How to Contribute to OpenEmbedded, Yocto, and Many Other Open Source Projects

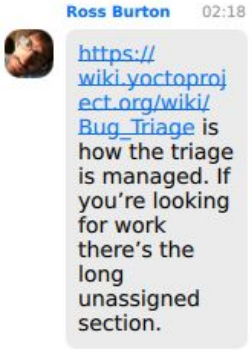
Michael Opdenacker, Bootlin

Yocto Project Summit, 2023.11

Yocto Project and OpenEmbedded Contributor Guide

- A new document added in August and September 2023
- Main goal: have a central, community reviewed reference instead of scattered, uncontrolled Wiki pages.
- Learned new Git tricks in the process and added my own

https://docs.yoctoproject.org/contributor-guide/



Goals of this presentation

- Explain how to contribute code to many projects like OpenEmbedded and Yocto Project: Linux, U-Boot, BusyBox, GNU, Git and so many others.
- Skip details specific to OE and Yocto Project.
  Read our contributor guide.
- Contributing to other projects is a way
  to contribute to OE and Yocto Project too!

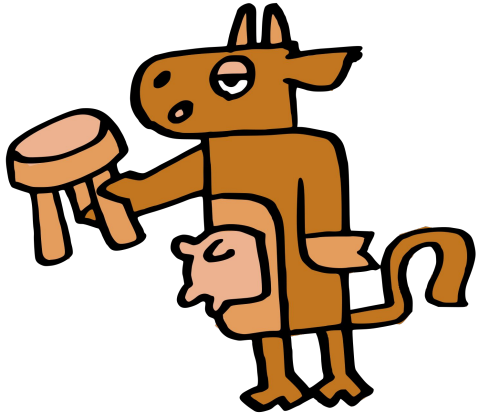
https://openclipart.org/detail/284209/jonas-6

Contributing through mailing lists

Advantages vs. web-based workflows:

- More eyes reviewing changes, instead of selected people
- Possibility to comment on specific parts of the changes
- Proven workflow that many long-time contributors are familiar with.
- Patchwork tool available to keep track of submitted patches.
- Patch testing tools for screening typical mistakes

Set up Git



Install packages first
(Debian / Ubuntu example):

```
$ sudo apt install git-core git-email
```

Then set add your name and e-mail address:

```
$ git config --global user.name "Ada Lovelace"
$ git config --global user.email "ada@bootlin.com"
```

Needed because we want signed commits!

https://openclipart.org/detail/283561/milk-me-please

Prepare your repository

Clone the repository where the code to modify lies:

```
$ git clone https://git.openembedded.org/openembedded-core
$ cd openembedded-core
```

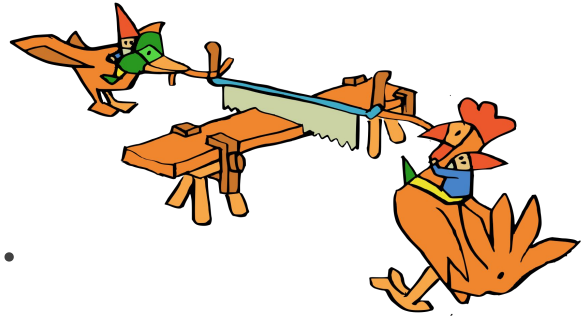
Then create one branch for each set of changes:

```
$ git checkout <ref-branch>
$ git checkout -b my-changes
```

Make sure you keep unrelated changes in different branches!

