

Computer Architecture and Technology Area (ARCOS)

Universidad Carlos III de Madrid



OPERATING SYSTEMS

Lab 1: System Calls

**Bachelor's Degree in Computer Science & Engineering
Bachelor's Degree in Applied Mathematics & Computing
Dual Bachelor's in Computer Science & Engineering & Business
Administration**

Academic Year 2023/2024

Index

1	Lab Statement	2
1.1	Lab Description	2
1.1.1	mywc	2
1.1.2	myls	3
1.1.3	myishere	4
1.1.4	Support Code	5
1.1.5	Program tester	6
2	Lab Submission	6
2.1	Delivery Deadline	6
2.2	Submission Procedure	6
2.3	Files to Deliver	6
3	Rules	9
4	Appendix – System calls	10
4.1	I/O system calls	10
4.2	File related system calls	11
4.3	Manual (man function)	11
5	Bibliography	11

1 Lab Statement

This lab allows the student to become familiar with the OS system calls (in particular, the file system) following the POSIX standard. Unix allows system calls to be made directly from a program written in a high-level language, particularly C. Most file input/output (I/O) operations in Unix can be performed using only five calls: `open`, `read`, `write`, `lseek` and `close`.

For the operating system kernel, all open files are identified by file descriptors. A file descriptor is a non-negative integer. When we open (`open`) an existing file, the kernel returns a file descriptor to the process. When we want to read or write to/from a file, we identify the file with the file descriptor returned by the previously described call.

Each open file has a current read/write position (“**current file offset**”). It is represented by a non-negative integer that measures the number of bytes from the beginning of the file. Read and write operations usually start at the current position and cause an increase in that position, equal to the number of bytes read or written. By default, this position is initialized to 0 (zero) when a file is opened, unless the `O_APPEND` option is specified. The current position (`current_offset`) of an open file can be changed explicitly using the `lseek` system call.

To manipulate directories, the system calls `opendir`, `readdir` and `closedir` can be used. An open directory is identified with a directory descriptor, which is a pointer to a type `DIR` (`DIR*`). When we open a directory with `opendir`, the kernel returns a directory descriptor, on which the entries of that directory can be read through calls to the `readdir` function. The `readdir` call returns a directory entry in a pointer to a `dirent` structure (`struct dirent*`). This structure will contain the fields corresponding to that entry such as the entry name, or the type (if it is a normal file, if it is another directory, symbolic links, etc.). Successive calls to the `readdir` function will return successive entries of an open directory.

1.1 Lab Description

The aim is to implement three C programs that use the system calls previously described. These programs will be **mywc**, **myls** and **myishere**. For this, you will have the corresponding code files `mywc.c`, `myls.c`, and `myishere.c`.

1.1.1 mywc

The first program, **mywc**, will open a file specified as an argument, **count the number of lines, words, and bytes on it**, and show these through the standard output (the console) using the appropriate system calls. To do this:

- It will open the file passed as a parameter.
- It will read the contents of the file **byte by byte**.
- It will update the counters based on the bytes read. It is understood that **two lines are separated by the character ‘\n’**, while **two words can be separated by the**

characters ' ' (blank space) or '\t'. In addition, there will be no more than one successive space or line break.

- It will show the results on the console, followed by the file name. Separating each value from the next with a blank space.
- Finally, it will close the file.

```
$ ./mywc p1_tests/f1.txt
2 15 85 p1_tests/f1.txt
```

- Usage: `./mywc <file_name>`
- Requirements:
 - The program must show the **number of lines, words, and bytes on the console, followed by the name of the read file.**
 - The program will show the data in the following format:
`<lines><space><words><space><bytes><space><file_name>`
 - The program must return -1 if no input argument has been passed.
 - The program must return -1 if there was an error opening the file (e.g. the file does not exist).
 - The program must return 0 if everything was correctly.
- **Suggestion of test:**¹ Check that the output of the program on a file matches the output of the ***wc*** command (without arguments) on that same file. To do this, it is recommended to compare the two outputs with the ***diff*** command.

