LOW LEVEL FILE PROCESSING

1. Overview

The learning objectives of this lab session are:

- To understand the functions provided for file processing by the lower level of the file management system, i.e. the functions which directly invoke the operating system.
- To acquire hands-on experience in using the low level IO in C

The example presented illustrates the main operations on a file: create, append, update, read.

2. Brief theory reminder

A **file** is an ordered collection of records, stored on external media. The main operations of files are:

- open / close an existing file;
- create a file;
- read / write a file;
- set current position in a file (file positioning);
- delete a file.

File processing can be done in two ways:

a) Using the low level functions provided by the file management system;
b) Using the high level functions provided by the file management system;

A file can be accessed in two modes:

- sequential;
- random.

2.1. File creation

To create a file, you can use the function **creat**, This function is obsolete. The call:

creat (filename, mode)

is equivalent to:

open (filename, O_WRONLY | O_CREAT | O_TRUNC, mode)

If on a 32 bit machine the sources are translated with _FILE_OFFSET_BITS == 64 the function **creat** returns a file descriptor opened in the large file mode which enables the file handling functions to use files up to 2^63 in size and offset from -2^63 to 2^63. This happens transparently for the user since all of the low level file handling functions are equally replaced. The function **creat64** is also obsolete:

int creat64 (const char *filename, mode_t mode)

This function is similar to **creat**. It returns a file descriptor which can be used to access the file named by filename. The only the difference is that on 32 bit systems the file is opened in the large file mode. I.e., file length and file offsets can exceed 31 bits. In these prototypes:

- **filename** - is a pointer to a character string which defines the path name of the file to be created.;
- **mode** - is an integer which can be specified by bitwise or-ing the following access rights:
  - S_IREAD - read right;
  - S_IWRITE - write right.

The function returns a file descriptor upon success, and -1 on error. To use the function, one must use the following directives:

#include <io.h>
#include <sys/stat.h>

Important note: opening an existing file by **creat** causes the loss of file contents.

The created file is exploited in mode text (O_TEXT) or binary (O_BINARY), depending on the value of the global variable **_fmode** (O_TEXT by default). This variable is also deprecated. We recommend the use of the **open** function.

2.2. Open (an existing) file

To open a file use the recommended **open** function. Its prototype is:

int open (const char *filename, int flags[, mode_t mode])

The open function creates and returns a new file descriptor for the file named by **filename**. Initially, the file position indicator for the file is at the beginning of the file. The argument mode is used only when a file is created, but it doesn't hurt to supply the argument in any case.

The **flags** argument controls how the file is to be opened. This is a **bit mask**; you create the value by the bitwise OR of the appropriate parameters (using the | operator in C. File status flags fall into three categories, which are described in the following sections.

- **Access Modes**, specify what type of access is allowed to the file: reading, writing, or both. They are set by open and are returned by **fcntl**, but cannot be changed. Access modes are:
  - **O_RDONLY**: Open the file for read access.
  - **O_WRONLY:** Open the file for write access.
  - **O_RDWR:** Open the file for both reading and writing.
- **Open-time Flags**, control details of what open will do. These flags are not preserved after the open call. They are:
  - **O_CREAT:** If set, the file will be created if it doesn't already exist.
  - **O_EXCL:** If both O_CREAT and O_EXCL are set, then open fails if the specified file already exists. This is guaranteed to never clobber an existing file.
  - **O_NONBLOCK:** This prevents open from blocking for a "long time" to open the file. This is only meaningful for some kinds of files, usually devices such as serial ports; when it is not meaningful, it is harmless and ignored. Often opening a port to a modem blocks until the modem reports carrier detection; if O_NONBLOCK is specified, open will return immediately without a carrier. **Note** that the O_NONBLOCK flag is overloaded as both an I/O operating mode and a file name translation flag. This means that specifying O_NONBLOCK in open also sets non-blocking I/O mode; see Operating Modes. To open the file without blocking but do normal I/O that blocks, you must call open with O_NONBLOCK set and then call **fcntl** to turn the bit off.
  - **O_TRUNC:** Truncate the file to zero length. This option is only useful for regular files, not special files such as directories or FIFOs. POSIX.1 requires that you open the file for writing to use O_TRUNC.

