COSC 3750
Linux Programming, Spring 2023
Homework 09

1 Basic info

This is Step II of writing the shell program. You have to accomplish two things here; validate the input **and execute programs**.

1.1 Validation

The possibility exists that you completed Step I with a program that cannot really be extended. You should be creating structures (of some type) for each command. If you have a better way, or at least one as good, fine, but it had better be extendable. However you put information into those structures, you should have done the checks to make sure that the commands comply with the shell syntax from the Homework 9 document. I have repeated that syntax in this document.

We will not be implementing pipes in this version.

1.2 Execution

At some point in this new version of the program you will have the input validated and have to start executing programs. You will need to implement the part that forks a new process for every program the user wants run. Then using `execvp`(), and you MUST use `execvp()`, you will execute that program. The parent program (shell) will use `waitpid`() to ensure that there are no zombie processes still hanging around. The difficult part here is making sure things are working.

You can see the list of items at the end of this document. The idea is to cause at least one program to be executed for every line of input. You will ignore (quietly) all redirection or stdin, stdout, or stderr. **That does not mean you will not validate it**, just that you will not do any redirection. You will take a command, "marshal" all the arguments into an argument vector, and use **execvp()** to cause the program to be executed.

You will make sure there are no zombies by using some version of `wait()` (not wait3() or wait4()) on the children. The one additional thing you **will** do, is implement the "do it in the background" operation.

$> executable arg1 arg2 arg3 &

This means that as soon as the user presses enter, the shell executes the command and immediately re-displays the prompt. Otherwise (without the &), the shell will NOT redisplay the prompt, or respond to user input, until the current command has completed. Running processes in the background is an easy way to create zombies. Make sure that at some point you wait on all the possible children.

2 What you will need

For this version you will need your program from Homework 9 (possibly with MAJOR revisions). As with the previous program you will need to include "wyscanner.h" in your program, add "wyscanner.c" to the compilation. The current installed version of **gcc** does NOT need any "std" options because it already supports are least through C11.

```
gcc -Wall -ggdb wyshell.c wyscanner.c -o wyshell
```

Please look in *wyscanner.h* for the return values from *parse_line()*. You will also see **extern** declarations of *parse_line*, the buffer *lexeme* which will contain a valid string if the return value is WORD and the character *error_char* which contains the offending error character if the return value is ERROR_CHAR. If the return is QUOTE_ERROR, the contents of *lexeme* and *error_char* are undefined. Note that the defined values for the tokens are such that the error values are less than EOL (end of line) and the valid tokens are all greater than EOL.

A last note. Do not attempt to parse the input on your own. Just use the scanner. It works, and you do not have to (or need to) reinvent the wheel.

3 What you will create

Your job is to write a program named **wyshell.c** that

1. Prints a prompt, "`$>` ". There is one space after the `>`.
2. Reads the lines of user input.
3. THIS VERSION NO LONGER PRINTS ANYTHING BUT ERROR MESSAGES! THERE WILL BE A 10% DEDUCTION IF YOU LEAVE IN ANY DEBUGGING OUTPUT, TESTING OUTPUT, OR OUTPUT FROM THE FIRST VERSION!

4. For each command (and its arguments) your shell will
   (a) fork() a new process
   (b) have the child create an argument vector
   (c) have the child cause the command to be executed using execvp()
   (d) dispose of any zombies by using wait?()
5. After the program execution, print the prompt again, just like the "real" shells.
6. If the command ends with "&", you will run the process(es) in the background. That means the shell will **not** immediately wait on its children and the background process will run "whenever." If the ampersand occurs at the end of a set of commands like

   ```
   grep shell homework7.tex | cut -f1 | wc -l &
   ```

   each process will be run in the background. You do not need to print anything out for this even though **tcsh** does.
7. This version will NOT handle redirection. If there is redirection for the command, just ignore it. The same applies to pipes. However, you will validate the ENTIRE command line before any execution. If there are pipes like in the previous example, ignore the pipe part and just run each command. This will result in strange things happening but do not worry about that.
8. After an ERROR_CHAR return, QUOTE_ERROR return, or detection of a grammatical error that line of text is abandoned (with an appropriate error message) and the next line is processed.
9. After SYSTEM_ERROR, the program exits.
10. Only when the read from `stdin` returns End-Of-File, will the program exit.

4 Turn in

Create a tar archive of all the files needed for your program, including your Makefile, NOT including wyscanner.c or wyscanner.h. Name this archive **hw09.tar** or **hw09.tgz**. Upload the archive to the WyoCourses assignment. DO NOT TAR ANY DIRECTORIES.

5 Syntax

Our command line syntax will be fairly simple. In the following, braces ({}) are used for grouping. The character $\vee$ means "or". Square brackets ([]) are used to enclose optional

items. An asterisk (*) following an item means "0 or more", a plus (+) means "1 or more". None of these are literally used in the input but are only there as elements of the grammar definition.

The symbols with special meaning to the shell are:

```
| ; > >> 2> 2>> 2>&1 < " ' &
```

The grammar is:

command	→	words [words]* [redir ∨ err_redir]* [&] `(see notes 1 and 5)`
redir	→	{input_redir ∨ output_redir ∨ output_append}
err_redir	→	{err_redir ∨ err_append ∨ err_to_output}
pipe `(note 2)`	→	command pipe_char command [pipe_char command]*
input_redir	→	`<` filename
output_redir	→	`>` filename `(see note 3)`
output_append	→	`>>` filename
err_redir	→	`2>` filename
err_append	→	`2>>` filename
err_to_output	→	`2>&1`
commands	→	command [`;` command]*
pipe_char	→	`|`
filename	→	words
words	→	{quoted_word ∨ word}
quoted_word	→	dq {ws* {word ∨ sq}$^+$}* ws* dq
quoted_word	→	sq {ws* {word ∨ dq}$^+$}* ws* sq
word	→	character$^+$
character `(note 4)`	→	`[^><&|;]`
dq	→	`"`
sq	→	`'`
ws	→	`[ \t]`

Notes:

1. There can be at most one output redirection, one error redirection and one input redirection.

2. That means the left side of a pipe must not also redirect output. The right size of a pipe must not also redirect input.

3. The order of redirection is important. The command line

```
command 2>&1 > filename
```

redirects *command*'s standard output to *filename* but its diagnostic output to standard output. On the other hand

```
command > filename 2>&1
```

redirects *command*'s standard output to *filename* and its diagnostic output will also go to *filename*.

4. The character class, indicated by characters enclosed in square brackets, is a short hand for the OR of the enclosed characters. That is `[abcd]` means {a ∨ b ∨ c ∨ d}. A dash between characters indicates a *range*. The previous class could have been written `[a-d]`.

5. The & symbol indicates the preceding command should run "in the background." This means the shell does not wait on the processes but immediately redisplays its prompt and is ready for more user input. The & must not occur except at the end of a command line. That means that

   ```
   command filename & | command arg arg
   ```

   is an error! It must be

   ```
   command1 filename | command2 arg arg &
   ```

   which causes both *command1* and *command2* to be run in the background.

   However, because the semicolon separates commands which are processed independently of each other,

   ```
   command1 filename &; command2 arg arg &; command3
   ```

   is perfectly valid.