Operating Systems
Programming Assignment – 01

Introduction and Learning Objectives

This assignment is a hands-on coding project that helps you learn how C programs are built and how real software projects work. You'll create a utility library by writing a collection of helpful C functions that do common tasks like working with text and files, then build a main program that actually uses your library functions to do something useful. The project also teaches you the build process understanding how C code gets turned into running programs and using tools like **`make`** to automate everything.

The goal isn't just writing code it's mastering the complete professional workflow for building and releasing multi-file C projects. You'll learn how different parts connect together, automate builds with Makefile, understand static vs dynamic linking, and use professional Git/GitHub practices including branching and creating releases. By the end of this assignment, you will have demonstrated proficiency in:

- **Modular Programming:** Structuring C code into reusable modules with separate header `(.h)` and source `(.c)` files.
- **Static & Dynamic Libraries:** Compiling object files and creating both static `(.a)` and dynamic `(.so)` libraries.
- **The Linking Process:** Understanding the roles of the `-I` (include), `-L` (library path), and `-l` (link library) flags to connect your program with your libraries.
- **Build Automation:** Writing a flexible and efficient Makefile(s) to manage the entire compilation, linking, and installation workflow for multiple targets.
- **Documentation:** Creating standard Linux man pages to document your library's functions and programs.
- **Binary Analysis:** Using tools like `nm`, `ar`, `readelf`, and `ldd` to inspect libraries and executables.
- **Version Control Workflow:** Applying `git` for project setup, using branches for features, merging completed work, and maintaining a clean project history.
- **Software Releasing:** Using `git tags` to mark stable versions of your project and creating official, downloadable releases on GitHub with compiled assets.

The Scenario: A Core C Utilities Library

You will create a general-purpose utility library named `libmyutils`. This library will be composed of two modules:

- **String Functions:** It will provide custom implementations of common string manipulation functions.
- **File Functions:** It will provide utility functions for file-based operations, like counting words or searching for a pattern.

You will then write a driver program that utilizes functions from both modules. By building and linking this library first statically and then dynamically, you will directly observe the trade-offs involved and analyze the resulting executables to see the linker's work firsthand.

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: Project Scaffolding and Version Control (05 Marks)

Concepts Covered: Linux File Hierarchy, Basic Shell Commands, Git/GitHub
What you will do: Your first task is to set up a professional project structure and place it under version control. This foundation is essential for your work.

Task 1. Create a GitHub Repository:

- Create a new **public** repository on GitHub.
- Name your file exactly: ROLL_NO-OS-A01 (If your roll number is **bsdsf23a001**, name it: **`BSDSF23A001-OS-A01` (ALL UPPERCASE)**
- Initialize the repository with a `README.md` file.

Task 2. Clone the Repository:

- Copy the HTTPS URL for your new repository and from your Linux terminal, use the appropriate `git` command to `clone` the repository to your local machine.

Task3. Create the Project Structure:

- Navigate inside your newly cloned project folder, and use shell commands to create the structure of your project as shown below:

```
ROLL_NO-OS-A01/
├── src/          # Source code for your library and application
├── include/      # Header files
├── lib/          # Compiled static and dynamic libraries
├── bin/          # The final executable programs
├── obj/          # Intermediate object files
└── REPORT.md     # Your analysis report (you will write this throughout)
```

Task 4. Save and Push Your Structure:

- Use the standard `git` workflow to save your changes to GitHub:
  - **Stage** all new files and directories for tracking.
  - **Commit** the staged files with an appropriate message.
  - **Push** your commit to the main branch on GitHub.

Deliverables:

- A GitHub repository URL that follows the specified ROLL_NO-OS-A01 naming convention.
- All work for the subsequent parts will be committed incrementally to this repository. Well-crafted, atomic commits are part of the grad.

Feature-2: Multi-file Project using Make Utility (20 Marks)

Concepts Covered: Multi-file Compilation, Makefile Basics, Git Branching, Git Tags, and GitHub Releases.

What you will do: Before creating a library, the first step is to learn how to compile a program that is split across multiple source files. In this part, you will write a simple Makefile that compiles all your `.c` files directly and links them into a single executable. This part establishes a baseline for your project and will be your first versioned release.

Marks Distribution (20 Marks):

- **Code Implementation (5 Marks):** All functions in the `.c` files are correctly implemented and tested in `main.c`.
- **Makefile Correctness (5 Marks):** The Makefile correctly compiles and links the project from all source files.
- **Git Workflow (5 Marks):** A separate `multifile-build` branch is used and work is committed correctly.
- **Tag and Release (5 Marks):** A `v1.0-multifile` tag and corresponding GitHub Release with the binary asset are created successfully.