Implement and test your changes

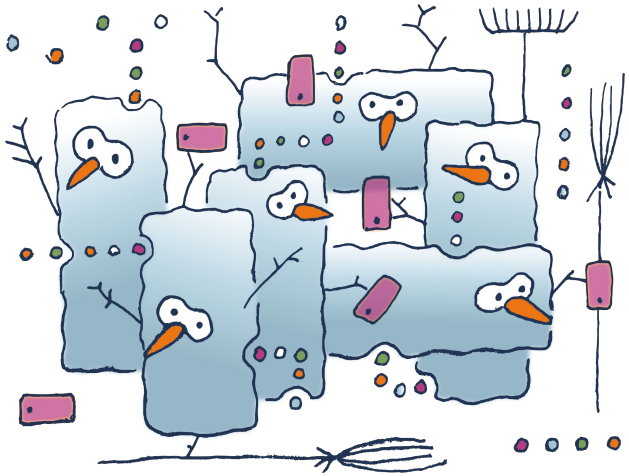
- Go ahead and implement your changes in your branch.
- Once again, only related changes
- Make sure you test them well enough.
- You don't want to ruin your reputation and make future patches harder to accept if people figure out you didn't test your changes!



https://openclipart.org/detail/284205/jonas-4

Group your changes

- Major rule: one commit per change. Commits should be atomic.
- This makes changes easier to review and accept. Some commits may be accepted in the first submission, some may need further work.
- A counter rule: possible to group changes of the same kind to different files, if they have the same maintainers.



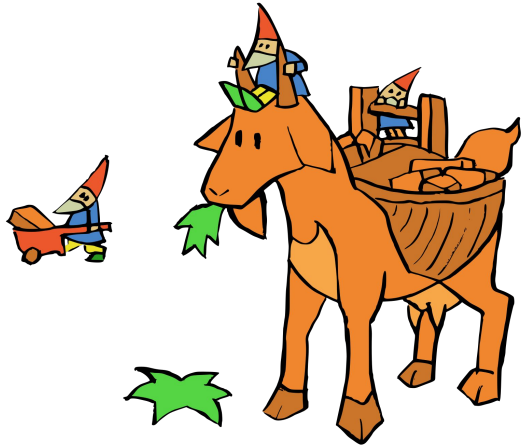
https://openclipart.org/detail/288697/smartsnow

Each change should be functional

- So that you can bisect changes (`git bisect`) to locate when a bug or regression started.
- If one of the commits doesn't build or prevents the execution of the tests, you won't be able to tell whether it falls into the "good" or "bad" sides.
- So, don't introduce a change that breaks something, followed by a change that repairs it in another way.

https://openclipart.org/detail/281312/dragonfest

Commit your changes (1)



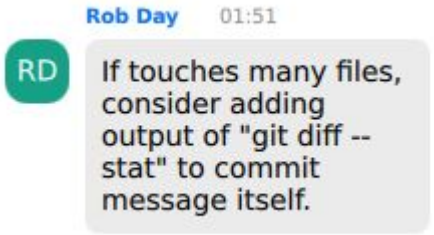
- Reminder: 1 commit per atomic change!
- For files already part of the repository, you can commit them directly:
  `git commit -s file1 file2`
- New files should be *staged* first:
  `$ git add newfile1 newfile2`
- If you want to add specific files to the next commit:
  `$ git add file1 file2` (`git add -A` for all modified files)
- Then you can commit all added files:
  `$ git commit -s`

https://openclipart.org/detail/284203/jonas-2

Describe your changes properly

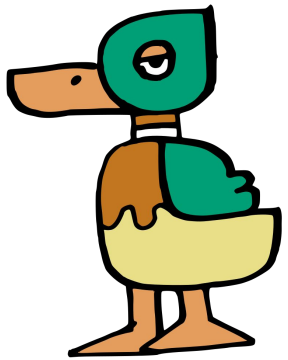
Give details such as:

- WHY the patch was created
- The consequences of not having the patch
- How it was tested
- Think about people studying the code history in the future
- Side note: use the present tense:
  `add new bike_shedding() function`



Developer's certificate of Origin

- `git commit -s` signs your commits messages:
  `Signed-off-by: Albert Dranac <dranac@bootlin.com>`
- This means you agree
  to the Developer's Certificate of Origin.
- This is required by most projects
  to trace their commits.

