# Operatii IO simple

## Contents



# Fiiere. Sisteme de fiiere

Fiierul este una din abstraciile fundamentale în lumea sistemelor de operare; cealaltă abstracie este procesul. Dacă procesul abstractizează execuia unei anumite sarcini pe procesor, fiierul abstractizează informaia persistentă a unui sistem de operare. Un fiier este folosit pentru a stoca informaiile necesare funcionării sistemului de operare i interaciunii cu utilizatorul.

Un sistem de fiiere este un mod de organizare a fiierelor i prezentare a acestora utilizatorului. Din punct de vedere al utilizatorului un sistem de fiiere are o structură ierarhică de fiiere i directoare, începând cu un director rădăcină. Localizarea unei intrări (fiier sau director) se realizează cu ajutorul unei căi în care sunt prezentate toate intrările de până atunci. Astfel, calea

```
/usr/local/app/file.txt
```

înseamnă că directorul rădăcină `/` are un subdirector `usr` urmat de subdirectorul `local`. Acesta are la rândul său are un subdirector `app` care conine un fiier `file.txt`.

Fiecare fiier are asociat, aadar, un nume cu ajutorul căruia se face identificarea, un set de drepturi de acces i zone coninând informaia utilă.

Sistemele de fiiere suportate de sistemele de operare de tip Unix i Windows sunt ierarhice. Caracterele interzise în nume sunt `/` în Unix i `?`, `"`, `/`, `\`, `<` , `>` , `*`, `|`, `:` în Windows. De altfel, este recomandat ca denumirile de fiiere să nu se termine cu punct sau spaiu în Windows.

O diferenă majoră între Linux/Unix i Windows este tratarea literelor mari sau mici. Linux/Unix sunt case-sensitive (`Data` este diferit de `data`), iar Windows nu.

Ierarhia sistemului de fiiere Unix are un singur director cunoscut sub numele de `root` i notat `/`, prin care se localizează orice fiier. Notaia Unix pentru căile fiierelor este un ir de nume de directoare despărite prin `/`, urmat de numele fiierului. Există i căi relative la directorul curent `.` sau la directorul părinte `..`.

În Unix nu se face nicio deosebire între fiierele aflate pe partiiile discului local, pe CD sau pe o maină din reea. Toate aceste fiiere vor face parte din ierarhia unică a directorului root. Acest lucru se realizează prin **montare**: sistemele de fiiere vor fi montate într-unul din directoarele sistemului de fiiere rădăcină.

În Windows există mai multe ierarhii, câte una pentru fiecare partiie i pentru fiecare loc din reea. Spre deosebire de Unix, delimitatorul între numele directoarelor dintr-o cale este `\`, i pentru căile absolute trebuie specificat numele ierarhiei în forma `C:\`, `E:\` sau `\\FILESERVER\myFile` (pentru reea). Ca i Unix, Windows folosete `.` i `..`.

# Operaii pe fiiere

Un **descriptor de fiier** în Unix este un întreg care indexează o tabelă cu pointeri spre structuri care descriu fiierele deschise de un proces. Un program care rulează într-un shell Unix îi deschide 3 fiiere standard cu file descriptori cu valori speciale:

- **standard input** (cu file descriptorul 0) (implicit, citit de la tastatură - de la terminal)
- **standard output** (cu file descriptorul 1) (implicit, afiat pe ecran - la terminal)
- **standard error** (cu file descriptorul 2) (implicit, afiat pe ecran - la terminal)

Pentru a asocia ali descriptori cu obiecte deschise, se folosete apelul de sistem `open`.

Pe Windows noiunea de bază pentru managementul fiierelor este **handle**-ul, o valoare din care se obine un pointer spre o structură descriptivă a fiierului. Aceleai 3 fiiere standard sunt deschise de fiecare program, i orice fiier suplimentar se deschide cu `OpenFile` sau `CreateFile`.

În continuare, pentru descrierea comportării operaiilor de intrare-ieire pe Windows, s-a ales ca toate apelurile să facă parte din API-ul Win32, care este cel mai aproape de kernelul Windows. Sistemul oferă ca alternativă apeluri standard (POSIX, de exemplu, compatibile între Windows i Linux), dar acestea se implementează în Windows prin apelurile Win32 i formează un nivel mai îndepărtat de kernel.

Un fiier are asociat **cursorul de fiier** (file pointer) care indică poziia curentă în cadrul fiierului. Cursorul de fiier este un întreg care reprezintă deplasamentul (offset-ul) faă de începutul fiierului.

Operaiile tipice de executat pe un fiier sunt prezentate în continuare. Exceptând deschiderea unui fiier, toate operaiile primesc ca argument descriptorul sau handle-ul acelui fiier. Pentru deschiderea fiierului se folosete ca argument numele acestuia.

