LINUX JOURNAL

Advanced search

Linux Journal Issue #17/September 1995

Kernel Korner System Calls *by Michael K. Johnson*

ncurses: Portable Screen-Handling for Linux

Eric S. Raymond

Issue #17, September 1995

Have you ever written a screen-oriented program under Linux and been annoyed by the limitations of the standard curses screen manipulation library? Here is a programmer's introduction to the vastly superior ncurses library.

Since you are clueful enough to subscribe to *Linux Journal*, you may already know that Linux includes a clone of the Unix screen-handling library, curses(3). You may also know that Linux's ncurses(3), like its ancestor, is a general screen-handling library that supports all popular asynchronous terminal types, including the color-ANSI and vt100-like capabilities of most Linux consoles. You've certainly run a program that used curses, vi, perhaps, or any of a dozen screen-oriented games.

You may not know that there are actually two major flavors of curses, with differences that are important for portability. You probably don't know (unless you have read ahead) that post-1.9 releases of ncurses include dramatic new features that will make them far more reliable, powerful, and useful than their predecessors.

curses History

The first curses library was hacked together at the University of California at Berkeley in about 1980 to support a screen-oriented dungeon game called rogue. It leveraged an earlier facility called **termcap**, the terminal capability library, which was used in the vi editor and elsewhere.

Termcap consisted of a simple text database format for describing the control sequences and capabilities of serial terminals. Before the ANSI standard hundreds of mutually-incompatible terminal types competed in the market; termcap described them in a standard way that made it possible for vi's screen-painting code to be terminal-independent.

The curses library took this one step further, by defining a relatively clean C API (applications programming interface) to hide the details of termcap-based screen painting. This facility was a terrific success, and gets almost all the credit for lifting Unix out of the line-oriented interface style inherited from its early days on ASR-33 Teletypes.

Enhancements and Controversy

The success of curses did not go unnoticed at Bell Labs. Later System III releases and System V Release 1 included an enhanced curses library with many new features. These included:

- Support for multiple screen highlights (BSD curses could only handle one “standout” highlight, usually reverse-video).
- Support for line- and box-drawing using forms characters.
- Recognition of function keys on input.
- Color support (in later versions).
- Full support for sub-windows.
- Support for pads (windows of larger than screen size on which the screen or a sub-window defines a viewport).

The Bell Labs curses was upward-compatible from BSD curses, and much more powerful. However, its designers made one controversial major change; they scrapped the all-text termcap format for terminal capability descriptions, opting instead for a binary format called terminfo.

The most important reason for changing formats was that the BSD termcap database was getting so large that sequential searching took significant overhead. The terminfo format (binary capability blocks living in a bushy directory tree), by contrast, was optimized for fast lookup and fast loading.

In doing changing to a binary format, however, Bell Labs curses gave up two valuable features: extensibility and the ability to edit terminal descriptions with standard text tools. The termcap routines had never actually cared what the capabilities meant to the program using them, whether it was curses or something else. Thus, it was easy to add new capabilities and new interpretations; in fact, later BSD releases used the same code for printer- and modem-capability databases. In terminfo, by contrast, additions to the database format required a recompile of the library.

AT&T's no-source-code policy meant that Bell Labs curses itself never got a chance to outcompete the BSD version in the hacker community. Worse, the tradeoffs in the terminfo format meant that the largely BSD-centric hacker culture had a passable excuse to denigrate the major new capabilities in Bell Labs curses.

For more than five years, then, freeware authors writing screen-oriented programs were in the vexing situation of having to design around the older, less-capable BSD interface even on machines that supported the Bell Labs curses.

Source Code Freedom

Around 1982, noted hacker Pavel Curtis (formerly of Xerox PARC, now perhaps best known for his MOO project) attacked this problem head-on by starting work on a freeware clone of Bell Labs curses. This package was part of Pavel's personal toolkit; it was not widely distributed, or even very well known, until Zeyd Ben-Halim, zmbenhal@netcom.com, took over the development effort in late 1991. I got involved in late 1993 to support a screen-oriented multi-user Unix BBS I was developing.

The early ncurses versions had some serious drawbacks. The biggest problem was that a good many important Bell Labs curses capabilities were absent, leaving annoying and unpredictable gaps in the API. The documentation was poor and not suited to on-line browsing. Very little systematic compatibility testing against Bell Labs curses had been done. Last, but not least, all versions up to 1.8.5 had serious bugs.

Quality Improvements

Much has changed in the last year, out of sight of the Linux community at large. We have fixed many bugs and filled out the API. We've written more complete documentation, including man pages. We've done extensive compatibility testing against SVR4, linking identical programs to both libraries and comparing behavior. We've audited the code carefully for cross-platform portability and replaced a rather clunky home-brewed configuration system with GNU autoconf. We've tested the code for memory leaks with Purify and improved argument validation in many critical functions. The overall quality of the code has been dramatically improved.

We've also added several Bell Labs-like utilities missing from older ncurses versions, including:

- infocmp(1) a terminfo entry lister and comparator.
- captoinfo(1) a termcap to terminfo translator.
- clear(1) a trivial screen clearer.
- tput(1) a terminfo capability access for shell scripts.

The captoinfo, clear, and tput utilities are based on code from Ross Ridge's mytinfo package (which we've effectively subsumed).

We chose System V Release 4.0 curses as our emulation target, and now support all of its very extensive features. We also include a clone of the SVR4 panels library, a curses extension that makes it easy to program stacks of windows with backing store.

The new ncurses package also includes a full and up-to-date set of man pages organized similarly to SVR4's.

In a few areas, we go beyond SVR4 curses, for example:

- The new cursor-movement optimizer and incremental-screen-update algorithms are smarter than the Bell Labs versions (and *much* smarter than the BSD versions) leading to significant update-speed gains for slow terminals.
- Unlike previous curses versions, ncurses can write the lower right hand corner cell of a terminal with automargin wrap (provided it has an insert-character capability).
- On Intel boxes, curses permits you to display not just the IBM high-half characters but also the ROM graphics in characters 0-31.
- Our terminfo tools recognize all the terminfo and termcap extensions found in GNU termcap, mytinfo, and the University of Waterloo libraries.
- We feature a C++ class derived from the libg++ CursesWindow class, but enhanced for ncurses.
- The ncurses suite now includes an ncurses program which allows you explicitly test most of ncurses's capabilities on any new platform.
- We've souped up the tracing feature. It is now possible to pick one of several levels of output tracing, and the trace log has been made easier to interpret (with symbolic dumping of screen attributes, colors, and so forth).
- Automatic fallback to the /etc/termcap file can be compiled in for systems without a terminfo tree. This feature is neither fast nor cheap, so you don't want to use it unless you have to.

Along with ncurses, you get a very complete terminfo file. I accepted the maintainer's baton for the 4.4BSD master termcap file from John Kunze in January 1995; I've since translated it to terminfo and added a lot of information from vendors like SCO, Digital Equipment Corporation, and Wyse. (The terminfo file is available separately from my WWW home page, www.ccil.org/~esr/home.html.)

How Do I Use ncurses?

Listing 1 gives a program skeleton that illustrates how to set up to use some of the new features.

Note the way we arrange for **endwin()** to get called by the signal catcher. This is necessary, otherwise a stray signal might leave the tty modes in a less than useful state.

Our next program fragment illustrates the use of the keypad mapping feature. This code, from an actual test program distributed with ncurses, is shown is Listing 2.

The example code illustrates how you can get function key tokens as single values outside the ASCII range from **getch()**. The call **keypad(stdscr, TRUE)** sets this up.

The ncurses distribution contains a more detailed tutorial in HTML form and also has several code examples in the test directory.

The Future of ncurses

The ncurses library seems to have achieved a stable plateau, suitable for production use, in the 1.9.1 and 1.9.2 releases. We're looking ahead to some interesting possibilities—most notably at supporting XPG4 Curses extended-level features for wide and multi-byte character sets. We're also thinking about MS-DOS and MS-Windows ports.

With the features we already have in a freeware library, you may be wondering whether there are good reasons for BSD curses to continue to exist. The answer is “probably not”.

Keith Bostic, maintainer of BSD curses, has agreed that if ncurses 1.9 proves stable and usable with nvi (new vi), he will switch to ncurses for the nvi distribution and pronounce BSD curses officially dead. The ncurses library has already been used to successfully support nvi for production, so it's a good bet that by the time you read this, Keith will have finished his testing—and BSD curses will be history.

This, in turn, means the Unix world will finally get a common freeware API that supports color, multiple highlights, and all the other capabilities common to PC consoles and today's character-cell terminals.

