Linux Kernel Development Documentation

Release 4.13.0-rc4+

The kernel development community

Sep 05, 2017

So you want to be a Linux kernel developer? Welcome! While there is a lot to be learned about the kernel in a technical sense, it is also important to learn about how our community works. Reading these documents will make it much easier for you to get your changes merged with a minimum of trouble.

Below are the essential guides that every developer should read.

CHAPTER

ONE

HOWTO 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If anything in this document becomes out of date, please send in patches to the maintainer of this file, who is listed at the bottom of the document.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-drive
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 at https://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).

   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).

2. 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.

Thanks to Paolo Ciarrocchi who allowed the "Development Process" (https://lwn.net/Articles/94386/) section to be based on text he had written, and to Randy Dunlap and Gerrit Huizenga for some of the list of things you should and should not say. Also thanks to Pat Mochel, Hanna Linder, Randy Dunlap, Kay Sievers, Vojtech Pavlik, Jan Kara, Josh Boyer, Kees Cook, Andrew Morton, Andi Kleen, Vadim Lobanov, Jesper Juhl, Adrian Bunk, Keri Harris, Frans Pop, David A. Wheeler, Junio Hamano, Michael Kerrisk, and Alex Shepard for their review, comments, and contributions. Without their help, this document would not have been possible.

Maintainer: Greg Kroah-Hartman <greg@kroah.com>

CHAPTER

TWO

CODE OF CONFLICT

The Linux kernel development effort is a very personal process compared to "traditional" ways of developing software. Your code and ideas behind it will be carefully reviewed, often resulting in critique and criticism. The review will almost always require improvements to the code before it can be included in the kernel. Know that this happens because everyone involved wants to see the best possible solution for the overall success of Linux. This development process has been proven to create the most robust operating system kernel ever, and we do not want to do anything to cause the quality of submission and eventual result to ever decrease.

If however, anyone feels personally abused, threatened, or otherwise uncomfortable due to this process, that is not acceptable. If so, please contact the Linux Foundation's Technical Advisory Board at <tab@lists.linux-foundation.org>, or the individual members, and they will work to resolve the issue to the best of their ability. For more information on who is on the Technical Advisory Board and what their role is, please see:

- http://www.linuxfoundation.org/projects/linux/tab

As a reviewer of code, please strive to keep things civil and focused on the technical issues involved. We are all humans, and frustrations can be high on both sides of the process. Try to keep in mind the immortal words of Bill and Ted, "Be excellent to each other."

CHAPTER

THREE

A GUIDE TO THE KERNEL DEVELOPMENT PROCESS

Contents:

* Introduction

* Executive summary

The rest of this section covers the scope of the kernel development process and the kinds of frustrations that developers and their employers can encounter there. There are a great many reasons why kernel code should be merged into the official ("mainline") kernel, including automatic availability to users, community support in many forms, and the ability to influence the direction of kernel development. Code contributed to the Linux kernel must be made available under a GPL-compatible license.

How the development process works introduces the development process, the kernel release cycle, and the mechanics of the merge window. The various phases in the patch development, review, and merging cycle are covered. There is some discussion of tools and mailing lists. Developers wanting to get started with kernel development are encouraged to track down and fix bugs as an initial exercise.

Early-stage planning covers early-stage project planning, with an emphasis on involving the development community as soon as possible.

Getting the code right is about the coding process; several pitfalls which have been encountered by other developers are discussed. Some requirements for patches are covered, and there is an introduction to some of the tools which can help to ensure that kernel patches are correct.

Posting patches talks about the process of posting patches for review. To be taken seriously by the development community, patches must be properly formatted and described, and they must be sent to the right place. Following the advice in this section should help to ensure the best possible reception for your work.

Followthrough covers what happens after posting patches; the job is far from done at that point. Working with reviewers is a crucial part of the development process; this section offers a number of tips on how to avoid problems at this important stage. Developers are cautioned against assuming that the job is done when a patch is merged into the mainline.

Advanced topics introduces a couple of "advanced" topics: managing patches with git and reviewing patches posted by others.

For more information concludes the document with pointers to sources for more information on kernel development.

* What this document is about

The Linux kernel, at over 8 million lines of code and well over 1000 contributors to each release, is one of the largest and most active free software projects in existence. Since its humble beginning in 1991, this kernel has evolved into a best-of-breed operating system component which runs on pocket-sized digital music players, desktop PCs, the largest supercomputers in existence, and all types of systems in between. It is a robust, efficient, and scalable solution for almost any situation.

With the growth of Linux has come an increase in the number of developers (and companies) wishing to participate in its development. Hardware vendors want to ensure that Linux supports their products well, making those products attractive to Linux users. Embedded systems vendors, who use Linux as a component in an integrated product, want Linux to be as capable and well-suited to the task at hand as possible. Distributors and other software vendors who base their products on Linux have a clear interest in the capabilities, performance, and reliability of the Linux kernel. And end users, too, will often wish to change Linux to make it better suit their needs.

One of the most compelling features of Linux is that it is accessible to these developers; anybody with the requisite skills can improve Linux and influence the direction of its development. Proprietary products cannot offer this kind of openness, which is a characteristic of the free software process. But, if anything, the kernel is even more open than most other free software projects. A typical three-month kernel development cycle can involve over 1000 developers working for more than 100 different companies (or for no company at all).

Working with the kernel development community is not especially hard. But, that notwithstanding, many potential contributors have experienced difficulties when trying to do kernel work. The kernel community has evolved its own distinct ways of operating which allow it to function smoothly (and produce a high-quality product) in an environment where thousands of lines of code are being changed every day. So it is not surprising that Linux kernel development process differs greatly from proprietary development methods.

The kernel's development process may come across as strange and intimidating to new developers, but there are good reasons and solid experience behind it. A developer who does not understand the kernel community's ways (or, worse, who tries to flout or circumvent them) will have a frustrating experience in store. The development community, while being helpful to those who are trying to learn, has little time for those who will not listen or who do not care about the development process.

It is hoped that those who read this document will be able to avoid that frustrating experience. There is a lot of material here, but the effort involved in reading it will be repaid in short order. The development community is always in need of developers who will help to make the kernel better; the following text should help you - or those who work for you - join our community.

* Credits

This document was written by Jonathan Corbet, corbet@lwn.net. It has been improved by comments from Johannes Berg, James Berry, Alex Chiang, Roland Dreier, Randy Dunlap, Jake Edge, Jiri Kosina, Matt Mackall, Arthur Marsh, Amanda McPherson, Andrew Morton, Andrew Price, Tsugikazu Shibata, and Jochen Voß.

This work was supported by the Linux Foundation; thanks especially to Amanda McPherson, who saw the value of this effort and made it all happen.

* The importance of getting code into the mainline

Some companies and developers occasionally wonder why they should bother learning how to work with the kernel community and get their code into the mainline kernel (the "mainline" being the kernel maintained by Linus Torvalds and used as a base by Linux distributors). In the short term, contributing code can look like an avoidable expense; it seems easier to just keep the code separate and support users directly. The truth of the matter is that keeping code separate ("out of tree") is a false economy.

As a way of illustrating the costs of out-of-tree code, here are a few relevant aspects of the kernel development process; most of these will be discussed in greater detail later in this document. Consider:

- Code which has been merged into the mainline kernel is available to all Linux users. It will automatically be present on all distributions which enable it. There is no need for driver disks, downloads, or the hassles of supporting multiple versions of multiple distributions; it all just works, for the developer and for the user. Incorporation into the mainline solves a large number of distribution and support problems.

- While kernel developers strive to maintain a stable interface to user space, the internal kernel API is in constant flux. The lack of a stable internal interface is a deliberate design decision; it allows fundamental improvements to be made at any time and results in higher-quality code. But one result of that policy is that any out-of-tree code requires constant upkeep if it is to work with new kernels. Maintaining out-of-tree code requires significant amounts of work just to keep that code working.

  Code which is in the mainline, instead, does not require this work as the result of a simple rule requiring any developer who makes an API change to also fix any code that breaks as the result of that change. So code which has been merged into the mainline has significantly lower maintenance costs.

- Beyond that, code which is in the kernel will often be improved by other developers. Surprising results can come from empowering your user community and customers to improve your product.
- Kernel code is subjected to review, both before and after merging into the mainline. No matter how strong the original developer's skills are, this review process invariably finds ways in which the code can be improved. Often review finds severe bugs and security problems. This is especially true for code which has been developed in a closed environment; such code benefits strongly from review by outside developers. Out-of-tree code is lower-quality code.
- Participation in the development process is your way to influence the direction of kernel development. Users who complain from the sidelines are heard, but active developers have a stronger voice - and the ability to implement changes which make the kernel work better for their needs.
- When code is maintained separately, the possibility that a third party will contribute a different implementation of a similar feature always exists. Should that happen, getting your code merged will become much harder - to the point of impossibility. Then you will be faced with the unpleasant alternatives of either (1) maintaining a nonstandard feature out of tree indefinitely, or (2) abandoning your code and migrating your users over to the in-tree version.
- Contribution of code is the fundamental action which makes the whole process work. By contributing your code you can add new functionality to the kernel and provide capabilities and examples which are of use to other kernel developers. If you have developed code for Linux (or are thinking about doing so), you clearly have an interest in the continued success of this platform; contributing code is one of the best ways to help ensure that success.

All of the reasoning above applies to any out-of-tree kernel code, including code which is distributed in proprietary, binary-only form. There are, however, additional factors which should be taken into account before considering any sort of binary-only kernel code distribution. These include:

- The legal issues around the distribution of proprietary kernel modules are cloudy at best; quite a few kernel copyright holders believe that most binary-only modules are derived products of the kernel and that, as a result, their distribution is a violation of the GNU General Public license (about which more will be said below). Your author is not a lawyer, and nothing in this document can possibly be considered to be legal advice. The true legal status of closed-source modules can only be determined by the courts. But the uncertainty which haunts those modules is there regardless.
- Binary modules greatly increase the difficulty of debugging kernel problems, to the point that most kernel developers will not even try. So the distribution of binary-only modules will make it harder for your users to get support from the community.
- Support is also harder for distributors of binary-only modules, who must provide a version of the module for every distribution and every kernel version they wish to support. Dozens of builds of a single module can be required to provide reasonably comprehensive coverage, and your users will have to upgrade your module separately every time they upgrade their kernel.
- Everything that was said above about code review applies doubly to closed-source code. Since this code is not available at all, it cannot have been reviewed by the community and will, beyond doubt, have serious problems.

Makers of embedded systems, in particular, may be tempted to disregard much of what has been said in this section in the belief that they are shipping a self-contained product which uses a frozen kernel version and requires no more development after its release. This argument misses the value of widespread code review and the value of allowing your users to add capabilities to your product. But these products, too, have a limited commercial life, after which a new version must be released. At that point, vendors whose code is in the mainline and well maintained will be much better positioned to get the new product ready for market quickly.

* Licensing

Code is contributed to the Linux kernel under a number of licenses, but all code must be compatible with version 2 of the GNU General Public License (GPLv2), which is the license covering the kernel distribution as a whole. In practice, that means that all code contributions are covered either by GPLv2 (with, optionally, language allowing distribution under later versions of the GPL) or the three-clause BSD license. Any contributions which are not covered by a compatible license will not be accepted into the kernel.

Copyright assignments are not required (or requested) for code contributed to the kernel. All code merged into the mainline kernel retains its original ownership; as a result, the kernel now has thousands of owners.

One implication of this ownership structure is that any attempt to change the licensing of the kernel is doomed to almost certain failure. There are few practical scenarios where the agreement of all copyright holders could be obtained (or their code removed from the kernel). So, in particular, there is no prospect of a migration to version 3 of the GPL in the foreseeable future.

It is imperative that all code contributed to the kernel be legitimately free software. For that reason, code from anonymous (or pseudonymous) contributors will not be accepted. All contributors are required to “sign off” on their code, stating that the code can be distributed with the kernel under the GPL. Code which has not been licensed as free software by its owner, or which risks creating copyright-related problems for the kernel (such as code which derives from reverse-engineering efforts lacking proper safeguards) cannot be contributed.

Questions about copyright-related issues are common on Linux development mailing lists. Such questions will normally receive no shortage of answers, but one should bear in mind that the people answering those questions are not lawyers and cannot provide legal advice. If you have legal questions relating to Linux source code, there is no substitute for talking with a lawyer who understands this field. Relying on answers obtained on technical mailing lists is a risky affair.

* How the development process works

Linux kernel development in the early 1990’s was a pretty loose affair, with relatively small numbers of users and developers involved. With a user base in the millions and with some 2,000 developers involved over the course of one year, the kernel has since had to evolve a number of processes to keep development happening smoothly. A solid understanding of how the process works is required in order to be an effective part of it.

* The big picture

The kernel developers use a loosely time-based release process, with a new major kernel release happening every two or three months. The recent release history looks like this:

2.6.38	March 14, 2011
2.6.37	January 4, 2011
2.6.36	October 20, 2010
2.6.35	August 1, 2010
2.6.34	May 15, 2010
2.6.33	February 24, 2010

Every 2.6.x release is a major kernel release with new features, internal API changes, and more. A typical 2.6 release can contain nearly 10,000 changesets with changes to several hundred thousand lines of code. 2.6 is thus the leading edge of Linux kernel development; the kernel uses a rolling development model which is continually integrating major changes.

A relatively straightforward discipline is followed with regard to the merging of patches for each release. At the beginning of each development cycle, the "merge window" is said to be open. At that time, code which is deemed to be sufficiently stable (and which is accepted by the development community) is merged into the mainline kernel. The bulk of changes for a new development cycle (and all of the major changes) will be merged during this time, at a rate approaching 1,000 changes ("patches," or "changesets") per day.

(As an aside, it is worth noting that the changes integrated during the merge window do not come out of thin air; they have been collected, tested, and staged ahead of time. How that process works will be described in detail later on).

The merge window lasts for approximately two weeks. At the end of this time, Linus Torvalds will declare that the window is closed and release the first of the "rc" kernels. For the kernel which is destined to be 2.6.40, for example, the release which happens at the end of the merge window will be called 2.6.40-rc1. The -rc1 release is the signal that the time to merge new features has passed, and that the time to stabilize the next kernel has begun.

Over the next six to ten weeks, only patches which fix problems should be submitted to the mainline. On occasion a more significant change will be allowed, but such occasions are rare; developers who try to merge new features outside of the merge window tend to get an unfriendly reception. As a general rule, if you miss the merge window for a given feature, the best thing to do is to wait for the next development cycle. (An occasional exception is made for drivers for previously-unsupported hardware; if they touch no in-tree code, they cannot cause regressions and should be safe to add at any time).

As fixes make their way into the mainline, the patch rate will slow over time. Linus releases new -rc kernels about once a week; a normal series will get up to somewhere between -rc6 and -rc9 before the kernel is considered to be sufficiently stable and the final 2.6.x release is made. At that point the whole process starts over again.

As an example, here is how the 2.6.38 development cycle went (all dates in 2011):

January 4	2.6.37 stable release
January 18	2.6.38-rc1, merge window closes
January 21	2.6.38-rc2
February 1	2.6.38-rc3
February 7	2.6.38-rc4
February 15	2.6.38-rc5
February 21	2.6.38-rc6
March 1	2.6.38-rc7
March 7	2.6.38-rc8
March 14	2.6.38 stable release

How do the developers decide when to close the development cycle and create the stable release? The most significant metric used is the list of regressions from previous releases. No bugs are welcome, but those which break systems which worked in the past are considered to be especially serious. For this reason, patches which cause regressions are looked upon unfavorably and are quite likely to be reverted during the stabilization period.

The developers' goal is to fix all known regressions before the stable release is made. In the real world, this kind of perfection is hard to achieve; there are just too many variables in a project of this size. There comes a point where delaying the final release just makes the problem worse; the pile of changes waiting for the next merge window will grow larger, creating even more regressions the next time around. So most 2.6.x kernels go out with a handful of known regressions though, hopefully, none of them are serious.

Once a stable release is made, its ongoing maintenance is passed off to the "stable team," currently consisting of Greg Kroah-Hartman. The stable team will release occasional updates to the stable release using the 2.6.x.y numbering scheme. To be considered for an update release, a patch must (1) fix a significant bug, and (2) already be merged into the mainline for the next development kernel. Kernels will typically receive stable updates for a little more than one development cycle past their initial release. So, for example, the 2.6.36 kernel's history looked like:

October 10	2.6.36 stable release
November 22	2.6.36.1
December 9	2.6.36.2
January 7	2.6.36.3
February 17	2.6.36.4

2.6.36.4 was the final stable update for the 2.6.36 release.

Some kernels are designated "long term" kernels; they will receive support for a longer period. As of this writing, the current long term kernels and their maintainers are:

2.6.27	Willy Tarreau	(Deep-frozen stable kernel)
2.6.32	Greg Kroah-Hartman	
2.6.35	Andi Kleen	(Embedded flag kernel)

The selection of a kernel for long-term support is purely a matter of a maintainer having the need and the time to maintain that release. There are no known plans for long-term support for any specific upcoming release.

* The lifecycle of a patch

Patches do not go directly from the developer's keyboard into the mainline kernel. There is, instead, a somewhat involved (if somewhat informal) process designed to ensure that each patch is reviewed for quality and that each patch implements a change which is desirable to have in the mainline. This process can happen quickly for minor fixes, or, in the case of large and controversial changes, go on for years. Much developer frustration comes from a lack of understanding of this process or from attempts to circumvent it.

In the hopes of reducing that frustration, this document will describe how a patch gets into the kernel. What follows below is an introduction which describes the process in a somewhat idealized way. A much more detailed treatment will come in later sections.

The stages that a patch goes through are, generally:

- Design. This is where the real requirements for the patch - and the way those requirements will be met - are laid out. Design work is often done without involving the community, but it is better to do this work in the open if at all possible; it can save a lot of time redesigning things later.
- Early review. Patches are posted to the relevant mailing list, and developers on that list reply with any comments they may have. This process should turn up any major problems with a patch if all goes well.
- Wider review. When the patch is getting close to ready for mainline inclusion, it should be accepted by a relevant subsystem maintainer - though this acceptance is not a guarantee that the patch will make it all the way to the mainline. The patch will show up in the maintainer's subsystem tree and into the -next trees (described below). When the process works, this step leads to more extensive review of the patch and the discovery of any problems resulting from the integration of this patch with work being done by others.
- Please note that most maintainers also have day jobs, so merging your patch may not be their highest priority. If your patch is getting feedback about changes that are needed, you should either make those changes or justify why they should not be made. If your patch has no review complaints but is not being merged by its appropriate subsystem or driver maintainer, you should be persistent in updating the patch to the current kernel so that it applies cleanly and keep sending it for review and merging.
- Merging into the mainline. Eventually, a successful patch will be merged into the mainline repository managed by Linus Torvalds. More comments and/or problems may surface at this time; it is important that the developer be responsive to these and fix any issues which arise.

- Stable release. The number of users potentially affected by the patch is now large, so, once again, new problems may arise.
- Long-term maintenance. While it is certainly possible for a developer to forget about code after merging it, that sort of behavior tends to leave a poor impression in the development community. Merging code eliminates some of the maintenance burden, in that others will fix problems caused by API changes. But the original developer should continue to take responsibility for the code if it is to remain useful in the longer term.

One of the largest mistakes made by kernel developers (or their employers) is to try to cut the process down to a single "merging into the mainline" step. This approach invariably leads to frustration for everybody involved.

* How patches get into the Kernel

There is exactly one person who can merge patches into the mainline kernel repository: Linus Torvalds. But, of the over 9,500 patches which went into the 2.6.38 kernel, only 112 (around 1.3%) were directly chosen by Linus himself. The kernel project has long since grown to a size where no single developer could possibly inspect and select every patch unassisted. The way the kernel developers have addressed this growth is through the use of a lieutenant system built around a chain of trust.

The kernel code base is logically broken down into a set of subsystems: networking, specific architecture support, memory management, video devices, etc. Most subsystems have a designated maintainer, a developer who has overall responsibility for the code within that subsystem. These subsystem maintainers are the gatekeepers (in a loose way) for the portion of the kernel they manage; they are the ones who will (usually) accept a patch for inclusion into the mainline kernel.

Subsystem maintainers each manage their own version of the kernel source tree, usually (but certainly not always) using the git source management tool. Tools like git (and related tools like quilt or mercurial) allow maintainers to track a list of patches, including authorship information and other metadata. At any given time, the maintainer can identify which patches in his or her repository are not found in the mainline.

When the merge window opens, top-level maintainers will ask Linus to "pull" the patches they have selected for merging from their repositories. If Linus agrees, the stream of patches will flow up into his repository, becoming part of the mainline kernel. The amount of attention that Linus pays to specific patches received in a pull operation varies. It is clear that, sometimes, he looks quite closely. But, as a general rule, Linus trusts the subsystem maintainers to not send bad patches upstream.

Subsystem maintainers, in turn, can pull patches from other maintainers. For example, the networking tree is built from patches which accumulated first in trees dedicated to network device drivers, wireless networking, etc. This chain of repositories can be arbitrarily long, though it rarely exceeds two or three links. Since each maintainer in the chain trusts those managing lower-level trees, this process is known as the "chain of trust."

Clearly, in a system like this, getting patches into the kernel depends on finding the right maintainer. Sending patches directly to Linus is not normally the right way to go.

* Next trees

The chain of subsystem trees guides the flow of patches into the kernel, but it also raises an interesting question: what if somebody wants to look at all of the patches which are being prepared for the next merge window? Developers will be interested in what other changes are pending to see whether there are any conflicts to worry about; a patch which changes a core kernel function prototype, for example, will conflict with any other patches which use the older form of that function. Reviewers and testers want access to the changes in their integrated form before all of those changes land in the mainline kernel. One could pull changes from all of the interesting subsystem trees, but that would be a big and error-prone job.

The answer comes in the form of -next trees, where subsystem trees are collected for testing and review. The older of these trees, maintained by Andrew Morton, is called "-mm" (for memory management, which is how it got started). The -mm tree integrates patches from a long list of subsystem trees; it also has some patches aimed at helping with debugging.

Beyond that, -mm contains a significant collection of patches which have been selected by Andrew directly. These patches may have been posted on a mailing list, or they may apply to a part of the kernel for which there is no designated subsystem tree. As a result, -mm operates as a sort of subsystem tree of last resort; if there is no other obvious path for a patch into the mainline, it is likely to end up in -mm. Miscellaneous patches which accumulate in -mm will eventually either be forwarded on to an appropriate subsystem tree or be sent directly to Linus. In a typical development cycle, approximately 5-10% of the patches going into the mainline get there via -mm.

The current -mm patch is available in the "mmotm" (-mm of the moment) directory at:

> http://www.ozlabs.org/~akpm/mmotm/

Use of the MMOTM tree is likely to be a frustrating experience, though; there is a definite chance that it will not even compile.

The primary tree for next-cycle patch merging is linux-next, maintained by Stephen Rothwell. The linux-next tree is, by design, a snapshot of what the mainline is expected to look like after the next merge window closes. Linux-next trees are announced on the linux-kernel and linux-next mailing lists when they are assembled; they can be downloaded from:

> http://www.kernel.org/pub/linux/kernel/next/

Linux-next has become an integral part of the kernel development process; all patches merged during a given merge window should really have found their way into linux-next some time before the merge window opens.

* Staging trees

The kernel source tree contains the drivers/staging/ directory, where many sub-directories for drivers or filesystems that are on their way to being added to the kernel tree live. They remain in drivers/staging while they still need more work; once complete, they can be moved into the kernel proper. This is a way to keep track of drivers that aren't up to Linux kernel coding or quality standards, but people may want to use them and track development.

Greg Kroah-Hartman currently maintains the staging tree. Drivers that still need work are sent to him, with each driver having its own subdirectory in drivers/staging/. Along with the driver source files, a TODO file should be present in the directory as well. The TODO file lists the pending work that the driver needs for acceptance into the kernel proper, as well as a list of people that should be Cc'd for any patches to the driver. Current rules require that drivers contributed to staging must, at a minimum, compile properly.

Staging can be a relatively easy way to get new drivers into the mainline where, with luck, they will come to the attention of other developers and improve quickly. Entry into staging is not the end of the story, though; code in staging which is not seeing regular progress will eventually be removed. Distributors also tend to be relatively reluctant to enable staging drivers. So staging is, at best, a stop on the way toward becoming a proper mainline driver.

* Tools

As can be seen from the above text, the kernel development process depends heavily on the ability to herd collections of patches in various directions. The whole thing would not work anywhere near as well as it does without suitably powerful tools. Tutorials on how to use these tools are well beyond the scope of this document, but there is space for a few pointers.

By far the dominant source code management system used by the kernel community is git. Git is one of a number of distributed version control systems being developed in the free software community. It is well tuned for kernel development, in that it performs quite well when dealing with large repositories and large numbers of patches. It also has a reputation for being difficult to learn and use, though it has gotten better over time. Some sort of familiarity with git is almost a requirement for kernel developers;

even if they do not use it for their own work, they'll need git to keep up with what other developers (and the mainline) are doing.

Git is now packaged by almost all Linux distributions. There is a home page at:

> http://git-scm.com/

That page has pointers to documentation and tutorials.

Among the kernel developers who do not use git, the most popular choice is almost certainly Mercurial:

> http://www.selenic.com/mercurial/

Mercurial shares many features with git, but it provides an interface which many find easier to use.

The other tool worth knowing about is Quilt:

> http://savannah.nongnu.org/projects/quilt/

Quilt is a patch management system, rather than a source code management system. It does not track history over time; it is, instead, oriented toward tracking a specific set of changes against an evolving code base. Some major subsystem maintainers use quilt to manage patches intended to go upstream. For the management of certain kinds of trees (-mm, for example), quilt is the best tool for the job.

* Mailing lists

A great deal of Linux kernel development work is done by way of mailing lists. It is hard to be a fully-functioning member of the community without joining at least one list somewhere. But Linux mailing lists also represent a potential hazard to developers, who risk getting buried under a load of electronic mail, running afoul of the conventions used on the Linux lists, or both.

Most kernel mailing lists are run on vger.kernel.org; the master list can be found at:

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

There are lists hosted elsewhere, though; a number of them are at lists.redhat.com.

The core mailing list for kernel development is, of course, linux-kernel. This list is an intimidating place to be; volume can reach 500 messages per day, the amount of noise is high, the conversation can be severely technical, and participants are not always concerned with showing a high degree of politeness. But there is no other place where the kernel development community comes together as a whole; developers who avoid this list will miss important information.

There are a few hints which can help with linux-kernel survival:

- Have the list delivered to a separate folder, rather than your main mailbox. One must be able to ignore the stream for sustained periods of time.
- Do not try to follow every conversation - nobody else does. It is important to filter on both the topic of interest (though note that long-running conversations can drift away from the original subject without changing the email subject line) and the people who are participating.
- Do not feed the trolls. If somebody is trying to stir up an angry response, ignore them.
- When responding to linux-kernel email (or that on other lists) preserve the Cc: header for all involved. In the absence of a strong reason (such as an explicit request), you should never remove recipients. Always make sure that the person you are responding to is in the Cc: list. This convention also makes it unnecessary to explicitly ask to be copied on replies to your postings.
- Search the list archives (and the net as a whole) before asking questions. Some developers can get impatient with people who clearly have not done their homework.
- Avoid top-posting (the practice of putting your answer above the quoted text you are responding to). It makes your response harder to read and makes a poor impression.
- Ask on the correct mailing list. Linux-kernel may be the general meeting point, but it is not the best place to find developers from all subsystems.

The last point - finding the correct mailing list - is a common place for beginning developers to go wrong. Somebody who asks a networking-related question on linux-kernel will almost certainly receive a polite suggestion to ask on the netdev list instead, as that is the list frequented by most networking developers. Other lists exist for the SCSI, video4linux, IDE, filesystem, etc. subsystems. The best place to look for mailing lists is in the MAINTAINERS file packaged with the kernel source.

* Getting started with Kernel development

Questions about how to get started with the kernel development process are common - from both individuals and companies. Equally common are missteps which make the beginning of the relationship harder than it has to be.

Companies often look to hire well-known developers to get a development group started. This can, in fact, be an effective technique. But it also tends to be expensive and does not do much to grow the pool of experienced kernel developers. It is possible to bring in-house developers up to speed on Linux kernel development, given the investment of a bit of time. Taking this time can endow an employer with a group of developers who understand the kernel and the company both, and who can help to train others as well. Over the medium term, this is often the more profitable approach.

Individual developers are often, understandably, at a loss for a place to start. Beginning with a large project can be intimidating; one often wants to test the waters with something smaller first. This is the point where some developers jump into the creation of patches fixing spelling errors or minor coding style issues. Unfortunately, such patches create a level of noise which is distracting for the development community as a whole, so, increasingly, they are looked down upon. New developers wishing to introduce themselves to the community will not get the sort of reception they wish for by these means.

Andrew Morton gives this advice for aspiring kernel developers

```
The #1 project for all kernel beginners should surely be "make sure
that the kernel runs perfectly at all times on all machines which
you can lay your hands on".  Usually the way to do this is to work
with others on getting things fixed up (this can require
persistence!) but that's fine - it's a part of kernel development.
```

(http://lwn.net/Articles/283982/).

In the absence of obvious problems to fix, developers are advised to look at the current lists of regressions and open bugs in general. There is never any shortage of issues in need of fixing; by addressing these issues, developers will gain experience with the process while, at the same time, building respect with the rest of the development community.

* Early-stage planning

When contemplating a Linux kernel development project, it can be tempting to jump right in and start coding. As with any significant project, though, much of the groundwork for success is best laid before the first line of code is written. Some time spent in early planning and communication can save far more time later on.

* Specifying the problem

Like any engineering project, a successful kernel enhancement starts with a clear description of the problem to be solved. In some cases, this step is easy: when a driver is needed for a specific piece of hardware, for example. In others, though, it is tempting to confuse the real problem with the proposed solution, and that can lead to difficulties.

Consider an example: some years ago, developers working with Linux audio sought a way to run applications without dropouts or other artifacts caused by excessive latency in the system. The solution they arrived at was a kernel module intended to hook into the Linux Security Module (LSM) framework; this module could be configured to give specific applications access to the realtime scheduler. This module was implemented and sent to the linux-kernel mailing list, where it immediately ran into problems.

