

# Sistemul de fișiere: partea 1

## Sisteme de Operare

Ciprian Oprea și Adrian Colea

Universitatea Tehnică din Cluj-Napoca  
Departamentul Calculatoare

Cursul 4



# Cuprins

- 1 Concepte de bază
- 2 Fișierul
- 3 Directorul
- 4 Apeluri de sistem Linux pentru sistemul de fișiere



# Cuprins

- 1 Concepte de bază
- 2 Fișierul
- 3 Directorul
- 4 Apeluri de sistem Linux pentru sistemul de fișiere



# Context

- orice aplicație lucrează cu date, păstrate în memorie
  - unele aplicații folosesc mai multe date decât încap în memoria principală (RAM)
  - anumite date trebuie stocate persistent, și după încheierea procesului
  - anumite date trebuie partajate cu alte aplicații




# Context

- orice aplicație lucrează cu date, păstrate în memorie
  - unele aplicații folosesc mai multe date decât încap în memoria principală (RAM)
  - anumite date trebuie stocate persistent, și după încheierea procesului
  - anumite date trebuie partajate cu alte aplicații
- sistemul de fișiere trebuie să ne permită
  - salvarea persistentă a datelor
  - regăsirea datelor salvate




# Medii de stocare

- discuri magnetice – folosite de mult timp, încât termenul *disc* a rămas încetățenit în domeniul stocării datelor
- Solid-State Drives (SSD) – acces aleator mai rapid la date; fără componente mobile
- stick-uri USB
- CD-uri, DVD-uri, dischete, benzi magnetice ... 



# Medii de stocare

- discuri magnetice – folosite de mult timp, încât termenul *disc* a rămas încetățenit în domeniul stocării datelor
- Solid-State Drives (SSD) – acces aleator mai rapid la date; fără componente mobile
- stick-uri USB
- CD-uri, DVD-uri, dischete, benzi magnetice ... 

Conceptual, mediile de stocare împart datele în **blocuri de dimensiune fixă** și suportă operații de tipul

- citește datele din blocul  $k$
- scrie datele în blocul  $k$



# Organizarea informației în blocuri

- de unde știm în ce bloc se găsește o anumită informație?
  - trebuie să “ținem minte” că am salvat-o în blocul 12493?
- cum știm dacă un bloc este liber?
- ce facem cu datele care ocupă mai multe blocuri?
- cum prevenim un utilizator să citească datele altui utilizator?





# Organizarea informației în blocuri

- de unde știm în ce bloc se găsește o anumită informație?
  - trebuie să “ținem minte” că am salvat-o în blocul 12493?
- cum știm dacă un bloc este liber?
- ce facem cu datele care ocupă mai multe blocuri?
- cum prevenim un utilizator să citească datele altui utilizator?

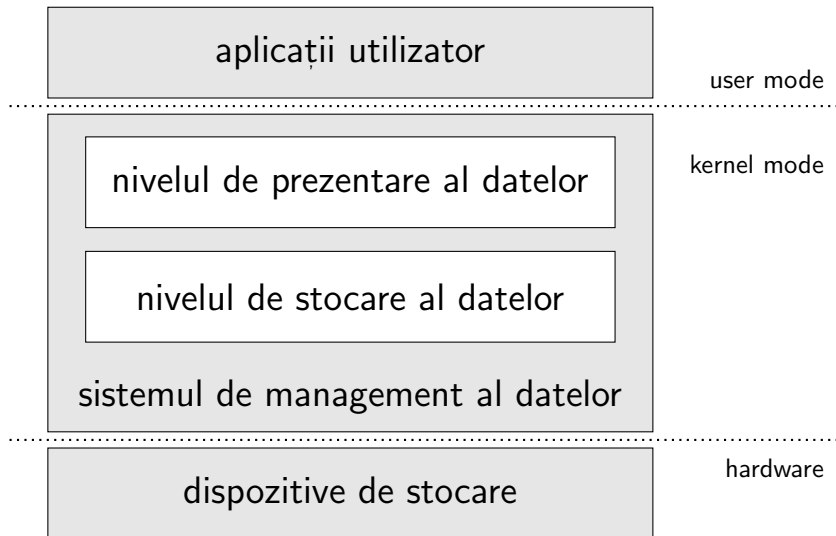
## Definiție

**Sistemul de fișiere** este o componentă a sistemului de operare care

- oferă utilizatorilor o interfață abstractă (simplificată) pentru stocarea și regăsirea datelor;
- gestionează datele de pe mediile de stocare.



# Arhitectura sistemului de fișiere (1)

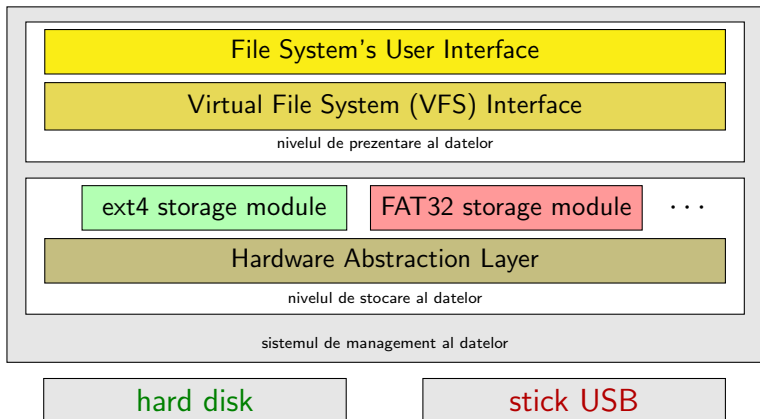




# Arhitectura sistemului de fișiere (2)

cat /home/so/f1

wc -l /media/so/usb/f2





# Cuprins

- 1 Concepte de bază
- 2 Fișierul**
- 3 Directorul
- 4 Apeluri de sistem Linux pentru sistemul de fișiere



# Fișierul din perspectiva utilizatorului

## Definiție

Fișierul este unitatea de bază pentru stocarea informației și suportă cel puțin operațiile de salvare (*store*) și citire (*retrieve*) a datelor.

- orice date pe care utilizatorul dorește să le stocheze trebuie să ajungă în fișiere



# Fișierul din perspectiva utilizatorului

## Definiție

Fișierul este unitatea de bază pentru stocarea informației și suportă cel puțin operațiile de salvare (*store*) și citire (*retrieve*) a datelor.

- orice date pe care utilizatorul dorește să le stocheze trebuie să ajungă în fișiere
- are un **nume**, pe baza căruia este identificat
- are un **conținut**, ce poate fi privit ca un container de date
- are **metadate** (attribute)



# Fișierul din perspectiva utilizatorului

## Definiție

Fișierul este unitatea de bază pentru stocarea informației și suportă cel puțin operațiile de salvare (*store*) și citire (*retrieve*) a datelor.

