Posting patches

Sooner or later, the time comes when your work is ready to be presented to the community for review and, eventually, inclusion into the mainline kernel. Unsurprisingly, the kernel development community has evolved a set of conventions and procedures which are used in the posting of patches; following them will make life much easier for everybody involved. This document will attempt to cover these expectations in reasonable detail; more information can also be found in the files :ref:`Documentation/process/submitting-patches.rst <submittingpatches>`, :ref:`Documentation/process/submitting-drivers.rst <submittingdrivers>` and :ref:`Documentation/process/submit-checklist.rst <submitchecklist>`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]5.Posting.rst`, line 6); ***backlink***

Unknown interpreted text role "ref".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]5.Posting.rst`, line 6); ***backlink***

Unknown interpreted text role "ref".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]5.Posting.rst`, line 6); ***backlink***

Unknown interpreted text role "ref".

When to post

There is a constant temptation to avoid posting patches before they are completely "ready." For simple patches, that is not a problem. If the work being done is complex, though, there is a lot to be gained by getting feedback from the community before the work is complete. So you should consider posting in-progress work, or even making a git tree available so that interested developers can catch up with your work at any time.

When posting code which is not yet considered ready for inclusion, it is a good idea to say so in the posting itself. Also mention any major work which remains to be done and any known problems. Fewer people will look at patches which are known to be half-baked, but those who do will come in with the idea that they can help you drive the work in the right direction.

Before creating patches

There are a number of things which should be done before you consider sending patches to the development community. These include:

- Test the code to the extent that you can. Make use of the kernel's debugging tools, ensure that the kernel will build with all reasonable combinations of configuration options, use cross-compilers to build for different architectures, etc.
- Make sure your code is compliant with the kernel coding style guidelines.
- Does your change have performance implications? If so, you should run benchmarks showing what the impact (or benefit) of your change is; a summary of the results should be included with the patch.
- Be sure that you have the right to post the code. If this work was done for an employer, the employer likely has a right to the work and must be agreeable with its release under the GPL.

As a general rule, putting in some extra thought before posting code almost always pays back the effort in short order.

Patch preparation

The preparation of patches for posting can be a surprising amount of work, but, once again, attempting to save time here is not generally advisable even in the short term.

Patches must be prepared against a specific version of the kernel. As a general rule, a patch should be based on the current mainline as found in Linus's git tree. When basing on mainline, start with a well-known release point - a stable or -rc release - rather than branching off the mainline at an arbitrary spot.

It may become necessary to make versions against -mm, linux-next, or a subsystem tree, though, to facilitate wider testing and review. Depending on the area of your patch and what is going on elsewhere, basing a patch against these other trees can require a significant amount of work resolving conflicts and dealing with API changes.

Only the most simple changes should be formatted as a single patch; everything else should be made as a logical series of changes. Splitting up patches is a bit of an art; some developers spend a long time figuring out how to do it in the way that the community expects. There are a few rules of thumb, however, which can help considerably:

- The patch series you post will almost certainly not be the series of changes found in your working revision control system. Instead, the changes you have made need to be considered in their final form, then split apart in ways which make sense. The developers are interested in discrete, self-contained changes, not the path you took to get to those changes.
- Each logically independent change should be formatted as a separate patch. These changes can be small ("add a field to this structure") or large (adding a significant new driver, for example), but they should be conceptually small and amenable to a one-line description. Each patch should make a specific change which can be reviewed on its own and verified to do what it says it does.
- As a way of restating the guideline above: do not mix different types of changes in the same patch. If a single patch fixes a critical security bug, rearranges a few structures, and reformats the code, there is a good chance that it will be passed over and the important fix will be lost.
- Each patch should yield a kernel which builds and runs properly; if your patch series is interrupted in the middle, the result should still be a working kernel. Partial application of a patch series is a common scenario when the "git bisect" tool is used to find regressions; if the result is a broken kernel, you will make life harder for developers and users who are engaging in the noble work of tracking down problems.
- Do not overdo it, though. One developer once posted a set of edits to a single file as 500 separate patches - an act which did not make him the most popular person on the kernel mailing list. A single patch can be reasonably large as long as it still contains a single *logical* change.
- It can be tempting to add a whole new infrastructure with a series of patches, but to leave that infrastructure unused until the final patch in the series enables the whole thing. This temptation should be avoided if possible; if that series adds regressions, bisection will finger the last patch as the one which caused the problem, even though the real bug is elsewhere. Whenever possible, a patch which adds new code should make that code active immediately.