1.1.2 myls

The second program, ***myls***, will open a directory specified as an argument (or the current directory if no directory is specified as an argument), and will display on the screen the name of all the entries of that directory, printing one entry per line. To do this:

- It will obtain the directory to list, from the program arguments, or from the current directory through the ***getcwd*** call. Use the `PATH_MAX` constant as the maximum size that the directory path can have.
- It will open it with ***opendir***.
- Then, it will read each of the directory entries through ***readdir*** and print the name of the entry using ***printf***.
- Finally, it will close the directory descriptor with ***closedir***.

¹Meeting this test is not a guarantee of getting the maximum grade in the exercise. It is only a suggestion for students to check the general operation of their program. Students must also meet the other requirements of the program, make the appropriate code, comment it, test extreme cases, and generally meet the other conditions described in the lab statement.

```
$ ./mys p1_tests/  
dirC  
f1.txt  
dirA  
f2.txt  
.  
..
```

- **Usage 1:** `./mys <directory_name>`
- **Usage 2:** `./mys`
- **Requirements:**
 - The program must list **all the entries** of the directory, in the order in which the call to `readdir` returns them, and display each entry line by line.
 - The program must list the entries of the directory passed as a parameter (usage 1) or the current directory if they have not passed any parameter (usage 2).
 - **mys** must show the current directory (`.`) and the parent directory (`..`).
 - The program must return **-1** if there was an error opening the directory (e.g. the directory does not exist).
- **Suggestion of test:**² Check that the output of the program on a directory matches that of the `ls -f -1` command on that same directory: `ls -f -1 <directory_name>`. To do this, it is recommended to compare the two outputs with the *diff* command.

1.1.3 myishere

The third program, **myishere**, receives the name of a directory as a first argument and checks if the file whose name is received as the second argument is in that directory. In order to do this:

- It will open the directory received as argument using *opendir*.
- It will read each entry by calling *readdir* comparing the entry name with the file received as argument. Do this until either a match is found or there are no more entries in the directory.
- If the file is found, the file name is going to be printed by the standard output (the console) in the following format:
`File <file_name> is in directory <directory_name>`
- If the file is not found, a message with the following format is going to be printed on the standard output (the console):
`File <file_name> is not in directory <directory_name>`

²Meeting this test is not a guarantee of getting the maximum grade in the exercise. It is only a suggestion for students to check the general operation of their program. Students must also meet the other requirements of the program, perform the appropriate code, comment it, test extreme cases, and generally meet the other conditions described in the lab statement.

- Finally, it will close the directory using *closedir*.

```
$/myishere p1_test f1.txt
File f1.txt is in directory p1_test
$/myishere p1_test f2.txt
File f2.txt is not in directory p1_test
```

- Usage: `./myishere <directory_name> <file_name>`
- Requirements:
 - The program must work with any directory and check if any of its entries match with the provided file.
 - The program must show one of the messages previously mentioned, depending on if the file has been found or not.
 - The program must return -1 if not enough arguments are provided.
 - The program must return -1 if there is any problem opening the directory.
 - The program must return 0 if everything works properly (working properly is intended as checking if the file is in the directory or not without generating any kind of error).
- Note: The library `string.h` contains the function `strcmp`. You can use it to compare two strings.

1.1.4 Support Code

The **p1_syscall_2024.zip** file is made available to support the development of this lab. To extract its content, run the following command:

```
unzip p1_syscall_2024.zip
```

After extracting its content, the `p1_syscall` directory is created, where the lab must be developed. The following files are included in this directory:

- **Makefile**
DO NOT modify. Source file for the `make` tool. It automatically recompiles the source files that are modified. Use `make` to compile the programs, and `make clean` to delete the compiled files.
- **mywc.c**
Must be modified. C source file where students must code the **mywc** program.
- **mys.c**
Must be modified. C source file where students must code the **mys** program.
- **myishere.c**
Must be modified. C source file where students must code the **myishere** program.
- **authors.txt**
Must be modified. `txt` format file where the authors of this practice will be included.

