Laborator 02 - Operaţii I/O simple

Materiale ajutătoare

- lab02-slides.pdf
- lab02-refcard.pdf
- Video Operaţii IO

Nice to read

- TLPI - Chapter 4, `File I/O: The Universal I/O model`
- WSP4 - Chapter 2, `Using the Windows File System`

Fişiere. Sisteme de fişiere

Fişierul este una din abstractizările fundamentale în lumea sistemelor de operare; cealaltă abstractizare este procesul. Dacă procesul abstractizează execuţia unei anumite sarcini pe procesor, fişierul abstractizează informaţia persistentă a unui sistem de operare. Un fişier este folosit pentru a stoca informaţiile necesare funcţionării sistemului de operare şi interacţiunii cu utilizatorul.

Un **sistem de fişiere** este un mod de organizare a fişierelor şi prezentare a acestora utilizatorului. Din punct de vedere al utilizatorului un sistem de fişiere are o structură ierarhică de fişiere şi directoare, începând cu un director rădăcină. Localizarea unei intrări (fişier sau director) se realizează cu ajutorul unei căi în care sunt prezentate toate intrările de până atunci. Astfel, pentru calea `/usr/local/file.txt` directorul rădăcină `'/'` are un subdirector `usr` urmat de subdirectorul `local` ce conţine un fişier `file.txt`.

Fiecare fişier are asociat, aşadar, un nume cu ajutorul căruia se face identificarea, un set de drepturi de acces şi zone conţinând informaţia utilă.

Sistemele de fişiere suportate de sistemele de operare de tip Unix şi Windows sunt ierarhice. Sistemele Linux/Unix sunt case-sensitive (`Data` este diferit de `data`), iar sistemele Windows sunt case-insensitive.

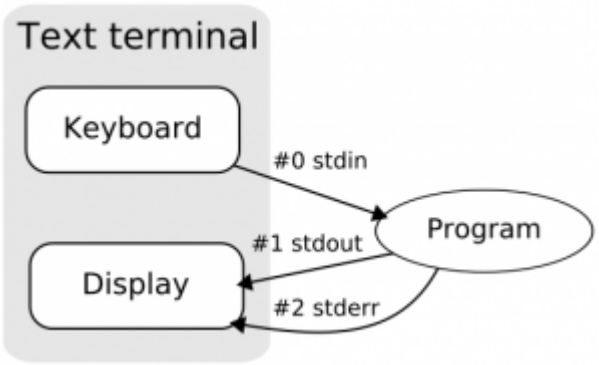
Ierarhia sistemului de fişiere Unix are un singur director cunoscut sub numele de `root` şi notat `'/'`, prin care se localizează orice fişier (a nu se confunda cu directorul `/root`, care este home-ul utilizatorului privilegiat, root). Notaţia Unix pentru căile fişierelor este un şir de nume de directoare despărţite prin `'/'`, urmat de numele fişierului. Există şi căi relative la directorul curent `'.'` sau la directorul părinte `'..'`.

În Unix nu se face nicio deosebire între fişierele aflate pe partiţiile discului local, pe CD sau pe o maşină din reţea. Toate aceste fişiere vor face parte din ierarhia unică a directorului `root`. Acest lucru se realizează prin `montare`: sistemele de fişiere vor fi montate într-unul din directoarele sistemului de fişiere rădăcină.

În Windows există mai multe ierarhii, câte una pentru fiecare partiţie şi pentru fiecare loc din reţea. Spre deosebire de Unix, delimitatorul între numele directoarelor dintr-o cale este `'\'`, şi pentru căile absolute trebuie specificat numele ierarhiei în forma `C:\`, `E:\` sau `\\FILESERVER\myFile` (pentru reţea). Ca şi Unix, Windows foloseşte `'.'` pentru directorul curent şi `'..'` pentru directorul părinte.

