Table of Contents

Table of Figures

1. ICT374 Project Declaration:

MURDOCH UNIVERSITY
PERTH, WESTERN AUSTRALIA

Discipline of Information Technology,

Media and Communications

College of Arts, Business, Law and Social Sciences

ICT374 ASSIGNMENT 2

PROJECT DECLARATION

Group Members (full name and student number):

Member 1: Daphne Tay ____

Member 2: Yin Zhanpeng ____

Tutor's Name: Dr Loo Poh Kok ____

Assignment Due Date: 25/11/2023 ____________ **Date Submitted: 25/11/2023** ____________

Your assignment should meet the following requirements. Please confirm this (by ticking boxes) before submitting your assignment.

☒ All details above are completed.
☒ We have read and understood the Documentation Requirements of this assignment
☒ **This assignment submission is compliant to the Documentation Requirements.**
☒ The archive file (a zip file) contains the file "Assignment2.pdf"
☒ We have included all relevant Linux source code, executables and test files in the tar archive. The file names are chosen according to the project specification.
☒ This archive file will be submitted to ICT374 Unit LMS.
☒ We have kept a copy of this assignment, including this archive file, in a safe place.
☒ We have completed Task Allocation and Completion Record below.
☒ **We have signed the Group Declaration in the next page.**

The unit coordinator may choose to use your submission as sample solutions to be viewed by other students, but only with your permission. Please indicate whether you give permission for this to be done.

☐ Yes, we are willing to have my submission without change be made public as a sample solution.
☒ Yes, we are willing to have my submission be made public as a sample solution, as long as my submission is edited to remove all mentions of my identity.
☐ No, we are not willing to have my submission made public.

Group Declaration

As a group assignment, each member of the group is expected to make an equal contribution to the assignment and receives the same mark for the assignment.

However, we recognise that on some occasions and due to various reasons, the actual contributions to the assignment from the members could be unequal despite the best efforts of each member. In this case, we can still accept your assignment provided that all members of the group reach an agreement on their percentages of contribution to the assignment, and the agreement accurately reflects the real contribution by each member. In such a case, a member's mark is linked to his or her agreed contribution to the assignment and is calculated using the following formula:

A member's mark = minimum (group mark x the member's percentage of contribution x 2, group mark + 10, 100)

On some rare occasions, the two members of the group fail to reach an agreement on their contributions to the assignment. In such a case, in order for your assignment to be marked, each member of the group must complete a detailed *Task Breakdown List* and state his or her own claim of the percentage of contribution to the assignment. Your tutor will then award each member a mark based on his assessment of the quality of the assignment as whole as well as his assessment of that member's contribution to the assignment based on the information provided.

Please complete and sign ***one*** of the three declarations below:

*We have made **equal** contributions to this assignment. We understand that each of us will receive the same mark for this assignment.* Signature (member 1): Daphne Tay______________________ Date: 25/11/2023____________________ Signature (member 2): Yin Zhanpeng ____________________ Date: 25/11/2023____________________
*We have made **unequal** contributions to this assignment. The percentage of contribution by each of us is given below (note the sum of the contributions by the two members must be equal to 100%):* Member's name: Contribution (%): Member's name: Contribution (%): *We understand that each of us will receive a mark for this assignment that is linked to our contributions to the assignment. The mark will be calculated using the following formula:* A member's mark = minimum (group mark x the member's percentage of contribution x 2, group mark + 10, 100) Signature (member 1): ________________________________Date: _____________________________ Signature (member 2): ________________________________Date: _____________________________
We are unable to reach an agreement on the percentage of our contributions to this assignment. However, in order for our tutor to be able to properly assess the work completed by each of us, each of us has completed a detailed Task Breakdown List which is included in this submission. We will accept our tutor's determination of our contributions to this assignment. Signature (member 1): ________________________________Date: _______________________ Signature (member 2): ________________________________Date: _______________________

2. Extension Granted:

No extension.

3. List of Files:

- makefile
- simpleShell.c
- simpleShell.o
- simpleShell
- command.h
- command.c
- command.o
- foo.txt
- error.txt

Apart from compiling the program with ‘make’, please add on this line on the command line as well – ‘gcc simpleShell.o -o simpleShell’.

So command line input goes like this:

make

gcc -Wall simpleShell.c -o simpleShell’

4. The project title and a brief description of the project:

ICT374 Assignment 2: A Simple Unix Shell

5. Self-diagnosis and evaluation:

Fully functional features

1. Simple commands

The simple commands such as ‘ls’, ‘ps’ and ‘who’ etc are executed and their outputs are displayed accordingly.

2. Reconfigurable shell prompt (default %)

The initial shell prompt is printed as “%” and calling the “prompt” command allows the shell prompt to be changed to an input of any kind - string, empty or a combination of words, numbers, special characters, and spaces.

3. The shell built-in command pwd

The “pwd” command displays the current working directory within the directory structure.

4. Directory walk

The “cd” command navigates to the home directory when just “cd” is entered and navigates to another directory when the full path is entered.

5. Tokenisation

Large command inputs are parsed into smaller units and are executed accordingly, e.g. ‘ls -l’ will display all the files and directories in the long format with additional information such as permissions etc.

6. Redirection of the standard input

The “<” command retrieves input from an external file to the standard output, alongside the use of other commands such as ‘cat’, ‘grep’ etc.

7. Redirection of the standard output

The “>” command redirects the standard output of a command to a file and automatically creates the file if it does not exist.

8. Redirection of the standard error

The “2>” command captures error messages into a file named ‘error’ and the file is automatically created if it does not exist.

9. Shell pipeline

The “|” command connects the standard output of the first command to the standard input of the second command via a pipe.

10. Sequential job execution

The “;” command allows multiple jobs to be carried out one by one (sequentially), with the latter having to wait for the former to complete the execution.

11. The shell built-in command exit

The “exit” command terminates the program.

12. Handling of Ctrl-C, Ctrl-\ and Ctrl-Z

Ctrl-C, Ctrl-\ and Ctrl-Z all do not terminate the program and instead catches the signal and allows the program to continue running.

13. Claim of zombies

The zombie processes have been claimed and do not take an entry in the Process Table once they are dead.

Not Fully Functional Features

1. Wildcard Characters

- The token “*” is implemented and provides a list of the files with the specified file formats when “ls” is entered alongside it.
- The token “?” is implemented but is not functional as it displays an error message instead of expanding the filename.

2. Background job execution

The “&” command allows commands to be carried out in the background while another command is concurrently executed without having to wait for process termination but does not print out the prompt if it is the last command being executed.

3. Command history:

- The up and down keys have not been implemented and therefore the previous commands cannot be navigated using that
- The “history” command displays all the past valid commands that have been entered.
- The “!n” command allows the nth command entered to be retrieved and executed again.
- The “!string” command allows the string entered to be retrieved and executed again

4. Handling of slow system calls

The code has been implemented but has not been tested for its functionality.

6. Discussion of your solution:

Solutions that is novel or unusual

Wildcard Expansion

The shell integrates glob.h for expanding wildcards, such as *.c and *.?, in command inputs. This functionality significantly enhances the shell's capabilities, enabling complex pattern matching for file names. It allows the execution of commands on multiple files simultaneously, thereby improving efficiency and user experience. The capability to handle wildcards demonstrates a high level of sophistication in command interpretation, catering to complex file management requirements. This feature simplifies file operations and empowers users with more advanced file handling capabilities.

History Management

A command history feature has been implemented in the shell, which stores and retrieves previously entered commands. This feature significantly improves the user experience by making the shell more user-friendly and efficient. It allows users to easily retrieve and re-execute past commands without the need to retype them, saving time and reducing the likelihood of errors when dealing with long or complex commands. This enhancement in command recalls and execution streamlines the overall user interaction with the shell.

Custom Prompt

The shell offers the functionality for users to dynamically change the shell prompt, highlighting a focus on customization and user preference. This flexibility allows users to personalise their command-line interface, tailoring the prompt to display relevant information or preferred formatting. Such customization enhances the usability and interactivity of the shell, providing users with a more informative and engaging command-line experience. The implementation of a customizable prompt is indicative of the shell's adaptability and user-centred design approach.

Technical Choices and Considerations:

Modular Function Design

Each major functionality (like changing directory, handling wildcards, history management) is encapsulated in separate functions to improve modularity. This helps to simplify a complex system and allows each feature to be individually managed.

This improves code reusability as the functions can be used throughout the codebase. Furthermore, this facilitates testing and debugging as issues are localised and problems can be easily fixed. It also allows for easier scalability as new functionalities can be added without affecting the rest of the codebase.

This was especially useful as it allowed us to collaborate on different functions without having to need to consult the other. Therefore, once we were done with our own sections, only the combination was required for a fully established shell.

Results and Expectations

The shell successfully fulfils most of the basic requirements and aptly manages several advanced features. This accomplishment stems from its intricate design, which incorporates a variety of functionalities.

To enhance its architecture further, an improvement could be made by adopting a more modular design approach. This would involve structuring each feature as a distinct class, allowing for clearer separation of concerns and more streamlined development. Ultimately, these individual classes would integrate cohesively, culminating in a robust and comprehensive system. This modular transformation holds the potential to elevate the shell's efficiency, maintainability, and scalability.

