Lab 5

Goals

- Get acquainted with command interpreters ("shell").
- Understand how Unix/Linux fork() works.
- Learn how to read the manual (man).
- Better proficiency with C, addressing non-trivial memory allocation/deallocation issues.

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 in lab 6, so try make your code readable and modular.

Note: this lab may be done in stable pairs from the same lab group (or solo if you wish). Obviously, joint work must still be explicitly declared.

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.
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 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

Write a signal handler that prints the signal that the shell receives with a message saying it was ignored. The signals you need to address are: SIGQUIT, SIGTSTP, SIGCHLD. 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.

Task 0c

Get yourself acquainted with job control, by running the commands as described by the following link: http://linuxg.net/how-to-manage-background-and-foreground-processes/

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

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 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). **Print appropriate error message to stderr if the cd operation fails.**
You will need to propagate the error messages of chdir to stderr.

Task 2 – Job Control

Before you start working on your implementation for Job Control tasks, make sure you read through this link in the reading material. Is shows what job control is and how it is used by shell users.

Each process executed by the shell is has a process group. Only one process group can run in the foreground, the rest run in the background or are suspended. Only the foreground process can receive signals (interrupts) and read from stdin. Managing these process groups and manipulating which of them runs in the background and which runs in the foreground is called job control, see reading material. In this task you will implement a simplified version of job control in the shell.

Another useful command that is a bit similar to jobs, 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 - Representation

We will need to store a list of all running/suspended jobs. To do that we use a linked list, where each node is a struct job:

```
typedef struct job{
    char *cmd;                              /* the whole command line as
typed by the user, including input/output redirection and pipes (lab 6)*/
    int idx;                             /* index of current job (starting
from 1) */
    pid_t pgid;                           /* process group id of the job*/
    int status;                              /* status of the job */
    struct termios *tmodes;                  /* saved terminal modes */
    struct job *next;                     /* next job in chain */
} job;
```

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

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

We provide you an implementation of a job list linked list: JobControl.h and JobControl.c, which includes creating a job List and adding nodes to it, removing nodes from a job list, and removing a job list from memory. However some of the parts are missing. You need to implement the following functions:

- `job* initializeJob(char* cmd);`: Receive a cmd (command line), initialize job fields (to NULLs, 0s etc.), and allocate memory for the new job and its fields: tmodes (will be used to save the shell attributes) and cmd.
- `void freeJob(job* job_to_remove);`: free all memory allocate for the job, include the job struct itself.
- `job* findJobByIndex(job * job_list, int idx);`: Receive a job list and an index, and return the job that has the given index.
- `void updateJobList(job **job_list, int remove_done_jobs);` This function is used to update the status of jobs running in the background to DONE. Go over all running jobs, and update their status, by waiting for any process in the process group with the option WNOHANG (using waitpid). if remove_done_jobs is set to 1

(TRUE) then DONE jobs are printed in the same format as in printJobs and are then removed from the job list. WNOHANG does not block the calling process, the process returns from the call to waitpid immediately. If there are no process with the given process group id, then waitpid returns -1.

Feel free to change signatures if needed.

Task 2b - Adding Jobs

In this task, we will support adding jobs to the list, and printing them:

- First, add any new job to the job list and set its status to `RUNNING`, whenever a user enters a new command to run a program.
- Now, implement the `jobs` shell command: whenever the user enters this command, call the printJobs function. Now test this as mentioned below.
- Finally, add code that frees the job list, before exiting the shell.

Task 2c - Initialization

To keep your shell running at all times, we are going to change the handling of signals in the shell, and set back the handlers to default in the child . In the shell, you'll use your signal handler from task 0, instead of the default handler. To be able to move jobs from running in the foreground to running in the background and vise versa, we need to set the process group id of each process, follow the steps under mandatory requirements.

Steps:

- Shell initialization: **At the beginning of the program** (at the beginning of the main!)
  - Ignore the following signals: SIGTTIN, SIGTTOU, SIGTSTP, so they can reach the foreground child process rather than the shell.
  - Use your signal handler from task0b to handle the following signals: SIGQUIT, SIGCHLD (We're not including SIGINT here so you can kill the shell with ^C if there's a bug somewhere)
  - Set the process group of the shell to its process id (getpid).
  - Save default terminal attributes to restore them back when a process running in the foreground ends.
- New processes: After each fork
  - In the child: Set the signal handlers back to default.

- In both the child and the parent (to avoid a racing condition): Set the process group id of the new process to be the same as the child process id, and save the group id in the job.