- `checker_os_p1.py`
DO NOT modify. Program tester provided for the practice.
- `p1_tests/`
This directory contains example files and directories to execute and test the programs.

1.1.5 Program tester

The **python (Version 3)** script **`checker_os_p1.py`** is given to the student to verify that the submission follows the format conventions (it has the correct names and it is well compressed) and run some functionality tests, printing on the screen a tentative grade obtained with the provided code. The tester must be executed in Ubuntu Linux Virtual Machines or in the Virtual Classrooms provided by the Informatic Department laboratory. To run the program tester, use the following command:

```
python3 checker_os_p1.py <submission.zip>
```

Where `deliverable.zip` is the file that will be delivered to Aula Global (see next section).
Example:

```
$ python3 checker_os_p1.py os_p1_100254896_100047014.zip
```

The program tester will print messages on the screen indicating whether the format is or is not correct.

2 Lab Submission

2.1 Delivery Deadline

The deadline for the submission of the laboratory in AULA GLOBAL will be **March 8th, 2024 at 23:55h**

2.2 Submission Procedure

The delivery of the laboratory must be done electronically and by **a single member of the group**. In AULA GLOBAL, links will be enabled to deliver the practices. Specifically, **a deliverer will be enabled for the practice code, and another with TURNITIN for the practice report.**

2.3 Files to Deliver

A compressed file in zip format must be delivered with the name:

```
os_p1_AAAAAAAAAA_BBBBBBBBBB_CCCCCCCC.zip
```

Where A...A, B...B and C...C are the NIAs of the members of the group. In case of doing the practice alone, the format will be **`os_p1_AAAAAAAAAA.zip`**. **The zip file will be delivered in the corresponding deliverer for the practice code.** The file must contain:

- Makefile
- mywc.c
- myls.c
- myishere.c
- **authors.txt:** Text file in CSV format with one author per line. The format is: NIA, Surnames, Name

NOTE

To compress these files and be processed correctly by the provided program tester, it is recommended to use the following command:

```
zip os_p1_AAA.BBB.CCC.zip Makefile mywc.c myls.c myishere.c authors.txt
```

The report will be delivered in PDF format in a file called:

os_p1_AAAAAAAAAA_BBBBBBBBBB_CCCCCC.pdf.

Only reports in pdf format will be corrected and graded. They must contain at least the following sections:

- **Description of the code** detailing the main functions implemented. DO NOT include any source code in this section of the report. Any code will be automatically ignored.
- **Test cases** used and results obtained from their execution: Higher scores will be given to advanced tests, extreme cases, and in general, to those tests that guarantee the correct operation of the functions in all cases. There are three clarifications to take into account:
 1. If a program compiles correctly and without warnings is not a guarantee that it will work correctly.
 2. Avoid duplicating tests that assess the same program flows. The score in this section is not measured by the number of tests, but by the degree of test coverage. Few tests that evaluate different cases are better than many tests that always evaluate the same case.
- **Conclusions**, problems found, how they have been solved, and personal opinions.

The following aspects related to the **presentation** of the lab report will also be scored:

- It must contain a cover page, with the authors and their NIAs.
- It must contain an index of contents.
- The report must have page numbers on all pages (except the cover).
- The text of the report must be justified.

The pdf file will be delivered in the corresponding deliverer for the practice report (TURNITIN deliverer).