To the audio developers, this security module was sufficient to solve their immediate problem. To the wider kernel community, though, it was seen as a misuse of the LSM framework (which is not intended to confer privileges onto processes which they would not otherwise have) and a risk to system stability. Their preferred solutions involved realtime scheduling access via the rlimit mechanism for the short term, and ongoing latency reduction work in the long term.

The audio community, however, could not see past the particular solution they had implemented; they were unwilling to accept alternatives. The resulting disagreement left those developers feeling disillusioned with the entire kernel development process; one of them went back to an audio list and posted this:

> There are a number of very good Linux kernel developers, but they tend to get outshouted by a large crowd of arrogant fools. Trying to communicate user requirements to these people is a waste of time. They are much too “intelligent” to listen to lesser mortals.

(http://lwn.net/Articles/131776/).

The reality of the situation was different; the kernel developers were far more concerned about system stability, long-term maintenance, and finding the right solution to the problem than they were with a specific module. The moral of the story is to focus on the problem - not a specific solution - and to discuss it with the development community before investing in the creation of a body of code.

So, when contemplating a kernel development project, one should obtain answers to a short set of questions:

- What, exactly, is the problem which needs to be solved?
- Who are the users affected by this problem? Which use cases should the solution address?
- How does the kernel fall short in addressing that problem now?

Only then does it make sense to start considering possible solutions.

* Early discussion

When planning a kernel development project, it makes great sense to hold discussions with the community before launching into implementation. Early communication can save time and trouble in a number of ways:

- It may well be that the problem is addressed by the kernel in ways which you have not understood. The Linux kernel is large and has a number of features and capabilities which are not immediately obvious. Not all kernel capabilities are documented as well as one might like, and it is easy to miss things. Your author has seen the posting of a complete driver which duplicated an existing driver that the new author had been unaware of. Code which reinvents existing wheels is not only wasteful; it will also not be accepted into the mainline kernel.
- There may be elements of the proposed solution which will not be acceptable for mainline merging. It is better to find out about problems like this before writing the code.
- It’s entirely possible that other developers have thought about the problem; they may have ideas for a better solution, and may be willing to help in the creation of that solution.

Years of experience with the kernel development community have taught a clear lesson: kernel code which is designed and developed behind closed doors invariably has problems which are only revealed when the code is released into the community. Sometimes these problems are severe, requiring months or years of effort before the code can be brought up to the kernel community’s standards. Some examples include:

- The Devicescape network stack was designed and implemented for single-processor systems. It could not be merged into the mainline until it was made suitable for multiprocessor systems. Retrofitting locking and such into code is a difficult task; as a result, the merging of this code (now called mac80211) was delayed for over a year.

- The Reiser4 filesystem included a number of capabilities which, in the core kernel developers' opinion, should have been implemented in the virtual filesystem layer instead. It also included features which could not easily be implemented without exposing the system to user-caused deadlocks. The late revelation of these problems - and refusal to address some of them - has caused Reiser4 to stay out of the mainline kernel.
- The AppArmor security module made use of internal virtual filesystem data structures in ways which were considered to be unsafe and unreliable. This concern (among others) kept AppArmor out of the mainline for years.

In each of these cases, a great deal of pain and extra work could have been avoided with some early discussion with the kernel developers.

* Who do you talk to?

When developers decide to take their plans public, the next question will be: where do we start? The answer is to find the right mailing list(s) and the right maintainer. For mailing lists, the best approach is to look in the MAINTAINERS file for a relevant place to post. If there is a suitable subsystem list, posting there is often preferable to posting on linux-kernel; you are more likely to reach developers with expertise in the relevant subsystem and the environment may be more supportive.

Finding maintainers can be a bit harder. Again, the MAINTAINERS file is the place to start. That file tends to not always be up to date, though, and not all subsystems are represented there. The person listed in the MAINTAINERS file may, in fact, not be the person who is actually acting in that role currently. So, when there is doubt about who to contact, a useful trick is to use git (and "git log" in particular) to see who is currently active within the subsystem of interest. Look at who is writing patches, and who, if anybody, is attaching Signed-off-by lines to those patches. Those are the people who will be best placed to help with a new development project.

The task of finding the right maintainer is sometimes challenging enough that the kernel developers have added a script to ease the process:

```
.../scripts/get_maintainer.pl
```

This script will return the current maintainer(s) for a given file or directory when given the "-f" option. If passed a patch on the command line, it will list the maintainers who should probably receive copies of the patch. There are a number of options regulating how hard get_maintainer.pl will search for maintainers; please be careful about using the more aggressive options as you may end up including developers who have no real interest in the code you are modifying.

If all else fails, talking to Andrew Morton can be an effective way to track down a maintainer for a specific piece of code.

* When to post?

If possible, posting your plans during the early stages can only be helpful. Describe the problem being solved and any plans that have been made on how the implementation will be done. Any information you can provide can help the development community provide useful input on the project.

One discouraging thing which can happen at this stage is not a hostile reaction, but, instead, little or no reaction at all. The sad truth of the matter is (1) kernel developers tend to be busy, (2) there is no shortage of people with grand plans and little code (or even prospect of code) to back them up, and (3) nobody is obligated to review or comment on ideas posted by others. Beyond that, high-level designs often hide problems which are only reviewed when somebody actually tries to implement those designs; for that reason, kernel developers would rather see the code.

If a request-for-comments posting yields little in the way of comments, do not assume that it means there is no interest in the project. Unfortunately, you also cannot assume that there are no problems with your idea. The best thing to do in this situation is to proceed, keeping the community informed as you go.

* Getting official buy-in

If your work is being done in a corporate environment - as most Linux kernel work is - you must, obviously, have permission from suitably empowered managers before you can post your company's plans or code to a public mailing list. The posting of code which has not been cleared for release under a GPL-compatible license can be especially problematic; the sooner that a company's management and legal staff can agree on the posting of a kernel development project, the better off everybody involved will be.

Some readers may be thinking at this point that their kernel work is intended to support a product which does not yet have an officially acknowledged existence. Revealing their employer's plans on a public mailing list may not be a viable option. In cases like this, it is worth considering whether the secrecy is really necessary; there is often no real need to keep development plans behind closed doors.

That said, there are also cases where a company legitimately cannot disclose its plans early in the development process. Companies with experienced kernel developers may choose to proceed in an open-loop manner on the assumption that they will be able to avoid serious integration problems later. For companies without that sort of in-house expertise, the best option is often to hire an outside developer to review the plans under a non-disclosure agreement. The Linux Foundation operates an NDA program designed to help with this sort of situation; more information can be found at:

> http://www.linuxfoundation.org/en/NDA_program

This kind of review is often enough to avoid serious problems later on without requiring public disclosure of the project.

* Getting the code right

While there is much to be said for a solid and community-oriented design process, the proof of any kernel development project is in the resulting code. It is the code which will be examined by other developers and merged (or not) into the mainline tree. So it is the quality of this code which will determine the ultimate success of the project.

This section will examine the coding process. We'll start with a look at a number of ways in which kernel developers can go wrong. Then the focus will shift toward doing things right and the tools which can help in that quest.

* Pitfalls

Coding style

The kernel has long had a standard coding style, described in *Documentation/process/coding-style.rst* . For much of that time, the policies described in that file were taken as being, at most, advisory. As a result, there is a substantial amount of code in the kernel which does not meet the coding style guidelines. The presence of that code leads to two independent hazards for kernel developers.

The first of these is to believe that the kernel coding standards do not matter and are not enforced. The truth of the matter is that adding new code to the kernel is very difficult if that code is not coded according to the standard; many developers will request that the code be reformatted before they will even review it. A code base as large as the kernel requires some uniformity of code to make it possible for developers to quickly understand any part of it. So there is no longer room for strangely-formatted code.

Occasionally, the kernel's coding style will run into conflict with an employer's mandated style. In such cases, the kernel's style will have to win before the code can be merged. Putting code into the kernel means giving up a degree of control in a number of ways - including control over how the code is formatted.

The other trap is to assume that code which is already in the kernel is urgently in need of coding style fixes. Developers may start to generate reformatting patches as a way of gaining familiarity with the process, or as a way of getting their name into the kernel changelogs - or both. But pure coding style fixes are seen as noise by the development community; they tend to get a chilly reception. So this type of patch is best avoided. It is natural to fix the style of a piece of code while working on it for other reasons, but coding style changes should not be made for their own sake.

The coding style document also should not be read as an absolute law which can never be transgressed. If there is a good reason to go against the style (a line which becomes far less readable if split to fit within the 80-column limit, for example), just do it.

Abstraction layers

Computer Science professors teach students to make extensive use of abstraction layers in the name of flexibility and information hiding. Certainly the kernel makes extensive use of abstraction; no project involving several million lines of code could do otherwise and survive. But experience has shown that excessive or premature abstraction can be just as harmful as premature optimization. Abstraction should be used to the level required and no further.

At a simple level, consider a function which has an argument which is always passed as zero by all callers. One could retain that argument just in case somebody eventually needs to use the extra flexibility that it provides. By that time, though, chances are good that the code which implements this extra argument has been broken in some subtle way which was never noticed - because it has never been used. Or, when the need for extra flexibility arises, it does not do so in a way which matches the programmer's early expectation. Kernel developers will routinely submit patches to remove unused arguments; they should, in general, not be added in the first place.

Abstraction layers which hide access to hardware - often to allow the bulk of a driver to be used with multiple operating systems - are especially frowned upon. Such layers obscure the code and may impose a performance penalty; they do not belong in the Linux kernel.

On the other hand, if you find yourself copying significant amounts of code from another kernel subsystem, it is time to ask whether it would, in fact, make sense to pull out some of that code into a separate library or to implement that functionality at a higher level. There is no value in replicating the same code throughout the kernel.

#ifdef and preprocessor use in general

The C preprocessor seems to present a powerful temptation to some C programmers, who see it as a way to efficiently encode a great deal of flexibility into a source file. But the preprocessor is not C, and heavy use of it results in code which is much harder for others to read and harder for the compiler to check for correctness. Heavy preprocessor use is almost always a sign of code which needs some cleanup work.

Conditional compilation with #ifdef is, indeed, a powerful feature, and it is used within the kernel. But there is little desire to see code which is sprinkled liberally with #ifdef blocks. As a general rule, #ifdef use should be confined to header files whenever possible. Conditionally-compiled code can be confined to functions which, if the code is not to be present, simply become empty. The compiler will then quietly optimize out the call to the empty function. The result is far cleaner code which is easier to follow.

C preprocessor macros present a number of hazards, including possible multiple evaluation of expressions with side effects and no type safety. If you are tempted to define a macro, consider creating an inline function instead. The code which results will be the same, but inline functions are easier to read, do not evaluate their arguments multiple times, and allow the compiler to perform type checking on the arguments and return value.

Inline functions

Inline functions present a hazard of their own, though. Programmers can become enamored of the perceived efficiency inherent in avoiding a function call and fill a source file with inline functions. Those functions, however, can actually reduce performance. Since their code is replicated at each call site, they end up bloating the size of the compiled kernel. That, in turn, creates pressure on the processor's memory caches, which can slow execution dramatically. Inline functions, as a rule, should be quite small and relatively rare. The cost of a function call, after all, is not that high; the creation of large numbers of inline functions is a classic example of premature optimization.

In general, kernel programmers ignore cache effects at their peril. The classic time/space tradeoff taught in beginning data structures classes often does not apply to contemporary hardware. Space *is* time, in that a larger program will run slower than one which is more compact.

More recent compilers take an increasingly active role in deciding whether a given function should actually be inlined or not. So the liberal placement of "inline" keywords may not just be excessive; it could also be irrelevant.

Locking

In May, 2006, the "Devicescape" networking stack was, with great fanfare, released under the GPL and made available for inclusion in the mainline kernel. This donation was welcome news; support for wireless networking in Linux was considered substandard at best, and the Devicescape stack offered the promise of fixing that situation. Yet, this code did not actually make it into the mainline until June, 2007 (2.6.22). What happened?

This code showed a number of signs of having been developed behind corporate doors. But one large problem in particular was that it was not designed to work on multiprocessor systems. Before this networking stack (now called mac80211) could be merged, a locking scheme needed to be retrofitted onto it.

Once upon a time, Linux kernel code could be developed without thinking about the concurrency issues presented by multiprocessor systems. Now, however, this document is being written on a dual-core laptop. Even on single-processor systems, work being done to improve responsiveness will raise the level of concurrency within the kernel. The days when kernel code could be written without thinking about locking are long past.

Any resource (data structures, hardware registers, etc.) which could be accessed concurrently by more than one thread must be protected by a lock. New code should be written with this requirement in mind; retrofitting locking after the fact is a rather more difficult task. Kernel developers should take the time to understand the available locking primitives well enough to pick the right tool for the job. Code which shows a lack of attention to concurrency will have a difficult path into the mainline.

Regressions

One final hazard worth mentioning is this: it can be tempting to make a change (which may bring big improvements) which causes something to break for existing users. This kind of change is called a "regression," and regressions have become most unwelcome in the mainline kernel. With few exceptions, changes which cause regressions will be backed out if the regression cannot be fixed in a timely manner. Far better to avoid the regression in the first place.

It is often argued that a regression can be justified if it causes things to work for more people than it creates problems for. Why not make a change if it brings new functionality to ten systems for each one it breaks? The best answer to this question was expressed by Linus in July, 2007:

```
So we don't fix bugs by introducing new problems.  That way lies
madness, and nobody ever knows if you actually make any real
progress at all. Is it two steps forwards, one step back, or one
step forward and two steps back?
```

(http://lwn.net/Articles/243460/).

An especially unwelcome type of regression is any sort of change to the user-space ABI. Once an interface has been exported to user space, it must be supported indefinitely. This fact makes the creation of user-space interfaces particularly challenging: since they cannot be changed in incompatible ways, they must be done right the first time. For this reason, a great deal of thought, clear documentation, and wide review for user-space interfaces is always required.

* Code checking tools

For now, at least, the writing of error-free code remains an ideal that few of us can reach. What we can hope to do, though, is to catch and fix as many of those errors as possible before our code goes into the mainline kernel. To that end, the kernel developers have put together an impressive array of tools which can catch a wide variety of obscure problems in an automated way. Any problem caught by the computer is a problem which will not afflict a user later on, so it stands to reason that the automated tools should be used whenever possible.

The first step is simply to heed the warnings produced by the compiler. Contemporary versions of gcc can detect (and warn about) a large number of potential errors. Quite often, these warnings point to real problems. Code submitted for review should, as a rule, not produce any compiler warnings. When silencing warnings, take care to understand the real cause and try to avoid "fixes" which make the warning go away without addressing its cause.

Note that not all compiler warnings are enabled by default. Build the kernel with "make EXTRA_CFLAGS=-W" to get the full set.

The kernel provides several configuration options which turn on debugging features; most of these are found in the "kernel hacking" submenu. Several of these options should be turned on for any kernel used for development or testing purposes. In particular, you should turn on:

- ENABLE_WARN_DEPRECATED, ENABLE_MUST_CHECK, and FRAME_WARN to get an extra set of warnings for problems like the use of deprecated interfaces or ignoring an important return value from a function. The output generated by these warnings can be verbose, but one need not worry about warnings from other parts of the kernel.
- DEBUG_OBJECTS will add code to track the lifetime of various objects created by the kernel and warn when things are done out of order. If you are adding a subsystem which creates (and exports) complex objects of its own, consider adding support for the object debugging infrastructure.
- DEBUG_SLAB can find a variety of memory allocation and use errors; it should be used on most development kernels.
- DEBUG_SPINLOCK, DEBUG_ATOMIC_SLEEP, and DEBUG_MUTEXES will find a number of common locking errors.

There are quite a few other debugging options, some of which will be discussed below. Some of them have a significant performance impact and should not be used all of the time. But some time spent learning the available options will likely be paid back many times over in short order.

One of the heavier debugging tools is the locking checker, or "lockdep." This tool will track the acquisition and release of every lock (spinlock or mutex) in the system, the order in which locks are acquired relative to each other, the current interrupt environment, and more. It can then ensure that locks are always acquired in the same order, that the same interrupt assumptions apply in all situations, and so on. In other words, lockdep can find a number of scenarios in which the system could, on rare occasion, deadlock. This kind of problem can be painful (for both developers and users) in a deployed system; lockdep allows them to be found in an automated manner ahead of time. Code with any sort of non-trivial locking should be run with lockdep enabled before being submitted for inclusion.

As a diligent kernel programmer, you will, beyond doubt, check the return status of any operation (such as a memory allocation) which can fail. The fact of the matter, though, is that the resulting failure recovery paths are, probably, completely untested. Untested code tends to be broken code; you could be much more confident of your code if all those error-handling paths had been exercised a few times.

The kernel provides a fault injection framework which can do exactly that, especially where memory allocations are involved. With fault injection enabled, a configurable percentage of memory allocations will be made to fail; these failures can be restricted to a specific range of code. Running with fault injection enabled allows the programmer to see how the code responds when things go badly. See Documentation/fault-injection/fault-injection.txt for more information on how to use this facility.

Other kinds of errors can be found with the "sparse" static analysis tool. With sparse, the programmer can be warned about confusion between user-space and kernel-space addresses, mixture of big-endian and small-endian quantities, the passing of integer values where a set of bit flags is expected, and so on.

Sparse must be installed separately (it can be found at https://sparse.wiki.kernel.org/index.php/Main_Page if your distributor does not package it); it can then be run on the code by adding "C=1" to your make command.

The "Coccinelle" tool (http://coccinelle.lip6.fr/) is able to find a wide variety of potential coding problems; it can also propose fixes for those problems. Quite a few "semantic patches" for the kernel have been packaged under the scripts/coccinelle directory; running "make coccicheck" will run through those semantic patches and report on any problems found. See Documentation/coccinelle.txt for more information.

Other kinds of portability errors are best found by compiling your code for other architectures. If you do not happen to have an S/390 system or a Blackfin development board handy, you can still perform the compilation step. A large set of cross compilers for x86 systems can be found at

> http://www.kernel.org/pub/tools/crosstool/

Some time spent installing and using these compilers will help avoid embarrassment later.

* Documentation

Documentation has often been more the exception than the rule with kernel development. Even so, adequate documentation will help to ease the merging of new code into the kernel, make life easier for other developers, and will be helpful for your users. In many cases, the addition of documentation has become essentially mandatory.

The first piece of documentation for any patch is its associated changelog. Log entries should describe the problem being solved, the form of the solution, the people who worked on the patch, any relevant effects on performance, and anything else that might be needed to understand the patch. Be sure that the changelog says *why* the patch is worth applying; a surprising number of developers fail to provide that information.

Any code which adds a new user-space interface - including new sysfs or /proc files - should include documentation of that interface which enables user-space developers to know what they are working with. See Documentation/ABI/README for a description of how this documentation should be formatted and what information needs to be provided.

The file Documentation/admin-guide/kernel-parameters.rst describes all of the kernel's boot-time parameters. Any patch which adds new parameters should add the appropriate entries to this file.

Any new configuration options must be accompanied by help text which clearly explains the options and when the user might want to select them.

Internal API information for many subsystems is documented by way of specially-formatted comments; these comments can be extracted and formatted in a number of ways by the "kernel-doc" script. If you are working within a subsystem which has kerneldoc comments, you should maintain them and add them, as appropriate, for externally-available functions. Even in areas which have not been so documented, there is no harm in adding kerneldoc comments for the future; indeed, this can be a useful activity for beginning kernel developers. The format of these comments, along with some information on how to create kerneldoc templates can be found at Documentation/doc-guide/ .

Anybody who reads through a significant amount of existing kernel code will note that, often, comments are most notable by their absence. Once again, the expectations for new code are higher than they were in the past; merging uncommented code will be harder. That said, there is little desire for verbosely-commented code. The code should, itself, be readable, with comments explaining the more subtle aspects.

Certain things should always be commented. Uses of memory barriers should be accompanied by a line explaining why the barrier is necessary. The locking rules for data structures generally need to be explained somewhere. Major data structures need comprehensive documentation in general. Non-obvious dependencies between separate bits of code should be pointed out. Anything which might tempt a code janitor to make an incorrect "cleanup" needs a comment saying why it is done the way it is. And so on.

* Internal API changes

The binary interface provided by the kernel to user space cannot be broken except under the most severe circumstances. The kernel's internal programming interfaces, instead, are highly fluid and can be changed when the need arises. If you find yourself having to work around a kernel API, or simply not using a specific functionality because it does not meet your needs, that may be a sign that the API needs to change. As a kernel developer, you are empowered to make such changes.

There are, of course, some catches. API changes can be made, but they need to be well justified. So any patch making an internal API change should be accompanied by a description of what the change is and why it is necessary. This kind of change should also be broken out into a separate patch, rather than buried within a larger patch.

The other catch is that a developer who changes an internal API is generally charged with the task of fixing any code within the kernel tree which is broken by the change. For a widely-used function, this duty can lead to literally hundreds or thousands of changes - many of which are likely to conflict with work being done by other developers. Needless to say, this can be a large job, so it is best to be sure that the justification is solid. Note that the Coccinelle tool can help with wide-ranging API changes.

When making an incompatible API change, one should, whenever possible, ensure that code which has not been updated is caught by the compiler. This will help you to be sure that you have found all in-tree uses of that interface. It will also alert developers of out-of-tree code that there is a change that they need to respond to. Supporting out-of-tree code is not something that kernel developers need to be worried about, but we also do not have to make life harder for out-of-tree developers than it needs to be.

* 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 process/submitting-patches.rst, process/submitting-drivers.rst, and process/submit-checklist.rst in the kernel documentation directory.

* 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 mentioned above are used to describe how various developers have been associated with the development of this patch. They are described in detail in the process/submitting-patches.rst document; what follows here is a brief summary. Each of these lines has the 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 Documentation/process/submitting-patches.rst. Code without a proper signoff cannot be merged into the mainline.

- 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 Documentation/process/submitting-patches.rst for more detail.
- 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.

  Documentation/process/email-clients.rst has some helpful hints on making specific mail clients work for sending patches.
- 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.

* Followthrough

At this point, you have followed the guidelines given so far and, with the addition of your own engineering skills, have posted a perfect series of patches. One of the biggest mistakes that even experienced kernel developers can make is to conclude that their work is now done. In truth, posting patches indicates a transition into the next stage of the process, with, possibly, quite a bit of work yet to be done.

It is a rare patch which is so good at its first posting that there is no room for improvement. The kernel development process recognizes this fact, and, as a result, is heavily oriented toward the improvement of posted code. You, as the author of that code, will be expected to work with the kernel community to ensure that your code is up to the kernel's quality standards. A failure to participate in this process is quite likely to prevent the inclusion of your patches into the mainline.

* Working with reviewers

A patch of any significance will result in a number of comments from other developers as they review the code. Working with reviewers can be, for many developers, the most intimidating part of the kernel development process. Life can be made much easier, though, if you keep a few things in mind:

- If you have explained your patch well, reviewers will understand its value and why you went to the trouble of writing it. But that value will not keep them from asking a fundamental question: what will it be like to maintain a kernel with this code in it five or ten years later? Many of the changes you may be asked to make - from coding style tweaks to substantial rewrites - come from the understanding that Linux will still be around and under development a decade from now.
- Code review is hard work, and it is a relatively thankless occupation; people remember who wrote kernel code, but there is little lasting fame for those who reviewed it. So reviewers can get grumpy, especially when they see the same mistakes being made over and over again. If you get a review which seems angry, insulting, or outright offensive, resist the impulse to respond in kind. Code review is about the code, not about the people, and code reviewers are not attacking you personally.
- Similarly, code reviewers are not trying to promote their employers' agendas at the expense of your own. Kernel developers often expect to be working on the kernel years from now, but they understand that their employer could change. They truly are, almost without exception, working toward the creation of the best kernel they can; they are not trying to create discomfort for their employers' competitors.

What all of this comes down to is that, when reviewers send you comments, you need to pay attention to the technical observations that they are making. Do not let their form of expression or your own pride keep that from happening. When you get review comments on a patch, take the time to understand what the reviewer is trying to say. If possible, fix the things that the reviewer is asking you to fix. And respond back to the reviewer: thank them, and describe how you will answer their questions.

Note that you do not have to agree with every change suggested by reviewers. If you believe that the reviewer has misunderstood your code, explain what is really going on. If you have a technical objection to a suggested change, describe it and justify your solution to the problem. If your explanations make sense, the reviewer will accept them. Should your explanation not prove persuasive, though, especially if others start to agree with the reviewer, take some time to think things over again. It can be easy to become blinded by your own solution to a problem to the point that you don't realize that something is fundamentally wrong or, perhaps, you're not even solving the right problem.

Andrew Morton has suggested that every review comment which does not result in a code change should result in an additional code comment instead; that can help future reviewers avoid the questions which came up the first time around.

One fatal mistake is to ignore review comments in the hope that they will go away. They will not go away. If you repost code without having responded to the comments you got the time before, you're likely to find that your patches go nowhere.

Speaking of reposting code: please bear in mind that reviewers are not going to remember all the details of the code you posted the last time around. So it is always a good idea to remind reviewers of previously raised issues and how you dealt with them; the patch changelog is a good place for this kind of information. Reviewers should not have to search through list archives to familiarize themselves with what was said last time; if you help them get a running start, they will be in a better mood when they revisit your code.

What if you've tried to do everything right and things still aren't going anywhere? Most technical disagreements can be resolved through discussion, but there are times when somebody simply has to make a decision. If you honestly believe that this decision is going against you wrongly, you can always try appealing to a higher power. As of this writing, that higher power tends to be Andrew Morton. Andrew has a great deal of respect in the kernel development community; he can often unjam a situation which seems to be hopelessly blocked. Appealing to Andrew should not be done lightly, though, and not before all other alternatives have been explored. And bear in mind, of course, that he may not agree with you either.

* What happens next

If a patch is considered to be a good thing to add to the kernel, and once most of the review issues have been resolved, the next step is usually entry into a subsystem maintainer's tree. How that works varies from one subsystem to the next; each maintainer has his or her own way of doing things. In particular, there may be more than one tree - one, perhaps, dedicated to patches planned for the next merge window, and another for longer-term work.

For patches applying to areas for which there is no obvious subsystem tree (memory management patches, for example), the default tree often ends up being -mm. Patches which affect multiple subsystems can also end up going through the -mm tree.

Inclusion into a subsystem tree can bring a higher level of visibility to a patch. Now other developers working with that tree will get the patch by default. Subsystem trees typically feed linux-next as well, making their contents visible to the development community as a whole. At this point, there's a good chance that you will get more comments from a new set of reviewers; these comments need to be answered as in the previous round.

What may also happen at this point, depending on the nature of your patch, is that conflicts with work being done by others turn up. In the worst case, heavy patch conflicts can result in some work being put on the back burner so that the remaining patches can be worked into shape and merged. Other times, conflict resolution will involve working with the other developers and, possibly, moving some patches between trees to ensure that everything applies cleanly. This work can be a pain, but count your blessings: before the advent of the linux-next tree, these conflicts often only turned up during the merge window and had to be addressed in a hurry. Now they can be resolved at leisure, before the merge window opens.

Some day, if all goes well, you'll log on and see that your patch has been merged into the mainline kernel. Congratulations! Once the celebration is complete (and you have added yourself to the MAINTAINERS file), though, it is worth remembering an important little fact: the job still is not done. Merging into the mainline brings its own challenges.

To begin with, the visibility of your patch has increased yet again. There may be a new round of comments from developers who had not been aware of the patch before. It may be tempting to ignore them, since there is no longer any question of your code being merged. Resist that temptation, though; you still need to be responsive to developers who have questions or suggestions.

More importantly, though: inclusion into the mainline puts your code into the hands of a much larger group of testers. Even if you have contributed a driver for hardware which is not yet available, you will be surprised by how many people will build your code into their kernels. And, of course, where there are testers, there will be bug reports.

The worst sort of bug reports are regressions. If your patch causes a regression, you'll find an uncomfortable number of eyes upon you; regressions need to be fixed as soon as possible. If you are unwilling or unable to fix the regression (and nobody else does it for you), your patch will almost certainly be removed during the stabilization period. Beyond negating all of the work you have done to get your patch into the mainline, having a patch pulled as the result of a failure to fix a regression could well make it harder for you to get work merged in the future.

After any regressions have been dealt with, there may be other, ordinary bugs to deal with. The stabilization period is your best opportunity to fix these bugs and ensure that your code's debut in a mainline kernel release is as solid as possible. So, please, answer bug reports, and fix the problems if at all possible. That's what the stabilization period is for; you can start creating cool new patches once any problems with the old ones have been taken care of.

And don't forget that there are other milestones which may also create bug reports: the next mainline stable release, when prominent distributors pick up a version of the kernel containing your patch, etc. Continuing to respond to these reports is a matter of basic pride in your work. If that is insufficient motivation, though, it's also worth considering that the development community remembers developers who lose interest in their code after it's merged. The next time you post a patch, they will be evaluating it with the assumption that you will not be around to maintain it afterward.

* Other things that can happen

One day, you may open your mail client and see that somebody has mailed you a patch to your code. That is one of the advantages of having your code out there in the open, after all. If you agree with the patch, you can either forward it on to the subsystem maintainer (be sure to include a proper From: line so that the attribution is correct, and add a signoff of your own), or send an Acked-by: response back and let the original poster send it upward.

If you disagree with the patch, send a polite response explaining why. If possible, tell the author what changes need to be made to make the patch acceptable to you. There is a certain resistance to merging patches which are opposed by the author and maintainer of the code, but it only goes so far. If you are seen as needlessly blocking good work, those patches will eventually flow around you and get into the mainline anyway. In the Linux kernel, nobody has absolute veto power over any code. Except maybe Linus.

On very rare occasion, you may see something completely different: another developer posts a different solution to your problem. At that point, chances are that one of the two patches will not be merged, and "mine was here first" is not considered to be a compelling technical argument. If somebody else's patch displaces yours and gets into the mainline, there is really only one way to respond: be pleased that your problem got solved and get on with your work. Having one's work shoved aside in this manner can be hurtful and discouraging, but the community will remember your reaction long after they have forgotten whose patch actually got merged.

* Advanced topics

At this point, hopefully, you have a handle on how the development process works. There is still more to learn, however! This section will cover a number of topics which can be helpful for developers wanting to become a regular part of the Linux kernel development process.

