4: GETTING THE CODE RIGHT

While there is much to be said for a solid and community-oriented design process, the proof of any kernel development project is in the resulting code. It is the code which will be examined by other developers and merged (or not) into the mainline tree. So it is the quality of this code which will determine the ultimate success of the project.

This section will examine the coding process. We'll start with a look at a number of ways in which kernel developers can go wrong. Then the focus will shift toward doing things right and the tools which can help in that quest.

4.1: PITFALLS

* Coding style

The kernel has long had a standard coding style, described in Documentation/CodingStyle. For much of that time, the policies described in that file were taken as being, at most, advisory. As a result, there is a substantial amount of code in the kernel which does not meet the coding style guidelines. The presence of that code leads to two independent hazards for kernel developers.

The first of these is to believe that the kernel coding standards do not matter and are not enforced. The truth of the matter is that adding new code to the kernel is very difficult if that code is not coded according to the standard; many developers will request that the code be reformatted before they will even review it. A code base as large as the kernel requires some uniformity of code to make it possible for developers to quickly understand any part of it. So there is no longer room for strangely-formatted code.

Occasionally, the kernel's coding style will run into conflict with an employer's mandated style. In such cases, the kernel's style will have to win before the code can be merged. Putting code into the kernel means giving up a degree of control in a number of ways - including control over how the code is formatted.

The other trap is to assume that code which is already in the kernel is urgently in need of coding style fixes. Developers may start to generate reformatting patches as a way of gaining familiarity with the process, or as a way of getting their name into the kernel changelogs - or both. But pure coding style fixes are seen as noise by the development community; they tend to get a chilly reception. So this type of patch is best avoided. It is natural to fix the style of a piece of code while working on it for other reasons, but coding style changes should not be made for their own sake.

The coding style document also should not be read as an absolute law which can never be transgressed. If there is a good reason to go against the style (a line which becomes far less readable if split to fit within the 80-column limit, for example), just do it.

* Abstraction layers

Computer Science professors teach students to make extensive use of abstraction layers in the name of flexibility and information hiding. Certainly the kernel makes extensive use of abstraction; no project involving several million lines of code could do otherwise and survive. But experience has shown that excessive or premature abstraction can be just as harmful as premature optimization. Abstraction should be used to the level required and no further.

At a simple level, consider a function which has an argument which is always passed as zero by all callers. One could retain that argument just in case somebody eventually needs to use the extra flexibility that it provides. By that time, though, chances are good that the code which implements this extra argument has been broken in some subtle way which was never noticed - because it has never been used. Or, when the need for extra flexibility arises, it does not do so in a way which matches the programmer's early expectation. Kernel developers will routinely submit patches to remove unused arguments; they should, in general, not be added in the first place.

Abstraction layers which hide access to hardware - often to allow the bulk of a driver to be used with multiple operating systems - are especially frowned upon. Such layers obscure the code and may impose a performance penalty; they do not belong in the Linux kernel.

On the other hand, if you find yourself copying significant amounts of code from another kernel subsystem, it is time to ask whether it would, in fact, make sense to pull out some of that code into a separate library or to implement that functionality at a higher level. There is no value in replicating the same code throughout the kernel.

* #ifdef and preprocessor use in general

The C preprocessor seems to present a powerful temptation to some C programmers, who see it as a way to efficiently encode a great deal of flexibility into a source file. But the preprocessor is not C, and heavy use of it results in code which is much harder for others to read and harder for the compiler to check for correctness. Heavy preprocessor use is almost always a sign of code which needs some cleanup work.

Conditional compilation with #ifdef is, indeed, a powerful feature, and it is used within the kernel. But there is little desire to see code which is sprinkled liberally with #ifdef blocks. As a general rule, #ifdef use should be confined to header files whenever possible.
Conditionally-compiled code can be confined to functions which, if the code is not to be present, simply become empty. The compiler will then quietly optimize out the call to the empty function. The result is far cleaner code which is easier to follow.

C preprocessor macros present a number of hazards, including possible multiple evaluation of expressions with side effects and no type safety. If you are tempted to define a macro, consider creating an inline function instead. The code which results will be the same, but inline functions are easier to read, do not evaluate their arguments multiple times, and allow

the compiler to perform type checking on the arguments and return value.

* Inline functions

Inline functions present a hazard of their own, though. Programmers can become enamored of the perceived efficiency inherent in avoiding a function call and fill a source file with inline functions. Those functions, however, can actually reduce performance. Since their code is replicated at each call site, they end up bloating the size of the compiled kernel. That, in turn, creates pressure on the processor's memory caches, which can slow execution dramatically. Inline functions, as a rule, should be quite small and relatively rare. The cost of a function call, after all, is not that high; the creation of large numbers of inline functions is a classic example of premature optimization.

In general, kernel programmers ignore cache effects at their peril. The classic time/space tradeoff taught in beginning data structures classes often does not apply to contemporary hardware. Space *is* time, in that a larger program will run slower than one which is more compact.

