Chapter 5. Tools for Kernel Programmers

The kernel programing kitbox starts with the C compiler, but writing and compiling code is only part of the story. If you want to code something relatively simple, and need to get up and running quickly, there are prototyping RAD tools, and to code effectively, you will want debugging, trace and profiling tools to ensure your work does what you expect. Finally, there is no magic bullet to comprehending where your modifications should fit into Linux architecture, but there are some very good reverse engineering tools to help you “call before you dig”.

> **Note:** This overview of programming tools, like most discussions on Linux, charts a moving target. New tools and updates are added almost daily; if your needs go beyond what is outlined here, do a fresh hunt through one of the Linux kernel support sites.

Because it is essentially an embedded application, working with the kernel means abandoning many comfort tools of the programmer’s toolkit. You cannot use standard libc functions or logging tools; you have only what is provided by the compiler and what is included in the kernel. Fortunately, there are other ways to get inside the box: Through careful analysis of the source files and, with minor restrictions, the use of debuggers and profilers, you can get some inside understanding of what your code is doing.

GCC, EGCS and GNU Assembler

The foundation of kernel programming is the GNU Compiler Collection (GCC). Linux is defined by the the interpretations of the C standard as they appear in GCC; as a general rule, Linux is portable to any platform where the Free Software Foundation has ported the GNU compiler. There are exceptions, but this is the official relationship between Linux and GNU: All patches and all bug reports are verified through the official, Linux-approved version of the GCC.

Linux programming may begin with GCC, but not just any GCC: At the time of writing, `Documentation/Changes` lists version 2.7.2.3 as the only compiler where the kernel is *expected* to compile. The sources are not neccessarily incompatible with other versions or even with other C compilers, but before you can report suspected kernel or compiler bugs, you must first verify your results using the 2.7.2.3 GNU compiler.

> **Note:** Many core developers will test the kernel build with proprietary compilers (for example, the IRIX and Solaris C compilers) and with the Experimental GNU Compiler System (EGCS), and each build will be tested in different configurations and platforms; unless you find problems specifically mentioned in the KernelNotes or kernel-list archives, the chances for success with another ANSI standard C compiler are reasonably good.

Linux 2.4 and egcs

Why not EGCS? After all, most Linux distributions released after the summer of 1999 bundle the EGCS version of GNU C. This is frequently debated in kernel circles. Yes, EGCS is the “official” GNU C, but no, it is not the same as GCC. The key word in “EGCS” is “Experimental”.

Some explaination: After the release of gcc2 in 1997, the Free Software Foundation halted further development of the compiler and the EGCS group was formed to carry the project forward, this time as a true *open source* initiative rather than as the cloistered workgroup model used for the earlier versions. In late April 1999, the Free Software Foundation formally appointed EGCS as official maintainers of the compiler. As a result, EGCS, because it is “new technology”, is the most widely distributed flavour of the GNU C compiler in modern distributions. It is still "new technology" and the merger of the old version 2 GCC code and the newer EGCS code is not yet complete. There are subtle differences.

Strange things can happen with egcs, often very subtle and hard to debug things. The most significant difference between these compilers is in optimization, especially with inline code. Prior to Linux 2.2, the differences were stark: The more aggressive optimizations on the Intel version caused wildly incorrect compilations. Since the primary goal of the 2.0/2.2 kernels was stability and EGCS presented too many unknowns, the kernel group voted to stay with 2.7.2.3 until these issues could be resolved.

This does not mean you *cannot* use the newer compiler. Many problematic sections have been rewritten and the most gross EGCS complications have vanished. Still, absence of evidence is not the same as evidence of absence: As recently as 2.3.18, optimization problems again pointed fingers at EGCS for miscompilations of deeply nested macros. It is like the dentist who tells patients they need not floss *all* their teeth, only those they wish to keep: You can use EGCS, but if you require stability, you reduce your unknowns with the official, albeit less clever compiler.

Using GNU Assembler

If you feel the urge to use assembler, get some friends to talk you out of it. Most tasks are completely practical using pure C, and much of the assembler code in the Linux kernel has migrated to C to ease portability (or EGCS conflict) issues. If you absolutely must have close coupling with your hardware and assembler is your only option, then, just as with the C compiler, the official assembler is the GNU Assembler (GAS) in the *binutils* package. Once again, `Documentation/Changes` lists the official and minimum version as *binutils-2.9.1.0.7*; other assemblers may be possible, and many are mentioned in the Assembly HOWTO, but the same caveat applies: Before you can report a bug, you must verify your results with the official toolset.

Assembler can be included inline in the C sources or in stand-alone files; both use the same assembler, in one case invoked by **gcc**, the other by **make** rules. The results and the rules are the same. Detailed information on mixing inline-assembler with C can be found in the GCC info pages under C Extensions::→Extended Asm:: with details about Intel-platform conventions under Invoking GCC::→Submodel Options::→i386 Options::.

Inline Assembler and Optimization

Be very careful with compiler optimization flags when using inline assembly language. GCC will not inline assembler functions unless optimization `-O` or `-On` flags are added; without optimization, the link stage will hang looking for unresolved references to any assembly routines that may have been embedded in extern inline functions. You can mix `-g` and `-O` flags for debugging, but you must include optimization for correct inline compilation.