Strengths

The shell's capability to tokenize user input is a critical aspect of its design, allowing for effective command recognition and argument processing. This is achieved through the tokenise_command function, which breaks down the user input into tokens, delineated by spaces. This tokenization process enables the shell to distinguish the first token as the command (e.g., ls) and subsequent tokens as its arguments (e.g., -lt).

Such parsing is pivotal in handling various commands with different argument structures, thereby enhancing the shell's usability. It also aids in error detection, allowing the shell to identify unrecognised commands or incorrect argument usage. Consequently, this tokenization mechanism lays the foundation for the shell's flexibility and robust command execution capabilities.

One of the shell's major strengths is its extensibility, primarily due to its modular code structure. Each significant function, such as changeDirectory for directory navigation, add_history and execute_history_command for managing command history, and expandWildcards for wildcard processing, is encapsulated in separate, dedicated functions. This modular approach not only simplifies understanding and modifying the code but also facilitates the addition of new features with minimal impact on existing functionalities. It significantly enhances the ease of maintenance and future upgrades, making the shell scalable and adaptable to new requirements. Moreover, this structure is conducive to collaborative development, allowing multiple contributors to work on different parts of the shell simultaneously without extensive code dependencies.

The implementation of sigchld_handler in the shell is a testament to its sophisticated process management. This signal handler is specifically designed to address the issue of zombie processes, which can occur when child processes terminate but are not properly reaped by the parent process. The sigchld_handler function ensures that the shell cleans up these defunct processes, thereby preventing resource leakage and maintaining system efficiency.

By effectively handling the SIGCHLD signal, the shell demonstrates a level of reliability and robustness that is crucial for long running and interactive applications. This feature underscores the shell's capability to manage concurrent processes and maintain system stability, making it a reliable tool for handling a variety of command-line tasks.

Weaknesses

Firstly, the current implementation does not have enough built-in shells and relies heavily on external commands (/bin/sh -c), limiting its functionality in environments where these might be restricted.

Secondly, for wildcard characters, the token "?" has been implemented but is not providing the input as compared to "*" and is providing an error message stating that the wildcard expansion has failed instead.

Thirdly, the background command "&" has been implemented and is able to successfully execute the commands concurrently in the background. However, a slight weakness is that the prompt is not printed again when there are two commands that are both to be run in the background or if the command to be run in the background is called upon last. This may cause confusion to the user who do not see the prompt and may assume that there is an error.

Fourthly, the Up Arrow key and Down Arrow key to navigate through and select one of the previous commands have not been implemented. We tried the usage of signals to capture the keys using event handling but were only successful in retrieving the input of the command and not being able to execute the command. We also tried to specify the keys by their human-readable inputs which are “^A[[“ for the Up Arrow key and “^B[[“ for the Down Arrow key and was unable to even retrieve the input at all. Therefore, we have removed that section completely and have it non-functional.

Finally, an external command line parser source file that combines all the different parsing required has been created but has not been utilised. We did not create the parser when we started on the code separately, so each function has been individually parsed on its own. Therefore, it gave rise to segmentation faults and errors when implemented upon. This indicates that our code is inefficient and potentially hard to maintain as each function must be updated individually if there are parsing issues.

Potential Improvements

Firstly, more built-in commands within the code are necessary to reduce dependency on external binaries and make the shell more versatile.

Secondly, enhanced error detection and handling must be implemented, particularly in memory allocation as it would allow easier troubleshooting of the code. Currently, we are unable to pinpoint where exactly the segmentation faults are occurring from so having a robust error handling implementation will help us to identify exactly where the error is stemming from.

Thirdly, features like tab-completion, syntax highlighting, or command suggestions could be implemented to significantly support the user experience and allow for an easier navigation of the Unix Shell.

Furthermore, implementing a help system, in the form of a user guide, within the shell to guide users through its features and usage would be a valuable addition. This will help the end-user to understand how to utilise the shell in depth through the provision of sequential instructions, examples, and explanations on the various features available.

Fourthly, a central command line parser is to be utilised and called upon instead of parsing each feature individually. Over time, individual parsing may lead to code redundancy and increased complexity if the codebase is to expand so a centralised parsing system will help to troubleshoot and update the codes in an easier manner.

Lastly, the current tokenization process may be sufficient for basic commands but can be enhanced to handle more complex syntax. For instance, better handling of nested quotes, escape characters, or special symbols (like semicolons within quotes) would improve the shell's capability to interpret intricate commands correctly.

7. Test evidence:

Test Case #1a: Behaviour of ‘prompt’ command - string input

Purpose: Verify that the ‘prompt’ command correctly changes the current prompt and displays the string input as the new prompt.

Steps:

1. Run the shell program.
2. Execute the ‘prompt’ command.
3. Input a string.

Expected Result: The ‘prompt’ command has been successfully implemented and the prompt should have been changed to the new string input.

Command Line Input and Test Output:

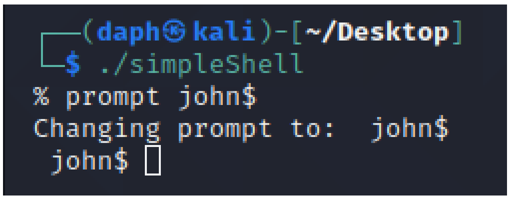


Figure 1: Changing the prompt to a string

Test Case #1b: Behaviour of ‘prompt’ command - empty input

Purpose: Verify that the ‘prompt’ command correctly changes the current prompt and displays the empty input as the new prompt.

Steps:

1. Run the shell program.
2. Execute the ‘prompt’ command.
3. Input a new line.

Expected Result: The ‘prompt’ command has been successfully implemented and the prompt should have been changed to the new input which is a combination of different keys with spaces in between.

Command Line Input and Test Output:

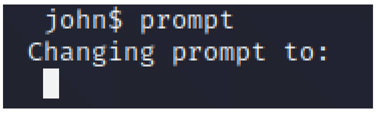


Figure 2: Changing the prompt to an empty input

Test Case #1c: Behaviour of ‘prompt’ command - combination input

Purpose: Verify that the ‘prompt’ command correctly changes the current prompt and displays the inputted keys as the new prompt.

Steps:

1. Run the shell program.
2. Execute the ‘prompt’ command.
3. Input a combination of keystrokes.

Expected Result: The ‘prompt’ command has been successfully implemented and the prompt should have been changed to the new input which is the combination of keys. This shows that the command is able to handle any type of input.

Command Line Input and Test Output:

```
  prompt 1A2 .?]_` TEST
Changing prompt to:  1A2 .?]_` TEST
 1A2 .?]_` TEST
```

Figure 3: Changing the prompt to a combination of words, numbers, special characters and spaces

Test Case #2: Behaviour of ‘pwd’ command

Purpose: Verify that the ‘pwd’ command correctly provides the current directory.

Steps:

1. Run the shell program.
2. Execute the ‘pwd’ command.

Expected Result: The ‘pwd’ command has been successfully implemented and the current directory is displayed.

Command Line Input and Test Output:

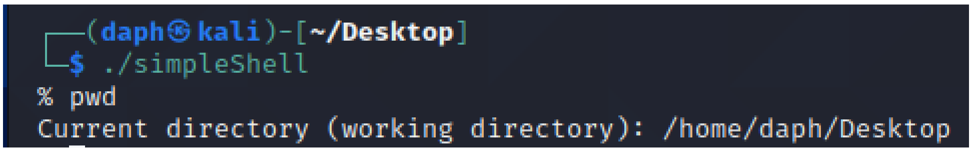


Figure 4: Printing the current/working directory

Test Case #3a: Behaviour of ‘cd’ command - home directory

Purpose: Verify that the ‘cd’ command correctly navigates to the home directory.

Steps:

1. Run the shell program.
2. Execute the ‘cd’ command.
3. Execute the ‘pwd’ command.

Expected Result: The ‘cd’ command has been successfully implemented and will provide an output that shows that the directory has been changed to the home directory. The ‘pwd’ command will then substantiate the change and provide the current directory which is the home directory.

Command Line Input and Test Output:

```
% cd
Changed current directory to: /home/daph
% pwd
Current directory (working directory): /home/daph
```

Figure 5: Navigating to the home directory and printing it

Test Case #3b: Behaviour of ‘cd’ command - /tmp

Purpose: Verify that the ‘cd’ command correctly navigates to the directory inputted with a ‘/’ in the path.

Steps:

1. Run the shell program.
2. Execute the ‘cd’ command.
3. Input ‘/tmp’ as the directory path.
4. Execute the ‘pwd’ command.

Expected Result: The ‘cd’ command has been successfully implemented and will provide an output that shows that the directory has been changed to the inputted directory with the ‘/’ in the path. The ‘pwd’ command will then substantiate the change and provide the current directory.

Command Line Input and Test Output:

```
% cd /tmp
Changed current directory to: /tmp
```

Figure 6: Navigating to a random directory

Test Case #3c: Behaviour of ‘cd’ command - tmp

Purpose: Verify that the ‘cd’ command will display an error message when an input is entered without a ‘/’ in the path.

Steps:

1. Run the shell program.
2. Execute the ‘cd’ command.
3. Input ‘tmp’ as the directory.

Expected Result: The ‘cd’ command has been successfully implemented and will display an error message as the directory does not exist.

