Laboratory Assignment 3: Basic concepts of file systems

Table of contents

1 Objectives

In this practice we are going to solve several exercises oriented to reinforce the basic concepts of files and directories in POSIX systems. We will work with the system calls: open, read, write, close, lstat, readlink, symlink, lseek, opendir and readdir.

The student is advised to create a directory for the laboratory with one subdirectory per exercise. The instructions assume that exercise N is done in a subdirectory named exercise*N* within the common directory for the laboratory.

The file files_p3.tar.gz contains a series of files that can be used as a starting point for the development of the exercises of this laboratory, as well as some makefiles that can be used for the compilation of the different projects.

2 Exercises

Exercise 1: POSIX API

Implement a *copy.c* program to make a copy of a regular file. The program will receive two parameters on the command line. The first will be the name of the source file and the second will be the name of the destination file.

The program must make the copy in 512B blocks. For this purpose a buffer of 512 bytes will be reserved as intermediate storage. The program must read 512 bytes from the source file and write them to the target file. The student must take into account that if the size of the file is not a multiple of 512, in the last iteration, 512 bytes will not be read. Therefore, `read` will return the number of bytes read.

The student should refer to the man pages of the following system calls: `open`, `read`, `write` and `close`. In the `open` man page pay special attention to the opening flags, having to use in this case: `O_RDONLY`, `O_WRONLY`, `O_CREAT` and `O_TRUNC`.

It is suggested to the student that before running his copy program, test the effect of `O_TRUNC`, using a file with any content named like the target file. Check that the contents of the file disappear when the `O_TRUNC` flag is used.

To check the correct operation of our program we can use the shell commands `diff` and `hexdump` (the latter for binary files).

Exercise 2: Symbolic links

The first thing we are going to do in this exercise is to create a symbolic link to any file using the `ln` command. For example, if we want to create a link that is called *mylink* and points to the file *../exercise1/Makefile* we will use the following shell command:

```
$ ln -s ../exercise1/Makefile mylink
```

By invoking `ls -l` we can check that the created file is really a symbolic link and we will see the file it points to:

```
$ ls -l
...
lrwxrwxrwx 1 christian christian   22 Jul 14 13:23 mylink -> ../exercise1/Makefile
...
```

Now we will use our copy program to copy the symbolic link. Assuming that the program is *../exercise1/copy*, we run:

```
$ ../exercise1/copy mylink mylinkcopy
```

What type of file is *mylinkcopy*? What is the content of the *mylinkcopy* file? You can use `ls`, `stat`, `cat` and `diff` commands to get the answers to these questions.

It is possible that this is the behavior we want, but it is also possible that we don't. What if we want the copy of a symbolic link to be another symbolic link pointing to the same file that the original symbolic link pointed to?

We are going to make a modification of our copy program from the previous exercise, which we will call *copy2.c*. We can start by copying the previous program and then modify it. We can make a copy using the `cp` command:

```
$ cp ../exercise1/copy.c copy2.c
```

Then we will modify the *copy2.c* file so that:

1. Before making the copy, identify whether the source file is a regular file, a symbolic link, or another type of file, using the `lstat` system call (see its manual page).
2. If the source file is a regular file, we will make the copy as in the previous exercise.
3. On the other hand, if the source file is a symbolic link we do not have to make a copy of the file pointed to but create a symbolic link that points to the same file to which the source file points to. To do this we have to follow the following steps:
   a. Reserve memory to make a copy of the pointed path. A call to `lstat` on the source file will allow us to know the number of bytes occupied by the symbolic link, which corresponds to the size of this path without the *null* character ('\0') at the end of the string (see the `lstat` manual page). Therefore we will add one to the size obtained from `lstat`.
   b. Copy into this buffer the path of the file pointed to using the `readlink` system call. We must manually add the null character at the end of the string.
   c. Use the `symlink` system call to create a new symlink pointing to this path.

   You should consult the `lstat`, `readlink` and `symlink` manual pages.
4. If the source file is of any other type (e.g. a directory), an error message will be displayed and the program will exit.

Exercise 3: Random file access

In this exercise we are going to create a program *show.c* similar to the `cat` command, that receives as parameter the name of a file and displays it by the standard output. In this case we will assume that it is a standard file. In addition, our program will receive two arguments that we will parse with `getopt` (see its manual page):

- `-n N`: indicates that we want to skip N bytes from the beginning of the file or show only the last N bytes of the file. Whether we do one or the other depends on the presence or not of a second `-e` flag. If the `-n` flag is not present, N will take the value 0.
- `-e`: if present, the last N bytes of the file will be read. If it is not present, the first N bytes of the file will be skipped.

The program must open the specified file on the command line (see the `optind` in the `getopt` manual page) and then place the file pointer in the correct position before reading. To do this we make use of the system call `lseek` (see the manual page). If the user has used the `-e` flag we must place the file pointer N bytes before the end of the file. If the user has not used the `-e` flag we must advance the pointer N bytes from the beginning of the file.

Once the file pointer has been placed, we must read byte by byte until the end of file, writing each byte read by the standard output (descriptor 1).

Exercise 4: Directory traversing

In this exercise we are going to create a program *space.c* that receives a list of filenames as input parameters, and will calculate for each one the total number of kilobytes reserved by the system. In case any of the processed files are a directory, the kilobytes occupied by the files contained in the directory will also be added up (note that this is recursive, because a directory can contain other directories).

