GIT
THE BASICS

Important Notice : Courseware - Legal

This courseware is both the product of the author and of freely available opensource and/or public domain materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of the **courseware PDFs are considered proprietary** and thus cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - **source code** and binaries (where applicable) - that form part of this courseware, and that are made available to the participant, are released under the terms of the permissive **MIT license**.
Under the terms of the MIT License, you can certainly use the source code provided here; you must just attribute the original source (author of this courseware and/or other copyright/trademark holders).

VERY IMPORTANT :: Before using this source(s) in your project(s), you *MUST* check with your organization's legal staff that it is appropriate to do so.

The courseware PDFs are *not* under the MIT License, they are to be kept confidential, non-distributable without consent, for your private internal use only.

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant VM are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind, and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

2000-2022 Kaiwan N Billimoria
kaiwanTECH, Bangalore, India.

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: http://bit.ly/ktcorp

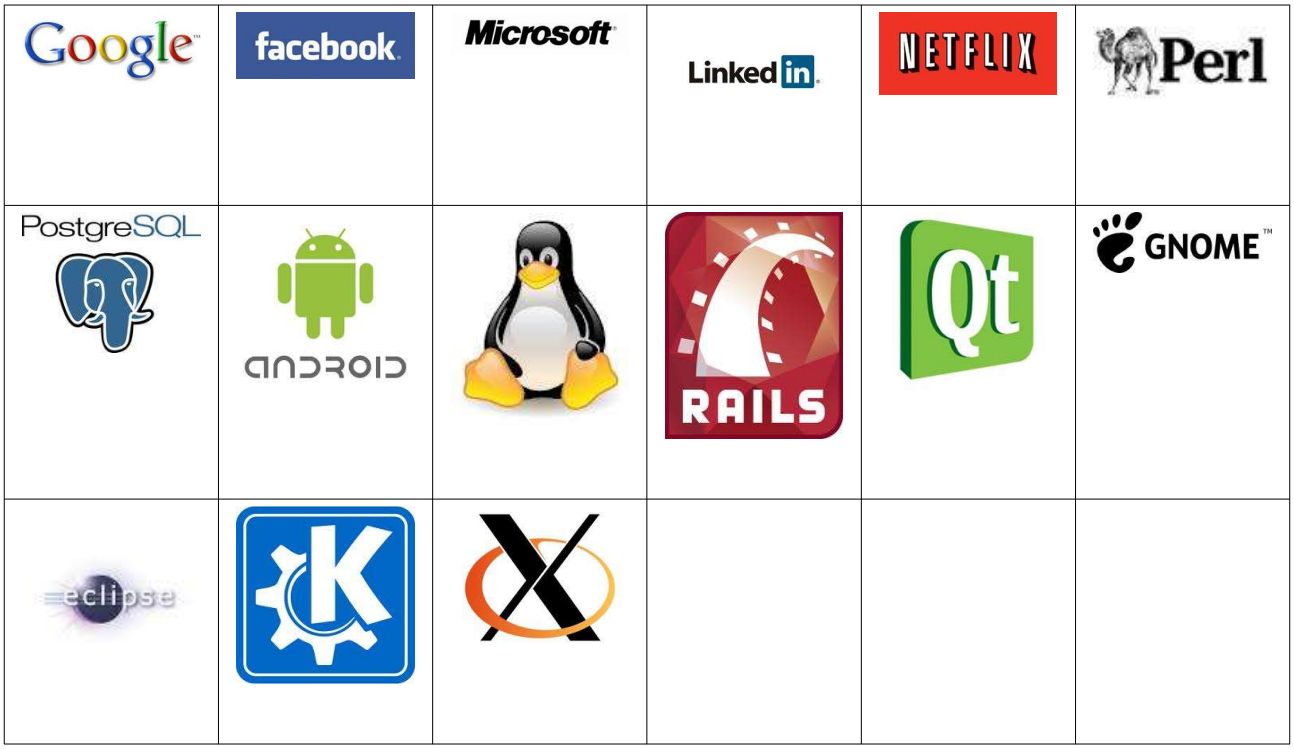
Git – A Very Brief Summary

Git – the SCM system created by Linus Torvalds – is of course the SCM used to track and manage the (enormous) Linux kernel project.

"Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.
Git is easy to learn and has a tiny footprint with lightning fast performance. ..."

Git has become very popular:

Companies & Projects Using Git



<< Above sourced from git-scm.org >>

How does one use git?
Some useful online resource to get started quickly with Git:

Firstly, A Gentle Introduction to Version Control

https://lab.github.com/

Oh Shit, Git!?! *- really helpful!*

Flight Rules for Git – what to do when things go wrong

Resources to learn Git

Become a Git pro in just one blog. A thorough guide to Git architecture and command line interface

Try Git in your browser – interactive tutorial!

An app on Google Play -Git Guide, by Raj Vaibhav.
Simple and quite clearly explains git workflow.

Most commonly used git tips and tricks on github

The "Git Magic" book, by Ben Lynn (PDF)

Aha! Moments when learning Git

How not to be afraid of Git anymore

"Every time you `add` a file, Git adds a new object to the object store. **This is exactly why you can't deal with very large files in Git**— it stores the entire file each time you `add` changes, not the diff (contrary to popular belief)."

Think Like a Git [read once you're familiar with the basics]

Getting The Most Out Of Git, Feb '21

Getting Git Right, Atlassian.com [bitbucket.org]

Beginning Git and Github for Linux Users – on Linux.com

The "official" git website

Get Started with Git

git Documentation

Everyday git with 20 Commands or so

Git Tips from kernelnewbies

Some Notes on Git

Master Version Control with git

GitHub Help

Google Git Cheatsheet

Git Ready – learn git one commit at a time

Git Foundations (the how and why)

10 Must Have Resources For Git Learners

Use gitk to understand git

A Very Brief Incomplete Guide to using Git

This assumes you've downloaded and installed git.

First time / just once:

```
git config --global user.name "Joey Joejoe"
git config --global user.email "joey@joejoe.com"
```

New project:

```
cd folder
git init
```

Tip:
Want a GUI git client?
Have a look at available gui clients for various platforms here!