- **Operating Modes**, affect how operations such as read and write are done. They are set by open, and can be fetched or changed with **fcntl**. The mode flags are:
    - **O_APPEND:** The bit that enables append mode for the file. If set, then all write operations write the data at the end of the file, extending it, regardless of the current file position. This is the only reliable way to append to a file. In append mode, you are guaranteed that the data you write will always go to the current end of the file, regardless of other processes writing to the file. Conversely, if you simply set the file position to the end of file and write, then another process can extend the file after you set the file position but before you write, resulting in your data appearing someplace before the real end of file.
    - **O_NONBLOCK:** The bit that enables non-blocking mode for the file. If this bit is set, read requests on the file can return immediately with a failure status if there is no input immediately available, instead of blocking. Likewise, write requests can also return immediately with a failure status if the output can't be written immediately. Note that the O_NONBLOCK flag is overloaded as both an I/O operating mode and a file name translation flag.

The normal return value from open is a non-negative integer file descriptor. In the case of an error, a value of -1 is returned instead. In addition to the usual file name errors, the following **errno** error conditions are defined for this function:

- **EACCES:** The file exists but is not readable/writable as requested by the flags argument, the file does not exist and the directory is not writable so it cannot be created.
- **EEXIST:** Both O_CREAT and O_EXCL are set, and the named file already exists.
- **EINTR:** The open operation was interrupted by a signal.
- **EISDIR:** The flags argument specified write access, and the file is a directory.
- **EMFILE:** The process has too many files open.
- **ENFILE:** The entire system, or perhaps the file system which contains the directory, cannot support any additional open files at the moment. (This problem cannot happen on the GNU system.)
- **ENOENT:** The named file does not exist, and O_CREAT is not specified.
- **ENOSPC:** The directory or file system that would contain the new file cannot be extended, because there is no disk space left.
- **ENXIO:** O_NONBLOCK and O_WRONLY are both set in the flags argument, the file named by filename is a FIFO, and no process has the file open for reading.
- **EROFS: T**he file resides on a read-only file system and any of O_WRONLY, O_RDWR, and O_TRUNC are set in the flags argument, or O_CREAT is set and the file does not already exist.

If on a 32 bit machine the sources are translated with _FILE_OFFSET_BITS == 64 the function open returns a file descriptor opened in the large file mode which enables the file handling functions to use files up to 2^63 bytes in size and offset from -2^63 to 2^63. This happens transparently for the user since all of the low level file handling functions are equally replaced.

The header files you need to include are **fcntl.h** and **io.h.** The function **open** returns a file descriptor if successful, and a -1 on error.
Example:

df=open("MyFile.dat", O_RDWR);

Note. The number of opened files is usually limited at any given moment.

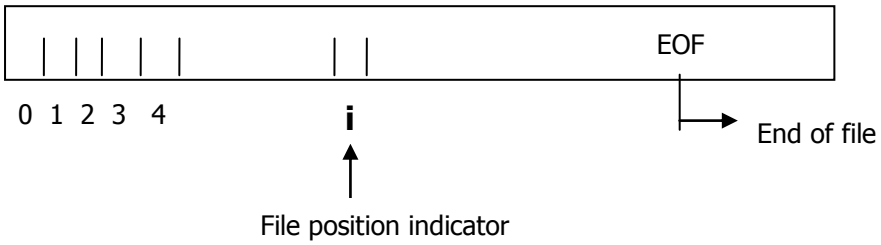
2.3. File read

To read from an existing file, previously opened using **open use the function read.** This has the following prototype:

ssize_t read (int filedes, void *buffer, size_t size);

where:
ssize_t is defined as long (see include\sys\types.h)
filedes - is a file descriptor as returned by **open**;
buffer - is a pointer to the buffer memory area where the read record is stored;
size - is the length in bytes of the record read.

A file is viewed as a sequence of bytes starting with byte numbered zero After each read operation the file position indicator is advanced to indicate the byte for next read.



When a file is opened with the flags O_RDONLY or O_RDWR, the file position indicator is positioned over byte 0 (beginning of file).

