Assignment 1 - Shell

Operating Systems - Fall '23

1 Introduction

In this assignment, you will have the opportunity to design and implement a minimal command-line interpreter, aka. shell that mimics the functionality of a real UNIX shell (e.g. BASH, ZSH etc). The purpose of this assignment is to provide you with hands-on experience in understanding the underlying principles of systems programming and to reinforce your understanding of operating systems concepts. You won't end up making a commercial shell equivalent, but you will have the core functionality down.

2 Objectives

For many people this project will be practice and/or a warm-up. For others, it will be a learning exercise. Regardless of your background, by the end of this assignment, we hope that you will comfortably and confidently be able to do the following:

- Understand the basic concepts of a UNIX shell and how it works.
- Develop clear, readable, well-documented and well-designed programs in the C Programming Language.
- Locate and interpret `man pages` applicable to application-level system programming.
- Understand the basic concepts of process creation and management in a UNIX-like operating system.
- Understand the basic concepts of input/output redirection and pipes.

3 Overview

3.1 Plan

This assignment is divided into three parts

Part	Brief description	Grade weightage	Recommended deadline
Basic	Builtin shell commands and process execution.	[30%]	17/09
Intermediate	Input/output redirection and pipes.	[40%]	24/09
Advanced	Chaining multiple commands	[30%]	01/10

None of the parts are optional but a major chunk of the assignment comprises the Basic and Intermediate sections.

The assignment is to be attempted individually. Plagiarism is strictly prohibited. Students are not allowed to discuss their solutions with others or look over the internet. However, they may seek help from the course staff ONLY.

4 Part 1 : The Basics

Like the shells you use daily (or rarely), yours should issue a prompt (when running interactively), at which it reads input commands from the user and executes them. It should also be able to read these commands one-by-one from a file as well. But what is a command? A simple shell command consists of a command name followed by optional arguments. The general syntax for executing a command is:

```
command [args]*
```

Where the `[args]*`[1] specifies that there can be an arbitrary (≥ 0) number of arguments. Shell supports two types of commands. A command can either be the name of an executable, or a path (either relative or absolute) to the executable, or it can be a shell built-in (which are the functions of the shell itself and modify the state of the shell directly).

Here's some example[2] commands. To list the contents of a directory, you would use the `ls` command:

```
$ ls
Documents  Downloads
```

To copy a file, you would use the `cp` command followed by the source and destination file names:

```
# copies the file source_file to destination_file
$ cp source_file destination_file
```

Your job is to implement a shell program that supports the execution of basic shell commands (including both the built-in as well as the external commands).

4.1 Specification

For this section, the shell should support the following features:

4.1.1 Interactive mode and Script mode

Your shell **must** support a script mode, where the shell reads commands from a file instead of the terminal. The script mode is activated by passing the name of the script file as an argument to the shell executable. For example, if the shell executable is named `shell`, then the script mode can be activated by running `./shell script.txt`. In this case, the shell should read the commands from the file `script.txt` and execute them. If the shell is run without any arguments, it should run in the interactive mode, where it reads commands from the terminal and executes them.

Note that this is very important, as the testing depends on the script mode. If your shell doesn't support the script mode, you won't be able to run the tests.

Side note on implementation

A single function like `getInput()` should be used that depending upon a flag, yields a command string, either from the user's input (in the case of the interactive mode), or from the script file provided as an argument. Have a look at the `int argc` and the `char** argv` arguments of a standard C `main` function.

4.1.2 Built-ins

1. `exit` : The `exit` command allows you to exit the shell gracefully and return to the parent shell or terminate the shell entirely.

[1] This is a syntax I adopted from UNIX man pages, with slight modifications, to precisely specify syntax. Find details in section A

[2] For this example, as well as all the subsequent examples, the `$` sign represents the shell prompt. The `#` sign represents the beginning of a comment. The `\` sign represents the continuation of a command over multiple lines, **and is not a part of the command, unless escaped**. Any line with no symbol at the start is an output

Syntax: `exit`

2. `pwd`: This command is used to display the current directory in which the shell is operating. This command is helpful when you need to know the exact path of the directory you are currently working in. The usage of the `pwd` command is straightforward. Simply enter `pwd` without any options or arguments and the output is the path to the current directory. Please don't add anything extra, you'll fail the tests.

   Syntax: `pwd`

3. `cd`: `cd` stands for change directory, and that's what it does. You specify the directory in the args, and the present working directory (or the current directory) of the shell is changed to that. If `cd` gets no args, it simply changes pwd to the `home` directory, and if there's more than 1 args, it should give an error. Think about why can't this be implemented as an external process.