- orice date pe care utilizatorul dorește să le stocheze trebuie să ajungă în fișiere
- are un **nume**, pe baza căruia este identificat
- are un **conținut**, ce poate fi privit ca un container de date
- are **metadate** (attribute)
- legătura cu blocurile mediului de stocare face parte din perspectiva internă a SO (se va studia la cursul următor)



## Numele fișierelor

- atunci când se salvează un fișier, i se dă un nume
- pe baza numelui, acesta poate fi regăsit ulterior





## Numele fișierelor

- atunci când se salvează un fișier, i se dă un nume
- pe baza numelui, acesta poate fi regăsit ulterior
- regulile pentru numele unui fișier variază de la un sistem de fișiere la altul
  - SF *FAT-16* al MS-DOS suporta nume de 8+3 caractere
  - SF moderne suportă nume de până la 255 caractere și Unicode
- unele SF sunt *case sensitive* (ex. *ext4* pe Linux), în timp ce altele sunt *case insensitive* (ex. *FAT-16* pe MS-DOS)



# Numele fișierelor

- atunci când se salvează un fișier, i se dă un nume
- pe baza numelui, acesta poate fi regăsit ulterior
- regulile pentru numele unui fișier variază de la un sistem de fișiere la altul
  - SF *FAT-16* al MS-DOS suporta nume de 8+3 caractere
  - SF moderne suportă nume de până la 255 caractere și Unicode
- unele SF sunt *case sensitive* (ex. *ext4* pe Linux), în timp ce altele sunt *case insensitive* (ex. *FAT-16* pe MS-DOS)
- extensia (partea de la ultimul '.' până la finalul numelui)
  - în general e doar o parte din nume, SO nu o tratează special
  - anumite programe se folosesc de extensie
    - *gcc* compilează diferit .c și .cpp
    - *Windows Explorer* alege cu ce program să deschidă fișierele în funcție de extensie (.txt cu *Notepad* și .docx cu *MS Word*)



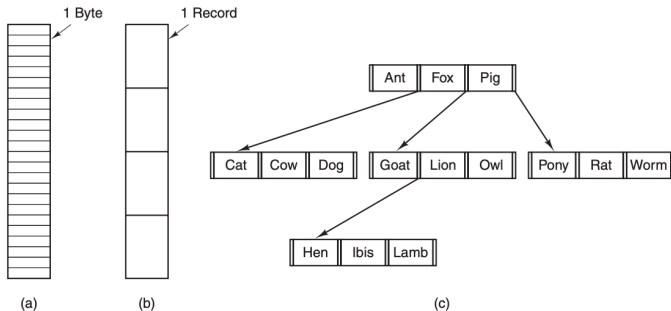
# Tipuri de fișiere

- fișiere obișnuite (eng. *regular files*)
  - documente (PDF, Word, ...), cod sursă, programe executabile, imagini, video, arhive, ...
- directoare
  - directoarele sunt tot fișiere
  - au o structură specializată (de obicei B-Tree)
- legături simbolice
  - fișiere ce “pointează” spre alte fișiere
- pipe-uri
  - fișiere de tip FIFO (scriem la un capăt, citim de la celălalt)
- fișiere speciale de tip caracter (modelează dispozitive I/O)
- fișiere speciale de tip bloc (modelează dispozitive de stocare)



# Structura fișierelor (1)

- orice fișier poate fi privit ca o **secvență de octeți**
- există și structuri specializate
  - secvență de înregistrări de dimensiune fixă
  - structură arborescentă



figură preluată din [MOS]



## Structura fișierelor (2)

Ce structură are un fișier text?

```
$ echo "Sisteme de Operare" >so.txt
```





## Structura fișierelor (2)

Ce structură are un fișier text?

```
$ echo "Sisteme de Operare" >so.txt
```



- pentru a vizualiza conținutul “raw” putem deschide mc, apăsăm F3 (View) pe fișier apoi F4 (Hex) – comandă echivalentă:  
hexdump -C so.txt

```
00000000 53 69 73 74 | 65 64 65 20 | 64 65 20 4F | 70 65 72 61 | 72 65 0A | 00000000 | Sisteme de Operare
```



## Structura fișierelor (2)

Ce structură are un fișier text?

```
$ echo "Sisteme de Operare" >so.txt
```



- pentru a vizualiza conținutul “raw” putem deschide mc, apăsăm F3 (View) pe fișier apoi F4 (Hex) – comandă echivalentă:  
hexdump -C so.txt

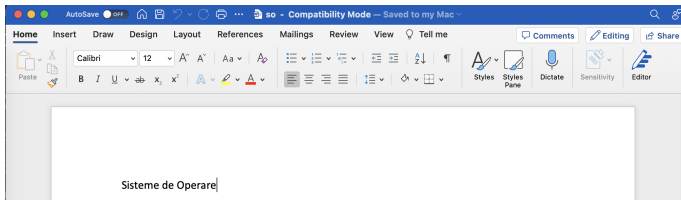
```
00000000 53 69 73 74 | 65 64 65 20 | 64 65 20 4F | 70 65 72 61 | 72 65 0A | Sisteme de Operare
```

- un fișier text e un fișier obișnuit (binar) al cărui conținut poate fi citit “cu ochiul liber”
- conține caractere afișabile (ASCII 32-126)
- liniile sunt separate prin 0A (Linux) sau 0D 0A (Windows)



# Structura fișierelor (3)

Ce structură are un document Word (.doc)?

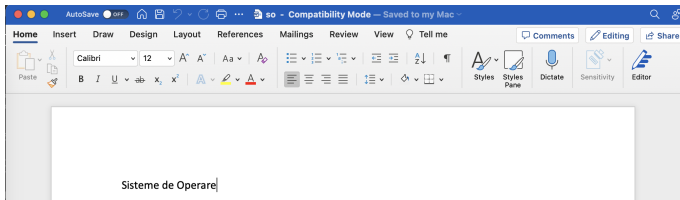






# Structura fișierelor (3)

Ce structură are un document Word (.doc)?

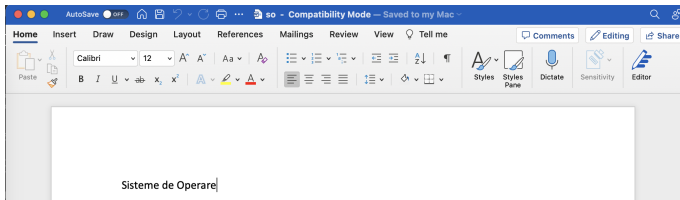


00000000	D0 CF 11 E0	A1 B1 1A E1	00 00 00 00	00 00 00 00	00 00 00 00	...	[1]	.....
00000014	00 00 00 00	3E 00 03 00	FE FF 09 00	06 00 00 00	00 00 00 00	...	>	.....
00000028	00 00 00 00	01 00 00 00	27 00 00 00	00 00 00 00	00 10 00 00	.....	'	.....
0000003C	29 00 00 00	01 00 00 00	FE FF FF FF	00 00 00 00	26 00 00 00	).....	.	&.....
00000050	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF		.....
00000064	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF		.....
00000A00	53 69 73 74	65 6D 65 20	64 65 20 4F	70 65 72 61	72 65 0D 00	Sisteme de Operare..		
00000A14	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....		.....
00000A28	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....		.....



# Structura fișierelor (3)

Ce structură are un document Word (.doc)?



