# Why Open Source Contributions can be an Asset to your Comany and How to make them Effectively

James Bottomley

SteelEye Technology

OSDL Kernel Sessions

17 August 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 how to persuade your boss to 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 an Application Vendor like SteelEye became home to two kernel maintainers

# 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.
  - You (or your employer) want to influence the direction something's moving in

# Influence, Franchise and Control

- No one has control over the kernel.
  - Not even Linus or the subsystem Maintainers
- Everyone can have influence
- Influence is via franchise
- Franchise goes to those who contribute code
  - That means that a company's franchise depends on individual people

# 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: Linux Storage

- Will cover two separate issues in this case study
  1. How did SteelEye get into Kernel Coding (as an application VAR)
     - and, more importantly, why do we continue to subsidise maintenance of the SCSI subsystem.
  2. What does the Linux Storage Roadmap actually look like
     - In so far as a roadmap exists
     - and, obviously, it's in terms of technology not features ...

# About SteelEye

- Founded in 1999
- mission to bring application HA to Linux
- Achieved by buying and porting the NCR LifeKeeper HA Cluster product to Linux.
- Company hired a large swath of NCR engineers for initial staffing
- Most of whom were kernel coders from the NCR UNIX SVR4MP OS called MP-RAS

# LifeKeeper and Linux

- Most porting application based ... not much of a problem
- However, base of LifeKeeper was shared storage clusters; two problems
  1. Shared Parallel SCSI buses didn't work in Linux in 1999-2000
  2. The storage ownership model (SCSI Reservations) LifeKeeper used in both MP-RAS and Solaris didn't exist in Linux
- Lucky accident: being mostly kernel engineers we can figure out what the problem is and how to correct it in both cases.

# Our Solution

- Found a set of Fixes ... easy
  - Elected to modify Linux to implement reservations at user level
    - This, accidentally, nicely aligns with the current kernel philosophy of moving policy to user space
  - Actually modified SCSI mid-layer
  - and `aic7xxx` driver
- Tried to get them into the kernel via Red Hat ... less so
  - But much easier in those days
  - Actually formed working relationships with Red Hat SCSI engineers

# Storage Ownership—Perhaps not so Accidental

- First tackled problem (Shared SCSI buses) taught us the difficulty of modifying kernel
  - Problem: SCSI so vital, so many interested parties, agreement on code changes hard to achieve.
  - This made us conclude that minimum and most generic changes were the ones most likely to be accepted.
    - This principle *still* applies today
- So concluded to comply with this
  - Storage ownership would be mediated at user level
  - with minimal necessary kernel support
  - Although kernel changes were still necessary

# The rest is History

- First changes taken by Red Hat (not vanilla Kernel) for 6.1 (June 2000)
- Next targeted OS were SuSE and TurboLinux.
- Realised that easier to apply changes to vanilla kernel ASAP and wait for all distros to pick them up
  - So, accidentally, we hit on "upstream first" policy
  - pragmatic: ease engineering support burden
- In 2003 became SCSI Maintainer
  - Shared Storage and Ownership model never broke again ...
- in 2006 SteelEye has 3 Linux kernel engineers (two maintainers) on staff.

# 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.