Update

Since this article was written, Keith Bostic tested ncurses with nvi, and has officially pronounced the old BSD curses dead. This means that ncurses will become the official standard curses for Linux, as well as for all the myriad free versions of BSD Unix.

Eric S. Raymond , esr@snark.thyrsus.com, is the volunteer editor of the Jargon File, a prolific author of freeware and FAQs, and a recent convert to Linux. His WWW home page is available at www.ccil.org/~esr/home.html.

Archive Index Issue Table of Contents

Advanced search

Advanced search

Writing a Mouse-Sensitive Application

Alessandro Rubini

Issue #17, September 1995

Terminal-oriented programs tend to have an unwieldy interface, while writing X-Windows applications is difficult. By using the gpm client library you can easily turn a text-oriented program into an easy-to-port mouse-sensitive application.

The Linux text console is more than a bare terminal. One of its most important features is the availability of a mouse device. In addition to supporting *selection*, the mouse can be used to interact with user programs. If your computer runs the **gpm** server your programs can easily benefit from mouse availability under both the Linux console and xterm, and run without complaint under other environments.

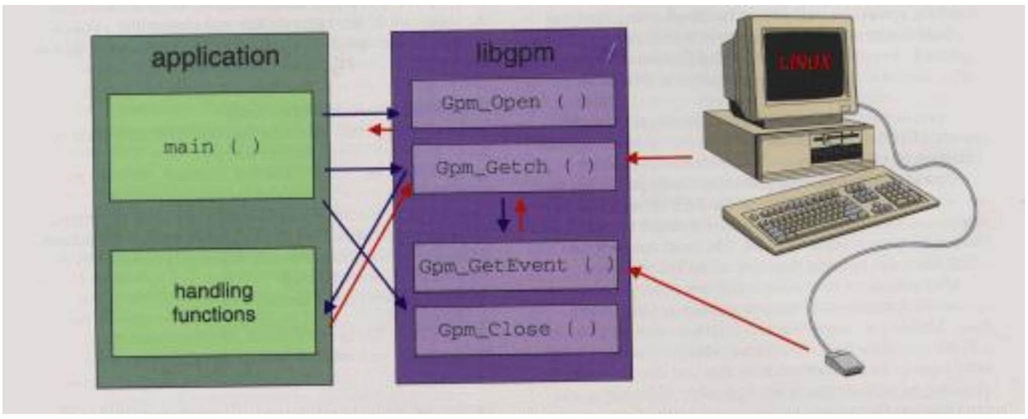


Figure 1. An application using libgpm

The main difference between a conventional text program and a mouse-sensitive application is the way they process input—while the former reads from **stdin** and writes to **stdout**, the latter must multiplex input from different sources—we can call it an “event-driven” application. I will consistently use “program” to refer to a **stdin**-driven process and “application” to refer to an event-driven one.

The gpm client library is meant to allow programmers to easily turn a program into an application by changing only a few lines of original source code. Alternatively, it offers complete support to developers designing an application from scratch. Portability is a major issue here, because you can be tempted to build a nice full-featured application for the Linux console, which reveals itself as completely unusable when you remotely log in to your PC from within an xterm or a bare vt100. Some care must be taken to avoid this, since a networked Linux computer can easily be used from a tty which has nothing to do with its own console.

The internal structure of a console application is represented in Figure 1, which outlines what changes required in the original program so it can respond to mouse events. As you can see, all mouse support code could be hidden in a separate module, and mouse-related code in the main body of the source is limited to the following calls:

- **Gpm_Open()** The open function should be called before reading any input. It is used to connect to the daemon's socket and performs all the setup needed to get back events from the **gpm** daemon.
- **Gpm_Getc()** Any call to **getc()** and to **getch()** should be replaced by the **Gpm_**-prefixed function. The replacement code manages multiple inputs and dispatches mouse events as needed—more on this later.
- **Gpm_Close()** Before **exit()**ing, the mouse connection should be closed. This call may be omitted, though it isn't nice to do so.

When writing portable code, these few modifications could be masked out as suggested in the code fragment below. Its role is to define function names which are independent of mouse availability. Such preprocessor-specific code would better reside in a header file, to avoid ugly **#ifdef**s in the actual source. The approach of choice is to hide **Gpm_Open** in **local_mouse_init**, because setting up is more than a function call; conversely, **local_mouse_close** is a syntactic place holder.

Any other code referring to the mouse can be put in a different source file from the general application code. A correct Makefile (possibly through **autoconf**) can easily choose which files need to be compiled and which preprocessor defines are needed, without cluttering the code with **#ifdef**/**#endif**.

```
#ifdef CFG_MOUSE<\n>
#    define local_wgetch(w) Gpm_Wgetch(w)<\n>
     extern int local_mouse_init(void);<\n>
#    define local_mouse_close() Gpm_Close()<\n>
#else<\n>
#    define local_wgetch(w) wgetch(w)<\n>
#    define local_mouse_init()  /* nothing */<\n>
#    define local_mouse_close() /* nothing */<\n>
#endif
```

Connecting through Gpm_Open

Choosing a good connection with your mouse device is tricky. The problem is getting the best event resolution while avoiding excessive of context switches. Some simple applications need to be told of only button-press events and can leave cursor-drawing to the server program; more complex applications, on the contrary, might want to be told of any single movement of the mouse, as well as button-press and button-release.

The **Gpm_Open** function gets as an argument a structure identifying the type of connection requested. The type of connection in turn is characterized by event masks—bitmaps identifying event types. With gpm, two mask are required—the mask of events you want to get and the mask of events you want to be handled in the default way.

The double mask is useful, because the default way is known. In particular, since you know that mouse motions cause the cursor to be drawn, you may often leave motion events to the default management, thus relieving your application of most of the work of handling the mouse.

In addition to event masks, the connecting application must specify two “modifier sets”, that is, sets of keyboard modifiers, such as shift, control, meta (alt) and so on. Within the gpm server, keyboard modifiers are used to multiplex applications on a single console. It is handy to be able to paste selected text in a mouse-sensitive application, while an application taking complete control of the user's pointer would irritate most of the customers.

Each gpm client is asked to specify a “minimum set” and a “maximum set”. The client specifically asks not to be informed about mouse events with less than the minimum set or more than the maximum set of attached modifiers. The minimum set will be 0 for most clients. The gpm-root menu drawer is a client with non-0 minimum mask. This gives selection mouse-only events when there is no other client. Thus, when running Emacs, you can use the Emacs mouse facility (by loading the library **t-mouse.el**, within the gpm distribution) and have access to selection and gpm-root; the lisp package accepts mouse-only and **alt**-mouse, the gpm-root server gets **ctrl**-mouse, and the internal selection mechanism gets any other events. Within this scheme, selection is a catch-all, as if it had an infinite maximum-modifiers mask.

Gpm_Open, then, keeps a stack of connection masks, so you can reopen the connection to modify your mask and get back to the previous behaviour on the next **Gpm_Close** invocation. This feature can be used to either increase or decrease the amount of events you get. Emacs, for example, disables event reporting in this way when it is stopped, to allow you to use selection normally with your shell. An application drawing a menu, by contrast, can only reopen the connection to get motion events while the menu is kept down. This stack-like feature is managed in the client library, so that misbehaving applications can't lock up the server.

Using mev

To test library features without loosing your youth in a compile-execute-understand-recompile loop, the **mev** program is distributed along with the gpm server. This tool is based on the idea of **xev**, the X event reporter. mev reports any event it gets on the current console, and you can specify on the command line the event and modifier masks to be used. Thus, mev can help in testing your connection parameters before you hardwire them in your application. Mev can also demonstrate use of the connection stack by getting push and pop commands from standard input.

Though mev was originally designed as a test case to debug the gpm server, it is now quite useful in itself and I use it as the engine for the emacs library.

Taking input from multiple sources

After connecting to the server, the application must respond to both keyboard events and mouse events. To ease this, libgpm offers replacement functions for getc and its relatives, but the application designer is not forced to use them.

Programmers who want to autonomously manage the two input channels must invariably use the **select()** system call, unless the application puts the input files in non-blocking mode and keeps polling them. Polling can be wise if your application takes a long time to do its job, and you want the user to be able to regain control with a single key or mouse press. A good example of this technique is to be found in the **Netscape** WWW browser.

Many screen-oriented applications, on the contrary, spend most of their time waiting for user input, and could well benefit from the functions in libgpm. Externally, **Gpm_Getc()** and its relatives behave just like the originals when keyboard events are received; internally, they can receive mouse events and handle them through a user-defined function.