00000000	D0 CF 11 E0	A1 B1 1A E1	00 00 00 00	00 00 00 00	00 00 00 00	...	[1]	.....
00000014	00 00 00 00	3E 00 03 00	FE FF 09 00	06 00 00 00	00 00 00 00	...	>	.....
00000028	00 00 00 00	01 00 00 00	27 00 00 00	00 00 00 00	00 10 00 00	...	'	.....
0000003C	29 00 00 00	01 00 00 00	FE FF FF FF	00 00 00 00	26 00 00 00	)	.....	&...
00000050	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF	FF FF FF	FF
00000064	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF	FF FF FF	FF
00000A00	53 69 73 74	65 6D 65 20	64 65 20 4F	70 65 72 61	72 65 0D 00	Sisteme de Operare..		
00000A14	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....		
00000A28	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....		

- fișier binar ( “înțeles” de anumite programe cum ar fi MS Word)
- *magic value*: D0 CF 11 E0
- textul începe la offset-ul 0xA00 (2560)



## Structura fișierelor (4)

Ar trebui SO să cunoască structura fișierelor?

- da, în cazul directoarelor
  - trebuie să permită navigarea și parcurgerea conținutului
- da, în cazul legăturilor simbolice și al pipe-urilor



## Structura fișierelor (4)

Ar trebui SO să cunoască structura fișierelor?

- da, în cazul directoarelor
  - trebuie să permită navigarea și parcurgerea conținutului
- da, în cazul legăturilor simbolice și al pipe-urilor
- dar în cazul fișierelor obișnuite?



## Structura fișierelor (4)

Ar trebui SO să cunoască structura fișierelor?

- da, în cazul directoarelor
  - trebuie să permită navigarea și parcurgerea conținutului
- da, în cazul legăturilor simbolice și al pipe-urilor
- nu, în cazul fișierelor obișnuite
  - multe formate de fișiere, dorim să păstrăm SO cât mai simplu
  - fiecare aplicație își gestionează propriile fișiere
  - $\Rightarrow$  **flexibilitate**
  - excepție: fiecare SO înțelege formatul propriilor executabile



# Metadatele fișierelor

metadata = date despre date

- tipul de fișier (obișnuit, director, ...)
- proprietarul fișierului
- permisiunile asupra fișierului (*read*, *write*, *execute*)
- dimensiune
- momente de timp (când a fost creat, când a fost modificat ultima dată, când a fost accesat ultima dată)
- ...



# Metode de acces

- acces secvențial
  - fișierul poate fi citit doar de la început, element cu element
  - medii de stocare de tipul banda magnetica / caseta audio
  - încă se folosește pentru directoare
- acces aleator
  - putem citi date de la orice poziție din fișier
  - fezabil, dar costisitor pe discurile magnetice (trebuie să se deplaseze mecanic capul de citire)
  - natural, pe discuri de tip SSD (adresabile electronic)



# Operații pe fișiere

- accesarea / manipularea fișierelor și a metadatelor
  - creare (*create*)
  - ștergere (*delete*)
  - redenumire (*rename*)
  - citirea/modificarea metadatelor





# Operații pe fișiere

- accesarea / manipularea fișierelor și a metadatelor
  - creare (*create*)
  - ștergere (*delete*)
  - redenumire (*rename*)
  - citirea/modificarea metadatelor
- accesarea / manipularea conținutului
  - citire (*read*)
  - scriere (*write*)



# Operații pe fișiere

- accesarea / manipularea fișierelor și a metadatelor
  - creare (*create*)
  - ștergere (*delete*)
  - redenumire (*rename*)
  - citirea/modificarea metadatelor
- accesarea / manipularea conținutului
  - deschidere (*open*)
  - închidere (*close*)
  - citire (*read*)
  - scriere (*write*)



# Operații pe fișiere

- accesarea / manipularea fișierelor și a metadatelor
  - creare (*create*)
  - ștergere (*delete*)
  - redenumire (*rename*)
  - citirea/modificarea metadatelor
- accesarea / manipularea conținutului
  - deschidere (*open*)
  - închidere (*close*)
  - citire (*read*)
  - scriere (*write*)
  - poziționare (*seek*)



# Operații pe fișiere

- accesarea / manipularea fișierelor și a metadatelor
  - creare (*create*)
  - ștergere (*delete*)
  - redenumire (*rename*)
  - citirea/modificarea metadatelor
- accesarea / manipularea conținutului
  - deschidere (*open*)
  - închidere (*close*)
  - citire (*read*)
  - scriere (*write*)
  - poziționare (*seek*)
  - adăugare la final (*append*)
  - reducerea dimensiunii (*truncate*)



# Cuprins

- 1 Concepte de bază
- 2 Fișierul
- 3 Directorul**
- 4 Apeluri de sistem Linux pentru sistemul de fișiere



# Directorul din perspectiva utilizatorului

## Definiție

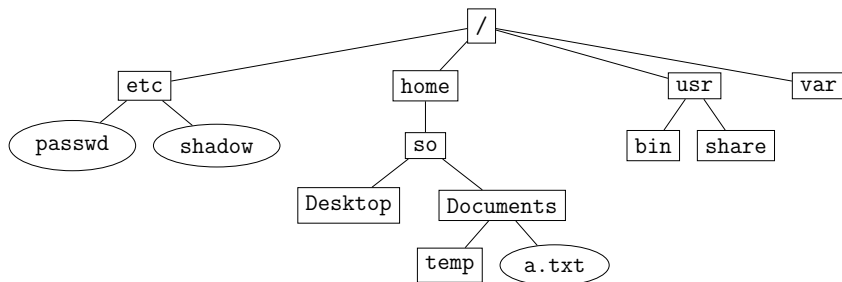
Directorul (denumit și *folder*) este un tip de fișier care ajută la organizarea și clasificarea datelor.

- atunci când avem multe fișiere este complicat să le regăsim doar pe baza numelui
- orice fișier poate fi stocat într-un director (sau în mai multe)
- SF moderne permit o structură ierarhică de directoare
  - un director poate conține alte directoare
  - istoric au existat SF cu doar unul sau două nivele de directoare



# Ierarhia de directoare

Exemplu (Linux):



- pentru a ajunge la fișierul 'a.txt', trebuie să trecem prin directoarele '/' (se citește *root*), 'home', 'so' și 'Documents'



# Calea spre un fișier

- căi absolute
  - specifică toate directoarele, începând cu rădăcina
  - exemplu Linux: `/home/so/Documents/a.txt`
  - exemplu Windows: `C:\Users\so\Documents\a.txt`





# Calea spre un fișier

- căi absolute
  - specifică toate directoarele, începând cu rădăcina
  - exemplu Linux: `/home/so/Documents/a.txt`
  - exemplu Windows: `C:\Users\so\Documents\a.txt`
- căi relative
  - orice proces are un director curent de lucru (*cwd* = *current working directory*)
  - orice director are două subdirectoare implicite:
    - `'.'` – referință la el însuși
    - `'..'` – referință la directorul părinte
  - căile care nu încep cu rădăcina se consideră relative la *cwd*
  - exemple:
    - *cwd* = `/home/so`, avem calea relativă `Documents/a.txt`
    - *cwd* = `/home/so/Documents` → `a.txt` (sau `./a.txt`)
    - *cwd* = `/home/so/Desktop` → `../Documents/a.txt`



