Operating Systems
Programming Assignment - 03

Introduction and Learning Objectives

This assignment is a hands-on project that will guide you through the creation of a functional command-line interpreter, or "shell," similar to `bash` or `zsh`. Building a shell is a quintessential rite of passage in the study of operating systems, as it touches upon many fundamental concepts: process creation and control, file descriptors, signal handling, I/O redirection, and inter-process communication. You will start with a basic shell that can execute external commands and incrementally add professional features, gaining a deep, practical understanding of how the operating system works from the user's perspective.

The goal of this project is not just to write a single program, but to master the complete development lifecycle of a complex piece of system software. You will learn how core shell features are implemented, how to manage a project's evolution with a disciplined Git workflow using tags and releases, and how to integrate powerful third-party libraries to add professional functionality like command history and tab completion.

By the end of this assignment, you will have demonstrated a comprehensive ability to:

- **Understand and Manage the Process Lifecycle:** Master the `fork-exec-wait` cycle that underpins the execution of nearly every command in a UNIX-like environment.
- **Implement Internal Shell Commands:** Distinguish between external commands and built-in commands (like `cd`, `exit`, `jobs`), and implement the logic for those that must run within the shell's own process.
- **Manage Command History:** Implement the mechanisms to store, display (`history`), and re-execute (`!n`) previous commands.
- **Integrate External Libraries:** Enhance your application by linking against and using a third-party library (GNU Readline) to provide advanced user-interface features.
- **Control I/O and Process Communication:** Implement the powerful shell features of input/output redirection (`<`, `>`) and pipes (`|`) by manipulating file descriptors.
- **Implement Multitasking Features:** Add support for command chaining (`;`) and background process execution (`&`), including the proper handling of "zombie" processes.
- **Create Basic Scripting Logic:** Build a foundational `if-then-else-fi` control structure, taking the first step towards creating a true scripting language.
- **Utilize a Professional Git Workflow:** Use Git tags to mark specific versions of your code and create formal, versioned software releases on GitHub.

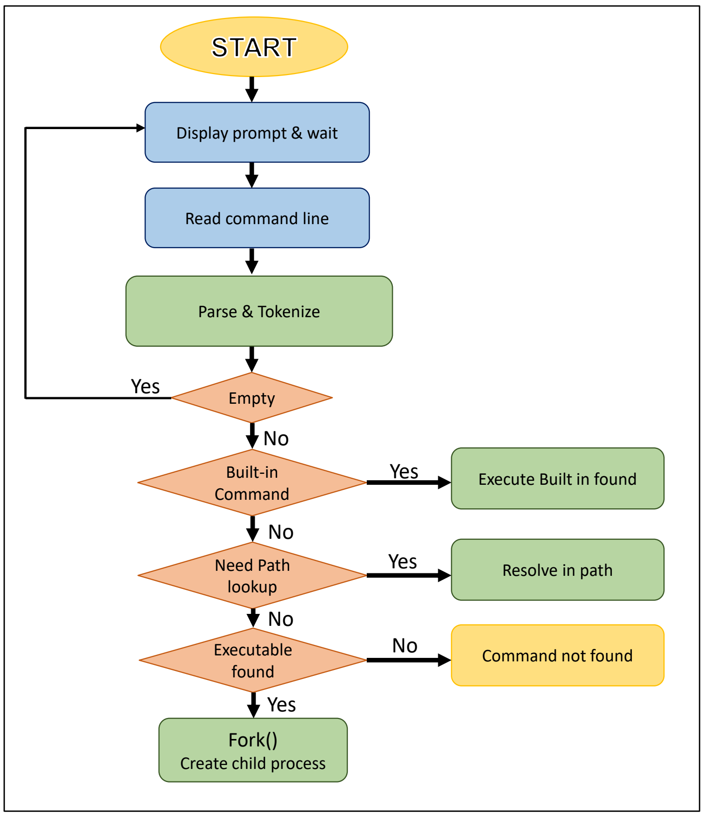
The Scenario & Resources

Your journey will begin with a simple but functional starter project that contains the core logic of a UNIX shell, already organized into a modular structure (`main.c`, `shell.c`, `execute.c`). Your task is to take this base shell and incrementally build upon it, adding a series of advanced features. Each completed feature will represent a new, versioned release of your shell, tracked meticulously through your Git repository.