These input procedures, as you can imagine, are built around **select()**. This is the only way to relieve the user from using select in the application body. When stdin is reported as readable, the original **getc** function is called; when the mouse connection is readable, **Gpm_GetEvent()** is called.

Applications can invoke **Gpm_GetEvent()** by themselves if they need to, but you must remember that **Get_Event** is based on a **read()** call, and thus is blocking. The normal gpm input functions (usually **Gpm_Getc**) in libgpm invoke

Gpm_GetEvent only when there is data to read. The getc-replacing function then delivers the event to a mouse-handling function specified by the application. Note that **Gpm_GetEvent()** is only in charge of reading the event from the current source and does not deliver the event to the mouse-handling function.

The user function in charge of handling mouse events—let's call it “mouse handler”---is registered by the user in a global variable before invoking the input functions, and its invocation doesn't interfere with the running **Gpm_Getc()** invocation. Be careful, however, that the input function is waiting for the handler to complete; long-running tasks don't fit into the mouse handler well.

Faking Keys

More often than not, the mouse is simply used as a shortcut for the keyboard: for example, clicking on a menu-button is like pressing **f1**, clicking on a listbox item is like entering its highlighted letter, pressing the button outside an active menu is like issuing the **esc** key, and using the scrollbar is like pressing the arrow keys many times. Application design is greatly simplified if the mouse can return keys to the input subsystem—the dual input mechanism is again joined into a single input stream, greatly reducing the amount of status information to be managed.

Within libgpm, this behaviour is enforced by the return value of the mouse handler. The handling function returns an integer value which gets interpreted in the following way:

- **EOF** This value is used to signal a fatal error and will cause the input function to return the same value to the caller.
- **0** A zero return value means that the input function should go on as before, without returning to the caller. The event is considered eaten by the handler and no key-press is simulated.
- **Anything else** Any other value is considered a “simulated” keystroke character, and is returned to the caller. Note that these values are not limited to ASCII characters—any integer value can be returned.

Before returning a fake key, the input functions set a global variable, which signals that the key is not a true key press; before returning a keyboard-generated character, the flag is cleared. Applications are free to use or ignore this information. Personally, I have never used it.

Note that the ability to return any integer value is powerful and is perfectly compatible with the libc environment, because **getc()** returns an integer by definition. Return values exceeding a character range can be used to encapsulate mouse activity into a generic “event” integer entity, and the same **switch** construct in the main loop can handle any input the application gets.

Using the fake-key capability, any mouse event can be packed in an integer value to be interpreted later in the main loop of the application. Personally, I prefer to interpret the event inside the mouse handler and return to the caller only a integer belonging to a small set of actions.

The mouse handler can also register its intent to return more keys, so it may be called without waiting for a new mouse event. Thus, a scrollbar can easily be implemented in the mouse handler by returning to the caller the right number of arrow key presses.

Smart use of the fake-key mechanism can greatly ease the design of a complex application, with negligible computational overhead. In practice, you must be careful when you write mouse-handling code which can't fit the fake-key mechanism; you must be sure that a user sitting on a vt100 tty without any pointing device won't loose control of the application by falling into a state which is unrecoverable without the mouse. It is best if the unfortunate mouseless user can exploit your application in full despite limited input capabilities.

Stacking Applications

Typically, a smart program allows itself to be temporarily stopped or offers the user the option of spawning a shell. This ability is often overlooked during program development, because programmers tend to concentrate on the application itself, rather than on escaping from it. Before giving away tty control, any mouse-sensitive program should release the mouse to avoid stealing events from a user trying to run the selection mechanism within a shell environment. The preferred way to release the mouse in this case is to invoke **Gpm_Open** with connection parameters indicating that all events are passed along to the next service. When the program resumes the user focus, it can simply **Gpm_Close** to restore the previous event masks. If the application forgets to release the mouse before releasing the tty, weird things happen.

Using curses

Usually mouse-sensitive applications manage the screen using **curses** or the compatible **ncurses** library. [See page ?? for an introduction to ncurses—Ed] From the point of view of mouse handling, this doesn't make much difference. You need only to call **Gpm_Getch()** or **Gpm_Wgetch()** in place of the getc or getchar. These replacement functions take the same arguments as the original curses calls.

From the mouse-handling point of view, the only difference between a full-featured curses application and one using normal tty is in the possible subdivision of the screen into different windows. Using a single mouse handler makes management non-trivial if the screen is split into multiple windows. The scenario is dealt with by the so-called high level library, which is a simple yet effective set of functions to manage a stack of “regions of interest”, easing the dispatch of events to multiple recipients.

The High-Level Library

The high-level part of the gpm library offers entry points to a centralized data structure responsible for delivering events to multiple mouse handlers.

In practice, a double linked list of ROIs (regions of interest) is maintained, and each ROI is responsible for handling events for a specific user function with specific “clientdata”. Each region is identified by its rectangular limits and by minimum-modifier and maximum-modifier sets. Thus you can choose to deliver events to different windows, according to either the event position or the modifiers used, in a way similar to the multiplexing of applications on a single console described earlier.

When you use a windowed interface, you can take full advantage of the high level library by creating one or more ROIs associated with each curses window. In addition to events happening in the ROI, the handler associated with a region will get “enter” events when the mouse cursor enters the region and “leave” events when it leaves. This means that a single mouse motion can generate multiple callbacks to help keep a consistent screen state without needing a huge set of global state variables.

Unfortunately, the high level library has been available only since gpm version 1.0. If you have an older version of gpm you would do best to upgrade. Lack of the high level library was the main reason that gpm's version numbers were 0.*x* for such a long time.

Xterm Support

Within the X Window System, terminal applications are run within **xterm**, and xterm is the only usable tty you can find on most workstations—usually workstations are terribly slow and unusably hostile before X-Windows is started.

Fortunately, xterm is able to report mouse events, made up of escape sequences, which are reported to the client application through the same channel as normal data.

Unfortunately, the range of events it is able to report is severely limited. Moreover, because events are reported through the same stream as keyboard events, all the nice design of multiple input channels breaks, and any application which wants to sense mouse and keyboard events independently fails.

Fortunately, using **Gpm_Getc()** and friends works quite well, as you can check by running mev under an xterm.

If you consider ever running your application under an xterm, you must be sure to not depend on a full event reporting. Specifically, you won't be informed of any motion or drag events, and button-release events won't specify which button of a set has been released. This means, in practice, that if you need precise reporting of a double-button press, your application will not work properly under xterm.

I strongly urge you to be careful; if the application can only run under the Linux console, it is of limited use, and you'll surely swear at yourself sooner than you may expect. If, on the contrary, the application is able to run under xterm, it is better exploiting the ability to (at least) invoke buttons by a simple mouse press, rather than forcing the user to use keyboard-only interaction.

Using GNU autoconf

And what if the environment is not a Linux computer? A pair of good design choices and a small investment of your time can make you a proficient user of the **autoconf** package, and your application can easily adapt to the following environments:

- **A Linux machine with gpm installed.** This is the best environment, and the application will be compiled with full support, under both the console and xterm. When invoked within a mouseless tty, the application will run in keyboard-only mode without needing runtime conditionals.
- **A Linux machine without gpm.** If the application is distributed in binary form, the gpm library will silently detect lack of the server and will run in keyboard-only mode on the console. Under xterm everything will work. If the application is distributed in source form, and thus can't link in the gpm library, the following case will apply.
- **Another Unix-like operating system.** The application will compile with xterm support built in, because autoconf will include **gpm-xterm.c** in the set of files to be compiled. This source replaces the most useful functions you find in libgpm (that is, the open, close, and getc functions) and **Gpm_Repeat()**, a support function used to provide event repetition while the button is kept pressed. The concept of “mouse handler” will still work.

- **A non-Unix operating system.** It seems like a lost battle... You have to include a lot of conditionally compiled code anyway. Are you sure you need a mouse-sensitive application? In any case, it will be no harder than making any application portable between significantly different operating systems.

The code excerpts in Listings 1 and 2 include the the relevant parts of **configure.in** and **Makefile.in** used to create the “portable” sample application distributed within the gpm package. They are reproduced here to give an idea of how easy it is to set up a portable compilation environment. In fact, you needn't be an autoconf expert to set up such an environment, because a little documentation and good amount of cut and paste can easily work.

This configure.in checks if **Gpm_Repeat** is found in libgpm and selects whether libgpm is linked in or **gpm-xterm.c** should be compiled. Note that the high-level library, though not managed in this configure.in, is independent of the low-level mechanisms, thus it can be included in the portable application as well.