# Operații pe directoare

- accesarea / manipularea directoarelor și a metadatelor
  - creare (*create*)
  - ștergere (*delete*)
  - redenumire (*rename*)
  - citirea/modificarea metadatelor



# Operații pe directoare

- accesarea / manipularea directoarelor și a metadatelor
  - creare (*create*)
  - ștergere (*delete*)
  - redenumire (*rename*)
  - citirea/modificarea metadatelor
- citirea conținutului
  - deschidere (*opendir*)
  - închidere (*closedir*)
  - citire (*readdir*)
  - re poziționare la început (*rewind*)



# Operații pe directoare

- accesarea / manipularea directoarelor și a metadatelor
  - creare (*create*)
  - ștergere (*delete*)
  - redenumire (*rename*)
  - citirea/modificarea metadatelor
- citirea conținutului
  - deschidere (*opendir*)
  - închidere (*closedir*)
  - citire (*readdir*)
  - re poziționare la început (*rewind*)
- modificarea conținutului
  - indirect, prin adăugarea și ștergerea fișierelor
  - crearea de legături (*link*)
  - ștergerea de legături (*unlink*)



# Cuprins

- 1 Concepte de bază
- 2 Fișierul
- 3 Directorul
- 4 Apeluri de sistem Linux pentru sistemul de fișiere



## Crearea, ștergerea, deschiderea și închiderea fișierelor

```
int open(const char *pathname, int flags);  
int open(const char *pathname, int flags, mode_t mode);  
int creat(const char *pathname, mode_t mode);  
int close(int fd);  
int unlink(const char *pathname);
```

- rezultatul lui `open` și `creat` e un descriptor de fișier
  - va fi folosit ca parametru în alte apeluri de sistem
- `flags` e o mască de biți care se referă la modul de deschidere
  - `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC`, ...
- `mode` specifică permisiunile fișierului în cazul în care este creat
  - cel mai simplu se pot specifica în octal (ex. 0644)



## Exemplu: creare şi ştergere

```
int fd;

// create a new file
fd = creat("/home/so/file.txt", 0600);
// if file exists, it is truncated
// newly file opened for WRONLY
// note permissions: rw-----

// remove the file (remove a link to the file)
unlink("/home/so/file.txt");
```



## Citirea şi scrierea fişierelor

```
ssize_t read(int fd, void *buf, size_t count);  
ssize_t write(int fd, const void *buf, size_t count);  
off_t lseek(int fd, off_t offset, int whence);
```

- read citeşte în buf, write scrie datele din buf
- count reprezintă numărul de octeţi care trebuie citiţi/scrişi
- read şi write returnează numărul de octeţi care au fost citiţi/scrişi
- dacă read returnează 0 (sau o valoare mai mică decât count) înseamnă că s-a ajuns la finalul fişierului
- lseek deplasează cursorul în cadrul fişierului cu offset octeţi
  - parametrul whence poate fi SEEK\_SET, SEEK\_CUR sau SEEK\_END





## Exemplu: fişier binar (1)

- programul *P1* scrie în fişier un caracter şi un întreg

```
int fd;
int number = 10;
char c = 'b';

if((fd = creat("file.bin", 0644)) < 0) {
    perror("Cannot create the file");
    exit(1);
}
// write a char on the first byte
write(fd, &c, sizeof(c));
// write an integer's representation on the next four bytes
write(fd, &number, sizeof(number));
close(fd);
```

- conţinutul lui *file.bin*: 62 0A 00 00 00



## Exemplu: fișier binar (2)

- programul *P2* citește din fișierul anterior doar întregul

```
int fd;
int number;

if((fd = open("file.bin", O_RDONLY)) < 0) {
    perror("Cannot open the file");
    exit(1);
}

// position where WE (MUST) KNOW the integer is
// i.e. one byte after beginning of file
lseek(fd, sizeof(char), SEEK_SET);
// read four bytes from crt position
// i.e. an integer's representation
read(fd, &number, sizeof(number));
close(fd);
```



## Exemplu: Prima linie dintr-un fișier text

```
#define MAX_LINE 1024
int fd, i;
char line[MAX_LINE + 1];

fd = open("file.txt", O_RDONLY);
if(fd < 0) { ... } // display error and exit
i=0;
while((i < MAX_LINE) && (read(fd, &line[i], 1) > 0)) {
    if(line[i] != '\n') // '\n' or 0xA is the line terminator
        i++;
    else break;
}
line[i] = 0; // a proper string should be NULL-terminated
printf("The read line is: %s", line);
close(fd);
```



## Gestiunea fișierelor deschise

- operațiile asupra conținutului unui fișier se fac pe fișiere deschise
  - SO poate ține o structură de date cu diverse informații despre fișierul deschis (ex. poziția cursorului)
  - se poate face buffering la operațiile de citire



## Gestiunea fișierelor deschise

- operațiile asupra conținutului unui fișier se fac pe fișiere deschise
  - SO poate ține o structură de date cu diverse informații despre fișierul deschis (ex. poziția cursorului)
  - se poate face buffering la operațiile de citire
- SO gestionează trei tabele:
  - *i-node File Table (IT)*
    - o singură tabelă la nivel de sistem
  - *Open File Table (OFT)*
    - o singură tabelă la nivel de sistem cu toate fișierele deschise
  - *File Descriptor Table (FDT)*
    - toți descriptorii de fișier folosiți de procesul curent
    - câte o tabelă pentru fiecare proces



## Gestiunea fişierelor deschise

- operaţiile asupra conţinutului unui fişier se fac pe fişiere deschise
  - SO poate ţine o structură de date cu diverse informaţii despre fişierul deschis (ex. poziţia cursorului)
  - se poate face buffering la operaţiile de citire
- SO gestionează trei tabele:
  - *i-node File Table (IT)*
    - o singură tabelă la nivel de sistem
  - *Open File Table (OFT)*
    - o singură tabelă la nivel de sistem cu toate fişierele deschise
  - *File Descriptor Table (FDT)*
    - toţi descriptorii de fişier folosiţi de procesul curent
    - câte o tabelă pentru fiecare proces
- fiecare proces “se naşte” cu trei descriptori de fişiere:
  - 0: *standard input*
  - 1: *standard output*
  - 2: *standard error*



# Duplicarea descriptorilor de fișier

```
int dup(int oldfd);  
int dup2(int oldfd, int newfd);
```

- dup duplică descriptorul oldfd folosind primul descriptor disponibil (în ordine crescătoare, începând de la 0)
- dup2 duplică descriptorul oldfd folosind neapărat descriptorul newfd
  - dacă newfd e folosit de un fișier deschis, se apelează implicit close pe acesta, întâi



```
//Process 1
```

```
int fd1, fd2;
```

```
char buf[] = "1234567890";
```

```
//Process 2
```

