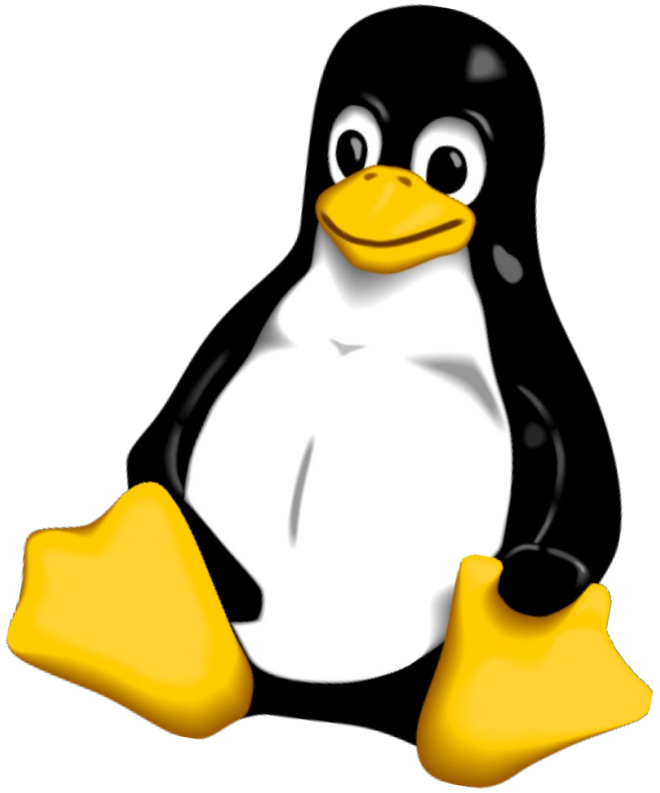
Git Like You Mean it

SOFTIRON



Alan Ott

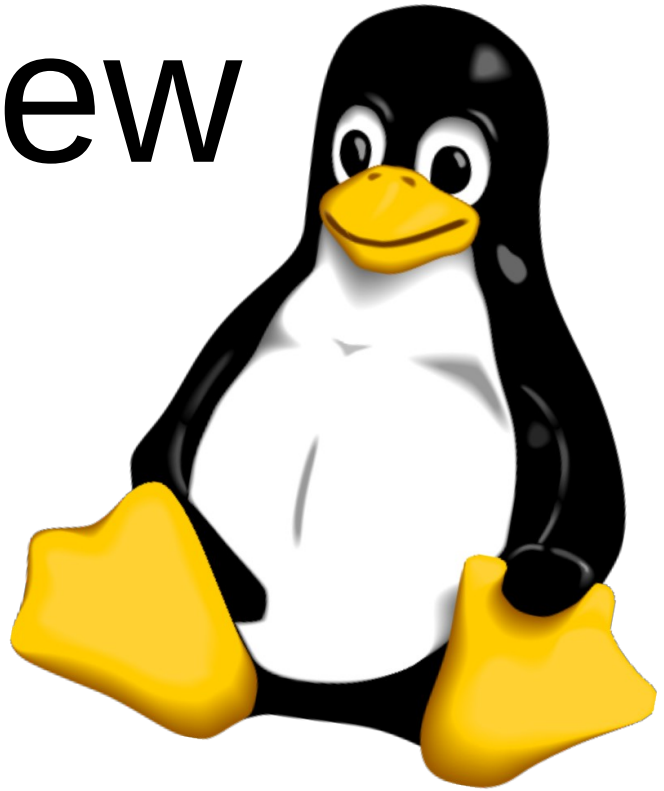
SCaLE 16x

March 8-11, 2018

About the Presenter

- Platform Software at **SoftIron**
  - Data center appliances (storage, transcoding)
  - Ceph-based storage appliances
  - OverDrive 3000/1000 servers
- **OSS Development**
  - Linux Kernel
  - Firmware
  - Training
  - USB
    - **M-Stack** USB Device Stack for PIC
  - 802.15.4 wireless

SCM Overview



Source Code Management

- **SCM**, or **Source Code Management** encompasses the tools and methods for doing revision control of software
  - Usually central, online **repository**
  - Developers commit source code changes (**commits**) to the repository
  - A **log** is maintained of all changes
  - **Branching**, **tagging** support
  - Usually the authoritative source

The Old Way

- Tools like CVS and Subversion use a **centralized** approach to version control
- The **repository** exists on a **server**
- Each user has a **working** copy
- Users commit changes from their working copies to the server
- History is maintained on the server

The Old Way

- Using a centralized system is easy to understand, but it has serious drawbacks:
  - It's hard to preview commits
    - Once commits are made, they are immutable
  - Anything involving the log must fetch information from the server
    - SLOW!

The Old Way

- drawbacks (cont'd):
  - Any kind of branching or merging involves going to the server. There are no local branches.
    - It's a slow process to switch branches
  - Merging is difficult
    - Varies by system
    - Has gotten better over the years
  - SVN lacks tags altogether, and only has “copies.”

Distributed SCM

- In a distributed SCM system, each user has a **full copy** of the repository:
  - Full history, branches, tags, etc.
  - Remote repo and local repo start out identical
- All the functionality of the remote repository is available on the local
  - Local branches, tags, commits, etc.
- A local repo can track multiple Remote repos

Distributed SCM

- *“This is great! I can commit code while I'm on the train!”*
  - This is true, but it's just the tip of the iceberg
- *“This is great, I can look at the log while I'm on the train!”*
  - Also true, but because looking at the log can be done offline, it's fast, and much more useful even when not on the train.

Distributed SCM

- *“This is great! I can create, switch to, and merge branches while I'm on the train!”*
  - Maybe you should move closer to work
  - ➢ *It’s not really about the train, but about all the other implications of being able to work locally.*

Distributed SCM

- What are the implications?
  - Since history is convenient and fast, there's potential for history to actually be used.
  - If history is going to be used, it's important that history be clean.
    - Linear
      - Commits don't depend on later commits
    - Concise commits
      - Germane to one thing
  - How do we create a clean history?

Centralized SCM

- In SVN and CVS, all commits are final and immutable.
  - The documentation says, if you want to change a commit, just make another commit with your change in it. Done!
  - While that “works,” wouldn't it be better to be able to make the commit correctly to begin with?
    - CVS and SVN provide no mechanism for getting it right the first time.

Subversion Example

- Some real SVN commits from my projects:

```
------------------------------------------------------------------------
r812 | alan | 2012-04-25 15:07:41 -0400 (Wed, 25 Apr 2012) | 2 lines

Integration fixes from [redacted] facility.



------------------------------------------------------------------------
r757 | alan | 2011-12-29 17:43:38 -0500 (Thu, 29 Dec 2011) | 2 lines

Bump version to 3.5.2.

------------------------------------------------------------------------
r756 | alan | 2011-12-29 17:43:02 -0500 (Thu, 29 Dec 2011) | 2 lines

Bump version to 3.5.2.
```

Subversion Example

- Real SVN Commits (cont'd):

```
------------------------------------------------------------------------
r449 | alan | 2010-07-30 13:21:12 -0400 (Fri, 30 Jul 2010) | 2 lines

Forgot to add.

------------------------------------------------------------------------
r448 | alan | 2010-07-30 12:46:43 -0400 (Fri, 30 Jul 2010) | 2 lines

Updated for version 1.2.4
```

- “Oops” commits
- Commits with too many things combined

Subversion Limitations

- Committing code:

```
svn commit file1.c file2.c file3.c
```

- What if you forget one?
- What if you have some debug code in those files that you don't want to commit?
- What if you munge the commit message?
  - *No way to go back and fix these issues*

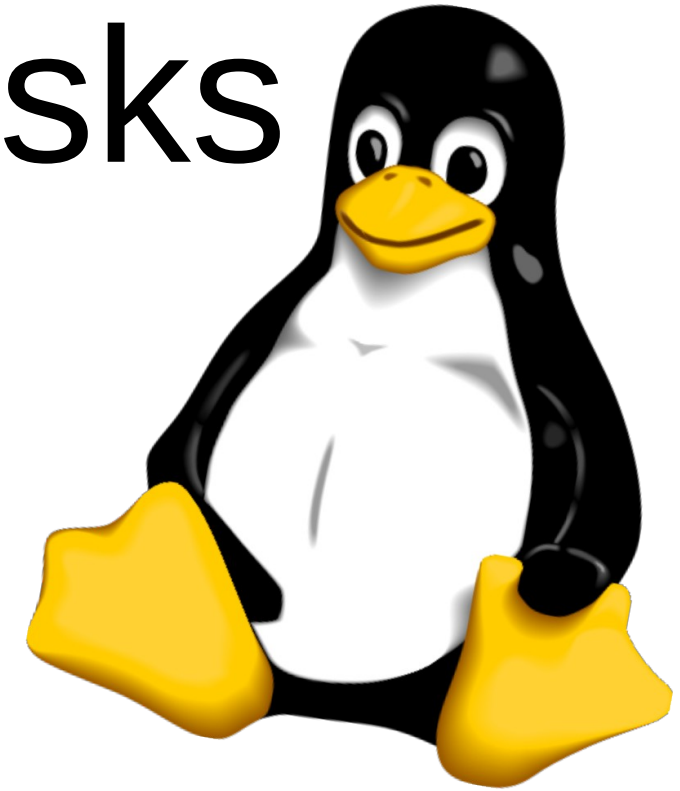
Git SCM

- Git provides **tools** to address these issues:
  - Local history is **editable**
  - **Separate add** and **commit** steps make it easy to **review** what is being committed
    - This allows **small**, atomic commits
    - This allows creation of a **linear history**
  - If you mess up, you can go back and fix it
  - Once it's right, push to the remote repository

But Why?

