University Carlos III of Madrid

BSc in Applied Mathematics and Computing

Lab 1: System Calls

Member 1: Sabela Rubert Docampo
100523102
100523102@alumnos.uc3m.es

Member 2: Roberto Hogas Goras
100523139
100523139@alumnos.uc3m.es

Member 3: Alejandro Tarbay Guadalupe
100523065
100523065@alumnos.uc3m.es

Academic Year 2024/25 Group 121

With love, made in LaTeX for Catherine, our Operating Systems' lab teacher.

Click here to see our Overleaf code.

Contents

1 Description of the codes

1.1 crear.c

In this code we are asked to create a new file with a specific protection mode, which must be included as an argument in the call to **crear**. On the other hand, we are asked to transform the mode into octal and then create the file with this transformed mode. Finally, we must return an error message in several cases, such as if the correct number of arguments is not entered or if there was an error when creating the file.

To make the code, in Aula Global we downloaded a folder called "**Support code**" which contained the files we needed to compile the codes, as well as the start of `crear.c` and `combine.c` (with the libraries, the main, and the form of the structure used in `combine.c`).

In `crear.c`, there were included the most common libraries used by any program that needs to open, close and edit files. These are:

- **#include <stdio.h>**: used to print, read and display error messages with `pererror`
- **#include <stdlib.h>**: includes functions such as `malloc` and `free` (to manage memory) and `exit` (to terminate the program)
- **#include <fcntl.h>**: includes the flags we need to create files with the `open` function
- **#include <sys/stat.h>**: allows you to manipulate file attributes, including functions such as `chmod`, which changes or checks a program's permissions, and `umask`, which modifies the default permissions 'mask' on a file)
- **#include <unistd.h>**: includes POSIX system calls, such as `close`, `open` and `read`
- **#include <errno.h>**: defines the global variable `errno` to store system error codes and print them with `perror`
- **#include <string.h>**: includes functions for handling strings, such as `strlen` (length of a string) or `strcmp` (compares two strings)

Once the libraries have been included, we have started the main. First, we want to check that the call to the program (`./crear <file> <mode>`) is made with exactly three arguments:

- **./crear**: is the first argument (argv[0]), which represents the program executable.
- **<file>**: is the second argument (argv[1]), which indicates the name of the file to create.
- **<mode>**: is the third argument (argv[2]), which represents the permissions that the created file should have.

To do this, we have used a simple `if` where we compare whether the variable `argc`, the number of arguments with which the program is called, is equal to 3. Next, we create the variable `mode` of type `mode_t`, which is a data type specifically designed to represent the permissions (the mode) of files on POSIX systems[1]. Later, with the `sscanf` function we pass

[1]`https://pubs.opengroup.org/onlinepubs/7908799/xsh/sysstat.h.html`

the third argument (the mode) to octal format, and we check if it has been done correctly by comparing it with 1, since this function returns this value when a value has been successfully transformed[2].

On the other hand, as the statement asks us, the operating system may have defined a default mask, so we must save a copy and remove it at the beginning. We do this with the function `umask(0)`, which sets the creation mask to 0. This means that the created files will not have permission bits removed. In addition, we save this mask in a variable for later recovery.

Next, we create the variables `fd` and `ret` with a value of -1. The first will be used to create from scratch the file whose name is in `argv[1]`, and the second to verify that we close the file correctly. To create the file, we use the `open` method, which is used to open (or create) a file with a name, some flags and a specific mode. The name is the one in `argv[1]`, the mode corresponds to the value previously converted to octal, and the flags are the following ones (taken from the class exercises):

- `O_EXCL`: Ensure that a file is only created if it does not already exist. If it does, it loads an error in 'errno' variable
- `O_CREAT`: Creates the file if it does not exist.
- `O_TRUNC`: Truncates (empties) a file if it already exists.

After trying to open the file, we check if it has been opened by comparing the variable `fd` with -1: if it is still open as it was at the beginning, then we return an error.

Finally, we restore the saved mask, use the `chmod` function to set permissions to the created file (since, according to our teacher in the practical classes, in Guernika the `umask` function does not work as expected) and then we close the created file. To do this, we set `ret` to the output if the `close` function. If there is an error closing the file, an error will appear. Afterwards, we return 0 since we have created the file correctly to end the program.

