Linux Maintainer Documentation

The kernel development community

Jun 10, 2024

CONTENTS

This document is the humble beginning of a manual for kernel maintainers. There is a lot yet to go here! Please feel free to propose (and write) additions to this manual.

CHAPTER

ONE

FEATURE AND DRIVER MAINTAINERS

The term “maintainer” spans a very wide range of levels of engagement from people handling patches and pull requests as almost a full time job to people responsible for a small feature or a driver.

Unlike most of the chapter, this section is meant for the latter (more populous) group. It provides tips and describes the expectations and responsibilities of maintainers of a small(ish) section of the code.

Drivers and alike most often do not have their own mailing lists and git trees but instead send and review patches on the list of a larger subsystem.

1.1 Responsibilities

The amount of maintenance work is usually proportional to the size and popularity of the code base. Small features and drivers should require relatively small amount of care and feeding. Nonetheless when the work does arrive (in form of patches which need review, user bug reports etc.) it has to be acted upon promptly. Even when a particular driver only sees one patch a month, or a quarter, a subsystem could well have a hundred such drivers. Subsystem maintainers cannot afford to wait a long time to hear from reviewers.

The exact expectations on the response time will vary by subsystem. The patch review SLA the subsystem had set for itself can sometimes be found in the subsystem documentation. Failing that as a rule of thumb reviewers should try to respond quicker than what is the usual patch review delay of the subsystem maintainer. The resulting expectations may range from two working days for fast-paced subsystems (e.g. networking) to as long as a few weeks in slower moving parts of the kernel.

1.1.1 Mailing list participation

Linux kernel uses mailing lists as the primary form of communication. Maintainers must be subscribed and follow the appropriate subsystem-wide mailing list. Either by subscribing to the whole list or using more modern, selective setup like lei.

Maintainers must know how to communicate on the list (plain text, no invasive legal footers, no top posting, etc.)

1.1.2 Reviews

Maintainers must review *all* patches touching exclusively their drivers, no matter how trivial. If the patch is a tree wide change and modifies multiple drivers - whether to provide a review is left to the maintainer.

When there are multiple maintainers for a piece of code an `Acked-by` or `Reviewed-by` tag (or review comments) from a single maintainer is enough to satisfy this requirement.

If the review process or validation for a particular change will take longer than the expected review timeline for the subsystem, maintainer should reply to the submission indicating that the work is being done, and when to expect full results.

1.1.3 Refactoring and core changes

Occasionally core code needs to be changed to improve the maintainability of the kernel as a whole. Maintainers are expected to be present and help guide and test changes to their code to fit the new infrastructure.

1.1.4 Bug reports

Maintainers must ensure severe problems in their code reported to them are resolved in a timely manner: regressions, kernel crashes, kernel warnings, compilation errors, lockups, data loss, and other bugs of similar scope.

Maintainers furthermore should respond to reports about other kinds of bugs as well, if the report is of reasonable quality or indicates a problem that might be severe -- especially if they have *Supported* status of the codebase in the MAINTAINERS file.

1.2 Selecting the maintainer

The previous section described the expectations of the maintainer, this section provides guidance on selecting one and describes common misconceptions.

1.2.1 The author

Most natural and common choice of a maintainer is the author of the code. The author is intimately familiar with the code, so it is the best person to take care of it on an ongoing basis.

That said, being a maintainer is an active role. The MAINTAINERS file is not a list of credits (in fact a separate CREDITS file exists), it is a list of those who will actively help with the code. If the author does not have the time, interest or ability to maintain the code, a different maintainer must be selected.

1.2.2 Multiple maintainers

Modern best practices dictate that there should be at least two maintainers for any piece of code, no matter how trivial. It spreads the burden, helps people take vacations and prevents burnout, trains new members of the community etc. etc. Even when there is clearly one perfect candidate, another maintainer should be found.

Maintainers must be human, therefore, it is not acceptable to add a mailing list or a group email as a maintainer. Trust and understanding are the foundation of kernel maintenance and one cannot build trust with a mailing list. Having a mailing list *in addition* to humans is perfectly fine.

1.2.3 Corporate structures

To an outsider the Linux kernel may resemble a hierarchical organization with Linus as the CEO. While the code flows in a hierarchical fashion, the corporate template does not apply here. Linux is an anarchy held together by (rarely expressed) mutual respect, trust and convenience.

All that is to say that managers almost never make good maintainers. The maintainer position more closely matches an on-call rotation than a position of power.

The following characteristics of a person selected as a maintainer are clear red flags:

- unknown to the community, never sent an email to the list before
- did not author any of the code
- (when development is contracted) works for a company which paid for the development rather than the company which did the work

1.3 Non compliance

Subsystem maintainers may remove inactive maintainers from the MAINTAINERS file. If the maintainer was a significant author or played an important role in the development of the code, they should be moved to the CREDITS file.

Removing an inactive maintainer should not be seen as a punitive action. Having an inactive maintainer has a real cost as all developers have to remember to include the maintainers in discussions and subsystem maintainers spend brain power figuring out how to solicit feedback.

Subsystem maintainers may remove code for lacking maintenance.

Subsystem maintainers may refuse accepting code from companies which repeatedly neglected their maintainership duties.

CHAPTER

TWO

CONFIGURING GIT

This chapter describes maintainer level git configuration.

Tagged branches used in pull requests (see *Creating Pull Requests*) should be signed with the developers public GPG key. Signed tags can be created by passing `-u <key-id>` to `git tag`. However, since you would *usually* use the same key for the project, you can set it in the configuration and use the `-s` flag. To set the default `key-id` use:

```
git config user.signingkey "keyname"
```

Alternatively, edit your `.git/config` or `~/.gitconfig` file by hand:

```
[user]
        name = Jane Developer
        email = jd@domain.org
        signingkey = jd@domain.org
```

You may need to tell `git` to use `gpg2`:

```
[gpg]
        program = /path/to/gpg2
```

You may also like to tell `gpg` which `tty` to use (add to your shell rc file):

```
export GPG_TTY=$(tty)
```

2.1 Creating commit links to lore.kernel.org

The web site https://lore.kernel.org is meant as a grand archive of all mail list traffic concerning or influencing the kernel development. Storing archives of patches here is a recommended practice, and when a maintainer applies a patch to a subsystem tree, it is a good idea to provide a Link: tag with a reference back to the lore archive so that people that browse the commit history can find related discussions and rationale behind a certain change. The link tag will look like this:

```
Link: https://lore.kernel.org/r/<message-id>
```

This can be configured to happen automatically any time you issue `git am` by adding the following hook into your git:

```
$ git config am.messageid true
$ cat >.git/hooks/applypatch-msg <<'EOF'
#!/bin/sh
. git-sh-setup
perl -pi -e 's|^Message-I[dD]:\s*<?([^>]+)>?$|Link: https://lore.kernel.org/r/
↪$1|g;' "$1"
test -x "$GIT_DIR/hooks/commit-msg" &&
        exec "$GIT_DIR/hooks/commit-msg" ${1+"$@"}
:
EOF
$ chmod a+x .git/hooks/applypatch-msg
```

CHAPTER

THREE

REBASING AND MERGING

Maintaining a subsystem, as a general rule, requires a familiarity with the Git source-code management system. Git is a powerful tool with a lot of features; as is often the case with such tools, there are right and wrong ways to use those features. This document looks in particular at the use of rebasing and merging. Maintainers often get in trouble when they use those tools incorrectly, but avoiding problems is not actually all that hard.