- *“So we can clean up the history, but who really cares? I don't look at it anyway.”*
  - Small, clean commits make **code review** possible.
  - They make problems harder to introduce
    - Fewer unintended consequences
  - They make **bisection** possible
    - *These together contribute to increased code quality!*

Basic Git Tasks



Basics of Git

- To create a git repository in a directory:

```
git init
```

- This creates a directory `.git` which contains the repository
- The actual files in the current directory are the **working copy**.
- No files are automatically added to the repository

Basics of Git

- Adding files to the git repository

  `git add <filename>`

- This adds a file to the **index** or the **staging area**.
- This file is not committed, but it is **staged for commit**.
- `git add` can be run multiple times to stage commits from **multiple** files for a **single** commit

Basics of Git

- Commit files:

```
git commit
```

- This **creates a commit** containing **all** the changes which are in the **index**, meaning all changes which are **staged for commit**.
- It will open your editor to prompt you for a commit message

Basics of Git

- After running the previous command, your repo has a single commit.
- You can view this log with:

```
git log
```

- This opens up the log in `less`. Press `q` to exit.
- View the **diffs** with each commit:

```
git log -p
```

Basics of Git

- After the initial commit, you will likely change and add code (ie: do normal development).
- Show the status of the working copy:

```
git status
```

- This shows:
  - Files added, changed, removed
  - Conflicts
  - Files staged for commit

Basics of Git

- Show all the code that's been changed in the working copy:

```
git diff
```

- This shows a colorized diff and pages it with `less`.

Basics of Git - Committing

- At some point you will have multiple commits worth of changes in your working copy
  - Once you've developed a new feature and debugged it
- **Individual changes** can be staged for commit with:

```
git add -p
```

- This will prompt to add each chunk to the index (ie: stage it).

Basics of Git

- Once you've added all the diff chunks for a commit, you should **review** them with:

```
git diff --cached
```

- This will show you all the changes **staged** for commit (ie: all the changes in the **index**) in diff format.
  - *It's important to do this review. Sometimes you will confuse yourself about what you added.*

Basics of Git

- If there's something you don't like, you can change it.
- To make a change, change the file, and then run `git add -p` on that file again.
  - The index will be updated.
- To remove changes in a file, run:

  `git reset HEAD <filename>`

  - This removes all changes for *filename* from the index.

Basics of Git

- Once you have reviewed your changes, you can commit them with:

```
git commit
```

- This will create a single commit
- It will pop up your editor and prompt you for a **commit message**.

Basics of Git

- Now that you've made this commit, run `git log` to look at it.
- Make sure it looks the way you expect it to look.
  - The actual commit message will be indented four characters
  - *Checking your work is always important!!*

Basics of Git

- If the commit message is not right, you can easily change it with:

```
git commit --amend
```

- This will edit the log message of the topmost commit (HEAD).
- Other commit messages can be changed with rebase (later).

Basics of Git

- The first commit is made. Run `git diff` again to see the remaining changes, and iterate until finished.
- To get more information, run

  ```
  git status
  ```

- This will tell you:
  - Changes in the working copy
  - Changes staged for commit
  - How to revert changes

Basics of Git

- Commit messages
  - Git has a de facto standard for commit messages
    - First line: subsystem and short description
    - Blank Line
    - Long description
  - Use the **imperative mood**
    - *Fancy English term for “commands”*
    - “Change the...”
    - “Add support for...”
    - “Handle exception in...”

Sample Commit

```
commit e09730a04ebc8d8a4cd436d2eaa6141b7d02c3bd
Author: Alan Ott <alan@signal11.us>
Date:   Fri Apr 28 19:51:37 2017 -0400

    cdc_acm: Handle getting and setting of line coding

    Handling of SET_LINE_CODING and GET_LINE_CODING are
    required, and Windows terminals will fail if those
    requests fail.

    This used to not fail on Windows, but that was by
    accident as there was no way to reject (stall) the
    data stage of control transfers.  The behavior which
    triggered this failure was likely caused in
    d1b95bc5c6dadec and then c685ebbec1cf267.
```

Commit Info

`commit` **`e09730a04ebc8d8a4cd436d2eaa6141b7d02c3bd`**

- SHA1 hash of the commit
  - The SHA1 hash of **everything** in this commit (patch, date, time, author, email, commit message, etc.), plus the hash of the **previous commit**.
  - This commit ID serves as a **unique identifier** of this point in the history.
  - It identifies the **commit**, and since it includes the previous commits, it also identifies indirectly the **branch**.

Commit Info

`commit` **`e09730a04ebc8d8a4cd436d2eaa6141b7d02c3bd`**

- The commit ID is used everywhere a git **revision** is called for.
  - Compare to the revision ID in SVN
- A **prefix** can be used (provided it's unique)
- Can be used with git checkout (for example):

  ```
  git checkout e09730a04eb
  ```

  - Will check out a specific version to the working copy.

Commit Info

```
Author: Alan Ott <alan@signal11.us>
Date:   Fri Apr 28 19:51:37 2017 -0400
```

- Author and commit date
- The date is the date of the commit, not the date that it's merged or pushed to a remote.
  - New commits can have old dates.
  - Dates are not necessarily increasing.
  - Commit dates are not a good way to determine the state of the repository on a particular date.

Commit Message

```
cdc_acm: Handle getting and setting of line coding

Handling of SET_LINE_CODING and GET_LINE_CODING are
required, and Windows terminals will fail if those
requests fail.

This used to not fail on Windows, but that was by
accident as there was no way to reject (stall) the
data stage of control transfers.  The behavior which
triggered this failure was likely caused in
d1b95bc5c6dadec and then c685ebbec1cf267.
```

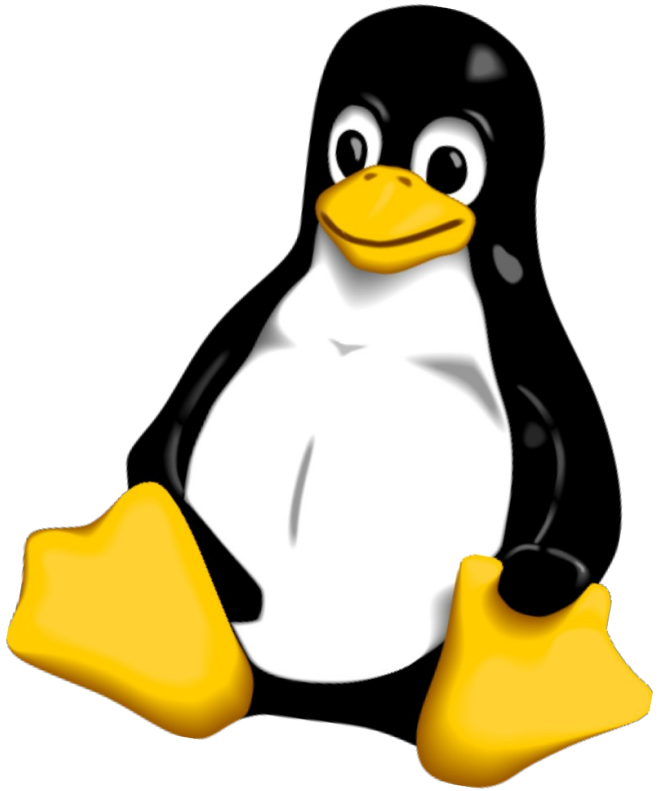
- Subsystem and Short description
- More Info in long description
- References other commit IDs

Basics of Git

- Interactive Demo #0

```
git init
git add
git commit
git log (-p)
git diff
git add -p
git diff --cached
git commit
```

Ranges



Ranges

- Some git commands take a **range** parameter
  - For example, you might want to look at a diff between two different commits
- Example:

```
git diff d1b95bc5..c6dadec1
```

  - This shows the differences between revisions `d1b95bc5` and `c6dadec1`

Ranges

- The same can be done with `git log`
- Example:

  ```
  git log d1b95bc5..c6dadec1
  ```

  - This shows all the **commits** which are in `c6dadec1` but not in `d1b95bc5`.
- Remember that ranges are:

  `older..newer`

Ranges

- You can also use symbolic names, meaning the name of a branch or tag:

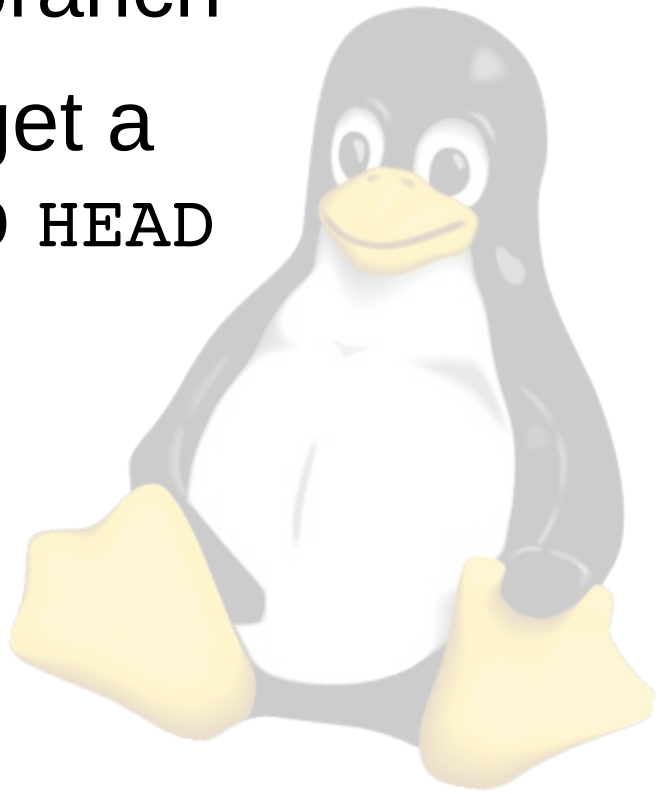
```
git log v4.13..v4.14
```

  - This shows the commits which are version v4.14 which are not in v4.13
    - V4.14 and v4.13 are **tags**