Operaţii pe fişiere

În Unix, un **descriptor de fişier** este un întreg care indexează o tabelă cu pointeri spre structuri care descriu fişierele deschise de un proces. În cazul în care un program rulează într-un shell Unix, procesul părinte (shell-ul) deschide pentru procesul copil (programul respectiv) 3 fişiere standard având descriptori de fişiere cu valori speciale:



- **standard input** (0) - citirea de la intrarea standard (tastatură)
- **standard output** (1) - afişarea la ieşirea standard (consolă)
- **standard error** (2) - afişarea la ieşirea standard de eroare (consolă)

Pe Windows, noţiunea de bază pentru managementul fişierelor este **handle**-ul, o valoare din care se obţine un pointer spre o structură descriptivă a fişierului. Aceleaşi 3 fişiere standard sunt deschise de fiecare proces.

În continuare, pentru descrierea comportamentului operaţiilor de intrare-ieşire 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 şi Linux), dar acestea se implementează în Windows prin apelurile Win32 şi formează un nivel de abstractizare aflat mai departe de kernel.

Un fişier are asociat cursorul de fişier (file pointer) care indică poziţia curentă în cadrul fişierului. Cursorul de fişier este un întreg care reprezintă deplasamentul (offset-ul) faţă de începutul fişierului.

Operaţiile specifice pentru lucru cu fişiere:

- **deschiderea/crearea unui fişier** - înseamnă asocierea unui descriptor de fişier sau a unui handle cu un fişier identificat prin numele său [1]. (Linux, Windows)
- **închiderea unui fişier** - înseamnă eliberarea structurilor de fişier asociate procesului şi a descriptorului (handle-ului) acelui fişier [2]. (Linux, Windows)
- **citirea dintr-un fişier** - înseamnă copierea unui bloc de date într-un buffer; după ce se realizează citirea se actualizează cursorul de fişier [3]. (Linux, Windows)
- **scrierea într-un fişier** - înseamnă copierea unui bloc de date dintr-un buffer în fişier; efectuarea scrierii înseamnă şi actualizarea cursorului de fişier [4]. (Linux, Windows)
- **poziţionarea într-un fişier** - înseamnă schimbarea valorii cursorului de fişier; citirile sau scrierile ulterioare vor porni din locul indicat de acest cursor de fişier [5]. (Linux, Windows)
- **schimbarea atributelor unui fişier** - înseamnă stabilirea unor parametri pentru fişier; apelurile [6]. (Linux)

Operaţii pe fişiere în Linux

Crearea, deschiderea şi închiderea fişierelor

open

Pentru deschiderea/crearea unui fişier se foloseşte funcţia open.

```
int open(const char *pathname, int flags);              /* deschidere */
int open(const char *pathname, int flags, mode_t mode);  /* creare */
```

creat

Pentru crearea de fişiere se poate utiliza şi creat:

```
int creat(const char *pathname, mode_t mode);
```

Funcţia este echivalentă cu apelul `open` unde flag-ul O_CREAT e setat şi fişierul nu exista deja:

```
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

close

Închiderea de fişiere se realizează cu close:

```
int close(int fd)
```

O greşeală 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`.

unlink

Ştergerea efectivă a unui fişier de pe disk se realizează cu funcţia unlink:

```
int unlink(const char *pathname);
```

Exemplu

Dacă, spre exemplu, dorim să deschidem fişierul `in.txt` pentru citire şi scriere, cu eventuala creare a acestuia, iar fişierul `out.txt` pentru scriere, cu trunchiere putem folosi următoarea secvenţă de cod:

io-01.c

```
#include <sys/types.h>  /* open */
#include <sys/stat.h>   /* open */
#include <fcntl.h>      /* O_RDWR, O_CREAT, O_TRUNC, O_WRONLY */
#include <unistd.h>     /* close */

#include "utils.h"

int main(void)
{
        int fd1, fd2;

        fd1 = open ("in.txt", O_RDWR | O_CREAT, 0644);
        DIE(fd1 < 0, "open in.txt");