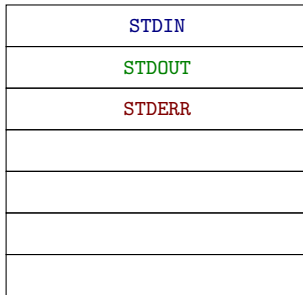
```
int fd1, fd2, fd3;
```

```
char buf[] = "ABCDEFGH IJ";
```

IT:



OFT:



FDT:



Proc 1

Proc 2





```
//Process 1
```

```
int fd1, fd2;
```

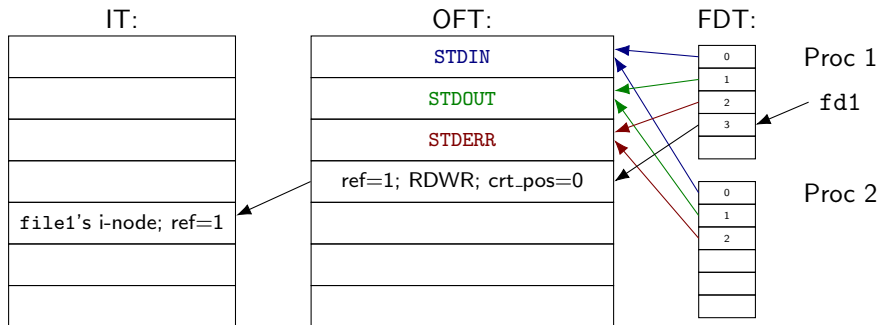
```
char buf[] = "1234567890";
```

```
fd1 = open("file1", O_RDWR);
```

```
//Process 2
```

```
int fd1, fd2, fd3;
```

```
char buf[] = "ABCDEFGH IJ";
```





```
//Process 1
```

```
int fd1, fd2;
```

```
char buf[] = "1234567890";
```

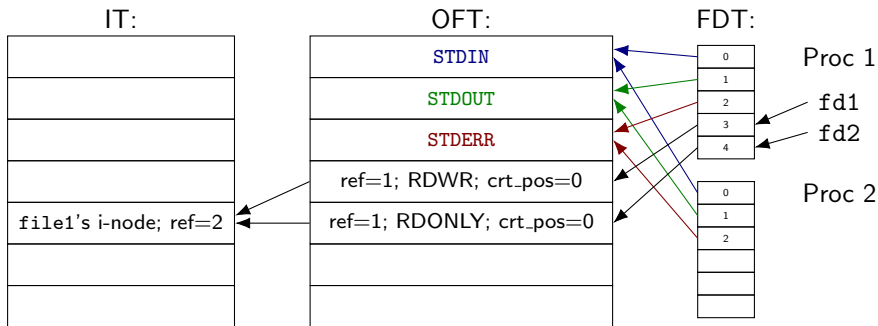
```
fd1 = open("file1", O_RDWR);
```

```
fd2 = open("file1", O_RDONLY);
```

```
//Process 2
```

```
int fd1, fd2, fd3;
```

```
char buf[] = "ABCDEFGHJIJ";
```





```
//Process 1
```

```
int fd1, fd2;
```

```
char buf[] = "1234567890";
```

```
fd1 = open("file1", O_RDWR);
```

```
fd2 = open("file1", O_RDONLY);
```

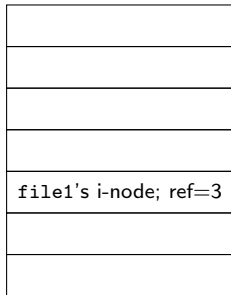
```
//Process 2
```

```
int fd1, fd2, fd3;
```

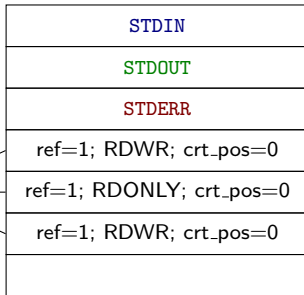
```
char buf[] = "ABCDEFGH IJ";
```

```
fd1 = open("file1", O_RDWR);
```

IT:



OFT:



FDT:



Proc 1

fd1

fd2



Proc 2

fd1



```
//Process 1
```

```
int fd1, fd2;
```

```
char buf[] = "1234567890";
```

```
fd1 = open("file1", O_RDWR);
```

```
fd2 = open("file1", O_RDONLY);
```

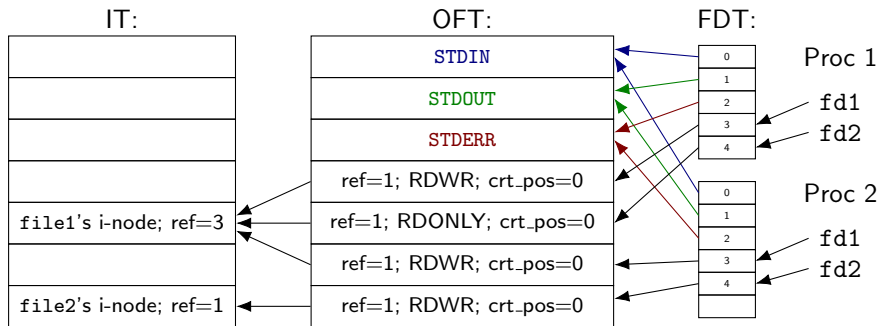
```
//Process 2
```

```
int fd1, fd2, fd3;
```

```
char buf[] = "ABCDEFGHJIJ";
```

```
fd1 = open("file1", O_RDWR);
```

```
fd2 = open("file2", O_RDWR);
```





```
//Process 1
```

```
int fd1, fd2;
```

```
char buf[] = "1234567890";
```

```
fd1 = open("file1", O_RDWR);
```

```
fd2 = open("file1", O_RDONLY);
```

```
//Process 2
```

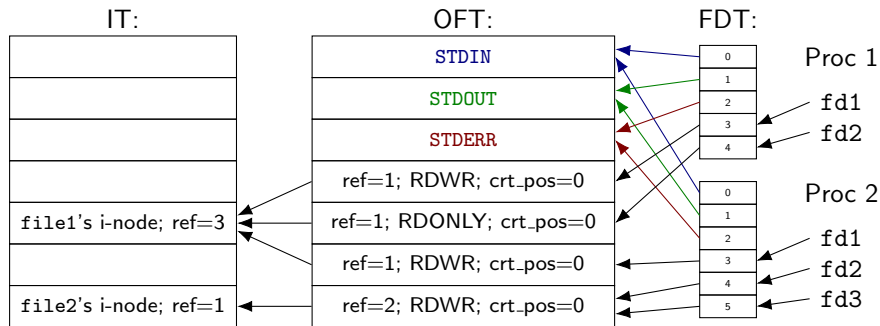
```
int fd1, fd2, fd3;
```

```
char buf[] = "ABCDEFGH IJ";
```

```
fd1 = open("file1", O_RDWR);
```

```
fd2 = open("file2", O_RDWR);
```