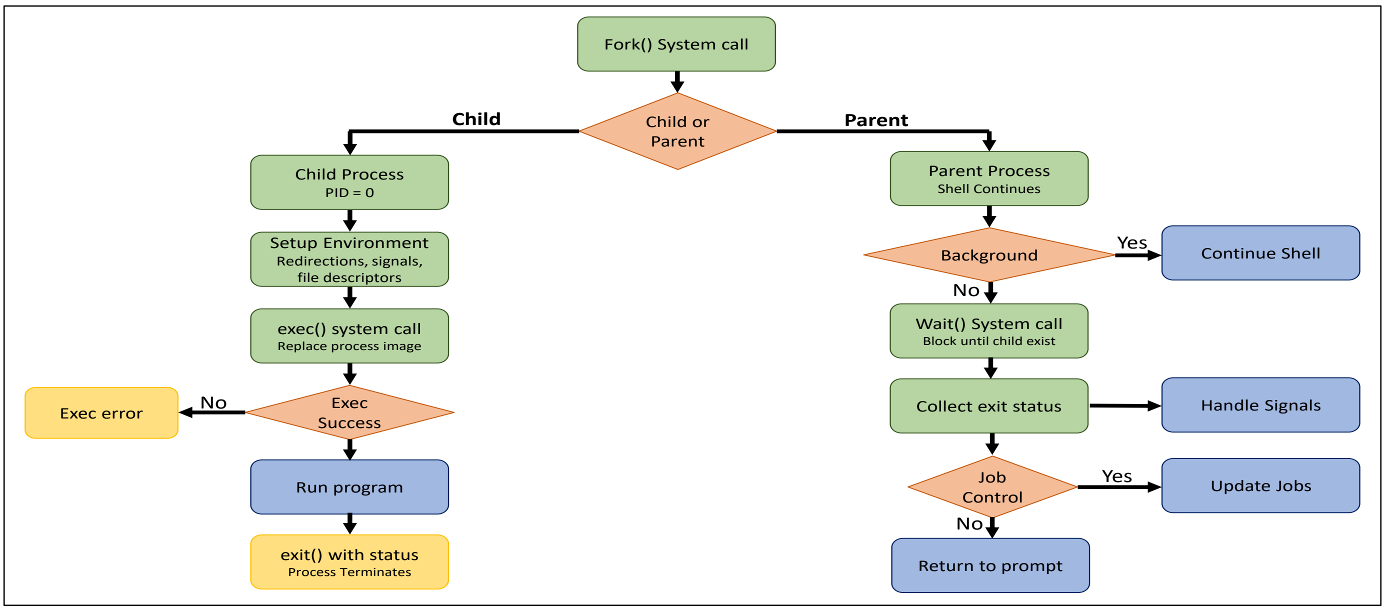
To succeed, you must first familiarize yourself with the foundational concepts and the provided starter code. The following resources are essential and will guide your work:

- To fully grasp the core logic of the starter code you will be working with, it is essential to first watch the **Instructor's video lecture on building a basic shell**. This video explains the `fork-exec-wait` cycle and the modular design of the initial code.
- Your journey begins by cloning the **Starter code repository on GitHub**. This provides you with the complete, working base shell that you will extend.

Flowcharts of the shell's core logic is provided below for your reference.



These two figures shows a flowchart that illustrate the complete lifecycle of executing an external command. The **first chart** (beginning with START) details the preparation phase, where the shell reads user input, parses it, and validates the command, culminating in the `fork()` system call. The **second chart** (beginning with `fork()` System call) then details the execution phase, showing the distinct responsibilities of the new Child Process (which runs the command) and the original Parent Process (which waits for and manages the child).



Do not blindly copy-paste code from AI. You are expected to understand the 'how' and 'why' of every line you submit, and failure to demonstrate this to a TA will result in a zero for the assignment.

Feature-1: The Starting Point (Release 1 - The Base Shell) **(05 Marks)**

What you will do: In this feature, you will set up your own GitHub repository, populate it with the provided starter shell code, build and test the shell, and then create your first formal release.

