_t` (tip definit în fișierul `header sys/types.h`)
- PPID-ul : PID-ul procesului *părinte*
- terminalul de control
- UID-ul utilizatorului proprietar *real* al procesului (proprietarul real este utilizatorul care l-a lansat în execuție)
- GID-ul grupului proprietar *real* al procesului

# Notiuni generale despre procese (cont.)

- EUID-ul și EGID-ul = UID-ul și GID-ul proprietarului **efectiv**

(adică acel utilizator ce determină drepturile procesului de acces la resurse)

*Notă:* dacă bitul *setuid* este 1, atunci, pe toată durata de execuție a fișierului respectiv, proprietarul efectiv al procesului va fi proprietarul fișierului, și nu utilizatorul care îl execută; similar pentru bitul *setgid*.

- *starea procesului* – poate fi una dintre următoarele:

- *ready* = pregătit pentru execuție
- *running* = în execuție
- *waiting* = în așteptarea producerii unui eveniment (e.g., terminare op. I/O)
- *finished* = terminare normală

- linia de comandă (parametrii cu care a fost lansat în execuție)

- variabilele de mediu transmise de către părinte

- s.a.

# Primitive utile referitoare la procese (1/3)

- Primitive pentru aflarea PID-urilor unui proces și a parintelui acestuia: `getpid`, `getppid`. Interfețele acestor funcții:  
`pid_t getpid(void);`  
`pid_t getppid(void);`
- Primitive pentru aflarea ID-urilor proprietarului unui proces și a grupului acestuia: `getuid`, `getgid` și `geteuid`, `getegid`. Interfețele acestor funcții:

```
uid_t getuid(void);  
gid_t getgid(void);  
uid_t geteuid(void);  
gid_t getegid(void);
```

# Primitive utile referitoare la procese (2/3)

- Primitive de suspendare a execuției pe o durată de timp specificată: `sleep` și `usleep`. Interfețele acestor funcții:

```
unsigned int sleep(unsigned int nr_secunde);  
int usleep(useconds_t nr_microsecunde);
```

Exemplu: a se vedea programul demonstrativ `info_ex.c`

*Notă:* În standardul POSIX s-a introdus și primitiva `nanosleep`, cu o precizie de ordinul nanosecunde și mai eficientă.

Interfața acestei funcții:

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

*Atenție:* durata efectivă de pauză în execuția programului poate depăși valoarea specificată în aceste apele (!). Măsurarea timpului scurs depinde (și) de precizia ceasului utilizat, i.e. de suportul hardware oferit; pentru detalii citiți `man 7 time`.

# Primitive utile referitoare la procese (3/3)

- Primitiva de terminare normală a execuției unui proces: `exit`.  
Interfața acestei funcții: `void exit(int status);`  
**Efect:** procesul apelant își încheie execuția normal, iar valoarea `status` este “trunchiată” (i.e., `status & 0xFF`) și returnată drept `cod_retur` către părintele procesului respectiv.
- Primitiva de terminare anormală a execuției unui proces: `abort`.  
Interfața acestei funcții: `void abort(void);`  
**Efect:** deblochează și apoi își livrează semnalul SIGABRT, ceea ce cauzează terminarea anormală a procesului.
- Funcția `system` permite lansarea de comenzi UNIX dintr-un program C, 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.

# Crearea proceselor – primitiva fork

Singura modalitate de creare a proceselor în UNIX/Linux este cu ajutorul apelului sistem `fork`. Prototipul lui este următorul:

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

**Efect:** prin acest apel se creează o copie a procesului *apelant*, și ambele procese – cel nou creat și cel apelant – își vor continua execuția cu următoarea instrucțiune (din programul executabil) ce urmează după apelul funcției `fork`.

Singura diferență dintre procese va fi valoarea returnată de funcția `fork`, precum și, bineînțeles, PID-urile proceselor.

Procesul apelant va fi *părintele* procesului nou creat, iar acesta va fi *fiul* procesului apelant (mai exact, unul dintre procesele fii ai acestuia).

# Crearea proceselor – primitiva fork

## *Observație importantă:*

Datorită acestei operații de “clonare”, imediat după apelul `fork` procesul fiu va avea aceleasi valori ale variabilelor din program și aceleasi fișiere deschise ca și procesul părinte. Mai departe însă, *fiecare proces va lucra pe propria sa zonă de memorie.*

Deci, dacă fiul modifică valoarea unei variabile, această modificare nu va fi vizibilă și în procesul tată (și nici invers).

În concluzie, nu avem memorie partajată (*shared memory*) între procesele părinte și fiu.

*Observație:* În Linux, apelul de sistem `fork` este implementat folosind pagini COW (*copy-on-write*), ceea ce optimizează timpul de creare a fiului, util mai ales când fiul apelează imediat o funcție `exec`.

# Crearea proceselor – primitiva fork

## Valoarea returnată:

Apelul `fork` returnează valoarea `-1`, în caz de eroare (dacă nu s-a putut crea un nou proces), iar în caz de succes, returnează respectiv următoarele valori în cele două procese, tată și fiu:

- $n$ , în procesul tată, unde  $n$  este PID-ul noului proces creat
- 0, în procesul fiu

Pe baza acestei valori returnate, ce diferă în cele două procese, se poate ramifica execuția astfel încât fiul să execute altceva decât tatăl.

Exemplu: a se vedea programul demonstrativ `fork_ex.c`

# Crearea proceselor – primitiva fork

*Observații:*

- PID-ul unui nou proces nu poate fi niciodată 0, deoarece procesul cu PID-ul 0 nu este fiul nici unui proces, ci este rădăcina arborelui proceselor, și este singurul proces din sistem ce nu se creează prin apelul `fork`, ci este creat atunci când se *boot-ează* sistemul UNIX/Linux pe calculatorul respectiv.
- Procesul nou creat poate afla PID-ul tatălui cu ajutorul primitivei `getppid`, pe când procesul părinte nu poate afla PID-ul noului proces creat, fiu al său, prin altă manieră decât prin valoarea returnată de apelul `fork`.

(Notă: nu s-a creat o primitivă pentru aflarea PID-ului fiului deoarece, spre deosebire de părinte, fiul unui proces nu este unic – un proces poate avea zero, unul, sau mai mulți fii la un moment dat.)

# Terminarea proceselor

Procesele se pot termina în două moduri:

- **terminarea normală**: se petrece în momentul întâlnirii în program a apelului primitivei `exit` (sau la sfârșitul funcției `main`).

Ca efect, procesul este trecut în starea *finished*, se închid fișierele deschise (și se salvează pe disc conținutul *buffer*-elor folosite), se dealocă zonele de memorie alocate procesului respectiv, și.a.m.d.

Codul de terminare este salvat în intrarea corespunzătoare procesului respectiv din tabela proceselor; intrarea respectivă nu este dealocată (“ștearsă”) imediat din tabelă, astfel încât codul de terminare a procesului respectiv să poată fi furnizat procesului părinte la cererea acestuia.

- **terminarea anomală**: se petrece în momentul primirii unui semnal de un anumit tip (e.g., semnalul generat cu un apel `abort`).

*Notă*: nu chiar toate tipurile de semnale cauzează terminarea procesului.

Și în acest caz se dealocă zonele de memorie ocupate de procesul respectiv, și se păstrează doar intrarea sa din tabela proceselor până când părintele său va cere codul de terminare (reprezentat în acest caz de numărul semnalului ce a cauzat terminarea anomală).

# Sincronizarea proceselor

În programarea concurrentă există noțiunea de *punct de sincronizare* a două procese: este un punct din care cele două procese au o execuție simultană (*i.e.* este un punct de așteptare reciprocă). Punctul de sincronizare nu este o noțiune dinamică, ci una statică (o noțiune fixă): este precizat în algoritm (*i.e.*, program) locul unde se găsește acest punct de sincronizare.

Primitiva `fork` este un exemplu de punct de sincronizare: cele două procese – procesul apelant al primitivei `fork` și procesul nou creat de apelul acestei primitive – își continuă execuția simultan din acest punct (*i.e.* punctul din program în care apare apelul funcției `fork`).

# Primitiva `wait`

Un alt exemplu de sincronizare, des întâlnită în practică:

Procesul părinte poate avea nevoie de valoarea de terminare returnată de procesul fiu.

Pentru a realiza această facilitate, trebuie stabilit un punct de sincronizare între sfârșitul programului fiu și punctul din programul părinte în care este nevoie de acea valoare, și apoi trebuie transferată acea valoare de la procesul fiu la procesul părinte.

# Primitiva `wait` (cont.)

Apelul sistem `wait` este utilizat pentru a aștepta un proces fiu să-și termine execuția.

Interfața acestei funcții:

```
pid_t wait(int* stat_loc);
```

**Efect:** apelul funcției `wait` suspendă execuția procesului apelant până în momentul în care unul dintre fiii aceluiași proces (oricare dintre ei), se termină sau este stopat (*i.e.*, terminat anormal printr-un semnal). Dacă există deja vreun fiu care s-a terminat sau a fost stopat, atunci funcția `wait` returnează imediat.

# Primitiva `wait` (cont.)

## Valoarea returnată:

Apelul `wait` returnează ca valoare PID-ul acelui proces fiu, iar în locația referită de pointerul `stat_loc` este salvată următoarea valoare:

- codul de terminare a acelui proces fiu (și anume, în octetul *high* al acelui `int`), dacă `wait` returnează deoarece un fiu s-a terminat normal
- codul semnalului (și anume, în octetul *low* al acelui `int`), dacă funcția `wait` returnează deoarece un fiu a fost stopat de un semnal

*Notă:* pentru a inspecta valoarea stocată în `*stat_loc` pot fi folosite macro-urile [WIFEXITED](#), [WEXITSTATUS](#), [WIFSIGNALED](#), [WTERMSIG](#), [WIFSTOPPED](#), [WSTOPSIG](#) și.a.

Dacă procesul apelant nu are procese fii, atunci funcția `wait` returnează valoarea `-1`, iar variabila `errno` este setată în mod corespunzător pentru a indica eroarea (*i.e.*, `ECHILD`).

# Primitiva `wait` (cont.)

*Observație:* dacă procesul părinte se termină înaintea vreunui proces fiu, atunci acestui fiu îi se va atribui ca părinte procesul `init` (ce are `PID-ul 1`), iar acest lucru se face pentru toate procesele fii neterminate în momentul terminării părintelui lor. Iar dacă un proces se termină înaintea părintelui său, atunci el devine *zombie*.

Mai există o primitivă, cu numele `waitpid`, care va aștepta terminarea fie unui anumit fiu (specificat prin `PID-ul său` dat ca argument), fie a oricărui fiu (dacă se specifică `-1` drept `PID`), și are un argument suplimentar care influențează modul de așteptare (e.g. opțiunea `WNOHANG` e utilă pentru a testa fără așteptare existența vreunui fiu deja terminat).

Exemple: a se vedea programele demonstrative `wait_ex1.c` , `wait_ex2.c` și `wait_ex3.c`

# Şabloane de cooperare și sincronizare

- Şablonul de cooperare '**Supervisor/workers**' (aka '**Master/slaves**'):  
Este un şablon de calcul paralel aplicabil atunci când avem o problemă complexă a cărei rezolvare se poate "diviza" în mai multe sub-probleme **ce pot fi apoi rezolvate, în paralel, independent una de alta**, iar la final rezultatele parțiale obținute pot fi "aggregate" pentru a obține rezultatul final al problemei inițiale.
- Şablonul de sincronizare '**Token ring**':  
Este un şablon de sincronizare care surprinde următoarea situație: avem un număr oarecare  $p$  de procese, fiecare având de executat, în mod repetitiv, câte o acțiune specifică  $A_i$ , cu  $i = 1, \dots, p$ , și se cere sincronizarea execuției lor în paralel, astfel încât ordinea de execuție (i.e., *trace-ul*) să fie precis următoarea:  $A_1, A_2, \dots, A_p$ , repetată de un anumit număr de ori.
- Alte şabloane: '**Producer-Consumer**', '**CREW**', s.a. (pentru detalii, recitați cursul teoretic #6).

# Bibliografie obligatorie

Cap.4, §4.1, §4.2 și §4.3 din manualul, în format PDF, accesibil din pagina disciplinei “Sisteme de operare”:

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

Programele demonstrative amintite pe parcursul acestei prezentări pot fi descărcate de la adresele următoare:

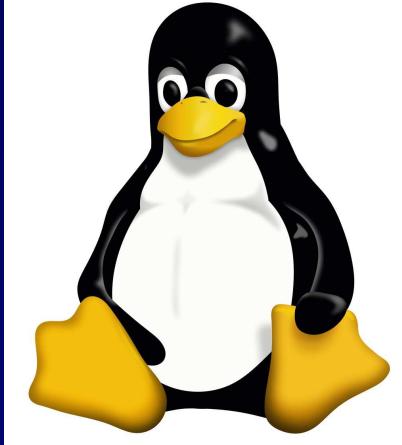
- <https://profs.info.uaic.ro/~vidrascu/SO/cursuri/C-programs/fork/>
- <https://profs.info.uaic.ro/~vidrascu/SO/cursuri/C-programs/wait/>

# PROGRAMARE DE SISTEM ÎN C PENTRU PLATFORMA LINUX (V)

## Gestiunea proceselor, partea a II-a: Reacoperirea proceselor – primitivele exec()

Cristian Vidrăscu  
[vidrascu@info.uaic.ro](mailto: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

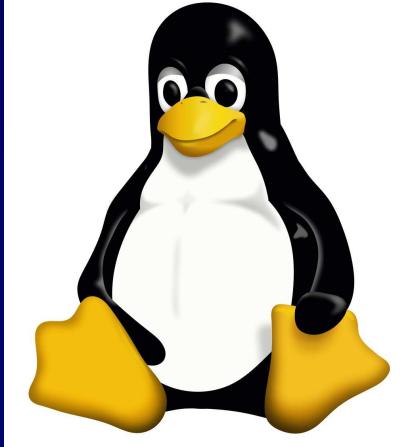
Exemplul #3: Reacoperirea unui program cu fișiere deschise

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](#)

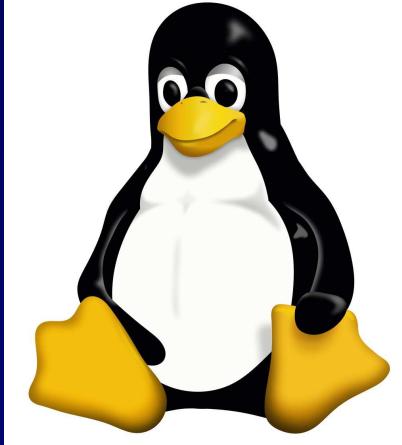
[Referințe bibliografice](#)

După cum am văzut în lectia 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](#)

Primitivile din familia exec  
Caracteristicile procesului  
după exec

[Demo: programe cu exec](#)

[Referințe bibliografice](#)

## Introducere

### **Reacoperirea proceselor**

Primitivile din familia exec

Caracteristicile procesului după exec

### **Demo: programe cu exec**

Exemplul #1: Reacoperirea unui program cu alt program

Exemplul #2: Reacoperirea recursivă

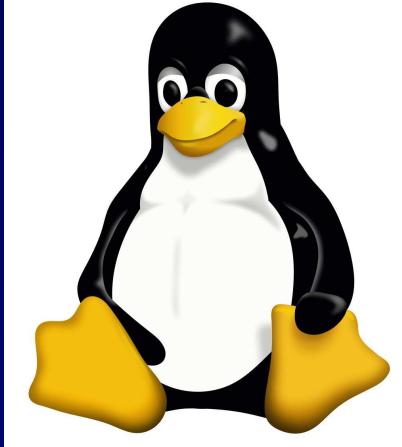
Exemplul #3: Reacoperirea unui program cu fisiere deschise

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

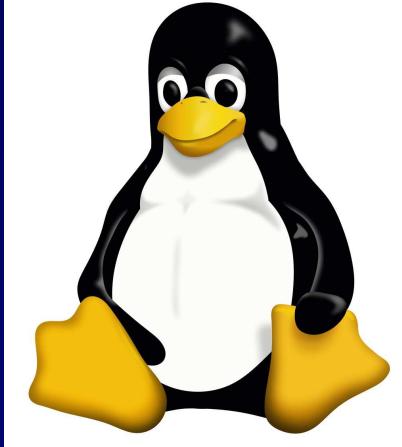
Familia de primitive exec “înlocuiește” programul rulat în cadrul procesului apelant cu un alt program, specificat prin numele fisierului executabil asociat, transmis ca argument al apelului exec.

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)



Introducere

Reacoperirea proceselor

Primitivele din familia exec

Caracteristicile procesului  
după exec

Demo: programe cu exec

Referințe bibliografice

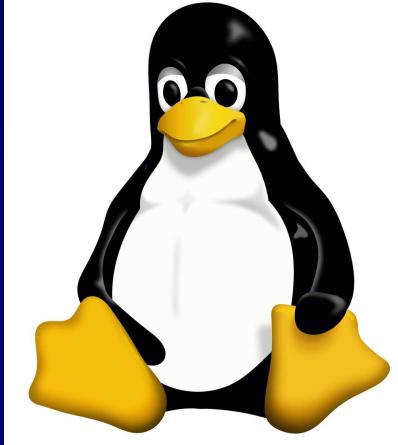
## Primitivele din familia exec (cont.)

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 afișat* (de comenzi precum `ps`, `pstree`, `w`, și.a.) al procesului “transformat” (*i.e.*, procesul rezultat în urma reacoperirii cu noul program).
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:

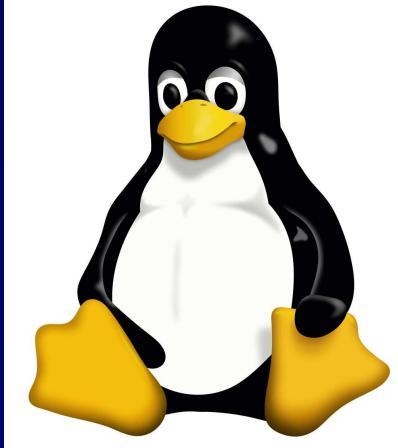
- `int execle(char* ref, char* argv0, ..., char* argvN, char* env[] )`
- `int execve(char* ref, char* argv[], char* env[] )`
  - *env* = parametru ce permite transmiterea unui *environment* (*i.e.*, un set de variabile de mediu) către noul program ce va reacoperi procesul apelant
  - celelalte argumente 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 consultă `man 2 execve`), iar celelalte funcții sunt de fapt niște *wrapper*e definite în STANDARD C LIBRARY, ce apelează la rândul lor funcția `execve` (a se consultă `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 `execlp` și `execvp`, ce au interfețele următoare:

- `int execlp(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. Si în acest caz au loc restricțiile din observațiile 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 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 reținut:* argumentul *ref* specifică *numele real* al fișierului executabil (sau scriptului) ce se va încărca și executa, iar *argv0*, respectiv *argv[0]*, specifică *numele afișat* (de comenzi precum *ps*, *pstree*, *w*, și.a.) al procesului “transformat” (i.e., procesul rezultat în urma reacoperirii programului apelant cu noul program).



