Chapter 6. Contributing to the Kernel Project (20)

Participation in the Linux kernel is essential. I'd like to say it is *required*. Your particular use of Linux may be intended for a commercial release or you may be looking to modify portions for a specific application within your own inhouse business or research projects, and the beauty of the GPL means any changes you may make to the kernel project are your own business. Still, if you are like most people who become involved in Linux, as time goes on, you will want to share what you've done to help advance the project. Such participation is the lifeblood of Linux. Participation can take many forms. You may start by joining mailing lists, discussion groups on USENET or on IRC to make contact with other developers working in your area. Later you may take an active role, contributing your own results with kernel patches or even taking ownership of a kernel subsystem. In all these instances, the kernel community has evolved some groundrules for participation. This section is a brief *HOWTO* guide to Linux participation.

Participating in Linux Development

Participation starts with choosing Linux and then choosing to become involved. Becoming involved means learning the technology and becoming involved in a community. If you have come this far, you already know about Linux Documentation Project and the kernel `linux/Documentation` directory; these collections are a goldmine of technical details. They are also a good reference of who's who in the kernel community.

Points of Contact

Most kernel developers are very approachable. The official list of developers and the subsystems under their wing is listed in the `MAINTAINERS` file in the root directory of the kernel source tree. Do keep in mind there are millions of Linux users, but only a few dozen maintainers; these people are the project managers and not technical support contacts for end-user problems or novice questions.For questions about the kernel sources, try to exhaust all other options before you ask directly. It may seem more expedient to send a fast email rather than sift twenty or thirty webpages, but consider what it is like to be on the other end of an email address which has been posted publically as an expert before an audience of millions. Linux has been around for many years, and each new revision is tested and probed by thousands of people. Chances are very good your question has been asked before. Your first stop should always be the kernel FAQ (http://www.tux.org), then the mailing list archives search page or Deja.com (http://www.deja.com), and only then should you approach the list or the contributors directly.Instead of directly approaching the top people, join the Linux Kernel mailing list at Tux.org (http://www.tux.org) and browse the archives to see who on the list is involved in your particular area of interest. Zach Brown's Kernel Notes (http://www.kernelnotes.org) publishes weekly summaries postings and a quick scan of his archives can usually identify someone who may answer your questions.

Mailing List Ettiquette The usual rules of nettiquette apply to the kernel development group; I don't need to tell you this. There are, however, other guidelines to keep in mind in your communications:

- Kernel programmers are busy on their own projects; they are not paid technical support people for Linux. Most are paid by their employer to get real work done on their company's agenda, not yours.
- If you absolutely need some feature added for your own use, be prepared to add it yourself. If you have a great idea with wide appeal, the developers may agree and change their focus to accommodate you, but they are under no obligation to do so. If you need the feature for a deliverable, be prepared to roll up your sleeves and contribute, or to arrange to hire someone on your own initiative. Kernel programmers are not hard to find (a pleasant side effect of an open source project) but, as a general rule, the kernel developers themselves are not available for hire.
- The `linux-kernel` mailing list is a very busy mailing list. There is already has far too much noise and its members, largely due to the above points, are not very tolerant of people adding more. Valid comments and ideas are more than welcome, but avoid “me too” replies or postings that stray from the charter of the group.

All these points of ettiquette aside, the kernel project values your input; a "kernel oops" report, even if the cause eludes you, is essential to ensuring the quality of Linux.

Joining the Inner Circle All *official* patches eventually find their way to Linus Torvalds before being folded into the master sources. Within each subsystem, developers will work on their own private source tree and collaborate with their core group and with the public to produce what they believe to be the best possible technical solution, but all their patches must go through the Torvalds gateway. As Linux advances, the work done enables participation by more people, and these in their turn enable even more advances and more participants. There is no absolute limit to the growth of Linux. To maintain sanity, a team structure has evolved over time to accommodate the flux of developers in and out of the project, and the ever increasing numbers of patches vying for inclusion in the kernel sources. While Torvalds still holds the keys to the source archive, there are now three or four trusted contributors who can submit patches which are almost automatically accepted. For example, promising candidate patches are often incorporated into the Alan Cox pre-release `-ac` patches before becoming part of the canonical source.Beyond this core group, the maintainers are the next level of access, and beyond them the subsystem developers. This hierarchy lets contributors work closely with the maintainer of a component, and the maintainers then work with Torvalds or one of his close assistants. As a result, Linux is able to beat the Frederick Brooks prognosis and manage a massive pool of developers. While entering the very top eschelon requires special talent, virtually anyone with the right mix of people skills, telework skills and programming savvy can become the maintainer for a kernel component.