The return value is the number of bytes actually read. This might be less than **size**; for example, if there aren't that many bytes left in the file or if there aren't that many bytes immediately available. The exact behavior depends on what kind of file it is. Note that reading less than **size** bytes is not an error. A value of zero indicates end-of-file (except if the value of the size argument is also zero). This is not considered an error. If you keep calling read while at end-of-file, it will keep returning zero and doing nothing else.

If read returns at least one character, there is no way you can tell whether end-of-file was reached. But if you did reach the end, the next read will return zero.

In case of an error, **read** returns -1.

The standard input file (**stdin**) has a file descriptor zero.

To use **read** you must include the header file **io.h**.

Example. Read 20 bytes from the file "MyFile.dat" located in the current directory:

```
df=open ("MyFile.dat", O_RDONLY);
n=read (df, adr, 20);
```

2.4. File write

To write to a file opened using **open** (or **creat**) use the function **write**. Its prototype is:

ssize_t write (int filedes, const void *buffer, size_t size);

The **write** function writes up to **size** bytes from **buffer** to the file with descriptor **filedes**. The data in buffer is not necessarily a character string and a null character is output like any other character.

The return value is the number of bytes actually written or -1 in case of an error. This may be **size**, but can always be smaller. Your program should always call write in a loop, iterating until all the data is written.

Once write returns, the data is enqueued to be written and can be read back right away, but it is not necessarily written out to permanent storage immediately. You can use **fsync** when you need to be sure your data has been permanently stored before continuing. (It is more efficient for the system

to batch up consecutive writes and do them all at once when convenient. Normally they will always be written to disk within a minute or less.)

The standard input file (**stdout**) has a file descriptor 1.

To use **write** you must include the header file **io.h**.

Example: Write in the file "MyFile.dat" located in the current directory 20 bytes from address **adr**:

```
df=open ("MyFile.dat", O_WRONLY | O_CREAT);
n=write (df, adr, 20);
```

2.5. Positioning in a file

Whne reading or writing starting with a certain position in a file is needed, use **lseek.** Its prototype is:

```
off_t lseek (int filedes, off_t offset, int whence);
```

The **lseek** function is used to change the file position of the file with descriptor **filedes**.

The **whence** argument specifies how the offset should be interpreted, the same way as for the **fseek** function. It must be one of the symbolic constants **SEEK_SET, SEEK_CUR**, or **SEEK_END**. In detail:

- SEEK_SET specifies that whence is a count of characters from the beginning of the file.
- SEEK_CUR specifies that whence is a count of characters from the current file position. This count may be positive or negative.
- SEEK_END specifies that whence is a count of characters from the end of the file. A negative count specifies a position within the current extent of the file; a positive count specifies a position past the current end. If you set the position past the current end, and actually write data, you will extend the file with zeros up to that position.

The return value from **lseek** is normally the resulting file position, measured in bytes from the beginning of the file or -1 in case an error occurs. You can use this feature together with **SEEK_CUR** to read the current file position.

If you want to append to the file, setting the file position to the current end of file with **SEEK_END** is not sufficient. Another process may write more data after you seek but before you write, extending the file so the position you write onto clobbers their data. Instead, use the **O_APPEND** operating mode.

To use **lseek** you must include the header file **io.h.**

Examples:

a) to position the indicator at the beginning of the file:

```
lseek( filedes, 0l, SEEK_SET);
```

b) to position the indicator at the end of the file: :

```
lseek( filedes, 0l, SEEK_END);
```

2.6. Closing a file

When done with the processing of a file, the file should be closed using the function **close**.

```
int close(int filedes);
```

where **filedes** is the file descriptor.