* Locking

In May, 2006, the "Devicescape" networking stack was, with great fanfare, released under the GPL and made available for inclusion in the mainline kernel. This donation was welcome news; support for wireless networking in Linux was considered substandard at best, and the Devicescape stack offered the promise of fixing that situation. Yet, this code did not actually make it into the mainline until June, 2007 (2.6.22). What happened?

This code showed a number of signs of having been developed behind corporate doors. But one large problem in particular was that it was not designed to work on multiprocessor systems. Before this networking stack (now called mac80211) could be merged, a locking scheme needed to be retrofitted onto it.

Once upon a time, Linux kernel code could be developed without thinking about the concurrency issues presented by multiprocessor systems. Now, however, this document is being written on a dual-core laptop. Even on single-processor systems, work being done to improve responsiveness will raise the level of concurrency within the kernel. The days when kernel code could be written without thinking about locking are long past.

Any resource (data structures, hardware registers, etc.) which could be accessed concurrently by more than one thread must be protected by a lock. New code should be written with this requirement in mind; retrofitting locking after the fact is a rather more difficult task. Kernel developers should take the time to understand the available locking primitives well enough to pick the right tool for the job. Code which shows a lack of attention to concurrency will have a difficult path into the mainline.

* Regressions

One final hazard worth mentioning is this: it can be tempting to make a change (which may bring big improvements) which causes something to break for existing users. This kind of change is called a "regression," and regressions have become most unwelcome in the mainline kernel. With few exceptions, changes which cause regressions will be backed out if the regression cannot be fixed in a timely manner. Far better to avoid the regression in the first place.

It is often argued that a regression can be justified if it causes things to work for more people than it creates problems for. Why not make a change if it brings new functionality to ten systems for each one it breaks? The best answer to this question was expressed by Linus in July, 2007:

> So we don't fix bugs by introducing new problems. That way lies madness, and nobody ever knows if you actually make any real progress at all. Is it two steps forwards, one step back, or one step forward and two steps back?

(http://lwn.net/Articles/243460/).

An especially unwelcome type of regression is any sort of change to the user-space ABI. Once an interface has been exported to user space, it must be supported indefinitely. This fact makes the creation of user-space interfaces particularly challenging: since they cannot be changed in incompatible ways, they must be done right the first time. For this reason, a great deal of thought, clear documentation, and wide review for user-space interfaces is always required.

4.2: CODE CHECKING TOOLS

For now, at least, the writing of error-free code remains an ideal that few of us can reach. What we can hope to do, though, is to catch and fix as many of those errors as possible before our code goes into the mainline kernel. To that end, the kernel developers have put together an impressive array of tools which can catch a wide variety of obscure problems in an automated way. Any problem caught by the computer is a problem which will not afflict a user later on, so it stands to reason that the automated tools should be used whenever possible.

The first step is simply to heed the warnings produced by the compiler. Contemporary versions of gcc can detect (and warn about) a large number of potential errors. Quite often, these warnings point to real problems. Code submitted for review should, as a rule, not produce any compiler warnings. When silencing warnings, take care to understand the real cause and try to avoid "fixes" which make the warning go away without addressing its cause.

Note that not all compiler warnings are enabled by default. Build the kernel with "make EXTRA_CFLAGS=-W" to get the full set.

The kernel provides several configuration options which turn on debugging features; most of these are found in the "kernel hacking" submenu. Several of these options should be turned on for any kernel used for development or

testing purposes. In particular, you should turn on:

- ENABLE_WARN_DEPRECATED, ENABLE_MUST_CHECK, and FRAME_WARN to get an extra set of warnings for problems like the use of deprecated interfaces or ignoring an important return value from a function. The output generated by these warnings can be verbose, but one need not worry about warnings from other parts of the kernel.

- DEBUG_OBJECTS will add code to track the lifetime of various objects created by the kernel and warn when things are done out of order. If you are adding a subsystem which creates (and exports) complex objects of its own, consider adding support for the object debugging infrastructure.

- DEBUG_SLAB can find a variety of memory allocation and use errors; it should be used on most development kernels.

- DEBUG_SPINLOCK, DEBUG_SPINLOCK_SLEEP, and DEBUG_MUTEXES will find a number of common locking errors.

There are quite a few other debugging options, some of which will be discussed below. Some of them have a significant performance impact and should not be used all of the time. But some time spent learning the available options will likely be paid back many times over in short order.

One of the heavier debugging tools is the locking checker, or "lockdep." This tool will track the acquisition and release of every lock (spinlock or mutex) in the system, the order in which locks are acquired relative to each other, the current interrupt environment, and more. It can then ensure that locks are always acquired in the same order, that the same interrupt assumptions apply in all situations, and so on. In other words, lockdep can find a number of scenarios in which the system could, on rare occasion, deadlock. This kind of problem can be painful (for both developers and users) in a deployed system; lockdep allows them to be found in an automated manner ahead of time. Code with any sort of non-trivial locking should be run with lockdep enabled before being submitted for inclusion.