If you know any other assembler, you will find few surprises in GAS. GNU assembler uses standard AT&T syntax, a little odd to Intel users, but completely familiar to those accustomed to the m68k assembler.

Assembler Caveats

The Assembler-HOWTO offers the following checklist on using GNU assembler:

- Register names are always prefixed with the percent (%) sign, as in `%eax`; this makes it possible to mix external C symbols without introducing underscore notation.
- The order of operands is *reversed* from the Intel convention; **gas** expects operand pairs as (source, destination)
- Operand datatypes can be disambiguated by appending a single-letter type identifier to the name of the operator, for example,

```
movw %ax,
  %dx
```

would move a 16-bit (word) value; the default behaviour will deduce the word-size from the operands or use 32-bit (long) values.

- Immediate operands are identified by a $ prefix
- Where there is no prefix, variable identifiers are taken as memory references, for example `$var` is the address of `var` whereas just `var` denotes the value stored at that location.
- Parenthesis around a variable name denote indirection or indexing, for example `17(%ebp)` is an offset of 17 bytes from the address stored in the base pointer.

Intel2GAS Assembler Translator

In some instances, assembler code from other platforms and from other assemblers can be machine translated to GAS format using Mikko Tiihonen's Intel2GAS (ttp://hermes.terminal.at/intel2gas/)

translator. Originally designed to translate NASM files to the GNU Assembler, this project has evolved into a general purpose system to translate several ix86 assemblers into GAS and can be scripted to translate between almost any two assembler syntax conventions.

Navigational Aids

You may understand the compiler and have a good notion what your code must accomplish. Next, you need to know where and how it will fit into the kernel architecture. Simply adding a new kernel device module may have an obvious best-location, but if your project spans module or subsystem boundaries, you need the big picture. Even if you are only extending existing code or adding a new minor service, without a current and accurate map, you run a risk of re-inventing services which may have been available had you chosen some other location for your code.

Any project of 1.6 million lines is going to have some tangles no matter how you slice it. Telling you to "start in `init/main.c`" is not much help, and even a printed reference such as this book is at best only a snapshot conceptual map through the labyrinth; the developers will emplore you to "just read the source" but in a project this size, finding your way around is a non-trivial problem.

Fortunately, several excellent open source packages exist for reverse-engineering large projects. There are also many commercial toolkits but start you thinking about architecture, I don't want to invite the excuse that you cannot afford them; this section will focus only on those tools which are freely available, with a preference for open source.

Code Visualization

Code visualization is not about pretty printouts of control structures in a source file. It is useful to have such a display for large and complex functions, but such code is not very common in the kernel sources (see StyleGuide). In our everyday life, we expect streets to be straight with periodic intersections and rows of driveways on either side; we don't buy maps to place our mailbox in front of the house, but we do buy maps and surveys to see the neighbourhood and we use architectural plans to understand any large construction project. Even if those plans are not accurate (they rarely are) we can work out the differences as we explore. This is the purpose of code visualization.

Unfortunately, most open source projects have no functional specifications and no architectural plans. At best, there are some loose guiding principles as decided by the steering committees. Linux is no different. Contrary to the best advice of software engineering, most open source projects are invented on the fly, made up as project members encounter each new design problem. Open source teams move organically with no master plan, and overall, the method works *very* well ... unless you arrived late.

I’m not advocating design process be thrust upon open source; not only does the open source “massively parallel solution space random walk” work, but Apache, Linux and many other large projects prove that it works very well, as good or better than the traditional software methodologies. I’m only suggesting that, for any non-trivial changes, we need to understand the *current* architecture, and this is especially important in a fast-moving project like Linux. Due to the great popularity of the O/S, we have limited contact with core developers, and a rapid pace of kernel development renders existing documents obsolete faster than they are written. Even for the seasoned team member, the escalating pace and scope of Linux development ensures, whatever conceptual model they may have had when they started, the reality is probably very different, and there is no one we can “call before we dig”: *On-the-fly* reverse engineering techniques are practically essential.

Reverse engineering is the process of deriving a map directly from the territory using an automated system to gather relational data, and a graph editor to help distill the jumble of call graphs and misinformation of directory structures into comprehensible interactions of well-defined subsystems. While common in object-oriented programming, the practice is relatively unknown in the open source community: Most projects have processes for moving conceptual consensus into source code, but there are virtually no reverse repair practices ensuring documents match the source reality.

Rigi Visual Software Understanding Tool

Rigi is a code visualization suite from the University of Victoria in British Columbia. The package has been ported to Linux and also runs on Sun SPARCstations (SunOS), IBM RISC System 6000 (AIX) workstations, and on Windows 95/NT. Rigi can extract and abstract structural information from the source code and display results in a variety of high-level views; the system is bundled with parsers for C, C++ and COBOL (a Y2K-boom legacy?) and can be extended using the Tcl interface.

Rigi includes two main components, a command-line source parsing **rigiparse** for creating the Rigi Standard Format (RSF) data tupples, and the Motif-based **regiedit** diagram editor for the graph manipulations, analysis and report generation. The kit also includes tools for massaging raw data produced by the parser to remove redundant information and for generating hypertext displays

The **regiedit** display window initially shows the project as an inverted tree where leaf nodes correspond to functions and structures, and intermediate nodes are assigned as preliminary subsystem boundaries. Selecting a subsystem will show contained elements with the call and reference arcs. You then use the editor to alter this projection and re-group nodes to disclose the underlying structure of the software.