Working to create the perfect patch series can be a frustrating process which takes quite a bit of time and thought after the "real work" has been done. When done properly, though, it is time well spent.

Patch formatting and changelogs

So now you have a perfect series of patches for posting, but the work is not done quite yet. Each patch needs to be formatted into a message which quickly and clearly communicates its purpose to the rest of the world. To that end, each patch will be composed of the following:

- An optional "From" line naming the author of the patch. This line is only necessary if you are passing on somebody else's patch via email, but it never hurts to add it when in doubt.
- A one-line description of what the patch does. This message should be enough for a reader who sees it with no other context to figure out the scope of the patch; it is the line that will show up in the "short form" changelogs. This message is usually formatted with the relevant subsystem name first, followed by the purpose of the patch. For example:

  ```
  gpio: fix build on CONFIG_GPIO_SYSFS=n
  ```

- A blank line followed by a detailed description of the contents of the patch. This description can be as long as is required; it should say what the patch does and why it should be applied to the kernel.
- One or more tag lines, with, at a minimum, one Signed-off-by: line from the author of the patch. Tags will be described in more detail below.

The items above, together, form the changelog for the patch. Writing good changelogs is a crucial but often-neglected art; it's worth spending another moment discussing this issue. When writing a changelog, you should bear in mind that a number of different people will be reading your words. These include subsystem maintainers and reviewers who need to decide whether the patch should be included, distributors and other maintainers trying to decide whether a patch should be backported to other kernels, bug hunters wondering whether the patch is responsible for a problem they are chasing, users who want to know how the kernel has changed, and more. A good changelog conveys the needed information to all of these people in the most direct and concise way possible.

To that end, the summary line should describe the effects of and motivation for the change as well as possible given the one-line constraint. The detailed description can then amplify on those topics and provide any needed additional information. If the patch fixes a bug, cite the commit which introduced the bug if possible (and please provide both the commit ID and the title when citing commits). If a problem is associated with specific log or compiler output, include that output to help others searching for a solution to the same problem. If the change is meant to support other changes coming in later patch, say so. If internal APIs are changed, detail those changes and how other developers should respond. In general, the more you can put yourself into the shoes of everybody who will be reading your changelog, the better that changelog (and the kernel as a whole) will be.

Needless to say, the changelog should be the text used when committing the change to a revision control system. It will be followed by:

- The patch itself, in the unified ("-u") patch format. Using the "-p" option to diff will associate function names with changes, making the resulting patch easier for others to read.

You should avoid including changes to irrelevant files (those generated by the build process, for example, or editor backup files) in the patch. The file "dontdiff" in the Documentation directory can help in this regard; pass it to diff with the "-X" option.

The tags already briefly mentioned above are used to provide insights how the patch came into being. They are described in detail in the :ref:`Documentation/process/submitting-patches.rst <submittingpatches>` document; what follows here is a brief summary.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]5.Posting.rst`, line 200); ***backlink***

Unknown interpreted text role "ref".

One tag is used to refer to earlier commits which introduced problems fixed by the patch:

```
Fixes: 1f2e3d4c5b6a ("The first line of the commit specified by the first 12 characters of its SHA-1 ID"
```

Another tag is used for linking web pages with additional backgrounds or details, for example a report about a bug fixed by the patch or a document with a specification implemented by the patch:

```
Link: https://example.com/somewhere.html  optional-other-stuff
```

Many maintainers when applying a patch also add this tag to link to the latest public review posting of the patch; often this is automatically done by tools like b4 or a git hook like the one described in 'Documentation/maintainer/configure-git.rst'.

A third kind of tag is used to document who was involved in the development of the patch. Each of these uses this format:

```
tag: Full Name <email address>  optional-other-stuff
```

The tags in common use are:

- Signed-off-by: this is a developer's certification that he or she has the right to submit the patch for inclusion into the kernel. It is an agreement to the Developer's Certificate of Origin, the full text of which can be found in :ref:`Documentation/process/submitting-patches.rst <submittingpatches>` Code without a proper signoff cannot be merged into the mainline.

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]5.Posting.rst`, line 228); ***backlink***

  Unknown interpreted text role "ref".

