HOW TO do Linux kernel development

This is the be-all, end-all document on this topic. It contains instructions on how to become a Linux kernel developer and how to learn to work with the Linux kernel development community. It tries to not contain anything related to the technical aspects of kernel programming, but will help point you in the right direction for that.

If anything in this document becomes out of date, please send in patches to the maintainer of this file, who is listed at the bottom of the document.

Introduction

So, you want to learn how to become a Linux kernel developer? Or you have been told by your manager, “Go write a Linux driver for this device.” This document’s goal is to teach you everything you need to know to achieve this by describing the process you need to go through, and hints on how to work with the community. It will also try to explain some of the reasons why the community works like it does.

The kernel is written mostly in C, with some architecture-dependent parts written in assembly. A good understanding of C is required for kernel development. Assembly (any architecture) is not required unless you plan to do low-level development for that architecture. Though they are not a good substitute for a solid C education and/or years of experience, the following books are good for, if anything, reference:

- “The C Programming Language” by Kernighan and Ritchie [Prentice Hall]
- “Practical C Programming” by Steve Oualline [O’Reilly]
- “C: A Reference Manual” by Harbison and Steele [Prentice Hall]

The kernel is written using GNU C and the GNU toolchain. While it adheres to the ISO C89 standard, it uses a number of extensions that are not featured in the standard. The kernel is a freestanding C environment, with no reliance on the standard C library, so some portions of the C standard are not supported. Arbitrary long long divisions and floating point are not allowed. It can sometimes be difficult to understand the assumptions the kernel has on the toolchain and the extensions that it uses, and unfortunately there is no definitive reference for them. Please check the gcc info pages (info gcc) for some information on them.

Please remember that you are trying to learn how to work with the existing development community. It is a diverse group of people, with high standards for coding, style and procedure. These standards have been created over time based on what they have found to work best for such a large and geographically dispersed team. Try to learn as much as possible about these standards ahead of time, as they are well documented; do not expect people to adapt to you or your company's way of doing things.

Legal Issues

The Linux kernel source code is released under the GPL. Please see the file, COPYING, in the main directory of the source tree, for details on the license. If you have further questions about the license, please contact a lawyer, and do not ask on the Linux kernel mailing list. The people on the mailing lists are not lawyers, and you should not rely on their statements on legal matters.

For common questions and answers about the GPL, please see:

> https://www.gnu.org/licenses/gpl-faq.html

Documentation

The Linux kernel source tree has a large range of documents that are invaluable for learning how to interact with the kernel community. When new features are added to the

kernel, it is recommended that new documentation files are also added which explain how to use the feature. When a kernel change causes the interface that the kernel exposes to userspace to change, it is recommended that you send the information or a patch to the manual pages explaining the change to the manual pages maintainer at mtk.manpages@gmail.com, and CC the list linux-api@vger.kernel.org.

Here is a list of files that are in the kernel source tree that are required reading:

README

This file gives a short background on the Linux kernel and describes what is necessary to do to configure and build the kernel. People who are new to the kernel should start here.

Documentation/process/changes.rst

This file gives a list of the minimum levels of various software packages that are necessary to build and run the kernel successfully.

Documentation/process/coding-style.rst

This describes the Linux kernel coding style, and some of the rationale behind it. All new code is expected to follow the guidelines in this document. Most

maintainers will only accept patches if these rules are followed, and many people will only review code if it is in the proper style.

Documentation/process/submitting-patches.rst and Documentation/process/submitting-drivers.rst

These files describe in explicit detail how to successfully create and send a patch, including (but not limited to):

- Email contents
- Email format
- Who to send it to

Following these rules will not guarantee success (as all patches are subject to scrutiny for content and style), but not following them will almost always prevent it.

Other excellent descriptions of how to create patches properly are:

“The Perfect Patch”

https://www.ozlabs.org/~akpm/stuff/tpp.txt

“Linux kernel patch submission format”

http://linux.yyz.us/patch-format.html

Documentation/process/stable-api-nonsense.rst

This file describes the rationale behind the conscious decision to not have a stable API within the kernel, including things like:

- Subsystem shim-layers (for compatibility?)
- Driver portability between Operating Systems.

