Assignment

Systems Spring Camp

Programming Club

February 2025

1 Introduction

In this assignment, you will be making a simpler version of git - **pclubgit**.

`pclubgit` can track individual files in the current working directory (no subdirectories!). It maintains a `.pclubgit/` subdirectory containing information about your repository. For each commit that the user makes, a directory is created inside the `.pclubgit/` directory (`.pclubgit/<ID>`, where `<ID>` is the ID of the commit). The `.pclubgit/` directory additionally contains two files: `.index` (a list of all currently tracked files, one per line, no duplicates) and `.prev` (contains the ID of the last commit, or 0..0 if there is no commit yet). Each `.pclubgit/<ID>` directory contains a copy of each tracked file (as well as the `.index` file) at the time of the commit, a `.msg` file that contains the commit message (one line) and a `.prev` file that contains the commit ID of the previous commit.

2 Main Assignment

2.1 Key differences between `git` and `pclubgit`

- Only supported commands are `init, add, rm, commit, status` and `log`. For each of them, only the most basic command line options are supported.
- `pclubgit` does not track diffs between files. Instead, each time you make a commit, it simply copies all files that are being tracked into the `.pclubgit/<ID>` directory (where `<ID>` is the commit ID).
- Commit IDs are not based on cryptographic hash functions, but instead are a fixed sequence of 40-character strings that only contain '6', '1' and 'c' (why we chose those characters is left as an exercise).
- Any commits with a commit message that does not contain "GO PCLUB!" (with exactly this capitalization and spelling) will be rejected with an error message.

- No user, date or other additional information is tracked by `pclubgit`. It does not allow to track subdirectories, or files starting with '.'.
- The `rm` command only causes `pclubgit` to stop tracking a file, but does not delete it from the file system.

2.2 Files:

- `pclubgit.c` - This is the file that you will fill in with your implementation of pclubgit.
- `pclubgit.h` - Do not edit - This file contains declarations of various constructs in `pclubgit.c` along with convenient `#defines`. See the "Important Numbers" section below.
- `main.c` - Do not edit - Contains the main for pclubgit (which parses command line options and calls into the functions defined in `pclubgit.c`).
- `Makefile` - Do not edit - This tells the program make how to build your code when you run the make command. This is a convenient alternative to having to repeatedly type long commands involving gcc.
- `util.h` - Do not edit - Contains helper functions that you may wish to use when completing the assignment.

2.3 Important Numbers: (see `pclubgit.h`)

In lecture, you learned about using `#define` to define constants as a single source of truth. You should use the appropriate constants from `pclubgit.h` whenever you are using any of the following numbers:

- Commit ID lengths are limited to 40 characters (not including the null terminator).
- Filenames are limited to 512 characters (including null terminator).
- Commit messages are limited to 512 characters in length (including null terminator).

2.4 Preliminaries:

For this assignment, you will be using some C functionality that you may or may not be familiar with. We will now highlight some of these features:

2.4.1 C library functions:

You may wish to familiarize yourself with the following C library functions: `fprintf`, `sprintf`, `fopen` (and `fclose`, `fwrite`, etc.), `strcmp`, `strlen`, `strtok`, and `fgets`. You can find documentation of the C library here (use the search box at the top to find out about each function). Make sure not to stray away from the "C library" section, as the linked website also contains C++ documentation.
When you look at the existing code in `pclubgit.c`, you will see examples of how these functions can be used to achieve the desired functionality. We recommend trying to understand the provided functions first, before starting to implement your own.

2.4.2 Handling I/O (more than just printf):

Unix machines use a concept called "streams" to handle arbitrary I/O. We will need two of these output streams in this assignment. The first is `stdout`, which is where your output goes when you call `printf`. We will use `stdout` to output all output indicating a "successful" action. The other output stream is `stderr`, which is where we will output all error messages. By default, both of these streams, `stdout` and `stderr`, are printed to your screen when you run a program.

Outputting to either `stdout` or `stderr` can be done similarly to using printf. The only change is that you use the fprintf function, and the first argument you pass in must be either "`stderr`" or "`stdout`" (without quotes).

```
[inside your C code]
fprintf(stdout, "%d\n", 3); // prints the number 3 to stdout, along with a newline
fprintf(stderr, "%d\n", 4); // prints the number 4 to stderr, along with a newline
```

If you want to know what messages went to `stdout` or `stderr`, you can forward them to a file instead of the terminal, by appending `2>log_err` and/or `1>log_out` to your command (e.g., `./my_program 2>log_err`). This will forward everything from `stderr` to the file `log_err` (and equivalently for `stdout` if you add `1>log_out`).

