Introduction to Version Control and Git

Dave McEwan

Tomorrow 99th May 2099

Introduction

Agenda

- .....1100 Meet and greet.
- 1100..1115 Introduction
- 1115..1200 Background: Revision control, version numbering.
- 1200..1300 Practical/interactive: RCS and git mechanics.
- 1300..1330 Lunch.
- 1330..1430 Local setup: GitLab, SemVer, Sub-modules, CI, etc.
- 1430..1600 Practical/interactive: Merging.

Assumptions

- You are comfortable working with Bash (or Zsh, Tcsh) on Linux.
- You have experience in collaboratively developing code.
- You have experience with some sort of version control before.

. . . and that you have setup these things before today:

- WiFi access.
- `/home/<username>/` with rwx permissions.
- Local or SSH access to Linux machine.

Objectives

- Understand fundamental mechanics of git.
- Be comfortable exploring the internals of git repositories.
- Know how to dig yourself out of common problems with git.
- Understand interactions between our systems.

Roundtable Introductions

- Who are you?
- Who am I?

Background

`diff` and `patch`

- Basis for all version control systems discussed today.
- POSIX since X/Open Portability Guide Issue 4 (1992).
- `diff` takes file1+file2, reports their differences in a patchfile.
- A patchfile precisely describes differences between two files.
  - Most common modern format is “unified” (vs “ed”, “context”).
  - Some context is often required.
  - Differences are identified line-by-line.
- `patch` takes a file+patchfile, applies the differences in-place.
- Inputs must be vaguely similar.

Garbage in, garbage out.

RCS Overview

- https://www.gnu.org/software/rcs/
- 1st generation, but not obsolete!
- Current version is 5.10.1 (2022-02-02).
- Implemented as a collection of utilities (not a single exe):
  - `ci`, `co`
  - `ident`, `rcs`, `rcsclean`, `rcsdiff`, `rcsfreeze`, `rcsmerge`, `rlog`

RCS Important Details

- Stored as separate/unmerged reverse deltas.
  - I.e. delta says how to get back to *previous* version.
  - Predecessor SCCS used forward deltas saying how to get to *next* version.
- Lock indicates that somebody *intends* to deposit a newer revision.
- Branches are on version numbers
  - Symbolic name (similar to a "tag") is a prefix shortcut.
- Concept of joining is similar to merging.
- Stamping, like `$Header$`, expands strings on checkin.
- An edit script is a patchfile.

CVS Overview

- https://cvs.nongnu.org/
- 2nd generation, mostly obsolete.
- Frontend to RCS which adds a client/server model.
- Centralised repository, usually accessed over RSH (predecessor to SSH).
- Composed of fewer executables (`cvs`, `cvspserver`).
- Delta compression distinguishes between text and binary.
- A “project” is a set of related files.
- Introduced “loginfo” similar to git’s hooks.

SVN Overview

- https://subversion.apache.org/
- 2nd generation, mostly obsolete.
- Intended to be successor to CVS, and mostly compatible.
- Many implementations of both clients and servers.
- Centralised repository, accessed over SSH, HTTP, or HTTPS
- Atomic commits to multiple files.
- Properties (key/value pairs) expand the concept of stamping.
- The SVN tool itself is version controlled
  - There are incompatibilities between versions (and that's ok).
- Relies heavily on conventions (trunk, branches, tags) so it's easy to make a mess.
- GUIs are all 3rd-party.

Git Overview

- https://git-scm.com/
- 3rd/current generation.
- Implemented as a single executable (`git`).
- Distributed network of repositories, accessed over SSH or HTTPS.
- Designed to support development of Linux kernel.
  - Thousands of developers, distributed globally, with restrictions on network access, on all sorts of operating systems.
- Every repository has a full copy of history.
- Concepts of branches and tags are first-class citizens with precise definitions.
- GUIs provided in standard distribution, but many 3rd-party tools exist.
- Locking concept doesn't translate well.
  - Close, but slightly different, approximations can be emulated via hooks, branches, and talking to each other.

Git Commits

- Each commit has a unique SHA1 hash.
- Concept of version numbering is intentionally avoided.
- Concept of stamping, like `$Header$`, doesn't exist.
  - Behaviour can be emulated via tags and hooks.
- Tags can be any format you like (please use SemVer).
  - "lightweight" tag is a convenient alias for a commit hash.
  - "annotated" tag is an object in its own right with a checksum, tag message, tagger's name, date, GPG signature.
  - In the same way as branches, tags must be pushed individually.
- Branches are fast, much more convenient than SVN and its predecessors.