1.2 combine.c

In this second code we were asked to combine data from the structure `students` saved in two files (the first two arguments) and save them with a **certain sort** in another file (the third argument). The structure students had a format that included the grade, the call and the name of each student. As a fundamental requirement, we had to count the number of students. If the number of these reached 100, we had to return an error and not put any students in the file specified in the third argument.

On the other hand, the program also asked us to save some statistics about these students in order to later classify and write them in a new file that we had to create called "`estadisticas.csv`". Finally, as in **crear.c**, the program must return an error if the number of arguments entered, which must be 4 if we include the call to **combine**, is not correct.

Regarding the libraries that come by default at the beginning of the `combine.c` file taken from Aula Global, there is only one library that was not previously explained in **crear.c**:

[2]`https://stackoverflow.com/questions/52310288/sscanf-returning-1-reading-a-string`

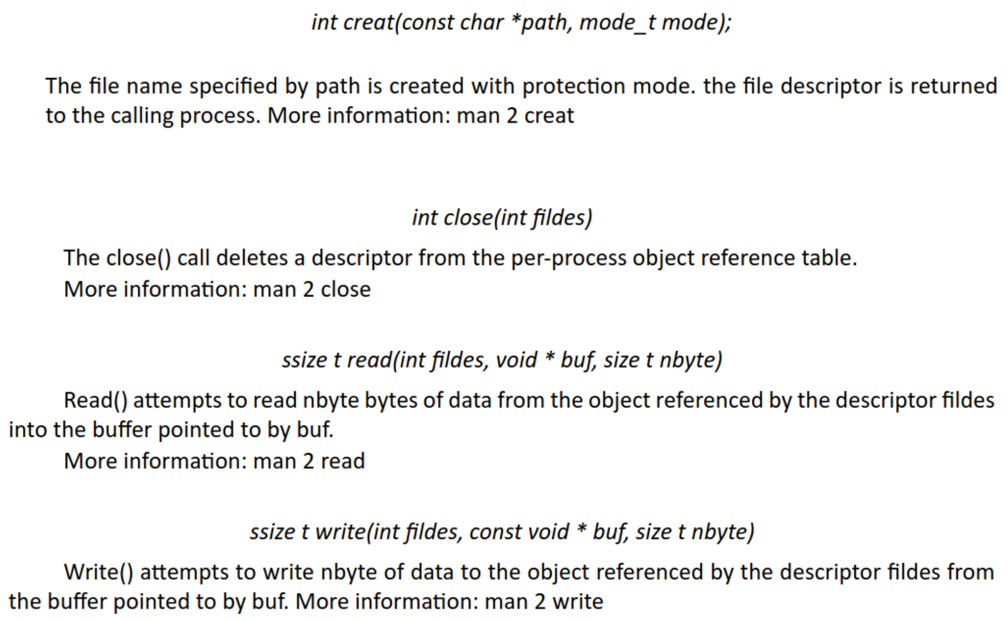
- **#include <sys/types.h>**: includes several data types used when working with system files and other processes that engage in low-level operations. For example, it includes the `size_t` variable, which we have used in the code.

Once all the libraries have been included, we have defined two important variables: `PERM`, which has the permissions **0644** (the ones most frequently used when creating or opening files), and `MAX_ALUMNS`, which has the maximum number of students that we will be able to count in the first two files (in our case it is **100**). Then, we have started to program the **main**. First of all, we check that the call to the program (`./combine <course file 1> <course file 2> <output file>`) is made with exactly 4 arguments:

- **./combine**: is the first argument (argv[0]), which represents the program's executable
- **<course file 1>**: is the second argument (argv[1]), which indicates the name of the first file from which we should retrieve the students
- **<course file 2>**: is the third argument (argv[2]), which indicates the name of the second file from which we should retrieve the students
- **<output file>**: is the fourth and last argument (argv[3]), which indicates the name of the file where we have to save the students from `<course file 1>` and `<course file 2>`.

To do this, as in **crear.c**, we have again compared whether the variable `argc`, which contains the number of arguments with which the program is called, is equal to 4. If not, it returns an error saying that the number of arguments is not correct.

Subsequently, we created the integers `infile1`, `infile2`, `outfile`, `estadisticas` and `ret`. The first two are used to open the first two arguments of the call to **create**, the third is used to open (or create if it does not exist) the file specified in the third argument, the fourth is used to create the file `estadisticas.csv` and last one to ensure we close the files succesfully. We also created the variables `nread` and `nwrite`, but in this case they are of type `ssize_t`. This is because, as the statement for the exercise says, `int` is usually used when we use variables as **file descriptors** to do **file operations** (for example, in the `open()` and `close()` functions), while when we want to read or write bytes, `ssize_t` is usually used (which, like the `mode_t` variable, is created specifically for POSIX systems). We have created all of these variables with a default value of -1. This way, if they can't be used, for example, to retrieve a student or write one to a file, we will be able to identify that there has been an error somewhere.


