

AOS Assignment 4

Mini Version Control System

Deadline :

30th October 2024, 11:59:00 PM

Guidelines :

- Languages Allowed: C/C++
- Modularize and Indent your codes. Also, add comments wherever necessary to promote readability.
- Handle error cases wherever required. (If not done marks will be deducted).
- If the code doesn't compile, no marks will be rewarded.
- Segmentation faults at the time of grading will be penalized.
- Add a README.md File (compulsory) which contains instructions to execute your code, the working procedure of your code, assumptions you have made, and a short description of each feature you have implemented.
- Viva will be conducted at the time of evaluation, so prepare well.
- Keep your imported headers CLEAN. Do not import libraries you are not using, especially banned ones.
- You are allowed to use the filesystem library, C++ STL data structures, Compression algorithm, and the SHA-1 algorithm. For any other libraries, if you're unsure whether they're allowed, ask on Moodle before using them, as they might be restricted.
- If you have doubts about any command, refer to the original Git command to understand how it works, what output it gives, and how it's implemented. Use compression techniques and the SHA-1 algorithm to compress files and compute their SHA-1 hash for integrity check.

Goal:

The objective of this assignment is to develop a mini version control system (VCS) that mimics the basic functionalities of Git. This system will allow users to add files, commit changes, and checkout specific commits. The project will help you understand the core concepts of version control systems and how they manage file changes over time.

The following are the specifications for the assignment. For each of the requirements, an appropriate example is given along with it:

1. Initialize Repository

Command: `./mygit init`

Purpose:

- 🛠️ Initializes a new repository by creating a directory called `.mygit`.
- Sets up the necessary structure to store objects, metadata, and references to track changes.

What to Implement:

- Ensure that running `./mygit init` creates a hidden `.mygit` directory.
- This directory should set up the framework for version control, allowing the repository to manage files and changes.
- This command should be the first one users run when starting a new project with your version control system.

Expected Outcome:

- A `.mygit` directory is created with the required structure for the repository.
-

2. Hash-Object

Command:

```
$ echo -n "hello world" > test.txt
$ ./mygit hash-object [-w] test.txt
```

Purpose:

- 🗝️ This command calculates the SHA-1 hash of a file, compresses it, and optionally stores it as a blob object in the repository when the `-w` option is used. If `-w` is not included, it simply prints the file's SHA-1 hash.

What to Implement:

- The `-w` flag writes the file as a blob object to the repository.
- Display the computed SHA-1 hash of the file when the command is executed.
- Ensure that each file is stored as a unique object based on its content.

Expected Outcome:

- The SHA-1 hash of the file is displayed, and the file is stored as a blob in the repository when the `-w` flag is used (e.g., `a3c9c2f58c4b6a2e5f0f48757c2f0e4ef9b8d6d4`).
-

3. Cat-File

Command: `./mygit cat-file <flag> <file_sha>`

Example: `./mygit cat-file -p 95d09f2b10159347eece71399a7e2e907ea3df4f`

Purpose:

- 📄 This command reads and displays the content or metadata of a file stored as an object using its SHA-1 hash.

What to Implement:

- Implement support for the following flags:
 - `-p`: Print the actual content of the file.
 - `-s`: Display the file size in bytes.
 - `-t`: Show the type of object (e.g., blob for file content, tree for directory).
- Ensure that the command correctly locates the object using its SHA-1 hash and displays the relevant information based on the provided flag.

Expected Outcome:

- For **-p**, the content of the file is displayed.
 - For **-s**, the size of the file is printed.
 - For **-t**, the object type (blob, tree) is shown.
-

4. Write Tree

Command: `./mygit write-tree`

Purpose:

- 🌳 Writes the current directory structure to a tree object, which represents the hierarchy of files and directories.
- Creates a new tree object and stores it in the repository.

What to Implement:

- Capture the current directory's state, including files and subdirectories.
- Create a tree object representing the directory structure and store it in the repository.
- Display the SHA-1 hash of the newly created tree object.