The base code for this assignment is located within the instructor's `OS-Codes` repository on GitHub. You will retrieve this code and use it to populate your own assignment repository. After setup, your project should contain the following structure and files.

Project Directory Structure

```
ROLL_NO-OS-A03/
├── src/
│   ├── main.c
│   ├── shell.c
│   └── execute.c
├── include/
│   └── shell.h
├── Makefile
└── README.md
```

Task 1: Set Up Repository and Add Starter Code

- Create a public GitHub repository named **ROLL_NO-OS-A03**.
- Clone it to your local machine.
- Get the starter code from the instructor's GitHub repository.
- Copy the following into your project folder:
  - `src`
  - `include`
  - `Makefile`

Task 2: Build and Test

- Navigate into your `ROLL_NO-OS-A03` directory.
- Use `make` to build the project.
- Run the shell: `./bin/myshell`. Test it with simple commands like `ls -l`, `pwd`, and `whoami` to verify it works.

Task 4: Commit and Push the Base Code

- Use the standard `git` workflow to add all the project files to the staging area.
- Commit the files with a message like `"feat: Initial project setup with base shell code"`.
- Push this commit to your `main` branch on GitHub.

Task 5: Create Your First Release (Release 1)

- Create an annotated **git tag** named **v1.0-base** for the initial version and push it to GitHub.
- Go to **Releases → Draft a new release**.
- Select the `v1.0-base` tag you just pushed.

- Title the release `Release 1 - The Base Shell`.
- Add a short description: Initial unmodified starter code.
- Attach the compiled **bin/myshell** binary.
- Finally, publish the release.

Development Workflow for All Subsequent Features

For every feature you implement from this point forward (Feature-2 through Feature-8), you are required to follow the complete professional Git workflow outlined below. These steps are a core part of the assignment and will be checked for each feature.

The workflow for EACH feature is as follows:

1. **Start the Feature:**
   - Ensure your `main` branch is up-to-date by merging the previous feature's branch into it.
   - Create and switch to a new, dedicated branch for the feature you are about to start (e.g., `feature-built-ins`).
2. **Implement and Commit:**
   - Write the code and make the necessary changes to implement the feature's requirements.
   - Build and test your implementation thoroughly.
   - Once the feature is working correctly, commit your changes to your feature branch with a clear, descriptive message.
3. **Finalize and Release the Feature:**
   - Merge your completed feature branch back into your `main` branch.
   - Create a new **annotated tag** corresponding to the feature's version (e.g., `v2`, `v3`, etc.).
   - Push the new tag to your GitHub repository.
   - On GitHub, navigate to "Releases" and **draft a new release**.
   - Select the tag you just pushed, give the release a descriptive title, attach your compiled `bin/myshell` binary as an asset, and publish it.

This entire cycle must be completed for each feature before you begin the next one.

Feature-2: Built-in Commands (10 Marks)

Concepts Covered: Process Model (Parent vs. Child), Built-in Commands, System Calls (`chdir`), Git Branching, Git Tagging & Releases

What you will do: Your base shell can only run external commands by creating a child process. However, some commands, like `cd` or `exit`, must be executed by the shell process itself. In this feature, you will add support for these essential "**built-in**" commands.

Task 1. Create a Development Branch

From your **`main`** branch, create and switch to a new branch for this feature. A suitable name would be **`feature-built-ins`**.

Task 2. Implement the Built-in Commands

You are required to implement the following commands. The logic for these commands should be added directly into your shell's source code.

- **`exit`**: Terminates the shell gracefully.
- **`cd <directory>`**: Changes the current working directory. Use the **`chdir()`** system call and remember to handle errors (e.g., if the directory does not exist).
- **`help`**: Displays a brief help message listing all the built-in commands you have implemented.
- **`jobs`**: For now, this is a placeholder. It should just print a message like **`"Job control not yet implemented."`**

Task 3. Modify the Main Loop to Handle Built-ins

This is the most critical logic change. Before your code calls `fork()` to run an external command, you must first check if the command entered by the user is one of your new built-ins.

