CS347M: Programming Assignment 1 - Command Line Interface

This assignment must be done in teams of two. Before you begin, make sure you are familiar with the fork system call, exec system call (and it's variants), file descriptors (opening, reading and writing files using **`open`**, **`read`**, **`write`** and **`close`** system calls). Make sure you have a Linux system. Those using Windows should try to get a Linux virtual machine setup or use a lab PC.

In this assignment you will code a program that will use process control API such as fork, exec and wait to build a command line interpreter (or a lightweight shell). It should repeatedly take commands as input from the user via the terminal and execute each command.

- The shell program, when invoked should display a command prompt as "**`myshell>`** " and then wait for user input.
- When an input is given, it should first check whether its a legal (one of the commands given below) command. If not, print a message "Illegal command or arguments" and go back to display the prompt and wait for user input. Do not exit your program.
- If the command is legal and has the right number of arguments, execute it using one of the two strategies shown below, wait until the command is finished (using the **`wait`** system call in the parent), and then print the command prompt and wait for the next input.

Each command can be coded to run in one of two ways:

a. To run the commands that are already present as executable files, you have to fork a new child process and use **`execve`** to execute the corresponding executable, while the main process waits for the child to finish. When the child exits, the wait call returns to the parent which will then print the command prompt again.
b. For commands which don't have an executable already, you must code these commands as the child process. We will indicate for each command whether there is an executable you must use or you must code it.

Boilerplate code that contain the code for reading user input and tokenizing the string has been provided in the file **`shell.c`**

1. Implement a function **`checkcpupercentage <pid>:`** This command will accept a process identifier as an argument and return the percentage CPU used by the process with identifier **`<pid>`** in user mode and system mode. Details regarding the CPU usage of every process is stored in a file **`/proc/[pid]/stat`** where [pid] is the process identifier of the process. Look up **`/proc/[pid]/stat`** in the man page for **`proc`** for

more information (**`man proc`**). The details regarding the CPU usage in general is stored in /proc/stat. Look up /proc/stat in the man page for proc for more information.
Example usage:

```
myshell> checkcpupercentage 12810
User Mode CPU Percentage: 38%
System Mode CPU Percentage: 11%
```

To implement this:

a. First, fork a new child process.
b. The child process should read the stat file for the specified pid at **`/proc/[pid]/stat`**. The 14th and the 15th word in this file should denote the time (in number of clocks ticks) spent in user mode and the time (in number of clocks ticks) spent in system mode respectively. Let's call these values **`utime_before`** and **`stime_before`**.
c. Now read the file /proc/stat/ to find out the total CPU time used so far. This is the sum of all numbers that follows "CPU" (note: CPU, not CPU1 etc.). Let's call this **`total_time_before`**.
d. Sleep for one second (using the **`sleep`** system call) and read **`/proc/[pid]/stat`** and get the same values again. Let's call these values **`utime_after`** and **`stime_after`**
e. Now read the file **`/proc/stat/`** to find out the total CPU time used so far. Let's call this **`total_time_after`**.
f. To find the user time:
```
ut = 100*(utime_after - utime_before)
     (total_time_after - total_time_before);
```
g. To find the system time:
```
st = 100*(stime_after - stime_before)
     (time_total_after - time_total_before);
```
h. Print out **`ut`** and **`st`** to the terminal with labels **`User Mode CPU Percentage:`** and **`System Mode CPU Percentage:`** respectively.
i. Exit the child, so that the parent who is waiting will be released.

2. Many bash commands are executable programs stored in /bin or /usr/bin. For example, the popular UNIX command **`ps`** that is used to list processes running in the system is one such executable. Implement a function **`checkresidentmemory <pid>:`** This command should accept a pid of a process and output the resident set size which is the physical memory that the task has used *(only that and no headers)*. Look up the manpage of **`ps`** and find out how to obtain the resident set size, without showing the header.

Example usage:

```
myshell> checkresidentmemory 13410
538290
```

To do this:

a. The main process should fork a child and do a **`wait`** on it. Once this call returns, print the command prompt and wait for user input.
b. The child process should use **`execve`** on the **`ps`** command with the appropriate flags to obtain the necessary output for the given **`pid`**. Note that **`execve`** will print the results to the terminal by itself.
c. In **`execve`**, the first argument is a string which points to the path to the executable (which here is just **`ps`** since ps is in your $PATH[1])
d. The second argument is a char** which is an array of strings that are to be given as arguments to the executable.
e. However, note that for most shell commands, the first argument is treated as the name with which the executable was called. The code to do **`ps -u user`** would look something like:

```
char *eargs[] = {"ps", "-u", "user"};
execve("ps", eargs, NULL);
```

3. Recall the **`ls`** is a bash command that lists all the files and folders in the given path. Using the **`ls`** executable, implement a command **`listFiles`** on your CLI which will save all the files and folders, in the current working directory to a file named files.txt. If the file exists already, overwrite its contents.