- **deschiderea unui fiier**: înseamnă asocierea unui descriptor de fiier sau un handle cu un fiier; acest descriptor sau handle este folosit în cadrul celorlalte operaii; fiierul este identificat prin nume; apeluri pentru deschiderea unui fiier sunt `fopen` (ISO C), `open` (POSIX), `CreateFile` (Win32 API); aceste apeluri pot fi folosite i pentru crearea unui fiier; alternativ există apelul `creat` (POSIX).
- **închiderea unui fiier**: înseamnă eliberarea structurilor de fiier asociate procesului i a descriptorului (handle-ului) acelui fiier; apelurile sunt `fclose` (ISO C), `close` (POSIX), `CloseFile` (Win32 API)
- **citirea dintr-un fiier**: înseamnă copierea unui bloc de date într-un buffer; după ce se realizează citirea se actualizează cursorul de fiier; apelurile sunt `fread` (ISO C), `read` (POSIX), `ReadFile` (Win32 API)
- **scrierea într-un fiier**: înseamnă copierea unui bloc de date dintr-un buffer în fiier; efectuarea scrierii înseamnă i actualizarea cursorului de fiier; apelurile sunt `fwrite` (ISO C), `write` (POSIX), `WriteFile` (Win32 API)
- **poziionarea într-un fiier**: înseamnă schimbarea valorii cursorului de fiier; citiri sau scrieri ulterioare vor porni din locul indicat de acest cursor de fiier; apelurile sunt `fseek` (ISO C), `lseek` (POSIX), `SetFilePointer` (Win32 API)
- **schimbarea atributelor unui fiier**: înseamnă stabilirea unor parametri pentru fiier; apelurile sunt `fcntl` (POSIX), `SetFileAttributes` (Win32 API)

# Operaii pe fiiere în Linux

În continuare sunt descrise operaiile de fiiere într-un sistem Linux. Apelurile descrise mai jos sunt valabile în orice sistem compatibil POSIX.

## Crearea, deschiderea i închiderea fiierelor

### open

Funcia open este folosită pentru deschiderea unui fiier; funcia este declarată în `fcntl.h` iar sintaxa apelului este una din următoareele:

```
int open(const char *FILENAME, int FLAGS);
int open(const char *FILANAME, int FLAGS, mode_t MODE);
```

Dacă, spre exemplu, dorim să deschidem fiierul `alina.txt` pentru scriere, cu trunchiere, i fiierul `dan.txt` pentru citire i scriere, cu eventuala creare a acestuia, vom folosi următoarea secvenă de cod:

**Exemplu 1. io-01.c**

```
[...]
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

[...]
int fd1, fd2;

fd1 = open ("alina.txt", O_WRONLY | O_TRUNC);
if (fd1 < 0) {
perror ("open");
exit (EXIT_FAILURE);
}

fd2 = open ("dan.txt", O_RDWR | O_CREAT, 0644);
if (fd1 < 0) {
perror ("open");
exit (EXIT_FAILURE);
}
```

**ATENIE!** O greeală frecventă este omiterea drepturilor de creare a fiierului (`0644` în exemplul de mai sus) când se apelează `open` cu flag-ul `O_CREAT` activat.

## creat

Declarată tot în `fcntl.h` este i funcia `creat`, care creează un fiier i are următoarea sintaxă:

```
int creat(const char *FILENAME, mode_t MODE);
```

Funcia este echivalentă cu:

```
open(FILENAME, O_WRONLY|O_CREAT|O_TRUNC, MODE);
```

## close

Funcia este declarată în `unistd.h` iar sintaxa apelului este următoarea:

```
int close(int FILEDES)
```

unde `FILEDES` este filedescriptorul care se dorete închis.

Dacă dealocarea are loc fără probleme atunci valoarea întoarsă este `0`, altfel `-1`. Ca i la `open`, mai multe informaii despre cum s-a terminat operaia se pot obine inspectând variabila `errno`.

O greeală frecventă de programare este neverificarea codului de eroare întors la `close`, pentru că se poate întâmpla ca o eroare la scriere (`EIO`) să fie întoarsă utilizatorului abia la `close`.

# Scrierea i citirea

Scrierea într-un fiier i citirea dintr-un fiier se realizează cu ajutorul apelurilor `read` i `write`. Aceste apeluri primesc trei argumente:

1. descriptorul fiierului folosit
2. buffer-ul ce conine datele pentru scriere sau unde se stochează datele citite din fiier
3. numărul de octei care vor fi citii/scrii

## read

Funcia read este declarată în `unistd.h` i are sintaxa de apel:

```
ssize_t read(int FILEDES, void *BUFFER, size_t SIZE);
```

Funcia întoarce numărul de octei citii. Valoarea minimă este de un octet, iar când se ajunge la sfârit se va întoarce `0`. Dacă se face read după ce s-a ajuns la sfârit se va întoarce în continuare `0`. Dacă apare o eroare se întoarce `−1` i variabila `errno` este setată corespunzător cauzei care a determinat eroarea.

## write

Funcia write este declarată, de asemenea, în `unistd.h` iar sintaxa este următoarea:

```
ssize_t write(int FILEDES, const void *BUFFER, size_t SIZE);
```

unde parametrii au semnificaii similare cu cei ai apelului `read`. Valoarea întoarsă este numărul de octeti ce au fost efectiv scrii. În mod implicit nu se garanteaza că la revenirea din write scrierea în fiier s-a terminat. Pentru a fora actualizarea se poate folosi `fsync` sau fiierul se poate deschide folosind flagul `O_FSYNC`, caz în care se garanteaza că după fiecare `write` fiierul a fost actualizat.

**Observaie**: pentru `read` i `write` există o versiune `pread`, respectiv `pwrite`, care permite specificarea unui offset în fiier de la care să se efectueaze operaia de citire / scriere. Mai există i o versiune `pread64` i `pwrite64` la care offset-urile sunt pe 64 de bii.

# Poziionarea în fiier (lseek)

Funcia lseek permite mutarea cursorului unui fiier la o poziie absolută sau relativă. Similar cu `read` i `write`, este declarată în `unistd.h`. Sintaxa este următoarea:

```
off_t lseek(int FILEDES, off_t OFFSET, int WHENCE)
```

Valoarea întoarsă reprezintă offset-ul la care s-a ajuns; un `lseek` pe `SEEK_CUR` cu `OFFSET` zero întoarce poziia curentă fără a o modifica.

