Project 1: The Shell: Process Creation, Pipes, and I/O Redirection

1. Background

As we've discussed in class, the OS command interpreter is the program that people interact with to launch and control other programs. On UNIX systems, the command interpreter is usually called the **shell**: it is a user-level program that gives people a command-line interface for launching, suspending, and killing other programs. sh, ksh, csh, tcsh, bash, etc. are all examples of UNIX shells.

Every shell is structured around a loop like the one below:

```
1. print out a prompt
2. read a line of input from the user (the next user command)
3. parse the line into the program name and an array of parameters
4. fork() the child process
4.1. the child process uses execv() to launch the specified program
4.2. the parent process (the shell) uses waitpid() to wait for the child's termination
5. once the child finishes, the shell repeats the loop by jumping to 1.
```

Although most of the commands people type on the prompt are the name of other UNIX programs (such as ls or more), shells recognize some special commands (called **internal commands**) which are not program names. For example, "**exit**" command terminates the shell, and the "**cd**" command changes the current working directory. Shells directly make system calls to execute these commands, instead of forking a child process to handle them.

2. Assignment

Implement a shell under Linux using C that has the following features:

1. It should parse the parameters and pass in the parameters to the exec'ed program.
2. It should recognize three internal commands: **exit**, **cd** and **time**. **exit** terminates the shell, i.e., the shell calls the **exit()** system call or returns from main. **cd** uses the **chdir()** system call to change to the new directory. **time** prints the current time, that would be done by first calling **time()** system call, which returns the current time as an integer, and then using **ctime()** to format this integer time as a string and printing it out.
3. It should do input/output redirection.
4. Must handle commands connected by pipes.

2.1 Basic Shell Functionality

The basic shell functionality is to read in the user's command, parse it and launch the appropriate command using fork + execv. Assume that full names of the programs, such as "/bin/ls", "/bin/cat" are given. Also, try to use the same prompt as given below; the output produced by your shell should look like the following:

```
CS340Shell% /bin/ls -l
  drwxrwxrwx 1 cakinlar cakinlar  4096 Mar  7 21:28 dir1
  -rwxrwxrwx 1 cakinlar cakinlar 16888 Mar  7 20:59 shell
  -rwxrwxrwx 1 cakinlar cakinlar  2986 Mar  7 21:20 shell.c
  -rwxrwxrwx 1 cakinlar cakinlar   527 Mar  7 21:28 test.txt

CS340Shell % /bin/wc -l shell.c test.txt
    96 shell.c
    19 test.txt
   115 total
```

To allow users pass command line arguments to programs you will have to parse the command line into words separated by whitespace (**spaces and tab characters**) and place these words into an array of strings. Then you'll need to pass the name of the command as well as the entire list of tokenized strings to one of the other variants of exec, such as **execv()**. These tokenized strings will then end up as the argv[] argument to the main() function of the new program executed by the child process. Look at the manual page for execv for more details. You may assume that the command string will always be correct. Further assume that each token in the command string is separated by a space character.

2.2 Shell Internal Commands

Your shell must recognize three internal commands: **exit**, **cd** and **time**. **exit** terminates the shell, i.e., the shell calls the exit() system call or returns from main. **cd** uses the chdir() system call to change to the new directory. **time** prints the current time. Printing the current time can be done by first calling time() system call, which returns the current time as an integer, and then using ctime() to format this integer time as a string and printing it out.

2.3 I/O Redirection

Your shell must also handle I/O redirection. Specifically, users should be able to have the standard input of the last command come from a file (input redirection) or send the standard output of the last command to a file (output redirection).

```
CS340Shell% /bin/wc < shell.c
  96  301 2986

CS340Shell% /bin/pwd > dir.txt

CS340Shell% /bin/wc < shell.c > out.txt
```

I/O redirection will be specified by **<** and **>** symbols as shown above. In the first example, we ask the shell to make "**wc**" read its standard input from "Project.tex". This can be achieved as follows: When the shell forks a child, it should open "Project.tex" using open() system call and redirect the **standard input** descriptor (descriptor 0) to the opened file descriptor using **dup2()** system call. The shell can then exec /bin/wc. This would make "wc" read Project.tex through its standard input descriptor.

In the second example, we ask the shell to make the standard output of "/bin/pwd" to go to file "dir.txt". Similar to input redirection, the shell will first fork a child, open file "dir.txt" and redirect the **standard output** descriptor (descriptor 1) of the child process to the opened file descriptor using dup2(). The shell can then exec "/bin/pwd". This would make the standard output of "/bin/pwd" to go to file "dir.txt".

Notice that it is possible to apply input and output redirection to the same command as shown in the third example.

2.4 Piped Commands

Recall from our discussion in class that **pipes are unidirectional communication links** used to have **2 related processes** communicate with each other. Using pipes it is possible to connect 2 commands together and have the output of the first command go to the second command. Consider the following example:

```
CS340Shell% /bin/cat shell.c | /bin/wc
  96     301    2986
```

Here we launch 2 processes: /bin/cat and /bin/wc, and connect them using a pipe (i.e., using the "**|**" symbol). The intention here is to have the standard output of the first command (/bin/cat) to go to the standard input of the second command (/bin/wc). To implement this functionality, the shell first creates a pipe using the **pipe()** system call. This would return two descriptors, the first one pointing to the **read-end** of the pipe, and the second one pointing to the **write-end** of the pipe. The shell then forks the first child and connects this child's **standard output descriptor (descriptor 1)** to point to the write-end of the pipe using dup2() system call. The shell then execs /bin/cat. Now the standard output of /bin/cat would go to the pipe rather than the screen. Next, the shell forks the second child and connects the child's **standard input (descriptor 0) to the pipe's read-end**. The shell then execs /bin/wc. Now the standard input of /bin/wc will come from the pipe, which is fed by the standard input of /bin/cat.

There may be multiple pipes within the command line as shown in the following example:

```
CS340Shell% /bin/cat shell.c | /bin/wc -l | /bin/wc
   1       1       3
```

In this case the shell must create 2 pipes and fork 3 children. The first pipe will connect the first 2 processes, and the second pipe will connect the last two processes. There could be any number of pipes in the command string.

3. Testing

Here is a list of commands I will use to test your shell. This is **test.txt** also posted in Brightspace.

```
time
/bin/ls
/bin/ls -l
/bin/wc shell.c test.txt
cd dir1
/bin/ls -a -l > out.txt
cd ..
/bin/wc < out.txt
/bin/cat < shell.c > out.txt
/bin/wc out.txt
/bin/pwd | /bin/wc
/bin/pwd | /bin/wc | /bin/wc
/bin/cat shell.c | /bin/wc -l -m -c
/bin/cat shell.c | /bin/wc -l -m -c | /bin/wc
/bin/ls -l -a | /bin/sort | /bin/wc -c -m -l
/bin/ls -l | /bin/sort | /bin/wc -l -m | /bin/wc -c -l -m | /bin/wc
/bin/cat shell.c | /bin/wc | /bin/wc | /bin/wc | /bin/wc | /bin/wc | /bin/wc
/bin/cat shell.c | /bin/wc > out.txt
/bin/wc out.txt
exit
```

4. Submission

Submit your application source code via Brightspace. This project is due on Sunday 3/30/2025 11:59pm. I will be testing your application on a **Linux** machine.