   Syntax: `cd [directory]`

4. `alias` The `alias` command allows you to create aliases for frequently used commands. An alias is a user-defined shortcut or alternative name for a command. When you create an alias, the shell replaces the alias with the actual command whenever you use it. Use the following syntax to create an alias:

```
alias <alias_name> "<input_string_to_alias>"
```

   You can assume that the alias names won't contain any operators, i.e. *pipeline, IO redirection or chaining operators* (explanation in the following sections), and can only be valid or invalid simple commands with or without arguments. You can also assume that no nesting of aliases will be done, i.e. no alias will contain another alias in its input string. For example, if you frequently use `ls -al` to list all contents of a directory, you can create an alias called `ll` by the command: `alias ll "ls -al"`. After creating this alias, whenever you enter `ll` in the shell, it will be expanded to `ls -al`.

   Syntax: `alias [alias_name [alias_value]]`

   If no arg is provided, `alias` lists all the aliases currently defined in the format `"%s='%s'\n", alias_name, alias_value` (`printf` format specifiers syntax). If only the `alias_name` is provided, list the expansion of that alias in the format : `"%s='%s'\n", alias_name, alias_value`. Please note that following this syntax is important, as this will be followed in the testing. Consider the following example:

```
$ alias ls "ls --color=tty"
$ alias grep "grep --color=tty"
$ alias
ls='ls --color=tty'
grep='grep --color=tty'
$ alias ls
ls='ls --color=tty'
$ alias grep
grep='grep --color=tty'
```

5. `unalias`: The `unalias` command allows you to remove aliases that have been defined. This command is useful if you no longer need an alias or if you want to redefine an alias with a different command. The syntax for removing an alias is straightforward: `unalias alias_name`. For example, if you want to remove the `ll` alias created earlier, you can use the command: `unalias ll`

   Syntax: `unalias alias_name`

6. `echo`: The `echo` command is used to display text or variables on the screen. It is commonly used for printing messages or displaying the values of variables. Following examples demonstrate the usage.

```
$ echo hello
hello
$ echo "hello world for real"
hello world for real
$ echo "hello1" "hello2"
hello1 hello2
```

   Note that everything inside the quotes is treated as a single argument, and the quotes are dropped. Multiple strings can also be passed to `echo`, and they all get printed on the same line.

   Syntax: `echo string`

7. History Feature: Implement a history feature that maintains a list of all executed commands (whether successful or not). This feature allows users to view and access their command history, providing convenience and the ability to recall and rerun commands. Please note that the history list should contain any *input string entered by the user*, and not the thing that actually gets executed. This becomes important in later sections.

   The `history` command, without any arg should display the history list in the format `"%d %s\n", index, stored_input`. For example, a history list would look like this (output of the first three commands hidden for clarity)

```
$ ls
$ cd Documents
$ echo "dang dude"
$ history
1 ls
2 cd Documents
3 echo "dang dude"
4 history
```

   The `history` command can receive an optional argument to execute a command from the history list. For example, `history 1`, in case of the previous history, would execute the 1st command from the history list, which is `ls`, and `history 3` should execute `echo "dang dude"`. Note that the history list should be updated with `history arg` instead of the command from the history list that it executes.

   All the commercial shells, allow the users to scroll through the history list by pressing the `UP` and `DOWN` keyboard keys, at the prompt. We expect you to add this functionality to your shell as well. This really adds to the convenience of using a shell, and is pretty cool as well not gonna lie. You can see the expected behaviour by going to BASH or your shell of choice, inputting a few commands, and then at the prompt, pressing the UP and DOWN keys.

   Syntax: `history [index]`

All the above commands, should handle argument errors as well, i.e. if the commands receive invalid number of arguments, or valid number of incorrect arguments, they should not execute, and should display the error.

4.1.3 Executable commands

Typing any command that is not built in executes a program from the filesystem in a new foreground process using the functions for process management we have studied.

- The first word of the command line is the executable to run. The entire command array should become the arguments to this executable command, including the command name.

- The executable should be located according to the `PATH` environment variable. Read the `man` page for the `exec` family of functions (run `man 3 exec`) to choose the variant of `exec` that will do this with minimal work.
- In case of the interactive mode, the shell should show the next command prompt when the command terminates. In case of scripting, it should execute the next command in the file/script.

You will essentially need to build a parser as well (ofcourse), which parses the input command string, and extracts the information out of it, including the command name, args etc.

5 Part 2 : Intermediate

NIX[3] shells also support a core feature called IO redirection (input output redirection), which refers to the ability of redirecting the input or output of commands. Generally commands take their input from `stdin`, and dump their output to the `stdout`. For our purposes, we can consider `stdin` to be the terminal's input coming from the keyboard, and `stdout` to be the terminal's display. This works well enough for standalone processes, that simply take input from the user, and show the output to the user as well. However, enabling basic IPC (inter process communication), where processes can communicate with each other, becomes a necessity in a modern multi-process operating system e.g. one process can communicate its findings (output) to another specialized process (as input) for further processing, or multiple processes can write their output directly to a file, which can then be used by another process as input etc. IO redirection allows this without changing the IO structure (in the code) of the program itself.