Becoming a Maintainer Torvalds makes a distinction between the author of a component and its maintainer. Anyone may contribute a piece of code, but maintainer status requires a commitment to supporting the component, and a willingness to playing by team rules. Supporting a component involves fielding questions, patches and public input, and also involves following reasonable procedure in the quality assurance and updates of the software. Project rules require timely updates be submitted to Linus Torvalds via email; Torvalds *never* goes to a website or FTP archive to fetch patches. The kernel is also grounded in the philosophy of "release early, release often": All components must be submitted incrementally to allow for open peer review. Even where a component may have its own website for core developers to discuss and develop the code, updates should never be held back and released in large chunks. Frequent interim releases also ensures the component development does not move in directions counter to the kernel design goals.Among those design goals are code re-use and generalization. In the past, several components developed outside of the kernel source tree required fundamental changes when brought into the core sources; while any project is free to remain independent of the kernel sources, to be included into the kernel sources, developers must be prepared to work *with* the kernel project on a personal level as much as at a design or an API level.

Programming Style

Programming style is often discussed in the mailing list, typically with no real resolution. Some postings recommend emulating the style used in the module where you are working, but such advice really only ascerbates the situation. In Linux 1.2, Linus Torvalds gave his first call for common coding style, and his advice has been collected and expanded into `linux/Documentation/CodingStyle`. This document which begins: "First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture."

Coding Style Guide The basic advice is to follow the traditional K&R rules set forth at the dawn of C; formatting programs such as GNU **indent** can be used, but need commandline modifiers to conform to the kernel conventions. Put briefly, these conventions are

- 8 character tabs
- Place opening braces at the end of the line, not on a line by themselves
- An exception to the above rule is to place opening braces on a new line *only* when the brace begins a function block.
- Closing braces are on a line by themselves unless they are followed by a continuing clause such as with an else or with the while in a do-while block.

- Variable names should be descriptive while not becoming verbose; some C compilers are only sensitive to the first 32 characters of an identifier, and long variable names only rarely add useful information.
- Mixed-case variable names are frowned upon. Multi-word variable names should be seperated by underscores.
- Global variables must be used sparingly and should have very clearly unique and descriptive names.
- Local variables should be short, lowercase tokens. Loop counters have been called "i" since Fortran days and no one ever got hurt ... providing your functions do not grow out of control.
- Functions should be short and single-purpose; the source should fit into one or two screens of text. The best assurance of code correctness is having code structures which can be instantly comprehended. If your function requires more than 10 local variables, you may want to reconsider your design. GNU C can always inline your functions if you need to save the overhead of a function call.
- Comments are good. Everyone likes comments, but only if those comments explain what the code should be *doing* and why; code which renders the algorithm in natural language is only rarely useful. Use small comments to warn future developers, but otherwise avoid placing comments inside the function body; comments should be placed in a block above the function declaration.

Comment Blocks New to the 2.4 kernel, Tim Waugh and Alan Cox are promoting GNOME-like comment block conventions; Tim has also provided a script utility, **kernel-doc** to format these comments into DocBook, HTML, man pages and text; all kernel contributors are encouraged to annotate their code with these blocks so new developers can generate accurate API and design docs in much the same way that Perl and Java programmers use **perldoc** and **javadoc**.Comment blocks follow the a simple template:

```
/**
 * function_prototype() (- short description)  <10122>
 * @parameter_x: (description of parameter x) <10123>
 *  <10124>
 * Description: (Description of function)
 * section header: (section description) <10125>
 **/
```

<10122>These elements are optional

<10124>Function prototype and parameter descriptions are separated froom the description and section header with a blank line.

<10123><10125>These elements are may be repeated

The header template will be familiar to Java and GNOME programmers. Kernel-Doc will also replace special markup for environment variables, structure names and contants according to the following patterns:

`funcname()`	function
`$ENVVAR`	environmental variable
`&struct_name`	name of a structure (can include “struct”)
`@parameter`	name of a parameter
`%CONST`	name of a constant.

Details on the use of the **kernel-doc** script can be found in the section called *Using kernel-doc* in Chapter 5.