To know the number of kilobytes reserved by the system to store a file, we can call `lstat`, which allows us to know the number of 512-byte blocks reserved by the system. To identify if a file is a directory, we must make a call to `lstat` and check the st_mode field (see the `lstat` manual page).

To traverse a directory, you must first open it using the `opendir` library function and then read its entries using the `readdir` library function. See the manual pages for these two functions. Note that the entries in a directory do not have a fixed order and that every directory has two entries "." and "..", which we should ignore if we do not want to have an infinite recursion.

The program should display by standard output one line per file, with the file name and the total size of the file in kilobytes. To check if our program works correctly we can compare its output with that of the `du -ks` command, passing to this command the same list of files as to ours. Note that you can use wildcards.

Example of use:

```
$ ls -l .
total 40
-rwxr-xr-x 1 christian christian 20416 Jul 15 12:41 space
-rw-r--r-- 1 christian christian  1639 Jul 15 12:41 space.c
-rw-r--r-- 1 christian christian  9056 Jul 15 12:41 space.o
-rw-r--r-- 1 christian christian   273 Jul 15 09:54 Makefile
$ ./space .
44K .
$ ./space *
20K space
4K  space.c
12K space.o
4K  Makefile
```

Exercise 5: File and directory management

In the last two exercises, we will work with typical file attributes, learning how to query and modify them from the command line, and observing their implications when interacting with programs written in C language.

To do so, we will develop a *script* called `prepare_files.sh`, which will perform the following preliminary actions:

1. Create and access a directory provided as the only argument to the *script*. If it does not exist, the directory will be created (`mkdir`). If it does exist, its entire contents will be deleted using the appropriate options of the `rm` command or by using the `rmdir` command.
2. Create in the specified directory a set of files with the following characteristics and names:

Name	Type	Remarks	Commands to consult (man)
subdir	Directory		mkdir
file1	Regular file	Created without content	touch
file2	Regular file	Created and with 10 characters written	echo
Slink	Symbolic link	Symbolic link to the file `file2`	ln
Hlink	Hard link	Hard link to file `file2`	ln

3. It will run through all the files created, displaying on screen all their attributes using the `stat` command, of which you can obtain more information by consulting the corresponding manual page (`man 1 stat`).

Based on the results observed after the execution of the *script*, give a reasoned answer to the following questions:

1. How many disk blocks does the file `file1` occupy? And the file `file2`? What is its size in bytes?
2. What is the reported size for `subdir`? Why is the *number of links* field in this case 2?
3. Do any of the created files or directories share an i-node number? Include in your answer the behavior with a directory, specifically of the hidden entries of the `subdir` directory (you can use `stat` for this, or the combination of modifiers (`-i` and `-a` of `ls`).
4. Display the contents of `Hlink` and `Slink` using the `cat` command. Then delete the file `file2` and repeat the procedure. Can you access the contents of the file in both cases?
5. Use the `touch` command to modify the access and modification dates of file `Hlink`. What changes are seen in the output of `stat` after execution? Investigate through the man page to modify only one of these dates (modification or access).

Exercise 6: Permissions and modes

We will work with the modification of access permissions to a given file. To do this, refer to the `chmod` command manual page. Note that the two supported modes for specifying the permissions you want to grant to a given file:

- **Symbolic mode**, which uses a mnemonic argument to specify the permissions to be granted to a given file.
- **Octal mode**, which uses an octal-based number to represent the bit pattern of the *mode* field of the i-node associated with the file.

Experiment with the `chmod` command and give any file read, write or execute permissions using its two modes of operation. Observe the implications of changing the mode when reading the contents of the file (`cat`) or writing to it (through, for example, `echo`), as well as the changes that occur in the corresponding i-node (`stat` or `ls -l`).

Finally, it is important to differentiate between the permissions granted to a given file or directory (static and stored in its associated i-node) and the way it is opened from a program written in C (dynamic and stored in the internal tables of the operating system), which ultimately restrict the operations that can be performed on the file from the running process.

To do this, write a C program called `open.c` (compiled as `open.x`) that, using POSIX system calls, meets the following characteristics:

1. The program will receive a mandatory argument (`-f`) followed by the name of the file to open.
2. The program will receive, through two optional arguments (`-r` for read and `-w` for write) the desired opening mode for the file. It is necessary to provide at least one opening mode, and both can be combined to open in read/write mode.
3. The program will try to open (`open`) the file with the indicated mode, reporting an error if it is not possible. If the file does not exist, it will be created. Otherwise, its entire contents will be deleted.
4. In any case, and independently of the opening mode selected, the program will try to write (`write`) from the file and then read (`read`), reporting an error if it is not possible to perform either of these operations.
5. Finally, the file will be closed (`close`).

Once developed, experiment with at least the following situations:

1. Grant read and write permissions to an existing file on the system, and run the `open` program with the three available options (`-r`, `-w` and `-rw`). Is it possible to read and/or write to the file in all cases? Which function returns the error in this case? Why?

2. Remove the read permission on the target file. Which function returns the error message in this case? Why? (you can also do the same by removing the write permission, or both simultaneously).

3. Remove the execute permission on the executable file `./open.x`. What is the implication for the execution of the executable file (`./open.x`)?