Advanced topics

At this point, hopefully, you have a handle on how the development process works. There is still more to learn, however! This section will cover a number of topics which can be helpful for developers wanting to become a regular part of the Linux kernel development process.

Managing patches with git

The use of distributed version control for the kernel began in early 2002, when Linus first started playing with the proprietary BitKeeper application. While BitKeeper was controversial, the approach to software version management it embodied most certainly was not. Distributed version control enabled an immediate acceleration of the kernel development project. In current times, there are several free alternatives to BitKeeper. For better or for worse, the kernel project has settled on git as its tool of choice.

Managing patches with git can make life much easier for the developer, especially as the volume of those patches grows. Git also has its rough edges and poses certain hazards; it is a young and powerful tool which is still being civilized by its developers. This document will not attempt to teach the reader how to use git; that would be sufficient material for a long document in its own right. Instead, the focus here will be on how git fits into the kernel development process in particular. Developers who wish to come up to speed with git will find more information at:

> http://git-scm.com/
>
> http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

and on various tutorials found on the web.

The first order of business is to read the above sites and get a solid understanding of how git works before trying to use it to make patches available to others. A git-using developer should be able to obtain a copy of the mainline repository, explore the revision history, commit changes to the tree, use branches, etc. An understanding of git's tools for the rewriting of history (such as rebase) is also useful. Git comes with its own terminology and concepts; a new user of git should know about refs, remote branches, the index, fast-forward merges, pushes and pulls, detached heads, etc. It can all be a little intimidating at the outset, but the concepts are not that hard to grasp with a bit of study.

Using git to generate patches for submission by email can be a good exercise while coming up to speed.

When you are ready to start putting up git trees for others to look at, you will, of course, need a server that can be pulled from. Setting up such a server with git-daemon is relatively straightforward if you have a system which is accessible to the Internet. Otherwise, free, public hosting sites (Github, for example) are starting to appear on the net. Established developers can get an account on kernel.org, but those are not easy to come by; see http://kernel.org/faq/ for more information.

The normal git workflow involves the use of a lot of branches. Each line of development can be separated into a separate "topic branch" and maintained independently. Branches in git are cheap, there is no reason to not make free use of them. And, in any case, you should not do your development in any branch which you intend to ask others to pull from. Publicly-available branches should be created with care; merge in patches from development branches when they are in complete form and ready to go - not before.

Git provides some powerful tools which can allow you to rewrite your development history. An inconvenient patch (one which breaks bisection, say, or which has some other sort of obvious bug) can be fixed in place or made to disappear from the history entirely. A patch series can be rewritten as if it had been written on top of today's mainline, even though you have been working on it for months. Changes can be transparently shifted from one branch to another. And so on. Judicious use of git's ability to revise history can help in the creation of clean patch sets with fewer problems.

Excessive use of this capability can lead to other problems, though, beyond a simple obsession for the creation of the perfect project history. Rewriting history will rewrite the changes contained in that history, turning a tested (hopefully) kernel tree into an untested one. But, beyond that, developers cannot easily collaborate if they do not have a shared view of the project history; if you rewrite history which other

developers have pulled into their repositories, you will make life much more difficult for those developers. So a simple rule of thumb applies here: history which has been exported to others should generally be seen as immutable thereafter.

So, once you push a set of changes to your publicly-available server, those changes should not be rewritten. Git will attempt to enforce this rule if you try to push changes which do not result in a fast-forward merge (i.e. changes which do not share the same history). It is possible to override this check, and there may be times when it is necessary to rewrite an exported tree. Moving changesets between trees to avoid conflicts in linux-next is one example. But such actions should be rare. This is one of the reasons why development should be done in private branches (which can be rewritten if necessary) and only moved into public branches when it's in a reasonably advanced state.

As the mainline (or other tree upon which a set of changes is based) advances, it is tempting to merge with that tree to stay on the leading edge. For a private branch, rebasing can be an easy way to keep up with another tree, but rebasing is not an option once a tree is exported to the world. Once that happens, a full merge must be done. Merging occasionally makes good sense, but overly frequent merges can clutter the history needlessly. Suggested technique in this case is to merge infrequently, and generally only at specific release points (such as a mainline -rc release). If you are nervous about specific changes, you can always perform test merges in a private branch. The git "rerere" tool can be useful in such situations; it remembers how merge conflicts were resolved so that you don't have to do the same work twice.

One of the biggest recurring complaints about tools like git is this: the mass movement of patches from one repository to another makes it easy to slip in ill-advised changes which go into the mainline below the review radar. Kernel developers tend to get unhappy when they see that kind of thing happening; putting up a git tree with unreviewed or off-topic patches can affect your ability to get trees pulled in the future. Quoting Linus:

```
You can send me patches, but for me to pull a git patch from you, I
need to know that you know what you're doing, and I need to be able
to trust things *without* then having to go and check every
individual change by hand.
```

(http://lwn.net/Articles/224135/).

To avoid this kind of situation, ensure that all patches within a given branch stick closely to the associated topic; a "driver fixes" branch should not be making changes to the core memory management code. And, most importantly, do not use a git tree to bypass the review process. Post an occasional summary of the tree to the relevant list, and, when the time is right, request that the tree be included in linux-next.

If and when others start to send patches for inclusion into your tree, don't forget to review them. Also ensure that you maintain the correct authorship information; the git "am" tool does its best in this regard, but you may have to add a "From:" line to the patch if it has been relayed to you via a third party.

When requesting a pull, be sure to give all the relevant information: where your tree is, what branch to pull, and what changes will result from the pull. The git request-pull command can be helpful in this regard; it will format the request as other developers expect, and will also check to be sure that you have remembered to push those changes to the public server.

Reviewing patches

Some readers will certainly object to putting this section with "advanced topics" on the grounds that even beginning kernel developers should be reviewing patches. It is certainly true that there is no better way to learn how to program in the kernel environment than by looking at code posted by others. In addition, reviewers are forever in short supply; by looking at code you can make a significant contribution to the process as a whole.

Reviewing code can be an intimidating prospect, especially for a new kernel developer who may well feel nervous about questioning code - in public - which has been posted by those with more experience. Even code written by the most experienced developers can be improved, though. Perhaps the best piece of advice for reviewers (all reviewers) is this: phrase review comments as questions rather than criticisms.

Asking "how does the lock get released in this path?" will always work better than stating "the locking here is wrong."

Different developers will review code from different points of view. Some are mostly concerned with coding style and whether code lines have trailing white space. Others will focus primarily on whether the change implemented by the patch as a whole is a good thing for the kernel or not. Yet others will check for problematic locking, excessive stack usage, possible security issues, duplication of code found elsewhere, adequate documentation, adverse effects on performance, user-space ABI changes, etc. All types of review, if they lead to better code going into the kernel, are welcome and worthwhile.