REPORT P1

Alexandra Perruchot-Triboulet Rodríguez
NIA: 100453024

Sheila Gallardo Redondo
NIA: 100472913

Index of contents

mywc

Description of the code

Firstly, we have included the corresponding libraries so that our code could be executed without any problem. Then we have defined our buffer size to be 1 byte (N=1) and once inside the main function, we have declared the variables we are going to use during the program. These ones are the following:

- *fd* → file descriptor of the input file.
- *n* → where we will store the actual number of bytes read from the file.
- *buffer* (as a character) → where the data read is going to be temporally stored.
- Number of *bytes*.
- Number of *lines*.
- Number of *words*.

Now, we have considered the case of an error if the number of arguments passed to the function is lower than expected, that is, *argc* is smaller than 2.
Once this is checked, we will take the first argument (*argv[1]*) which is the file name, use the *open* system call to open the file for reading only (*O_RDONLY*) and assign *fd* as its descriptor. Then we check that there is no error opening the file and start to read it using a while loop which checks if each byte read is one of these three cases:

- ' ' → Add 1 to the number of words.
- '\t' → Add 1 to the number of words.
- '\n' → Add 1 to the number of lines and words because when we change of line, we change of word too.
- In each iteration we add 1 to the number of bytes.
- The first 3 points only happen if the number of bytes is greater than one considering the case that the file could start with one of these three first cases.

Once outside the loop, we add the last word of the file only if there is something written in the file (bytes>0) and if there is no ' ', '\t' or '\n' at the end of the word. This is because if there wasn't any of these, the word wouldn't be counted on the reading loop since we count words with ' ', '\t' or '\n' and so we add it at the end.

Finally, we check any error reading and if everything is okay, we close the file and print the results as asked:

<lines><space><words><space><bytes><space><file name>

Tests cases

- **TEST 1:** Check what happens with a totally empty file (tests.txt).
  - **Expected output:** *0 0 0 tests.txt*
  - **Actual output:** *0 0 0 tests.txt*

- **TEST 2:** Checks a file (tests.txt) with only one word (without ‘ ‘, ‘\t’ or ‘\n’) with 4 characters.
  - **Expected output:** *0 1 4 tests.txt*
  - **Actual output:** *0 1 4 tests.txt*

- **TEST 3:** Checks a normal file (tests.txt) where ‘ ‘, ‘\t’ and ‘\n’ are mixed.
  - **Expected output:** *3 5 35 tests.txt*
  - **Actual output:** *3 5 35 tests.txt*

- **TEST 4:** Checks a file (tests.txt) only with ‘ ‘.
  - **Expected output:** *0 0 1 tests.txt*
  - **Actual output:** *0 0 1 tests.txt*

We have obtained the expected output with the *wc* command on Linux.

Conclusions

When working through this function, we have encountered some problems such as:

- When we had a file with just a word without any blank space neither in front nor after, our program counted it as 0 words while it should return 1 word (as checked with command *wc*). We have fix it by adding an additional word if the number of bytes on the file is greater than 0 and the last byte is not ‘ ‘ (blank space).
- We also had problems when counting the last word of each line, as when it reaches the ‘\n’ it jumped to the next line without counting the last word of the previous line. To fix this, we add a word each time we find a ‘\n’.

myenv

Description of the code

In this second function, we have again included the corresponding libraries so that our code could be executed without any problem. Then we have defined our output file permissions (Perm) to be 0644, which means:

- User permission (6) → read and write (rw-).
- Group permission (4) → read (r--).
- Other permission (4) → read (r--).

We have defined our buffer size to be 1 byte (N=1) and once inside the main function, we have declared the variables we are going to use during the program. These ones are the following:

- *env* → array of 30 characters to store the variable received as an argument.
- *vect* → array of 30 characters to store the characters we read in the file until we find '='.
- *fd* → file descriptor of the input file (env.txt).
- *out* → file descriptor of the output file (created by us later).
- *n* → where we will store the actual number of bytes read from the file.
- *i* → vect index (to go through the elements of vect) and counter to store characters we are reading from buffer to vect.

To start with, we have considered the case of an error if the number of arguments passed to the function is lower than expected, that is, *argc* is smaller than 3.
Then, with a for loop, we have assigned the variable passed as an argument to the constant size string *env.*
Moreover, we open the file 'env.txt' using the *open* system call for reading only (*O_RDONLY*) and assign *fd* as its descriptor and check that there is no error opening the file. If the file is open properly, we create an output file with the name given as an argument (*argv[2])* and the permissions (*Perm*) defined above and assign *out* as its descriptor. Then check if there is an error creating the file and if everything is correct, we start the reading loop, but before we set *i*=0.

During the reading loop, we store the characters before the '='. When it arrives to '=' we compare the string stored in *vect* with the one stored in *env* with the function *strcmp*.
If they are equal, we write this string in the output file checking each time if there is an error when writing. If all is good, we write '=' (stored in *buffer[0]*) in the output file too and keep writing the whole line with another while loop. When *buffer[0]*='\n' (the line has finished)

then we write '\n' into the file, set *i* back to 0, clear out *vect* with the function *memset* and go back to the first while loop to repeat the process with the following lines.
If the string in *vect* is different from the one stored in *env,* it keeps reading the line with another while loop, until *buffer[0]*='\n', then set *i* back to 0, clear out *vect* with the function *memset* and go back to the first while loop to repeat the process with the following lines.
To finish the program we close both files (*fd* and *out*) and return 0.

