Maintainers Don't Scale

January 20, 2017

This is the write-up of my talk at LCA 2017 in Hobart. It's not exactly the same, because this is a blog and not a talk, but the same contents. The slides for the talk are here, and I will link to the video as soon as it is available. **Update:** Video is now uploaded.

Linux Kernel Maintainers

First let's look at how the kernel community works, and how a change gets merged into Linus Torvalds' repository. Changes are submitted as patches to mailing list, then get some review and eventually get applied by a maintainer to that maintainer's git tree. Each maintainer then sends pull request, often directly to Linus. With a few big subsystems (networking, graphics and ARM-SoC are the major ones) there's a second or third level of sub-maintainers in. 80% of the patches get merged this way, only 20% are committed by a maintainer directly.

Most maintainers are just that, a single person, and often responsible for a bunch of different areas in the kernel with corresponding different git branches and repositories. To my knowledge there are only three subsystems that have embraced group maintainership models of different kinds: TIP (x86 and core kernel), ARM-SoC and the graphics subsystem (DRM).

The radical change, at least for the kernel community, that we implemented over a year ago for the Intel graphics driver is to hand out commit rights to all regular contributors. Currently there are 19 people with commit rights to the drm-intel repository. In the first year of ramp-up 70% of all patches are now committed directly by their authors, a big change compared to how things worked before, and still work everywhere else outside of the graphics subsystem. More recently we also started to manage the drm-misc tree for subsystem wide refactorings and core changes in the same way.

I've covered the details of the new process in my Kernel Recipes talk "Maintainers Don't Scale", and LWN has covered that, and a few other talks, in their article on linux kernel maintainer scalability. I also covered this topic at the kernel summit, again LWN covered the group maintainership discussion. I don't want to go into more detail here, mostly because we're still learning, too, and not really experts on commit rights for everyone and what it takes to make this work well. If you want to enjoy what a community does who really has this all figured out, watch Emily Dunham's talk "Life is better with Rust's community automation" from last year's LCA.

What we are experts on is the Linux Kernel's maintainer model - we've run things for years with the traditional model, both as single maintainers and small groups, and now gained the outside perspective by switching to something completely different. Personally, I've come to believe that the maintainer model as implemented by the kernel community just doesn't scale. Not in the technical sense of big-O scalability, because obviously the kernel community scales to a rather big size. Much larger organizations, entire states are organized in a hierarchical way, the kernel maintainer hierarchy is not anything special. Besides that, git was developed specifically to support the Linux maintainer hierarchy, and git won. Clearly, the linux maintainer model scales to big numbers of contributors. Where I think it falls short is the constant factor of how efficiently contributions are reviewed and merged, especially for non-maintainer contributors. Which do 80% of all patches.

Cult of Busy

The first issue that routinely comes out when talking about maintainer topics is that everyone is overloaded. There's a pervasive spirit in our industry (especially in the US) hailing overworked engineers as heroes, with an entire "cult of busy" around. If you have time, you're a slacker and probably not worth it. Of course this doesn't help when being a maintainer, but I don't believe it's a cause of why the Linux maintainer model doesn't work. This cult of busy leads to burnout, which is in my opinion a prime risk when you're an open source person. Personally I've gone through a few difficult phases until I understood my limits and respected them. When you start as a maintainer for 2-3 people, and it increases to a few dozen within a couple of years, then getting a bit overloaded is rather natural - it's a new job, with a different set of responsibilities and I had no clue about a lot of things. That's no different from suddenly being a leader of a much bigger team anywhere else. A great talk on this topic is “What part of “… for life” don’t you understand?” from Jacob Kaplan-Moss since it’s by a former maintainer. It also contains a bunch of links to talks on burnout specifically. Ignoring burnout is not healthy, or not knowing about the early warning signs, it is rampant in our communities, but for now I’ll leave it at that.

Boutique Trees and Bus Factors

The first issue I see is how maintainers usually are made: You scratch an itch somewhere, write a bit of code, suddenly a few more people find it useful, and “tag” you’re the maintainer. On top, you often end up being stuck in that position “for life”. If the community keeps growing, or your maintainer becomes otherwise busy with work&life, you have your standard-issue overloaded bottleneck.

That’s the point where I think the kernel community goes wrong. When other projects reach this point they start to build up a more formal community structure, with specialized roles, boards for review and other bits and pieces. One of the oldest, and probably most notorious, is Debian with its constitution. Of course a small project doesn’t need such elaborate structures. But if the goal is world domination, or at least creating something lasting, it helps when there’s solid institutions that cope with people turnover. At first just documenting processes and roles properly goes a long way, long before bylaws and codified decision processes are needed.

The kernel community, at least on the maintainer side, entirely lacks this.