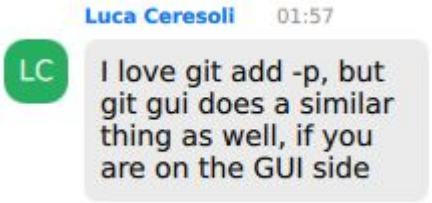


https://openclipart.org/detail/283593/duckling

Commit your changes (2)

- In case have multiple changes in the same file,
  but want to add them to separate commits.
- You can use `git add -p` to stage only selected *hunks* at a time.

```
$ git add -p
diff --git a/documentation/README b/documentation/README
index 2f077fa4bf..67dac233eb 100644
--- a/documentation/README
+++ b/documentation/README
@@ -32,6 +32,7 @@ Manual Organization
 Here the folders corresponding to individual manuals:
  * brief-yoctoprojectqs - Yocto Project Quick Start
 +* contributor-guide      - Yocto Project and OpenEmbedded Contributor Guide
  * overview-manual        - Yocto Project Overview and Concepts Manual
  * ref-manual      - Yocto Project Reference Manual
  * bsp-guide       - Yocto Project Board Support Package (BSP) Developer's Guide
(1/1) Stage this hunk [y,n,q,a,d,e,?]?
```



- Then commit the staged files, then stage the next files, etc.

Find a good commit title

- Important: commit titles are shown in the list of commits.
- Such a title should give the commit list reader a clear idea of what the commit is about.
- Some projects also expect to put a prefix to identify the modified directory, file, or subsystem. Read the contributing guidelines or see prior commit titles for the same files:

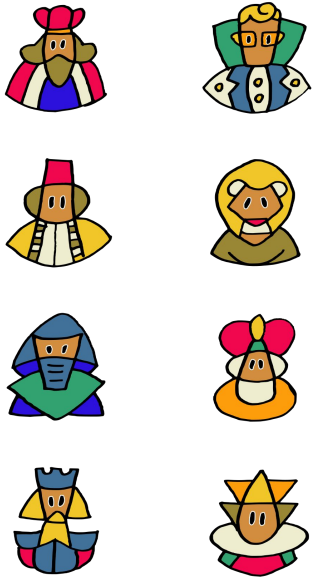
```
$ git log --oneline <paths>
13d9551ba6 vim: use upstream generated .po files
3c0deafcfc useradd_base: Fix sed command line for passwd-expire
faa32bbb35 ffmpeg: Upgrade 6.0 -> 6.1
```

Credit contributors

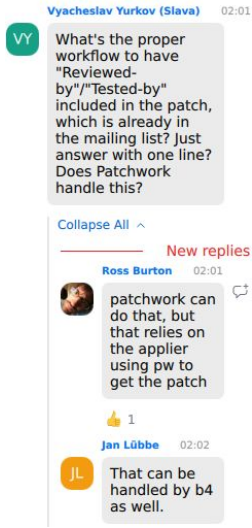
Use `git commit --amend` to add tags to the commit description

- `Reported-by`:
  Name and email of a person reporting a bug
  that your commit is trying to fix.
- `Suggested-by`:
  People to credit for the idea of making the change.
- `Tested-by`, `Reviewed-by`:
  People having tested your changes or reviewed your code.
- `CC`:
  People you want to send a copy of your changes to.

All these people will be copied when you send your patches with `git send-email`. This helps to get reviews too!

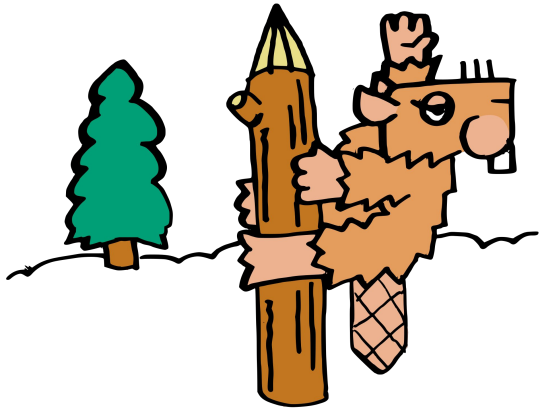


https://openclipart.org/detail/289210/its-good-to-be-king



Describe your branch

- Sending a cover letter is useful to explain your proposal if you have multiple commits in your branch.
- A convenient way is to add a description to your branch:
  `$ git branch --edit-description`
- Start the description with a first line which will be used as the subject line for the cover letter.