While it is easy enough to manipulate source code graphs using **rigiedit**, populating the graph with the relational data gleened from source code is not so straightforward. For an application the size and complexity of Linux, this requires considerable massaging of data before you can get a meaningful result; the built-in RCL scripting language allows you to automate much of the process, but creating a useful map of the entire Linux kernel will require some effort and experiment.

To prepare the database for the editor, the basic procedure is

1. Create a shell script wrapper around **gcc** to substitute the preprocessor output and for all the compiler invokations in the Linux make files.
2. Recompile Linux setting the `CC` to your script wrapper. This will create corresponding **regiparse** output files for every C-language file.
3. After compilation, the results can be collected together, sorted and filtered, or processed into hypertext information (for a hypertext navigation)
4. Once loaded into the editor, the built-in scripting system can also be used to filter the dataset for redundant or misleading links before displaying the project graph.

The process does not require changes to the Linux source tree and will only map those options which were selected in the kernel `.config` file.

It is worth the effort. Once loaded into the graph editor, you have a very powerful tool for distilling the source files into manageable bits. For example, you can generate forward dependency maps out from `main()` or any other function to see the call or reference scope of that item.

A very useful feature of Rigi is the ability to reframe any data structure into an object oriented view: The structure is portrayed as the attributes of a virtual class with the functions involving this structure as its methods. The whole virtual class definition is grouped as a single subsystem and folding this back into the overall graph will allow you to project higher-order forward dependency and interactions maps. This process can be repeated to reduce the entire kernel into the subsystems listed in Part II.

For your documentation and presentation purposes, Rigi will also produce graph view screen dumps which can be massaged through postscript tools such as **gv** or **xv** or included in DocBook files; Rigi's output may be somewhat rustic, but the program more than compensates for aesthetics by being a solid piece of technical wizardry.

The package also includes scripted demos of the abstraction features using a simple application, a ray tracer and a portion of the IBM SQL/DS sources. The latter is a dramatic demonstration of distilling a nightmarish soup of a flat call graph into an elegant and instantly comprehensible display. Other examples of applying Rigi to very large projects and a collection of research papers can be found on the Rigi Homepage (http://www.rigi.csc.uvic.ca) at the University of Victoria.

PBS: The Portable Bookshelf

Conceptually and functionally similar to Rigi, the Portable Book Shelf (PBS) from Ric Holt's group at the University of Waterloo is another reverse-engineering Swiss Army Knife for gathering and presenting structural information from a large software project. PBS has also been ported to Solaris, Linux and NT and can be obtained from the PBS Homepage (http://www.turing.toronto.edu/pbs/index.html) at the University of Toronto; this site also sports a pre-built demo (and downloadable datasets) using the 2.0 Linux kernel as well as many excellent software architecture research documents describing PBS as specifically applied to Linux.

(sidebar) Do we *need* reverse engineering?

Do we need elaborate schemes to tell us what is already outlined in other kernel documentation without the benefit of complicated tools like Rigi or PBS? Yes, other documentation does divide the kernel into these same subsystems, and we even depended on that knowledge in distilling our concrete model, but what is most interesting is where we differ: Our *empirical* model shows many interactions and dependencies not reported in other documents or even in the source code comments. By appealing to reality, we discover non-obvious (or forgotten) interactions such as the kmod dependencies between the network subsystem and IPC.

This should not imply our empirical view is any less prone to omissions or errors; our tools can and do produce artifacts which must be verified in the source code logic, and because these tools work on the kernel being built (using our current `.config` file) we cannot be certain other configurations will not include other unseen relationships. What we can do is gain a more accurate map of what is known and to illustrate Richard Feynman's famous remark ""...it doesn't matter how beautiful your theory is, it doesn't matter how smart you are – if it doesn't agree with experiment, it's wrong."

Source Code Navigation

Once you understand kernel structure, and while you do your actual coding, you need to manage that tangle of symbols and function calls. Souce code navigation tools give you various methods for hypertext views of source files; you need to know where you are going, but for stepping through static source structures, these products are essential

These tools can be divided into two camps, the HTML browser-based systems and the ctags editor-extension systems. Each method has its advantages and disadvantages, but all have the common objective of point-and-click navigation through cross-references between many source files spread over many directories.

The Linux Source Navigator

One of the first systems to render Linux in hypertext is a lone Metalab UNC webpage called simply "the Linux Source Navigator". There are no mentions of authorship, available source code or any explanation why this site ceased to follow Linux at some unspecified incarnation of the 2.0 kernel; the only things I can say about the the Linux Source Navigator (http://metalab.unc.edu/navigator-bin/navigator.cgi?) are that it is fast and that it probably also served as an inspiration for more useful projects such as LXR.

LXR: The Linux Cross-Reference project

The Source Navigator was the seed of an idea to use HTML as a means to present Linux source code. LXR takes this further, wrapping the idea into a useful and elegant package for presenting multiple versions and multiple branches of a large project.

LXR is a suite of Perl5 scripts which, when coupled with a search engine such as Glimpse or Ht/Dig, lets you navigate kernel sources by searching directories, searching on keywords, identifiers, strings and perl regular expressions (regexps), and even just plain old hyperlink browsing. The display renders C code with all project identifiers and function names as HTML anchors linking to the definitions or to cross reference lists on other uses of the symbol.

