Version Control

Consider the following simple scenario:

- a software system contains a source code file `x.c`
- system is worked on by several teams of programmers
- Ann in Sydney adds a new feature in her copy of `x.c`
- Bob in Singapore fixes a bug in his copy of `x.c`

Ultimately, we need to ensure that

- all changes are properly recorded (when, who, why)
- both the new feature and the bug fix are in the next release
- if we later find bugs in old release, they can be fixed

Version Control Systems

A *version control system* allows software developers to:

- work simultaneously and coperatively on a system
- document when, who, & why changes made
- discuus and approve changes
- recreate old versions of a system when needed
- multiple versions of system can be distributed, tested, merged

This allows change to be managed/controlled in a systematic way.
VCSs also try to minimise resource use in maintaining multiple versions.

Unix VCS - Generation 1 (Unix)

1970's ... *SCCS* (source code control system)

- first version control system
- centralized VCS - single central repository
- introduced idea of multiple versions via delta's
- single user model: lock - modify - unlock
- only one user working on a file at a time

1980's ... *RCS* (revision control system)

- similar functionality to SCCS
  (essentially a clean open-source re-write of SCCS)
- centralized VCS - single central repository
- single user model: lock - modify - unlock
- only one user working on a file at a time
- still available and in use

Unix VCS - Generation 2 (Unix)

1990 ... *CVS* (concurrent version system)

- centralized VCS - single central repository
- locked check-out replaced by copy-modify-merge model
- users can work simultaneously and later merge changes
- allows remote development essential for open source projects
- web-accessible interface promoted wide-dist projects
- poor handling of file metadata, renames, links

Early 2000's ... *Subversion* (svn)

- depicted as "CVS done right"
- many cvs weakness fixed
- solid, well documented, widely used system
- but essentially the same model as CVS
- centralized VCS - single central repository
- svn is suitable for assignments/small-medium projects
- easier to understand than distributed VCSs, well supported
- but Andrew recommends git

Unix VCS - Generation 3

Early 2000s... *Bitkeeper*

- distributed VCS - multiple repositories, no "master"
- every user has their own repository
- written by Larry McVoy
- Commercial system but allowed limited use for Linux kernel until dispute over licensing issues
- Linus Torvalds + others then wrote GIT open source distributed VCS
- Other open source distributed VCS's appeared, e.g: bazaar (Canonical!), darcs (Haskell!), Mercurial

Git

- distributed VCS - multiple repositories, no "master"
- every user has their own repository
- created by Linux Torvalds for Linux kernel
- external revisions imported as new branches
- flexible handling of branching
- various auto-merging algorithms
- Andrew recommends you use git unless good reason not to
- not better than competitors but better supported/more widely used (e.g. github/bitbucket)
- at first stick with a small subset of commands
- substantial time investment to learn to use Git's full power

Repository

Many VCSs use the notion of a *repository*

- store all versions of all objects (files) managed by VCS
- may be single file, directory tree, database,...
- possibly accessed by filesystem, http, ssh or custom protocol
- possibly structured as a collection of *projects*

Git Repository

Git uses the sub-directory `.git` to store the repository.
Inside `.git` there are:

- **blobs** file contents identified by SHA-1 hash
- **tree objects** links blobs to info about directories, link, permissions (limited)
- **commit objects** links trees objects with info about parents, time, log message

- Create repository **git init**
- Copy exiting repository **git clone**

Tracking a Project with Git

- Project must be in single directory tree.
- Usually don't want to track all files in directory tree
- Don't track binaries, derived files, temporary files, large static files
- Use **.gitignore** files to indicate files never want to track
- Use **git add** *file* to indicate you want to track *file*
- Careful: **git add** *directory* will every file in *file* and sub-directories

Git Commit

- A git commit is a snapshot of all the files in the project.
- Can return the project to this state using **git checkout**
- Beware if you accidentally add a file with confidential info to git - need to remove it from all commits.
  机密的
- **git add** copies file to staging area for next commit
- **git commit -a** if you want commit current versions of all files being tracked
- commits have parent commit(s), most have 1 parent
- merge produce commit with 2 parents, first commit has no parent
- merge commits labelled with SHA-1 hash

Git Branch

- Git branch is pointer to a commit
- Allows you to name a series of commits
- Provides convenient tracking of versions of parallel version of projects
- New features can be developed in a branch and eventually merged with other branches
- Default branch is called master.
- **HEAD** is a reference to the last commit in the current branch
- **git branch** *name* creates branch *name*
- **git checkout** *name* changes all project files to their version on this branch

Git Merge

- merges branches
- git mergetool shows conflicts
- configure your own mergetool - many choices meld, kdiff3, , p4merge
- kaleidoscope popular on OSX

