A day in the life of a Linux kernel hacker...

Why who does what when and how!

John W. Linville
Red Hat, Inc.
linville@redhat.com

Abstract

The Linux kernel is a huge project with contributors spanning the globe. Its usefulness and other advantages continue to draw new users on a daily basis. But some users will discover problems with the code, and others will eventually find a need to add their own features to Linux. Whether you are a user in need of support or a developer trying to enhance the kernel, it is good to know something about who is in the community and how they work together.

This topic will introduce the newcomer to some of the characters in the Linux community and some of the roles they play. It will highlight some of the tasks Linux hackers perform on a day-to-day basis, and give a general overview of how work gets done within the community.

1 Introduction

Have you ever wondered how Linux kernel hackers spend their days? I am sure that many people have a picture in their mind: a pale-faced, shaggy, sandal-clad man basking in the glow of his LCD in a corner of his mother's basement while C code for drivers, schedulers, and memory allocators oozes from his fingertips. That image is, of course, not entirely inaccurate. However, there is much more to the community than that stereotype. Not only is there a wide diversity amongst the participants, there are far more roles to play than merely that of sunlight-deprived developer!

1.1 Why is this interesting?

You might ask, "So what?" Many might be satisfied to use Linux (either directly or indirectly) without knowing any details about how it came to be or how it continues to evolve. But one must realize that the Linux kernel is a huge software project with literally millions of lines of code, thousands of adjunct developers, hundreds of regular developers, and dozens of core developers. The Linux kernel is a study in management of complex projects, and there are many lessons to be drawn from observing how Linux is developed.

Beyond one's own intellectual expansion, there are more practical reasons why one might want to understand how Linux is developed. A direct user of Linux (or any other system) is bound to encounter a problem eventually, and that user will probably want to see that problem fixed. Also, many systems are now developed using Linux as a component. Developers working on such systems will want to understand how the community works, not only to help themselves get problems fixed, but also to understand how to get their own code incorporated into Linux in order to reap the benefits of community maintenance. Finally, one may wish to become an active member of the Linux community. In fact there are many reasons for joining the community. These include "scratching your own itch," making the world a better place, or simply building your own public profile.

1.2 What is so different?

Of course, lots of software is developed behind closed doors in any number of companies around the world. Surely there are any number of people who know how to develop software? That is true, but there are some key differences between those practices and how software is developed in the open source community. Some important differences exist in terms of the role of profit, the hierarchy of authority, and how technical decisions are made.

One of the more obvious points about the open source community is the role of financial profit. Such profit is not necessarily the main motivator for participation in

the community. Many people participate solely as hobbyists, while others have specific needs and participate in order to "scratch their own itch." Still others honestly believe that they are making the world a better place. The presence of such players in the community creates a far different dynamic than one finds in a traditional closed software shop.

Another poignant difference between traditional software development and the open source community is the lack of central authority figures. Even Linus Torvalds himself has no inherent authority beyond his own skill and participation. If Linus were to make irrational decisions or were simply to lose interest in the Linux kernel project, the community would simply reform around one or more other leaders. The need to recruit and retain contributors through the merit of the leadership is another key difference between traditional software development and the open source community.

A final difference worth noting is the notion of meritocracy. In a traditional software development shop, a project of any considerable size will quickly be subdivided into component parts and each part will be assigned to an individual or team. Those teams will typically toil in isolation until they have something working, then they will submit that to the final product with little or no external review. In the open source community, open review is part of the process both during development and before the final merge. In many cases, alternative implementations compete with one another and the community chooses between them based on technical merit and individual needs. Such "wasted" effort would not be tolerated in most traditional software development shops, but the open source community is big enough to afford it and is stronger for it.

1.3 Let's explore!

Hopefully the point has been made that the open source community in general and the Linux community in particular is worth studying. Below is a discussion of what types of people are involved in that community and what roles they play. Also discussed are some of the tools they use and how they spend their time. Finally, some time is spent discussing the actual processes used for development and how they interact with one another. By the end, the reader should have a good idea of who in the community does what, when they do it, why they do so, and how.