2.4.3 Included helper functions

To make life easier for you, we provide helper functions for common operations that you will encounter while implementing `pclubgit`. You will find these in `utils.h`. Here is a brief overview of each of these functions:

- `void fs_mkdir(const char* dirname)`: Create a new directory named `dirname`.
- `void fs_rm(const char* filename)`: Delete the file `filename`.
- `void fs_mv(const char* src, const char* dst)`: Move the file `src` to `dst`, potentially overwriting it.
- `void fs_cp(const char* src, const char* dst)`: Copy the file `src` to `dst`, potentially overwriting it.

- `void write_string_to_file(const char* filename, const char* str)`: Create or overwrite the file `filename` and write `str` into it, including the NULL character.
- `void read_string_from_file(const char* filename, char* str, int size)`: Open the file `filename` and read its content into the location pointed to by `str`; limit the amount read to at most `size` bytes, including the NULL character.

The last two functions should only be used together. Specifically, don't try to use `read_string_from_file` to read multi-line files, but only for single strings that you previously wrote into a file using `write_string_to_file`.

While these functions perform some basic checks to prevent you from accidentally overwriting important files, be careful whenever you call any function that modifies the file system. There is always a risk of unintentionally deleting or overwriting files, especially when working on your own machine!

2.5 Required functionality:

While the version of `pclubgit` that we've given to you compiles, you can't do much except call `pclubgit init` to create a new repository, and call `pclubgit add <file>` to start tracking a file. Everything else you need to implement yourself!

We recommend that you implement the `pclubgit` commands in this order, as this makes testing easier:

- `pclubgit status`
- `pclubgit rm`
- `pclubgit commit`
- `pclubgit log`

For each of these, you need to implement one of the functions below (but feel free to define new helper functions to make things easier). We give you an outline of each function's job, as well as the errors you need to be able to detect, and the output you need to produce.

2.6 Step 1: The status command

2.6.1 Functionality:

The status command in `pclubgit` should read the file `.pclubgit/.index` and print a line for each tracked file. The exact format is described below. Unlike `git status`, `pclubgit status` should not print anything about untracked files.

2.6.2 Output to stdout:

```
$ pclubgit status
Tracked files:

  <file1>
  [...]
  <fileN>

<N> files total
```

For each file in the above output, `<file*>` should be replaced with the filename of that file.

2.6.3 Return value and output to `stderr`:

This function should always return 0 (indicating success) and should never output to stderr.

2.7 Step 2: The rm command

Hint: You may want to have a look at the provided implementation of pclubgit add before implementing this command.

2.7.1 Functionality:

The `rm` command in `pclubgit` takes in a single argument, which specifies the file to remove from the index (which is stored in the file `.pclubgit/.index`). If the filename passed in is not currently being tracked, you should print an error as indicated below. Note that this behavior is different from git in that it doesn't delete the file from your file system.

2.7.2 Output to stdout:

None.

2.7.3 Return value and output to stderr:

If the filename specified in the provided argument exists in the index, the function should return 0 and produce no output on `stderr`. If the filename specified does not exist in the index, the function should return 1 and output the following to `stderr`:

```
$ pclubgit rm FILE_THAT_IS_NOT_TRACKED.txt
ERROR: File <filename> not tracked
```

2.8 Step 3: The commit command

2.8.1 Functionality:

The commit command involves a couple of steps:

- First, check whether the commit string contains "GO PCLUB!". If not, display an error message.
- Read the ID of the previous last commit from .pclubgit/.prev
- Generate the next ID (newid) in such a way that:
- All characters of the id are either 6, 1 or *c*.
- Generating 100 IDs in a row will generate 100 IDs that are all unique (Hint: you can do this in such a way that you go through all possible IDs before you repeat yourself. Some of the ideas from the number representation may help you!)
- Generate a new directory `.pclubgit/<newid>` and copy `.pclubgit/.index, .pclubgit/.prev` and all tracked files into the directory.
- Store the commit message (`<msg>`) into `.pclubgit/<newid>/.msg`
- Write the new ID into `.pclubgit/.prev`.

2.8.2 Output to `stdout`:

None.

2.8.3 Return value and output to `stderr`:

If the commit message does not contain the exact string "GO PCLUB!", then you must output the following to `stderr` and return 1:

```
 $ pclubgit commit -m "G-O- -P-C-L-U-B-!"
ERROR: Message must contain "GO PCLUB!"
```

If the commit message does contain the string "GO PCLUB!", then the function should produce no output and return 0.