* Managing patches with git

The use of distributed version control for the kernel began in early 2002, when Linus first started playing with the proprietary BitKeeper application. While BitKeeper was controversial, the approach to software version management it embodied most certainly was not. Distributed version control enabled an immediate acceleration of the kernel development project. In current times, there are several free alternatives to BitKeeper. For better or for worse, the kernel project has settled on git as its tool of choice.

Managing patches with git can make life much easier for the developer, especially as the volume of those patches grows. Git also has its rough edges and poses certain hazards; it is a young and powerful tool which is still being civilized by its developers. This document will not attempt to teach the reader how to use git; that would be sufficient material for a long document in its own right. Instead, the focus here will be on how git fits into the kernel development process in particular. Developers who wish to come up to speed with git will find more information at:

> http://git-scm.com/
>
> http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

and on various tutorials found on the web.

The first order of business is to read the above sites and get a solid understanding of how git works before trying to use it to make patches available to others. A git-using developer should be able to obtain a copy of the mainline repository, explore the revision history, commit changes to the tree, use branches, etc. An understanding of git's tools for the rewriting of history (such as rebase) is also useful. Git comes with its own terminology and concepts; a new user of git should know about refs, remote branches, the index, fast-forward merges, pushes and pulls, detached heads, etc. It can all be a little intimidating at the outset, but the concepts are not that hard to grasp with a bit of study.

Using git to generate patches for submission by email can be a good exercise while coming up to speed.

When you are ready to start putting up git trees for others to look at, you will, of course, need a server that can be pulled from. Setting up such a server with git-daemon is relatively straightforward if you have a system which is accessible to the Internet. Otherwise, free, public hosting sites (Github, for example) are starting to appear on the net. Established developers can get an account on kernel.org, but those are not easy to come by; see http://kernel.org/faq/ for more information.

The normal git workflow involves the use of a lot of branches. Each line of development can be separated into a separate "topic branch" and maintained independently. Branches in git are cheap, there is no reason to not make free use of them. And, in any case, you should not do your development in any branch which you intend to ask others to pull from. Publicly-available branches should be created with care; merge in patches from development branches when they are in complete form and ready to go - not before.

Git provides some powerful tools which can allow you to rewrite your development history. An inconvenient patch (one which breaks bisection, say, or which has some other sort of obvious bug) can be fixed in place or made to disappear from the history entirely. A patch series can be rewritten as if it had been written on top of today's mainline, even though you have been working on it for months. Changes can be transparently shifted from one branch to another. And so on. Judicious use of git's ability to revise history can help in the creation of clean patch sets with fewer problems.

Excessive use of this capability can lead to other problems, though, beyond a simple obsession for the creation of the perfect project history. Rewriting history will rewrite the changes contained in that history, turning a tested (hopefully) kernel tree into an untested one. But, beyond that, developers cannot easily collaborate if they do not have a shared view of the project history; if you rewrite history which other developers have pulled into their repositories, you will make life much more difficult for those developers. So a simple rule of thumb applies here: history which has been exported to others should generally be seen as immutable thereafter.

So, once you push a set of changes to your publicly-available server, those changes should not be rewritten. Git will attempt to enforce this rule if you try to push changes which do not result in a fast-forward merge (i.e. changes which do not share the same history). It is possible to override this check, and there may be times when it is necessary to rewrite an exported tree. Moving changesets between trees to avoid conflicts in linux-next is one example. But such actions should be rare. This is one of the reasons why development should be done in private branches (which can be rewritten if necessary) and only moved into public branches when it's in a reasonably advanced state.

As the mainline (or other tree upon which a set of changes is based) advances, it is tempting to merge with that tree to stay on the leading edge. For a private branch, rebasing can be an easy way to keep up with another tree, but rebasing is not an option once a tree is exported to the world. Once that happens, a full merge must be done. Merging occasionally makes good sense, but overly frequent merges can clutter the history needlessly. Suggested technique in this case is to merge infrequently, and generally only at specific release points (such as a mainline -rc release). If you are nervous about specific changes, you can always perform test merges in a private branch. The git "rerere" tool can be useful in such situations; it remembers how merge conflicts were resolved so that you don't have to do the same work twice.

One of the biggest recurring complaints about tools like git is this: the mass movement of patches from one repository to another makes it easy to slip in ill-advised changes which go into the mainline below the review radar. Kernel developers tend to get unhappy when they see that kind of thing happening; putting up a git tree with unreviewed or off-topic patches can affect your ability to get trees pulled in the future. Quoting Linus:

```
You can send me patches, but for me to pull a git patch from you, I
need to know that you know what you're doing, and I need to be able
to trust things *without* then having to go and check every
individual change by hand.
```

(http://lwn.net/Articles/224135/).

To avoid this kind of situation, ensure that all patches within a given branch stick closely to the associated topic; a "driver fixes" branch should not be making changes to the core memory management code. And, most importantly, do not use a git tree to bypass the review process. Post an occasional summary of the tree to the relevant list, and, when the time is right, request that the tree be included in linux-next.

If and when others start to send patches for inclusion into your tree, don't forget to review them. Also ensure that you maintain the correct authorship information; the git "am" tool does its best in this regard, but you may have to add a "From:" line to the patch if it has been relayed to you via a third party.

When requesting a pull, be sure to give all the relevant information: where your tree is, what branch to pull, and what changes will result from the pull. The git request-pull command can be helpful in this regard; it will format the request as other developers expect, and will also check to be sure that you have remembered to push those changes to the public server.

* Reviewing patches

Some readers will certainly object to putting this section with "advanced topics" on the grounds that even beginning kernel developers should be reviewing patches. It is certainly true that there is no better way to learn how to program in the kernel environment than by looking at code posted by others. In addition, reviewers are forever in short supply; by looking at code you can make a significant contribution to the process as a whole.

Reviewing code can be an intimidating prospect, especially for a new kernel developer who may well feel nervous about questioning code - in public - which has been posted by those with more experience. Even code written by the most experienced developers can be improved, though. Perhaps the best piece of advice for reviewers (all reviewers) is this: phrase review comments as questions rather than criticisms. Asking "how does the lock get released in this path?" will always work better than stating "the locking here is wrong."

Different developers will review code from different points of view. Some are mostly concerned with coding style and whether code lines have trailing white space. Others will focus primarily on whether the change implemented by the patch as a whole is a good thing for the kernel or not. Yet others will check for problematic locking, excessive stack usage, possible security issues, duplication of code found elsewhere, adequate documentation, adverse effects on performance, user-space ABI changes, etc. All types of review, if they lead to better code going into the kernel, are welcome and worthwhile.

* For more information

There are numerous sources of information on Linux kernel development and related topics. First among those will always be the Documentation directory found in the kernel source distribution. The top-level process/howto.rst file is an important starting point; process/submitting-patches.rst and process/submitting-drivers.rst are also something which all kernel developers should read. Many internal kernel APIs are documented using the kerneldoc mechanism; "make htmldocs" or "make pdfdocs" can be used to generate those documents in HTML or PDF format (though the version of TeX shipped by some distributions runs into internal limits and fails to process the documents properly).

Various web sites discuss kernel development at all levels of detail. Your author would like to humbly suggest http://lwn.net/ as a source; information on many specific kernel topics can be found via the LWN kernel index at:

> http://lwn.net/Kernel/Index/

Beyond that, a valuable resource for kernel developers is:

> http://kernelnewbies.org/

And, of course, one should not forget http://kernel.org/, the definitive location for kernel release information.

There are a number of books on kernel development:

> Linux Device Drivers, 3rd Edition (Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman). Online at http://lwn.net/Kernel/LDD3/.
>
> Linux Kernel Development (Robert Love).
>
> Understanding the Linux Kernel (Daniel Bovet and Marco Cesati).

All of these books suffer from a common fault, though: they tend to be somewhat obsolete by the time they hit the shelves, and they have been on the shelves for a while now. Still, there is quite a bit of good information to be found there.

Documentation for git can be found at:

> http://www.kernel.org/pub/software/scm/git/docs/
>
> http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

* Conclusion

Congratulations to anybody who has made it through this long-winded document. Hopefully it has provided a helpful understanding of how the Linux kernel is developed and how you can participate in that process.

In the end, it's the participation that matters. Any open source software project is no more than the sum of what its contributors put into it. The Linux kernel has progressed as quickly and as well as it has because it has been helped by an impressively large group of developers, all of whom are working to make it better. The kernel is a premier example of what can be done when thousands of people work together toward a common goal.

The kernel can always benefit from a larger developer base, though. There is always more work to do. But, just as importantly, most other participants in the Linux ecosystem can benefit through contributing to the kernel. Getting code into the mainline is the key to higher code quality, lower maintenance and distribution costs, a higher level of influence over the direction of kernel development, and more. It is a situation where everybody involved wins. Fire up your editor and come join us; you will be more than welcome.

The purpose of this document is to help developers (and their managers) work with the development community with a minimum of frustration. It is an attempt to document how this community works in a way which is accessible to those who are not intimately familiar with Linux kernel development (or, indeed, free software development in general). While there is some technical material here, this is very much a process-oriented discussion which does not require a deep knowledge of kernel programming to understand.

CHAPTER

FOUR

SUBMITTING PATCHES: THE ESSENTIAL GUIDE TO GETTING YOUR CODE INTO THE KERNEL

For a person or company who wishes to submit a change to the Linux kernel, the process can sometimes be daunting if you're not familiar with "the system." This text is a collection of suggestions which can greatly increase the chances of your change being accepted.

This document contains a large number of suggestions in a relatively terse format. For detailed information on how the kernel development process works, see *Documentation/process* . Also, read *Documentation/process/submit-checklist.rst* for a list of items to check before submitting code. If you are submitting a driver, also read *Documentation/process/submitting-drivers.rst* ; for device tree binding patches, read Documentation/devicetree/bindings/submitting-patches.txt.

Many of these steps describe the default behavior of the `git` version control system; if you use `git` to prepare your patches, you'll find much of the mechanical work done for you, though you'll still need to prepare and document a sensible set of patches. In general, use of `git` will make your life as a kernel developer easier.

* 0) Obtain a current source tree

If you do not have a repository with the current kernel source handy, use `git` to obtain one. You'll want to start with the mainline repository, which can be grabbed with:

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

Note, however, that you may not want to develop against the mainline tree directly. Most subsystem maintainers run their own trees and want to see patches prepared against those trees. See the **T:** entry for the subsystem in the MAINTAINERS file to find that tree, or simply ask the maintainer if the tree is not listed there.

It is still possible to download kernel releases via tarballs (as described in the next section), but that is the hard way to do kernel development.

* 1) `diff -up`

If you must generate your patches by hand, use `diff -up` or `diff -uprN` to create patches. Git generates patches in this form by default; if you're using `git`, you can skip this section entirely.

All changes to the Linux kernel occur in the form of patches, as generated by *`diff(1)`*. When creating your patch, make sure to create it in "unified diff" format, as supplied by the `-u` argument to *`diff(1)`*. Also, please use the `-p` argument which shows which C function each change is in - that makes the resultant `diff` a lot easier to read. Patches should be based in the root kernel source directory, not in any lower subdirectory.

To create a patch for a single file, it is often sufficient to do:

```
SRCTREE= linux
MYFILE=  drivers/net/mydriver.c

cd $SRCTREE
cp $MYFILE $MYFILE.orig
vi $MYFILE      # make your change
cd ..
diff -up $SRCTREE/$MYFILE{.orig,} > /tmp/patch
```

To create a patch for multiple files, you should unpack a "vanilla", or unmodified kernel source tree, and generate a `diff` against your own source tree. For example:

```
MYSRC= /devel/linux

tar xvfz linux-3.19.tar.gz
mv linux-3.19 linux-3.19-vanilla
diff -uprN -X linux-3.19-vanilla/Documentation/dontdiff \
        linux-3.19-vanilla $MYSRC > /tmp/patch
```

`dontdiff` is a list of files which are generated by the kernel during the build process, and should be ignored in any *`diff(1)`*-generated patch.

Make sure your patch does not include any extra files which do not belong in a patch submission. Make sure to review your patch -after- generating it with *`diff(1)`*, to ensure accuracy.

If your changes produce a lot of deltas, you need to split them into individual patches which modify things in logical stages; see *3) Separate your changes* . This will facilitate review by other kernel developers, very important if you want your patch accepted.

If you're using git, `git rebase -i` can help you with this process. If you're not using git, `quilt` <http://savannah.nongnu.org/projects/quilt> is another popular alternative.

* 2) Describe your changes

Describe your problem. Whether your patch is a one-line bug fix or 5000 lines of a new feature, there must be an underlying problem that motivated you to do this work. Convince the reviewer that there is a problem worth fixing and that it makes sense for them to read past the first paragraph.

Describe user-visible impact. Straight up crashes and lockups are pretty convincing, but not all bugs are that blatant. Even if the problem was spotted during code review, describe the impact you think it can have on users. Keep in mind that the majority of Linux installations run kernels from secondary stable trees or vendor/product-specific trees that cherry-pick only specific patches from upstream, so include anything that could help route your change downstream: provoking circumstances, excerpts from dmesg, crash descriptions, performance regressions, latency spikes, lockups, etc.

Quantify optimizations and trade-offs. If you claim improvements in performance, memory consumption, stack footprint, or binary size, include numbers that back them up. But also describe non-obvious costs. Optimizations usually aren't free but trade-offs between CPU, memory, and readability; or, when it comes to heuristics, between different workloads. Describe the expected downsides of your optimization so that the reviewer can weigh costs against benefits.

Once the problem is established, describe what you are actually doing about it in technical detail. It's important to describe the change in plain English for the reviewer to verify that the code is behaving as you intend it to.

The maintainer will thank you if you write your patch description in a form which can be easily pulled into Linux's source code management system, `git`, as a "commit log". See *15) Explicit In-Reply-To headers* .

Solve only one problem per patch. If your description starts to get long, that's a sign that you probably need to split up your patch. See *3) Separate your changes* .

When you submit or resubmit a patch or patch series, include the complete patch description and justification for it. Don't just say that this is version N of the patch (series). Don't expect the subsystem maintainer to refer back to earlier patch versions or referenced URLs to find the patch description and put that into the patch. I.e., the patch (series) and its description should be self-contained. This benefits both the maintainers and reviewers. Some reviewers probably didn't even receive earlier versions of the patch.

Describe your changes in imperative mood, e.g. "make xyzzy do frotz" instead of "[This patch] makes xyzzy do frotz" or "[I] changed xyzzy to do frotz", as if you are giving orders to the codebase to change its behaviour.

If the patch fixes a logged bug entry, refer to that bug entry by number and URL. If the patch follows from a mailing list discussion, give a URL to the mailing list archive; use the https://lkml.kernel.org/ redirector with a `Message-Id`, to ensure that the links cannot become stale.

However, try to make your explanation understandable without external resources. In addition to giving a URL to a mailing list archive or bug, summarize the relevant points of the discussion that led to the patch as submitted.

If you want to refer to a specific commit, don't just refer to the SHA-1 ID of the commit. Please also include the oneline summary of the commit, to make it easier for reviewers to know what it is about. Example:

```
Commit e21d2170f36602ae2708 ("video: remove unnecessary
platform_set_drvdata()") removed the unnecessary
platform_set_drvdata(), but left the variable "dev" unused,
delete it.
```

You should also be sure to use at least the first twelve characters of the SHA-1 ID. The kernel repository holds a *lot* of objects, making collisions with shorter IDs a real possibility. Bear in mind that, even if there is no collision with your six-character ID now, that condition may change five years from now.

If your patch fixes a bug in a specific commit, e.g. you found an issue using `git bisect`, please use the 'Fixes:' tag with the first 12 characters of the SHA-1 ID, and the one line summary. For example:

```
Fixes: e21d2170f366 ("video: remove unnecessary platform_set_drvdata()")
```

The following `git config` settings can be used to add a pretty format for outputting the above style in the `git log` or `git show` commands:

```
[core]
        abbrev = 12
[pretty]
        fixes = Fixes: %h (\"%s\")
```

* 3) Separate your changes

Separate each **logical change** into a separate patch.

For example, if your changes include both bug fixes and performance enhancements for a single driver, separate those changes into two or more patches. If your changes include an API update, and a new driver which uses that new API, separate those into two patches.

On the other hand, if you make a single change to numerous files, group those changes into a single patch. Thus a single logical change is contained within a single patch.

The point to remember is that each patch should make an easily understood change that can be verified by reviewers. Each patch should be justifiable on its own merits.

If one patch depends on another patch in order for a change to be complete, that is OK. Simply note **"this patch depends on patch X"** in your patch description.

When dividing your change into a series of patches, take special care to ensure that the kernel builds and runs properly after each patch in the series. Developers using `git bisect` to track down a problem can end up splitting your patch series at any point; they will not thank you if you introduce bugs in the middle.

If you cannot condense your patch set into a smaller set of patches, then only post say 15 or so at a time and wait for review and integration.

* 4) Style-check your changes

Check your patch for basic style violations, details of which can be found in *Documentation/process/coding-style.rst* . Failure to do so simply wastes the reviewers time and will get your patch rejected, probably without even being read.

One significant exception is when moving code from one file to another – in this case you should not modify the moved code at all in the same patch which moves it. This clearly delineates the act of moving the code and your changes. This greatly aids review of the actual differences and allows tools to better track the history of the code itself.

Check your patches with the patch style checker prior to submission (scripts/checkpatch.pl). Note, though, that the style checker should be viewed as a guide, not as a replacement for human judgment. If your code looks better with a violation then its probably best left alone.

The checker reports at three levels:

- ERROR: things that are very likely to be wrong
- WARNING: things requiring careful review
- CHECK: things requiring thought

You should be able to justify all violations that remain in your patch.

* 5) Select the recipients for your patch

You should always copy the appropriate subsystem maintainer(s) on any patch to code that they maintain; look through the MAINTAINERS file and the source code revision history to see who those maintainers are. The script scripts/get_maintainer.pl can be very useful at this step. If you cannot find a maintainer for the subsystem you are working on, Andrew Morton (akpm@linux-foundation.org) serves as a maintainer of last resort.

You should also normally choose at least one mailing list to receive a copy of your patch set. linux-kernel@vger.kernel.org functions as a list of last resort, but the volume on that list has caused a number of developers to tune it out. Look in the MAINTAINERS file for a subsystem-specific list; your patch will probably get more attention there. Please do not spam unrelated lists, though.

Many kernel-related lists are hosted on vger.kernel.org; you can find a list of them at http://vger.kernel.org/vger-lists.html. There are kernel-related lists hosted elsewhere as well, though.

Do not send more than 15 patches at once to the vger mailing lists!!!

Linus Torvalds is the final arbiter of all changes accepted into the Linux kernel. His e-mail address is <torvalds@linux-foundation.org>. He gets a lot of e-mail, and, at this point, very few patches go through Linus directly, so typically you should do your best to -avoid- sending him e-mail.

If you have a patch that fixes an exploitable security bug, send that patch to security@kernel.org. For severe bugs, a short embargo may be considered to allow distributors to get the patch out to users; in such cases, obviously, the patch should not be sent to any public lists.

Patches that fix a severe bug in a released kernel should be directed toward the stable maintainers by putting a line like this:

```
Cc: stable@vger.kernel.org
```

into the sign-off area of your patch (note, NOT an email recipient). You should also read *Documentation/process/stable-kernel-rules.rst* in addition to this file.

Note, however, that some subsystem maintainers want to come to their own conclusions on which patches should go to the stable trees. The networking maintainer, in particular, would rather not see individual developers adding lines like the above to their patches.

If changes affect userland-kernel interfaces, please send the MAN-PAGES maintainer (as listed in the MAINTAINERS file) a man-pages patch, or at least a notification of the change, so that some information makes its way into the manual pages. User-space API changes should also be copied to linux-api@vger.kernel.org.

For small patches you may want to CC the Trivial Patch Monkey trivial@kernel.org which collects "trivial" patches. Have a look into the MAINTAINERS file for its current manager.

Trivial patches must qualify for one of the following rules:

- Spelling fixes in documentation
- Spelling fixes for errors which could break `grep(1)`
- Warning fixes (cluttering with useless warnings is bad)
- Compilation fixes (only if they are actually correct)
- Runtime fixes (only if they actually fix things)
- Removing use of deprecated functions/macros
- Contact detail and documentation fixes
- Non-portable code replaced by portable code (even in arch-specific, since people copy, as long as it's trivial)
- Any fix by the author/maintainer of the file (ie. patch monkey in re-transmission mode)

* 6) No MIME, no links, no compression, no attachments. Just plain text

Linus and other kernel developers need to be able to read and comment on the changes you are submitting. It is important for a kernel developer to be able to "quote" your changes, using standard e-mail tools, so that they may comment on specific portions of your code.

For this reason, all patches should be submitted by e-mail "inline".

Warning:

Be wary of your editor's word-wrap corrupting your patch, if you choose to cut-n-paste your patch.

Do not attach the patch as a MIME attachment, compressed or not. Many popular e-mail applications will not always transmit a MIME attachment as plain text, making it impossible to comment on your code. A MIME attachment also takes Linus a bit more time to process, decreasing the likelihood of your MIME-attached change being accepted.

Exception: If your mailer is mangling patches then someone may ask you to re-send them using MIME.

See *Documentation/process/email-clients.rst* for hints about configuring your e-mail client so that it sends your patches untouched.

* 7) E-mail size

Large changes are not appropriate for mailing lists, and some maintainers. If your patch, uncompressed, exceeds 300 kB in size, it is preferred that you store your patch on an Internet-accessible server, and provide instead a URL (link) pointing to your patch. But note that if your patch exceeds 300 kB, it almost certainly needs to be broken up anyway.

* 8) Respond to review comments

Your patch will almost certainly get comments from reviewers on ways in which the patch can be improved. You must respond to those comments; ignoring reviewers is a good way to get ignored in return. Review comments or questions that do not lead to a code change should almost certainly bring about a comment or changelog entry so that the next reviewer better understands what is going on.

Be sure to tell the reviewers what changes you are making and to thank them for their time. Code review is a tiring and time-consuming process, and reviewers sometimes get grumpy. Even in that case, though, respond politely and address the problems they have pointed out.

* 9) Don't get discouraged - or impatient

After you have submitted your change, be patient and wait. Reviewers are busy people and may not get to your patch right away.

Once upon a time, patches used to disappear into the void without comment, but the development process works more smoothly than that now. You should receive comments within a week or so; if that does not happen, make sure that you have sent your patches to the right place. Wait for a minimum of one week before resubmitting or pinging reviewers - possibly longer during busy times like merge windows.

* 10) Include PATCH in the subject

Due to high e-mail traffic to Linus, and to linux-kernel, it is common convention to prefix your subject line with [PATCH]. This lets Linus and other kernel developers more easily distinguish patches from other e-mail discussions.

* 11) Sign your work - the Developer's Certificate of Origin

To improve tracking of who did what, especially with patches that can percolate to their final resting place in the kernel through several layers of maintainers, we've introduced a "sign-off" procedure on patches that are being emailed around.

The sign-off is a simple line at the end of the explanation for the patch, which certifies that you wrote it or otherwise have the right to pass it on as an open-source patch. The rules are pretty simple: if you can certify the below:

* Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

1. The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

2. The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or
3. The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.
4. I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

then you just add a line saying:

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

using your real name (sorry, no pseudonyms or anonymous contributions.)

Some people also put extra tags at the end. They'll just be ignored for now, but you can do this to mark internal company procedures or just point out some special detail about the sign-off.

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), 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 submitter's 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.

* 12) When to use Acked-by: and Cc:

The Signed-off-by: tag indicates that the signer was involved in the development of the patch, or that he/she was in the patch's delivery path.

If a person was not directly involved in the preparation or handling of a patch but wishes to signify and record their approval of it then they can ask to have an Acked-by: line added to the patch's changelog.

Acked-by: is often used by the maintainer of the affected code when that maintainer neither contributed to nor forwarded the patch.

Acked-by: is not as formal as Signed-off-by:. It is a record that the acker has at least reviewed the patch and has indicated acceptance. Hence patch mergers will sometimes manually convert an acker's "yep, looks good to me" into an Acked-by: (but note that it is usually better to ask for an explicit ack).

Acked-by: does not necessarily indicate acknowledgement of the entire patch. For example, if a patch affects multiple subsystems and has an Acked-by: from one subsystem maintainer then this usually indicates acknowledgement of just the part which affects that maintainer's code. Judgement should be used here. When in doubt people should refer to the original discussion in the mailing list archives.

If a person has had the opportunity to comment on a patch, but has not provided such comments, you may optionally add a `Cc:` tag to the patch. This is the only tag which might be added without an explicit action by the person it names - but it should indicate that this person was copied on the patch. This tag documents that potentially interested parties have been included in the discussion.

* 13) Using Reported-by:, Tested-by:, Reviewed-by:, Suggested-by: and Fixes:

The Reported-by tag gives credit to people who find bugs and report them and it hopefully inspires them to help us again in the future. Please note that if the bug was reported in private, then ask for permission first before using the Reported-by tag.

A Tested-by: tag indicates that the patch has been successfully tested (in some environment) by the person named. This tag informs maintainers that some testing has been performed, provides a means to locate testers for future patches, and ensures credit for the testers.

Reviewed-by:, instead, indicates that the patch has been reviewed and found acceptable according to the Reviewer's Statement:

* Reviewer's statement of oversight

By offering my Reviewed-by: tag, I state that:

1. I have carried out a technical review of this patch to evaluate its appropriateness and readiness for inclusion into the mainline kernel.
2. Any problems, concerns, or questions relating to the patch have been communicated back to the submitter. I am satisfied with the submitter's response to my comments.
3. While there may be things that could be improved with this submission, I believe that it is, at this time, (1) a worthwhile modification to the kernel, and (2) free of known issues which would argue against its inclusion.
4. While I have reviewed the patch and believe it to be sound, I do not (unless explicitly stated elsewhere) make any warranties or guarantees that it will achieve its stated purpose or function properly in any given situation.

A Reviewed-by tag is a statement of opinion that the patch is an appropriate modification of the kernel without any remaining serious technical issues. Any interested reviewer (who has done the work) can offer a Reviewed-by tag for a patch. This tag serves to give credit to reviewers and to inform maintainers of the degree of review which has been done on the patch. Reviewed-by: tags, when supplied by reviewers known to understand the subject area and to perform thorough reviews, will normally increase the likelihood of your patch getting into the kernel.

A Suggested-by: tag indicates that the patch idea is suggested by the person named and ensures credit to the person for the idea. Please note that this tag should not be added without the reporter's permission, especially if the idea was not posted in a public forum. That said, if we diligently credit our idea reporters, they will, hopefully, be inspired to help us again in the future.

A Fixes: tag indicates that the patch fixes an issue in a previous commit. It is used to make it easy to determine where a bug originated, which can help review a bug fix. This tag also assists the stable kernel team in determining which stable kernel versions should receive your fix. This is the preferred method for indicating a bug fixed by the patch. See *2) Describe your changes* for more details.

* 14) The canonical patch format

This section describes how the patch itself should be formatted. Note that, if you have your patches stored in a `git` repository, proper patch formatting can be had with `git format-patch`. The tools cannot create the necessary text, though, so read the instructions below anyway.

The canonical patch subject line is:

```
Subject: [PATCH 001/123] subsystem: summary phrase
```

The canonical patch message body contains the following:

- A `from` line specifying the patch author (only needed if the person sending the patch is not the author).
- An empty line.
- The body of the explanation, line wrapped at 75 columns, which will be copied to the permanent changelog to describe this patch.
- The `Signed-off-by:` lines, described above, which will also go in the changelog.
- A marker line containing simply `---`.
- Any additional comments not suitable for the changelog.
- The actual patch (`diff` output).

The Subject line format makes it very easy to sort the emails alphabetically by subject line - pretty much any email reader will support that - since because the sequence number is zero-padded, the numerical and alphabetic sort is the same.

The `subsystem` in the email's Subject should identify which area or subsystem of the kernel is being patched.

The `summary phrase` in the email's Subject should concisely describe the patch which that email contains. The `summary phrase` should not be a filename. Do not use the same `summary phrase` for every patch in a whole patch series (where a `patch series` is an ordered sequence of multiple, related patches).

Bear in mind that the `summary phrase` of your email becomes a globally-unique identifier for that patch. It propagates all the way into the `git` changelog. The `summary phrase` may later be used in developer discussions which refer to the patch. People will want to google for the `summary phrase` to read discussion regarding that patch. It will also be the only thing that people may quickly see when, two or three months later, they are going through perhaps thousands of patches using tools such as `gitk` or `git log --oneline`.

For these reasons, the `summary` must be no more than 70-75 characters, and it must describe both what the patch changes, as well as why the patch might be necessary. It is challenging to be both succinct and descriptive, but that is what a well-written summary should do.

The `summary phrase` may be prefixed by tags enclosed in square brackets: "Subject: [PATCH <tag>...] <summary phrase>". The tags are not considered part of the summary phrase, but describe how the patch should be treated. Common tags might include a version descriptor if the multiple versions of the patch have been sent out in response to comments (i.e., "v1, v2, v3"), or "RFC" to indicate a request for comments. If there are four patches in a patch series the individual patches may be numbered like this: 1/4, 2/4, 3/4, 4/4. This assures that developers understand the order in which the patches should be applied and that they have reviewed or applied all of the patches in the patch series.

A couple of example Subjects:

```
Subject: [PATCH 2/5] ext2: improve scalability of bitmap searching
Subject: [PATCH v2 01/27] x86: fix eflags tracking
```

The `from` line must be the very first line in the message body, and has the form:

> From: Original Author <author@example.com>

The `from` line specifies who will be credited as the author of the patch in the permanent changelog. If the `from` line is missing, then the `From:` line from the email header will be used to determine the patch author in the changelog.

The explanation body will be committed to the permanent source changelog, so should make sense to a competent reader who has long since forgotten the immediate details of the discussion that might have led to this patch. Including symptoms of the failure which the patch addresses (kernel log messages, oops messages, etc.) is especially useful for people who might be searching the commit logs looking for the applicable patch. If a patch fixes a compile failure, it may not be necessary to include _all_ of the compile failures; just enough that it is likely that someone searching for the patch can find it. As in the `summary phrase`, it is important to be both succinct as well as descriptive.