Mandatory Requirements

- Signals: To set signals back to default use the command signal, see signal(2).
  - Relevant signal handler: SIG_DFL, SIG_IGN, and your signal handler from task0b.
- Process groups: use the following commands to set and get process groups: setpgid(2), getpgid (2).
- Use tcgetattr with STDIN_FILENO as fd, to save the shell terminal attributes.

Test your code using the following scenario, in your shell:

```
$>test1&
Start of test1
$>test2&
Start of test2
$>test3&
Start of test3
$>jobs
[1]     Running         ./test1&
[2]     Running         ./test2&
[3]     Running         ./test3&
$>End of test3
End of test2
End of test1
quit
```

The order of printing will not necessarily be as mentioned above.
Download test1, test2, test3, and use the above scenario to test your code. Each of the files prints a start message, sleeps for awhile (30, 20, 10 seconds respectively), and then prints an end message. Obviously, you need to give these executable files execute permission before you try to run them…

Task 2d - Run in the foreground

Running jobs in the foreground is done in 2 cases:

1. If a job is run without & (non-blocking).
2. If the command fg is run.

Add support to running jobs in the foreground and to the `fg <job_number>` command that receives a job index, and runs the job in the foreground. Use findJobByIndex to get the job with the given index, and implement a function called runJobInForeground:

```
void runJobInForeground (job** job_list, job *j, int cont, struct termios* shell_tmodes, pid_t shell_pgid);
```

- Receive the job list, a pointer to the job with the given index, the shell process group, and the shell's saved attributes, see task2c.
- Check if the job is done, by running waitpid and checking its return status. waitpid in this case must **not block** (see WNOHANG in the man page). It should return immediately. If it fails (returns -1) then there are no processes with the given process group id. If that is the case, then it prints a Done message in the same format as in printJobs and remove the job from the job list.
- If the job has not finished yet, then put it in the foreground using: `tcsetpgrp (STDIN_FILENO, <job pgid>);`.
- if cont is 1 and the job's status was SUSPENDED, then set the attributes of the terminal to that of the job's using: `tcsetattr (STDIN_FILENO, TCSADRAIN, <job tmodes>);`.
- Use kill, see man 2 kill, to send SIGCONT signal to the process group of the job.
- Wait for the job to change status using waitpid (need to block). Change the status of the job to SUSPENDED if the process group receives: a SIGTSTP (ctrl-z) - see WUNTRACED and WIFSTOPPED in the man page of waitpid.
  If it receives a SIGINT (ctrl-c), change the job status to DONE.

  Now the child process is running in the foreground and the shell is waiting for it to complete (or be stopped). Once the shell returns from `waitpid`:

- Put the shell back in the foreground.
- Save the terminal attributes in the job tmodes.
- Restore the shell’s terminal attributes using the shell tmodes, which were saved during initialization. This is done to prevent leaving the shell in an unstable mode. For example if one of the jobs changes the tmodes of the terminal (the text reader used by `man` for examples does this).
- Check for status update of jobs that are running in the background using your updateJobList() function.

You need to call this function, when the fg command is used and when a command is executed in the foreground (in the function `execute`), in a blocking mode (if a command is run without &, instead of the parent waiting for the job to end).
Hint: You'll probably need to refactor `execute` a bit for this to work.

Example:

```
$> test1
Start of test1
^Z
$> jobs
[1]     Suspended               test1
$> fg 1
End of test1
[1]     Done                test1
$>quit
```

Task 2e - Run in the background

Add support to running jobs in the background and to the `bg <job_number>` command that receives a job index, and sends a `SIGCONT` to the process group.

Steps

- Implement a function called runJobInBackground that receives a job (that has the given index using findJobByIndex), sets its status to RUNNING, and sends it a SIGCONT. `void runJobInBackground (job *j, int cont);`
- Use kill, see man 2 kill, to send SIGCONT signal to the process group of the job.

You need to call this function, when the bg command is used and when a command is executed in the background (in the function `execute`), in a non-blocking mode (if a command is run with &).

Example

```
$> test1
Start of test1
^Z
$> test2
Start of test2
^Z
$> test3
Start of test3
```

```
^Z
$> jobs
[1]     Suspended               test1
[2]     Suspended               test2
[3]     Suspended               test3
$> fg 3
End of test3
[3]    Done               test3
$> jobs
[1]     Suspended               test1
[2]     Suspended               test2
$> bg 2
$> End of test2
fg 1
End of test1
[1]     Done                test1
$>jobs
[2]     Done                test2
$>jobs
$> quit
```

Deliverables:

Tasks1, 2a, 2b, and 2c must be completed during the regular lab. Tasks 2d and 2e 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 day.
You must submit source files for task 1, and task 2 and a makefiles that compile it. The source files must be named task1.c, task2.c, JobControl.c, JobControl.h, makefile1, makefile2