NOTE: It is possible to deliver the practice code as often as you want within the delivery period, with the last delivery being considered the definitive version. **THE PRACTICE REPORT CAN ONLY BE DELIVERED ONCE THROUGH TURNITIN.**

3 Rules

1. Programs that do not compile or do not satisfy the requirements will receive a mark of **zero**.
2. **Programs that use library functions (fopen, fread, fwrite, etc.) or similar, instead of system calls, will receive a grade of zero. It is also not allowed to use statements or functions such as goto or stat.**
3. Students are expected to submit original work. In case plagiarism is detected between two assignments both groups will fail the continuous evaluation. Additional administrative charges of academic misconduct may be filled.
4. All programs should compile without reporting any **warnings**.
5. The programs implemented must work in a virtual machine running Ubuntu Linux or in the Virtual Aulas provided by the informatics lab at the university platform. It is the student responsibility to be sure that the delivered code works correctly in those places.
6. Programs without comments will receive a **very low grade**.
7. The assignment must be submitted using the available links in Aula Global. Submitting the assignments by mail is not allowed without prior authorization.
8. It is mandatory to follow the input and output formats indicated in each program implemented. In case this is not fulfilled there will be a penalization to the mark obtained.
9. It is mandatory to implement error handling methods, **beyond what is explicitly requested, in each of the programs**.

Failing to follow these rules will be translated into zero marks in the affected programs.

4 Appendix – System calls

A system call allows user programs to request services from the operating system. In this sense, system calls can be seen as the interface between the user and kernel spaces. In order to invoke a system call it is necessary to employ the functions offered by the underlying operating system. This section overviews a subset of system calls offered by Linux operating systems that can be invoked in a C program. As any other function, the typical syntax of a system calls follows:

```
status = function (arg1, arg2,.....);
```

4.1 I/O system calls

```
int open(const char * path, int flag, ...)
```

The file name specified by `path` is opened for reading and/or writing, as specified by the argument `flag`; the file descriptor is returned to the calling process.

More information: `man 2 open`

```
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`

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

The `lseek()` function repositions the offset of the file descriptor `fildes` to the argument `offset`, according to the directive `whence`. The argument `fildes` must be an open file descriptor. `Lseek()` repositions the file pointer `fildes` as follows:

- If `whence` is `SEEK.SET`, the offset is set to `offset` bytes.
- If `whence` is `SEEK.CUR`, the offset is set to its current location plus `offset` bytes.
- If `whence` is `SEEK.END`, the offset is set to the size of the file plus `offset` bytes.

More information: `man 2 lseek`

4.2 File related system calls

```
DIR * opendir(const char * dirname)
```

The `opendir()` function opens the directory named by `dirname`, associates a directory stream with it, and returns a pointer to be used to identify the directory stream in subsequent operations.

More information: `man opendir`

```
struct dirent * readdir(DIR * dirp)
```

The `readdir()` function returns a pointer to the next directory entry. It returns `NULL` upon reaching the end of the directory or detecting an invalid `seekdir()` operation. The *dirent* structure contains a field *d_name* (`char * d_name`) with the filename and a *d_type* field (*unsigned char d_type*) with the type of file.

More information: `man readdir`

```
int closedir(DIR * dirp)
```

The `closedir()` function closes the named directory stream and frees the structure associated with the *dirp* pointer, returning 0 on success.

More information: `man closedir`

4.3 Manual (man function)

man is a command that formats and displays the online manual pages of the different commands, libraries and functions of the operating system. If a section is specified, man only shows information about name in that section. Syntax:

```
man [section] open
```

A man page includes the synopsis, the description, the return values, example usage, bug information, etc. about a name. The utilization of man is recommended for the realization of all lab assignments. **To exit a man page, press q.**

5 Bibliography

- El lenguaje de programación C: diseño e implementación de programas Félix García, Jesús Carretero, Javier Fernández y Alejandro Calderón. Prentice-Hall, 2002.
- The UNIX System S.R. Bourne Addison-Wesley, 1983.
- Advanced UNIX Programming M.J. Rochkind Prentice-Hall, 1985.
- Sistemas Operativos: Una visión aplicada Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez. McGraw-Hill, 2001.
- Programming Utilities and Libraries SUN Microsystems, 1990.
- Unix man pages (`man function`)