What instead most often happens is that a new set of ad-hoc, chosen-by-default maintainers start to crop up in a new level of the hierarchy, below your overload bottleneck. Because becoming your own maintainer is the only way to help out and to get your own features merged. That only perpetuates the problem, since the new maintainers are as likely to be otherwise busy, or occupied with plenty of other kernel parts already. If things go well that area becomes big, and you have another git tree with another overloaded maintainer. More often than not people move around, and accumulate small bits allover under their maintainership. And then the cycle repeats.

The end result is a forest of boutique trees, each covering a tiny part of the project, maintained by a bunch of notoriously overloaded people. The resulting cross-tree coordination issues are pretty impressive - in the graphics subsystem we fairly often end up with with simple drivers that somehow need prep patches in 5 different trees before you can even land that simple driver in the graphics tree.

Unfortunately that's not the bad part. Because these maintainers are all busy with other trees, or their work, or life in general, you're guaranteed that one of them is not available at any given time. Worse, because their tree has relatively little activity because it covers a small area, many only pick up patches once per kernel release, which means a built-in 3 month delay. That's all because each tree and area has just one maintainer. In the end you don't even need the proverbial bus to hit anyone to feel the pain of having a single point of failure in your organization - there's so many maintainer trees around that that absence always happens, and constantly.

Of course people get fed up trying to get features merged, and often the fix is trying to become a maintainer yourself. That takes a while and isn't easy - only 20% of all patches are authored by maintainers - and after the new code landed it makes it all worse: Now there's one more semi-absent maintainer with one more boutique tree, adding to all the existing troubles.

Checks and Balances

All patches merged into the Linux kernel are supposed to be reviewed, and rather often that review is only done by the maintainers who merges the patch. When maintainers send out pull requests the next level of maintainers then reviews those patch piles, until they land in Linus' tree. That's an organization where control flows entirely top-down, with no checks and balances to reign in maintainers who are not serving their contributors well. History of dicatorships tells us that despite best intentions, the end result tends to heavily favour the few over the many. As a crude measure for how much maintainers subject themselves to some checks&balances by their peers and contributors I looked at how many patches authored and committed by the same person (probably a maintainer) do not also carry a reviewed or acked tag. For the Intel driver that's less than 3%. But even within the core graphics code it's only 5%, and that covers the time before we started to experiment with commit rights for that area. And for the graphics subsystem overall the ratio is still only about 25%, including a lot of drivers with essentially just one contributor, who is always volunteered as the maintainer, and hence somewhat natural that those maintainers lack reviewers.

Outside of graphics only roughly 25% of all patches written by maintainers are reviewed by their peers - 75% of all maintainer patches lack any kind of recorded peer review, compared to just 25% for graphics alone. And even looking at core areas like `kernel/` or `mm/` the ratio is only marginally better at about 30%. In short, in the kernel at large, peer review of maintainers isn't the norm.

And there's nothing outside of the maintainer hierarchy that could provide some checks and balance either. The only way to escalate disagreement is by starting a revolution, and revolutions tend to be long, drawn-out struggles and generally not worth it. Even Debian only recently learned that they lack a way to depose maintainers, and that maybe going maintainerless would be easier (again, LWN has you covered).

Of course the kernel is not the only hierarchy where there's no meaningful checks and balances. Professor at universities, and managers at work are in a fairly similar position, with minimal options for students or employers to meaningfully appeal decisions. But that's a recognized problem, and at least somewhat countered by providing ways to provide anonymous feedback, often through regular surveys. The results tend to not be all that significant, but at least provide some control and accountability to the wider masses of first-level dwellers in the hierarchy. In the kernel that amounts to about 80% of all contributions, but there's no such survey. On the contrary, feedback sessions about maintainer happiness only reinforce the control structure, with e.g. the kernel summit featuring an "Is Linus happy?" session each year.

Another closely related aspect to all this is how a project handles personal conflicts between contributors. For a very long time Linux didn't have any formal structures in this area either, with the only options available to unhappy people to either take it or leave it. Well, or usurping a maintainer with a small revolution, but that's not really an option. For two years we've now had the "Code of Conflict", which de facto just throws up its hands and declares that conflict are the normal outcome, essentially just encoding the status quo. Refusing to handle conflicts in a project with thousands of contributors just doesn't work, except that it results in lots of frustration and ultimately people trying to get away. Again, the lack of a poised board to enforce a strong code of conduct, independent of the maintainer hierarchy, is in line with the kernel community unwillingness to accept checks and balances.

