Followthrough

At this point, you have followed the guidelines given so far and, with the addition of your own engineering skills, have posted a perfect series of patches. One of the biggest mistakes that even experienced kernel developers can make is to conclude that their work is now done. In truth, posting patches indicates a transition into the next stage of the process, with, possibly, quite a bit of work yet to be done.

It is a rare patch which is so good at its first posting that there is no room for improvement. The kernel development process recognizes this fact, and, as a result, is heavily oriented toward the improvement of posted code. You, as the author of that code, will be expected to work with the kernel community to ensure that your code is up to the kernel's quality standards. A failure to participate in this process is quite likely to prevent the inclusion of your patches into the mainline.

Working with reviewers

A patch of any significance will result in a number of comments from other developers as they review the code. Working with reviewers can be, for many developers, the most intimidating part of the kernel development process. Life can be made much easier, though, if you keep a few things in mind:

- If you have explained your patch well, reviewers will understand its value and why you went to the trouble of writing it. But that value will not keep them from asking a fundamental question: what will it be like to maintain a kernel with this code in it five or ten years later? Many of the changes you may be asked to make - from coding style tweaks to substantial rewrites - come from the understanding that Linux will still be around and under development a decade from now.
- Code review is hard work, and it is a relatively thankless occupation; people remember who wrote kernel code, but there is little lasting fame for those who reviewed it. So reviewers can get grumpy, especially when they see the same mistakes being made over and over again. If you get a review which seems angry, insulting, or outright offensive, resist the impulse to respond in kind. Code review is about the code, not about the people, and code reviewers are not attacking you personally.
- Similarly, code reviewers are not trying to promote their employers' agendas at the expense of your own. Kernel developers often expect to be working on the kernel years from now, but they understand that their employer could change. They truly are, almost without exception, working toward the creation of the best kernel they can; they are not trying to create discomfort for their employers' competitors.

What all of this comes down to is that, when reviewers send you comments, you need to pay attention to the technical observations that they are making. Do not let their form of expression or your own pride keep that from happening. When you get review comments on a patch, take the time to understand what the reviewer is trying to say. If possible, fix the things that the reviewer is asking you to fix. And respond back to the reviewer: thank them, and describe how you will answer their questions.

Note that you do not have to agree with every change suggested by reviewers. If you believe that the reviewer has misunderstood your code, explain what is really going on. If you have a technical objection to a suggested change, describe it and justify your solution to the problem. If your explanations make sense, the reviewer will accept them. Should your explanation not prove persuasive, though, especially if others start to agree with the reviewer, take some time to think things over again. It can be easy to become blinded by your own solution to a problem to the point that you don't realize that something is fundamentally wrong or, perhaps, you're not even solving the right problem.

Andrew Morton has suggested that every review comment which does not result in a code change should result in an additional code comment instead; that can help future reviewers avoid the questions which came up the first time around.

One fatal mistake is to ignore review comments in the hope that they will go away. They will not go away. If you repost code without having responded to the comments you got the time before, you're likely to find that your patches go nowhere.

Speaking of reposting code: please bear in mind that reviewers are not going to remember all the details of the code you posted the last time around. So it is always a good idea to remind reviewers of previously raised issues and how you dealt with them; the patch changelog is a good place for this kind of information. Reviewers should not have to search through list archives to familiarize themselves with what was said last time; if you help them get a running start, they will be in a better mood when they revisit your code.

What if you've tried to do everything right and things still aren't going anywhere? Most technical disagreements can be resolved through discussion, but there are times when somebody simply has to make a decision. If you honestly believe that this decision is going against you wrongly, you can always try appealing to a higher power. As of this writing, that higher power tends to be Andrew Morton. Andrew has a great deal of respect in the kernel development community; he can often unjam a situation which seems to be hopelessly blocked. Appealing to Andrew should not be done lightly, though, and not before all other alternatives have been explored. And bear in mind, of course, that he may not agree with you either.

What happens next

If a patch is considered to be a good thing to add to the kernel, and once most of the review issues have been resolved, the next step is usually entry into a subsystem maintainer's tree. How that works varies from one subsystem to the next; each maintainer has his or her own way of doing things. In particular, there may be more than one tree - one, perhaps, dedicated to patches planned for the next merge window, and another for longer-term work.