Task 1. Create a Development Branch:

- From your `main` branch, it is essential to create a new branch for this new feature. Use the appropriate `git` command to create and switch to a new branch named **`multifile-build`**.

Task 2. Implement the Library and Driver Code:

- **Create Header Files:** In the `include/` directory, create `mystrfunctions.h` and `myfilefunctions.h` with the exact content provided below.

```
// File: include/mystrfunctions.h
int mystrlen(const char* s);
int mystrcpy(char* dest, const char* src);
int mystrncpy(char* dest, const char* src, int n);
int mystrcat(char* dest, const char* src);
```

```
// File: include/myfilefunctions.h

// Count the number of lines, words and characters in the passed file
stream pointer. Return 0 on success and -1 on failure.
int wordCount(FILE* file, int* lines, int* words, int* chars);

// Search lines containing search_str in a file, and fills the matches
array. Return the count of matches and  -1 on failure.
int mygrep(FILE* fp, const char* search_str, char*** matches);
```

- **Create Source Files:** In your `src/` directory, create `mystrfunctions.c`, `myfilefunctions.c`, and `main.c`. Your main task is to write the C code to implement all the functions and create a driver program in `main.c` that effectively tests them. Use the templates below as your starting point.

```c
// File: src/mystrfunctions.c
#include "../include/mystrfunctions.h"

// Your main task is to implement the logic for the string
functions mentioned in the header file ...
```

```c
// File: src/myfilefunctions.c
#include "../include/myfilefunctions.h"

// Your main task is to implement the logic for the string
functions mentioned in the header file ...
```

```c
// File: src/main.c
#include <stdio.h>
#include "../include/mystrfunctions.h"
#include "../include/myfilefunctions.h"

int main() {
    printf("--- Testing String Functions ---\n");
    // Add code here to test each of your string functions.

    printf("\n--- Testing File Functions ---\n");
    // Add code here to test your file functions.


    return 0;
}
```

Task 3. Create a Simple Makefile for Direct Compilation:

- For this project write multiple `makefiles` using Recursive approach as discussed in class. You need to use multiple targets and make use of macros to give your `makefiles` a professional look.

Task 4. Build, Run, and Commit:

- Now use make utility to build the entire project and run the resulting executable (`./bin/client`) to ensure all your functions work correctly.
- Once working, use the standard `git` workflow to add all your new and modified files to the staging area, and then commit them to the `multifile-build` branch with a clear message.

Task 5. Tag and Release Your First Version:

This is your first stable point. You will now create your first versioned release, by following the given steps:

- Use the `git tag` command to create a new **annotated tag**. Name it `v0.1.1-multifile` and include a message (e.g., "Version 1.0 - Basic multifile compilation").

- Push your new tag to your remote repository on GitHub. (Note: `git push` does not push tags by default; you need to use a specific option or push the tag explicitly).
- Go to your repository page on the GitHub website. Navigate to the "Releases" section (usually on the right-hand side).
- Click "Draft a new release."
- In the "Choose a tag" dropdown, select the `v1.0-multifile` tag you just pushed.
- Give it a "Release title," such as **Version 0.1.0: Multi-file Build**.
- In the description, briefly mention what this version does (e.g., "A simple, directly compiled executable from all source files.").
- In the "Attach binaries" section, drag and drop or upload your compiled `bin/client` executable file. This is how you distribute the compiled program.
- Finally, click "Publish release".

Report Questions (for REPORT.md):

- Explain the linking rule in this part's Makefile: `$(TARGET): $(OBJECTS)`. How does it differ from a Makefile rule that links against a library?
- What is a `git tag` and why is it useful in a project? What is the difference between a simple tag and an annotated tag?
- What is the purpose of creating a "Release" on GitHub? What is the significance of attaching binaries (like your `client` executable) to it?

Feature-3: Creating and using Static Library (25 Marks)

Concepts Covered: Static Libraries (`.a`), Archiving (`ar`, `ranlib`), Makefile Modification, Git Branching & Merging

What you will do: You will now refactor your project to use a more scalable structure. Instead of compiling all source files together, you will first bundle your utility functions into a **static library**, and then link your main program against it.

Marks Distribution (25 Marks):

- **Makefile Correctness (10 Marks):** The `Makefile` is correctly modified to build and link the static library.
- **Code and Functionality (5 Marks):** The program compiles and runs correctly.
- **Git Workflow (5 Marks):** A separate `static-build` branch is used and work is merged correctly.
- **Tag and Release (5 Marks):** A `v2.0-static` tag and corresponding GitHub Release with assets are created successfully.