Ranges

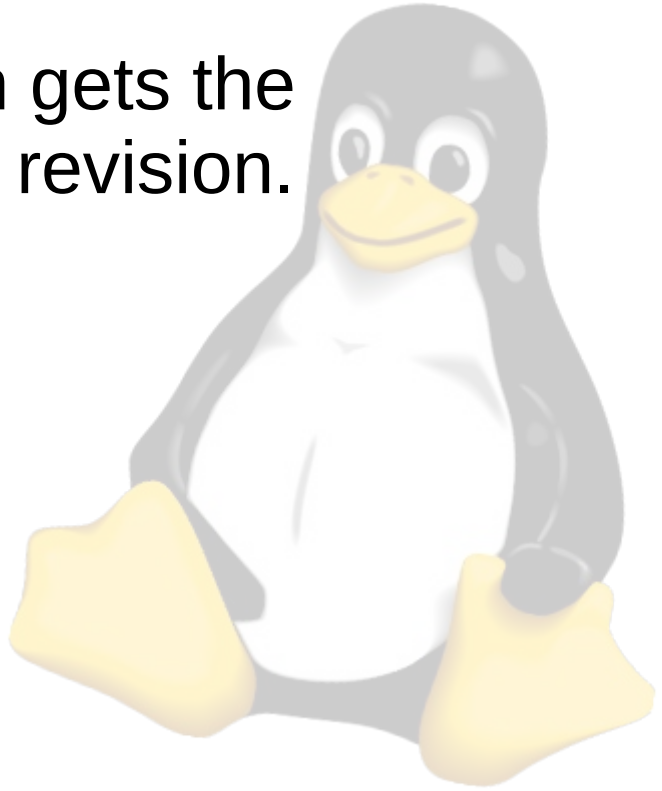
- One revision has a special name: `HEAD`
- `HEAD` describes the most recent revision which is checked out in your current branch
- Sometimes you might want to get a log or diff from some revision to `HEAD`

```
git log v4.14..HEAD
```



Ranges

- You can reference a commit previous to a specified revision.
  - This is often done relative to HEAD:
  - Putting a carat (^) after a revision gets the revision previous to the specified revision.
  - Multiple carats can be used:
    - `git log HEAD^..HEAD`
    - `git log HEAD^^^^..HEAD`



Ranges

- Instead of using multiple carats, a tilde can be used with a number, indicating the number of commits to go back.

```
git log HEAD~4..HEAD
```

  - is equivalent to:

```
git log HEAD^^^^..HEAD
```

  - which will show the last 4 commits.

Ranges

- As a shortcut, the later revision can be omitted, and HEAD will be **assumed**. Thus:

```
git log HEAD~4..HEAD
```

  - is equivalent to:

```
git log HEAD~4
```

- See the man page for gitrevisions:

```
man 7 gitrevisions
```

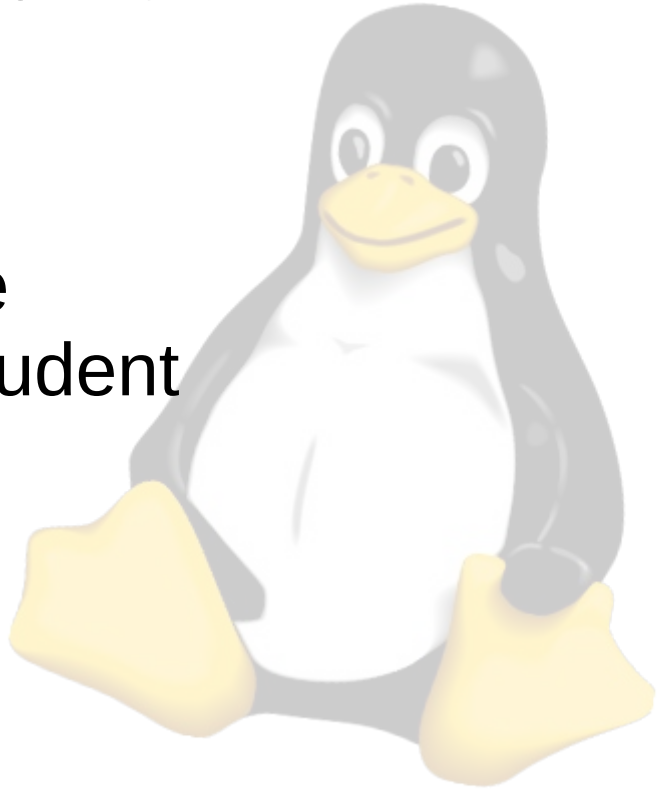
Branch and Merge

Branches

- Remember that in a distributed SCM system, the **entire repo** exists on your workstation.
  - This includes branches and tags
- Branches can be made **locally** which do not correspond to upstream
- Branches are extremely **fast** to create and fast to switch between
  - It's very convenient to create local feature branches for work.

Branches

- Git will always give you a default branch called `master`.
  - This is typically the branch synced with upstream.
  - It's your “trunk”
  - There's nothing special about the mechanism of `master`, but it's prudent to use the standard policy.



Branches

- To view local branches

  `git branch`

  - *Current branch is starred*
- To create a branch:

  `git branch <branch_name>`

  - The branch is not checked out yet!
- To check out the new branch:

  `git checkout <branch_name>`

Branches

- Checking out a branch checks out the branch to the working copy.
  - The working copy will contain the branch
  - The log will reflect the branch
  - Commits made will be made to the branch
- You can switch between branches easily with git checkout.
  - *Make sure to commit your changes!*

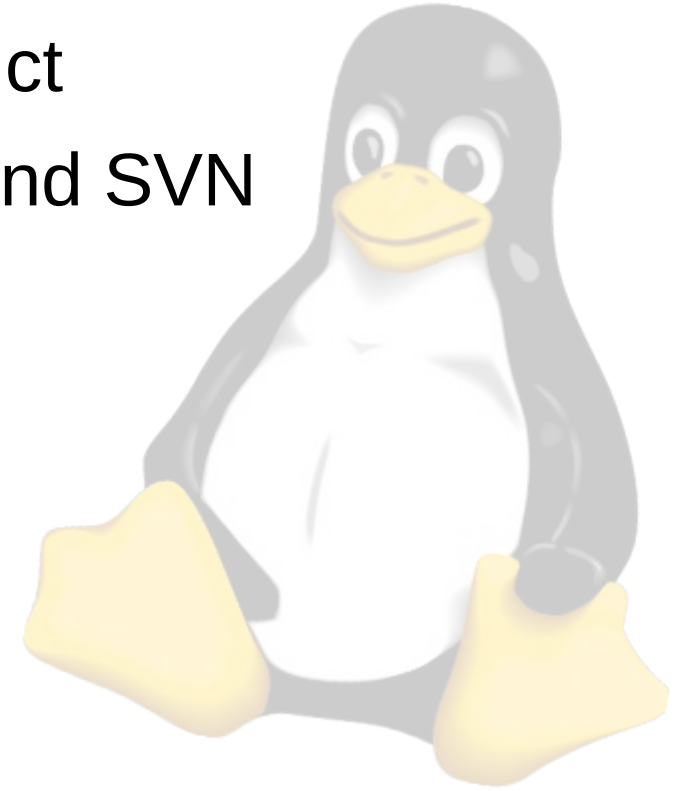
Branches

- It's convenient to use branches to work on a feature:
  - Make a branch to work on a feature
  - Check out the branch
  - Commit changes to the branch
  - Checkout the `master`
  - Merge the branch to the `master`

```
git merge <branch_name>
```

Branches

- Hopefully your branch can be merged. If not, git will often tell you what needs to be done to make it happen.
  - Sometimes you will have a conflict
  - Conflicts work similarly to CVS and SVN

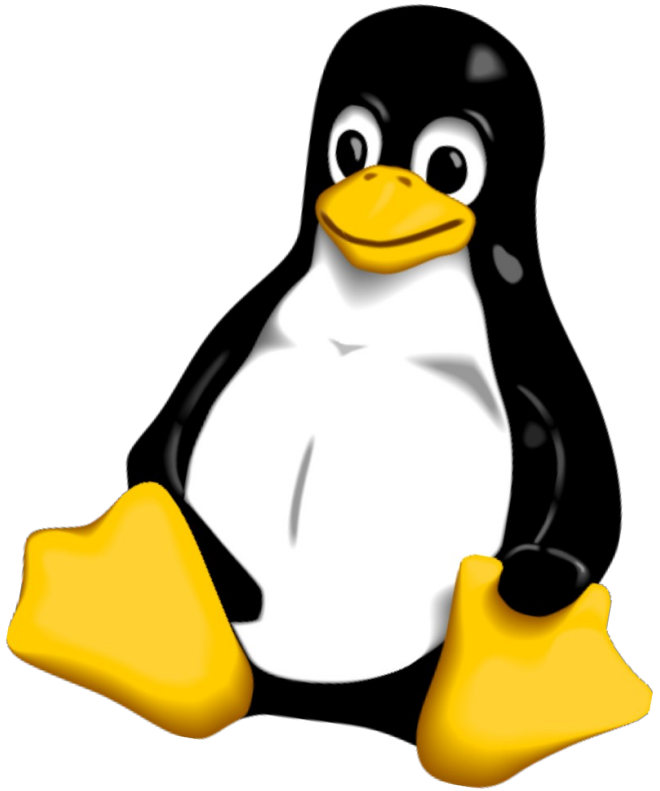


Branch and Merge

- Interactive Demo #1

```
git branch
git log (with ranges)
git diff (with ranges)
git merge
```