The function returns zero upon success, and -1 if an error occurred. To use it, include **io.h.**

2.7. Example

The following program illustrates a number of operations on a file:

- File creation.
- Appending new records.
- Modification of records. (In this case the file position indicator is set to the beginning of a record and the new record is written.)
- File sort using the field **media** as a key. Because a large number of records were assumed to be stored in the file, first the records were read, and the record number and sort key were extracted from each record. This way ordering the keys can be achieved in the internal memory, as the data needed is less. Then the sorted file is obtained.

```
/* Program L10Ex1.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <io.h>
#include <ctype.h>
#include <string.h>
#include <sys/stat.h>
typedef struct
{
    char name[60];
    float media;
} StudentT;

typedef union
{
    StudentT stud;
    char st[sizeof(StudentT)];
} BufferT;

typedef struct
{
    int nb;
    float media;
} ElementT;

typedef enum {FALSE, TRUE} BooleanT;

void sort(const char filename[], const char sorted_filename[])
{
    ElementT el, t[100];
    char msgBuf[201];
    int i, j, n, fd1, fd2;
    BooleanT flag;
    BufferT stu;
    /* read file to array t */
    j = 0;
    fd1 = open(filename, O_RDONLY);
    while ( read(fd1, stu.st, sizeof(BufferT)) > 0 && j < 100)
    {
        t[j].nb = j;
        t[j].media = stu.stud.media;
        j++;
    }
```

```
    n = j;
    /* sort array t by media */
    j = 0;
    do
    {
        flag = TRUE;
        for (i = 1; i < n - j ; i++)
            if (t[i-1].media < t[i].media)
            {
                memcpy(&el, &t[i-1], sizeof(ElementT));
                memcpy(&t[i-1], &t[i], sizeof(ElementT));
                memcpy(&t[i], &el, sizeof(ElementT));
                flag = FALSE;
            }
    } while (flag == FALSE);
    /* write sorted output */
    if ((fd2 = open(sorted_filename, O_CREAT | O_WRONLY | O_TRUNC, S_IREAD |
S_IWRITE)) < 0)
    {
     sprintf(msgBuf, "\nCannot create %s. ", sorted_filename);
        perror(msgBuf);
        getchar();
        exit(1);
    }
    for (i = 0; i < n; i++)
    {
        lseek(fd1, (long)(t[i].nb *sizeof(BufferT)), SEEK_SET);
        read(fd1, stu.st, sizeof(BufferT));
        write(fd2, stu.st, sizeof(BufferT));
    }
    close(fd1); close(fd2);
}

void show_students(char filename[])
{
    BufferT stu;
    char s[41];
    int j, fd1;

    if ((fd1 = open(filename, O_RDONLY)) < 0)
    {
     sprintf(s, "Cannot open %s for reading. ", filename);
     perror(s);
     getchar();
     exit(3);
    }

     memset(&stu, '\0', sizeof(BufferT));
    j = 0;
    while ( read(fd1, stu.st, sizeof(BufferT)) > 0)
    {
        printf("\n%d. %-60s %7.2f", j++, stu.stud.name, stu.stud.media);
        memset(&stu, '\0', sizeof(BufferT));
    }
    close(fd1);
}
int main(int argc, char *argv[])
```

```
{
  unsigned int i, n;
    int fd1;
    long l;
    char ch;
    BufferT stu;
    float avgAux;
    char nameAux[sizeof(stu.stud.name)];

    char filename[] = "Group.dat";
    char sorted_filename[] = "SortedGroup.dat";

    printf("\nNumber of students in the group= "); scanf("%u%*c", &n);
    /* create group file */
    if ((fd1 = open(filename, O_CREAT | O_WRONLY | O_TRUNC, S_IREAD |
S_IWRITE)) < 0)
    {
        perror("\nCannot create Group.dat. ");
        getchar();
        exit(1);
    }
    /* student data input */
    for (i = 1; i <= n; i++)
    {
        printf("Name: ");
        scanf("%s", stu.stud.name);
        printf("media: "); scanf("%f%*c", &stu.stud.media);
        if (write(fd1, stu.st, sizeof(BufferT)) < sizeof(BufferT))
        {
          perror("Short write. ");
          getchar();
          exit(5);
        }
    }
    close(fd1);
    printf("\nGroup.dat contents after creation\n");
    show_students(filename);
    /* add new records to file */
    printf("\nNumber of students to add to group= "); scanf("%u%*c", &n);
    if ((fd1 = open(filename, O_RDWR)) < 0)
    {
        perror("\nCannot append to Group.dat. ");
        getchar();
        exit(1);
    }
    if (lseek(fd1, 0L, SEEK_END) < 0)
    {
      perror("\nlseek to end failed. ");
      getchar();
      exit(4);
    }
    /* student data input */
    for (i = 1; i <= n; i++)
    {
        printf("Name: ");
        scanf("%s", stu.stud.name);
        printf("media: "); scanf("%f%*c", &stu.stud.media);
```

```
        write(fd1, stu.st, sizeof(BufferT));
    }
    close(fd1);
    printf("\nGroup.dat contents after addition\n");
    show_students(filename);
    printf("\nUpdate student info [Yes=Y/y, No=other char]? ");
    ch=getchar();
    fd1 = open(filename, O_RDWR);
    while( toupper(ch) == 'Y')
    {
        printf("\nStudent number= "); scanf("%d%*c", &i);
        l = lseek(fd1, (long)(i * sizeof(BufferT)), SEEK_SET);
        printf("\toffset=%ld for i=%d\n", l, i);
        read(fd1, stu.st, sizeof(BufferT));
        printf("Current name is: %s\nNew name: ", stu.stud.name);
        scanf("%s", nameAux);
        if (*nameAux) strcpy(stu.stud.name, nameAux);
        printf("Current media is: %f\nNew media: ", stu.stud.media);
        scanf("%f", &avgAux);
        if (avgAux > 0) stu.stud.media = avgAux;
        l = lseek(fd1, (long)(i * sizeof(BufferT)), SEEK_SET);
        printf("\toffset=%ld for i=%d\n", l, i);
     write(fd1, stu.st, sizeof(BufferT));
        printf("\nUpdate more student info [Yes=Y/y, No=other char]? ");
        scanf("%*c%c", &ch);
    }
    close(fd1);
    printf("\nGroup.dat contents after changes\n");
    show_students(filename);
    getchar();
    sort(filename, sorted_filename);
    printf("\nGroup sorted by media\n");
    show_students(sorted_filename);
    getchar();
    return 0;
}
```