Try using *gitk*; have it invoke *git-gui* (File/Start git gui)

Trivial git workflow

[
- create/edit your project files

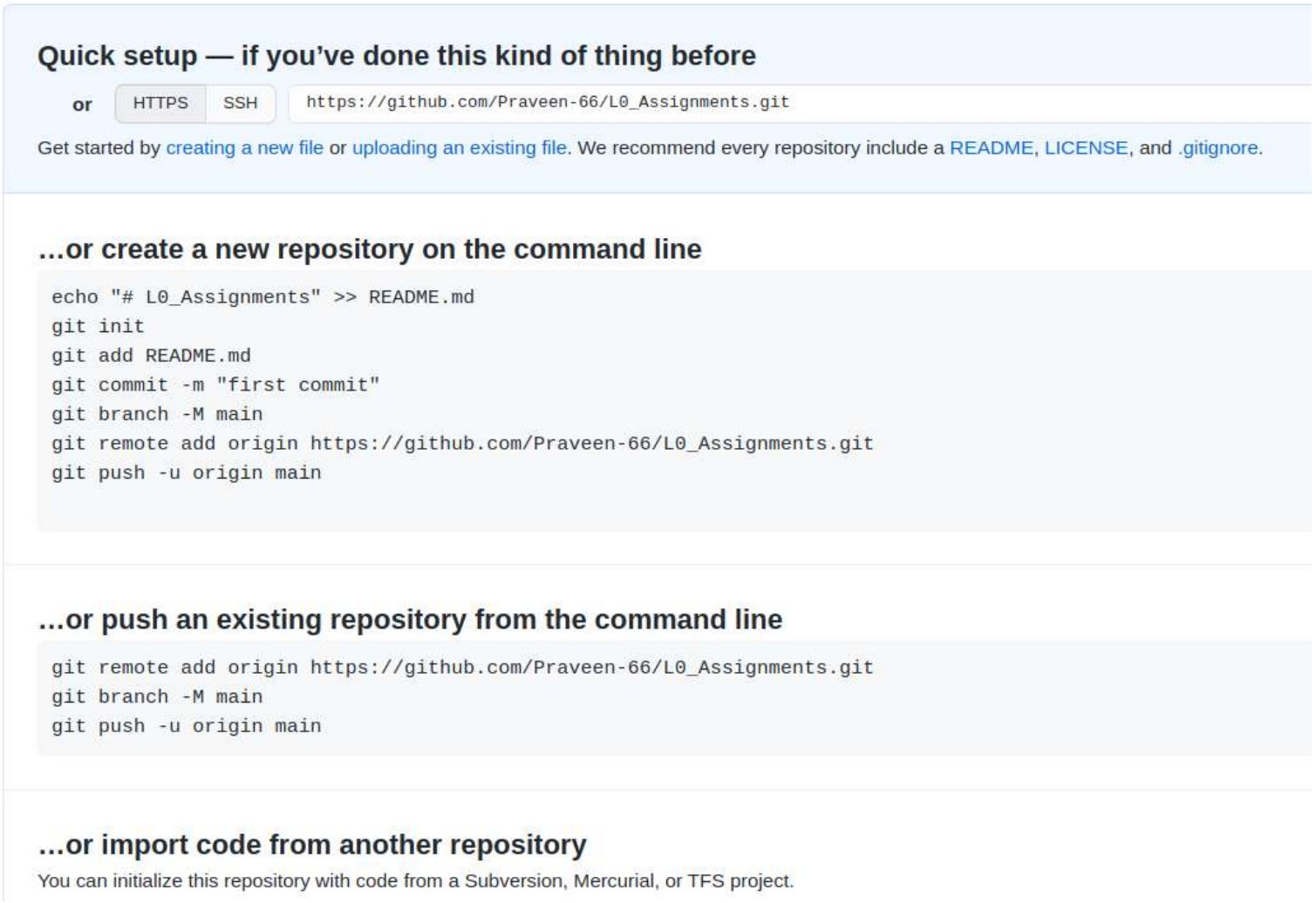
- add modified/new files by
git add
This is known as "staging" your work; implies that git knows about it, but changes are not committed.

- commit files (local only); should do this often..
git commit
(will ask for a commit comment/summary here)

- See your status, changes, etc:
git status
git log
]

*[Tip: **Please see http://githowto.com** - "a guided tour that walks through the fundamentals of Git, inspired by the premise that to know a thing is to do it."]*

From GitHub :: (screenshot):



[
Source: "Git Magic", by Ben Lynn

About to attempt something drastic? Before you do, take a snapshot of all files in the current directory with:

```
$ git init
$ git add .
$ git commit -m "My first backup"
```

Now if your new edits go awry, restore the pristine version:

```
$ git reset --hard
```

To save the state again:
```
$ git commit -a -m "Another backup"
```

]

Git: Sample Session 1

Imagine we've begun work on an existing project and decided we'd like to setup git as the SCM.

```
$ cd git_fifo/
$ ls
bkp/  fifo_common.h  Makefile  reader_fifo*  reader_fifo.c  restart_lib.h
test_fifo.sh*  writer_fifo*  writer_fifo.c
$
```

Here's the project's “root” soure folder, where we'd like to setup git

One-time username/email configuration; all your commits will appear under this name/email id..

```
$ git config user.name "Joey Joejoe"
$ git config user.email "joey@joejoe.com"
```

First initialize a git repo

```
$ git init
Initialized empty Git repository in </...>/git_fifo/.git/
$
```

Add all files into the repo

```
$ git add .
```

Hey, don't add anything that can be re-generated; typically, this includes binary executables, backup files and folders, etc. So lets tell git to ignore them with 'git rm'

```
$ git rm reader_fifo writer_fifo
error: 'reader_fifo' has changes staged in the index
(use --cached to keep the file, or -f to force removal)
error: 'writer_fifo' has changes staged in the index
(use --cached to keep the file, or -f to force removal)
$
$ git rm --cached reader_fifo writer_fifo
```

```
rm 'reader_fifo'
rm 'writer_fifo'
$
$ git rm --cached -r bkp/
rm 'bkp/fifo_common.h'
rm 'bkp/reader_fifo.c'
rm 'bkp/restart_lib.h'
rm 'bkp/writer_fifo.c'
$
```

How're we doing?

```
$ git log
fatal: bad default revision 'HEAD'
$
```

Oops, we haven't even committed once. Commit yourself now!! :-)

```
$ git commit -m "Initial commit"
[master (root-commit) 67dc822] Initial commit
 6 files changed, 561 insertions(+)
 create mode 100644 Makefile
 create mode 100644 fifo_common.h
 create mode 100644 reader_fifo.c
 create mode 100644 restart_lib.h
 create mode 100755 test_fifo.sh
 create mode 100644 writer_fifo.c
$
$ git log
commit 67dc8228266711203496 2419f4ccd1918d46398f
Author: Joey Joejoe <joey@joejoe.com>
Date:   Wed Mar 13 20:03:25 2013 +0530

    Initial commit
$
```

Okay!

Do commit often; as a thumb-rule, any change requiring more than one line to describe it requires a commit!

[Repeat indefinitely, the 3 steps:
edit <file> / git add <file> / git commit]

- see current status
git status

- see the diff
git diff

- see log
git log

- push (publish) changes upstream
git push

- pull (sync) local repo with remote
git pull

Pushing to a remote repository

We'll use the (famous!) http://github.com as an example of a remote repo.
We'll assume you've setup an account there (can be a free one); we expect you know the username and password, of course.