## Caracteristicile procesului după exec

Introducere

[Reacoperirea proceselor](#)

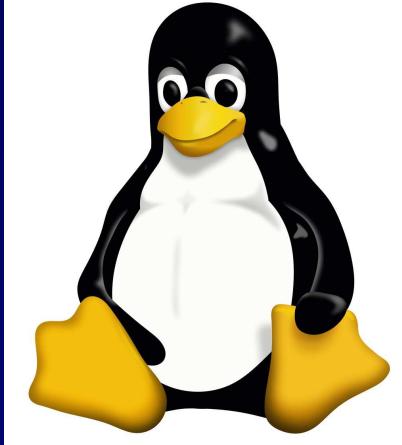
Primitivele din familia exec  
Caracteristicile procesului  
după exec

[Demo: programe cu exec](#)

[Referințe bibliografice](#)

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, și.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.
Handler-ele de semnale	Sunt reinstalate <i>handler-ele</i> implicate pentru acele semnale ce erau “corupte” (i.e., 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 <i>fcntl</i> , atunci descriptorul respectiv este închis la exec (ceilalți descriptori de fișiere rămân deschisi).



# 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**

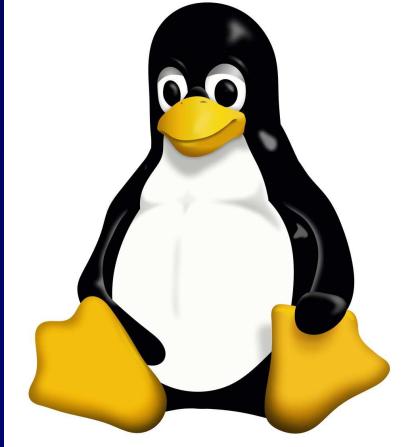
**Exemplul #3: Reacoperirea unui program cu fișiere deschise**

**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

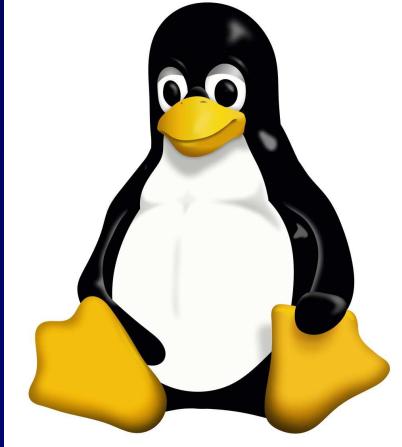
[Referințe bibliografice](#)

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

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

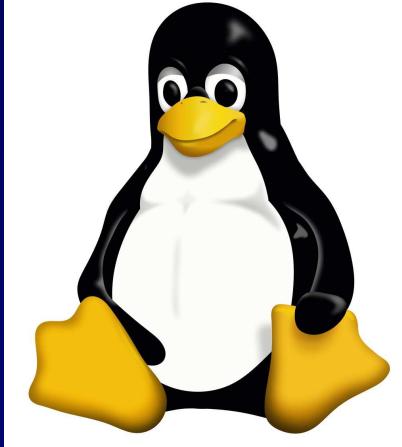
Exemplul #5: Reacoperirea unui program cu un script

Alte programe demonstrative

Referințe bibliografice

Un 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

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 apărea în acel fișier și nu pe ecran.

Introducere

Reacoperirea proceselor

Demo: programe cu exec

Exemplul #1: Reacoperirea unui program cu alt program

Exemplul #2: Reacoperirea recursivă

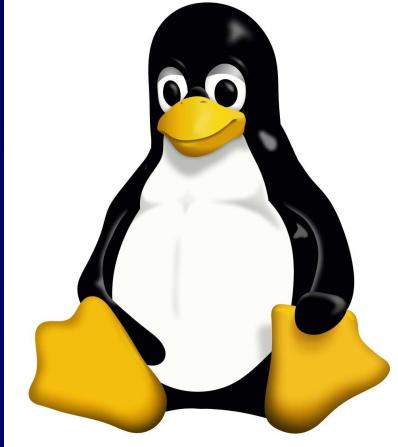
Exemplul #3: Reacoperirea unui program cu fișiere deschise

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 (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

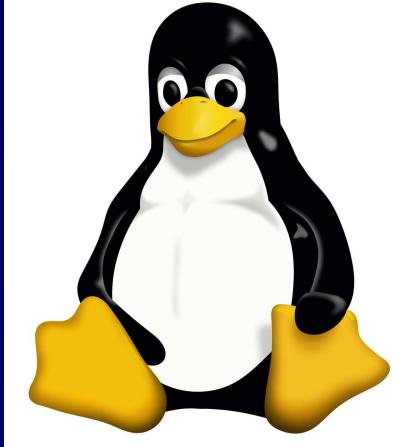
[Referințe bibliografice](#)

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

*Notă:* în mod normal *buffer-ul* este scris în fisier abia în momentul când s-a umplut, sau la întâlnirea caracterului '\n'. Dar se poate forța scrierea *buffer-ului* în fisier 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

Pe lângă primitiva dup, mai există o primitivă, cu numele dup2, utilă pentru duplicarea unui descriptor de fișier ([5]). Cu ajutorul lor se poate realiza redirectarea fluxurilor standard de I/O, aşa 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`).

Introducere

[Reacoperirea proceselor](#)

[Demo: programe cu exec](#)

Exemplul #1: Reacoperirea unui program cu alt program

Exemplul #2: Reacoperirea recursivă

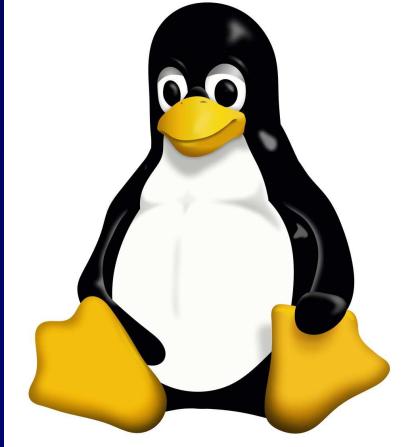
Exemplul #3: Reacoperirea unui program cu fișiere deschise

Exemplul #4: Redirectarea fluxului stdout

Exemplul #5: Reacoperirea unui program cu un script

Alte programe demonstrative

[Referințe bibliografice](#)



## Exemplul #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

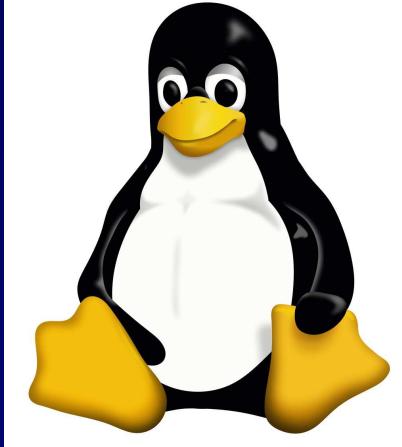
Alte programe demonstrative

[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: Redirectarea fluxului stdout

Exemplul #5: Reacoperirea unui program cu un script

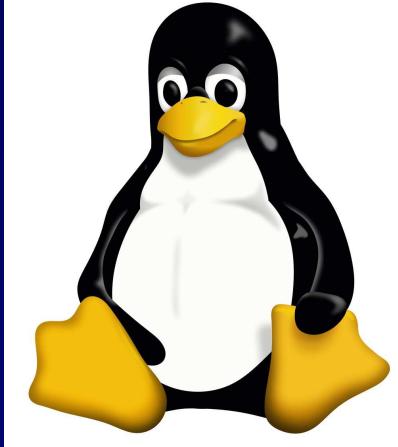
Alte programe demonstrative

Referințe bibliografice

lată un exemplu de program care *apelează prin exec o comandă* oarecare, însotită de o listă de argumente, iar apoi prelucrează statusul execuției comenzi 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); }
    /* În 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 în 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 eșec (cod: %d).\n",WEXITSTATUS(ret));
        }
    }
    else printf("Comanda ls a fost terminata fortat (semnal: %d).\n",WTERMSIG(ret));
    return 0;
}
```

*Notă:* acest program este disponibil în întregime în exemplul [Exec command #1: ls], prezentat în suportul de laborator #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: Redirectarea 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 de laborator #11, ilustrează alte două programe care apelează prin exec o comandă simplă și, respectiv, o secvență de două comenzi simple, însotite fiecare de câte o listă de argumente. Iar la final, fiecare program prelucrează statusul execuției comenzi respective (*i.e.*, succes vs. eșec).

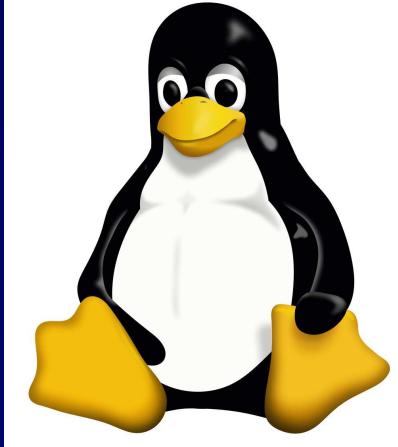
*Observație:* funcția system permite lansarea de comenzi uzuale de UNIX/Linux dintr-un program C, printr-un apel de forma: `system(comanda);`

**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 de laborator #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 şablonului de cooperare 'Supervisor / workers' pentru realizarea unui calcul paralel.



## Bibliografie obligatorie

[Introducere](#)

[Reacoperirea proceselor](#)

[Demo: programe cu exec](#)

[Referinte bibliografice](#)

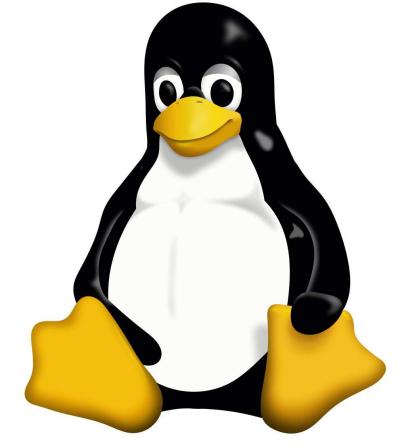
- [1] Capitolul 4, §4.4 din cartea “Sisteme de operare – manual pentru ID”, autor C. Vidrașcu, editura UAIC, 2006. Acest manual este accesibil, în format PDF, din pagina disciplinei “Sisteme de operare”:
- <https://profs.info.uaic.ro/~vidrascu/S0/books/ManualID-S0.pdf>
- [2] Programele demonstrative amintite pe parcursul acestei prezentări pot fi descărcate de la adresa:
- <https://profs.info.uaic.ro/~vidrascu/S0/cursuri/C-programs/exec/>
- [3] Suportul online de laborator asociat acestei prezentări:
- [https://profs.info.uaic.ro/~vidrascu/S0/labs/suport\\_lab11.html](https://profs.info.uaic.ro/~vidrascu/S0/labs/suport_lab11.html)
- [4] Capitolul 27 din cartea “The Linux Programming Interface : A Linux and UNIX System Programming Handbook”, autor M. Kerrisk, editura No Starch Press, 2010.
- Această carte este accesibilă, în format PDF, din pagina disciplinei “Sisteme de operare”:
- <https://profs.info.uaic.ro/~vidrascu/S0/books/TLPI1.pdf>
- [5] POSIX API: `man 2 execve`, `man 3 exec`, `man 2 dup`.

# **PROGRAMARE DE SISTEM ÎN C PENTRU PLATFORMA LINUX (V)**

## **Comunicația inter-procese: Canale de comunicație anonte și cu nume**

Cristian Vidrașcu  
[vidrascu@info.uaic.ro](mailto:vidrascu@info.uaic.ro)

Mai, 2021



# Sumar

Introducere

Canale anoneme

Canale cu nume (fifo)

Caracteristici comune pentru ambele tipuri de canale

Şabloane de comunicaţie între procese

Aplicaţii ale canalelor de comunicaţie

Referinţe bibliografice

Introducere

## Canale anone

Crearea lor, cu primitiva pipe

Modul de utilizare a unui canal anonim

*Demo:* exemple de comunicaţie între două procese

## Canale cu nume (fifo)

Crearea lor, cu primitiva mkfifo

Modul de utilizare a unui canal cu nume

Despre persistenţa informaţiei dintr-un fişier fifo

Deosebiri ale canalelor cu nume faţă de cele anone

## Caracteristici comune pentru ambele tipuri de canale

Caracteristici şi restricţii ale canalelor de comunicaţie

Comportamentul implicit, de tip blocant

Comportamentul de tip neblocant

## Şabloane de comunicaţie între procese

Clasificarea şabloanelor de comunicaţie inter-procese

Şablonul de comunicaţie *unul-la-unul*

Şablonul de comunicaţie *unul-la-mulţi*

Şablonul de comunicaţie *mulţi-la-unul*

Şablonul de comunicaţie *mulţi-la-mulţi*

## Aplicaţii ale canalelor de comunicaţie

Aplicaţia #1: implementarea unui semafor

Aplicaţia #2: implementarea unei aplicaţii client/server

## Referinţe bibliografice



# Introducere

Introducere

Canale anonte

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

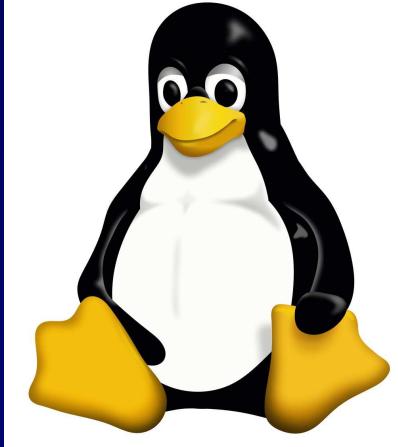
Şabioane de comunicaţie între  
procese

Aplicaţii ale canalelor de  
comunicaţie

Referinţe bibliografice

## Tipuri de comunicaţie între procese:

- **comunicaţia prin memorie partajată** (“*shared-memory communication*”)
  - e.g. prin fişiere mapate în memorie, sau mapări anonime și cu nume, s.a.
- **comunicaţia prin schimb de mesaje** (“*message-passing communication*”)
  - *comunicaţie locală*
    - ▲ canale anonime (numite, uneori, și canale interne)
    - ▲ canale cu nume, i.e. fișiere fifo (numite, uneori, și canale externe)
  - *comunicaţie la distanţă*
    - ▲ *socket*-uri



## Introducere (cont.)

Introducere

Canale anone

Canale cu nume (*fifo*)

Caracteristici comune pentru ambele tipuri de canale

Şabloane de comunicaţie între procese

Aplicaţii ale canalelor de comunicatie

Referinţe bibliografice

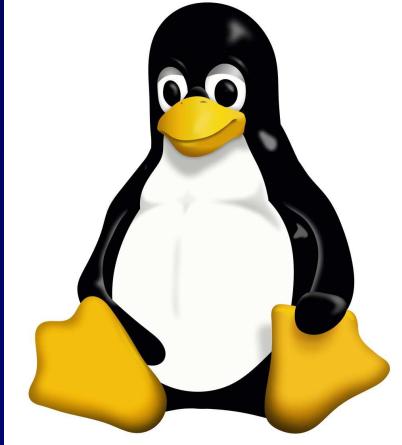
Un *canal de comunicaţie* UNIX, sau *pipe*, este o “conductă” prin care pe la un capăt se scriu mesajele (ce constau în secvenţe de octeţi), iar pe la celălalt capăt acestea sunt citite (cu extracţia lor din canal) – deci practic se comportă ca o structură de tip coadă, adică o listă FIFO (*First-In,First-Out*).

*Notă:* de fapt, un *pipe* chiar este implementat de nucleul UNIX/Linux ca o listă FIFO, cu o capacitate constantă, gestionată în *kernel-space*.

**Rolul unui canal:** o asemenea “conductă” FIFO poate fi folosită pentru comunicare de către două (sau mai multe) procese, pentru a transmite date de la unul la altul (!).

Canalele de comunicaţie UNIX se împart în două subcategorii:

- ***canale anone*:** aceste “conducte” sunt create în memoria internă a sistemului UNIX respectiv, fără niciun nume asociat lor în sistemul de fișiere;
- ***canale cu nume*:** aceste “conducte” sunt create tot în memoria internă a sistemului, dar au asociate câte un nume, reprezentat printr-un fișier de tipul special *fifo*, care este păstrat în sistemul de fișiere (din acest motiv, aceste fișiere *fifo* se mai numesc și *pipe-uri cu nume*).



# Agenda

Introducere

Canale anoneme

Crearea lor, cu primitiva pipe

Modul de utilizare a unui canal anonim

*Demo:* exemple de comunicatie intre doua procese

Canale cu nume (fifo)

Crearea lor, cu primitiva mkfifo

Modul de utilizare a unui canal cu nume

Despre persistenta informatiei dintr-un fisier fifo

Deosebiri ale canalelor cu nume fata de cele anone

Sabloane de comunicatie intre procese

Aplicatii ale canalelor de comunicatie

Referinte bibliografice

Introducere

**Canale anone**me

Crearea lor, cu primitiva pipe

Modul de utilizare a unui canal anonim

*Demo:* exemple de comunicatie intre doua procese

**Canale cu nume (fifo)**

Crearea lor, cu primitiva mkfifo

Modul de utilizare a unui canal cu nume

Despre persistenta informatiei dintr-un fisier fifo

Deosebiri ale canalelor cu nume fata de cele anone

**Caracteristici comune pentru ambele tipuri de canale**

Caracteristici si restrictii ale canalelor de comunicatie

Comportamentul implicit, de tip blocant

Comportamentul de tip neblocant

**Sabloane de comunicatie intre procese**

Clasificarea sabloanelor de comunicatie inter-procese

Sablonul de comunicatie *unul-la-unul*

Sablonul de comunicatie *unul-la-mulți*

Sablonul de comunicatie *mulți-la-unul*

Sablonul de comunicatie *mulți-la-mulți*

**Aplicatii ale canalelor de comunicatie**

Aplicatia #1: implementarea unui semafor

Aplicatia #2: implementarea unei aplicatii client/server

**Referinte bibliografice**



## Crearea lor, cu primitiva pipe

Introducere

Canale anonime

**Crearea lor, cu primitiva pipe**

Modul de utilizare a unui canal anonim

Demo: exemple de comunicatie intre doua procese

Canale cu nume (fifo)

Caracteristici comune pentru ambele tipuri de canale

Sabloane de comunicatie intre procese

Aplicatii ale canalelor de comunicatie

Referinte bibliografice

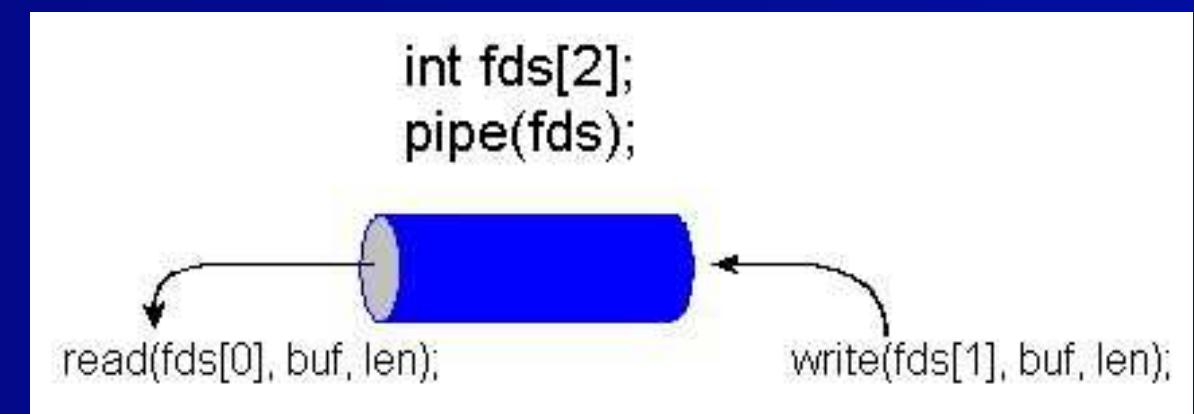
Un *canal anonim* se creeaza cu ajutorul primitivei **pipe**.

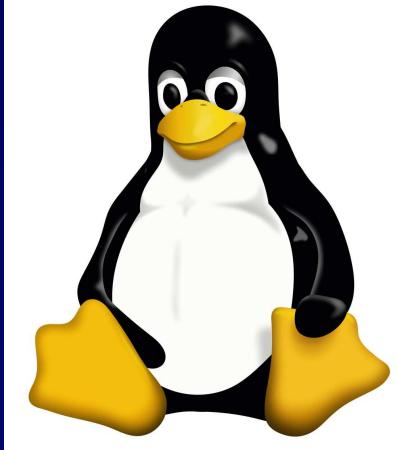
Interfața acestei funcții este următoarea ([5]):