https://openclipart.org/detail/283592/pole-dancing

Describing a single commit

For a single commit, you probably won't have a cover letter, but there's an easy way to provide more information, by adding extra text at the end of the commit message:

```
---
Changes in V2:
- Use devm_kmalloc() instead of kmalloc()
```

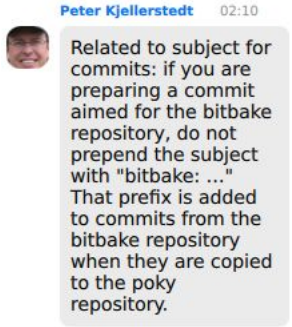
The text starting from --- won't be included in the commit message when the patch is merged.

Generate patches for your branch

- If your branch didn't need a description:
  ```
  $ git format-patch <ref-branch>
  ```
- Otherwise, generate a cover letter too:
  ```
  $ git format-patch --cover-letter \
    --cover-from-description=auto <ref-branch>
  ```
- This generates multiple patch files:
  ```
  0000-cover-letter.patch
  0001-first-change.patch
  0002-second-change.patch
  ```
- Review your patches to find last minute issues!

Add a specific prefix to patch titles

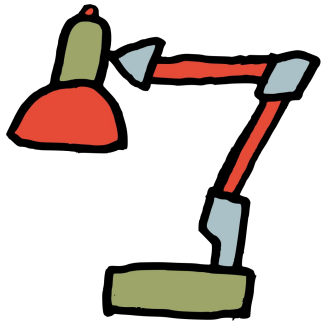
- For e-mail review, can be required to prefix the commit with the branch it applies to.
- However, you don't want this prefix to appear in the final commit lists.
- Bad example:
  ```
  $ git commit -s "[kirkstone] manuals: add 4.0.12 release notes"
  $ git format-patch kirkstone
  ```
  Patch title: `[PATCH] [kirkstone] manuals: add 4.0.12 release notes`
  Applied patch commit title: `[kirkstone] manuals: add 4.0.12 release notes`
- Good example:
  ```
  $ git commit -s "manuals: add 4.0.12 release notes"
  $ git format-patch –subject-prefix="kirkstone][PATCH" kirkstone
  ```
  Patch title: `[kirkstone][PATCH] manuals: add 4.0.12 release notes`
  Applied patch commit title: `manuals: add 4.0.12 release notes`



Check your patches

Some projects have tools to inspect patches
(coding rules, required information, typical mistakes…)

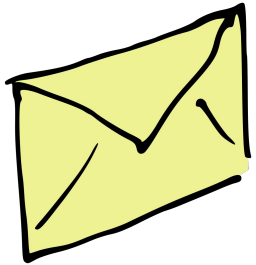
- Linux kernel / U-Boot example:
  ```
  $ scripts/checkpatch.pl *.patch
  ```
- OE / Yocto project:
  ```
  $ patchtest -patch file.patch
  ```



https://openclipart.org/detail/284026/desk-lamp

Send patches by e-mail

- Important to send patches as *inline attachments* (in the e-mail body, not as regular attachments). This way reviewers can easily "reply" to any of your patch lines.
- Only `git send-email` can do this reliably without making the received patches inapplicable.



https://openclipart.org/detail/283663/mail

Configuring git send-email

- Either use a local Mail Transport Agent such as `msmtp` or `sendmail`.
- Or let `git send-email` connect to your regular SMTP server.
  Example for Google Mail:

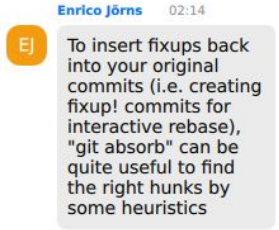
```
$ git config --global sendemail.smtpserver smtp.gmail.com
$ git config --global sendemail.smtpserverport 587
$ git config --global sendemail.smtpencryption tls
$ git config --global sendemail.smtpuser william.gates@gmail.com
$ git config --global sendemail.smtppass = XXXXXXXX
```