        /* will fail if out.txt does not exist */
        fd2 = open ("out.txt", O_WRONLY | O_TRUNC);
        DIE(fd2 < 0, "open out.txt");

        rc = close(fd1);
        DIE(rc < 0, "close fd1");

        rc = close(fd2);
        DIE(rc < 0, "close fd2");

        return 0;
}
```

Atenţie! O greşeală frecventă este omiterea drepturilor de creare a fişierului (0644 în exemplul de mai sus) când se apelează open cu flag-ul O_CREAT setat.

Scrierea şi citirea

read

Funcţia read e folosită pentru citirea din fişier a maxim `count` octeţi:

```
ssize_t read(int fd, void *buf, size_t count);
```

Funcţia read întoarce numărul de octeţi efectiv citiţi, cel mult `count`. Valoarea minimă este de `1` octet, iar când se ajunge la sfârşitul de fişier se va întoarce `0`.

write

Funcţia write e folosită pentru scrierea în fişier a maxim `count` octeţi:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Valoarea întoarsă este numărul de octeţi ce au fost efectiv scrişi, cel mult `count`. În mod implicit nu se garantează că la

revenirea din write scrierea în fişier s-a terminat. Pentru a forţa actualizarea se poate folosi fsync sau fişierul se poate deschide folosind flagul `O_FSYNC`, caz în care se garanteaza că după fiecare write fişierul a fost actualizat.

Observaţie:
Pentru read/write există versiunile pread/pwrite, care permit specificarea unui offset în fişier de la care să se efectueaze operaţia de citire/scriere. (De asemenea, există şi versiunile `pread64`/`pwrite64` care folosesc offset-uri de 64 de biţi - pentru a putea specifica offset-uri mai mari decât 4GB).

Poziţionarea în fişier (lseek)

lseek

Funcţia lseek permite mutarea cursorului unui fişier la o poziţie absolută sau relativă.

```
off_t lseek(int fd, off_t offset, int whence)
```

Parametrul `whence` reprezintă poziţia relativă de la care se face deplasarea:

- `SEEK_SET` - faţă de poziţia de început
- `SEEK_CUR` - faţă de poziţia curentă
- `SEEK_END` - faţă de poziţia de sfârşit

Observaţie lseek permite şi poziţionări după sfârşitul fişierului. Scrierile care se fac în astfel de zone nu se pierd, ceea ce se obţine fiind un fişier cu *goluri*, o zonă care este *sărită* - nu este alocată pe disc.

Pentru această funcţie există şi o versiune lseek64 la care offset-ul este pe 64 de biţi.

Trunchierea fişierelor

Pe lângă trunchierea la 0 care se poate face prin apelul `open` cu flag-ul `O_TRUNC`, se poate specifica trunchierea unui fişier la o dimensiune specificată, prin apelurile de sistem ftruncate şi truncate:

```
int ftruncate(int fd, off_t length);
int truncate(const char *path, off_t length);
```

În cazul ftruncate, parametrul `fd` este file descriptorul obţinut un apel open care a asigurat drept de scriere. În cazul truncate, fişierul reprezentat prin `path` trebuie să aibă drept de scriere.

Exemplu utilizare operaţii I/O

io-2.c

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

#include <sys/types.h>  /* open */
#include <sys/stat.h>   /* open */
#include <fcntl.h>      /* O_CREAT, O_RDONLY */
#include <unistd.h>     /* close, lseek, read, write */

#include "utils.h"

/* Print the last 100 bytes from a file */

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

        /* alocate space for the read buffer */
        buf = malloc(101);
        DIE(buf == NULL, "malloc");

        /* open file */
        fd = open("file.txt", O_RDONLY);
        DIE(fd < 0, "open");
```

```
                /* set file pointer at 100 characters
                 _before_ the end of the file */
                rc = lseek(fd, -100, SEEK_END);
                DIE(rc < 0, "lseek");

                /* read the last 100 characthers */
                bytes_read = read(fd, buf, 100);
                DIE(bytes_read < 0, "read");

                /* set '
        int dup(int oldfd);
        int dup2(int oldfd, int newfd);
        ' at end of buffer for printing purspouses*/
        buf[bytes_read] = '
        fd = open("output.txt", O_RDWR|O_CREAT|O_TRUNC, 0600);
        dup2 (fd, STDOUT_FILENO);
        ';

        <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("the last %ld bytes:
        \n%s\n", bytes_read, buf);

        /* close file */
        rc = close(fd);
        DIE(rc < 0, "close");

        /* cleanup */
        free(buf);

        return 0;
        }
```

