Custom Shell Documentation

Programming Project

Raheim Burkett 2210198

Raul Miller 2210179

Breanna Smith 2207630

Iyana Taylor 2209566

School of Computing and Information Technology (SCIT)

Faculty of Engineering and Computing (FENC)

University of Technology, Jamaica (UTech)

CIT3002 Operating Systems (UM4)

Tutor: Mr. Philip Smith

November 22, 2024

Table of Contents

Source Code

Source code submitted via UTech Moodle by the name "**`custom_shell.c`**". Each group member's name is commented in the first few lines.

Details of Shell Features and Testing

Details

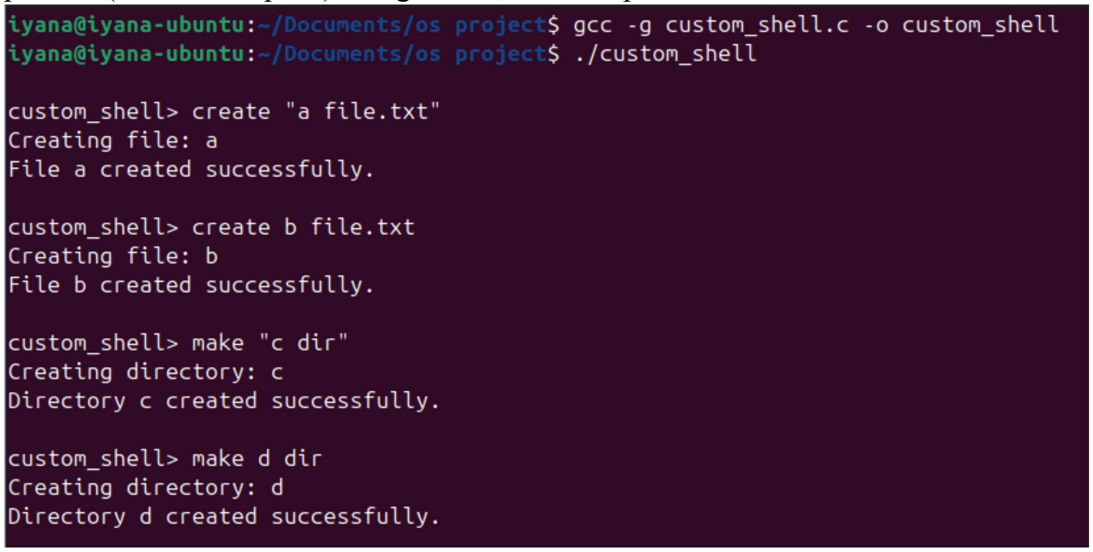
Our Custom Shell Project is a user-centric program designed to mirror the functionality of standard Unix/Linux shells while being simple and practical. It's developed in C for a Linux environment and this shell supports essential file system operations to manipulate files and directories, environment variable handling is also included. Key features include creating, renaming, and deleting files, making, changing and removing directories, managing file permissions, and setting and retrieving environment variables. The shell also includes a helpful command reference menu (help) for user guidance and employs robust error handling to manage invalid inputs and unforeseen issues effectively, at almost every step of the code.

Modularity is used extensively throughout the code, with functions clearly defined for each operation to ensure readability and ease of maintenance. Error messages are descriptive, aiming to provide immediate feedback and suggestions for user correction.

The point of this shell is to help get a deeper understanding and hands-on experience with core operating system principles, like system calls, process control, and file system interactions and to explore the mechanics behind modern command-line interfaces.

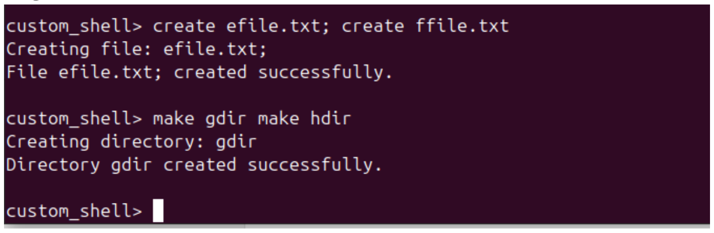
Testing

- Passing file and directory names with spaces – the spaces are ignored, only the first portion (before the space) is registered, even in quotes.



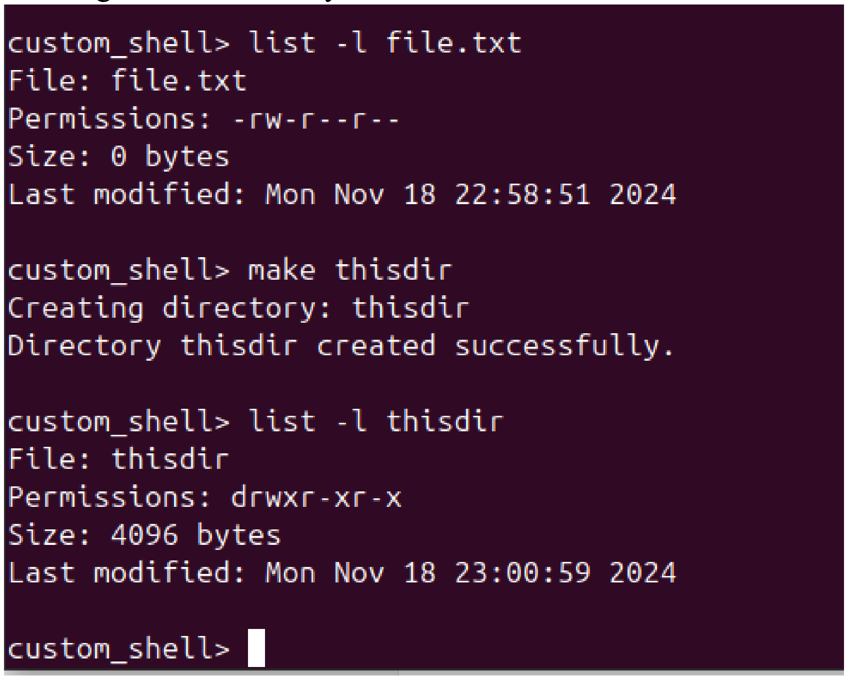
```
iyana@iyana-ubuntu:~/Documents/os project$ gcc -g custom_shell.c -o custom_shell
iyana@iyana-ubuntu:~/Documents/os project$ ./custom_shell

custom_shell> create "a file.txt"
Creating file: a
File a created successfully.

custom_shell> create b file.txt
Creating file: b
File b created successfully.

custom_shell> make "c dir"
Creating directory: c
Directory c created successfully.

custom_shell> make d dir
Creating directory: d
Directory d created successfully.
```