Git Push

- **git push** *repository-name branch* adds commits from your *branch* to remote repository *repository-name*
- Can set defaults, e.g. **git push -u origin master** then run **git push**
- **git remote** lets you give names to other repositories
- Note **git clone** sets `origin` to be name for cloned repository

Git Fetch/Pull

- **git fetch** *repository-name branch* adds commits from *branch* in remote repository *repository-name*
- Usually **git pull** - combines fetch and merge

Example: Git & conflicts

```
$ git init /home/cs2041
Initialized empty Git repository in /home/cs2041/.git/
$ git add main.c
$ git commit main.c
Aborting commit due to empty commit message.
$ git commit main.c -m initial
[master (root-commit) 8c7d287] initial
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 main.c
```

Example: Git & conflicts

Suppose Fred does:

```
$ cd /home/fred
$ git clone /home/cs2041/.git /home/fred
Cloning into /home/fred...
done.
$ echo >fred.c
$ git commit -m 'created fred.c'
```

Example: Git & conflicts

Suppose Jane does:

```
$ cd /home/jane
$ git clone /home/cs2041/.git /home/jane
Cloning into /home/jane...
done.
$ echo '/* Jane Rules */' >>main.c
$ git commit -a  -m 'did some documentation'
[master 1eb8d32] did some documentation
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Example: Git & conflicts

Fred can now get Jane's work like this:

```
$ git pull /home/jane/.git
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/jane/.git
 * branch            HEAD       -> FETCH_HEAD
Merge made by recursive.
 main.c |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Example: Git & conflicts

And Jane can now get Fred's work like this:

```
$  git pull /home/fred/.git
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From /home/fred/.git
 * branch            HEAD       -> FETCH_HEAD
Updating 1eb8d32..63af286
Fast-forward
 fred.c |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 fred.c
```

Example: Git & conflicts

But if Fred does this:

```
$ echo '// Fred Rules' >fred.c
$ git commit -a  -m 'added documentation'
```

And Jane does this:

```
$ echo '// Jane Rules' >fred.c
$ git commit -a  -m 'inserted comments'
```

Example: Git & conflicts

When Fred tries to get Jane's work:

```
$ git pull /home/jane/.git
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From ../jane/
 * branch            HEAD       -> FETCH_HEAD
Auto-merging fred.c
CONFLICT (content): Merge conflict in fred.c
Automatic merge failed; fix conflicts and then commit the r
$ git  mergetool
```

Example: making Git Repository Public via Github

Github popular repo hosting site (see competitors e.g. bitbucket)
Github free for small number of public repos
Github and competitors also let you setup collaborators, wiki, web pages, issue tracking
Web access to git repo e.g. https://github.com/mirrors/linux

Example: making Git Repository Public via Github

Its a week after the 2041 assignment was due and you want to publish your code to the world.
Create github account - assume you choose *2041rocks* as your login
Create a repository - assume you choose *my_code* for the repo name
Add your ssh key (`.ssh/id_rsa.pub`) to github (Account Settings - SSH Public Keys - Add another public key)

```
$ cd ~/cs2041/ass2
$ git remote add origin git@github.com:2041rocks/my_code.gi
$ git push -u origin master
```

Now anyone anywhere can clone your repository by

```
git clone git@github.com:2041rocks/my_code.git
```

Labelling Versions

How to name an important version? (unique identifier)

Common approach: file name + version "number" (e.g. Perl 5.8.1)

No "standard" for *a.b.c* version numbers, but typically:

- *a* is major version number (changes when functionality/API changes)
- *b* is minor version number (changes when internals change)
- *c* is patch level (changes after each set of bug fixes are added)

Examples: Oracle 7.3.2, 8.0.5, 8.1.5, ...

Labelling Versions

How to store multiple versions (e.g. v3.2.3 and v3.2.4)?
We *could* simply store one complete file for each version.
Alternative approach:

- store complete file for version 3.2.3
- store differences between 3.2.3 and 3.2.4

A *delta* is the set of differences between two successive versions of a file.
Many VCSs store a combination of complete versions and deltas for each file.

Using Deltas

Creating version N of file F (F_N) from a collection of

- complete copies of F whose versions $< N$
- deltas for all versions in between complete copies

is achieved via:

```
get list of complete copies of F
choose highest complete version V << N
f = copy of F{V}
foreach delta between V .. N {
   f = f + delta
}
# f == F{N} (i.e. version N of F)
```

Programs like `patch` can apply deltas.

Delta Bandwidth Efficiency

An example of why deltas are useful:

- Google Chrome for Windows upgrades in the background almost daily
- Google Chrome is 10MB
- bsdiff delta = 700KB
- google custom delta (Courgette) = 80KB
- 200 full upgrades/year = 2GB/year
- 200 bsdiff upgrades/year = 140Mb/year
- 200 Courgette upgrades/year = 16Mb/year