- **Implementation Hint:** A good way to organize this is to create a new function, for example: **`int handle_builtin(char** arglist)`**`;`. This function can check if **`arglist[0]`** is a built-in command (like "cd", "exit", etc.). If it is, the function should execute the command's logic and return `1` (to signal that the command was handled). If it's not a built-in, it should return `0`. In your **`main`** loop, you will now only call your external **`execute()`** function if **`handle_builtin()`** returns `0`.

Task 4. Build, Run, and Commit

- Use **`make`** to build your updated shell.
- Test your new commands thoroughly. Verify that `exit` closes the shell, `cd` changes the directory (you can check with `pwd` executed as an external command), and `help` and `jobs` print their messages.
- Once everything is working correctly, use the standard `git` workflow to add your modified files and commit them to your **`feature-built-ins`** branch with a clear message.

Feature-3: Command History (10 Marks)

Concepts Covered: Data Structures (Arrays/Lists), String Manipulation, Built-in Commands, Git Branching, Git Tagging & Releases

What you will do: A good shell remembers what you have typed. In this feature, you will add memory to your shell by implementing a command history. This includes storing recent commands, displaying them with a new `history` command, and re-executing them with the `!n` syntax.

Task 1. Create a Development Branch

- First, merge your completed feature-built-ins branch into main.
- From your updated main branch, create and switch to a new branch for this feature. A suitable name would be feature-history.

Task 2. Implement History Storage

- You need a data structure to store the commands. A simple array of strings (e.g., `char* history[HISTORY_SIZE];`) is a great place to start.
- Modify your main loop so that after a command is successfully read, it is added to this history storage. You will need to manage the array, keeping track of the current command count and handling the case where the history is full (e.g., by overwriting the oldest command).
- Your shell must store at least the last 20 commands.

Task 3. Implement the history Built-in Command

- Add `history` to your list of built-in commands (in your `handle_builtin` function or equivalent).
- When the user types `history`, your shell should loop through your history storage and print each command, preceded by its line number (starting from 1).

Task 4. Implement the !n Re-execution Feature

- This is a special type of command that needs to be handled *before* tokenization or even adding the command to the history.
- In your main loop, after reading a command line, check if the first character is `!`.
  - If it is, parse the number `n` that follows.
  - Retrieve the *n*-th command string from your history storage.
  - Replace the `!n` command line with the retrieved command string, and then let your shell's main loop proceed to tokenize and execute it as if the user had typed it directly.
  - Remember to handle errors, such as if `n` is out of bounds.

Task 5. Build, Run, and Commit

- Use `make` to build your shell.
- Test the history features thoroughly:
  - Type several commands.
  - Use the `history` command to see if they are listed correctly with numbers.
  - Use `!n` to re-execute a command from the list.
- Once everything works, add your modified files and commit them to your `feature-history` branch.

Feature-4: Tab Completion with Readline (15 Marks)

Concepts Covered: External Library Integration (GNU Readline), Linking (`-l` flag), API Usage, Command-line Editing, Tab Completion

What you will do: This feature is a hallmark of modern shells and dramatically improves usability. You will integrate the powerful GNU Readline library to replace your basic command input function, providing professional features like tab completion and advanced history navigation "out of the box."

Task 1. Merge and Create a New Branch:

- First, merge your completed `feature-history` branch into `main`.
- From your updated `main` branch, create and switch to a new branch for this feature. A suitable name would be `feature-readline`.

Task 2. Implement the Readline Integration:

- You are required to integrate the readline library into your project. This involves the following steps:
  1. **Replace your `read_cmd` function** with calls to `readline()`.
     - In your C source files, you will need to include the necessary Readline headers: `#include <readline/readline.h>` and `#include <readline/history.h>`.
     - In your `main` loop, replace the call to your own `read_cmd()` function with a call to `readline()`. The `readline()` function handles the prompt, user input, and line editing automatically.
  2. **You must modify your `Makefile`** to link against the library. Add `-lreadline` to your linker flags variable (e.g., `LDFLAGS`).