Lets say, for example :

```
Remote git url: https://github.com/
username: userme
password: @mypass@
```

The “project” or repo name: **misc**
Information on your remote repo(s):

```
$ git remote -v
$
```

Nothing there? Add it..

```
$ git remote add misc https://github.com/userme/misc.git
$

$ git remote -v
misc  https://github.com/userme/misc.git (fetch)
misc  https://github.com/userme/misc.git (push)
$
```

We'll also assume you've done some work and have a file or two ready to push upstream.

```
<< Since Git 2.0, Git defaults to the more conservative 'simple'
behavior, which only pushes the current branch to the corresponding
remote branch that 'git pull' uses to update the current branch. >>

$ git config --global push.default simple
$
```

Okay. Lets try and push!

```
$ git push --set-upstream misc master
Username for 'https://github.com': userme
Password for 'https://kaiwan@github.com': ...
To https://github.com/kaiwan/misc.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'https://github.com/kaiwan/misc.git'
```

```
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
$
```

Okay, so lets try and “pull” first:

```
Syntax:
git pull <git-url>
  <git-url> : https://github.com/<username>/<git-repo-name>.git
```

```
$ git pull https://github.com/userme/misc.git
From https://github.com/userme/misc
 * branch            HEAD       -> FETCH_HEAD
Merge made by the 'recursive' strategy.
 LICENSE   | 22 ++++++++++++++++++++++
 README.md |  2 ++
 2 files changed, 24 insertions(+)
 create mode 100644 LICENSE
 create mode 100644 README.md
$
```

Good. *Now* lets retry the “push”:

```
$ git push --set-upstream misc master
Username for 'https://github.com': userme
Password for 'https://kaiwan@github.com': ...
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 1.50 KiB | 0 bytes/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To https://github.com/userme/misc.git
   9b2dedf..be464f5  master -> master
Branch master set up to track remote branch master from misc.
$
```

Success!

SIDEBAR | Authentication Problems with 'git push'

Both from the cmd-line as well as git-gui I get the failure message:

"fatal: remote error: Invalid username/password.
You may need to use your generated googlecode.com password; see
https://code.google.com/hosting/settings"

The SOLUTION is here!

http://stackoverflow.com/questions/9344566/authenticating-sourcetree-with-a-google-code-project

"The problem is that by default Google tells (in .git/config) you to use a URL which includes the

"USER@" portion. SourceTree sees this and assumes that the Username is the user in the URL and that there is no password. To fix the problem, simply remove the "USER@" portion in the URL. Then when you try pushing your changes, SourceTree will prompt for a username/password.

For example, instead of:

https://USER@code.google.com/p/PROJECT/

use:

https://code.google.com/p/PROJECT/

For more info, see: https://answers.atlassian.com/questions/36585/entering-in-a-password-in-sourcetree "

Backing out of a change

We make a change in a source file; here, as an example, we change an '#if 0' to an '#if 1' (effectively compiling in that code).

```
$ git diff
diff --git a/reader_fifo.c b/reader_fifo.c
index 6c3f24d..20e4c0b 100644
--- a/reader_fifo.c
+++ b/reader_fifo.c
@@ -45,7 +45,7 @@ int main(int argc, char **argv)
        if ((argc == 4) && (atoi(argv[3]) == 1))
                verbose = 1;

-#if 0
+#if 1
        unlink(FIFO_FILE);
        if (mkfifo(FIFO_FILE, FIFO_FILE_MODE) < 0)
                err_exit(argv[0], "mkfifo failed", FAILURE);
$
```

We commit the change; notice that this changes the HEAD; also notice that every commit has an associated SHA1_HASH associated with it (here it's the number 826ad70fa217d3e06baec10b0946c0750dfc995a !).

When specifying the commit by hask key, just using the first few numerals is (usually) sufficient to guarantee uniqueness.

```
$ git commit -a -m "create the FIFO in the reader as well"
[master 826ad70] create the FIFO in the reader as well
 1 file changed, 1 insertion(+), 1 deletion(-)
$
$ git log
commit 826ad70fa217d3e06baec10b0946c0750dfc995a  <-- the SHA1_HASH for this commit
Author: Joey Joejoe <joey@joejoe.com>
Date:   Thu Mar 14 08:42:48 2013 +0530

    create the FIFO in the reader as well
```

```
commit 992fce30d53f93dc11fa12dc17aa76862ebfb7ca
Author: Joey Joejoe <joey@joejoe.com>
Date:   Thu Mar 14 08:41:45 2013 +0530

    indent done on src files

commit 4712d45b53e489273a01cf910a100e56ac459ec3
Author: Joey Joejoe <joey@joejoe.com>
Date:   Thu Mar 14 08:34:27 2013 +0530

    Makefile changed to reflect better Makefile practice

commit 67dc82282667112034962419f4ccd1918d46398f
Author: Joey Joejoe <joey@joejoe.com>
Date:   Wed Mar 13 20:03:25 2013 +0530

    Initial commit
$
```

Now we do our testing of the project and what do we realize?
That the change we made ('#if 0' to an '#if 1') was a mistake!
How do we get rid of that particular change??

Easy- one way is to tell git to "revert" that commit.

$ git revert 826a *<-- it's enough to supply just the first few numerals of the commit's SHA1_HASH key*

<< git launches vi so that the 'Revert' commit message can be edited if required ... >>

```
[master 5ee6e9b] Revert "create the FIFO in the reader as well"
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git diff
```

<< See? Now there's no diff with the (original) HEAD!
(Note that the source file was auto-updated of course) >>

```
$ git log
commit 5ee6e9bb07705f1afcc20b7b3cd54631c3b17a9e      <-- current HEAD
Author: Joey Joejoe <joey@joejoe.com>
Date:   Thu Mar 14 08:44:29 2013 +0530

    Revert "create the FIFO in the reader as well"

    This reverts commit 826ad70fa217d3e06baec10b0946c0750dfc995a.

commit 826ad70fa217d3e06baec10b0946c0750dfc995a
Author: Joey Joejoe <joey@joejoe.com>
Date:   Thu Mar 14 08:42:48 2013 +0530

    create the FIFO in the reader as well

--snip--

$
```

Now it's fine!