Before proceeding, I must present a subtle technicality here. "Redirection" specifically referes to the ability of passing output of a command to, or taking input of a process from, a file or a stream. If the IO of a command needs to be passed to another command, that can be done by using pipes, and is generally called "piping commands" which will be described in detail below. Note that this definition is still a bit simplified, but it should suffice for our purposes.

5.1 Specification

Although modern shells provide a lot of redirection facilities, we will be mainly concerned with the operators `>`, `<`, `»`, `|`. The following specification pertains to this custom shell only, and is not necessarily true for all (atleast POSIX compliant) shells.

1. Output redirection
   - `command > file`: Redirects output of the `command` to `file`, replacing the file if it exists. If the file doesn't exist, it is created.
   - `command » file`: Redirects output of the `command` to `file`, appending to the file if it exists. If it doesn't exist, it is created.

   Consider the example usage of output redirection:

```
$ echo "hello world"
hello world
$ echo "hello world" > file.txt
$ cat file.txt
hello world
$ echo "hello world 2" >> file.txt
$ cat file.txt
hello world
hello world 2
```

[3] NIX is an informal shorthand for UNIX and UNIX like systems

2. Input redirection
   - `command < file`: Redirects input of the `command` to `file`. If the input file doesn't exist, the command should not execute, and report the error.

   Consider the example usage of input redirection:

```
# assuming the file created earlier. this specifies that cat should read
# input from file.txt, instead of the keyboard
$ cat < file.txt
hello world
hello world 2
```

3. Pipes
   - `command1 | command2`: Pipes the output of `command1` to the input of `command2`. The output of `command1` should not be displayed on the terminal, and should be passed as input to `command2`. The output of `command2` should be displayed on the terminal (unless it is redirected to a file or piped with another command). The general syntax of piped commands (more commonly referred to as a **pipeline**) is:

```
command [args]* [ | command [args]* ]*
```

   The `[x]` means that `x` is optional, and the * means that the preceding element can be repeated any number of times. So, the above syntax means that a pipeline can have a command, followed by any number of args, followed by any number of commands (each followed by any number of args), each separated by a `'|'` symbol.
   Consider the following example pipelines:

```
# list the contents of a directory and then filter
# the results using the grep command
$ ls
build  include  Makefile  src  test
$ ls | grep include
include
# Find the pid of the zsh instance with the lowest pid and kill it
# Note that this is not the standard way to find the desired thing at all
# it's just a long pipeline I crafted to show how we can use pipes
$ ps -opid,comm | grep -v PID | grep '[z]sh' | cut -d ' ' -f 3 | sort | head
    -1 | xargs kill
```

Shells also support a combination of pipes and IO redirects. For example, a long pipeline can dump its results (`stdout` output) to a file instead of the terminal screen. Similarly, the start of a pipeline can read input from a file instead of `stdin`, and then propagate the output to the next command in the pipeline. Consider the example usages:

```
$ echo "hello line1 hello line2" | wc
1       4      24
$ alias ls "ls --color=tty"
$ alias grep "grep --color=tty"
$ alias | grep color > greplog.log
$ cat greplog.log
ls='ls --color=tty'
grep='grep --color=tty'
# from a file with a bunch of names in the format 'Fname Lname', extract all the
    firstnames that are not 'Abdullah' and write them to a file in a sorted order
$ cut -d ' ' -f 1 < names.txt | grep -v Abdullah | sort > extracted.txt
```

For our simple shell, we can assume that the file IO redirection operator, and pipes, come after the name of the command, and the operator impacts the command that precedes it.

Note that for each process, the input can come from only one file, and the output can only go to one file as well (file in this case refers to anything which has an associated file descriptor, it can either be `stdin/stdout`, a filesystem file, or a pipe). So, if a process is in the middle of a pipeline, it can only receive input from the previous process'/command's pipe, and can only send its output to the next command's/process' pipe. It cannot have any filesystem file IO redirection. Similarly, if a command has an associated output file redirection, it cannot be piped to another command, it must be the last command in a pipeline (since the output is already redirected to a file, it cannot be sent to another command). And similarly, if a command has an associated input redirection, it cannot have a previous piped command, it must be the first command in the pipeline (since the input is already redirected from a file, it cannot be received from another command). If an invalid combination of IO redirections and pipes is encountered, the shell should report an error and not execute the command.

