Project 5: A Shell

CS/COE 0449 — Introduction to System Software

See Due Date on the CourseWeb

The Basics

For this project you are going to implement a Unix shell program. A shell is simply a program that conveniently allows you to run other programs. Read up on your favorite shell to see what it does.

The requirements for completing this assignment successfully are described below.

The Input

A shell, at its simplest, is a program that reads input from the user and tries to do the commands. The commands for our shell will be of one of two categories: 1.) Internal commands the shell knows how to do itself and 2.) Other programs on the system to execute.

Your shell should read in a line of input using the `fgets()` command. You then need to tokenize it and interpret it according to the features we are supporting described in the section below.

To tokenize the input, use the C standard library function `strtok()`. This function behaves oddly, so make sure you understand the right way to invoke it. Our delimiter set will be all whitespace characters (space, tab, newline).

The Details

Your shell must support the following:

1. The internal shell command "exit" which terminates the shell.
   Example: `exit`
   Concepts: shell commands, exiting the shell
   System calls: `exit()`

2. The internal shell command "`cd`" which changes the present working directory
   Example: `cd private`
   Details: This command takes a relative or absolute path and changes the present working directory to that path
   Concepts: present working directory, absolute and relative paths
   System calls: `chdir()`

3. Any UNIX command, with or without arguments
   Example commands: `ls`, `pico`, `pwd`, `ls -l`, `wc -l`, `ps -a`, `gcc -m32 -o ...`
   Details: Your shell must block until the command completes and, if the return code is abnormal, print out a message to that effect. Argument 0 is the name of the command
   Concepts: Forking a child process, waiting for it to complete, synchronous execution, Command-line parameters
   System calls: `fork()`, `execvp()`, `exit()`, `wait()`

Note: You must check and correctly handle all return values. This means that you need to read the man pages for each function to figure out what the possible return values are, what errors they indicate, and what you must do when you get that error.

Please note that the commands in the list above are examples. Do not hard code support for them in. You should be attempting to run whatever program the user has typed as the first word on the command line.

Requirements and Hints

- The prompt of your shell should be `"USERNAME's shell> "` where `USERNAME` is your Pitt user id. For example, if your user name is `abc123`, when you run the shell, this will be what you see on the console screen:

  ```
  abc123's shell> _
  ```

  where underscore symbol shown above is the blinking cursor.

- You must read in input using `fgets()`. Note that the signature of the function `fgets()` is as follows:

  ```
  char *fgets(char *s, int size, FILE *stream);
  ```

  To use `fgets()` function, you must supply an array of characters for storing user input, the size (usually the size of the array of characters), and the FILE stream. Since we are going to read from keyboard, the `stream` will simply be `stdin`. For example:

  ```
  char buffer[200];
  :
  fgets(buffer, 200, stdin);
  ```

  For `fgets()` function, the newline character will be included and follows by the null-terminated character. For example, if user type `ls` and press Enter, your `buffer[0]` will be `'l'`, `buffer[1]` will be `'s'`, `buffer[2]` will be `'\n'`, and `buffer[3]` will be `'\0'`.

- You must tokenize the input using `strtok()`. The signature of the function `strtok()` is as follows:

  ```
  char *strtok(char *str, const char *delim);
  ```

Note the way `strtok()` works is a little bit strange. Here is a short example of how to use the function `strtok()`:

```
char buffer[] = "gcc -m32 -o hello hello.c";
char *args[20];
:
args[0] = strtok(buffer, " ");
args[1] = strtok(NULL, " ");
args[2] = strtok(NULL, " ");
:
```

The first call to `strtok()`, you must supply the string that you want to tokenize and the delimiter (the above example is a space). But the second time you call to get the next token, you have to supply `NULL` instead of the original string with the same delimiter as shown above. Note that the `strtok()` will return `NULL` if there is no more token.

- You must use the function `chdir()` to change the current directory. The signature of the function `chdir()` is as follows:

```
int chdir(const char *path);
```

For this function, simply supply the relative or absolute path in the form of string as its argument. Make sure you check the return value. If the return value is -1, an internal error number will be set. To display the error message, you can simply use the function `perror` as shown below:

```
if(chdir(path) == -1)
{
    perror("abc123's shell: ");
}
```

which will print `abc123's shell:` and follow by an error message. Note that you have to include `errno.h`.

- You do not need any special environment to run or execute this program (beyond using `thoth.cs.pitt.edu` as always). A shell can run another shell inside of it.

- To make your shell execute an external command, you must use `fork()`, `execvp()` and `wait()` function as shown below:

```
#include <unistd.h>
:
if(fork() == 0)
{
    if(execvp(args[0], args) == -1)
    {
        perror("abc123's shell: ");
        exit(0);
    }
}
```

```
else
{
    int status;
    wait(&status)
}
```

Recall that the `fork()` will create an exact copy of your program (your shell in this case) and start running from where `fork()` is called. The only way to check whether this is your original shell or a copy of your shell is to check the return value of `fork()`. If it is a child process (a copy of your shell), the return value will be 0. If it is the parent process, the return value will be the process id (PID) of the child process. As shown in the code above, if `fork()` return 0, the process will execute `execvp()`, otherwise it is the parent process, it execute `wait()` to wail until the child process is done.

For the function `execvp()`, the first argument is the command and the second argument is an array of strings represent arguments including the command itself. Here is a short program that execute an external command `"ls -l"`:

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
        char *args[10];
        args[0] = "ls";
        args[1] = "-l";
        args[2] = NULL;

        if(fork() == 0)
        {
                if(execvp(args[0], args) == -1)
                {
                        perror("abc123's shell: ");
                        exit(0);
                }
        }
        else
        {
                int status;
                wait(&status);
        }

        return 0;
}
```

Note that an error may occur when you call `execvp()` function. Therefore, you **MUST check its return value. If the return value is -1, simply exit the child process. If you forgot to do this, the child process will keep running and will behave like**

another shell that may not terminate.

- As mention earlier, to print an error message, use the function `perror()`. Read the `man` page about this function.

What to turn in

- The `myshell.c` file
- Any documentation you provide to help us grade your project
- All in a tar.gz file, named with your user id (`USERNAME_project4.tar.gz`)
- Copy your archive to the appropriate directory:

```
/afs/cs.pitt.edu/public/incoming/CS0449/tkosiyat/sec1
```

No late submission will be accepted.