Revisions



Revisions

- `git checkout` can be used to check out a specific revision, a branch or a tag:
  - git checkout `d1b95bc5`
  - git checkout `working_branch`
  - git checkout `HEAD^`
  - git checkout `HEAD~20`
    - *This makes it very easy to work with old revisions*

Stashing

- When switching between revisions with `git checkout`, sometimes there will be a conflict.
  - git checkout will refuse to alter the working tree in this case
- To get around this case, `git stash` can be used to store away all your working directory changes.
  - This leaves a clean working directory

Stashing

- Once the directory is clean, you can try the `git checkout` again.
- Once the new version is checked out, you can then get your stashed changes back with `git stash pop`

Stashing

- Git maintains the **stash list** (a stack) of stashed diffs.
  - `git stash list`
  - `git stash show`
    - View an individual stash diff

    ➢ *Stashing is a good way to temporarily clean your working tree*

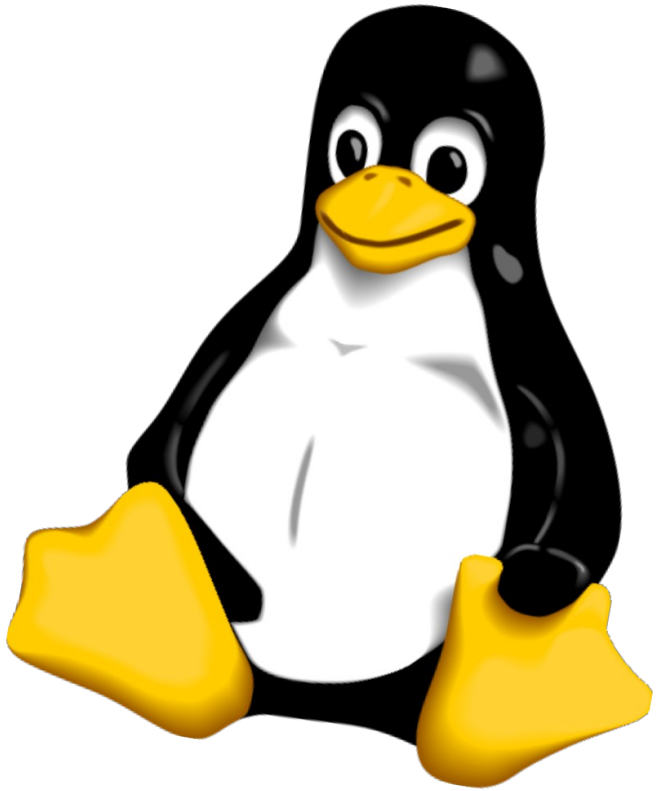
Reset

- `git reset` will manipulate the history to set your current revision and working copy to a certain state.
  - It has several uses, but here we focus on resetting to a previous revision
  - `git reset <revision>`
    - Set the current `HEAD` to <revision>
    - **Leave** the working copy as-is
    - **Remove** the log history for all commits **after** <revision>

Reset

- `git reset` has several modes.
  - By default, `--mixed` mode is used.
  - `--hard` resets the commit history and also the working tree
    - Be careful, you will lose data!
  - `--soft` resets the commit history and **leaves** the changes but also puts them in the index (staged for commit)
  - See man page for `git-reset`

Rebase



Rebase

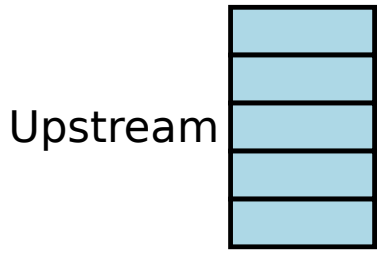
- Probably the single greatest feature of git is the ability to **rebase**.
  - History is editable
    - This might seem scary, but it's not.
    - Remember that with distributed SCM, there is **local** history.
      - Commits which are not pushed to a remote
    - Usually you will only edit **local** history **before** it is pushed to a remote repo
      - Get the commit set right, and then push!

Rebase

- What is rebasing?
  - Simple rebasing involves inserting commits (from another branch, or upstream) ***before*** your local commits.
  - Use case
    - You have a working branch, and have made commits
    - Someone else pushes commits to the remote repo
    - You want your commits to be on top of the remote commits

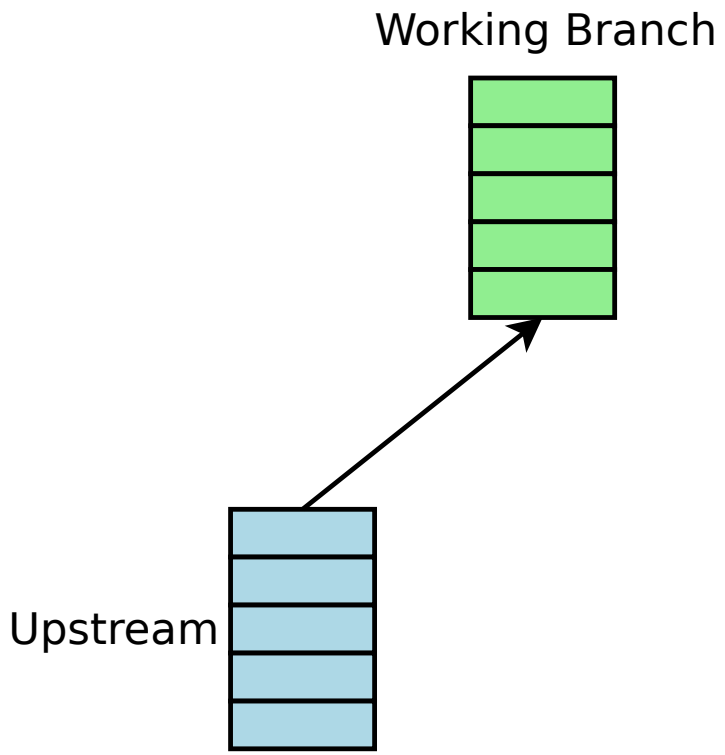
Rebase

- In this example, the state of the repo before starting work just contains a master branch.
- The master branch contains the main line of development.
- It is potentially fast-moving.