Consider the following examples:

```
# invalid, the first command has two output destinations
$ echo "hello world" > file.txt | grep hello
# valid, since there's only one input source, and one output destination
$ grep hello < file.txt | grep world
# invalid, since the second command has two input sources
$ grep hello < file.txt | grep world < file2.txt
# valid, since there's only one input source, and one output destination for each command
$ grep hello < file.txt | grep world > file2.txt
# invalid, the middle command has two input sources
$ grep hello < hello.txt | wc < hello.txt | grep 10
```

Based on the above information, for part 2, your shell should be able to correctly handle and execute a command input of the following form:

```
command [args]* [ < file] [ | command [args]*]* [(> OR >>) file]
```

The first command can have an optional input redirect, and only the last command (which can very well be the first command as well, since the piped commands are optional) can have an optional output redirect (either of the two, not the both). The shell should report an error otherwise, and not execute the command string. Notice that this generalized syntax also includes commands from part 1, where we looked at basic commands without any IO redirection and pipes. **From now on, we will be referring to this specification as a *pipeline*.** So having a correct *pipeline* execution automatically implies a working part 1 implementation.

Note the careful use of whitespaces in the above specification. You can assume that all commands, args and operators are *separated by atleast one space character*. There can be more than one whitespace characters though, but they should get ignored by the shell.

Note: The above specification holds true for all commands, both built-in as well as the external commands from the first part. For example, in the usage of echo command in above examples, echo is a built-in. The shell grammar makes no distinction between an external or an internal command, and thus a pipeline can have a mix of both built-ins as well as external commands.

6 Part 3 : The Advanced bits

It's time to extend the functionality of the shell even further by adding another core feature called "chaining". We'll add a few more bells and whistles to the shell as well.

NIX shells allow multiple pipelines to be chained together via different operators. Suppose you want to execute a series of commands and pipelines, such that each next command depends on the success of the

previous one. One way is to sequentially execute them one-by-one, i.e. enter one command/pipeline, wait for its completion, and then at the next prompt, enter the next command/pipeline and so on, until you encounter an error, and its time to panic. However, you can be more productive by chaining all the commands via a logical chaining operator (more details in the specification section). Similarly, a lot of the cases arise, where you want a particular command to be executed only in case an error occurs in the previous command. This can also be achieved by the use of chaining. However, chaining has many more use cases than just that, which you'll hopefully figure out once you start using those in your actual shell usage.

Another very useful feature of modern shells is wildcard substitution. This allows you to specify a "pattern" for a file, or a bunch of files, or directories in a single argument, and the shell substitutes the argument with the list of all the things which obey the pattern. A common usage is to remove/copy/move etc. all the files with a particular extension by using a wildcard. For example, the wildcard pattern `*.log` gets substituted by a space separated list of all the filenames with a `.log` extension. The possibilites are endless with the wildcards, and it makes shell usage super productive.

6.1 Specification

A detailed specification of the features you are required to implement in this part is provided below.

6.1.1 Chaining

We need to support three main chaining operators : `&&, ||, ;`.

1. Logical Chaining Operators

   - `operator &&`: The `&&` operator allows for chaining of pipelines in a logical AND fashion. If the last return status is zero (last executed command succeeded), only then the immediate right hand side of the `&&` is executed, otherwise the right hand side pipeline is skipped, and the next link in the chain is tested w.r.t. its chaining operator.
   - `operator ||`: The `||` operator allows for chaining of pipelines in a logical OR fashion. If the last return status is non-zero (last executed command failed), only then the immediate right hand side of the `||` is executed, otherwise the right hand side pipeline is skipped, and the next link in the chain is tested w.r.t. its chaining operator.

   Consider the following example usages of Logical chaining operators.

```
# first command fails, so the second one is not executed
$ false && echo "This should not be printed" && echo "Neither this"
# the first command succeeds, so the second one is executed
$ true && echo "This should be printed"
This should be printed
# only execute, and print if compilation was successful
$ gcc file.c && echo "Compilation successful" && ./a.out
# the first command succeeds, so the second one is not executed
$ true || echo "This should not be printed" || echo "Neither this"
# failure of first command results in only the second one being executed
$ false || echo "This should be printed" || echo "This should not be printed"
This should be printed
```

2. Sequential Chaining Operator `;` : The `;` operator allows for sequential chaining of pipelines. Irrespective of the return status of the previous pipeline, the next pipeline is always executed. Chaining multiple commands/pipelines with `;` is equivalent to entering them one-by-one at the prompt irrespective of the return status of the previous command/pipeline.

   Example usage:

```
$ echo "Hello" ; false ; echo "World" ; true ; echo "great"
Hello
World
great
```

All of the above chaining operators have the same priority and are *left associative*, i.e. the chaining is done from left to right. These chaining operators can be used to chain together multiple pipelines, all these chaining operators can be used in a single command input string, in different combinations.

