On submitting kernel patches

Andi Kleen
Intel Open Source Technology Center
ak@linux.intel.com

Abstract

A lot of groups and individual developers work on improving the Linux kernel. Many innovative new features are developed all the time. The best and smoothest way to distribute and maintain new kernel features is to incorporate them into the standard mainline source tree. This involves a review process and some standard conventions. Unfortunately actually getting innovative new features through review can be a rough ride and sometimes they don't make it in at all. This paper examines some common problems for submitting larger changes and some strategies to avoid problems.

1 Introduction

Many people and groups want to contribute to the Linux kernel.

Sometimes it can be difficult to get larger changes into the mainline[1] sources, especially for new contributors. This paper examines some of the challenges in submitting larger patches, and outlines some solutions for them.

This paper assumes that the reader already knows the basics of patch submissions. These are covered in detail in various documents in the Linux kernel source tree (see [4], [3], [5]). Instead of repeating the material in there this paper covers some strategic higher level points in patch submission.

Some of the procedures suggested here are also only applicable to complex or controversal changes. Linux kernel development isn't (yet) a bureaucracy with fixed complicated procedures that cannot be diverged from and there is quite some flexibility in the process.

[1] mainline refers to the kernel.org tree as maintained by Linus Torvalds

2 Why submit kernel patches to mainline?

There are many good reasons not to keep changes private but to submit them to Linus Torvalds' mainline source tree.

- A common approach for companies new to Linux is to take a snapshot of one kernel version (and its associated userland) and try to standardize on that version forever. But freezing on old versions for a long time is not a good idea. New features and bug fixes are constantly being added to mainline and some of them will be eventually needed. Freezing on a old version cuts off from a lot of free development.

  While some changes from mainline can be relatively easily backported to older source trees, many others (and that will likely be the bug fix or feature you want) can be very difficult to backport. The Linux kernel infrastructure is constantly evolving and new changes often rely on newer infrastructure which is not available in old kernels. And even relatively simple backports tend to eventually become a maintenance problem when they add up in numbers because such a tree will diverge quite a lot from a standard kernel and invalidate previous testing. Then there are also security fixes that will be missed in trees that are frozen too long. It is possible to keep up with the security changes in mainline, but it's quite expensive requiring constant effort. And missed security fixes can be very costly to fix later.

  At some point usually, it is required to resync the code base with mainline when updating to a new version. Forward porting private changes during such a version update tend to be hard, especially if they are complicated or affect many parts of the kernel. To avoid this problem submit changes to mainline relatively quickly after developing them,

then they will be part of the standard code base and still be there after version updates. This will also give additional code review, additional improvements and bug fixing and a lot of testing for free.

- For changes and redesigns done in mainline, usually only the requirements of in-tree code are considered. So, even if an enhancement works first externally with just standard exported kernel symbols, these might change or be taken away at any time. The only sure way to avoid that or at least guarantee an equivalent replacement interface to prevent breaking your code is to merge into mainline.

  When the code is in mainline, it will be updated to any interface changes directly or in consultation with the maintainer. And in mainline, the user base will also test on the code and provide free quality-assurance.

- Writing a driver for some device and getting it into mainline means that all the distributions will automatically pick it up and support that device. That makes it much easier for users to use in the end because installing external drivers tends to be complicated and clumsy.

For another perspective on the why to submit changes to mainline see also Andrew Morton's presentation [1]

3 Basics of maintenance

All code in the kernel has some maintenance overhead. Once code is submitted the kernel maintainers commit themselves to keeping it functional for a long time.[2] It is usually expected that the person who submits new code does most of the maintenance for that code at least initially. Some of the procedures described here are actually to demonstrate that the patch submitter is trustworthy enough and they they not just plan to "throw code over the fence."

All code has bugs, so initially when code is submitted, it is assumed contain new defects. Exposing code to mainline also tends to generate a lot of new testing in new unexpected circumstances, which will expose new problems. One important part of patch submission is to make sure these bugs will be handled adequately.