Redirectări

În Linux redirectările se realizează cu ajutorul funcţiilor de duplicare a file descriptorilor dup şi dup2:

```
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

De exemplu, pentru redirectarea ieşirii în fişierul `output.txt`, sunt necesare două linii de cod:

```
HANDLE CreateFile(
   LPCTSTR lpFileName,
   DWORD dwDesiredAccess,
   DWORD dwShareMode,
   LPSECURITY_ATTRIBUTES lpSecAttributes,
   DWORD dwCreationDisposition,
   DWORD dwFlagsAndAttributes,
   HANDLE hTemplateFile
);
```

Operaţii speciale

Funcţia fcntl permite efectuarea unor operaţii speciale asupra descriptorilor de fişier.

Operaţii pe fişiere în Windows

Crearea, deschiderea şi închiderea

CreateFile

Pentru a crea un handle asociat cu un fişier, director sau altă resursă abstractizată sub forma unui fişier (port COM, pipe, modem etc.) se foloseşte funcţia CreateFile. Funcţia se ocupă atât de crearea, cât şi de deschiderea unui fişier (şi întoarce în ambele cazuri un handle asociat cu fişierul):

`BOOL CloseHandle(HANDLE hObject);`	`CloseHandle(hFile);` `DeleteFile("myfile.txt");`

Atenţie! Explicaţiile complete se găsesc pe pagina de manual pentru CreateFile. În continuare vom prezenta cele mai importante proprietăţi.

Drepturile de acces cerute la deschiderea fişierului sunt specificate în `dwDesiredAccess`:

- `GENERIC_WRITE`
- `GENERIC_READ`

Lista completa aici

Parametrul `dwCreationDisposition` precizează modul în care apelul acţionează în cazul în care fişierul există sau nu; poate avea valori de forma:

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

Dacă fişierul există deja şi `dwCreationDisposition` este `CREATE_ALWAYS` sau `OPEN_ALWAYS`, apelul NU eşuează, dar `GetLastError` returnează `ERROR_ALREADY_EXISTS`.

Mai multe aici

Pentru copierea şi mutarea fişierelor există apelurile CopyFile, MoveFile şi ReplaceFile.
Un exemplu de schimbare a atributelor găsiţi aici.

CloseHandle

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

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

DeleteFile

Ştergerea se face prin închiderea fişierului şi folosirea apelului de sistem DeleteFile

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

unde DeleteFile are signatura

```
bRet = ReadFile(
   hFile,        /* open file handle */
   lpBuffer,     /* where to put data */
   dwBytesToRead,/* number of bytes to read */
```

```
   &dwBytesRead, /* number of bytes that were read */
   NULL          /* no overlapped structure */
);
```

Citirea şi scrierea

ReadFile

ReadFile operează asupra unui fişier care are drepturi de acces cel puţin pentru citire, copiind un număr de octeţi (începând cu poziţia curentă a cursorului de fişier) într-un buffer şi întoarce într-o variabilă numărul de octeţi citiţi.

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