<< Also:
Look up section 2.2 “Advanced Undo/Redo” in the “Git Magic” book.
>>

Miscellaneous

Looking up the git log

A very good sample format for the “git log” command:

```
git [--no-pager] log --pretty=format:"%h %ad | %s%d [%an]" --graph --
date=short
```

Eg. output from the Linux kernel git tree:

```
$ git --no-pager log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short
*   c8d17b4 2015-06-12 | Merge git://git.kernel.org/pub/scm/linux/kernel/git/davem/net (HEAD,
origin/master, origin/HEAD, master) [Linus Torvalds]
|\
| * b07d496 2015-06-13 | Doc: networking: Fix URL for wiki.wireshark.org in udplite.txt
[Masanari Iida]
| * ae36806 2015-06-11 | sctp: allow authenticating DATA chunks that are bundled with
COOKIE_ECHO [Marcelo Ricardo Leitner]
| * fb05e7a 2015-06-11 | net: don't wait for order-3 page allocation [Shaohua Li]
| * 0fae3bf 2015-06-11 | mpls: handle device renames for per-device sysctls [Robert Shearman]
| * 58c98be 2015-06-11 | net: igb: fix the start time for periodic output signals [Richard
Cochran]
| * 8b13b4e 2015-06-11 | enic: fix memory leak in rq_clean [Govindarajulu Varadarajan]
| * 19b596b 2015-06-11 | enic: check return value for stat dump [Govindarajulu Varadarajan]
| * 6286e82 2015-06-11 | enic: unlock napi busy poll before unmasking intr [Govindarajulu
Varadarajan]
| * 5d75361 2015-06-10 | net, swap: Remove a warning and clarify why sk_mem_reclaim is required
when deactivating swap [Mel Gorman]
...
...
```

Remove the --graph to get rid of the “graph tree” ASCII art.

Very useful!

Look for a particular feature commit, for example, GRO (Generic Receive Offload – a hardware/OS offload technique used for better network performance):

```
$ git --no-pager log --pretty=format:"%h %ad | %s%d [%an]" --date=short |egrep
--color=auto  "net.*Generic Receive| +GRO "  << the regex “ +GRO ” looks to match >= 1
spaces followed by 'GRO ' >>
45c9b3c 2015-03-23 | bgmac: implement GRO and use build_skb [Felix Fietkau]
1037ebb 2015-03-02 | net/mlx4_en: Disbale GRO for incoming loopback/selftest packets
[Ido Shamay]
6db93ea 2015-02-10 | udp: Set SKB_GSO_UDP_TUNNEL* in UDP GRO path [Tom Herbert]
26c4f7d 2015-02-10 | net: Fix remcsum in GRO path to not change packet [Tom Herbert]
a4c9ea5 2014-12-30 | geneve: Add Geneve GRO support [Joe Stringer]
```

```
9b174d8 2014-12-30 | net: Add Transparent Ethernet Bridging GRO support. [Jesse Gross]
...
...
e1c096e 2009-01-06 | vlan: Add GRO interfaces [Herbert Xu]
89c88b1 2008-12-15 | e1000e: Add GRO support [Herbert Xu]
bf296b1 2008-12-15 | tcp: Add GRO support [Herbert Xu]
73cc19f 2008-12-15 | ipv4: Add GRO infrastructure [Herbert Xu]
d565b0a 2008-12-15 | net: Add Generic Receive Offload infrastructure [Herbert Xu]
$
```

See the commit details

```
$ git show d565b0a
commit d565b0a1a9b6ee7dff46e1f68b26b526ac11ae50
Author: Herbert Xu <herbert@gondor.apana.org.au>
Date:   Mon Dec 15 23:38:52 2008 -0800

    net: Add Generic Receive Offload infrastructure

    This patch adds the top-level GRO (Generic Receive Offload) infrastructure.
    This is pretty similar to LRO except that this is protocol-independent.
 [...]
```

Which kernel version was GRO first committed to?

```
$ git describe --contains d565b0a
v2.6.29-rc1~581^2~181
$
```

To see the difference between two commits

git diff <commit1> <commit2>

('HEAD' is the top of the tree history).

Eg.
Made a change in malloc's argument in a source file and committed the same. Look up the change with:

```
$ git log
commit f3d68b5e5f8fb25d621d6b51e8542e82796cdab7            <-- current HEAD
Author: Joey Joejoe <joey@joejoe.com>
Date:   Thu Mar 14 09:02:49 2013 +0530

    malloc sz

commit 5ee6e9bb07705f1afcc20b7b3cd54631c3b17a9e
Author: Joey Joejoe <joey@joejoe.com>
Date:   Thu Mar 14 08:44:29 2013 +0530

    Revert "create the FIFO in the reader as well"

    This reverts commit 826ad70fa217d3e06baec10b0946c0750dfc995a.

--snip--

$ git diff HEAD 5ee6
diff --git a/reader_fifo.c b/reader_fifo.c
index fd29ce0..6c3f24d 100644
--- a/reader_fifo.c
+++ b/reader_fifo.c
@@ -51,7 +51,7 @@ int main(int argc, char **argv)
                err_exit(argv[0], "mkfifo failed", FAILURE);
 #endif

-       buf = malloc(sz-10000);
+       buf = malloc(sz);
        if (!buf)
                err_exit(argv[0], "malloc failed", FAILURE);
        memset(buf, 'x', sz);
$

$ git revert f3d6
[master 2d1cec5] Revert "malloc sz"
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git diff HEAD 5ee6
$
...
$ git whatchanged --since="10 minutes ago"
commit 2d1cec592e0ea57281a570ba637f4268085cf393
Author: Joey Joejoe <joey@joejoe.com>
Date:   Thu Mar 14 09:07:04 2013 +0530

    Revert "malloc sz"

    This reverts commit f3d68b5e5f8fb25d621d6b51e8542e82796cdab7.

:100644 100644 fd29ce0... 6c3f24d... M  reader_fifo.c
```

```
$
```

<<

Source: "Git Magic", by Ben Lynn

2.7 What Have I Done?

Find out what changes you've made since the last commit with:

```
$ git diff
```

Or since yesterday:
```
$ git diff "@{yesterday}"
```

Or between a particular version and 2 versions ago:
```
$ git diff SHA1_HASH "master~2"
```

<<
Or, what changed last with:
```
$ git diff master~1 master
```
>>

In each case the output is a patch that can be applied with git apply. Try also:

```
$ git whatchanged --since="2 weeks ago"
```

Often I'll browse history with qgit (http://sourceforge.net/projects/qgit) instead, due to its slick photogenic interface, or tig (http://jonas.nitro.dk/tig/), a text-mode interface that works well over slow connections. Alternatively, install a web server, run git instaweb and fire up any web browser.

>>

<<

Humour ! When Artificial Intelligence Meets git [Comic]
Source
>>

Source: "Git Magic", by Ben Lynn

Branching and Merging

4.2 Dirty Work

Say you're working on some feature, and for some reason, you need to go back to an old version and temporarily put in a few print statements to see how something works. Then:

```
$ git commit -a
$ git checkout SHA1_HASH
```

<< OR

```
$ git branch dirty
```

also does the trick; we now have a new "branch" called "dirty" !

```
$ git branch
* master
  dirty
$
```

Switch to the *dirty* branch:

```
$ git branch dirty
$ git branch
master
* dirty
$
```

>>

Now you can add ugly temporary code all over the place. You can even commit these changes. When you're done,

```
$ git checkout master
```

<< the meaning of the 'checkout' command in git is very different from what it means in subversion (svn). In git, checkout implies switching to another branch & working within that branch. >>

to return to your original work. Observe that any uncommitted changes are carried over.
What if you wanted to save the temporary changes after all? Easy:

```
$ git checkout -b dirty
```

and commit before switching back to the master branch. Whenever you want to return to the dirty changes, simply type

```
$ git checkout dirty
```

We touched upon this command in an earlier chapter, when discussing loading old states. At last we can tell the whole story: the files change to the requested state, but we must leave the master branch. Any commits made from now on take your files down a different road, which can be named later.