Gpm_Repeat is a software aid to repeating events on a timely basis up to button release. It works also under xterm and is used here as a test because it appeared only when the library and server were quite stable. I presume you don't want to link your application with libgpm 0.01, in the unfortunate case that some early alpha tester has one lying around on his or her hard drive.

Is This Pain Worth the Effort?

Before you actually start coding, however, it is worth understanding the pros and cons of mouse programming using libgpm, and being warned against common pitfalls.

If you need to write a friendly interface, using libgpm is *really* difficult compared to writing a **Tk** script. If your interface is going to run on a powerful workstation, you are better off running X-Windows and Tk. Moreover, it is completely portable—free Macintosh and MS Windows ports of Tk are in development.

If your application will run on a general-purpose workstation which does not run X-Windows, you should take into account the trend to upgrade existing hardware. Thus, if your application is a medium- or long-term project, you might be better off to start with Tk anyway.

But then, what applications need libgpm, if the author himself discourages its use? As a simple rule, I suggest writing text-only applications when you need to support the whole range of Linux computers. System management tools are good candidates for libgpm—remember that Linux-1.2 still runs happily with 2MB of RAM and 10MB of disk.

Another field which could benefit from a simple mouse-sensitive front end is the field of embedded systems and dedicated machinery. For example, an inexpensive Linux box can be used as an NFS (Network File System) or WWW server for a small company, and usage reports will be queried by novice users. Avoiding X-Windows and writing a gpm-based interface is a win here.

If I've not discouraged you from using libgpm, go for it, but remember to pay attention to portability, simplicity, and the user's proficiency.

Portability is a major issue when developing a Unix application. Particularly, remember to build a tty-independent application—this means you must always provide a keyboard alternative to mouse events. There are hundreds of tty types, and you can't force a user to use the Linux console. Besides, a user might need to drive your application in “unsupervised mode” through stdin.

Another important issue is keeping it simple: don't depend on things like pressing two buttons at once, for example, which won't work under xterm.

Finally, remember that the user must *feel* the mouse. You should redraw the mouse cursor (possibly by means of **Gpm_DrawPointer**) after each write to the screen. This is important because users tend to use the mouse for selecting text, and using a mouse-sensitive application in the same way as **selection** can make for disasters.

Where to Get the Software

The gpm package is available by ftp from sunsite.unc.edu/pub/Linux/system/Daemons/gpm-1.06.tar.gz. Sometimes small improvements don't get to sunsite, because I don't want to fuss up the maintainer. The very latest release is always available from iride.unipv.it/pub/gpm/gpm-1.06.tar.gz.

The source package includes a full **info** file and a PostScript manual describing the library much more thoroughly. A sample portable application is included as well.

The package is also distributed in binary form (but with the full documentation) with Slackware. If you have had the Slackware distribution through floppy disks, you may want to get the source; otherwise it is in the cdrom. Recently, I've also heard a proposal to “debianize” gpm, so it may appear in the Debian distribution in the near future.

For any question not answered in the documentation, feel free to contact me.

Listing 1. Simple configure.in for a Mouse-Aware Application

```
dnl configure.in for sample gpm client
dnl This will only run with autoconf-2.0. or later
AC_INIT(rmev.c)
AC_PROG_CC
AC_PROG_CPP
CFLAGS="-O"
LIBS=""
dnl look for libgpm.a; if found assume to have
dnl <gpm.h> as well. Gpm_Repeat is only present
dnl after gpm-0.18
AC_CHECK_LIB(gpm, Gpm_Repeat,[
    GPMXTERM=""
    LIBS="$LIBS -lgpm"],[
    GPMXTERM="gpm-xterm.o"
    if test "-uname-" = Linux
      then AC_MSG_WARN("libgpm.a is missing or old")
    fi
    ])
dnl subsitute @GPMXTERM@ in Makefile
AC_SUBST(GPMXTERM)
```

Listing 2. Simple Makefile.in for a Mouse-Aware Application

```
# simple Makefile.in - autoconf will
# replace any @symbol@ with the right value
# include standard stuff
srcdir = @srcdir@
VPATH = @srcdir@
CC = @CC@
CFLAGS = @CFLAGS@
LDFLAGS = @LDFLAGS@
LIBS = @LIBS@
prefix = @prefix@
OBJS = rmev.o @GPMXTERM@
TARGET = rmev
all: configure Makefile $(TARGET)
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) $(LDFLAGS) \
               -o $(TARGET) ($LIBS)
clean:
    rm -f *.o $(TARGET) config.*
### rules to automatically rerun autoconf
Makefile: Makefile.in
    ./configure
configure: configure.in
    autoconf
    configure
distrib: clean
    rm -f config.* *~ core
    autoconf
    rm -f Makefile
```

Alessandro Rubini (rubini@ipvvis.unipv.it) is taking his PhD course in computer science and is breeding two small Linux boxes at home. Wild by his very nature, he loves trekking, canoeing, and riding his bike. He wrote **gpm**.

Porting DOS Applications to Linux

Alan Cox

Issue #17, September 1995

Trying to port a DOS application to Linux? Alan Cox gives you hints and practical help.

With a little care, the average DOS application can be easily ported to the Linux system. This article looks at some of the techniques involved, and tries to provide a small “builder's kit” of handy little DOS routines people always want under Linux.

Memory Models

DOS programs written in the C and C++ languages generally run in a variety of different memory models with their own segmentation semantics. The simplest is the “tiny” model, where all of the program and data are referenced off one segment. All three segment registers (CS, DS, and SS) point to the same place to suit the way the processor wishes to work. The Linux kernel executes programs in the 32-bit equivalent of tiny mode. Because offsets are 32-bit, not 16-bit, a program can utilise 4GB of address space before segmentation becomes an issue. Thus you get the simplicity of tiny model without the limitations.

As a result of this the DOS keywords **near**, **far**, and **huge**, have no meaning to Linux. These can be removed, or if you are trying to maintain a common source tree, you can add these lines instead:

```
#if defined(__linux__)
#define far
#define near
#define huge
#define register
#endif
```

gcc, the normal Linux C compiler, understands the **register** keyword, but the code optimiser is sufficiently good that using **register** is normally a bad idea.

Many DOS C compilers support an **inline** keyword. gcc also supports this.

C Types Supported

gcc supports all the ANSI C types you would expect and some extensions. The size of the normal types is, however, different from that of DOS compilers, and frequently causes problems when porting. Here is a summary for sizes on Linux/i386 (Linux on other architectures, such as the 64-bit Alpha, will differ in some respects):

Type Name	Linux	DOS time/small	DOS large	DOS huge
char	8 bits	8 bits	8 bits	8 bits
short	16 bits	16 bits	16 bits	16 bits
int	32 bits	16 bits	16 bits	16 bits
long	32 bits	32 bits	32 bits	32 bits
pointer	32 bits	16 bits	32 bits	32 bits
largest array	4GB*	64KB	64KB	640KB

* Actually, because some of the address space is reserved and used for other things, you can't get above about 2GB at the moment.

DOS programmers generally make good use of prototypes to avoid mysterious crashes caused by passing the wrong type. Mixing **short** and **long** under Linux normally just results in mysterious value changes in passed parameters, so the habit of prototyping is a good one to get into. Furthermore, you can tell gcc to warn you about any routine which has no prototype by adding the compiler flag **-Wstrict-prototypes**. All of the C library and system calls have prototypes, provided the correct header files are included.

gcc: the GNU C Compiler

The GNU C compiler is an extremely flexible tool. Although it compiles much slower than most of the DOS compilers, and is (intentionally) without an Integrated Development Environment, it has a wide range of abilities and flexibility that few DOS compilers can touch. People who have used DJGPP to write 32-bit DOS extender programs will be familiar with gcc, although in its Linux and Unix form it is somewhat easier to work with.

It is worth knowing how to tell gcc how to cope with different “flavours” of code. It can become a traditional K&R C compiler, by using the **-traditional** option, a strict ANSI compiler, by using the **-ansi** option, or a GNU C compiler—ANSI + GNU extensions. In addition, you can ask it to perform a wide range of sanity checks with the **-pedantic** and **-Wall** options. For a typical program, the compiler will generate a lot of warnings, many of which will give insights into potential problems. For example, the compiler will check to see that the conversion options in the format strings of **printf()/scanf()** and their family of functions match the types of the variables they will interpret.

The optimiser is controllable both by a general level of optimisations, using the **-O1** or **-O2** options, and on a per-optimisation basis for those speed-critical special cases. The optimiser performs a wide range of peephole and global optimisations, including intelligent allocation of registers, loop unrolling, and even instruction scheduling on RISC CPUs.

