PROGRAMARE DE SISTEM ÎN C PENTRU PLATFORMA LINUX (I)

Gestiunea fișierelor, partea I-a:

Primitivele I/O pentru lucrul cu fișiere

Cristian Vidrașcu
vidrascu@info.uaic.ro

Aprilie, 2020

Sumar

Introducere

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 UNIXce implementează standardul POSIX.
  - Avantaj: funcțiile din acest API oferă, practic, acces la toate funcționalitățile "exportate" către *user-mode* de către nucleul Linux.
  - 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 SUBSYTEM 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

4 / 25

Principalele categorii de primitive I/O

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`
- primitive de duplicare a unei *sesiuni de lucru* cu un fișier: `dup`, `dup2`

5 / 25

Principalele categorii de primitive I/O (cont.)

- primitive pentru consultarea "stării" unor *sesiuni de lucru* cu fișiere (operații I/O sincrone multiplexate): `select`, `poll`
- primitive de modificare a unor atribute 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`
- ș.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()`.

6 / 25

Primitiva access

- *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(ele) verificat(e) este/sunt permis(e), respectiv `-1` în caz de eroare.

7 / 25

Primitiva creat

- *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);`

8 / 25

Primitiva open

- *"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`, ș.a.
  - valoarea returnată este descriptorul de fișier deschis (*i.e.*, **indexul în tabela locală de fișiere deschise**), sau `-1` în caz de eroare.

9 / 25

Primitiva read

- *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 octeții citiți
  - *nr_oct* = numărul de octeți de citit din fișier
  - valoarea returnată este numărul de octeți efectiv citiți, dacă citirea a reușit (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 citiți 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 poziția `EOF` (*i.e.*, *end-of-file*).

Primitiva write

- *Scrierea într-un fișier*: primitiva `write` ([3]).
  Interfața funcției `write`:
  `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

- *Poziționarea cursorului într-un fișier* (*i.e.* ajustarea deplasamentului curent în fișier): 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 fișier (întotdeauna, în raport cu începutul fișierului), sau `-1` în caz de eroare.

Primitiva close

- *"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] prezentate 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

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 // Exact dimensiunea paginii de memorie, din motive de eficienta a operatiilor cu discul!

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;
}
```

Notă: acest exemplu este disponibil pentru descărcare de aici: `cp_POSIX.c` ([2]).

Alte primitive I/O pentru fișiere

- *"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/ștergerea 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.
- ș.a.

Primitive I/O pentru directoare

- *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".

Șablonul de lucru cu directoare

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
}

/* prelucrarea secventiala 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 șablon – 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`

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 repetate ale aceluiași bloc de disc, la intervale de timp foarte scurte, vor regăsi informația direct din *cache*-ul din memorie; ii) scrieri repetate 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 octeți). 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 în preambulul din pagina Laboratorului #6.

Biblioteca standard de C: funcții pentru operații I/O cu fișiere 19 / 25

Agenda

Despre biblioteca standard de C

- 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

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 utilizați 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.)

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 apeluri ale 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

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 / scriere într-un fișier ;
- 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 fiecăruia:

- %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 `float` (un număr "real" cu semn), reprezentarea *textuală* fiind cea corespunzătoare scrierii numărului în notația cu punct zecimal
- %f : un `float` (un număr "real" cu semn), reprezentarea *textuală* fiind cea corespunzătoare scrierii numărului în notația cu mantisă E
- ș.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` .

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

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. C library implementation. */
#include <stdio.h>
#define BUF_SIZE 4096 // Exact dimensiunea paginii de memorie, din motive de eficienta a operatiilor cu discul!

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 atime. */
    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>`.

Referințe bibliografice

Bibliografie obligatorie

[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/SO/books/ManualID-SO.pdf

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

- https://profs.info.uaic.ro/~vidrascu/SO/cursuri/C-programs/file/

[3] POSIX API: `man 2 open`, `man 2 read`, `man 2 write`, ș.a.

[4] STANDARD C LIBRARY: `man 3 stdio`, `man 3 string`, `man 0p stdlib.h`.