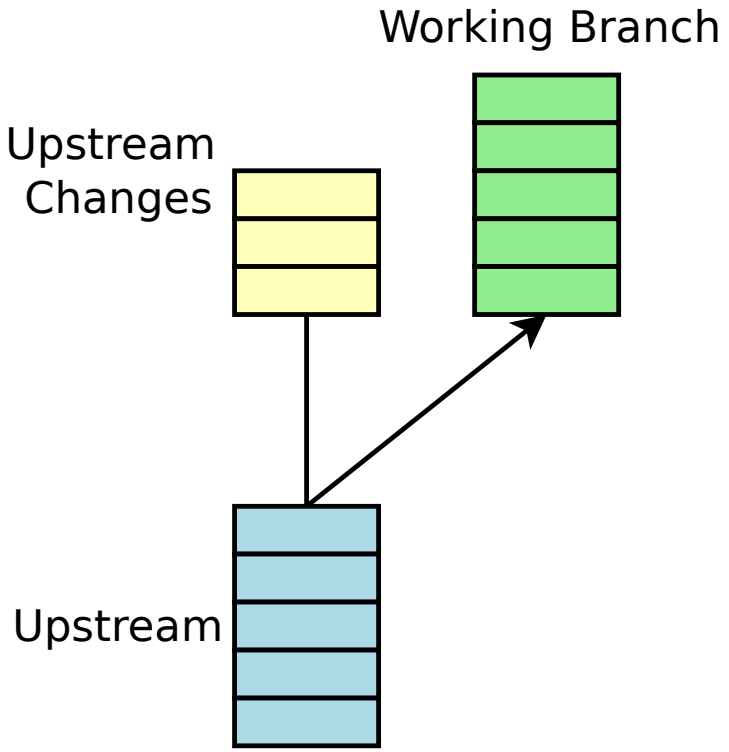


Rebase

- Working branch is created
- Branch-local commits are made to it

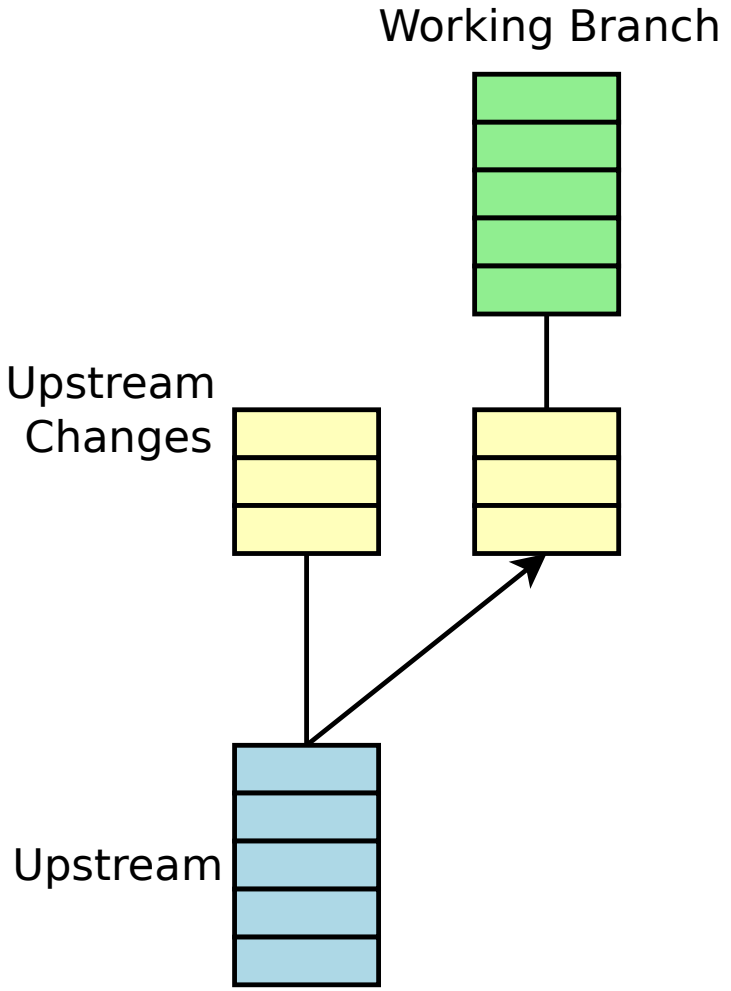


Rebase



- Commits are made to the master branch, maybe from upstream.
- Master and working branch now diverge.
- (If you want the commits in the working branch to go into the master (upstream), they may have to be modified to remove conflicts.)

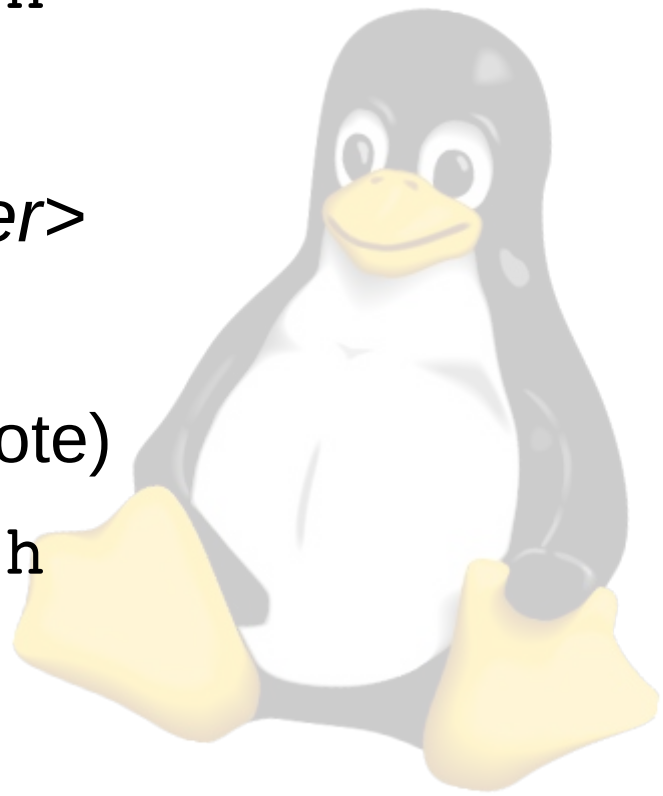
Rebase



- A `rebase` operation on the working branch will:
  - Pop the branch-local commits out of the history and save them
  - Apply the commits from master to the working branch
  - Re-apply the saved branch-local commits to the working branch

Rebase

- Procedure used:
  - `git branch working_branch`
  - `git checkout working_branch`
  - *<make new commits>*
  - *<assume others commit to master>*
  - `git checkout master`
  - `git pull` (get changes from remote)
  - `git checkout working_branch`
  - `git rebase master`



Rebase

- `git rebase master`
  - **Save** all commits which are in `working_branch` but not in `master` to a temporary area
  - **Reset** the `working_branch` to the most recent commit which is common to `working_branch` and `master`
  - **Apply** all the commits from `master` which are not in `working_branch` to `working_branch`
  - **Apply** the saved commits (from above) to `working_branch`.

Rebase

- It's possible that there was a **conflict** when applying the saved commits.
  - Rebase will stop on the conflicting commit and give you an opportunity to fix it.
  - At this point you can run:
    `git rebase --abort`
    to revert back to before the rebase
  - You can fix the commit and run:
    `git rebase --continue`

Rebase

- If conflicts or exception cases occur, remember that:
  - Git will often tell you what to do.
  - Running `git status` will tell you what to do.

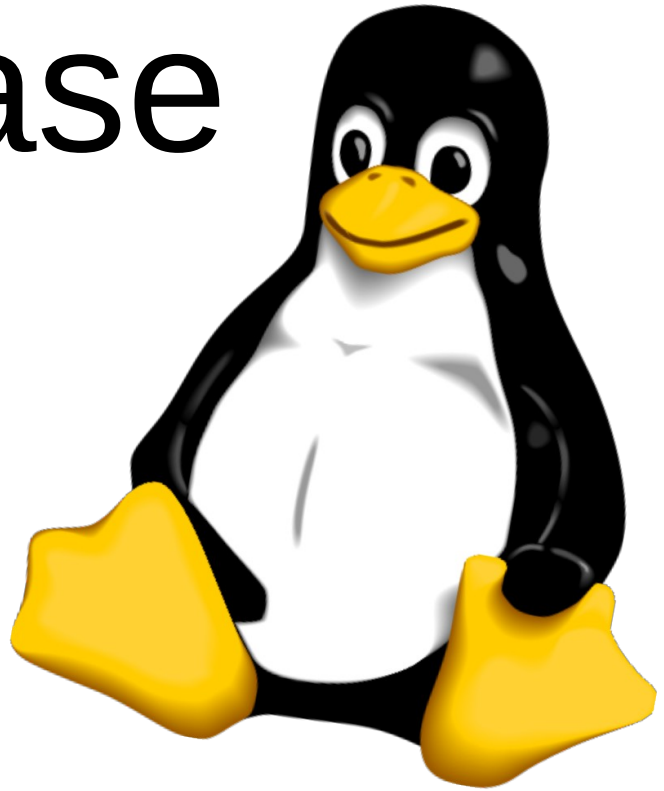
    - *It helps to open a **second window** in an exception case, so that any instructions remain on the screen and you don't forget the state*

Rebase

- Interactive Demo #2

```
git add -p
git commit
git log (with ranges)
git diff (with ranges)
git rebase
```

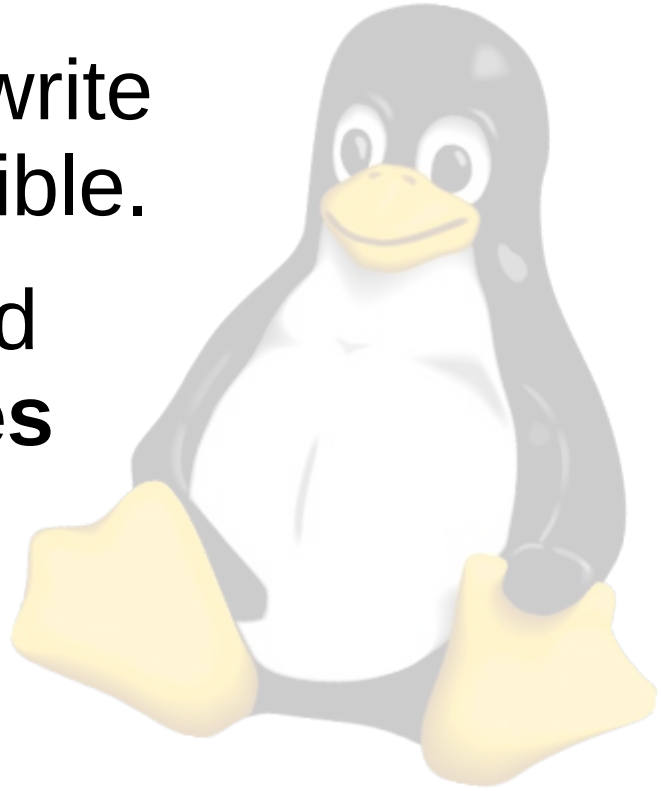
Interactive Rebase



SCALE 16x

Rebase

- **Simple rebase** is used to integrate the history of one branch into another branch
  - History is rewritten
- **Interactive rebase** will also rewrite history, but it is much more flexible.
- Remember that rebasing should only be done on **local branches** which are not pushed to a remote!!



Rebase

- Interactive rebasing will allow you to drastically rewrite history.
  - Only do this on a local working branch
  - Don't rebase commits that have been pushed to a remote repository!
- Use rebase to integrate changes suggested during code review.
  - This can happen before a branch is merged into master!