Every time and error occurs when dealing with files, we close those files and return -1.

Test cases

- **TEST 1:** ./myenv PATH output.txt (word in the left side of a '=')
  - **Expected output:**
    PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games:/snap/bin
  - **Actual output:**
    PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games:/snap/bin

- **TEST 2:** ./myenv tomate output.txt (word that is not in the file)
  - **Expected output:** empty output file
  - **Actual output:** empty output file

- **TEST 3:** ./myenv bin output.txt (word that is in the right side of the '=')
  - **Expected output:** empty output file
  - **Actual output:** empty output file

- **TEST 4:** ./myenv PATH output.doc (to check a different extension for the file)
  - **Expected output:**
    PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games:/snap/bin (file with .doc extension)
  - **Actual output:**
    PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games:/snap/bin (file with .doc extension)

- **TEST 5:** ./myenv (checks error of no arguments)
  - **Expected output:** Too few arguments (in shell)
  - **Actual output:** Too few arguments (in shell)

- **TEST 6:** ./myenv PATH out.txt (without the env.txt file to check the error)
  - **Expected output:** Error opening the file (in shell)
  - **Actual output:** Error opening the file (in shell)

We have taken as outputs the content of the created output file.

Conclusions

When working through this function, we have encountered some problems such as:

- One of them, was how to compare two strings to check if the input variable was in the env.txt file. We found a function (*strcmp*) which takes as input two strings and returns 0 if they are equal.
- To use the *strcmp* function, we found a problem with the format of the first argument passed (the variable we are searching) to our program. As it is passed as a pointer, we cannot use it as an input for the *strcmp* function, so we had to do a for loop to store the variable pointed by the argument as a string in the array *env*.
- Another problem was that we didn't clear out the array *vect* where we stored the variables of each line from the file. Therefore, it wasn't reading the variables correctly, since each time it was overlapping the new characters with the ones that were already read. To fix this, we have use the function *memset* at the end of each line (every time we start to read a new variable) to clear out the positions of the array and restore the index *i* to store the new variable in *vect*.

myls

Description of the code

In this last function, we have included more libraries so that our code could be executed without any problem. We have declared the variables we are going to use during the program. These ones are the following:

- **dirname* → Name of the directory we want to access. We have defined it as a pointer.
- **dir* → Directory descriptor. We have defined it as a pointer.
- *Buffer[PATH_MAX]* → Where the data read is going to be temporally stored. It is defined as a character vector with maximum size of PATH_MAX (included in *limits.h* library).
- **read* → *Struct dirent* to access the contents of the directory. Defined as a pointer.

Now we can have two cases. First, if the number of argument passed is less than 2 then the name of the directory is not given so we take the current directory. To do that, we set *dirname* with the *getcwd* function and set the directory descriptor (*dir*) with the *opendir* system call. In the second case the directory is specified and so we set dirname as the first argument passed (*argv[1]*) and again sets the directory descriptor (*dir*) with the *opendir* system call.

Moreover, we check there is no error opening the directory (if there is an error, we return -1). If no error has occurred, then we use a while loop with the *read* structure and the *readdir* system call. With the *read* structure we access to the names of the entries of the directory and prints them.

To finish the program, we close the directory and return 0.

Test cases

- **TEST 1:** ./myls (checks the case without arguments)
  - **Expected output:**

    .
    myls.c
    myls
    out.doc
    output.txt
    ..
  - **Actual output:**

    .
    myls.c
    myls
    out.doc
    output.txt
    ..

- **TEST 2:** ./myls p1_tests (Checks the case with a directory as an argument)
  - **Expected output:**

    .
    dirB
    f1.txt:Zone.Identifier
    dirA
    f1.txt
    ..
  - **Actual output:**

    .
    dirB
    f1.txt:Zone.Identifier
    dirA
    f1.txt
    ..

- **TEST 3:** ./myls hu23 (Try to open a directory that doesn't exist)
  - **Expected output:** Error opening the directory
  - **Actual output:** Error opening the directory

- **TEST 4:** ./myls output.txt (Try to open a file instead of a directory
  - **Expected output:** Error opening the directory
  - **Actual output:** Error opening the directory

We have obtained the expected output with the *ls -f -l* command on Linux.

Conclusions

When working through this function, we have encountered some problems such as:

- Firstly, we had to add the library *<limits.h>* to be able to use *PATH_MAX.*
- We had problems assigning the name of the directory passed as an argument to the variable *dirname* because we defined *dirname* as an array of characters and the argument was given as a pointer. To solve this, we declared *dirname* as a pointer (**dirname*).
- In order to declare the directory descriptor, we had to use a special data type (*DIR*). Again, it is defined as a pointer.
- We had problems accessing the contents of the directory. We found that to do this, we had to use the *struct dirent.* This allow us to define the structure where we are going to store the contents of the directory we want to read (**read*). Then, once we have the structure, we can access the name of the entries with '*d_name*' and so print them on the screen.
- Finally, we had a segmentation fault error when verifying errors in the *opendir* function. We forgot to put the *else* after the *if* condition to verify the error.

To end up this report, we have found the project useful to learn how to use LINUX and how to implement functions in C and execute them in LINUX.