Git Hooks

- Run scripts on your local repo on specific events.
- Local events: `pre-commit`, `commit-msg`, `post-merge`, `pre-push`, and more ...
- Remote/server events: `pre-receive`, `update`, `post-receive`
- Must be setup per repository.

SemVer and SemVerSoC

- https://semver.org/
- `<major>.<minor>.<patch>-<pre-release>+<build>`
- Version numbers and the way they change have convey well-defined meaning - that tools can rely on.
- Used for many/most open-source projects.
- . . . See the tiny webpage. . .
- Releases may use SemVerSoC.

CI/CD

- CI: Continuous Integration
  - Run regression on specific events, and complain (loudly, via email) if something is broken.
  - "Push branch X", "Commit to branch Y", "Every Monday at 1200"
  - GitHub has Actions.
  - Great for anything code-based.
- CD: Continous Deployment
  - If the new version passes regression, then deploy it.
  - Can be built into CI flow.
  - GitHub has Dependabot.
  - Great for web development, but not for hardware or critical software.

Practical/Interactive: RCS and Git Mechanics

Preliminary

1. Let's create a space for all of today's practical work.

   - `cd /work/${USER}/`
   - `mkdir -p VCG/morningRCS`
   - `mkdir -p VCG/morningGit VCG/morningSomewhereElse`
   - `mkdir -p VCG/afternoon`

2. `unset noclobber` or `set +o noclobber`
3. Look inside `~/.gitconfig` if it exists.

RCS 1of4

- Objective 1: Feel familiar with inspecting the system.
- Objective 2: Give a point of comparison against git.
- Estimated time: 10 minutes.
- There's a similar tutorial here:
  https://www.madboa.com/geek/rcs/

RCS 2of4

Let's start by making a couple of files to play with.

- `cd VCG/morningRCS/`
- `printf 'hello\nworld\n' > foo.txt`
- `printf 'red\nblue\n' > bar.txt`
- `ls -al ./ RCS/`

And provide a place for RCS to keep its data.

- `mkdir RCS`

RCS 3of4

Next, let's tell RCS to manage these files. By default, RCS will delete the original files on checkin, so you usually want the `-u` option to keep them in the working directory.

- `ci -u foo.txt bar.txt`
- `ls -al ./ RCS/`
- `cat RCS/foo.txt,v`

RCS 4of4

Now, let's try making a change to `foo.txt`

- `echo writableNo >> foo.txt`

But wait! You need to checkout first.

- `co -l foo.txt`
- `echo writableYes >> foo.txt`
- `ls -al ./ RCS/`

And finally, we can observe that we've made changes.

- `rcsdiff`

Git 1of11

- Objective 1: Feel familiar with inspecting the system.
- Objective 2: Be comfortable starting a new repository.
- Estimated time: 30 minutes.
- Follows then extends what we just did with RCS.

Git 2of11

Let's start by making a couple of files to play with...

- `cd VCG/morningGit/`
- `printf 'hello\nworld\n' > foo.txt`
- `printf 'red\nblue\n' > bar.txt`
- `ls -al`

... and initialising the repository.

- `git init`

Git 3of11

Let's have a look what that did to our workspace.

- `find . | sort`

Immediately noticable is that the management structure looks more like a database. Git commands mostly work on these files, so let's see an example of that with `.git/config`.

- `cat .git/config` and/or `git config user.name`
- `git config user.name "Sam Smith"`
- `cat .git/config` and/or `git config user.name`

Git 4of11

Next, let's tell git to manage our files. This has two steps (add to the staging area, then the atomic commit).

- `git add foo.txt bar.txt`
- `git commit -m 'Little message about the changes.'`

Also different from RCS is that we can always edit our working files.

- `echo writableYes >> foo.txt`

Git 5of11

We can see changes between the working directory and the staging area (also known as the cache).

- `git diff`

And changes between the staging area and the commited files.

- `git add foo.txt`
- `git diff --cached`

Git 6of11

The actions of tracking files and viewing differences between revisions in Git are immediately comparable to RCS.

- Can see exactly what is meant by atomicity?
- Can you see how things are arranged in your local workspace?

Git 7of11

Now let's have a look at the distributed and decentralised features. Create a *bare* repository that can work like GitLab/BitBucket/GitHub, then have a look inside. Do this somewhere else on the filesystem - not in the repo you've just created.