[2]There is a procedure to deprecate functionality too, but it is rarely used and only for very strong reasons.

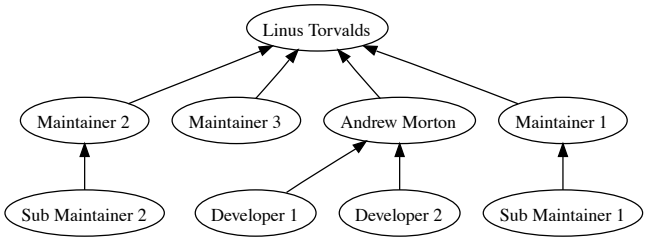

Figure 1: Kernel maintainer hierarchy (example) and patch flow. Andrew Morton is the fallback maintainer taking care of areas with no own maintainer or of patches crossing many subsystems

The mainline kernel changes at a very high rate (see [7] for detailed numbers), and it is very important for the overall quality of the Linux kernel to keep the bug rate under control. See [2] for details on the mechanics of QA and bug reporting in mainline Linux. Because new code often has more bugs than old code, the maintainers tend to use various procedures to make sure the bugs in the new code are minimized early on and handled quickly. One common way to do this is extensive code review.

All new kernel code has to go through standard code review to make sure it follows the kernel coding style[6] and avoids deprecated kernel interfaces and some common mistakes. Code review will also look for other functional problems (although that is not the main focus) and include a design review. Coding style is already covered extensively in [6], and I won't cover it in detail in this paper. In the end, coding style makes only a small part of a successful piece of kernel code anyways and is commonly overrated. Still it is a good idea to follow the coding style in the initial posting so that discussions about white space and variable naming do not distract from the actual functionality of the change.

This paper takes a higher level look on the mechanics of merging larger functionality, and assumes the basic Linux code requirements (like coding style, using standard kernel interfaces etc.) are already covered.

4 Types of submissions

4.1 Easy cases

- The easiest case is a clear bug fix. The need for a bug fix is obvious, and the only argument might be how the bug is actually fixed. But getting clear bug fixes merged is usually no problem. Sometimes, the maintainers might want to fix a particular bug differently than with the original patch, but then the bug will usually be fixed in a different way by someone else. The end result is that the bug is fixed.

  Occasionally, there can be differences on what is considered to be a bug and what is not. In this case, the submitter has to argue for its case in the review mail thread.

- Then there are cleanups. Cleanups can range from very simple as in fixing coding style or making code sparse[3] clean to removing unused code to refactoring an outdated subsystem. Getting such cleanups in is not normally a problem, but they have to be timed right to not conflict with other higher priority changes during the merge windows[4]. The maintainers can normally coordinate that.

- Optimizations are usually welcome, and not too hard to merge, but there are some caveats. When the optimization applies only to a very special case, it is important that it does not impact other more common cases. And there should be benchmark numbers (or other data) showing that the optimization is useful. The impact of the optimization on the code base should also be limited (unless it is a major advantage for an important use-case). Generally optimizations should not impact maintainability too much. Especially when the optimization is not a dramatic improvement or does apply only to some special cases, it is important that it is clean code and its impact is limited. Cleaning up some code while doing the optimization will make the optimization much more attractive.

[3] Sparse is a static code analysis tool written for the Linux kernel. See [8]. But like most static checking tools it needs a large amount of work initially to eliminate false positives.

[4] Merge window is the two week period after each major releases where maintainers send major features to Linus Torvalds tree. Most features and code changes go in at that time

4.2 Hardware drivers

The need can be easily established for submitting hardware drivers for standard devices like network cards or SCSI adapters. The hardware exists and Linux should support it and the driver (if correctly written) will in most cases not impact any other code.

The increasing the bug rate argument in Section 3 is fortunately not a serious problem in this special case If the hardware is not there, the driver will not be used, and can then also not cause bugs.[5] Luckily, this means that because most of the kernel source base are drivers, the effective bug rate is not raising as quickly as you would expect from the raw source size growth. Still, this is a problem maintainers are concerned about, especially for core kernel code that is used on nearly[6] all hardware and on drivers for hardware used very widely in the user-base.