Pentru această funcie există i o versiune `lseek64` la care `OFFSET`-ul este pe 64 de bii. Tipul `off_t` este un tip aritmetic folosit pentru a reprezenta dimensiuni de fisiere. În sistemele GNU el este un `long int`.

`lseek` permite i poziionări după sfâritul fiierului. Scrierile care se fac în astfel de zone nu se pierd ceea ce se obine fiind un fisier cu... goluri, o zonă care este "sărită" nu este alocată pe disc.

# Exemplu utilizare operaii I/O

**Exemplu 2. io-02.c**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main (void)
{
int fd;
char *buf;
        ssize_t bytes_read;

/* alocam spatiu pentru buffer-ul de citire */
      malloc=(101);
if (buf == NULL) {
perror ("malloc");
exit (EXIT_FAILURE);
             }

             /* deschidem fisierul */
        fd =      open ("gabi.txt", O_RDONLY);
             if (fd < 0) {
perror ("open");
             exit (EXIT_FAILURE);
             }

/* ne poziionăm în fiier */
if (lseek (fd, -100, SEEK_END) < 0) {
perror ("lseek");
             exit (EXIT_FAILURE);
             }

             /* citim ultimele 100 caractere in buffer */
        bytes_read =      read (fd, buf, 100);
             if (bytes_read < 0) {
perror ("read");
             exit (EXIT_FAILURE);
             }
        buf     [bytes_read] = '\0';

             /* afisam sirul citit */
             printf("file contents [%ld bytes]: \n%s\n", bytes_read, buf);

             /* inchidem fisierul */
             close (fd);

             /* eliberam buffer-ul alocat */
             free (buf);

             return 0;
```

```
}
```

## Operaii speciale (fcntl)

Funcia fcntl permite efectuarea unor operaii speciale asupra descriptorilor de fiier. Sintaxa este următoarea:

```
int fcntl(int FILEDES, int COMMAND, ...);
```

`COMMAND` este tipul operaiei; o parte din valorile posibile i semnificaiile acestora sunt prezentate în urmatorul tabel:

| **COMMAND** | **efect** |
|---|---|
| `F_DUPFD` | duplicarea unui filedescriptor |
| `F_GETFD` | obinerea flagului `CLOSE_ON_EXEC` (`FD_CLOEXEC`); semnificaia flagului va fi discutată în cadrul laboratorului de procese |
| `F_SETFD` | setarea flagului `CLOSE_ON_EXEC` |
| `F_GETFL` | obinerea flagurilor file-descriptorului |
| `F_SETFL` | setarea flagurilor file-descriptorului |
| `F_GETLK` | obinerea informaiilor despre un lock; lock-urile sunt folosite pentru reglementarea accesului concurent la un fiier |
| `F_SETLK` | obinerea / eliberarea unui lock |
| `F_SETLKW` | similar cu `F_SETLK` dar se ateaptă terminarea operaiei |
| `F_GETOWN` | obinerea PID-ul procesului care primete semnalul `SIGIO`; acest semnal este trimis de nucleu atunci când se lucrează cu fiierul în mod asincron; despre operaii I/O asincrone vom discuta într-un alt laborator |
| `F_SETOWN` | stabilirea procesului care va primi semnalul `SIGIO` |

# Operaii pe fiiere în Windows

## Generalităi

### NTFS

Sistemele de fiiere native folosite pe Windows sunt FAT32 i NTFS.

NTFS consideră un fiier ca fiind o colecie de atribute i valori pentru acestea. Exemple de atribute: meta-date despre fiier (nume, lungime, timestamp), coninutul fiierului, informaii de securitate. Fiierele sunt stocate folosind **clustere**, care sunt grupări de sectoare fizice de pe disc; cluster-ele sunt folosite pentru o mai bună

performană. Numărul de sectoare care formează un cluster depinde de dimensiunea partiiei, dar poate fi adaptat i la alte valori (cu cât partiia e mai mare, cu atât un cluster conine mai multe sectoare).

Mai multe detalii despre NTFS putei găsi aici.

Fiierele i directoarele sunt i în Windows obiecte securizate; cererea de drepturi asupra unui fiier se face prin intermediul a două flag-uri, date ca parametri către funciile de lucru cu fiiere:

- **modul de acces** specifică setul de operaii pe care un proces îl poate face pe un obiect.
- **modul de partajare** specifică felul în care mai multe procese pot partaja un acelai obiect

### Restricii asupra numelor de fiiere

Windows nu permite folosirea de nume `rezervate` pentru a denumi fiiere. Numele rezervate sunt ale unor device-uri speciale de sistem (cum ar fi LPT2, COM1). Numele unui fiier este case-insensitive i lungimea maximă a unui nume este de `MAXPATH` (260) de caractere (incluzand prefix-ul de drive si caracterul `'\0'` de sfârit de string).

### Handle-uri către fiiere

Un `handle` este un identificator pentru un obiect gestionat de kernel-ul Windows. În particular, pentru a ne referi la un fiier, vom folosi un `handle` către acesta, care va avea un rol asemanător cu descriptorul de fiier din Unix. Vom discuta mai multe detalii despre `handle`-uri la laboratorul de procese.

## Crearea, deschiderea i închiderea fiierelor

### Crearea / deschiderea

Pentru a crea un handle asociat cu un fiier, director sau altă resursă abstractizată sub forma unui fiier (port COM, pipe, modem, etc.) se folosete funcia CreateFile. Funcia se ocupă atât de crearea, cât i de deschiderea unui fiier (i întoarce în ambele cazuri un handle asociat cu fiierul):

