The Economic Benefits of making Open Source Contributions to the Linux Kernel

James Bottomley

SteelEye Technology

6 April 2006

Introduction

- The Linux Kernel is becoming an increasingly complex place
  - The number of "core subsystem" maintainers is growing
  - The number of supported features is growing
  - The rate of change of code is also (currently) growing
- Often difficult to understand what you're changing.
- Even more difficult to work out what the correct way to change it is.
- However, the kernel has a basic need for talented and motivated contributors

Agenda

1. Why you should contribute code to the Kernel (and why your Employer should pay you to do it).
2. Why the Kernel needs you to contribute.
3. Why it isn't as simple as it sounds
4. Case Study: how SteelEye got replication for Disaster Recovery into 2.6.14

Why Contribute

- Direct contributions:
  - There's a bug and it's affecting you personally
  - There's a bug and its affecting your employer
  - You (or your employer) has a new feature/driver
- Indirect contributions
  - You have an area in the kernel that you want to work on.
  - You want your employer to sponsor your work on it.

Alternatives (and misconceptions)

- My Product only supports RedHat, SUSE etc. Linux Distributions, so I only need to patch their distribution.
- Distributions are commercially motivated so they're much easier to deal with than Linux Kernel Developers.
- The Distributions are a direct channel to the users, so they're the obvious place to start.
- I can just patch the kernel and ship it myself.

Upstream First Policy

- Major distributions have agreed not to incorporate features or drivers unless they are on "upstream track" for the vanilla Linux Kernel
  - Obviously there's some flexibility in interpretation of this for their best customers
- Primary reason is that it keeps the distribution kernel code and the vanilla kernel code as close as possible, so
  - Maintenance is reduced: the distro can file a bug with the upstream maintainer if there's a problem.
  - Testing is enhanced: users of all distributions are testing the same code
  - Code Review burden is greatly reduced: Can rely on upstream maintainers to review and accept.

What is "Upstream Track"?

- In the vanilla Kernel (Linus Tree)
- In Andrew Morton's -mm tree
  - With the proviso that Andrew has accepted it for onward transmission to Linus.
  - Not everything in -mm is designated for onward transmission.
- In a Subsystem Maintainer Tree.
  - Again, it must be designated for onward transmission.
  - Policy on this varies from subsystem to subsystem
- Interpretation within gift of Distribution

The Bottom Line

- You must either
  - Get your code accepted into the Vanilla Kernel
    - Either directly to Linus (very hard nowadays)
    - Or via Andrew Morton or one of the Subsystem Maintainers.
  - Or, distribute it yourself
    - Will expand more in case study.
    - Summary: If you need a new kernel, don't bother; If you can just ship a module, may be feasible.

Why the Kernel Needs you to Contribute

- The Linux Kernel Code base is incredibly complex.
- No-one understands it all fully
- It maintains its forward momentum and "buzz" because of innovative advances contributed by individuals.
- The more experts the kernel has contributing and assessing the contributions of others, the better it becomes.
- Maintaining the flow of innovation requires a constant stream of fresh talent.

Contributing To The Kernel

- Know where to start
  - Look in the `MAINTAINERS` file
  - Find your driver, or subsystem and see if it has a mailing list.
  - if it doesn't, you have to begin on the Linux kernel mailing list
    * `linux-kernel@vger.kernel.org`
    * very high volume
    * Slightly lower signal to noise ratio.
- Begin by reading the mailing list **not** by coding.
  - Get a sense of where the code is going and what might be acceptable.
  - Read previous acceptances and rejections.

Your First Contribution

- First, make sure you've lurked on the email list for a while to get the feel of the subsystem and the patches.
- Then, your initial patch should be small, just to get the feel of the process
  - Find a tiny bug or misfeature and fix it.
  - Will give others confidence in trusting you.
  - Will get you used to the patch submission process
- If all goes well, and you think you understand how the subsystem is working, then you can begin your big driver/feature.

Rules for Coding your Feature/Driver

- Release Early, release often
  - Your first patch, doesn't even need to be a patch, just a "this is how I'm thinking of coding this" email.
  - Makes sure you're going in the right direction
  - Gets feedback (and buy in) from others in the development
  - Allows any corrections to be made easily (before you've coded another 10,000 lines of code dependent on the piece that the maintainer wants changed)

Accepting Feedback

- Pay attention to feedback on your code
  - Even if you know your own driver/feature, others probably know the kernel better.
  - Even in your own code, another pair of eyes may spot a bug you missed.
- Some feedback is more valuable that others
  - Every mailing list has its share of armchair coders.
  - If you studied the list first, you should have a pretty good idea who they are.
  - Can also tell by what type of reply from others the feedback elicits.