*int creat(const char *path, mode_t mode);*

The file name specified by path is created with protection mode. the file descriptor is returned to the calling process. More information: man 2 creat

int close(int fildes)

The close() call deletes a descriptor from the per-process object reference table. More information: man 2 close

*ssize t read(int fildes, void * buf, size t nbyte)*

Read() attempts to read nbyte bytes of data from the object referenced by the descriptor fildes into the buffer pointed to by buf.
More information: man 2 read

*ssize t write(int fildes, const void * buf, size t nbyte)*

Write() attempts to write nbyte of data to the object referenced by the descriptor fildes from the buffer pointed to by buf. More information: man 2 write

Figure 1: Screenshot from the statement of the assignment stating that `ssize_t` is used for functions like `read` and `write`, and `int` for functions like `creat`, `open` or `close`.

Then, as we do in the exercises of the practical classes, we are going to use the **buffer** to read students from the first two files. To do this, we create a variable `bufferSize` that contains the bytes that any student occupies and we create the variable **buffer** with the size saved in `bufferSize`. We must remember that the **buffer** is a variable that is normally used to temporarily save data that we want to transfer from one file to another. In addition, we have created an array called `alumns` of structures of type `alumnos`. This array contains by default 100 students with an empty name, and the grades and the call equal to -1. We do this to do **error handling**, since we want to make sure that all students have a name, a grade and a real call. For example, there cannot be students with a grade less than 0 or greater than 10, or students with an empty name. Finally, we also create the variable `count` since we need to count the number of students that we are going to save, since the array of structures will always have a size of 100 and it is not useful to us.

Afterwards, we proceeded to open the first file. Since we want to open it only for reading, we first save its mask (in the same way as in **crear.c**) and use only the `O_RDONLY` flag, which indicates that we want to open a file only for reading. If there is any error when opening the file, -1 is returned.

After opening the file, we need to read the students inside it. For them, we use the `nread` variable, in which we save the output of the `read` function, which will read students from the first file (`infile1`). To read the students correctly, we use the `bufferSize` variable, which contains the number of bytes that each student occupies, to know how many bytes we temporarily save in `buffer` to later pass them to the `nread` variable. If at any time during this process something fails, `nread` will continue to be -1, so we return an error. We do this whole process with a `while` loop, because we want to read all the students. If at any time a student cannot be read, it will exit the `while` and go into an `if` that detects errors in reading students.

One may wonder why we don't use the lseek function, which, as stated in the statement, moves the offset pointer of an open file to a specific position. This is because when we use the `write` and `read` functions, the file descriptor (`infile1`, `infile2`, `outputfile`...) **automatically updates its file offset** to the last byte read/written. So, knowing that the **file offset** is 0 at the beginning, when we finish reading or writing a student, the **offset** pointer will automatically start reading/writing at the first byte of the next student, thus making the program work as expected since we don't want the students' information to be overwritten.

On the other hand, while reading students, we will keep saving them in the list structure created before, called `alumns` with the `memcpy` function. If at any time the number of students reaches more than 100, the program will return an error. In addition, we have done **error handling** with the students, since we also control that their names are not empty and the grades and the call have logical values (for example, the grade must be between 0 and 10). If a student has an illogical data at any time, we return an error. To finish with this file, we close it in the samy way as in `crear.c`, with the `close` function. Then, we move on to the second one.

In the second file, we do exactly the same as in the first, but in this case the student counter (variable `count`) does not start from 0, but instead we add students to those in the first file. For all this, we use the variable `infile2` to open the file and the same variable `nread` to read and save students.

As the statement says, after reading all the students we must write them in the file specified in the third argument. To do this, we must first **sort them** in **ascending order** by grade, then by call and then by name. There are many types of **sorting** to do this, but we have decided to use the famous **bubble sort**. As we believe that this assignment does not talk about programming techniques, we are not going to explain much about this sorting. It is based on making **n iterations** in which we are going to **make comparisons between an element and the one on its right**, where a swap between both may or may not be made, being able to make a maximum of **n swaps**. With this algorithm we can sort the students that are in the array `alumns` as the exercise asks us. Although it is true that it is not the most efficient algorithm to do the sorting and it can cause problems in very large arrays, we have thought that as there can only be a maximum of 100 students, we are not going to have time or memory problems. After all, **bubble sort** has a **time complexity of** $\mathbf{O(n^2)}$, which, although not the best, is useful.