One thing to be aware of in general is that, unlike many other projects, the kernel community is not scared by seeing merge commits in its development history. Indeed, given the scale of the project, avoiding merges would be nearly impossible. Some problems encountered by maintainers result from a desire to avoid merges, while others come from merging a little too often.

3.1 Rebasing

"Rebasing" is the process of changing the history of a series of commits within a repository. There are two different types of operations that are referred to as rebasing since both are done with the `git rebase` command, but there are significant differences between them:

- Changing the parent (starting) commit upon which a series of patches is built. For example, a rebase operation could take a patch set built on the previous kernel release and base it, instead, on the current release. We'll call this operation "reparenting" in the discussion below.
- Changing the history of a set of patches by fixing (or deleting) broken commits, adding patches, adding tags to commit changelogs, or changing the order in which commits are applied. In the following text, this type of operation will be referred to as "history modification"

The term "rebasing" will be used to refer to both of the above operations. Used properly, rebasing can yield a cleaner and clearer development history; used improperly, it can obscure that history and introduce bugs.

There are a few rules of thumb that can help developers to avoid the worst perils of rebasing:

- History that has been exposed to the world beyond your private system should usually not be changed. Others may have pulled a copy of your tree and built on it; modifying your tree will create pain for them. If work is in need of rebasing, that is usually a sign that it is not yet ready to be committed to a public repository.

  That said, there are always exceptions. Some trees (linux-next being a significant example) are frequently rebased by their nature, and developers know not to base work on them.

Developers will sometimes expose an unstable branch for others to test with or for automated testing services. If you do expose a branch that may be unstable in this way, be sure that prospective users know not to base work on it.

- Do not rebase a branch that contains history created by others. If you have pulled changes from another developer's repository, you are now a custodian of their history. You should not change it. With few exceptions, for example, a broken commit in a tree like this should be explicitly reverted rather than disappeared via history modification.
- Do not reparent a tree without a good reason to do so. Just being on a newer base or avoiding a merge with an upstream repository is not generally a good reason.
- If you must reparent a repository, do not pick some random kernel commit as the new base. The kernel is often in a relatively unstable state between release points; basing development on one of those points increases the chances of running into surprising bugs. When a patch series must move to a new base, pick a stable point (such as one of the -rc releases) to move to.
- Realize that reparenting a patch series (or making significant history modifications) changes the environment in which it was developed and, likely, invalidates much of the testing that was done. A reparented patch series should, as a general rule, be treated like new code and retested from the beginning.

A frequent cause of merge-window trouble is when Linus is presented with a patch series that has clearly been reparented, often to a random commit, shortly before the pull request was sent. The chances of such a series having been adequately tested are relatively low - as are the chances of the pull request being acted upon.

If, instead, rebasing is limited to private trees, commits are based on a well-known starting point, and they are well tested, the potential for trouble is low.

3.2 Merging

Merging is a common operation in the kernel development process; the 5.1 development cycle included 1,126 merge commits - nearly 9% of the total. Kernel work is accumulated in over 100 different subsystem trees, each of which may contain multiple topic branches; each branch is usually developed independently of the others. So naturally, at least one merge will be required before any given branch finds its way into an upstream repository.

Many projects require that branches in pull requests be based on the current trunk so that no merge commits appear in the history. The kernel is not such a project; any rebasing of branches to avoid merges will, most likely, lead to trouble.

Subsystem maintainers find themselves having to do two types of merges: from lower-level subsystem trees and from others, either sibling trees or the mainline. The best practices to follow differ in those two situations.

3.2.1 Merging from lower-level trees

Larger subsystems tend to have multiple levels of maintainers, with the lower-level maintainers sending pull requests to the higher levels. Acting on such a pull request will almost certainly generate a merge commit; that is as it should be. In fact, subsystem maintainers may want to use the --no-ff flag to force the addition of a merge commit in the rare cases where one would not normally be created so that the reasons for the merge can be recorded. The changelog for the merge should, for any kind of merge, say *why* the merge is being done. For a lower-level tree, "why" is usually a summary of the changes that will come with that pull.

Maintainers at all levels should be using signed tags on their pull requests, and upstream maintainers should verify the tags when pulling branches. Failure to do so threatens the security of the development process as a whole.

As per the rules outlined above, once you have merged somebody else's history into your tree, you cannot rebase that branch, even if you otherwise would be able to.

3.2.2 Merging from sibling or upstream trees

While merges from downstream are common and unremarkable, merges from other trees tend to be a red flag when it comes time to push a branch upstream. Such merges need to be carefully thought about and well justified, or there's a good chance that a subsequent pull request will be rejected.

It is natural to want to merge the master branch into a repository; this type of merge is often called a "back merge". Back merges can help to make sure that there are no conflicts with parallel development and generally gives a warm, fuzzy feeling of being up-to-date. But this temptation should be avoided almost all of the time.

Why is that? Back merges will muddy the development history of your own branch. They will significantly increase your chances of encountering bugs from elsewhere in the community and make it hard to ensure that the work you are managing is stable and ready for upstream. Frequent merges can also obscure problems with the development process in your tree; they can hide interactions with other trees that should not be happening (often) in a well-managed branch.

That said, back merges are occasionally required; when that happens, be sure to document *why* it was required in the commit message. As always, merge to a well-known stable point, rather than to some random commit. Even then, you should not back merge a tree above your immediate upstream tree; if a higher-level back merge is really required, the upstream tree should do it first.

