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.

:ref:`development_process` 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.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]1.Intro.rst`, line 17); *backlink*

Unknown interpreted text role "ref".

:ref:`development_early_stage` covers early-stage project planning, with an emphasis on involving the development community as soon as possible.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]1.Intro.rst`, line 24); *backlink*

Unknown interpreted text role "ref".

:ref:`development_coding` 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.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]1.Intro.rst`, line 27); *backlink*

Unknown interpreted text role "ref".

:ref:`development_posting` 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.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]1.Intro.rst`, line 32); *backlink*

Unknown interpreted text role "ref".

:ref:`development_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.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]1.Intro.rst`, line 38); *backlink*

Unknown interpreted text role "ref".

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]1.Intro.rst`, line 44); *backlink*

Unknown interpreted text role "ref".

:ref:`development_conclusion` concludes the document with pointers to sources for more information on kernel development.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]1.Intro.rst`, **line 47);** ***backlink***

Unknown interpreted text role "ref".

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 VoÃŸ.

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.