The GNU C compiler, linker, and debugger are all described in complete documents available from the Free Software Foundation, which you can either buy as books (the money goes to fund more free software work) or print yourself.

To cover the compiler, debugging tools, make, and other programs in full would require several more articles. If the documentation and documentation viewer are all installed, typing **info gdb**, **info gcc**, and **info ld** should give you a good start. (If the **info** program is not installed, the Emacs editor can also be used to read documentation in the info format.) Fans of graphical user interfaces may also like to pick up **tgdb** as a graphical front end for the gdb debugger, and **xwpe**, a look-alike of a well-known DOS C development environment, built on top of gcc, make, and gdb.

Where is the foo() Function?

This is commonly asked of a specific set of DOS C library and system functions, most notably the various text mode window packages, **kbhit()**, **getch()**, **getche()**, and the string functions **stricmp()** and **strnicmp()**.

Not surprisingly, equivalent functionality exists under Linux. The text mode window case warrants a section of its own, so you will have to wait a minute or skip on ahead. The string functions are nice and easy. **stricmp()** is also known as **strcasecmp()** and **strnicmp()** as **strncasecmp()**---there is just a naming difference.

The keyboard I/O routines cause problems because Unix terminal I/O is a lot more flexible than DOS terminal I/O, and in the case of **kbhit()** it does not suit the multitasking nature of the system to have the CPU spend all its time in loops polling the keyboard. Furthermore, unlike in DOS, the terminal mode is set explicitly, rather implicitly by each call. A set of routines (the standard POSIX **termios** functions) exist to manipulate the control structures for each device.

You cannot use the various privileged instructions that might otherwise harm the machine integrity. Controlling I/O devices is done by devices in the kernel, with access through file abstraction (the “special” files in /dev/), rather than directly. It is possible (but dangerous) to use **ioperm()** to allow access to devices in a process running as root if absolutely necessary. Any code that does this will be non-portable, and thus this should be used for special purposes only. **mmap()** can be used for equivalent access to the device memory window on a

PC (640KB-1MB), but this is equally as bad an idea for normal use. In particular, you should never, ever, attempt to do screen output this way.

Executing Other Programs

The Linux environment is built on the basis of small, effective, programs working together. Therefore, there are a wide variety of program execution facilities. They differ from the DOS environment in very specific ways. There are no equivalents to the various “swap out existing program and spawn another task” facilities found under DOS. The Linux virtual memory system will automatically decide by itself what to swap out and when. Secondly, the basic constructs of Linux process execution have no equivalents in DOS, even though there are library routines which do.

The simplest way to run another process is via the **system()** function, which calls a shell and feeds it a command string for interpretation and execution. All of the normal shell parsing and redirection is performed. This means that you should be careful of the arguments you pass if you don't want the shell to misinterpret any special characters.

This is a simple example of a program that shows who is logged on, and then who is logged on to the remote machine called “thrudd”.

```
void main(int argc, char *argv[])
{
  system("who");
  system("rsh thrudd who");
}
```

The Linux system makes heavy use of pipes—a way of feeding the output of one command into another. This is not just a shell facility. Any program can read or write from another using the **popen()** and **pclose()** calls. These work the same way as **fopen()** and **fclose()** do, save that **popen()** is passed a program as its argument. Because **pclose()** handles the termination of the process created by **popen()**, it is important to use the right close routine.

This brings us conveniently back to printing for our next example. Here is a set of subroutines for printing a file:

```
FILE *open_printer(char *printername)
{
  char buf[256];
  sprintf(buf,"lpr -P%s",printername);
  return popen(buf,"w");
}
void print_line(FILE *printer, char *line)
{
  fprintf(printer,"%s\n",line);
}
void close_printer(FILE *printer)
{
  pclose(printer);
}
```

Unlike DOS, Linux has no support for overlays in the linker, nor for loading overlays in the C libraries. The memory management and virtual memory system will swap unused program segments out of memory without assistance from the program being swapped and will automatically bring them back in when they are needed. Using overlays would not speed up program startup time, either. Program code is read from disk whenever the page of code in question is needed and not found resident in memory. The new ELF library format does support dynamic linking, and you could conceivably implement overlays using it. This is, however, pointless.

The core routines Linux provides for process execution are **execve()**, **fork()**, and **wait()**. The **execve()** call replaces the running program image with another one. The original is destroyed totally. The **fork()** call creates a new copy of the existing process. The only difference between the copies is the value returned by **fork()** (the process id of the new process in the parent, 0 in the child). Finally, the **wait()** call lets you wait for a process to complete. This level of control is unlikely to be needed when porting DOS programs, so they are not covered here.

Multitasking Politely

In general, the kernel automatically blocks programs, thus avoids having them use CPU time when they are waiting for I/O. A device can be opened with the extra option **O_NDELAY** to indicate that an error **EWOULDBLOCK** should be returned to indicate the lack of ready data (or for writing lack of buffer space). When a program is using I/O in this manner, it must take great care not to sit in a tight loop.

Avoid the following DOS-style constructs:

```
while(1)
{
  if(kbhit())
    do_something(getch());
  if(timer_expired())
    time_event();
}
```

Linux instead provides the very useful **select()** system call, which allows you to wait for multiple I/O events, a timeout, or both, in a manner that avoids polling and enables the kernel to avoid allocating processor resources to the task in question.

select() allows you to wait for a given time, or wait until one of a set of files has data ready to read or space to write, or wait until an exceptional condition occurs on that file. As the Linux system sees everything within reason as a file, this is extremely flexible.

Listing 1 shows is a “trivial” example. This is an implementation of **kbhit()**. For exact DOS behaviour, it assumes the terminal is already in **raw** mode, which we will discuss later. Otherwise, it will return 1 after **ENTER** is pressed, which is when data becomes available in the line-by-line **cooked** mode.

Sharp-eyed DOS programmers might wonder, “What happens with this **kbhit()** if we redirect the input of the program from a file? It's not a keyboard, but input is available.” The answer is simple enough—a disk file is always ready for reading, and at this level, there is no difference between reading a keyboard and reading a file. The program carries on and works fine. Indeed, you could redirect a program to run reading input from the mouse and **select()** would still behave consistently.

Files and Devices

File I/O under Linux is somewhat simpler than DOS. DOS emulates the Unix low level (**open()**, **close()**, **read()**, and **write()**) and high level “stdio” facilities, but DOS C libraries have their own ascii/binary awareness to handle the carriage return/line feed differences. Under Linux these are gone and there is no need to worry about specifying these (although the ascii/binary specification will be accepted). All of the DOS device names are different under Linux. Linux systems keep their devices in /dev. Here is a rough conversion chart:

```
      CON:          /dev/tty
      LPT1:         /dev/lp0
      LPT2:         /dev/lp1
      LPT3:         /dev/lp2
      COM1:         /dev/ttyS0 /dev/cua0
      COM2:         /dev/ttyS1 /dev/cua1
      COM3:         /dev/ttyS2 /dev/cua2
      COM4:         /dev/ttyS3 /dev/cua3
      NUL:          /dev/null
```

Note that it is normal to print by queueing jobs via the printing service (**lpr**) rather than writing directly to ports. On typical Linux systems the /dev/lp* files are protected so that a normal user cannot access them directly.

Terminal Input

Terminal I/O is distinctly different in Linux than it is under DOS. First, the POSIX terminal system is more modal than DOS. To switch from one-character-at-a-time mode (**getch()** in DOS) to a line-based editing mode requires an actual termios request, which gives the new terminal parameters to use. In addition, a program is responsible for restoring the terminal state before it runs other programs and when it exits. If you forget to do this, you may well need to switch screens and kill the process, or you may find that the shell gets confused by your terminal state and logs you out (which also fixes the problem).

Listing 2 includes some sample code for managing terminal I/O settings.

Terminal Output

On output, Unix programs traditionally avoid using direct cursor control codes and cannot write directly to video memory. The reasons for this are obvious. The terminal in question may be a different type of machine, in a different part of the world. Handling all the different terminal types by hand is unpleasant, so a library called **curses** is available. A more modern library called **ncurses**, which has such things as colour support, is also available for Linux. Older versions of this have had many bugs, but the latest appears very good indeed. See article "ncurses: Portable Screen Handling for Linux", *Linux Journal* issue 17, September, 1995, for an introduction.

ncurses provides you with simple output control, colour (if the terminal supports it), function keys, and other manipulations in a terminal-independent manner. In addition, it optimises the updates it does to minimize traffic over slow networks or serial links. It is free and comes with a nice set of examples and good documentation. As it is an implementation of System V curses, you can pick up a book on curses from a library and use that as a reference or tutorial (as appropriate).