3. Lab Tasks

3.1. Execute and analyze the example program, L10Ex1.cpp

3.2. Read from the standard input a text, and write it to a text file, e.g. file "text.dat". Then display the file contents, prefixing each line with its order number.

3.3. Read the real part and imaginary part of ***n*** complex numbers. Then create a file containing the complex numbers your program read including the real part, the imaginary part, the modulus and argument for each number.

3.4. Two companies maintain information concerning their stocks (product code, name, amount, price per unit) in the files "stock1.dat" and "stock2.dat" respectively, ordered ascending by product code. The two companies merge, and there results a common stock which must be stored in the file "stock.dat", again ordered by the product code.

a) Create the original files input data from the keyboard, and then create the merged file "stock.dat". For products of the same code consider that the name and unit price are identical in both companies.

b) Walk sequentially through the file "stock.dat", and print for each product its name and amount.

c) For a component selected by the order number change the price per unit directly.

3.5. Write a program to concatenate two or more files containing real numbers. Print the information stored in the resulted file.

3.6. A simple message compression method is the following (run-length): any sequence of more than three identical characters is replaced by the following triple:
- A control character (marker);
- The repeated character;
- A natural number indicating the number of repetitions of that character. For example, the sequence **uuuuuu** becomes **#u6** (the control character was chosen to be **#**)

Note there are some issues here:
- The choice of the control character and its representation if it occurs in the message.
- The maximum number of repetitions which can be represented by a single triple.

3.7. Write a program to compress/decompress a file using functions for these operations. The name of the file to process must be passed as a parameter. Analyze the efficiency of compression by comparing the contents of the original with the compressed version.

3.8. Write a program to accomplish the following actions:
- Create a file with data concerning cars in a garage. A car is represented by a structure containing its brand name, owner name, color and number (from the number plate).
- Display on the screen an alphabetic list of all cars of a given color, ordered by owner's name.

3.9. Write a program which should read a C source text file and prints the occurrence frequencies for the reserved keywords in the given text.

3.10. Write a program which compares the contents of two existing text files an prints the lines which are different.