2.9 Step 4: The log command

2.9.1 Functionality:

The goal of the log command is to print out all recent commits. See below for the individual steps:

- List all commits, latest to oldest. `.pclubgit/.prev` contains the ID of the latest commit, and each directory `.pclubgit/` contains a .prev file pointing to that commit's predecessor.

- For each commit, print the commit's ID followed by the commit message (see below for the exact format).

Output to stdout:

```
$ pclubgit log
[BLANK LINE]
commit <ID1>
    <msg1>
[BLANK LINE]
commit <ID2>
    <msg2>
[...]
commit <IDN>
    <msgN>
[BLANK LINE]
```

2.9.2 Return value and output to stderr:

If there are no commits to the `pclubgit` repo, `pclubgit` should return 1 and output the following to `stder`:

```
[assume that no commits have been made]
$ pclubgit log
ERROR: There are no commits!
```

If there are commits, you should produce the output indicated in the "Output to stdout" section above and return 0.

3 Harder extension

Last week, you implemented a basic version of `pclubgit` that supports init, add, rm, status, log and commit. However, this is not very useful yet – while you can create a `pclubgit` repository and commit data to it, there is no way to retrieve it and go back to the state of a previous commit. In the easy version, you will complete your `pclubgit` implementation to make this possible. And you will implement support for branches as well!

3.1 New Additions

- We added a new numerical constant: `BRANCHNAME_SIZE`, the maximum length of a branch name (including NULL terminator)

- We refined the `fs_*` functions, they now also show their arguments at an error. They also don't require all files to be within .pclubgit anymore.
- We added a new helper function: `int fs_check_dir_exists(const char* dirname)`. This function tests whether a given directory exists.

3.2 How branches and checkouts work in git

You can go to any commit in the history of time if you know its ID. This is called "checking out a commit". The current state of the working directory will be completely restored to how it was during the time of that commit.

Branches in git are basically just diverging commit histories. You have an "alternate history" depending on which branch you are on. One way to think about branches is that they allow multiple commits to point to the same previous commit: two branches can have a shared history, and then at some point they do different things starting from a certain point in time.
So every commit has a predecessor, but multiple commits can actually have the same predecessor. In fact, branches themselves are just identifiers for specific commits (which are called the "HEAD" of a branch). Just like commits, you can also check out a branch: in that case, you switch to that branch's HEAD commit. You can also check out commits that are not the HEAD of any branch – in that case, you say you are "detached", because you are not on any specific branch.

To add branches in `pclubgit`, not much changes: every commit still has exactly one predecessor (`.prev`), but multiple commits can have the same predecessor now. Branches in `pclubgit` are just pointers to specific commits. To keep things simple, we only allow `pclubgit` to commit when you are at the HEAD of a branch (i.e., when you are not detached). This allows you to "grow" each branch forwards.

When you are at any commit, you can start a new branch from there: you can say

```
git checkout -b <new_branchname>
```

to start a new branch that has the current commit as its HEAD. You can then start an alternative history by committing on this branch. When you initialize a new `pclubgit` repository, a default branch master is created, and its HEAD points to the 00.0 commit ID. To help you get a better sense of how branches actually work, you should work through the following tutorial until you are satisfied that you understand what branches do: http://pcottle.github.io/learnGitBranching/.

3.3 Required functionality:

While implementing branches may sound very complicated, it is not much additional work to what you have already implemented – in your last homework, you have created a solid foundations to build upon, so now things get easier. Directory structure

We will implement branches very similarly to how we implemented tracking of files. All we have to do is add a few files to our directory structure:

- `.pclubgit/.branches` is a file that contains a line for every branch that exists. We will call the line number on which the branch exists in this file the "branch number" (starting from 0).
- `.pclubgit/.current_branch` contains a single string with the name of the current branch if we are at the HEAD of some branch, or is an empty string if we are not on some branch HEAD.
- `.pclubgit/.branch_<branchname>` (one for every branch). This is a copy of the .prev file that belongs to the branch head (i.e., the HEAD commit of the branch)

With this information, we can now implement pclubgit branch and pclubgit checkout.

3.4 The branch command:

3.4.1 Functionality:

`pclubgit` branch prints all the branches and puts a star in front of the current one. Do you remember `pclubgit status`? This is almost the same: you need to read the entire .branches file line by line and output it. However, you also need to check each line against the string in `.current_branch`. If they are the same, you need to print a * in front of it.