Should you decide to use ncurses to do your output, it will also provide all the routines necessary to do DOS style character-by-character input via the functions **cbreak()**, **nocbreak()**, **echo()**, and **noecho()**. The ncurses documentation explains all four.

Standard APIs for Mice

Until recently there was no standard control API for mice in text mode (in graphics mode X-Windows runs the mouse and provides all the facilities you would imagine a GUI to have). The normal mouse behaviour is to provide text mode cut and paste. This is managed by a program known as **selection** and more recently by **gpm**.

The gpm library allows your text mode application to handle mouse events both on the console and under X-Windows in xterm shell windows. Writing programs that use gpm to control the mouse is explained in article "Writing a Mouse-Sensitive Application" issue 17, September, 1995.

Terminate and Stay Resident Programs

Occasionally you will have to port a DOS terminate-and-stay-resident (TSR) program. If you have been accustomed to using undocumented DOS calls and switching stacks and other horrible assembly language goings-on you will be glad to know you can forget the experience.

First, the whole concept of terminate and stay resident is gone. When a program exits all its resources are freed, and the process ceases to exist. That does not mean the same facilities are not present; they are present in different ways that are more appropriate to a system which is already multitasking.

There are three main reasons for TSR programs under DOS.

1. To provide a library of subroutines for supporting some extended facility. Several loadable graphics libraries have used this facility. Under Linux you can create a new shared library and it will be available for linking with applications and sharing between multiple users.
2. To add a device driver. Device drivers are kernel code. Porting a DOS device driver will almost certainly be a major rewrite. Linux also has loadable device drivers via the **modules** support. Porting a DOS device driver is definitely beyond the scope of this article. In some cases the driver may be adding a high level facility that can be provided as a library or as an actual program left running all the time.
3. To create pop-up "hot key" based mini-applications like phone books. There is no reason for these under Linux. You have multiple console screens, the ability to have multiple screens on even a fairly dumb terminal with the **iscreen** program, and can run any application at any time. Thus, there is no need for mini-applications carefully patched into the kernel. You can just port it as a normal program.

For the second example, some TSR programs can be ported as if they were applications that provided services. The **gpm** mouse management is a fine example of this. It provides the core equivalents of the DOS mouse services interrupt facilities as an application program that runs in the background and a library of support routines which interface with the server.

Porting Graphical Applications

Graphical programs are much more complex to port, because the graphics hardware interface is not available. You can approach this two ways. First, **svgalib** provides the basic functions you need to port your application. Note that you cannot use the BIOS functions, because they are only available in 16bit mode. An svgalib application can be very fast (see **linuxsdoom** for a superb example), but cannot run remotely and is not easily ported beyond PC-based systems.

The second approach is to use X-Windows. This makes for a much harder port, as you will need to move to an event-based paradigm (akin to programming for MS Windows) and rewrite your interface dialog and menus into X widgets. Furthermore X-Windows programming is—initially at least—quite hard to get the hang of. The result, however, is a graphical program that is portable and can run remotely. X-Windows is generally slower than raw SVGA, as one might expect. There is, however, an extension (called Xshm) that Linux and most Unix systems include, which supports fast bitmap updating as occurs commonly in games.

In the “cheat box” there is also Tcl/Tk, a front end language for writing easy X-Windows interfaces. Its applicability to a given program is very hard to summarise. However, applications that are basically modal can generally make best use of Tcl/Tk. Using Tcl/Tk to write front ends for programs is covered in “Using Tcl and Tk from Your C Programs”, *Linux Journal* issue 10, February, 1995.

Alan Cox has been working on Linux since version 0.95, when he installed it to do further work on the AberMUD game. He now manages the Linux Networking, SMP, and Linux/8086 projects and hasn't done any work on AberMUD since November 1993. In real life he hacks ISDN routers for I2IT.

Resources

A commercial clone of Borland's BGI library is also available for Linux. If your program uses BGI graphics, this may be an attractive option. A shareware version (US$15 registration) should be available at sunsite.unc.edu in the file /pub/Linux/apps/graphics/bgi_library.tar.gz by the time you read this.

The sunsite.unc.edu ftp site also carries a wide variety of database tools, from simple libraries for handling reading/writing PC-style XBase files to SQL systems.

A Commercial package called FlagShip for porting clipper database programs directly to Linux is available. A demo version is available from ftp://ftp.wgs.com/pub2/wgs/Filelist

Two Eiffel Implementations

Dan Wilder

Issue #17, September 1995

Dan's previous article (*Linux Journal* #14) explains why Eiffel is a language of great interest. In this article Dan reviews two Linux implementations of Eiffel-3.

The two packages reviewed, ISE Eiffel-3 from Interactive Software Engineering (ISE), of Santa Barbara, California, and TowerEiffel from Tower Technology, of Austin, Texas, have much in common, including:

- A short “Ace” file that describes the makeup of a system of classes comprising a program and that contains options to the compiler, most critically, the name of the root class and its creation procedure, and the class directories. Based on the Ace file, the compiler handles all dependency analysis itself.
- A command line interface (not evaluated in this review). The graphical interfaces are nice, but not absolutely needed, and you can get around them if you find they are getting in your way.
- Between 50KB and 100KB of static runtime binary libraries. Both ISE and Tower Technology plan to offer shared libraries in the ELF binary format when ELF stabilizes.

Class Libraries

Eiffel class libraries are delivered in source or sometimes encrypted source form, with one file per class. Lots of classes are present in such a library, offering things such as linked lists, trees, file I/O, iterators, classes for object persistence and transmission, and much more.

Except for the kernel library, which furnishes hooks to the operating system and the runtime binary library, class libraries are planned to be portable across compilers. With the adoption of PELKS, the Proposed Eiffel Kernel Library Standard, all vendors now move toward a common kernel library interface supporting their other class libraries.

TowerEiffel

The Tower development environment is tailored to lemacs or its successor, xemacs. Neither is included in the distribution, but these editors are are commonly available from ftp sites and CD-ROMs. Look for lemacs-19.10 or xemacs-19.11. You can also use emacs, though the fit is less precise.

Tower is working on a non-emacs integrated development environment, to be in beta test by press time.

Installation was smooth, except for a little difficulty in the license setup. After some exchange of e-mail, a working license code was installed and TowerEiffel was up and running.

Running TowerEiffel

Program edit, class browsing, compilation, and debug are initiated from lemacs pull-down menu selections. A nice touch is the “Send performance report to Tower” menu item which e-mails a bug report. I mailed some, and got immediate responses.

The “Browse” menu item gives you library browsing capability, as well as allowing you to work with your own classes. Some very useful views were missing from the menu. Tower agreed, and had already implemented the most difficult of these. More choices will soon be available.

With all optimizations enabled, a small test application compiled from scratch in about one minute. The resulting binary used about 77KB of code space. A rough breakdown of this showed most of the code coming from the runtime library.

With all assertion checking on and no compiler optimizations, the binary file from compiling this application occupied about a megabyte. It still took one minute to compile from scratch.

Compiler errors list to a window, with explanations and references to *Eiffel: The Language*. If you click on the errors, the related program text appears in another window.

The Tower debugger, egdb, adapted from GNU gdb, is a command-line symbolic debugger. **Xemacs** or **xegdb** provide graphical front ends. Breakpoints may be set at any reasonable line of the Eiffel source code. When execution encounters a breakpoint the corresponding program text appears in a window. You can single-step one Eiffel statement at a time, and the cursor in the display window steps to the statement executed. When execution enters C code functions, egdb becomes a C source code debugger. All the other amenities familiar to gdb users are present.

There were a few mysteries about how to display the contents of some objects. Setting some breakpoints was also obscure. Other than that, egdb was straightforward. Source code awareness was good, and the ability to switch over to C mode was also nice, as time-critical or platform dependent parts of Eiffel systems are often implemented in C.

The TowerEiffel tools are rich in run-time options, not all accommodated in the menus. In particular, report generating tools, such as eshort, can produce output in text, LaTeX, plain roff, manpage roff, or LaTeX, or Rich Text Format for use in Word or NeXT. The options are well documented in **man** pages.

Tower Class Libraries

The Tower class library has 102 classes, including the 47 kernel classes, providing the standard Eiffel language hooks into the underlying runtime library and operating system, as well as certain basic amenities, such as **ANY**, the ancestor of all user classes. Also present here are classes for strings, numbers, arrays, and so on.

Another 22 encrypted classes comprise a subset of the Booch components from Rational (Santa Clara, CA). These provide data structure and support classes giving you containers, lists, searching and sorting, pattern matching, and so on. Tower sells the full set of Booch components in unencrypted form.