- More than one command in the same command line – only the first command is registered.



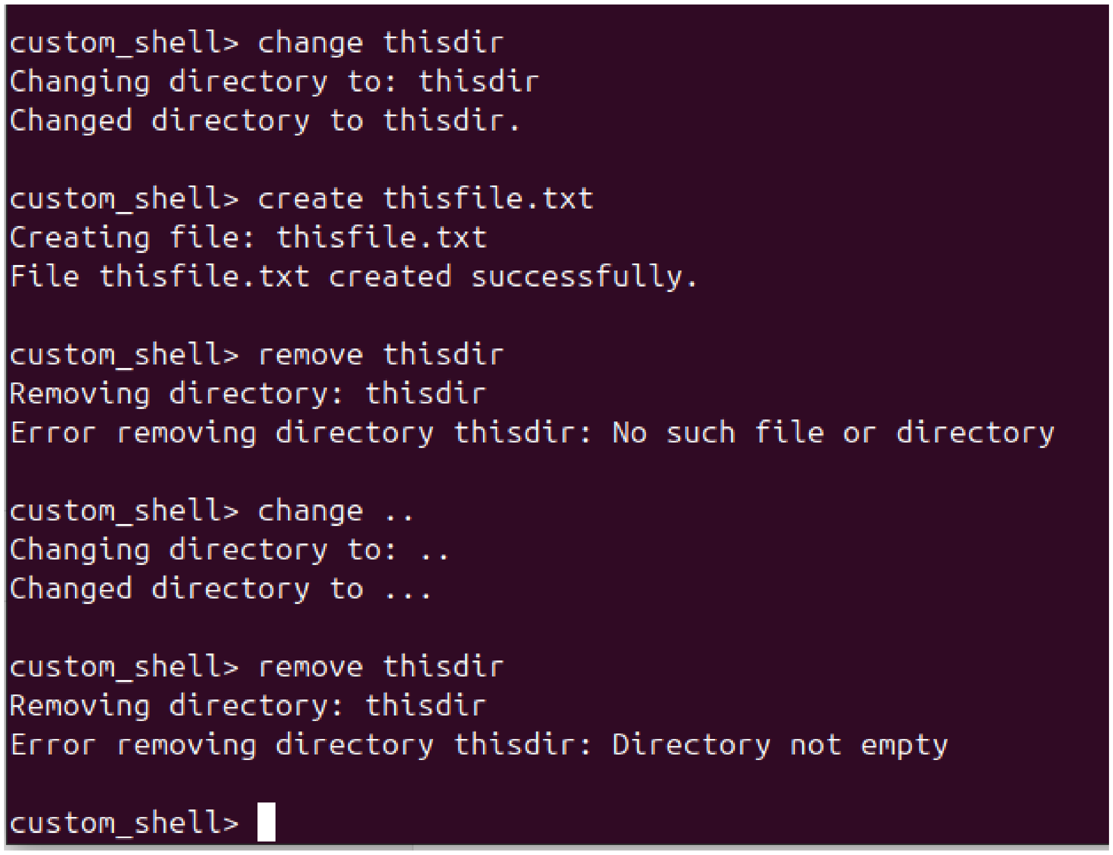
```
custom_shell> create efile.txt; create ffile.txt
Creating file: efile.txt;
File efile.txt; created successfully.

custom_shell> make gdir make hdir
Creating directory: gdir
Directory gdir created successfully.

custom_shell>
```

- Viewing file and directory details



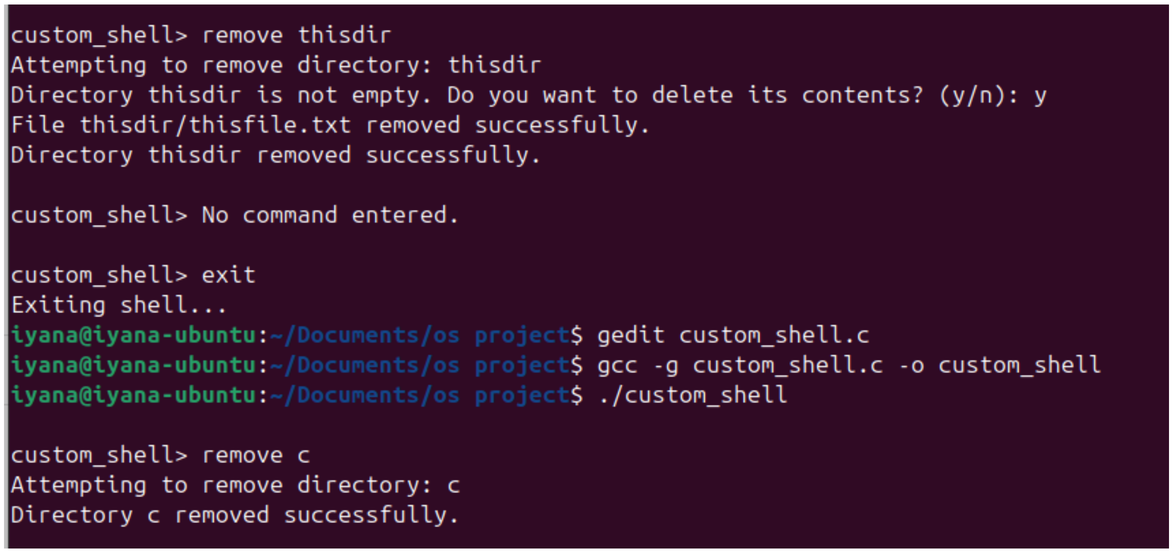
```
custom_shell> list -l file.txt
File: file.txt
Permissions: -rw-r--r--
Size: 0 bytes
Last modified: Mon Nov 18 22:58:51 2024

custom_shell> make thisdir
Creating directory: thisdir
Directory thisdir created successfully.

custom_shell> list -l thisdir
File: thisdir
Permissions: drwxr-xr-x
Size: 4096 bytes
Last modified: Mon Nov 18 23:00:59 2024

custom_shell>
```