```
#include <windows.h>
HANDLE CreateFile(
        LPCTSTR lpFileName,
        DWORD dwDesiredAccess,
        DWORD dwShareMode,
        LPSECURITY_ATTRIBUTES lpSecurityAttributes,
        DWORD dwCreationDisposition,
        DWORD dwFlagsAndAttributes,
        HANDLE hTemplateFile
);
```

Drepturile de acces cerute la deschiderea fiierului sunt specificate in dwDesiredAccess [in]. Există variabile generice care încapsulează drepturile de acces, cum ar fi `GENERIC_EXECUTE`, `GENERIC_READ` i `GENERIC_WRITE`.

Un fiier poate fi deschis de mai multe ori (de procese diferite, sau de acelai proces). În acest caz, la prima deschidere, parametrul `dwShareMode [in]` va avea una din valorile:

- `FILE_SHARE_DELETE` permite unor operaii de deschidere ulterioare să capete acces de tip *delete*.
- `FILE_SHARE_READ` permite unor operaii de deschidere ulterioare să capete acces de tip *read*.
- `FILE_SHARE_WRITE` permite unor operaii de deschidere ulterioare să capete acces de tip *write*.

La deschiderea unui fiier se poate preciza prin parametrul `lpSecurityAttributes [in]` modul în care handle-ul returnat de apel poate fi motenit de procesele fii ale procesului apelant. Daca `lpSecurityAttributes` este `NULL`, handle-ul nu poate fi motenit.

Parametrul `dwCreationDisposition [in]` precizează modul în care apelul acionează în cazul în care fiierul există sau nu; poate avea valori de forma:

- `CREATE_ALWAYS` creează un fiier nou; dacă fiierul există, apelul îl suprascrie, tergând atributele existente;
- `CREATE_NEW` creează un fiier nou; apelul euează dacă fiierul există deja;
- `OPEN_ALWAYS` deschide fiierul, daca acesta există; altfel, se comportă ca i `CREATE_NEW`;
- `OPEN_EXISTING` deschide fiierul; dacă nu există, apelul euează;
- `TRUNCATE_EXISTING` deschide fiierul (cu drept de acces `GENERIC_WRITE`) i îl trunchiază la dimensiunea zero; dacă fiierul nu există, apelul euează.

Un set de flaguri i atribute suplimentare (valabile numai în cazul fiierelor) pot fi precizate în `dwFlagsAndAttributes [in]`. Valori uzuale sunt:

- `FILE_ATTRIBUTE_NORMAL` fiierul nu are alte atribute setate (folosit numai singur)
- `FILE_ATTRIBUTE_READONLY` fiierul va fi read only pentru toate procesele

`hTemplateFile [in]` este un handle la un fiier template cu drepturi de acces pentru citire, template ale cărui atribute i flaguri vor fi folosite i pentru fiierul nou creat.

Apelul returnează în caz de succes un handle al fiierului. Dacă `dwCreationDisposition` este `CREATE_ALWAYS` sau `OPEN_ALWAYS`, apelul nu euează, dar `GetLastError` returnează `ERROR_ALREADY_EXISTS`. În caz de eroare, apelul returnează `INVALID_HANDLE_VALUE`, iar informaii suplimentare despre eroare pot fi returnate de `GetLastError`.

Următoarea secvenă de cod deschide fiierul `alina.txt` pentru scriere i îl trunchiază. Fiierul `dan.txt` este deschis pentru citire i scriere i este creat dacă nu există:

**Exemplu 3. io-03.c**

```
#include <windows.h>

 HANDLE handle1, handle2;

 handle1 = CreateFile(
         "alina.txt",
         GENERIC_READ,     /* mod acces */
         FILE_SHARE_READ,  /* tip partajare */
         NULL/* atribute securitate */
         OPEN_EXISTING,    /* creare daca nu exista? */
         FILE_ATTRIBUTE_NORMAL,         /* atribute fisier */
         NULL);/* sablon pentru GENERIC_READ */
```

```
if (handle1 == INVALID_HANDLE_VALUE)
        HandleError();

handle2 = CreateFile(
        "dan.txt",
        GENERIC_READ | GENERIC_WRITE,
        0,
        NULL,
        CREATE_NEW,
        FILE_ATTRIBUTE_NORMAL,
        NULL);
if (handle2 == INVALID_HANDLE_VALUE)
        HandleError();
```

Un apel `FormatMessage` cu `GetLastError()` printează *The system cannot find the file specified* dacă fiierul nu există.

Pentru copierea i mutarea fiierelor există apelurile `CopyFile`, `MoveFile` i `ReplaceFile`.

## Închiderea

Când fiierul nu mai este folosit, fiierul este închis cu apelul generic pentru orice tip de handle-uri CloseHandle

```
BOOL CloseHandle(
        HANDLE hObject
 );
```

### tergerea

tergerea se face prin închiderea fiierului i folosirea apelului de sistem DeleteFile

```
CloseHandle(hFile);
DeleteFile("myfile.txt");
```

unde `DeleteFile` are signatura

