Create a Shell

DESCRIPTION

Shell

The shell is a command interpreter that provides Linux/Unix users with an interface to the underlying operating system. It interprets user commands and executes them as independent processes. The shell also allows users to code an interpretive script using simple programming constructs such as if, while and for. With shell scripting, users can coordinate a scenario of their jobs. The behavior of shell simply repeats:

1. Displaying a prompt to indicate that it is ready to accept a next command from its user,
2. Reading a line of keyboard input as a command, and
3. Spawning and having a new process execute the user command.
4.

The prompt symbols frequently used include for example:

```
cs3230%
```

like a name followed by a % symbol, or simply

```
$
```

How does the shell execute a user command?

The mechanism follows the steps given below:

1. The shell locates an executable file whose name is specified in the first string given from a keyboard input.
2. It creates a child process by duplicating itself.
3. The duplicated shell overloads its process image with the executable file.
4. The overloaded process receives all the remaining strings given from a keyboard input, and starts a command execution.

For instance, assume that your current working directory includes the a.out executable file and you have typed the following line input from your keyboard.

```
cs3230% a.out a b c
```

This means that the shell duplicates itself and has this duplicated process execute a.out that receives a, b, and c as its arguments.

Shell Built-in Commands

The shell has some built-in commands that rather changes its current status than executes a user program. (Note that user programs are distinguished as external commands from the shell built-in commands.) For instance,

```
cs3230% cd public_html
```

changes the shell's current working directory to public_html. Thus, cd is one of the shell built-in command.

Command Delimiters

The shell can receive two or more commands at a time from a keyboard input. The symbols ';' and '&' are used as a delimiter specifying the end of each single command. If a command is delimited by ';', the shell spawns a new process to execute this command, waits for the process to be terminated, and thereafter continues to interpret the next command. If a command is delimited by '&', the shell goes forward and interprets the next command without waiting for the completion of the current command.

```
cs3230% who & ls & date
```

executes who, ls, and date concurrently. Note that, if the last command does not have a delimiter, the shell assumes that it is implicitly delimited by ';'. In the above example, the shell waits for the completion of date.

Taking everything in consideration, the more specific behavior of the shell is therefore:

1. Displaying a prompt to show that it is ready to accept a new line input from the keyboard,
2. Reading a keyboard input,
3. Repeating the following interpretation till reaching the end of input line:
    i. Execute a command if it is a shell built-in program, otherwise
    ii. Spawning a new process and having it execute this external command.
    iii. Waiting for the process to be terminated if the command is delimited by ';'.

GOALS

1. Create your own shell using c language and system calls
    a. fork()
    b. execvp() or execlp()
    c. wait()
    d. exit()
    e. pipe() and dup()/dup2()

TASKS

In this lab, **you will be given an incomplete C program "main.c"** and you need to complete it to create a simple shell. The shell takes an input command, and then executes it. Your shell should implement the following functions:

Task1: A new prompt: Your shell should start from a prompt, "CS3230-Your name>>"
Task2: A single external command: Your shell should be able to interpret a single external command, such as ls, emacs, and so on. *–This is already implemented in the main.c file*.

Task3: A single built-in command: Your shell should have at least one built-in command other than "cd", "help", "exit" in the given code, mian.c. Some reference built-in commands that you can reimplement include

a. "about" command to print out the program information. From example, you can print out the programmer's name, create time, program's commands, …;
b. "history" command to print out the list of command you used;
c. "cat filename" to print out the content of the file;

You are not limited to these options.

Task4: "&" connector: Use "&" connector to implement multiple commands in one line. The basic requirement is to implement one "&" to connect two commands in a line and execute the two commands at the same time. *You receive 5 extra points if "&" can be used limitless times to chain commands in one line.*

Task5: ";" connector: Use ";" to terminate a process and continue a new one. The basic requirement is to implement one ";" to connect two commands in a line and execute the first command and then the second one in a sequence. *You receive 5 extra points if ";" can be used limitless times to chain commands in one line.*

Task6: "|" connector: Use "|" connector to implement pipeline commands. The basic requirement is to implement one "|" to connect two commands in a line and execute the two commands as parent and child with a one-directional pipe to input from child to parent as we explained in class. *You receive 5 extra points if "|" can be used limitless times to chain commands in one line.*

Task7: Input requirements:

a. The input to the shell is a sequence of lines. The shell must correctly handle lines of up to 100 characters. If a line containing more than 100 characters is submitted to the shell, it should print some kind of error message and truncate it to 100 characters.
b. Each line consists of tokens. Tokens are separated by one or more spaces. A line may contain as many tokens as can fit into 100 characters.
c. Words consist of the characters **A–Z**, **a–z**, **0–9**, **dash**, **dot**, **forward slash**, and **underscore**. If a word in a line of input to the shell contains any character not in this set, then the shell should print an error message and then ignore the rest.
d. Lines of input are divided into token groups. Each token group will result in the shell forking a new process and then executing a program.
e. Every token group must begin with a word that is called the command. The words immediately following a command are called arguments and each argument belongs to the command it most closely follows. The order of arguments matters, and they are numbered from left to right, starting at one.