Why Contributions Usually Fail

- One of the most classic is Coding Style
  - Read the kernel coding style document `Documentation/CodingStyle` and follow it.
  - Not conforming really does matter, because it makes your contribution harder to follow and more difficult to maintain.
  - This really, **really** does matter, so people will be anal about it.
  - Redoing the style is fairly easy and, hey, if that's all they complain about, they must have liked the code ...

Design Issues

- Code that fails for a basic design reason is the hardest to correct
  - Usually requires a fairly thorough rewrite
- Design problems can be picked up early on, so releasing early can avoid this.
- Just because you wrote a driver this way on 15 other platforms doesn't mean that Linux will automatically accept it.
- Design issues are hard to foresee and are usually within the gift of the Maintainer to adjudicate.

Glue Layers

- A “Glue Layer” is a layer that sits between your driver/feature and the Linux Kernel.
- Usually, the reason for it existing is so that the driver/feature can be common across several platforms.
- **Don’t do it!**
- Glue layers may be nice for you to maintain, but they’re a nightmare for anyone else after you move on to different projects.

Case Study: SteelEye Data Replication

- Had some experience of Linux Kernel work
  - Mostly in fixes to Linux 2.2. for Shared SCSI
- in 2001 Decided we needed Replication in Linux (already had it as proprietary kernel extensions for Windows and MP-RAS (SVR4MP).
- Immediate temptation was just to do another binary driver for Linux.
- However, with a bit of persuasion, we decided to try open source development methods.

The Persuasion

- All prior projects (Windows, UNIX) took over two person years to develop and bring to market kernel based replication.
- Linux, being sufficiently different would require a completely new (as in write from scratch) driver.
- We had two resources to assign to the project, one full time and one half time, so that would give us an end date about sixteen months.
- However, using pre-existing open source components (`md` and `nbd`) we produced a project plan predicting GA in 8 months (i.e. half the time)

The Concerns

- Won't the GPL contaminate our entire Product?
  - This was simple: as long as we open source all our kernel components, we're clear to keep our own user level components that make use of the kernel proprietary.
- We're a product company, how can we make money from something we'll give away for free?
  - Solved by separating the problem as above.
  - Key is that the proprietary user components contain sufficient value to protect our investment.

The Value Proposition

- For an engineering shop, engineer time is our most precious asset.
- Using Open Source components saved us 50% in terms of engineer time
- In cash terms, this probably equals about $150k in engineer costs plus increased Opportunity costs
  - Opportunity is the amount of money the product made in the additional 8 months it had on the market
  - Conservatively this is estimated at another $150k

Other Lessons Learned

- Fixing `md` and `nbd` was daunting.
- Delivering the fixes in a timely fashion was painful
  - The distributions and even the kernel cycle is too long for our release
  - Thus, had to ship our own modules
  - `md` is non-modular in every distribution, so changes to it have to be delivered by complete kernel replacement.
  - Fortunately, most of the bugs were in `nbd`

Replacing Distribution Kernels

- Have to generate the kernel in its entirety and package it
- Kernel is often the most complex and difficult distribution package to build.
- It is also the fastest turning one ... almost every update includes a new kernel.
- Distributions often forbid kernel replacement (it voids the support agreement).
- In general, kernel replacement has too many drawbacks to be viable in the marketplace.

Delivering Individual Modules

- Much easier ... doesn't void the support agreement (if you're careful).
- SteelEye does this with `nbd.o` and `nfsd.o` for 2.4
- Even this is hard. Currently have 150 separate kernel build directories on our build machines for all the distributions we support.
- Even for modules, this is rapidly becoming untenable
- Great incentive to get all our patches upstream for 2.6

Working With Distributions

- As Linux becomes more mainstream, this becomes harder
  - As the revenues grow and the money stakes become higher, distributions become less likely to listen to smaller companies.
- Upstream first policy means the patches must be upstream anyway before a distribution will pick them up.
- Therefore, simplest just to work in upstream knowing that distributions will be forced to incorporate them (eventually).

Conclusions

- Submitting patches is different from any other industrial process you'll have been through before
- The trick is to understand the constituency you're trying to convince to accept your patches.
  - i.e. study the mailing list
- Release early and release often.
- Leveraging existing open source components can dramatically shorten project cycles and time to market
  - Providing you're willing to open source your feature.
- Working in the Vanilla kernel is the simplest method for distribution of your feature.