Lab 5

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).

In this lab you will create a simple shell and implement **input/output redirection** and **pipelines** (see reading material). Your shell will then be able to execute non-trivial commands such as **"tail -n 2 in.txt| cat > out.txt"**, demonstrating the power of these simple concepts.

Lab 5 tasks

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

> You should read and understand the reading material and do task 0 before attending the lab.
> It is extremely important to check whether a command fails before using variables it alters.

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** (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 **parse_cmd_lines()** (line_parser.h). The result is a structure **cmd_line** that contains all necessary parsed data.
4. Write a function **execute(cmd_line *line)** that receives a parsed line and invokes the program specified in the cmd_line 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 cmd_line 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.

> **Memory Leaks**
> In this lab and **onwards** you are required to use valgrind to make sure your program is "memory-leak" free. You should use valgrind in the following manner: valgrind --leak-check=full --show-reachable=yes -v [your-program] [your-program-options]

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 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.

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 cmd_line. 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**).

Example:

```
$ ./myshell
/home/admin/caspl182/lab5/task1$:cat
hello
hello
/home/admin/caspl182/lab5/task1$:     <------------pay attention: here 'Enter' is pressed
/home/admin/caspl182/lab5/task1$:
/home/admin/caspl182/lab5/task1$:quit
$
```

"/home/admin/caspl182/lab5/task1" should be replaced with your current working directory.
"<------------pay attention: here 'Enter' is pressed" is not part of your shell's output. It's just a note, telling you that your shell must not crash if enter is pressed without any command!

Task 1c

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 **input_redirect** and **output_redirect** fields in cmd_line 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).
- Output redirection ">" creates the output file if it doesn't exist (see BASH behavior as an example).

Example:

```
$ ./myshell
/home/admin/caspl182/lab5/task1$:
/home/admin/caspl182/lab5/task1$:cat < in.txt > out.txt
open failed: No such file or directory
/home/admin/caspl182/lab5/task1$:echo "some input" > in.txt
/home/admin/caspl182/lab5/task1$:cat in.txt
"some input"
/home/admin/caspl182/lab5/task1$:ls -l
total 64
-rwxrwxr-x 1 admin admin    13 Apr 29 05:51 in.txt
-rw------- 1 admin admin  3898 Apr 29 02:01 line_parser.c
-rw------- 1 admin admin  1218 Apr 29 02:01 line_parser.h
-rw-rw-r-- 1 admin admin 10072 Apr 29 05:27 line_parser.o
-rw-r--r-- 1 admin admin   298 Apr 29 05:27 Makefile
-rwxrwxr-x 1 admin admin 19498 Apr 29 05:47 myshell
-rw-rw-r-- 1 admin admin  1968 Apr 29 05:47 myshell.c
-rw-rw-r-- 1 admin admin  8952 Apr 29 05:47 myshell.o
-rwxrwxr-x 1 admin admin     0 Apr 29 05:51 out.txt
/home/admin/caspl182/lab5/task1$:rm out.txt
/home/admin/caspl182/lab5/task1$:ls -l
total 64
-rwxrwxr-x 1 admin admin    13 Apr 29 05:51 in.txt
-rw------- 1 admin admin  3898 Apr 29 02:01 line_parser.c
-rw------- 1 admin admin  1218 Apr 29 02:01 line_parser.h
-rw-rw-r-- 1 admin admin 10072 Apr 29 05:27 line_parser.o
-rw-r--r-- 1 admin admin   298 Apr 29 05:27 Makefile
-rwxrwxr-x 1 admin admin 19498 Apr 29 05:47 myshell
-rw-rw-r-- 1 admin admin  1968 Apr 29 05:47 myshell.c
-rw-rw-r-- 1 admin admin  8952 Apr 29 05:47 myshell.o
/home/admin/caspl182/lab5/task1$:cat < in.txt > out.txt
/home/admin/caspl182/lab5/task1$:ls -l
total 68
-rwxrwxr-x 1 admin admin    13 Apr 29 05:51 in.txt
-rw------- 1 admin admin  3898 Apr 29 02:01 line_parser.c
-rw------- 1 admin admin  1218 Apr 29 02:01 line_parser.h
-rw-rw-r-- 1 admin admin 10072 Apr 29 05:27 line_parser.o
-rw-r--r-- 1 admin admin   298 Apr 29 05:27 Makefile
-rwxrwxr-x 1 admin admin 19498 Apr 29 05:47 myshell
-rw-rw-r-- 1 admin admin  1968 Apr 29 05:47 myshell.c
-rw-rw-r-- 1 admin admin  8952 Apr 29 05:47 myshell.o
-rwxrwxr-x 1 admin admin    13 Apr 29 05:52 out.txt
/home/admin/caspl182/lab5/task1$:cat out.txt
"some input"
/home/admin/caspl182/lab5/task1$:quit
$
```

The output of `ls -l` doesn't need to match the output given here! It only needs to have the created file in the case of in.txt, after its creation, and not have out.txt after removing it, but have it after creating it using the redirection.

Task 2

Pipes

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.

Here we wish to explore the implementation of a pipeline. In order to achieve such a pipeline, one has to create pipes and properly redirect the standard outputs and standard inputs of the processes.
Please refer to the 'Introduction to Pipelines' section in the reading material.

Your task: Write a short program called **mypipeline** which creates a pipeline of 2 child processes.
Essentially, you will implement the shell call **"ls -l | tail -n 2"**.
(A question: what does "ls -l" do, what does "tail -n 2" do, and what should their combination produce?)

Follow the given steps as closely as possible to avoid synchronization problems:

1. Create a pipe.
2. Fork to a child process (child1).

3. On the child1 process:
    a. Close the standard output.
    b. Duplicate the write-end of the pipe using **dup** (see man).
    c. Close the file descriptor that was duplicated.
    d. Execute "ls -l".
4. **On the parent process: Close the write end of the pipe.**
5. Fork again to a child process (child2).
6. On the child2 process:
    a. Close the standard input.
    b. Duplicate the read-end of the pipe using **dup**.
    c. Close the file descriptor that was duplicated.
    d. Execute "tail -n 2".
7. **On the parent process: Close the read end of the pipe.**
8. Now wait for the child processes to terminate, in the same order of their execution.

Mandatory Requirements

1. Compile and run the code and make sure it does what it's supposed to do.
2. How does the following affect your program:
    a. Comment out step 4 in your code (i.e. on the parent process:**do not** Close the write end of the pipe). Compile and run your code. (Also: see "man 7 pipe")
    b. Undo the change from the last step. Comment out step 7 in your code. Compile and run your code.
    c. Undo the change from the last step. Comment out step 4 and step 8 in your code. Compile and run your code.

Task 3

Go back to your shell and add support to a single pipe. Your shell must be able now to run commands like: `ls|wc -l` which basically counts the number of files/directories under the current working dir, and `cat < makefile|wc -l>out1` which basically counts the number of lines you have in the makefile and writes it to a file called out1. The most important thing to remember about pipes is that the write-end of the pipe needs to be closed in all processes, otherwise the read-end of the pipe will not receive EOF, unless the main process terminates.

Deliverables

Tasks 1,2 must be completed during the regular lab. Task 3 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 lab session. You must submit source files for task 1, task 2, task 3, and a makefile that compiles them. The source files must be named task1c.c, task2.c, task3.c, makefile1, makefile2, and makefile3.

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.