2 Why who does what...

People from all over the world become involved in the Linux community for any number of reasons. These people apply their diversity of talents to a number of different roles within the community, many of which are not directly related to writing software. A number of tools facilitate this cooperation. An overview of these topics will guide the reader's understanding of the community.

2.1 Motivations

It is probably impossible to enumerate every possible motivation for becoming a Linux contributor. Still, most motivations fall into a few loosely defined categories. These motivations run the gamut from commercialism to volunteerism, and span from self-interest to altruism.

The most well-known reason and most commonly cited one for getting involved with Linux is "scratching an itch." Nearly everyone needs software for something nowadays, and many people and organizations need software either that is unavailable or that those people and organizations cannot afford to obtain from a vendor. In many such situations, this software is developed and deployed internally by those organizations.

The nature of software is such that once it exists, the cost of duplicating and distributing that software is negligible. This is especially true when that distribution is done electronically and done by others at their own expense. Such distribution also allows those with similar software needs to find and support each other, sharing resources to develop and improve software for the widest possible audience. As the audience widens, so increases the potential benefits of such sharing. Since the kernel is the central component of a Linux-based system, "scratch your own itch" contributors are quite common in the Linux community.

A large number of Linux contributors are working on commercial software development projects that use the Linux kernel. Those employed by distribution vendors such as Red Hat are obvious examples of this. But there are any number of smaller software vendors developing embedded systems or other specialized products and making contributions to the kernel. These contributions are often small bug fixes or specialized device drivers,

but can be more generic components such as filesystems, compression algorithms, networking subsystems, etc. Just like "scratch your own itch" contributors, commercial developers recognize the value of community software development and maintenance of kernel components.

A few other motivations are commonly found within the community. Some number of community contributors make their living doing contract development work. This typically involves short-term work on behalf of product-focused companies that need specific features developed for the Linux kernel. Some other contributors are fortunate enough to be sponsored to do Linux kernel work under their own direction due to the good graces of some company or other benefactor. Finally, some number of contributors work on Linux because they believe they are making the world a better place or for some other similarly altruistic reason.[1]

2.2 Roles

In a community-based software development project, it is important to recognize that not everyone is writing code for the project. While this is certainly an important and necessary skill, it is generally insufficient for a successful project. People are needed to test the software and to report bugs, to write and review the code, to manage the code, and to document and write about the code. Each of these roles plays an important part within the community.

2.2.1 Bug Reporter

One of the most important roles in the community is the bug reporter. While many people will run new kernels and many of those will experience one problem or another, few will bother to report those problems and fewer still will not only provide useful information but also remain engaged long enough to find fixes. Those individuals are invaluable in the creation and maintenance of high quality software.

[1] In any case, the author suspects that the sum of these groups is dwarfed by the number of commercial and "scratch your own itch" contributors.

2.2.2 Tester

In a sense, every user is also a tester. But in reality most users exercise little more than the core functionality of a given piece of software. True testing requires repetition, documentation, and skill as well as the dedication to apply those resources. Testers not only find problems but also assist in analysis by finding the boundaries of the problems they identify. Testers are highly valued members of the community.

2.2.3 Coder

The coder is the most celebrated member of the community. The coder tackles the problem of producing source code changes to fix a problem or implement a new feature. Coders provide the raw material for the Linux kernel. Obviously, without coders the project would not exist.

2.2.4 Reviewer

An often overlooked person in the community is the reviewer. When the coder does his job, he posts his product (i.e., a patch) to a mailing list. The reviewer evaluates the change, comments upon its form and impact, and often makes suggestions for changes or refinements. The reviewer has one of the most important roles in ensuring initial code quality.

