CS 137: Algorithms in the Real World — Spring 2024

The purpose of this project is to develop a deep understanding of the internals of `git` by implementing core parts of its functionality and reading and understanding the provided starter code.

Project Overview

In this project, you will implement a basic version of `git` , sophisticated enough to use with (most of) your existing local repositories and remotes (e.g. GitLab). A working git implementation involves a lot of incidental complexity, which we have attempted to remove for you in order to get at the core of what makes git work. You will leave this project with a mostly magic-free understanding of `git` , and the ability to readily fill in any remaining knowledge gaps.

Implementing `git` requires that you know almost everything to implement anything, so the learning curve will be steep in the beginning, and smoother towards the end.

This project runs best on Linux. If you do not have a Linux machine, contact Adam to get a Labradoodle account for the purposes of working on this project.

Requirements

A working proficiency with git is recommended to provide motivation and background for the project. If you are not already comfortable with crafting `git` commits and playing with branches, you may have a difficult time learning the advanced git concepts.

The project must be implemented in C so that you can make use of the starter code.

Deliverables

You will implement the core functionality of seven `git` subcommands: `add`, `status`, `commit`, `log`, `checkout`, `fetch`, and `push`, which will be subcommands of a `mygit` binary.

Testing

All git subcommands you implement should be compatible with the ordinary `git` installed on your computer. To that end, you should be able to verify your code works by inspecting the output of regular `git` commands. For instance, you can run `mygit add` followed by a `git status`, or `mygit commit` followed by `git log`. You may also find `git fsck` useful; it verifies all kinds of things about the `git` objects in the current repository.

Task 0: Browse the Starter Code

Begin by familiarizing yourself with the provided starter code. We will call out highlights, but looking yourself will payoff well once you get started coding.

Commandline parsing and file resolution is handled for you. The starter code also changes the working directory to the root of the current `git` repository, so you can access special `git` files using a relative path like `.git/<path>`.

Additionally, there are routines to parse some file formats from disk (though you will need to make methods to *write* them yourself). For this purpose, there are also helpful utility functions provided.

Finally, note that data structures are provided. These are sufficient to complete the project as-is, though nothing is stopping you from modifying them, of course.

Task 1: `git` Objects, the Index, and `add`

Review *Pro Git* Section 10.2. This covers the process of committing, except for the specifics of the index and the format of commit and tree objects, which you won't need yet.

The `git` Index

The index's primary purpose is to define exactly the tree object for the next commit. Thus, although we are typically only concerned with the contents of index that differs from the current commit, the index has entries for files unchanged from the current commit as well. You can inspect the current state of

the index using `git ls-files -stage [-debug]`. You can also set the `GIT_INDEX_FILE` environment variable when running this to inspect an arbitrary index file.

You should know from *Pro Git* that the index is a collection of filenames, along with the objects hashes those filenames refer to. The in-depth definition of the index format is found in `git` 's technical documentation, which you may choose to read if you are interested. Otherwise, we will provide you with all the necessary details. The full specification (we are implementing a subset of version 2) is available at

`https://github.com/git/git/blob/master/Documentation/gitformat-index.txt`

The index format consists of a header, followed by a number of index entries. All numbers are encoded in network order (big-endian). This also means that bitfields have their highest bits in the first byte. Index entries should be sorted in ascending order on the name field.

Index Header

The beginning of the index file consists of the following bytes:

Field Name	Length	Description
signature	4 bytes	The ASCII string `DIRC`
version Number	4 bytes	Use 2
number of entries	4 bytes	The number of index entries that follow

This is immediately followed by the specified number of index entries.

Index Entry

Field Name	Length	Description
ctime seconds	4 bytes	From `stat(2)`, but we will use 0
ctime nanoseconds	4 bytes	From `stat(2)`, but we will use 0
mtime seconds	4 bytes	From `stat(2)`, use the correct value
mtime nanoseconds	4 bytes	From `stat(2)`, but we will use 0
dev	4 bytes	From `stat(2)`, but we will use 0
ino	4 bytes	From `stat(2)`, but we will use 0
mode: unused	16 bits	Use 0
mode: object type	4 bits	Use `0b1000`, the value for a regular file
mode: unused	3 bits	Use 0
mode: UNIX permissions	9 bits	`0644` for non-executable or `0755` for executable
uid	4 bytes	From `stat(2)`, but we will use 0
gid	4 bytes	From `stat(2)`, but we will use 0
file size	4 bytes	The on-disk size of the file, from `stat(2)` or other means.
object `SHA1`	20 bytes	The hash of the object storing this file
flags	16 bits	For the highest 4 bits use 0; the other 12 bits store the length of the filename, saturated to `0xFFF`
file name	variable	The file name as a string of bytes
null termination	1 byte	A null byte
padding	0-7 bytes	Pad the length of the entry to a multiple of 8 bytes.