```
BOOL DeleteFile( LPCTSTR lpFileName );
```

# Citirea i scrierea unui fiier

Un proces poate face operaii de citire i de scriere asupra unui fiier prin apelurile de sistem `ReadFile`, `WriteFile` i cele extinse `ReadFileEx`, `WriteFileEx`. Apelurile primesc un handle de fiier creat cu drepturi corespunzătoare i scriu sau citesc un număr specificat de octei începând cu poziia curentă a cursorului (sau dintr-o altă poziie specificată).

## Citirea

`ReadFile` operează asupra unui fiier care are drepturi de acces cel puin pentru citire, copiind un număr de octei (începând cu poziia curentă a cursorului de fiier) într-un buffer i întoarce într-o variabilă numărul de

octei citii. În mod normal, după o astfel de operaie, cursorul de fiier este actualizat cu numărul de octei citii. Singura excepie este cazul în care fiierul este deschis pentru operaii de I/O de tip OVERLAPPED, caz în care conceptul de cursor de fiier nu mai este folositor (i deci nu mai este actualizat).

Apelul `ReadFile` poate opera atât sincron, cât i asincron; `ReadFileEx` este exclusiv asincronă. În cadrul acestui laborator vom insista pe folosirea sincronă, urmând ca cea asincronă (mai complicată, dar care oferă performane mai bune în cazul accesului concurent) să fie prezentată într-un laborator ulterior:

```
BOOL ReadFile(
        HANDLE hFile,
        LPVOID lpBuffer,
        DWORD nNumberOfBytesToRead,
        LPDWORD lpNumberOfBytesRead,
        LPOVERLAPPED lpOverlapped
);
```

`ReadFile` primete un handle de fiier `hFile [in]`, care trebuie să fi fost creat cu un drept de acces cel puin egal cu `GENERIC_READ`. Pentru ca operaia să fie asincronă, hFile trebuie să fi fost deschis cu FILE_FLAG_OVERLAPPED precizat în `CreateFile`. Bufferul în care se copiază rezultatul citirii este `lpBuffer [out]`.

Numărul de octei care se dorete a fi citit se precizează în `nNumberOfBytesToRead [in]`, iar numărul efectiv citit este returnat în variabila pointată de `lpNumberOfBytesRead [out]`.

`ReadFile` returnează o valoare diferită de zero în caz de succes, i zero altfel. Dacă se returnează o valoare diferită de zero, dar numărul de octei citii este zero, atunci s-a ajuns la sfâritul de fiier.

O operaie de citire simplă dintr-un fiier poate arăta astfel:

## Scrierea

Apelul `WriteFile` copiază în mod sincron sau asincron un număr specificat de octei dintr-un buffer în coninutul unui fiier i returnează într-o variabilă numărul efectiv de octei copiai. Scrierea în fiier se face în general începând din poziia curentă a cursorului i după terminarea operaiei poziia cursorului fiierului este actualizată (rămân valabile observaiile anterioare despre operaii OVERLAPPED).

```
BOOL WriteFile(
        HANDLE hFile,
        LPCVOID lpBuffer,
        DWORD nNumberOfBytesToWrite,
        LPDWORD lpNumberOfBytesWritten,
        LPOVERLAPPED lpOverlapped
);
```

Handle-ul de fiier în care se scrie `hFile [in]` trebuie să fi fost creat cu drepturi de acces `GENERIC_WRITE`. Parametrii `WriteFile` au aceleai semnificaii cu parametrii `ReadFile`, adaptate pentru operaii de scriere.

# Poziionarea in fiier

Fiecare fiier deschis are asociat un cursor (memorat pe 64 de bii) care reprezintă poziia curentă de citire/scriere. Un proces poziionează cursorul la un offset specificat cu SetFilePointer (pentru apelarea căruia trebuie inclus windows.h):

```
DWORD SetFilePointer(
         HANDLE hFile,
         LONG lDistanceToMove,
         PLONG lpDistanceToMoveHigh,
         DWORD dwMoveMethod
 );
```

Fiierul destinaie al operaiei are handle-ul hFile [in] i trebuie să fi fost în prealabil creat cu unul din drepturile de acces GENERIC_READ sau GENERIC_WRITE.

Numărul de octei cu care se mută cursorul este specificat de lDistanceToMove [in] i lpDistanceToMoveHigh [in, out]; cele două câmpuri de 32 de bii formează o valoare de 64 de bii. Uzual cel de-al doilea câmp este NULL. O valoare pozitivă înseamnă o deplasare înainte, iar una negativă, înapoi.

Parametrul dwMoveMethod specifică punctul de start pentru mutarea cursorului, i poate avea una din valorile:

- FILE_BEGIN punctul de start este începutul fiierului; liDistanceToMove este considerat unsigned
- FILE_CURRENT punctul de start este valoarea curentă a cursorului
- FILE_END punctul de start este valoarea curentă a sfâritului de fiier

Apelul returnează noua valoare a cursorului, dacă lpDistanceToMoveHigh este NULL; altfel, se returnează jumătatea low a valorii, jumătatea high luând locul lpDistanceToMoveHigh. În caz de eroare, valoarea returnată este egală cu INVALID_SET_FILE_POINTER. SetFilePointer poate fi folosit i pentru a interoga poziia curentă a cursorului, precizând punctul de start FILE_CURRENT i offsetul zero:

```
DWORD currentPosition;
 currentPosition = SetFilePointer(myFileHandle,
                                  0,
                                  NULL,
                                  FILE_CURRENT);
 if (INVALID_SET_FILE_POINTER == currentPosition)
         HandleError();
 else
         printf("current position is %ld\n", currentPosition);
```

Varianta extinsă SetFilePointerEx a apelului SetFilePointer memorează valoarea cursorului într-un singur câmp, în loc de două câmpuri separate, apelul extins făcând lucrul cu valorile cursorului mai uor.

Un fiier poate fi trunchiat sau extins folosind apelul SetEndOfFile, care face poziia sfâritului de fiier EOF egală cu poziia curentă a cursorului fiierului. În cazul extinderii fiierului peste limita sa, coninutul adăugat este nedefinit.