Using GNU Emacs

If you are using GNU Emacs or one of its variants, you can modify your c-mode support to follow the kernel guidelines, either globally for all your C language coding or to limit this style to just the kernel sources. To enable this within the scope of kernel files, you can add the following code to your `.emacs` file:

```
(defun linux-c-mode ()
   "C mode with adjusted defaults for use with the Linux kernel."
   (interactive)
   (c-mode)
   (c-set-style "K&amp;R")
   (setq c-basic-offset 8))

(setq auto-mode-alist
       (append ’(("/usr/src/linux.*/.*\\.[ch]$" . linux-c-mode))
   auto-mode-alist))
```

Using GNU indent

If you are using some other editor or if you get tangled about using Emacs, you can recover the proper formatting of your C language files with the **indent** command. In its basic mode, **indent** will backup your source files and produce new copies; **indent** can also be used to transform source files between different coding styles or to reformat your comment blocks (see **man indent**). For our purposes, the command to reformat a file into Linux kernel style is

```
indent -kr -i8
<sourcefile>
```

Asserts and Exceptions

As a general rule, avoid `assert()` statements; where they work, they are bloat, and where they don’t they are trouble. Besides, Linus does not like them.

Other Programming Guidelines

Your very best source on C language programming style for working with any highly-portable and large-team-effort project is to appeal to "K&R", the classic "Programming in C" by Kernighan and Richie, and a final word on Linux programming guidelines belongs to legendary open source programmer, Henry Spencer and his oft-cited Ten Commandments of C Programming (http://www.ee.ryerson.ca:8080/~elf/hack/The10.html).

Preparing a Patch

The patch should be co-measurate with the hole.

Thomas Jefferson

What do you do with your brilliant kernel hack? As noted above (the section called *Joining the Inner Circle*), final say on what is and is not included into the Linux kernel rests with Linus Torvalds, although as the size, complexity and popularity of Linux grows, Linus is shifting some of that burden to able contributors. Although the more subordinate roles do change, every module has a maintainer listed in the `linux/MAINTAINERS` file, and any patches intended for general distribution should go to the relevent people on that list.The ease with which the maintainers can apply your patch is directly related to the speed and likelihood of the patch being included. Patches should always be submitted as a *context diff*, the output from the **diff -c** command. A convenient way to generate these patches is to keep your local Linux sources in a versioning repository such as CVS and to generate the patch with the **cvs diff** command.

Bug Reporting

Bug reports are crucial Linux developers, but only when the reports include relevent information. Linux runs on many platforms and many configurations; simply saying that “SMP crashes” is about as useful as including the weather report. Context is everything, and developers bestow special blessings on those who can supply a short, repeatable example.Don’t let me disuade you from submitting bug reports on nebulous or sporadic problems. For many past bugs, the most obtuse observation has been critically useful. Still, where possible, provide as much information about your machine, the tasks it was doing and any other hardware or process information relevent to the bug. If the crash includes an “OOPS:” message, follow the procedure in the section called *Tracing from a Kernel OOPS* to create a more useful description of your machine and the state of the kernel when the problem occurred. If possible, create a test case which will reliably trip the OOPS condition; a repeatable example of a problem is worth far more than any amount of diagnostic data.

Kernel First-Aid

Before reporting any suspected bug, check the obvious causes.

- If you are using EGCS, a newer gcc or any other unofficial development tools, verify the bug using the recommended toolkit as described in Chapter 5.
- A frequent cause of trouble is a mismatch between the kernel and the supporting software itemized in `Documentation/Changes`; the script file `scripts/ver_linux` will generate a partial list of version numbers to check against the recommended list.
- Check symlink paths for `/usr/include/linux` and other relevant headerfiles, libraries and binaries to ensure you do not have version mis-matching or older versions masking out newer versions on the search path. If possible, verify the bug on another machine.

Reporting a Bug

Detailed reporting is very important; kernel developers want to help, and in most cases can help very quickly, providing they are given enough information to localize and identify the problem code. When preparing a bug report, include the following information:

1. One-line description of the problem
2. A more detailed description including any processes which were going on at the time, any events or odd hardware which may have triggered the bug and the sequence of events which lead to the bug.
3. If you are posting to the mailing list, include keywords to help identify the relevent maintainer, for example, “modules, networking, NFS”
4. The kernel version reported by `/proc/version` or reported when the kernel boots.
5. If the bug results in an OOPS, include the output with the symbolic information resolved (see the section called *Tracing from a Kernel OOPS*).
6. If you can repeat the problem conditions, include a shell script or the sequence of steps that will evoke the bug.

