Operating Systems

Grado en Informática 2026/2027

Lab Assignment 1: File systems

We'll start to code the shell that we will continue in following lab assigments. Check the supplied shell to check the workings and syntax for the commands. You can use the help command to get help.

In addition to the commands described below, the shell will keep a list with the files opened (keeping AT LEAST the name of the file and the file descriptor) by the open comand (and closed by the close command). That list must be coherent with the system list that we can get with **lsof -p pid** (in linux) or **procstat -f pid** (in freeBSD). The commands read, write, and lseek ressemble the system calls of the same name and operate on descriptors in that file. Note that every process inherits files opened by its parent process, and thus should be in the list as well

Please download the latest version of the reference shell and use it to check the workings of the different commads

command name	**arguments**	**description**
exit **bye**		exits the shell
date	 **-d** **-t**	shows the present date and time shows only the date shows only the time
pid **getpid**	 **-p**	shows ths shell's pid shows the shell's parent process pid
authors	 **-l** **-n**	shows the shell's authors (names and logins) shows only the logins shows only the names
sysinfo		shows information on the machine (something like uname -a)
help	 *cmd*	shows the shells help. Lists all available commands gives information on the command
chdir	 dir	changes or shows the current working directory changes the currect working directory of the shell to dir
open	file m1 m2 ...	the shells opens a file and adds the file (with its descriptor) to an internal list, kept by the shell, of open files file is the name of the file m1, m2..are some of the following opening modes: cr: O_CREAT ap: O_APPEND ex: O_EXCL ro: O_RDONLY rw: O_RDWR wo: O_WRONLY tr: O_TRUNC without arguments open lists the open files
close	df df -f	closes the file descriptor df, and eliminates the corresponding list entry does the closing even if df corresponds to an active mapping
listopen		lists the shells open files

read	df addr cont	transfers cont bytes from the openfile df to the address addr. addr is expressed in HEX and need not be a valid address.
write	df addr cont	transfers cont bytes from addr to the open file described by df. addr is expressed in HEX and need not be a valid address.
lseek	df pos ref	positions the offset if file descriptor to pos. ref is the reference and can be: SEEK_SET: pos is relative to the beginning of the file SEEK_CUR: pos is relative to the current position in the file SEEK_END: pos is relative to the end of the file
readstr	df cont	reads cont bytes from file descrptor df and prints them on screen as if it were a string
writestr	df str	writes the string str to the open file described by df
makefile	**name**	creates an empy file of name nam
makedir	**nam**	creates a directory of name nam
delete	name1 name2	deletes filesystem objects (files, links, directories) of name name1, name2... For a directory to be deleted it must be empty
deltree	name1 name2	deletes filesystem objects (files, links, directories) of name name1, name2... If a directory is not empty it is deleted together with all of its content
listfile	[-long][-link][-acc] nam1 nam2...	gives information of filesystem objetcs name1 name2....In the case one of the names is a directory, information on the directory file itself is given (not of its contents). For any item only its name an size are printed. Unless the following parameters are used -acc: last access time is used -long: long listing. In adition to name and size: creation (or access) time, number of links, mode, owner, group. Only ONE line per file -link: if file were to represent a symbolic link, the file it points to is also printed
list	[-reca] [-recb] [-hid][-long] [-link][-acc] name1 name2 ..	same as stat (listfile, listfich) except that if any of the names represents a directory it's contents are listed instead of the directory file itself -hid: hidden files in directories are listed as well -reca: directories are listed recursively. recursion takes place AFTER listing the directory --recb: directories are listed recursively. recursion takes place BEFORE listing the directory

IMPORTANT:

- Information on the system calls and library functions needed to code these programs is available through man: (open, opendir, readdir, lstat, stat, unlink, mkdir, rmdir, realpath, readlink . . .) (**should you choose to use ftw or nftw, you must be able to COMPLY WITH THE SHELL SPECIFICATIONS AND BEHAVIOUR OF THE REFERENCE SHELL, and be assured you'll be questioned about how does it work!!!**)
- A reference shell is provided (for various platforms) for students to check how the shell should perform. This program should be checked to find out the syntax of the various commands
- An additional C file (ayudaP1.txt) is provided with some useful functions
- To check what type of filesystem object a name is, one of the *stat* system calls must be used. **DO NOT USE THE FIELD d_type IN THE DIRECTORY ENTRY**
- The program should compile cleanly (produce no warnings even when compiling with "gcc -Wall")
- **NO RUNTIME ERROR WILL BE ALLOWED** (segmentation, bus error . . .). Programs with runtime errors will yield no score.
- These programs can have no memory leaks (you can use valgrind to check)
- When the program cannot perform its task (for whatever reason, for example, lack of privileges) it should inform the user (See errors section)
- All input and output is done through the standard input and output

ERRORS:

When a system call cannot perform (for whatever reason) the task it was asked to do, it returns a special value (usually -1), and sets an external integer variable (errno) to an error code. The man page of the system call explains the reason why the system call produced such an error code.

A generic message explaning the error can be obtained with any of these methods:

- the **perror()** function prints that message to the standard error (the screen, if the standard error has not been redirected)
- the **strerror()** function returns a string with the error description if we supply it with the error code
- the external array of pointers, `extern char * sys_errlist[]`, contains the error descriptions indexed by error number so that `sys_errlist[errno]` has a description of the error associated with errno

WORK SUBMISSION

- Work must be done in pairs.
- **campusvirtual.udc.gal** will be used to submit the source code: a zipfile containing ONLY a directory named **P1 (capitals)** where all the source files of the lab assignment reside. The name of the zip file should be **p1.zip (lowercase)**
- The name of the main program will be p1.c (lowercase). Program must be able to be compiled with **gcc p1.c**, Optionally a Makefile can be supplied so that all of the source code can be compiled with just make. Should that be the case, the compiled program should be called p1 and placed on the same P1 directory (no build subdirs or similar)
- **ONLY ONE OF THE MEMBERS OF THE GROUP** will submit the source code. The names and logins of all the members of the group should appear in the source code of the main programs (at the top of the file) (as well as output to the command *authors*)
- **Works submited not conforming to these rules will be disregarded.**
- DEADLINE: 23:00, Friday October the 9th, 2026
- Failure to deliver the lab assignment following the above instructios will yield a score of 0

INTERESTING THINGS TO TRY ONCE YOU HAVE PROGRAMMED THIS SHELL

*** REDIRECTION:** In adition to the terminal where we are executing the shell, we open another terminal. Let's imagine that the terminal where the shell is is /dev/pts/1 and the other is /dev/pts/2. Consider the following shell commands (the C like comments, /*..*/, are not part of the commands)

```
#) open /dev/pts/2 rw   /*we asume we get 3 as the file descriptor */
#) dup 1                        /*we asume we get 4 as the dupped file descriptor */
#) close 1                      /*now the shell prints nothing, it has no standard output*/
#) dup 3                        /*from now on, all the shell output goes to the other terminal */
.....
. . . . . .
#) close 1
#) dup 4                        /*we restore the standard output*/
```

*** SPARSE FILES:** Lets do the following shell commands

```
#) open  prueba.txt cr rw      /*create file prueba.txt, we asume we get 3 as the file descriptor */
#) writestr  3  Holaaaaaa
#) lseek 3 1000000000 SEEK_SET
#) writestr 3 ***Adios
```

prueba.txt is a sparse file, its size is 10000000008, has 9 bytes at offset 0 (Holaaaaaa) and 8 bytes at offset 1000000000. What does it have in between? (try to cat it). What size does *ls -l* report? How much disk space is it using up? (try *du -k*)? What disk usage did report the system when it only had the first bytes?

*** FILES WITH NO OWNER:** Create new user (execute as root `useradd -m nadie`). Delete that user but not his home directory (f.e., execute as root `userdel nadie`). Check who now owns its home directory (`ls -l /home`). Check what your shell thinks of it (**`list -long /home`**); if you get Segmentation Fault: you have misprogrammed, check your code!!