- `cd VCG/morningSomewhereElse/`
- `git init --bare MyRepo`
- `ls MyRepo/`

Git 8of11

Back in our original repo, we can add that as a remote for our original. We're using the name `myremote`, but BitBucket and GitHub suggest the name `origin` in their documentation. You can use whatever alias you like - it's your choice and doesn't affect anybody else.

- `cd VCG/morningGit/`
- `git remote add myremote path/to/MyRepo`

Note that this does nothing to MyRepo yet. When we push our original repo, that's when the network access occurs.

- `git push myremote master`

Now have a look around MyRepo where you can see that the contents of `objects/` is identical to those in our local copy. You can add as many remotes as you like, name them however you like, and synchronise them however you like.

Git 9of11

To see a nice overview, you can use the standard GUIs for working with local changes and seeing what your local repo knows about other repos. Let's have a quick look now. Both `git gui` and `gitk`, implemented in Tcl/Tk, are included in the usual distributions of Git and provide a consistent interface on Linux, BSDs, MacOS, and Windows.

- `git gui &`
- `gitk --all &`

Git 10of11

Finally, let's play with a merge which has a little conflict. Merging the changes from a branch in your local repo is the same process as merging the changes from a branch on a remote repo - just fetch first.

- `git checkout -b firstBranch`
- `echo apples >> foo.txt && git add . && git commit -m 'Add apples.'`
- `git checkout master && git checkout -b secondBranch`
- `echo oranges >> foo.txt && git add . && git commit -m 'Add oranges.'`

Finally, return our working directory to the HEAD of `master`.

- `git checkout master`

Git 11of11

We'll try merging the first branch, then the second.

- `git merge firstBranch`
- `git merge secondBranch`

That didn't work because, in the second merge, changes had been made to the same part of the file touched by a commit since their common ancestor. We could also have made this conflict by adding the "apples" line directly on the `master` branch.

To fix the conflict, we need to choose what is right. The simplest way to do this is to accept either change using `git gui`, then modify it manually before the merge is committed. We'll go over different ways of merging in the afternoon.

Local Setup

Outline

- GitLab, BitBucket, GitHub
- SemVer and SemVerSoC
- Jenkins
- Q and A

GitLab, BitBucket, GitHub

- Hooks may prevent rewriting history, like amending pushed commits.
- Hooks may enforce branch our naming scheme.
- Markdown is rendered on all web views.
  - README, commit messages, pull requests
- PDF is also viewable in the web browser, but not DOCX.
- SVG,PNG,JPG are rendered, but not VSDX.
  - SVG can be diffed – Inkscape is recommended for most diagrams and Wavedrom for waveforms.
  - Visio obfuscates SVG!

Git Submodules

- Incorporate other git repositories into your repository.

SemVer and SemVerSoC

- SemVer ascribes meaning to software version numbers.
- SemVerSoC clarifies SemVer's rules for HDL/RTL projects.
- You've probably used SemVer constraint solvers before - in package managers like `apt` (Debian), `rpm` (RHEL/CentOS), `cargo` (Rust), `pip` (Python).

Jenkins

- https://github.com/jenkinsci
- A continuous integration tool.
- Free, open-source software written in Java.

Practical/Interactive: Git Merging

Outline

- Preliminaries
- Example 1: Diffing with tkdiff.
- Example 2: Merging without change.
- Example 3: Merging without conflict.
- Example 4: Merging with conflict.
- Example 5: Merging with kdiff3.

Preliminaries

1. Examples with FSF's `reuse-tool`.
2. Non-merge vs merge commits.
3. GUIs.
4. Stashing.
5. `reset` vs `revert`.
6. Commit message format.
7. `fetch`, `merge`, `rebase`, and `pull`.
8. Nordic's BitBucket branch naming scheme.

Preliminary 1of7: Examples with FSF's `reuse-tool`

- Create a new directory to work on the repository.
  - `cd VCG/afternoon/`
  - `git clone https://github.com/fsfe/reuse-tool.git`
  - `cd reuse-tool`
- Now we can see some history.
  - `git log`
- As well as the branches and remotes.
  - `git branch`
  - `git remote -v`
  - `git branch -a`

Preliminary 2of7: Non-Merge vs Merge Commits

- There two types of commits in Git: non-merge, and merge.
- Non-merge commits are the usual kind.
  - Extend a branch's history.
  - Single ancestor.
  - Must contain differences.
  - Created with `git commit`.
- Merge commits join two branch's histories.
  - Two ancestors.
  - *May* contain differences.
  - Created with `git merge`.
- Have a look at merge commits: `git log`
  - A merge commit with no differences: `git show 95b6dd09`
  - A merge commit with differences: `git show 65b3206a`