```
bRet = WriteFile(
   hFile,          /* open file handle */
   lpBuffer,       /* start of data to write */
   dwBytesToWrite, /* number of bytes to write */
   &dwBytesWritten,/* number of bytes that were written */
   NULL            /* no overlapped structure */
);
```

ReadFile primeşte un handle de fişier `hFile`, creat anterior cu drepturi cel puţin de citire. Rezultatul citirii este copiat în `lpBuffer`, iar numărul de octeţi efectiv citiţi este întors în variabila pointată de `lpNumberOfBytesRead`. Numărul de octeţi efectiv citiţi poate fi mai mic decât numărul de octeţi care se doresc a fi citiţi - `nNumberOfBytesToRead`.

În mod normal, după acest apel, cursorul de fişier este actualizat cu numărul de octeţi citiţi. Singura excepţie este cazul în care fişierul este deschis pentru operaţii de I/O de tip `OVERLAPPED` - asincrone, caz în care conceptul de cursor de fişier nu mai este folositor (şi deci nu mai este actualizat). Mai multe detalii despre operaţiile asincrone în Laborator 10 - Operatii IO avansate - Windows.

ReadFile returnează o valoare diferită de zero în caz de succes, şi zero altfel. Dacă se returnează o valoare diferită de zero, dar numărul de octeţi citiţi este zero, atunci s-a ajuns la sfârşitul de fişier.

WriteFile

Apelul WriteFile copiază în mod sincron sau asincron un număr specificat de octeţi dintr-un buffer în conţinutul unui fişier şi returnează într-o variabilă numărul efectiv de octeţi copiaţi. Scrierea în fişier se face în general începând din poziţia curentă a cursorului şi după terminarea operaţiei, poziţia cursorului fişierului este actualizată (rămân valabile observaţiile anterioare despre operaţii `OVERLAPPED`).

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

```
/* Example: How to get current position */
currentPos = SetFilePointer(
               myFileHandle,
               0,           /* offset 0 */
               NULL,        /* no 64bytes offset */
               FILE_CURRENT
);
```

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

Poziţionarea în fişier

SetFilePointer

Fiecare fişier deschis are asociat un cursor (memorat pe 64 de biţi) care reprezintă poziţia curentă de citire/scriere. Un proces poziţionează cursorul la un offset specificat cu SetFilePointer:

```
BOOL SetEndOfFile(HANDLE hFile);
```

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

#include "utils.h"

#define BUF_SIZE        100

int main (void)
{
        HANDLE hFile;
        DWORD dwBytesRead, dwPos, dwBytesToRead = BUF_SIZE, dwRet;
        BOOL bRet;
        CHAR outBuffer[BUF_SIZE+1];

        /* deschidem fisierul */
        hFile = CreateFile(
                        "file.txt",
                        GENERIC_READ,
                        FILE_SHARE_READ,
                        NULL,        /* no security attributes */
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL        /* no pattern */
                );
        DIE(hFile == INVALID_HANDLE_VALUE, "CreateFile");

        /* set file pointer at 100 bytes
        _before_ the end of file */
        dwPos = SetFilePointer(
                        hFile,
                        -100,
                        NULL,        /* used only for offsets on 64bytes */
                        FILE_END
        );
        DIE(dwPos == INVALID_SET_FILE_POINTER, "SetFilePointer");

        /* read last 100 bytes into buffer */
        dwRet = ReadFile(
                        hFile,
                        outBuffer,
                        dwBytesToRead,
                        &dwBytesRead,
                        NULL);        /* do nothing asyncronious */
        DIE(dwRet == FALSE, "ReadFile");

        /* print buffer */
        outBuffer[dwBytesRead] = '
#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);
';
        <a
href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("last
%ld bytes: \n%s\n", dwBytesRead, outBuffer);
        fflush(stdout);

        /* close file */
        bRet = CloseHandle (hFile);
        DIE(bRet == FALSE, "CloseHandle");

        return 0;
}
```

Deplasarea se face asupra unui fişier reprezentat prin handle-ul `hFile` deschis în prealabil, creat cu unul din drepturile de acces `GENERIC_READ` sau `GENERIC_WRITE`. O valoare pozitivă înseamnă o deplasare înainte, iar una negativă, înapoi.