Task 3. Implement Completion and Verify History:

- You must now ensure the new features provided by Readline are working as expected.
  - **Implement filename and command completion.** When the user presses the `Tab` key, the shell should automatically complete the name of a command or a file in the current directory.
  - **The readline library provides history functionality out of the box.** Ensure the up/down arrow keys now correctly cycle through your command history.
- **Implementation Hint:** `readline` has its own functions for managing history (like `add_history()`) and providing default completion. Read the documentation to see how to properly initialize it. Start with the default completion behavior. The `add_history()` function should be called after a command has been successfully read and is not empty.
- **Useful resource:** GNU Readline Library Documentation.

Task 4. Build, Run, and Commit:

- Use `make` to build your shell. If you get "undefined reference" errors for `readline` or `add_history`, it means your `Makefile` was not updated correctly.
- Run your new shell and test the features:
  - Press the **Up and Down Arrow keys**. You should be able to navigate through your command history seamlessly.
  - Type the beginning of a filename or a command (like `ls`) and press the **Tab key**. Readline should automatically complete it for you.
- Once everything works, add your modified files and commit them to your `feature-readline` branch.

Feature-5: I/O Redirection and Pipes **(15 Marks)**

Concepts Covered: File Descriptors (`STDIN_FILENO`, `STDOUT_FILENO`), I/O Redirection, Pipes, System Calls (`open`, `close`, `dup2`, `pipe`), Process Management

What you will do: This feature is fundamental to the UNIX philosophy of small tools working together. You'll implement the ability for your shell to redirect a command's input and output, and to chain commands together using pipes. This will be version `v5`.

Task 1. Merge and Create a New Branch

- First, merge your completed `feature-readline` branch into `main`. From your updated `main` branch, create and switch to a new branch for this feature. A suitable name would be `feature-io-redirection`.

Task 2. Modify the Parser

- Your `tokenize` function (or a new parsing layer you create) must be updated to recognize the special characters: `<`, `>`, and `|`. The parser needs to not only separate the command arguments but also identify the redirection and pipe operators and the filenames or commands associated with them. You may need to adjust your data structures to store this information.

Task 3. Implement I/O Redirection (< and >):

- **Output Redirection (>):** Implement the `>` operator. The command's standard output should be written to the specified file instead of the console.
    - *Example:* `ls -l > file_list.txt`
- **Input Redirection (<):** Implement the `<` operator. The command should take its standard input from the specified file instead of the keyboard.
    - *Example:* `sort < file_list.txt`

Task 4. Implement Pipes (|):

Implement the `|` operator. The standard output of the command on the left of the pipe must become the standard input of the command on the right.

- *Example:* `cat /etc/passwd | grep "root"`

- **Implementation Hints:**
  This task is all about **file descriptor management**. The key system calls are `open()`, `close()`, `dup2()`, and `pipe()`.
    - **For < and >:** After you `fork()`, but *before* you `execvp()`, the child process must:
        - `open()` the target file to get a new file descriptor.
        - Use `dup2()` to clone this new file descriptor onto `STDIN_FILENO` (for `<`) or `STDOUT_FILENO` (for `>`).
        - `close()` the original file descriptor returned by `open()`.
    - **For pipes:**
        - Call `pipe()` in the parent shell to create a read/write pair of file descriptors.
        - `fork()` twice, creating two child processes (one for each side of the pipe).
        - In the *left* child (the writer), `dup2()` the pipe's write-end to its `STDOUT_FILENO`.
        - In the *right* child (the reader), `dup2()` the pipe's read-end to its `STDIN_FILENO`.
        - **Crucially, `close()` all unused pipe ends in all three processes (the parent and both children) to avoid hangs.**

Task 5. Build, Run, and Commit

- Use `make` to build your shell.
- Test each new feature thoroughly using the examples provided. Verify that files are created correctly for `>` and that commands read from them for `<`. Ensure the pipe example works as expected.
- Once everything works, add your modified files and commit them to your feature branch.

Feature-6: Command Chaining and Background Execution (15 Marks)

Concepts Covered: Process Management, Job Control, Zombie Proc esses, `waitpid()` with `WNOHANG`, Parsing

What you will do: This feature adds multitasking capabilities to your shell. You will implement the ability to run multiple commands sequentially from a single line using the semicolon (`;`) and to execute commands in the background using the ampersand (`&`). This will also require you to properly manage these background jobs and update your `jobs` command. This will be version `v6`.