Rebase

- Rewriting history is important during the development process.
  - Making history simple (simple commits)
  - Making commits linear
    - Each commit builds on previous commits
    - Commits don't undo one another
    - No “oops, forgot to add” fixes
      - Just rebase and fix it!
  - Making every point in history valid
    - No commit breaks the build

Rebase

- To start, run `git rebase -i <revision>` which will rebase all commits newer than `<revision>`.
- Git will open up your editor with a list of commits, one per line.
- Each line indicates:
  - What to do with the commit
  - The sha1 and commit message

Rebase

- From this window, you can:
  - Delete commits
    - Delete the line
  - Reorder commits
    - Move the lines around (copy/paste)
  - Modify the commits
    - Edit the first word of the line

➢ *The file tells you what to do!*

Rebase

- Modification commands:
  - p (pick), leave as is (default)
  - r (reword), change commit message
  - e (edit), change a commit
  - s (squash), combine with previous commit
    - There are others, but these are the basics

Rebase

- It's easy to move commits around.
  - The problem comes when there's a conflict
    - Normal conflict resolution will occur.
    - You will be able to resolve the conflicts manually.
  - Be careful when editing the rebase command file.
    - Deleting a line will remove that commit!

Rebase

- It's best to use two windows for this:
  - One window for the `git rebase` commands
    - When rebase stops (either for error or for editing) it will tell you the next steps. It's good not to lose this information.
  - Another window to do the edits
    - Reset, edit, add, etc.

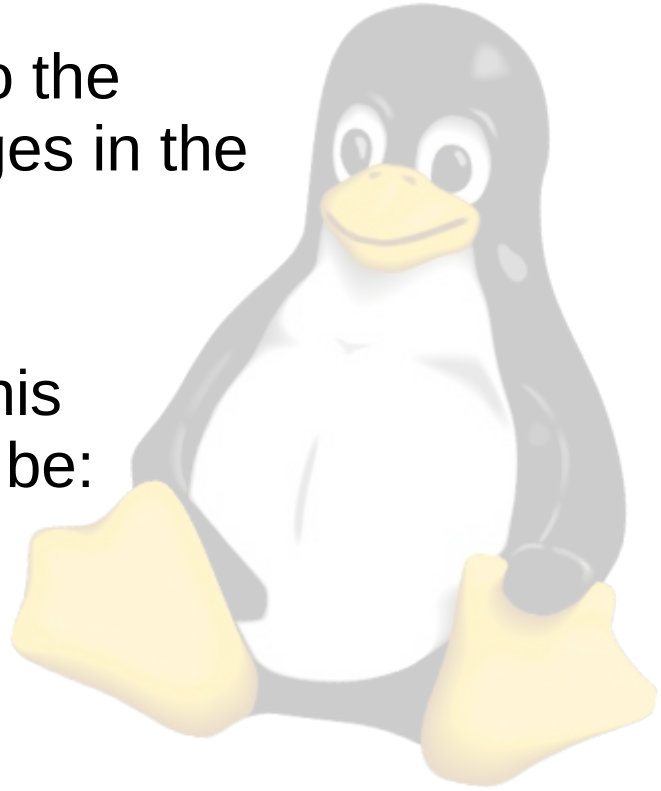
Rebase

- Splitting Commits
  - Interactive rebase can be used to split commits
    - Mark a commit with “e” for edit.
    - When rebase stops, reset the log to the previous commit, leaving the changes in the working tree:

```
git reset HEAD^
```

    - With the commit removed, create this commits the way you want them to be:

```
git add -p
git commit
...
```



Rebase

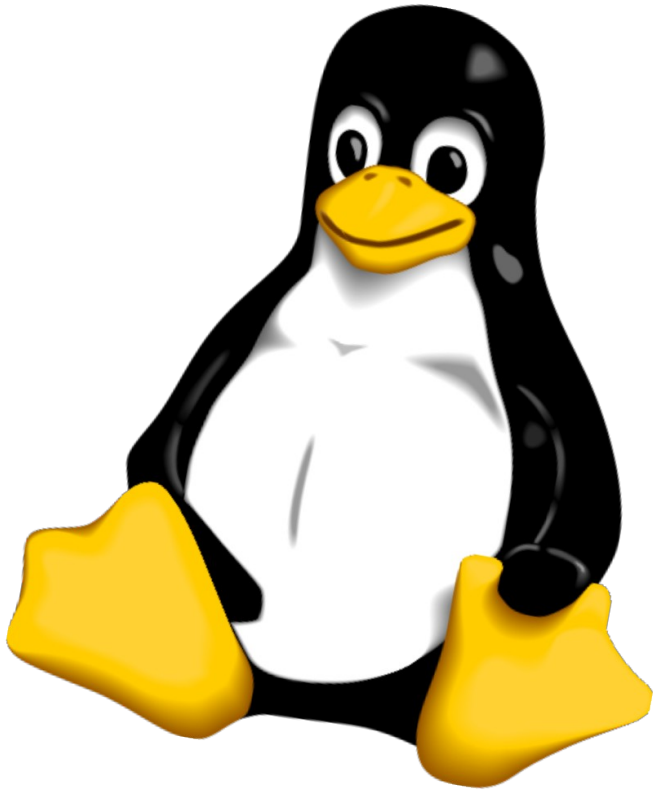
- Interactive Demo #3

```
git rebase -i
git reset HEAD^
git add
git commit
git rebase --continue
```

Rebase

- Important things to remember:
  - Rebase is destructive!
  - Make sure you backup
    - Make a new branch as a backup
    - Copy the whole repo (cheap insurance)
  - Double-check what branch you're on
    - It’s easy to be wrong
  - Know whether you're up to date

Remotes



Remotes

- So far we've dealt with **local** repositories
- Git supports **remote** repositories
- Several protocols are supported:
  - git, http, ssh, local filesystem
    - For remotes that you can authenticate to, you will often use **ssh**
    - **http/git** is often be used for anonymous access (for OSS projects)
    - **Filesystem** is good for cloning a repo that exists on your disk

Remotes

- Git supports **multiple** remotes
  - Push to and pull from each
- This is because git can **detect** when repos have a **common history**.
  - It’s easy using the sha1
  - The sha1 revision is not tied to the repo, but to the actual history.
- Some example use cases of multiple remotes...

Remotes

- Linux Kernel and Stable repos
  - Pull master from linux
  - Pull branches from linux-stable
- Public/private repo
  - Push/pull to/from private repo
  - Push to Github
- Working Repo
  - Pull from upstream OSS repo
  - Push/pull to/from private working repo
  - Send patches to mailing list

Remotes

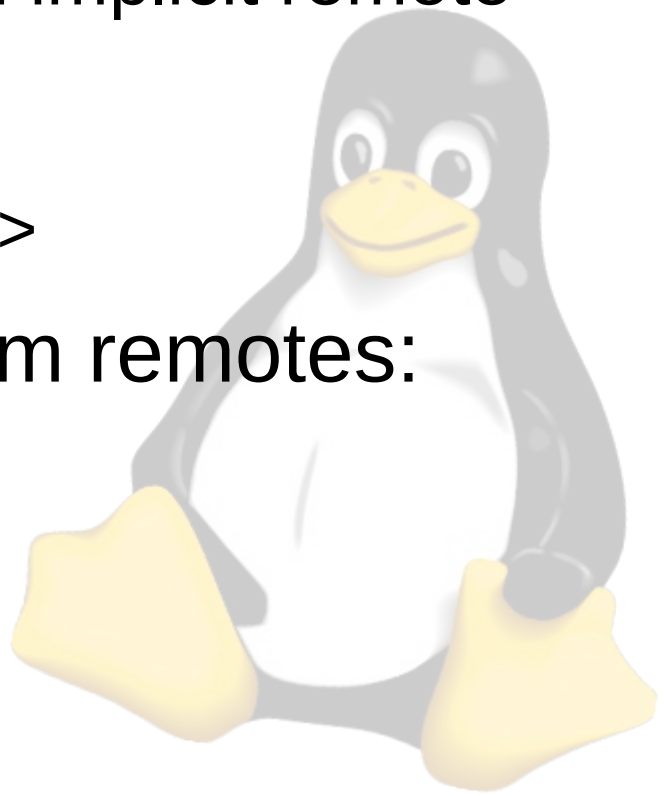
- Public Working Repo
  - Pull from upstream OSS repo
  - Push to personal, public working repo
    - git.kernel.org, for example
  - Send pull requests to maintainers

Remotes

- Getting a repo from a remote is called **cloning**.
  - `git clone git@github.com:signal11/m-stack.git`
    - **ssh** syntax, requires ssh public key
  - `git clone git://git.kernel.org/pub/.../linux.git`
    - **git** protocol; anonymous
  - `git clone https://git.kernel.org/pub/.../linux.git`
    - **https** protocol; use behind firewalls
- These commands clone the repo, including all history of the master branch.

Remotes

- Cloning a repo gives a single remote called `origin`.
  - `origin` is sometimes used as an implicit remote
- Other remotes can be added:
  - `git add remote` *<name>* *<URL>*
- You can checkout branches from remotes:
  - `git checkout remote/branch`



Remotes

- Inspect which remotes are being tracked
  - `git remote -a`
- Branches checked out will automatically **track** the remotes from which they came.
  - Push/pull will operate as expected
- See which branches are available on the remote:
  - `git branch -a`

Remotes

- Clone with reference (optimization)
  - Sometimes you want a new copy of a repo you already have.
  - Use `--reference` to provide clone a local repository to use as a reference.
    - git will use hard-linking to save disk space
    - Clone operation will be faster
  - `git clone --reference /path/to/linux \ git@git.kernel.org/pub/.../linux.git`

Remotes

- Fetching
  - Remember that your local git has the capacity to be a complete repository
    - branches, tags, etc, (collectively called **refs**)
  - `git fetch` will fetch all refs from a remote to your local repo.
    - It will not modify your working copy

Remotes

- Push and pull
  - To get updates from a remote, **pull** from the remote:
    - `git pull origin master`
      - Pull from the remote **origin**, the **master** branch
      - Origin and master are implicit
        - Could be simply `git pull`
  - It's best to not do it like this.
    - If your local branch has local commits your history will diverge from upstream
    - In this case you'll get a **merge commit** which is probably not what you want.