Preliminary 3of7: GUIs

- CLI will give you canonical status/results, but GUIs can make some tasks easier.
- `git status`: Visualise status with `git gui`.
  - In the default distribution.
  - Also useful for amending the previous commit.
- `git log`: Visualise branch history with `gitk --all`.
  - In the default distribution.
- `git diff`: Navigate diffs one file at a time with `git difftool`.
  - `git config --global diff.tool tkdiff`
- `git merge`: Resolve conflicts with `git mergetool`.
  - `git config --global merge.tool kdiff3`

Note: `--global` changes `~/.gitconfig`, instead of `./.git/config`

Preliminary 4of7: Stashing

- A handy tool for digging yourself out of a mess.
- Save uncommited and unstaged changes temporarily.
- Easier than saving files to temporary locations.
- Implemented as a stack.
  - Push unstaged changes with `git stash`
  - Pop with `git stash pop`
- Let's try that quickly by making a change, then attempting to pull.
  - `echo BREAKTHINGS >> ./README.md`
  - `git pull --rebase`
- For a rebasing pull to succeed, tracked files must be unmodified. Save our change onto the stash stack.
  - `git stash`
  - `git pull --rebase`
- Now we can get our change back.
  - `git stash pop`

Preliminary 5of7: `reset` vs `revert`

- To undo unwanted changes that aren't commited, reset all tracked files.
  - `git reset --hard`
- That puts your working repo back to `HEAD`.
- To undo changes made by existing commits, we need a new commit which reverses the diff.
  - `git revert <commit>`
- To help yourself and colleagues:
  - Write useful commit messages in the de-facto standard form.
  - Use searchable keywords in commit messages.
  - Review the diff before you make each commit.
  - Keep unrelated changes in separate commits.

Preliminary 6of7: Commit Message Format

- De-facto format supported by GitLab, BitBucket, GitHub, 3rd-party tools, etc.
- First line is a subject, up to 50 characters.
- Second line is empty/blank.
- Subsequent lines are the body, up to 72 characters.
- Subject is in imperitive mood, i.e. commanding.
  - Good: "*Add* support for Foo."
  - Bad: "*Adds* support for Foo."
- Alternatively, if the commit is very simple and doesn't need a body, word the subject as a phrasal noun.
  - "Foo support"
- Both subject and body should be formatted in Markdown.

Preliminary 7of7: Fetch, Merge, Rebase, and Pull

- `git fetch`: Fetch/copy the newest branch and commit data from a remote, but don't change any branches or working files.
- `git merge`: Performs a 3-way merge between two branches and their most recent common ancestor.
- `git rebase`: Take the series of commit patches and apply them to rewrite the branch's history.
  - Rewriting history is not allowed on some server setups. In those systems, you'll rarely (if ever) use the rebase command.
- `git pull`: Execute `git fetch` then `git merge` (or `git rebase`).
  - If you're unsure, use separate steps for fetching and merging.

Diffing with `tkdiff` 1of1

- Let's introduce a harmless change to a tracked file.
  - `echo '## HARMLESS' >> src/reuse/lint.py`
  - `git difftool`
- We can also use `tkdiff` to diff over branches.
  - `F=src/reuse/lint.py`
  - `B1=origin/master`
  - `B2=origin/summary-help`
  - `git difftool $B1:$F $B2:$F`
- First argument is LHS, second argument is RHS.
- Use `n` and `p` to move quickly to next and previous changes.
- See the documentation for all the ways you can diff.
  - https://git-scm.com/docs/git-diff
  - https://git-scm.com/docs/git-difftool

Merging Without Changes 1of3

- This is the simplest kind of merge, where the same changes have been made on converging branches.
- First, let's get back to a known state and make a change on one branch.
  - `git checkout master`
  - `git reset --hard`
  - `git checkout -b myBranchA`
  - `echo '## foo' >> README.md`
  - `git add !$`
  - `git commit -m 'Add foo comment, by Alice.'`

Merging Without Changes 2of3

- Second, get back to a known state and make the same change on another branch.
  - `git checkout master`
  - `git reset --hard`
  - `git checkout -b myBranchB`
  - `echo '## foo' >> README.md`
  - `git add !$`
  - `git commit -m 'Add foo comment, by Bob.'`

Merging Without Changes 3of3

- Now that our changes are commited, we can freely switch branches without losing work.
  - `git checkout myBranchA`
  - `git checkout master`
- To start the merge, first switch to the "destination" branch, i.e. the one that will still be worked upon. Let's choose to merge `myBranchA` into `myBranchB`.
  - `git checkout myBranchB`
  - `git status`
  - `git merge myBranchA`
  - Save and quit the editor
- Now, have a look at the history with `git log` and/or `gitk`.