If you have a reasonably fast Internet connection, you can enjoy the benefits of LXR through the LXR Homepage (http://lxr.linux.no/). This site sports a full demonstration of the software, complete with Glimpse-search capabilities, and shows the very latest Linux sources for both the production and the development kernels. Other demos at the LXR site include source browsing of the GIMP, BSD and several other large-scale open source projects.

Installing the LXR

If, on the other hand, you need rapid and/or local access to the display, a full installation of the LXR search engine is simple to do, but the installation is large: The identifier indexes alone for the 2.3 Linux kernel total 30Mb and take nearly a half hour to generate on a Celeron 300 PC. The LXR documents are sparse, but not impossible to follow if taken completely literally. An overview of the installation goes as follows:

1. Install the LXR web files in your webserver document tree and set the webserver config to recognize the CGI programs
2. create the index directories outside of the document tree, typically in `/usr/local/lxr` and create index subdirectories for each kernel version to be indexed; these kernel versions must also be listed in the `versions` text file in the base directory of the indexes
3. change to the version-specific index directory and run the genxref program giving the full path of the kernel source tree as the target of the command.

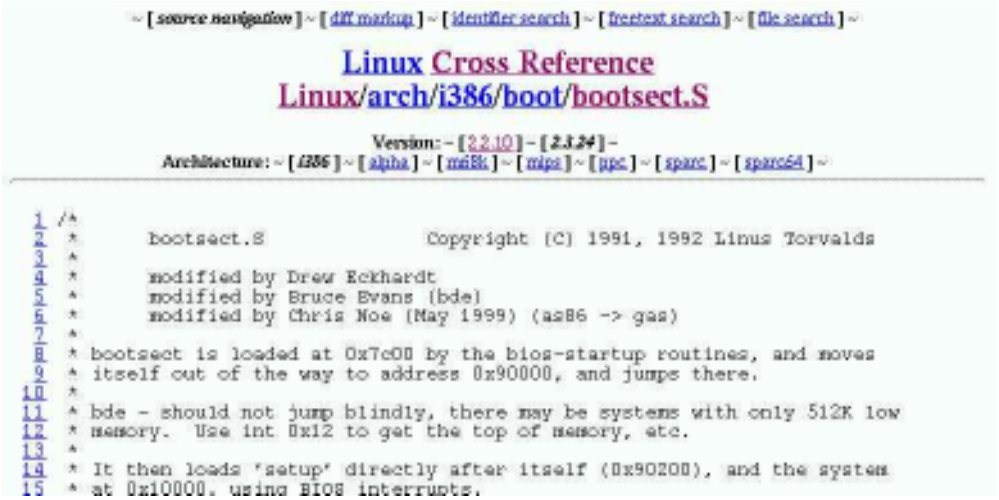
This installs the basic system to browse the source code and switch between alternate versions, plus it will let you do basic identifier lookups and to navigate the hyperlinks LXR will embed into the source-code display

The next-level of LXR use requires a search engine, and for this purpose, the LXR documents recommend the Glimpse search engine, but they concede that any text-based search engine will do just as well; the LXR config allows for any CGI program to be called in response to the full-text search form[1]. Given this backend, the system is able to answer just about any pattern-search query on the kernel in a matter of seconds.

Using LXR

There are no surprises in using LXR. The screen display shows the current page and lists all installed kernel versions and architectures (see Figure 5-1). As you would expect, while viewing a file for one version or architecture, selecting the option for an alternate source tree will display the corresponding file (if available) in that new context. This makes LXR an indispensable tool for making source comparisons, for example, when adapting instructions for previous kernels to the situation in a more modern source tree.

Figure 5-1. Screenshot of an LXR page



In-Editor Source Browsing

A source browser such as LXR may be useful for large sweeps of your sources or for identifier or comment searching via the ad-hoc perl regexps, but it is read-only and browser based, and not very practical during your hands-on coding. To integrate source navigation into the practice of writing software, many programmers already use code navigation tools such as **ctags** in vi or **etags** in EMACS but these tools are not quite up to indexing tags lists as large and complex as those of the Linux kernel.

Global Source Tags

You can think of Shigio Yamaguchi's Global as ctags on steroids: Global is a portable source code tag system for C, YACC and Java source files and can be integrated into vi or Emacs, or used with a

stand-alone hypertext generator to quickly locate identifiers, object references and even preprocessor symbols.

Global tags may be invoked within your editor or from the command line. Its features include

- Reference scope can be limited to directories allowing duplicate identifiers within different source areas.
- Search space can include library paths
- Tag files are platform independent and can be moved or shared freely between different architectures (for example, tag files may be shared via NFS)
- An optional compact format can be used to reduce diskspace requirements

Obtaining Global

This is one of those gem applications where development has all but ceased only because the current version is about as good as it gets. The most recent version dates from 1996 and is available at the Global Homepage (http://wafu.netgate.net/tama/unix/global.html); it is also available through large toolkit sites such as MetaLab.

Building Kernel GTags

The advanced uses of the Global system for generating HTML output, leveraging the Extended VI and EMACS specific features or extending the parsers are beyond the scope of this chapter; the documents which accompany Global are unusually complete and well written. All you need know here is, to the end user, preparing a complete `GTAGS` index for even a large project such as Linux is no more complicated than the traditional `CTAGS`; all the complexity of the Global btree indexing is completely and delightfully hidden.