One of the most frequent causes of merge-related trouble is when a maintainer merges with the upstream in order to resolve merge conflicts before sending a pull request. Again, this temptation is easy enough to understand, but it should absolutely be avoided. This is especially true for the final pull request: Linus is adamant that he would much rather see merge conflicts than unnecessary back merges. Seeing the conflicts lets him know where potential problem areas are. He does a lot of merges (382 in the 5.1 development cycle) and has gotten quite good at conflict resolution - often better than the developers involved.

So what should a maintainer do when there is a conflict between their subsystem branch and the mainline? The most important step is to warn Linus in the pull request that the conflict will happen; if nothing else, that demonstrates an awareness of how your branch fits into the whole. For especially difficult conflicts, create and push a *separate* branch to show how you

would resolve things. Mention that branch in your pull request, but the pull request itself should be for the unmerged branch.

Even in the absence of known conflicts, doing a test merge before sending a pull request is a good idea. It may alert you to problems that you somehow didn't see from linux-next and helps to understand exactly what you are asking upstream to do.

Another reason for doing merges of upstream or another subsystem tree is to resolve dependencies. These dependency issues do happen at times, and sometimes a cross-merge with another tree is the best way to resolve them; as always, in such situations, the merge commit should explain why the merge has been done. Take a moment to do it right; people will read those changelogs.

Often, though, dependency issues indicate that a change of approach is needed. Merging another subsystem tree to resolve a dependency risks bringing in other bugs and should almost never be done. If that subsystem tree fails to be pulled upstream, whatever problems it had will block the merging of your tree as well. Preferable alternatives include agreeing with the maintainer to carry both sets of changes in one of the trees or creating a topic branch dedicated to the prerequisite commits that can be merged into both trees. If the dependency is related to major infrastructural changes, the right solution might be to hold the dependent commits for one development cycle so that those changes have time to stabilize in the mainline.

3.3 Finally

It is relatively common to merge with the mainline toward the beginning of the development cycle in order to pick up changes and fixes done elsewhere in the tree. As always, such a merge should pick a well-known release point rather than some random spot. If your upstream-bound branch has emptied entirely into the mainline during the merge window, you can pull it forward with a command like:

```
git merge --ff-only v5.2-rc1
```

The guidelines laid out above are just that: guidelines. There will always be situations that call out for a different solution, and these guidelines should not prevent developers from doing the right thing when the need arises. But one should always think about whether the need has truly arisen and be prepared to explain why something abnormal needs to be done.

CHAPTER

FOUR

CREATING PULL REQUESTS

This chapter describes how maintainers can create and submit pull requests to other maintainers. This is useful for transferring changes from one maintainers tree to another maintainers tree.

This document was written by Tobin C. Harding (who at that time, was not an experienced maintainer) primarily from comments made by Greg Kroah-Hartman and Linus Torvalds on LKML. Suggestions and fixes by Jonathan Corbet and Mauro Carvalho Chehab. Misrepresentation was unintentional but inevitable, please direct abuse to Tobin C. Harding <me@tobin.cc>.

Original email thread:

```
https://lore.kernel.org/r/20171114110500.GA21175@kroah.com
```

4.1 Create Branch

To start with you will need to have all the changes you wish to include in the pull request on a separate branch. Typically you will base this branch off of a branch in the developers tree whom you intend to send the pull request to.

In order to create the pull request you must first tag the branch that you have just created. It is recommended that you choose a meaningful tag name, in a way that you and others can understand, even after some time. A good practice is to include in the name an indicator of the subsystem of origin and the target kernel version.

Greg offers the following. A pull request with miscellaneous stuff for drivers/char, to be applied at the Kernel version 4.15-rc1 could be named as `char-misc-4.15-rc1`. If such tag would be produced from a branch named `char-misc-next`, you would be using the following command:

```
git tag -s char-misc-4.15-rc1 char-misc-next
```

that will create a signed tag called `char-misc-4.15-rc1` based on the last commit in the `char-misc-next` branch, and sign it with your gpg key (see *Configuring Git*).

Linus will only accept pull requests based on a signed tag. Other maintainers may differ.

When you run the above command `git` will drop you into an editor and ask you to describe the tag. In this case, you are describing a pull request, so outline what is contained here, why it should be merged, and what, if any, testing has been done. All of this information will end up in the tag itself, and then in the merge commit that the maintainer makes if/when they merge the pull request. So write it up well, as it will be in the kernel tree for forever.

As said by Linus:

```
Anyway, at least to me, the important part is the *message*. I want
to understand what I'm pulling, and why I should pull it. I also
want to use that message as the message for the merge, so it should
not just make sense to me, but make sense as a historical record
too.

Note that if there is something odd about the pull request, that
should very much be in the explanation. If you're touching files
that you don't maintain, explain _why_. I will see it in the
diffstat anyway, and if you didn't mention it, I'll just be extra
suspicious.  And when you send me new stuff after the merge window
(or even bug-fixes, but ones that look scary), explain not just
what they do and why they do it, but explain the _timing_. What
happened that this didn't go through the merge window..

I will take both what you write in the email pull request _and_ in
the signed tag, so depending on your workflow, you can either
describe your work in the signed tag (which will also automatically
make it into the pull request email), or you can make the signed
tag just a placeholder with nothing interesting in it, and describe
the work later when you actually send me the pull request.

And yes, I will edit the message. Partly because I tend to do just
trivial formatting (the whole indentation and quoting etc), but
partly because part of the message may make sense for me at pull
time (describing the conflicts and your personal issues for sending
it right now), but may not make sense in the context of a merge
commit message, so I will try to make it all make sense. I will
also fix any speeling mistaeks and bad grammar I notice,
particularly for non-native speakers (but also for native ones
;^). But I may miss some, or even add some.

                Linus
```

Greg gives, as an example pull request:

```
Char/Misc patches for 4.15-rc1

Here is the big char/misc patch set for the 4.15-rc1 merge window.
Contained in here is the normal set of new functions added to all
of these crazy drivers, as well as the following brand new
subsystems:
        - time_travel_controller: Finally a set of drivers for the
          latest time travel bus architecture that provides i/o to
          the CPU before it asked for it, allowing uninterrupted
          processing
        - relativity_shifters: due to the affect that the
          time_travel_controllers have on the overall system, there
          was a need for a new set of relativity shifter drivers to
          accommodate the newly formed black holes that would
```

```
          threaten to suck CPUs into them.  This subsystem handles
          this in a way to successfully neutralize the problems.
          There is a Kconfig option to force these to be enabled
          when needed, so problems should not occur.

All of these patches have been successfully tested in the latest
linux-next releases, and the original problems that it found have
all been resolved (apologies to anyone living near Canberra for the
lack of the Kconfig options in the earlier versions of the
linux-next tree creations.)

Signed-off-by: Your-name-here <your_email@domain>
```

The tag message format is just like a git commit id. One line at the top for a "summary subject" and be sure to sign-off at the bottom.