After doing the sort, before writing the students we have decided to save the **student statistics** because we will need later to write them in the file `estadisticas.csv`. First, we have created an array with **5** zeros called `grades`, in which we will relate each position of the array with how many students there are with that grade:

- **F's** (grades[0])

- **A's** (grades[1])
- **N's** (grades[2])
- **S's** (grades[3])
- **M's** (grades[4])

Once the array is created, we simply iterate through all the students we have read and we increase each value in the array with a `switch` that identifies the **grade** of each student.

After saving the necessary statistics, we have decided to proceed to save all the students already sorted in the file asociated to the third argument of the call to **combine**. To do this, as before, we save the mask of the third file, open it with the variable `outfile` created previously, and check if there is any error in the process. In this case, as the statement does not specify whether the third file is created or not, we are going to use a flag, `O_CREAT`, which needs to be accompanied by some permissions (variable `PERM` defined at the beginning of the program) so that the system knows what permissions to assign to the file. The flags we use in this case are the following:

- `O_WRONLY`: Open a file, if it is already created, for **writing** only
- `O_CREAT`: Creates the file if it does not exist.
- `O_TRUNC`: Truncates (empties) a file if it already exists.

Once the third argument file is opened or created, we only need to write all the students that we already have sorted into it. To do this, we are going to use the variable we created before of type `ssize_t` called `nwrite` and the function `write`, which as arguments needs the file to write to (`outfile`), the variable to be written (`alumns[i]`) and the number of bytes we are going to write from the variable (in this case, since we are going to write the entire student, `sizeof(alumns[i]` bytes are written). With a simple `for` we iterate through all the students saved in the `alumns` array and we write to the file until we reach the number represented by the variable `count`. If at any point during writing there is an error, the program will return an error. Finally, when we have written to all the students, we close the file and continue to the last section.

The last thing we need to do is write the statistics we have classified before in the file `estadisticas.csv`. To do this, as always, we first save its mask to then restore it, and **create** the file with the same flags we used in the file asociated to the third argument (`outfile`). Later, as the statement asks us to write to the file with a specific format, we create an array called `result` with enough byte size to be able to save what we want to write. In addition, we create the array `differentGrades` to identify the students of each grade ('F', 'A', 'N', 'S', 'M'). In other words, we create it to have a 1 to 1 association with the `grades` array.

Finally, with the `sprintf` function we concatenate all the information with the requested format and save it in the `result` variable. To calculate the total number of students with a specific grade, we are going to divide the number of students with that grade by the total. Since the total can be 0, we have put an `if` before doing the calculation to avoid having to divide by 0. In other words, if there are 0 students, we do not return an error, but we do show a message saying that there are no students to write and we write in `estadisticas.csv` that there are 0 students in each grade. With a `for` we iterate through all the different grades and we write in the file with the `write` function the information requested with the requested format, which will always be in the `result` variable. Since this variable is 20 bytes in size, but we only want to write the bytes that have text coming from the `sprintf` function, we only write `strlen(result)` bytes in each iteration of the `for` since the `strlen` function returns the number of characters before finding `'\0'`. Then we close the `estadisticas.csv` file and end the program returning a 0, since we have finished correctly.

2 Test cases

2.1 crear.c

Data to enter	**Description of the test**	**Result expected**	**Result obtained**
./crear myfile.dat 777	Default test to see if it works	Creation of **myfile.dat** with permissions: **rwxrwxrwx**	Creation of **myfile.dat** with permissions: **rwxrwxrwx**
./crear mydat.txt 444	Another default test to see if it works	Creation of **mydat.txt** with permissions: r--r--r--	Creation of **mydat.txt** with permissions: r--r--r--
./crear	Call of **crear** without file name and mode	Error with message: The number of arguments is not exact!	Error with message: The number of arguments is not exact!
./crear myfile.dat	Call of **crear** without mode	Error with message: The number of arguments is not exact!	Error with message: The number of arguments is not exact!
./crear ”” 777	Creation of a file with an empty name	Error with message: There has been an error creating the file: 2 (errno = 2)	Error with message: There has been an error creating the file: 2 (errno = 2)
./crear existing-file.dat 777	Creation of a file which already exists!	Error with message: There has been an error creating the file: 17 (errno = 17)	Error with message: There has been an error creating the file: 17 (errno = 17)

