Technical vs Social Engineering

What This means for Japanese Developers

Parallels™

Profit from the Cloud

James Bottomley

CTO, Server Virtualization; SCSI Subsystem, Parisc Kernel Maintainer

8 June 2012

In the Beginning

- In 1992, the kernel began life as a very technical place.
- It had very few features and desperately needed others adding.
- Getting patches in was very easy simply because so much work needed to be done.
- Reviews were mostly done by Linus before he put your patch into his kernel tree.
- Reviews tended to concentrate on the technical substance of the patch rather than feature justification.

The Bottom Line

- Anyone could get a patch into the kernel
- For almost any feature
- The only requirement was that you be able to write the code to implement it.
- Most of this early code wasn't of the highest quality
  - SCSI old error handler and IDE driver full of busy waits
  - Block layer had a single lock to protect all devices
  - TTY layer had a static array for ttys and grew a bit like spaghetti.
- Emphasis on enabling features rather than getting the code perfect

The Problems

- This anything goes style produced a full featured kernel very fast
- But it left a lot of problems in its wake.
- Robustness and Scaling were really bad
- I mean really:
  - The kernel was liable to crash frequently
  - More than one disk worked really slowly
  - If you had an error on your disk or cable, error recovery rarely actually recovered the error.
  - SMP, while functional rarely delivered the performance of more than one of your processors.

About ten years later

- Around the time of the first kernel summit in 2001 fixing the problems was becoming urgent
- Eric Youngdale rewrote the entire SCSI layer to give it a well defined API and a threaded error handler
- Jens Axboe rewrote the block layer to divide the single monolithic io lock into a fast, robust, per-queue locked system that would be able to scale.
- The USB subsystem got rewritten several times
- A programme of fine grained locking was introduced so we could reliably scale beyond a single CPU
- Unfortunately, everyone was too afraid to touch the TTY layer!

Attitudes Change

- It's no longer about code and features
- It's about code quality and feature justification
- It also becomes more about ensuring that new code doesn't disrupt the old code
  - i.e. doesn't cause regressions
- Linus isn't the only one reviewing the code any more
  - The kernel now has ~100 Maintainers
  - Each of whom is supposed to make sure the code going into their subsystem is correct and tested.
- Review rises in importance as a vital function for code cleanliness in the kernel

Fast Forward to Today

- The kernel is incredibly feature rich
- Which makes it very complex
  - And thus, adding to the complexity with a new feature gets looked at very closely.
- A lot of our effort goes into preventing regressions
- We've developed elaborate processes for all of this and a host of static checking tools
- It's no longer just about code, it's about style and process as well.
  - i.e. it's no longer technical, it's also social

To Expand on This

- Open Source isn't just a licence, it's a process
- Actually, it's exactly like ISO9001 but worse
  - Over time we've added lots of little things
    - Signed off by
    - Coding styles
    - Dos and Don'ts for patches
- Most people who are maintainers today grew up evolving this process
  - So we all understand what it is and why we're doing it
- However, it can look daunting to outsiders

So How do you get patches in

- Firstly, this is mostly about features
- Bug Fixes are easy
  - Provided you can describe the bug and its effects
  - Not every bug patch does this …
- Need to Socialise the feature first
  - Build a community of users preferably vocal.
  - But if not users, then a community of interested people
  - Be prepared to argue for the feature, explaining what it is, what you'll use it for and why it is useful.
- Conferences are great venues to meet people outside the mailing list environment and talk about what you're tring to do

Of course it goes without saying that

- You first identify and read the relevant mailing list
- You read all the necessary conventions
  - Documentation/HOWTO
  - Documentation/CodingStyle
  - Documentation/SubmittingPatches
- These are even (thanks to the kernel translation project) available in Japanese.
- Following these to the letter is very important
  - `scripts/checkpatch.pl`
  - Does this automatically for you

The Importance of Coding Style

- Mailing lists can be very hostile places
- There are some elements who believe attacking others demonstrates their own cleverness
- Any CodingStyle violation that is flagged by checkpatch.pl is easy meat for them
  - They don't have to think about anything, just feed the mailing list into checkpatch and flame if the result isn't right
- If you adhere to the rules and run your own patches through checkpatch, you forestall this
  - Means that hopefully the arguments will be about the contents of your patch not its style.