```
int pipe(int *p)
```

- $p$  = parametrul efectiv de apel trebuie să fie un vector `int [2]`, care va fi actualizat de funcție în felul următor:
  - $p[0]$  va fi descriptorul de fisier deschis pentru *capătul de citire* al canalului
  - $p[1]$  va fi descriptorul de fisier deschis pentru *capătul de scriere* al canalului
- valoarea returnată este 0, în caz de succes, sau -1, în caz de eroare.

**Efect:** în urma executiei primitivei `pipe` se creeaza un canal anonim și este deschis automat la ambele capete – în citire la capătul referit prin descriptorul  $p[0]$  și, respectiv, în scriere la capătul referit prin descriptorul  $p[1]$ .





## Modul de utilizare a unui canal anonim

Introducere

Canale anonte

Crearea lor, cu primitiva pipe

Modul de utilizare a unui canal  
anonim

Demo: exemple de  
comunicatie intre doua  
procese

Canale cu nume (fifo)

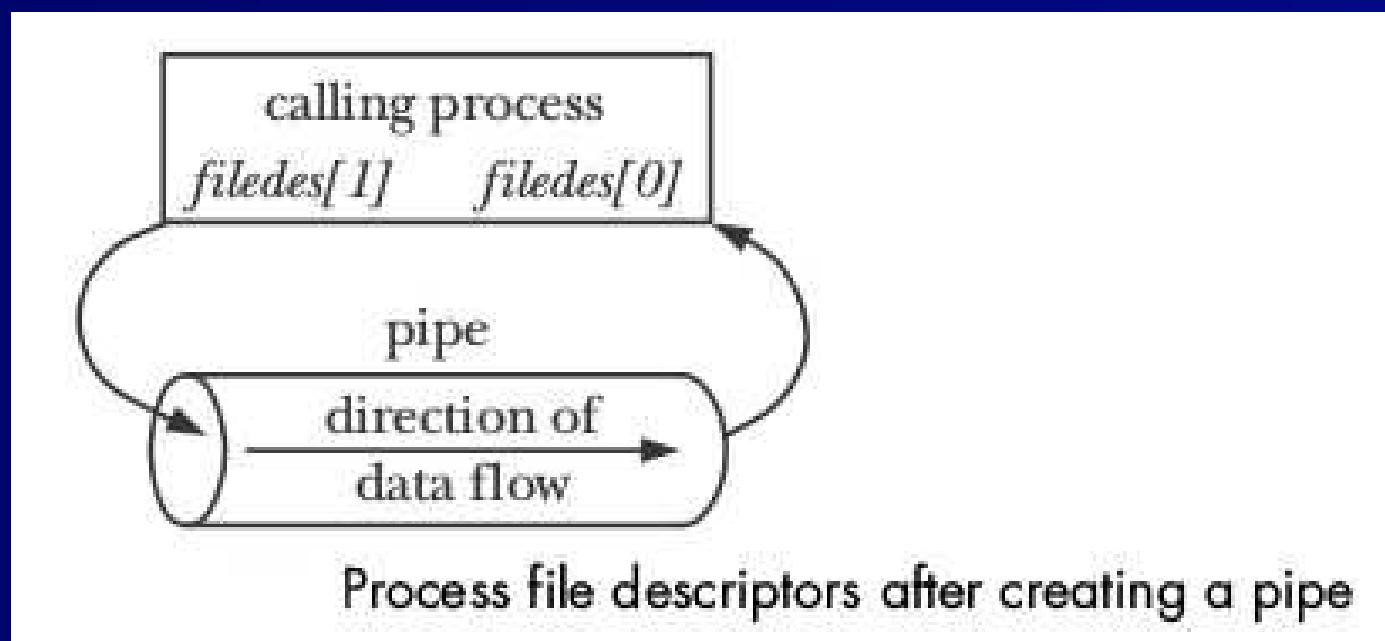
Caracteristici comune pentru  
ambele tipuri de canale

Sabloane de comunicatie intre  
procese

Aplicatii ale canalelor de  
comunicatie

Referinte bibliografice

După crearea unui canal anonim, folosirea sa pentru comunicatia locală intre două (sau mai multe) procese se face prin scrierea informației în acest canal și, respectiv, prin citirea informației din canal.



Iar scrierea în canal și respectiv citirea din canal, prin intermediul celor doi descriptori  $p[0]$  și  $p[1]$ , se efectuează la fel ca pentru fișierele obișnuite, i.e. folosind apelurile `read` și `write`, sau cu funcțiile I/O din biblioteca stdio.

\* \* \*

### *Restricție importantă:*

Deoarece acest tip de canale sunt *anonte* (i.e., nu au nume), pot fi utilizate pentru comunicatie doar de către procese "înrudite" prin apeluri `fork`/`exec`.

De ce? Motivația este următoarea: ... (vezi slide-ul următor)



## Modul de utilizare a unui canal anonim (cont.)

Introducere

Canale anonime

Crearea lor, cu primitiva pipe

Modul de utilizare a unui canal  
anonim

Demo: exemple de  
comunicatie intre doua  
procese

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

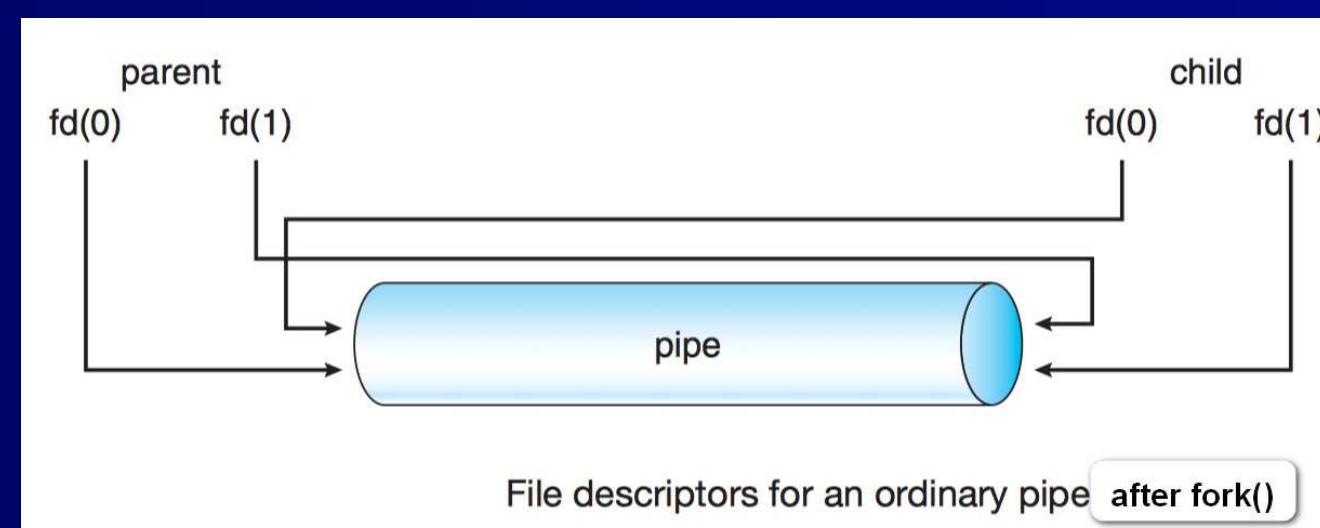
Şabloane de comunicatie intre  
procese

Aplicaţii ale canalelor de  
comunicatie

Referinte bibliografice

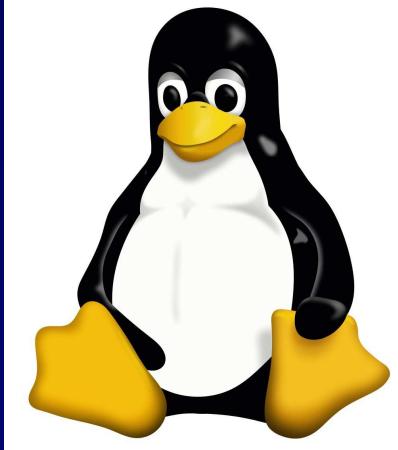
*Motivatie:* pentru ca două (sau mai multe) procese să poată folosi un canal anonim pentru a comunica între ele, acele procese trebuie să aibă la dispoziție cei doi descriptori  $p[0]$  și  $p[1]$  obținuți prin crearea canalului. Deci procesul care a creat canalul prin apelul pipe, va trebui să le “transmită” cumva celuilalt proces.

De exemplu, în cazul când se dorește să se utilizeze un canal anonim pentru comunicarea între două procese de tipul părinte-fiu, atunci este suficient să se apeleze primitiva pipe de creare a canalului *înaintea* apelului primitivei fork de creare a procesului fiu. În acest fel, prin clonare, avem la dispoziție și în procesul fiu cei doi descriptori necesari pentru comunicare prin intermediul aceluia canal anonim.



*Notă:*

“Transmiterea” descriptorilor canalului are loc și în cazul apelului primitivelor exec (deoarece descriptorii de fișiere deschise se moștenesc prin exec).



# Modul de utilizare a unui canal anonim (cont.)

Introducere

Canale anonyme

Crearea lor, cu primitiva pipe

Modul de utilizare a unui canal  
anonim

Demo: exemple de  
comunicatie intre doua  
procese

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

Sabloane de comunicatie intre  
procese

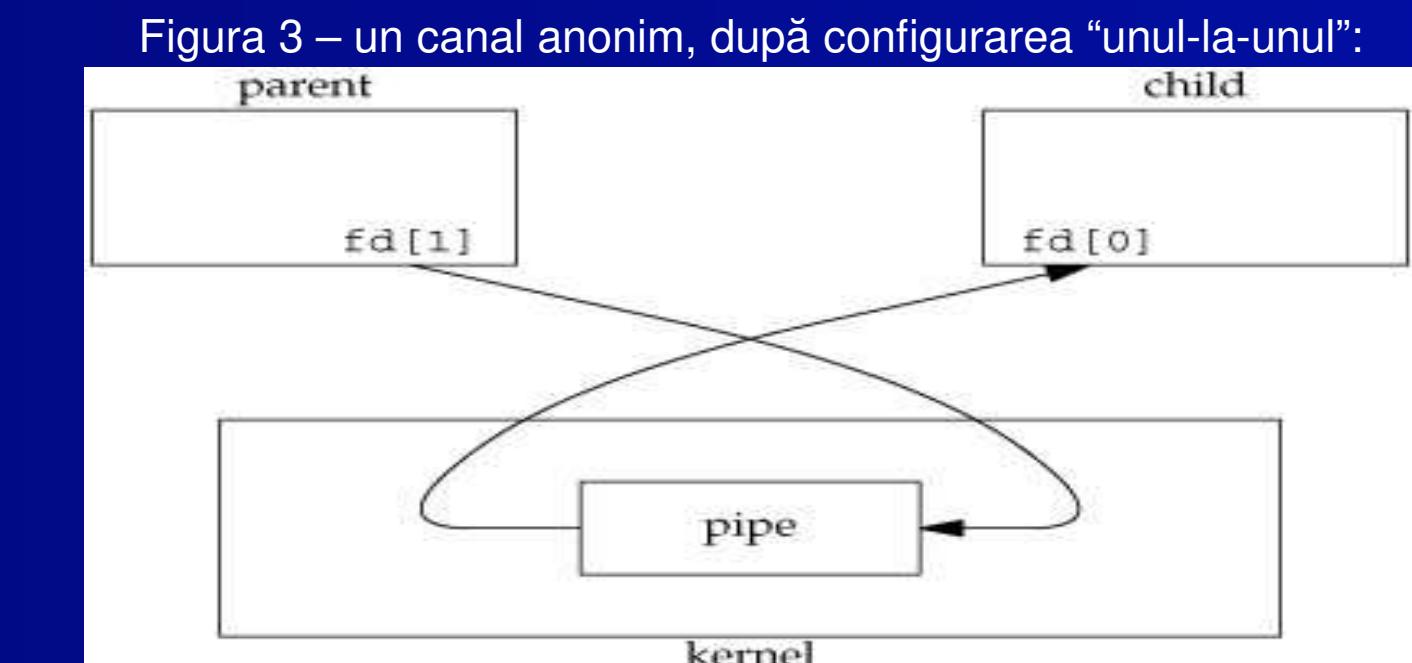
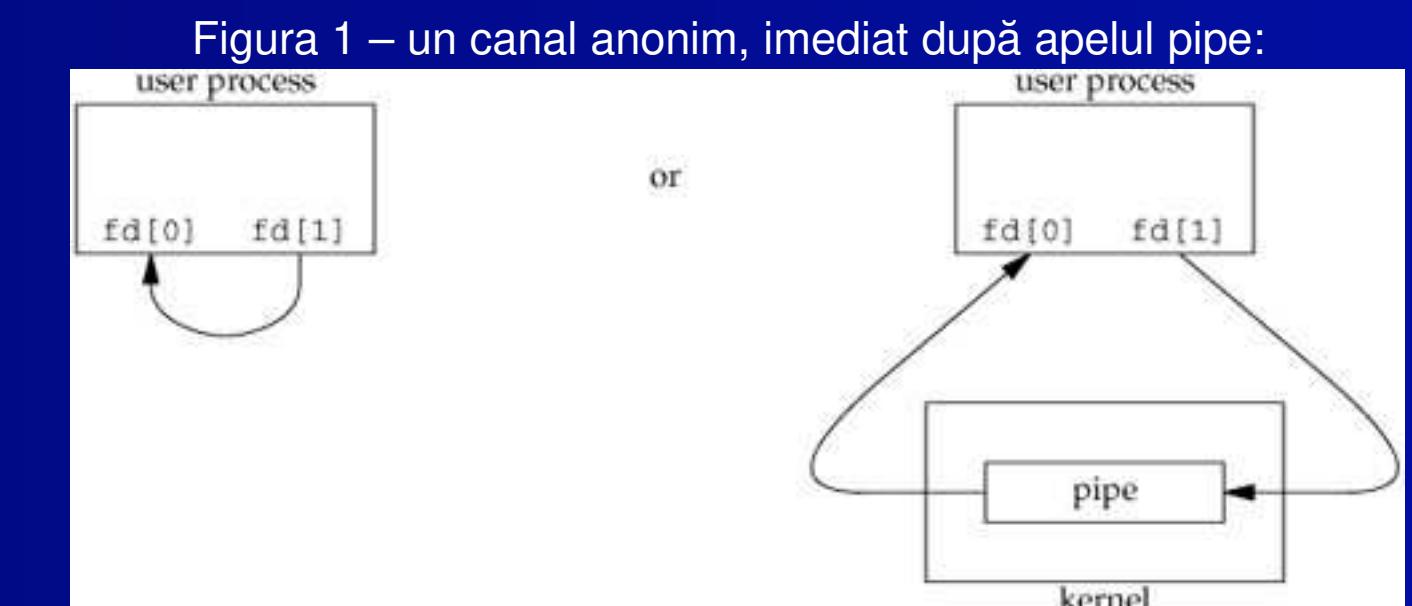
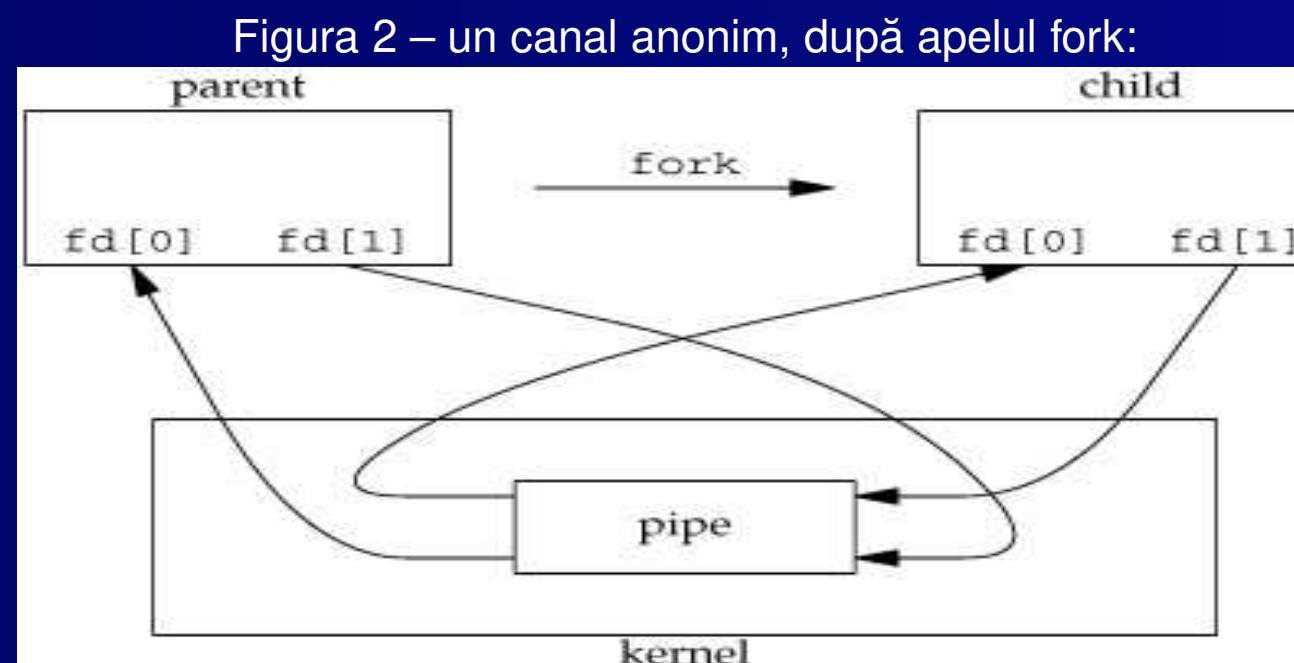
Aplicatii ale canalelor de  
comunicatie

Referinte bibliografice

## Altă restricție:

Dacă un proces își închide vreunul dintre  
capetele unui canal anonim, atunci nu  
mai are nicio posibilitate de a redeschide  
ulterior acel capăt al canalului.

\* \* \*



*Notă:* vom vedea ulterior că aceste două restricții de folosire a canalelor anonyme nu  
mai sunt valabile și în cazul canalelor cu nume (!).



## Demo: exemple de comunicație între două procese

Introducere

Canale anonime

Crearea lor, cu primitiva pipe

Modul de utilizare a unui canal anonim

*Demo: exemple de comunicatie între două procese*

Canale cu nume (fifo)

Caracteristici comune pentru ambele tipuri de canale

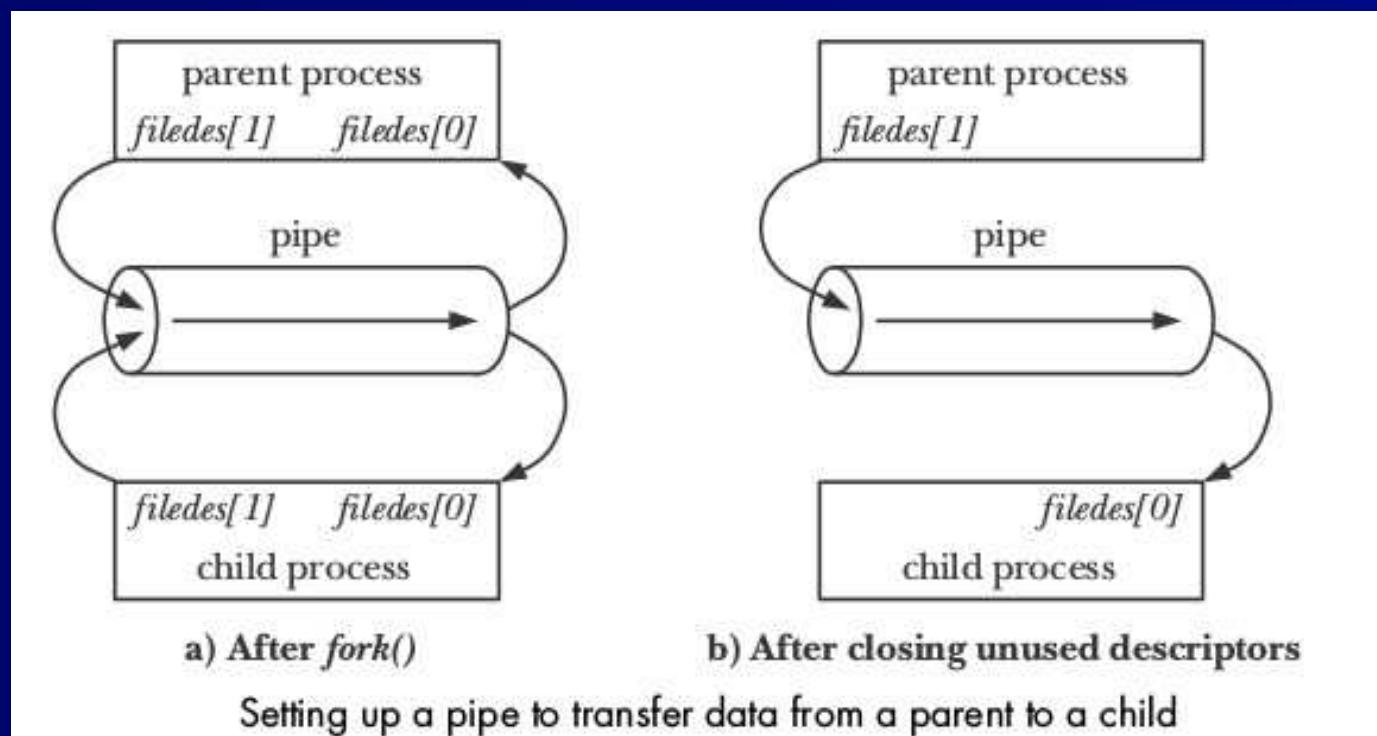
Şabioane de comunicație între procese

Aplicații ale canalelor de comunicatie

Referințe bibliografice

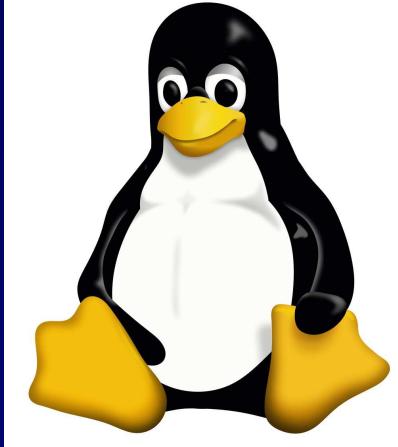
- Exemplul #1: un program care exemplifică modul de utilizare a unui canal anonim pentru comunicația între două procese, de tipul producător-consumator. În acest exemplu se ilustrează folosirea primitivelor `read` și `write` (*i.e.*, funcțiile din API-ul POSIX) pentru a citi din canal, respectiv pentru a scrie în canal.

A se vedea fișierul sursă `pipe_ex1.c` ([2]).



Efectul acestui program: mai întâi se creează un canal anonim și un proces fiu. Apoi, procesul părinte citește o secvență de caractere de la tastatură, secvență terminată cu combinația de taste CTRL+D (*i.e.*, caracterul EOF în UNIX), și le transmite procesului fiu, prin intermediul canalului anonim, doar pe acele care sunt litere mici. În procesul fiu citește din canal caracterele trasmise de procesul părinte și le afișează pe ecran.

*Observație:* pentru explicații suplimentare despre acest program demonstrativ, consultați exemplul [FirstDemo – `pipe_ex1`] din suportul online de laborator ([3]).



## Demo: exemple de comunicație între două procese (cont.)

Introducere

Canale anonte

Crearea lor, cu primitiva pipe  
Modul de utilizare a unui canal  
anonim

*Demo: exemple de  
comunicație între două  
procese*

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

Şabioane de comunicație între  
procese

Aplicații ale canalelor de  
comunicație

Referințe bibliografice

Pentru comunicația prin intermediul canalelor anonte se pot folosi și funcțiile I/O de nivel înalt :

- **Exemplul #2:** un alt program care exemplifică folosirea unui canal anonim pentru comunicăția între două procese, de tipul producător-consumator.

De această dată, se utilizează funcțiile fscanf și, respectiv, fprintf (*i.e.*, din biblioteca stdio) pentru a citi din canal și, respectiv, pentru a scrie în canal.

A se vedea fișierul sursă [pipe\\_ex2.c](#) ([2]).

*Notă:* în acest caz, este necesară conversia descriptorilor de fișiere de la tipul int (*i.e.*, descriptorii foloșiți de apelurile I/O din API-ul POSIX) la descriptori de tipul FILE\* (*i.e.*, descriptorii foloșiți de funcțiile I/O din biblioteca stdio), lucru realizabil cu ajutorul funcției de bibliotecă fdopen.

Efectul acestui program: mai întâi, se creează un canal anonim și un proces fiu. Apoi, procesul tată citește o secvență de numere de la tastatură, secvență terminată cu combinația de taste CTRL+D (*i.e.*, caracterul EOF în UNIX), și le transmite procesului fiu, prin intermediul canalului anonim. În procesul fiu citește din canal numerele trasmise de procesul părinte și le afișează pe ecran.

*Observație:* pentru explicații suplimentare despre acest program demonstrativ, consultați exemplul [\[SecondDemo – pipe\\_ex2\]](#) din suportul online de laborator ([3]).



# Agenda

Introducere

Canale anoneme

Canale cu nume (fifo)

Crearea lor, cu primitiva

`mkfifo`

Modul de utilizare a unui canal  
cu nume

Despre persistența informației  
dintr-un fișier *fifo*

Deosebiri ale canalelor cu  
nume față de cele anonime

Caracteristici comune pentru  
ambele tipuri de canale

Şabloane de comunicație între  
procese

Aplicații ale canalelor de  
comunicație

Referințe bibliografice

Introducere

**Canale anone**me

Crearea lor, cu primitiva pipe

Modul de utilizare a unui canal anonim

*Demo:* exemple de comunicație între două procese

**Canale cu nume (fifo)**

Crearea lor, cu primitiva `mkfifo`

Modul de utilizare a unui canal cu nume

Despre persistența informației dintr-un fișier *fifo*

Deosebiri ale canalelor cu nume față de cele anonime

**Caracteristici comune pentru ambele tipuri de canale**

Caracteristici și restricții ale canalelor de comunicație

Comportamentul implicit, de tip blocant

Comportamentul de tip neblocant

**Şabloane de comunicație între procese**

Clasificarea şabloanelor de comunicație inter-procese

Şablonul de comunicație *unul-la-unul*

Şablonul de comunicație *unul-la-mulți*

Şablonul de comunicație *mulți-la-unul*

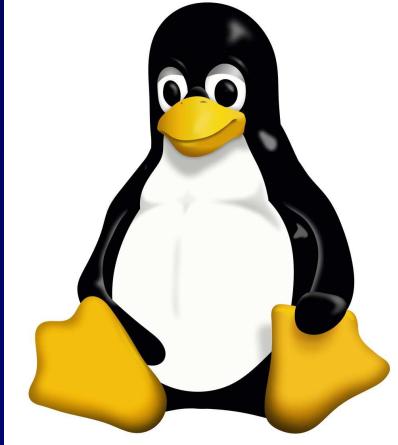
Şablonul de comunicație *mulți-la-mulți*

**Aplicații ale canalelor de comunicație**

Aplicația #1: implementarea unui semafor

Aplicația #2: implementarea unei aplicații client/server

**Referințe bibliografice**



## Crearea lor, cu primitiva mkfifo

Introducere

Canale anonime

Canale cu nume (fifo)

Crearea lor, cu primitiva  
`mkfifo`

Modul de utilizare a unui canal  
cu nume

Despre persistența informației  
dintr-un fișier *fifo*

Deosebiri ale canalelor cu  
nume față de cele anonime

Caracteristici comune pentru  
ambele tipuri de canale

Sabloane de comunicație între  
procese

Aplicații ale canalelor de  
comunicație

Referințe bibliografice

Un *canal cu nume* se creează cu ajutorul primitivei `mkfifo`.

Interfața acestei funcții este următoarea ([5]):