Continued on next page

(Continued)

./crear myfile.dat 999	Creation of a file with permissions that cannot be transformed to octal (octal max is 777)	Error with message: The input could not be converted to octal	Error with message: The input could not be converted to octal
./crear myfile.dat abc	Creation of a file with non-numeric permissions	Error with message: The input could not be converted to octal	Error with message: The input could not be converted to octal
./crear /root/my-file.dat 777	Creation of a file in a denied path (root)	Error with message: There has been an error creating the file: 13 (errno = 13)	Error with message: There has been an error creating the file: 13 (errno = 13)

2.2 combine.c

Data to enter	**Description of the test**	**Result expected**	**Result obtained**
./combine list1 list2 sortedfile.bin	Normal test with 8 alumnos. **In list1:** {"Pedro Perez", 8, 1}, {"Carlos Lopez", 8, 3}, {"Alfonso Garcia", 7, 1}, {"Maribel Fernandez", 5, 3} **In list2:** {"Juan Perez", 5, 1}, {"Maria Lopez", 9, 2}, {"Carlos Garcia", 7, 1}, {"Ana Fernandez", 4, 3}	Creation of '**sortedfile.bin**' with the following sorting: {"Ana Fernandez", 4, 3}, {"Juan Perez", 5, 1}, {"Maribel Fernandez", 5, 3} , {"Carlos Garcia", 7, 1}, {"Alfonso Garcia", 7, 1} , {"Pedro Perez", 8, 1}, {"Carlos Lopez", 8, 3}, {"Maria Lopez", 9, 2}. And **estadisticas.csv:** M;0;0.00% S;1;12.50% N;4;50.00% A;2;25.00% F;1;12.50%	Creation of '**sortedfile.bin**' with the following sorting: {"Ana Fernandez", 4, 3}, {"Juan Perez", 5, 1}, {"Maribel Fernandez", 5, 3}, {"Carlos Garcia", 7, 1}, {"Alfonso Garcia", 7, 1} , {"Pedro Perez", 8, 1}, {"Carlos Lopez", 8, 3}, {"Maria Lopez", 9, 2}. And **estadisticas.csv**: M;0;0.00% S;1;12.50% N;4;50.00% A;2;25.00% F;1;12.50%

Continued on next page

(Continued)

./combine list1 list2 sortedfile.bin	Test where there are 0 alumnos between the two input files	Creation of the two asked files with the message: "There are no students to write in **estadisticas.csv**!!"	Creation of the two asked files with the message: "There are no students to write in **estadisticas.csv**!!"
./combine list1 list2 sortedfile.bin	Test where the grade of an alumno is greater than 10	"Error: alumno with name %s can't have a grade lower than 0 or bigger than 10"	"Error: alumno with name %s can't have a grade lower than 0 or bigger than 10"
./combine list1 list2 sortedfile.bin	Test where the call is negative	"Error: alumno with name %s can't have a convocatoria less or equal than 0"	"Error: alumno with name %s can't have a convocatoria less or equal than 0"
./combine list1 list2 sortedfile.bin	Test where there is an alumno with no name	"Error: here is a student without name!"	"Error: here is a student without name!"
./combine list1 list2 sortedfile.bin	Test where there are more than 100 alumnos	"Error: there can't be more than 100 students!!"	"Error: there can't be more than 100 students!!"
./combine list1 list2	Test where there are not the exact number of arguments	"Error: the number of arguments is not exact!"	"Error: the number of arguments is not exact!"
./combine list1 list2 sortedfile.bin	Test where there is an alumno without call	file_creator can't create a file with an alumno without initializing name, grade and call	file_creator can't create a file with an alumno without initializing name, grade and call

3 Conclusion

We found this assignment very interesting as it was the first time we programmed in C. The system calls are the main feature of every operating system, as in any computer we constantle read and save information. At first we had numerous problems as we did not know how **Guernika** worked, but after we managed to understand its functions we could upload and test our code correctly. The most tedious part of this project was searching for information about which variables and functions we should be using in each case, for examle we should use in different occasions `strlen` and `sizeof`, or it is not the same to open a file with some permisions or others. The time spent in this project was about 40 hours.