

# Joining the herd of cats

## How to work with the kernel development process

Jonathan Corbet  
LWN.net  
[corbet@lwn.net](mailto:corbet@lwn.net)

### I: Introduction

Your author has been watching the kernel development process closely for many years. Over that time, he has submitted numerous patches, participated in every kernel summit, and gotten to know many of the people involved. Slowly, a realization has dawned: while the kernel process seems relatively straightforward to most of the people who are deeply involved with it, many others have an exceedingly difficult time engaging with it. As a result of this impedance mismatch, a number of capable and well-meaning developers have quit in frustration, unable to understand why their attempts to make the kernel better have failed. Others find the process sufficiently intimidating that they do not even attempt to work with it, choosing to keep their code outside of the mainline kernel.

We cannot afford either of those outcomes. We need the best developers we can get, and we need to encourage them to merge their code into the mainline kernel. We may be stronger than we have ever been, but there is still much to be done, and we will not continue to thrive if we alienate those who are trying to work with us. This work is my attempt to bridge the gap between the kernel development community and those who would participate in it.

Many aspiring kernel developers come initially from the proprietary world, but they can fail to realize just how different the free software development environment is. Proprietary software tends to be product driven, while free software is much more about the ongoing process. In the proprietary world, requirements are imposed from the top; with free software, they come up from the bottom. The time horizon in the free software community is much longer: we know that we'll still be maintaining that code ten years from now. Quality assurance is an internal function in the proprietary environment, but it's done externally in the free software community. Proprietary software decisions are made hierarchically, in private, in an environment of complete control; free software decisions are consensus-based, public, and nobody has any great level of control.

The kernel is even more different than most free software projects. It is a large process involving literally thousands of developers, with nobody dominating the contributions to the mainline. The community is global and highly commercial. It is also growing quickly. Traditionally the kernel has been the wild west of almost any operating system process – it's a place where many of the normal rules do not apply. That is becoming less true over time with the Linux kernel, however.

## II: Process issues

Vast amounts of kernel developer pain comes from a simple lack of understanding of the process by which code gets into the mainline kernel. The life cycle of a kernel patch is complicated, and can take years to reach its conclusion. From its beginning as a gleam in some developer's eye, into early discussions and code postings, through its arrival into subsystem trees and the -mm tree, and into its possible merger into the mainline, a patch goes through a very iterative and public process. Developers from the corporate environment (and their managers!) often see the process a little differently: they would like a mechanism where a patch arrives fully-formed and goes straight into the mainline. But things do not work that way.

One of the biggest mistakes that many developers make is to do their early work privately, without engaging the community. Early communication is a crucial part of the process; it will help developers avoid duplication of effort and merge-blocking mistakes. Developers have worked on code for years which had no chance of getting into the mainline because they never talked to anybody about what they were doing. Early communication can prevent wasting vast amounts of developer time.

Similarly, early release of code is important. There are companies out there which have policies requiring that any code release go through their internal quality assurance process first. But the company's process will be vastly different from the QA that the community will apply to the code; holding code until the internal testers have signed off on it makes the eventual release too late.

The “hold for internal QA” approach also misses a fundamental point: there *will* be changes required before any non-trivial submission gets into the mainline. No initial submission is perfect. If those changes have to go through the whole QA process again, the people involved will suffer a lot of pain.

Failure to understand the development cycle also leads to problems. Each kernel release goes through a 2-3 month cycle, where the first two weeks are called the “merge window.” New features and significant changes can only be added during the merge window; any changes which miss the window have to wait until the next cycle. Attempts to merge changes after the merge window closes will draw an unfriendly response.

## III: Patch submission

Once upon a time, a developer with a patch of interest would send it directly to Linus Torvalds and hope for the best. Things no longer work that way. Patches flow through a hierarchy of subsystem maintainers on their way to the mainline, being reviewed at each step. Most patches also spend some time in Andrew Morton's -mm tree; there is, in fact, resistance to merging patches which have

not found their way into -mm before the merge window opens.

The first step, thus, is to be sure to send patches to the correct maintainer. Just sending it out onto a mailing list may not be enough to get it the attention it needs to get into the system. If you do not know which maintainer you should be working with, the MAINTAINERS file in the kernel source tree should tell you. For some areas of the kernel there is no obvious maintainer – there is no virtual filesystem tree, for example. In such cases, sending patches to Andrew Morton is usually the right thing to do.

It is also important to find the correct mailing list. Ideally a developer should be a member of the relevant list long before getting to the point of posting code; again, the MAINTAINERS file can be a useful guide. Often the developers working in a specific subsystem will not participate on the main linux-kernel list at all; the volume of traffic there is simply too high. So patches posted to linux-kernel will often not be seen by the people who would be most interested.