Task 1. Merge and Create a New Branch:

- First, merge your completed feature-io-redirection branch into main.
- From your updated main branch, create and switch to a new branch for this feature. A suitable name would be feature-multitasking.

Task 2. Implement Command Chaining (;):

- **Requirement:** Allow multiple commands to be entered on a single line, separated by semicolons. They must be executed sequentially. The shell should wait for the first command to complete before starting the next.
  - *Example:* `echo "First command" ; sleep 2 ; echo "Second command"`
- **Implementation:** Your parser should first split the input line into separate command strings based on the `;` delimiter. Your main loop will then iterate through these command strings, executing each one sequentially using your existing `fork-exec-wait` logic.

Task 3. Implement Background Execution (&):

- **Requirement:** If a command is followed by an ampersand, the shell should execute it in the background and immediately display the prompt again for the next command, without waiting for the background command to finish.
  - *Example:* `sleep 10 &`
- **Implementation:** When your parser detects an `&` at the end of a command, it should set a flag. When you `fork()` a process for a background command, the parent shell must **not** call `waitpid()` for that child. Instead, it should store the child's PID in a list of background jobs and continue its main loop immediately.

Task 4. Handle Zombie Processes and Update `jobs`:

- **Requirement:** Your `jobs` built-in command should now list all commands currently running in the background, along with their Process IDs (PIDs).
- **Implementation:** You must handle "**zombie**" processes. A zombie is a completed background process whose status has not yet been collected by the parent. In your main loop, *before* prompting for new input, you must call `waitpid(-1, &status, WNOHANG)` in a loop.
  - The `WNOHANG` option makes the call non-blocking; it will "reap" any completed child process and return immediately if none have finished. This prevents your shell from freezing while waiting for background jobs.
  - As jobs are reaped, you should remove them from your list of active background jobs. Your `jobs` command should then print the contents of this list.

Task 5. Build, Run, and Commit:

- Use `make` to build your shell.
- Test the features thoroughly: run chained commands with `;`, run commands in the background with `&`, and use `jobs` to see the list of active background processes.
- Once everything works, add your modified files and commit them to your `feature-multitasking` branch.

Feature-7: The if-then-else-fi Control Structure (10 Marks)

Concepts Covered: Parsing, State Management, Conditional Logic, Process Exit Status

What you will do: This is your first step into creating a true scripting language. You will add the ability for your shell to understand and execute a multi-line `if-then-else-fi` control structure, allowing for conditional execution of commands based on the success or failure of another command. This will be version `v7`.

Task 1. Merge and Create a New Branch*:*

- First, merge your completed `feature-multitasking` branch into `main`. From your updated `main` branch, create and switch to a new branch for this feature. A suitable name would be `feature-if-then-else`.

Task 2. Implement the if-then-else-fi Structure:

- **Requirement:** Implement an `if-then-else-fi` control structure. The shell must be able to parse this multi-line block.
    - **`if` block:** The shell executes the command following the `if` keyword.
    - **`then` block:** If the command from the `if` block exits with a status of `0` (success), the shell executes the commands between `then` and `else` (or `fi`).
    - **`else` block:** If the command from the `if` block exits with a non-zero status (failure), the shell executes the commands between `else` and `fi`. The `else` block is optional.
    - **`fi` block:** This keyword marks the end of the entire `if` statement.
- **Example Usage:**

```
# Example with else
if grep "user" /etc/passwd > /dev/null
then
  echo "Found user"
else
  echo "User not found"
fi

# Example without else
if [ -f "myfile.txt" ]
then
  echo "myfile.txt exists."
fi
```

- **Implementation Hints:**
  This is primarily a **parsing and state management challenge**. Your `read_cmd` function (or equivalent) will need to be significantly smarter. When it sees an `if`, it must enter a special mode to continue reading lines until it finds a matching `fi`.
    - Store the lines for the `then` and `else` blocks in separate buffers (e.g., arrays of strings).
    - Execute the command from the `if` block using your standard `fork-exec-wait` logic.
    - After the command finishes, use the **`WEXITSTATUS(status)`** macro on the status variable returned by `waitpid()` to get the child's actual exit code.
    - Based on the exit code (`0` for success, non-zero for failure), your shell must then choose which block of commands to execute.