Finally, 16 classes provide for storage and transmission of objects, and 10 more provide support functions.

Customer Service

Tower staff were consistently prompt in responding to e-mail, even when the answer to a question was not immediately available. The longest delay between sending mail and receiving a response was about four days, for a non-urgent matter. Most e-mail responses were within 24 hours. I phoned Tower only twice, about matters unrelated to any preceding e-mail. The person I wished to speak with answered the phone himself the first time, and the second, returned my call within a couple of hours.

Interactive Software Engineering

Interactive Software Engineering is the company of Bertrand Meyer, Eiffel's designer.

Included with the pre-release document set I read were *Eiffel: The Language, Eiffel: The Reference, Eiffel: The Libraries,* and *Eiffel: The Environment,* all by Bertrand Meyer.

The Language is the definitive work on Eiffel, required reading for every Eiffel developer. Best read after obtaining some initial familiarity with the language, it is the language standard, and the place where all discussions of the fine points turn for answers.

The remaining books are necessary if you plan to use the ISE package, and are worthwhile reading in their own right. Bertrand Meyer's writing is often entertaining, always very insightful.

Installation

Interactive Software Engineering Eiffel-3 version 3.3.4 for Linux arrived on a DAT tape. I took it over to the *Linux Journal* offices to read it onto disk, then compressed it to six floppies for **klister** to read. Unpacking was easy, and I filled out the registration form and sent it off by e-mail. In due time the license codes arrived by return e-mail, I installed them and fired up the license manager daemon, and ISE Eiffel-3 was up and running.

Running ISE Eiffel-3

Program edit, class browsing, compilation, and debug are performed from **ebench** or the tools it launches. These static linked Motif applications require no Motif shared libraries.

The user interface is thoughtful and innovative. Many of the things you need to do are initiated by pushbuttons or drag and drop. Dragging is performed without holding down a button, by clicking button three once to drag and again to drop, reducing the carpal tunnel wear and tear. Drag is easily aborted by clicking button 1. Pushbuttons have distinctive icons, and a phrase describing each pushbutton appears in the tool frame when the cursor is over the button. Those who have worked with graphical tools will appreciate that multiple level menus, pop-ups, and pull-downs are few. Instead, most of the controls needed at any point are immediately available on the frames of the tools present on the desktop.

You can edit program text using the native Motif capabilities, or you can click on a button that launches your favorite editor in a subshell. vi is the default editor. If you prefer emacs, you might run it in its own window to avoid starting it every time you want to edit a file.

ISE includes the following tools:

- Project tool: Launch other tools, debug programs.
- System tool: Edit Ace file; look at system statistics. Provide class names, for launching or targeting class tools.
- Class tool: Edit and browse classes.
- Feature tool: Closer look at features, including ancestry.
- Object tool: Examine objects at debug time.
- Explanation tool: Help on compilation errors

The class, one of the more interesting tools, gives you alternate views of a class at the push of a button. Views include:

- Text: Editable class text.
- Short: Feature definitions and feature header comments.
- Short flat: Short, with all ancestors rolled in.
- Ancestors: Indented list of all ancestors.
- Descendents: Indented list of descendents.
- Clients: List of all class clients in system.
- Suppliers: List of classes whose client the target class is.

And more. Within each of these views, many things can be clicked and dragged to various tools. The effect is a very good hypertext view of a system, including the supporting classes from the libraries. Users of ctags may know what I mean if I call this “ctags on steroids”.

My small test application took sixteen minutes to compile from scratch with full optimizations and no assertions. The resulting binary file was about 130KB.

With assertions enabled and optimizations off, full compile time from scratch was about six minutes, and the binary file was about 1.7MB.

ISE ebench uses “melting ice technology”, which allows incremental changes to run in an interpreted mode. Only modified classes are recompiled. Changing one class and clicking the Melt button caused only a few seconds of compilation. The entire application can be run in this mode, which runs slower but compiles faster. My test application took 20 seconds to compile from scratch in “melt” mode.

Compiler errors are shown in the project tool, whence you can drag and drop class and feature names into class tools, letting you find the affected program code quickly. For more information, drag and drop the error code itself into an explanation tool, and very likely you'll see a paragraph and page number from *The Language*.

Debug uses the same drag and drop technique used elsewhere. Breakpoints can be set regardless of whether the system is executing, in any class whose text is accessible to the tools. At each breakpoint the stack frame is shown, and features may be displayed by dragging from the stack frame to an object tool. Once displayed, further references may be followed by dragging from the object tool to itself. For example, to traverse a linked list, drag its root object to an object tool, then repeatedly drag the forward link to the same tool.

A breakpoint can be set only at a routine's entry point. As most Eiffel routines are short, this gives better granularity than a C source debugger able to break only at function entry points, but can sometimes require insertion of feature calls and recompilation. Single stepping is available, but I was unable to discover any indication of which source statement had been reached at a step.

The EiffelBench environment is a very nice piece of work. If you are interested in integrated development environments, ISE Eiffel-3 is worth the purchase price just to get a good look at what they've done here. While not perfect, this environment gives a stable, quick, and user friendly, but not confining, basis for doing projects in Eiffel.

ISE Class Libraries

The ISE class library is large, with 752 classes, all in source form. Of these, the ones of most general interest will be the 157 classes in the base hierarchy, comprising the kernel classes, support classes, and containers and data structures. Beyond that there are libraries for building parsers, GUIs under X, Motif, or MS Windows, or for use in EiffelBuild, their Motif GUI builder.

Customer Service

ISE gave a satisfactory response to every significant question I have raised to date. Sometimes this required persistence on my part, and once or twice it required a phone call to follow up the e-mail.

> Dan Wilder has been employed as a software engineer since 1975. Dan resides in Seattle, where he divides his time between work, family, and a home Linux system. Any time left is spent ignoring his moss collection, which his neighbors think he calls a lawn. He can be reached via e-mail as dan@gasboy.com.

Resources

Linux Tips

Robert A. Dalrymple

Issue #17, September 1995

Familiar tools, such as textedit and commandtool, provide some comfort to those who are used to a Sun system.

The OpenLook window manager (olwm or olvwm) makes your Linux box look like a Sun workstation. Familiar tools, such as textedit and commandtool, provide some comfort to those who are used to a Sun system. A couple of tips can raise this comfort level even higher.

For example, the Slackware distribution redefines some of the keys across the top of your PC keyboard so that you have the cut, copy, and paste functions that are part of textedit. To see this, examine the .Xmodmap file in your root directory, which is run when olwm starts:

```
! F1=Help (move pointer on panel, press F1 to show
! help on the item)
! F2=Find (after having selected some text, press F2
! to do a search)
! F3=Cut  (select text, press F3 to move text into
! clipboard)
! F4=Copy (select text, press F4 to copy text into
! clipboard)
! F5=Paste (insert text from clipboard at caret
! position)
keysym F1 = Help
keysym F2 = F19
keysym F3 = F20
keysym F4 = F16
keysym F5 = F18
```

So, some of the same functions are available, but with different keys than on the Sun keyboard. But what about that critically necessary key for undo? To get it, add the following to your .Xmodmap file:

```
! F6=Undo
keysym F6 = F14
```

This change will be implemented the next time you fire up Open Look.

To provide a key (**f8** say) to pop up a buried window, add the following to your .Xdefaults file:

```
OpenWindows.KeyboardCommand.RaiseLower:   F8
```

This is especially important if you have AutoRaise active, which immediately brings forward the window your cursor is in (In .Xdefaults this is **OpenWindows.AutoRaise: True**). Now type **xrdb .Xdefaults** to make the changes immediately.

Does your keyboard have the caps lock key where your Sun has the control key? Do you keep hitting the wrong one? No problem to interchange them–insert the following lines in your xmodmap file (taken from the man page for xmodmap):

```
! Swap Caps_Lock and Control_L
remove Lock = Caps_Lock
remove Control = Control_L
keysym Control_L = Caps_Lock
keysym Caps_Lock = Control_L
add Lock = Caps_Lock
add Control = Control_L
```

Do you want to define a meta key? **keysym F9=Meta_L** placed in the .Xmodmap file will do it. This gives you some flexibility with textedit commands, which are defined also for a **meta** key combination—**meta**-x for cut, **meta**-v for paste, etc. See the man page for textedit for more. This meta key would also be available for emacs. Another option for the meta key is to define it in the Keyboard section of /usr/X11R6/lib/X11/XF86Config (as root, of course):

```
LeftAlt   Meta
```

The XFree86kbd man page describes other keys that can be defined there.