- Changing directory – this directory was created previously, then changed into. A file was created inside it successfully. Used "**`change ..`**" to go back to previous working directory then tried to remove the directory with content inside. It could not be removed.



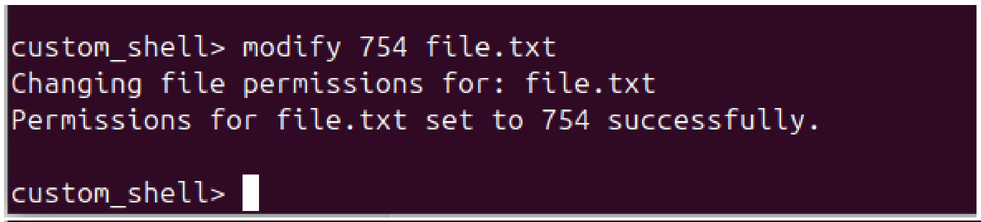
```
custom_shell> change thisdir
Changing directory to: thisdir
Changed directory to thisdir.

custom_shell> create thisfile.txt
Creating file: thisfile.txt
File thisfile.txt created successfully.

custom_shell> remove thisdir
Removing directory: thisdir
Error removing directory thisdir: No such file or directory

custom_shell> change ..
Changing directory to: ..
Changed directory to ...

custom_shell> remove thisdir
Removing directory: thisdir
Error removing directory thisdir: Directory not empty

custom_shell> 
```

- The "**`remove_directory`**" method was updated to allow the removal of directories with and without content.

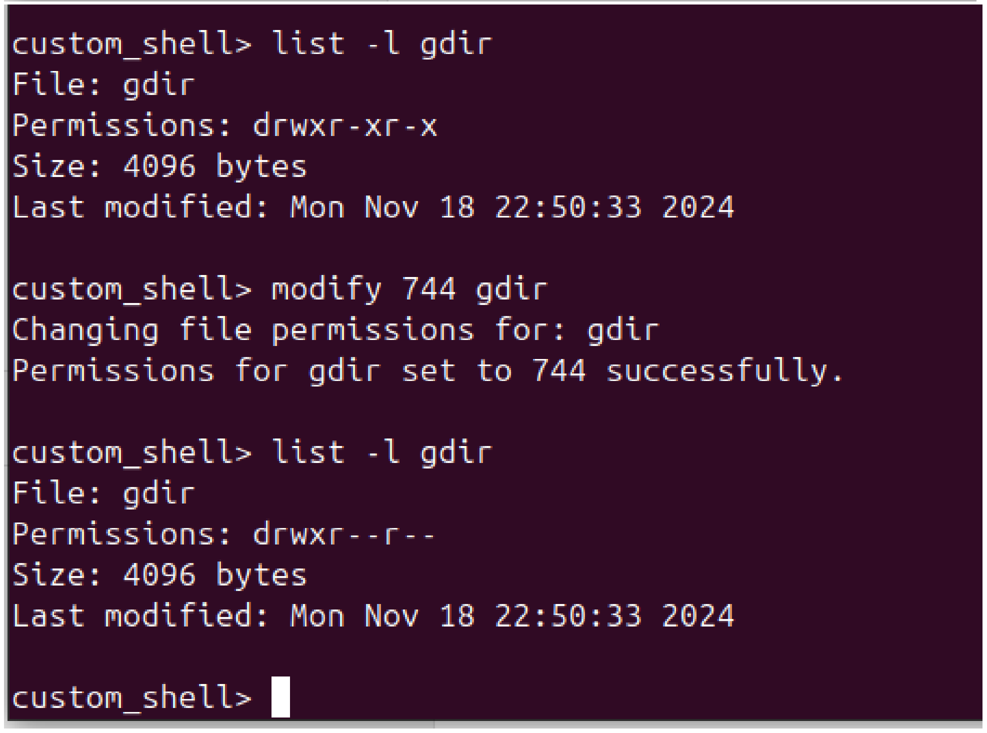


```
custom_shell> remove thisdir
Attempting to remove directory: thisdir
Directory thisdir is not empty. Do you want to delete its contents? (y/n): y
File thisdir/thisfile.txt removed successfully.
Directory thisdir removed successfully.

custom_shell> No command entered.

custom_shell> exit
Exiting shell...
iyana@iyana-ubuntu:~/Documents/os project$ gedit custom_shell.c
iyana@iyana-ubuntu:~/Documents/os project$ gcc -g custom_shell.c -o custom_shell
iyana@iyana-ubuntu:~/Documents/os project$ ./custom_shell

custom_shell> remove c
Attempting to remove directory: c
Directory c removed successfully.
```

