lab 2

In this lab, you should perform **task 0 before attending the lab session**. This lab may be done in pairs (as a rule, members of a pair must be in the same lab group, and should inform the TA(s)).

Goals

- Get acquainted with command interpreters ("shell") by implementing a simple command interpreter.
- Understand how Unix/Linux fork() and exec() work.
- Introduction to Linux signals.
- Redirection, and introduction to pipes.
- Learn how to read the manual (man).

Note
You will be extending your code from lab 2 for use in lab C, 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 (2 and C), 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, suspend 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 2 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 appropriately refer to LineParser.c in your makefile. You can find a detailed explanation in the reading material for lab 2.

Throughout the lab pay close attention to the difference between **processes**(things that you run with execvp() after fork()) and **shell commands**. Think about when do you need a new process and when to use the process of the shell. Running things in a different process preserves inter-activeness with the shell. However, not all things can be run in a new process.

Task 0a

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** (it's defined in **linux/limits.h**, so you'll need to include it).
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 program specified in the cmdLine 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 will 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, signal(2), and waitpid(2), before attending the "official" lab session.

Task 0b

Add the signal handler looper.c that prints the signal with a message saying it was received, and propagates the signal to the default signal handler. This is what really makes the process sleep/continue. The signals you need to address are: SIGTSTP, SIGINT, SIGCONT. The signals will be sent to the looper by the shell that you are going to write to test the functionality of the signal commands in task 3 and the process manager that you are going to implement in lab C.

- Use strsignal (see: man strsignal) to get the signal name.
- See signal(2) you will need it to set your handler to handle these signals.
- Use signal(signum, SIG_DFL) to make the default handler handle the signal.
- Use raise() to send the signal again, so that the default signal handler can handle it.
- After handling SIGCONT, make sure you reinstate the custom handler for SIGTSTP
- After handling SIGTSTP, make sure you reinstate the custom handler for SIGCONT

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.
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 alive (recall mandatory lecture 2) by forking before **execvp**, while handling the return code appropriately (again as stated in the lecture). (Although if fork( ) fails you are in real trouble anyway (e.g. fork bomb!), so you might as well ignore this case).
- 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 (cat in this case runs in the **background**), but "cat myshell.c" will (cat runs in the **foreground**).

Task 1c

Add a **shell command** "cd" that allows the user to change the current working directory. Essentially, you need to emulate a simplified version of the "cd" internal shell command: use **chdir** for that purpose (see man). No need to implement anything beyond transferring the argument of "cd" to "chdir". **Print appropriate error message to stderr if the cd operation fails.**

Task 2: Redirection

Add standard input/output redirection capabilities to your shell (e.g., **"cat < in.txt > out.txt"**). Guidelines on I/O redirection can be found in the reading material.

Notes:

- The **inputRedirect** and **outputRedirect** fields in cmdLine do the parsing work for you. They hold the redirection file names if exist, NULL otherwise.
- Remember to redirect input/output only in the child process. We do not want to redirect the I/O of the shell itself (parent process).

Task 3 - Signals

Every program you run using the shell runs as a process. You can get a list of the running processes using the ps program (see: man 1 ps and man 2 ps). In this task we are going

to implement shell commands to help manage the processes using signals. Implement and test the following commands:

- wakeup <process id> - Wake up a sleeping process (SIGCONT).
- nuke <process id> - Terminate a running/sleeping process.

In both cases, use the kill( ) system call wrapper, see man 2 kill, to send the relevant signal to the given process id. Check if kill( ) succeeded and print an appropriate message.

Test your shell using your looper code from task0b in the following scenario:

```
#> ./looper&
#> ./looper&
#> ./looper&
#> ps
PID TTY        TIME CMD
17998 pts/11   00:00:00 bash
24207 pts/11   00:00:00 task2
24246 pts/11   00:00:00 looper
24279 pts/11   00:00:00 looper
24326 pts/11   00:00:00 looper
24336 pts/11   00:00:00 ps
#> nuke 24326
#> ps
PID TTY        TIME CMD
17998 pts/11   00:00:00 bash
24207 pts/11   00:00:00 task2
24246 pts/11   00:00:00 looper
24279 pts/11   00:00:00 looper
24326 pts/11   00:00:00 looper <defunct>
24336 pts/11   00:00:00 ps
```

Now that you finished tasks 1, 2, 3, save your code aside. You will need it for submission and for Lab C.

Task 4: Exercise in Pipe System Call

Recall from the lecture that a pipe is a pair of input stream/output stream, such that one stream feeds the other stream directly. All data that is written to one side (the "write end") can be read from the other side (the "read end"). This sort of feed becomes pretty useful when one wishes to communicate between processes, such as when implementing a shell pipe as described in the lecture. This task is to help you execise the basic pipe mechanism, towards achieving a shell pipe implementation (part of what you will be doing in lab C).

Your task: Implement a simple program called **mypipe** (a program **separate** from your shell), which creates a child process that sends a message such as "hello" to its parent process. The parent then prints the incoming message and terminates. Use the **pipe** system call (see man) to create the pipe.

Deliverables: lab 2

Tasks 1, 2, 3, 4 must be completed during the regular lab. The deliverables must be submitted until the end of the lab session.
You must submit 2 source files:
The first one is the shell including task 1,2, 3. Name it myshell.c
The second one is mypipe.c (task 4).
Also, submit a makefile that compile both of them, that is: running "make myshell" should generate the "myshell" executable, and "make mypipe" should generate the "mypipe" executable

Submission instructions

- Create a zip file with the relevant files (only) (named either [student-id-num].zip [student1-id-num_student2-id-num.zip] in case of pair submission).
- 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.