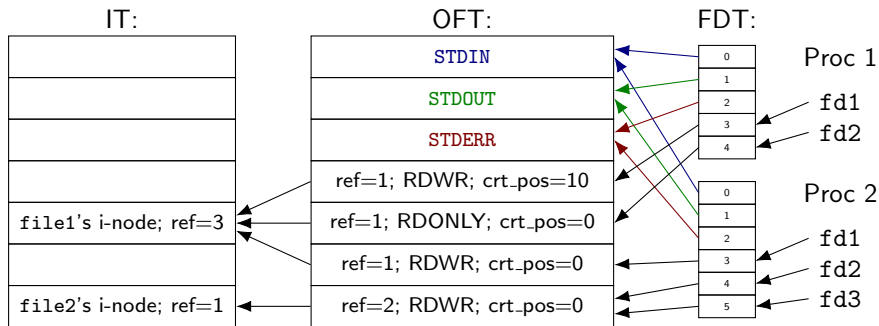
```
fd3 = dup(fd2);
```





```
//Process 1
int fd1, fd2;
char buf[] = "1234567890";
fd1 = open("file1", O_RDWR);
fd2 = open("file1", O_RDONLY);
write(fd1, buf, 10);
```

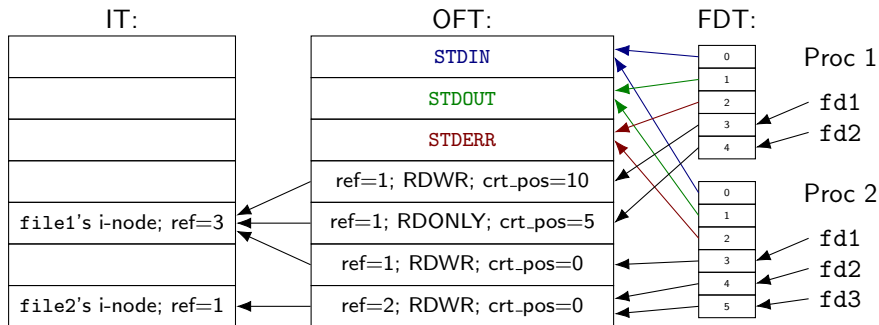
```
//Process 2
int fd1, fd2, fd3;
char buf[] = "ABCDEFGH IJ";
fd1 = open("file1", O_RDWR);
fd2 = open("file2", O_RDWR);
fd3 = dup(fd2);
```





```
//Process 1
int fd1, fd2;
char buf[] = "1234567890";
fd1 = open("file1", O_RDWR);
fd2 = open("file1", O_RDONLY);
write(fd1, buf, 10);
read(fd2, buf, 5);
```

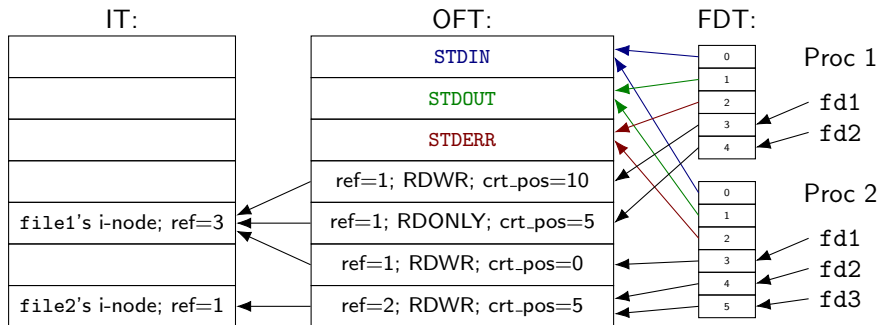
```
//Process 2
int fd1, fd2, fd3;
char buf[] = "ABCDEFGH IJ";
fd1 = open("file1", O_RDWR);
fd2 = open("file2", O_RDWR);
fd3 = dup(fd2);
```





```
//Process 1
int fd1, fd2;
char buf[] = "1234567890";
fd1 = open("file1", O_RDWR);
fd2 = open("file1", O_RDONLY);
write(fd1, buf, 10);
read(fd2, buf, 5);
```

```
//Process 2
int fd1, fd2, fd3;
char buf[] = "ABCDEFGH IJ";
fd1 = open("file1", O_RDWR);
fd2 = open("file2", O_RDWR);
fd3 = dup(fd2);
write(fd2, buf, 5);
```

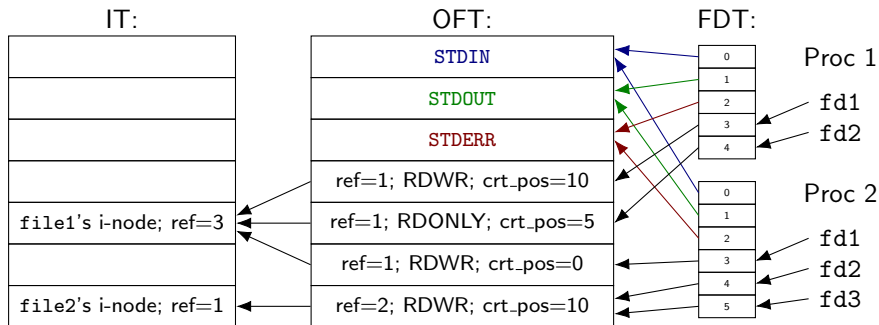






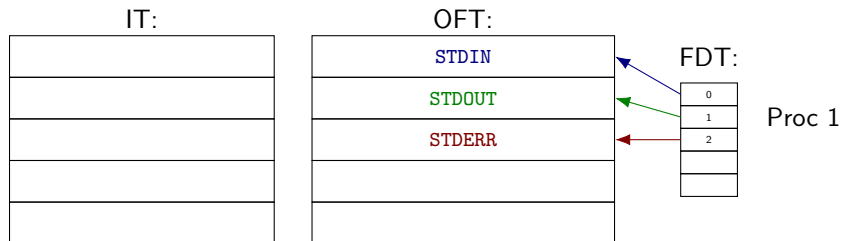
```
//Process 1
int fd1, fd2;
char buf[] = "1234567890";
fd1 = open("file1", O_RDWR);
fd2 = open("file1", O_RDONLY);
write(fd1, buf, 10);
read(fd2, buf, 5);
```

```
//Process 2
int fd1, fd2, fd3;
char buf[] = "ABCDEFGH IJ";
fd1 = open("file1", O_RDWR);
fd2 = open("file2", O_RDWR);
fd3 = dup(fd2);
write(fd2, buf, 5);
write(fd3, buf, 5);
```



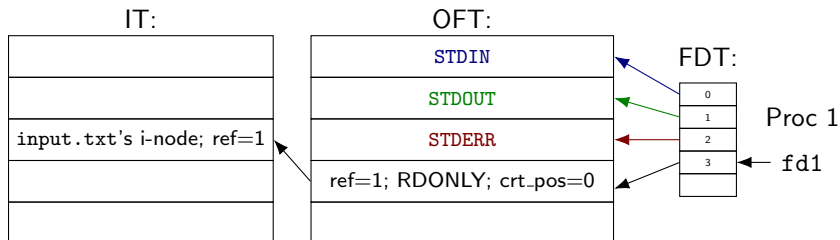


```
int fd1, fd2, n1, n2;
```



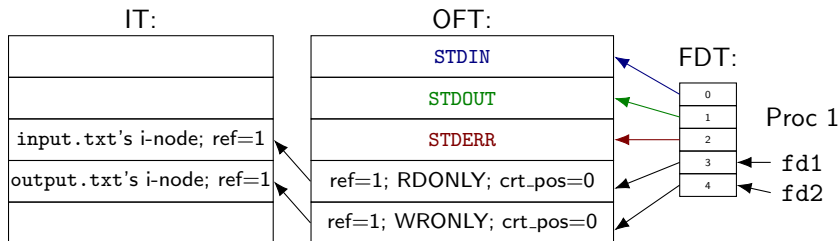


```
int fd1, fd2, n1, n2;
fd1 = open("input.txt", O_RDONLY);
```