- Setting and listing file or directory permissions

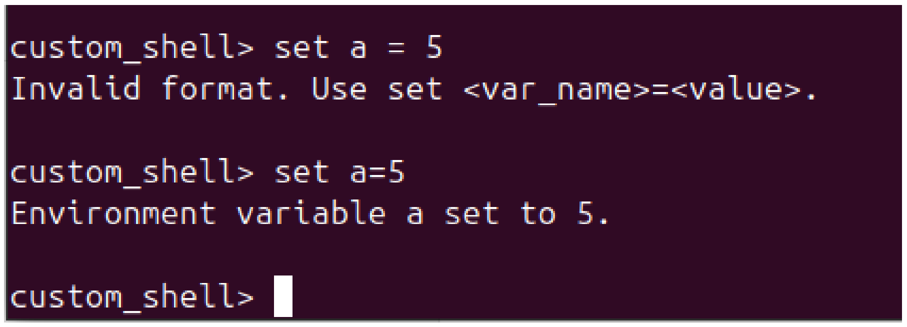


```
custom_shell> modify 754 file.txt
Changing file permissions for: file.txt
Permissions for file.txt set to 754 successfully.

custom_shell> 
```



```
custom_shell> list -l gdir
File: gdir
Permissions: drwxr-xr-x
Size: 4096 bytes
Last modified: Mon Nov 18 22:50:33 2024

custom_shell> modify 744 gdir
Changing file permissions for: gdir
Permissions for gdir set to 744 successfully.

custom_shell> list -l gdir
File: gdir
Permissions: drwxr--r--
Size: 4096 bytes
Last modified: Mon Nov 18 22:50:33 2024

custom_shell> 
```

- Setting variable – variables cannot be set with spaces between the name, the equal sig and the value.



```
custom_shell> set a = 5
Invalid format. Use set <var_name>=<value>.

custom_shell> set a=5
Environment variable a set to 5.

custom_shell> 
```

Architecture

The source code follows a modular and procedural architecture designed in the simulation of a custom shell environment in Linux. There are methods to manage functionalities such as parsing user input, executing commands, and handling errors, these are separate from the methods to manipulate files and directories, modify permissions, list attributes and set and get environment variables.

There are heavy uses of system calls for process creation. The code uses conditional checks and loops for interactive command execution and error handling. Each command entered by the user is processed through a parsing function, with built-in commands handled internally and external commands delegated to system utilities via fork-exec patterns.

The table below shows some of the system commands used:

System Call	Description	Header/ Library
chdir	Changes the current working directory.	`<unistd.h>`
chmod	Changes the permissions of a file or directory.	`<sys/stat.h>`
close	Closes a file descriptor, freeing resources.	`<unistd.h>`
closedir	Closes a directory stream, releasing resources.	`<dirent.h>`
ctime	Converts a time_t value to a human-readable string representing local time.	`<time.h>`
fgets	Reads a line of input from stdin and stores it in the provided buffer.	`<stdio.h>`
fprintf	Writes formatted output to stderr, often used for error messages.	`<stdio.h>`
getenv	Retrieves the value of an environment variable.	`<stdlib.h>`
mkdir	Creates a directory with specified permissions.	`<sys/stat.h>`
open	Opens or creates a file. Can specify flags like O_CREAT, O_WRONLY.	`<fcntl.h>`
opendir	Opens a directory stream for reading its contents.	`<dirent.h>`
readdir	Reads the next entry in a directory stream (e.g., file or subdirectory).	`<dirent.h>`
rename	Renames or moves a file or directory.	`<stdio.h>`
rmdir	Removes an empty directory.	`<unistd.h>`
scanf	Reads formatted input from the user.	`<stdio.h>`
setenv	Sets an environment variable with a value.	`<stdlib.h>`
snprintf	Formats and writes data to a string safely, avoiding buffer overflows.	`<stdio.h>`
stat	Retrieves file or directory attributes, such as size and permissions.	`<sys/stat.h>`
strcmp	Compares two strings for equality, used to check commands.	`<string.h>`

strcspn	Finds the first occurrence of a character in a string. Used here to replace the newline.	`<string.h>`
strlen	Returns the length of a string, used to check for empty input.	`<string.h>`
strtok	Tokenizes a string based on delimiters, used for parsing commands.	`<string.h>`
strtol	Converts a string to a long integer, used here for permissions.	`<stdlib.h>`
unlink	Deletes a file.	`<unistd.h>`

Interfacing with OS Functionality

The code interfaces with the operating system (OS) primarily through **system calls** to handle shell related tasks and **standard library functions** that act as intermediaries between user-level applications and OS kernel functionality.

Developers' Plan, Notes & Recommendations

Plan

A step-by-step development plan was created to ensure building the project was organized and all major shell features were implemented well. It also makes tracking errors and doing tests easier.

1. Foundation and Shell Loop

- Goal: Set up the basic loop and input handling.
- Tasks:
  - Initialize the shell.
  - Create a prompt display function.
  - Implement input reading and basic command parsing.
  - Add a basic `exit` command to end the shell loop.

2. Command Parsing and Structure

- Goal: Break down user input into commands and arguments.
- Tasks:
  - Tokenize user input to separate command and arguments.

- Set up a command structure to handle different commands.

- Start defining function stubs for each type of command (e.g., file operations, directory management).

3. File Operations Implementation

- Goal: Implement basic file operations (`create`, `delete`, `rename`).

- Tasks:

  - Code functions for `create <file_name>`, `delete <file_name>`, and `rename <old_name> <new_name>`.

  - Use system calls like `open()`, `unlink()`, and `rename()` to interact with files.

  - Add error handling for cases like missing arguments or inaccessible files.

4. Directory Management Implementation

- Goal: Implement directory management commands (`make`, `remove`, `change`).