```
int mkfifo(char *nume, int permisiuni)
```

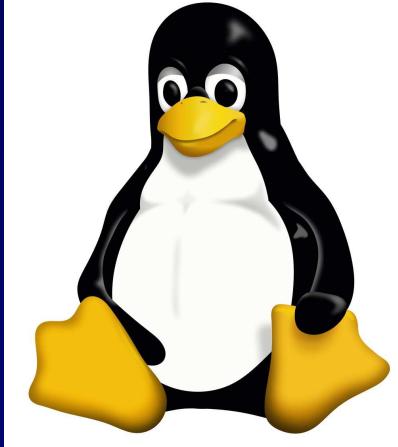
- *nume* = numele fișierului (de tip *fifo*) ce va fi creat
- *permisiuni* = permisiunile pentru fișierul ce va fi creat
- valoarea returnată este 0, în caz de succes, sau -1, în caz de eroare.

**Efect:** în urma execuției primitivei `mkfifo` se creează un canal cu nume, dar *fără a fi deschis la ambele capete* (!), precum se întâmplă în cazul creării unui canal anonim.

*Notă:* crearea unui fișier *fifo* se mai poate face cu ajutorul primitivei `mknod` apelată cu *flag-ul S\_IFIFO*. De asemenea, mai poate fi creat și direct de la linia de comandă (*i.e.*, prompterul *shell*-ului), cu comenzile `mkfifo` sau `mknod`.

Exemplu de creare a unui fișier *fifo*: a se vedea fișierul sursă `mkfifo_ex.c` ([2]).

*Observație:* pentru explicații suplimentare despre acest program demonstrativ, consultați exemplul [`ThirdDemo – mkfifo_ex`] din suportul online de laborator ([3]).



## Modul de utilizare a unui canal cu nume

Introducere

Canale anonime

Canale cu nume (fifo)

Crearea lor, cu primitiva  
`mkfifo`

Modul de utilizare a unui canal  
cu nume

Despre persistența informației  
dintr-un fișier *fifo*

Deosebiri ale canalelor cu  
nume față de cele anonime

Caracteristici comune pentru  
ambele tipuri de canale

Şabloane de comunicație între  
procese

Aplicații ale canalelor de  
comunicație

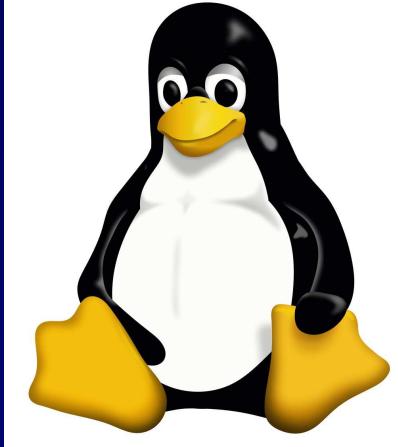
Referințe bibliografice

După crearea unui canal cu nume, folosirea sa pentru comunicația locală între două (sau mai multe) procese se face prin scrierea informației în acest canal și, respectiv, prin citirea informației din canal.

Iar scrierea în canal și, respectiv, citirea din canal se efectuează la fel ca pentru fișierele obișnuite. Și anume: mai întâi se deschide *explicit* fișierul la “capătul” dorit (cel de citire și/sau cel de scriere), pentru a se obține descriptorul necesar, apoi se scrie în el și/sau se citește din el, prin intermediul descriptorului obținut explicit, i.e. folosind apelurile de sistem `read` și `write`, sau cu funcțiile de citire/scriere din biblioteca `stdio`, iar la sfârșit se închide descriptorul respectiv.

\* \* \*

*Observație importantă:* deoarece acest tip de canale nu sunt *anonime* (i.e., au nume prin care pot fi referite), pot fi utilizate pentru comunicație între **orice procese care cunosc numele fișierului *fifo* respectiv**, deci nu mai avem restricția de la canale anonime, aceea că procesele trebuiau să fie “înrudite” prin `fork`/`exec`.



## Modul de utilizare a unui canal cu nume (cont.)

Introducere

Canale anone

Canale cu nume (*fifo*)

Crearea lor, cu primitiva

`mkfifo`

Modul de utilizare a unui canal  
cu nume

Despre persistența informației  
dintr-un fișier *fifo*

Deosebiri ale canalelor cu  
nume față de cele anone

Caracteristici comune pentru  
ambele tipuri de canale

Şabloane de comunicație între  
procese

Aplicații ale canalelor de  
comunicație

Referințe bibliografice

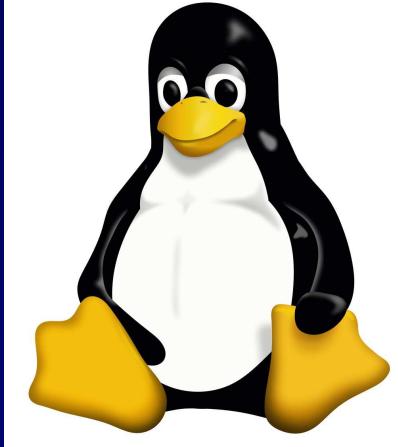
Așadar, operațiile asupra canalelor *fifo* se vor face fie cu primitivele I/O de nivel scăzut (i.e., `open`, `read`, `write`, `close`), fie cu funcțiile I/O de nivel înalt din biblioteca standard de I/O din C (i.e., `fopen`, `fread/fscanf`, `fwrite/fprintf`, `fclose`, și.a.).

La fel ca pentru fișiere obișnuite, “deschiderea” unui fișier *fifo* se face explicit, printr-un apel al funcției `open` sau `fopen`, într-unul din următoarele trei moduri posibile, specificat prin parametrul transmis funcției de deschidere:

- *read & write* (i.e., deschiderea ambelor capete ale canalului)
- *read-only* (i.e., deschiderea doar a capătului de citire)
- *write-only* (i.e., deschiderea doar a capătului de scriere)

*Observație importantă:*

Implicit, deschiderea se face în mod *blocant*, i.e. o deschidere *read-only* trebuie să se “sincronizeze” cu una *write-only*. Cu alte cuvinte, dacă un proces încearcă să deschidă un capăt al canalului, apelul funcției de deschidere rămâne blocat (i.e., funcția nu returnează) până când un alt proces va deschide celălalt capăt al canalului.



## Despre persistența informației dintr-un fișier *fifo*

Introducere

Canale anonime

Canale cu nume (*fifo*)

Crearea lor, cu primitiva  
`mkfifo`

Modul de utilizare a unui canal  
cu nume

Despre persistența informației  
dintr-un fișier *fifo*

Deosebiri ale canalelor cu  
nume față de cele anonime

Caracteristici comune pentru  
ambele tipuri de canale

Şabloane de comunicație între  
procese

Aplicații ale canalelor de  
comunicație

Referințe bibliografice

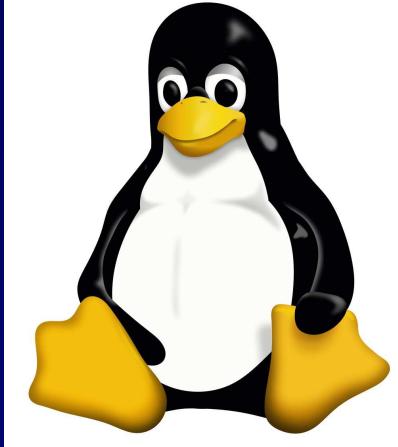
*Observație:* un *canal cu nume* este creat tot în memoria internă a sistemului (ca și unul anonim), dar în plus are asociat un nume, reprezentat printr-un fișier de tipul special *fifo*, care este păstrat în sistemul de fișiere.

*Concluzie:* informațiile conținute în acest tip de fișiere sunt stocate în memoria principală, nu pe disc, și ca urmare nu sunt persistente. (Practic, conținutul unui fișier *fifo* este gestionat, de către nucleul SO-ului, tot ca o coadă FIFO aflată în memorie, la fel ca și în cazul canalelor anonime.)

Așadar, **perioada de retenție a informației stocate într-un canal este următoarea:** Spre deosebire de fișierele obișnuite ( ce păstrează informația scrisă în ele pe o perioadă nedeterminată – mai precis, până la o eventuală operație de modificare sau ștergere ), în cazul unui fișier *fifo* informația scrisă în el se păstrează doar din momentul scrierii și până în momentul când atât procesul care a scris acea informație, cât și orice alt proces ce-l accesa, termină accesul la acel canal *fifo* (închizându-și capetele canalului), iar aceasta numai dacă informația nu este consumată mai devreme, prin citire.

*Demo:* a se vedea fișierul sursă `testare_retentie_fifo.c` ([2]).

*Notă:* pentru mai multe explicații despre acest program și descrierea comportamentului său la execuție, consultați exemplul [FourthDemo – `testare_retentie_fifo`] din suportul online de laborator ([3]).



# Deosebiri ale canalelor cu nume față de cele anonime

Introducere

Canale anonime

Canale cu nume (*fifo*)

Crearea lor, cu primitiva  
`mkfifo`

Modul de utilizare a unui canal  
cu nume

Despre persistența informației  
dintr-un fișier *fifo*

**Deosebiri ale canalelor cu  
nume față de cele anonime**

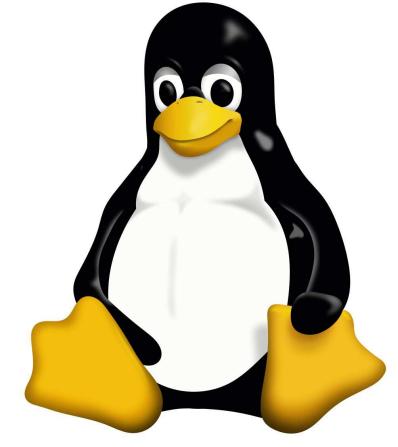
Caracteristici comune pentru  
ambele tipuri de canale

Şabloane de comunicație între  
procese

Aplicații ale canalelor de  
comunicație

Referințe bibliografice

- Funcția de creare a unui fișier *fifo* (i.e., canal cu nume) nu realizează și deschiderea automată a celor două capete ale canalului, precum la canalele anonime, ci acestea trebuie să fie deschise explicit, după creare, prin apelul unei funcții de deschidere a aceluui fișier.
- Un canal *fifo* poate fi deschis, la oricare dintre capete, de orice proces, indiferent dacă acel proces are sau nu vreo legătură de “rudenie” (prin `fork` / `exec`) cu procesul care a creat canalul respectiv.  
Aceasta este posibil deoarece un proces trebuie doar să cunoască numele fișierului *fifo* pe care dorește să-l deschidă, pentru a-l putea deschide. Evident, mai trebuie și ca procesul respectiv să aibă drepturi de acces pentru acel fișier *fifo*.
- După ce un proces închide un capăt al unui canal *fifo*, acel proces poate redeschide din nou acel capăt al canalului.  
Motivul pentru care ar dori aceasta: poate constata, ulterior momentului închiderii, că are nevoie să mai efectueze și alte operații I/O asupra aceluui capăt.



# Agenda

Introducere

Canale anoneme

Canale cu nume (fifo)

Caracteristici comune pentru ambele tipuri de canale

Caracteristici și restricții ale canalelor de comunicație

Comportamentul implicit, de tip blocant

Comportamentul de tip neblocant

Sabloane de comunicație între procese

Aplicații ale canalelor de comunicație

Referințe bibliografice

Introducere

**Canale anone**me

Crearea lor, cu primitiva pipe

Modul de utilizare a unui canal anonim

*Demo:* exemple de comunicație între două procese

**Canale cu nume (fifo)**

Crearea lor, cu primitiva mkfifo

Modul de utilizare a unui canal cu nume

Despre persistența informației dintr-un fișier fifo

Deosebiri ale canalelor cu nume față de cele anone

**Caracteristici comune pentru ambele tipuri de canale**

Caracteristici și restricții ale canalelor de comunicație

Comportamentul implicit, de tip blocant

Comportamentul de tip neblocant

**Sabloane de comunicație între procese**

Clasificarea sabloanelor de comunicație inter-procese

Şablonul de comunicație *unul-la-unul*

Şablonul de comunicație *unul-la-mulți*

Şablonul de comunicație *mulți-la-unul*

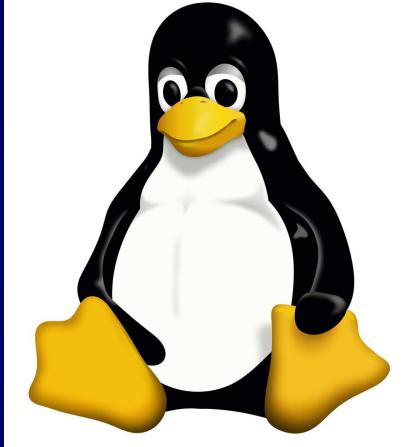
Şablonul de comunicație *mulți-la-mulți*

**Aplicații ale canalelor de comunicație**

Aplicația #1: implementarea unui semafor

Aplicația #2: implementarea unei aplicații client/server

**Referințe bibliografice**



# Caracteristici și restricții ale canalelor de comunicație

Introducere

Canale anone

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

Caracteristici și restricții ale  
canalelor de comunicație

Comportamentul implicit, de tip  
blocant

Comportamentul de tip  
neblocant

Şabloane de comunicație între  
procese

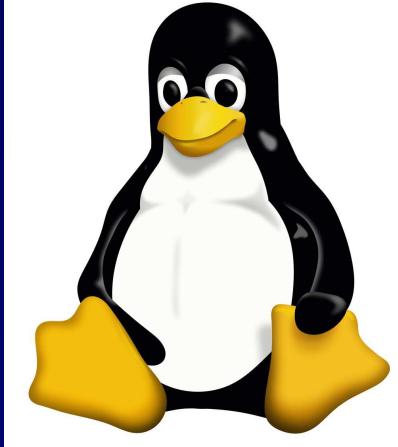
Aplicații ale canalelor de  
comunicație

Referințe bibliografice

- Ambele tipuri de canale sunt canale *unidirectionale*, adică pe la un capăt se scrie informația în canal, iar pe la capătul opus se citește.

*Notă:* însă putem avea mai mulți scriitori (*i.e.*, toate procesele ce au acces la capătul de scriere, pot să scrie în canal), și/sau mai mulți cititori (*i.e.*, toate procesele ce au acces la capătul de citire, pot să citească din canal).

- Unitatea de informație pentru ambele tipuri de canale este *octetul*. Cu alte cuvinte, cantitatea minimă de informație ce poate fi scrisă în canal, respectiv citită din canal, este de 1 octet.
- Capacitatea unui canal de comunicație este limitată la o anumită dimensiune maximă (*e.g.*, 4 Ko, 16 Ko, 64 Ko, *s.a.*), ce este configurabilă. Spre exemplu, în Linux (începând de la versiunea 2.6.35) se poate afla, respectiv configura, capacitatea unui canal de comunicație prin operațiile `F_GETPIPE_SZ`, respectiv `F_SETPIPE_SZ`, disponibile prin apelul de sistem `fcntl`. Pentru detalii, consultați documentația acestui apel (*i.e.*, `man 2 fcntl`), precum și explicațiile prezentate în exercițiul rezolvat [[A pipe's capacity](#)].



## Caracteristici și restricții ale canalelor de comunicație (cont.)

Introducere

Canale anone

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

Caracteristici și restricții ale  
canalelor de comunicație

Comportamentul implicit, de tip  
blocant

Comportamentul de tip  
neblocant

Şabloane de comunicație între  
procese

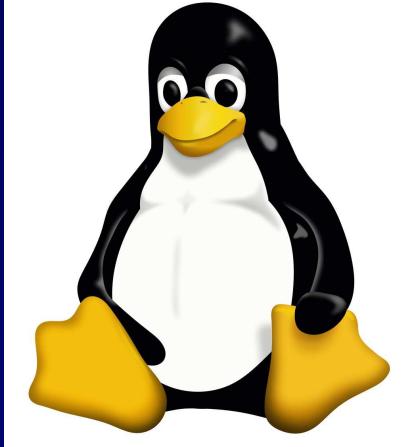
Aplicații ale canalelor de  
comunicație

Referințe bibliografice

- Practic, ambele tipuri de canale (*i.e.*, și cele anone, și cele cu nume) funcționează ca o coadă, adică o listă FIFO (*First-In, First-Out*), deci citirea din canal se face cu “distrugerea” (*i.e.*, *consumul din canal a*) informației citite (!), iar scrierea în canal se face prin “inserarea” în coadă a informației scrise.

*Concluzie:* aşadar, citirea dintr-un fișier *fifo* diferă de citirea din fișiere obișnuite, pentru care citirea se face fără consumarea informației din fișier.

- În cazul fișierelor obișnuite am văzut că există noțiunea de *offset* (*i.e.*, poziția curentă în fișier, de la care se efectuează operația curentă de citire sau scriere). În schimb, nici pentru fișierele *fifo*, nici pentru canalele anone nu există această noțiune de *offset*, ele funcționând precum o coadă FIFO.



## Comportamentul implicit, de tip blocant

Introducere

Canale anonime

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

Caracteristici și restricții ale  
canalelor de comunicație

Comportamentul implicit, de tip  
blocant

Comportamentul de tip  
neblocant

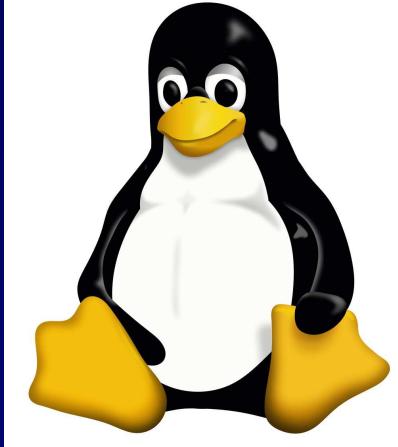
Şabloane de comunicație între  
procese

Aplicații ale canalelor de  
comunicație

Referințe bibliografice

### ■ Citirea dintr-un canal de comunicație funcționează în felul următor:

- Apelul de citire `read` va citi din canal și va returna imediat, fără să se blocheze, numai dacă mai este suficientă informație în canal, iar în acest caz valoarea returnată reprezintă numărul de octeți citiți din canal.
- Altfel, dacă canalul este gol, sau nu conține suficientă informație, apelul de citire `read` va rămâne blocat până când va avea suficientă informație în canal pentru a putea citi cantitatea de informație specificată, ceea ce se va întâmpla în momentul când un alt proces va scrie în canal.
- Alt caz de excepție la citire: dacă un proces încearcă să citească din canal și niciun proces nu mai este capabil să scrie în canal (deoarece toate procesele și-au închis deja capătul de scriere), atunci apelul `read` returnează imediat valoarea 0 prin care se semnalizează că “a citit EOF” din canal.  
*Concluzie: pentru a se putea citi EOF din canal, trebuie ca mai întâi toate procesele să închidă canalul în scriere* (adică să închidă descriptorul corespunzător capătului de scriere).



## Comportamentul implicit, de tip blocant (cont.)

Introducere

Canale anonime

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

Caracteristici și restricții ale  
canalelor de comunicație

Comportamentul implicit, de tip  
blocant

Comportamentul de tip  
neblocant

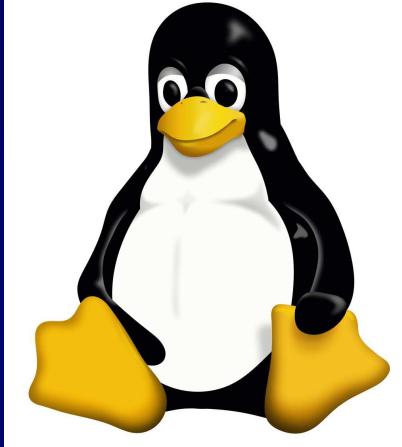
Şabloane de comunicație între  
procese

Aplicații ale canalelor de  
comunicație

Referințe bibliografice

### ■ Scrierea într-un canal de comunicație funcționează în felul următor:

- Apelul de scriere `write` va scrie în canal și va returna imediat, fără să se blocheze, numai dacă mai este suficient spațiu liber în canal, iar în acest caz valoarea returnată reprezintă numărul de octeți efectiv scriși în canal (care poate să nu coincidă întotdeauna cu numărul de octeți ce se doreau a se scrie, căci pot apărea eventuale erori I/O).
- Altfel, dacă canalul este plin, sau nu conține suficient spațiu liber, apelul de scriere `write` va rămâne blocat până când va avea suficient spațiu liber în canal pentru a putea scrie informația specificată ca argument, ceea ce se va întâmpla în momentul când un alt proces va citi din canal.
- Alt caz de excepție la scriere: dacă un proces încearcă să scrie în canal și niciun proces nu mai este capabil să citească din canal (deoarece toate procesele și-au închis deja capătul de citire), atunci sistemul va trimite aceluui proces semnalul **SIGPIPE**, ce cauzează terminarea forțată a procesului, fără a afișa însă vreun mesaj de eroare (Notă: versiunile mai vechi de Linux afișau “**Broken pipe**”).



## Comportamentul implicit, de tip blocant (cont.)

Introducere

Canale anonime

Canale cu nume (fifo)

Caracteristici comune pentru ambele tipuri de canale

Caracteristici și restricții ale canalelor de comunicație

Comportamentul implicit, de tip blocant

Comportamentul de tip neblocant

Şabloane de comunicație între procese

Aplicații ale canalelor de comunicație

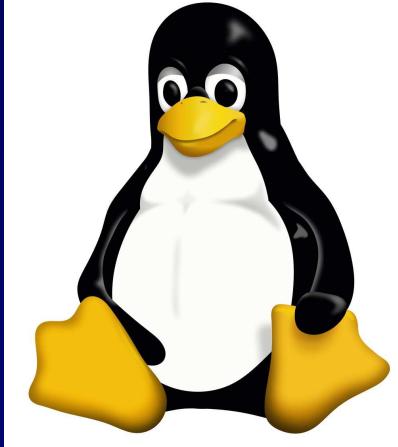
Referințe bibliografice

*Observație:* în locul primitivelor `read` și `write` din API-ul POSIX, putem folosi funcțiile I/O de nivel înalt din biblioteca `stdio` pentru a citi din canal (e.g., cu `fread`, `fscanf`, ș.a.) și, respectiv, pentru a scrie în canal (e.g., cu `fwrite`, `fprintf`, ș.a.). Si aceste funcții de bibliotecă au un *comportament implicit blocant*, similar cu cel descris mai sus, singura diferență fiind aceea că, reamintiți-vă, aceste funcții lucrează *buffer-izat* (!), i.e. folosind un *cache local* în *user-space*.

*Consecință:* modul de lucru *buffer-izat* al funcțiilor I/O din `stdio`, poate cauza uneori erori logice (i.e., bug-uri) dificil de depistat, *datorate neatentiei programatorului*, care poate uita să forțeze “golirea” *buffer-ului* în canal cu ajutorul funcției `fflush`, imediat după apelul funcției de scriere utilizate pentru a scrie acea informație în canal.

Si astfel, un proces cititor al acelei informații va rămâne blocat în apelul de citire, deoarece informația încă nu a ajuns în canal, iar programatorul va căuta cauza blocajului în altă parte, crezând că informația, pe care o scrisese, a ajuns “instantaneu” (i.e., *fără nicio întârziere sesizabilă*) în canal (!).

*Recomandare:* acordați mare atenție să nu comiteți acest gen de greșeli logice, căci le-am observat de nenumărate ori, pe parcursul anilor, în programele scrise de studenți.



## Comportamentul de tip neblocant

Introducere

Canale anonime

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

Caracteristici și restricții ale  
canalelor de comunicație

Comportamentul implicit, de tip  
blocant

Comportamentul de tip  
neblocant

Sabloane de comunicație între  
procese

Aplicații ale canalelor de  
comunicație

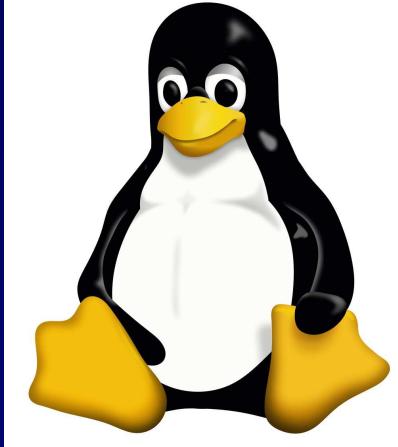
Referințe bibliografice

Cele descrise mai devreme, despre blocarea apelurilor de citire, respectiv de scriere, în cazul canalului gol, respectiv plin, corespund comportamentului implicit, de tip **blocant**, al canalelor de comunicație.

Acest comportament implicit poate fi modificat, pentru ambele tipuri de canale de comunicație, într-un comportament de tip **neblocant**, situație în care apelurile de citire și, respectiv, de scriere, nu mai rămân blocate în cazul canalului gol și, respectiv, în cazul canalului plin, ci returnează imediat valoarea -1, setând în mod corespunzător variabila errno.

Mai mult, putem modifica *separat* comportamentul pentru oricare dintre cele două capete ale unui canal, nu suntem limitați doar la a schimba *simultan* comportamentul pentru ambele capete (!).

În plus, în cazul canalelor cu nume, o deschidere *neblocantă* a uneia dintre capetele canalului va reuși imediat, fără să mai aștepte ca vreun alt proces să deschidă celălalt capăt, precum se întâmplă în cazul deschiderii *implicite*, de tip blocant.



## Comportamentul de tip neblocant (cont.)

Introducere

Canale anone

Canale cu nume (*fifo*)

Caracteristici comune pentru  
ambele tipuri de canale

Caracteristici și restricții ale  
canalelor de comunicație

Comportamentul implicit, de tip  
blocant

Comportamentul de tip  
neblocant

Sabloane de comunicație între  
procese

Aplicații ale canalelor de  
comunicație

Referințe bibliografice

Modificarea comportamentului implicit în comportament **neblocant** se realizează prin setarea atributului **O\_NONBLOCK** pentru descriptorul corespunzător aceluia capăt al canalului de comunicație pentru care se dorește modificarea comportamentului.

Setarea atributului **O\_NONBLOCK** pentru descriptorul dorit, se poate face astfel:

1. fie direct la deschiderea explicită a canalului, e.g. printr-un apel de forma:

```
fd_out = open("canal_fifo", O_WRONLY | O_NONBLOCK);
```

care va seta la deschidere atributul **O\_NONBLOCK** doar pentru capătul de scriere.

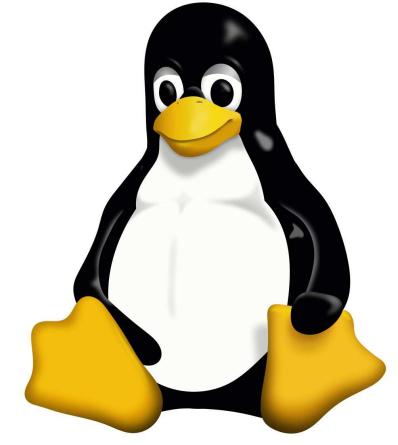
Această modalitate este posibilă numai pentru canale cu nume (i.e., fisiere *fifo*).

2. fie după deschiderea, implicită sau explicită, a canalului, utilizând primitiva `fcntl`, e.g. printr-un apel de forma: `fcntl(fd_out, F_SETFL, O_NONBLOCK);`

Această modalitate este posibilă pentru ambele tipuri de canale.

*Exercițiu:* scrieți un program prin care să determinați capacitatea ambelor tipuri de canale de comunicație pe sistemul Linux pe care lucrați.

*Rezolvare:* dacă nu reușiți să-l rezolvați singuri, citiți exercițiile rezolvate [A pipe's capacity] și [A fifo's capacity] prezentate în suportul de laborator.



# Agenda

Introducere

Canale anoneme

Canale cu nume (fifo)

Caracteristici comune pentru ambele tipuri de canale

Şabloane de comunicaţie între procese

Clasificarea şabloanelor de comunicaţie inter-procese

Şablonul de comunicaţie *unul-la-unul*

Şablonul de comunicaţie *unul-la-multi*

Şablonul de comunicaţie *multi-la-unul*

Şablonul de comunicaţie *multi-la-multi*

Aplicaţii ale canalelor de comunicaţie

Referinţe bibliografice

Introducere

## Canale anone

Crearea lor, cu primitiva pipe

Modul de utilizare a unui canal anonim

*Demo:* exemple de comunicaţie între două procese

## Canale cu nume (fifo)

Crearea lor, cu primitiva mkfifo

Modul de utilizare a unui canal cu nume

Despre persistenţa informaţiei dintr-un fişier fifo

Deosebiri ale canalelor cu nume faţă de cele anone

## Caracteristici comune pentru ambele tipuri de canale

Caracteristici şi restricţii ale canalelor de comunicaţie

Comportamentul implicit, de tip blocant

Comportamentul de tip neblocant

## Şabloane de comunicaţie între procese

Clasificarea şabloanelor de comunicaţie inter-procese

Şablonul de comunicaţie *unul-la-unul*

Şablonul de comunicaţie *unul-la-multi*

Şablonul de comunicaţie *multi-la-unul*

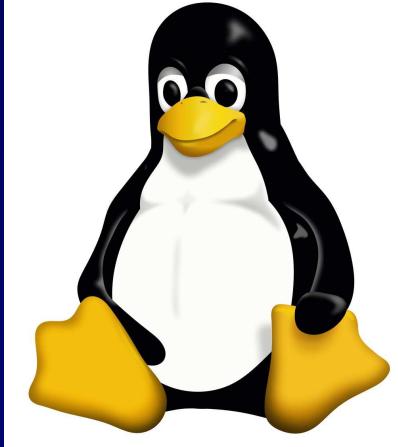
Şablonul de comunicaţie *multi-la-multi*

## Aplicaţii ale canalelor de comunicaţie

Aplicaţia #1: implementarea unui semafor

Aplicaţia #2: implementarea unei aplicaţii client/server

## Referinţe bibliografice



# Clasificarea șabloanelor de comunicație inter-procese

Introducere

Canale anone

Canale cu nume (fifo)

Caracteristici comune pentru ambele tipuri de canale

Şabloane de comunicație între procese

Clasificarea șabloanelor de comunicație inter-procese

Şablonul de comunicație *unul-la-unul*

Şablonul de comunicație *unul-la-mulți*

Şablonul de comunicație *mulți-la-unul*

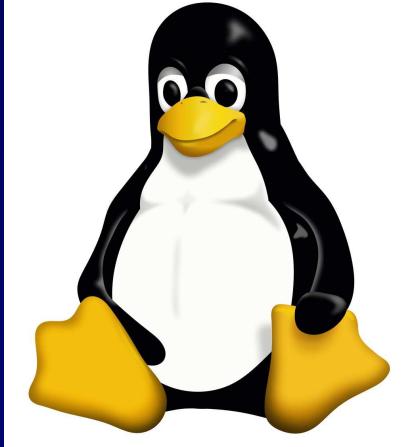
Şablonul de comunicație *mulți-la-mulți*

Aplicații ale canalelor de comunicație

Referințe bibliografice

După numărul de procese “scriitori” și, respectiv, de procese “cititori” ce utilizează un anumit canal de comunicație (anonim sau cu nume) pentru a comunica între ele, putem diferenția următoarele șabloane de comunicație inter-procese:

- Comunicație ***unul-la-unul***: canalul este folosit de un singur proces “scriitor” pentru a transmite date unui singur proces “cititor”.
- Comunicație ***unul-la-mulți***: canalul este folosit de un singur proces “scriitor” pentru a transmite date mai multor procese “cititori”.
- Comunicație ***mulți-la-unul***: canalul e folosit de mai multe procese “scriitori” pentru a transmite date unui singur proces “cititor”.
- Comunicație ***mulți-la-mulți***: canalul e folosit de mai multe procese “scriitori” pentru a transmite date mai multor procese “cititori”.



## Şablonul de comunicaţie *unul-la-unul*

Comunicaţia *unul-la-unul* reprezintă şablonul cel mai simplu, neridicând probleme deosebite de implementare. Din acest motiv, este şi cel mai folosit în practică.

*Exemplu:* cele două programe demonstrative prezentate anterior, în secţiunea despre canale anonime, se încadrează în acest şablon de comunicaţie.

*Demo:* exerciţiile rezolvate [['Producer-consumer' pattern #1, \(v2, using fifos for IPC\)](#)] și, respectiv, [['Producer-consumer' pattern #2, \(v2, using fifos for IPC\)](#)] din suportul online de laborator ([\[3\]](#)), ilustrează alte două programe care, fiecare în parte, utilizează un canal cu nume pentru comunicaţia *unul-la-unul* între două procese, unul cu rol de producător, iar celălalt cu rol de consumator.

\* \* \*

Celealte trei şabloane ridică anumite probleme de sincronizare, datorate accesului concurent al mai multor procese la câte unul, sau la ambele, dintre capetele canalului, probleme de care trebuie să se țină cont la implementarea acestor şabloane.

Vom trece în revistă, pe rând, aceste probleme de sincronizare, ce pot avea efecte asupra *integrității datelor* transmise prin canal (i.e., “coruperea” mesajelor).



## Şablonul de comunicaţie *unul-la-mulţi*

Factori ce pot genera anumite probleme de sincronizare, cu efecte asupra *integrității datelor*:

### ■ *lungimea mesajelor:*

- **mesaje de lungime *constantă***

Nu ridică probleme deosebite – fiecare mesaj poate fi citit *atomic* (i.e., dintr-o dată, printr-un singur apel `read`).

- **mesaje de lungime *variabilă***

Pot apărea probleme de sincronizare, deoarece mesajele nu mai pot fi citite *atomic*. Solutia este folosirea mesajelor formataste astfel:

MESAJ = HEADER + MESAJUL PROPRIU-ZIS ,

*header-ul* fiind un mesaj de lungime fixă ce conține lungimea mesajului propriu-zis.

*Protocolul de comunicație* utilizat: sunt necesare 2 apeluri `read` pentru a citi un mesaj în întregime, de aceea trebuie garantat accesul exclusiv la canal (folosind, spre exemplu, blocaje pe fișiere).

Introducere

Canale anone

Canale cu nume (*fifo*)

Caracteristici comune pentru  
ambele tipuri de canale

Şabloane de comunicaţie între  
procese

Clasificarea şabloanelor de  
comunicaţie inter-procese

Şablonul de comunicaţie  
*unul-la-unul*

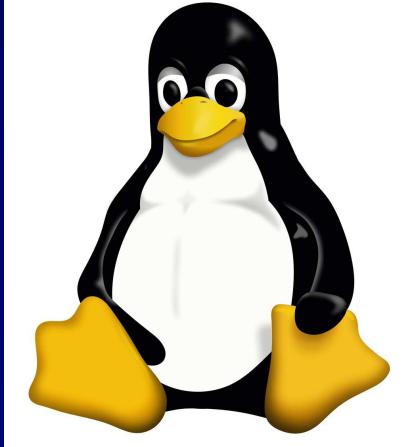
Şablonul de comunicaţie  
*unul-la-mulţi*

Şablonul de comunicaţie  
*mulţi-la-unul*

Şablonul de comunicaţie  
*mulţi-la-mulţi*

Aplicaţii ale canalelor de  
comunicaţie

Referințe bibliografice



## Şablonul de comunicaţie *unul-la-mulţi* (cont.)

Factori ce pot genera anumite probleme de sincronizare, cu efecte asupra *integrităţii datelor*:

### ■ *destinatarul mesajelor:*

#### — *mesaje cu destinatar oarecare*

Nu ridică probleme deosebite – fiecare mesaj poate fi citit și prelucrat de oricare dintre procesele “cititori”.

#### — *mesaje cu destinatar specificat*

Trebuie asigurat faptul că mesajul este citit exact de către “cititorul” căruia îi era destinat. Soluția – am putea folosi mesaje formataste astfel:

MESAJ = HEADER + MESAJUL PROPRIU-ZIS ,

*header-ul conținând un identificator al destinatarului.*

Pentru citire, s-ar putea aplica protocolul de comunicație discutat la mesaje de lungime variabilă. Însă, apare o *problemă suplimentară*: dacă un “cititor” a citit un mesaj care nu-i era destinat lui, cum facem să-l livrăm celui căruia îi era destinat? O soluție ar fi să îl scrie înapoi în canal, și apoi va face o pauză aleatoare înainte de a încerca să citească din nou din canal. *Notă*: această soluție poate suferi de fenomenul de *starvation*.

Introducere

Canale anone

Canale cu nume (fifo)

Caracteristici comune pentru ambele tipuri de canale

Sabloane de comunicație între procese

Clasificarea şabloanelor de comunicație inter-procese

Şablonul de comunicație *unul-la-unul*

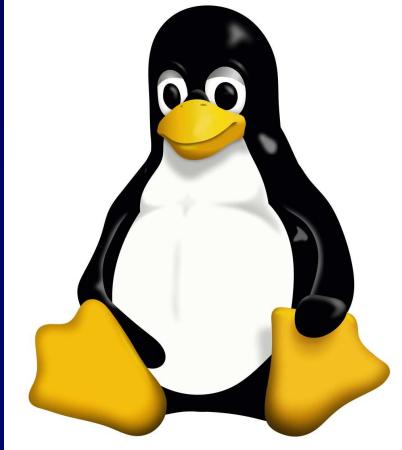
Şablonul de comunicație *unul-la-mulți*

Şablonul de comunicație *mulți-la-unul*

Şablonul de comunicație *mulți-la-mulți*

Aplicații ale canalelor de comunicație

Referințe bibliografice



## Şablonul de comunicaţie *multi-la-unul*

Factori ce pot genera anumite probleme de sincronizare, cu efecte asupra *integrității datelor*:

### ■ *lungimea mesajelor:*

- **mesaje de lungime *constantă***

Nu ridică probleme deosebite – fiecare mesaj poate fi scris *atomic* (i.e., dintr-o dată, printr-un singur apel `write`).

- **mesaje de lungime *variabilă***

Trebuie indicată “cititorului” lungimea fiecărui mesaj. Soluția este folosirea mesajelor formatate astfel:

$$\text{MESAJ} = \text{HEADER} + \text{MESAJUL PROPRIU-ZIS} ,$$

*header-ul* fiind un mesaj de lungime fixă ce conține lungimea mesajului propriu-zis.

Nu ridică probleme deosebite – fiecare mesaj, astfel formatat, poate fi scris *atomic*, printr-un singur apel `write`, deci nu trebuie garantat accesul exclusiv la canal.

Introducere

Canale anone

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

Şabloane de comunicaţie între  
procese

Clasificarea şabloanelor de  
comunicaţie inter-procese

Şablonul de comunicaţie  
*unul-la-unul*

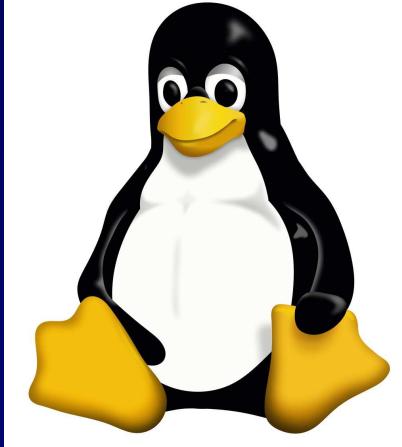
Şablonul de comunicaţie  
*unul-la-multi*

Şablonul de comunicaţie  
*multi-la-unul*

Şablonul de comunicaţie  
*multi-la-multi*

Aplicații ale canalelor de  
comunicație

Referințe bibliografice



## Şablonul de comunicaţie *multi-la-unul* (cont.)

Factori ce pot genera anumite probleme de sincronizare, cu efecte asupra *integrității datelor*:

### ■ *destinatarul* mesajelor:

- **mesaje cu expeditor *oarecare***

Nu ridică probleme deosebite – fiecare mesaj poate fi citit de procesul “cititor” și prelucrat în același fel, indiferent de la care dintre procesele “scriitori” provine acel mesaj.

- **mesaje cu expeditor *specificat***

Trebuie asigurat că mesajul îi indică “cititorului” care este “scriitorul” care i l-a trimis. Solutia – mesaje formataate în felul următor:

MESAJ = HEADER + MESAJUL PROPRIU-ZIS ,

*header*-ul conținând un identificator al expeditorului.

*Notă*: scrierea mesajului astfel formatat se va face prinr-un singur apel write, la fel ca la mesaje de lungime variabilă.

Introducere

Canale anone

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

Şabloane de comunicaţie între  
procese

Clasificarea şabloanelor de  
comunicaţie inter-procese

Şablonul de comunicaţie  
*unul-la-unul*

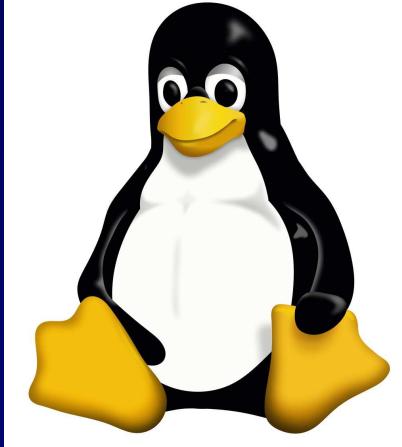
Şablonul de comunicaţie  
*unul-la-multi*

Şablonul de comunicaţie  
*multi-la-unul*

Şablonul de comunicaţie  
*multi-la-multi*

Aplicații ale canalelor de  
comunicație

Referințe bibliografice



## Şablonul de comunicaţie *mulţi-la-mulţi*

Introducere

Canale anone

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

Şabloane de comunicaţie între  
procese

Clasificarea şabloanelor de  
comunicaţie inter-procese

Şablonul de comunicaţie  
*unul-la-unul*

Şablonul de comunicaţie  
*unul-la-mulţi*

Şablonul de comunicaţie  
*mulţi-la-unul*

Şablonul de comunicaţie  
*mulţi-la-mulţi*

Aplicaţii ale canalelor de  
comunicaţie

Referinţe bibliografice

Problemele de sincronizare ce pot apărea în cazul acestui şablon, pot fi cauzate de oricare dintre factorii discutaţi la şabloanele *unul-la-mulţi* și *mulţi-la-unul*:

- *lungimea* mesajelor
- *expeditorul* mesajelor
- *destinatarul* mesajelor

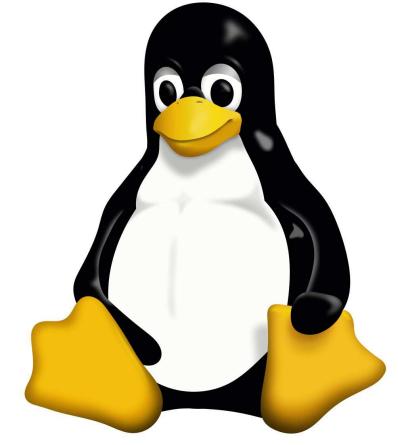
Tratarea acestora se poate face prin combinarea soluţiilor prezentate la şabloanele precedente.

*Notă:* pentru simplitatea programării, uneori se poate prefera înlocuirea unui singur canal folosit pentru comunicaţie *unul-la-mulţi*, cu mai multe canale folosite pentru comunicaţie *unul-la-unul*, i.e. cu câte un canal pentru fiecare proces “cititor” existent.

Evident, se poate proceda similar și pentru şabloanele *unul-la-mulţi* și *mulţi-la-mulţi*.

*Demo:* a se vedea programele `suma_pipes.c` și `suma_fifos.c` ([2]), care reprezintă rescrieri ale programului `suma_files.c` din lecția practică despre fork și wait, prin înlocuirea fișierelor obișnuite cu canale (anone și, respectiv, cu nume) pentru comunicațiile dintre supervisor și workeri.

Comunicațiile dinspre workeri spre supervisor folosesc şablonul *mulţi-la-unul*. În schimb, şablonul *unul-la-mulţi* pentru comunicațiile dinspre supervisor spre workeri l-am implementat pe baza ideii de mai sus. ( Pentru explicații suplimentare despre aceste două programe, puteți consulta exemplele [FifthDemo – suma\_pipes] și [SixthDemo – suma\_fifos] din suportul online de laborator ([3]). )



# Agenda

Introducere

Canale anoneme

Canale cu nume (fifo)

Caracteristici comune pentru ambele tipuri de canale

Şabloane de comunicaţie între procese

Aplicaţii ale canalelor de comunicatie

Aplicaţia #1: implementarea unui semafor

Aplicaţia #2: implementarea unei aplicaţii client/server

Referinţe bibliografice

Introducere

## Canale anone

Crearea lor, cu primitiva pipe

Modul de utilizare a unui canal anonim

*Demo:* exemple de comunicaţie între două procese

## Canale cu nume (fifo)

Crearea lor, cu primitiva mkfifo

Modul de utilizare a unui canal cu nume

Despre persistenţa informaţiei dintr-un fişier fifo

Deosebiri ale canalelor cu nume faţă de cele anone

## Caracteristici comune pentru ambele tipuri de canale

Caracteristici şi restricţii ale canalelor de comunicaţie

Comportamentul implicit, de tip blocant

Comportamentul de tip neblocant

## Şabloane de comunicaţie între procese

Clasificarea şabloanelor de comunicaţie inter-procese

Şablonul de comunicaţie *unul-la-unul*

Şablonul de comunicaţie *unul-la-mulţi*

Şablonul de comunicaţie *mulţi-la-unul*

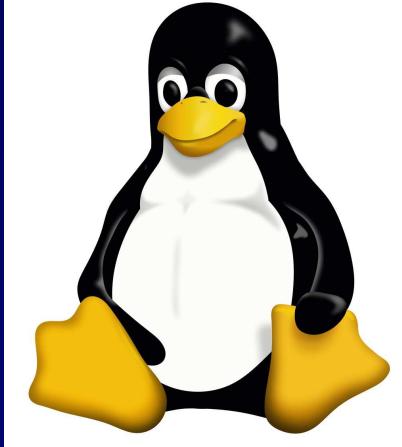
Şablonul de comunicaţie *mulţi-la-mulţi*

## Aplicaţii ale canalelor de comunicatie

Aplicaţia #1: implementarea unui semafor

Aplicaţia #2: implementarea unei aplicaţii client/server

## Referinţe bibliografice



## Aplicația #1: implementarea unui semafor

Introducere

Canale anonime

Canale cu nume (*fifo*)

Caracteristici comune pentru ambele tipuri de canale

Şabloane de comunicație între procese

Aplicații ale canalelor de comunicatie

Aplicația #1: implementarea unui semafor

Aplicația #2: implementarea unei aplicații client/server

Referințe bibliografice

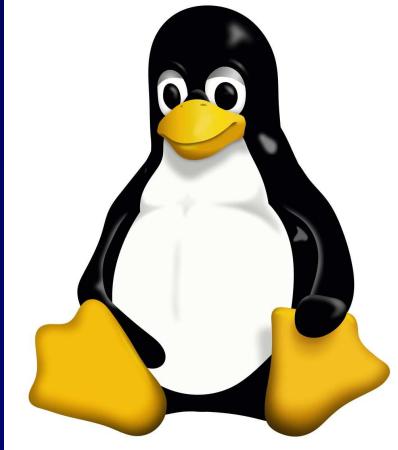
Cum am putea implementa un semafor folosind canale *fifo* ?

O posibilă implementare ar consta în următoarele idei:

**Inițializarea semaforului** s-ar realiza prin crearea unui fișier *fifo* de către un proces cu rol de *supervizor* (acesta poate fi oricare dintre procesele ce vor folosi acel semafor, sau poate fi un proces separat).

Acest proces *supervizor* va scrie inițial în canal 1 octet oarecare, dacă e vorba de un semafor binar (sau  $n$  octeți oarecare, dacă e vorba de un semafor general  $n$ -ar).