There are well-established procedures to get new drivers in, and doing so is normally not a problem. In some cases, depending on the code quality of the driver, it can take a long time with many iterations of reviews and fixes.

One more difficult issue are special optimizations for specific drivers. Most drivers won't need any, for example, a standard NIC or block driver is usually not a problem to add because it only plugs into the well established network driver interface. There will be no changes on other code.

On the other hand, if a hardware device does for example RDMA[7], and needs special support in the core networking stack to support that, merging that will be much more difficult because these code changes could impact other setups too. One recommended strategy in this case is to first get basic support in while minimizing changes to core infrastructure

Sometimes, there is first a rough consensus in the kernel community that particular optimizations should not be supported for various reasons. One example of this

[5] This ignores the case of non-essential drivers for common hardware. Adding them could risk increasing the bug rate.

[6] Nearly because there are some special cases like devices without a block driver or MMU-less devices that disable significant parts of a standard kernel.

[7] Remote DMA, DMA directly controlled by a remote machine over the network

is stateful TCP/IP stack off-loading[8] or native support for hashed page tables in the core VM. Of course such consensus can be eventually re-evaluated when the facts change (or it is demonstrated that the optimization is really needed), but it is typically difficult to do so.

Once the basic support is in, and you need some specific changes to optimize for your special case, one reasonable way to get this done is to do clean-up or redesign that improves the standard subsystem code (not considering your changes), and then just happens to make your particular optimization easier. The trade-off is here that offseting the maintainability impact on the subsystem by cleaning it up and improving it first, later re-adding some complexity for special optimizations can be justified.

Assuming your proposed change does not fall into one of these difficult areas, it should be relatively easy to get it included in mainline once the driver passes the basic code review.

If it is in a difficult area, it is usually better to at least try to merge it, but will require much more work. In a few extreme cases the actual merge will be very hard to impossible too, for instance when you're planning to submit a patch supporting a full TCP offload engine. On the other hand, if the arguments are good maintainers sometimes reconsider.

4.3 New core functionality

An especially touchy topic is adding new hooks into the core kernel, like new notifier chains or new functions calling into specific subsystems. Very generic notifiers and hooks tend to have a large maintenance impact because they have the potential to alter the code flow in unexpected ways, lead to lock order problems, bugs, and unexpected behavior, and generally making code harder to maintain and debug later. That is why maintainers tend to be not very happy about adding them to their subsystems. If you really need the hooks anyways trading cleanups for hooks as described in section 4.2 is a reasonable (but not guaranteed to be successful) strategy

Usually, there will be a discussion on the need for the hooks on the mailing list, with commenters suggesting design alternative if the case is not very clear. This may result in you having to redesign some parts if you cannot convince the maintainer of the benefits of your particular design. A redesign might include moving some parts to userland or doing it altogether differently.

To avoid wasting too much work, it is a good idea to discuss the basic design publicly before spending too much time on real production code. Of course doing prototypes first to measure basic feasibility is still a good idea. Just do not expect these prototypes to be necessarily merged exactly as they are. As usual prototype code tends to require some work to make it production ready.

5 Splitting submissions into pieces

It is also fairly important to submit larger changes in smaller pieces so that reviewers and maintainers can process the changes step-by-step. Normally this means splitting a larger patch series into smaller logical chunks than can be reviewed together. There are exceptions to this. For example a single driver source file that is completely new is is normally reviewed together, even when it is large. But this protocol is fairly important for any changes to existing code.

These changes must be bisectable:[9] that is applying or unapplying any patch in the sequence must still produce a buildable and working kernel.

Usually, you will need to revise patches multiple times based on the feedback before they can be accepted. Avoid patch splitting methods that cause you a lot of work each time you post. Ideally you keep the split patches in change set oriented version control system like git or mercurial or in a patch management system like quilt[9]. Personally, I prefer quilt for patch management which has the smoothest learning curve. [10] has a introduction on using quilt.