Command Line Input and Test Output:

```
% cd tmp
chdir() error: No such file or directory
Directory change failed.
```

Figure 7: Navigating to a random directory without the correct path syntax usage

Test Case #4a(i): Behaviour of ‘*’ command - Wildcard Character

Purpose: Verify that the **‘‘*’** command will display all the files with the specific format when an input is entered with a ‘ls’’ input before.

Steps:

1. Run the shell program.
2. Input ‘ls’ to list all the files in the directory.
3. Execute the ‘*’ command.
4. Input ‘.c’ to specifically look for all the C source files in the directory.

Expected Result: The ‘*’’ command has been successfully implemented and will display all the C source files found in the directory when ‘ls’ is inputted.

Command Line Input and Test Output:

```
% ls *.c
command.c
simpleShell.c
```

Figure 8: Using wildcard characters to find certain files

Test Case #4a(ii): Behaviour of ‘*’ command - Wildcard Character

Purpose: Verify that the **‘‘*’** command will display an error message stating that the command is not found when ‘ls’ is not inputted beforehand.

Steps:

1. Run the shell program.
2. Execute the ‘*’ command.
3. Input ‘.c’ to specifically look for all the C source files in the directory.

Expected Result: The ‘*’’ command has been successfully implemented and will display a message stating that the command is not found when ‘ls’ is not inputted beforehand.

Command Line Input and Test Output:

```
% *.c
sh: 1: command.c: not found
```

Figure 9: Error message when using wildcard characters without an additional command

Test Case #4b: Behaviour of ‘?’ command - Wildcard Character

Purpose: Verify that the **‘‘?’** command will display an error message stating that the wildcard expansion has failed.

Steps:

1. Run the shell program.
2. Input ‘ls’ to list all the files in the directory
3. Execute the ‘?’ command.
4. Input ‘.c’ to specifically look for all the C source files in the directory.

Expected Result: The ‘?’’ command has not been successfully implemented and will display an error message stating that the wildcard expansion has failed.

Command Line Input and Test Output:

```
% ls ?.c
Wildcard expansion failed.
```

Figure 10: Error message printed when the wildcard character implementation is unsuccessful

Test Case #5a: Behaviour of ‘>’ command - Redirection of the Standard Output

Purpose: Verify that the **‘‘>’** command will redirect the standard output.

Steps:

1. Run the shell program.
2. Input ‘ls’ to list all the files in the directory
3. Execute the ‘>’ command.
4. Input ‘foo’ as the file for the output to be redirected to.

Expected Result: The ‘>’’ command has been successfully implemented and will display all the files in the directory in the ‘foo’ file.

Command Line Input and Test Output:

Command Line Input:

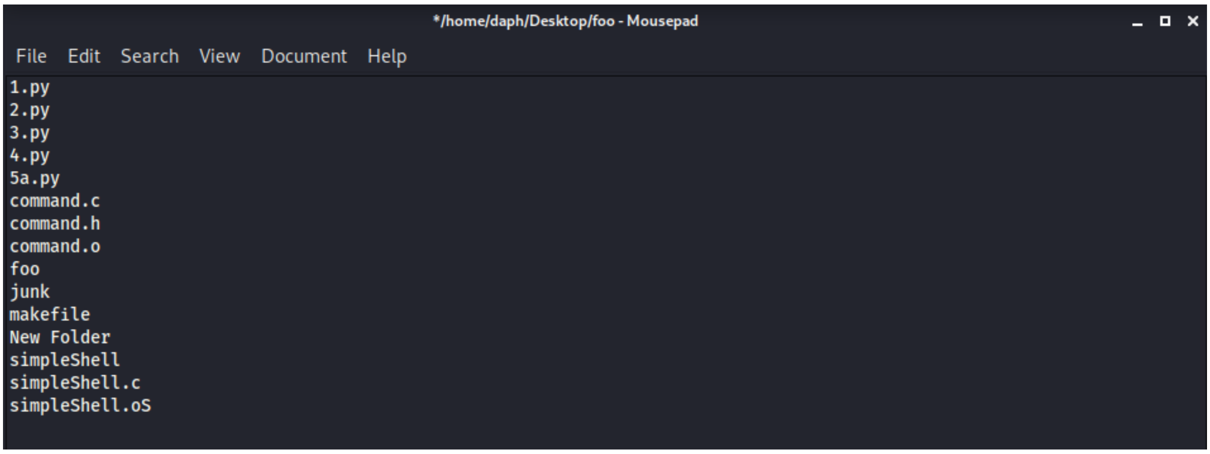
```
% ls > foo
```

Figure 11: Command line input to redirect output to another file

‘foo’ file created as a result of the ‘>’ command:

Figure 12: "foo" file created

Contents of the ‘foo’ file:




```
1.py
2.py
3.py
4.py
5a.py
command.c
command.h
command.o
foo
junk
makefile
New Folder
simpleShell
simpleShell.c
simpleShell.oS
```

Figure 13: Contents of the "foo" file

Test Case #5b: Behaviour of ‘<’ command - Redirection of the Standard Input

Purpose: Verify that the **‘‘<’** command will redirect the standard input.

Steps:

1. Run the shell program.
2. Input ‘cat’
3. Execute the ‘<’ command.
4. Input ‘foo’ as the file for the input to be redirected from.

Expected Result: The ‘<’’ command has not been successfully implemented and will display the contents of the ‘foo’ file in the standard output.

Command Line Input and Test Output:

```
% cat < foo
1.py
2.py
3.py
4.py
5a.py
command.c
command.h
command.o
foo
junk
makefile
New Folder
simpleShell
simpleShell.c
simpleShell.o
```

Figure 14: Redirection of input to the command line from the file "foo"

Test Case #5c: Behaviour of ‘2>’ command - Redirection of the Standard Error

Purpose: Verify that the “**2>’** command will redirect the standard error.

Steps:

1. Run the shell program.
2. Input ‘ls /xxxx’
3. Execute the ‘2>’ command.
4. Input ‘error’ as the file for the standard error to be redirected to.

Expected Result: The ‘2>’’ command has not been successfully implemented and will display the error message as the contents of the ‘error’ file.

Command Line Input and Test Output:

Command Line Input:

```
% ls /xxxx 2> error
```

Figure 15: Command line input for redirection of error message to the "error" file

‘error’ file created as a result of the ‘2>’ command:

Figure 16: "error" file created

Contents of the ‘error’ file:

```
/home/daph/Desktop/error - Mousepad
File  Edit  Search  View  Document  Help
ls: cannot access '/xxxx': No such file or directory
```

Figure 17: Contents of the "error" file

Test Case #6: Pipeline (Pipe Operator)

Purpose: Verify that the pipe operator (|) correctly connects the standard output of one process to the standard input of another, allowing data to be transferred between processes.

Steps:

1. Open the shell program.
2. Execute a command that produces a list of data as output
3. Pipe the output of the first command to a second command using the | operator.

Expected Result:

The first command (ls -lt) should produce a list of data, and the pipe operator should connect its standard output to the standard input of the second command (more). The second command should display the data in a paginated manner, allowing the user to scroll through it.

Command Line Input and Output:

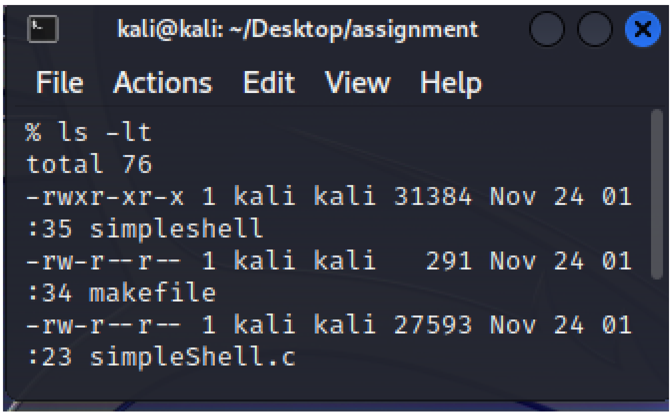


Figure 18: Part 1/2 of the output of the shell pipeline with the scroll function

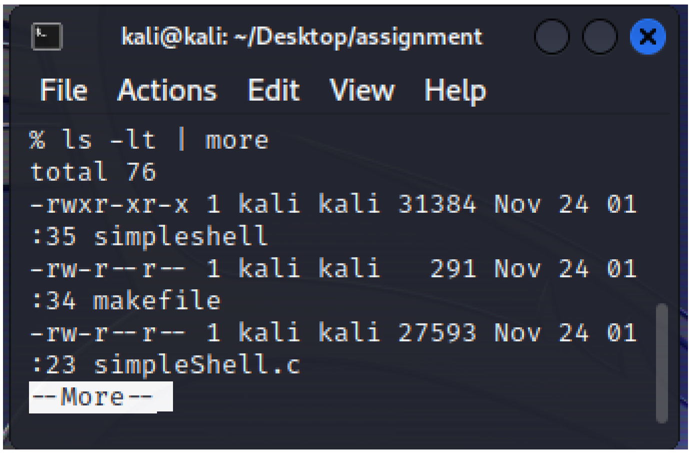


Figure 19: Part 2/2 of the output of the shell pipeline with the scroll function

Test Case #7: Background Job Execution

Purpose: Verify that commands executed in the background do not block the shell, allowing the concurrent execution of other commands.