Note that many of these will be *incorrect* values. You may serialize accurate file metadata if you choose to, but we only require "size" and "mtime seconds". This doesn't substantially affect functionality, but it makes the code simpler.

Finally, the index file may be terminated with a 160-bit SHA-1 hash of the content of the rest of the index file. This is not tested for or required but some interop with `git` may be lost without it.

Creating `git` objects requires `SHA1` hashing and `zlib` compression. To save you the time messing around with libraries, we have provided `get_object_hash()` and `write_object()` in `object_io.c`.
Assuming you have handled creation of blob objects and indexes, the behavior of `git add` is quite simple:

(1) Read the current index

(2) Create, modify, or delete the index entries for the added files. This will involve creating blob objects for current state of the files. Be sure to handle the case where 'git add' removes a file from the index if it no longer exists in the working tree.

(3) Write the new index.

You do not need to specially handle ignored files (e.g. from `.gitignore`), even though the real `git add` does.

Task 2: `status`

`git status` displays the differences between the working tree, the index, and the current commit. The current commit is kept track of by the special file `.git/HEAD`. If you need a refresher on how HEAD works conceptually, see Chapter 3.1 of *Pro Git*.
Almost all commands going forward depend on the current commit as well, so it's worth ensuring you have a good understanding of HEAD and what `git` calls "refs". *Pro Git* Section 10.3 contains an explanation of the internals of both refs and HEAD. Because the format of `git` refs is so simple, the starter code includes several methods for working with them in `ref_io.h`, which you are advised to read and then make use of going forward. You will also find some of the starter code in `object_io.h` useful, as you will need to inspect the current commit.

status

Our implementation of `status` prints three sections of files: "Staged for commit", "Not staged for commit", and "Untracked files". Each file in the "staged" or "unstaged" section is also listed with an action: either "new file", "modified", or "delete." Of course, a "new file" that is "Not staged for commit" is considered untracked.
The state of the working tree, HEAD, and the index together determine the categories and actions of all files. For instance, a file present in the work tree, not present in HEAD, but present in the index would be considered "Staged for commit" with the "added" action. Critically, "staged" and "not staged" are not mutually exclusive. A file present in HEAD, differing from HEAD in the index, and differing from the index in the worktree, would be considered both "staged" and "unstaged", both with action "modified". How would this change if the file was not present in the HEAD, or not present in the work tree?
When a repository is first initialized, there is a valid HEAD, but it refers to a non-existent branch (because there are no commits for the branch to point to). Your `git status` is expected to still function in this scenario.
Your implementation should print the status of the repository in the following format:

```
Staged for commit:
<TAB><action>: <filename>
<TAB><action>: <filename>
...

Not staged for commit:
<TAB><action>: <filename>
<TAB><action>: <filename>
...
```

```
Untracked files:
<TAB><filename>
<TAB><filename>
...
```

Each section is separated by an empty line. The sections must be printed in this order, but the order of files is unimportant. Additionally, empty sections should be omitted (do not print a header). Do not print any information about the state of HEAD. This means that nothing will be output if the working tree is fully clean. You also do not need to print paths relative to the directory the command was invoked in; simply print them relative to the root of the repository (which is the active working directory).
As with `add`, your implementation does not need to respect ignored files.

Note on testing with existing repositories

At this point, your `git` is reading objects from the content-addressed database. If you try this on some of your existing repositories, especially ones where you have pushed and/or pulled, you will find that some objects aren't where we would expect in the filesystem, (`.git/objects/<SHA1-first-2>/<SHA1-rest>`) resulting in errors. This is usually because the object in question has been placed along with other objects into a "packfile". Packfiles are stores of many objects that enable delta compression between them to save space (similar to LZ77 from the zip project, if you've done it). If you want to use your `git` on these repositories, you can expand the packfiles using the provided `expand-all-packfiles` script. Packfiles will come up again later when we push and fetch from remotes, though you won't implement them in this project.

