Alexandria University
Faculty of Engineering
Comp. & Comm. Engineering
CC373: Operating Systems

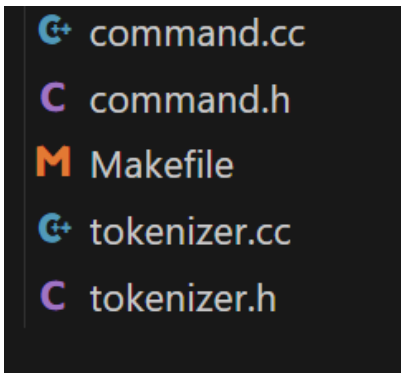
جامعة الاسكندرية
كلية الهندسة
برنامج هندسة الحاسب والاتصالات
مادة نظم التشغيل

Lab3
Mini shell

Objective:

In this lab, you are required to extend a pre-existing **mini shell** program in C++. The base code has been provided to you, which includes structures for the tokenizer, basic command handling, and simple shell functionality. Your task is to **complete** the shell by adding full functionality for **command parsing**, **redirection**, **pipes**, **background processes**, and **error handling**. Which will be in file **command.cc.**

Let's give you an overview about the project architecture before doing your first run.



You will start with the following base code:

1. **Tokenizer:** The `tokenizer.cc` and `tokenizer.h` files contain the implementation of the tokenizer that converts user input into tokens.

2. **Command Parsing:** The `command.cc` and `command.h` files define the basic structure for commands but lack full parsing and execution logic.

3. **Makefile:** A Makefile that should be provided to help you compile the program, and it is set up for a C++ build process (Empty you are going to do it).

Your Task:

You need to implement the following functionality:

1- Command Parsing:

- Complete the `parse()` function in `command.cc` to properly parse the tokens into a `Command` object.
- Handle command arguments, redirections (`>`, `>>`, `<`, `2>`, `>>&`), pipes (`|`), and background execution (`&`).
- Support for compound commands (multiple commands connected by pipes, redirection with multiple operators, etc.).

2- Command Execution:

- Implement the `execute()` function in `command.cc` to handle command execution.
- Use **fork()** and **execvp()** to spawn child processes and execute commands.
- Implement file redirection using **dup2()**.
- Handle pipes by creating a chain of processes, where the output of one command is passed to the input of the next using pipes.
- Implement background processes (`&`), where the shell runs a command in the background without waiting for it to complete.

3- Error Handling:

- Ensure that your shell handles invalid input gracefully and prints appropriate error messages (e.g., for invalid commands, missing files, and malformed redirection or pipes).
- Provide clear error messages for invalid redirection or pipe usage.

Try to understand how the program works. First read the file **command.h** implements the data structure that represents a shell command. The struct ***SimpleCommand*** implements the list of arguments of a simple command. Usually a shell command can be represented by only one ***SimpleCommand***. However, when pipes are used, a command will consist of more than one ***SimpleCommand***. The struct *Command* represents a list of *SimpleCommand* structs. Other fields that the *Command* struct has are ***_outFile***, ***_inputFile***, and ***_errFile*** that represent input, output, and error redirection.

You will have to modify shell.y to implement a more complex grammar

```
cmd [arg]* [ | cmd [arg]* ]* [ [> filename] [< filename] [>> filename] ]* [&]
```

Insert the necessary actions in `parse()` to fill in the *Command* struct. Make sure that the *Command* struct is printed correctly.

Run your program against the following commands:

```
ls -al
ls -al > output.txt
cat < input.txt
echo "Hello, World!" | grep "Hello"
ls -al &
ls | grep "file" > output.txt
```

Second part: Process Creation, Execution, File Redirection, Pipes, and Background

Starting from the command table produced in Part 1, in this part you will execute the simple commands, do the file redirection, piping and if necessary wait for the commands to end.

1. For every simple command create a new process using *fork()* and call *execvp()* to execute the corresponding executable. If the _bakground flag in the Command struct is not set then your shell has to wait for the last simple command to finish using *waitpid()*. Check the manual pages of *fork()*, *execvp()*, and *waitpid()*. Also there is an example file that executes processes and does redirection in cat_grep.cc. After this part is done you have to be able to execute commands like:

```
ls -al

ls -al /etc &
```

2. Now do the file redirection. If any of the input/output/error is different than 0 in the Command struct, then create the files, and use *dup2()* to redirect file descriptors 0, 1, or 2 to the new files. See the example ls_output.cc to see how to do redirection. After this part you have to be able to execute commands like:

```
ls -al > out
ls -al >> out

cat out

cat < out

cat < out > out2
```

3. Now do the pipes. Use the call *pipe()* to create pipes that will interconnect the output of one simple command to the input of the next simple command. use *dup2()* to do the redirection. See the example cat_grep.cc to see how to construct pipes and do redirection. After this part you have to be able to execute commands like:

```
ls -al | grep command

ls -al | grep command | grep command.o

ls -al | grep command

ls -al | grep command | grep command.o > out

cat out
```

Third part: Control-C ,Exit, Change Directory, Process creation log file

1. Your shell has to ignore ctrl-c. When ctrl-c is typed, a signal SIGINT is generated that kills the program.
2. You will also have to implement also an internal command called *exit* that will exit the shell when you type it. Remember that the *exit* command has to be executed by the shell itself without forking another process.

```
myshell> exit

    Good bye!!
```

3. Implement the *cd [dir]* command. This command changes the current directory to *dir*. When *dir* is not specified, the current directory is changed to the home directory.
4. It's required to create a log file that contains Logs when every child is terminated you can use SIGCHLD signal to do so.

Bonus part:

1. Do the wildcarding. The wildcarding will work in the same way that it works in shells like csh. The "*" character matches 0 or more nonspace characters. The "?" character matches one nonspace character. The shell will expand the wildcards to the file names that match the wildcard where each matched file name will be an argument.

```
echo *          // Prints all the files in the current directory

echo *.cc       // Prints all the files in the current

                // director that end with cc

echo c*.cc

echo M*f*

echo /tmp/*     // Prints all the files in the tmp directory

echo /*t*/*

echo /dev/*
```

How to Submit

1. Join the GitHub Classroom:

· You will use the **link** to get in the GitHub Classroom Assignment.
· Click on the link to join the classroom and accept the Assignment.

2. Access Your Repository:

· Once your repository is created, you will be redirected to it. The repository will be named with your **student ID** (e.g., 12345-lab2).
· **Important**: Only you can access this repository to push your commits.

3. Work on Your Lab:

· Download the repository to your local machine and make the required changes as specified in the lab instructions.
· **Make commits regularly** as you complete different parts of the project.

4. Push Your Changes:

· After making a commit locally, push your changes to GitHub.
· **Each commit counts as a submission**. Ensure that each commit includes all the necessary code and files to complete the assignment..

5. Deadline and Late Submissions:

· **Deadline for submission**: The final date and time for submissions will be provided by your instructor.
· After the deadline, **no further commits will be accepted**. Ensure you submit all your work **before the deadline**.
· If you need to make additional changes, **make sure to do so before the deadline** and push the final commit.

6. Final Submission:

· **You don't need to send anything manually**. The repository you've been working on will be used for grading.
· **Make sure your work is final** before the deadline, as no more commits will be allowed after that.