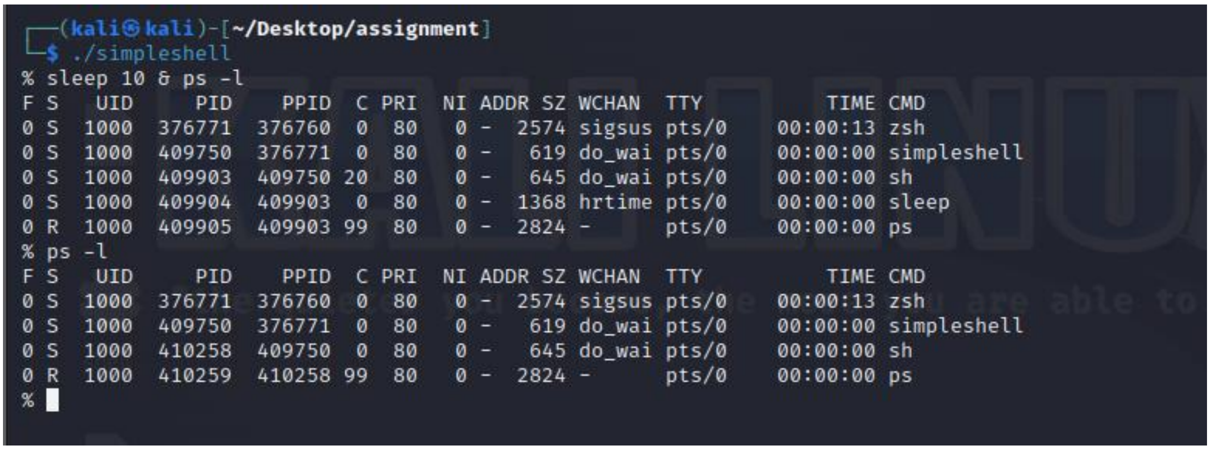
Steps:

- Open the shell program.
- Execute a command in the background using the & operator
- Immediately execute another command without waiting for the background command to finish.

Expected Result:

The shell should start the background command (sleep 10) without waiting for it to complete. The shell should then immediately execute the subsequent command (“ps -l") while the background command continues running independently. Both commands can run concurrently.

Command Line Input and Output:



```
┌──(kali㉿kali)-[~/Desktop/assignment]
└─$ ./simpleshell
% sleep 10 & ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  376771  376760  0  80   0 -  2574 sigsus pts/0    00:00:13 zsh
0 S  1000  409750  376771  0  80   0 -   619 do_wai pts/0    00:00:00 simpleshell
0 S  1000  409903  409750 20  80   0 -   645 do_wai pts/0    00:00:00 sh
0 S  1000  409904  409903  0  80   0 -  1368 hrtime pts/0    00:00:00 sleep
0 R  1000  409905  409903 99  80   0 -  2824 -      pts/0    00:00:00 ps
% ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  376771  376760  0  80   0 -  2574 sigsus pts/0    00:00:13 zsh
0 S  1000  409750  376771  0  80   0 -   619 do_wai pts/0    00:00:00 simpleshell
0 S  1000  410258  409750  0  80   0 -   645 do_wai pts/0    00:00:00 sh
0 R  1000  410259  410258 99  80   0 -  2824 -      pts/0    00:00:00 ps
%
```

Figure 20: Background execution

Further explanations: run ps -l twice one before and after indicating that the shell indeed run both commands concurrently

Test Case #8: Sequential Job Execution

Purpose: Verify that commands are executed sequentially in the order they are provided, with the second command waiting for the first one to finish.

Steps:

- Open the shell program.
- Enter two command and connect them using ";"

Expected Result:

The shell should execute them sequentially .

Command Line Input and Output:

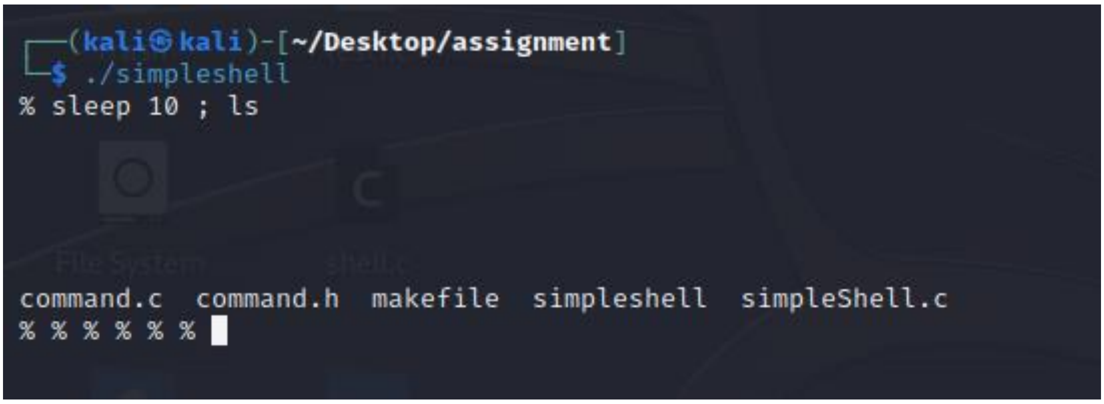


Figure 21: Sequential Execution

Further explanations: I press "Enter" multiple times after executing the command, and the prompt displays multiple "%" after listing the items. This indicated that its sequential instead of all at once

Test Case #9a: Using the “history” Built-in Command

Purpose: Verify that the history built-in command correctly displays a list of past commands.

Steps:

- Open the shell program.
- Enter multiple commands.
- Run the history command.

Expected Result:

The shell should display a numbered list of previously executed commands, including their command numbers and the commands themselves.

Command Line Input and Output:

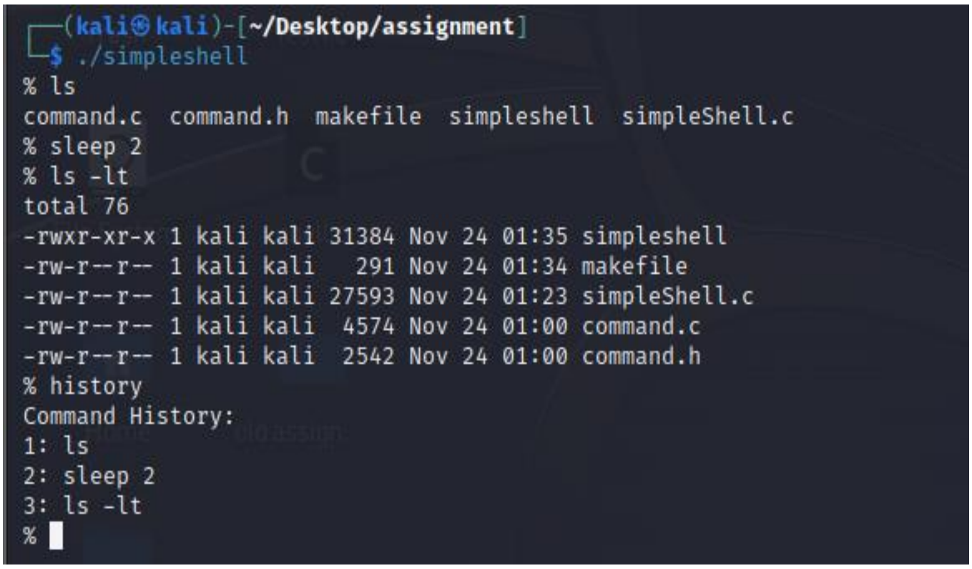


Figure 22: Displaying all the past commands

Test Case #9b(i): Using the "!n" Built-in Command

Purpose: Verify that the "!n" built-in command correctly displays the nth command and execute the command again.

Steps:

- Open the shell program.
- Enter "history" to identify the previous commands and their respective numbers
- Run the "!2" command

Expected Result:

The shell should display a numbered list of previously executed commands, including their command numbers and the commands themselves. The "!2" command should display the 2nd command entered and execute the specific command again.

Command Line Input and Output:

```
% history
Command History:
1: pwd
2: who
3: ls
% !2
who
daph     tty7         2023-11-25 13:07 (:0)
```

Figure 23: Displaying the "!2" command from a list of previous command

Test Case #9b(ii): Using the “!n” Built-in Command

Purpose: Verify that the “!n” built-in command displays an error message.

Steps:

- Open the shell program.
- Enter “history” to identify the previous commands and their respective numbers
- Run the “!4” command

Expected Result:

The shell should display a numbered list of previously executed commands, including their command numbers and the commands themselves. The “!4” command should display an error message as the 4th command has not been entered yet and does not exist in the history array.

Command Line Input and Output:

```
% history
Command History:
1: pwd
2: who
3: ls
% !4
Invalid command number entered.
```

Figure 24: Error message displayed when the command does not exist

Test Case #9c(i): Using the "!string" Built-in Command

Purpose: Verify that the "!string" built-in command correctly displays the string command and execute the command again.

Steps:

- Open the shell program.
- Enter "history" to identify the previous commands and their respective strings
- Run the "!pwd" command

Expected Result:

The shell should display a numbered list of previously executed commands, including their command numbers and the commands themselves. The "!pwd" command should display the first command 'pwd' entered and execute the specific command again.