Examples:

```
myshell> listFiles
```

This should make a files.txt in the same directory, and should contain something similar to:

```
Desktop Downloads Documents Music Photos Videos
```

To do this:

a. Save the file descriptor of STDOUT using **`dup2`**. Read about dup from **`man dup2`.**
b. Now close STDOUT.
c. Open the file your have to write to. It will use the lowest available fd number, which is 1. Now, all the output meant for STDOUT will be written to this file.
d. Execute the required command.
e. Once you're done, you need to restore the file descriptor 0 to STDOUT. To do this, first flush stdout by using **`fflush(stdout);`**

[1]PATH is an environmental variable in Linux and other Unix-like operating systems that tells the shell which directories to search for executable files in response to commands issued by a user. Since the directory /usr/bin is in the $PATH for most Linux systems, you just need to mention the commands and not the entire path to it.

   f. Then close the new file descriptor.
   g. Finally, use dup to restore STDOUT to FD1 by using the variable you saved in step a. Dup in the first step would have created a duplicate of the STDOUT file descriptor, which can now be restored to FD1 since it has been freed. Recall that dup uses the lowest available FD number, which is guaranteed to be FD 1 here.

4. The shell utility **`sort`** sorts the input alphabetically, line by line, given to it from STDIN and outputs it to STDOUT. Add a **`sortFile`** command in your shell using **`sort`**, but this should accept input from a file. So instead of giving input for sort from STDIN, you should be able to write several lines into a file and then type **`sortFile [file]`** to get the file's contents sorted line by line printed into STDOUT (**`sort`** prints to STDOUT by default so you don't have to change anything regarding the output). For this do what you did for the previous task, but instead of closing and replacing STDOUT with a file, do it for STDIN.

Examples:
```
myshell> sortFile aFileThatHasBCA.txt
A
B
C
```

5. Your shell should be able to execute two commands simultaneously by spawning two child process while the parent shell process waits for both of them to complete. Implement this by an operator **`;`** which when called as **`command1 ; command2`** should execute both commands in parallel. The shell should only display the prompt for the next command once both of the commands are done executing. You can assume that there won't be more than two commands executed in parallel. To do this:
   a. Fork two processes and execute the required commands in both of them. Save the pid of both the processes.
   b. The parent should call a wait on both the pids. Lookup the syntax of 'waitpid' to implement this.

Example:
```
myshell> checkresidentmemory 13410 ; listFiles
Desktop Downloads Documents Music Photos Videos
325235
myshell>
```
6. You should be able to exit the CLI by typing in a command named **`exit`**.

In all of the commands, make sure to reap the dead child (using **`wait`**) processes and avoid zombies. Make sure that if execution of any commands results in an error, the error message thrown by the program are printed.

Submission instructions:

Write your program in a file called shell.c. Also write a file readme.txt with any special instructions on how to execute your program. Include your names and roll numbers in the readme.txt file. Place these files in a folder with your roll numbers as the name. Eg: 183050011_183050012. Now make an archive of these two files:

```
tar -cvzf 183050011_183050012.tar.gz 183050011_183050012/
```

Upload this .tar.gz file to Moodle. If you are doing the assignment in pairs, only one of you should submit the file.

CS347M Assignment 1 Grading Rubric:

1. **Part 1 - checkcpupercentage: 20 Marks**
    a. Reading /proc/[pid]/stat and parsing the input correctly: 10 marks
    b. Reading /proc/stat and parsing the input correctly: 10 marks
    c. Doing the fork, sleep correctly and ultimately getting the required result: 5 marks
2. **Part 2 - checkresidentmemory: 10 Marks**
    a. Fork, wait and exec done correctly: 5 marks
    b. Execve done with ps command: 5 marks
    c. Header not hidden/wrong parameter printed in ps: -3 marks
3. **Part 3 - ls and output redirection: 25 Marks**
    a. Implementing ls using execve and: 5 marks
    b. Correctly closing the STDOUT file descriptor, opening the file descriptor for the new file and successfully getting the output in a file - 20 marks.
    c. STDOUT not reopened properly/any other file descriptors left hanging: -5 marks
4. **Part 4 - sort and input redirection: 25 Marks**
    a. Implementing sort using exec: 5 marks
    b. Correctly closing STDIN, opening the file descriptor for the new file and successfully getting the sorted output after reading the input from a file - 20 marks
    c. STDIN not reopened properly/any other file descriptors left hanging: -5 marks
5. **Part 5 - parallel execution: 10 Marks**
    a. Forking two processes and executing them: 5 marks
    b. Properly waiting for children: 5 marks
    c. Shell displaying prompt incorrectly/before the processes finished: -5 marks
6. **General execution of the shell: 10 marks.**
    a. Includes the shell recognizing the different commands/operands and executing them correctly without *any* unexpected outputs or errors.
    b. Prompt being proper.