Consider the following examples. The example with a more complicated combination of chaining operators

```
$ echo "Hello" && echo "World" || echo "Bye" && echo "World"
Hello
World
$ false || echo "prints1" && echo "prints2"
prints1
prints2
```

The following example demonstrates a more complicated/confusing combination of chaining operators.

```
ls  && false || echo "doit" && false && echo "won't work" || echo "works" ; echo "done"
|_|    |___|    |_________|    |___|    |_________________|    |__________|   |_________|
 1       2           3           4             5                   6             7
1 -> returns 0
2 -> last status 0, gets executed (because &&), returns 1 (false always returns 1)
3 -> last status 1, gets executed (because ||), returns 0
4 -> last status 0, gets executed (because &&), returns 1
5 -> last status 1, doesn't get executed (because &&)
6 -> last status 1, gets executed (because ||), returns 0
7 -> gets executed regardless of the last status, returns 0

# The output of the above command string would be
<output of ls>
doit
works
done
```

And finally, an example with chaining of multiple pipelines.

```
# If either of the strings "Forgis" and "Jeep" is found
# in the input string, then this is it
$ echo "I just put new Forgiato wheels on my vehicle from the company Jeep" > file.txt ;
    cat < file.txt | grep Forgis || cat file.txt | grep Jeep && echo "Woah.. that's..
    what??" || echo "clearly not" ; echo "done"
# Guess what would the output be.
# pls dont @ me, the manual was written when this was funny
```

6.1.2 Wildcards

You are required to support two types of wildcards : `*` and `?`.

1. `*` : The `*` wildcard matches any string, including the empty string. It can be used to match any number of characters, including zero characters. For example, `*.txt` matches all the files with a

`.txt` extension, and `*.c` matches all the files with a `.c` extension. Similarly, `*.` matches all the files with a dot at the end, and `*` matches all the files with any extension.

2. `?` : The `?` wildcard matches any single character. For example, `?.txt` matches all the files with a single character followed by a `.txt` extension, and `?.c` matches all the files with a single character followed by a `.c` extension. Similarly, `?.` matches all the files with a single character and no extension, and `?` matches all the files with a single character and any extension.

Both the wildcards can be used in a single command input string, in different combinations, inside a single argument, or across multiple arguments. For the purpose of this assignment, you need to expand wildcards, by replacing them with the list of files or directories that match the wildcard.

Consider the following examples.

```
$ ls
build file.txt_ez.c include Makefile src test
$ ls build
main.o  shell
# The following command would list all the files in the current directory
# as well as the files in the subdirectories
$ ls *
file.txt_ez.c  Makefile

build:
main.o  shell

include:

src:
main.c  shell.c

test:
README.md  test.ipynb
# The following command would list all the files with a .c extension in the src directory
$ ls src/*.c
src/main.c  src/shell.c
# echo can be used to test wildcard expansions
# A peculiar filename, but it's just to demonstrate the wildcard expansion
$ echo *.txt_*.?
file.txt_ez.c
# Explanation : *.txt_*.? matches all the files (or directories) that match the pattern :
# <any string>.txt_<any string>.<any single char>
# Possible other matches : hhahaha.txt_lmao.a, woah123.txt_xyz.b etc.
```

Please note that the wildcard expansion would be done by your shell, and not by the commands that you execute. For example, the following command would not work, if the `*.c` is passed as it is to the `ls` command.

```
$ ls *.c
ls: cannot access '*.c': No such file or directory
```

Any token/arg which has one of the wildcards in it, should be expanded to the list of files/directories that match the wildcard. If there are no files/directories that match the wildcard, then the wildcard should be left as it is. For example, if there are no files with a `.c` extension in the current directory, then the following command should be executed as it is.

```
# no C source files in current working directory
$ ls *.c
ls: cannot access '*.c': No such file or directory
$ echo ?.c
?.c
```

Expansion of the wildcard can be done during the parsing of the command string, or during the execution of the command, it's upto you. (Note, however, a weird artifact from parsing during the parsing stage, and why, it is done during the execution in any standard shell. You won't be tested on this)

```
# an example session
$ ls *
Makefile

build:
main.o  shell

include:

src:
main.c  shell.c
# if expanded during parsing stage
$ cd src && echo *.c
*.c
# if expanded right before execution
$ cd src && echo *.c
main.c shell.c
```

This happens because if the wildcard is expanded during parsing, the wildcard expands to all the matching `C` files in the CWD, but there aren't any C files in the example's top directory. However, if expanded during the execution stage, the wildcard expands to all the matching `C` files in the `src` directory, as the `cd` command has already executed, in which case there are two. This is a very subtle difference, but it is important to be aware of this.

Combining the chaining operators and the wildcards, we can write some pretty complicated command input strings. The general syntax for a valid command input string then becomes :

```
pipeline [(&& OR || OR ;) pipeline]*
where pipeline := command [args]* [ < file] [ | command [args]*]* [[ > file][ >> file]]
where command := shell builtin OR executable
and args := [arg OR wildcard]*
```

Your code should be able to handle any input string of the form specified above correctly, and execute the commands in the correct order, with the correct arguments, with the correct redirections and with the correct conditions according to the chaining operators.