REFERENCES:

You can refer to an online tutorial below. You can start with this online tutorial.

https://brennan.io/2015/01/16/write-a-shell-in-c/

SAMPLE OUT

```
[02/22/24]seed@VM:.../cs3230$ gcc main1.c -o myshell
[02/22/24]seed@VM:.../cs3230$ myshell
CS3230>> ls & cat test.c
a.out          mymulti.c  pip.c       retlib     test1.c              test.txt
c.c            myshell    pip_dup1    retlib.c   test2.c
cs3230hello.c  myshell.c  pip_dup1.c  runcode    test3.c
main1.c        myShell.c  pip_dup.c   runCode.c  test4.c
main2.c        pip1.c     pipe0.c     test       test.c
multihello.c   pip2.c     reader.c    test1      test-reverseShell.c
#include<stdio.h>

int main(int argc, char *argv[])
{

    printf("This test.c called by execvp() to receive message : %s", argv[1]);

    return 0;
}
CS3230>>
```

```
CS3230>> ls
a.out          mymulti.c  pip.c       retlib     test1.c              test.txt
c.c            myshell    pip_dup1    retlib.c   test2.c
cs3230hello.c  myshell.c  pip_dup1.c  runcode    test3.c
main1.c        myShell.c  pip_dup.c   runCode.c  test4.c
main2.c        pip1.c     pipe0.c     test       test.c
multihello.c   pip2.c     reader.c    test1      test-reverseShell.c
CS3230>> ls ; cat test.c
a.out          mymulti.c  pip.c       retlib     test1.c              test.txt
c.c            myshell    pip_dup1    retlib.c   test2.c
cs3230hello.c  myshell.c  pip_dup1.c  runcode    test3.c
main1.c        myShell.c  pip_dup.c   runCode.c  test4.c
main2.c        pip1.c     pipe0.c     test       test.c
multihello.c   pip2.c     reader.c    test1      test-reverseShell.c
#include<stdio.h>

int main(int argc, char *argv[])
{

    printf("This test.c called by execvp() to receive message : %s", argv[1]);

    return 0;
}
CS3230>>
```

```
CS3230>> who | wc -l
1
CS3230>> cat test.c | wc -l
9
CS3230>>
```

SUBMISSION

1. Your complete source code.

2. A program report: Please complete the template by filling in the blanks. If any blank is not applicable to the assignment, you may leave it empty, but no more than three blanks should be left unanswered. Your report should include at least screenshots, the source code, and an explanation of your program.

Rubric

Criteria	Ratings			Pts
Execute: Compile and execute correctly with no errors	**10 pts** Correct	**0 pts** Not correct or no submission		10 pts
Task1:Prompt Prompt is changed to the required format	**10 pts** Correct	**0 pts** Not correct or no submission		10 pts
Task3: shell built-in command At least one is correctly implemented	**10 pts** One or more built-in program are implemented correctly.	**0 pts** Not correct or no submission		10 pts
Task4:"&" It is correctly implemented	**15 pts** More than one "&" is implemented in the command line	**10 pts** Correct	**0 pts** Not correct or no submission	10 pts
Task5:";" It is correctly implemented	**15 pts** More than one ";" is implemented in the command line	**10 pts** Correct	**0 pts** Not correct or no submission	10 pts
Task6:"|" It is correctly implemented	**25 pts** More than one "|" is implemented in the command line	**20 pts** Correct	**0 pts** Not correct or no submission	20 pts

Criteria	Ratings			Pts
Task7:input 1. Length, 100, is checked 2. Character set is checked	**10 pts** 1.and 2. are checked or more error handlings are implemented.	**5 pts** 1. or 2. is checked	**0 pts** No error handling for input	10 pts
Report: Fill all blanks with detailed and reasonable explanation	**20 pts** Correct	**0 pts** Not correct or no submission		20 pts
Total Points: 100 + 15				

CONGRATULATIONS, YOU'VE COMPLETED PROGRAM 2!