- Mitigating rapid change within the kernel source tree (or preventing rapid change)

This document is crucial for understanding the Linux development philosophy and is very important for people moving to Linux from development on other Operating Systems.

Documentation/admin-guide/security-bugs.rst

If you feel you have found a security problem in the Linux kernel, please follow the steps in this document to help notify the kernel developers, and help solve the issue.

Documentation/process/management-style.rst

This document describes how Linux kernel maintainers operate and the shared ethos behind their methodologies. This is important reading for anyone new to kernel development (or anyone simply curious about it), as it resolves a lot of common misconceptions and confusion about the unique behavior of kernel maintainers.

Documentation/process/stable-kernel-rules.rst

> This file describes the rules on how the stable kernel releases happen, and what to do if you want to get a change into one of these releases.

Documentation/process/kernel-docs.rst

> A list of external documentation that pertains to kernel development. Please consult this list if you do not find what you are looking for within the in-kernel documentation.

Documentation/process/applying-patches.rst

> A good introduction describing exactly what a patch is and how to apply it to the different development branches of the kernel.

The kernel also has a large number of documents that can be automatically generated from the source code itself or from ReStructuredText markups (ReST), like this one. This includes a full description of the in-kernel API, and rules on how to handle locking properly.

All such documents can be generated as PDF or HTML by running:

```
make pdfdocs
make htmldocs
```

respectively from the main kernel source directory.

The documents that uses ReST markup will be generated at Documentation/output. They can also be generated on LaTeX and ePub formats with:

```
make latexdocs
make epubdocs
```

Becoming A Kernel Developer

If you do not know anything about Linux kernel development, you should look at the Linux KernelNewbies project:

> https://kernelnewbies.org

It consists of a helpful mailing list where you can ask almost any type of basic kernel development question (make sure to search the archives first, before asking something that has already been answered in the past.) It also has an IRC channel that you can use to ask questions in real-time, and a lot of helpful documentation that is useful for learning about Linux kernel development.

The website has basic information about code organization, subsystems, and current projects (both in-tree and out-of-tree). It also describes some basic logistical information, like how to compile a kernel and apply a patch.

If you do not know where you want to start, but you want to look for some task to start doing to join into the kernel development community, go to the Linux Kernel Janitor's project:

https://kernelnewbies.org/KernelJanitors

It is a great place to start. It describes a list of relatively simple problems that need to be cleaned up and fixed within the Linux kernel source tree. Working with the developers in charge of this project, you will learn the basics of getting your patch into the Linux kernel tree, and possibly be pointed in the direction of what to go work on next, if you do not already have an idea.

If you already have a chunk of code that you want to put into the kernel tree, but need some help getting it in the proper form, the kernel-mentors project was created to help you out with this. It is a mailing list, and can be found at:

https://selenic.com/mailman/listinfo/kernel-mentors

Before making any actual modifications to the Linux kernel code, it is imperative to understand how the code in question works. For this purpose, nothing is better than reading through it directly (most tricky bits are commented well), perhaps even with the help of specialized tools. One such tool that is particularly recommended is the Linux Cross-Reference project, which is able to present source code in a self-referential, indexed webpage format. An excellent up-to-date repository of the kernel code may be found at:

http://lxr.free-electrons.com/

The development process

Linux kernel development process currently consists of a few different main kernel “branches” and lots of different subsystem-specific kernel branches. These different branches are:

- main 4.x kernel tree

- 4.x.y -stable kernel tree
- 4.x -git kernel patches
- subsystem specific kernel trees and patches
- the 4.x -next kernel tree for integration tests

4.x kernel tree

4.x kernels are maintained by Linus Torvalds, and can be found on https://kernel.org in the pub/linux/kernel/v4.x/ directory. Its development process is as follows:

