CS252: Mini Shell (15 points)

Due: Tuesday April 11, classtime

You are to write a C program that will act as a mini Linux shell

**Basic Shell:**

Your shell program should display a prompt, get the input from the user, execute the specified command, and then repeat.

For the prompt I want you to display the current working directory followed by a >, as in (although your directory path will be different): 

/USERS/STAFF/S-Z/S/STEVENDE/cs252>

You may use the getcwd function to get the current working directory. It takes in a character buffer (e.g. array) and the size of the buffer as its two arguments and will fill in this buffer.

To read the line from the user you can use fgets. It also takes in a character buffer and the size of that buffer. As a 3rd argument it takes in the stream it is reading from. For our purposes this can simply be the built-in stream stdin.

In order to do any processing of the input, it will be helpful to parse the string into its individual tokens. Start by using the following code I found on the internet. You pass in your original string buffer and it fills in an array of strings with each of the tokens as a separate slot in the array. It will have a final NULL as the last token. This code should work as given and you can use it directly in your code (string.h needs to be included). Note that it, like a lot of systems style code found on the internet, is poorly written. Check out the section at the bottom of this document on getting extra credit for rewriting it.

```c
// buffer is the input string of characters
// args is the output array of arguments. It has already been created with argsSize slots.
// nargs as the number filled in that is passed back
void parseArgs(char *buffer, char** args, int argsSize, int *nargs) {
    char *bufArgs[argsSize];
    char **cp;
    char *wbuf;
    int i, j;

    wbuf=buffer;
    bufArgs[0]=buffer;
    args[0]=buffer;
```

```c
```
for(cp=bufArgs; (*cp=strsep(&wbuf, "\n\t")) != NULL ;){
    if (((*cp != '0') && (++cp >= &bufArgs[argsSize])))
        break;
}

for (j=i=0; bufArgs[i]!=NULL; i++){
    if(strlen(bufArgs[i])>0)
        args[j++]=bufArgs[i];
}

// Add the NULL as the end argument because we need that for later
*nargs=j;
args[j]=NULL;
}

Now that you have the input broken into tokens, you should do something different for each requested command. There are two basic types of commands. For this section we will focus just on shell specific commands. To start, you should include 2 shell specific commands: exit and cd.

Exit should simply stop your shell program from running.

Cd should do a standard change directory. You will need to use the function chdir to accomplish this task. You simply pass it the string of the directory to which you wish to change. Note that this function will return a non-zero value if it cannot change to that directory and you should inform the user this occurred.

Test your basic shell by changing directories a few times and then exiting.

**Executing Programs:**

We can get a lot more power out of our shell by allowing it to execute programs. This will not only include programs that we write, but also built-in systems ones such as ‘ls –l’ or ‘ps’. To do this, you will need to fork a child process, have the child process perform an execvp, and have the parent process wait for the child to finish.

Forking is accomplished by the use of the fork command, no parameters. It will return the process id (pid) of the child created to the parent and a pid of 0 to the child.

To have a process start executing another programs code you use the execvp function. This takes two arguments. The first is the name of the program to run, such as “ls”. The second should be an array of strings with all the arguments necessary to run it. For example, slot 0 could be “ls” and slot 1 could be “-1”. Note that this type of array is exactly what you got back from the supplied parser.
Waitpid can be used to have the parent wait for the child. This takes 3 parameters. The first is either the pid of the child to wait for or else a -1 indicating it should wait for any child it created. The second should just be a pointer to an int in which it will place a status code. And the last parameter should either be a 0 if we want it to block, or a WNOHANG (stdlib.h) if we wish it to return immediately.

Input/Output Redirection:

For this piece you are to add input and output redirection. It sounds complex but it is really simple. If there is a redirection it will show up as a > or < followed by the name of the file that the output/input should go to/come from. Thus, your first task would be to search for a ‘<’ or a ‘>’ as a token. I highly suggest you make two helper methods to check for output redirection and input redirection. Note that you can have both in the same command. You only need to handle redirection in the case of executing linux programs (i.e. you don’t need to redirect shell specific commands).

If you find the redirection token then the next token is simply the filename. Note that since your shell will be taking care of the redirection, you should remove the token and associated filename from your token array so when the command is executed (execvp) they do not exist in the array.

To redirect use the freopen function. This remaps one stream to another stream. In our case we want to remap either stdin or stdout to the file specified. The first parameter is always the name of the file. The second and third parameters are either “w” and stdout, or “r” and stdin.

Test out your code by redirecting something like ls –l to foo.txt.

Background Processes:

Lastly, you need to add the ability for programs to be run in the background. This will be indicated by a & at the end of the command. It can only occur at the end. You already have the shell setup to handle making processes and waiting for them to finish. This is simply an adjustment on that. The shell should allow the user to execute other commands while the background command is being executed.

You may want to add a signal handler into your code to work with the SIGCHLD signal. This signal occurs when a process dies. You can setup a handler to catch it with the signal function. This function takes two parameters. The first is the signal type. For us, SIGCHLD is probably what you want to use (signal.h). The second is the name of the function to handle that signal.
You will need to find a way to reap zombies. That is your shell is a long running program and cannot simply leave zombies processes in the system. A good way to test to see how your code is doing is to create a simple test program that simply loops or sleeps for a set amount time. Then you can place this program in the background. See if you can run other programs while it is running in the background. And then make sure it doesn’t end up a zombie when it is done executing. Recall that you can always execute the command ‘ps’ through your shell to check on the processes.

Here is a list of things you do not need to worry about. You do not need to inform the user when the background program ends. You do not need to handle suspending (ctrl-z) processes or placing them into the background after the command has already be run (i.e. bg in the built-in shells). You do not need to be able to take a shell that was in the background and put it into the foreground (i.e. fg). You do not need to handle programs that do interactive input/output while in the background (i.e. emacs, vi, vim). You do not need to list all the processes you have placed in the background (i.e. jobs) If you have any other questions as to what you should be able to handle just ask.

**Extra Credit (up to 2 points):**

To get 1 point of extra credit, I want you to completely rewrite the parse function. Even though this code works, it is an example of some of the ugly coding styles you will find on the internet. This will require exploring how this function works and then rewriting the loops, variables, etc so that accomplishes the same tasks but it of acceptable coding standards for this course.

To get 1 point of extra credit, I want you do add an additional shell specific command that will deal with history. You should include a ‘history’ command that will list the last 25 commands that were run. And then a ‘!5’ command that will execute the 5th item from this history list.

**Submission:**

You may work with a partner.

You are to submit a zip with your source code to the W: drive. You are also to turn in a cover sheet detailing your submission.