Now that you have a local signed tag, you need to push it up to where it can be retrieved:

```
git push origin char-misc-4.15-rc1
```

4.2 Create Pull Request

The last thing to do is create the pull request message. `git` handily will do this for you with the `git request-pull` command, but it needs a bit of help determining what you want to pull, and on what to base the pull against (to show the correct changes to be pulled and the diffstat). The following command(s) will generate a pull request:

```
git request-pull master git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/
↪char-misc.git/ char-misc-4.15-rc1
```

Quoting Greg:

```
This is asking git to compare the difference from the
'char-misc-4.15-rc1' tag location, to the head of the 'master'
branch (which in my case points to the last location in Linus's
tree that I diverged from, usually a -rc release) and to use the
git:// protocol to pull from.  If you wish to use https://, that
can be used here instead as well (but note that some people behind
firewalls will have problems with https git pulls).

If the char-misc-4.15-rc1 tag is not present in the repo that I am
asking to be pulled from, git will complain saying it is not there,
a handy way to remember to actually push it to a public location.

The output of 'git request-pull' will contain the location of the
git tree and specific tag to pull from, and the full text
description of that tag (which is why you need to provide good
information in that tag).  It will also create a diffstat of the
pull request, and a shortlog of the individual commits that the
pull request will provide.
```

Linus responded that he tends to prefer the `git://` protocol. Other maintainers may have different preferences. Also, note that if you are creating pull requests without a signed tag then `https://` may be a better choice. Please see the original thread for the full discussion.

4.3 Submit Pull Request

A pull request is submitted in the same way as an ordinary patch. Send as inline email to the maintainer and CC LKML and any sub-system specific lists if required. Pull requests to Linus typically have a subject line something like:

```
[GIT PULL] <subsystem> changes for v4.15-rc1
```

CHAPTER

FIVE

HANDLING MESSY PULL-REQUEST DIFFSTATS

Subsystem maintainers routinely use `git request-pull` as part of the process of sending work upstream. Normally, the result includes a nice diffstat that shows which files will be touched and how much of each will be changed. Occasionally, though, a repository with a relatively complicated development history will yield a massive diffstat containing a great deal of unrelated work. The result looks ugly and obscures what the pull request is actually doing. This document describes what is happening and how to fix things up; it is derived from The Wisdom of Linus Torvalds, found in Linus1 and Linus2.

A Git development history proceeds as a series of commits. In a simplified manner, mainline kernel development looks like this:

```
... vM --- vN-rc1 --- vN-rc2 --- vN-rc3 --- ... --- vN-rc7 --- vN
```

If one wants to see what has changed between two points, a command like this will do the job:

```
$ git diff --stat --summary vN-rc2..vN-rc3
```

Here, there are two clear points in the history; Git will essentially "subtract" the beginning point from the end point and display the resulting differences. The requested operation is unambiguous and easy enough to understand.

When a subsystem maintainer creates a branch and commits changes to it, the result in the simplest case is a history that looks like:

```
... vM --- vN-rc1 --- vN-rc2 --- vN-rc3 --- ... --- vN-rc7 --- vN
                        |
                        +-- c1 --- c2 --- ... --- cN
```

If that maintainer now uses `git diff` to see what has changed between the mainline branch (let's call it "linus") and cN, there are still two clear endpoints, and the result is as expected. So a pull request generated with `git request-pull` will also be as expected. But now consider a slightly more complex development history:

```
... vM --- vN-rc1 --- vN-rc2 --- vN-rc3 --- ... --- vN-rc7 --- vN
             |          |
             |          +-- c1 --- c2 --- ... --- cN
             |                      /
             +-- x1 --- x2 --- x3
```

Our maintainer has created one branch at vN-rc1 and another at vN-rc2; the two were then subsequently merged into c2. Now a pull request generated for cN may end up being messy

indeed, and developers often end up wondering why.

What is happening here is that there are no longer two clear end points for the `git diff` operation to use. The development culminating in cN started in two different places; to generate the diffstat, `git diff` ends up having pick one of them and hoping for the best. If the diffstat starts at vN-rc1, it may end up including all of the changes between there and the second origin end point (vN-rc2), which is certainly not what our maintainer had in mind. With all of that extra junk in the diffstat, it may be impossible to tell what actually happened in the changes leading up to cN.

Maintainers often try to resolve this problem by, for example, rebasing the branch or performing another merge with the linus branch, then recreating the pull request. This approach tends not to lead to joy at the receiving end of that pull request; rebasing and/or merging just before pushing upstream is a well-known way to get a grumpy response.

So what is to be done? The best response when confronted with this situation is to indeed to do a merge with the branch you intend your work to be pulled into, but to do it privately, as if it were the source of shame. Create a new, throwaway branch and do the merge there:

```
... vM --- vN-rc1 --- vN-rc2 --- vN-rc3 --- ... --- vN-rc7 --- vN
             |         |                                       |
             |         +-- c1 --- c2 --- ... --- cN            |
             |                    /                |            |
             +-- x1 --- x2 --- x3                  +------------+-- TEMP
```

The merge operation resolves all of the complications resulting from the multiple beginning points, yielding a coherent result that contains only the differences from the mainline branch. Now it will be possible to generate a diffstat with the desired information:

```
$ git diff -C --stat --summary linus..TEMP
```

Save the output from this command, then simply delete the TEMP branch; definitely do not expose it to the outside world. Take the saved diffstat output and edit it into the messy pull request, yielding a result that shows what is really going on. That request can then be sent upstream.

CHAPTER

SIX

MAINTAINER ENTRY PROFILE

The Maintainer Entry Profile supplements the top-level process documents (submitting-patches, submitting drivers...) with subsystem/device-driver-local customs as well as details about the patch submission life-cycle. A contributor uses this document to level set their expectations and avoid common mistakes; maintainers may use these profiles to look across subsystems for opportunities to converge on common practices.

6.1 Overview

Provide an introduction to how the subsystem operates. While MAINTAINERS tells the contributor where to send patches for which files, it does not convey other subsystem-local infrastructure and mechanisms that aid development.

Example questions to consider:

- Are there notifications when patches are applied to the local tree, or merged upstream?
- Does the subsystem have a patchwork instance? Are patchwork state changes notified?
- Any bots or CI infrastructure that watches the list, or automated testing feedback that the subsystem uses to gate acceptance?
- Git branches that are pulled into -next?
- What branch should contributors submit against?
- Links to any other Maintainer Entry Profiles? For example a device-driver may point to an entry for its parent subsystem. This makes the contributor aware of obligations a maintainer may have for other maintainers in the submission chain.