Task 3: `commit`

`git commit` consists of two basic steps:

(1) Create a commit object from the state of the index and HEAD

(2) If HEAD is detached, HEAD now points to the new commit. If HEAD is not detached, update the ref that HEAD references to the new commit.

As covered in *Pro Git* Section 10.2, a commit object consists of a root tree object, along with some metadata. Both commit objects and tree objects are made in a similar manner to blob objects, except their content differs.

Tree Objects

A tree object's content is a series of entries, sorted in ascending lexicographic order by name. Each entry looks like

```
{MODE} {NAME}\0{OBJECT-SHA1}
```

where

- `{MODE}` is the mode integer, printed using ASCII numbers in octal.
- `{NAME}` is the file or directory name
- `{OBJECT-SHA1}` is the hash of the object (blob or tree) with that name, as raw bytes.

For instance, after inflating the `zlib` stream, the start of a tree object might look like

```
tree 1288<00>100644 .gitignore<00><9A><E8><ED><81><DE><64><2B><F0>E<1A><74>
<5D><A3><27><0C><4B><9F><D5><65><93>100644 Makefile<00><73><BB><A0><08><F8>
<50><4D><66><DF><2D><2F><F1><4E><F9><BD><34><38><74><CB><24>40000 dirname
<00><F6><8F><DC><EB><37><E1><3A><39><FF><EF><13><5C><BD><8C><34><3F><F4><8C>
<5B><7C>
```

where selected bytes are displayed as their hex values in angle brackets, and no newlines are actually present. You can inspect the raw form of `git` objects yourself by simply decompressing the on-disk `zlib` stream. On Linux, 'pigz' is a commandline tool that makes decompressing zlib streams simple. For instance,

```
pigz -d < .git/objects/28/e005c09647bbbe5eba0f5b770b3c25842a8f55 | hexdump -C
```

Remember that you can find the hashes of tree objects by running `git cat-file -p <commit-or-branch>`

Commit Objects

Unlike trees, commit objects have purely textual content. The general format is

```
tree {TREE-SHA1}
{PARENTS}
author {AUTHOR-NAME} <{AUTHOR-EMAIL}> {AUTHOR-DATE-UNIX} {AUTHOR-TIMEZOME}
committer {COMMITTER-NAME} <{COMMITTER-EMAIL}> {COMMITTER-DATE-UNIX} {COMMITTER-DATE-TIMEZONE}

{COMMIT MESSAGE}
{BLANK LINE}
```

where

- `{PARENTS}` is zero-or-more lines like `parent {COMMIT-SHA1}`, identifying the parent commits of this commit. There are zero parents in the case of an initial commit, and two or more (though more than two is rare) in the case of a merge commit.
- `{*-NAME}` and `{*-EMAIL}` are pulled from the configuration, and will have the same values in our implementation.
- `{*-DATE-UNIX}` are base-10 ASCII integers representing the current time in number of seconds since the UNIX epoch.
- `{*-DATE-TIMEZONE}` identifies the timezone of the timestamp relative to UTC.
- other fields should be self-explanatory.

Here's a full example from `git` itself:

```
tree cda2c48629c1e926d5b971954a038e79f0f0a325
parent eb804cd405618ef78b772072685c39392aea4ac1
parent bc22d845c4328f5bd896d019b3729f776ad4be4c
author Junio C Hamano <gitster@pobox.com> 1648251504 -0700
committer Junio C Hamano <gitster@pobox.com> 1648251505 -0700
```

```
Merge branch 'ps/fsync-refs'

Updates to refs traditionally weren't fsync'ed, but we can
configure using core.fsync variable to do so.

* ps/fsync-refs:
  core.fsync: new option to harden references
```

Tips for Committing

- Remember that the index stores all the necessary information to create a tree object, and that as a result, all the blobs should already exist.
- The user's name and email can be found using methods given to you in `config_io.h`. They are located in the section "user" with the keys "name", and "email", respectively. Remember that `git` fails to create a commit without these values.
- When there are no commits in a repository, HEAD will typically refer to a non-existent branch. For instance, `git init` initializes HEAD to the non-existent ref `refs/heads/master`.