- As soon as a new kernel is released a two weeks window is open, during this period of time maintainers can submit big diffs to Linus, usually the patches that have already been included in the -next kernel for a few weeks. The preferred way to submit big changes is using git (the kernel's source management tool, more information can be found at https://git-scm.com/) but plain patches are also just fine.
- After two weeks a -rc1 kernel is released and the focus is on making the new kernel as rock solid as possible. Most of the patches at this point should fix a regression. Bugs that have always existed are not regressions, so only push these kinds of fixes if they are important. Please note that a whole new driver (or filesystem) might be accepted after -rc1 because there is no risk of causing regressions with such a change as long as the change is self-contained and does not affect areas outside of the code that is being added. git can be used to send patches to Linus after -rc1 is released, but the patches need to also be sent to a public mailing list for review.

- A new -rc is released whenever Linus deems the current git tree to be in a reasonably sane state adequate for testing. The goal is to release a new -rc kernel every week.
- Process continues until the kernel is considered “ready”, the process should last around 6 weeks.

It is worth mentioning what Andrew Morton wrote on the linux-kernel mailing list about kernel releases:

> *“Nobody knows when a kernel will be released, because it’s released according to perceived bug status, not according to a preconceived timeline.”*

4.x.y -stable kernel tree

Kernels with 3-part versions are -stable kernels. They contain relatively small and critical fixes for security problems or significant regressions discovered in a given 4.x kernel.

This is the recommended branch for users who want the most recent stable kernel and are not interested in helping test development/experimental versions.

If no 4.x.y kernel is available, then the highest numbered 4.x kernel is the current stable kernel.

4.x.y are maintained by the “stable” team <stable@vger.kernel.org>, and are released as needs dictate. The normal release period is approximately two weeks, but it can be longer if there are no pressing problems. A security-related problem, instead, can cause a release to happen almost instantly.

The file Documentation/process/stable-kernel-rules.rst in the kernel tree documents what kinds of changes are acceptable for the -stable tree, and how the release process works.

4.x -git patches

These are daily snapshots of Linus' kernel tree which are managed in a git repository (hence the name.) These patches are usually released daily and represent the current state of Linus' tree. They are more experimental than -rc kernels since they are generated automatically without even a cursory glance to see if they are sane.

Subsystem Specific kernel trees and patches

The maintainers of the various kernel subsystems — and also many kernel subsystem developers — expose their current state of development in source repositories. That way, others can see what is happening in the different areas of the kernel. In areas where development is rapid, a developer may be asked to base his submissions onto such a subsystem kernel tree so that conflicts between the submission and other already ongoing work are avoided.

Most of these repositories are git trees, but there are also other SCMs in use, or patch queues being published as quilt series. Addresses of these subsystem repositories are listed in the MAINTAINERS file. Many of them can be browsed at https://git.kernel.org/.

Before a proposed patch is committed to such a subsystem tree, it is subject to review which primarily happens on mailing lists (see the respective section below). For several kernel subsystems, this review process is tracked with the tool patchwork. Patchwork offers a web interface which shows patch postings, any comments on a patch or revisions to it, and maintainers can mark patches as under review, accepted, or rejected. Most of these patchwork sites are listed athttps://patchwork.kernel.org/.

4.x -next kernel tree for integration tests

Before updates from subsystem trees are merged into the mainline 4.x tree, they need to be integration-tested. For this purpose, a special testing repository exists into which virtually all subsystem trees are pulled on an almost daily basis:

https://git.kernel.org/?p=linux/kernel/git/next/linux-next.git

This way, the -next kernel gives a summary outlook onto what will be expected to go into the mainline kernel at the next merge period. Adventurous testers are very welcome to runtime-test the -next kernel.

Bug Reporting

https://bugzilla.kernel.org is where the Linux kernel developers track kernel bugs. Users are encouraged to report all bugs that they find in this tool. For details on how to use the kernel bugzilla, please see:

https://bugzilla.kernel.org/page.cgi?id=faq.html

The file admin-guide/reporting-bugs.rst in the main kernel source directory has a good template for how to report a possible kernel bug, and details what kind of information is needed by the kernel developers to help track down the problem.