```
BOOL SetEndOfFile(
```

```
        HANDLE hFile
);
```

Fiierul este precizat de parametrul `hFile`; acesta trebuie să fi fost în prealabil creat cu dreptul `GENERIC_WRITE`.

Apelul întoarce o valoare diferită de zero în caz de succes, i zero altfel.

# Exemplu utilizare operaii I/O

**Exemplu 4. io-04.c**

```
#include <stdio.h>
 #include <string.h>
 #include <stdlib.h>
 #include <windows.h>

 #define BUF_SIZE       100

 static void HandleError (void)
 {
        CHAR errBuf[1024];

        FormatMessage(
                FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_MAX_WIDTH_MASK,
                NULL/* nicio definire a mesajului */
                GetLastError(),
                0,
                errBuf, sizeof(errBuf) - 1,
                NULL);/* fara argumente variabile */

             fprintf (stderr, "%s\n", errBuf);
 }

 int main (void)
 {
    HANDLE myFileHandle;
    CHAR outBuffer[BUF_SIZE+1];
    DWORD bytesRead;
    DWORD currentPosition;
    SIZE_T bytesToRead = BUF_SIZE;
    BOOL retRead;

    /* deschidem fisierul */
    myFileHandle = CreateFile(
            "myfile.txt",
            GENERIC_READ,
            FILE_SHARE_READ,
            NULL/* fara atribute de securitate */
            OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL,
            NULL/* fara sablon */
    );
    if (myFileHandle == INVALID_HANDLE_VALUE) {
       HandleError();
       ExitProcess(-1);
    }
```

```
    /* ne pozitionam în fisier */
    currentPosition = SetFilePointer(
            myFileHandle,
            -100,
            NULL,/* caz pozitionare pe 64 biti */
            FILE_END
    );
    if (currentPosition == INVALID_SET_FILE_POINTER) {
        HandleError();
        ExitProcess (-1);
    }

    /* citim ultimele 100 caractere in buffer */
    retRead = ReadFile(
            myFileHandle,
            outBuffer,
            bytesToRead,
            &bytesRead,
            NULL);/* nimic asincron */
    if (retRead == FALSE) {
            HandleError();  /* functie de tratare a erorii */
            ExitProcess (-1);
    }

    /* afisam sirul citit */
    outBuffer[bytesRead] = '\0';
    printf("file contents [%ld bytes]: \n%s\n", bytesRead, outBuffer);

    /* inchidem fisierul */
    CloseHandle (myFileHandle);

    return 0;
}
```

# Redirectări

Redirectările sunt operaiile prin care ieirea, intrarea sau erorile sunt redirectate în fiiere sau către intrarea altui proces. Câteva exemple de redirectări sunt prezentate mai jos:

```
$ ls > output 2> error
$ cat /etc/services | grep tcp
```

Redirectarea ieirii către un fiier se realizează cu ajutorul operatorului >. Redirectarea intrării dintr-un fiier se realizează cu ajutorul operatorului <. Redirectarea erorii într-un fiier se realizează cu ajutorul operatorului `2>`. În fine, redirectarea ieirii unei comenzi către o altă comandă se realizează cu ajutorul operatorului `|`.

## Redirectări în Linux

În Linux redirectările se realizează simplu, cu ajutorul funciilor de duplicare a file descriptorilor. De exemplu, pentru redirectarea ieirii în fiierul output.txt, sunt necesare două linii de cod:

```
fd = open("output.txt", O_RDWR|O_CREAT|O_TRUNC, 0600);
dup2 (fd, STDOUT_FILENO);
```

Dacă se dorete să se redirecteze intrarea, ieirea sau eroarea unui proces nou creat, aceai secvenă trebuie efectuată în procesul copil, după `fork`, dar înainte de `exec`. Trebuie de asemenea să ne asigurăm că descriptorii de fiier nu au fost deschii cu flagul `CLOSE_ON_EXEC`.

# Redirectări în Windows

În Windows, redirectarea i crearea unui proces cu unul din handlerele standard redirectat se face după metode diferite. Astfel, pentru redirectarea unuia din handlerele standard se vor folosi funciile:

```
HANDLE GetStdHandle(int std_handle);
BOOL SetStdHandle(int std_handle, HANDLE handle);
```

unde `std_handle` poate fi `STD_INPUT_HANDLE`, `STD_OUTPUT_HANDLE` sau `STD_ERROR_HANDLE`.

Pentru redirectări într-un proces nou creat, fiierul în/din care se redirectează trebuie creat astfel încât să fie motenibil, pentru ca handle-ul să poată fi motenit în procesele copii ale procesului curent.

Apoi, apelul `CreateProcess` de creare a procesului care va avea handle-ul redirectat trebuie să aibă parametrul `bInheritHandles` cu valoarea `TRUE` i câmpurile prezentate mai jos trebuie să aibă valorile corespunzătoare:

```
struct _STARTUPINFO {
        DWORD cb;
        DWORD dwFlags;
        HANDLE hStdInput;
        HANDLE hStdOutput;
        HANDLE hStdError;
} STARTUPINFO;
```

Câmpul cb trebuie să fie lungimea în octei a structurii. Pentru a lua în considerare noile handlere, dwFlags trebuie setat la STARTF_USESTDHANDLES. Handlerele pentru redirectare se specifică în `hStdInput`, `hStdOutput` i `hStdError`.

Un exemplu de redirectare a ieirii unui proces nou creat este prezentat mai jos:

**Exemplu 3. io-05.c**

```
STARTUPINFO si;
 PROCESS_INFORMATION pi;
 HANDLE myFileHandle;
 SECURITY_ATTRIBUTES sa;

 [...]

