Lab 6: Reading Material

Implementing a Command Interpreter (Shell)

You should read the introductory material from the task descriptions as part of the reading material.
Download the attached code in the task file and learn how to use it.

Thoroughly understand the following:

1. All the material from previous labs related to the C language, especially using pointers and structures in C.
2. Read the man pages and understand how to use the following system calls and library functions: getcwd, fork, execv, execvp, perror, waitpid, pipe, dup, fopen, close.

Basic introduction to the notion of fork can be found here and here.

Attached code documentation

LineParser:

This package supports parsing of a given string to a structure which holds all the necessary data for a shell program.
For instance, parsing the string **"cat file.c > output.txt &"** results in a cmdLine struct, consisting of arguments = {"cat", "file.c"}, outputRedirect = "output.txt", blocking = 0 etc.

```
typedef struct cmdLine
{
    char * const arguments[MAX_ARGUMENTS];
    int argCount;
    char const *inputRedirect;
    char const *outputRedirect;
    char blocking;
    int idx;
    struct cmdLine* next;
} cmdLine;
```

Included functions:

1. **cmdLine* parseCmdLines(const char *strLine)**
   Returns a parsed structure cmdLine from a given strLine string, NULL when there was nothing to parse. If the strLine indicates a pipeline (e.g. "ls | grep x"), the function will return a linked list of cmdLine structures, as indicated by the **next** field.

2. **void freeCmdLines(cmdLine *pCmdLine)**
   Releases the memory that was allocated to accomodate the linked list of cmdLines, starting with the head of the list pCmdLine.

3. **int replaceCmdArg(cmdLine *pCmdLine, int num, const char *newString)**
   Replaces the argument with index *num* in the arguments field of pCmdLine with a given string.
   If successful returns 1, otherwise - returns 0.

MAN: Relevant Functions and Utilities

fork(2), wait(2), waitpid(2), _exit, exit(3)

Lab 6

Lab Goals

- Get acquainted with command interpreters ("shell").
- Understand how Unix/Linux fork() works.
- Learn how to read the manual (man).
- Write tedious code in C, and address non-trivial memory allocation/deallocation issues.
- Learn to debug your code!

Note
Labs 6 is independent of Lab 5.
You are indeed expected to link your code with stdlib, and use the C standard library wrapper functions which invoke the system calls.
Your code is expected to run without any memory leaks or errors.

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 6 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.
You will need to print to `stderr` the following debug information in your task:

- PID
- Executing command

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 (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.
6. Release the cmdLine resources when finished.
7. End the infinite loop once the command "quit" is entered in the shell.

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?)

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 and 1d).

Task 1a

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.

- Use fork to maintain the shell's activeness by forking before **execvp**, while handling the return code appropriately.
- 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: "kate&" will not wait for the cat process to end, but "kate" 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). **Write an appropriate error message 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).

Hint: define array with history maximum size and insert new command line with O(1).

You can partially test your code by using this input scenario:

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

Task 2

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.

You can partially test your code by using this input scenario:

```
ls -l
echo hello world
cd .
!0
!1
history
```

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).