6.2 Submit Checklist Addendum

List mandatory and advisory criteria, beyond the common “submit-checklist”, for a patch to be considered healthy enough for maintainer attention. For example: “pass checkpatch.pl with no errors, or warning. Pass the unit test detailed at $URI”.

The Submit Checklist Addendum can also include details about the status of related hardware specifications. For example, does the subsystem require published specifications at a certain revision before patches will be considered.

6.3 Key Cycle Dates

One of the common misunderstandings of submitters is that patches can be sent at any time before the merge window closes and can still be considered for the next -rc1. The reality is that most patches need to be settled in soaking in linux-next in advance of the merge window opening. Clarify for the submitter the key dates (in terms of -rc release week) that patches might be considered for merging and when patches need to wait for the next -rc. At a minimum:

- Last -rc for new feature submissions: New feature submissions targeting the next merge window should have their first posting for consideration before this point. Patches that are submitted after this point should be clear that they are targeting the NEXT+1 merge window, or should come with sufficient justification why they should be considered on an expedited schedule. A general guideline is to set expectation with contributors that new feature submissions should appear before -rc5.
- Last -rc to merge features: Deadline for merge decisions Indicate to contributors the point at which an as yet un-applied patch set will need to wait for the NEXT+1 merge window. Of course there is no obligation to ever accept any given patchset, but if the review has not concluded by this point the expectation is the contributor should wait and resubmit for the following merge window.

Optional:

- First -rc at which the development baseline branch, listed in the overview section, should be considered ready for new submissions.

6.4 Review Cadence

One of the largest sources of contributor angst is how soon to ping after a patchset has been posted without receiving any feedback. In addition to specifying how long to wait before a resubmission this section can also indicate a preferred style of update like, resend the full series, or privately send a reminder email. This section might also list how review works for this code area and methods to get feedback that are not directly from the maintainer.

6.5 Existing profiles

For now, existing maintainer profiles are listed here; we will likely want to do something different in the near future.

6.5.1 Documentation subsystem maintainer entry profile

The documentation “subsystem” is the central coordinating point for the kernel’s documentation and associated infrastructure. It covers the hierarchy under Documentation/ (with the exception of Documentation/devicetree), various utilities under scripts/ and, at least some of the time, LICENSES/.

It’s worth noting, though, that the boundaries of this subsystem are rather fuzzier than normal. Many other subsystem maintainers like to keep control of portions of Documentation/, and many more freely apply changes there when it is convenient. Beyond that, much of the kernel’s

documentation is found in the source as kerneldoc comments; those are usually (but not always) maintained by the relevant subsystem maintainer.

The mailing list for documentation is linux-doc@vger.kernel.org. Patches should be made against the docs-next tree whenever possible.

Submit checklist addendum

When making documentation changes, you should actually build the documentation and ensure that no new errors or warnings have been introduced. Generating HTML documents and looking at the result will help to avoid unsightly misunderstandings about how things will be rendered.

Key cycle dates

Patches can be sent anytime, but response will be slower than usual during the merge window. The docs tree tends to close late before the merge window opens, since the risk of regressions from documentation patches is low.

Review cadence

I am the sole maintainer for the documentation subsystem, and I am doing the work on my own time, so the response to patches will occasionally be slow. I try to always send out a notification when a patch is merged (or when I decide that one cannot be). Do not hesitate to send a ping if you have not heard back within a week of sending a patch.

6.5.2 LIBNVDIMM Maintainer Entry Profile

Overview

The libnvdimm subsystem manages persistent memory across multiple architectures. The mailing list is tracked by patchwork here: https://patchwork.kernel.org/project/linux-nvdimm/list/ ...and that instance is configured to give feedback to submitters on patch acceptance and upstream merge. Patches are merged to either the 'libnvdimm-fixes' or 'libnvdimm-for-next' branch. Those branches are available here: https://git.kernel.org/pub/scm/linux/kernel/git/nvdimm/nvdimm.git/

In general patches can be submitted against the latest -rc; however, if the incoming code change is dependent on other pending changes then the patch should be based on the libnvdimm-for-next branch. However, since persistent memory sits at the intersection of storage and memory there are cases where patches are more suitable to be merged through a Filesystem or the Memory Management tree. When in doubt copy the nvdimm list and the maintainers will help route.

Submissions will be exposed to the kbuild robot for compile regression testing. It helps to get a success notification from that infrastructure before submitting, but it is not required.

Submit Checklist Addendum

There are unit tests for the subsystem via the ndctl utility: https://github.com/pmem/ndctl Those tests need to be passed before the patches go upstream, but not necessarily before initial posting. Contact the list if you need help getting the test environment set up.

ACPI Device Specific Methods (_DSM)

Before patches enabling a new _DSM family will be considered, it must be assigned a format-interface-code from the NVDIMM Sub-team of the ACPI Specification Working Group. In general, the stance of the subsystem is to push back on the proliferation of NVDIMM command sets, so do strongly consider implementing support for an existing command set. See drivers/acpi/nfit/nfit.h for the set of supported command sets.

Key Cycle Dates

New submissions can be sent at any time, but if they intend to hit the next merge window they should be sent before -rc4, and ideally stabilized in the libnvdimm-for-next branch by -rc6. Of course if a patch set requires more than 2 weeks of review, -rc4 is already too late and some patches may require multiple development cycles to review.

Review Cadence

In general, please wait up to one week before pinging for feedback. A private mail reminder is preferred. Alternatively ask for other developers that have Reviewed-by tags for libnvdimm changes to take a look and offer their opinion.

6.5.3 arch/riscv maintenance guidelines for developers

Overview

The RISC-V instruction set architecture is developed in the open: in-progress drafts are available for all to review and to experiment with implementations. New module or extension drafts can change during the development process - sometimes in ways that are incompatible with previous drafts. This flexibility can present a challenge for RISC-V Linux maintenance. Linux maintainers disapprove of churn, and the Linux development process prefers well-reviewed and tested code over experimental code. We wish to extend these same principles to the RISC-V-related code that will be accepted for inclusion in the kernel.

Patchwork

RISC-V has a patchwork instance, where the status of patches can be checked:

> https://patchwork.kernel.org/project/linux-riscv/list/

If your patch does not appear in the default view, the RISC-V maintainers have likely either requested changes, or expect it to be applied to another tree.