- Tasks:

  - Code functions for `make <dir_name>`, `remove <dir_name>`, and `change <dir_name>`.

  - Use system calls like `mkdir()`, `rmdir()`, and `chdir()`.

  - Implement error handling for invalid directory paths or permissions.

5. File Access and Permissions

- Goal: Implement commands to modify and view file permissions.

- Tasks:

  - Code a function for `modify <permissions> <file_name>` using `chmod()`.

  - Code a function for `list -l` using `stat()` to display file attributes.

  - Test each command and add error handling.

6. Environment Variable Handling

- Goal: Allow users to set and get environment variables within the shell.

- Tasks:

  - Implement `set <var_name>=<value>` to create/update environment variables.

  - Implement `get <var_name>` to retrieve and display the value of a variable.

  - Use standard library functions like `setenv()` and `getenv()`.

7. I/O Redirection and Piping

- Goal: Enable redirection and piping for commands.
- Tasks:
  - Implement I/O redirection using `dup2()` for `<`, `>`, and `>>`.
  - Set up piping using `pipe()` and handle simple command chaining.
  - Add error handling for invalid redirection or pipe syntax.

8. Built-in Commands (Help and Exit)

- Goal: Provide built-in support for essential commands.
- Tasks:
  - Implement `help` to display a list of available commands with usage examples.
  - Ensure `exit` fully terminates the shell process.

9. Comprehensive Error Handling and Testing

- Goal: Make sure the shell is stable and user-friendly.
- Tasks:
  - Check for invalid commands, incorrect syntax, and handle edge cases.
  - Use `perror()` to give clear feedback on failed system calls.
  - Test all functions to confirm they work as expected in various scenarios.

10. Final Review and Documentation

- Goal: Finalize the project with documentation for clarity and maintainability.
- Tasks:
  - Add comments to all code, explaining each function and major code blocks.
  - Include in documentation: setup instructions and usage examples.

Notes

- The custom commands only work for files and directories inside the same directory the custom_shell.c executable file is in (the working directory the executable is in).
- Some of the challenges faced were simple in nature and provided many learning points. One such challenge was with the **`execute_command`** method:
  - The issue: there was a mismatch between the type of argument expected by execute_command() and what's actually being passed.
  - Error explanation: In the declaration at the top (void execute_command(char *input);), execute_command was set up to receive a single char * (a single string).

- However, later in the code (where we define execute_command), it's defined to accept a char ** (an array of strings), because we want it to process both the command and its arguments (like create myfile.txt).
- The solution: update the function declaration of execute_command at the top to take a char ** instead of char *, so it matches the definition and usage.

- There were challenges faced concerning I/O redirection and piping. Different implementations were tested but despite those efforts the logic errors encountered could not be resolved in time.

The first implementation was done using system calls – `dup2()` for I/O redirection and `pipe()` for piping (command chaining in shells). After receiving compilation errors suggesting the wait() and waitpid() commands used were not being recognized, the <sys/wait.h> header was included in the code (these calls are in the POSIX standard library, defined in the afore-mentioned header). But even with the addition of this header I/O redirection and piping via system calls continued to fail. Then we tried rewriting the custom commands to be better integrated with the redirection and piping but that did not yield the desired results, the output-input between commands was not being connected – similar issue in testing when trying to execute more than one command in a single command line. Which the fault of falls on the developers, we did not implement any feature to allow multiple commands in a ingle line to keep up with the simplicity of the overall project. After updating the custom commands, compiling and running the code again then retrying the redirection and command functions were entered, error messages were being displayed in a never-ending loop.

We went on to consider other implementation options:

Option 1: Sequential Execution

Instead of true piping (where the output of one command is passed as input to another), we tried parsing multiple commands separated by | or ; and execute them sequentially, one after another.

Steps:

1. **Split the input on | or ;.**
   - Treat each segment as an independent command.
2. **Iterate through the commands and execute them one by one.**
   - Use existing execute_command() function to handle each command.
3. **Error Handling:**

  - If one command fails, decide whether to stop or continue executing the remaining commands.

Positives:

- Very simple to implement using basic string splitting.
- Doesn't require additional system calls or managing file descriptors.

Example Flow:

Input:
make tdir; change tdir

Process:

- Split into make tdir and change tdir.
- Execute make tdir → check for success.
- Execute change tdir.

Output:
Commands execute independently, and errors are printed as they arise.

Option 2: Command Chaining with Logical Operators

Add support for chaining commands with logical operators like && and ||:

- cmd1 && cmd2: Execute cmd2 only if cmd1 succeeds.
- cmd1 || cmd2: Execute cmd2 only if cmd1 fails.

Implementation:

1. **Split the input on && or ||.**
2. **Evaluate commands based on return values of execute_command().**
3. **Provide appropriate feedback (e.g., success/failure) for logical conditions.**

Positives:

- Doesn't require true piping but adds meaningful functionality.
- Easy to implement using existing execute_command() logic.

Example Flow:

Input:
make tdir && change tdir

Process:

- Execute make tdir.

- If successful, execute change tdir.

But just as before, the second (and any other succeeding commands) were being ignored. The last resort was to have all commands and arguments entered in the same line, and after each is executed, the user is asked if they would like to execute the next command in the line (after the delimiter). This solution would confirm the commands were being received and possibly interpreted by the shell and ensure their execution, but this did not work.

Recommendations

- Get proper confirmation from users before deleting files and removing directories. Its currently implemented for directories that have content but not for those that are empty, nor for files.

- Redo **`read_input`** and **`parse_input`** functions to understand why spaces inside quotes are not being registered.