Command Line Input and Output:

```
1: pwd
2: who
3: ls
% !pwd
pwd
/home/daph/Desktop
```

Figure 25: Displaying the "!pwd" command from a list of previous commands

Test Case #9c(ii): Using the "!string" Built-in Command

Purpose: Verify that the "!string" built-in command displays an error message.

Steps:

- Open the shell program.
- Enter "history" to identify the previous commands and their respective numbers
- Run the "!ps" command

Expected Result:

The shell should display a numbered list of previously executed commands, including their command numbers and the commands themselves. The "!ps" command should display an error message as the string command "ps" has not been entered yet and does not exist in the history array.

Command Line Input and Output:

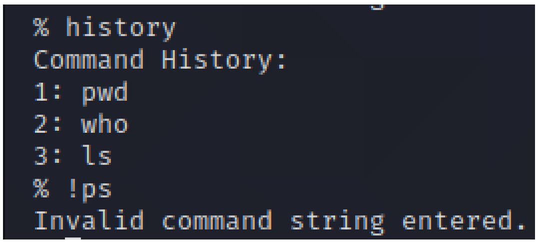


Figure 26: Error message displayed when the command does not exist

Test Case #10: Inheriting Environment Variables

Purpose: Verify that the shell inherits its environment from its parent process, including environment variables.

Steps:

- Open the shell program.
- Set an environment variable in the parent shell session.
- Execute the echo command inside the shell to check if the environment variable is accessible.

Expected Result:

The shell program should inherit the environment variable set in the parent process, and the echo command should display the value of the environment variable.

Command Line Input and Output:

Figure 27: Inheriting the environment from the parent process

Test Case #11: Behaviour of exit Command

Purpose: Verify that the exit command correctly terminates the shell program and displays any exit status or message.

Steps:

- Open the shell program.
- Execute various commands within the shell.
- Execute the exit command.

Expected Result:

The shell program should terminate, and if an exit status or message is provided, it should be displayed.

Command Line Input and Output:

Figure 28: Exiting the shell

Test Case #12(a): Behaviour of CTRL-C Command

Purpose: Verify that pressing CTRL-C does not terminate the shell program

Steps:

- Open the shell program.
- Execute various commands within the shell.
- Execute the CTRL-C command.

Expected Result:

The shell should not terminate and should continue to run as expected.

Command Line Input and Output:

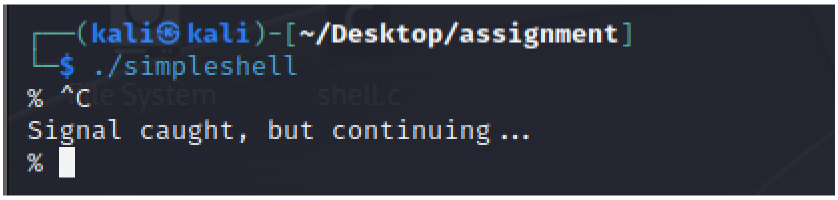


Figure 29: "CTRL-C" command executed

Test Case #12(b) Behaviour of CTRL-\ Command

Purpose: Verify that pressing CTRL-\ does not terminate the shell program

Steps:

- Open the shell program.
- Execute various commands within the shell.
- Execute the CTRL-\ command.

Expected Result:

The shell should not terminate and should continue to run as expected.

Command Line Input and Output:

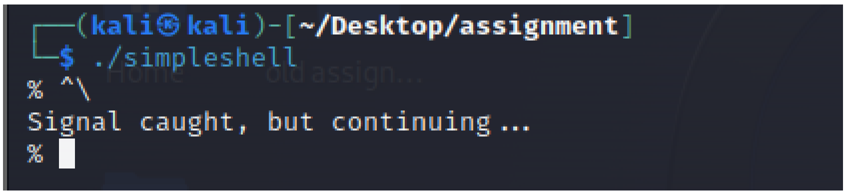


Figure 30: "CTRL-\" command executed

Test Case #12(c): Behaviour of CTRL-Z Command

Purpose: Verify that pressing CTRL-Z should not suspend the shell or terminate it.

Steps:

- Open the shell program.
- Execute various commands within the shell.
- Execute the CTRL-Z command.

Expected Result:

The shell should not be suspended, and a message indicating suspension should be displayed. The shell should not terminate.

Command Line Input and Output:

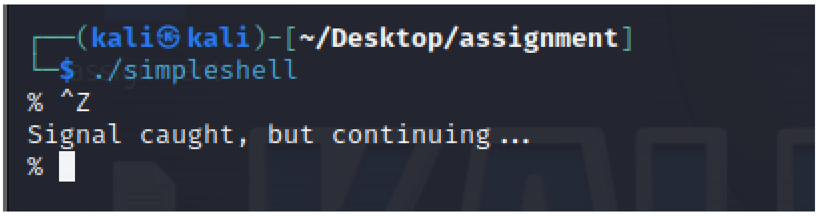


Figure 31: "CTRL-Z" command executed

Test Case #13: Simple Commands

Purpose: Verify that simple commands can be inputted and executed.

Steps:

- Open the shell program.
- Execute the “ls” command.

Expected Result:

The “ls” command will be successfully executed and a list of files existent in the directory is printed out.

Command Line Input and Output:

```
% ls
  1.py    3.py    5a.py        command.h    error    junk        'New Folder'    simpleShell.c
  2.py    4.py    command.c    command.o    foo      makefile     simpleShell    simpleShell.o
```

Figure 32: Output of the simple command "ls"

Test Case #14: Complex Command Lines

Purpose: Verify that complex commands can be inputted and executed.

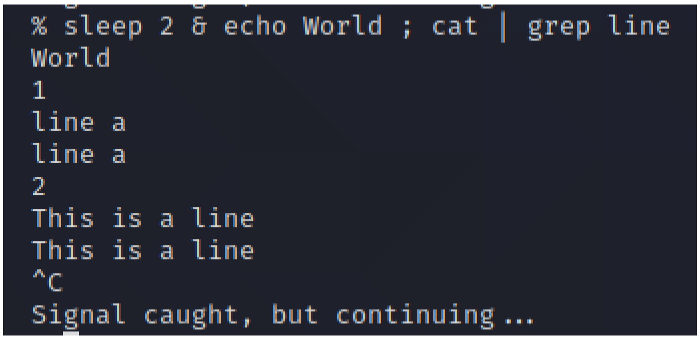
Steps:

- Open the shell program.
- Execute the “sleep 2 & echo World ; cat | grep line” command.

Expected Result:

The “sleep 2” command will be executed in the background while the “echo World” prints out the word “World” and carries out the command “cat” simultaneously. The command “cat” allows us to input indefinitely and the “grep line” command will print out the input again in the word “line” is found in the input. The input is then ended with a “CTRL-C” signal.

Command Line Input and Output:



```
% sleep 2 & echo World ; cat | grep line
World
1
line a
line a
2
This is a line
This is a line
^C
Signal caught, but continuing...
```

Figure 33: Output of the complex command line "sleep 2 & echo World ; cat | grep line”

Source Code Listings

1. makefile:

```
#makefile

simpleShell: simpleShell.o command.o
	gcc -std=c99 simpleShell.o command.o -o simpleShell

simpleShell.o: simpleShell.c command.c
	gcc -std=c99 -c simpleShell.c

command.o: command.c command.h
	gcc -std=c99 -c command.c

clean:
	rm *.o
```