Iar apoi va păstra deschise ambele capete ale canalului pe toată durata de execuție a proceselor ce vor folosi acel semafor (cu scopul de a nu se pierde pe parcurs informația din canal, datorită inexistenței la un moment dat a măcar unui proces care să aibă deschis măcar vreunul dintre capete, conform celor discutate anterior legat de perioada de retentie a informației într-un canal *fifo*).



## Aplicația #1: implementarea unui semafor (cont.)

Introducere

Canale anone

Canale cu nume (fifo)

Caracteristici comune pentru ambele tipuri de canale

Sabloane de comunicație între procese

Aplicații ale canalelor de comunicatie

Aplicația #1: implementarea unui semafor

Aplicația #2: implementarea unei aplicații client/server

Referințe bibliografice

**Operația wait va consta în citirea unui octet din fișierul *fifo*.**

Mai precis, întâi se va face deschiderea lui, urmată de citirea efectivă a unui octet, și apoi eventual închiderea fișierului.

**Operația signal va consta în scrierea unui octet în fișierul *fifo*.**

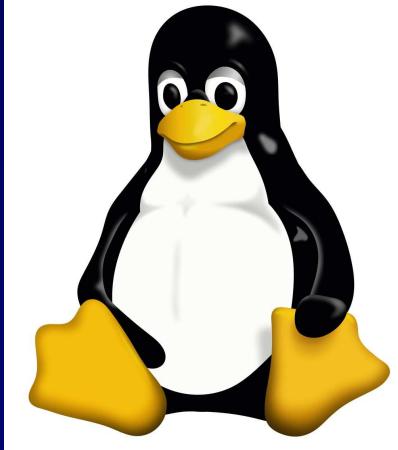
Mai precis, întâi se va face deschiderea lui, urmată de scrierea efectivă a unui octet, și apoi eventual închiderea fișierului.

*Observații:*

i) citirea se va face, în modul implicit, *blocant*, ceea ce va asigura așteptarea procesului la punctul de intrare în secțiunea sa critică atunci când semaforul este “pe roșu”, adică dacă canalul *fifo* este gol.

ii) scrierea nu se va putea bloca (cu condiția ca  $n$ -ul semaforului general să nu depășească capacitatea maximă pe care o putem configura pentru un canal).

*Temă:* implementați în C un semafor binar pe baza ideilor de mai sus și scrieți un program demonstrativ în care să utilizați semaforul astfel implementat pentru asigurarea excluderii mutuale a unei secțiuni critice de cod (pentru “inspirație” în scrierea programului demonstrativ, revedeți problemele de sincronizare discutate în cursurile teoretice #5 și #6).



## Aplicația #2: implementarea unei aplicații client/server

Introducere

Canale anone

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

Şabloane de comunicație între  
procese

Aplicații ale canalelor de  
comunicatie

Aplicația #1: implementarea  
unui semafor

Aplicația #2: implementarea  
unei aplicații client/server

Referințe bibliografice

O *aplicație cu arhitectură de tip client/server* este compusă din două componente:

- **serverul**: este un program care dispune de un anumit număr de *servicii* (i.e., funcții, operații, etc.), pe care le pune la dispoziția clientilor.
- **clientul**: este un program care “interroghează” serverul, solicitându-i *efectuarea unui serviciu* (dintre cele puse la dispoziție de acel server).

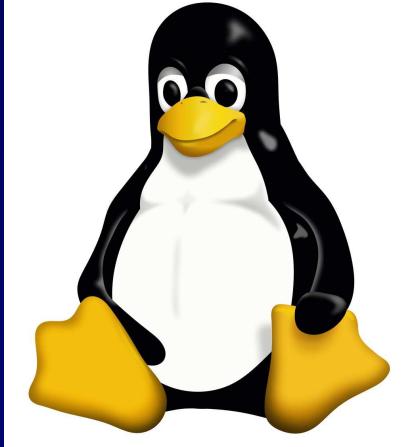
*Exemplu*: Browserele pe care le folosiți pentru a naviga pe INTERNET sunt un exemplu de program client, care se conectează la un program server, numit *server de web*, solicitându-i transmiterea unei pagini *web*, care apoi este afișată în fereastra grafică a browserului.

*Implementarea unei aplicații de tip client/server* o puteți face în felul următor:

Programul server va fi rulat în *background*, și va sta în așteptarea cererilor din partea clientilor, putând servi mai mulți clienti simultan.

Iar clientii vor putea fi rulați mai mulți simultan (din același cont și/sau din conturi utilizator diferite), și se vor “conecta” la serverul rulat în *background*.

*Notă*: pot exista, la un moment dat, mai multe procese client care încearcă, fiecare independent de celelalte, să folosească serviciile puse la dispoziție de același proces server.



## Aplicația #2: implementarea unei aplicații client/server (cont.)

Introducere

Canale anone

Canale cu nume (fifo)

Caracteristici comune pentru ambele tipuri de canale

Şabloane de comunicație între procese

Aplicații ale canalelor de comunicatie

Aplicația #1: implementarea unui semafor

Aplicația #2: implementarea unei aplicații client/server

Referințe bibliografice

*Observație:* în realitate, programul server este rulat pe un anumit calculator, iar clienții pe diverse alte calculatoare, conectate la INTERNET, comunicația realizându-se folosind *socket-uri*, prin intermediul rețelelor de calculatoare.

Însă puteți simula această “realitate” folosind **comunicație prin canale cu nume și executând toate procesele (i.e., serverul și clientii) pe un același calculator**, eventual din conturi utilizator diferite.

Tipurile de servere existente în realitate, d.p.d.v. al *servirii “simultane” a mai multor clienti*, se împart în două categorii:

### ■ **server *iterativ***

Cât timp durează efectuarea unui serviciu (i.e., rezolvarea unui client), serverul este blocat: nu poate răspunde cererilor venite din partea altor clienți. Deci nu poate rezolva mai mulți clienți în același timp!

### ■ **server *concurrent***

Pe toată durata de timp necesară pentru efectuarea unui serviciu (i.e., rezolvarea unui client), serverul nu este blocat, ci poate răspunde cererilor venite din partea altor clienți. Deci poate rezolva mai mulți clienți în același timp!



## Aplicația #2: implementarea unei aplicații client/server (cont.)

Introducere

Canale anonte

Canale cu nume (fifo)

Caracteristici comune pentru  
ambele tipuri de canale

Şabloane de comunicație între  
procese

Aplicații ale canalelor de  
comunicatie

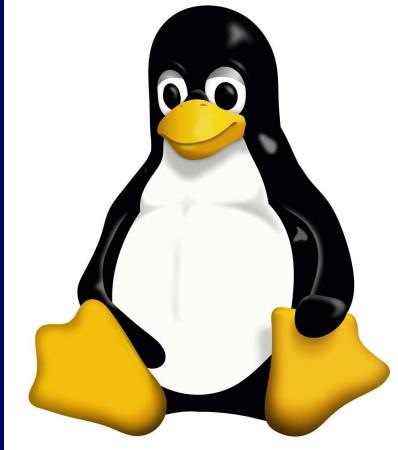
Aplicația #1: implementarea  
unei semafor

Aplicația #2: implementarea  
unei aplicații client/server

Referințe bibliografice

### Detalii legate de implementare:

- Pentru implementarea unui server de tip iterativ este suficient un singur proces secvențial. În schimb, pentru implementarea unui server de tip concurrent este nevoie de mai multe procese secvențiale: un proces *supervisor*, care așteaptă sosirea cererilor din partea clientilor, și la fiecare cerere sosită, el va crea un nou proces fiu, un *worker* care va fi responsabil cu rezolvarea propriu-zisă a clientului respectiv, iar *supervisor-ul* va relua imediat așteptarea sosirii unei noi cereri, fără să aștepte terminarea procesului fiu. (Sau, alternativ, se poate implementa printr-un singur proces *multi-threaded*.)
- Pentru comunicarea între procesele client și procesul server este necesar să se utilizeze, drept canale de comunicație, fișiere *fifo*. (*Motivul*: nu se pot folosi canale anonte deoarece procesul server și procesele clienti nu sunt înrudite prin fork/exec.)
- Permisiiile fișierelor *fifo* folosite pentru comunicație trebuie configurate adekvat, astfel încât să permită execuția proceselor client din *conturi utilizator diferite* (!).



## Aplicația #2: implementarea unei aplicații client/server (cont.)

Introducere

Canale anone

Canale cu nume (fifo)

Caracteristici comune pentru ambele tipuri de canale

Şabloane de comunicație între procese

Aplicații ale canalelor de comunicatie

Aplicația #1: implementarea unui semafor

Aplicația #2: implementarea unei aplicații client/server

Referințe bibliografice

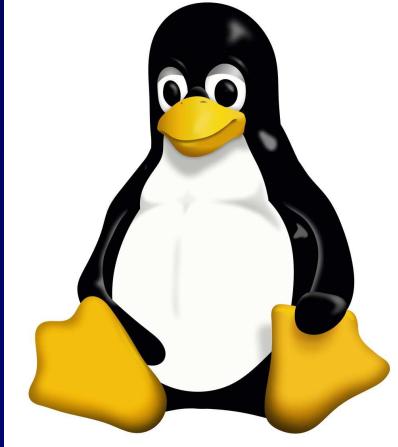
- Un alt aspect legat tot de comunicație: serverul nu cunoaște în avans clientii ce se vor conecta la el pentru a le oferi servicii, în schimb clientul trebuie să cunoască serverul la care se va conecta pentru a beneficia de serviciul oferit de el.

Ce înseamnă aceasta d.p.d.v. practic ?

Serverul va crea un canal *fifo* cu un nume fixat, cunoscut în programul client, și va aștepta sosirea informațiilor pe acest canal.

Un client oarecare se va conecta la acest canal *fifo* cunoscut și va transmite informații de identificare a sa, care vor fi folosite ulterior pentru realizarea efectivă a comunicațiilor implicate de serviciul solicitat ( cel mai probabil va fi nevoie să utilizați canale suplimentare, particolare pentru acel client, pentru a nu se “amesteca” între ele comunicațiile destinate unui client cu cele destinate altui client conectat la server în același timp cu primul ).

*Temă:* implementați un joc *multi-player* “în rețea”, pe baza ideilor descrise mai sus.



# Bibliografie obligatorie

[Introducere](#)

[Canale anonime](#)

[Canale cu nume \(\*fifo\*\)](#)

[Caracteristici comune pentru ambele tipuri de canale](#)

[Şabloane de comunicaţie între procese](#)

[Aplicaţii ale canalelor de comunicaţie](#)

[Referinţe bibliografice](#)

- [1] Capitolul 5, §5.1, §5.2, §5.3 și §5.5 din cartea “Sisteme de operare – manual pentru ID”, autor C. Vidrăscu, editura UAIC, 2006. Acest manual este accesibil, în format PDF, la adresa:
- <https://profs.info.uaic.ro/~vidrascu/S0/books/ManualID-S0.pdf>
- [2] Programele demonstrative amintite pe parcursul acestei prezentări pot fi descărcate de la adresele:
- <https://profs.info.uaic.ro/~vidrascu/S0/cursuri/C-programs/pipe/>
  - <https://profs.info.uaic.ro/~vidrascu/S0/cursuri/C-programs/fifo/>
- [3] Suportul online de laborator asociat acestei prezentări:
- [https://profs.info.uaic.ro/~vidrascu/S0/labs/suport\\_lab12.html](https://profs.info.uaic.ro/~vidrascu/S0/labs/suport_lab12.html)
- [4] Capitolul 44 din cartea “The Linux Programming Interface : A Linux and UNIX System Programming Handbook”, autor M. Kerrisk, editura No Starch Press, 2010. Cartea este accesibilă la adresa:
- <https://profs.info.uaic.ro/~vidrascu/S0/books/TLPI1.pdf>
- [5] POSIX API: `man 2 pipe`, `man 2 mkfifo`, `man 2 fcntl`.

# **Programare concurentă în C (VII) :**

## *Gestiunea proceselor, partea a III-a: Semnale UNIX*

Cristian Vidrașcu

[vidrascu@info.uaic.ro](mailto:vidrascu@info.uaic.ro)

# Sumar

- Introducere
- Categorii de semnale
- Tipurile de semnale predefinite ale UNIX-ului
- Cererea explicită de generare a unui semnal
  - primitiva `kill`
- Crearea semnalelor – primitiva `signal`
- Definirea propriilor *handle*re de semnal
- Blocarea semnalelor
- Așteptarea unui semnal

# Introducere

Semnalele UNIX reprezintă un mecanism fundamental de manipulare a proceselor și de comunicare între procese, ce asigură tratarea evenimentelor asincrone apărute în sistem.

Un **semnal UNIX** este o *întrerupere software* generată în momentul producerii unui anumit eveniment și transmisă de sistemul de operare unui anumit proces.

# Introducere (cont.)

Un semnal este *generat* de apariția unui eveniment excepțional (care poate fi o eroare, un eveniment extern sau o cerere explicită).

Orice semnal are asociat un *tip*, reprezentat printr-un număr întreg pozitiv (ce codifică cauza sa), și un proces *destinatar*.

Odată generat, semnalul este pus în *coada de semnale* a sistemului, de unde este extras și transmis procesului destinatar de către sistemul de operare.

*Transmiterea* semnalului către destinatar se face imediat după ce semnalul a ajuns în coada de semnale, cu o excepție: dacă primirea semnalelor de tipul respectiv a fost *blocată* de către procesul destinatar, atunci transmiterea semnalului se va face abia în momentul când procesul destinatar va debloca primirea acelui tip de semnal.

# Introducere (cont.)

În momentul în care procesul destinatar primește acel semnal, el își *întrerupe execuția* și va executa o anumită acțiune (i.e., o funcție de tratare a aceluia semnal, funcție numită *handler de semnal*) care este atașată tipului de semnal primit, după care procesul își va relua execuția din punctul în care a fost îintrerupt (cu anumite excepții: unele semnale vor cauza terminarea forțată a aceluui proces).

În concluzie, fiecare tip de semnal are asociat o acțiune (un *handler*) specifică aceluui tip de semnal.

# Categorii de semnale

Evenimentele ce genereaza semnale se împart în trei categorii:

- **erori (în procesul destinatar)**

O **eroare** înseamnă că programul a făcut o operație invalidă și nu poate să-și continue execuția. Nu toate erorile generează semnale, ci doar acele erori care pot apărea în orice punct al programului, cum ar fi: împărțirea la zero, accesarea unei adrese de memorie invalide, etc.

- **evenimente externe (procesului destinatar)**

**Evenimentele externe** sunt în general legate de operațiile I/O sau de acțiunile altor procese, cum ar fi: sosirea datelor (pe un *socket* sau *pipe*), terminarea unui proces fiu, expirarea intervalului de timp setat pentru o alarmă, sau suspendarea ori terminarea programului de către utilizator (prin apăsarea tastelor ^Z ori ^C).

- **cereri explicite**

O **cerere explicită** înseamnă generarea unui semnal de către un (alt) proces, prin apelul primitivei `kill`.

# Categorii de semnale (cont.)

Semnalele pot fi generate **sincron** sau **asincron**.

- Un semnal **sincron** este generat de o anumită acțiune specifică în program și este livrat (dacă nu este blocat) în timpul acelei acțiuni.
  - Evenimente ce generează semnale sincrone: erorile și cererile explicite ale unui proces de a genera semnale pentru el însuși.
- Un semnal **asincron** este generat de un eveniment din afara zonei de control a procesului care îl recepționează; cu alte cuvinte, un semnal ce este recepționat, în timpul execuției procesului destinatar, la un moment de timp ce nu poate fi anticipat.
  - Evenimente ce generează semnale asincrone: evenimentele externe și cererile explicite ale unui proces de a genera semnale destinate altor procese.

# Categorii de semnale (cont.)

Pentru fiecare tip de semnal există o acțiune implicită de tratare a aceluia semnal, specifică sistemului de operare UNIX respectiv. Această acțiune este denumită *handlerul implicit de semnal* atașat aceluui tip de semnal.

Atunci când semnalul este livrat procesului, acesta este întrerupt și are trei posibilități de comportare:

- fie să execute această acțiune implicită,
- fie să ignore semnalul,
- fie să execute o anumită funcție *handler* utilizator.

Setarea unuia dintre cele trei comportamente se face cu ajutorul apelului primitivelor `signal` sau `sigaction`.

## Tipurile de semnale predefinite ale UNIX-ului

Tipurile predefinite de semnale din UNIX se clasifică în mai multe categorii:

- semnale standard de eroare: SIGFPE, SIGILL, SIGSEGV, SIGBUS
- semnale de terminare a proceselor: SIGHUP, SIGINT, SIGQUIT, SIGTERM, SIGKILL
- semnale de alarmă: SIGALRM, SIGVTALRM, SIGPROF
- semnale asincrone I/O: SIGIO, SIGURG
- semnale pentru controlul proceselor: SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU
- alte tipuri de semnale: SIGPIPE, SIGUSR1, SIGUSR2

## Tipurile de semnale predefinite ale UNIX-ului (cont.)

Lista semnalelor UNIX predefinite, mai exact numărul întreg asociat fiecărui tip de semnal, poate fi obținută cu comanda următoare:

```
UNIX> kill -l
```

iar pagina de manual ce conține descrierea semnalelor este:

```
UNIX> man 7 signal
```

*Observație:* o parte dintre aceste tipuri de semnale depind și de suportul oferit de partea de *hardware* a calculatorului respectiv, nu numai de sistemul de operare de pe acel calculator. Din acest motiv, există mici deosebiri în modul de implementare a acestor semnale pe diferite tipuri de arhitecturi de calculatoare (adică unele semnale se poate să nu fie implementate deloc, sau să fie implementate cu mici diferențe).

Exemple de semnale ce pot差别 de la un tip de arhitectură la altul: cele generate de erori, cum ar fi SIGBUS (care nu este implementat în Linux-ul pentru *hardware*-ul i386). În concluzie: trebuie studiată documentația tipului de calculator pe care îl utilizați pentru a vedea ce semnale aveți la dispoziție.

## Cererea explicită de generare a unui semnal – primitiva kill

Apelul de sistem `kill` este utilizat pentru a cere explicit generarea unui semnal. Interfața acestei funcții:

```
int kill ( int pid, int id-signal )
```

- *pid* = PID-ul procesului destinatar
- *id-signal* = tipul semnalului
- valoarea returnată este 0, în caz de reușită, sau -1, în caz de eroare.

Efect: în urma execuției funcției `kill` se generează un semnal de tipul specificat, destinat procesului specificat.

## Cererea explicită de generare a unui semnal – primitiva kill

*Observație:* prin apelul `kill(pid, 0)`; nu se trimite nici un semnal, dar este util pentru verificarea validității PID-ului respectiv (i.e., dacă există un proces cu acel PID în momentul apelului, sau nu): se returnează 0 dacă PID-ul specificat este valid, sau -1, în caz contrar.

Pentru cererea explicită de generare a unui semnal se poate folosi și comanda `kill`:

```
UNIX> kill -semnal pid
```

Un proces își poate trimite semnale și însuși folosind funcția `raise`, ce are interfața:

```
int raise(int id-signal)
```

Efect: este echivalent cu apelul `kill(getpid(), id-signal);`.

## Coruperea semnalelor – primitiva signal

Acțiunea asociată unui semnal poate fi:

- o acțiune implicită (specifică sistemului de operare respectiv),
- sau ignorarea semnalului,
- sau un *handler* propriu, definit de programator.

Se utilizează termenul de *corupere a unui semnal* cu sensul de: setarea unui *handler* propriu pentru acel tip de semnal.

*Notă:* uneori, se folosește și termenul de *tratare a semnalului*.

*Observație:* semnalele SIGKILL și SIGSTOP nu pot fi corupte, ignoreate sau blocate!

## Coruperea semnalelor – primitiva `signal` (cont.)

Specificarea acțiunii asociate unui semnal se poate face cu apelurile de sistem `signal` sau `sigaction`.

Interfața primitivei `signal` este:

```
sighandler_t signal (int id-signal, sighandler_t action)
```

- *id-signal* = tipul semnalului căruia î se asociază acea acțiune
- *action* = acțiunea (*i.e.*, *handler*ul de semnal) ce se asociază semnalului; poate fi numele unei funcții definite de programator, sau poate lua una dintre valorile:
  - SIG\_DFL : specifică acțiunea implicită (cea stabilită de către sistemul de operare) la recepționarea semnalului
  - SIG\_IGN : specifică faptul că procesul va ignora acel semnal
- valoarea returnată este vechiul *handler* pentru semnalul specificat, sau constanta simbolică SIG\_ERR în caz de eroare.

## Coruperea semnalelor – primitiva signal (cont.)

Interfața primitivei signal este:

```
sighandler_t signal (int id-signal, sighandler_t action)
```

Efect: se asociază *handlerul* specificat pentru acel tip de semnal.

Ca urmare, ulterior (până la o nouă recorupere), ori de câte ori procesul va receptiona semnalul *id-signal*, se va executa *handlerul* de semnal *action*.

*Observație:* în general nu este bine ca programul să ignore semnalele (mai ales pe acelea care reprezintă evenimente importante). Dacă se dorește ca programul să nu receptioneze semnale în timpul executiei unei anumite porțiuni de cod (pentru a nu fi întreruptă), soluția cea mai indicată este să se *blocheze* primirea semnalelor, nu ca ele să fie ignorate.

## Coruperea semnalelor – primitiva signal (cont.)

Interfața primitivei signal este:

```
sighandler_t signal (int id-signal, sighandler_t action)
```

Dacă argumentul *action* este numele unei funcții definite de utilizator, această funcție trebuie să aibă prototipul `sighandler_t`, definit astfel:

```
typedef void (*sighandler_t)(int);
```

i.e., tipul “funcție ce întoarce tipul `void`, și are un argument de tip `int`”.

*Notă:* la momentul executiei unui *handler* de semnal, acest argument va avea ca valoare numărul semnalului ce a determinat execuția acelui *handler*. În acest fel, se poate asigna o aceeași funcție ca și *handler* pentru mai multe semnale, în corpul ei putând ști, pe baza argumentului primit, care dintre acele semnale a cauzat apelul respectiv.

## Coruperea semnalelor – primitiva `signal` (cont.)

Exemplu: un program care să ignore îintreruperile de tastatură, adică semnalul SIGINT (generat de tastele CTRL+C) și semnalul SIGQUIT (generat de tastele CTRL+\ ).

A se vedea programul `sig-ex1.c` fără ignorarea celor două semnale (i.e., poate fi întrerupt/oprit cu CTRL+C, respectiv CTRL+\ ), și respectiv programul `sig-ex2.c` cu ignorarea celor două semnale (i.e., va rula fără a putea fi întrerupt/oprit cu CTRL+C, respectiv CTRL+\ ).

Să modificăm exemplul anterior astfel: corupem semnalele să execute un *handler* propriu, care să afișeze un anumit mesaj. Iar apoi refacem comportamentul implicit al semnalelor.

A se vedea programul `sig-ex3.c`

## Definirea propriilor *handler* de semnal

Un *handler* de semnal propriu este o funcție definită de programator, ce va fi apelată atunci când procesul recepționează semnalul căruia îi este asociată.

Strategii principale folosite în scrierea de *handler* proprii:

- Se poate ca *handlerul* să notifice primirea semnalului prin setarea unei variabile globale și apoi să returneze imediat, urmând ca în bucla principală a programului, să se verifice periodic dacă acea variabilă a fost setată, în care caz se vor efectua operațiile dorite.
- Se poate ca *handlerul* să termine execuția procesului, sau să transfere execuția într-un punct în care procesul poate să-si recupereze starea în care se afla în momentul recepționării semnalului.

## Definirea propriilor *handle*re de semnal (cont.)

*Atenție:* trebuie luate măsuri speciale atunci când se scrie codul pentru *handler*ele de semnal, deoarece acestea pot fi apelate asincron, adică la momente de timp imprevizibile.

Spre exemplu, în timp ce se execută *handler*ul asociat unui semnal primit, acesta poate fi întrerupt prin receptia unui alt semnal (al doilea semnal trebuie să fie de alt tip decât primul; dacă este același tip de semnal, el va fi blocat până când se termină tratarea primului semnal).

*Important:* prin urmare, primirea unui semnal poate întrerupe nu doar execuția programului respectiv, ci chiar execuția *handler*ului unui semnal anterior primit, sau poate întrerupe execuția unui apel de sistem efectuat de program în acel moment.

# Blocarea semnalelor

*Blocarea semnalelor* înseamnă că procesul spune sistemului de operare să nu îi transmită anumite semnale (ele vor rămâne în coada de semnale, până când procesul va debloca primirea lor).

*Notă:* nu este recomandabil ca un program să blocheze semnalele pe tot parcursul execuției sale, ci numai pe durata execuției unor porțiuni critice ale codului său. Astfel, dacă un semnal ajunge în timpul execuției acelei porțiuni de program, el va fi livrat procesului abia după terminarea execuției acesteia și deblocarea acelui tip de semnal.

# Blocarea semnalelor

*Blocarea semnalelor* înseamnă că procesul spune sistemului de operare să nu îi transmită anumite semnale (ele vor rămâne în coada de semnale, până când procesul va debloca primirea lor).

Blocarea semnalelor se realizează cu funcția `sigprocmask`, ce utilizează structura de date `sigset_t` (care este o mască de biți, cu semnificația de set de semnale ales pentru blocare).

Cu primitiva `sigpending` se poate verifica existența, în coada de semnale, a unor semnale blocate (deci care așteaptă să fie deblocate pentru a putea fi livrate procesului).

Exemplu: a se vedea fișierul sursă `sig-ex4.c`

# Așteptarea unui semnal

Dacă aplicația este influențată de evenimente externe, sau folosește semnale pentru sincronizare cu alte procese, atunci ea nu trebuie să facă altceva decât să aștepte semnale.

Se poate folosi în acest scop funcția `pause`, ce are prototipul:

```
int pause( )
```

Efect: suspendarea execuției programului până la sosirea unui semnal.

*Observație:* simplitatea acestei funcții poate ascunde erori greu de detectat. Deoarece programul principal nu face altceva decât să apeleze `pause()`, înseamnă că cea mai mare parte a activității utile în program o realizează *handlelele de semnal*. Însă, codul acestor *handle* nu este indicat să fie prea lung, deoarece poate fi întrerupt de alte semnale.

# Așteptarea unui semnal (cont.)

Modalitatea cea mai indicată, pentru așteptarea unui anumit semnal (*i.e.*, așteptarea primului semnal primit, dintr-o mulțime fixată de semnale), este de a folosi funcția `sigsuspend`, ce are prototipul:

```
int sigsuspend(const sigset_t *set)
```

Efect: se înlocuiește masca de semnale curentă a procesului cu cea specificată de parametrul *set* și apoi se suspendă execuția procesului până la recepționarea unui semnal, de către proces (deci un semnal care nu este blocat, adică nu este cuprins în masca de semnale curentă).

Masca de semnale rămâne la valoarea setată (*i.e.*, valoarea lui *set*) numai până când funcția `sigsuspend()` returnează, moment în care este reinstalată, în mod automat, vechea mască de semnale.

# Așteptarea unui semnal (cont.)

Modalitatea cea mai indicată, pentru așteptarea unui anumit semnal (i.e., așteptarea primului semnal primit, dintr-o mulțime fixată de semnale), este de a folosi funcția `sigsuspend`, ce are prototipul:

```
int sigsuspend(const sigset_t *set)
```

Valoarea returnată: 0, în caz de succes, respectiv -1, în caz de eșec (iar variabila `errno` este setată în mod corespunzător: EINVAL, EFAULT sau EINTR).

Exemplu: un program care își suspendă execuția în așteptarea semnalului SIGQUIT (generat de tastele CTRL+\ ), fără a fi întrerupt de alte semnale. A se vedea fișierul sursă `sig-ex5.c`

# Bibliografie obligatorie

Cap.4, §4.5 din manualul, în format PDF, accesibil din pagina disciplinei “Sisteme de operare”:

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

Programele demonstrative amintite pe parcursul acestei prezentări pot fi descărcate de la adresa următoare:

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

Întrebări?

# Programare concurentă în C (VIII)

*Gestiunea terminalelor:*

*Biblioteca NCURSES – ferestre ecran în mod text*

Cristian Vidrașcu

[vidrascu@info.uaic.ro](mailto:vidrascu@info.uaic.ro)

# Sumar

- Vedere generală asupra bibliotecii NCURSES
- Principalele categorii de funcții din NCURSES
- Exemple de lucru cu biblioteca NCURSES
- Lucrul de nivel scăzut cu terminalul

## Vedere generală asupra bibliotecii NCURSES

Biblioteca NCURSES pune la dispoziția utilizatorilor o metodă de gestiune a ecranelor în mod text, independentă de terminal și cu optimizări rezonabile.

Ideea: ecranul terminalului este “separat” în două concepte:

- *ecranul fizic*  
reprezintă ceea ce vede la un moment dat un utilizator pe ecranul terminalului fizic (*i.e.*, pe ecranul monitorului);
- *ecranul virtual*  
reprezintă ceea ce ar fi văzut utilizatorul dacă ecranul terminalului fizic ar fi reflectat întocmai toate operațiile de I/O executate de el prin intermediul bibliotecii NCURSES.

## Vedere generală asupra bibliotecii NCURSES (cont.)

Ecranul virtual este o imagine în memoria calculatorului a ecranului fizic.

Toate operațiile de I/O executate de utilizator prin intermediul bibliotecii NCURSES, se execută asupra acestei imagini în memorie, fără a se sincroniza cu imaginea de pe ecranul fizic.

Sincronizarea celor două ecrane (*i.e.*, “copierea” imaginii ecranului virtual pe ecranul fizic) se execută în mod explicit prin operatia de *refresh*.

## Vedere generală asupra bibliotecii NCURSES (cont.)

Această bibliotecă permite lucrul cu **ferestre în mod text**.

O **fereastră** este o zonă dreptunghiulară de pe ecran, care este păstrată în propria sa zonă de memorie. Pentru aceasta se utilizează o structură de date numită **WINDOW**.

*Observație:* ferestrele au alocate zone de memorie distincte, chiar dacă imaginile lor pe ecran se suprapun (parțial sau total).

Toate operațiile de I/O și operația de *refresh* se execută asupra ferestrei specificate ca argument al operației respective.

Operația de *refresh* a unei ferestre presupune următoarele două faze:

- “copierea” ferestrei pe ecranul virtual,
- “copierea” ecranului virtual pe ecranul fizic.

## Vedere generală asupra bibliotecii NCURSES (cont.)

Schema generală a unui program care utilizează biblioteca NCURSES:

```
#include <ncurses.h>

int main( )
{
    ...

    initscr(); // inițializare mod de lucru ncurses
    while( ! gata() )
    {
        executa_operatii_IO(); // op. I/O asupra imaginii din memorie
        executa_refresh(); // "copie" ecranul virtual pe cel fizic
    }
    endwin(); // sfârșit mod de lucru ncurses
    ...
}
```

## Vedere generală asupra bibliotecii NCURSES (cont.)

Compilarea unui program care utilizează biblioteca NCURSES:

```
UNIX> gcc sursa.c -lncurses [ -o executabil ]
```

Pagina de manual principală ce descrie biblioteca NCURSES:

```
UNIX> man ncurses
```

Această pagină conține o descriere de ansamblu a bibliotecii, împreună cu lista tuturor funcțiilor disponibile în această bibliotecă.

*Atenție:* lista cuprinde numele fiecărei funcții, împreună cu *keyword*-ul ce trebuie folosit ca parametru în comanda `man` pentru a vizualiza pagina de manual pentru funcția respectivă.

Astfel, spre exemplu, funcțiile `initscr()` și `endwin()`, cu care se initializează, respectiv se sfîrșește, lucrul în mod ncurses, sunt descrise pe pagina de manual:

```
UNIX> man curs_initscr
```

## Vedere generală asupra bibliotecii NCURSES (cont.)

- O fereastră este practic o matrice bidimensională de caractere, având tipul `chtype`  $\equiv$  `char + attribute mod video + o pereche de culori (ink & paper colors)`.
- Majoritatea funcțiilor bibliotecii NCURSES care operează cu ferestre, sunt definite cu două forme:

`wfunctie(WINDOW *, ...)` și `functie(...)`,  
cea de-a doua formă fiind un macro care apelează prima formă  
pentru fereastra standard `stdscr` (`stdscr` este o variabilă globală a  
bibliotecii care identifică o fereastră ce “acoperă” întregul ecran).

## Vedere generală asupra bibliotecii NCURSES (cont.)

- Majoritatea funcțiilor bibliotecii NCURSES care operează cu ferestre, sunt definite cu două forme:

*wfuncție*( WINDOW \* , . . . ) și *funcție*( . . . ),  
cea de-a doua formă fiind un macro care apelează prima formă  
pentru fereastra standard `stdscr` (`stdscr` este o variabilă globală a  
bibliotecii care identifică o fereastră ce “acoperă” întregul ecran).

- Majoritatea funcțiilor bibliotecii care operează (scriu sau citesc) la  
poziția cursorului în ferestre, sunt definite cu două forme:

*funcție*( . . . ) și *mvfuncție*( . . . , int `y`, int `x`),  
cea de-a doua formă fiind un macro care întâi mută cursorul la  
poziția (`y`, `x`), și apoi apelează prima formă, ce realizează operația  
respectivă la poziția curentă a cursorului.

## Principalele categorii de funcții din NCURSES

- operații de initializare / terminare a modului de lucru NCURSES:  
`initscr`, `endwin`
- operația de *refresh* a ferestrelor: `[w]refresh`
- operația de “*touch*” a ferestrelor (prin care se “scurtcircuitează” mecanismul de optimizare a operației de *refresh*): `touchwin`, `touchline`
- operații de creare / distrugere a ferestrelor: `newwin`, `delwin`
- operații pentru poziția cursorului în fereastră: `getyx`, `[w]move`
- operații pentru desenare de margini interioare și de linii orizontale și verticale în ferestre: `[w]border`, `box`, `[w]hline`, `[w]vline`
- operații de citire scriere a unui caracter de pe/pe fereastră:  
`[mv][w]addch`, `[w]echochar`, `[mv][w]inch`

# Principalele categorii de funcții din NCURSES

- operații de citire scriere a unui sir de caractere de pe/pe fereastră:  
`[mv][w]add[n]str, [mv][w]in[n]str`
- operații de citire scriere formatare de pe/pe fereastră:  
`[mv][w]printw, [mv][w]scanw`
- operația de stergere a ferestrelor: `[w]clear`
- operații de inserare extractie a unui caracter (la nivel de linie):  
`[mv][w]insch, [mv][w]delch`
- operații de inserare extractie a unei linii (la nivel de fereastră):  
`[w]insertln, [w]deleteln, [w]insdelln`
- operații de citire de la tastatura terminalului: `[mv][w]getch,`  
`[mv][w]get[n]str`

## Principalele categorii de funcții din NCURSES

- operații de control a caracteristicilor legăturii de intrare a terminalului:  
`echo`, `noecho`, `cbreak`, `nocbreak`, `keypad`, s.a.
- operații de control a caracteristicilor legăturii de ieșire a terminalului:  
`nl`, `nonl`, `scrolllok`, `[w]setscreg`, s.a.
- operații de manipulare a atributelor și culorilor caracterelor:  
`[w]attron`, `[w]attroff`, `[w]attrset`, `[w]color_set`,  
`start_color`, `init_pair`, s.a.
- operații de copiere de ferestre: `overlay`, `overwrite`, `copywin`
- s.a.

# Exemple de lucru cu NCURSES

- **Exemplul 1:** programul `test1.c` ilustrează lucrul cu principalele funcții de afișare, folosind doar fereastra standard `stdscr`.

# Exemple de lucru cu NCURSES

- **Exemplul 1:** programul `test1.c` ilustrează lucrul cu principalele funcții de afișare, folosind doar fereastra standard `stdscr`.
- **Exemplul 2:** programul `test2.c` ilustrează lucrul cu ferestre suprapuse și fenomenele ce pot apărea datorită optimizărilor folosite de operația de *refresh*.

# Exemple de lucru cu NCURSES

- **Exemplul 1:** programul `test1.c` ilustrează lucrul cu principalele funcții de afișare, folosind doar fereastra standard `stdscr`.
- **Exemplul 2:** programul `test2.c` ilustrează lucrul cu ferestre suprapuse și fenomenele ce pot apărea datorită optimizărilor folosite de operația de *refresh*.
- **Exemplul 3:** programele `test3.c` și `test3b.c` ilustrează lucrul cu principalele funcții de afișare, folosind o fereastră definită și fereastra standard `stdscr`; al doilea program ilustrează și o ieșire temporară din modul ncurses.

# Exemple de lucru cu NCURSES

- **Exemplul 1:** programul `test1.c` ilustrează lucrul cu principalele funcții de afișare, folosind doar fereastra standard `stdscr`.
- **Exemplul 2:** programul `test2.c` ilustrează lucrul cu ferestre suprapuse și fenomenele ce pot apărea datorită optimizărilor folosite de operația de *refresh*.
- **Exemplul 3:** programele `test3.c` și `test3b.c` ilustrează lucrul cu principalele funcții de afișare, folosind o fereastră definită și fereastra standard `stdscr`; al doilea program ilustrează și o ieșire temporară din modul ncurses.
- **Exemplul 4:** programul `test4.c` ilustrează lucrul cu attribute și culori, folosind fereastra standard `stdscr`.

# Exemple de lucru cu NCURSES (cont.)

- **Exemplul 5:** programul `test5.c` ilustrează lucrul cu *background* și culori pentru ferestre.

# Exemple de lucru cu NCURSES (cont.)

- **Exemplul 5:** programul `test5.c` ilustrează lucrul cu *background* și culori pentru ferestre.
- **Exemplul 6:** programul `test6.c` ilustrează lucrul cu zone de *scroll* pentru ferestre.

# Exemple de lucru cu NCURSES (cont.)

- **Exemplul 5:** programul `test5.c` ilustrează lucrul cu *background* și culori pentru ferestre.
- **Exemplul 6:** programul `test6.c` ilustrează lucrul cu zone de *scroll* pentru ferestre.
- **Exemplul 7:** programul `test7.c` ilustrează lucrul cu *input* interactiv de la tastatură.

# Exemple de lucru cu NCURSES (cont.)

- **Exemplul 5:** programul `test5.c` ilustrează lucrul cu *background* și culori pentru ferestre.
- **Exemplul 6:** programul `test6.c` ilustrează lucrul cu zone de *scroll* pentru ferestre.
- **Exemplul 7:** programul `test7.c` ilustrează lucrul cu *input* interactiv de la tastatură.
- **Exemplul 8:** programul `menu.c` + `menu.h` ilustrează o mică aplicație interactivă cu meniuri, capabilă să răspundă la apăsarea tastelor săgeți, ENTER, F1 sau H (pentru *help*), și Q (pentru *quit*).

Temă: scrieți un program de administrare a fișierelor, asemănător cu mc-ul.

# Lucrul de nivel scăzut cu terminalul

- Comenzi UNIX referitoare la terminal:
  - Aflarea terminalului la care ești conectat:  
UNIX> `tty`
  - Aflarea/modificarea caracteristicilor terminalului:  
UNIX> `stty [-a]` (pentru aflare)  
UNIX> `stty ...` (pentru modificare)
- Primitive (apelabile din programe C):
  - Se utilizează structura de date `termios` și funcțiile de manipulare a ei, descrise în pagina de manual:  
UNIX> `man termios`

Exemple de programe care ilustrează lucrul de nivel scăzut cu terminalul  
– vezi lectia [ncurses.htm](#)

# Bibliografie obligatorie

Lectia ncurses.htm

Programele demonstrative amintite pe parcursul acestei prezentări pot fi descărcate de la adresa următoare:

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