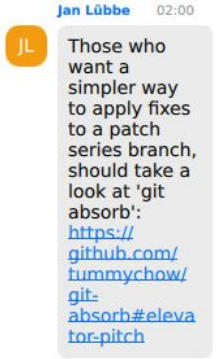
Sending the patches

- Some projects such as the Linux kernel have a nice command to figure out who to send a patch to:
  ```
  $ ./scripts/get_maintainer.pl *.patch
  ```
- For other projects, read the contributing guidelines
- Then send the patches to the expected recipients:
  ```
  $ git send-email --to … --cc … *.patch
  ```
- For a repository with always the same contact address:
  ```
  $ git config --local sendemail.to \
     openembedded-core@lists.openembedded.org
  ```

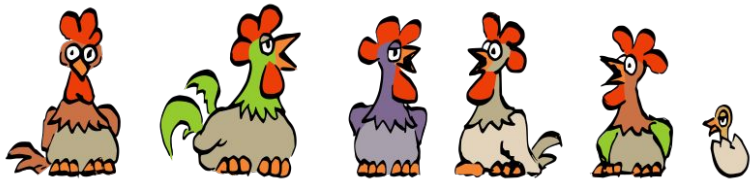
Taking reviews into account

- Reviews will make you modify your changes, and may even change your list of commits (number, order, description).
- Don't add fixes as extra commits at the end of your branch. Instead, rework your commits as if you were sending them for the first time.
- A branch with a single commit is easy to modify:
  `$ git commit --amend`
- `git rebase` in *interactive mode* is a great tool to rework a branch with multiple commits, reordering, editing, merging or deleting them:
  `$ git rebase -i <ref-branch>`





https://openclipart.org/detail/283814/chick-chat



Submitting a new iteration

- Update branch information (`git branch --edit-description`) to help people who have reviewed previous iterations:

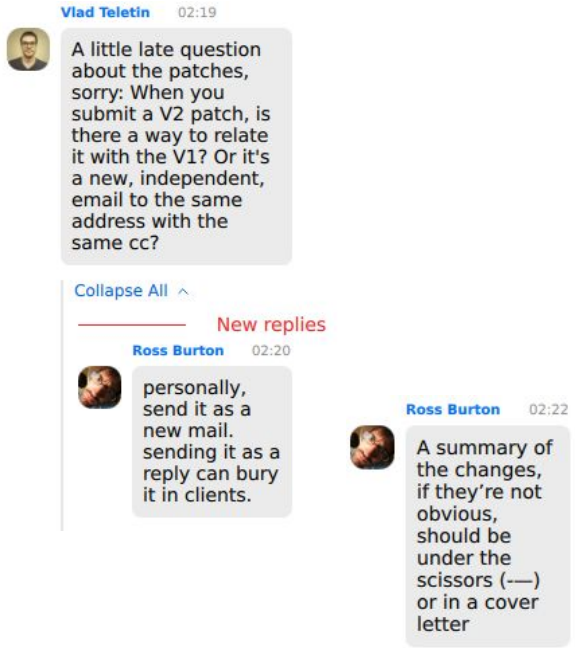
```
- Changes in V3:
    Fix null pointer dereference
- Changes in V2:
    Fix checkpatch issues
```

- Regenerate patches with `-v3` (for version 3)…

```
$ git format-patch -v3 ...
```

This generates `v3-0***.patch` files with a `[PATCH v3]` subject line.

- Send the patches in the usual way.



What to remember

- 1 commit per atomic change
- 1 branch per change set
- `git add -p`
  to stage specific hunks
- Each change should work
- Test your changes
- Describe your changes well
- Use `git commit -s`
- Credit contributors
- Store cover letter in branch
- Send with `git send-email`
- `git commit --amend`
  to edit a single patch
- `git rebase -i`
  to rework an entire branch

Questions?

PDF and editable sources:
https://bootlin.com/pub/conferences/2023/yp-summit/
Image credits:
https://openclipart.org/artist/klaro





yocto PROJECT | THE LINUX FOUNDATION