The command-line use of Global is similarly straight forward and with no surprises. Using the included **global** utility from a shell, you can quickly locate identifiers and references ... try this the next time a kernel patch leaves you without some symbol defined! It gets even better for the Kernel developer: By creating separate indexes in each of your `linux-VERSION` directories and a third index in the parent `/usr/src` directory, searching done in the context of one version will be limited to that scope while a search done from the root directory will locate all references or definitions in all your installed versions.

There’s more good news: Global does not require a complete rebuild of the database every time there is a change to a few files. There is an initial hit to build your tags index, but thereafter, tag files can be updated incrementally by changing to the parent directory of the changed or patched files and issuing the `-i` command option. Solidly cool.

XRef-Speller for EMACS

Similar in intent to Global under the EMACS editors, XRef-Speller also provides in-editor tracing of identifiers and function calls in C, C++ and Java source code. XRef extends this basic lookup function to include other interesting features such as the ability to highlight a segment of code within a function and to auto-generate a new function to factor out that code, or to do the opposite and adjust a function to be inserted into some other function. Like the Source Code Browser, XRef can also be used to generate hypertext (HTML) editions of source code files to cross reference symbols to the definitions and other uses; their website includes a demo of this using the 2.2.5 Linux sources (http://guma.ii.fmph.uniba.sk/linux.html).

Obtaining XRef

XRef is distributed through the XRef-Tech Homepage (ttp://www.xref-tech.com/) and while demonstration versions are available for download, XRef is commercial software and is licenced either in binary or in source-code form.

Using kernel-doc

Understanding a large program requires grasping the high-level design and navigating the token zoo, but it also requires understanding the source code itself and the intentions of the developers. While not followed throughout the Linux kernel (it wouldn't be any fun if there were rules), many kernel developers, and everyone reading this book, are adopting the kernel-doc conventions for embedded comments; Tim Waugh and Michael Zucchi have contributed a perl script, `scripts/kernel-doc`, to parse these comments into DocBook, HTML, man page and text formats. This utility will also let you single out specific functions in the C code to be included or excluded.

parport_set_timeout illustrates the man-page like output returned from the **kernel-doc** command **perl scripts/kernel-doc -docbook -function parport_set_timeout drivers/parport/ieee1284.c**.

The **kernel-doc** is the core application behind the new `Documentation/DocBook` project headed by Tim Waugh and Alan Cox. This directory uses a simple template system to insert kernel-doc generated reference pages into DocBook sources. A few "Kernel Books" can be found in the `Documentation/DocBook` directory where there is also a `Makefile` for generating PDF and Postscript output

> **Note:** Programmer guidelines for coding kernel-doc comment blocks can be found in the section called *Comment Blocks* in Chapter 6.

parport_set_timeout

Name `parport_set_timeout` — set the inactivity timeout for a device

Synopsis

```
long parport_set_timeout (struct pardevice * dev, long inactivity);
```

Arguments

`dev`

device on a port

`inactivity`

inactivity timeout (in jiffies)

Description

This sets the inactivity timeout for a particular device on a port. This affects functions like `parport_wait_peripheral`. The special value 0 means not to call `schedule` while dealing with this device.

The return value is the previous inactivity timeout.

Any callers of `parport_wait_event` for this device are woken up.

Description

This sets the inactivity timeout for a particular device on a port. This affects functions like `parport_wait_peripheral`. The special value 0 means not to call `schedule` while dealing with this device.

The return value is the previous inactivity timeout.

Any callers of `parport_wait_event` for this device are woken up.

Development Tools

In a perfect world, you have all the time you need to carefully craft blemish-free software with a *carte blanche* budget. I don't know about your world, but in mine, my clients want things yesterday, and they are more likely to approve a project budget after they can see something tangible, only rarely before. Some readers may be raptured in a labour of love working on kernel code, but others may need that extra boost of development tools. Fortunately, there is help.

Development tools can be divided into three domains: RAD tools, performance evaluation, and proofing simulators. The first get you coding quickly, the second let you verify and probe your program, and the last group give you a test-harness that avoids the bother and risk of continually rebooting and lets you observe your code on almost any interface instead of crossing your fingers and inflicting it on a live machine.

Linux offers the most in the middle category, in software probes and debugging tools; there is a simple RAD kit for developing new driver modules, but there is only one. There is also only one 'sandbox' application to run your kernel in userspace. You can probably tell from these counts where the average programmer spends most of their time, and it is not in either initial coding or in the final proofing of their code.

Rapid Application Development in Kernel Programming

A kernel RAD tool needs to capture knowledge of (or extract) current kernel APIs, structures and function calls. It would know what is valid and in what sequence, and it would build source and document templates for common design patterns. In actual practice, however, you are not likely to create many new kernel services in your kernel programming career, and you are much more likely to spend most of your time using source code navigation tools such as Global. This probably explains why there are so few (exactly one) kernel development kits: There is no demand. It may also simply be an application who's time has not yet come.

Linux Driver Development Kit

Let's suppose you have some new piece of hardware which will sit on some I/O or network port awaiting some handshake sequence of events; this is the scenario of the Linux Driver Development Kit. A project of Linux Labs, the LDDK simplifies creating device drivers by generating the basic module source code from a a driver description file and corresponding code-templates.