In other words, after checking out an old state, Git automatically puts you in a new, unnamed branch, which can be named and saved with git checkout -b.

4.3. Quick Fixes

You're in the middle of something when you are told to drop everything and fix a newly discovered bug:

```
$ git commit -a
$ git checkout -b fixes SHA1_HASH
```

Then once you've fixed the bug:

```
$ git commit -a -m "Bug fixed"
$ git push            # to the central repository
```

```
$ git checkout master
```

and resume work on your original task.

4.4. Uninterrupted Workflow

Often in hardware projects, the second step of a plan must wait for the first step to be completed before it can begin. A car undergoing repairs might sit idly in a garage until a particular part arrives from the factory. A prototype might wait for a chip to be fabricated before construction can continue.

Software projects can be similar. The second part of a new feature may have to wait until the first part has been released and tested. Some projects require your code to be reviewed before accepting it, so you might wait until the first part is approved before starting the second part.

In Git, thanks to painless branching and merging, we can bend the rules and work on Part II before Part I is officially ready. Suppose you have committed Part I and sent it for review. Let's say you're in the master branch. Then branch off:

```
$ git checkout -b part2
```

Next, work on Part II, committing your changes along the way. To err is human, and often you'll want to go back and fix something in Part I. If you're lucky, or very good, you can skip these lines.

```
$ git checkout master                # Go back to Part I.
<< edit files >>                                          # Fix Part I.
<< git commit -a -m ... >>
$ git checkout part2                 # Go (switch) back to Part II.
$ git merge master                   # Merge in those fixes.
```

Eventually, Part I is approved:

```
$ git checkout master       # Go (switch) back to Part I.
$ some_command              # Some command you're supposed to run when the
                            # current working directory is officially ready.


$ git merge part2           # Merge in Part II.
$ git branch -d part2       # delete the branch when done
```

Now you're in the master branch again, with Part II in the working directory.

It's easy to extend this trick for any number of parts. It's also easy to branch off retroactively: suppose you belatedly realize you should have created a branch several commits ago. Then type:

```
$ git branch -m master part2      # Rename "master" branch to "part2".
$ git checkout SHA1 -b master     # The commit representing Part I.
```

The master branch now contains just Part I, and the part2 branch contains the rest.

Git Sample Session – Branching & Merging

```
$ git branch
* master
$
```

<< Not happy with the current state; want to try something different...
Branch off!
>>

```
$ git branch poll4fifo      <-- create a branch called “poll4fifo”
$ git branch                <-- list branches
* master
  poll4fifo
$
$ git checkout poll4fifo         <-- switch to the 'poll4fifo' branch
Switched to branch 'poll4fifo'
$ git branch
  master
* poll4fifo
$
```

```
...
<< edit files, do your stuff, then commit >>
...
```

```
$ git commit -a
[poll4fifo d0a2691] Poll for the presence of the FIFO before attempting to open it.
This way, we don't race with the writer that creates the FIFO object..
 1 file changed, 9 insertions(+), 12 deletions(-)
$
```

```
$ git checkout master        <-- switch to the 'master' branch
Switched to branch 'master'
$ git branch
* master
  poll4fifo
$
```

<<
Now we're back in the 'master' branch.
If you WANT to keep the changes you've just made while in the newly created branch, use the

```
git merge <new-branch-name>
```

command.

If you do NOT want to keep the changes you just made while in the newly created branch, do Not merge in the new branch.
>>

```
$ git merge poll4fifo        <-- merge 'poll4fifo' branch into master
```

```
Updating 2727177..d0a2691
Fast-forward
 reader_fifo.c |   21 +++++++++------------
 1 file changed, 9 insertions(+), 12 deletions(-)
$
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#      .reader_fifo.c.swp
#      reader_fifo
#      reader_fifo.o
#      tags
#      writer_fifo
#      writer_fifo.o
nothing added to commit but untracked files present (use "git add" to track)
$
$ git log
commit d0a2691eb1382161f1072651d6ea701e5d6a80aa
Author: Joey Joejoe <joey@joejoe.com>
Date:   Thu Mar 14 11:49:26 2013 +0530

    Poll for the presence of the FIFO before attempting to open it.
     This way, we don't race with the writer that creates the FIFO object..

commit 27271779a8b2d7ea3e56582ffa1fd00a6a2f0322
Author: Joey Joejoe <joey@joejoe.com>
Date:   Thu Mar 14 09:53:12 2013 +0530

    have the reader retry the FIFO open thrice before reporting failure (due to a poss
race w/ the writer not having cr8 the FIFO yet). RELOOK?

--snip--

$

<< Done. >>
```

Source File History

Method 1

Viewing the detailed history of a source file (line-by-line) is easily achieved with the 'git blame' command.

Eg.

```
$ git blame shm_read.c
...
^9751d63 (Kaiwan Billimoria 2013-08-02 21:06:53 +0530  99)             if (verbose) {
0e407dd3 (Kaiwan Billimoria 2013-08-02 21:39:31 +0530 100)                     printf("%s
[%d] : loop #%3d buf read, %d bytes (from 0x%08x)\n",
0e407dd3 (Kaiwan Billimoria 2013-08-02 21:39:31 +0530 101)
argv[0], pid, i, sz, (unsigned int)shm_vaddr);
0e407dd3 (Kaiwan Billimoria 2013-08-02 21:39:31 +0530 102)
//printf(" :: \"%.*s\"\n", sz, buf);
^9751d63 (Kaiwan Billimoria 2013-08-02 21:06:53 +0530 103)             }
0e407dd3 (Kaiwan Billimoria 2013-08-02 21:39:31 +0530 104)             shm_vaddr += sz;
...
```

```
$
```

How does one figure the kernel version given only the patch (SHA1) id?

Ref: Given a git patch id, how to find out which kernel release contains it?

Short Ans:
Use '`git describe`' like so (on the local repo of course; it can take a while!):

```
$ git describe --contains 3aa551c9
v2.6.30-rc1~3^2~5
$
```

BTW, 3aa551c9 is the (first 8 digits) of the unique SHA1 id of the "Add support for threaded interrupt handlers - V3" by Thomas Gleixner !

