Operating Systems

Grado en Informática. Course 2021-2022

Lab assignment 0: Introduction to C programming language

To get acquainted with the C programming language we'll start to code a shell, coding of this shell will be continued in next lab assignments.

We'll start with a nearly empty shell, which is basically a loop that

- prints a *prompt*
- reads from the standard input a line of text which includes a command (with its arguments).
- stores this command in a list of commands that it has been given, each with its order number in a list we'll call the *historic* of commands.
- separates the command and its arguments
- processes the comand with its arguments

At this moment this *shell* has to understand only the following commands. In the next lab assignments we will be COMPLETING this shell: we BUILD lab assignment 1 ON the code of lab assigment 0, lab assigment 2 ON the code of lab assigment 1 and so on

autores [-l|-n] Prints the names and logins of the program authors. **authors -l** prints only the logins and **authors -n** prints only the names

pid [-p] Prints the pid of the process executing the shell. **pid -p** rints the pid of the shell's parent process.

carpeta [direct] Changes the current working directory of the shell to *direct* (using the *chdir* system call). When invoked without auguments it prints the current working directory (using the *getcwd* system call.

fecha [-d|-h] Without arguments it prints both the current date and the current time. **fecha -d** prints the current date in the format DD/MM/YYYY. **fecha -h** prints the current time in the format hh:mm:ss.

hist [-c|-N] Shows/clears the *historic* of commands executed by this shell. In order to do this, a list to store all the commands input to the shell must be implemented. *historic -c* clears the historic, that's to say, empties the list

  - `hist` Prints all the comands that have been input with their order number
  - `hist -c` Clears (empties) the list of *historic* commands

– `hist -N` Prints the first N comands

comando N Repeats command number N (from historic list) Example:

```
*) carpeta
/home/antonio/Practicas
*) pid -p
11562
*) pid
11581
*) carpeta /root
Cannot change dir /root: permission denied
*) carpeta pru.c
Cannot change dir pru.c: not a directory
*) hist
0->carpeta
1->pid -p
2->pid
3->carpeta /root
4->carpeta pru.c
6->hist
*) hist -2
0->pwd
1->getppid
2->getpid
*) comando 3
 carpeta /root
 Cannot change dir /root: permission denied
```

Students are free to decide whether *historic* stars numbering commands at 0 or at 1. Hypothetically, there's a scenario where trying to repeat a *historic* command could yield an infinite loop or a stack overflow (depending on how it is coded), so students may choose to not store calls to *comando* itself in the historic list if they want so (such scenario would be `comando 9` in the previous example)
(See the *NOTES ON LIST IMPLEMENTATIONS* at the end of this document)

infosis Prints information on the machine running the shell (as obtained via the *uname* system call/library function)

ayuda [cmd] *ayuda* displays a list of available commands. *ayuda cmd* gives a brief help on the usage of comand *cmd*

fin Ends the shell

salir Ends the shell

bye Ends the shell

IMPORTANT

- This program should compile cleanly (produce no warnings even when compiling with `gcc -Wall`)
- **NO RUNTIME ERROR WILL BE ALLOWED (segmentation, bus error ...)**, unless where explicitly spcified. **Programs with runtime errors will yield no score.**
- This program can have no memory leaks (please use valgrind to check)
- When the program cannot perform its task (for whatever reason, for example, trying to change the current working directory to a directory that does not exist or that shell has not enough privileges) it should inform the user, giving an appropiate description of the error such as the one given by *strerror(), perror(),* `sys_errlist[errno]` ...
- All input and output is done through the standard input and output
- Executable files of an implementation of this shell are provided. Please check out them for any doubts.

Information on the system calls and library functions needed to code this program is available through `man`: (*printf, gets, read, write, exit, getpid, getppid, getcwd, chdir, time ...).*

WORK SUBMISSION

- Work must be done in pairs.
- The name of the main program file will be `p0.c`. Program must be able to be compiled with `gcc p0.c` Alternatively a `Makefile` can be supplied so that the program can be compiled with just `make`
- Only one of the members of the workgroup will submit the source code. **The names and logins of all the members of the group should be in the source code of the main program** (at the top of the file, as comments)

DEADLINE: September, Friday the 24th . This lab assignment will yield no score, neither will it be evaluated. However all the code for this assigmnet must be reutilized for the following assigments. This assigment will also help get acquainted wit the submission procedure of all of the following lab assignments (from the next assignment on, work wrongly submitted will no be evaluated). Submission procedure will be announced at a later date

CLUES

A shell is basically a loop

```
while (!terminado){
   imprimirPrompt();
   leerEntrada();
   procesarEntrada();
}
```

imprimirPrompt() and *leerEntrada()* can be as simple as calls to `printf` y `gets` (there's a reason why *fgets()* should be used instead fog *gets()*)

The first step when processing the input string is splitting it into words. For this, the `strtok` library function comes in handy. Please notice that `strtok` nor allocates memory neither does copy strings, it just breaks the input string by inserting end of string ('\0') characters. The following function splits the string pointed by *cadena* (suposedly not null) into a NULL terminated array of pointers (*trozos*). The function returns the number of words that were in *cadena*

```
int TrocearCadena(char * cadena, char * trozos[])
{ int i=1;

  if ((trozos[0]=strtok(cadena," \n\t"))==NULL)
    return 0;
  while ((trozos[i]=strtok(NULL," \n\t"))!=NULL)
    i++;
  return i;
}
```

NOTES ON LIST IMPLEMENTATION

- the implementations of list should consist of the data types and the access funtions. All access to the list should be done used the aforementioned access functions.
- students can choose from one of these three list implementations
    1) **linked list:** The list is composed of dynamically allocated nodes. Each node has some item of information and a pointer to the following node. The list itself is a pointer to the first node, when the list is empty this pointer is NULL, so creating the list is asigning NULL to the list pointer, thus the functions `CreateList`, `InsertElement` and `RemoveElement` must receive the list by reference as they may have (case of inserting or removing the first

element) to modify the list. There can also be used a double linked version of this list (each node has two pointers)

2) **linked list with head node:** Similar to the linked list except that the list itself is a pointer to an *empty* (with no information) first node. Creating the list is allocating this first element (head node). `CreateList` must receive the list by reference whereas `InsertElement` and `RemoveElement` can receive the list by value. There can also be used a double linked version of this list (each node has two pointers)

3) **array of pointers;** The list is an array (statically allocated) of pointers. Each pointer points to one element in the list which is allocated dynamically. For the purpose of this lab assigment we can assume this statically allocated array dimension to be 4096, which should be declared a named constant, and thus easily modifiable. To implement the list with this array we can use either a NULL terminated array or we can use aditional integers. We could also make the list completely dynamic by using a dynamically allocated pointer instead the fixed size array of pointers