A common mistake is to post large, multi-purpose patches. Code in this form is difficult to review, and, as a result, is often ignored. A big change must be split into small pieces, each of which does thing which can be reviewed and verified independently. The small pieces will not be the series of commits on the way to the current version of the patch; instead, the patch must be teased apart into logical chunks. It is important that the kernel can be built and run after every step in the process; a patch series which breaks partway through will cause frustration for developers attempting to use the powerful git-bisect tool to find problems.

Basic care should also be taken in the packaging of patches for the mailing lists. A patch should be included inline (not as an attachment), not be word-wrapped, contain a proper description and justification, etc. See the SubmittingPatches file in the Documentation directory; the new checkpatch.pl script should also be used to help avoid simple problems.

Once a patch is submitted, reviews and comments will (with luck) come back. Many developers go wrong in their handling of comments, with the result that they are far less effective in the community than they would otherwise be. Reviewing patches is hard and thankless work, but it is a crucial part of how the process works. Developers should take reviews in the proper spirit: they are an attempt to maintain the quality of the kernel we all use. Among other things, that means:

- Treat reviewers politely; thank them for their time. A developer should not criticize or attack reviewers even if they seem rude.
- Reviews require responses, they cannot be ignored. Most of the time, the reviewer will raise a legitimate point which requires changes to the patch. Should a reviewer make a suggestion which shows a misunderstanding of the patch or which cannot be implemented, a polite explanation should be

provided.

- Criticism should not be taken personally. Developers who review patches are not driven by a dislike of the developer or their employer. Employees whose code is harshly reviewed have not brought shame upon themselves or their companies; they have just experienced a part of the process.

Reviewers will occasionally ask for major changes, sometimes extending beyond the scope of the current patch. For example, reviewers may suggest that certain functionalities should be moved up into a higher-level subsystem where everybody can benefit from them. Satisfying such requests can be an expensive and unwelcome diversion, but, if possible, it should be done. The end result will be a better kernel for everybody involved.

#### **IV: Coding issues**

In the early days of Linux kernel development, few people cared much about coding style; as a result, many different styles of code were merged. That situation has changed, and a patch which ignores the established kernel coding style is unlikely to be merged. `Documentation/CodingStyle` should be consulted for the details of the accepted style.

An occasional mistake is to introduce levels of abstraction which are not required to make the desired functionality work. Hardware abstraction layers, extra function parameters “just in case,” and lots of single-line functions are an example of this sort of problem. Reviewers will usually ask that these abstraction layers be removed. Code which contains preprocessor directives intended to help it compile with multiple versions of the kernel will suffer a similar fate.

With regard to multiple versions: every kernel developer should understand that there is no stable internal kernel API. That is just how development is done in this project. The best way to cope with API instability is to get code into the mainline kernel at the earliest possible date; until that time, it will be necessary to update the code for every kernel release. During this period, the documentation at <http://lwn.net/Articles/2.6-kernel-api> may prove helpful.

Another common mistake is to introduce regressions. A regression is often justified by saying that fixes a different problem, that that reasoning no longer flies with the kernel maintainers. Keeping existing systems working is seen as being more important than fixing systems which have never worked. So patches which create regressions will fail to get into the kernel.

Finally, all code will have bugs, but many kinds of problems can be avoided. There is an increasingly large set of tools which can be used to find problems before they affect users. These include warnings from the compiler, the sparse static analysis tool, the lock checker, the fault injection framework, and more. These tools should be used to check code before it is posted for inclusion into the

kernel.

## **V: Some final notes**

It goes without saying that the submission of any sort of code which cannot be legitimately added to the kernel will not be appreciated. All code must include a Signed-off-by line which indicates that the developer has submitted the code under the terms of the Developers Certificate of Origin (found in the SubmittingPatches file). If a developer cannot certify the code as described there, the code should not be submitted at all.

One of the biggest mistakes is keeping code outside of the mainline and shipping binary-only modules to customers. There is a long list of problems – both technical and legal – associated with this practice, and there is no real benefit to it. The best approach is to show respect for customers, give them the source, and get that source into the mainline.

There are many resources available to developers who wish to work with the kernel process. These include the mailing lists, the introductory information available at [KernelNewbies.org](http://KernelNewbies.org), the Kernel Mentors project, and the kernel programming formation maintained by your author at [LWN.net](http://LWN.net). Making full use of these resources will help to ensure that one's engagement with the process will go well.

The kernel development process is a busy, chaotic, and sometimes intimidating thing to watch. But its rules are relatively straightforward and its goals are clear: the creation of the best kernel we can. By paying attention to the avoidance of common mistakes, any developer should be able to have a satisfying experience working with the kernel. Jump in and have fun!