Task 1. Merge and Create a New Branch:

- First, merge your completed `multifile-build` branch into your `main` branch.
- From your updated `main` branch, **create and switch to** a new branch named `static-build`.

Task 2. Implement the Static Library and Driver Code:

- As discussed in class create a static library using the `ar` utility that contains all the string and file functions that you have already written, with the name of `libmyutils.a` inside the `/lib/` directory.

Task 3. Modify the `Makefile` appropriately:

- You need to modify the existing `Makefile` to incorporate the changes made in this feature, i.e., using a static library.

Task 4. Build, Run, Analyze and Commit:

- Now use `make` utility to build the entire project and run the resulting executable (`./bin/client_static`) to ensure all your functions work correctly.
- Use `ar -t` to list the contents of your `lib/libmyutils.a` file. Use `nm` and `readelf` to analyze the symbols in your object files and the final executable.
- Once working, use the standard `git` workflow to add all your new and modified files to the staging area, and then commit them to the `static-build` branch with a clear message.

Task 5. Tag and Release Your Second Version:

- Create a new annotated `git` tag named `v0.2.1-static`
- Push this new tag to GitHub.
- On GitHub, draft a new release titled **Version 0.2.1: Static Library Build**.
- Attach both your compiled library (`lib/libmyutils.a`) and your executable (`bin/client_static`) as binary assets.
- Publish the release.

Report Questions (for `REPORT.md`):

- Compare the `Makefile` from Part 2 and Part 3. What are the key differences in the variables and rules that enable the creation of a static library?
- What is the purpose of the `ar` command? Why is `ranlib` often used immediately after it?
- When you run `nm` on your `client_static` executable, are the symbols for functions like `mystrlen` present? What does this tell you about how static linking works?

Feature-4: Creating and using Dynamic Library (25 Marks)

Concepts Covered: Dynamic Libraries (`.so`), Position-Independent Code (`-fPIC`), Dynamic Loader (`ldd`), `LD_LIBRARY_PATH`, Git Branching & Merging

What you will do: In this part, you will evolve your project to use a **dynamic library** (or shared object, `.so`). Unlike a static library, a dynamic library's code is not copied into your final program. Instead, the operating system loads it into memory once when the program starts. This leads to smaller executables and more efficient memory usage.

Marks Distribution (25 Marks):

- **Makefile Correctness (10 Marks):** The `Makefile` is correctly modified to build and link both static and dynamic libraries.
- **Code and Functionality (5 Marks):** The dynamically linked program compiles and runs correctly after setting the library path.
- **Git Workflow (5 Marks):** A separate `dynamic-build` branch is used, work is merged correctly, and commits are clear.
- **Tag and Release (5 Marks):** A `v0.3.1-dynamic` tag and corresponding GitHub Release with assets are created successfully.

Task 1. Merge and Create a New Branch:

- First, ensure your `main` branch is fully updated. Use the appropriate `git` commands to switch to your `main` branch and then merge your completed `static-build` branch into it.
- From your updated `main` branch, create and switch to a new branch for this part. Name it `dynamic-build`.

Task 2. Implement the Dynamic Library and Driver Code:

- As discussed in class create a dynamic library by compiling using the `--shared` option of `gcc`, with the name of `libmyutils.so` inside the `/lib/` directory.

Task 3. Modify the `Makefile` appropriately:

You need to modify the existing `Makefile` to incorporate the changes made in this feature, i.e., using a dynamic library.

Task 4. Build, Run, Analyze and Commit:

- Now use `make` utility to build the entire project and run the resulting executable (`./bin/client_dynamic`) to ensure all your functions work correctly.
- Compare the file sizes of `bin/client_static` and `bin/client_dynamic` using the `ls -lh bin/` command. Note the significant difference.
- Attempt to run your new dynamic executable: `./bin/client_dynamic`. You will see a "cannot open shared object file" error. This is expected and is a core concept of dynamic linking. The operating system's loader does not know where to find your custom `libmyutils.so` file.
- To fix this, you must tell the loader where to look. Use the `export` command to temporarily add your project's `lib` directory to the `LD_LIBRARY_PATH` environment variable.
- Run `./bin/client_dynamic` again. It should now execute successfully.
- Finally, use the `ldd` tool on your dynamic client (`ldd bin/client_dynamic`) to see how the system resolved the library paths and confirm it's using the one from your `lib` directory.
- Once working, use the standard `git` workflow to add all your new and modified files to the staging area, and then commit them to the `dynamic-build` branch with a clear message.

