Lab 5

Note
Labs 5 is independent of Lab 4.
You are indeed expected to compile your code with stdlib, and use the c-library wrapper functions which invoke the system calls.

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 **shells** 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 task 1 during the lab and task 2 during the completion lab.

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 (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 (man **execv**).
5. Use **perror** (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 (man) and try again .

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

Task 1a

We would like our shell to remain active after invoking another program. The **fork** system call (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()** (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 (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 wait.

Task 1c

Add a shell feature "chdir" that enables the user to change the current working directory. Essentially, you need to emulate the "cd" shell command. Use **chdir** for that purpose (man).

Task 1d

Add a "hist" command which

- Prints the list of the last 10 commands you typed, in a decreasing chronological order. Namely, if the last command you typed is "ls", then "ls" is the first command to appear. Naturally, if you typed less than 10 commands thus far (say, 3), only 3 commands should appear on screen.
- Adds a command to the list ONLY if it differs from the last command executed by the shell. This means that if you run "ls" twice consecutively, only one of those instances should appear in the history list.

Tip
Keep a record of the last 10 commands you typed in STRING form, prior to parsing via the cmdLine parser. This should make things much easier.

Task 1e

Add a "rep" command which repeats the last command executed by the shell. The "rep" command **should not appear** the history list.

Task 2a

Add a "print" command which prints to output all the arguments that follow.
For instance: "print Hello world" will print "Hello world".

Task 2b

Here we wish to emulate the command shell variables environment. Briefly, the 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 enlisted value.
For instance, let variable **i** be mapped to value **Hello**. "print $i" will then print "Hello", whereas "print i" will print "i".

The following mini-tasks require the usage of library functions **getenv**, **setenv** and **unsetenv** (see man). You will also need to use the externally declared variable, **extern char **environ**, which is an array of strings, pointing to the environment variables associated with the running process. This string array ends with a **pointer to a NULL string** (please take notice: a pointer to a NULL string is not a NULL pointer).

1. Add a 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 unset command, which removes a variable from the environment. Write an appropriate error message when variables are not found.
   Usage: "unset x" removes variable x.

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

   

   environ contains all the environment variables, including those not defined by your shell.
   Make sure you **print only the variables defined by your shell**.

4. Enforce the variable environment on 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.

Page last modified on 6 May 2012, 18:13 by **apsel**
powered by **CourseWiki 2.4.2** , **gzip** supported
generated in 0.0013038 sec.