For patches applying to areas for which there is no obvious subsystem tree (memory management patches, for example), the default tree often ends up being -mm. Patches which affect multiple subsystems can also end up going through the -mm tree.

Inclusion into a subsystem tree can bring a higher level of visibility to a patch. Now other developers working with that tree will get the patch by default. Subsystem trees typically feed linux-next as well, making their contents visible to the development community as a whole. At this point, there's a good chance that you will get more comments from a new set of reviewers; these comments need to be answered as in the previous round.

What may also happen at this point, depending on the nature of your patch, is that conflicts with work being done by others turn up. In the worst case, heavy patch conflicts can result in some work being put on the back burner so that the remaining patches can be worked into shape and merged. Other times, conflict resolution will involve working with the other developers and, possibly, moving some patches between trees to ensure that everything applies cleanly. This work can be a pain, but count your blessings: before the advent of the linux-next tree, these conflicts often only turned up during the merge window and had to be addressed in a hurry. Now they can be resolved at leisure, before the merge window opens.

Some day, if all goes well, you'll log on and see that your patch has been merged into the mainline kernel. Congratulations! Once the celebration is complete (and you have added yourself to the MAINTAINERS file), though, it is worth remembering an important little fact: the job still is not done. Merging into the mainline brings its own challenges.

To begin with, the visibility of your patch has increased yet again. There may be a new round of comments from developers who had not been aware of the patch before. It may be tempting to ignore them, since there is no longer any question of your code being merged. Resist that temptation, though; you still need to be responsive to developers who have questions or suggestions.

More importantly, though: inclusion into the mainline puts your code into the hands of a much larger group of testers. Even if you have contributed a driver for hardware which is not yet available, you will be surprised by how many people will build your code into their kernels. And, of course, where there are testers, there will be bug reports.

The worst sort of bug reports are regressions. If your patch causes a regression, you'll find an uncomfortable number of eyes upon you; regressions need to be fixed as soon as possible. If you are unwilling or unable to fix the regression (and nobody else does it for you), your patch will almost certainly be removed during the stabilization period. Beyond negating all of the work you have done to get your patch into the mainline, having a patch pulled as the result of a failure to fix a regression could well make it harder for you to get work merged in the future.

After any regressions have been dealt with, there may be other, ordinary bugs to deal with. The stabilization period is your best opportunity to fix these bugs and ensure that your code's debut in a mainline kernel release is as solid as possible. So, please, answer bug reports, and fix the problems if at all possible. That's what the stabilization period is for; you can start creating cool new patches once any problems with the old ones have been taken care of.

And don't forget that there are other milestones which may also create bug reports: the next mainline stable release, when prominent distributors pick up a version of the kernel containing your patch, etc. Continuing to respond to these reports is a matter of basic pride in your work. If that is insufficient motivation, though, it's also worth considering that the development community remembers developers who lose interest in their code after it's merged. The next time you post a patch, they will be evaluating it with the assumption that you will not be around to maintain it afterward.

Other things that can happen

One day, you may open your mail client and see that somebody has mailed you a patch to your code. That is one of the advantages of having your code out there in the open, after all. If you agree with the patch, you can either forward it on to the subsystem maintainer (be sure to include a proper From: line so that the attribution is correct, and add a signoff of your own), or send an Acked-by: response back and let the original poster send it upward.

If you disagree with the patch, send a polite response explaining why. If possible, tell the author what changes need to be made to make the patch acceptable to you. There is a certain resistance to merging patches which are opposed by the author and maintainer of the code, but it only goes so far. If you are seen as needlessly blocking good work, those patches will eventually flow around you and get into the mainline anyway. In the Linux kernel, nobody has absolute veto power over any code. Except maybe Linus.

On very rare occasion, you may see something completely different: another developer posts a different solution to your problem. At that point, chances are that one of the two patches will not be merged, and "mine was here first" is not considered to be a compelling technical argument. If somebody else's patch displaces yours and gets into the mainline, there is really only one way to respond: be pleased that your problem got solved and get on with your work. Having one's work shoved aside in this manner can be hurtful and discouraging, but the community will remember your reaction long after they have forgotten whose patch actually got merged.