But Remember

- The Perfect is the enemy of the good
- The patch doesn't have to be perfect
- Submit Early and Often … even before you've developed all the code
- It's often easier to have constructive arguments over incomplete code
  - Because everyone sees they can still give input
- Just remember to follow the rules and the coding style.

Arguing on Mailing Lists

- First, be technical, never personal
  - Remember you're the expert on the patch
- Only respond to the technical content (if any) in an email
  - If there's no technical content, don't respond at all
- Lurk on the lists to identify who the important people are and pay attention to them
  - They submit lots of patches that get accepted
  - They provide feedback which is often considered in discussions
  - They come up with sensible, constructive suggestions

Defusing Aggression on Mailing Lists

- Arguments sometimes get very heated
  - Especially on LKML where we have a dedicated community of flamers
- Always keep it technical, never personal
- Knowing and being known to people on mailing lists really helps
  - You're no longer an email address, you're a person they've met
- So going to Conferences or other gatherings just to meet people will really assist you
  - If you don't speak English very well, they'll understand

You Must Be Prepared to Argue

- Know why you need the feature and be prepared to explain it
  - Practice beforehand with friends and colleagues.
  - Give seminars to your local LUG explaining what you want to do.
  - Preferably in English because English is the language of mailing list exchange
- Ideally have a list of other communities it will help
  - It's even better if you contact them ahead of time and get them to chime in
- Stick to being polite and technical, but also firm
  - If you have a problem understanding some comment, say so

Final words about Arguing on Lists

- Make sure you argue with the right people
  - i.e. the people you've previously observed to be influential
  - They may be hard to persuade, but they'll be reasonable
  - Remember they may be arguing simply because they don't understand the patch, so make sure to explain itl
- Don't waste time arguing with the wrong people
  - Even if you finally win, no-one useful will be paying attention.
- Be prepared to accept feedback and update your patch accordingly.
- Many patches go through several iterations before being accepted.

Writing Good Change Logs
(my pet maintainer peeve)

- A Good change log should describe what you're doing and why
- It should not describe the code
  - We can all read C, so, unless the code is badly commented or very obscure, we can simply read it.
- Bad:
  - Insert a spinlock into foo_bar function
- Good:
  - An oops was observed removing the foo device while playing music because multiple threads were altering the same data. Fix by using a spinlock to make the foo_bar function single threaded

Splitting your patch into a series

- The object of a patch series is to make the feature easy to review
- Split the patch into functional areas which can be reviewed independently.
- Think about how you explain your patch: first you talk about X, then Y then Z
  - can you split the patch into an X piece a Y piece and a Z piece to match your explanation
- If you can split your patch into a series that follows how you would explain it, then the patch series will be easier to understand

Repeat: Try this out on your peers first

- If you follow all these rules, it's still best to try it with a narrow audience first
- So explain your patch to the local linux users group or work place seminar
  - You can do this in Japanese too first time around
  - Although you'll need to use English for the lists
- It will help you organise your thoughts and also hear what people don't understand about it
- Because you've already argued for the patch, you'll be more confident on the mailing list
- You'll also understand some of the criticism you'll get back because you've heard it before

General Conclusions

- Follow the Rules
- Identify the important people
  - And the people to ignore
  - Meeting the people in your community is also important for improving communications
- Practice arguing for your patch in a friendly environment
  - Before you try it out on the mailing lists
- Build consensus for your feature on the list
  - Remember to explain what it does and why you need it
  - Modify it to make it more useful to others
- Everyone's still afraid to touch the tty layer

Questions?