Managing bug reports

One of the best ways to put into practice your hacking skills is by fixing bugs reported by other people. Not only you will help to make the kernel more stable, you'll learn to fix real world problems and you will improve your skills, and other developers will be aware of your presence. Fixing bugs is one of the best ways to get merits among other developers, because not many people like wasting time fixing other people's bugs.

To work in the already reported bug reports, go to https://bugzilla.kernel.org. If you want to be advised of the future bug reports, you can subscribe to the bugme-new mailing list

(only new bug reports are mailed here) or to the bugme-janitor mailing list (every change in the bugzilla is mailed here)

https://lists.linux-foundation.org/mailman/listinfo/bugme-new

https://lists.linux-foundation.org/mailman/listinfo/bugme-janitors

Mailing lists

As some of the above documents describe, the majority of the core kernel developers participate on the Linux Kernel Mailing list. Details on how to subscribe and unsubscribe from the list can be found at:

http://vger.kernel.org/vger-lists.html#linux-kernel

There are archives of the mailing list on the web in many different places. Use a search engine to find these archives. For example:

http://dir.gmane.org/gmane.linux.kernel

It is highly recommended that you search the archives about the topic you want to bring up, before you post it to the list. A lot of things already discussed in detail are only recorded at the mailing list archives.

Most of the individual kernel subsystems also have their own separate mailing list where they do their development efforts. See the MAINTAINERS file for a list of what these lists are for the different groups.

Many of the lists are hosted on kernel.org. Information on them can be found at:

http://vger.kernel.org/vger-lists.html

Please remember to follow good behavioral habits when using the lists. Though a bit cheesy, the following URL has some simple guidelines for interacting with the list (or any list):

http://www.albion.com/netiquette/

If multiple people respond to your mail, the CC: list of recipients may get pretty large. Don't remove anybody from the CC: list without a good reason, or don't reply only to the list address. Get used to receiving the mail twice, one from the sender and the one from the list, and don't try to tune that by adding fancy mail-headers, people will not like it.

Remember to keep the context and the attribution of your replies intact, keep the "John Kernelhacker wrote ...:" lines at the top of your reply, and add your statements between the individual quoted sections instead of writing at the top of the mail.

If you add patches to your mail, make sure they are plain readable text as stated in Documentation/process/submitting-patches.rst. Kernel developers don't want to deal with attachments or compressed patches; they may want to comment on individual lines of your patch, which works only that way. Make sure you use a mail program that does not mangle spaces and tab characters. A good first test is to send the mail to yourself and try to apply your own patch by yourself. If that doesn't work, get your mail program fixed or change it until it works.

Above all, please remember to show respect to other subscribers.

Working with the community

The goal of the kernel community is to provide the best possible kernel there is. When you submit a patch for acceptance, it will be reviewed on its technical merits and those alone. So, what should you be expecting?

- criticism
- comments
- requests for change
- requests for justification
- silence

Remember, this is part of getting your patch into the kernel. You have to be able to take criticism and comments about your patches, evaluate them at a technical level and either rework your patches or provide clear and concise reasoning as to why those changes should not be made. If there are no responses to your posting, wait a few days and try again, sometimes things get lost in the huge volume.

What should you not do?

- expect your patch to be accepted without question
- become defensive
- ignore comments
- resubmit the patch without making any of the requested changes

In a community that is looking for the best technical solution possible, there will always be differing opinions on how beneficial a patch is. You have to be cooperative, and willing to adapt your idea to fit within the kernel. Or at least be willing to prove your idea

is worth it. Remember, being wrong is acceptable as long as you are willing to work toward a solution that is right.

It is normal that the answers to your first patch might simply be a list of a dozen things you should correct. This does **not** imply that your patch will not be accepted, and it is **not** meant against you personally. Simply correct all issues raised against your patch and resend it.