Numărul de octeţi cu care se mută cursorul este specificat de `lDistanceToMove` [in] şi `lpDistanceToMoveHigh`; cele două câmpuri de 32 de biţi formează o valoare de 64 de biţi. Uzual cel de-al doilea câmp este NULL.

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

- `FILE_BEGIN` - punctul de start este începutul fişierului; `lDistanceToMove` este considerat unsigned
- `FILE_CURRENT` - punctul de start este valoarea curentă a cursorului
- `FILE_END` - punctul de start este valoarea curentă a sfârşitului de fişier

Apelul returnează noua valoare a cursorului, dacă `lpDistanceToMoveHigh` este NULL; altfel, se returnează jumătatea low a valorii, jumătatea high luând locul lpDistanceToMoveHigh.

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 uşor.

Trunchierea fişierelor

SetEndOfFile

Un fişier poate fi trunchiat sau extins folosind apelul SetEndOfFile, care face poziţia sfârşitului de fişier EOF egală cu poziţia curentă a cursorului fişierului. În cazul extinderii fişierului peste limita sa, conţinutul adăugat este nedefinit.

```
#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
```

Exemplu

win_io.c

```
        watch -d lsof -p $(pidof redirect)
```

Wrapper-e

Exerciţii

În rezolvarea laboratorului folosiţi arhiva de sarcini lab02-tasks.zip

Observaţii: Pentru a vă ajuta la implementarea exerciţiilor din laborator, în directorul `utils` din arhivă există un fişier `utils.h` cu funcţii utile.

Folosiţi `man`/`MSDN` pentru informaţii despre apelurile de sistem

Verificaţi valorile de retur a apelurilor de sistem
Puteţi folosi macro-ul DIE(valoare_retur == eroare, "mesaj eroare");

LINUX

1. (**1 punct**) Intraţi în directorul `1-redirect`
   - Urmăriţi conţinutul fişierului `redirect.c`.
   - Compilaţi fişierul (folosiţi make).
   - Deschideţi alt terminal şi rulaţi comanda:
     - ./set_nasty.sh
   - lsof este un utilitar care afişează informaţii despre fişierele deschise (ce fişiere sunt deschise în sistem, ce fişiere a deschis un anumit user etc)
     - Ce semnificaţie are coloana FD ? Dar coloana TYPE ? (man 8 lsof, cautare după FD şi TYPE)
   - Folosiţi comanda ENTER pentru a continua programul. În paralel urmăriţi cum se modifică tabela de file-descriptori.
   - Observaţi în cod parametrii cu care s-a realizat redirectarea cu dup2. Ce se întampla dacă erau în ordine inversă ? (dup2(fd2, STDERR_FILENO))
     - Hint: Revedeţi secţiunea de redirectări
2. (**1 punct**) Intraţi în directorul `2-lseek`
   - Citiţi sursa `lseek.c`.
     - Ce valoare va întoarce al doilea apel al funcţiei `lseek` ? De ce ?
   - Decomentaţi linia de afişare, compilaţi şi rulaţi pentru verificare.
   - Sursa închide doar file descriptorul `fd1`. Este nevoie să se închidă şi file descriptorul `fd2` ? De ce?
3. (**4 puncte**)
   - Intraţi în directorul `3-mcat`.
   1. (**1 punct**) Completaţi fişierul astfel încat programul rezultat `mcat` să aibă funcţionalitate similară cu a utilitarului `cat` (urmăriţi comentariile cu TODO 1)
      - Programul `mcat` va primi ca argument în linia de comandă numele unui fişier al cărui conţinut îl va afişa la ieşirea standard.
      - Nu aveţi voie să citiţi tot fişierul în memorie. Puteţi citi doar bucăţi de dimensiune maximum BUFSIZE.
      - **Verificaţi codul de eroare** întors de apelurile de sistem. Puteţi folosi macro-ul DIE.
      - Revedeţi secţiunile Crearea, deschiderea şi închiderea fişierelor şi Scrierea şi citirea fişierelor.
      - Testaţi cu o comandă de genul: ./mcat /dev/nasty
        ./mcat /dev/nasty out ; ./mcat out
   2. (**1 punct**) Extindeţi funcţionalitatea astfel încât output-ul să fie redirectat într-un fişier primit ca al doilea argument - funcţionalitate similară cu a utilitarului `cp`. (urmăriţi comentariile cu TODO 2)
      - Revedeţi secţiunea de redirectări.
      - Testaţi funcţionalitatea: ./mcat Makefile /dev/nasty ; cat /dev/nasty
   3. (**2 puncte**) Iniţializaţi fişierul `/dev/nasty`:./singular & sleep 3 ; ./singular
   - Încercaţi funcţionalitatea de copiere pe fişierul `/dev/nasty`: ls.exe ..
   - De unde apare diferenţa ?
   - Hints:
     - Ce întorc funcţiile read şi write? (eventual afişaţi aceste valori)
   - Testaţi **scrierea** cu: ./mcat Makefile /dev/nasty ; cat /dev/nasty

