Lab 5

In this lab, you should perform **task 0, task 1a and task 1b before attending the lab session**. This lab may be done in pairs (members of a pair must be in the same lab group, and should inform the TA so as to be placed in the same breakout room).

Goals

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

> **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 - task1 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, 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 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 appropriately refer to LineParser.c in your makefile. You can find a detailed explanation here.

> Through out the lab pay close attention to the difference between **processes**(things that you run with execvp() after fork()) and **command lines**. Think 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'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, signal(2), and waitpid(2), before attending the "official" lab session.

Task 0b

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`.

Add a signal handler to the [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 process manager that you are going to implement in task2.

- 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'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

Add a shell feature "cd" that allows the user to change the current working directory. Essentially, you need to emulate the "cd" internal 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 1c

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**).

Now that you finished task1, save it aside. You will need it for Lab 6.

Task 2 - Process Manager

In this task we are going to implement an internal shell "process manager" to manage the process we run in our shell (everything you fork). The process manager will provide 4 operations:

- `procs` - prints current processes including sleeping, running, and "freshly" terminated processes.
- `wake <process id>` - wakes up a sleeping process.
- `suspend <process id>` - suspends a running process.
- `kill <process id>` - terminates a running/sleeping process.

> Another useful command that is a bit similar to procs, but offers different functionality is the `history` command. The shell saves the history of shell command lines. The shell's history also allows you to run a command from the history by typing its number, instead of typing the whole command again. It can be useful for example when you need to run valgrind, but you don't remember all the flags. You can run: `history|grep valgrind` and it will print all the commands in the history that have the word valgrind in them.

Task 2a - Process List

In this task will create and print a list of all processes that have been forked by your shell.

Representation

Create a linked list to store running/suspended processes. Each node in the list is a struct process:

```
typedef struct process{
    cmdLine* cmd;                         /* the parsed command line*/
    pid_t pid;                            /* the process id that is running the command*/
    int status;                           /* status of the process: RUNNING/SUSPENDED/TERMINATED */
    struct process *next;                         /* next process in chain */
} process;
```

The field *status* can have one of the following values:

```
#define TERMINATED  -1
#define RUNNING 1
#define SUSPENDED 0
```

Implementation

Implement the following functions that create and print the process list:

- `void addProcess(process** process_list, cmdLine* cmd, pid_t pid);` : Receive a process list (process_list), a command (cmd), and the process id (pid) of the process running the command.

- `void printProcessList(process** process_list);` : print the processes.
- Add support for the command `procs` to the shell which prints processes using printProcessList()in the following format:
  `<index in process list> <process id> <process status> <the command together with its arguments>`

  Example:

```
#> sleep 3  # foreground, takes 3 seconds until we get prompt back
#> procs
PID          Command      STATUS
14952        sleep        Terminated
#>
#> sleep 5& # background, we get prompt back immediately
#> procs
PID          Command      STATUS
14962        sleep        Running
#> # Wait for the process to finish
#>
#> procs
PID          Command      STATUS
14962        sleep        Terminated
```

Task 2b - Updating the Process List

Implement the following to add some functionality to your process list:

- `void freeProcessList(process* process_list);` : free all memory allocated for the process list.
- `void updateProcessList(process **process_list);` : go over the process list, and for each process check if it is done, you can use waitpid with the option WNOHANG. WNOHANG does not block the calling process, the process returns from the call to waitpid immediately. If no process with the given process id exists, then waitpid returns -1.
  In order to learn if a process was stopped (SIGTSTP), resumed (SIGCONT) or terminated (SIGINT), It's highly essential you read and understand how to use waitpid(2) before implementing this function
- `void updateProcessStatus(process* process_list, int pid, int status)` : find the process with the given id in the process_list and change its status to the received status.
- update `void printProcessList(process** process_list);` :
  - Run updateProcessList() at the beginning of the function.

- If a process was "freshly" terminated, delete it after printing it.

Task 2c - Manipulating the Processes

In this task you will manipulate processes, using one of the following commands:

- `suspend <process id>` - suspends a running process. Send SIGTSTP to the respective process. This is similar to typing CTRL-Z in the shell when running the process.
- `kill <process id>` - terminates a running/sleeping process. Send SIGINT to the respective process. This is similar to typing CTRL-C in the shell when running a process.
- `wake <process id>` - wakes up a sleeping process. Send SIGCONT to the respective process. This is similar to typing `fg` in a standard shell, right after typing CTRL-Z.

Use kill(), see man 2 kill, to send the relevant signal to the given process id. Check if kill() succeeded and print an appropriate message. Remember to update the status of the process in the process_list.

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

```
#> ./looper&
#> ./looper&
#> ./looper&
#> procs
PID         Command     STATUS
18170       ./looper    Running
18171       ./looper    Running
18174       ./looper    Running
#> kill 18170
#> Looper handling SIGINT       # Message from the child process

#> suspend 18174
#> Looper handling SIGTSTP      # Message from the child process
procs
PID         Command     STATUS
18170       ./looper    Terminated
18171       ./looper    Running
```

```
18174       ./looper    Suspended
#> wake 18174
#> Looper handling SIGCONT      # Message from the child process

#> wake 18171   # What will happen to the process? (it is already running)
#> Looper handling SIGCONT      # Message from the child process
procs
PID         Command     STATUS
18171       ./looper    Running
18174       ./looper    Running
```

Deliverables:

Task1c and 2a must be completed during the regular lab. Tasks 2b and 2c 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, and task 2 and makefiles that compile it. The source files must be named task1.c, task2.c, makefile1 (for task1), and makefile2 (for task2)

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.