Task 5. Tag and Release Your Third Version:

- Create a new annotated `git` tag named `v0.3.1-dynamic`
- Push this new tag to GitHub.

- On GitHub, draft a new release titled **Version0.3.1: Dynamic Library Build**.
- For a dynamic release, a user needs both the executable and the shared library to run the program. As assets, upload both your compiled executable **(`bin/client_dynamic`)** and your dynamic library **(`lib/libmyutils.so`)**.
- Publish the release.

Report Questions (for `REPORT.md`):

- What is **Position-Independent Code (`-fPIC`)** and why is it a fundamental requirement for creating shared libraries?
- Explain the difference in file size between your static and dynamic clients. Why does this difference exist?
- What is the `LD_LIBRARY_PATH` environment variable? Why was it necessary to set it for your program to run, and what does this tell you about the responsibilities of the operating system's dynamic loader?

Feature-5: Creating and Accessing Man Pages (15 Marks)

Concepts Covered: Linux Documentation, `man` pages, `groff` formatting, Makefile `install` Target
What you will do: A hallmark of a professional software project is good documentation and a simple installation method. In this part, you will create a standard Linux "man page" for your client application. Then, you will add an `install` target to your `Makefile` to simulate how a user would install your program and its documentation on their system.
Marks Distribution (15 Marks):

- **Man Page Content (5 Marks):** The man page is created correctly with all required sections.
- **Makefile install Target (5 Marks):** The `install` target correctly copies the executable and man page.
- **Git Workflow (5 Marks):** A separate `documentation` branch is used, and work is merged correctly.

Task 1. Merge and Create a New Branch:

- From your updated `main` branch, create and switch to a new branch for this part. Name it `man-pages`.

Task 2. Create the Man Page:

- Create a new directory named `man` in your project's root. Inside it, create another directory named `man3` (for manual section 3, which covers executable programs).
- Inside the `man/man3/` directory, create one file each for all of your functions, e.g., `mycat.1`.
- Man pages are written using a formatting language called **`groff`**. Research its basic syntax and structure your `man` files to include at least the following standard sections:
  - `.TH`: The title header
  - `.SH NAME`: A section for the function name and a one-line description.
  - `.SH SYNOPSIS`: A section showing how the function is typically called.
  - `.SH DESCRIPTION`: A more detailed explanation of what the function does.
  - `.SH AUTHOR`: A section with your name and email.
- You can preview your formatted man page locally by running the appropriate command from your project's root directory (e.g., `man -l man/man3/mycat.1`).

Task 3. Create an `install` Target in the Makefile:

- Make appropriate changes in your Makefile

Task 4. Test the Installation and Commit Your Work*:*

- Run `sudo make install`.
- After it completes, try running your program from anywhere by just typing `client`.
- Try viewing the man page by typing `man mycat`.
- Once your `install` target works, add and commit your new `man` directory and your modified `Makefile` to the `documentation` branch.

Task 5. Tag and Release Your Fourth Version:

- Create a new annotated `git` tag named `v0.4.1-final`
- Push this new tag to GitHub.
- On GitHub, draft a new release titled **Version0.4.1: Final Build**.
- Publish the release.

Viva-Voce and Final Submission (10 Marks)

What you will do: This final part covers the steps to merge all your work, ensure your report is complete, and submit the project for grading.

Marks Distribution (10 Marks):

- **Final Report (5 Marks):** All questions in `REPORT.md` are answered completely, clearly, and thoughtfully.
- **Final Git Workflow (5 Marks):** The `documentation` branch is merged correctly, and all branches are pushed to GitHub.

Task 1.

- **Finalize your REPORT.md file,** ensuring all questions from all parts of the assignment are answered. Review your answers to make sure they reflect your understanding of the concepts.

Task 2. Merge Your Final Branch:

- Merge your completed documentation branch back into your main branch. This ensures that main contains the complete and final version of your project, including the installation capability.

Step 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 branches.
- Use the `git push` command to update your `main` branch on GitHub.
- Also, use the `git push` command to push all your feature branches (`multifile-build`, `static-build`, `dynamic-build`, and `man-pages`) to your remote repository.

Grading Rubric (Total: 100 Marks)

Part	Task	Marks
Part 1	Project Scaffolding & Version Control	**5**
Part 2	Multi-file Project (Code, Makefile, Git, Release)	**20**
Part 3	Static Library (Makefile, Git, Release)	**25**
Part 4	Dynamic Library (Makefile, Git, Release)	**25**
Part 5	Man Pages & Installation (Man Page, Makefile)	**15**
Part 6	Final Submission (Report, Final Git Workflow)	**10**
Total		**100**

Happy Learning with Arif Butt