Automation runs against this patchwork instance, building/testing patches as they arrive. The automation applies patches against the current HEAD of the RISC-V *for-next* and *fixes* branches, depending on whether the patch has been detected as a fix. Failing those, it will use the RISC-V *master* branch. The exact commit to which a series has been applied will be noted on patchwork. Patches for which any of the checks fail are unlikely to be applied and in most cases will need to be resubmitted.

Submit Checklist Addendum

We'll only accept patches for new modules or extensions if the specifications for those modules or extensions are listed as being unlikely to be incompatibly changed in the future. For specifications from the RISC-V foundation this means "Frozen" or "Ratified", for the UEFI forum specifications this means a published ECR. (Developers may, of course, maintain their own Linux kernel trees that contain code for any draft extensions that they wish.)

Additionally, the RISC-V specification allows implementers to create their own custom extensions. These custom extensions aren't required to go through any review or ratification process by the RISC-V Foundation. To avoid the maintenance complexity and potential performance impact of adding kernel code for implementor-specific RISC-V extensions, we'll only consider patches for extensions that either:

- Have been officially frozen or ratified by the RISC-V Foundation, or
- Have been implemented in hardware that is widely available, per standard Linux practice.

(Implementers, may, of course, maintain their own Linux kernel trees containing code for any custom extensions that they wish.)

6.5.4 Media Subsystem Profile

Overview

The media subsystem covers support for a variety of devices: stream capture, analog and digital TV streams, cameras, remote controllers, HDMI CEC and media pipeline control.

It covers, mainly, the contents of those directories:

- drivers/media
- drivers/staging/media
- Documentation/admin-guide/media
- Documentation/driver-api/media
- Documentation/userspace-api/media

- Documentation/devicetree/bindings/media/[1]
- include/media

Both media userspace and Kernel APIs are documented and the documentation must be kept in sync with the API changes. It means that all patches that add new features to the subsystem must also bring changes to the corresponding API files.

Due to the size and wide scope of the media subsystem, media's maintainership model is to have sub-maintainers that have a broad knowledge of a specific aspect of the subsystem. It is the sub-maintainers' task to review the patches, providing feedback to users if the patches are following the subsystem rules and are properly using the media kernel and userspace APIs.

Patches for the media subsystem must be sent to the media mailing list at linux-media@vger.kernel.org as plain text only e-mail. Emails with HTML will be automatically rejected by the mail server. It could be wise to also copy the sub-maintainer(s).

Media's workflow is heavily based on Patchwork, meaning that, once a patch is submitted, the e-mail will first be accepted by the mailing list server, and, after a while, it should appear at:

- https://patchwork.linuxtv.org/project/linux-media/list/

If it doesn't automatically appear there after a few minutes, then probably something went wrong on your submission. Please check if the email is in plain text[2] only and if your emailer is not mangling whitespaces before complaining or submitting them again.

You can check if the mailing list server accepted your patch, by looking at:

- https://lore.kernel.org/linux-media/

Media maintainers

At the media subsystem, we have a group of senior developers that are responsible for doing the code reviews at the drivers (also known as sub-maintainers), and another senior developer responsible for the subsystem as a whole. For core changes, whenever possible, multiple media maintainers do the review.

The media maintainers that work on specific areas of the subsystem are:

- **Remote Controllers (infrared):**
  Sean Young <sean@mess.org>
- **HDMI CEC:**
  Hans Verkuil <hverkuil@xs4all.nl>
- **Media controller drivers:**
  Laurent Pinchart <laurent.pinchart@ideasonboard.com>
- **ISP, v4l2-async, v4l2-fwnode, v4l2-flash-led-class and Sensor drivers:**
  Sakari Ailus <sakari.ailus@linux.intel.com>
- **V4L2 drivers and core V4L2 frameworks:**
  Hans Verkuil <hverkuil@xs4all.nl>

[1] Device tree bindings are maintained by the OPEN FIRMWARE AND FLATTENED DEVICE TREE BINDINGS maintainers (see the MAINTAINERS file). So, changes there must be reviewed by them before being merged via the media subsystem's development tree.

[2] If your email contains HTML, the mailing list server will simply drop it, without any further notice.

The subsystem maintainer is:
Mauro Carvalho Chehab <mchehab@kernel.org>

Media maintainers may delegate a patch to other media maintainers as needed. On such case, checkpatch's `delegate` field indicates who's currently responsible for reviewing a patch.

Submit Checklist Addendum

Patches that change the Open Firmware/Device Tree bindings must be reviewed by the Device Tree maintainers. So, DT maintainers should be Cc:ed when those are submitted via devicetree@vger.kernel.org mailing list.

There is a set of compliance tools at https://git.linuxtv.org/v4l-utils.git/ that should be used in order to check if the drivers are properly implementing the media APIs:

Type	Tool
V4L2 drivers[3]	`v4l2-compliance`
V4L2 virtual drivers	`contrib/test/test-media`
CEC drivers	`cec-compliance`

Other compilance tools are under development to check other parts of the subsystem.

Those tests need to pass before the patches go upstream.

Also, please notice that we build the Kernel with:

```
make CF=-D__CHECK_ENDIAN__ CONFIG_DEBUG_SECTION_MISMATCH=y C=1 W=1 CHECK=check_
↪script
```

Where the check script is:

```
#!/bin/bash
/devel/smatch/smatch -p=kernel $@ >&2
/devel/sparse/sparse $@ >&2
```

Be sure to not introduce new warnings on your patches without a very good reason.

Style Cleanup Patches

Style cleanups are welcome when they come together with other changes at the files where the style changes will affect.

We may accept pure standalone style cleanups, but they should ideally be one patch for the whole subsystem (if the cleanup is low volume), or at least be grouped per directory. So, for example, if you're doing a big cleanup change set at drivers under drivers/media, please send a single patch for all drivers under drivers/media/pci, another one for drivers/media/usb and so on.

[3] The `v4l2-compliance` also covers the media controller usage inside V4L2 drivers.

Coding Style Addendum

Media development uses `checkpatch.pl` on strict mode to verify the code style, e.g.:

```
$ ./scripts/checkpatch.pl --strict --max-line-length=80
```

In principle, patches should follow the coding style rules, but exceptions are allowed if there are good reasons. On such case, maintainers and reviewers may question about the rationale for not addressing the `checkpatch.pl`.

Please notice that the goal here is to improve code readability. On a few cases, `checkpatch.pl` may actually point to something that would look worse. So, you should use good sense.