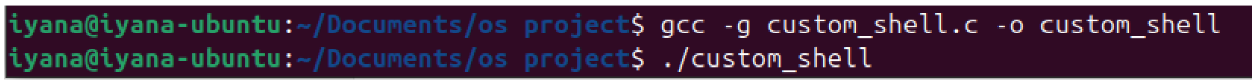
- Display the name of the current working directory, so when the "**`change`**" command is used users know where they are.

- Use an object-oriented language (like C++ or Java) next time so operations can be separated by classification or category or based on specific use cases (general, directory, file), and so the source code will not be so long (makes it easier to manage).

User Guide

General Information

- Ensure a C file with the custom shell source code is saved on your operating system (Linux). Use the following commands to compile and run in the Linux terminal.



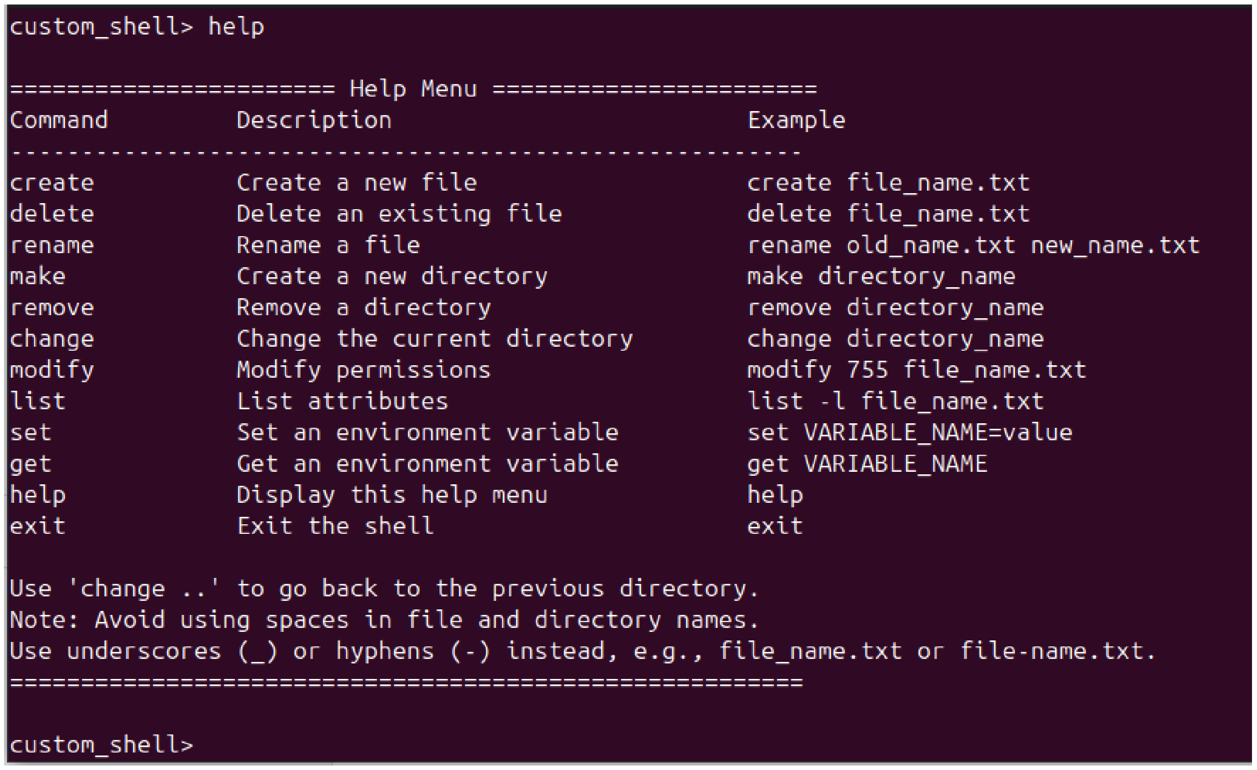
```
iyana@iyana-ubuntu:~/Documents/os project$ gcc -g custom_shell.c -o custom_shell
iyana@iyana-ubuntu:~/Documents/os project$ ./custom_shell
```

- The custom commands only work for files and directories inside the same directory the custom_shell.c executable file is in (the working directory the executable is in).

- When creating files, attaching an extension is recommended, else the argument reads like a directory. The commands to create, rename and delete files will still work without the extension but to be safe, include one.

- Do not give files and directories the same name and always use the correct extensions for files – a recommendation.

- The "**`help`**" command can be used to quickly get information about the custom commands and how to use them.