The toolkit currently contains only one program, the command-line based compiler for their Driver Definition Language (DDL); other tools such as a GUI and a language editor are planned, but at the time of writing, the recommended path is to just use SWIG (see below) to add widget access to the compiler. Given your hand-crafted DDL file, the **ddl2c** compiler will generate a complete driver project source tree, complete with all your makefiles, libraries and online documentation, all ready to compile into a

new kernel module.

While limited, the LDDK is still very useful for creating (or just bootstrapping) global devices (single major number) or simple submodules within a single hardware device (minor numbered devices). The generated files automatically include `/proc` pseudo-file reporting, `/proc/sys` Sysctl I/O, **ioctl()** functions and the user libraries. Generated drivers also include definitions for multi-level debugging code which can be manipulated via **insmod** command line parameters.

The LDDK package includes example code for simple and submodule drivers, block devices, memory mapping, `/proc` functions and a guide to passing signals between the kernel and user applications. The full documentation and download information for the LDDK is available from the Linux Labs Software Page (http://www.llp.fu-berlin.de/pool/software/dutil/); Tcl/Tk, Perl, Phyton or Guile support for your can also be added through the SWIG wrapper generator from http://www.cs.utah.edu/~beazley/SWIG/ (http://www.cs.utah.edu/~beazley/SWIG/) or from the Linux Lab Project.

Debugging and Kernel Probes

> Linus Torvalds
>
> "I’m afraid that I’ve seen too many people fix bugs by looking at debugger output, and that almost inevitably leads to fixing the symptoms rather than the underlying problems."

Torvalds may be right, but you can also have bugs which cannot be discovered through static analysis of the source code. Very often, especially in real-time or multithreaded applications, the only way to gather information about system behaviour is to watch it. Put another way, software is never under any obligation to behave the way it was intended to behave.

Working on the operating system itself presents some obvious problems for debugging. Because the kernel is what provides the basic I/O services for all applications software, we are not able to stand outside of its space in the same way we can stand outside of another application; outside of the kernel, there is no place to stand. Fortunately, this is not precisely true: We are able to gain some knowledge about the running kernel through the kernel’s own I/O services such as **printk()** or **sysctl()**, or we can indeed stand outside the kernel by standing in the I/O space of some other kernel either on the same machine or on some other box.

When printk() fails

Simple diagnostics and tracing can usually be done by peppering suspect code with the kernel library **printk()** function. As expected, **printk()** provides a 'printf'-like facility to log messages through **syslogd**. Where it works, and where calling the function is allowed, this technique works quite well, but as we all know, like any instrumentation, print statements inserted into code do not monitor the code you wrote, they measure new code with print statements inserted into it. The difference can lead to code which may work when traced, but cease to work when the statements are removed.

The same warning holds true for the use of **assert()** macros to add pre- and post-conditions to your functions: While wonderful in theory, when these macros are enabled they can fundamentally change your program. I have seen cases where inserting **assert()** shifted stack addresses enough to prevent the bug from occuring and could as easily caused new instabilities to appear.

Sometimes, the only way to understand why our code behaves the way it does is to strap it to the lab bench and stuff it full of probes and monitors. This section describes some methods in all of these scenarios, from the very low-level probes to ways to enable full symbolic debugging and port taps on a running kernel image.

Using ptrace

Unless you are creating custom debugging tools or extending the facilities of the existing tools, you are unlikely to work directly with this function. **ptrace()** is the standard unix basis of all breakpoint debugging and is used to manipulate the runtime state of a process through the protocol of signals defined in the header files. For example, a controlling process can attach another using the `PTRACE_ATTACH` request and initiate a single instruction step with `PTRACE_SINGLESTEP` or manipulate data within the child process or change the register values within a stopped process.

Most Linux distributions include man-page documentation for the ptrace system call, but for the 2.4 kernel support, you will need to find the most recent edition of this page. `<sys/ptrace.h>>` defines the interface to all the process trace and manipulations functions located in `Linux/arch/ARCH/kernel/ptrace.c`.

Subterfugue (http://subterfugue.org/) is a related open source project which provides am object-oriented scriptable interface to `ptrace()` using extensible Python scripts. This can be used to do **strace**-line monitoring kernel communications of user-space programs or to wrap an application in a secure sandbox. Subterfugue is able to provide services unable to the usual **strace** utility, for example, tracing through clone system calls and following job control.

Debugging over the Serial Console

At long last, there is a good excuse to dust off that VT52 in the closet: New to the 2.4 kernel, a serial console can be enabled to monitor the system messages and manipulate the system using a simple terminal or terminal emulator program. With the serial console enabled, you can also issue `SysRQ` commands: Sending a **BREAK** followed by enter will enable the serial SysRQ and print a short command summary; to execute a SysRQ command, send **BREAK** again followed by the command key.

The serial console SysRQ facilities are also available as a patch for the 2.2 series kernal from Miquel van Smoorenburg's FTP site. (ftp://ftp.cistron.nl/pub/people/miquels/kernel/v2.2/)

Using the GNU Symbolic Debugger (gdb)

Rather than manipulating the kernel directly through **ptrace()**, if you must monitor or correct a running kernel image, you will probably do this using **gdb**. The good news is the GNU symbolic debugger can easily monitor a running Linux kernel; the bad news is the access is read-only; you cannot set breakpoints or modify values.