2.2.5 Maintainer

The maintainers are the "old men"[2] in the community. The maintainers are responsible for taking what the coders produce, deciding when the reviewers have added enough value, and merging the results into the upstream kernel tree. They also communicate with testers, bug reporters, and others to ensure that code quality is maintained at a good level and that good processes are being followed. Maintainers perform a role similar to a manager or team leader in a traditional software development shop.

[2] Of course, they are not all old and not all men...

2.2.6 Technical Writer

A technical writer is one who produces documentation of technical details surrounding the kernel. This primarily includes documenting both those APIs for internal use within the kernel and those for communicating with userland programs. Without such documentation, it would be difficult to sustain the development community.

2.2.7 Journalist

An important aspect of maintaining a community is keeping contributors aware of what else is happening within that community. The Linux kernel is a huge project, and it can be difficult to know what new developments are in progress and what old components are being revamped or removed. Following all of the relevant mailing lists is a daunting task by itself, much less if one is trying to write code or run tests. The journalists in the community keep everyone abreast of what is happening now and what is coming next.

2.3 Tools

It should not be surprising that a software development community uses a number of software tools to keep itself running smoothly. Still, it is worthwhile to enumerate some of them and discuss how they are used. Important tools include both those for communications and those specific to software development.

2.3.1 Email

Email is the single most important tool within the community. The Linux kernel community is diverse and spread across the globe. It is generally difficult to assemble people in one place or even to gather for a conveniently timed teleconference. Consequently discussions are held on mailing lists. This has the added benefits of documenting and archiving such discussions as well as generally keeping such discussions short and direct.

2.3.2 Bugzilla, etc.

Bugzilla and other bug-tracking tools are often used for their intended purpose. The kernel has its own Bugzilla instance,[3] but the bug trackers of distributions and other kernel-related projects are often used as well. Such tools help to organize bug report information and to segregate one bug's information from reports of other bugs, as well as from other discussions that would otherwise clutter a mailing list.

2.3.3 IRC

IRC is commonly used by active kernel contributors for real-time communications. Chat bridges the gap between email and telephony, allowing precise communications in a timely and direct fashion.

2.3.4 Wikis

A wiki is the documentary analog of open source development. As such, it fits nicely with the general mindset of the Linux kernel development community. Many parts of the project use wikis to document designs, user interfaces, API changes, and other information pertinent to users and/or other developers.

2.3.5 Code Analyzers

Policing large bodies of code can be daunting, and line-by-line analysis of code for trivial or subtle coding errors can be tedious and error prone. Fortunately, many such problems can be identified algorithmically. The kernel includes `checkpatch.pl` which can be used to find many simple problems, especially those relating to coding style. Tools such as Sparse[4] are often used for deeper code analysis.

2.3.6 Git

No discussion of tools used in the Linux kernel community would be complete without mentioning `git`, the primary revision control tool used within the community. Unlike traditional revision control tools, `git` uses

[3] `http://bugzilla.kernel.org`

[4] `http://www.kernel.org/pub/software/devel/sparse/`

a distributed development model. Among other ramifications, this means that every copy of the `git` repository can operate independently. Further, formerly independent repositories can be merged at any time, allowing for development efforts to proceed according to their own schedules without requiring lots of work to resynchronize with the upstream kernel tree. `Git` is probably the single most important tool in use by the community today.

2.4 Patches

An important point must be made regarding patches. A patch is a unit of change to source code.[5] Typically patches are sent in email for review and then later applied to a tree in `git` to form the basis of future development. Patches contain limited context used to identify affected pieces of source code files. This helps to make them resilient against unrelated changes in the same files.

The great thing about a patch is that it focuses attention just on those pieces of code that are being changed. This allows for direct review of a proposed change without lots of effort in locating or identifying that change. The alternative[6] is to pass around changed versions of complete files. These files are awkward to handle and their use makes identifying changes difficult and error prone. The use of the simple patch is an elegant enabling technology for distributed development on a large scale.