The `---` marker line serves the essential purpose of marking for patch handling tools where the changelog message ends.

One good use for the additional comments after the `---` marker is for a `diffstat`, to show what files have changed, and the number of inserted and deleted lines per file. A `diffstat` is especially useful on bigger patches. Other comments relevant only to the moment or the maintainer, not suitable for the permanent changelog, should also go here. A good example of such comments might be `patch changelogs` which describe what has changed between the v1 and v2 version of the patch.

If you are going to include a `diffstat` after the `---` marker, please use `diffstat` options `-p 1 -w 70` so that filenames are listed from the top of the kernel source tree and don't use too much horizontal space (easily fit in 80 columns, maybe with some indentation). (`git` generates appropriate diffstats by default.)

See more details on the proper patch format in the following references.

* 15) Explicit In-Reply-To headers

It can be helpful to manually add In-Reply-To: headers to a patch (e.g., when using `git send-email`) to associate the patch with previous relevant discussion, e.g. to link a bug fix to the email with the bug report. However, for a multi-patch series, it is generally best to avoid using In-Reply-To: to link to older versions of the series. This way multiple versions of the patch don't become an unmanageable forest of references in email clients. If a link is helpful, you can use the https://lkml.kernel.org/ redirector (e.g., in the cover email text) to link to an earlier version of the patch series.

* 16) Sending `git pull` requests

If you have a series of patches, it may be most convenient to have the maintainer pull them directly into the subsystem repository with a `git pull` operation. Note, however, that pulling patches from a developer requires a higher degree of trust than taking patches from a mailing list. As a result, many subsystem maintainers are reluctant to take pull requests, especially from new, unknown developers. If in doubt you can use the pull request as the cover letter for a normal posting of the patch series, giving the maintainer the option of using either.

A pull request should have [GIT] or [PULL] in the subject line. The request itself should include the repository name and the branch of interest on a single line; it should look something like:

```
Please pull from

    git://jdelvare.pck.nerim.net/jdelvare-2.6 i2c-for-linus

to get these changes:
```

A pull request should also include an overall message saying what will be included in the request, a `git shortlog` listing of the patches themselves, and a `diffstat` showing the overall effect of the patch series. The easiest way to get all this information together is, of course, to let `git` do it for you with the `git request-pull` command.

Some maintainers (including Linus) want to see pull requests from signed commits; that increases their confidence that the request actually came from you. Linus, in particular, will not pull from public hosting sites like GitHub in the absence of a signed tag.

The first step toward creating such tags is to make a GNUPG key and get it signed by one or more core kernel developers. This step can be hard for new developers, but there is no way around it. Attending conferences can be a good way to find developers who can sign your key.

Once you have prepared a patch series in `git` that you wish to have somebody pull, create a signed tag with `git tag -s`. This will create a new tag identifying the last commit in the series and containing a signature created with your private key. You will also have the opportunity to add a changelog-style message to the tag; this is an ideal place to describe the effects of the pull request as a whole.

If the tree the maintainer will be pulling from is not the repository you are working from, don't forget to push the signed tag explicitly to the public tree.

When generating your pull request, use the signed tag as the target. A command like this will do the trick:

```
git request-pull master git://my.public.tree/linux.git my-signed-tag
```

* References

Andrew Morton, "The perfect patch" (tpp). <http://www.ozlabs.org/~akpm/stuff/tpp.txt>

Jeff Garzik, "Linux kernel patch submission format". <http://linux.yyz.us/patch-format.html>

Greg Kroah-Hartman, "How to piss off a kernel subsystem maintainer". <http://www.kroah.com/log/linux/maintainer.html>

<http://www.kroah.com/log/linux/maintainer-02.html>

<http://www.kroah.com/log/linux/maintainer-03.html>

<http://www.kroah.com/log/linux/maintainer-04.html>

<http://www.kroah.com/log/linux/maintainer-05.html>

<http://www.kroah.com/log/linux/maintainer-06.html>

NO!!!! No more huge patch bombs to linux-kernel@vger.kernel.org people! <https://lkml.org/lkml/2005/7/11/336>

Kernel Documentation/process/coding-style.rst: *Documentation/process/coding-style.rst*

Linus Torvalds's mail on the canonical patch format: <http://lkml.org/lkml/2005/4/7/183>

Andi Kleen, "On submitting kernel patches" Some strategies to get difficult or controversial changes in.

http://halobates.de/on-submitting-patches.pdf

CHAPTER

FIVE

LINUX KERNEL CODING STYLE

This is a short document describing the preferred coding style for the linux kernel. Coding style is very personal, and I won't **force** my views on anybody, but this is what goes for anything that I have to be able to maintain, and I'd prefer it for most other things too. Please at least consider the points made here.

First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.

Anyway, here goes:

* 1) Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

In short, 8-char indents make things easier to read, and have the added benefit of warning you when you're nesting your functions too deep. Heed that warning.

The preferred way to ease multiple indentation levels in a switch statement is to align the `switch` and its subordinate `case` labels in the same column instead of `double-indenting` the `case` labels. E.g.:

```
switch (suffix) {
case 'G':
case 'g':
        mem <<= 30;
        break;
case 'M':
case 'm':
        mem <<= 20;
        break;
case 'K':
case 'k':
        mem <<= 10;
        /* fall through */
default:
        break;
}
```

Don't put multiple statements on a single line unless you have something to hide:

```
if (condition) do_this;
  do_something_everytime;
```

Don't put multiple assignments on a single line either. Kernel coding style is super simple. Avoid tricky expressions.

Outside of comments, documentation and except in Kconfig, spaces are never used for indentation, and the above example is deliberately broken.

Get a decent editor and don't leave whitespace at the end of lines.

* 2) Breaking long lines and strings

Coding style is all about readability and maintainability using commonly available tools.

The limit on the length of lines is 80 columns and this is a strongly preferred limit.

Statements longer than 80 columns will be broken into sensible chunks, unless exceeding 80 columns significantly increases readability and does not hide information. Descendants are always substantially shorter than the parent and are placed substantially to the right. The same applies to function headers with a long argument list. However, never break user-visible strings such as printk messages, because that breaks the ability to grep for them.

* 3) Placing Braces and Spaces

The other issue that always comes up in C styling is the placement of braces. Unlike the indent size, there are few technical reasons to choose one placement strategy over the other, but the preferred way, as shown to us by the prophets Kernighan and Ritchie, is to put the opening brace last on the line, and put the closing brace first, thusly:

```
if (x is true) {
        we do y
}
```

This applies to all non-function statement blocks (if, switch, for, while, do). E.g.:

```
switch (action) {
case KOBJ_ADD:
        return "add";
case KOBJ_REMOVE:
        return "remove";
case KOBJ_CHANGE:
        return "change";
default:
        return NULL;
}
```

However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)
{
        body of function
}
```

Heretic people all over the world have claimed that this inconsistency is ... well ... inconsistent, but all right-thinking people know that (a) K&R are **right** and (b) K&R are right. Besides, functions are special anyway (you can't nest them in C).

Note that the closing brace is empty on a line of its own, **except** in the cases where it is followed by a continuation of the same statement, ie a `while` in a do-statement or an `else` in an if-statement, like this:

```
do {
        body of do-loop
} while (condition);
```

and

```
if (x == y) {
        ..
} else if (x > y) {
        ...
} else {
        ....
}
```

Rationale: K&R.

Also, note that this brace-placement also minimizes the number of empty (or almost empty) lines, without any loss of readability. Thus, as the supply of new-lines on your screen is not a renewable resource (think 25-line terminal screens here), you have more empty lines to put comments on.

Do not unnecessarily use braces where a single statement will do.

```
if (condition)
        action();
```

and

```
if (condition)
        do_this();
else
        do_that();
```

This does not apply if only one branch of a conditional statement is a single statement; in the latter case use braces in both branches:

```
if (condition) {
        do_this();
        do_that();
} else {
        otherwise();
}
```

* 3.1) Spaces

Linux kernel style for use of spaces depends (mostly) on function-versus-keyword usage. Use a space after (most) keywords. The notable exceptions are sizeof, typeof, alignof, and __attribute__, which look somewhat like functions (and are usually used with parentheses in Linux, although they are not required in the language, as in: `sizeof info` after `struct fileinfo info;` is declared).

So use a space after these keywords:

```
if, switch, case, for, do, while
```

but not with sizeof, typeof, alignof, or __attribute__. E.g.,

```
s = sizeof(struct file);
```

Do not add spaces around (inside) parenthesized expressions. This example is **bad**:

```
s = sizeof( struct file );
```

When declaring pointer data or a function that returns a pointer type, the preferred use of * is adjacent to the data name or function name and not adjacent to the type name. Examples:

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

Use one space around (on each side of) most binary and ternary operators, such as any of these:

```
=  +  -  <  >  *  /  %  |  &  ^  <=  >=  ==  !=  ?  :
```

but no space after unary operators:

```
&  *  +  -  ~  !  sizeof  typeof  alignof  __attribute__  defined
```

no space before the postfix increment & decrement unary operators:

```
++  --
```

no space after the prefix increment & decrement unary operators:

```
++  --
```

and no space around the `.` and `->` structure member operators.

Do not leave trailing whitespace at the ends of lines. Some editors with `smart` indentation will insert whitespace at the beginning of new lines as appropriate, so you can start typing the next line of code right away. However, some such editors do not remove the whitespace if you end up not putting a line of code there, such as if you leave a blank line. As a result, you end up with lines containing trailing whitespace.

Git will warn you about patches that introduce trailing whitespace, and can optionally strip the trailing whitespace for you; however, if applying a series of patches, this may make later patches in the series fail by changing their context lines.

* 4) Naming

C is a Spartan language, and so should your naming be. Unlike Modula-2 and Pascal programmers, C programmers do not use cute names like ThisVariableIsATemporaryCounter. A C programmer would call that variable `tmp`, which is much easier to write, and not the least more difficult to understand.

HOWEVER, while mixed-case names are frowned upon, descriptive names for global variables are a must. To call a global function `foo` is a shooting offense.

GLOBAL variables (to be used only if you **really** need them) need to have descriptive names, as do global functions. If you have a function that counts the number of active users, you should call that `count_active_users()` or similar, you should **not** call it `cntusr()`.

Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged - the compiler knows the types anyway and can check those, and it only confuses the programmer. No wonder MicroSoft makes buggy programs.

LOCAL variable names should be short, and to the point. If you have some random integer loop counter, it should probably be called `i`. Calling it `loop_counter` is non-productive, if there is no chance of it being mis-understood. Similarly, `tmp` can be just about any type of variable that is used to hold a temporary value.

If you are afraid to mix up your local variable names, you have another problem, which is called the function-growth-hormone-imbalance syndrome. See chapter 6 (Functions).

* 5) Typedefs

Please don't use things like `vps_t`. It's a **mistake** to use typedef for structures and pointers. When you see a

```
vps_t a;
```

in the source, what does it mean? In contrast, if it says

```
struct virtual_container *a;
```

you can actually tell what a is.

Lots of people think that typedefs `help readability`. Not so. They are useful only for:

1. totally opaque objects (where the typedef is actively used to **hide** what the object is).

   Example: `pte_t` etc. opaque objects that you can only access using the proper accessor functions.

   > ***Note:***
   >
   > *Opaqueness and `accessor functions` are not good in themselves. The reason we have them for things like pte_t etc. is that there really is absolutely **zero** portably accessible information there.*

2. Clear integer types, where the abstraction **helps** avoid confusion whether it is `int` or `long`.

   u8/u16/u32 are perfectly fine typedefs, although they fit into category (d) better than here.

   > ***Note:***
   >
   > *Again - there needs to be a **reason** for this. If something is `unsigned long`, then there's no reason to do*
   >
   > *typedef unsigned long myflags_t;*

   but if there is a clear reason for why it under certain circumstances might be an `unsigned int` and under other configurations might be `unsigned long`, then by all means go ahead and use a typedef.

3. when you use sparse to literally create a **new** type for type-checking.

4. New types which are identical to standard C99 types, in certain exceptional circumstances.

   Although it would only take a short amount of time for the eyes and brain to become accustomed to the standard types like `uint32_t`, some people object to their use anyway.

   Therefore, the Linux-specific `u8/u16/u32/u64` types and their signed equivalents which are identical to standard types are permitted – although they are not mandatory in new code of your own.

   When editing existing code which already uses one or the other set of types, you should conform to the existing choices in that code.

5. Types safe for use in userspace.

   In certain structures which are visible to userspace, we cannot require C99 types and cannot use the u32 form above. Thus, we use __u32 and similar types in all structures which are shared with userspace.

Maybe there are other cases too, but the rule should basically be to NEVER EVER use a typedef unless you can clearly match one of those rules.

In general, a pointer, or a struct that has elements that can reasonably be directly accessed should **never** be a typedef.

* 6) Functions

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.

However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can ask the compiler to in-line them if you think it's performance-critical, and it will probably do a better job of it than you would have done).

Another measure of the function is the number of local variables. They shouldn't exceed 5-10, or you're doing something wrong. Re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know you're brilliant, but maybe you'd like to understand what you did 2 weeks from now.

In source files, separate functions with one blank line. If the function is exported, the **EXPORT** macro for it should follow immediately after the closing function brace line. E.g.:

```
int system_is_up(void)
{
        return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

In function prototypes, include parameter names with their data types. Although this is not required by the C language, it is preferred in Linux because it is a simple way to add valuable information for the reader.

* 7) Centralized exiting of functions

Albeit deprecated by some people, the equivalent of the goto statement is used frequently by compilers in form of the unconditional jump instruction.

The goto statement comes in handy when a function exits from multiple locations and some common work such as cleanup has to be done. If there is no cleanup needed then just return directly.

Choose label names which say what the goto does or why the goto exists. An example of a good name could be `out_free_buffer:` if the goto frees `buffer`. Avoid using GW-BASIC names like `err1:` and `err2:`, as you would have to renumber them if you ever add or remove exit paths, and they make correctness difficult to verify anyway.

The rationale for using gotos is:

- unconditional statements are easier to understand and follow
- nesting is reduced
- errors by not updating individual exit points when making modifications are prevented
- saves the compiler work to optimize redundant code away ;)

```
int fun(int a)
{
        int result = 0;
        char *buffer;

        buffer = kmalloc(SIZE, GFP_KERNEL);
        if (!buffer)
                return -ENOMEM;

        if (condition1) {
                while (loop1) {
                        ...
                }
                result = 1;
                goto out_free_buffer;
        }
        ...
out_free_buffer:
        kfree(buffer);
        return result;
}
```

A common type of bug to be aware of is one `err` bugs which look like this:

```
err:
        kfree(foo->bar);
        kfree(foo);
        return ret;
```

The bug in this code is that on some exit paths `foo` is NULL. Normally the fix for this is to split it up into two error labels `err_free_bar:` and `err_free_foo:`:

```
err_free_bar:
       kfree(foo->bar);
err_free_foo:
       kfree(foo);
       return ret;
```

Ideally you should simulate errors to test all exit paths.

* 8) Commenting

Comments are good, but there is also a danger of over-commenting. NEVER try to explain HOW your code works in a comment: it's much better to write the code so that the **working** is obvious, and it's a waste of time to explain badly written code.

Generally, you want your comments to tell WHAT your code does, not HOW. Also, try to avoid putting comments inside a function body: if the function is so complex that you need to separately comment parts of it, you should probably go back to chapter 6 for a while. You can make small comments to note or warn about something particularly clever (or ugly), but try to avoid excess. Instead, put the comments at the head of the function, telling people what it does, and possibly WHY it does it.

When commenting the kernel API functions, please use the kernel-doc format. See the files at Documentation/doc-guide/ and `scripts/kernel-doc` for details.

The preferred style for long (multi-line) comments is:

```
/*
 * This is the preferred style for multi-line
 * comments in the Linux kernel source code.
 * Please use it consistently.
 *
 * Description:  A column of asterisks on the left side,
 * with beginning and ending almost-blank lines.
 */
```

For files in net/ and drivers/net/ the preferred style for long (multi-line) comments is a little different.

```
/* The preferred comment style for files in net/ and drivers/net
 * looks like this.
 *
 * It is nearly the same as the generally preferred comment style,
 * but there is no initial almost-blank line.
 */
```

It's also important to comment data, whether they are basic types or derived types. To this end, use just one data declaration per line (no commas for multiple data declarations). This leaves you room for a small comment on each item, explaining its use.

* 9) You've made a mess of it

That's OK, we all do. You've probably been told by your long-time Unix user helper that `GNU emacs` automatically formats the C sources for you, and you've noticed that yes, it does do that, but the defaults it uses are less than desirable (in fact, they are worse than random typing - an infinite number of monkeys typing into GNU emacs would never make a good program).

So, you can either get rid of GNU emacs, or change it to use saner values. To do the latter, you can stick the following in your .emacs file:

```
(defun c-lineup-arglist-tabs-only (ignored)
  "Line up argument lists by tabs, not spaces"
  (let* ((anchor (c-langelem-pos c-syntactic-element))
         (column (c-langelem-2nd-pos c-syntactic-element))
         (offset (- (1+ column) anchor))
         (steps (floor offset c-basic-offset)))
    (* (max steps 1)
       c-basic-offset)))

(add-hook 'c-mode-common-hook
          (lambda ()
            ;; Add kernel style
            (c-add-style
             "linux-tabs-only"
             '("linux" (c-offsets-alist
                        (arglist-cont-nonempty
                         c-lineup-gcc-asm-reg
                         c-lineup-arglist-tabs-only))))))

(add-hook 'c-mode-hook
          (lambda ()
            (let ((filename (buffer-file-name)))
              ;; Enable kernel mode for the appropriate files
              (when (and filename
                         (string-match (expand-file-name "~/src/linux-trees")
                                       filename))
                (setq indent-tabs-mode t)
                (setq show-trailing-whitespace t)
                (c-set-style "linux-tabs-only")))))
```

This will make emacs go better with the kernel coding style for C files below `~/src/linux-trees`.

But even if you fail in getting emacs to do sane formatting, not everything is lost: use `indent`.

Now, again, GNU indent has the same brain-dead settings that GNU emacs has, which is why you need to give it a few command line options. However, that's not too bad, because even the makers of GNU indent recognize the authority of K&R (the GNU people aren't evil, they are just severely misguided in this matter), so you just give indent the options `-kr -i8` (stands for K&R,`8 character indents`), or use `scripts/Lindent`, which indents in the latest style.

`indent` has a lot of options, and especially when it comes to comment re-formatting you may want to take a look at the man page. But remember: `indent` is not a fix for bad programming.

* 10) Kconfig configuration files

For all of the Kconfig* configuration files throughout the source tree, the indentation is somewhat different. Lines under a `config` definition are indented with one tab, while help text is indented an additional two spaces. Example:

```
config AUDIT
	bool "Auditing support"
	depends on NET
	help
	  Enable auditing infrastructure that can be used with another
	  kernel subsystem, such as SELinux (which requires this for
	  logging of avc messages output).  Does not do system-call
	  auditing without CONFIG_AUDITSYSCALL.
```

Seriously dangerous features (such as write support for certain filesystems) should advertise this prominently in their prompt string:

```
config ADFS_FS_RW
	bool "ADFS write support (DANGEROUS)"
	depends on ADFS_FS
	...
```

For full documentation on the configuration files, see the file Documentation/kbuild/kconfig-language.txt.

* 11) Data structures

Data structures that have visibility outside the single-threaded environment they are created and destroyed in should always have reference counts. In the kernel, garbage collection doesn't exist (and outside the kernel garbage collection is slow and inefficient), which means that you absolutely **have** to reference count all your uses.

Reference counting means that you can avoid locking, and allows multiple users to have access to the data structure in parallel - and not having to worry about the structure suddenly going away from under them just because they slept or did something else for a while.

Note that locking is **not** a replacement for reference counting. Locking is used to keep data structures coherent, while reference counting is a memory management technique. Usually both are needed, and they are not to be confused with each other.

Many data structures can indeed have two levels of reference counting, when there are users of different `classes`. The subclass count counts the number of subclass users, and decrements the global count just once when the subclass count goes to zero.

Examples of this kind of `multi-level-reference-counting` can be found in memory management (`struct mm_struct`: mm_users and mm_count), and in filesystem code (`struct super_block`: s_count and s_active).

Remember: if another thread can find your data structure, and you don't have a reference count on it, you almost certainly have a bug.

* 12) Macros, Enums and RTL

Names of macros defining constants and labels in enums are capitalized.

```
#define CONSTANT 0x12345
```

Enums are preferred when defining several related constants.

CAPITALIZED macro names are appreciated but macros resembling functions may be named in lower case.

Generally, inline functions are preferable to macros resembling functions.

Macros with multiple statements should be enclosed in a do - while block:

```
#define macrofun(a, b, c)                       \
        do {                                    \
                if (a == 5)                     \
                        do_this(b, c);          \
        } while (0)
```

Things to avoid when using macros:

1. macros that affect control flow:

```
#define FOO(x)                                  \
        do {                                    \
                if (blah(x) < 0)                \
                        return -EBUGGERED;      \
        } while (0)
```

is a **very** bad idea. It looks like a function call but exits the `calling` function; don't break the internal parsers of those who will read the code.

2. macros that depend on having a local variable with a magic name:

```
#define FOO(val) bar(index, val)
```

might look like a good thing, but it's confusing as hell when one reads the code and it's prone to breakage from seemingly innocent changes.

3) macros with arguments that are used as l-values: FOO(x) = y; will bite you if somebody e.g. turns FOO into an inline function.

4) forgetting about precedence: macros defining constants using expressions must enclose the expression in parentheses. Beware of similar issues with macros using parameters.

```
#define CONSTANT 0x4000
#define CONSTEXP (CONSTANT | 3)
```

5) namespace collisions when defining local variables in macros resembling functions:

```
#define FOO(x)                          \
({                                      \
        typeof(x) ret;                  \
        ret = calc_ret(x);              \
        (ret);                          \
})
```

ret is a common name for a local variable - __foo_ret is less likely to collide with an existing variable.

The cpp manual deals with macros exhaustively. The gcc internals manual also covers RTL which is used frequently with assembly language in the kernel.

* 13) Printing kernel messages

Kernel developers like to be seen as literate. Do mind the spelling of kernel messages to make a good impression. Do not use crippled words like `dont`; use `do not` or `don't` instead. Make the messages concise, clear, and unambiguous.

Kernel messages do not have to be terminated with a period.

Printing numbers in parentheses (%d) adds no value and should be avoided.

There are a number of driver model diagnostic macros in <linux/device.h> which you should use to make sure messages are matched to the right device and driver, and are tagged with the right level: dev_err(), dev_warn(), dev_info(), and so forth. For messages that aren't associated with a particular device, <linux/printk.h> defines pr_notice(), pr_info(), pr_warn(), pr_err(), etc.

Coming up with good debugging messages can be quite a challenge; and once you have them, they can be a huge help for remote troubleshooting. However debug message printing is handled differently than printing other non-debug messages. While the other pr_XXX() functions print unconditionally, pr_debug() does not; it is compiled out by default, unless either DEBUG is defined or CONFIG_DYNAMIC_DEBUG is set. That is true for dev_dbg() also, and a related convention uses VERBOSE_DEBUG to add dev_vdbg() messages to the ones already enabled by DEBUG.

Many subsystems have Kconfig debug options to turn on -DDEBUG in the corresponding Makefile; in other cases specific files #define DEBUG. And when a debug message should be unconditionally printed, such as if it is already inside a debug-related #ifdef section, printk(KERN_DEBUG ...) can be used.

* 14) Allocating memory

The kernel provides the following general purpose memory allocators: kmalloc(), kzalloc(), kmalloc_array(), kcalloc(), vmalloc(), and vzalloc(). Please refer to the API documentation for further information about them.

The preferred form for passing a size of a struct is the following:

```
p = kmalloc(sizeof(*p), ...);
```

The alternative form where struct name is spelled out hurts readability and introduces an opportunity for a bug when the pointer variable type is changed but the corresponding sizeof that is passed to a memory allocator is not.

Casting the return value which is a void pointer is redundant. The conversion from void pointer to any other pointer type is guaranteed by the C programming language.

The preferred form for allocating an array is the following:

```
p = kmalloc_array(n, sizeof(...), ...);
```

The preferred form for allocating a zeroed array is the following:

```
p = kcalloc(n, sizeof(...), ...);
```

Both forms check for overflow on the allocation size n * sizeof(...), and return NULL if that occurred.

* 15) The inline disease

There appears to be a common misperception that gcc has a magic "make me faster" speedup option called `inline`. While the use of inlines can be appropriate (for example as a means of replacing macros, see Chapter 12), it very often is not. Abundant use of the inline keyword leads to a much bigger kernel, which in turn slows the system as a whole down, due to a bigger icache footprint for the CPU and simply because there is less memory available for the pagecache. Just think about it; a pagecache miss causes a disk seek, which easily takes 5 milliseconds. There are a LOT of cpu cycles that can go into these 5 milliseconds.

A reasonable rule of thumb is to not put inline at functions that have more than 3 lines of code in them. An exception to this rule are the cases where a parameter is known to be a compiletime constant, and as a result of this constantness you *know* the compiler will be able to optimize most of your function away at compile time. For a good example of this later case, see the kmalloc() inline function.

Often people argue that adding inline to functions that are static and used only once is always a win since there is no space tradeoff. While this is technically correct, gcc is capable of inlining these automatically without help, and the maintenance issue of removing the inline when a second user appears outweighs the potential value of the hint that tells gcc to do something it would have done anyway.

* 16) Function return values and names

Functions can return values of many different kinds, and one of the most common is a value indicating whether the function succeeded or failed. Such a value can be represented as an error-code integer (-Exxx = failure, 0 = success) or a succeeded boolean (0 = failure, non-zero = success).

Mixing up these two sorts of representations is a fertile source of difficult-to-find bugs. If the C language included a strong distinction between integers and booleans then the compiler would find these mistakes for us... but it doesn't. To help prevent such bugs, always follow this convention:

```
If the name of a function is an action or an imperative command,
the function should return an error-code integer.  If the name
is a predicate, the function should return a "succeeded" boolean.
```

For example, add `work` is a command, and the add_work() function returns 0 for success or -EBUSY for failure. In the same way, PCI `device present` is a predicate, and the pci_dev_present() function returns 1 if it succeeds in finding a matching device or 0 if it doesn't.

All EXPORTed functions must respect this convention, and so should all public functions. Private (static) functions need not, but it is recommended that they do.

Functions whose return value is the actual result of a computation, rather than an indication of whether the computation succeeded, are not subject to this rule. Generally they indicate failure by returning some out-of-range result. Typical examples would be functions that return pointers; they use NULL or the ERR_PTR mechanism to report failure.