Remotes

- You are better off pulling with `--ff-only`
  - This ensures that only **fast-forward** merges are allowed
    - This means the local repo has no local commits.
  - `git pull --ff-only`
  - If it can't fast-forward, it will simply fail
    - Then you can figure out why, fix it, and try again.

Remotes

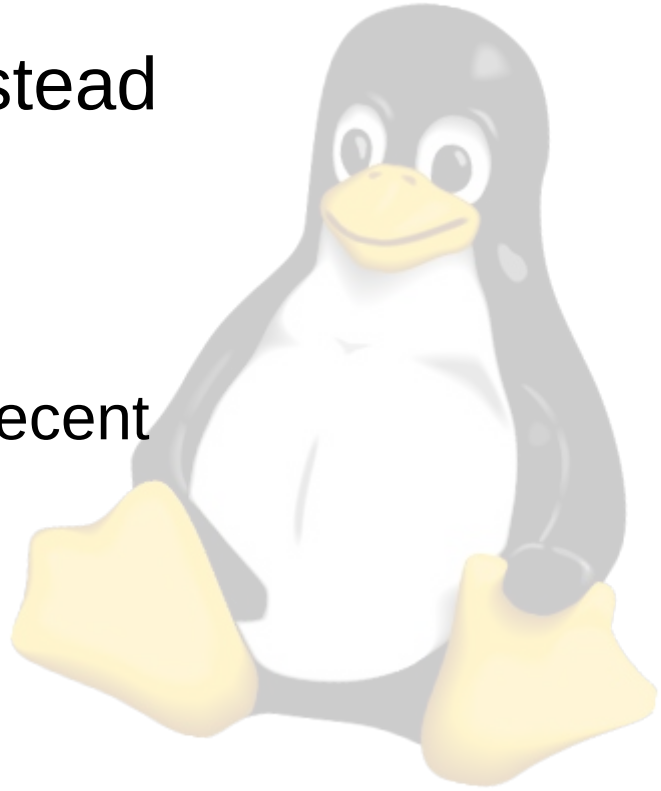
- Avoiding pull issues:
  - Have a **master** branch that tracks an upstream
  - Do your work on a **local** working branch
  - When the upstream changes:
    - `git checkout master`
    - `git pull --ff-only`
    - `git checkout working`
    - `git rebase master`
  - Your working branch is now up-to-date.

Remotes

- **Pushing** is sending your commits to a **remote**, most often the **origin**
- When it's time to push:
  - `git checkout master`
  - *update master using git pull*
  - `git merge working --ff-only`
  - `git push origin master`
- This will push your changes to the `origin` on the `master` branch

Remotes

- Sometimes you don't have push access to the remote.
  - Often the case for OSS projects
  - You'll need to send patch files instead
  - Git can do this for you!
  - `git format-patch HEAD^^^^^`
    - Generate a patch file for the most recent five commits



Remotes

- Git will generate a patch file for each commit.
  - It can also generate a cover letter, with stats, to describe the patch series.
    - `git format-patch --cover-letter HEAD^^^^`
    - Make sure to edit the cover letter before sending
  - To send a patch series, use:
    - `git send-email`

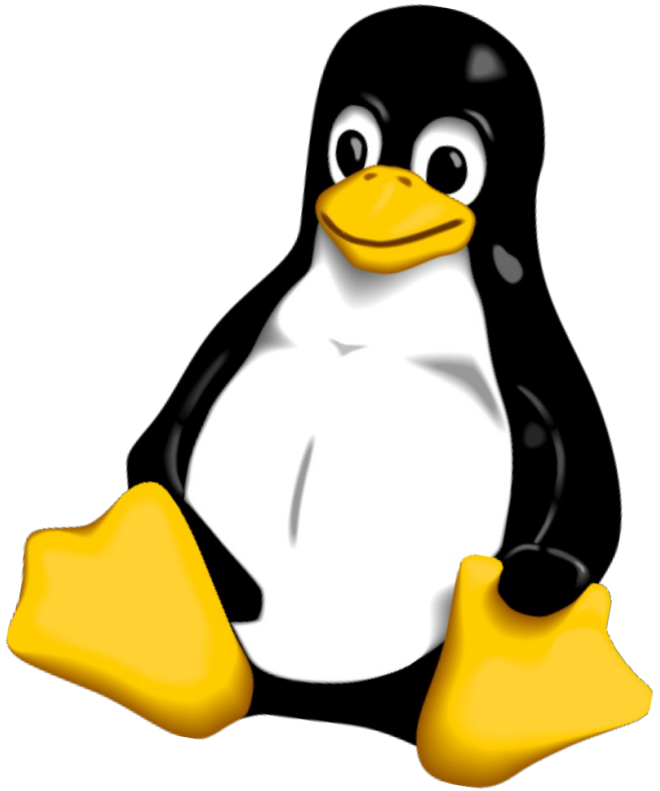
Remotes

- For submitting a patch upstream, see Greg Kroah-Hartman's excellent talk:
  - *Write and Submit your first Linux Kernel Patch*
  - https://www.youtube.com/watch?v=LLBrBBImJt4
    - *Watch it a couple times*

Remotes

- To accept a set of patch files into your repository, use `git am`
  - `git am *.patch`
- This will append the set of patches into your commit history on your current branch.
- `git am` preserves the **author names** present in the patches

Reflog



Reflog

- Git keeps track of when branches change.
  - Commits, pulls, merges, and rebases
- This information is stored in the **reflog**
- Each change gets an entry into the reflog, and the repository can be **reset** to that state.
- The reflog is stored locally
  - It is not pushed

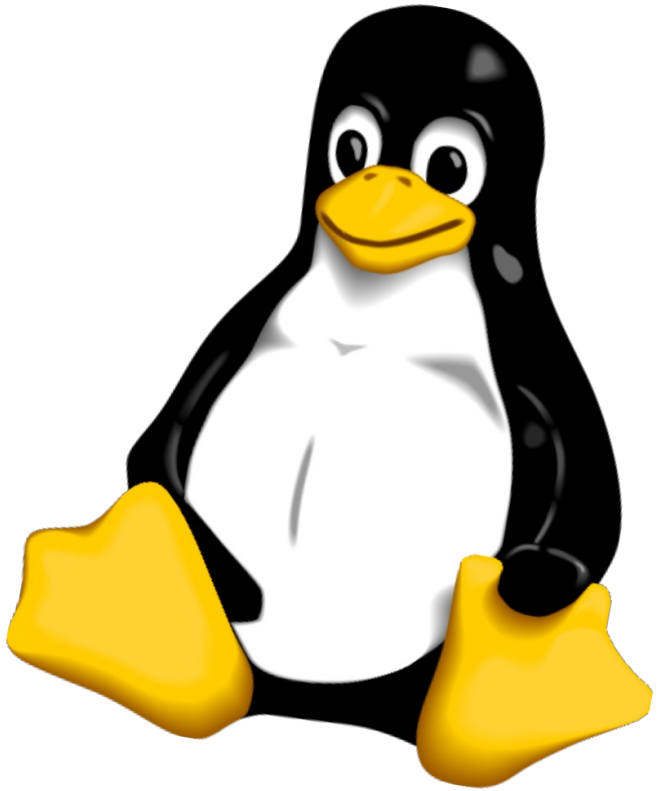
Reflog

- To see the reflog, run:
  - `git reflog`
- This shows all the entries and their sha1
  - This sha1 is different than a commit sha1
- To reset the repo to a previous state:
  - `git reset <sha1>`

Reflog

- The reflog is great if you mess up a rebase or pull.
  - Simply reset to a known good state
  - It can be tricky to find the last good state
- The reflog is not guaranteed to exist forever.
  - It can be garbage-collected with `git gc`

Bisect



Bisect

- For large projects, a few thousand commits might go by before noticing a bug.
  - How do you determine where the bug was introduced?
- Git provides a mechanism called **bisect**, which will perform a binary search between known good and known bad.

Bisect

- To start bisect, give it a known **good** revision and a **bad** revision, and tell it to **start**.
- Git will then give you the revision which is centered between them.
- Build, run, and test this revision
- Tell git if it's good or bad
- Git will continue a **binary search** based on the answer, giving you the next commit to try.

Bisect

- Since bisect uses a binary search, it will allow you to find the error in $O(\log_2 n)$ time, where n is the number of commits.
  - Search 1000 commits in ~10 steps
  - Very efficient!