 ZeroMemory (&sa, sizeof (sa));
 sa.bInheritHandle = TRUE;
 myFileHandle = CreateFile(
         fileName,
         GENERIC_READ|GENERIC_WRITE,
         FILE_SHARE_READ,
         &amp;sa,
         OPEN_ALWAYS,
         FILE_ATTRIBUTE_NORMAL,
         NULL
```

```
        );
if (myFileHandle == INVALID_HANDLE_VALUE) {
        HandleError(); // functie proprie de tratare a erorii
}

[...]
ZeroMemory (&pi, sizeof (pi));
ZeroMemory (&si, sizeof (si));
si.cb = sizeof(si);
si.dwFlags = STARTF_USESTDHANDLES;
si.hStdOutput = myFileHandle;
if(!CreateProcess(
        NULL,
        comm,
        NULL,
        NULL,
        TRUE,
        0,
        NULL,
        NULL,
        &si,
        &pi)) {
        HandleError();
}
[...]
```

# Wrapper-e

În domeniul sistemelor de operare, prin wrapper înelegem un layer software subire (a se citi: care nu aduce un overhead prea mare) peste sistemul de operare, cu scopul de a abstractiza serviciile oferite de acesta, adaptându-le la o interfaă comună. Interfaa comună este astfel definită incât să se potrivească cu mai multe sisteme de operare. Programele pe care le scriem ulterior nu vor folosi direct apelurile de sistem specifice fiecărui sistem de operare, ci interfaa comună.

Un wrapper este folositor atunci când dorim să scriem software portabil pe mai multe platforme (spre exemplu, temele de la Sisteme de Operare) cu un "overhead" minim de portare i fară a plati un pre prea scump măsurat în performană (după cum tii, există i alte soluii pentru această problemă, de exemplu, maina virtuală Java - JVM).

Una din metodele posibile pentru realizarea unui wrapper este folosirea preprocesorului. Să presupunem că incercăm să abstractizăm conceptul de fiier i operaiile disponibile cu el. Vom exemplifica doar operaiile de read/write.

**Exemplu 6. io-wrapper.h**

```
#ifdef __linux__

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
typedef int os_handle;
typedef size_t os_size;
typedef ssize_t  os_ssize;

#elif defined(_WIN32)
```

```
#include <windows.h>
typedef HANDLE os_handle;
typedef DWORD os_size;
typedef DWORD os_ssize;

#else
#error "Unknown OS!"
#endif

os_ssize os_read(os_handle fd, void* buffer, os_size count);
os_ssize os_write(os_handle fd, const void* buffer, os_size count);
```

Observăm că în funcie de sistemul de operare definit, diferă:

- fiierele header incluse
- definiiile tipurilor cu care lucrează wrapper-ul

De asemenea, observăm că semnăturile funciilor definite sunt identice pentru ambele sisteme de operare. Iată un exemplu de implementare a lor:

**Exemplu 6. io-wrapper.c**

```
#include "io-wrapper.h"

#ifdef __linux__
os_ssize os_read(os_handle fd, void *buffer, os_size count) {
    return read(fd, buffer, count);
}

os_ssize os_write(os_handle fd, const void *buffer, os_size count) {
    return write(fd, buffer, count);
}

#elif defined(_WIN32)
os_ssize os_read(os_handle fd, void *buffer, os_size count) {
    os_ssize result = -1;
    ReadFile(fd, buffer, count, &result, NULL);
    return result;
}

os_ssize os_write(os_handle fd, void *buffer, os_size count) {
    os_ssize result = -1;
    WriteFile(fd, buffer, count, &result, NULL);
    return result;
}