It is also recommended to time submissions of large patch kits. Posting more than 20 patches at a time will overwhelm the review capacity of the mailing lists. Group them into logical groups and post these one at a time with at least a day grace time in between.

Patches should be logical steps with their own descriptions, but they don't need to be too small. Only create

[8] Partial stateless offloads like Large-Receive-Offload (LRO) or TCP Segmentation Offload (TSO) on the other hand are already supported.

[9] Typically used with the "git bisect" command for semi-automated debugging.

very small (less than 10 lines change) patches if they are really a logically an self-contained change. On the other hand, new files typically do not need to be split up, except when parts of them logically belong to other changes in other areas.

When you post revised patches, add a version number to the subject, and also maintain a brief change log at the end of the patch description.

6 A good description

Submitting a Linux kernel patch is like publishing a paper. It will be scrutinized by a sceptical public. That is why the description is very important. You should spend some time writing a proper introduction explaining what your change is all about, what your motivations were and what the important design decisions and trade-offs are.

Ideally you will also already address expected counter arguments in the description. It is a good idea to browse the mailing list archives beforehand and to study a few successful submissions there.

Hard numbers quantifying improvements are always appreciated, too, in the description. If there is something to measure, measure it and include the numbers. If your patch is an optimization, measure the improvement and include the numbers.

The quality of a description can make or break whether your kernel patch is accepted. If you cannot convince the kernel reviewers that your work is interesting and worthwhile, then there will be no code review and without visible code review the code will likely not be merged.

The linux-kernel mailing list (and other kernel project mailing lists) are an attention economy, and it is important to be competitive here.

7 Establishing trust

If your patch is larger than just a change to an existing system (like a new driver or similar), you will be expected to be the maintainer of that code for at least some time. A maintainer is someone who takes care of the code of some subsystem, incorporates patches, makes sure they are up to standard, does some release engineering to stabilize the code for releases, and sends changes bundled to the next level maintainer.

This relationship involves some level of trust. The next level maintainer trusts you to do this job properly, and keep linux code quality high. If the person is not known yet from other projects, the only way to get such trust is to do something publically visible. That could be done by submitting some self-written code that is high quality or by fixing bugs.

The maintainers in general favor people who do not only care about their little piece of code but who take a little larger view of the code base, and are known to improve, and fix other areas of the kernel too. This does not necessarily mean a lot of work, and could be just an occasional bug fix.

8 Setting up a community

For larger new kernel features like a file system or a network protocol, it is a good idea to have a small user base first. This user base is needed for testing, and to make sure there is actually interest in the new feature.

It is recommended to at least initially (while your community is still young) keep most of the technical discussion on the linux-kernel mailing list. This way the maintainers can see there is an active group working on the particular feature, fixing bugs, and caring about user's needs which then builds up trust for an eventual merge.

9 When to post

Publishing a patch is a very important event in its lifetime. There is a delicate balance between posting too early and posting too late.

First, once the code basically works, it is a good idea to post it as a Request-for-Comments (RFC), clearly marking it as such. For more complicated changes or you're not sure yet what will be acceptable to the maintainers you can also post a rough prototype or even just a design overview what you're planning as RFC. This would not be intended to be merged, but just to invite initial comments, and to make other developers aware you have something in the pipeline. There should already be valuable feedback in that stage, and when the feedback includes requests for larger changes, you do not need to throw as much work away when you redo code, as you

would had posted a finished patch. For simple changes the RFC stage can be skipped, but for anything more complex it is typically a good idea. For very complex or very controversial changes you will likely go through multiple RFC stages.

Also, conducting more of the development process visible on the mailing list is good for building trust, and for establishing a community as discussed earlier.

On the other hand, actually merging (submitting to a upstream maintainer) too early when you still know the code is unstable is a bad idea. The problem is that even when some subsystem is marked experimental, people will start using it, and if it does not work or only works very badly or worse corrupts data, it will get a bad name. Getting a bad name early on can be a huge problem later for a project and it will be hard to ever get rid of that taint again[10]

So there should be some basic stability before actually submitting it to merge. On the other hand, it definitely does not need to be finished. Feature completeness or final performance is not a requirement.