Differences between the kernel community and corporate structures

The kernel community works differently than most traditional corporate development environments. Here are a list of things that you can try to do to avoid problems:

Good things to say regarding your proposed changes:

- “This solves multiple problems.”
- “This deletes 2000 lines of code.”
- “Here is a patch that explains what I am trying to describe.”
- “I tested it on 5 different architectures...”
- “Here is a series of small patches that...”
- “This increases performance on typical machines...”

Bad things you should avoid saying:

- “We did it this way in AIX/ptx/Solaris, so therefore it must be good...”

- “I’ve being doing this for 20 years, so...”
- “This is required for my company to make money”
- “This is for our Enterprise product line.”
- “Here is my 1000 page design document that describes my idea”
- “I’ve been working on this for 6 months...”
- “Here’s a 5000 line patch that...”
- “I rewrote all of the current mess, and here it is...”
- “I have a deadline, and this patch needs to be applied now.”

Another way the kernel community is different than most traditional software engineering work environments is the faceless nature of interaction. One benefit of using email and irc as the primary forms of communication is the lack of discrimination based on gender or race. The Linux kernel work environment is accepting of women and minorities because all you are is an email address. The international aspect also helps to level the playing field because you can’t guess gender based on a person’s name. A man may be named Andrea and a woman may be named Pat. Most women who have worked in the Linux kernel and have expressed an opinion have had positive experiences.

The language barrier can cause problems for some people who are not comfortable with English. A good grasp of the language can be needed in order to get ideas across properly on mailing lists, so it is recommended that you check your emails to make sure they make sense in English before sending them.

Break up your changes

The Linux kernel community does not gladly accept large chunks of code dropped on it all at once. The changes need to be properly introduced, discussed, and broken up into

tiny, individual portions. This is almost the exact opposite of what companies are used to doing. Your proposal should also be introduced very early in the development process, so that you can receive feedback on what you are doing. It also lets the community feel that you are working with them, and not simply using them as a dumping ground for your feature. However, don't send 50 emails at one time to a mailing list, your patch series should be smaller than that almost all of the time.

The reasons for breaking things up are the following:

1. Small patches increase the likelihood that your patches will be applied, since they don't take much time or effort to verify for correctness. A 5 line patch can be applied by a maintainer with barely a second glance. However, a 500 line patch may take hours to review for correctness (the time it takes is exponentially proportional to the size of the patch, or something).
2. Small patches also make it very easy to debug when something goes wrong. It's much easier to back out patches one by one than it is to dissect a very large patch after it's been applied (and broken something).
3. It's important not only to send small patches, but also to rewrite and simplify (or simply re-order) patches before submitting them.

Here is an analogy from kernel developer Al Viro:

> *"Think of a teacher grading homework from a math student. The teacher does not want to see the student's trials and errors before they came up with the solution. They want to see the cleanest, most elegant answer. A good student knows this, and would never submit her intermediate work before the final solution.*

> *The same is true of kernel development. The maintainers and reviewers do not want to see the thought process behind the solution to the problem one is solving. They want to see a simple and elegant solution."*

It may be challenging to keep the balance between presenting an elegant solution and working together with the community and discussing your unfinished work. Therefore it is good to get early in the process to get feedback to improve your work, but also keep your changes in small chunks that they may get already accepted, even when your whole task is not ready for inclusion now.

Also realize that it is not acceptable to send patches for inclusion that are unfinished and will be "fixed up later."

Justify your change

Along with breaking up your patches, it is very important for you to let the Linux community know why they should add this change. New features must be justified as being needed and useful.

Document your change

When sending in your patches, pay special attention to what you say in the text in your email. This information will become the ChangeLog information for the patch, and will be preserved for everyone to see for all time. It should describe the patch completely, containing:

- why the change is necessary
- the overall design approach in the patch
- implementation details

- testing results

For more details on what this should all look like, please see the ChangeLog section of the document:

“The Perfect Patch”

http://www.ozlabs.org/~akpm/stuff/tpp.txt

All of these things are sometimes very hard to do. It can take years to perfect these practices (if at all). It’s a continuous process of improvement that requires a lot of patience and determination. But don’t give up, it’s possible. Many have done it before, and each had to start exactly where you are now.