Lab 5

Lab Goals

- Get acquainted with command interpreters ("shell").
- Understand how Unix/Linux fork() works.
- Learn how to read the manual (man).
- Better profficiency with C, addressing non-trivial memory allocation/deallocation issues.

> **Note**
> Labs 5 is independent of Lab 4.
> You are indeed expected to link your code with stdlib, and use the C standard library wrapper functions which invoke the system calls.
> Nevertheless, you will be extending your code from lab 5 in lab 6, so try make your code readable and modular.

Motivation

Perhaps the most important system program is the **command interpreter**, that is, the program that gets user commands and executes them. The command interpreter is thus the major interface between the user and the operating system services. There are two main types of command interpreters:

- Command-line interpreters, which receive user commands in text form and execute them (also called **shell** in UNIX-like systems).
- Menu-based interpreters, where the user selects commands from a menu. At the most basic level, menus are text driven. At the most extreme end, everything is wrapped in a nifty graphical display (e.g. Windows or KDE command interpreters).

Lab Goals

In this sequence of labs, you will be implementing a simple shell (command-line interpreter). Like traditional UNIX shells, your shell program will **also** be a **user level** process (just like all your programs to-date), that will rely heavily on the operating system's services. Your shell should do the following:

- Receive commands from the user.
- Interpret the commands, and use the operating system to help starting up programs and processes requested by the user.
- Manage process execution (e.g. run processes in the background, kill them, etc.), using the operating system's services.

The complicated tasks of actually starting up the processes, mapping their memory, files, etc. are strictly a responsibility of the operating system, and as such you will study these issues in the Operating Systems course. Your responsibility, therefore, is limited to telling the operating system which processes to run, how to run these processes (run in the background/foreground) etc.

Starting and maintaining a process involves many technicalities, and like any other command interpreter we will get assistance from system calls, such as execv, fork, waitpid (see **man** on how to use these system calls).

Lab 5 tasks

First, download LineParser.c and LineParser.h. These files contain some useful parsing and string management functions that will simplify your code substantially. Make sure you include the c file in your makefile. You can find a detailed explanation here.

> **Deliverables**
> You should read and understand the reading material and do task 0 before attending the lab. To be eligible for a full grade, you must complete at least all of task 1 during the regular lab. Task 2 may be done in a completion lab if you run out of time.

Task 0

Here you are required to write a basic shell program **myshell**. Keep in mind that you are expected to extend this basic shell during the next tasks. In your code write an infinite loop and carry out the following:

1. Display a prompt - the current working directory (see man getcwd). The path name is not expected to exceed **PATH_MAX**.
2. Read a line from the "user", i.e. from stdin (no more than 2048 bytes). It is advisable to use **fgets** (see man).
3. Parse the input using **parseCmdLines()** (LineParser.h). The result is a structure **cmdLine** that contains all necessary parsed data.
4. Write a function **execute(cmdLine *pCmdLine)** that receives a parsed line and invokes the command using the proper system call (see man **execv**).
5. Use **perror** (see man) to display an error if the execv fails, and then exit "abnormally".
6. Release the cmdLine resources when finished.
7. End the infinite loop of the shell if the command "quit" is entered in the shell, and exit the shell "normally".

Once you execute your program, you'll notice a few things:

- Although you loop infinitely, the execution ends after execv. Why is that?
- You must place the full path of an executable file in-order to run properly. For instance: "ls" won't work, whereas "/bin/ls" runs properly. (Why?)

Now replace execv with execvp (see man) and try again .

- Wildcards, as in "ls *", are not working. (Again, why?)

In addition to the reading material, please make sure you read up on and understand the system calls: fork(2), exec(2) and its variants, and waitpid(2), before attending the "official" lab session.

Task 1

In this task, you will make your shell work like a real command interpreter (tasks 1a and 1b), and then add various features (task 1c through 1e).
When executed with the "-d" flag, your shell will also print the debug output to stderr (if "-d" is not given, you should not print anything to stderr).

Task 1a

Building up on your code from task 0, we would like our shell to remain active after invoking another program. The **fork** system call (see man) is the key: it 'duplicates' our process, creating an almost identical copy (**child**) of the issuing (**parent**) process. For the parent process, the call returns the process ID of the newly-born child, whereas for the child process - the value 0 is returned.

> **You will need to print to `stderr` the following debug information in your task:**
>
> - PID
> - Executing command

Notes:

- Use fork to maintain the shell's activeness by forking before **execvp**, while handling the return code appropriately. (Although if fork( ) fails you are in real trouble!).
- If execvp fails, use **_exit()** (see man) to terminate the process. (Why?)