#endif
```

Acum putem genera fiiere executabile compatibile cu o platformă Linux sau Windows, în funcie de un singur macro, definit automat de catre compilator.

Observăm că folosind această tehnică putem să convertim inclusiv între procedură i funcie (funciile de pe Windows primesc ca parametru transmis prin referină numărul de octei citii/scrii, iar cele de pe Linux îl întorc direct). Desigur, abordarea de mai sus este incompletă, pentru că ar fi trebuit convertite i codurile de eroare într-un format comun.

Odată scris acest wrapper, putem folosi în continuare funciile os_read i os_write pentru a citi / scrie din fiiere, fară a ne preocupa de sistemul de operare pe care rulează programul nostru. Acesta este însă un caz fericit, pentru că aa după cum vei observa la laboratorul de procese, nu toate serviciile oferite de sisteme de operare diferite se pot "unifica" atât de uor (este vorba de fork() + exec() vs. CreateProcess).

# Exerciii

## Quiz

Pentru autoevaluare raspundeti la intrebarile din acest quiz.

## Exercitii pre-laborator

1. (Linux) Rulati **strace** pentru urmatoarele comezi, pentru a observa apelurile de sistem utilizate (identificati-le pe cele relevante).
    - touch test_file
    - cp /etc/network/interfaces test_file
    - ls -l test_file
    - rm test_file
2. (Platform Independent) Realizai un program denumit mycat care simulează comanda cat de afiare a coninutului unui fiier. Comanda primete ca argument fiierul de afiat. Pentru testare puteti folosi chiar fisierul sursa C.
3. (Platform Independent) Realizai un program denumit `mysize` care afiează dimensiunea fiierului primit ca argument prin citirea coninutului acestuia (folosind `read`, respectiv `ReadFile`). De ce nu se recomandă aflarea dimensiunii unui fiier în acest fel? Care funcii se recomandă să se folosească pe Linux, respectiv Windows?
4. (Platform independent) Sa se scrie un program care citeste si afiseaza un octet de la o pozitie aleatoare dintr-un fisier folosind o singura parcurgere a fisierului si fara sa citeasca tot fisierul in memorie.

## Exercitii de laborator

In rezolvarea exercitiilor puteti folosi arhiva de sarcini a laboratorului.

1. (Platform Independent) Intrai în directorul 01_xfile al arhivei. Studiai coninutul său i completai în fiierul xfile.c funciile xread i xwrite astfel încât programul `xtest`, obinut prin compilare, să producă mesajul "OK". Hints:
    - programul testează implementarea voastră pentru cele două funcii; o descriere a funcionalităii lor găsii în `xfile.h`
    - revedei documentaia pentru apelurile de sistem read i write (pentru Linux), respectiv ReadFile i WriteFile pentru Windows
    - nu vă putei baza pe faptul că read/write/ReadFile/WriteFile funcionează corect (fără eroare), sau că scriu / citesc câi octei le-ai trimis, i nu mai puini. Verificai rezultatul întors de ele!

    **(1 punct)**

2. (Platform Independent) Intrai în directorul 02_fcomp al arhivei. Studiai coninutul său. Examinai fiierul xfile.h i observai că a mai apărut o funcie în el. Copiai fiierul xfile.c din directorul 01_xfile în acest director i implementai i funcia xlen. Completai apoi fiierul fcomp.c pentru a obine un program care compară două fiiere, afiând un mesaj care spune dacă ele sunt identice ca i coninut sau diferă. Hints:
    - revedei funciile lseek/SetFilePointer, pentru poziionarea în fiier, respectiv open/close/CreateFile/CloseHandle, pentru închiderea i deschiderea fiierelor
    - nu avei voie să citii fiierele în întregime în memorie. Putei citi doar bucăi limitate din ele, folosind un buffer.
    - putei folosi funciile xread i xwrite din xfile.c

    **(1 punct)**
3. (Platform Independent) Intrai în directorul 03_fcp al arhivei. Studiai coninutul său. Copiai fiierul xfile.c obinut la exerciiul 2 (din directorul 02_fcomp) i completai fiierul fcp.c astfel încât să obinei un program care copiază o poriune dintr-un fiier de intrare, descrisă prin (offset, lungime) într-un fiier de ieire. Ordinea parametrilor o putei afla examinând fiierul fcp.c. Hints:
    - putei folosi funciile definite în xfile.h
    - nu avei voie să citii toată poriunea din fiierul de intrare în memorie; folosii un buffer
    - atenie la drepturile cu care creai fiierul de ieire, dacă acesta nu există; recomandăm 644; revedei documentaia funciei creat pentru mai multe detalii

    **(1 punct)**
4. (Platform Independent) Intrai în directorul 04_fscatter al arhivei. Studiai coninutul său. Completai fiierul fscatter.c astfel încât să obinei un program care "sparge în bucăi" de o dimensiune maximă dată un fiier de intrare. Dacă fiierul de intrare se numete file.txt, bucăile se vor numi file.txt1, file.txt2,.., file.txtN, unde N este numărul de bucăi. Testai apoi funcionarea corectă a programului vostru folosind operatorul din shell ">>" i utilitarul "diff". Hints:
    - putei folosi funcia fcp, scrisă la exerciiul anterior (nu uitai să copiai xfile.h i xfile.c)
    - avand in vedere ca itoa nu este independenta de platforma si golurile de memorie ar trebui evitate, puteti folosi pentru concatenarea de string-uri si int-uri functia sprintf : char buffer[BUF_SIZE]; sprintf(buffer, "%s%d", nume_fisier, id);

    **(2 puncte)**
5. (Platform independent) Intrai în directorul 05_ffind al arhivei. Studiai coninutul său. Completai fiierul ffind.c astfel încât să obinei un program care are un efect similar cu comanda "ls". Pentru fiecare fiier, afiai: numele, dimensiunea i dacă este director sau nu ("Y"/"N"). Nu afiai fiierele speciale "." i "..". Hints:
    - pe Linux putei folosi funciile opendir, readdir i closedir pentru a enumera fiierele din director. Ele se regăsesc în fiierul header dirent.h
    - pe Linux putei folosi funcia stat pentru a obine informaiile despre fiiere
    - pe Linux, putei porni de la următorul exemplu
    - pe Windows putei folosi funciile FindFirstFile i FindNextFile pentru a enumera fiierele dintr-un director
    - pe Windows nu e nevoie să apelai o funcie suplimentară pentru a obine informaiile despre fiiere; ele se află în structura întoarsă de funciile de enumerare a fiierelor
    - pe Windows, pattern-ul de căutare a fiierelor trebuie să conină calea absolută până la directorul curent. Pentru a o obine, folosii funcia GetCurrentDirectory
    - pe Windows, putei porni de la următorul exemplu

    **(1 punct)**

    **Punctajul maxim se ia cu 10 puncte. Bonusul poate recupera lipsa de activitate de la alte laboratoare**.

    Pe Windows, putei compila în linie de comandă cu:

```
cl /EHsc file.c[pp]
```

# Soluții

Soluții exercițiii laborator 2

# Resurse utile

- Low-Level I/O (`info libc "Low-Level I/O"`)
- Duplicating Descriptors (`info libc "Duplicating Descriptors"`)
- File Management Functions