Note that addressing one `checkpatch.pl` issue (of any kind) alone may lead to having longer lines than 80 characters per line. While this is not strictly prohibited, efforts should be made towards staying within 80 characters per line. This could include using re-factoring code that leads to less indentation, shorter variable or function names and last but not least, simply wrapping the lines.

In particular, we accept lines with more than 80 columns:

- on strings, as they shouldn't be broken due to line length limits;
- when a function or variable name need to have a big identifier name, which keeps hard to honor the 80 columns limit;
- on arithmetic expressions, when breaking lines makes them harder to read;
- when they avoid a line to end with an open parenthesis or an open bracket.

Key Cycle Dates

New submissions can be sent at any time, but if they intend to hit the next merge window they should be sent before -rc5, and ideally stabilized in the linux-media branch by -rc6.

Review Cadence

Provided that your patch is at https://patchwork.linuxtv.org, it should be sooner or later handled, so you don't need to re-submit a patch.

Except for bug fixes, we don't usually add new patches to the development tree between -rc6 and the next -rc1.

Please notice that the media subsystem is a high traffic one, so it could take a while for us to be able to review your patches. Feel free to ping if you don't get a feedback in a couple of weeks or to ask other developers to publicly add Reviewed-by and, more importantly, `Tested-by:` tags.

Please note that we expect a detailed description for `Tested-by:`, identifying what boards were used at the test and what it was tested.

6.5.5 Acceptance criteria for vfio-pci device specific driver variants

Overview

The vfio-pci driver exists as a device agnostic driver using the system IOMMU and relying on the robustness of platform fault handling to provide isolated device access to userspace. While the vfio-pci driver does include some device specific support, further extensions for yet more advanced device specific features are not sustainable. The vfio-pci driver has therefore split out vfio-pci-core as a library that may be reused to implement features requiring device specific knowledge, ex. saving and loading device state for the purposes of supporting migration.

In support of such features, it's expected that some device specific variants may interact with parent devices (ex. SR-IOV PF in support of a user assigned VF) or other extensions that may not be otherwise accessible via the vfio-pci base driver. Authors of such drivers should be diligent not to create exploitable interfaces via these interactions or allow unchecked userspace data to have an effect beyond the scope of the assigned device.

New driver submissions are therefore requested to have approval via sign-off/ack/review/etc for any interactions with parent drivers. Additionally, drivers should make an attempt to provide sufficient documentation for reviewers to understand the device specific extensions, for example in the case of migration data, how is the device state composed and consumed, which portions are not otherwise available to the user via vfio-pci, what safeguards exist to validate the data, etc. To that extent, authors should additionally expect to require reviews from at least one of the listed reviewers, in addition to the overall vfio maintainer.

6.5.6 Linux NVMe feature and and quirk policy

This file explains the policy used to decide what is supported by the Linux NVMe driver and what is not.

Introduction

NVM Express is an open collection of standards and information.

The Linux NVMe host driver in drivers/nvme/host/ supports devices implementing the NVM Express (NVMe) family of specifications, which currently consists of a number of documents:

- the NVMe Base specification
- various Command Set specifications (e.g. NVM Command Set)
- various Transport specifications (e.g. PCIe, Fibre Channel, RDMA, TCP)
- the NVMe Management Interface specification

See https://nvmexpress.org/developers/ for the NVMe specifications.

Supported features

NVMe is a large suite of specifications, and contains features that are only useful or suitable for specific use-cases. It is important to note that Linux does not aim to implement every feature in the specification. Every additional feature implemented introduces more code, more maintenance and potentially more bugs. Hence there is an inherent tradeoff between functionality and maintainability of the NVMe host driver.

Any feature implemented in the Linux NVMe host driver must support the following requirements:

1. The feature is specified in a release version of an official NVMe specification, or in a ratified Technical Proposal (TP) that is available on NVMe website. Or if it is not directly related to the on-wire protocol, does not contradict any of the NVMe specifications.
2. Does not conflict with the Linux architecture, nor the design of the NVMe host driver.
3. Has a clear, indisputable value-proposition and a wide consensus across the community.

Vendor specific extensions are generally not supported in the NVMe host driver.

It is strongly recommended to work with the Linux NVMe and block layer maintainers and get feedback on specification changes that are intended to be used by the Linux NVMe host driver in order to avoid conflict at a later stage.

Quirks

Sometimes implementations of open standards fail to correctly implement parts of the standards. Linux uses identifier-based quirks to work around such implementation bugs. The intent of quirks is to deal with widely available hardware, usually consumer, which Linux users can't use without these quirks. Typically these implementations are not or only superficially tested with Linux by the hardware manufacturer.

The Linux NVMe maintainers decide ad hoc whether to quirk implementations based on the impact of the problem to Linux users and how it impacts maintainability of the driver. In general quirks are a last resort, if no firmware updates or other workarounds are available from the vendor.

Quirks will not be added to the Linux kernel for hardware that isn't available on the mass market. Hardware that fails qualification for enterprise Linux distributions, ChromeOS, Android or other consumers of the Linux kernel should be fixed before it is shipped instead of relying on Linux quirks.

6.5.7 XFS Maintainer Entry Profile

Overview

XFS is a well known high-performance filesystem in the Linux kernel. The aim of this project is to provide and maintain a robust and performant filesystem.

Patches are generally merged to the for-next branch of the appropriate git repository. After a testing period, the for-next branch is merged to the master branch.

Kernel code are merged to the xfs-linux tree[0]. Userspace code are merged to the xfsprogs tree[1]. Test cases are merged to the xfstests tree[2]. Ondisk format documentation are merged to the xfs-documentation tree[3].

All patchsets involving XFS *must* be cc'd in their entirety to the mailing list linux-xfs@vger.kernel.org.

Roles

There are eight key roles in the XFS project. A person can take on multiple roles, and a role can be filled by multiple people. Anyone taking on a role is advised to check in with themselves and others on a regular basis about burnout.

- **Outside Contributor**: Anyone who sends a patch but is not involved in the XFS project on a regular basis. These folks are usually people who work on other filesystems or elsewhere in the kernel community.
- **Developer**: Someone who is familiar with the XFS codebase enough to write new code, documentation, and tests.

  Developers can often be found in the IRC channel mentioned by the `C:` entry in the kernel MAINTAINERS file.
- **Senior Developer**: A developer who is very familiar with at least some part of the XFS codebase and/or other subsystems in the kernel. These people collectively decide the long term goals of the project and nudge the community in that direction. They should help prioritize development and review work for each release cycle.