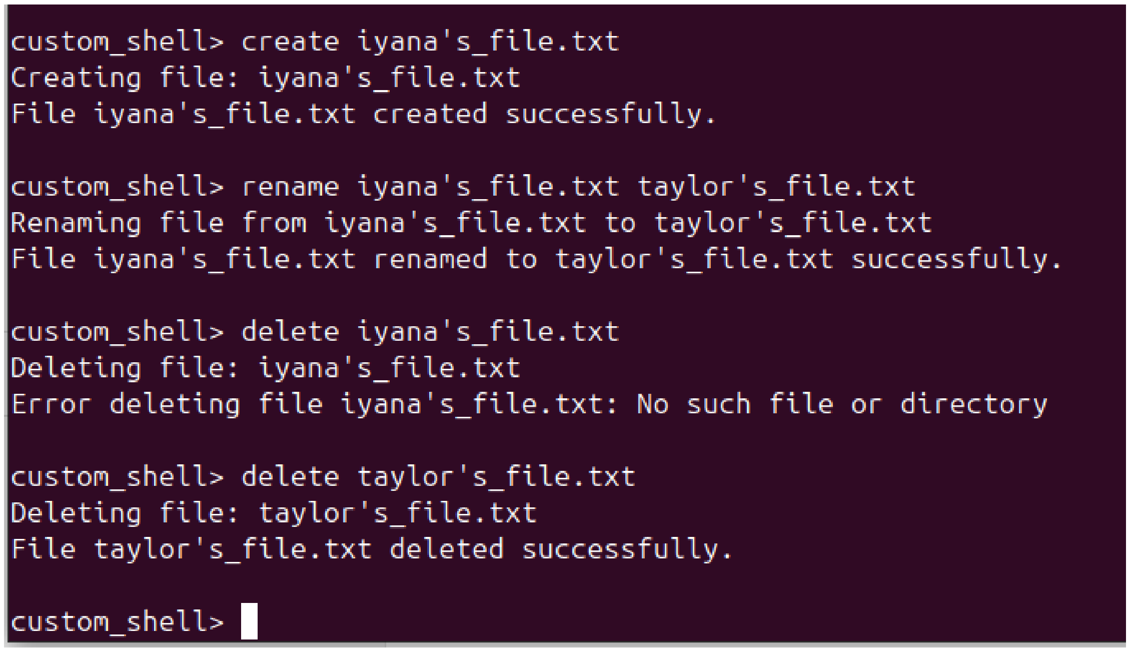


```
custom_shell> help

====================== Help Menu ======================
Command         Description                         Example
--------------------------------------------------------
create          Create a new file                   create file_name.txt
delete          Delete an existing file             delete file_name.txt
rename          Rename a file                       rename old_name.txt new_name.txt
make            Create a new directory              make directory_name
remove          Remove a directory                  remove directory_name
change          Change the current directory        change directory_name
modify          Modify permissions                  modify 755 file_name.txt
list            List attributes                     list -l file_name.txt
set             Set an environment variable         set VARIABLE_NAME=value
get             Get an environment variable         get VARIABLE_NAME
help            Display this help menu              help
exit            Exit the shell                      exit

Use 'change ..' to go back to the previous directory.
Note: Avoid using spaces in file and directory names.
Use underscores (_) or hyphens (-) instead, e.g., file_name.txt or file-name.txt.
========================================================

custom_shell>
```

Files

- There is no space support for file and directory name with spaces, even in quotes. Only the first portion will be registered. Avoid using spaces, if you must include spaces, it is recommended to use underscores (_) or hyphens (-).

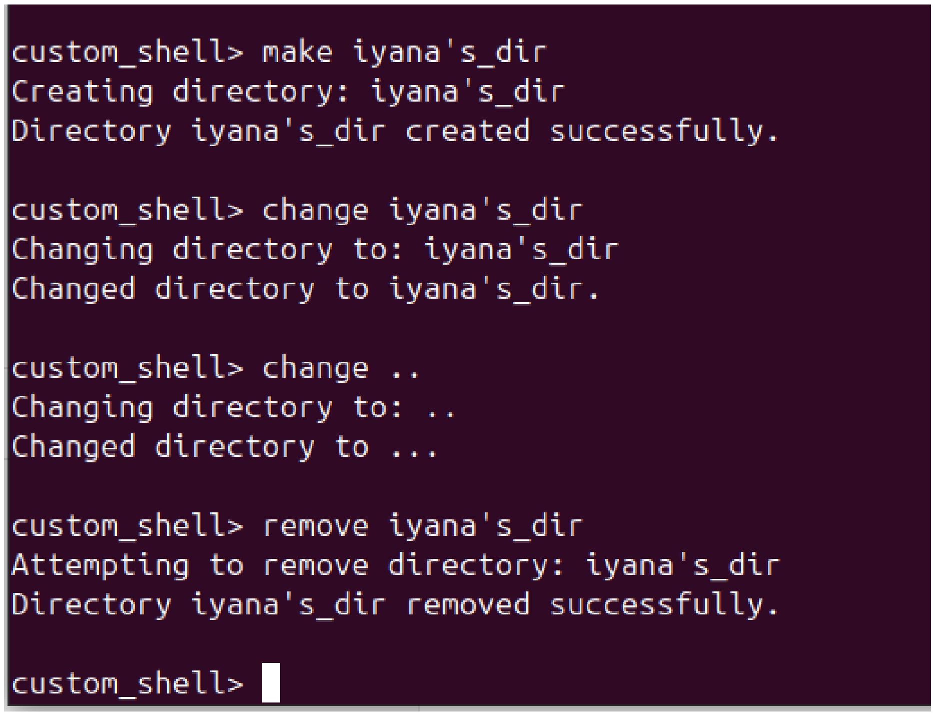
Using file commands:



```
custom_shell> create iyana's_file.txt
Creating file: iyana's_file.txt
File iyana's_file.txt created successfully.

custom_shell> rename iyana's_file.txt taylor's_file.txt
Renaming file from iyana's_file.txt to taylor's_file.txt
File iyana's_file.txt renamed to taylor's_file.txt successfully.

custom_shell> delete iyana's_file.txt
Deleting file: iyana's_file.txt
Error deleting file iyana's_file.txt: No such file or directory

custom_shell> delete taylor's_file.txt
Deleting file: taylor's_file.txt
File taylor's_file.txt deleted successfully.

custom_shell> 
```

Directories

Using directory commands:



```
custom_shell> make iyana's_dir
Creating directory: iyana's_dir
Directory iyana's_dir created successfully.

custom_shell> change iyana's_dir
Changing directory to: iyana's_dir
Changed directory to iyana's_dir.

custom_shell> change ..
Changing directory to: ..
Changed directory to ...

custom_shell> remove iyana's_dir
Attempting to remove directory: iyana's_dir
Directory iyana's_dir removed successfully.

custom_shell> 
```

Modify permissions and List attributes

Permissions Table		
Octal Version	**Letter Version**	**Meaning**
7	rwx	Sets read, write and execute permissions.
6	rw	Sets read and write permissions.
5	rx	Sets read and execute permissions.
4	r	Sets read permission
3	wx	Sets write and execute permissions.
2	w	Sets write permission.
1	x	Sets execute permission.
0	-	No permission.