Note that we require you to print branches in the order of creation, from oldest to latest. Also note that if you have checked out a commit previously (in contrast to a branch), you are detached from the HEAD and don't have to print a star in front of any branch. This is even true if the commit you checked out is actually the HEAD of a branch.

3.4.2 Output to stdout:

```
$ pclubgit branch
  <branch1>
  <branch2>
  [...]
* <current_branch>
  [...]
  <branchN>
```

3.4.3 Return value and output to stderr:

This function should always return 0 (indicating success) and should never output to stderr.

3.5 The checkout command

3.5.1 Functionality:

This is the command that is the most important feature of pclubgit. It allows you to restore the state of any commit in time, as well as to switch and create branches. pclubgit checkout has three different behaviors:

- `pclubgit checkout <commit_id>`: Check out a particular commit (i.e., leaving a branch HEAD if you are on it; this is called a "detached" state. You can assume that whenever you call this, you become detached, even if the commit you are checking out is some commit's HEAD).
- `pclubgit checkout <branch>`: Check out an existing branch and check out its head.
- `pclubgit checkout -b <newbranch>`: Start a new branch at the current commit.

While these behaviors look very different, they are actually very similar. First, you need to find out which of the three cases it is. We give you whether the user has provided -b (the new_branch bool parameter) and then the other argument, which can be either a commit ID or a branch name.

So pclubgit first needs to find out if you are giving it a commit or a branch name. For this, we have prepared a function *is_it_a_commit_id*, which you need to fill in. The function takes a string and returns true if and only if the string is 40 characters that are each 6, 1 or c.
Once you know whether you are dealing with a branch or a commit, you have to do one of two things:

- If it's a commit, check out the commit by replacing the currently tracked files with those from the time of the commit.
- If it's a branch (and you're not creating a new one), first check whether it exists. If yes, you need to switch to that branch. This means that you first store the latest commit of the current branch into the branch_branchname file, and then replace the content of current_branch by the new branch. You then read the branch_newbranch file to find out the HEAD commit of that branch, and then you check that commit out just like in 1).
- You are creating a new branch. This is very similar to 2), but you also have to add the branch to the .branches file and instead of reading the HEAD ID from .branch_branchname, you make the current prev ID the head ID for that branch and store it into that file.

Since we are nice people, we actually implemented the functionality above for you, except for the implementation of the actual checkout! But because we had to write this homework in a rush, there are three mistakes in the `pclubgit_checkout` function – you need to find and correct them for everything to run (one line per mistake).

Note: The `pclubgit_checkout` function is taking two arguments: `new_branch` is true if and only if -b was supplied to the command, and arg contains the other command line argument.

After you found the mistakes, you have to write a function `checkout_commit` which will do the actual checkout of a commit by:

- Going through the index of the current index file, delete all those files (in the current directory; i.e., the directory where we ran `pclubgit`).
- Copy the index from the commit that is being checked out to the `.pclubgit` directory, and use it to copy all that commit's tracked files from the commit's directory into the current directory.
- Write the ID of the commit that is being checked out into `.prev`.
- In the special case that the new commit is the 00.0 commit, there are no files to copy and there is no index. Instead empty the index (but still write the ID into `.prev` and delete the current index files). You may wonder how we could ever check out the 00.0 commit, since it is not a valid commit ID; the answer is that if you check out a branch whose HEAD is the 00.0 commit, that checkout is expected to work (while 00.0 would not be recognized as a commit ID).

Once you are done, you should experiment with the checkout and branch functionality by creating new branches, checking out old commits and see how you can commit to different branches individually. There is a lot that can go wrong, so we recommend testing thoroughly.

3.5.2 Output to stdout:

None.

3.5.3 Return value and output to stderr:

If the argument is a commit ID (40 characters, each of which is '6', '1' or 'c') of a commit that exists, a branch that exists and new_branch is false, or a branch that doesn't exist and new_branch is true, the function should return 0 and produce no output on stderr.

If the argument is a commit ID but the commit does not exist, the function should return 1 and produce the following error:

```
$ pclubgit checkout 6666.66
ERROR: Commit <commit_id> does not exist
```

If the argument is a branch that exists but new_branch is true, the function should return 1 and produce the following error:

```
$ pclubgit checkout -b <branch_name>
ERROR: A branch named <branch_name> already exists
```

If the argument is a branch that does not exist but new_branch is false, the function should return 1 and produce the following error:

```
$ pclubgit checkout <branch_name>
ERROR: No branch <branch_name> exists
```