In addition to the bug synopsis, list a summary of your operating environment:

1. Software versions as reported by `scripts/ver_linux`
2. Output from `/proc/cpuinfo`
3. Active modules listed from `/proc/modules` or **lsmod**
4. Current hardware and driver information output from

- `/proc/ioports`
- `/proc/interrupts`
- `/proc/iomem`

5. Where relevent, include the PCI information from **lspci -vvv** (**lspcidrake** on Mandrake machines), SCSI details from `/proc/scsi/scsi` and any other `/proc` reports which may shed some light on the state of your machine.
6. Any other notes, patches or workarounds. If you can include the fix for this bug as a context-diff against the current kernel sources, expect a very warm welcome.

The completed bug report should be sent to the current maintainer for the relevent kernel component as listed in the `MAINTAINERS` file; bugs specific to any one module should be sent to the technical lead listed in that file, or, where the bug seems specific to a particular source file, reports can be sent to the author listed in the file header. As a last resort, or for bugs which appear to transcend module boundaries, reports can be sent to the Linux-kernel mailing list (http://www.tux.org/lkml) at <`linux-kernel@vger.rutgers.edu`>.

Tracing from a Kernel OOPS

The main trick is having 5 years of experience with those pesky oops messages

Linus

On those rare occasions where the kernel does fail, the dying gasp lets out a register dump. The addresses in this dump can be resolved against the current `/proc/ksyms` using the **ksymoops** program; because this utility does not depend on the version of the kernel, **ksymoops** is no longer bundled with the Linux source code and now lives with many other kernel debugging and reportint tools at the OCS FTP site (ftp://ftp.ocs.com.au/pub/ksymoops).Depending on the cause and severity of the OOPS condition, the register dump may have been read by `klogd` and recorded by `syslogd` in the `/var/log/messages` log. If `klogd` fails, the message may be available using **dmesg** or can be read directly from the kernel buffers using **cat /proc/kmsg**; be aware that the latter method is reading a continuously generated source and must be interrupted with Ctrl-C.If the kernel OOPS happens during the kernel boot or is otherwise unavailable using any of the above methods, you have three options:

1. Configure your system for a serial console and capture the output on a second machine.
2. Transcribe the message off the screen by hand.
3. Apply one of the non-standard “crash dump” patches such as `kmsgdump`, `lkcd` or `oops+smram` to enable saving kernel messages to a floppy disk, video RAM or the swap partition. (See the section

called *Analyzing Core Dumps* in Chapter 5

Given the output from **ksymoops**, the next step is to load the errant code into the debugger and to compare the disassembled binary with the register values given in the panic message to gain some insight into what should be happening vs what is actually happening. Running **gdb vmlinux** (ie, on the uncompressed kernel image), the gdb **disassemble** command will generate a source listing of the given function; the given offset into that function will pin-point the failing code.

In `Documentation/kernel-oops.txt`, Linus provides two tips for generating assembly code to compare with the code generated with the gdb **disassembly** command:

1. **make fs/buffer.s** (or whatever the errant file is) will stop the compilation after generating the GNU assembler. If the kernel bug report and the current kernel were created using the same development environment, this will provide a reference for comparison with the disassembled code.
2. The `Code:` values given in the panic message can be translated into assembler by creating a short C sourcefile, compiling it with **gcc -g** and running it through the gdb **disassemble**. Linus recommends an empty `main()` and a static `char str[]` set to the code values; he suggests using cut and paste to place the code numbers into the C source and replacing the spaces in the pasted string with `\x`:

   ```
   char str[] = "\xXX\xXX\xXX...";
   main(){}
   ```

LinusNow, if somebody gets the idea that this is time-consuming and requires some small amount of concentration, you’re right. Which is why I will mostly just ignore any panic reports that don’t have the symbol table info etc looked up: it simply gets too hard to look it up (I have some programs to search for specific patterns in the kernel code segment, and sometimes I have been able to look up those kinds of panics too, but that really requires pretty good knowledge of the kernel just to be able to pick out the right sequences etc..) *Sometimes* it happens that I just see the disassembled code sequence from the panic, and I know immediately where it’s coming from. That’s when I get worried that I’ve been doing this for too long ;-)