On submitting kernel patches

July 2008

Andi Kleen

Ottawa Linux Symposium 2008

Scope

- Describing Linux kernel specific procedures
- Many points apply to other large OS projects
  - Details usually differ
    - Each project has an own culture
  - Smaller projects typically use simpler procedures
    - But basics tend to be similar
- In general not all points apply to all patches
  - For simple bug fixes much of this can be skipped
  - Guideline, not fixed procedure
  - Complex changes should try to avoid shortcuts

Why submit patches to Linux mainline?

- Review (usually) increases code quality
- Free testing by the user base
- Avoids user interface conflicts
  - Can be very painful if not avoided
- Free forward porting service
  - Interfaces won't go away
  - For intrusive changes often required
- Best way to distribute a change
  - Change will be often just available in next release of popular distributions
  - Best case for hardware vendors
  - And even convenient for private features

•General overview

- Write/test code
- Code review
- Code gets fixed as needed
- Maintainer merges code
- Code gets tested
- Gets incorporated into release

Preliminaries

- Coding Style should be correct
- Change should work of course
- Extensively documented elsewhere
  - See resources, paper
- Prepare for some additional work
  - And to do some revisions
  - Use some way to allows patch revision (quilt, git)
- There will be criticism
  - It's not meant personally even if harsh

Getting attention

- Patch is more like publishing a scientific paper
  - than a traditional checkin
  - Exceptions: when you become the maintainer
- There is a shortage of reviewers
  - But without review it is difficult to get something in
- And maintainers are often very busy
  - And sometimes there is no clear maintainer for some area
  - Needs other reviewers
- Linux kernel is an attention economy
- Who can sell their patches best gets the reviewers

Case study: dprobes

- Dynamic instrumentation framework
  - Attach probes in RPN language to kernel/user space
  - Originally ported from OS/2
  - Submitted in 2.4 time frame
- No user community, very little interest
  - Dropped from major distribution due to lack of interest
- Team posted many versions of the patches in 2.4
  - Didn't attract significant reviews
  - Main contentious area: VM interface for user probes
  - Byte code interpreter not popular
  - No clear maintainer to process the code

•Dprobes: lessons

- •You have to sell the feature
  - – Especially if it's new and innovative
  - – Only became popular when others started to hype
  - – Adopt a user base early
- •When parts are problematic split them out
- •Don't wait too long to redesign
- •Don't try to submit all features in the first step

•Dprobes: the solution

•Finally redesigned to kprobes

- No byte code, only kernel probes in C
- Went in relatively quickly due to simplicity

•Quickly used by kernel community with C probes

•Lives on as kprobes/systemtap

- Systemtap as a user friendly script language frontend
- User base now due to independent hyping effort
- But still no user probes

Types of submissions

- Clear bug fix
  - Easiest case: Can be usually added quickly
- Cleanups
  - Timing is important
  - Don't overdo it. Bug fixes are more important!
- Optimizations
  - Depends on how much it helps
  - And for what workload
  - And how intrusive it is

Hardware Drivers

- Most common code in the kernel
- Most important part is code style, basic interfaces
  - Look at existing drivers for guidances
- Must be Linux code
  - Follow standard Linux design patterns
  - Avoid adaption layers
  - Coding Style
- Well established procedures for the standard types
  - Networking, block devices, etc.
  - Sometimes more difficult for more exotic ones
- Difficult areas:
  - Needing special hooks in core code

•New core functionality

•Hooks, hooks, hooks

- “I just want to add this hook to improve the world”
- Each hook has large maintenance overhead
- Breaks coding assumptions, makes it difficult to follow coding flow, requires all hook users to check etc.
- Maintainers usually not sympathetic

•First try to avoid hooks

- If you do them they need very careful design

•One way is to trade cleanups for such controversial changes

- Do some significant work to clean up subsystem or resolve existing problems
- Then as part of that add your hooks in a clean way
- That is how Xen paravirt ops got in

•Splitting submissions into pieces

- Large patches cannot be effectively reviewed
- Split patches into logical chunks
  - File boundaries are not logical chunks
  - Exceptions are for new drivers
- Patches must be bisectable
- Don't mix cleanups/refactoring with functional changes
- Don't post too many patches at a time
  - Space out posting of larger patchkits
  - Post in logical chunks

Case study: perfmon2

- Performance counter interface
- Original simpler in tree version on ia64
- “Second system” version out of tree
  - Years out of tree development
  - User base with feature development
  - Very complicated code
- Very complicated interfaces for all the features
  - Scared reviewers away
- Now new merge attempt with a much simplified version
  - But interface still very complicated