Example:

```
$ mkdir test_tree && cd test_tree
$ /path/to/mygit.sh init
$ echo "AOS TA's are the best" > test_tree_file_1.txt
$ mkdir test_tree_1
$ echo "PG1 is GOAT" > test_tree_1/test_tree_file_2.txt
$ mkdir test_tree_2
$ echo "Buidling VCS is fun" > test_tree_2/test_tree_file_3.txt
```

Expected Outcome:

- The command outputs the SHA-1 hash of the tree object representing the current directory structure (e.g., `a3c9c2f58c4b6a2e5f0f48757c2f0e4ef9b8d6d4`).
-

5. List Tree (ls-tree)

Command: `./mygit ls-tree [--name-only] <tree_sha>`

Purpose:

- 📁 Lists the contents of a tree object (directory) using its SHA-1 hash.
- Displays detailed information about the files and subdirectories or just their names when using `--name-only`.

What to Implement:

- If the `--name-only` flag is provided, only the names of files and directories within the tree should be shown.
- Without the flag, display the mode (permissions), type (blob or tree), SHA-1 hash, and name of each entry in the tree.
- Ensure that the tree object represents directories correctly, showing both files and subdirectories.

Expected Outcome:

- Without flags: Detailed information about each object in the tree is displayed (mode, type, SHA, name).
- With `--name-only`: Only the names of files and directories are listed.

Example: `./mygit ls-tree 6d5c10cb0693c0f9ea3ef477651a15e40698bad8`

Expected Outcome:

```
Without Flag:
100644 blob efa3675ec1227827b21b4b54628a67c0571b3a28 README.md
040000 tree 6d5c10cb0693c0f9ea3ef477651a15e40698bad8 build

With Flag:
README.md
build/
```

6. Add Files

Command:

```
./mygit add .  
./mygit add main.cpp utils.cpp
```

Purpose:

-  Adds files or directories to the staging area, preparing them for the next commit.

What to Implement:

- Implement the functionality to stage all files in the current directory (`./mygit add .`) or specific files (`./mygit add file1 file2`).
- Ensure that the files' metadata (name, SHA, etc.) are correctly recorded in an index file to track staged changes.

Expected Outcome:


- The specified files are added to the staging area, making them ready to be committed in the next snapshot.
-

7. Commit Changes

Command:

```
./mygit commit -m "Commit message"  
./mygit commit
```

Purpose:

-  Creates a commit object, representing a snapshot of the staged changes, and updates the repository's history.

What to Implement:

- The commit object should store a reference to the tree object representing the current directory structure.
- If a message is provided with the `-m` flag, use it as the commit message. If not, assign a default message.
- After the commit, update the `HEAD` to point to the new commit.
- Ensure the commit contains metadata such as SHA, timestamp, and parent commit SHA (if applicable).


Expected Outcome:

- A commit SHA is displayed, and the repository's history is updated (e.g., `a3c9c2f58c4b6a2e5f0f48757c2f0e4ef9b8d6d4`).
-

8. Log Command

Command: `./mygit log`

Purpose:

-  Displays the commit history from the latest commit to the oldest.

What to Implement:

- Implement the log command to print details of all previous commits in reverse chronological order (latest to oldest).
- For each commit, display:
 - Commit SHA
 - Parent SHA (if applicable)
 - Commit message
 - Timestamp
 - Committer information

Expected Outcome:


- The commit history is shown, detailing each commit's metadata.
-

9. Checkout Command

Command: `./mygit checkout <commit_sha>`

Example: `./mygit checkout 95d09f2b10159347eece71399a7e2e907ea3df4f`

Purpose:

-  Checks out a specific commit, restoring the state of the project as it was at that commit.

What to Implement:

- Implement the checkout functionality to switch to a specific commit using its SHA.
- Extract the tree and blob objects associated with the commit and restore the directory structure and file content.
- Ensure errors are handled gracefully if an invalid or nonexistent commit SHA is provided.


Expected Outcome:

- The directory structure and files are restored to the state of the specified commit.
- The working directory reflects the exact snapshot of the project at the given commit.

Submission:

- **Upload format:** `<roll_number>_Assignment4.zip`
- Ensure you create a makefile to compile all your code, using the appropriate flags and linker options.
- Please follow the naming guidelines and directory structure below:

```
<roll_number>_Assignment4
├── README.md
├── makefile
└── Other files and Directories
```

- Include a **README.md** file that briefly describes your work and explains which file corresponds to each part of the assignment. Including the README is mandatory.
-  **Do NOT take codes from seniors or your batchmates, by any chance. We will extensively evaluate cheating scenarios along with the previous few year's submissions**