Task 3. Build, Run, and Commit:

- Use `make` to build your shell.
- Test the `if` structure thoroughly. Create test cases with and without the `else` block. Use commands that are known to succeed (like `true`) and fail (like `false` or `grep` on a non-existent pattern) to verify both branches of logic.
- Once it works correctly, add your modified files and commit them to your `feature-if-then-else` branch..

Feature-8: Shell Variables (10 Marks)

Concepts Covered: Data Structures (Linked Lists/Hash Tables), String Manipulation, Parsing, Memory Management

What you will do: A scripting language isn't complete without variables. In this final feature, you'll add the ability for your shell to create, use, and modify variables. This involves implementing variable assignment and expanding variables (e.g., `$VARNAME`) before a command is executed. This will be version `v8`.

Task 1. Merge and Create a New Branch:

- First, merge your completed `feature-if-then-else` branch into `main`. From your updated `main` branch, create and switch to a new branch for this feature. A suitable name would be `feature-variables`.

Task 2. Implement Variable Assignment and Storage:

- **Requirement:** Implement a mechanism to assign values to variables. The standard shell syntax is `VARNAME=value` (with no spaces around the `=`). This should be treated as a built-in action.
  - *Example:* `MESSAGE="Hello, World!"`
- **Implementation:**
  - You will need an internal data structure to store the variables. A simple linked list of key-value pairs is a good place to start. For better performance, a hash table could be used.
  - Your parser must be updated to identify assignment statements. A simple check is to see if a token contains an `=` and has no spaces around it.
  - When an assignment is detected, it should not be executed via `fork()`. Instead, a built-in function should parse the `VARNAME` and `value` and store them in your data structure.

Task 3. Implement Variable Expansion:

- **Requirement:** Before executing any command, the shell must scan the arguments for words beginning with a `$`. If found, it should replace the word (e.g., `$MESSAGE`) with the variable's stored value.
  - *Example:* `echo $MESSAGE` should print "Hello, World!".
- **Implementation:**
  - Create a pre-execution "expansion" step. After tokenizing but *before* executing, loop through the `arglist`. If an argument starts with `$`, look it up in your variable storage and replace it with its value.
  - Be very careful with memory management here. You will likely need to allocate new memory for the expanded string.

Task 4. Implement the `set` Command:

- **Requirement:** Create a new built-in command (e.g., `set` or `export`) that prints all currently defined variables and their values.
- **Implementation:** Add `set` to your `handle_builtin` function. When called, it should iterate through your variable data structure and print each key-value pair.

Task 5. Build, Run, and Commit:

- Use `make` to build your shell.
- Test the variable features thoroughly:
  1. Assign a value to a variable: `MSG="hello"`
  2. Use the `set` command to verify it was stored.
  3. Use the variable in a command: `echo $MSG`
  4. Re-assign the variable and test again.
- Once it works correctly, add your modified files and commit them to your `feature-variables` branch.

Feature-9: Git Workflow (10 Marks)

Concepts Covered: Collaborative Git Workflow (Forks, Issues, Pull Requests), Code Review

What you will do: This final part covers the unique submission process for this assignment, which simulates a professional, collaborative Git workflow. You will use a secondary GitHub account to `fork` your own project, report a mock "bug" by creating an `issue`, fix it, and submit a `pull request`. Finally, you will act as the project maintainer to review and merge the fix.

Task 1. Prepare for the Pull Request Workflow

- For this task, you will need a **secondary GitHub account**. If you do not have one, you must create a new one using a different email address. This is a mandatory step for simulating a collaborative environment.
- Log out of your primary GitHub account (the one that owns the `ROLL_NO-OS-A03` repository) and log in with your new, secondary account.