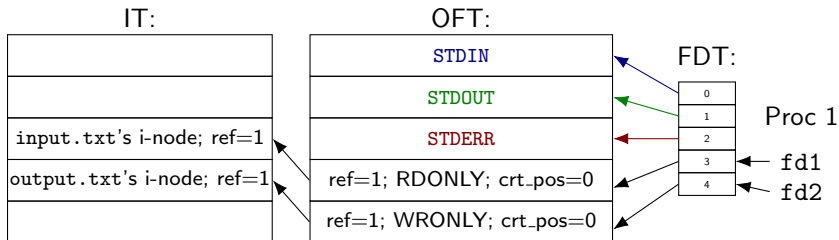


```
int fd1, fd2, n1, n2;
fd1 = open("input.txt", O_RDONLY);
fd2 = creat("output.txt", 0600);
```



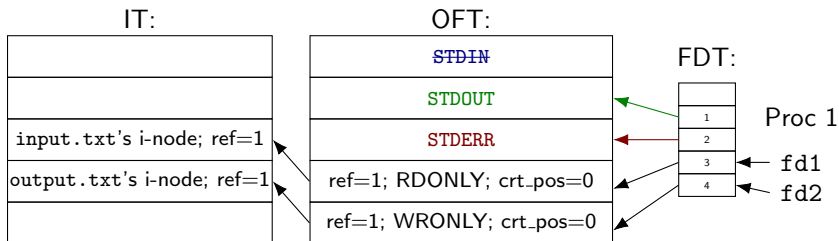


```
int fd1, fd2, n1, n2;
fd1 = open("input.txt", O_RDONLY);
fd2 = creat("output.txt", 0600);
scanf("%d", &n1); //calls read(0, ...);
```



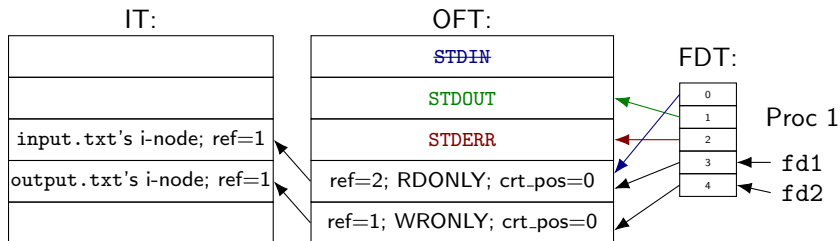


```
int fd1, fd2, n1, n2;
fd1 = open("input.txt", O_RDONLY);
fd2 = creat("output.txt", 0600);
scanf("%d", &n1); //calls read(0, ...);
close(0);
```



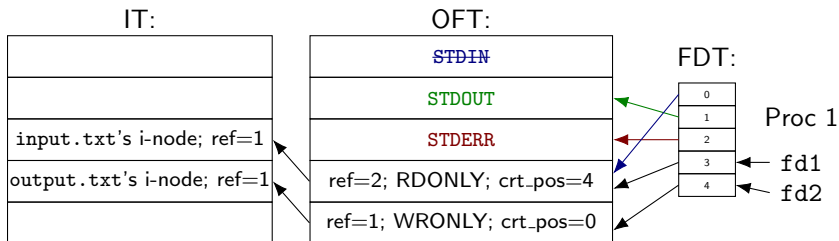


```
int fd1, fd2, n1, n2;
fd1 = open("input.txt", O_RDONLY);
fd2 = creat("output.txt", 0600);
scanf("%d", &n1); //calls read(0, ...);
close(0);
dup(fd1);
```





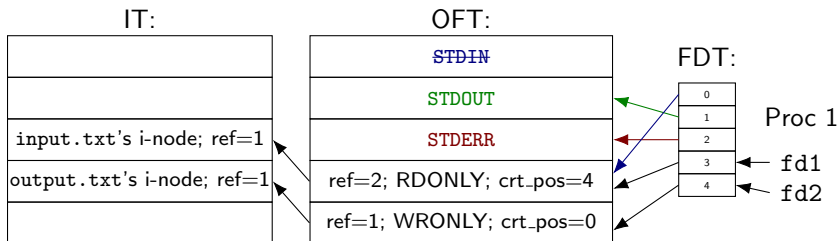
```
int fd1, fd2, n1, n2;
fd1 = open("input.txt", O_RDONLY);
fd2 = creat("output.txt", 0600);
scanf("%d", &n1); //calls read(0, ...);
close(0);
dup(fd1);
scanf("%d", &n2); //calls read(0, ...); <- "input.txt"
```





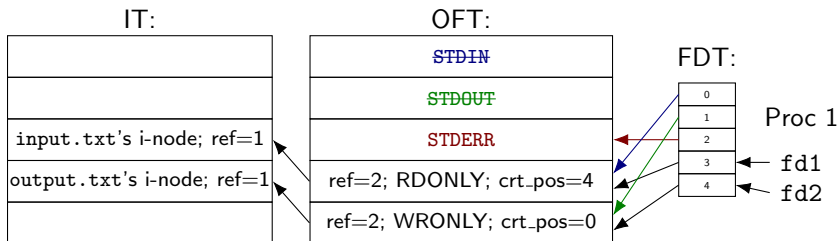


```
int fd1, fd2, n1, n2;
fd1 = open("input.txt", O_RDONLY);
fd2 = creat("output.txt", 0600);
scanf("%d", &n1); //calls read(0, ...);
close(0);
dup(fd1);
scanf("%d", &n2); //calls read(0, ...); <- "input.txt"
printf("%d\n", n1); //calls write(1, ...);
```





```
int fd1, fd2, n1, n2;
fd1 = open("input.txt", O_RDONLY);
fd2 = creat("output.txt", 0600);
scanf("%d", &n1); //calls read(0, ...);
close(0);
dup(fd1);
scanf("%d", &n2); //calls read(0, ...); <- "input.txt"
printf("%d\n", n1); //calls write(1, ...);
dup2(fd2, 1);
```