- Co-developed-by: states that the patch was co-created by several developers; it is a used to give attribution to co-authors (in addition to the author attributed by the From: tag) when multiple people work on a single patch. Every Co-developed-by: must be immediately followed by a Signed-off-by: of the associated co-author. Details and examples can be found in :ref:`Documentation/process/submitting-patches.rst <submittingpatches>`.

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]5.Posting.rst`, line 234); ***backlink***

  Unknown interpreted text role "ref".

- Acked-by: indicates an agreement by another developer (often a maintainer of the relevant code) that the patch is appropriate for inclusion into the kernel.
- Tested-by: states that the named person has tested the patch and found it to work.
- Reviewed-by: the named developer has reviewed the patch for correctness; see the reviewer's statement in :ref:`Documentation/process/submitting-patches.rst <submittingpatches>` for more detail.

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]5.Posting.rst`, line 248); ***backlink***

  Unknown interpreted text role "ref".

- Reported-by: names a user who reported a problem which is fixed by this patch; this tag is used to give credit to the (often underappreciated) people who test our code and let us know when things do not work correctly.
- Cc: the named person received a copy of the patch and had the opportunity to comment on it.

Be careful in the addition of tags to your patches: only Cc: is appropriate for addition without the explicit permission of the person named.

Sending the patch

Before you mail your patches, there are a couple of other things you should take care of:

- Are you sure that your mailer will not corrupt the patches? Patches which have had gratuitous white-space changes or line wrapping performed by the mail client will not apply at the other end, and often will not be examined in any detail. If there is any doubt at all, mail the patch to yourself and convince yourself that it shows up intact.

  :ref:`Documentation/process/email-clients.rst <email_clients>` has some helpful hints on making specific mail clients work for sending patches.

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]5.Posting.rst`, line 276); ***backlink***

  Unknown interpreted text role "ref".

- Are you sure your patch is free of silly mistakes? You should always run patches through scripts/checkpatch.pl and address the complaints it comes up with. Please bear in mind that checkpatch.pl, while being the embodiment of a fair amount of thought about what kernel patches should look like, is not smarter than you. If fixing a checkpatch.pl complaint would make the code worse, don't do it.

Patches should always be sent as plain text. Please do not send them as attachments; that makes it much harder for reviewers to quote sections of the patch in their replies. Instead, just put the patch directly into your message.

When mailing patches, it is important to send copies to anybody who might be interested in it. Unlike some other projects, the kernel encourages people to err on the side of sending too many copies; don't assume that the relevant people will see your posting on the mailing lists. In particular, copies should go to:

- The maintainer(s) of the affected subsystem(s). As described earlier, the MAINTAINERS file is the first place to look for these people.
- Other developers who have been working in the same area - especially those who might be working there now. Using git to see who else has modified the files you are working on can be helpful.
- If you are responding to a bug report or a feature request, copy the original poster as well.
- Send a copy to the relevant mailing list, or, if nothing else applies, the linux-kernel list.
- If you are fixing a bug, think about whether the fix should go into the next stable update. If so, stable@vger.kernel.org should get a copy of the patch. Also add a "Cc: stable@vger.kernel.org" to the tags within the patch itself; that will cause the stable team to get a notification when your fix goes into the mainline.

When selecting recipients for a patch, it is good to have an idea of who you think will eventually accept the patch and get it merged. While it is possible to send patches directly to Linus Torvalds and have him merge them, things are not normally done that way. Linus is busy, and there are subsystem maintainers who watch over specific parts of the kernel. Usually you will be wanting that maintainer to merge your patches. If there is no obvious maintainer, Andrew Morton is often the patch target of last resort.

Patches need good subject lines. The canonical format for a patch line is something like:

```
[PATCH nn/mm] subsys: one-line description of the patch
```

where "nn" is the ordinal number of the patch, "mm" is the total number of patches in the series, and "subsys" is the name of the affected subsystem. Clearly, nn/mm can be omitted for a single, standalone patch.

If you have a significant series of patches, it is customary to send an introductory description as part zero. This convention is not universally followed though; if you use it, remember that information in the introduction does not make it into the kernel changelogs. So please ensure that the patches, themselves, have complete changelog information.

In general, the second and following parts of a multi-part patch should be sent as a reply to the first part so that they all thread together at the receiving end. Tools like git and quilt have commands to mail out a set of patches with the proper threading. If you have a long series, though, and are using git, please stay away from the --chain-reply-to option to avoid creating exceptionally deep nesting.