Project 1

Due April 13, 2020 at 11:59 PM
Early Extra Credit Due April 6, 2020 at 11:59PM
You will be working alone for this project. This specification is subject to change at anytime for additional clarification. For this project, you will be implementing a fairly simple shell called **ashell**. The shell must be able to execute applications, be able to setup pipes between executed applications, redirect standard input/output from/to files for the executed applications, and perform simple commands internally. You must provide a make file (not a cmake file) to compile your shell. Your program must be written in C or C++. You may not use scanf, fscanf, printf, fprintf, cin, cout, stringstream, or their derivatives; you must use read and write system calls for all I/O. You may use the C++ containers. If you have any concerns if a standard library function is allowed, ask before using it. Submit your project on Canvas as tgz file.

The shell must handle commands cd, ls, pwd, ff, and exit internally. Below are descriptions of the commands:

`cd [directory]` – Changes the current working directory to specified directory. If the optional directory is not specified, then the current working directory is changed to the directory specified by the HOME environmental variable.

`ls [directory]` – Lists the files/directories in the directory specified, if no directory is specified, it lists the contents of the current working directory. The order of files listed does not matter, but the type and permissions must precede the file/directory name with one entry per line. Below is an example listing

```
drwxr-xr-x .
drwxr-xr-x ..
-rwxr-xr-x ashell
-rwxr-xr-x main.cpp
```

`pwd` – Prints the current working directory name.

`ff filename [directory]` – Prints the path to all files matching the filename parameter in the directory (or subdirectories) specified by the directory parameter. If no directory is specified the current working directory is used. The order of the files listed does not matter. Below is an example of a listing of `ff main.cpp ECS150`:

```
ECS150/Project1/main.cpp
ECS150/Project2/main.cpp
ECS150/main.cpp
```

`exit` – Exits the shell.

The shell needs to be able to handle up and down arrows to select through the history of commands. Only the most recent 10 commands need to be stored in the history. The up arrow must toggle to the previous command, down arrow to the next command. If the user attempts to navigate beyond the end of the list the audible bell character '\a' (ASCII 0x07) must be output. In

addition, if a user enters a backspace or delete and no characters exist on the command to delete, the audible bell must be output. You will find the noncanmode.c program helpful converting being able to read in a single character at a time. This will also be helpful in discovering the VT100 escape codes for arrows. Remember that characters aren't echoed out in non-canonical mode.

The shell prompt must have the current working directory followed by the percent symbol. If the full current working directory path is longer than 16 characters, then "/…" should be prepended to the last level directory name. Below are examples:

```
/home/cjnitta%
/.../Project01%
```

A working example can be found on the CSIF at /home/cjnitta/ecs150/ashell.

You should avoid using existing source code as a primer that is currently available on the Internet. You **must** specify in your readme file any sources of code that you have viewed to help you complete this project. Any copied code snippets from online sources **must** have the URL source in comments next to its use. All class projects will be submitted to MOSS to determine if pairs of students have excessively collaborated with other pairs. Excessive collaboration, or failure to list external code sources will result in the matter being transferred to Student Judicial Affairs.

Early Extra Credit

Due April 6, 2020 at 11:59PM

You can receive extra credit by starting early on the project. Submit your project that can take in user input, from the command line. It should be able to change directories with cd, output the current directory, and handle exit commands. Up/down arrows should work for this as well. All other commands should output the command followed by each argument separated by only a single space. For example the command:

```
/home/cjnitta% ls  -la   foo
```

Should output:

```
Command: ls -la foo
```

A working example of the prompt only version can be found on the CSIF at /home/cjnitta/ecs150/ashell-prompt-only.

Helpful Resources

System calls will be necessary for this project. You will definitely use the following system calls:

`open()` – Opens a file and potentially creates it if specified. See man page at http://man7.org/linux/man-pages/man2/open.2.html

`close()` – Closes a file descriptor. See man page at http://man7.org/linux/man-pages/man2/close.2.html

`read()` – Reads data from a file descriptor (file, pipe, etc.). See man page at http://man7.org/linux/man-pages/man2/read.2.html

`write()` – Writes data to a file descriptor (file, pipe, etc.). See man page at http://man7.org/linux/man-pages/man2/write.2.html

`pipe()` – Creates a pipe. See man page at http://man7.org/linux/man-pages/man2/pipe.2.html

`dup2()` – Duplicates a file descriptor (note use `dup2` instead of `dup` it is easier). See man page at http://man7.org/linux/man-pages/man2/dup.2.html

`fork()` – Makes a copy of a process. See man page at http://man7.org/linux/man-pages/man2/fork.2.html

`execvp()` – Replaces the calling process with a new program (note `execvp` may be the easiest to use but there are many derivatives of `exec`). See man page at http://man7.org/linux/man-pages/man3/exec.3.html

`wait()` – Waits for a child to terminate. See man page at http://man7.org/linux/man-pages/man2/wait.2.html

`getcwd()` – Gets the current working directory. See man page at http://man7.org/linux/man-pages/man2/getcwd.2.html

`chdir()` – Changes the current working directory. See man page at http://man7.org/linux/man-pages/man2/chdir.2.html

`opendir()` – Opens a directory for reading. See man page at http://man7.org/linux/man-pages/man3/opendir.3.html

`readdir()` – Reads a directory entry in from a directory opened with opendir. See man page at http://man7.org/linux/man-pages/man3/readdir.3.html

`closedir()` – Closes a directory opened with opendir. See man page at http://man7.org/linux/man-pages/man3/opendir.3.html

`stat()` – Gets the information about a file. See man page at http://man7.org/linux/man-pages/man2/stat.2.html

`getenv()` – Gets an environmental variable, helpful for getting HOME directory. See man page at http://man7.org/linux/man-pages/man3/getenv.3.html

You may find `getpid()` http://man7.org/linux/man-pages/man2/getpid.2.html

helpful in determining which process is running what when outputting. During your debugging you will probably want to output the pid of the process with each output.

Overview of simple shell command

1. Print prompt
2. Wait for command input
3. Parse into command and arguments
4. fork the shell
5. Child and parent determine who they are by fork return
6. Parent waits while child executes with call to wait
7. Child calls exec to become the command
8. New exec process runs and finally terminates
9. Parent shell returns to step 1

Internal commands and forking

Some of the internal commands you will not want the child to handle, while others you will. The parent should handle `cd` and `exit`, the reason for this is that if the child handles `cd` the directory will only be changed for the child, and not the parent. Also, if the child handles `exit`, then nothing will have actually happened. The commands that the child should handle are `ls`, `pwd`, and `ff`. The reason that the child should handle these is that the output of them might be redirected or piped to another process. The parent shell output should not be redirected, and therefore the child needs to handle these commands.

Redirecting overview

Redirecting opens a file and redirects either stdin or stdout to it. Here are some things to remember when trying to implement redirecting.

- You will probably want to open the file after forking because the parent will not necessarily need access to it
- You will want to set the mode for the file if you are creating it, failure to do so may not grant read permission to the file afterward.
- dup2 should be used to redirect the file to either stdin or stdout

Piping overview

Piping is similar to redirecting but uses a pipe and multiple children. Piping between two children processes requires multiple calls to fork, creating a pipe, and redirection of stdin/stdout before exec. Here are some things to remember when trying to implement piping.

- Pipe must be created before a fork (remember fork clones, so if pipe is done afterward the child or parent doesn't know the pipe created)
- dup2 should be used to redirect an end of the pipe to either stdin or stdout (each pipe has a read and a write end)
- Don't forget to close the pipe ends that are no longer needed after calling dup2 before calling exec. An open write end can make a process hang expecting that input is still possible.