  Senior developers tend to be more active participants in the IRC channel.
- **Reviewer**: Someone (most likely also a developer) who reads code submissions to decide:

  0. Is the idea behind the contribution sound?
  1. Does the idea fit the goals of the project?
  2. Is the contribution designed correctly?
  3. Is the contribution polished?
  4. Can the contribution be tested effectively?

  Reviewers should identify themselves with an `R:` entry in the kernel and fstests MAINTAINERS files.
- **Testing Lead**: This person is responsible for setting the test coverage goals of the project, negotiating with developers to decide on new tests for new features, and making sure that developers and release managers execute on the testing.

  The testing lead should identify themselves with an `M:` entry in the XFS section of the fstests MAINTAINERS file.
- **Bug Triager**: Someone who examines incoming bug reports in just enough detail to identify the person to whom the report should be forwarded.

  The bug triagers should identify themselves with a `B:` entry in the kernel MAINTAINERS file.

- **Release Manager**: This person merges reviewed patchsets into an integration branch, tests the result locally, pushes the branch to a public git repository, and sends pull requests further upstream. The release manager is not expected to work on new feature patchsets. If a developer and a reviewer fail to reach a resolution on some point, the release manager must have the ability to intervene to try to drive a resolution.

  The release manager should identify themselves with an `M:` entry in the kernel MAINTAINERS file.

- **Community Manager**: This person calls and moderates meetings of as many XFS participants as they can get when mailing list discussions prove insufficient for collective decisionmaking. They may also serve as liaison between managers of the organizations sponsoring work on any part of XFS.

- **LTS Maintainer**: Someone who backports and tests bug fixes from uptream to the LTS kernels. There tend to be six separate LTS trees at any given time.

  The maintainer for a given LTS release should identify themselves with an `M:` entry in the MAINTAINERS file for that LTS tree. Unmaintained LTS kernels should be marked with status `S: Orphan` in that same file.

Submission Checklist Addendum

Please follow these additional rules when submitting to XFS:

- Patches affecting only the filesystem itself should be based against the latest -rc or the for-next branch. These patches will be merged back to the for-next branch.
- Authors of patches touching other subsystems need to coordinate with the maintainers of XFS and the relevant subsystems to decide how to proceed with a merge.
- Any patchset changing XFS should be cc'd in its entirety to linux-xfs. Do not send partial patchsets; that makes analysis of the broader context of the changes unnecessarily difficult.
- Anyone making kernel changes that have corresponding changes to the userspace utilities should send the userspace changes as separate patchsets immediately after the kernel patchsets.
- Authors of bug fix patches are expected to use fstests[2] to perform an A/B test of the patch to determine that there are no regressions. When possible, a new regression test case should be written for fstests.
- Authors of new feature patchsets must ensure that fstests will have appropriate functional and input corner-case test cases for the new feature.
- When implementing a new feature, it is strongly suggested that the developers write a design document to answer the following questions:
  - **What** problem is this trying to solve?
  - **Who** will benefit from this solution, and **where** will they access it?
  - **How** will this new feature work? This should touch on major data structures and algorithms supporting the solution at a higher level than code comments.
  - **What** userspace interfaces are necessary to build off of the new features?

  - **How** will this work be tested to ensure that it solves the problems laid out in the design document without causing new problems?

  The design document should be committed in the kernel documentation directory. It may be omitted if the feature is already well known to the community.

- Patchsets for the new tests should be submitted as separate patchsets immediately after the kernel and userspace code patchsets.
- Changes to the on-disk format of XFS must be described in the ondisk format document[3] and submitted as a patchset after the fstests patchsets.
- Patchsets implementing bug fixes and further code cleanups should put the bug fixes at the beginning of the series to ease backporting.

Key Release Cycle Dates

Bug fixes may be sent at any time, though the release manager may decide to defer a patch when the next merge window is close.

Code submissions targeting the next merge window should be sent between -rc1 and -rc6. This gives the community time to review the changes, to suggest other changes, and for the author to retest those changes.

Code submissions also requiring changes to fs/iomap and targeting the next merge window should be sent between -rc1 and -rc4. This allows the broader kernel community adequate time to test the infrastructure changes.

Review Cadence

In general, please wait at least one week before pinging for feedback. To find reviewers, either consult the MAINTAINERS file, or ask developers that have Reviewed-by tags for XFS changes to take a look and offer their opinion.

References

[0] https://git.kernel.org/pub/scm/fs/xfs/xfs-linux.git/
[1] https://git.kernel.org/pub/scm/fs/xfs/xfsprogs-dev.git/
[2] https://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git/
[3] https://git.kernel.org/pub/scm/fs/xfs/xfs-documentation.git/

CHAPTER

SEVEN

MODIFYING PATCHES

If you are a subsystem or branch maintainer, sometimes you need to slightly modify patches you receive in order to merge them, because the code is not exactly the same in your tree and the submitters'. If you stick strictly to rule (c) of the developers certificate of origin, you should ask the submitter to rediff, but this is a totally counter-productive waste of time and energy. Rule (b) allows you to adjust the code, but then it is very impolite to change one submitters code and make him endorse your bugs. To solve this problem, it is recommended that you add a line between the last Signed-off-by header and yours, indicating the nature of your changes. While there is nothing mandatory about this, it seems like prepending the description with your mail and/or name, all enclosed in square brackets, is noticeable enough to make it obvious that you are responsible for last-minute changes. Example:

```
Signed-off-by: Random J Developer <random@developer.example.org>
[lucky@maintainer.example.org: struct foo moved from foo.c to foo.h]
Signed-off-by: Lucky K Maintainer <lucky@maintainer.example.org>
```

This practice is particularly helpful if you maintain a stable branch and want at the same time to credit the author, track changes, merge the fix, and protect the submitter from complaints. Note that under no circumstances can you change the author's identity (the From header), as it is the one which appears in the changelog.

Special note to back-porters: It seems to be a common and useful practice to insert an indication of the origin of a patch at the top of the commit message (just after the subject line) to facilitate tracking. For instance, here's what we see in a 3.x-stable release:

```
Date:   Tue Oct 7 07:26:38 2014 -0400

  libata: Un-break ATA blacklist

  commit 1c40279960bcd7d52dbdf1d466b20d24b99176c8 upstream.
```

And here's what might appear in an older kernel once a patch is backported:

```
Date:   Tue May 13 22:12:27 2008 +0200

    wireless, airo: waitbusy() won't delay

    [backport of 2.6 commit b7acbdfbd1f277c1eb23f344f899cfa4cd0bf36a]
```

Whatever the format, this information provides a valuable help to people tracking your trees, and to people trying to troubleshoot bugs in your tree.