Please note that the previous specifications should still be compatible with above described specifications, i.e. the previous specifications (described in part 1 and 2) should be a subset of this specification. This can be easily shown as follows:

```
pipeline [(&& OR || OR ;) pipeline]*
| |
\ / no wildcards, no chaining
 V
command [args]* [ < file] [ | command [args]*]* [[ > file][ >> file]]
```

```
| |
\ / no wildcards, and IO redirection in input string
 V
command [args]* [ | command [args]*]*
| |
\ / no pipes in the input string
 V
command [args]*

## did i just draw a pencil instead of an arrow??
```

7 Getting started

The handout contains the following files:

```
Shell/
|-- Makefile
|-- dockerfile
|-- src/
|   |-- main.c
|-- include/
|   |-- utils.h
|   |-- log.h
|-- build/
|-- test/
|   |-- Tests/
|   |-- test.py
|   |-- config.json
|-- Manual/
    |-- manual.pdf
```

The `src` directory should contain all of your source files (all the `.c` files), while all the header (`.h`) files go to `include` directory. The build files (object files, and the executable) are placed inside the `build` directory. The `test` directory contains the test files. The `dockerfile` is used to build the docker image, and the `Makefile` is used to build the project. The `README.md` file contains the instructions to build and run the project.

7.1 Setting up Docker

Although the assignment can be built on any Linux system, we recommend using Docker to build and run the project. The project, as well as the test files, have been extensively tested on a docker image, and we can guarantee that it would work on the docker image irrespective of your environment. Therfore, your submission would be tested on the docker container provided. So, we recommend using Docker to build and run the project. If you are not familiar with Docker, you can read about it here. You can install Docker on your system by following the instructions here. Once you have installed Docker, you can build the docker image by running the following command in the root directory of the project.

1. Install docker on your machine
2. Clone this repository
3. Open a terminal and navigate to the root of this repository
4. Build and run the docker container by typing `docker-compose up -build -d` in the terminal.

5. Type `docker exec -it os-fall-2023 /bin/bash` to enter the container.
6. Navigate to the mounted directory by typing `cd /home/os-fall-2023` in the terminal.
7. The container has the following packages installed: `build-essential, valgrind, binutils, libreadline-dev and git`. You can install more packages if you need to (e.g. `apt-get install vim`).
8. To exit the container, type '`exit`' in the terminal.

7.2 Build system

For building the project, you can use the `Makefile` provided. To build the project, run the following command in the project directoy, (containing the `Makefile`):

```
$ make
```

This would create the executable `Shell` (or whatever is defined to be the target's name in the `makefile`) inside the `build` directory, in `debug` mode by default. Debug mode is helpful for debugging, as it provides necessary macros, and useful flags to the compiler, and also disables compiler optimizations. You can also build the project in `release` mode, which enables compiler optimizations, and disables debugging macros. To build the project in `release` mode, run the following command:

```
$ make BUILD_DEFAULT=release
```

Or, you can change the value of `BUILD_DEFAULT` variable in the `Makefile` to `release`, and then run `make` without any arguments. Always make sure to run `make clean` before switching between the build modes, as the object files are not compatible between the two modes.

To run the project, just use `make run` in the project directory. To clean the `build`, to force a rebuild from scratch, use `make clean`, or `make clean; make`.

Please note that the provided `Makefile` assumes the provided directory structure. So, the above instructions are only valid if you stick to that structure.

7.3 Testing

The `test` directory contains the test files. The `test.py` script is used to test the project. The testing script assumes that the executable to be tested is located inside the `build` folder, with the name `"Shell"`. So, make sure that you have built the project, in `release` mode, before running the tests. To run the tests, run the following command in the project directory:

```
$ make test
```

This runs all the tests (for all 3 sections), and reports the score. There are three categories of the tests: `easy`, `medium`, `advanced`. So, to run a specific test, pass the name as an argument. For example, to run the `easy` tests, run the following command:

```
# Runs the easy tests
$ make test ARGS="easy"
# Runs only the medium and advanced tests
$ make test ARGS="medium advanced"
```

If for some reason, you are not happy with the test results, you can see the intermediate results for yourself as well, by looking at the files in the directory `test/Tests/test_output`, which stores the output of the commands ran by your shell, and the reference shell.

8 Submission

The submission should be done through LMS on the assignment tab. The submission should be a `zip` file containing the following:

1. The `src` directory containing all the source files.
2. The `include` directory containing all the header files.

Please don't include any other files, or directories, as they'd just be making our lives harder, and would be removed nonetheless. Also, please make sure to follow the specified directory structure and not just dump all the files directly in the archive. The zip file should be named as `<roll_number>.zip`, where `<roll_number>` is your roll number. For example, if your roll number is `24100173`, then the zip file should be named as `24100173.zip`. The zip file should not contain any subdirectories, i.e. the zip file should not contain a directory named `24100173`, which contains the `src` and `include` directories. The zip file should be submitted on LMS before the deadline. **Late submissions will not be accepted.**

Good Luck, and Happy Coding!