Method 2

USEFUL! Get the complete commit history of a particular line(s) of a given file

Sometimes, we'd like to figure out the complete history of a full, or a range of lines, within a file. Git can do this with:

```
$ git log --pretty=short -u -L <start_line#>,<end_line#>:<filename>
```

Example:

```
$ git log --pretty=short -u -L 1668,1706:include/linux/fs.h
commit 723c038475b78edc9327eb952f95f9881cc9d79d
Author: Christoph Hellwig <hch@lst.de>

    fs: remove the never implemented aio_fsync file operation

diff --git a/include/linux/fs.h b/include/linux/fs.h
--- a/include/linux/fs.h
+++ b/include/linux/fs.h
@@ -1695,40 +1695,39 @@
 struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
[...]
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
-       int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
[...]
        int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t,
                        u64);
        ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,
                        u64);
 };

commit 6192269444ebfbfb42e23c7a6a93c76ffe4b5e51
```

```
Author: Al Viro <viro@zeniv.linux.org.uk>

    introduce a parallel variant of ->iterate()

diff --git a/include/linux/fs.h b/include/linux/fs.h
--- a/include/linux/fs.h
+++ b/include/linux/fs.h
@@ -1669,39 +1669,40 @@
 struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iterate) (struct file *, struct dir_context *);
+       int (*iterate_shared) (struct file *, struct dir_context *);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);

[...]

commit 54dbc15172375641ef03399e8f911d7165eb90fb
Author: Darrick J. Wong <darrick.wong@oracle.com>

    vfs: hoist the btrfs deduplication ioctl to the vfs

diff --git a/include/linux/fs.h b/include/linux/fs.h
--- a/include/linux/fs.h
+++ b/include/linux/fs.h
@@ -1600,37 +1600,39 @@
 struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
[...]
        ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
                        loff_t, size_t, unsigned int);
        int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t,
                        u64);
+       ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,
+                       u64);
 };

commit 04b38d601239b4d9be641b412cf4b7456a041c67
Author: Christoph Hellwig <hch@lst.de>

    vfs: pull btrfs clone API to vfs layer

diff --git a/include/linux/fs.h b/include/linux/fs.h
--- a/include/linux/fs.h
+++ b/include/linux/fs.h
@@ -1600,34 +1600,37 @@
 struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);

[...]

commit b19dd42faf413b4705d4adb38521e82d73fa4249    << fyi: in 2.6.36 >>
Author: Arnd Bergmann <arnd@arndb.de>
```

```
    bkl: Remove locked .ioctl file operation

diff --git a/include/linux/fs.h b/include/linux/fs.h
--- a/include/linux/fs.h
+++ b/include/linux/fs.h
@@ -1489,28 +1489,27 @@
 struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long,
loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
-       int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);

[...]

commit 1da177e4c3f41524e886b7f1b8a0c1fc7321cac2
Author: Linus Torvalds <torvalds@ppc970.osdl.org>

    Linux-2.6.12-rc2

diff --git a/include/linux/fs.h b/include/linux/fs.h
--- /dev/null
+++ b/include/linux/fs.h
@@ -0,0 +924,29 @@
+struct file_operations {
+       struct module *owner;
+       loff_t (*llseek) (struct file *, loff_t, int);
+       ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
+       ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
+       ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
+       ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
+       int (*readdir) (struct file *, void *, filldir_t);
+       unsigned int (*poll) (struct file *, struct poll_table_struct *);
+       int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
+       long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
+       long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
+       int (*mmap) (struct file *, struct vm_area_struct *);
+       int (*open) (struct inode *, struct file *);
+       int (*flush) (struct file *);
+       int (*release) (struct inode *, struct file *);
+       int (*fsync) (struct file *, struct dentry *, int datasync);
+       int (*aio_fsync) (struct kiocb *, int datasync);
+       int (*fasync) (int, struct file *, int);
+       int (*lock) (struct file *, int, struct file_lock *);
+       ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t
*);
+       ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t
*);
+       ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
+       ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
+       unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
```

```
unsigned long, unsigned long);
+       int (*check_flags)(int);
+       int (*dir_notify)(struct file *filp, unsigned long arg);
+       int (*flock) (struct file *, int, struct file_lock *);
+};
$
```

Ref [SO]: Retrieve the commit log for a specific line in a file?

Method 3

GUI

Lets say you'd like to view the detailed history of the kernel source file *kernel/fork.c* .

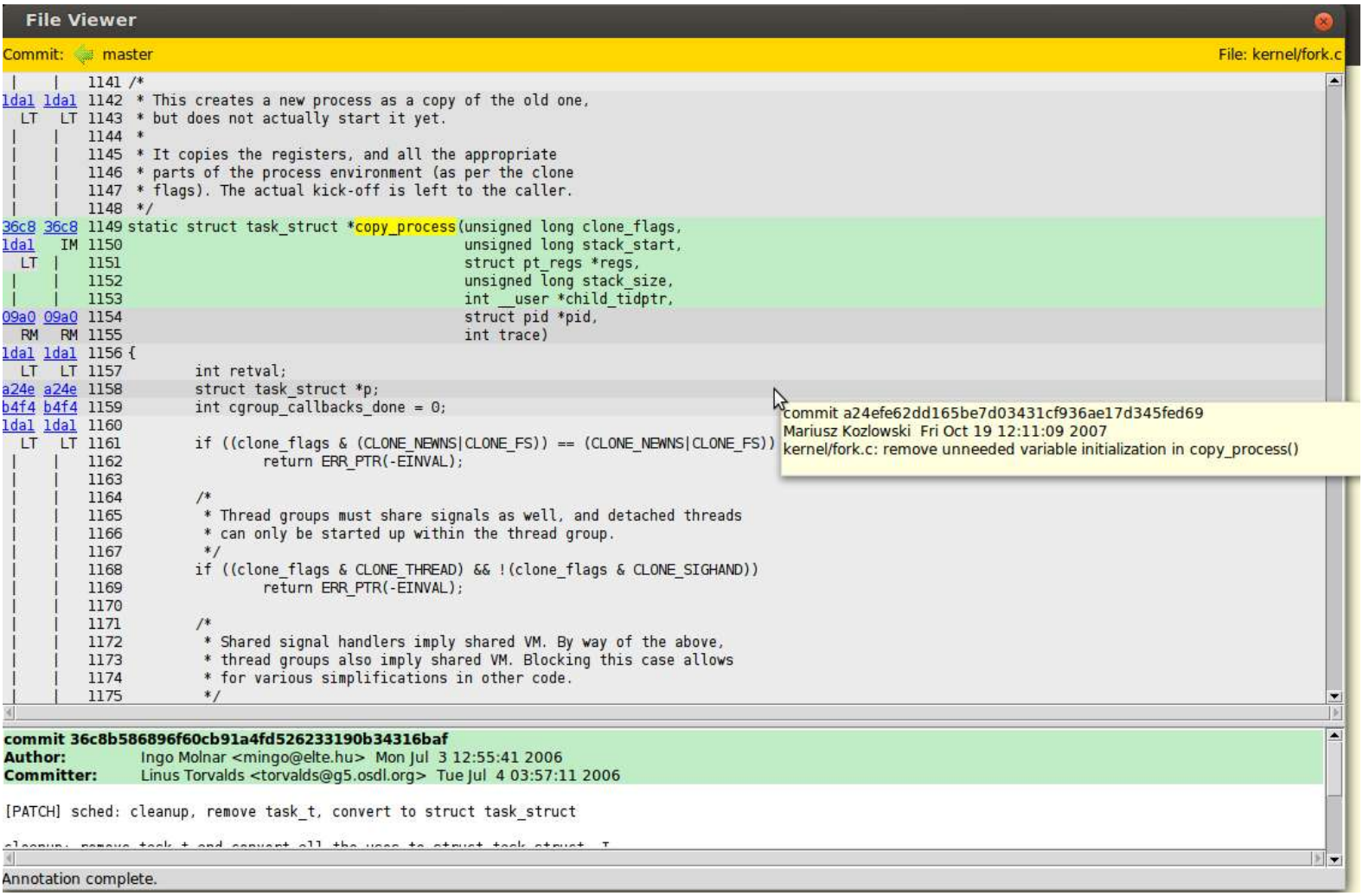
Pre-requisites:

- Clone the Linux kernel git repository onto your box with:
  `$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git`
  (this can take a while!)
- Ensure you have installed *gitk* and *git-gui* applications

Steps:

1. cd to the git kernel root source tree folder
2. Run gitk
3. Within gitk select the menu item: *File / Start git gui*
   The git-gui program runs
4. Within git-gui select the menu item: ~~*File*~~ *Repository / Browse master's Files*
5. A “File Browser” dialog comes up, displaying the contents of master:
6. Double-click the folder/file you'd like to browse
7. A “File Viewer” dialog now displays the source file contents, and annotates it (with all it's associated commits!)
   Browse through.

   (See the screenshot).



[
The commit highlighted above happened in which kernel version?

```
$ git describe --contains a24efe62
v2.6.24-rc1~144
```

]

Miscellaneous

Stashing and Applying

Sometimes, we'll have a codebase with some file(s) modified, but intentionally not staged for commit. Then, if we attempt to 'pull' deltas or switch a branch, we'll get this error:

Eg.

```
$ git status
On branch memsanit4
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
```

```
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   init/main.c
        modified:   mm/Kconfig.debug
        modified:   mm/page_poison.c
        modified:   mm/slub.c
...

$ git checkout -b memsanit5 next-20170223   << attempting to pull in recent deltas >>
error: Your local changes to the following files would be overwritten by checkout:
        init/main.c
        mm/Kconfig.debug
        mm/slub.c
Please, commit your changes or stash them before you can switch branches.
Aborting
$
```

Ref: http://stackoverflow.com/questions/15745045/how-do-i-resolve-git-saying-commit-your-changes-or-stash-them-before-you-can-me

```
$ git stash
Saved working directory and index state WIP on memsanit4: 421cf05 Add linux-next
specific files for 20170203
HEAD is now at 421cf05 Add linux-next specific files for 20170203
$
$ git checkout -b memsanit5 next-20170223   << now succeeds >>
Checking out files: 100% (4900/4900), done.
Switched to a new branch 'memsanit5'
$

$ git stash apply             << revert your local mods >>
Auto-merging mm/slub.c
Auto-merging mm/Kconfig.debug
Auto-merging init/main.c
On branch memsanit5
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   init/main.c
        modified:   mm/Kconfig.debug
        modified:   mm/page_poison.c
        modified:   mm/slub.c
...
no changes added to commit (use "git add" and/or "git commit -a")
$
```

Working on the Linux kernel with git

Download (clone) the very latest kernel source tree with:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
```

To download the latest -rc tree

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

<<

You can use the *github tree* but it's not recommended by Linus; also, you cannot fetch "subversions" (ver *x.y.z – major.minor.sublevel*) with the github tree:

```
$ git clone https://github.com/torvalds/linux.git
```
>>

Tutorials:

KernelBuild on kernelnewbies

Kernel Hackers' Guide to git

From Tobin Harding's blog

Getting Started with Linux Kernel Development - Part 1: First Patch.
Jul 19, 2017
This is the first in a series of posts about getting started in Linux kernel development. Most of what is written in this post is already available on the web. It is provided here for completeness and as a pre-amble to the next post in the series.

Getting Started with Linux Kernel Development - Part 2: The Process.
Jul 20, 2017
Part 2 of this series outlines a method for starting to learn the process of Linux kernel development. As stated in part 1, this is but one method. The aim of this post is to illuminate a pathway starting at the point when you have had your first patch merged into the mainline. If you have not had your first patch merged you may like to read part 1 of this series....

Useful: ***working at the bleeding edge! With the linux-next tree***
Working with linux-next

Same, with more details explained:

Cloning linux next tree

Tracking linux next

Linux Kernel - How to obtain a particular version (right upto SUBLEVEL)

SIDEBAR :: a simple script to help setup for Linux kernel development

---

$ cat git-clone-linux-kernel.sh

```
#!/bin/sh
# Ref: Kernel Hackers' Guide to git
#  http://linux.yyz.us/git-howto.html , and
#  http://pradheepshrinivasan.github.io/2011/12/29/cloning-linux-next-tree-i-wanted-to-
do/
name=$(basename $0)

REGULAR_TREE=0
LINUX_NEXT_TREE=1  # linux-next: working with the bleeding edge?

if [ ${REGULAR_TREE} -eq 1 -a ${LINUX_NEXT_TREE} -eq 1 ] ; then
  echo "${name}: Both 'regular' and 'linux-next' can't be cloned, choose one of them
pl.."
  exit 1
fi
if [ ${REGULAR_TREE} -eq 0 -a ${LINUX_NEXT_TREE} -eq 1 ] ; then
  [ $# -ne 1 ] && {
     echo "Usage: ${name} new-branch-to-work-under"
     exit 1
  }
  NEW_BRANCH=$1
fi

[ ${REGULAR_TREE} -eq 1 ] && {
  GITURL=https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
  echo "${name}: cloning 'regular' linux kernel now ..."
  time git clone ${GITURL}
}
# For 'regular': to update to latest:
# git pull ${GITURL}
# or just
# git pull

[ ${LINUX_NEXT_TREE} -eq 1 ] && {
  GITURL=git://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git
  echo "${name}: cloning latest 'linux-next' linux kernel now ..."
  time git clone ${GITURL}
  cd linux-next || exit 1
  echo " Running: git checkout master"
  git checkout master
  echo " Running: git remote update"
  git remote update
  LATEST_TAG=$(git tag -l next-* | tail -n1)
  echo " Latest tag: ${LATEST_TAG}"
  echo " Running: git checkout -b ${NEW_BRANCH} ${LATEST_TAG}"
  git checkout -b ${NEW_BRANCH} ${LATEST_TAG}
}