* 17) Don't re-invent the kernel macros

The header file include/linux/kernel.h contains a number of macros that you should use, rather than explicitly coding some variant of them yourself. For example, if you need to calculate the length of an array, take advantage of the macro

```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

Similarly, if you need to calculate the size of some structure member, use

```
#define FIELD_SIZEOF(t, f) (sizeof(((t*)0)->f))
```

There are also min() and max() macros that do strict type checking if you need them. Feel free to peruse that header file to see what else is already defined that you shouldn't reproduce in your code.

* 18) Editor modelines and other cruft

Some editors can interpret configuration information embedded in source files, indicated with special markers. For example, emacs interprets lines marked like this:

```
-*- mode: c -*-
```

Or like this:

```
/*
Local Variables:
compile-command: "gcc -DMAGIC_DEBUG_FLAG foo.c"
End:
*/
```

Vim interprets markers that look like this:

```
/* vim:set sw=8 noet */
```

Do not include any of these in source files. People have their own personal editor configurations, and your source files should not override them. This includes markers for indentation and mode configuration. People may use their own custom mode, or may have some other magic method for making indentation work correctly.

* 19) Inline assembly

In architecture-specific code, you may need to use inline assembly to interface with CPU or platform functionality. Don't hesitate to do so when necessary. However, don't use inline assembly gratuitously when C can do the job. You can and should poke hardware from C when possible.

Consider writing simple helper functions that wrap common bits of inline assembly, rather than repeatedly writing them with slight variations. Remember that inline assembly can use C parameters.

Large, non-trivial assembly functions should go in .S files, with corresponding C prototypes defined in C header files. The C prototypes for assembly functions should use `asmlinkage`.

You may need to mark your asm statement as volatile, to prevent GCC from removing it if GCC doesn't notice any side effects. You don't always need to do so, though, and doing so unnecessarily can limit optimization.

When writing a single inline assembly statement containing multiple instructions, put each instruction on a separate line in a separate quoted string, and end each string except the last with `\n\t` to properly indent the next instruction in the assembly output:

```
asm ("magic %reg1, #42\n\t"
     "more_magic %reg2, %reg3"
     : /* outputs */ : /* inputs */ : /* clobbers */);
```

* 20) Conditional Compilation

Wherever possible, don't use preprocessor conditionals (#if, #ifdef) in .c files; doing so makes code harder to read and logic harder to follow. Instead, use such conditionals in a header file defining functions for use in those .c files, providing no-op stub versions in the #else case, and then call those functions unconditionally from .c files. The compiler will avoid generating any code for the stub calls, producing identical results, but the logic will remain easy to follow.

Prefer to compile out entire functions, rather than portions of functions or portions of expressions. Rather than putting an ifdef in an expression, factor out part or all of the expression into a separate helper function and apply the conditional to that function.

If you have a function or variable which may potentially go unused in a particular configuration, and the compiler would warn about its definition going unused, mark the definition as __maybe_unused rather than wrapping it in a preprocessor conditional. (However, if a function or variable *always* goes unused, delete it.)

Within code, where possible, use the IS_ENABLED macro to convert a Kconfig symbol into a C boolean expression, and use it in a normal C conditional:

```
if (IS_ENABLED(CONFIG_SOMETHING)) {
        ...
}
```

The compiler will constant-fold the conditional away, and include or exclude the block of code just as with an #ifdef, so this will not add any runtime overhead. However, this approach still allows the C compiler to see the code inside the block, and check it for correctness (syntax, types, symbol references, etc). Thus, you still have to use an #ifdef if the code inside the block references symbols that will not exist if the condition is not met.

At the end of any non-trivial #if or #ifdef block (more than a few lines), place a comment after the #endif on the same line, noting the conditional expression used. For instance:

```
#ifdef CONFIG_SOMETHING
...
#endif /* CONFIG_SOMETHING */
```

* Appendix I) References

The C Programming Language, Second Edition by Brian W. Kernighan and Dennis M. Ritchie. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8 (paperback), 0-13-110370-9 (hardback).

The Practice of Programming by Brian W. Kernighan and Rob Pike. Addison-Wesley, Inc., 1999. ISBN 0-201-61586-X.

GNU manuals - where in compliance with K&R and this text - for cpp, gcc, gcc internals and indent, all available from http://www.gnu.org/manual/

WG14 is the international standardization working group for the programming language C, URL: http://www.open-std.org/JTC1/SC22/WG14/

Kernel process/coding-style.rst, by greg@kroah.com at OLS 2002: http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/

CHAPTER

SIX

EMAIL CLIENTS INFO FOR LINUX

* Git

These days most developers use `git send-email` instead of regular email clients. The man page for this is quite good. On the receiving end, maintainers use `git am` to apply the patches.

If you are new to `git` then send your first patch to yourself. Save it as raw text including all the headers. Run `git am raw_email.txt` and then review the changelog with `git log`. When that works then send the patch to the appropriate mailing list(s).

* General Preferences

Patches for the Linux kernel are submitted via email, preferably as inline text in the body of the email. Some maintainers accept attachments, but then the attachments should have content-type `text/plain`. However, attachments are generally frowned upon because it makes quoting portions of the patch more difficult in the patch review process.

Email clients that are used for Linux kernel patches should send the patch text untouched. For example, they should not modify or delete tabs or spaces, even at the beginning or end of lines.

Don't send patches with `format=flowed`. This can cause unexpected and unwanted line breaks.

Don't let your email client do automatic word wrapping for you. This can also corrupt your patch.

Email clients should not modify the character set encoding of the text. Emailed patches should be in ASCII or UTF-8 encoding only. If you configure your email client to send emails with UTF-8 encoding, you avoid some possible charset problems.

Email clients should generate and maintain References: or In-Reply-To: headers so that mail threading is not broken.

Copy-and-paste (or cut-and-paste) usually does not work for patches because tabs are converted to spaces. Using xclipboard, xclip, and/or xcutsel may work, but it's best to test this for yourself or just avoid copy-and-paste.

Don't use PGP/GPG signatures in mail that contains patches. This breaks many scripts that read and apply the patches. (This should be fixable.)

It's a good idea to send a patch to yourself, save the received message, and successfully apply it with 'patch' before sending patches to Linux mailing lists.

* Some email client (MUA) hints

Here are some specific MUA configuration hints for editing and sending patches for the Linux kernel. These are not meant to be complete software package configuration summaries.

Legend:

- TUI = text-based user interface
- GUI = graphical user interface

* Alpine (TUI)

Config options:

In the *Sending Preferences* section:

- *Do Not Send Flowed Text* must be `enabled`
- *Strip Whitespace Before Sending* must be `disabled`

When composing the message, the cursor should be placed where the patch should appear, and then pressing `CTRL-R` let you specify the patch file to insert into the message.

* Claws Mail (GUI)

Works. Some people use this successfully for patches.

To insert a patch use *Message→Insert* File (`CTRL-I`) or an external editor.

If the inserted patch has to be edited in the Claws composition window "Auto wrapping" in *Configuration→Preferences→Compose→Wrapping* should be disabled.

* Evolution (GUI)

Some people use this successfully for patches.

When composing mail select: Preformat from *Format→Paragraph Style→Preformatted* (`CTRL-7`) or the toolbar

Then use: *Insert→Text File...* (`ALT-N x`) to insert the patch.

You can also `diff -Nru old.c new.c | xclip`, select *Preformat*, then paste with the middle button.

* Kmail (GUI)

Some people use Kmail successfully for patches.

The default setting of not composing in HTML is appropriate; do not enable it.

When composing an email, under options, uncheck "word wrap". The only disadvantage is any text you type in the email will not be word-wrapped so you will have to manually word wrap text before the patch. The easiest way around this is to compose your email with word wrap enabled, then save it as a draft. Once you pull it up again from your drafts it is now hard word-wrapped and you can uncheck "word wrap" without losing the existing wrapping.

At the bottom of your email, put the commonly-used patch delimiter before inserting your patch: three hyphens (`---`).

Then from the *Message* menu item, select insert file and choose your patch. As an added bonus you can customise the message creation toolbar menu and put the *insert file* icon there.

Make the composer window wide enough so that no lines wrap. As of KMail 1.13.5 (KDE 4.5.4), KMail will apply word wrapping when sending the email if the lines wrap in the composer window. Having word wrapping disabled in the Options menu isn't enough. Thus, if your patch has very long lines, you must make the composer window very wide before sending the email. See: https://bugs.kde.org/show_bug.cgi?id=174034

You can safely GPG sign attachments, but inlined text is preferred for patches so do not GPG sign them. Signing patches that have been inserted as inlined text will make them tricky to extract from their 7-bit encoding.

If you absolutely must send patches as attachments instead of inlining them as text, right click on the attachment and select properties, and highlight *Suggest automatic display* to make the attachment inlined to make it more viewable.

When saving patches that are sent as inlined text, select the email that contains the patch from the message list pane, right click and select *save as*. You can use the whole email unmodified as a patch if it was properly composed. There is no option currently to save the email when you are actually viewing it in its own window – there has been a request filed at kmail's bugzilla and hopefully this will be addressed. Emails are saved as read-write for user only so you will have to chmod them to make them group and world readable if you copy them elsewhere.

* Lotus Notes (GUI)

Run away from it.

* IBM Verse (Web GUI)

See Lotus Notes.

* Mutt (TUI)

Plenty of Linux developers use `mutt`, so it must work pretty well.

Mutt doesn't come with an editor, so whatever editor you use should be used in a way that there are no automatic linebreaks. Most editors have an *insert file* option that inserts the contents of a file unaltered.

To use `vim` with mutt:

```
set editor="vi"
```

If using xclip, type the command:

```
:set paste
```

before middle button or shift-insert or use:

```
:r filename
```

if you want to include the patch inline. (a)ttach works fine without `set paste`.

You can also generate patches with `git format-patch` and then use Mutt to send them:

```
$ mutt -H 0001-some-bug-fix.patch
```

Config options:

It should work with default settings. However, it's a good idea to set the `send_charset` to:

```
set send_charset="us-ascii:utf-8"
```

Mutt is highly customizable. Here is a minimum configuration to start using Mutt to send patches through Gmail:

```
# .muttrc
# ================  IMAP ====================
set imap_user = 'yourusername@gmail.com'
set imap_pass = 'yourpassword'
set spoolfile = imaps://imap.gmail.com/INBOX
set folder = imaps://imap.gmail.com/
set record="imaps://imap.gmail.com/[Gmail]/Sent Mail"
set postponed="imaps://imap.gmail.com/[Gmail]/Drafts"
set mbox="imaps://imap.gmail.com/[Gmail]/All Mail"

# ================  SMTP  ====================
set smtp_url = "smtp://username@smtp.gmail.com:587/"
set smtp_pass = $imap_pass
set ssl_force_tls = yes # Require encrypted connection

# ================  Composition  ====================
set editor = `echo \$EDITOR`
set edit_headers = yes  # See the headers when editing
set charset = UTF-8     # value of $LANG; also fallback for send_charset
# Sender, email address, and sign-off line must match
unset use_domain        # because joe@localhost is just embarrassing
set realname = "YOUR NAME"
set from = "username@gmail.com"
set use_from = yes
```

The Mutt docs have lots more information:

> http://dev.mutt.org/trac/wiki/UseCases/Gmail
>
> http://dev.mutt.org/doc/manual.html

* Pine (TUI)

Pine has had some whitespace truncation issues in the past, but these should all be fixed now.

Use alpine (pine's successor) if you can.

Config options:

- `quell-flowed-text` is needed for recent versions
- the `no-strip-whitespace-before-send` option is needed

* Sylpheed (GUI)

- Works well for inlining text (or using attachments).
- Allows use of an external editor.
- Is slow on large folders.
- Won't do TLS SMTP auth over a non-SSL connection.
- Has a helpful ruler bar in the compose window.
- Adding addresses to address book doesn't understand the display name properly.

* Thunderbird (GUI)

Thunderbird is an Outlook clone that likes to mangle text, but there are ways to coerce it into behaving.

- Allow use of an external editor: The easiest thing to do with Thunderbird and patches is to use an "external editor" extension and then just use your favorite `$EDITOR` for reading/merging patches into the body text. To do this, download and install the extension, then add a button for it using *View→Toolbars→Customize...* and finally just click on it when in the *Compose* dialog.

  Please note that "external editor" requires that your editor must not fork, or in other words, the editor must not return before closing. You may have to pass additional flags or change the settings of your editor. Most notably if you are using gvim then you must pass the -f option to gvim by putting `/usr/bin/gvim -f` (if the binary is in `/usr/bin`) to the text editor field in *external editor* settings. If you are using some other editor then please read its manual to find out how to do this.

To beat some sense out of the internal editor, do this:

- Edit your Thunderbird config settings so that it won't use `format=flowed`. Go to *edit→preferences→advanced→config editor* to bring up the thunderbird's registry editor.
- Set `mailnews.send_plaintext_flowed` to `false`
- Set `mailnews.wraplength` from 72 to `0`
- *View→Message Body As→Plain Text*
- *View→Character Encoding→Unicode (UTF-8)*

* TkRat (GUI)

Works. Use "Insert file..." or external editor.

* Gmail (Web GUI)

Does not work for sending patches.

Gmail web client converts tabs to spaces automatically.

At the same time it wraps lines every 78 chars with CRLF style line breaks although tab2space problem can be solved with external editor.

Another problem is that Gmail will base64-encode any message that has a non-ASCII character. That includes things like European names.

Other guides to the community that are of interest to most developers are:

CHAPTER

SEVEN

MINIMAL REQUIREMENTS TO COMPILE THE KERNEL

* Intro

This document is designed to provide a list of the minimum levels of software necessary to run the 4.x kernels.

This document is originally based on my “Changes” file for 2.0.x kernels and therefore owes credit to the same people as that file (Jared Mauch, Axel Boldt, Alessandro Sigala, and countless other users all over the ‘net).

* Current Minimal Requirements

Upgrade to at **least** these software revisions before thinking you’ve encountered a bug! If you’re unsure what version you’re currently running, the suggested command should tell you.

Again, keep in mind that this list assumes you are already functionally running a Linux kernel. Also, not all tools are necessary on all systems; obviously, if you don’t have any ISDN hardware, for example, you probably needn’t concern yourself with isdn4k-utils.

Program	Minimal version	Command to check the version
GNU C	3.2	gcc –version
GNU make	3.81	make –version
binutils	2.20	ld -v
util-linux	2.10o	fdformat –version
module-init-tools	0.9.10	depmod -V
e2fsprogs	1.41.4	e2fsck -V
jfsutils	1.1.3	fsck.jfs -V
reiserfsprogs	3.6.3	reiserfsck -V
xfsprogs	2.6.0	xfs_db -V
squashfs-tools	4.0	mksquashfs -version
btrfs-progs	0.18	btrfsck
pcmciautils	004	pccardctl -V
quota-tools	3.09	quota -V
PPP	2.4.0	pppd –version
isdn4k-utils	3.1pre1	isdnctrl 2>&1|grep version
nfs-utils	1.0.5	showmount –version
procps	3.2.0	ps –version
oprofile	0.9	oprofiled –version
udev	081	udevd –version
grub	0.93	grub –version || grub-install –version
mcelog	0.6	mcelog –version
iptables	1.4.2	iptables -V
openssl & libcrypto	1.0.0	openssl version
bc	1.06.95	bc –version
Sphinx[1]	1.3	sphinx-build –version

* Kernel compilation

GCC

The gcc version requirements may vary depending on the type of CPU in your computer.

Make

You will need GNU make 3.81 or later to build the kernel.

Binutils

The build system has, as of 4.13, switched to using thin archives (*ar T*) rather than incremental linking (*ld -r*) for built-in.o intermediate steps. This requires binutils 2.20 or newer.

Perl

You will need perl 5 and the following modules: `Getopt::Long`, `Getopt::Std`, `File::Basename`, and `File::Find` to build the kernel.

BC

You will need bc to build kernels 3.10 and higher

[1] Sphinx is needed only to build the Kernel documentation

OpenSSL

Module signing and external certificate handling use the OpenSSL program and crypto library to do key creation and signature generation.

You will need openssl to build kernels 3.7 and higher if module signing is enabled. You will also need openssl development packages to build kernels 4.3 and higher.

* System utilities

Architectural changes

DevFS has been obsoleted in favour of udev (http://www.kernel.org/pub/linux/utils/kernel/hotplug/)

32-bit UID support is now in place. Have fun!

Linux documentation for functions is transitioning to inline documentation via specially-formatted comments near their definitions in the source. These comments can be combined with ReST files the Documentation/ directory to make enriched documentation, which can then be converted to PostScript, HTML, LaTex, ePUB and PDF files. In order to convert from ReST format to a format of your choice, you'll need Sphinx.

Util-linux

New versions of util-linux provide `fdisk` support for larger disks, support new options to mount, recognize more supported partition types, have a fdformat which works with 2.4 kernels, and similar goodies. You'll probably want to upgrade.

Ksymoops

If the unthinkable happens and your kernel oopses, you may need the ksymoops tool to decode it, but in most cases you don't. It is generally preferred to build the kernel with `CONFIG_KALLSYMS` so that it produces readable dumps that can be used as-is (this also produces better output than ksymoops). If for some reason your kernel is not build with `CONFIG_KALLSYMS` and you have no way to rebuild and reproduce the Oops with that option, then you can still decode that Oops with ksymoops.

Module-Init-Tools

A new module loader is now in the kernel that requires `module-init-tools` to use. It is backward compatible with the 2.4.x series kernels.

Mkinitrd

These changes to the `/lib/modules` file tree layout also require that mkinitrd be upgraded.

E2fsprogs

The latest version of `e2fsprogs` fixes several bugs in fsck and debugfs. Obviously, it's a good idea to upgrade.

JFSutils

The jfsutils package contains the utilities for the file system. The following utilities are available:

- fsck.jfs - initiate replay of the transaction log, and check and repair a JFS formatted partition.
- mkfs.jfs - create a JFS formatted partition.
- other file system utilities are also available in this package.

Reiserfsprogs

The reiserfsprogs package should be used for reiserfs-3.6.x (Linux kernels 2.4.x). It is a combined package and contains working versions of mkreiserfs, resize_reiserfs, debugreiserfs and reiserfsck. These utils work on both i386 and alpha platforms.

Xfsprogs

The latest version of xfsprogs contains mkfs.xfs, xfs_db, and the xfs_repair utilities, among others, for the XFS filesystem. It is architecture independent and any version from 2.0.0 onward should work correctly with this version of the XFS kernel code (2.6.0 or later is recommended, due to some significant improvements).

PCMCIAutils

PCMCIAutils replaces pcmcia-cs. It properly sets up PCMCIA sockets at system startup and loads the appropriate modules for 16-bit PCMCIA devices if the kernel is modularized and the hotplug subsystem is used.

Quota-tools

Support for 32 bit uid's and gid's is required if you want to use the newer version 2 quota format. Quota-tools version 3.07 and newer has this support. Use the recommended version or newer from the table above.

Intel IA32 microcode

A driver has been added to allow updating of Intel IA32 microcode, accessible as a normal (misc) character device. If you are not using udev you may need to:

```
mkdir /dev/cpu
mknod /dev/cpu/microcode c 10 184
chmod 0644 /dev/cpu/microcode
```

as root before you can use this. You'll probably also want to get the user-space microcode_ctl utility to use with this.

udev

udev is a userspace application for populating /dev dynamically with only entries for devices actually present. udev replaces the basic functionality of devfs, while allowing persistent device naming for devices.

FUSE

Needs libfuse 2.4.0 or later. Absolute minimum is 2.3.0 but mount options `direct_io` and `kernel_cache` won't work.

* Networking

General changes

If you have advanced network configuration needs, you should probably consider using the network tools from ip-route2.

Packet Filter / NAT

The packet filtering and NAT code uses the same tools like the previous 2.4.x kernel series (iptables). It still includes backwards-compatibility modules for 2.2.x-style ipchains and 2.0.x-style ipfwadm.

PPP

The PPP driver has been restructured to support multilink and to enable it to operate over diverse media layers. If you use PPP, upgrade pppd to at least 2.4.0.

If you are not using udev, you must have the device file /dev/ppp which can be made by:

```
mknod /dev/ppp c 108 0
```

as root.

Isdn4k-utils

Due to changes in the length of the phone number field, isdn4k-utils needs to be recompiled or (preferably) upgraded.

NFS-utils

In ancient (2.4 and earlier) kernels, the nfs server needed to know about any client that expected to be able to access files via NFS. This information would be given to the kernel by `mountd` when the client mounted the filesystem, or by `exportfs` at system startup. exportfs would take information about active clients from `/var/lib/nfs/rmtab`.

This approach is quite fragile as it depends on rmtab being correct which is not always easy, particularly when trying to implement fail-over. Even when the system is working well, `rmtab` suffers from getting lots of old entries that never get removed.

With modern kernels we have the option of having the kernel tell mountd when it gets a request from an unknown host, and mountd can give appropriate export information to the kernel. This removes the dependency on `rmtab` and means that the kernel only needs to know about currently active clients.

To enable this new functionality, you need to:

```
mount -t nfsd nfsd /proc/fs/nfsd
```

before running exportfs or mountd. It is recommended that all NFS services be protected from the internet-at-large by a firewall where that is possible.

mcelog

On x86 kernels the mcelog utility is needed to process and log machine check events when `CONFIG_X86_MCE` is enabled. Machine check events are errors reported by the CPU. Processing them is strongly encouraged.

* Kernel documentation

Sphinx

Please see sphinx_install in `Documentation/doc-guide/sphinx.rst` for details about Sphinx requirements.

* Getting updated software

* Kernel compilation

gcc

- <ftp://ftp.gnu.org/gnu/gcc/>

Make

- <ftp://ftp.gnu.org/gnu/make/>

Binutils

- <https://www.kernel.org/pub/linux/devel/binutils/>

OpenSSL

- <https://www.openssl.org/>

* System utilities

Util-linux

- <https://www.kernel.org/pub/linux/utils/util-linux/>

Ksymoops

- <https://www.kernel.org/pub/linux/utils/kernel/ksymoops/v2.4/>

Module-Init-Tools

- <https://www.kernel.org/pub/linux/utils/kernel/module-init-tools/>

Mkinitrd

- <https://code.launchpad.net/initrd-tools/main>

E2fsprogs

- <http://prdownloads.sourceforge.net/e2fsprogs/e2fsprogs-1.29.tar.gz>

JFSutils

- <http://jfs.sourceforge.net/>

Reiserfsprogs

- <http://www.kernel.org/pub/linux/utils/fs/reiserfs/>

Xfsprogs

- <ftp://oss.sgi.com/projects/xfs/>

Pcmciautils

- <https://www.kernel.org/pub/linux/utils/kernel/pcmcia/>

Quota-tools

- <http://sourceforge.net/projects/linuxquota/>

Intel P6 microcode

- <https://downloadcenter.intel.com/>

udev

- <http://www.freedesktop.org/software/systemd/man/udev.html>

FUSE

- <http://sourceforge.net/projects/fuse>

mcelog

- <http://www.mcelog.org/>

* Networking

PPP

- <ftp://ftp.samba.org/pub/ppp/>

Isdn4k-utils

- <ftp://ftp.isdn4linux.de/pub/isdn4linux/utils/>

NFS-utils

- <http://sourceforge.net/project/showfiles.php?group_id=14>

Iptables

- <http://www.iptables.org/downloads.html>

Ip-route2

- <https://www.kernel.org/pub/linux/utils/net/iproute2/>

OProfile

- <http://oprofile.sf.net/download/>

NFS-Utils

- <http://nfs.sourceforge.net/>

* Kernel documentation

Sphinx

- <http://www.sphinx-doc.org/>

CHAPTER

EIGHT

SUBMITTING DRIVERS FOR THE LINUX KERNEL

This document is intended to explain how to submit device drivers to the various kernel trees. Note that if you are interested in video card drivers you should probably talk to XFree86 (http://www.xfree86.org/) and/or X.Org (http://x.org/) instead.

Note:

This document is old and has seen little maintenance in recent years; it should probably be updated or, perhaps better, just deleted. Most of what is here can be found in the other development documents anyway.
Oh, and we don't really recommend submitting changes to XFree86 :)

Also read the Documentation/process/submitting-patches.rst document.

* Allocating Device Numbers

Major and minor numbers for block and character devices are allocated by the Linux assigned name and number authority (currently this is Torben Mathiasen). The site is http://www.lanana.org/. This also deals with allocating numbers for devices that are not going to be submitted to the mainstream kernel. See Documentation/admin-guide/devices.rst for more information on this.

If you don't use assigned numbers then when your device is submitted it will be given an assigned number even if that is different from values you may have shipped to customers before.

* Who To Submit Drivers To

Linux 2.0: No new drivers are accepted for this kernel tree.

Linux 2.2: No new drivers are accepted for this kernel tree.

Linux 2.4: If the code area has a general maintainer then please submit it to the maintainer listed in MAINTAINERS in the kernel file. If the maintainer does not respond or you cannot find the appropriate maintainer then please contact Willy Tarreau <w@1wt.eu>.

Linux 2.6 and upper: The same rules apply as 2.4 except that you should follow linux-kernel to track changes in API's. The final contact point for Linux 2.6+ submissions is Andrew Morton.

* What Criteria Determine Acceptance

Licensing: The code must be released to us under the GNU General Public License. We don't insist on any kind of exclusive GPL licensing, and if you wish the driver to be useful to other communities such as BSD you may well wish to release under multiple licenses. See accepted licenses at include/linux/module.h

Copyright: The copyright owner must agree to use of GPL. It's best if the submitter and copyright owner are the same person/entity. If not, the name of the person/entity authorizing use of GPL should be listed in case it's necessary to verify the will of the copyright owner.

Interfaces: If your driver uses existing interfaces and behaves like other drivers in the same class it will be much more likely to be accepted than if it invents gratuitous new ones. If you need to implement a common API over Linux and NT drivers do it in userspace.

Code: Please use the Linux style of code formatting as documented in *Documentation/process/coding-style.rst* . If you have sections of code that need to be in other formats, for example because they are shared with a windows driver kit and you want to maintain them just once separate them out nicely and note this fact.

Portability: Pointers are not always 32bits, not all computers are little endian, people do not all have floating point and you shouldn't use inline x86 assembler in your driver without careful thought. Pure x86 drivers generally are not popular. If you only have x86 hardware it is hard to test portability but it is easy to make sure the code can easily be made portable.

Clarity: It helps if anyone can see how to fix the driver. It helps you because you get patches not bug reports. If you submit a driver that intentionally obfuscates how the hardware works it will go in the bitbucket.

PM support: Since Linux is used on many portable and desktop systems, your driver is likely to be used on such a system and therefore it should support basic power management by implementing, if necessary, the .suspend and .resume methods used during the system-wide suspend and resume transitions. You should verify that your driver correctly handles the suspend and resume, but if you are unable to ensure that, please at least define the .suspend method returning the -ENOSYS ("Function not implemented") error. You should also try to make sure that your driver uses as little power as possible when it's not doing anything. For the driver testing instructions see Documentation/power/drivers-testing.txt and for a relatively complete overview of the power management issues related to drivers see Documentation/power/admin-guide/devices.rst .

Control: In general if there is active maintenance of a driver by the author then patches will be redirected to them unless they are totally obvious and without need of checking. If you want to be the contact and update point for the driver it is a good idea to state this in the comments, and include an entry in MAINTAINERS for your driver.

* What Criteria Do Not Determine Acceptance

Vendor: Being the hardware vendor and maintaining the driver is often a good thing. If there is a stable working driver from other people already in the tree don't expect 'we are the vendor' to get your driver chosen. Ideally work with the existing driver author to build a single perfect driver.

Author: It doesn't matter if a large Linux company wrote the driver, or you did. Nobody has any special access to the kernel tree. Anyone who tells you otherwise isn't telling the whole story.

* Resources

Linux kernel master tree: ftp.*country_code*.kernel.org:/pub/linux/kernel/...

where *country_code* == your country code, such as **us**, **uk**, **fr**, etc.

http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git

Linux kernel mailing list: linux-kernel@vger.kernel.org [mail majordomo@vger.kernel.org to subscribe]

Linux Device Drivers, Third Edition (covers 2.6.10): http://lwn.net/Kernel/LDD3/ (free version)

LWN.net: Weekly summary of kernel development activity - http://lwn.net/

2.6 API changes:

http://lwn.net/Articles/2.6-kernel-api/

Porting drivers from prior kernels to 2.6:

http://lwn.net/Articles/driver-porting/

KernelNewbies: Documentation and assistance for new kernel programmers

http://kernelnewbies.org/

Linux USB project: http://www.linux-usb.org/

How to NOT write kernel driver by Arjan van de Ven: http://www.fenrus.org/how-to-not-write-a-device-driver-paper.pdf

Kernel Janitor: http://kernelnewbies.org/KernelJanitors

GIT, Fast Version Control System: http://git-scm.com/

CHAPTER

NINE

THE LINUX KERNEL DRIVER INTERFACE

(all of your questions answered and then some)

Greg Kroah-Hartman <greg@kroah.com>

This is being written to try to explain why Linux **does not have a binary kernel interface, nor does it have a stable kernel interface**.

Note:

*Please realize that this article describes the **in kernel** interfaces, not the kernel to userspace interfaces.*
*The kernel to userspace interface is the one that application programs use, the syscall interface. That interface is **very** stable over time, and will not break. I have old programs that were built on a pre 0.9something kernel that still work just fine on the latest 2.6 kernel release. That interface is the one that users and application programmers can count on being stable.*

* Executive Summary

You think you want a stable kernel interface, but you really do not, and you don't even know it. What you want is a stable running driver, and you get that only if your driver is in the main kernel tree. You also get lots of other good benefits if your driver is in the main kernel tree, all of which has made Linux into such a strong, stable, and mature operating system which is the reason you are using it in the first place.

* Intro

It's only the odd person who wants to write a kernel driver that needs to worry about the in-kernel interfaces changing. For the majority of the world, they neither see this interface, nor do they care about it at all.

First off, I'm not going to address **any** legal issues about closed source, hidden source, binary blobs, source wrappers, or any other term that describes kernel drivers that do not have their source code released under the GPL. Please consult a lawyer if you have any legal questions, I'm a programmer and hence, I'm just going to be describing the technical issues here (not to make light of the legal issues, they are real, and you do need to be aware of them at all times.)

So, there are two main topics here, binary kernel interfaces and stable kernel source interfaces. They both depend on each other, but we will discuss the binary stuff first to get it out of the way.

* Binary Kernel Interface

Assuming that we had a stable kernel source interface for the kernel, a binary interface would naturally happen too, right? Wrong. Please consider the following facts about the Linux kernel:

- Depending on the version of the C compiler you use, different kernel data structures will contain different alignment of structures, and possibly include different functions in different ways (putting functions inline or not.) The individual function organization isn't that important, but the different data structure padding is very important.
- Depending on what kernel build options you select, a wide range of different things can be assumed by the kernel:
  - different structures can contain different fields
  - Some functions may not be implemented at all, (i.e. some locks compile away to nothing for non-SMP builds.)
  - Memory within the kernel can be aligned in different ways, depending on the build options.
- Linux runs on a wide range of different processor architectures. There is no way that binary drivers from one architecture will run on another architecture properly.

Now a number of these issues can be addressed by simply compiling your module for the exact specific kernel configuration, using the same exact C compiler that the kernel was built with. This is sufficient if you want to provide a module for a specific release version of a specific Linux distribution. But multiply that single build by the number of different Linux distributions and the number of different supported releases of the Linux distribution and you quickly have a nightmare of different build options on different releases. Also realize that each Linux distribution release contains a number of different kernels, all tuned to different hardware types (different processor types and different options), so for even a single release you will need to create multiple versions of your module.

Trust me, you will go insane over time if you try to support this kind of release, I learned this the hard way a long time ago...

* Stable Kernel Source Interfaces

This is a much more "volatile" topic if you talk to people who try to keep a Linux kernel driver that is not in the main kernel tree up to date over time.

Linux kernel development is continuous and at a rapid pace, never stopping to slow down. As such, the kernel developers find bugs in current interfaces, or figure out a better way to do things. If they do that, they then fix the current interfaces to work better. When they do so, function names may change, structures may grow or shrink, and function parameters may be reworked. If this happens, all of the instances of where this interface is used within the kernel are fixed up at the same time, ensuring that everything continues to work properly.

As a specific examples of this, the in-kernel USB interfaces have undergone at least three different reworks over the lifetime of this subsystem. These reworks were done to address a number of different issues:

- A change from a synchronous model of data streams to an asynchronous one. This reduced the complexity of a number of drivers and increased the throughput of all USB drivers such that we are now running almost all USB devices at their maximum speed possible.
- A change was made in the way data packets were allocated from the USB core by USB drivers so that all drivers now needed to provide more information to the USB core to fix a number of documented deadlocks.

This is in stark contrast to a number of closed source operating systems which have had to maintain their older USB interfaces over time. This provides the ability for new developers to accidentally use the old interfaces and do things in improper ways, causing the stability of the operating system to suffer.

In both of these instances, all developers agreed that these were important changes that needed to be made, and they were made, with relatively little pain. If Linux had to ensure that it will preserve a stable source interface, a new interface would have been created, and the older, broken one would have had to be maintained over time, leading to extra work for the USB developers. Since all Linux USB developers do their work on their own time, asking programmers to do extra work for no gain, for free, is not a possibility.

Security issues are also very important for Linux. When a security issue is found, it is fixed in a very short amount of time. A number of times this has caused internal kernel interfaces to be reworked to prevent the security problem from occurring. When this happens, all drivers that use the interfaces were also fixed at the same time, ensuring that the security problem was fixed and could not come back at some future time accidentally. If the internal interfaces were not allowed to change, fixing this kind of security problem and insuring that it could not happen again would not be possible.

Kernel interfaces are cleaned up over time. If there is no one using a current interface, it is deleted. This ensures that the kernel remains as small as possible, and that all potential interfaces are tested as well as they can be (unused interfaces are pretty much impossible to test for validity.)

* What to do

So, if you have a Linux kernel driver that is not in the main kernel tree, what are you, a developer, supposed to do? Releasing a binary driver for every different kernel version for every distribution is a nightmare, and trying to keep up with an ever changing kernel interface is also a rough job.

Simple, get your kernel driver into the main kernel tree (remember we are talking about GPL released drivers here, if your code doesn't fall under this category, good luck, you are on your own here, you leech <insert link to leech comment from Andrew and Linus here>.) If your driver is in the tree, and a kernel interface changes, it will be fixed up by the person who did the kernel change in the first place. This ensures that your driver is always buildable, and works over time, with very little effort on your part.

The very good side effects of having your driver in the main kernel tree are:

- The quality of the driver will rise as the maintenance costs (to the original developer) will decrease.
- Other developers will add features to your driver.
- Other people will find and fix bugs in your driver.
- Other people will find tuning opportunities in your driver.
- Other people will update the driver for you when external interface changes require it.
- The driver automatically gets shipped in all Linux distributions without having to ask the distros to add it.

As Linux supports a larger number of different devices "out of the box" than any other operating system, and it supports these devices on more different processor architectures than any other operating system, this proven type of development model must be doing something right :)

Thanks to Randy Dunlap, Andrew Morton, David Brownell, Hanna Linder, Robert Love, and Nishanth Aravamudan for their review and comments on early drafts of this paper.

CHAPTER

TEN

LINUX KERNEL MANAGEMENT STYLE

This is a short document describing the preferred (or made up, depending on who you ask) management style for the linux kernel. It's meant to mirror the process/coding-style.rst document to some degree, and mainly written to avoid answering [1] the same (or similar) questions over and over again.

Management style is very personal and much harder to quantify than simple coding style rules, so this document may or may not have anything to do with reality. It started as a lark, but that doesn't mean that it might not actually be true. You'll have to decide for yourself.

Btw, when talking about "kernel manager", it's all about the technical lead persons, not the people who do traditional management inside companies. If you sign purchase orders or you have any clue about the budget of your group, you're almost certainly not a kernel manager. These suggestions may or may not apply to you.

First off, I'd suggest buying "Seven Habits of Highly Effective People", and NOT read it. Burn it, it's a great symbolic gesture.

Anyway, here goes:

* 1) Decisions

Everybody thinks managers make decisions, and that decision-making is important. The bigger and more painful the decision, the bigger the manager must be to make it. That's very deep and obvious, but it's not actually true.

The name of the game is to **avoid** having to make a decision. In particular, if somebody tells you "choose (a) or (b), we really need you to decide on this", you're in trouble as a manager. The people you manage had better know the details better than you, so if they come to you for a technical decision, you're screwed. You're clearly not competent to make that decision for them.

(Corollary:if the people you manage don't know the details better than you, you're also screwed, although for a totally different reason. Namely that you are in the wrong job, and that **they** should be managing your brilliance instead).

So the name of the game is to **avoid** decisions, at least the big and painful ones. Making small and non-consequential decisions is fine, and makes you look like you know what you're doing, so what a kernel manager needs to do is to turn the big and painful ones into small things where nobody really cares.

It helps to realize that the key difference between a big decision and a small one is whether you can fix your decision afterwards. Any decision can be made small by just always making sure that if you were wrong (and you **will** be wrong), you can always undo the damage later by backtracking. Suddenly, you get to be doubly managerial for making **two** inconsequential decisions - the wrong one **and** the right one.

And people will even see that as true leadership (*cough* bullshit *cough*).

[1] This document does so not so much by answering the question, but by making it painfully obvious to the questioner that we don't have a clue to what the answer is.

Thus the key to avoiding big decisions becomes to just avoiding to do things that can't be undone. Don't get ushered into a corner from which you cannot escape. A cornered rat may be dangerous - a cornered manager is just pitiful.

It turns out that since nobody would be stupid enough to ever really let a kernel manager have huge fiscal responsibility **anyway**, it's usually fairly easy to backtrack. Since you're not going to be able to waste huge amounts of money that you might not be able to repay, the only thing you can backtrack on is a technical decision, and there back-tracking is very easy: just tell everybody that you were an incompetent nincompoop, say you're sorry, and undo all the worthless work you had people work on for the last year. Suddenly the decision you made a year ago wasn't a big decision after all, since it could be easily undone.

It turns out that some people have trouble with this approach, for two reasons:

- admitting you were an idiot is harder than it looks. We all like to maintain appearances, and coming out in public to say that you were wrong is sometimes very hard indeed.
- having somebody tell you that what you worked on for the last year wasn't worthwhile after all can be hard on the poor lowly engineers too, and while the actual **work** was easy enough to undo by just deleting it, you may have irrevocably lost the trust of that engineer. And remember: "irrevocable" was what we tried to avoid in the first place, and your decision ended up being a big one after all.

Happily, both of these reasons can be mitigated effectively by just admitting up-front that you don't have a friggin' clue, and telling people ahead of the fact that your decision is purely preliminary, and might be the wrong thing. You should always reserve the right to change your mind, and make people very **aware** of that. And it's much easier to admit that you are stupid when you haven't **yet** done the really stupid thing.

Then, when it really does turn out to be stupid, people just roll their eyes and say "Oops, he did it again".

This preemptive admission of incompetence might also make the people who actually do the work also think twice about whether it's worth doing or not. After all, if **they** aren't certain whether it's a good idea, you sure as hell shouldn't encourage them by promising them that what they work on will be included. Make them at least think twice before they embark on a big endeavor.

Remember: they'd better know more about the details than you do, and they usually already think they have the answer to everything. The best thing you can do as a manager is not to instill confidence, but rather a healthy dose of critical thinking on what they do.

Btw, another way to avoid a decision is to plaintively just whine "can't we just do both?" and look pitiful. Trust me, it works. If it's not clear which approach is better, they'll eventually figure it out. The answer may end up being that both teams get so frustrated by the situation that they just give up.

That may sound like a failure, but it's usually a sign that there was something wrong with both projects, and the reason the people involved couldn't decide was that they were both wrong. You end up coming up smelling like roses, and you avoided yet another decision that you could have screwed up on.

* 2) People

Most people are idiots, and being a manager means you'll have to deal with it, and perhaps more importantly, that **they** have to deal with **you**.

It turns out that while it's easy to undo technical mistakes, it's not as easy to undo personality disorders. You just have to live with theirs - and yours.

However, in order to prepare yourself as a kernel manager, it's best to remember not to burn any bridges, bomb any innocent villagers, or alienate too many kernel developers. It turns out that alienating people is fairly easy, and un-alienating them is hard. Thus "alienating" immediately falls under the heading of "not reversible", and becomes a no-no according to *1) Decisions* .

There's just a few simple rules here:

1. don't call people d*ckheads (at least not in public)
2. learn how to apologize when you forgot rule (1)

The problem with #1 is that it's very easy to do, since you can say "you're a d*ckhead" in millions of different ways [2], sometimes without even realizing it, and almost always with a white-hot conviction that you are right.

And the more convinced you are that you are right (and let's face it, you can call just about **anybody** a d*ckhead, and you often **will** be right), the harder it ends up being to apologize afterwards.

To solve this problem, you really only have two options:

- get really good at apologies
- spread the "love" out so evenly that nobody really ends up feeling like they get unfairly targeted. Make it inventive enough, and they might even be amused.

The option of being unfailingly polite really doesn't exist. Nobody will trust somebody who is so clearly hiding his true character.

* 3) People II - the Good Kind

While it turns out that most people are idiots, the corollary to that is sadly that you are one too, and that while we can all bask in the secure knowledge that we're better than the average person (let's face it, nobody ever believes that they're average or below-average), we should also admit that we're not the sharpest knife around, and there will be other people that are less of an idiot than you are.

Some people react badly to smart people. Others take advantage of them.

Make sure that you, as a kernel maintainer, are in the second group. Suck up to them, because they are the people who will make your job easier. In particular, they'll be able to make your decisions for you, which is what the game is all about.

So when you find somebody smarter than you are, just coast along. Your management responsibilities largely become ones of saying "Sounds like a good idea - go wild", or "That sounds good, but what about xxx?". The second version in particular is a great way to either learn something new about "xxx" or seem **extra** managerial by pointing out something the smarter person hadn't thought about. In either case, you win.

One thing to look out for is to realize that greatness in one area does not necessarily translate to other areas. So you might prod people in specific directions, but let's face it, they might be good at what they do, and suck at everything else. The good news is that people tend to naturally gravitate back to what they are good at, so it's not like you are doing something irreversible when you **do** prod them in some direction, just don't push too hard.

* 4) Placing blame

Things will go wrong, and people want somebody to blame. Tag, you're it.

It's not actually that hard to accept the blame, especially if people kind of realize that it wasn't **all** your fault. Which brings us to the best way of taking the blame: do it for another guy. You'll feel good for taking the fall, he'll feel good about not getting blamed, and the guy who lost his whole 36GB porn-collection because of your incompetence will grudgingly admit that you at least didn't try to weasel out of it.

Then make the developer who really screwed up (if you can find him) know **in_private** that he screwed up. Not just so he can avoid it in the future, but so that he knows he owes you one. And, perhaps even more importantly, he's also likely the person who can fix it. Because, let's face it, it sure ain't you.

Taking the blame is also why you get to be manager in the first place. It's part of what makes people trust you, and allow you the potential glory, because you're the one who gets to say "I screwed up". And if you've followed the previous rules, you'll be pretty good at saying that by now.

[2] Paul Simon sang "Fifty Ways to Leave Your Lover", because quite frankly, "A Million Ways to Tell a Developer He Is a D*ckhead" doesn't scan nearly as well. But I'm sure he thought about it.

* 5) Things to avoid

There's one thing people hate even more than being called "d*ckhead", and that is being called a "d*ckhead" in a sanctimonious voice. The first you can apologize for, the second one you won't really get the chance. They likely will no longer be listening even if you otherwise do a good job.

We all think we're better than anybody else, which means that when somebody else puts on airs, it **really** rubs us the wrong way. You may be morally and intellectually superior to everybody around you, but don't try to make it too obvious unless you really **intend** to irritate somebody [3].

Similarly, don't be too polite or subtle about things. Politeness easily ends up going overboard and hiding the problem, and as they say, "On the internet, nobody can hear you being subtle". Use a big blunt object to hammer the point in, because you can't really depend on people getting your point otherwise.

Some humor can help pad both the bluntness and the moralizing. Going overboard to the point of being ridiculous can drive a point home without making it painful to the recipient, who just thinks you're being silly. It can thus help get through the personal mental block we all have about criticism.

* 6) Why me?

Since your main responsibility seems to be to take the blame for other peoples mistakes, and make it painfully obvious to everybody else that you're incompetent, the obvious question becomes one of why do it in the first place?

First off, while you may or may not get screaming teenage girls (or boys, let's not be judgmental or sexist here) knocking on your dressing room door, you **will** get an immense feeling of personal accomplishment for being "in charge". Never mind the fact that you're really leading by trying to keep up with everybody else and running after them as fast as you can. Everybody will still think you're the person in charge.

It's a great job if you can hack it.

[3] Hint: internet newsgroups that are not directly related to your work are great ways to take out your frustrations at other people. Write insulting posts with a sneer just to get into a good flame every once in a while, and you'll feel cleansed. Just don't crap too close to home.

CHAPTER

ELEVEN

EVERYTHING YOU EVER WANTED TO KNOW ABOUT LINUX -STABLE RELEASES

Rules on what kind of patches are accepted, and which ones are not, into the "-stable" tree:

- It must be obviously correct and tested.
- It cannot be bigger than 100 lines, with context.
- It must fix only one thing.
- It must fix a real bug that bothers people (not a, "This could be a problem…" type thing).
- It must fix a problem that causes a build error (but not for things marked CONFIG_BROKEN), an oops, a hang, data corruption, a real security issue, or some "oh, that's not good" issue. In short, something critical.
- Serious issues as reported by a user of a distribution kernel may also be considered if they fix a notable performance or interactivity issue. As these fixes are not as obvious and have a higher risk of a subtle regression they should only be submitted by a distribution kernel maintainer and include an addendum linking to a bugzilla entry if it exists and additional information on the user-visible impact.
- New device IDs and quirks are also accepted.
- No "theoretical race condition" issues, unless an explanation of how the race can be exploited is also provided.
- It cannot contain any "trivial" fixes in it (spelling changes, whitespace cleanups, etc).
- It must follow the *Documentation/process/submitting-patches.rst* rules.
- It or an equivalent fix must already exist in Linus' tree (upstream).

* Procedure for submitting patches to the -stable tree

- If the patch covers files in net/ or drivers/net please follow netdev stable submission guidelines as described in Documentation/networking/netdev-FAQ.txt
- Security patches should not be handled (solely) by the -stable review process but should follow the procedures in Documentation/admin-guide/security-bugs.rst .

* For all other submissions, choose one of the following procedures

* Option 1

To have the patch automatically included in the stable tree, add the tag

```
Cc: stable@vger.kernel.org
```

in the sign-off area. Once the patch is merged it will be applied to the stable tree without anything else needing to be done by the author or subsystem maintainer.

* Option 2

After the patch has been merged to Linus' tree, send an email to stable@vger.kernel.org containing the subject of the patch, the commit ID, why you think it should be applied, and what kernel version you wish it to be applied to.

* Option 3

Send the patch, after verifying that it follows the above rules, to stable@vger.kernel.org. You must note the upstream commit ID in the changelog of your submission, as well as the kernel version you wish it to be applied to.

Option 1 is **strongly** preferred, is the easiest and most common. *Option 2* and *Option 3* are more useful if the patch isn't deemed worthy at the time it is applied to a public git tree (for instance, because it deserves more regression testing first). *Option 3* is especially useful if the patch needs some special handling to apply to an older kernel (e.g., if API's have changed in the meantime).

Note that for *Option 3* , if the patch deviates from the original upstream patch (for example because it had to be backported) this must be very clearly documented and justified in the patch description.

The upstream commit ID must be specified with a separate line above the commit text, like this:

```
commit <sha1> upstream.
```

Additionally, some patches submitted via Option 1 may have additional patch prerequisites which can be cherry-picked. This can be specified in the following format in the sign-off area:

```
Cc: <stable@vger.kernel.org> # 3.3.x: a1f84a3: sched: Check for idle
Cc: <stable@vger.kernel.org> # 3.3.x: 1b9508f: sched: Rate-limit newidle
Cc: <stable@vger.kernel.org> # 3.3.x: fd21073: sched: Fix affinity logic
Cc: <stable@vger.kernel.org> # 3.3.x
Signed-off-by: Ingo Molnar <mingo@elte.hu>
```

The tag sequence has the meaning of:

```
git cherry-pick a1f84a3
git cherry-pick 1b9508f
git cherry-pick fd21073
git cherry-pick <this commit>
```

Also, some patches may have kernel version prerequisites. This can be specified in the following format in the sign-off area:

```
Cc: <stable@vger.kernel.org> # 3.3.x
```

The tag has the meaning of:

```
git cherry-pick <this commit>
```

For each "-stable" tree starting with the specified version.

Following the submission:

- The sender will receive an ACK when the patch has been accepted into the queue, or a NAK if the patch is rejected. This response might take a few days, according to the developer's schedules.

- If accepted, the patch will be added to the -stable queue, for review by other developers and by the relevant subsystem maintainer.

* Review cycle

- When the -stable maintainers decide for a review cycle, the patches will be sent to the review committee, and the maintainer of the affected area of the patch (unless the submitter is the maintainer of the area) and CC: to the linux-kernel mailing list.
- The review committee has 48 hours in which to ACK or NAK the patch.
- If the patch is rejected by a member of the committee, or linux-kernel members object to the patch, bringing up issues that the maintainers and members did not realize, the patch will be dropped from the queue.
- At the end of the review cycle, the ACKed patches will be added to the latest -stable release, and a new -stable release will happen.
- Security patches will be accepted into the -stable tree directly from the security kernel team, and not go through the normal review cycle. Contact the kernel security team for more details on this procedure.

* Trees

- The queues of patches, for both completed versions and in progress versions can be found at:

  https://git.kernel.org/pub/scm/linux/kernel/git/stable/stable-queue.git
- The finalized and tagged releases of all stable kernels can be found in separate branches per version at:

  https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git

* Review committee

- This is made up of a number of kernel developers who have volunteered for this task, and a few that haven’t.

CHAPTER

TWELVE

LINUX KERNEL PATCH SUBMISSION CHECKLIST

Here are some basic things that developers should do if they want to see their kernel patch submissions accepted more quickly.

These are all above and beyond the documentation that is provided in *Documentation/process/submitting-patches.rst* and elsewhere regarding submitting Linux kernel patches.

1. If you use a facility then #include the file that defines/declares that facility. Don't depend on other header files pulling in ones that you use.
2. Builds cleanly:
1. with applicable or modified `CONFIG` options `=y`, `=m`, and `=n`. No `gcc` warnings/errors, no linker warnings/errors.
2. Passes `allnoconfig`, `allmodconfig`
3. Builds successfully when using `O=builddir`
3. Builds on multiple CPU architectures by using local cross-compile tools or some other build farm.
4. ppc64 is a good architecture for cross-compilation checking because it tends to use `unsigned long` for 64-bit quantities.
5. Check your patch for general style as detailed in *Documentation/process/coding-style.rst* . Check for trivial violations with the patch style checker prior to submission (`scripts/checkpatch.pl`). You should be able to justify all violations that remain in your patch.
6. Any new or modified `CONFIG` options don't muck up the config menu.
7. All new `Kconfig` options have help text.
8. Has been carefully reviewed with respect to relevant `Kconfig` combinations. This is very hard to get right with testing – brainpower pays off here.
9. Check cleanly with sparse.
10. Use `make checkstack` and `make namespacecheck` and fix any problems that they find.

    Note:

    `checkstack` does not point out problems explicitly, but any one function that uses more than 512 bytes on the stack is a candidate for change.

11. Include kernel-doc to document global kernel APIs. (Not required for static functions, but OK there also.) Use `make htmldocs` or `make pdfdocs` to check the kernel-doc and fix any issues.
12. Has been tested with `CONFIG_PREEMPT`, `CONFIG_DEBUG_PREEMPT`, `CONFIG_DEBUG_SLAB`, `CONFIG_DEBUG_PAGEALLOC`, `CONFIG_DEBUG_MUTEXES`, `CONFIG_DEBUG_SPINLOCK`, `CONFIG_DEBUG_ATOMIC_SLEEP`, `CONFIG_PROVE_RCU` and `CONFIG_DEBUG_OBJECTS_RCU_HEAD` all simultaneously enabled.

13. Has been build- and runtime tested with and without `CONFIG_SMP` and `CONFIG_PREEMPT`.
14. If the patch affects IO/Disk, etc: has been tested with and without `CONFIG_LBDAF`.
15. All codepaths have been exercised with all lockdep features enabled.
16. All new `/proc` entries are documented under `Documentation/`
17. All new kernel boot parameters are documented in `Documentation/admin-guide/kernel-parameters.rst`.
18. All new module parameters are documented with `MODULE_PARM_DESC()`
19. All new userspace interfaces are documented in `Documentation/ABI/`. See `Documentation/ABI/README` for more information. Patches that change userspace interfaces should be CCed to linux-api@vger.kernel.org.
20. Check that it all passes `make headers_check`.
21. Has been checked with injection of at least slab and page-allocation failures. See `Documentation/fault-injection/`.

    If the new code is substantial, addition of subsystem-specific fault injection might be appropriate.
22. Newly-added code has been compiled with `gcc -W` (use `make EXTRA_CFLAGS=-W`). This will generate lots of noise, but is good for finding bugs like “warning: comparison between signed and unsigned”.
23. Tested after it has been merged into the -mm patchset to make sure that it still works with all of the other queued patches and various changes in the VM, VFS, and other subsystems.
24. All memory barriers {e.g., `barrier()`, `rmb()`, `wmb()`} need a comment in the source code that explains the logic of what they are doing and why.
25. If any ioctl’s are added by the patch, then also update `Documentation/ioctl/ioctl-number.txt`.
26. If your modified source code depends on or uses any of the kernel APIs or features that are related to the following `Kconfig` symbols, then test multiple builds with the related `Kconfig` symbols disabled and/or `=m` (if that option is available) [not all of these at the same time, just various/random combinations of them]:

    `CONFIG_SMP`, `CONFIG_SYSFS`, `CONFIG_PROC_FS`, `CONFIG_INPUT`, `CONFIG_PCI`, `CONFIG_BLOCK`, `CONFIG_PM`, `CONFIG_MAGIC_SYSRQ`, `CONFIG_NET`, `CONFIG_INET=n` (but latter with `CONFIG_NET=y`).

CHAPTER

THIRTEEN

INDEX OF DOCUMENTATION FOR PEOPLE INTERESTED IN WRITING AND/OR UNDERSTANDING THE LINUX KERNEL

Juan-Mariano de Goyeneche <jmseyas@dit.upm.es>

The need for a document like this one became apparent in the linux-kernel mailing list as the same questions, asking for pointers to information, appeared again and again.

Fortunately, as more and more people get to GNU/Linux, more and more get interested in the Kernel. But reading the sources is not always enough. It is easy to understand the code, but miss the concepts, the philosophy and design decisions behind this code.

Unfortunately, not many documents are available for beginners to start. And, even if they exist, there was no "well-known" place which kept track of them. These lines try to cover this lack. All documents available on line known by the author are listed, while some reference books are also mentioned.

PLEASE, if you know any paper not listed here or write a new document, send me an e-mail, and I'll include a reference to it here. Any corrections, ideas or comments are also welcomed.

The papers that follow are listed in no particular order. All are cataloged with the following fields: the document's "Title", the "Author"/s, the "URL" where they can be found, some "Keywords" helpful when searching for specific topics, and a brief "Description" of the Document.

Enjoy!

Note:

The documents on each section of this document are ordered by its published date, from the newest to the oldest.

* Docs at the Linux Kernel tree

The Sphinx books should be built with `make {htmldocs | pdfdocs | epubdocs}`.

- Name: **linux/Documentation**

  Author Many.

  Location Documentation/

  Keywords text files, Sphinx.

  Description Documentation that comes with the kernel sources, inside the Documentation directory. Some pages from this document (including this document itself) have been moved there, and might be more up to date than the web version.

* On-line docs

- Title: **Linux Kernel Mailing List Glossary**

  Author various

  URL http://kernelnewbies.org/glossary/

  Date rolling version

  Keywords glossary, terms, linux-kernel.

  Description From the introduction: "This glossary is intended as a brief description of some of the acronyms and terms you may hear during discussion of the Linux kernel".

- Title: **Tracing the Way of Data in a TCP Connection through the Linux Kernel**

  Author Richard Sailer

  URL https://archive.org/details/linux_kernel_data_flow_short_paper

  Date 2016

  Keywords Linux Kernel Networking, TCP, tracing, ftrace

  Description A seminar paper explaining ftrace and how to use it for understanding linux kernel internals, illustrated at tracing the way of a TCP packet through the kernel.

  Abstract *This short paper outlines the usage of ftrace a tracing framework as a tool to understand a running Linux system. Having obtained a trace-log a kernel hacker can read and understand source code more determined and with context. In a detailed example this approach is demonstrated in tracing and the way of data in a TCP Connection through the kernel. Finally this trace-log is used as base for more a exact conceptual exploration and description of the Linux TCP/IP implementation.*

- Title: **On submitting kernel Patches**

  Author Andi Kleen

  URL http://halobates.de/on-submitting-kernel-patches.pdf

  Date 2008

  Keywords patches, review process, types of submissions, basic rules, case studies

  Description This paper gives several experience values on what types of patches there are and how likley they get merged.

  Abstract [...]. This paper examines some common problems for submitting larger changes and some strategies to avoid problems.

- Title: **Overview of the Virtual File System**

  Author Richard Gooch.

  URL http://www.mjmwired.net/kernel/Documentation/filesystems/vfs.txt

  Date 2007

  Keywords VFS, File System, mounting filesystems, opening files, dentries, dcache.

  Description Brief introduction to the Linux Virtual File System. What is it, how it works, operations taken when opening a file or mounting a file system and description of important data structures explaining the purpose of each of their entries.

- Title: **Linux Device Drivers, Third Edition**

  Author Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

  URL http://lwn.net/Kernel/LDD3/

  Date 2005

Description A 600-page book covering the (2.6.10) driver programming API and kernel hacking in general. Available under the Creative Commons Attribution-ShareAlike 2.0 license.

note You can also *purchase a copy from O'Reilly or elsewhere* .

- Title: **Writing an ALSA Driver**

  Author Takashi Iwai <tiwai@suse.de>

  URL http://www.alsa-project.org/~iwai/writing-an-alsa-driver/index.html

  Date 2005

  Keywords ALSA, sound, soundcard, driver, lowlevel, hardware.

  Description Advanced Linux Sound Architecture for developers, both at kernel and user-level sides. ALSA is the Linux kernel sound architecture in the 2.6 kernel version.

- Title: **Linux PCMCIA Programmer's Guide**

  Author David Hinds.

  URL http://pcmcia-cs.sourceforge.net/ftp/doc/PCMCIA-PROG.html

  Date 2003

  Keywords PCMCIA.

  Description "This document describes how to write kernel device drivers for the Linux PCMCIA Card Services interface. It also describes how to write user-mode utilities for communicating with Card Services.

- Title: **Linux Kernel Module Programming Guide**

  Author Ori Pomerantz.

  URL http://tldp.org/LDP/lkmpg/2.6/html/index.html

  Date 2001

  Keywords modules, GPL book, /proc, ioctls, system calls, interrupt handlers .

  Description Very nice 92 pages GPL book on the topic of modules programming. Lots of examples.

- Title: **Global spinlock list and usage**

  Author Rick Lindsley.

  URL http://lse.sourceforge.net/lockhier/global-spin-lock

  Date 2001

  Keywords spinlock.

  Description This is an attempt to document both the existence and usage of the spinlocks in the Linux 2.4.5 kernel. Comprehensive list of spinlocks showing when they are used, which functions access them, how each lock is acquired, under what conditions it is held, whether interrupts can occur or not while it is held...

- Title: **A Linux vm README**

  Author Kanoj Sarcar.

  URL http://kos.enix.org/pub/linux-vmm.html

  Date 2001

  Keywords virtual memory, mm, pgd, vma, page, page flags, page cache, swap cache, kswapd.

  Description Telegraphic, short descriptions and definitions relating the Linux virtual memory implementation.

- Title: **Video4linux Drivers, Part 1: Video-Capture Device**

  Author Alan Cox.

  URL http://www.linux-mag.com/id/406

  Date 2000

  Keywords video4linux, driver, video capture, capture devices, camera driver.

  Description The title says it all.

- Title: **Video4linux Drivers, Part 2: Video-capture Devices**

  Author Alan Cox.

  URL http://www.linux-mag.com/id/429

  Date 2000

  Keywords video4linux, driver, video capture, capture devices, camera driver, control, query capabilities, capability, facility.

  Description The title says it all.

- Title: **Linux IP Networking. A Guide to the Implementation and Modification of the Linux Protocol Stack.**

  Author Glenn Herrin.

  URL http://www.cs.unh.edu/cnrg/gherrin

  Date 2000

  Keywords network, networking, protocol, IP, UDP, TCP, connection, socket, receiving, transmitting, forwarding, routing, packets, modules, /proc, sk_buff, FIB, tags.

  Description Excellent paper devoted to the Linux IP Networking, explaining anything from the kernel's to the user space configuration tools' code. Very good to get a general overview of the kernel networking implementation and understand all steps packets follow from the time they are received at the network device till they are delivered to applications. The studied kernel code is from 2.2.14 version. Provides code for a working packet dropper example.

- Title: **How To Make Sure Your Driver Will Work On The Power Macintosh**

  Author Paul Mackerras.

  URL http://www.linux-mag.com/id/261

  Date 1999

  Keywords Mac, Power Macintosh, porting, drivers, compatibility.

  Description The title says it all.

- Title: **An Introduction to SCSI Drivers**

  Author Alan Cox.

  URL http://www.linux-mag.com/id/284

  Date 1999

  Keywords SCSI, device, driver.

  Description The title says it all.

- Title: **Advanced SCSI Drivers And Other Tales**

  Author Alan Cox.

  URL http://www.linux-mag.com/id/307

  Date 1999

Keywords SCSI, device, driver, advanced.

Description The title says it all.

- Title: **Writing Linux Mouse Drivers**

  Author Alan Cox.

  URL http://www.linux-mag.com/id/330

  Date 1999

  Keywords mouse, driver, gpm.

  Description The title says it all.

- Title: **More on Mouse Drivers**

  Author Alan Cox.

  URL http://www.linux-mag.com/id/356

  Date 1999

  Keywords mouse, driver, gpm, races, asynchronous I/O.

  Description The title still says it all.

- Title: **Writing Video4linux Radio Driver**

  Author Alan Cox.

  URL http://www.linux-mag.com/id/381

  Date 1999

  Keywords video4linux, driver, radio, radio devices.

  Description The title says it all.

- Title: **I/O Event Handling Under Linux**

  Author Richard Gooch.

  URL http://web.mit.edu/~yandros/doc/io-events.html

  Date 1999

  Keywords IO, I/O, select(2), poll(2), FDs, aio_read(2), readiness event queues.

  Description From the Introduction: "I/O Event handling is about how your Operating System allows you to manage a large number of open files (file descriptors in UNIX/POSIX, or FDs) in your application. You want the OS to notify you when FDs become active (have data ready to be read or are ready for writing). Ideally you want a mechanism that is scalable. This means a large number of inactive FDs cost very little in memory and CPU time to manage".

- Title: **(nearly) Complete Linux Loadable Kernel Modules. The definitive guide for hackers, virus coders and system administrators.**

  Author pragmatic/THC.

  URL http://packetstormsecurity.org/docs/hack/LKM_HACKING.html

  Date 1999

  Keywords syscalls, intercept, hide, abuse, symbol table.

  Description Interesting paper on how to abuse the Linux kernel in order to intercept and modify syscalls, make files/directories/processes invisible, become root, hijack ttys, write kernel modules based virus... and solutions for admins to avoid all those abuses.

  Notes For 2.0.x kernels. Gives guidances to port it to 2.2.x kernels.

- Name: **Linux Virtual File System**

Author Peter J. Braam.

URL http://www.coda.cs.cmu.edu/doc/talks/linuxvfs/

Date 1998

Keywords slides, VFS, inode, superblock, dentry, dcache.

Description Set of slides, presumably from a presentation on the Linux VFS layer. Covers version 2.1.x, with dentries and the dcache.

- Title: **The Venus kernel interface**

  Author Peter J. Braam.

  URL http://www.coda.cs.cmu.edu/doc/html/kernel-venus-protocol.html

  Date 1998

  Keywords coda, filesystem, venus, cache manager.

  Description "This document describes the communication between Venus and kernel level file system code needed for the operation of the Coda filesystem. This version document is meant to describe the current interface (version 1.0) as well as improvements we envisage".

- Title: **Design and Implementation of the Second Extended Filesystem**

  Author Rémy Card, Theodore Ts'o, Stephen Tweedie.

  URL http://web.mit.edu/tytso/www/linux/ext2intro.html

  Date 1998

  Keywords ext2, linux fs history, inode, directory, link, devices, VFS, physical structure, performance, benchmarks, ext2fs library, ext2fs tools, e2fsck.

  Description Paper written by three of the top ext2 hackers. Covers Linux filesystems history, ext2 motivation, ext2 features, design, physical structure on disk, performance, benchmarks, e2fsck's passes description... A must read!

  Notes This paper was first published in the Proceedings of the First Dutch International Symposium on Linux, ISBN 90-367-0385-9.

- Title: **The Linux RAID-1, 4, 5 Code**

  Author Ingo Molnar, Gadi Oxman and Miguel de Icaza.

  URL http://www.linuxjournal.com/article.php?sid=2391

  Date 1997

  Keywords RAID, MD driver.

  Description Linux Journal Kernel Korner article. Here is its

  Abstract *A description of the implementation of the RAID-1, RAID-4 and RAID-5 personalities of the MD device driver in the Linux kernel, providing users with high performance and reliable, secondary-storage capability using software*.

- Title: **Linux Kernel Hackers' Guide**

  Author Michael K. Johnson.

  URL http://www.tldp.org/LDP/khg/HyperNews/get/khg.html

  Date 1997

  Keywords device drivers, files, VFS, kernel interface, character vs block devices, hardware interrupts, scsi, DMA, access to user memory, memory allocation, timers.

  Description A guide designed to help you get up to speed on the concepts that are not intuitevly obvious, and to document the internal structures of Linux.

- Title: **Dynamic Kernels: Modularized Device Drivers**

  Author Alessandro Rubini.

  URL http://www.linuxjournal.com/article.php?sid=1219

  Date 1996

  Keywords device driver, module, loading/unloading modules, allocating resources.

  Description Linux Journal Kernel Korner article. Here is its

  Abstract *This is the first of a series of four articles co-authored by Alessandro Rubini and Georg Zezchwitz which present a practical approach to writing Linux device drivers as kernel loadable modules. This installment presents an introduction to the topic, preparing the reader to understand next month's installment*.

- Title: **Dynamic Kernels: Discovery**

  Author Alessandro Rubini.

  URL http://www.linuxjournal.com/article.php?sid=1220

  Date 1996

  Keywords character driver, init_module, clean_up module, autodetection, mayor number, minor number, file operations, open(), close().

  Description Linux Journal Kernel Korner article. Here is its

  Abstract *This article, the second of four, introduces part of the actual code to create custom module implementing a character device driver. It describes the code for module initialization and cleanup, as well as the open() and close() system calls*.

- Title: **The Devil's in the Details**

  Author Georg v. Zezschwitz and Alessandro Rubini.

  URL http://www.linuxjournal.com/article.php?sid=1221

  Date 1996

  Keywords read(), write(), select(), ioctl(), blocking/non blocking mode, interrupt handler.

  Description Linux Journal Kernel Korner article. Here is its

  Abstract *This article, the third of four on writing character device drivers, introduces concepts of reading, writing, and using ioctl-calls*.

- Title: **Dissecting Interrupts and Browsing DMA**

  Author Alessandro Rubini and Georg v. Zezschwitz.

  URL http://www.linuxjournal.com/article.php?sid=1222

  Date 1996

  Keywords interrupts, irqs, DMA, bottom halves, task queues.

  Description Linux Journal Kernel Korner article. Here is its

  Abstract *This is the fourth in a series of articles about writing character device drivers as loadable kernel modules. This month, we further investigate the field of interrupt handling. Though it is conceptually simple, practical limitations and constraints make this an ''interesting" part of device driver writing, and several different facilities have been provided for different situations. We also investigate the complex topic of DMA*.

- Title: **Device Drivers Concluded**

  Author Georg v. Zezschwitz.

  URL http://www.linuxjournal.com/article.php?sid=1287

  Date 1996

Keywords address spaces, pages, pagination, page management, demand loading, swapping, memory protection, memory mapping, mmap, virtual memory areas (VMAs), vremap, PCI.

Description Finally, the above turned out into a five articles series. This latest one's introduction reads: "This is the last of five articles about character device drivers. In this final section, Georg deals with memory mapping devices, beginning with an overall description of the Linux memory management concepts".

- Title: **Network Buffers And Memory Management**

  Author Alan Cox.

  URL http://www.linuxjournal.com/article.php?sid=1312

  Date 1996

  Keywords sk_buffs, network devices, protocol/link layer variables, network devices flags, transmit, receive, configuration, multicast.

  Description Linux Journal Kernel Korner.

  Abstract *Writing a network device driver for Linux is fundamentally simple—most of the complexity (other than talking to the hardware) involves managing network packets in memory*.

- Title: **Analysis of the Ext2fs structure**

  Author Louis-Dominique Dubeau.

  URL http://teaching.csse.uwa.edu.au/units/CITS2002/fs-ext2/

  Date 1994

  Keywords ext2, filesystem, ext2fs.

  Description Description of ext2's blocks, directories, inodes, bitmaps, invariants...

* Published books

- Title: **Linux Treiber entwickeln**

  Author Jürgen Quade, Eva-Katharina Kunst

  Publisher dpunkt.verlag

  Date Oct 2015 (4th edition)

  Pages 688

  ISBN 978-3-86490-288-8

  Note German. The third edition from 2011 is much cheaper and still quite up-to-date.

- Title: **Linux Kernel Networking: Implementation and Theory**

  Author Rami Rosen

  Publisher Apress

  Date December 22, 2013

  Pages 648

  ISBN 978-1430261964

- Title: **Embedded Linux Primer: A practical Real-World Approach, 2nd Edition**

  Author Christopher Hallinan

  Publisher Pearson

Date November, 2010

Pages 656

ISBN 978-0137017836

- Title: **Linux Kernel Development, 3rd Edition**

  Author Robert Love

  Publisher Addison-Wesley

  Date July, 2010

  Pages 440

  ISBN 978-0672329463

- Title: **Essential Linux Device Drivers**

  Author Sreekrishnan Venkateswaran

  Published Prentice Hall

  Date April, 2008

  Pages 744

  ISBN 978-0132396554

- Title: **Linux Device Drivers, 3rd Edition**

  Authors Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

  Publisher O'Reilly & Associates

  Date 2005

  Pages 636

  ISBN 0-596-00590-3

  Notes Further information in http://www.oreilly.com/catalog/linuxdrive3/ PDF format, URL: http://lwn.net/Kernel/LDD3/

- Title: **Linux Kernel Internals**

  Author Michael Beck

  Publisher Addison-Wesley

  Date 1997

  ISBN 0-201-33143-8 (second edition)

- Title: **Programmation Linux 2.0 API systeme et fonctionnement du noyau**

  Author Remy Card, Eric Dumas, Franck Mevel

  Publisher Eyrolles

  Date 1997

  Pages 520

  ISBN 2-212-08932-5

  Notes French

- Title: **The Design and Implementation of the 4.4 BSD UNIX Operating System**

  Author Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman

  Publisher Addison-Wesley

  Date 1996

ISBN 0-201-54979-4

- Title: **Unix internals – the new frontiers**

  Author Uresh Vahalia

  Publisher Prentice Hall

  Date 1996

  Pages 600

  ISBN 0-13-101908-2

- Title: **Programming for the real world - POSIX.4**

  Author Bill O. Gallmeister

  Publisher O'Reilly & Associates, Inc

  Date 1995

  Pages 552

  ISBN I-56592-074-0

  Notes Though not being directly about Linux, Linux aims to be POSIX. Good reference.

- Title: **UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers**

  Author Curt Schimmel

  Publisher Addison Wesley

  Date June, 1994

  Pages 432

  ISBN 0-201-63338-8

- Title: **The Design and Implementation of the 4.3 BSD UNIX Operating System**

  Author Samuel J. Leffler, Marshall Kirk McKusick, Michael J Karels, John S. Quarterman

  Publisher Addison-Wesley

  Date 1989 (reprinted with corrections on October, 1990)

  ISBN 0-201-06196-1

- Title: **The Design of the UNIX Operating System**

  Author Maurice J. Bach

  Publisher Prentice Hall

  Date 1986

  Pages 471

  ISBN 0-13-201757-1

* Miscellaneous

- Name: **Cross-Referencing Linux**

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

  Keywords Browsing source code.

  Description Another web-based Linux kernel source code browser. Lots of cross references to variables and functions. You can see where they are defined and where they are used.

- Name: **Linux Weekly News**

  URL http://lwn.net

  Keywords latest kernel news.

  Description The title says it all. There's a fixed kernel section summarizing developers' work, bug fixes, new features and versions produced during the week. Published every Thursday.

- Name: **The home page of Linux-MM**

  Author The Linux-MM team.

  URL http://linux-mm.org/

  Keywords memory management, Linux-MM, mm patches, TODO, docs, mailing list.

  Description Site devoted to Linux Memory Management development. Memory related patches, HOWTOs, links, mm developers... Don't miss it if you are interested in memory management development!

- Name: **Kernel Newbies IRC Channel and Website**

  URL http://www.kernelnewbies.org

  Keywords IRC, newbies, channel, asking doubts.

  Description #kernelnewbies on irc.oftc.net. #kernelnewbies is an IRC network dedicated to the 'newbie' kernel hacker. The audience mostly consists of people who are learning about the kernel, working on kernel projects or professional kernel hackers that want to help less seasoned kernel people. #kernelnewbies is on the OFTC IRC Network. Try irc.oftc.net as your server and then /join #kernelnewbies. The kernelnewbies website also hosts articles, documents, FAQs...

- Name: **linux-kernel mailing list archives and search engines**

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

  URL http://www.uwsg.indiana.edu/hypermail/linux/kernel/index.html

  URL http://groups.google.com/group/mlist.linux.kernel

  Keywords linux-kernel, archives, search.

  Description Some of the linux-kernel mailing list archivers. If you have a better/another one, please let me know.

Document last updated on Tue 2016-Sep-20

This document is based on: http://www.dit.upm.es/~jmseyas/linux/kernel/hackers-docs.html

These are some overall technical guides that have been put here for now for lack of a better place.

CHAPTER

FOURTEEN

APPLYING PATCHES TO THE LINUX KERNEL

Original by: Jesper Juhl, August 2005

Note:

This document is obsolete. In most cases, rather than using `patch` manually, you'll almost certainly want to look at using Git instead.

A frequently asked question on the Linux Kernel Mailing List is how to apply a patch to the kernel or, more specifically, what base kernel a patch for one of the many trees/branches should be applied to. Hopefully this document will explain this to you.

In addition to explaining how to apply and revert patches, a brief description of the different kernel trees (and examples of how to apply their specific patches) is also provided.

* What is a patch?

A patch is a small text document containing a delta of changes between two different versions of a source tree. Patches are created with the `diff` program.

To correctly apply a patch you need to know what base it was generated from and what new version the patch will change the source tree into. These should both be present in the patch file metadata or be possible to deduce from the filename.

* How do I apply or revert a patch?

You apply a patch with the `patch` program. The patch program reads a diff (or patch) file and makes the changes to the source tree described in it.

Patches for the Linux kernel are generated relative to the parent directory holding the kernel source dir.

This means that paths to files inside the patch file contain the name of the kernel source directories it was generated against (or some other directory names like "a/" and "b/").

Since this is unlikely to match the name of the kernel source dir on your local machine (but is often useful info to see what version an otherwise unlabeled patch was generated against) you should change into your kernel source directory and then strip the first element of the path from filenames in the patch file when applying it (the `-p1` argument to `patch` does this).

To revert a previously applied patch, use the -R argument to patch. So, if you applied a patch like this:

```
patch -p1 < ../patch-x.y.z
```

You can revert (undo) it like this:

```
patch -R -p1 < ../patch-x.y.z
```

* How do I feed a patch/diff file to `patch`?

This (as usual with Linux and other UNIX like operating systems) can be done in several different ways.

In all the examples below I feed the file (in uncompressed form) to patch via stdin using the following syntax:

```
patch -p1 < path/to/patch-x.y.z
```

If you just want to be able to follow the examples below and don't want to know of more than one way to use patch, then you can stop reading this section here.

Patch can also get the name of the file to use via the -i argument, like this:

```
patch -p1 -i path/to/patch-x.y.z
```

If your patch file is compressed with gzip or xz and you don't want to uncompress it before applying it, then you can feed it to patch like this instead:

```
xzcat path/to/patch-x.y.z.xz | patch -p1
bzcat path/to/patch-x.y.z.gz | patch -p1
```

If you wish to uncompress the patch file by hand first before applying it (what I assume you've done in the examples below), then you simply run gunzip or xz on the file – like this:

```
gunzip patch-x.y.z.gz
xz -d patch-x.y.z.xz
```

Which will leave you with a plain text patch-x.y.z file that you can feed to patch via stdin or the `-i` argument, as you prefer.

A few other nice arguments for patch are `-s` which causes patch to be silent except for errors which is nice to prevent errors from scrolling out of the screen too fast, and `--dry-run` which causes patch to just print a listing of what would happen, but doesn't actually make any changes. Finally `--verbose` tells patch to print more information about the work being done.

* Common errors when patching

When patch applies a patch file it attempts to verify the sanity of the file in different ways.

Checking that the file looks like a valid patch file and checking the code around the bits being modified matches the context provided in the patch are just two of the basic sanity checks patch does.

If patch encounters something that doesn't look quite right it has two options. It can either refuse to apply the changes and abort or it can try to find a way to make the patch apply with a few minor changes.

One example of something that's not 'quite right' that patch will attempt to fix up is if all the context matches, the lines being changed match, but the line numbers are different. This can happen, for example, if the patch makes a change in the middle of the file but for some reasons a few lines have been added or removed near the beginning of the file. In that case everything looks good it has just moved up or down a bit, and patch will usually adjust the line numbers and apply the patch.

Whenever patch applies a patch that it had to modify a bit to make it fit it'll tell you about it by saying the patch applied with **fuzz**. You should be wary of such changes since even though patch probably got it right it doesn't /always/ get it right, and the result will sometimes be wrong.

When patch encounters a change that it can't fix up with fuzz it rejects it outright and leaves a file with a `.rej` extension (a reject file). You can read this file to see exactly what change couldn't be applied, so you can go fix it up by hand if you wish.

If you don't have any third-party patches applied to your kernel source, but only patches from kernel.org and you apply the patches in the correct order, and have made no modifications yourself to the source files, then you should never see a fuzz or reject message from patch. If you do see such messages anyway, then there's a high risk that either your local source tree or the patch file is corrupted in some way. In that case you should probably try re-downloading the patch and if things are still not OK then you'd be advised to start with a fresh tree downloaded in full from kernel.org.

Let's look a bit more at some of the messages patch can produce.

If patch stops and presents a `File to patch:` prompt, then patch could not find a file to be patched. Most likely you forgot to specify -p1 or you are in the wrong directory. Less often, you'll find patches that need to be applied with `-p0` instead of `-p1` (reading the patch file should reveal if this is the case – if so, then this is an error by the person who created the patch but is not fatal).

If you get `Hunk #2 succeeded at 1887 with fuzz 2 (offset 7 lines).` or a message similar to that, then it means that patch had to adjust the location of the change (in this example it needed to move 7 lines from where it expected to make the change to make it fit).

The resulting file may or may not be OK, depending on the reason the file was different than expected.

This often happens if you try to apply a patch that was generated against a different kernel version than the one you are trying to patch.

If you get a message like `Hunk #3 FAILED at 2387.`, then it means that the patch could not be applied correctly and the patch program was unable to fuzz its way through. This will generate a `.rej` file with the change that caused the patch to fail and also a `.orig` file showing you the original content that couldn't be changed.

If you get `Reversed (or previously applied) patch detected! Assume -R? [n]` then patch detected that the change contained in the patch seems to have already been made.

If you actually did apply this patch previously and you just re-applied it in error, then just say [n]o and abort this patch. If you applied this patch previously and actually intended to revert it, but forgot to specify -R, then you can say [**y**]es here to make patch revert it for you.

This can also happen if the creator of the patch reversed the source and destination directories when creating the patch, and in that case reverting the patch will in fact apply it.

A message similar to `patch: **** unexpected end of file in patch` or `patch unexpectedly ends in middle of line` means that patch could make no sense of the file you fed to it. Either your download is broken, you tried to feed patch a compressed patch file without uncompressing it first, or the patch file that you are using has been mangled by a mail client or mail transfer agent along the way somewhere, e.g., by splitting a long line into two lines. Often these warnings can easily be fixed by joining (concatenating) the two lines that had been split.

As I already mentioned above, these errors should never happen if you apply a patch from kernel.org to the correct version of an unmodified source tree. So if you get these errors with kernel.org patches then you should probably assume that either your patch file or your tree is broken and I'd advise you to start over with a fresh download of a full kernel tree and the patch you wish to apply.

* Are there any alternatives to `patch`?

Yes there are alternatives.

You can use the `interdiff` program (http://cyberelk.net/tim/patchutils/) to generate a patch representing the differences between two patches and then apply the result.

This will let you move from something like 4.7.2 to 4.7.3 in a single step. The -z flag to interdiff will even let you feed it patches in gzip or bzip2 compressed form directly without the use of zcat or bzcat or manual decompression.

Here's how you'd go from 4.7.2 to 4.7.3 in a single step:

```
interdiff -z ../patch-4.7.2.gz ../patch-4.7.3.gz | patch -p1
```

Although interdiff may save you a step or two you are generally advised to do the additional steps since interdiff can get things wrong in some cases.

Another alternative is `ketchup`, which is a python script for automatic downloading and applying of patches (http://www.selenic.com/ketchup/).

Other nice tools are diffstat, which shows a summary of changes made by a patch; lsdiff, which displays a short listing of affected files in a patch file, along with (optionally) the line numbers of the start of each patch; and grepdiff, which displays a list of the files modified by a patch where the patch contains a given regular expression.

* Where can I download the patches?

The patches are available at http://kernel.org/ Most recent patches are linked from the front page, but they also have specific homes.

The 4.x.y (-stable) and 4.x patches live at

> https://www.kernel.org/pub/linux/kernel/v4.x/

The -rc patches live at

> https://www.kernel.org/pub/linux/kernel/v4.x/testing/

* The 4.x kernels

These are the base stable releases released by Linus. The highest numbered release is the most recent.

If regressions or other serious flaws are found, then a -stable fix patch will be released (see below) on top of this base. Once a new 4.x base kernel is released, a patch is made available that is a delta between the previous 4.x kernel and the new one.

To apply a patch moving from 4.6 to 4.7, you'd do the following (note that such patches do **NOT** apply on top of 4.x.y kernels but on top of the base 4.x kernel – if you need to move from 4.x.y to 4.x+1 you need to first revert the 4.x.y patch).

Here are some examples:

```
# moving from 4.6 to 4.7

$ cd ~/linux-4.6                # change to kernel source dir
$ patch -p1 < ../patch-4.7      # apply the 4.7 patch
$ cd ..
$ mv linux-4.6 linux-4.7        # rename source dir

# moving from 4.6.1 to 4.7

$ cd ~/linux-4.6.1              # change to kernel source dir
$ patch -p1 -R < ../patch-4.6.1 # revert the 4.6.1 patch
                                # source dir is now 4.6
$ patch -p1 < ../patch-4.7      # apply new 4.7 patch
$ cd ..
$ mv linux-4.6.1 linux-4.7      # rename source dir
```

* The 4.x.y kernels

Kernels with 3-digit versions are -stable kernels. They contain small(ish) 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.

Note:

The -stable team usually do make incremental patches available as well as patches against the latest mainline release, but I only cover the non-incremental ones below. The incremental ones can be found at https://www.kernel.org/pub/linux/kernel/v4.x/incr/

These patches are not incremental, meaning that for example the 4.7.3 patch does not apply on top of the 4.7.2 kernel source, but rather on top of the base 4.7 kernel source.

So, in order to apply the 4.7.3 patch to your existing 4.7.2 kernel source you have to first back out the 4.7.2 patch (so you are left with a base 4.7 kernel source) and then apply the new 4.7.3 patch.

Here's a small example:

```
$ cd ~/linux-4.7.2              # change to the kernel source dir
$ patch -p1 -R < ../patch-4.7.2 # revert the 4.7.2 patch
$ patch -p1 < ../patch-4.7.3    # apply the new 4.7.3 patch
$ cd ..
$ mv linux-4.7.2 linux-4.7.3    # rename the kernel source dir
```

* The -rc kernels

These are release-candidate kernels. These are development kernels released by Linus whenever he deems the current git (the kernel's source management tool) tree to be in a reasonably sane state adequate for testing.

These kernels are not stable and you should expect occasional breakage if you intend to run them. This is however the most stable of the main development branches and is also what will eventually turn into the next stable kernel, so it is important that it be tested by as many people as possible.

This is a good branch to run for people who want to help out testing development kernels but do not want to run some of the really experimental stuff (such people should see the sections about -next and -mm kernels below).

The -rc patches are not incremental, they apply to a base 4.x kernel, just like the 4.x.y patches described above. The kernel version before the -rcN suffix denotes the version of the kernel that this -rc kernel will eventually turn into.

So, 4.8-rc5 means that this is the fifth release candidate for the 4.8 kernel and the patch should be applied on top of the 4.7 kernel source.

Here are 3 examples of how to apply these patches:

```
# first an example of moving from 4.7 to 4.8-rc3

$ cd ~/linux-4.7                        # change to the 4.7 source dir
$ patch -p1 < ../patch-4.8-rc3          # apply the 4.8-rc3 patch
$ cd ..
$ mv linux-4.7 linux-4.8-rc3            # rename the source dir
```

```
# now let's move from 4.8-rc3 to 4.8-rc5

$ cd ~/linux-4.8-rc3                    # change to the 4.8-rc3 dir
$ patch -p1 -R < ../patch-4.8-rc3       # revert the 4.8-rc3 patch
$ patch -p1 < ../patch-4.8-rc5          # apply the new 4.8-rc5 patch
$ cd ..
$ mv linux-4.8-rc3 linux-4.8-rc5        # rename the source dir

# finally let's try and move from 4.7.3 to 4.8-rc5

$ cd ~/linux-4.7.3                      # change to the kernel source dir
$ patch -p1 -R < ../patch-4.7.3         # revert the 4.7.3 patch
$ patch -p1 < ../patch-4.8-rc5          # apply new 4.8-rc5 patch
$ cd ..
$ mv linux-4.7.3 linux-4.8-rc5          # rename the kernel source dir
```

* The -mm patches and the linux-next tree

The -mm patches are experimental patches released by Andrew Morton.

In the past, -mm tree were used to also test subsystem patches, but this function is now done via the *linux-next <https://www.kernel.org/doc/man-pages/linux-next.html>* tree. The Subsystem maintainers push their patches first to linux-next, and, during the merge window, sends them directly to Linus.

The -mm patches serve as a sort of proving ground for new features and other experimental patches that aren't merged via a subsystem tree. Once such patches has proved its worth in -mm for a while Andrew pushes it on to Linus for inclusion in mainline.

The linux-next tree is daily updated, and includes the -mm patches. Both are in constant flux and contains many experimental features, a lot of debugging patches not appropriate for mainline etc., and is the most experimental of the branches described in this document.

These patches are not appropriate for use on systems that are supposed to be stable and they are more risky to run than any of the other branches (make sure you have up-to-date backups – that goes for any experimental kernel but even more so for -mm patches or using a Kernel from the linux-next tree).

Testing of -mm patches and linux-next is greatly appreciated since the whole point of those are to weed out regressions, crashes, data corruption bugs, build breakage (and any other bug in general) before changes are merged into the more stable mainline Linus tree.

But testers of -mm and linux-next should be aware that breakages are more common than in any other tree.

This concludes this list of explanations of the various kernel trees. I hope you are now clear on how to apply the various patches and help testing the kernel.

Thank you's to Randy Dunlap, Rolf Eike Beer, Linus Torvalds, Bodo Eggert, Johannes Stezenbach, Grant Coady, Pavel Machek and others that I may have forgotten for their reviews and contributions to this document.

CHAPTER

FIFTEEN

ADDING A NEW SYSTEM CALL

This document describes what's involved in adding a new system call to the Linux kernel, over and above the normal submission advice in *Documentation/process/submitting-patches.rst* .

* System Call Alternatives

The first thing to consider when adding a new system call is whether one of the alternatives might be suitable instead. Although system calls are the most traditional and most obvious interaction points between userspace and the kernel, there are other possibilities – choose what fits best for your interface.

- If the operations involved can be made to look like a filesystem-like object, it may make more sense to create a new filesystem or device. This also makes it easier to encapsulate the new functionality in a kernel module rather than requiring it to be built into the main kernel.
  - If the new functionality involves operations where the kernel notifies userspace that something has happened, then returning a new file descriptor for the relevant object allows userspace to use `poll`/`select`/`epoll` to receive that notification.
  - However, operations that don't map to *`read(2)`*/*`write(2)`*-like operations have to be implemented as *`ioctl(2)`* requests, which can lead to a somewhat opaque API.
- If you're just exposing runtime system information, a new node in sysfs (see `Documentation/filesystems/sysfs.txt`) or the `/proc` filesystem may be more appropriate. However, access to these mechanisms requires that the relevant filesystem is mounted, which might not always be the case (e.g. in a namespaced/sandboxed/chrooted environment). Avoid adding any API to debugfs, as this is not considered a 'production' interface to userspace.
- If the operation is specific to a particular file or file descriptor, then an additional *`fcntl(2)`* command option may be more appropriate. However, *`fcntl(2)`* is a multiplexing system call that hides a lot of complexity, so this option is best for when the new function is closely analogous to existing *`fcntl(2)`* functionality, or the new functionality is very simple (for example, getting/setting a simple flag related to a file descriptor).
- If the operation is specific to a particular task or process, then an additional *`prctl(2)`* command option may be more appropriate. As with *`fcntl(2)`*, this system call is a complicated multiplexor so is best reserved for near-analogs of existing `prctl()` commands or getting/setting a simple flag related to a process.

* Designing the API: Planning for Extension

A new system call forms part of the API of the kernel, and has to be supported indefinitely. As such, it's a very good idea to explicitly discuss the interface on the kernel mailing list, and it's important to plan for future extensions of the interface.

(The syscall table is littered with historical examples where this wasn't done, together with the corresponding follow-up system calls – `eventfd/eventfd2`, `dup2/dup3`, `inotify_init/inotify_init1`, `pipe/pipe2`, `renameat/renameat2` – so learn from the history of the kernel and plan for extensions from the start.)

For simpler system calls that only take a couple of arguments, the preferred way to allow for future extensibility is to include a flags argument to the system call. To make sure that userspace programs can safely use flags between kernel versions, check whether the flags value holds any unknown flags, and reject the system call (with `EINVAL`) if it does:

```
if (flags & ~(THING_FLAG1 | THING_FLAG2 | THING_FLAG3))
    return -EINVAL;
```

(If no flags values are used yet, check that the flags argument is zero.)

For more sophisticated system calls that involve a larger number of arguments, it's preferred to encapsulate the majority of the arguments into a structure that is passed in by pointer. Such a structure can cope with future extension by including a size argument in the structure:

```
struct xyzzy_params {
    u32 size; /* userspace sets p->size = sizeof(struct xyzzy_params) */
    u32 param_1;
    u64 param_2;
    u64 param_3;
};
```

As long as any subsequently added field, say `param_4`, is designed so that a zero value gives the previous behaviour, then this allows both directions of version mismatch:

- To cope with a later userspace program calling an older kernel, the kernel code should check that any memory beyond the size of the structure that it expects is zero (effectively checking that `param_4 == 0`).
- To cope with an older userspace program calling a newer kernel, the kernel code can zero-extend a smaller instance of the structure (effectively setting `param_4 = 0`).

See *`perf_event_open(2)`* and the `perf_copy_attr()` function (in `kernel/events/core.c`) for an example of this approach.

* Designing the API: Other Considerations

If your new system call allows userspace to refer to a kernel object, it should use a file descriptor as the handle for that object – don't invent a new type of userspace object handle when the kernel already has mechanisms and well-defined semantics for using file descriptors.

If your new *`xyzzy(2)`* system call does return a new file descriptor, then the flags argument should include a value that is equivalent to setting `O_CLOEXEC` on the new FD. This makes it possible for userspace to close the timing window between `xyzzy()` and calling `fcntl(fd,F_SETFD,FD_CLOEXEC)`, where an unexpected `fork()` and `execve()` in another thread could leak a descriptor to the exec'ed program. (However, resist the temptation to re-use the actual value of the `O_CLOEXEC` constant, as it is architecture-specific and is part of a numbering space of `O_*` flags that is fairly full.)

If your system call returns a new file descriptor, you should also consider what it means to use the *`poll(2)`* family of system calls on that file descriptor. Making a file descriptor ready for reading or writing is the normal way for the kernel to indicate to userspace that an event has occurred on the corresponding kernel object.

If your new *`xyzzy(2)`* system call involves a filename argument:

```
int sys_xyzzy(const char __user *path, ..., unsigned int flags);
```

you should also consider whether an *`xyzzyat(2)`* version is more appropriate:

```
int sys_xyzzyat(int dfd, const char __user *path, ..., unsigned int flags);
```

This allows more flexibility for how userspace specifies the file in question; in particular it allows userspace to request the functionality for an already-opened file descriptor using the AT_EMPTY_PATH flag, effectively giving an *fxyzzy(3)* operation for free:

```
- xyzzyat(AT_FDCWD, path, ..., 0) is equivalent to xyzzy(path,...)
- xyzzyat(fd, "", ..., AT_EMPTY_PATH) is equivalent to fxyzzy(fd, ...)
```

(For more details on the rationale of the *at() calls, see the *openat(2)* man page; for an example of AT_EMPTY_PATH, see the *fstatat(2)* man page.)

If your new *xyzzy(2)* system call involves a parameter describing an offset within a file, make its type `loff_t` so that 64-bit offsets can be supported even on 32-bit architectures.

If your new *xyzzy(2)* system call involves privileged functionality, it needs to be governed by the appropriate Linux capability bit (checked with a call to `capable()`), as described in the *capabilities(7)* man page. Choose an existing capability bit that governs related functionality, but try to avoid combining lots of only vaguely related functions together under the same bit, as this goes against capabilities' purpose of splitting the power of root. In particular, avoid adding new uses of the already overly-general `CAP_SYS_ADMIN` capability.

If your new *xyzzy(2)* system call manipulates a process other than the calling process, it should be restricted (using a call to `ptrace_may_access()`) so that only a calling process with the same permissions as the target process, or with the necessary capabilities, can manipulate the target process.

Finally, be aware that some non-x86 architectures have an easier time if system call parameters that are explicitly 64-bit fall on odd-numbered arguments (i.e. parameter 1, 3, 5), to allow use of contiguous pairs of 32-bit registers. (This concern does not apply if the arguments are part of a structure that's passed in by pointer.)

* Proposing the API

To make new system calls easy to review, it's best to divide up the patchset into separate chunks. These should include at least the following items as distinct commits (each of which is described further below):

- The core implementation of the system call, together with prototypes, generic numbering, Kconfig changes and fallback stub implementation.
- Wiring up of the new system call for one particular architecture, usually x86 (including all of x86_64, x86_32 and x32).
- A demonstration of the use of the new system call in userspace via a selftest in `tools/testing/selftests/`.
- A draft man-page for the new system call, either as plain text in the cover letter, or as a patch to the (separate) man-pages repository.

New system call proposals, like any change to the kernel's API, should always be cc'ed to linux-api@vger.kernel.org.

* Generic System Call Implementation

The main entry point for your new *xyzzy(2)* system call will be called `sys_xyzzy()`, but you add this entry point with the appropriate `SYSCALL_DEFINEn()` macro rather than explicitly. The 'n' indicates the number of arguments to the system call, and the macro takes the system call name followed by the (type, name) pairs for the parameters as arguments. Using this macro allows metadata about the new system call to be made available for other tools.

The new entry point also needs a corresponding function prototype, in include/linux/syscalls.h, marked as asmlinkage to match the way that system calls are invoked:

```
asmlinkage long sys_xyzzy(...);
```

Some architectures (e.g. x86) have their own architecture-specific syscall tables, but several other architectures share a generic syscall table. Add your new system call to the generic list by adding an entry to the list in include/uapi/asm-generic/unistd.h:

```
#define __NR_xyzzy 292
__SYSCALL(__NR_xyzzy, sys_xyzzy)
```

Also update the __NR_syscalls count to reflect the additional system call, and note that if multiple new system calls are added in the same merge window, your new syscall number may get adjusted to resolve conflicts.

The file kernel/sys_ni.c provides a fallback stub implementation of each system call, returning -ENOSYS. Add your new system call here too:

```
cond_syscall(sys_xyzzy);
```

Your new kernel functionality, and the system call that controls it, should normally be optional, so add a CONFIG option (typically to init/Kconfig) for it. As usual for new CONFIG options:

- Include a description of the new functionality and system call controlled by the option.
- Make the option depend on EXPERT if it should be hidden from normal users.
- Make any new source files implementing the function dependent on the CONFIG option in the Makefile (e.g. obj-$(CONFIG_XYZZY_SYSCALL) += xyzzy.c).
- Double check that the kernel still builds with the new CONFIG option turned off.

To summarize, you need a commit that includes:

- CONFIG option for the new function, normally in init/Kconfig
- SYSCALL_DEFINEn(xyzzy, ...) for the entry point
- corresponding prototype in include/linux/syscalls.h
- generic table entry in include/uapi/asm-generic/unistd.h
- fallback stub in kernel/sys_ni.c

* x86 System Call Implementation

To wire up your new system call for x86 platforms, you need to update the master syscall tables. Assuming your new system call isn't special in some way (see below), this involves a "common" entry (for x86_64 and x32) in arch/x86/entry/syscalls/syscall_64.tbl:

```
333   common   xyzzy     sys_xyzzy
```

and an "i386" entry in arch/x86/entry/syscalls/syscall_32.tbl:

```
380   i386     xyzzy     sys_xyzzy
```

Again, these numbers are liable to be changed if there are conflicts in the relevant merge window.

* Compatibility System Calls (Generic)

For most system calls the same 64-bit implementation can be invoked even when the userspace program is itself 32-bit; even if the system call's parameters include an explicit pointer, this is handled transparently.

However, there are a couple of situations where a compatibility layer is needed to cope with size differences between 32-bit and 64-bit.

The first is if the 64-bit kernel also supports 32-bit userspace programs, and so needs to parse areas of (`__user`) memory that could hold either 32-bit or 64-bit values. In particular, this is needed whenever a system call argument is:

- a pointer to a pointer
- a pointer to a struct containing a pointer (e.g. `struct iovec __user *`)
- a pointer to a varying sized integral type (`time_t`, `off_t`, `long`, …)
- a pointer to a struct containing a varying sized integral type.

The second situation that requires a compatibility layer is if one of the system call's arguments has a type that is explicitly 64-bit even on a 32-bit architecture, for example `loff_t` or `__u64`. In this case, a value that arrives at a 64-bit kernel from a 32-bit application will be split into two 32-bit values, which then need to be re-assembled in the compatibility layer.

(Note that a system call argument that's a pointer to an explicit 64-bit type does **not** need a compatibility layer; for example, *`splice(2)`*'s arguments of type `loff_t __user *` do not trigger the need for a `compat_` system call.)

The compatibility version of the system call is called `compat_sys_xyzzy()`, and is added with the `COMPAT_SYSCALL_DEFINEn()` macro, analogously to SYSCALL_DEFINEn. This version of the implementation runs as part of a 64-bit kernel, but expects to receive 32-bit parameter values and does whatever is needed to deal with them. (Typically, the `compat_sys_` version converts the values to 64-bit versions and either calls on to the `sys_` version, or both of them call a common inner implementation function.)

The compat entry point also needs a corresponding function prototype, in `include/linux/compat.h`, marked as asmlinkage to match the way that system calls are invoked:

```
asmlinkage long compat_sys_xyzzy(...);
```

If the system call involves a structure that is laid out differently on 32-bit and 64-bit systems, say `struct xyzzy_args`, then the include/linux/compat.h header file should also include a compat version of the structure (`struct compat_xyzzy_args`) where each variable-size field has the appropriate `compat_` type that corresponds to the type in `struct xyzzy_args`. The `compat_sys_xyzzy()` routine can then use this `compat_` structure to parse the arguments from a 32-bit invocation.

For example, if there are fields:

```
struct xyzzy_args {
    const char __user *ptr;
    __kernel_long_t varying_val;
    u64 fixed_val;
    /* ... */
};
```

in struct xyzzy_args, then struct compat_xyzzy_args would have:

```
struct compat_xyzzy_args {
    compat_uptr_t ptr;
    compat_long_t varying_val;
    u64 fixed_val;
    /* ... */
};
```

The generic system call list also needs adjusting to allow for the compat version; the entry in `include/uapi/asm-generic/unistd.h` should use `__SC_COMP` rather than `__SYSCALL`:

```
#define __NR_xyzzy 292
__SC_COMP(__NR_xyzzy, sys_xyzzy, compat_sys_xyzzy)
```

To summarize, you need:

- a `COMPAT_SYSCALL_DEFINEn(xyzzy, ...)` for the compat entry point
- corresponding prototype in `include/linux/compat.h`
- (if needed) 32-bit mapping struct in `include/linux/compat.h`
- instance of `__SC_COMP` not `__SYSCALL` in `include/uapi/asm-generic/unistd.h`

* Compatibility System Calls (x86)

To wire up the x86 architecture of a system call with a compatibility version, the entries in the syscall tables need to be adjusted.

First, the entry in `arch/x86/entry/syscalls/syscall_32.tbl` gets an extra column to indicate that a 32-bit userspace program running on a 64-bit kernel should hit the compat entry point:

```
380   i386     xyzzy     sys_xyzzy    compat_sys_xyzzy
```

Second, you need to figure out what should happen for the x32 ABI version of the new system call. There's a choice here: the layout of the arguments should either match the 64-bit version or the 32-bit version.

If there's a pointer-to-a-pointer involved, the decision is easy: x32 is ILP32, so the layout should match the 32-bit version, and the entry in `arch/x86/entry/syscalls/syscall_64.tbl` is split so that x32 programs hit the compatibility wrapper:

```
333   64       xyzzy     sys_xyzzy
...
555   x32      xyzzy     compat_sys_xyzzy
```

If no pointers are involved, then it is preferable to re-use the 64-bit system call for the x32 ABI (and consequently the entry in arch/x86/entry/syscalls/syscall_64.tbl is unchanged).

In either case, you should check that the types involved in your argument layout do indeed map exactly from x32 (-mx32) to either the 32-bit (-m32) or 64-bit (-m64) equivalents.

* System Calls Returning Elsewhere

For most system calls, once the system call is complete the user program continues exactly where it left off – at the next instruction, with the stack the same and most of the registers the same as before the system call, and with the same virtual memory space.

However, a few system calls do things differently. They might return to a different location (`rt_sigreturn`) or change the memory space (`fork/vfork/clone`) or even architecture (`execve/execveat`) of the program.

To allow for this, the kernel implementation of the system call may need to save and restore additional registers to the kernel stack, allowing complete control of where and how execution continues after the system call.

This is arch-specific, but typically involves defining assembly entry points that save/restore additional registers and invoke the real system call entry point.

For x86_64, this is implemented as a `stub_xyzzy` entry point in `arch/x86/entry/entry_64.S`, and the entry in the syscall table (`arch/x86/entry/syscalls/syscall_64.tbl`) is adjusted to match:

```
333   common   xyzzy     stub_xyzzy
```

The equivalent for 32-bit programs running on a 64-bit kernel is normally called `stub32_xyzzy` and implemented in `arch/x86/entry/entry_64_compat.S`, with the corresponding syscall table adjustment in `arch/x86/entry/syscalls/syscall_32.tbl`:

```
380   i386     xyzzy     sys_xyzzy    stub32_xyzzy
```

If the system call needs a compatibility layer (as in the previous section) then the `stub32_` version needs to call on to the `compat_sys_` version of the system call rather than the native 64-bit version. Also, if the x32 ABI implementation is not common with the x86_64 version, then its syscall table will also need to invoke a stub that calls on to the `compat_sys_` version.

For completeness, it's also nice to set up a mapping so that user-mode Linux still works – its syscall table will reference stub_xyzzy, but the UML build doesn't include `arch/x86/entry/entry_64.S` implementation (because UML simulates registers etc). Fixing this is as simple as adding a #define to `arch/x86/um/sys_call_table_64.c`:

```
#define stub_xyzzy sys_xyzzy
```

* Other Details

Most of the kernel treats system calls in a generic way, but there is the occasional exception that may need updating for your particular system call.

The audit subsystem is one such special case; it includes (arch-specific) functions that classify some special types of system call – specifically file open (open/`openat`), program execution (`execve`/`exeveat`) or socket multiplexor (`socketcall`) operations. If your new system call is analogous to one of these, then the audit system should be updated.

More generally, if there is an existing system call that is analogous to your new system call, it's worth doing a kernel-wide grep for the existing system call to check there are no other special cases.

* Testing

A new system call should obviously be tested; it is also useful to provide reviewers with a demonstration of how user space programs will use the system call. A good way to combine these aims is to include a simple self-test program in a new directory under `tools/testing/selftests/`.

For a new system call, there will obviously be no libc wrapper function and so the test will need to invoke it using `syscall()`; also, if the system call involves a new userspace-visible structure, the corresponding header will need to be installed to compile the test.

Make sure the selftest runs successfully on all supported architectures. For example, check that it works when compiled as an x86_64 (-m64), x86_32 (-m32) and x32 (-mx32) ABI program.

For more extensive and thorough testing of new functionality, you should also consider adding tests to the Linux Test Project, or to the xfstests project for filesystem-related changes.

- https://linux-test-project.github.io/
- git://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git

* Man Page

All new system calls should come with a complete man page, ideally using groff markup, but plain text will do. If groff is used, it's helpful to include a pre-rendered ASCII version of the man page in the cover email for the patchset, for the convenience of reviewers.

The man page should be cc'ed to linux-man@vger.kernel.org For more details, see https://www.kernel.org/doc/man-pages/patches.html

* References and Sources

- LWN article from Michael Kerrisk on use of flags argument in system calls: https://lwn.net/Articles/585415/
- LWN article from Michael Kerrisk on how to handle unknown flags in a system call: https://lwn.net/Articles/588444/
- LWN article from Jake Edge describing constraints on 64-bit system call arguments: https://lwn.net/Articles/311630/
- Pair of LWN articles from David Drysdale that describe the system call implementation paths in detail for v3.14:
  - https://lwn.net/Articles/604287/
  - https://lwn.net/Articles/604515/
- Architecture-specific requirements for system calls are discussed in the *syscall(2)* man-page: http://man7.org/linux/man-pages/man2/syscall.2.html#NOTES
- Collated emails from Linus Torvalds discussing the problems with `ioctl()`: http://yarchive.net/comp/linux/ioctl.html
- "How to not invent kernel interfaces", Arnd Bergmann, http://www.ukuug.org/events/linux2007/2007/papers/Bergmann.pdf
- LWN article from Michael Kerrisk on avoiding new uses of CAP_SYS_ADMIN: https://lwn.net/Articles/486306/
- Recommendation from Andrew Morton that all related information for a new system call should come in the same email thread: https://lkml.org/lkml/2014/7/24/641
- Recommendation from Michael Kerrisk that a new system call should come with a man page: https://lkml.org/lkml/2014/6/13/309
- Suggestion from Thomas Gleixner that x86 wire-up should be in a separate commit: https://lkml.org/lkml/2014/11/19/254
- Suggestion from Greg Kroah-Hartman that it's good for new system calls to come with a man-page & selftest: https://lkml.org/lkml/2014/3/19/710
- Discussion from Michael Kerrisk of new system call vs. *prctl(2)* extension: https://lkml.org/lkml/2014/6/3/411
- Suggestion from Ingo Molnar that system calls that involve multiple arguments should encapsulate those arguments in a struct, which includes a size field for future extensibility: https://lkml.org/lkml/2015/7/30/117
- Numbering oddities arising from (re-)use of O_* numbering space flags:
  - commit 75069f2b5bfb ("vfs: renumber FMODE_NONOTIFY and add to uniqueness check")
  - commit 12ed2e36c98a ("fanotify: FMODE_NONOTIFY and __O_SYNC in sparc conflict")
  - commit bb458c644a59 ("Safer ABI for O_TMPFILE")
- Discussion from Matthew Wilcox about restrictions on 64-bit arguments: https://lkml.org/lkml/2008/12/12/187
- Recommendation from Greg Kroah-Hartman that unknown flags should be policed: https://lkml.org/lkml/2014/7/17/577

- Recommendation from Linus Torvalds that x32 system calls should prefer compatibility with 64-bit versions rather than 32-bit versions: https://lkml.org/lkml/2011/8/31/244

CHAPTER

SIXTEEN

LINUX MAGIC NUMBERS

This file is a registry of magic numbers which are in use. When you add a magic number to a structure, you should also add it to this file, since it is best if the magic numbers used by various structures are unique.

It is a **very** good idea to protect kernel data structures with magic numbers. This allows you to check at run time whether (a) a structure has been clobbered, or (b) you've passed the wrong structure to a routine. This last is especially useful — particularly when you are passing pointers to structures via a void * pointer. The tty code, for example, does this frequently to pass driver-specific and line discipline-specific structures back and forth.

The way to use magic numbers is to declare then at the beginning of the structure, like so:

```
struct tty_ldisc {
        int     magic;
        ...
};
```

Please follow this discipline when you are adding future enhancements to the kernel! It has saved me countless hours of debugging, especially in the screwy cases where an array has been overrun and structures following the array have been overwritten. Using this discipline, these cases get detected quickly and safely.

Changelog:

```
                                      Theodore Ts'o
                                      31 Mar 94

The magic table is current to Linux 2.1.55.

                                      Michael Chastain
                                      <mailto:mec@shout.net>
                                      22 Sep 1997

Now it should be up to date with Linux 2.1.112. Because
we are in feature freeze time it is very unlikely that
something will change before 2.2.x. The entries are
sorted by number field.

                                      Krzysztof G. Baranowski
                                      <mailto: kgb@knm.org.pl>
                                      29 Jul 1998

Updated the magic table to Linux 2.5.45. Right over the feature freeze,
but it is possible that some new magic numbers will sneak into the
kernel before 2.6.x yet.

                                      Petr Baudis
                                      <pasky@ucw.cz>
                                      03 Nov 2002
```

```

Updated the magic table to Linux 2.5.74.

                                Fabian Frederick
                                <ffrederick@users.sourceforge.net>
                                09 Jul 2003
```

Magic Name	Number	Structure	File
PG_MAGIC	'P'	pg_{read,write}_hdr	include/linux/pg.h
CMAGIC	0x0111	user	include/linux/a.out.h
MKISS_DRIVER_MAGIC	0x04bf	mkiss_channel	drivers/net/mkiss.h
HDLC_MAGIC	0x239e	n_hdlc	drivers/char/n_hdlc.c
APM_BIOS_MAGIC	0x4101	apm_user	arch/x86/kernel/apm_32.c
CYCLADES_MAGIC	0x4359	cyclades_port	include/linux/cyclades.h
DB_MAGIC	0x4442	fc_info	drivers/net/iph5526_novram.
DL_MAGIC	0x444d	fc_info	drivers/net/iph5526_novram.
FASYNC_MAGIC	0x4601	fasync_struct	include/linux/fs.h
FF_MAGIC	0x4646	fc_info	drivers/net/iph5526_novram.
ISICOM_MAGIC	0x4d54	isi_port	include/linux/isicom.h
PTY_MAGIC	0x5001		drivers/char/pty.c
PPP_MAGIC	0x5002	ppp	include/linux/if_pppvar.h
SERIAL_MAGIC	0x5301	async_struct	include/linux/serial.h
SSTATE_MAGIC	0x5302	serial_state	include/linux/serial.h
SLIP_MAGIC	0x5302	slip	drivers/net/slip.h
STRIP_MAGIC	0x5303	strip	drivers/net/strip.c
X25_ASY_MAGIC	0x5303	x25_asy	drivers/net/x25_asy.h
SIXPACK_MAGIC	0x5304	sixpack	drivers/net/hamradio/6pack.
AX25_MAGIC	0x5316	ax_disp	drivers/net/mkiss.h
TTY_MAGIC	0x5401	tty_struct	include/linux/tty.h
MGSL_MAGIC	0x5401	mgsl_info	drivers/char/synclink.c
TTY_DRIVER_MAGIC	0x5402	tty_driver	include/linux/tty_driver.h
MGSLPC_MAGIC	0x5402	mgslpc_info	drivers/char/pcmcia/synclin
TTY_LDISC_MAGIC	0x5403	tty_ldisc	include/linux/tty_ldisc.h
USB_SERIAL_MAGIC	0x6702	usb_serial	drivers/usb/serial/usb-seri
FULL_DUPLEX_MAGIC	0x6969		drivers/net/ethernet/dec/tu
USB_BLUETOOTH_MAGIC	0x6d02	usb_bluetooth	drivers/usb/class/bluetty.
RFCOMM_TTY_MAGIC	0x6d02		net/bluetooth/rfcomm/tty.c
USB_SERIAL_PORT_MAGIC	0x7301	usb_serial_port	drivers/usb/serial/usb-seri
CG_MAGIC	0x00090255	ufs_cylinder_group	include/linux/ufs_fs.h
RPORT_MAGIC	0x00525001	r_port	drivers/char/rocket_int.h
LSEMAGIC	0x05091998	lse	drivers/fc4/fc.c
GDTIOCTL_MAGIC	0x06030f07	gdth_iowr_str	drivers/scsi/gdth_ioctl.h
RIEBL_MAGIC	0x09051990		drivers/net/atarilance.c
NBD_REQUEST_MAGIC	0x12560953	nbd_request	include/linux/nbd.h
RED_MAGIC2	0x170fc2a5	(any)	mm/slab.c
BAYCOM_MAGIC	0x19730510	baycom_state	drivers/net/baycom_epp.c
ISDN_X25IFACE_MAGIC	0x1e75a2b9	isdn_x25iface_proto_data	drivers/isdn/isdn_x25iface.
ECP_MAGIC	0x21504345	cdkecpsig	include/linux/cdk.h
LSOMAGIC	0x27091997	lso	drivers/fc4/fc.c
LSMAGIC	0x2a3b4d2a	ls	drivers/fc4/fc.c
WANPIPE_MAGIC	0x414C4453	sdla_{dump,exec}	include/linux/wanpipe.h
CS_CARD_MAGIC	0x43525553	cs_card	sound/oss/cs46xx.c
LABELCL_MAGIC	0x4857434c	labelcl_info_s	include/asm/ia64/sn/labelcl
ISDN_ASYNC_MAGIC	0x49344C01	modem_info	include/linux/isdn.h
CTC_ASYNC_MAGIC	0x49344C01	ctc_tty_info	drivers/s390/net/ctctty.c

Continu

Table 16.1 – continued from previous page

Magic Name	Number	Structure	File
ISDN_NET_MAGIC	0x49344C02	isdn_net_local_s	drivers/isdn/i4l/isdn_net_l
SAVEKMSG_MAGIC2	0x4B4D5347	savekmsg	arch/*/amiga/config.c
CS_STATE_MAGIC	0x4c4f4749	cs_state	sound/oss/cs46xx.c
SLAB_C_MAGIC	0x4f17a36d	kmem_cache	mm/slab.c
COW_MAGIC	0x4f4f4f4d	cow_header_v1	arch/um/drivers/ubd_user.c
I810_CARD_MAGIC	0x5072696E	i810_card	sound/oss/i810_audio.c
TRIDENT_CARD_MAGIC	0x5072696E	trident_card	sound/oss/trident.c
ROUTER_MAGIC	0x524d4157	wan_device	[in wanrouter.h pre 3.9]
SAVEKMSG_MAGIC1	0x53415645	savekmsg	arch/*/amiga/config.c
GDA_MAGIC	0x58464552	gda	arch/mips/include/asm/sn/gd
RED_MAGIC1	0x5a2cf071	(any)	mm/slab.c
EEPROM_MAGIC_VALUE	0x5ab478d2	lanai_dev	drivers/atm/lanai.c
HDLCDRV_MAGIC	0x5ac6e778	hdlcdrv_state	include/linux/hdlcdrv.h
PCXX_MAGIC	0x5c6df104	channel	drivers/char/pcxx.h
KV_MAGIC	0x5f4b565f	kernel_vars_s	arch/mips/include/asm/sn/kl
I810_STATE_MAGIC	0x63657373	i810_state	sound/oss/i810_audio.c
TRIDENT_STATE_MAGIC	0x63657373	trient_state	sound/oss/trident.c
M3_CARD_MAGIC	0x646e6f50	m3_card	sound/oss/maestro3.c
FW_HEADER_MAGIC	0x65726F66	fw_header	drivers/atm/fore200e.h
SLOT_MAGIC	0x67267321	slot	drivers/hotplug/cpqphp.h
SLOT_MAGIC	0x67267322	slot	drivers/hotplug/acpiphp.h
LO_MAGIC	0x68797548	nbd_device	include/linux/nbd.h
OPROFILE_MAGIC	0x6f70726f	super_block	drivers/oprofile/oprofilefs
M3_STATE_MAGIC	0x734d724d	m3_state	sound/oss/maestro3.c
VMALLOC_MAGIC	0x87654320	snd_alloc_track	sound/core/memory.c
KMALLOC_MAGIC	0x87654321	snd_alloc_track	sound/core/memory.c
PWC_MAGIC	0x89DC10AB	pwc_device	drivers/usb/media/pwc.h
NBD_REPLY_MAGIC	0x96744668	nbd_reply	include/linux/nbd.h
ENI155_MAGIC	0xa54b872d	midway_eprom	drivers/atm/eni.h
CODA_MAGIC	0xC0DAC0DA	coda_file_info	fs/coda/coda_fs_i.h
DPMEM_MAGIC	0xc0ffee11	gdt_pci_sram	drivers/scsi/gdth.h
YAM_MAGIC	0xF10A7654	yam_port	drivers/net/hamradio/yam.c
CCB_MAGIC	0xf2691ad2	ccb	drivers/scsi/ncr53c8xx.c
QUEUE_MAGIC_FREE	0xf7e1c9a3	queue_entry	drivers/scsi/arm/queue.c
QUEUE_MAGIC_USED	0xf7e1cc33	queue_entry	drivers/scsi/arm/queue.c
HTB_CMAGIC	0xFEFAFEF1	htb_class	net/sched/sch_htb.c
NMI_MAGIC	0x48414d4d455201	nmi_s	arch/mips/include/asm/sn/nm

Note that there are also defined special per-driver magic numbers in sound memory management. See `include/sound/sndmagic.h` for complete list of them. Many OSS sound drivers have their magic numbers constructed from the soundcard PCI ID - these are not listed here as well.

IrDA subsystem also uses large number of own magic numbers, see `include/net/irda/irda.h` for a complete list of them.

HFS is another larger user of magic numbers - you can find them in `fs/hfs/hfs.h`.

CHAPTER

SEVENTEEN

WHY THE “VOLATILE” TYPE CLASS SHOULD NOT BE USED

C programmers have often taken volatile to mean that the variable could be changed outside of the current thread of execution; as a result, they are sometimes tempted to use it in kernel code when shared data structures are being used. In other words, they have been known to treat volatile types as a sort of easy atomic variable, which they are not. The use of volatile in kernel code is almost never correct; this document describes why.

The key point to understand with regard to volatile is that its purpose is to suppress optimization, which is almost never what one really wants to do. In the kernel, one must protect shared data structures against unwanted concurrent access, which is very much a different task. The process of protecting against unwanted concurrency will also avoid almost all optimization-related problems in a more efficient way.

Like volatile, the kernel primitives which make concurrent access to data safe (spinlocks, mutexes, memory barriers, etc.) are designed to prevent unwanted optimization. If they are being used properly, there will be no need to use volatile as well. If volatile is still necessary, there is almost certainly a bug in the code somewhere. In properly-written kernel code, volatile can only serve to slow things down.

Consider a typical block of kernel code:

```
spin_lock(&the_lock);
do_something_on(&shared_data);
do_something_else_with(&shared_data);
spin_unlock(&the_lock);
```

If all the code follows the locking rules, the value of shared_data cannot change unexpectedly while the_lock is held. Any other code which might want to play with that data will be waiting on the lock. The spinlock primitives act as memory barriers - they are explicitly written to do so - meaning that data accesses will not be optimized across them. So the compiler might think it knows what will be in shared_data, but the spin_lock() call, since it acts as a memory barrier, will force it to forget anything it knows. There will be no optimization problems with accesses to that data.

If shared_data were declared volatile, the locking would still be necessary. But the compiler would also be prevented from optimizing access to shared_data _within_ the critical section, when we know that nobody else can be working with it. While the lock is held, shared_data is not volatile. When dealing with shared data, proper locking makes volatile unnecessary - and potentially harmful.

The volatile storage class was originally meant for memory-mapped I/O registers. Within the kernel, register accesses, too, should be protected by locks, but one also does not want the compiler “optimizing” register accesses within a critical section. But, within the kernel, I/O memory accesses are always done through accessor functions; accessing I/O memory directly through pointers is frowned upon and does not work on all architectures. Those accessors are written to prevent unwanted optimization, so, once again, volatile is unnecessary.

Another situation where one might be tempted to use volatile is when the processor is busy-waiting on the value of a variable. The right way to perform a busy wait is:

```
while (my_variable != what_i_want)
    cpu_relax();
```

The cpu_relax() call can lower CPU power consumption or yield to a hyperthreaded twin processor; it also happens to serve as a compiler barrier, so, once again, volatile is unnecessary. Of course, busy- waiting is generally an anti-social act to begin with.

There are still a few rare situations where volatile makes sense in the kernel:

- The above-mentioned accessor functions might use volatile on architectures where direct I/O memory access does work. Essentially, each accessor call becomes a little critical section on its own and ensures that the access happens as expected by the programmer.
- Inline assembly code which changes memory, but which has no other visible side effects, risks being deleted by GCC. Adding the volatile keyword to asm statements will prevent this removal.
- The jiffies variable is special in that it can have a different value every time it is referenced, but it can be read without any special locking. So jiffies can be volatile, but the addition of other variables of this type is strongly frowned upon. Jiffies is considered to be a "stupid legacy" issue (Linus's words) in this regard; fixing it would be more trouble than it is worth.
- Pointers to data structures in coherent memory which might be modified by I/O devices can, sometimes, legitimately be volatile. A ring buffer used by a network adapter, where that adapter changes pointers to indicate which descriptors have been processed, is an example of this type of situation.

For most code, none of the above justifications for volatile apply. As a result, the use of volatile is likely to be seen as a bug and will bring additional scrutiny to the code. Developers who are tempted to use volatile should take a step back and think about what they are truly trying to accomplish.

Patches to remove volatile variables are generally welcome - as long as they come with a justification which shows that the concurrency issues have been properly thought through.

* References

[1] http://lwn.net/Articles/233481/

[2] http://lwn.net/Articles/233482/

* Credits

Original impetus and research by Randy Dunlap

Written by Jonathan Corbet

Improvements via comments from Satyam Sharma, Johannes Stezenbach, Jesper Juhl, Heikki Orsila, H. Peter Anvin, Philipp Hahn, and Stefan Richter.