2. simpleShell.c:

```c
#include <glob.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <ctype.h>

#include <fcntl.h>

#include <signal.h>

#include <errno.h>

#include "command.h"


// --------------------------------------------------


#define _POSIX_C_SOURCE 200809L

#define _GNU_SOURCE

#define MAX_COMMAND_LENGTH 100

#define MAX_ARGUMENT_LENGTH 1000

#define MAX_INPUT_LENGTH 1024

#define MAX_HISTORY_LENGTH 100

#define MAX_PATH_LENGTH 4096

#define MAX_NUM_TOKENS 100

#define MAX_PROMPT_LENGTH 100
```

```
// --------------------------------------------------

typedef struct
{
    char prompt[MAX_PROMPT_LENGTH];
    char currentDirectory[MAX_PATH_LENGTH];
    char command_history[MAX_HISTORY_LENGTH][MAX_COMMAND_LENGTH];

} Shell;

// --------------------------------------------------

int total_history = 0; // total number of commands in command_history
int history_index = 0; // index of command_history index
int total_command = 0; // total number of commands

// ----------------------------------------------------

Shell* createShell();
void changePrompt(Shell* shell, const char* newPrompt);
void printCurrentDirectory(Shell* shell);
int changeDirectory(Shell* shell, const char* path);
char** expandWildcards(const char* command, int* numExpanded);
int executeSequentially(Shell* shell, const char* command);
```

```
int handleRedirection(const char* command);

void add_history(Shell* shell, const char *command);

void execute_history_command(char *arg[]);

char* history_by_number(Shell* shell, int num);

char* history_by_string(Shell* shell, const char *str);

void execute_history_by_string(Shell* shell, const char *str);

void execute_history(Shell* shell);

int execute_piped_commands(char* commands[], int num_commands);

int executeCommand(Shell* shell, const char* command);

void handleSignal(Shell* shell, int signum);

void sigchld_handler(Shell* shell, int signum);

int tokenise_command(char* input, char* tokens[]);

void runShell(Shell* shell);

void destroyShell(Shell* shell);


// ------------------------------------------------------------


int main()

{

    signal(SIGCHLD, sigchld_handler);

    Shell* myShell = createShell();


    if (myShell)

    {

        runShell(myShell);
```

```
        destroyShell(myShell);
    }

    return 0;
}

// ------------------------------------------------------------

Shell* createShell()
{
    // allocating memory for the 'Shell' struct
    Shell* newShell = (Shell*)malloc(sizeof(Shell));
    if (newShell)
    {
        // setting the current prompt as '%'
        strcpy(newShell->prompt, "% ");

        // setting the current directory
        if (getcwd(newShell->currentDirectory, sizeof(newShell->currentDirectory)) == NULL)
        {
            // error handling
            perror("getcwd() error");

            // deallocate memory
            free(newShell);
```

```
            return NULL;
        }
    }
    return newShell;
}
// ---------------------------------------------------------
/*
 * changing the shell prompt from '%' to user input
 */
void changePrompt(Shell* shell, const char* newPrompt)
{
    if (newPrompt)
    {
        snprintf(shell->prompt, sizeof(shell->prompt), "%s ", newPrompt);
        printf("Changing prompt to: %s\n", shell->prompt);
    }
}

// -----------------------------------------------------------

/*
 * printing the current directory - pwd
 */
void printCurrentDirectory(Shell* shell)
{
```

```
    printf("Current directory (working directory): %s\n", shell->currentDirectory);
}

// -----------------------------------------------------------

/*
 * directory walk - cd
 */
int changeDirectory(Shell* shell, const char* path)
{
    // changing to the home directory
    if (!path || !path[0])
    {
        path = getenv("HOME");

        // error handling for if the directory does not exist
        if (!path)
        {
            fprintf(stderr, "Could not determine user's home directory.\n");
            return 0;
        }
    }

    // changing to the new directory given by the user input
    if (chdir(path) == 0 && getcwd(shell->currentDirectory, sizeof(shell->currentDirectory)) != NULL)
```

```
    {
        printf("Changed current directory to: %s\n", shell->currentDirectory);
        return 1;
    }

    perror("chdir() error");

    return 0;
}

// -----------------------------------------------------------

/*
 * wildcard characters - *.c or *.?
 */
char** expandWildcards(const char* command, int* numExpanded)
{
    // Initial setup and allocation
    char** expandedCommands = NULL;
    *numExpanded = 0;

    // Separate command and pattern
    char* cmdCopy = strdup(command);
    char* spacePos = strchr(cmdCopy, ' ');
```

```
if (!spacePos)
{
    free(cmdCopy);
    return NULL;  // No space found, not a valid command for expansion
}

*spacePos = '\0';  // Split command and arguments/pattern

char* commandPart = cmdCopy;
char* patternPart = spacePos + 1;

// Check for wildcards - *.?
if (strpbrk(patternPart, "*?"))
{
    glob_t glob_result;
    if (glob(patternPart, GLOB_TILDE, NULL, &glob_result) == 0)
    {
        // Allocate memory for expanded commands
        expandedCommands = malloc((glob_result.gl_pathc + 1) * sizeof(char*));

        // error handling for if the memory cannot be allocated
        if (!expandedCommands)
        {
            perror("Memory allocation failed");
            globfree(&glob_result);
```

```
            free(cmdCopy);
            return NULL;
        }

        // Reconstruct commands with expanded paths
        for (size_t i = 0; i < glob_result.gl_pathc; i++)
        {
            size_t cmdLength = strlen(commandPart) + strlen(glob_result.gl_pathv[i]) + 2;
            expandedCommands[i] = malloc(cmdLength * sizeof(char));

            if (expandedCommands[i])
            {
                snprintf(expandedCommands[i], cmdLength, "%s %s", commandPart,
glob_result.gl_pathv[i]);
            }
        }
        expandedCommands[glob_result.gl_pathc] = NULL;  // Null-terminate the array
        *numExpanded = glob_result.gl_pathc;

        globfree(&glob_result);
    }
    else
    {
        fprintf(stderr, "Wildcard expansion failed.\n");
    }
```

```
    }

    // deallocate memory
    free(cmdCopy);

    return expandedCommands;
}

// -------------------------------------------------------------

/*
 * sequential job execution - ;
 */
int executeSequentially(Shell* shell, const char* command)
{
    char* cmdCopy = strdup(command);
    char* token = strtok(cmdCopy, ";");
    int exitCode = 0;

    while (token != NULL)
    {
        // Trim leading and trailing spaces from the token
        while (*token && (*token == ' ' || *token == '\t'))
        {
            token++;
```

```
}

size_t tokenLen = strlen(token);

while (tokenLen > 0 && (token[tokenLen - 1] == ' ' || token[tokenLen - 1] == '\t'))
{
    tokenLen--;
    printf("%s", shell->prompt);
    token[tokenLen] = '\0';
}

if (tokenLen > 0)
{
    int code = executeCommand(shell, token);

    // error handling
    if (code == -1)
    {
        free(cmdCopy);
        return -1;
    }

    exitCode = code;
}
```

```
        token = strtok(NULL, ";");
    }

    // deallocate memory
    free(cmdCopy);

    return exitCode;
}

// ----------------------------------------------------------

/*
 * redirection of the standard input, standard output and standard error <, > and 2>
 */
int handleRedirection(const char* command)
{
    int stdout_backup = dup(fileno(stdout)); // Backup the original standard output
    int stderr_backup = dup(fileno(stderr)); // Backup the original standard error


    // error handling - unable to back up file descriptors
    if (stdout_backup == -1 || stderr_backup == -1)
    {
        perror("Failed to backup file descriptors");
        return -1;
```

```
}

char* cmd = strdup(command);
char* token = strtok(cmd, " ");

int output_redirect = 0;
int error_redirect = 0;

while (token)
{
  // redirection of the standard output
  if (strcmp(token, ">") == 0)
  {
    token = strtok(NULL, " ");

    if (token)
    {
      output_redirect = 1;

      int fd = open(token, O_WRONLY | O_CREAT | O_TRUNC, 0644);

      if (fd == -1)
      {
        perror("Error opening output file.");
        free(cmd);
```

```
            return -1;
        }

        dup2(fd, fileno(stdout));

        // error handling - unable to redirect output
        if (dup2(fd, fileno(stdout)) == -1)
        {
            perror("Error redirecting output.");
            close(fd);
            free(cmd);
            return -1;
        }

        close(fd);
    }
}
// redirection of the standard error
else if (strcmp(token, "2>") == 0)
{
    token = strtok(NULL, " ");

    if (token)
    {
        error_redirect = 1;
```

```
        int fd = open(token, O_WRONLY | O_CREAT  | O_TRUNC, 0644);

        if (fd == -1)
        {
          perror("Error opening error file");
          free (cmd); // daph added this (for reprintf("%s", shell->prompt);
          return -1;
        }

        dup2(fd, fileno(stderr));

        // error handling - unable to redirect standard error
        if (dup2(fd, fileno(stderr)) == -1)
        {
          perror("Error redirecting error.");
          close(fd);
          free(cmd);
          return -1;
        }

        close(fd);
    }
}
```

```
        token = strtok(NULL, " ");
    }

    free(cmd);

    // Restore the standard output and standard error
    if (output_redirect || error_redirect)
    {
        dup2(stdout_backup, fileno(stdout)); // Restore standard output
        dup2(stderr_backup, fileno(stderr)); // Restore standard error
    }

    // error handling - unable to restore file descriptors
    if (output_redirect || error_redirect)
    {
        if (dup2(stdout_backup, fileno(stdout)) == -1 || dup2(stderr_backup, fileno(stderr)) == -1)
        {
            perror("Error restoring file descriptors");
            return -1;
        }
    }

    return (output_redirect || error_redirect) ? (stdout_backup | (stderr_backup << 16)) : 0;
}
```

```
// ----------------------------------------------------------

/*
 * adding the commands entered into a command_history array
 */
void add_history(Shell* shell, const char *command)
{
   if (total_history < MAX_HISTORY_LENGTH)
   {
      strcpy(shell->command_history[total_history], command);
      total_history++;
   }
   else
   {
      // deallocate memory
      free(shell->command_history[0]);

      // if the history is full, overwrite the oldest command in a circular manner
      strcpy(shell->command_history[history_index], command);
      history_index = (history_index + 1) % MAX_HISTORY_LENGTH;
   }
}

// ----------------------------------------------------------
```

```
/*
 * command execution for the !12 and !string commands
 */
void execute_history_command(char *arg[])
{
   pid_t pid = fork();

   if (pid<0)
   {
      perror("fork()");
      exit(0);
   }
   else if (pid == 0)
   {
      printf("Child process executing: %s", arg[0]);

      if (execvp(arg[0],arg) < 0)
      {
         perror("execvp()");
         exit(1);
      }
   }
   else
   {
      int status;
```

```
        waitpid(pid, &status, 0);
    }
}
// ------------------------------------------------------------

/*
 * providing the nth command entered
 */
char * history_by_number(Shell* shell, int num)
{
    if (num > 0 && num <= total_history)
    {
        return shell->command_history[num - 1];
    }
    return NULL;
}

// ------------------------------------------------------------

/*
 * providing output of the nth command
 */
void execute_history_by_number(Shell* shell, int num)
{
    // finding the nth command
```

```
    char *command_to_execute = shell->command_history[num -1];
    char *command[MAX_ARGUMENT_LENGTH];

    // getting the output of the nth command
    int num_args = tokenise_command(command_to_execute, command);
    execute_history_command(command);
}

// ----------------------------------------------------------

/*
 * providing the string command entered
 */
char* history_by_string(Shell* shell, const char *str)
{
    for (int i = total_history - 1; i >= 0; --i)
    {
        if (strncmp(shell->command_history[i], str, strlen(str)) == 0)
        {
            return shell->command_history[i];
        }
    }
    return NULL;
}
```

```
// ------------------------------------------------------------

/*
 * providing output of the string command
 */
void execute_history_by_string(Shell* shell, const char *str)
{
    char* command_to_execute = NULL;

    // finding the string command
    for (int i = total_history - 1; i >= 0; --i)
    {
        if (strncmp(shell->command_history[i], str, strlen(str)) == 0)
        {
            command_to_execute = shell->command_history[i];
        }
    }

    // getting the output of the string command
    if (command_to_execute != NULL)
    {
        char *command[MAX_ARGUMENT_LENGTH];

        int num_args = tokenise_command(command_to_execute, command);
```

```
        execute_history_command(command);
    }
}

// ------------------------------------------------------------

/*
 * provide all the history entered
  */
void execute_history(Shell* shell)
{
    printf("Command History: \n");

    for (int i = 0; i < total_history; i++)
    {
        printf("%d: %s \n", i + 1, shell->command_history[i]);
    }
}

// ------------------------------------------------------------

/*
 * shell pipeline - '|'
 */
int execute_piped_commands(char* commands[], int num_commands)
```

```
{
    // need more than 2 commands for shell pipeline
    if (num_commands < 2)
    {
        fprintf(stderr, "Not enough commands for piping.\n");
        return -1;
    }

    int pipes[num_commands - 1][2];

    for (int i = 0; i < num_commands - 1; i++)
    {
        // error handling if pipe cannot be created
        if (pipe(pipes[i]) < 0)
        {
            perror("pipe");
            return -1;
        }
    }

    for (int i = 0; i < num_commands; i++)
    {
        pid_t pid = fork();

        if (pid == -1)
```

```
{
    // error handling - forking failed
    perror("fork");
    return -1;
}
else if (pid == 0)
{
    // child process
    if (i > 0)
    {
        // set stdin from the previous pipe
        dup2(pipes[i - 1][0], 0);
        close(pipes[i - 1][0]);
    }
    if (i < num_commands - 1)
    {
        // set stdout to the next pipe
        dup2(pipes[i][1], 1);
        close(pipes[i][1]);
    }

    // close all the other pipes
    for (int j = 0; j < num_commands - 1; j++)
    {
        if (j != i - 1)
```

```
            close(pipes[j][0]);
          if (j != i)
            close(pipes[j][1]);
        }

        // execute the command after tokinisation
        char* args[100];
        int num_args = tokenise_command(commands[i], args);

        if (num_args < 0)
        {
          exit(1);
        }

        // error handling - execution failed
        execvp(args[0], args);
        perror("execvp");
        exit(1);
      }
    }

    // Parent process: close all pipes
    for (int i = 0; i < num_commands - 1; i++)
    {
      close(pipes[i][0]);
```

```
        close(pipes[i][1]);
    }

    // Wait for all child processes
    for (int i = 0; i < num_commands; i++)
    {
        wait(NULL);
    }

    return 0;
}
// -------------------------------------------------------------

/*
 * command execution for background &, redirection < > 2>, widlcard *.? and other commands
 */
int executeCommand(Shell* shell, const char* command)
{
    int exitCode = 0;
    int background = 0;
    char* modifiedCommand = strdup(command);

    // Check for background execution
    if (modifiedCommand[strlen(modifiedCommand) - 1] == "&")
    {
```

```
        background = 1;
        modifiedCommand[strlen(modifiedCommand) - 1] = '\0'; // Remove the '&' character
    }

    if (background)
    {
        pid_t pid = fork();
        if (pid == -1)
        {
            perror("fork() error");
            free(modifiedCommand);
            return -1;
        }
        else if (pid == 0)
        {
            execlp("/bin/sh", "sh", "-c", modifiedCommand, (char*)0);
            perror("execlp() error");
            exit(EXIT_FAILURE);
        }
        else
        {
            printf("Background job started with PID: %d\n", pid);
            free(modifiedCommand);
            return 0; // Return immediately, do not wait for child
        }
```

```
}

// Handle redirection
int redirection_mask = handleRedirection(modifiedCommand);

// Check for wildcards and expand
int numExpanded;
char** expandedCommands = expandWildcards(modifiedCommand, &numExpanded);

if (expandedCommands)
{
    for (int i = 0; i < numExpanded; i++)
    {
        if (expandedCommands[i])
        {
            // Execute the expanded command
            pid_t pid = fork();
            if (pid == -1)
            {
                perror("fork() error");
                exitCode = -1;
                break;
            }
            else if (pid == 0)
            {
```

```
            execlp("/bin/sh", "sh", "-c", expandedCommands[i], (char*)0);
            perror("execlp() error");
            exit(EXIT_FAILURE);
          }
          else
          {
            int status;
            waitpid(pid, &status, 0);
            exitCode = WEXITSTATUS(status);
          }
          free(expandedCommands[i]);
        }
      }
      free(expandedCommands);
    }
    else
    {
      // Execute the command without wildcard expansion
      pid_t pid = fork();
      if (pid == -1)
      {
        perror("fork() error");
        free(modifiedCommand);
        return -1;
      }
```

```
        else if (pid == 0)
        {
            execlp("/bin/sh", "sh", "-c", modifiedCommand, (char*)0);
            perror("execlp() error");
            exit(EXIT_FAILURE);
        }
        else
        {
            int status;
            waitpid(pid, &status, 0);
            exitCode = WEXITSTATUS(status);
        }
    }

    free(modifiedCommand);
    return exitCode;
}

// -----------------------------------------------------------
/*
 * signal handling for CTRL-C, CTRL-Z and CTRL-\
 */
volatile sig_atomic_t signalReceived = 0;
void handleSignal(Shell* shell, int signum)
{
```

```
    signalReceived = 1;

    write(STDOUT_FILENO, "\nSignal caught, but continuing...\n", 35);

}


// ---------------------------------------------------------


/*

 * handling zombie processes

 */

void sigchld_handler(Shell* shell, int signum)

{

    int status;


    while (waitpid(-1, &status, WNOHANG) > 0)

    {

        // Reap the zombie process

    }

}

// ---------------------------------------------------------


/*

 * tokenising - dividing the commands into tokens

 */

int tokenise_command(char* input, char* tokens[])

{
```

```
// copy the input
char *input_copy = strdup(input);

// error handling - unable to copy input
if(input_copy == NULL)
{
   perror("strdup()");
   exit(1);
}

int num_arg = 0;
char* token = strtok(input_copy, " ");

while (token != NULL && num_arg < MAX_ARGUMENT_LENGTH - 1)
{
   // dynamically allocate memory
   tokens[num_arg] = malloc(sizeof(char)* MAX_TOKEN_LENGTH);

   // error handling - dynamic memory allocation failed
   if(tokens[num_arg] == NULL)
   {
      perror("malloc()");
      exit(1);
   }
```

```
        strncpy(tokens[num_arg], token, MAX_TOKEN_LENGTH -1);
        tokens[num_arg][MAX_TOKEN_LENGTH -1] = '\0';

        token = strtok(NULL, " ");
        num_arg++;
    }

    tokens[num_arg] = NULL;

    // deallocate memory
    free(input_copy);

    return num_arg;
}

// ------------------------------------------------------------
/*
 * differentiate the commands and execute them
 */

void runShell(Shell* shell)
{
    // signal handling for CTRL-C, CTRL-Z and CTRL-\
    struct sigaction sa;
```

```
// Set up the sigaction structure
sa.sa_handler = handleSignal;  // Set the handler function
sigemptyset(&sa.sa_mask);      // Initialize the mask to empty
sa.sa_flags = 0;               // No special flags

// Set up signal handling for SIGINT (Ctrl+C)
if (sigaction(SIGINT, &sa, NULL) == -1)
{
    perror("Error setting SIGINT");
    // Handle error
}

// Set up signal handling for SIGQUIT (Ctrl+\)
if (sigaction(SIGQUIT, &sa, NULL) == -1)
{
    perror("Error setting SIGQUIT");
    // Handle error
}

// Set up signal handling for SIGTSTP (Ctrl+Z)
if (sigaction(SIGTSTP, &sa, NULL) == -1)
{
    perror("Error setting SIGTSTP");
    // Handle error
}
```

```
int exitShell = 0;

while (!exitShell)
{

    printf("%s", shell->prompt);

    char input[MAX_PROMPT_LENGTH];

    // handling slow system calls e.g background executions and signals being caught
    int again = 1;
    char *linept; // pointer to the line buffer

    while (again)
    {
        again = 0;
        linept = fgets(input, sizeof(input), stdin);

        if (linept == NULL)
        {
            if(errno == EINTR)
            {
                again = 1; // signal interruption, read again;
                printf("%s", shell->prompt);
```

```
        }
        else
        {
            printf("Invalid input entered. \n");
            exit(1);
        }
    }
}

// removing the new line
input[strcspn(input, "\n")] = '\0';

// adding the commands into a command_history array if '!' and 'history' is not entered
if (input[0] != '!' && (strcmp(input, "history") != 0))
{
    add_history(shell, input);
}

// tokenising the commands
char *tokenise[100];
int token_num = 0;

token_num = tokenise_command(input, tokenise);

// prompt change
```

```
if (strncmp(input, "prompt", 6) == 0)
{
    changePrompt(shell, input + 6);
}
// directory walk
else if (strncmp(input, "cd", 2) == 0)
{
    // Find the start of the path argument
    const char* path = input + 2;

    while (*path == ' ')
    {
        // Skip leading spaces
        path++;
    }

    // If there's no path argument, path will point to '\0' (end of string)
    if (*path == '\0' || strcmp(path, " ") == 0)
    {
        path = NULL;  // Handle 'cd' with no arguments to go to HOME
    }

    if (!changeDirectory(shell, path))
    {
        printf("Directory change failed.\n");
```

```
    }
}
// print current directory
else if (strcmp(input, "pwd") == 0)
{
    printCurrentDirectory(shell);


}
// exit the program
else if (strcmp(input, "exit") == 0)
{
    printf("Exiting the shell.\n");
    exitShell = 1;
}
// history - print out all the commands entered
else if (strcmp(input, "history") == 0)
{
    execute_history(shell);
}
else if (input[0] == '!')
{
    // if the input is a digit
    if (isdigit(input[1]))
    {
        // get the nth number entered
```

```
    int num_command = atoi(input+1);
    char *commands = history_by_number(shell, num_command);

    if (commands != NULL)
    {
        printf("%s \n", history_by_number(shell, num_command));

        execute_history_by_number(shell, num_command);
    }
    else
    {
        printf("Invalid command number entered. \n");
        continue;
    }
}
// if the input is a string
else
    // if (strncmp(command_history[i], str, strlen(str)) == 0)
{
    // get the string entered
    char *commands = history_by_string(shell, input + 1);

    if (commands != NULL)
    {
        printf("%s \n", history_by_string(shell, input + 1));
```

```
            execute_history_by_string(shell, input+1);
        }
        else
        {
            printf("Invalid command string entered. y\n");
            continue;
        }
    }
}
// shell pipeline
else if (strcmp(input, "|") == 0)
{
    int num_commands = 0;

    while (input[num_commands] != NULL && strcmp(input[num_commands], "|") == 0)
    {
        num_commands++;
    }

    // allocate memory for the commands array
    char* commands[num_commands];
    execute_piped_commands(commands, num_commands);
}
// executing other commands e.g ls, ps, who
else
```

```
        {
            if (executeCommand(shell, input) == -1)
            {
                printf("Unknown command: %s\n", input);

            }
        }
    } // end of exitShell loop
}

// -----------------------------------------------------------


/*
 * deallocate memory for the 'Shell' struct
 */
void destroyShell(Shell* shell)
{

    if (shell)
    {
        free(shell);
    }
}

// -----------------------------------------------------------
```

