Advance OS: Shell Project

Objectives:

The objective of this assignment is to understand the workings of a command-line interface (CLI) and to obtain a working knowledge of process forking and signalling.

Assignment:

In this assignment we will build a command shell in stages. Our command shell will be a simple implementation of the commonly used command shells such as sh, bash, csh, and tcsh.

The assignment is split-up in tasks to implement separate features of our shell program. The code fragments for these tasks do not need to be separately submitted. Only the final implementation should be submitted for grading. If you are not able to complete the final tasks, you should be able to submit a partial but working shell program.

Warning: Do not wait until the end to submit this assignment. You will not have enough time.

Programming Environment: This assignment assumes you are working in a *nix environment. If you are using a different type of Operating System (Windows), you have different choices:

1. Virtualize a *nix OS. I recommend downloading VirtualBox and installing Ubuntu.
2. Install Ubuntu on Windows through the microsoft store.

Warning: Both of these options take awhile to install, prepare accordingly!

Requirements:

You need to be familair with a UNIX command shell, such as bash, and you must be able to write, compile, and debug C or C++ programs.

Consult the manual pages to understand the following system calls and library functions:

- int chdir(const char *path)
- int execvp(const char *file, char *const argv[])
- void exit(int status)
- pid_t fork(void)
- char *getcwd(char *buf, size_t size)
- char getenv(const char *name)
- void perror(const char *string)
- int setenv(const char *name, const char *value, int overwrite)
- sig_t signal(int sig, sig_t func)
- pid_t wait(int *status)

- pid_t waitpid(pid_t wpid, int *status, int options)

Use the man command to access the manual pages for each command. For example: "man execvp" will bring up the manual page for execvp. You man also use Google.

Note that perror prints an error message when errno is set. It should only be used when a system call or library function call fails and errno is set.

Another useful function we will use is assert to assist us with debugging. For example:

assert(argc > 0);

checks if argc > 0. If the assertion fails, a diagnostic message is displayed. The error message points to the location in your program of the failed assertion. Whenever you have specific assumptions about the state of (your variables in) your program, it is a good idea to add an assertion.

The examples shown in this document are all in C. But it is not required to use C for your assignments and you may use C++ constructs if you wish. Note that the C examples use the following functions that are rarely used in C++:

- char *fgets(char *str, int size, FILE *stream)
- int printf(const char *format, ...)
- int fprintf(FILE *stream, const char *format)

You should consult the manual pages to understand these library functions. The stdin, stdout, and stderr streams are frequently used with these functions.

Task 1 - Adding Built-in Commands

Create a new file shell.c (or shell.cpp if you prefer C++). This will be our main shell program that will execute commands from a command line. Our shell should intepret the following built-in commands:

- cd: changes the current working directory
- pwd: prints the current working directory
- echo: prints a message and the values of environment variables
- exit: terminates the shell
- env: prints the current values of the environment variables
- setenv: sets an environment variable

For example:

```
1> echo hello world
hello world
2> cd test
3> pwd
/home/pwest/test
3> setenv greeting hello
4> echo $greeting $OSTYPE
hello linux
5> env
...
greeting=hello
...
6> exit
```

To implement the first part you will need to write a parser. If you haven't taken a compilers course before I have stubbed out a simple parser in main.c. If you have taken compilers before, please implement the parsing as you see fit. The parsing is not meant to be hard (a simple space deliminter will suffice.)

Add the main routine, the tokenizeer, and interpret functions to your shell.c program. Here are some helpful hints:

To implement $var variables, you will have a token for DOLLAR_SIGN to start the variable. After the the '$', use getenv to obtain the variable value and place it in the argument array. In this way, variables can be used with all commands, for example echo, cd, and setenv.
To implement echo, write a loop that runs over the command arguments to print the string values.
To implement env, you need to access to the global environment variables array. To this end, add the following declaration to your code: extern char **environ; (see manual pages for environ) For example, environ[0] contains the first variable name=value pair, environ[1] contains the second pair, and so on until you hit a terminating NULL string.
To implement setenv, use the setenv function with overwrite=1. You may assume that the assigned value is a single string token (i.e. the first argument is the variable name and the second argument is the value).

After testing your code, you can advance to task 2. You don't need to submit your code at this point. Just make sure it works before you continue with task 2 by testing the example input above (we will test your code with different input).

Task 2 - Adding Processes

In this part of the assignment we will extend our shell program with process forking. The shell takes a command, checks if it is not a built-in command, forks a process, loads the program from the file system, passes the arguments, and executes it. The shell must wait for the child process to terminate and report any errors that might have occurred.

To implement process forking we will use the process.c program as an example. This program takes a command as an argument and executes it by forking a process.

For example:

```
shiloh% gcc -Wall -o process process.c
```

```
shiloh% ./process ls
Doxyfile html/ parser.c process process.c
shiloh% ./process ls process.c
process.c
engelen% ./process ./process ls
Doxyfile html/ parser.c process process.c
```

Implement process forking in your shell.c program. Your shell program must be able to execute built-in commands and run executables from the command line.

For example:

```
1> ls
Doxyfile html/ parser.c process process.c
2> cp parser.c test.c
3> ls
Doxyfile html/ parser.c process process.c test.c
4> abc
execvp() failed: No such file or directory
An error occurred.
5> exit
```

Hint: The execvp function will be helpful here.

Task 3 - Adding Background Processes

Processes can be run on the background by appending a '&' at the end of a command line.
For example:

```
1> xterm &
2>
```

This starts an xterm window and the shell returns immediately to prompt for more input. Implement background processes by checking (and removing) a '&' at the end of the command line. I recommend extending the "next_token" function

Hint: Look up the fork command.

Task 4 - Signal Handling

Our current shell implementation has a significant drawback: when we want to terminate a foreground process by depressing ctrl-C, the shell will quit as well. To prevent the shell from quitting, we need a signal handler to catch SIGINT (ctrl-C). Consult the manual page of signal and the internet to implement this feature.

For example:

```
1> top    (type ctrl-C)
2>
```

When depressing ctrl-C, the shell should return to the prompt.

Task 5 - Extra Credit

You can earn extra credit (5%) for this assignment by implementing a timer that terminates a foreground process after 10 seconds have elapsed and the process has not completed. You should only terminate foreground processes that exceed the alloted time limit. To terminate a process, you can use int kill(pid_t, int sig). If the process completes before the timer expires, the shell should immediately return with the prompt.

How to turn in:

Turn in via your Github. Commit your code to the existing shell directory on your forked github repository.

Due Date: Nov. 8, 2022 2359

Teamwork: No teamwork, you work must be your own.