Task 1b

Until now we've executed commands without waiting for the process to terminate. You will now use the **waitpid** call (see man), in order to implement the wait. Pay attention to the **blocking** field in cmdLine. It is set to 0 if a "&" symbol is added at the end of the line, 1 otherwise.

Invoke waitpid when you're required, and only when you're required. For example: "cat myshell.c &" will not wait for the cat process to end, but "cat myshell.c" will.

Task 1c

Add a shell feature "cd" that allows the user to change the current working directory. Essentially, you need to emulate the "cd" shell command. Use **chdir** for that purpose (see man). **Print appropriate error message to stderr if the cd operation fails.**

> **You will need to propagate the error messages of chdir to stderr.**

Task 1d

Add a "history" command which prints the list of all the command lines you've typed, in an increasing chronological order. Namely, if the last command typed is "ls", then "ls" is the last command to appear. A printout of N commands should consist of N lines - numbered from #0 (the first) to #N-1 (the last). You can partially test your code by using this input scenario:

```
ls -l
echo hello world
cd .
history
ls -l
history
```

> **Tip**
> Use an array to store your list. You may assume history to contain at most 10 commands, but do not use the number in your code more than once (use #define).

You will need to propagate the error messages to stderr.
such as out of bounds index.

Task 1e

History list is useful if one wishes to re-use previously typed commands. To use a previously typed command in say, index #0, one should invoke the command "!0". Likewise for 1,2 etc.

Add the "!" command implementation, as explained above. **Make sure the history list adds the reused command line as it was initially entered.** That is - if the command "ls -l" currently occupies index #2 in the log list, then typing "!2" should: (i) Invoke the "ls -l" command, and (ii) Add "ls -l" to the bottom of the log list. **Do not add !-commands to the history list, instead, add the original command.**

Remember to **print an error message when a non-existing log index is invoked** (e.g. when you have only 5 history entries, "!5" should trigger an appropriate error message).
You can partially test your code by using this input scenario:

```
ls -l
echo hello world
cd .
!0
!1
history
ls
wrong_command
cd ~/
ls -a
pwd
echo echo echo echo
!0
history
```

You will need to propagate the error messages to stderr.

Add the full command to the history list, including all of its parameters

Task 2

Here we wish to emulate the internal variables environment of the command shell. Briefly, the internal variables environment is a list of string pairs (name, value), which associates names with values. When variable names appear in the command line with a "$" prefix, they are replaced with their associated value.
For instance, let variable **i** be mapped to value **Hello.** "`echo $i`" will then print "Hello", whereas "`echo i`" will print "i". Similarly, the command "`ls $i`" will be translated to "ls Hello", etc.

The task is divided into four mini-tasks, which define operations relevant to the internal variables environment. Although the environment can be implemented in several ways, **you are strictly required to base your implementation on a linked list of (name, value) string pairs.** Your implementation should be general enough to support unlimited number of variables and unlimited length of names and values.

1. Add an set command to the shell, which associates a given name with a given value. **If the name already exists, the command should override the existing value.**
   Usage: "set x y" creates an environment variable with name x and value y.

2. Add an env command, which prints all current associations in the environment.
   Usage: "env" prints out all environment associations.

3. Activate the internal variables in each executed command line, by replacing each argument that starts with a **$** sign with its proper environment value. **Write an appropriate error message when variables are not found.**

4. Add a delete command, which delete a variable from the environment. **Write an appropriate error message when variables are not found.**
   Usage: "delete x" deletes the variable x from the environment.

5. Adding support of `~` for `cd`. Initiating the command `cd ~` in your shell will result in an `error`. You need to add support for converting ~ to your home directory. Find the environment variable for your home directly, **do not hard code it.**

You will need to propagate the error messages to stderr.
- Deleting an environment variable that does not exist.
- Activating an environment variable that does not exist.

Deliverables:

Tasks1 must be completed during the regular lab. Task 2 may be done in a completion lab, but only if you run out of time during the regular lab. The deliverables must be submitted until the end of the day. You must submit source files for task 1e, and task 2 and a makefiles that compile it. The source files must be named task1e.c, task2.c, makefile1, makefile2

Submission instructions

- Create a zip file with the relevant files (only).
- Upload zip file to the submission system.
- Download the zip file from the submission system and extract its content to an empty folder.
- Compile and test the code to make sure that it still works.