Reference

A Syntax Specifcation reference

As mentioned earlier, the syntax used to specify syntax in this manual is an adoption from `man` pages conventions, with a few changes. Following is a brief description of different operators used.

1. `[]`: The `[]` operator specifies that the enclosed token or expression is optional. It is used to specify optional args, or optional operators. The operator can be nested inside another `[]` operator as well. For example, `[arg1 [arg2]]` specifies an optional arg1, which can be followed by an optional arg2, (but `arg2` can only be present if there's an `arg1` in the first place) So, empty string, `arg1`, and `arg1 arg2` are all valid examples but `arg2` or `arg1 arg2 arg2` etc. are invalid examples.

2. `()`: The `()` operator specifies that the enclosed token/expression is mandatory. However, it is not always used, and if a `token` is found without any operator, that is also considered mandatory. So, for example, the syntax specifications : `(command) [arg]` and `command [arg]` are equivalent, both specifying that the valid syntax comprises of a `mandatory` command, followed by an optional argument. The main use of the operator `()` is to specify an expression, that reduces to a single token. The scope of both `[]` and `()` is local.

3. `OR, AND`: These logical operators are self-explanatory. They are used to form an expression. The `OR` operator specifies that *only* of the operands can be present, and the `AND` operator specifies that all of the operands need to be present. For example, the syntax specification `[(> OR ») file]` specifies optional output redirection. Note the nesting, and the use of an expression in the syntax specification.

4. `*`: The operator `*` represents a repetition of the preceding expression. Note that, it doesn't say that the copies of the token are to be duplicated arbitrary number of times, but the preceding expression can be repeated arbitrary number of times. For example, `command [args]*` specifies that a command can take arbitrary number of *optional* commands, i.e. `command arg1 arg2 ... argn`, where n is 0 or a positive integer.

5. `:=`: This operator represents equivalence. It is used to mainly use shorthands, to breakdown a complicated syntax into smaller parts.

B System Calls

System calls, or more commonly "syscalls", are specialized functions. System calls are different from a regular procedure call in that the callee is executed in a privileged state, i.e, that the callee is within the operating system. Because, for security and sanity, calls into the operating system must be carefully controlled, there is a well-defined and limited set of system calls. This restriction is enforced by the hardware through trap vectors: only those OS addresses entered, at boot time, into the trap (interrupt) vector are valid destinations of a system call. Thus, a system call is a call that trespasses a protection boundary in a controlled manner.

The process abstraction, as well as some operations related to processes, are managed by the operating system, therefore, in order to work with the processes, and control them, system calls are essential. The following UNIX syscalls are probably going be to useful for this project.

System Call	Brief Description
`chdir`	Changes the current working directory of the process.
`close`	Closes a file specified by the file descriptor.
`dup`	The `dup()` system call creates a copy of the file descriptor provided, using the lowest-numbered unused file descriptor for the new descriptor.
`dup2`	The `dup2()` system call performs the same task as `dup()`, but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in `newfd`.
`exec`	`exec` is a family of syscalls, which are responsible for replacing the current process with a new process. The different functions differ in terms of the params they receive.
`exit`	Terminates the current process, with the specified exit status code.
`fork`	Creates a new child process, identical to the parent process.
`getcwd`	Returns the current working directory of the process.
`getenv`	Returns the value of an environment variable e.g. `getenv(HOME)` can be used to retrieve the value of the home directory.
`getpid`	Returns the process ID of the current process.
`getppid`	Returns the process ID of the parent of the current process.
`open`	Opens a file. On success, returns a file descriptor.
`pipe`	Creates a pipe. On success, returns two file descriptors, one for reading and one for writing.
`read`	Reads data from a file specified by a file descriptor.
`setenv`	Sets the value of an environment variable.
`wait`	`wait` family of functions wait for a child process to terminate.
`write`	Writes data to a file specified by a file descriptor.

Please note that this is a very brief description of some useful syscalls that you may or may not use in the implementation. For more details, you can always visit the `man` pages. The `man` pages also mention the header files that need to be included in order to use the syscalls, as well as the return values, parameters and other details.

C GNU Readline

GNU Readline is a library that provides line-editing and history capabilities for interactive programs with a command-line interface, such as the Unix shell and the programming languages Python, Ruby and Haskell. It is free software, distributed under the terms of the GNU GPL, version 3 or later.