As a diligent kernel programmer, you will, beyond doubt, check the return status of any operation (such as a memory allocation) which can fail. The fact of the matter, though, is that the resulting failure recovery paths are, probably, completely untested. Untested code tends to be broken code; you could be much more confident of your code if all those error-handling paths had been exercised a few times.

The kernel provides a fault injection framework which can do exactly that, especially where memory allocations are involved. With fault injection enabled, a configurable percentage of memory allocations will be made to fail; these failures can be restricted to a specific range of code. Running with fault injection enabled allows the programmer to see how the code responds when things go badly. See Documentation/fault-injection/fault-injection.text for more information on how to use this facility.

Other kinds of errors can be found with the "sparse" static analysis tool. With sparse, the programmer can be warned about confusion between

user-space and kernel-space addresses, mixture of big-endian and small-endian quantities, the passing of integer values where a set of bit flags is expected, and so on. Sparse must be installed separately (it can be found at http://www.kernel.org/pub/software/devel/sparse/ if your distributor does not package it); it can then be run on the code by adding "C=1" to your make command.

Other kinds of portability errors are best found by compiling your code for other architectures. If you do not happen to have an S/390 system or a Blackfin development board handy, you can still perform the compilation step. A large set of cross compilers for x86 systems can be found at

http://www.kernel.org/pub/tools/crosstool/

Some time spent installing and using these compilers will help avoid embarrassment later.

4.3: DOCUMENTATION

Documentation has often been more the exception than the rule with kernel development. Even so, adequate documentation will help to ease the merging of new code into the kernel, make life easier for other developers, and will be helpful for your users. In many cases, the addition of documentation has become essentially mandatory.

The first piece of documentation for any patch is its associated changelog. Log entries should describe the problem being solved, the form of the solution, the people who worked on the patch, any relevant effects on performance, and anything else that might be needed to understand the patch.

Any code which adds a new user-space interface - including new sysfs or /proc files - should include documentation of that interface which enables user-space developers to know what they are working with. See Documentation/ABI/README for a description of how this documentation should be formatted and what information needs to be provided.

The file Documentation/kernel-parameters.txt describes all of the kernel's boot-time parameters. Any patch which adds new parameters should add the appropriate entries to this file.

Any new configuration options must be accompanied by help text which clearly explains the options and when the user might want to select them.

Internal API information for many subsystems is documented by way of specially-formatted comments; these comments can be extracted and formatted in a number of ways by the "kernel-doc" script. If you are working within a subsystem which has kerneldoc comments, you should maintain them and add them, as appropriate, for externally-available functions. Even in areas which have not been so documented, there is no harm in adding kerneldoc comments for the future; indeed, this can be a useful activity for beginning kernel developers. The format of these comments, along with some information on how to create kerneldoc templates can be found in the file Documentation/kernel-doc-nano-HOWTO.txt.

Anybody who reads through a significant amount of existing kernel code will note that, often, comments are most notable by their absence. Once again, the expectations for new code are higher than they were in the past; merging uncommented code will be harder. That said, there is little desire for verbosely-commented code. The code should, itself, be readable, with comments explaining the more subtle aspects.

Certain things should always be commented. Uses of memory barriers should be accompanied by a line explaining why the barrier is necessary. The locking rules for data structures generally need to be explained somewhere. Major data structures need comprehensive documentation in general. Non-obvious dependencies between separate bits of code should be pointed out. Anything which might tempt a code janitor to make an incorrect "cleanup" needs a comment saying why it is done the way it is. And so on.

4.4: INTERNAL API CHANGES

The binary interface provided by the kernel to user space cannot be broken except under the most severe circumstances. The kernel's internal programming interfaces, instead, are highly fluid and can be changed when the need arises. If you find yourself having to work around a kernel API, or simply not using a specific functionality because it does not meet your needs, that may be a sign that the API needs to change. As a kernel developer, you are empowered to make such changes.

There are, of course, some catches. API changes can be made, but they need to be well justified. So any patch making an internal API change should be accompanied by a description of what the change is and why it is necessary. This kind of change should also be broken out into a separate patch, rather than buried within a larger patch.

The other catch is that a developer who changes an internal API is generally charged with the task of fixing any code within the kernel tree which is broken by the change. For a widely-used function, this duty can lead to literally hundreds or thousands of changes - many of which are likely to conflict with work being done by other developers. Needless to say, this can be a large job, so it is best to be sure that the justification is solid.

When making an incompatible API change, one should, whenever possible, ensure that code which has not been updated is caught by the compiler. This will help you to be sure that you have found all in-tree uses of that interface. It will also alert developers of out-of-tree code that there is a change that they need to respond to. Supporting out-of-tree code is not something that kernel developers need to be worried about, but we also do not have to make life harder for out-of-tree developers than it needs to be.