3. command.h

```
#define MAX_NUM_COMMANDS  1000

#define MAX_TOKENS 100

#define MAX_TOKEN_LENGTH 100


// command separators

#define pipeSep  "|"                         // pipe separator "|"

#define conSep   "&"                         // concurrent execution separator "&"

#define seqSep   ";"                         // sequential execution separator ";"


struct CommandStruct

{

   int first;        // index to the first token in the array "token" of the command

   int last;         // index to the first token in the array "token" of the command

   char *sep;           // the command separator that follows the command,  must be one of "|", "&",
and ";"

   char **argv;      // an array of tokens that forms a command

   char *stdin_file;   // if not NULL, points to the file name for stdin redirection

   char *stdout_file;  // if not NULL, points to the file name for stdout redirection

};


typedef struct CommandStruct Command;  // command type



// purpose:
```

```
//                separate the list of token from array "token" into a sequence of commands, to be
//                stored in the array "command".
//
// return:
//                1) the number of commands found in the list of tokens, if successful, or
//                2) -1, if the the array "command" is too small.
//                3) < -1, if there are following syntax errors in the list of tokens.
//                        a) -2, if any two successive commands are separated by more than one
command separator
//                        b) -3, the first token is a command separator
//                        c) -4, the last command is followed by command separator "|"
//
// assume:
//                the array "command" must have at least MAX_NUM_COMMANDS number of
elements
//
//  note:
//                1) the last command may be followed by "&", or ";", or nothing. If nothing is
//                  followed by the last command, we assume it is followed by ";".
//                2) if return value, nCommands >=0, set command[nCommands] to NULL,
//
int separateCommands(char *token[], Command command[]);
```