Patches take some time to travel through review and then the various maintainer trees. If you want a change in a particular release it is really too late when you only post it during the two week merge window. Rather when the merge window opens the patch should be already through review and traveled up the maintainer hierarchy. That will take some time. And during the merge window reviewer capacity tends to be in short order and there might be none left for you.

10 Dealing with reviews

Code review is an integral part of the Linux code submission process. It proceeds on public mailing lists with everyone allowed to comment on your patches. Most of the comments will be useful and help to improve the code. This is usually fairly obvious, in this case just make the changes they request.

Sometimes even when the comments are useful, they might cause you excessive work: for example when they ask for a redesign. Sometimes there can be very good reasons for the redesign, sometimes not. It is your judgment. Or sometimes the reviewer just missed something and the suggestion wouldn't actually improve the code or not handle some case correctly. If the reason is convincing, you should just make the changes if feasible.

In some cases, the reviewer might not realize how much work it would be to implement a particular change. If you are not convinced of the reasons for the redesign or it is just not feasible because it would take too long, explain that clearly with pure technical arguments on the mailing list (but do not get yourself dragged into a flame war).

Then if even after discussion the reviewer still insist on you making such a change you don't like, you have to judge the request: If the the maintainer of the subsystem or a upstream maintainer requests the change, and you cannot convince them to change their mind, you'll have to implement the change or drop the submission. If someone other than the maintainer requests the change, it is useful to ask the maintainer for their opinion in a private mail before embarking on large projects addressing review comments.

A reasonable rule of thumb (but there are exceptions of course!) for how serious to take a reviewer is to check how much code the person contributed. As a first approximation, if someone never contributed any patches themselves their comments are less important,[11] and you could ignore them partly or completely after describing why on the mailing list. You can look up the contributions of a particular person in the git version history. But you really should do that only in extreme cases when there is no other choice. Linux kernel development unfortunately suffers from a shortage of reviewers so you should consider well before you ignore one and only do that for very strong reasons.

And sometimes you will notice that the comments from a particular reviewer are just not constructive. This comes from the fact that review is open to all on the internet, and there are occasionally bad reviewers too. These cases tend to be usually clear, and it is reasonable not to address such unproductive comments.

11 Merging plans

For complicated patch kits, especially when they depend on other changes, and after the basic reviews are done, it is also a good idea to negotiate a merging plan with

[10] There are several high quality subsystems in the kernel like JFS or ACPI who suffer from this problem.

[11] Some people call Linux a "codeocracy" because of that.

the various stake holders. This is especially important if changes touch multiple subsystems, and might need in theory to go through multiple maintainers (which can be quite tedious to coordinate). Merging plan would be a simple email telling them in what order and when the patches will go in and get their approval. If a change touches a lot of subsystems you don't necessarily need the approval from all maintainers. Such tree wide sweeps are normally coordinated by the upstream maintainers.

12 Interfaces

Reviewers and maintainers focus on the interface designs for user space programs. This is because merging an externally visible interface commits the kernel to the interface because other code will depend on it. Changing a published interface later is much harder and often special backwards compatibility code is required. This usually leads to very close scrutiny of interfaces by reviewers and maintainers before merge. To avoid delays, it is best to get interface designs right on the first try. On the other hand this first try should be as simple as possible and not include ever feature you plan on the first iteration.

For many submissions, like device drivers for standard devices or a file system, user space interfaces will not be relevant because they don't have user interfaces of their own.