# Could use 'gitk' (or gitg) to see git repos in a GUI
# http://gitk.sourceforge.net/
# http://lostechies.com/joshuaflanagan/2010/09/03/use-gitk-to-understand-git/
```

$

Applying a git Patch

Ref: How to create and apply a patch with Git

The short ver:

1. Ensure you have the patch (in a file)
2. Look up the stats – the changes it wroughts in the codebase:
   `git apply --stat <patch>`
3. Verify that it will apply:
   `git apply --check <patch>`
4. Apply it:
   `git apply <patch>`

A quick example (applying the S.A.R.A. security LSM patch on the v4.14-rc1 kernel source):

```
$ head -n5 Makefile
VERSION = 4
PATCHLEVEL = 14
SUBLEVEL = 0
EXTRAVERSION = -rc1
NAME = Fearless Coyote

$ git tag -l next-* |tail -n1
next-20170921
$ gdiff
$ ls -l SARA_21sept2017.patch
-rw-r--r-- 1 seawolf seawolf 105543 Sep 21 20:15 SARA_21sept2017.patch
$ git apply --stat SARA_21sept2017.patch
 Documentation/admin-guide/LSM/SARA.rst          |  170 ++++++
 Documentation/admin-guide/LSM/index.rst         |    1
 Documentation/admin-guide/kernel-parameters.txt |   24 +
 include/linux/lsm_hooks.h                       |    5
 security/Kconfig                                |    1
 security/Makefile                               |    2
 security/sara/Kconfig                           |   43 +
 security/sara/Makefile                          |    3
 security/sara/include/sara.h                    |   29 +
 security/sara/include/securityfs.h              |   59 ++
 security/sara/include/utils.h                   |   69 ++
 security/sara/main.c                            |  105 ++++
 security/sara/securityfs.c                      |  560 ++++++++++++++++++++
 security/sara/utils.c                           |  151 +++++
 security/security.c                             |    1
 include/linux/lsm_hooks.h                       |    7
 include/linux/security.h                        |    6
 mm/mmap.c                                       |   13
 security/security.c                             |    5
 include/linux/cred.h                            |    3
 security/sara/Makefile                          |    2
 security/sara/include/sara_data.h               |   47 ++
```

```
 security/sara/main.c                          |    6
 security/sara/sara_data.c                     |   79 +++
 security/sara/Kconfig                         |   75 +++
 security/sara/Makefile                        |    1
 security/sara/include/utils.h                 |   11
 security/sara/include/wxprot.h                |   27 +
 security/sara/main.c                          |    6
 security/sara/wxprot.c                        |  683 ++++++++++++++++++++++++
 arch/Kconfig                                  |    6
 arch/x86/Kconfig                              |    1

[...]

 security/sara/wxprot.c                        |  150 +++++
 42 files changed, 2777 insertions(+), 3 deletions(-)
$ git apply --check  SARA_21sept2017.patch        << no issues >>
$ git apply  SARA_21sept2017.patch
$ echo $?
0
$ ls security/sara/
include/  Kconfig  main.c  Makefile  sara_data.c  securityfs.c  utils.c  wxprot.c
$
```

Done.

Useful aliases / functions

```
# shortcut for git SCM
# aliases: see https://www.linux.com/blog/git-success-stories-and-tips-kvm-maintainer-paolo-bonzini

alias gdiff='git diff -r'
alias gfiles='git diff --name-status -r'
alias gstat='git status ; echo ; git diff --stat -r'


# add a file(s) and then commit it with a commit msg
function gitac()
{
 [ $# -ne 2 ] && {
  echo "Usage: gitac filename \"commit-msg\""
  return
 }
 echo "git add $1 ..."
 git add $1
 echo "git commit -m ..."
 git commit -m "$2"
}
```

Edit or Remove Commits etc
https://sethrobertson.github.io/GitFixUm/fixup.html

...
To **remove** the **last commit** from **git**, you can simply run

git reset --hard HEAD^

If you are removing multiple **commits** from the top, you can run

git reset --hard HEAD~2

to **remove** the **last** two **commits**. You can increase the number to **remove** even more **commits**.
...

Generate Patch

git format-patch -X [--cover-letter] ; where X = # of (last) commits

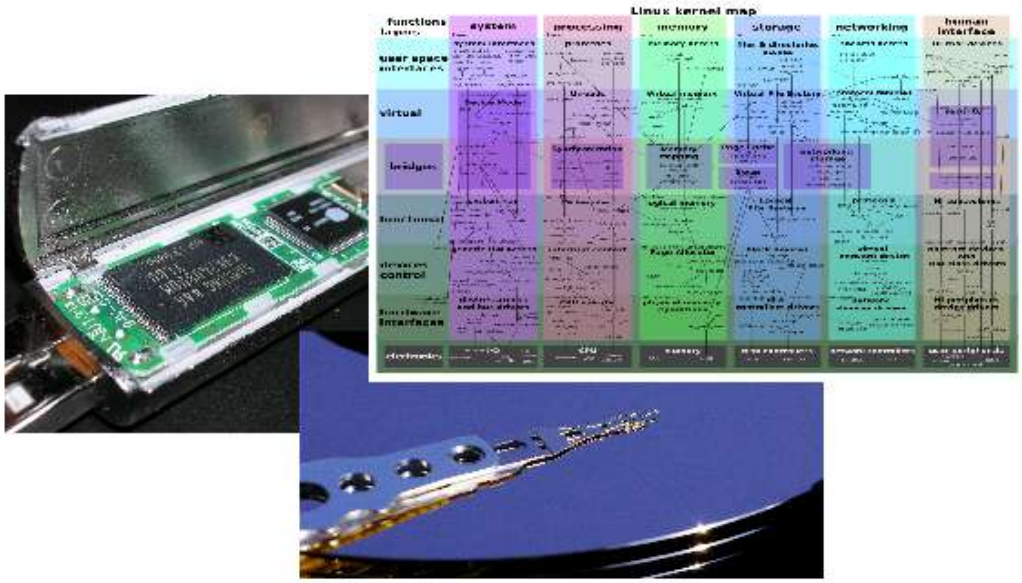
Linux Operating System Specialized

kaiwanTECH

The highest quality Training on:

Linux Fundamentals, CLI and Scripting
Linux Systems Programming
Linux Kernel Internals
Linux Device Drivers
Embedded Linux
Linux Debugging Techniques
New! *Linux OS for Technical Managers*

Please do visit our website for details:
http://kaiwantech.in



kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: http://bit.ly/ktcorp