•perfmon2 lessons

- Submit quickly
- Be conservative with novel design patterns
  - Like output plug-ins
- Don't add too many features out of tree
  - Later it's hard to untangle them
  - And rationales will be lost
- Have a basic functionality version

•Interfaces

- Reviewers focus on user space interfaces
  - “Code changes, but interfaces stay forever”
  - Often very difficult discussions
  - Doesn't matter for many drivers
- KISS: Keep it simple, ..
  - Have a rationale for all aspects
  - Remove unnecessary debug interfaces
- Different interface styles
  - file system, ioctl, sysfs, syscalls
- Compromise with en-vogue interface styles
  - Should make sense for the problem
  - Should not unduly complicate your code

•Interfaces II

•Consider the 32bit compat layer

•Have some design/user documentation

- Manpage for syscalls
- And ideally test code, especially for syscalls

•On the other hand in kernel interfaces are less critical

- Can easily change later
- But when widely used should be still well designed

•A good description

- Patch submission is a publication
  - Must compete in the attention economy
  - People like to read good stories
- Description of the patches is important
  - When applicable hard numbers quantifying a improvement are good
- For larger patch series write an introduction
  - Explain what the patch does and how it improves Linux
  - Describe rationale of contentious design decisions
  - Describe open problems
- When you have problems with English get help
  - Of course only for larger submissions
- Document changes over time

•Establishing trust

- •Accepting a patch means that the maintainer trusts you
  - That you know what you're doing
  - That you deal with problems
- •Trust is built up over time
- •More trust makes the process easier
  - Extreme case maintainer
- •Do development publicly on mailing lists
  - Including bug processing
- •Ideally single engineer should be main interface
- •Working on other areas can establish trust
  - For example fix bugs elsewhere, do cleanups

•Timing: when to post

- •Post early patches as RFC
  - – When it basically works but still has problems
  - – For complex code even multiple RFC stages
  - – Gives you early feedback
  - – Good description still important
- •Ideally do parts of the development process on the mailing list
- •Don't merge when it's too unstable
  - – Can give a bad reputation (“ACPI/JFS effect”)
  - – But doesn't have to be perfect either
  - – Crashes not good, missing functionality is
- •Don't post shortly before/during merge windows
  - – It's too late then
  - – Unless it's a small incremental change

•Dealing with code reviewers

- Reviewing is open for all
  - Actually there is a shortage of reviewers
  - But sometimes there are bad reviews
    - You have to recognize that
- Main focus on the interfaces
  - Both user interfaces and kernel interfaces
- Don't rely on them for logic bugs

•When the reviewer asks for a redesign...

- First often they are right
  - You might to have to just do it.
- They often don't realize how much work it is
  - Try to negotiate if it's unreasonable
- Sometimes they are wrong
- You have to judge it:
  - is it worth it
  - Does it make sense?
- Who asks for it?
  - Maintainer is more important than random reviewer
  - Can also check git logs to judge person

•Resolving problems

- Sometimes submissions get stuck
  - Not enough interest
  - Maintainer loses interest
- Ask the maintainer in private mail for advice
  - Most are reasonable and willing to help
  - If the maintainer doesn't cooperate you can also go higher up the food chain
- For complicated features negotiate a merging plan
  - Especially for dependencies in different trees

•Dealing with controversial features

•Discuss the basic design in advance

•But if discussion is fruitless having working code is also good

•Only part of the submission is controversial?

  - Can you split it out and get the uncontentious parts in first?
  - Later there might be a chance to resubmit them again once the code is established
  - Or you need to redesign only these parts

Resources

- /usr/src/linux/Documentation/
  - SubmittingDrivers
  - SubmitChecklist
  - SubmittingPatches
  - CodingStyle
- OLS paper from proceedings
  - http://halobates.de/on-submitting-kernel-patches.pdf
  - Has more details and further references
- Questions?

Backup

The all-powerful maintainer

- Maintainers have the power over the code
  - They merge or reject your code
  - There are (difficult) ways to appeal
- Who watches the watchmen?
  - Judged by the results
- Maintainers are (usually) constructive
  - But there can be (rare) exceptions
- Don't get into conflict with the maintainer
  - But do not everything mindlessly they ask for
  - Sometimes they are wrong or didn't think something through
  - Explain issues politely
- When there is no clear maintainer merging is difficult
  - Some catchall maintainers as fallback
  - Usually have to attract reviewers unless it's simple

What is code review?

- Linux review is
  - Design review
  - Coding style review
  - Interface review
  - Obvious bugs review

Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY RELATING TO SALE AND/OR USE OF INTEL PRODUCTS, INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel may make changes to specifications, product descriptions, and plans at any time, without notice.

All dates provided are subject to change without notice.

intel®

Leap ahead™