To prep the kernel for debugging,

1. Add debugging symbols to your object files by manually inserting the **-g** compiler option into `arch/i368/Makefile`.
2. Enable the `/proc` filesystem in the kernel options; this will be your window into the kernel.
3. Issue a **make clean** and then rebuild and install the new kernel image

With these changes, the new kernel can be monitored through the `/proc/kcore` memory image pseudo-file using the command

```
gdb vmlinux -core /proc/kcore
```

If you need more control over the runtime state or you need to monitor a kernel that is going to crash and burn (taking your debugger with it), **gdb** can be run from a second machine and connected to your test box through a serial port; to facilitate this connection and to enable you to attach to the kernel as early in the boot process as possible, Silicon Graphics offers a kernel patch which adds the gdb option to the **LILO** prompt; this patch will also add the option to define `CONFIG_GDB` in the kernel config so the feature can be turned off when no longer required. The SGI kGDB patch can be downloaded from the kGDB Project Homepage (http://oss.sgi.com/projects/kgdb/).

Built-in Kernel Debugger

SGI people just don’t know when to quit. If your kernel is among the list of supported versions, they also offer the KDB patch (see oss.sgi.com (http://oss.sgi.com/projects/kdb/)) which will add configuration options for `CONFIG_KDB` and `CONFIG_KDB_FRAMEPTR`. When enabled, these options provide symbolic debugging services directly in the kernel, without the need for a remote **gdb** process; the KDB patch kit also includes a `Documentation/kdb` directory with copious release and usage information.

Kernel Sandboxes

What if your changes may damage your hardware, or involve handshake sequences you would like to test through a regression test suite and a device stub? If your changes kernel fit within the capabilities of a simulator, it may make sense to test your code a user-space virtual machine, or ’sandbox’, to observe its behaviour. In many cases, the same tests can be done through a debugger or even by simply tracing loading and unloading your kernel module, but in some circumstances, you may want to verify your code before you inflict it on a real machine and to proof your new kernel in a nice safe bubble.

When the kernel is running in user-space, it is much easier to use standard debugging and verification tools. Debugging with **gdb** or profiling your code using **gprof** is no more involved than any other application. Other diagnostics, for example branch testing and other automated regression tests, can also be applied.

User Mode Kernel (UMK)

The only existing "Kernel in a Sandbox" system, the "User Mode Kernel" by Jeff Dike, started out as an experiment to prove doing such madness was actually possible; it was later recognized to have real debugging value and has since evolved into a major project and a major voice behind mainstream kernel facilities such as **ptrace()**.

When run in on a console or in a terminal window, the UMK will appear just like any kernel boot, but with some devices adjusted around the host machine. For example, during the boot sequence, the new kernel may report serial ports as

```
serial line 0 assigned pty /dev/ptyp7
```

; to connect to this running kernel, you can either login at the prompt in your window or connect via the serial port. In our example this would be by pointing **minicom** or some other termal program at `/dev/ptyp7`; when you exit from this session, the user mode kernel will close that port and open a new one, for example, on `/dev/ttyp6`

The UMK is available for Linux 2.2 but is primarily aimed at the 2.3 kernels; if you only need to test loadable modules, you can also fetch a pre-compiled kernel image. The software supports network emulation, but does not (yet) simulate multi-processors and many hardware devices are in need of a hero to code them in. Your installation will require the UMK patches or the binary and a root filesystem for

the new runtime kernel; this can be built on any existing filesystem using the scripts provided. More information on UMK and all the necessary files can be fetched from Jeff Dike's UML[2] website (http://www.mv.com/ipusers/karaya/uml/uml.html).

CPU Emulators

Depending on your application and its device requirements, it may be possible to boot your experimental kernel within a ix86 (or other architecture) emulator such as VMWare or FreeVMWare/Bochs, although the stark division between the operating space of such emulators will severely limit the amount of information you can extract from the test; considering the extra cost of the software and of the machine requirements needed to run the virtual machine, it is probably easier, cheaper and more fruitful to purchase a cheap beater-box and to use the remote debugging technique.

Useful Utilities

You have your new custom kernel up and running, it gets safely past initialization and now you need to know if it is doing what it should and is not doing what it shouldn't. In this stage of development, you may not need the bother of a symbolic debugger to know your program has gone astray (or to prove that it hasn't) and in some cases, a debugger would be giving information too low level or too specific to be useful.

While it is true that there is not much you can do about the behaviour of your changes while the kernel is booting, most often it is enough to just watch the progress of your code and the state of your machine using the traditional means of the programmers' toolkit: Since most devices are readily implemented as dynamically loaded modules, there are relatively few applications outside of inner kernel subsystems development where plain old system monitor tools are not enough to diagnose the problem ... and as expected, it is those deep core procedures which are the most intractable to identify, but let's try to stay positive: 90% of everyday kernel diagnosis will be obvious from the system behaviour. For that other 10%, the following sections describe some of the available options available to Linux programmers.

System and Process Monitors

Assuming your modified kernel will boot far enough to be accessible, and also assuming your changes are amiable to being tracked by the usual system monitor tools. These include the normal arsenal of unix commands such as **top**, **free** and the direct reading of the `/proc` pseudo files these utilities reference. Very recently, other standard unix tools for process accounting, programs such as **sar**, have been ported to Linux and will generally give a more accurate picture of what the machine was doing while it ran your code.