WINDOWS

Executabilele sunt generate în directorul `win/Debug` (în directorul `Debug` al soluţiei, nu al fiecărui proiect în parte).

1. (**0.5 punct**) cat

- Deschideţi soluţia laboratorului 2 din folderul `win`
- Intraţi în proiectul `1-cat` şi urmăriţi sursa `cat.c`
- Compilaţi şi testaţi executabilul `cat.exe`
- Hint:
  - Porniţi command prompt-ul de Visual Studio: Tools Visual Studio Command Prompt

2. (**3.5 puncte**) CRC
   - Exerciţiul are ca scop realizarea unui utilitar care, pentru un fişier dat, calculeaza CRC-ul pentru fiecare bucată de 512 bytes din fişier şi o salvează într-un fişier de output.
   1. (**1.5 puncte**) Deschideţi fişierul `crc.c` din proiectul `2-crc` şi completaţi funcţia `GenerateCrc`
      - Funcţia primeşte ca prim argument fişierul pentru care trebuie calculat CRC-ul, iar ca al doilea argument fişierul în care se salvează CRC-urile pentru fiecare bucată de câte 512 bytes .
        - La ultima bucată se va face padding.
      - Hint:
        - Revedeţi secţiunile Crearea, deschiderea şi închiderea fişierelor, cât şi Citirea şi scrierea fişierelor.
        - Urmăriţi comentariile cu `TODO 1`
   2. (**2 puncte**) - Odată calculat fişierul cu CRC, vrem să vedem dacă două fişiere de CRC sunt egale.
      - Extindeţi funcţionalitatea programului anterior astfel încât să compare 2 fişiere - completaţi funcţia `CompareFiles`.
      - Iniţial comparaţi dimensiunile fişierelor.
      - Hint:
        - Completaţi funcţia `GetSize` pentru calcuarea dimensiunii unui fişier
        - Folosiţi doar funcţia `SetFilePointer`
        - Urmăriţi comentariile cu `TODO - 2`
      - Dacă dimensiunile sunt egale, comparaţi cele 2 fişiere bucată cu bucată (nu citiţi tot fişierul în memorie)
      - Hint:
        - Urmăriţi comentariile cu `TODO - 3`