- There are various interface styles (e.g. file systems, files in sysfs, system calls, ioctls, netlink, character devices) which have different trade offs and are the right choice for different areas. It is important to chose the interface style that fits the application best.
- There should be some design documentation on the interface included with the submission.
- It is recommended to make any new interfaces as simple as possible before submission. This will make reviewing easier and they can be still later extended.
- 64bit architectures typically use a compat layer to translate system calls from 32bit application to the internal 64bit ABI of the kernel. The needs of the compat layer needs to be considered for new interfaces.
- For new system calls or system call extensions there should be a manpage and some submittable test code.
- Remove any private debug interfaces before submission if it's not clear they will be useful long term for code maintenance. Alternatively maintain them as a separate add-on patch.

On the other hand internal kernel interfaces are considered subject to change and are much less critical.

13 Resolving problems

Sometimes a submission gets stuck during the submission process. In this case, it is a good idea to just send private mail to the maintainer who is responsible and ask advice on how to proceed with the merge. Alternatively, it is also possible to ask one of the upstream maintainers (in case the problem is with the maintainer)

There is also the linux-mentors[11] program and the kernelnewbies project[12] who can help with process issues.

14 Case studies

14.1 dprobes

Dprobes [13] was a dynamic tracing project from IBM originally ported from OS/2 and released as a kernel patch in 2000. It allowed to attach probes written in a simple stack based byte code language to arbitrary code in the kernel or user space, to collect information on kernel behavior non intrusively. Dynamic tracing is a very popular topic now[12], but back then, dprobes was clearly ahead of its time. There was not much interest in it.

The dprobes team posted various versions with increasing functionality, but could not really build a user community. dprobes was a comprehensive solution, covering both user space and kernel tracing. The user space tracing required some changes to the Virtual Memory subsystem that were not well received by the VM developers. Putting a byte-code interpreter into the kernel was also unpopular, both from its code and because

[12] Especially due to the marketing efforts of a particular operating system vendor for a much later similar project

the kernel developers as potential user base preferred to write such code in C. That lead to the original dprobes code never being merged to mainline and never getting a real user base.

After a few years of unsuccessful merging attempts, maintaining the code out of tree and existing in relative obscurity the project reinvented itself. One part of dprobes was the ability to set non intrusive breakpoints to any kernel code. They extracted that facility and allowed it to attach handlers in kernel modules written in C to arbitrary kernel functions. This new facility was called kprobes [15]. kprobes became quickly relatively popular with the kernel community and was merged after a short time It attracted also significant contributors from outside the original dprobes project. This was both because it was a much simpler patch, touching less code, none of its changes were controversial, and that other competing operating systems had made dynamic tracing popular by then, so there was generally much more interest.

Kprobes then was the kernel infrastructure used by the [14] project started shortly after the merge. systemtap is a scripting language that generates C source for a kernel module that then then uses kprobes to trace kernel functions. systemtap and kprobes are popular Linux features now, but it took a long time to get there. Also, significant features of the original dprobes (attaching probes to user space code) are still missing. Support for user probes is currently developed as part of the utrace project, but unfortunately utrace development proceeds slowly and has unlikely prospects for merge because utrace is a gigantic "redesign everything" patch developed on a private mailing list.

The lessons to learn from the dprobes story are:

- When building something new and radical, it is needed to "sell" it at least a little, to get a user base and interest from reviewers. dprobes, while being technically a very interesting project, did not compete well initially in the attention economy.
- Don't try to put in all features on the first step. Submit features step-by-step.
- When a particular part of a submission is very unpopular, strip it out to not let it delay submission of the other parts.
- Don't wait too long to redesign if the original design is too unpopular.

14.2 perfmon2

perfmon1 was a relatively simple architecture specific interface for the performance counters of the IA64 port. That code was integrated into the mainline Linux source. Later, it was redesigned as perfmon2[16] with many more features, support for more performance monitoring hardware, support for flexible output formats using plug-ins, and its cross architecture support[13].

perfmon2 was developed outside the normal Linux kernel, together with its user land components at [17]. While it was able to attract some code contributions and made some attempts to be merged into the mainline kernel, it has not succeeded yet. This is mainly because it has acquired many features on its long out-of-tree development path that are difficult to untangle now[14]

This lead to the perfmon2 patch submissions being very large patches with many interdependencies which are very difficult to understand and review and debug later. It also didn't help that its very flexible plug-in-based output module architecture was a design pattern not previously used in Linux. Kernel programmer's are relatively conservative regarding design patterns. And the subsystem came with a very complicated system call-based interface that was difficult to evaluate. Also, many perfmon2 features were relatively old and it was sometimes impossible to reconstruct why they were even added. In turn, the patches were unable to either attract enough reviewers or to satisfy the comments of the few reviews it got. That in turn, didn't lead to maintainer confidence in the code and there was no mainline merge so far, except for some trivial separatable changes.