As far as I know, Bash, and a lot of other commercial shell programs use Readline to provide the standard command line interface. The Readline library includes additional functions to maintain a list of previously-entered command lines, to recall and perhaps reedit those lines, and perform csh-like history expansion on previous commands. It also provides the UP/DOWN scrolling through history by default. The significance of this library will become apparent instantly when you'll use `scanf` (a highly discouraged way to read input from stdin), and won't be able to edit the entered input in any way.

You are free to implement these features on your own if you want. That itself, can be a very interesting learning opportunity but you are also allowed to use GNU readline. The following C program demonstrates the usage of GNU readline. The provided `makefile` has been configured to link the program with the readline library. Readline is a very extensive library, and is full of useful/useless features. Make sure to check out the official docs.

```c
#include <stdlib.h>
#include <stdio.h>
#include <readline/readline.h>
#include <readline/history.h>

int main()
{
    // Configure readline to auto-complete paths when the tab key is hit.
    rl_bind_key('\t', rl_complete);

    // Enable history
    using_history();

    while (1) {
        // Display prompt and read input
        char* input = readline("prompt> ");

        // Check for EOF.
        if (!input)
            break;

        // Add input to readline history.
        add_history(input);

        // Do stuff...

        // Free buffer that was allocated by readline
        free(input);
    }
    return 0;
}
```

D glob glob

Globbing is the operation that expands a glob into the list of pathnames matching the glob. A glob is a pattern that matches a set of filenames with a single expression. (Wait, this sounds oddly familiar to the wildcards in the shell, doesn't it? That's because they are the same thing.)

For your custom shell, you are required to implement the globbing feature. You are free to implement it on your own, or use the `glob` function provided by the GNU C library. The following C program demonstrates the usage of `glob`. Make sure to check the `man` page, or the online documentation to find out more.

```c
#include <stdio.h>
#include <glob.h>

int main() {
    glob_t glob_result;
    char *pattern = "*.txt";  // Example wildcard pattern

    // Perform wildcard expansion
    int ret = glob(pattern, 0, NULL, &glob_result);
    if (ret != 0) {
```

```
        printf("Error in glob: %d\n", ret);
        return 1;
    }

    // Iterate over the matched files
    for (size_t i = 0; i < glob_result.gl_pathc; ++i) {
        printf("Matched file: [%s]\n", glob_result.gl_pathv[i]);
    }

    // Free the allocated memory
    globfree(&glob_result);

    return 0;
}
```

E Debugging

"Debugging is like being the crime detective in a crime movie where you're also the murderer."

Debugging is the process of finding and resolving bugs (defects or problems that prevent correct operation) within computer programs, software, or systems. Since this will be the first time writing a non-trivial C (read C, not C++) program for a lot of you people, you will be making a lot of mistakes. And that's okay. That's how you learn. Often times you will find yourself staring at a Segmentation fault message that shows up out of nowhere. Or maybe your program will just crash without any error message. Or maybe your program will just hang and not do anything. Or maybe your program will just not do what you want it to do. These are all very common problems that you will face while writing your shell. And the only way to solve these problems is to debug your program.

E.1 Print-and-hunt

One of the most common (and probably the most primitive) methods of debugging is print-and-hunt. This is something you probably are the most used to. A proper way to do printf debugging is to use logs instead. With logs, you can separate debug statements from the actual error/non-error print statements. In a debug build the debug statements are printed, and in a release build they are excluded, by simply changing the value of some control variable instead of removing all the print statements manually. For this purpose, we have provided two macros, `LOG_DEBUG`, and `LOG_ERROR`, which are wrappers around `printf` and can be used how you would use `printf`. These macros are defined in `include/log.h`. You can also use `assert` statements to check for certain conditions. If the condition is false, the program will terminate and print the line number and the file name where the assertion failed (only in debug mode).

E.2 GDB

GDB is the GNU Project debugger. It allows you to see what is going on 'inside' another program while it executes – or what another program was doing at the moment it crashed. GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

You can use GDB to look at the values of different variables at different times in a program. The program flow is also in your control instead of the program just running from start to finish. You can also set breakpoints in your program, which will pause the execution of the program at that point and give you control. You can then step through the program line by line, or jump to a specific line, or jump to a specific function etc.

I am sure all the practice from doing the bomb lab, and the attack lab (all by yourself) will pay off.

E.3 Valgrind

Valgrind is a programming tool for memory debugging, memory leak detection, and profiling. Valgrind is a dynamic binary instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. Valgrind installation comes with a lot of tools, but we are concerned the most with `memcheck`, which is also the default tool.

Valgrind can be used to detect memory leaks, as well as their sources in a program. Valgrind is also particularly useful to find the sources of Segmentation faults in your programs. As soon as you hit a segfault, you should re run your program with valgrind, and valgrind will tell you exactly where the segfault occurred in the code (provided the same initial conditions). To run valgrind on the shell program (preferably the debug build), run the following command:

```
$ make valgrind
```