4. command.c

```
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>


#include "command.h"


// return 1 if the token is a command separator

// return 0 otherwise

//


void initialiseCommand(Command *cp)

{

   cp->first = 0;

   cp->last =0;

   cp->sep = NULL;

   cp-> stdin_file = NULL;

   cp->stdout_file = NULL;

   cp-> argv = malloc(sizeof(char*)*MAX_TOKENS);


   int i;

   for (int i = 0; i < MAX_TOKENS; ++i)
```

```
    {
        cp->argv[i] = malloc(sizeof(char) * MAX_TOKEN_LENGTH);
    }
}

void freeCommand(Command *cp)
{
    for (int i = 0; i < MAX_TOKENS; ++i)
    {
        free(cp->argv[i]);
    }

    free(cp->argv);

    cp->argv = NULL;

    cp->first = NULL;
    cp->last = NULL;
    cp->sep = NULL;
    cp->stdin_file = NULL;
    cp->stdout_file = NULL;
}


int separator(char *token)
```

```
{
   int i = 0;
   char *commandSeparators[] = {pipeSep, conSep, seqSep, NULL};

   while (commandSeparators[i] != NULL)
   {
      if (strcmp(commandSeparators[i], token) == 0)
      {
         return 1;
      }
      ++i;
   }

   return 0;
}

// fill one command structure with the details
//
void fillCommandStructure(Command *cp, int first, int last, char *sep)
{
   cp->first = first;
   cp->last = last - 1;
   cp->sep = sep;
}
```

```
// process standard in/out redirections in a command
void searchRedirection(char *token[], Command *cp)
{
   int i;

   for (i=cp->first; i<=cp->last; ++i)
   {
      if (strcmp(token[i], "<") == 0)
      {
         // standard input redirection
         cp->stdin_file = token[i+1];
         ++i;
      }
      else if (strcmp(token[i], ">") == 0)
      {
         // standard output redirection
         cp->stdout_file = token[i+1];
         ++i;
      }
   }
}

// build command line argument vector for execvp function
void buildCommandArgumentArray(char *token[], Command *cp)
{
```

```
int n = (cp->last - cp->first + 1); // the number of tokens in the command

if (cp->stdin_file != NULL)  // remove 2 tokens for stdin redirection
{
   n -= 2;
}

if (cp->stdout_file != NULL) // remove 2 tokens for stdout redirection
{
   n -=2;
}

n = n + 1; // the last element in argv must be a NULL

// re-allocate memory for argument vector
cp->argv = (char **) realloc(cp->argv, sizeof(char *) * n);

if (cp->argv == NULL)
{
   perror("realloc");
   exit(1);
}

// build the argument vector
int i;
```

```
    int k = 0;

    for (i=cp->first; i<= cp->last; ++i )
    {
        if (strcmp(token[i], ">") == 0 || strcmp(token[i], "<") == 0)
        {
            ++i;    // skip off the std in/out redirection
        }
        else
        {
            cp->argv[k] = token[i];
            ++k;
        }
    }
    cp->argv[k] = NULL;
}

int separateCommands(char *token[], Command command[])
{
    int i;
    int nTokens;

    // find out the number of tokens
    i = 0;
```

```
while (token[i] != NULL)
    ++i;
nTokens = i;

// if empty command line
if (nTokens == 0)
    return 0;

// check the first token
if (separator(token[0]))
    return -3;

// check last token, add ";" if necessary
if (!separator(token[nTokens-1]))
{
    token[nTokens] = seqSep;
    ++nTokens;
}

int first=0;   // points to the first tokens of a command
int last;      // points to the last  tokens of a command
char *sep;     // command separator at the end of a command
int c = 0;     // command index

for (i=0; i<nTokens; ++i)
```

```
{
   last = i;

   if (separator(token[i]))
   {
      sep = token[i];

      if (first==last)  // two consecutive separators
         return -2;

      fillCommandStructure(&(command[c]), first, last, sep);
      ++c;

      first = i+1;
   }
}

// check the last token of the last command
if (strcmp(token[last], pipeSep) == 0)
{
   // last token is pipe separator
   return -4;
}

// calculate the number of commands
```

```
    int nCommands = c;


    // handle standard in/out redirection and build command line argument vector
    for (i=0; i<nCommands; ++i)
    {
        searchRedirection(token, &(command[i]));
        buildCommandArgumentArray(token, &(command[i]));
    }


    return nCommands;
}
```