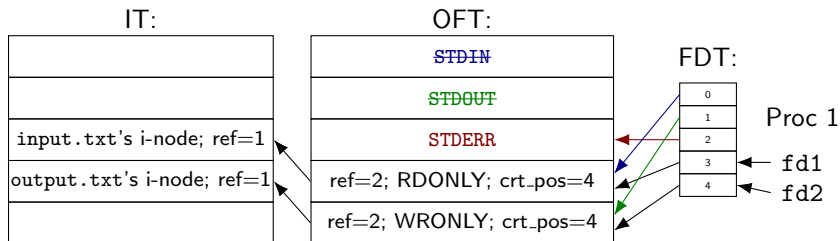




```

int fd1, fd2, n1, n2;
fd1 = open("input.txt", O_RDONLY);
fd2 = creat("output.txt", 0600);
scanf("%d", &n1); //calls read(0, ...);
close(0);
dup(fd1);
scanf("%d", &n2); //calls read(0, ...); <- "input.txt"
printf("%d\n", n1); //calls write(1, ...);
dup2(fd2, 1);
printf("%d\n", n2); //calls write(1, ...); -> "output.txt"

```





## Citirea metadatelor unui fișier

```
int stat(const char *pathname, struct stat *statbuf);  
int fstat(int fd, struct stat *statbuf);  
int lstat(const char *pathname, struct stat *statbuf);  
S_ISREG(st_mode) // is it a regular file?  
S_ISDIR(st_mode) // is it a directory?  
S_ISLNK(st_mode) // is it a symbolic link?
```

- `stat` și `lstat` primesc ca prim parametru numele fișierului, în timp ce `fstat` primește un descriptor de fișier
- `stat` urmează legăturile simbolice
- metadatele se completează în struct `stat`:
  - tipul de fișier și permisiunile (`.st_mode`)
  - dimensiunea (`.st_size`)
  - numărul *i-node*-ului (`.st_ino`)
  - id-ul proprietarului (`.st_uid`)
  - ...



## Exemplu: Afişarea metadatelor unui fişier

```
int res;
struct stat statbuf;
res = lstat(argv[1], &statbuf);
if(res < 0) {
    perror("Cannot get the stat buffer");
    exit(1);
}
// identify the file type
if(S_ISREG(statbuf.st_mode)) {
    printf("It is a regular file\n");
    printf("File's size [bytes]: %d\n", statbuf.st_size);
} else if(S_ISDIR(statbuf.st_mode)) {
    printf("It is a directory\n");
} else if(S_ISLNK(statbuf.st_mode)) {
    printf("It is a symbolic link\n");
}
```



## Parcurgerea unui director

```
DIR *opendir(const char *name);  
int closedir(DIR *dirp);  
struct dirent *readdir(DIR *dirp);  
void rewinddir(DIR *dirp);
```

- la deschiderea cu `opendir` se returnează un pointer de tip `DIR*` (intern se alocă memorie, va fi eliberată de `closedir`)
- `readdir` citește următorul element din director, returnând un pointer la o structură de tipul `struct dirent*`
  - această structură conține cel puțin membrul `d_name`, reprezentând numele elementului
- pentru a reveni la începutul directorului se apelează `rewinddir`
- atenție la elementele `"."` și `".."`, în special când parcurgem recursiv – pot introduce cicluri



## Exemplu: Parcurgerea unui director

```
DIR* dir = NULL;
struct dirent *entry = NULL;
char path[MAX_PATH];
struct stat statbuf;
if((dir = opendir("/home/so")) == NULL) { // open directory
    perror("Cannot open the directory");
    exit(1);
}
// read one-by-one dir entries until NULL returned
while((entry = readdir(dir)) != NULL) {
    // avoid "." and ".." as they are not useful
    if(strcmp(entry->d_name, ".") && strcmp(entry->d_name, "..")) {
        // build the complete path = dirpath + dirent's name
        snprintf(path, MAX_PATH, "%s/%s", "/home/so", entry->d_name);
        stat(path, &statbuf);
        if(S_ISREG(statbuf.st_mode))
            printf("%s is a file\n", path);
        else if(S_ISDIR(statbuf.st_mode))
            printf("%s is a dir\n", path);
    }
}
closedir(dir);
```



## Problemă de securitate: *dir traversal*

- un server web servește imagini din directorul `/var/www/images`
- baza de date cu informații confidențiale se află în fișierul `/var/db/secret.db`
- codul de mai jos citește imaginea cerută de utilizator prin variabila `image_name`

```
const char *image_name = ...; //read the name of the image
char path[MAX_PATH];
snprintf(path, MAX_PATH, "/var/www/images/%s", image_name);
int fd = open(path, O_RDONLY);
...
```





## Problemă de securitate: *dir traversal*

- un server web servește imagini din directorul `/var/www/images`
- baza de date cu informații confidențiale se află în fișierul `/var/db/secret.db`
- codul de mai jos citește imaginea cerută de utilizator prin variabila `image_name`

```
const char *image_name = ...; //read the name of the image
char path[MAX_PATH];
snprintf(path, MAX_PATH, "/var/www/images/%s", image_name);
int fd = open(path, O_RDONLY);
...
```

- cum ajunge un utilizator rău intenționat la baza de date secretă?



## Problemă de securitate: *dir traversal*

- un server web servește imagini din directorul `/var/www/images`
- baza de date cu informații confidențiale se află în fișierul `/var/db/secret.db`
- codul de mai jos citește imaginea cerută de utilizator prin variabila `image_name`

```
const char *image_name = ...; //read the name of the image
char path[MAX_PATH];
snprintf(path, MAX_PATH, "/var/www/images/%s", image_name);
int fd = open(path, O_RDONLY);
...
```

- cum ajunge un utilizator rău intenționat la baza de date secretă?
- cum ne protejăm de astfel de atacuri?



## Urmărirea apelurilor de sistem (1)

- se poate analiza activitatea unei aplicații urmărind ce apeluri de sistem face
- *reverse engineering* sau *analiză dinamică*
- nu e nevoie nici măcar să avem codul sursă
- folosim comanda `strace` pentru a rula aplicația
- exemplu:
  - `dd if=/dev/zero of=file1 bs=1024 count=200`
  - `strace cp file1 file2`



## Urmărirea apelurilor de sistem (2)

```
stat("file2", {st_mode=S_IFREG|0664, st_size=131072, ...}) = 0
stat("file1", {st_mode=S_IFREG|0664, st_size=204800, ...}) = 0
...
opent(AT_FDCWD, "file1", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0664, st_size=204800, ...}) = 0
opent(AT_FDCWD, "file2", O_WRONLY|O_TRUNC) = 6
fstat(6, {st_mode=S_IFREG|0664, st_size=0, ...}) = 0
...
read(3, "\\0\\0\\0\\0\\0\\0\\0\\0"..., 131072) = 131072
write(6, "\\0\\0\\0\\0\\0\\0\\0\\0"..., 131072) = 131072
read(3, "\\0\\0\\0\\0\\0\\0\\0\\0"..., 131072) = 73728
write(6, "\\0\\0\\0\\0\\0\\0\\0\\0"..., 73728) = 73728
read(3, "", 131072) = 0
close(6) = 0
close(3) = 0
```

# Bibliografie

- [MOS] A. Tanenbaum și H. Bos, *Modern Operating Systems*, 4th Edition, Pearson Education, 2015, Secțiunile 4.1 și 4.2
- [LAB] C. Oprea și A. Colea, *Sisteme de operare - îndrumător de laborator*, UTPress, 2021, Capitolele 4 și 5