3 When and How...

Now that we have identified the motivations of the players, the roles they play, and the tools they use, we should look at the processes used to coordinate their efforts. We will discuss how development needs are identified, what process is used to vet patches, and the route patches follow on their way to the "official" Linux kernel.

[5] While the word "patch" can connote something shoddy or temporary to a native English speaker, in the context of Linux kernel development it has no such connotations. Instead, the term patch derives from the name of the non-interactive editor used to apply the changes to the source code.

[6] The author's experience suggests that this code review alternative is used all too often in the traditional software development world.

3.1 Identify a Need

Perhaps it goes without saying, but the first step in any development process is to identify a need. In many cases, the need will originate with a bug report. This might be in Bugzilla or another bug tracker, or it might come via email or IRC. In other cases, the need will derive from an external project requirement such as the need for a driver for a new device or a new networking subsystem for a certain application. In other cases the "need" is because some other operating system has a feature that is deemed desirable for Linux. Finally, in many cases the development need originates with someone saying "wouldn't it be cool if...?" In any case, once a need is identified, the next step is to write some code and post a patch.

3.2 Development Cycle

Many people find the Linux development process to be daunting. In reality it is quite simple, although personalities can make the process a bit humbling. The basic process begins with creating and posting a patch to an appropriate mailing list. With any luck this provokes someone to review the patch and make appropriate comments. Or, reviewers may simply indicate their approval with an "ACK."[7] In many cases it will be necessary to make revisions to the patch and post a new version to the same mailing list. This process should be repeated until the patch is accepted by the maintainer.

Often new contributors are either intimidated by the above process or they simply do not believe it to be the best use of their time. The temptation is to develop in solitude until the developer is completely confident in the soundness of a patch. Do not make this mistake! Inevitably someone will find legitimate problems with any significant patch series. Trying to avoid the review-revise-repost cycle will only waste a developer's time and create frustration between the developer and the community when the patch is finally posted for review.

3.3 Path Through the Trees

The first stop for an accepted patch is in a maintainer's tree. A variety of maintainers' trees exist for subsystems

[7] *ACK* is short for *acknowledged*.

like networking and SCSI, features like SELinux and realtime, and architectures like ARM, MIPS, SPARC, etc. Maintainers' trees are usually limited to usage by interested parties such as developers and users with specific needs or interests.

To expand test coverage and community exposure, other trees aggregate input from the various maintainers' trees. In the past this role was primarily filled by Andrew Morton's `-mm` tree, but more recently the `linux-next` tree has become more popular. The `linux-next` tree pulls the current versions of many (probably most) maintainers' trees to create a preview of what will soon be available in the "official" Linux kernel.

Periodically Linus will pronounce a kernel ready for release. Prior to that time, maintainers will have been staging changes for the next Linux release and making them available through the `linux-next` tree. After the release, Linus spends two weeks merging the patches the maintainers have been staging. Between the end of that period and the next release, only necessary bug fixes are merged into the "official" tree by Linus. Any changes that are not necessary bug fixes are again staged by the maintainers for the following release. This period lasts several weeks as the kernel is exposed to more testing and as bugs are uncovered and fixed. After 2–3 months, Linus will pronounce the kernel ready for another release, and then the cycle begins again.

4 Conclusion

The reader has been provided with an overview of how the Linux kernel is developed. We have discussed why the contributors are involved, and what jobs they perform. We discussed many of the tools the community uses to manage itself, and specifically discussed the importance of the patch as a unit of work.

With all that background information, we went on to discuss the development cycle for the kernel. We touched on how development needs are identified, and discussed how patches are proposed, reviewed, and accepted into the kernel. Finally we discussed the various trees a patch has to traverse before making its way to Linus.

The author hopes this information has been useful. The community needs to sustain itself with contributors. A variety of roles need to be filled—no special training is required. The author hopes that the reader will be inspired to find a way to join us! A well informed and active community continues to be the force behind the continued success of Linux and the open source community in general.

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*