Task 2. Fork and Create an Issue

- While logged in as your **secondary user**, navigate to the GitHub page of your primary assignment repository (`github.com/YourPrimaryUsername/ROLL_NO-OS-A03`).
- In the top-right corner, click the **`Fork`** button. This will create a personal copy of your own project under your secondary user's account.
- Now, go back to your **primary repository's** GitHub page. Navigate to the "Issues" tab and create a **`New issue`**.
  - **Title:** Choose a title for a mock bug or a minor feature. For example: `Bug: 'help' command does not list the 'set' command`.
  - **Comment:** Briefly describe the issue. For example: "The `set` command was added in a previous feature, but the `help` command was not updated to include it in the list of available built-ins."
  - Submit the new issue.

Task 3. Fix the Issue and Submit a Pull Request

- Clone the **forked repository** (the one on your secondary account) to a new, separate location on your local machine.
- In this new local copy, create a new branch to work on the fix (e.g., `fix-help-command-bug`).
- Implement the code changes necessary to "fix" the issue you reported. In the example above, this would mean updating your `help` command's logic.
- Commit and push this new branch to your **forked (secondary) repository**.
- On the GitHub page of your forked repository, you will see a prompt to **"Compare & pull request"**. Click it.
- Review the changes and create the pull request. This will formally submit your fix for review to the original (primary) repository.

Task 4. Review and Merge the Pull Request

- Log out of your secondary GitHub account and log back in with your **primary account**.
- Navigate to your primary assignment repository (`ROLL_NO-OS-A03`). You will see a new entry in the "Pull requests" tab.
- As the project maintainer, click on the pull request to **review** it. Look at the "Files changed" tab to see the proposed fix and ensure it is correct.
- If the fix is correct, **merge the pull request**. This will integrate the changes from your secondary account's branch into your primary repository's `main` branch.
- Finally, on your local machine, navigate back to your original `ROLL_NO-OS-A03` directory and use `git pull` to sync the merged changes from GitHub.

Task 5. Push All Branches to GitHub

- For final grading, your instructor needs to see the history of your entire development process.
- Use the `git push` command to push **all** of your feature branches (e.g., `feature-built-ins`, `feature-history`, `feature-readline`, etc.) from your local primary repository to GitHub.

Viva-Voce and Final Submission

What you will do: This final part covers the steps to finalize your project's Git history and prepare for the viva-voce. The submission will be followed by an oral defense where you will be expected to demonstrate your shell and explain your code to determine your final grade.

Important Note on Grading: Your performance in the viva-voce will determine the marks awarded for **all preceding features (Features 2 through 8)**. You will be asked questions during this session to assess your true understanding of the concepts you've implemented and the code you have written. As stated in the academic integrity policy, failure to explain your own code will result in a zero.

Task 1. Prepare for the Viva-Voce

- You must be prepared to demonstrate your shell and verbally explain the concepts behind each feature you have implemented.
- Thoroughly review your own code, your Git history, and the core OS concepts related to each feature. You must be able to explain the "how" and "why" of every part of your implementation, from the `fork-exec-wait` cycle and built-in commands to I/O redirection and shell variables.

Task 2. Merge Your Final Branch

- Merge your final completed feature branch, which should be **`feature-variables`** (or your equivalent for Tag v8), back into your `main` branch. This ensures that `main` contains the complete and final version of your project with all features integrated.

Task 3. Push Everything to GitHub

- For grading, your instructor needs to see not only the final code but also the history of your development process contained in your feature branches.
- Use the appropriate `git` command to push your updated `main` branch to GitHub.
- Also, use the `git` command to push **all** of your feature branches to your remote repository. This includes branches like `feature-built-ins`, `feature-history` & `feature-readline` etc.

Grading Rubric (Total: 100 Marks)

Feature	Task	Marks
Feature-1	The Starting Point (Release - 1)	05
Feature-2	Built-in Commands (Tag v2)	10
Feature-3	Command History (Tag v3)	10
Feature-4	Tab Completion with Readline (Tag v4)	15
Feature-5	I/O Redirection and Pipes (Tag v5)	15
Feature-6	Command Chaining and Background Execution (Tag v6)	15
Feature-7	The if-then-else-fi Control Structure (Tag v7)	10
Feature-8	Shell Variables (Tag v8)	10
Feature-9	Git Workflow	10
Total		**100**

Happy Learning with Arif Butt