Analyzing Core Dumps

It ran for hours, the process monitors all looked normal and then ... ooops. There are several means available to deduce the final state of the kernel after an oops, ranging in their diagnostic detail from showing only where the program failed, to being able to recreate the state of the machine.

At one end of the post-mortem spectrum, the **ksymoops** utility, which was included in the Linux source tree up until the 2.2 distribution, will map the output of the kernel oops to the System.map file to show the name fo the function where the failure occured (See the section called *Tracing from a Kernel OOPS* in Chapter 6. Very often, knowing the location of a failure is enough to diagnose a problem, but for other situations, there is no substitute for a full backtrace and snapshot of your machine at the time of death. In the traditional user-space programming toolkit, this is the domain of the **gdb –core** command; thanks to the contributions of the Linux Kernel Crash Dump project at Silicon Graphics, it is now also possible to apply this analysis to the Linux kernel

The LKCD kit includes kernel patches and user-space tools to capture a core dump of a failing kernel to a dedicated SCSI partition, and it can also perform some analysis on the generated core dump or to reload the kernel state when the machine reboots.

In its simplest form, LKCD will store the core file snapshot to `/dev/vmdump`, and then, on reboot, it will fetch this image and generate a report on `/var/log/vmdump`. RPM files and the necessary kernel patches can be downloaded from the LKCD homepage (http://oss.sgi.com/projects/lkcd/).

Kernel Profiling

Your program does not oops, and it doesn’t exhibit any gross behaviours recognizable during process monitoring, but it still fails to make the proper handshake or just needs to go faster; this is the domain of profiling.

The stock Linux kernel offers very basic facilities for gathering timing statistics on certain subsystems such as the SCSI parallel interface and the network layer, but all such code is limited in resolution by the 100Hz kernel clock. As a result, the standard kernel profiling is pretty much useless. Several authors have contributed patches to allow fine grain profiling but in all cases, these patches are highly version-specific and are not be available for all kernels; the code is often portable to other kernel versions, and the authors are almost always helpful, but if you are tinkering with this stuff, be prepared for surprises (which is to say, don’t promise the results to anyone by any specific deadline date).

Profiling with gprof

If you have been following this chapter from the start, you may have already concluded that Silicon Graphics is a big player in the realm of kernel debugging; kernel profiling is yet another area made

possible by the kind contributions from SGI engineers. The **gprof** patches provide detailed trace information through the `/proc` filesystem; once the system is prepared, **kernprof** is used to set the sampling rates and to toggle the data collection and this data is then analyzed by running **gprof** on the kernel image and the generated data.

Prepping the kernel is done by applying the SGI patch to the source tree to collect the call arc data; unfortunately, due to a bug in all known editions of GCC, including the EGCS compiler, **gcc** must also be modified using the patches found in the EGCS patch archive (http://egcs.cygnus.com/ml/egcs-patches/2000-04/msg00332.html). If you are unable to modify your compiler, an alternative fix is to remove all **FASTCALL** support in the kernel sources; the SGI patch kit includes a patch to do this removal but keep in mind that this will change the behaviour of all systems which might use the FASTCALL API.

As of this writing, there is no FTP archive for this patch other than in the mailing list archives. If you are looking for it in some other archive, the **gprof** patch was posted to the Linux Kernel Mailing List (http://linuxwww.db.erau.edu/mail_archives/linux-kernel/May_99/2383.html) by Dimitris Michailidis on the 14th of May, 1999.

Linux Trace Toolkit

Karim Yaghmour's Trace Toolkit is another kernel instrumentation package designed to catalog CPU and allocation events down to microsecond resolution. Like other instrumented-kernel patch sets, LTT adds a trace capability option to the standard kernel configuration script and lets you choose the level of detail and the specifics on the events to be monitored.

Because the LTT tracks the system over a specific time period, it is particularly useful for debugging I/O latency and synchronization problems. Generated reports include event flow, process analysis and raw event logs; these reports are all available in user-space via the `/proc` interface and can be read as text tables or analyzed with the included GUI reporting module.

The kit consists of the kernel instrumentation patches, a kernel module and trace daemon to handle the data collection, and a GUI report reader which will translate the trace output into a readable report. Source files and documentation can be downloaded from the Linux Trace Toolkit Homepage (http://www.info.polymtl.ca/~karym/trace/)

IKD Debugging Tools Patch

For the Linux 2.2.x kernels, Andrea Archangelli compiled a single patch to bundle together the most common kernel debugging facilities. The included tools cover

- Debug kernel stack overflows

- Detect software lockups
- Procedure flow tracing
- Set breakpoints and single step
- Print-EIP on video ram
- Stack monitor
- Fine-grain profiling

Unfortunately, all of these services are highly version dependent and while the patch may be portable to other kernel versions, Andrea tends to only support a few as the needs arise.

The latest versions of the IKD patch are available from e-mind.com through Andrea's FTP area (ftp://e-mind.com/pub/andrea/ikd/)

Notes

1. In 1999, the Glimpse search engine was pulled from the free software world and re-released as a commercial product with the usual restrictions on for-profit use. Before you install Glimpse and its companion WebGlimpse, read over the new license to ensure your use will comply with their requirements.
2. While Jeff refers to his system as the User-Mode Kernel, he most often abbreviates this as UML (user-mode linux); I have chosen the short form of UMK to avoid confusion with the Universal Modelling Language