- Be conservative with novel design patterns (like the perfmon2 output plug-ins) Kernel programmers are conservative regarding coding style and not very open to novel programming styles.
- Don't combine too many novelties into a single patch. If you have a lot of new ideas do them step by step.

[13] perfmon2 is generally considered to be showing some signs of the "Second System effect".

[14] Often programmers don't remember ever detail on why they did some code years before.

- Don't make the code too flexible internally. Too much flexibility makes it harder to evaluate.
- Don't do significant follow-on development outside the mainline kernel tree. Concentrate on developing the basic subsystem without too many bells and whistles and merge that quickly into the mainline. Then any additional features requested by users should be submitted to the mainline incrementally.
- If any optional features are already implemented up front before merging, develop them as incremental patches to the core code base, not in an integrated source tree.

In the author's opinion, the only promising strategy for a perfmon2 merge now would be similar to the evolution from dprobes to kprobes in section 14.1. Go back to the core competencies: define the core functionality of a performance counter interface, of develop a "perfmon3", which only implements the core functionality in a very clean way, and submit that. Then port forward features requested by the user step-by-step from perfmon2, each time you reevaluate the design of a particular feature and its user interface. This process would likely lead to a much cleaner perfmon2 code base. There is some work underway now by the perfmon2 author to do this.

15 Conclusion

Submitting code to the mainline Linux kernel is rewarding, but also takes some care to be successful. With the rough guidelines in this paper, I hope some commons problems while submitting Linux kernel changes can be avoided.

References

[1] Andrew Morton
kernel.org development and the embedded world
http://userweb.kernel.org/~akpm/rants/elc-08.odp

[2] Andrew Morton
Production process, bug tracking and quality assurance of the Linux kernel
http://userweb.kernel.org/~akpm/rants/hannover-kernel-qa.odp

[3] *Documentation/SubmittingDrivers* in the Linux kernel source from
http://www.kernel.org
Living document. Always refer to the version from the latest kernel release.

[4] *Documentation/SubmitChecklist* in your Linux kernel source from
http://www.kernel.org
Living document. Always refer to the version from the latest kernel release.

[5] *Documentation/SubmittingPatches* in the Linux kernel source from
http://www.kernel.org
Living document. Always refer to the version from the latest kernel release.

[6] *Documentation/CodingStyle* in the Linux kernel source from http://www.kernel.org
Living document. Always refer to the version from the latest kernel release.

[7] Kroah-Hartmann *et al.*,
https://www.linux-foundation.org/publications/linuxkerneldevelopment.php.

[8] Sparse, the semantic parser,
http://www.kernel.org/pub/software/devel/sparse/

[9] Quilt, the patch manager,
http://savannah.nongnu.org/projects/quilt

[10] Grünbacher,
How to survive with many patches or Introduction to Quilt,
http://www.suse.de/~agruen/quilt.pdf

[11] http://www.linuxmentors.org

[12] http://www.kernelnewbies.org

[13] http://dprobes.sourceforge.net

[14] Prasad
Locating System Problems using Dynamic Instrumentation
Proceedings of the Ottawa Linux Symposium 2005

[15] Keniston
Documentation/kprobes.txt in the Linux kernel source from `http://www.kernel.org`

[16] Eranian
Perfmon2: A flexible performance monitoring interface for Linux
Proceedings of the Ottawa Linux Symposium 2006

[17] `http://perfmon2.sourceforge.net`

This work represents the opinion of the author and not of Intel.