BONUS LINUX

1. (**1 so karma**) Troubleshooting
   - Intraţi în directorul `4-trouble`
   - Compilaţi şi rulaţi programul `trouble`
   - Programul ar trebui să afişeze în fişierul `tmp1.txt` mesajul din `msg`
   - Afişaţi fişierul `tmp1.txt`
   - Identificaţi şi remediaţi problema
   - Hint:
     - Revedeţi secţiunea: Crearea, deschiderea şi închiderea fişierelor
2. (**1 so karma**) File lock
   - Vrem să ne asigurăm că doar **o instanţă** a unui program rulează la un moment dat.
   - Pentru asta se creează un fişier temporar pe care se încearcă obţinerea unui lock folosind apelul flock
   - În acest scop intraţi în directorul `5-singular` şi completaţi sursa `singular.c` (urmăriţi comentariile cu TODO)
   - Testaţi rulând executabilul din două terminale diferite, sau cu comanda: ./singular & sleep 3 ; ./singular
   - Cum ne mai putem asigura că programul nostru are doar o singura instanţă, folosind mai **puţine** apeluri de sistem?

BONUS WINDOWS

1. Utilitar echivalent cu `ls -a -R`
   1. (**1 so karma**) Creare utilitar `ls`
      - Deschideţi din soluţia laboratorului 2 proiectul `3-ls`
      - Completaţi fişierul `ls.c` pentru ca programul `3-ls.exe` să se comporte ca utilitarul `ls`.
      - Afişarea fişierelor dintr-un director se face în doi paşi:
        - se obţine un handle la o primă intrare din lista de fişiere a directorului cu funcţia: FindFirstFile
        - se iterează această listă folosind funcţia: FindNextFile
      - Hint:
        - Urmăriţi comentariile marcate cu `TODO 1`
        - Pentru testare folosiţi dintr-un prompt Visual Studio: ls.exe ..
   2. (**1 so karma**) Afişare detalii pentru parametrul `-a`
      - Pentru fişiere afişaţi numele, dimensiunea şi data la care au fost modificate ultima oară
      - Pentru directoare afişaţi numele şi un indicator de director (ex: nume)
      - Hint:
        - Atributele unui fişier sunt definite într-o structură de forma: WIN32_FIND_DATA

    - Pentru a verifica dacă un fişier e director, trebuie să aibă bitul "FILE_ATTRIBUTE_DIRECTORY" din câmpul "dwFileAttributes" (vezi File Attributes)
    - Urmăriţi comentriile marcate cu `TODO 2`
3. (**1 so karma**) Afişare detalii pentru parametrul `-R`
    - Realizaţi parcurgerea recursivă a directoarelor prin apelarea recursivă a funcţiei `ListFile`
    - Hint:
        - Urmăriţi comentariile marcate cu `TODO 3`
        - Aveţi grijă să concatenaţi numele noului director la calea deja existentă
2. (**1 so karma**) Troubleshooting
    - Deschideţi din soluţia laboratorului 2 proiectul `4-trouble`
    - Programul ar trebui să creeze un fişier cu mesajul "Testing 123"
    - Compilaţi şi rulaţi programul `trouble`
    - Identificaţi şi remediaţi problema
    - Hint:
        - Revedeţi secţiunea: Crearea, deschiderea şi închiderea fişierelor

EXTRA

Operaţii cu fişiere în Python
Studiaţi exemplele din arhivă, citiţi documentaţia şi observaţi diferenţele între API-uri

Soluţii

lab02-sol.zip

Resurse utile

1. Low level I/O (info libc "Low-Level I/O")
2. Duplicating descriptors (info libc "Duplicating Descriptors")
3. Low level I/O (Advanced Linux Programming)
4. File management functions

[1] `fopen` (ISO C), `open`, `creat` (POSIX), `CreateFile` (Win32 API)

[2] `fclose` (ISO C), `close` (POSIX), `CloseFile` (Win32 API)

[3] `fread` (ISO C), `read` (POSIX), `ReadFile` (Win32 API)

[4] `fwrite` (ISO C), `write` (POSIX), `WriteFile` (Win32 API)

[5] `fseek` (ISO C), `lseek` (POSIX), `SetFilePointer` (Win32 API)

[6] `fcntl` (POSIX), SetFileAttributes (Win32 API)