Mesh vs. Hierarchy

The last big issue I see with the Linux kernel model, featuring lots of boutique trees and overloaded maintainer, is that it seems to harm collaboration and integration of new contributors. In the Intel graphics, driver maintainers only ever reviewed a small minority of all patches over the last few years, with the goal to foster direct collaboration between contributors. Still, when a patch was stuck, maintainers were the first point of contact, especially, but not only, for newer contributors. No amount of explaining that only the lack of agreement with the reviewer was the gating factor could persuade people to fully collaborate on code reviews and rework the code, tests and documentation as needed. Especially when they're coming with previous experience where code review is more of a rubber-stamp step compared to the distributed and asynchronous pair-programming it often resembles in open-source. Instead, new contributors often just ended up falling back to pinging maintainers to make a decision or just merge the patches as-is.

Giving all regular contributors commit rights and fully trusting them to do the right thing entirely fixed that: If the reviewer or author have commit rights there's no easy excuse anymore to involve maintainers when the author and reviewer can't reach agreement. Of course that requires a lot of work in mentoring people, making sure requirements for merging are understood and documented, and automating as much as possible to avoid screw ups. I think maintainers who lament their lack of review bandwidth, but also state they can't trust anyone else aren't really doing their jobs.

At least for me, review isn't just about ensuring good code quality, but also about diffusing knowledge and improving understanding. At first there's maybe one person, the author (and that's not a given), understanding the code. After good review there should be at least two people who fully understand it, including corner cases. And that's also why I think that group maintainership is the only way to run any project with more than one regular contributor.

On the topic of patch review and maintainers, there's also the habit of wholesale rewrites of patches written by others. If you want others to contribute to your project, then that means you need to accept other styles and can't enforce your own all the time. Merging first and polishing later recognizes new contributions, and if you engage newcomers for the polish work they tend to stick around more often. And even when a patch really needs to be reworked before merging it's better to ask the author to do it: Worst case they don't have time, best case you've improved your documentation and training procedure and maybe gained a new regular contributor on top.

A great take on the consequences of having fixed roles instead of trying to spread responsibilities more evenly is Alice Goldfuss' talk "Rock Stars, Builders, and Janitors: You're doing it wrong". I also think that rigid roles present a bigger bar for people with different backgrounds, hampering diversity efforts and in the spirit of

Sage Sharp's post on what makes a good community, need to be fixed first.

Towards a Maintainer's Manifest

I think what's needed in the end is some guidelines and discussions about what a maintainer is, and what a maintainer does. We have ready-made licenses to avoid havoc, there's code of conducts to copypaste and implement, handbooks for building communities, and for all of these things, lots of conferences. Maintainer on the other hand you become by accident, as a default. And then everyone gets to learn how to do it on their own, while hopefully not burning too many bridges - at least I myself was rather lost on that journey at times. I'd like to conclude with a draft on a maintainer's manifest.

It's About the People

If you're maintainer of a project or code area with a bunch of full time contributors (or even a lot of drive-by contributions) then primarily you deal with people. Insisting that you're only a technical leader just means you don't acknowledge what your true role really is.

And then, trust them to do a good job, and recognize them for the work they're doing. The important part is to trust people just a bit more than what they're ready for, as the occasional challenge, but not too much that they're bound to fail. In short, give them the keys and hope they don't wreck the car too badly, but in all cases have insurance ready. And insurance for software is dirt cheap, generally a `git revert` and the maintainer profusely apologizing to everyone and taking the blame is all it takes.

Recognize Your Power

You're a maintainer, and you have essentially absolute power over what happens to your code. For successful projects that means you can unleash a lot of harm on people who for better or worse are employed to deal with you. One of the things that annoy me the most is when maintainers engage in petty status fights against subordinates, thinly veiled as technical discussions - you end up looking silly, and it just pisses everyone off. Instead recognize your powers, try to stay on the good side of the force and make sure you share it sufficiently with the contributors of your project.

Accept Your Limits

At the beginning you're responsible for everything, and for a one-person project that's all fine. But eventually the project grows too much and you'll just become a dictator, and then failure is all but assured because we're all human. Recognize what you don't do well, build institutions to replace you. Recognize that the responsibility you initially took on might not be the same as that which you'll end up with and either accept it, or move on. And do all that before you start burning out.

Be a Steward, Not a Lord

I think one of key advantages of open source is that people stick around for a very long time. Even when they switch jobs or move around. Maybe the usual "for life" qualifier isn't really a great choice, since it sounds more like a mandatory sentence than something done by choice. What I object to is the "dictator" part, since if your goal is to grow a great community and maybe reach world domination, then you as the maintainer need to serve that community. And not that the community serves you.

Thanks a lot to Ben Widawsky, Daniel Stone, Eric Anholt, Jani Nikula, Karen Sandler, Kimmo Nikkanen and Laurent Pinchart for reading and commenting on drafts of this text.

Share this on → Twitter Google+
Tags: Maintainer-Stuff, Conferences