- In the custom shell, permissions can only be set using the octal version but are displayed in the letter version, its starts with 'd' for directories and '-'for files. Then the permissions for each (users, groups and all others) are displayed as a string, separate them into threes to get the permissions for each.

  Directory example: **`drwxrw-r-x`**

  In the above example, we're viewing the permissions for a directory. The user has read, write and execute permissions; groups have read and write permissions; all others have read and execute permissions.

  File example: **`-rwxrw-r-x`**

  In the above example, we're viewing the permissions for a file. The user has read, write and execute permissions; groups have read and write permissions; all others have read and execute permissions.

- Please remember, permissions are set for users, groups and all others, in that exact order. The first number is for users, the second is for groups and the third for all others.

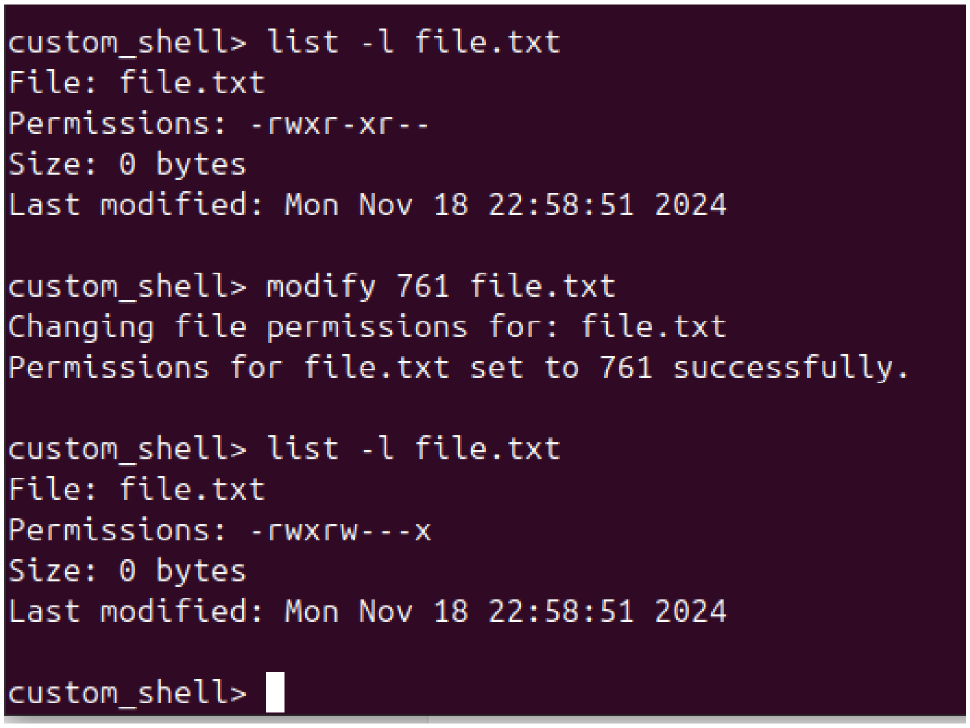
  Example:

  ```
  custom_shell> modify 755 file.txt
  ```

  You can then check the permissions and other attributes like size and last modified time for files and directories by using the list command.

  Example:

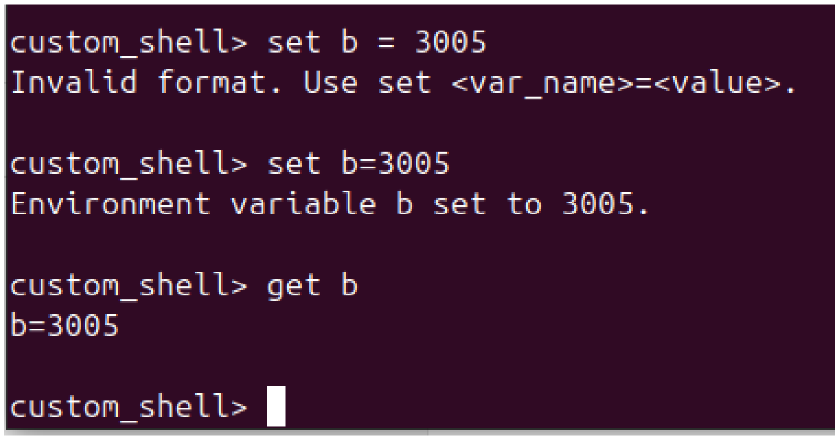
`custom_shell> list -l file.txt`



```
custom_shell> list -l file.txt
File: file.txt
Permissions: -rwxr-xr--
Size: 0 bytes
Last modified: Mon Nov 18 22:58:51 2024

custom_shell> modify 761 file.txt
Changing file permissions for: file.txt
Permissions for file.txt set to 761 successfully.

custom_shell> list -l file.txt
File: file.txt
Permissions: -rwxrw---x
Size: 0 bytes
Last modified: Mon Nov 18 22:58:51 2024

custom_shell>
```

Setting and Getting Environment Variables

- Follow this format to set variables **`<var_name>=value`**. If spaces are entered after the name and equal sign, a message will be displayed.



```
custom_shell> set b = 3005
Invalid format. Use set <var_name>=<value>.

custom_shell> set b=3005
Environment variable b set to 3005.

custom_shell> get b
b=3005

custom_shell>
```

Team Contributions

Name	*Contribution*
Raheim Burkett	create: create a new file delete: delete an existing file
Raul Miller	remove: delete a directory change: change the current working directory
Breanna Smith	rename: rename a file make: create a new directory
Iyana Taylor	modify: modify file permissions list -l: display file attribute handle & manage environment variables implement I/O redirection & piping testing