Bisect

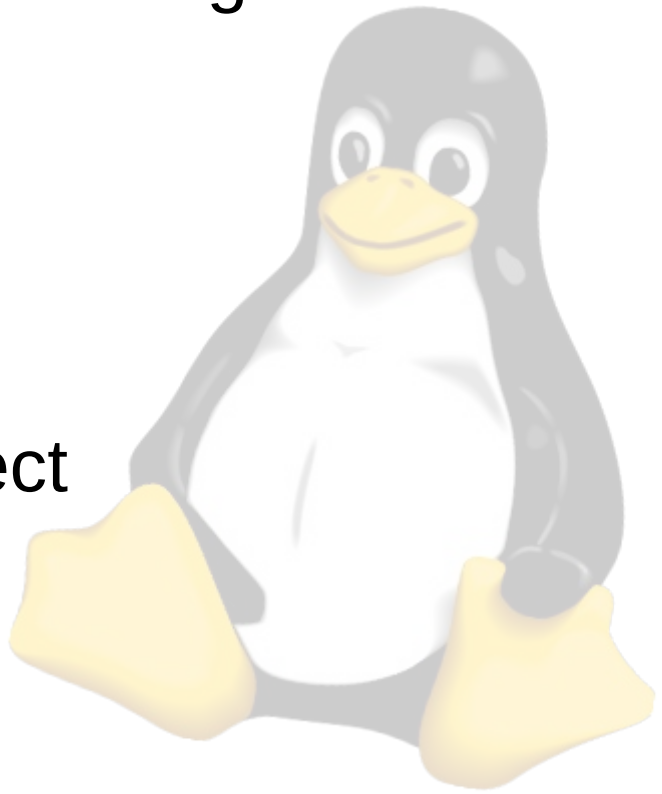
- Start the bisect
  - `git bisect start`
- Give git a good and bad revision
  - `git bisect good` <*revision*>
  - `git bisect bad` <*revision*>
- Git will then start the bisection
  - Git gives you the center commit between the good and bad revisions

Bisect

- Test the revision. If it's good, run:
  - `git bisect good`
- If it's bad, run:
  - `git bisect bad`
- Git will give you another commit to test
  - Goto the top of this page

Bisect

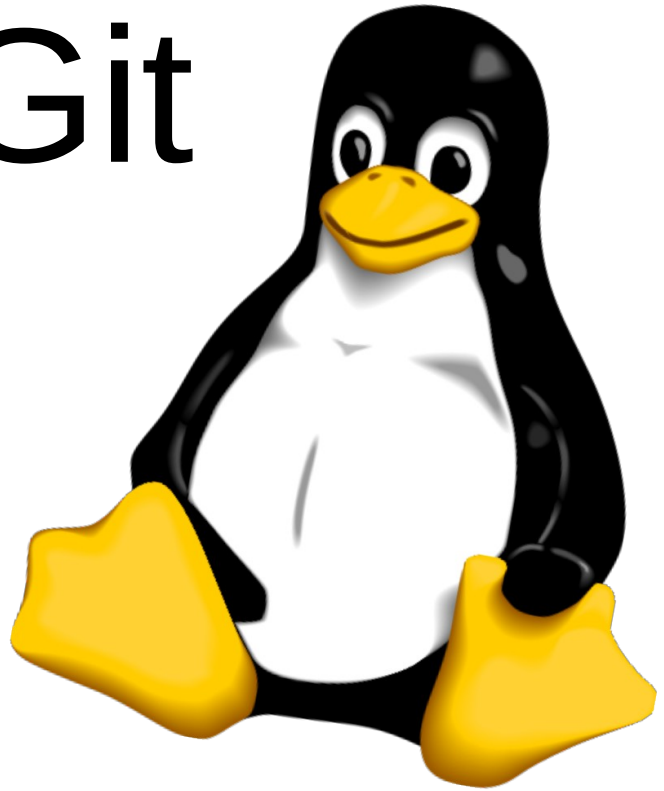
- Eventually git will tell you which is the first bad commit.
  - This is the one which introduced the bug
- When done, run:
  - `git bisect reset`
- It’s easier than it looks.
  - The first try will work as you expect



Bisect

- During a bisect, Git can potentially give you **any revision** in the history to test.
- This is why it's important that every commit build, run, and pass tests.
  - Imagine bisect giving you the commit before an “oops, forgot to add this file”
    - It would be much less fun.
  - This is also why it's important that history be **linear**, and have older commits not dependent on newer.

Teamwork in Git



SCALE 16x

Teams

- Git provides the tools to enable large, distributed teams of developers to work together.
  - Rebase
  - Cheap, local branching
  - Multiple remotes

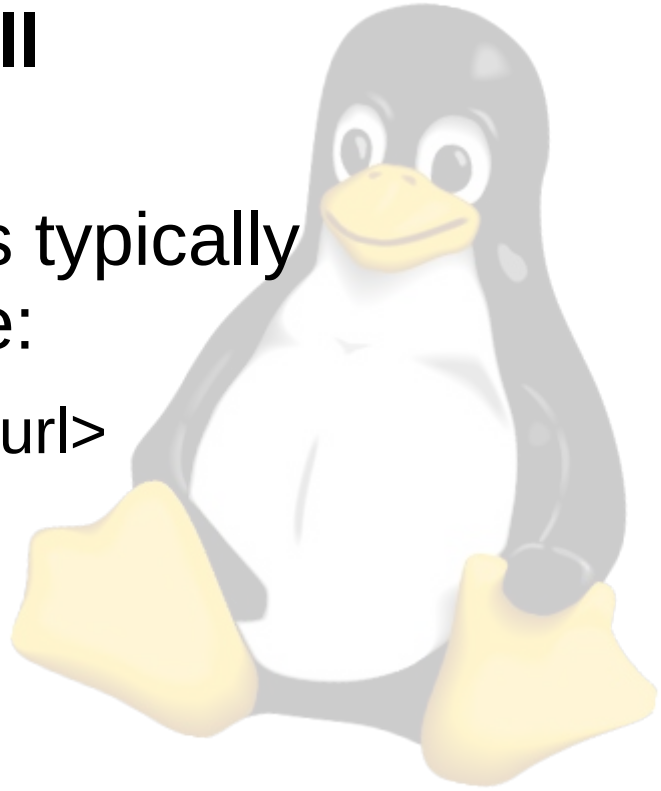
➢ *The Linux kernel is likely the largest software team in the world.*

Teams

- The typical method:
  - Each developer has their own online git repository.
  - Developers work on code and push it to their own repositories.
  - When it comes time to submit code to the maintainer for integration, the developer uses **interactive rebase** to **linearize** and **clean up** the commits.
    - Typically this will happen on a new branch.

Teams

- The typical method (cont'd):
  - Once the new branch has been rebased and is clean, the developer pushes the branch to their own public repository and sends a **pull request** to the maintainer.
  - Unlike on github, a pull request is typically an email that says something like:
    - Hi <maintainer>, please pull from <url>
  - The maintainer can then pull this branch into their local repo.

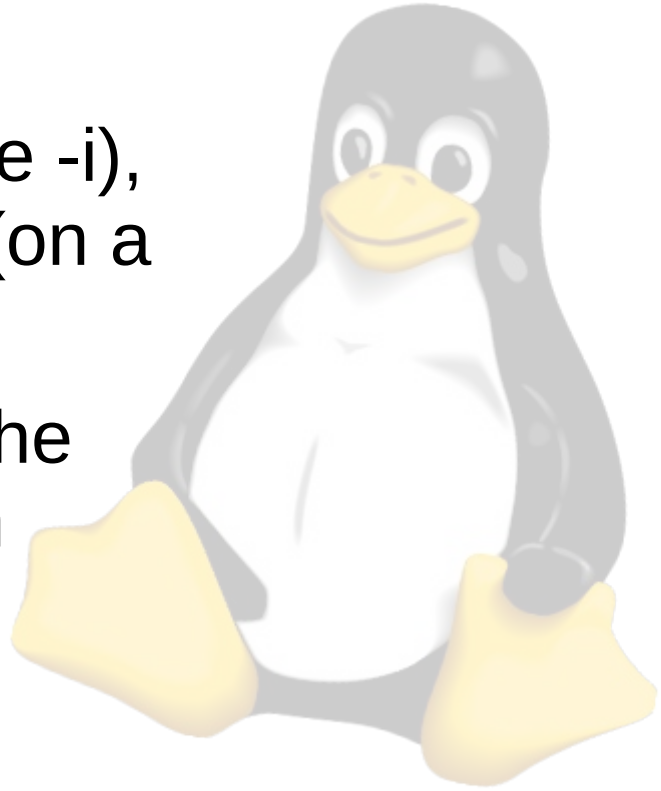


Teams

- The typical method (cont'd):
  - The maintainer can then look at the changes, in log format, using a range:
    - `git log master..pull_request_branch`
  - Even if the developer's branch is based on a revision older than what's on the maintainer's master it still works.
    - This case is likely!
    - Git will recognize the common history and show only the developer's commits

Teams

- The typical method (cont'd):
  - The maintainer will then provide feedback on the commits.
  - The developer can integrate the suggested changes (using rebase -i), and submit another pull request (on a different branch).
  - The maintainer can then re-pull the branch, review, and merge it with their own master.



Teams

- What are the advantages?
  - Code review is possible
    - Code can be reviewed at the commit level **before** commits are actually in the upstream master.
    - Commits can be **modified** by the developer an unlimited number of times before merging.
    - Code quality is improved!
  - The maintainer has complete control of the repository.
    - No surprise commits

Resources

- *Pro Git* book, available online:
  - https://git-scm.com/book/en/v2
- *git ready* website:
  - http://gitready.com/
- Man pages (use a hyphen):
  - `git-log(1)`
  - `git-clone(1)`
  - etc

SOFTIRON

Alan Ott
alan@softiron.com
www.softiron.com
+1 407-222-6975 (GMT -5)