Merging Without Conflict(s) 1of3

- This is also a simple kind of merge, where the completely different changes have been made on converging branches.
- First, let's get back to a known state and make a change on one branch.
  - `git checkout master`
  - `git reset --hard`
  - `git checkout -b myBranchC`
  - `echo '## foo' >> README.md`
  - `git add !$`
  - `git commit -m 'Add foo comment, by Alice.'`

Merging Without Conflict(s) 2of3

- Second, get back to a known state and make a different change on another branch.
  - `git checkout master`
  - `git reset --hard`
  - `git checkout -b myBranchD`
  - `echo '## foo' >> Makefile`
  - `git add !$`
  - `git commit -m 'Add foo comment, by Bob.'`

Merging Without Conflict(s) 3of3

- To start the merge, first switch to the "destination" branch, i.e. the one that will still be worked upon. Let's choose to merge `myBranchD` into `myBranchC`.
  - `git checkout myBranchC`
  - `git status`
  - `git merge myBranchD`
  - Save and quit the editor
- Now, have a look at the history with `git log` and/or `gitk`.

Merging With Conflict(s) 1of4

- This is the usual kind of merge, where the same changes have been made on converging branches.
- First, let's get back to a known state and make a change on one branch.
  - `git checkout master`
  - `git reset --hard`
  - `git checkout -b myBranchE`
  - `echo '## foo' >> README.md`
  - `git add !$`
  - `git commit -m 'Add foo comment, by Alice.'`

Merging With Conflict(s) 2of4

- Second, get back to a known state and make a conflicting change on another branch.
  - `git checkout master`
  - `git reset --hard`
  - `git checkout -b myBranchF`
  - `echo '## bar' >> README.md`
  - `git add !$`
  - `git commit -m 'Add foo comment, by Bob.'`

Merging With Conflict(s) 3of4

- To start the merge, first switch to the "destination" branch, i.e. the one that will still be worked upon. Let's choose to merge `myBranchE` into `myBranchF`.
  - `git checkout myBranchF`
  - `git status`
  - `git merge myBranchE`
  - ... This needs some attention.

Merging With Conflict(s) 4of4

- Look at the conflicts with `git gui`.
- Search for conflicts with `<<<<<<<`, `=======`, and `>>>>>>>`.
  - That's 7 characters for each kind of conflict marker.
- The format is always the same.
  - Conflicts are marked beginning with `<<<<<<<` (opening chevrons). and ending with `>>>>>>>` (closing chevrons).
  - Each conflict has two parts, partitioned by `=======` (the separator).
  - Lines outside of chevrons belongs to the most recent common ancestor (`master`).
  - First, lines between the opening chevrons and the separator are from the current/destination branch (`myBranchF`)
  - Second, lines between the separator and the closing chevrons are from the branch we want to merge in (`myBranchE`).
- Edit the file, commit, then look at the history.
- A simple strategy is to accept the "remote" version, then make edits, stage, then commit.

Merge Using `kdiff3` 1of3

- Let's repeat the previous example, but use a GUI tool for the merge step.
  - `git checkout master`
  - `git reset --hard`
  - `git checkout -b myBranchG`
  - `echo '## foo' >> README.md`
  - `git add !$`
  - `git commit -m 'Add foo comment, by Alice.'`
  - `git checkout master`
  - `git checkout -b myBranchH`
  - `echo '## bar' >> README.md`
  - `git add !$`
  - `git commit -m 'Add foo comment, by Bob.'`

Merge Using `kdiff3` 2of3

- To start the merge, first switch to the "destination" branch, i.e. the one that will still be worked upon.
  - `git checkout myBranchG`
  - `git status`
  - `git merge myBranchH`
  - `git mergetool`

Merge Using `kdiff3` 3of3

- There are 4 panes.
  - Left (Base) is the most recent common ancestor.
  - Middle (Local) is the current/destination branch (`myBranchG`)
  - Right (Remote) is the branch we want to merge in (`myBranchH`)
  - Bottom is the output of the merge.
- Controls and keyboard shortcuts are well described and intuitive.
- Right-click on conflicts in the bottom pane to select between solutions.
- You can type in the bottom pane.
- Save the output, then use `git gui` to view your changes as usual.

Q and A