Task 4: `log`

You will implement a basic version of `git log` that prints a linear history of commits reachable from the given ref or commit name (using HEAD if none is specified) along the first listed parent. When a commit with more than one parent is encountered, simply print the `SHA1` of all parents, and proceed to traverse the first parent.

Ordinary `git log` prints all reachable commits in reverse-chronological order. You are not required to do this.

Your output should be formatted as a series of commit entries, each of which looks like

```
commit <SHA1>
[Merge: <SHA1> <SHA1> [SHA1] ...]
Author: <name> <email-in-angle-brackets>
Date: <date>

<commit message>
<blank line>
```

The merge line is of course only printed if the commit is a merge commit. `<date>` should be formatted using `strftime` with the string `%a %b %d %H:%M:%S %Y %z`.

Task 5: `checkout`

Your implementation of `git checkout` should have the following basic flow:

(a) If the "make_branch" flag is set, then create a new branch pointing to the current value of HEAD, and update HEAD to the new branch.

(b) If the flag is not set, then update HEAD to the checkout name (which may be *either* a ref or a commit hash), and update the work tree and index accordingly.

Files with unstaged modifications that are identical in the old HEAD and the new HEAD should not be modified by the checkout. That is, if the `checkout` won't affect a file the user has modified, then

the checkout should succeed and leave the users changes in the working tree intact. In ordinary `git checkout`, this is also true for staged modifications, but in your implementation you are allowed to just raise an error if there are any staged modifications for simplicity.

For added speed, you are encouraged to leverage the `mtime` field on the index when querying if there are local changes.

Task 6: `fetch`

Our final two commands will implement the basic `git` transport protocol, enabling you to push and pull from remotes, including GitLab. You will implement `fetch` rather than `pull`, as `pull` simply combines `fetch` with a merge or rebase; the fundamental complexity of `git` history exchange is encapsulated in `fetch`.

You are provided with robust starter code that handles the particulars of the network protocol; your job is to initiate and write callbacks for particular events during the client/server exchange. The details of this are documented in `transport.h`. You have also been provided with some code that handles setting up an individual fetch from a remote, based on its configuration.

Your `fetch` should fetch and update all the branches from the remote (this means any remote ref with the `refs/heads` prefix), but avoid requesting any commit hashes that already exist locally. All these refs should be created locally, or modified if they already exist. Recall that you are provided code to deal with ref creation and modification.

As you probably read in `transport.h`, the server ultimately sends a packfile, which, as noted before, is a conglomeration of many `git` objects, possibly with delta compression between them. These are identified by `type == OBJ_REF_DELTA`, and can be decoded using the provided `apply_ref_delta` function.

Task 7: `push`

The procedure for pushing is again described in `transport.h`.

An additional nuance is the question of how to know which branches to push to which remotes. If you look inside `.git/config` in any of your existing repositories, you'll see that most branches have a configured "remote" and "merge" value. Together, these define the "upstream" branch of this local branch, which, in real `git` , affects defaults for `fetch`, `pull`, `rebase`, and sometimes `push`. Note that `push` should only push to branches that have these values configured. See `branch.<name>.merge` in `man git config` for full details.

For your implementation, only concern yourself with the "remote" value, which tells you which remote to push to. Update the remote branch with the same name as the local branch. For a given `push` invocation, each remote should be connected to at most once, so you will need to read these config values upfront. Be sure to process the updating of branch remotes before doing this. If a branch is new, it may not have a config section yet; if its branch remote needs to be updated, this section should be created. If a pushed branch has no remote, your implementation should raise an error.

For compatibility with `git` , when a remote is set, you should also set the "merge" key to `refs/heads/<branch-name>`.

Finally, you should only locally update the remote refs when the server verifies that it has updated them. This is discovered when calling `check_updates()`‘ at the end of the push procedure.

Note that even though `mygit` doesn't implement merging, `push` is required to correctly handle repositories which have merge commits created by `git merge`.

Task 8: Use your `git`

The most satisfying part is exercising your implementation. You should be able to `git init` an empty repository, do some work, and push using only your own git implementation. You will need use regular `git` to configure the remote URL, however, using something like `git remote add origin <URL>`.