Finally, if you use a Sun machine at work, change the .Xmodmap files on it, so that the function keys at the top of the keyboard are defined the same way on both machines, just in case you get too used to the Linux keyboard layout! This may entail using keycodes rather than keysyms:

```
keycode 13 = F19
keycode 15 = F20
keycode 17 = F16
keycode 19 = F18
keycode 21 = F14
```

Robert A. Dalrymple teaches coastal engineering at the University of Delaware. His address is rad@coastal.udel.edu; also http://coastal.udel.edu.

Moo-Tiff Development Environment

Bogdan Urma

Issue #17, September 1995

Priced at US$99, Moo-Tiff is so far the cheapest solution for Linux users who want to have a Motif development environment on their PCs.

Moo-Tiff is a new distribution of OSF/Motif 2.0 for Linux, distributed by InfoMagic Inc. in the U.S. and by Lasermoon Ltd. in Europe. (For a more in-depth explanation of Motif, see my review of SWiM and MetroLink Motif in the July 1995 issue of the *Linux Journal*.) Priced at US$99, Moo-Tiff is so far the cheapest solution for Linux users who want to have a Motif development environment on their PCs.

Moo-Tiff comes on a single CD with installation instructions on the CD sleeve and also on the CD itself. Similar to the MetroLink and SWiM distributions, Moo-Tiff comes with the Mwm window manager, shared and static libraries, header files, the Uil compiler, Mrm, man pages, and source to the OSF/Motif demos.

What it doesn't include is the OSF Motif 2.0 User's Guide booklet that accompanies both the SWiM and MetroLink distributions. This is not surprising, since Moo-Tiff is at least $50 cheaper than the other distributions. The Motif 2.0 User's Guide is, however, supplied in Postscript format on the CD, as is other OSF documentation, including the Motif Programming Guide. Moo-Tiff is geared toward an X11R6 system, but also includes X11R5 libraries for users still running X11R5. Also included on the CD is the complete XFree 3.1.1 distribution, ELF libraries, and an archive of freeware Motif programs. There is also an option to run Moo-Tiff directly from the CD, by mounting the “live” directories from the CD on your system.

Installation involves simply running an installation script, which takes care of everything for you. While poking around in the installation directory, I noticed a striking similarity between Moo-Tiff and SWiM. It seems that the Moo-Tiff libraries are identical to SWiM's. In fact, many of the installation packages were identical to the ones on the SWiM CD. Since Lasermoon is involved with distributing both SWiM and Moo-Tiff, this is not really surprising. There are differences, however, and these lie mainly in the location of the installed components and Moo-Tiff's cleaner installation. Moo-Tiff's installation went by very smoothly, without any errors or quirks. [Lasermoon says that the libraries are on different development paths and may diverge, and there is no guarantee that applications linked with SWiM shared libraries will work with Moo-Tiff, and vice-versa.[Ed]

After installing Moo-Tiff, I fired up Mwm and went straight to the installed demos directory to compile some of the demos and test the product. Using the supplied mxmkmf program, all I had to do was type **mxmkmf** to turn Imake files into Makefiles, then **make** to compile. Sure enough, the demos compiled and ran correctly. I also recompiled most of the Motif programs I already had, such as Mosaic, just to make sure things worked okay. They did.

Moo-Tiff seems to be a great product, especially considering its $99 price tag. I have had no problems with it at all, except for one little thing. One of the installed programs, xmbind, is linked with the X11R5 libraries instead of the X11R6 ones, and I couldn't find one linked with the R6 libraries anywhere on the CD. This is not that big a deal, except that I must have R5 shared libs on my system if I want to use xmbind.

Support and update/bugfix information for Moo-Tiff can be had from:

```
   (UK)
   Lasermoon Ltd
   2a Beaconsfield Road,
   Fareham, Hants
   PO16 0QB ENGLAND
   Phone: +44-0-329 826444
   Fax: +44-0-329 825936
   E-mail:
   info@lasermoon.co.uk
   (USA)
   Infomagic, Inc.
   PO Box 30370
   Flagstaff, Arizona 86004-0370
   Phone: 800-800-6613 or 602-526-9565
   Fax: 602-526-9573
   E-mail:
   support@infomagic.com
```

Bogdan Urma is studying physics and computer science at Cornell University and hopes to get his BS by next year. He has been using Linux since 1993 and spending way too much time with it. He welcomes your comments by e-mail sent to burma@newton.ruph.cornell.edu.

Reader Survey Results

LJ Staff

Issue #17, September 1995

Responses showed that the first priority for Linux users is the porting of a WYSIWYG word processor.

Last March we asked readers for feedback about what they wanted to see happen with Linux. We got responses from all over the world—Israel, Italy, Belgium, UK, Netherlands, Japan, Thailand, Austria, Germany, and the US. Although our survey wasn't entirely scientific, this response hints at the wide-spread implementation of Linux.

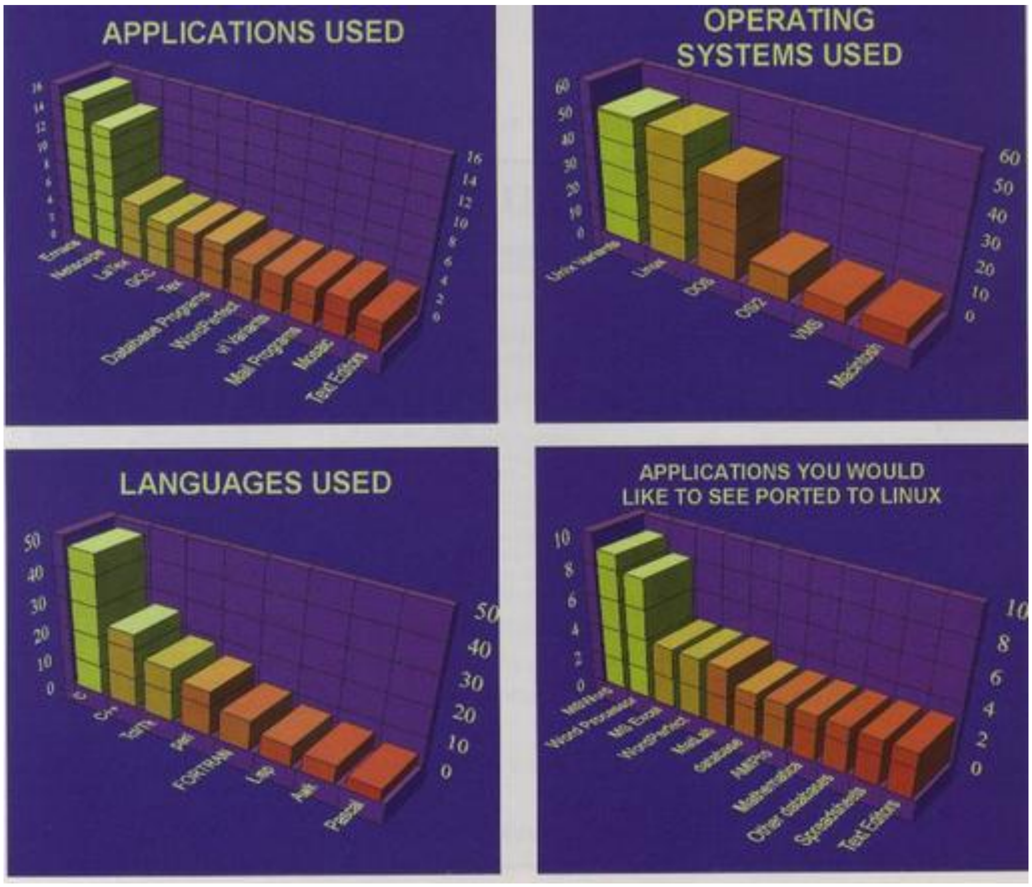
Responses showed that the first priority for Linux users is the porting of a WYSIWYG word processor. Microsoft Word was the number one choice, followed by WordPerfect, AMIPro, and FrameMaker. The second category on the wish list was a good database program, with the number one choice Microsoft Excel. Requests for other ports included CAD programs, Motif, fax software, and Lotus 1[hy]2[hy]3. See “Stop the Presses”, this issue, for an update on WordPerfect for Linux, and “Moo-tiff Development Environment”, this issue, and “Motif for Linux”, July, 1995, for reviews of Motif ports for Linux.

Emacs was the most frequently-listed application used by readers. Maybe it has a built-in command for filling in “emacs” on any questionnaire. Netscape was the next most frequently-listed application. Many other Net and Web tools were included in the 63 different applications people listed. Text processing tools, e-mail tools, graphics tools, and databases were also popular. Only four readers listed vi variants, maybe because vi users think of it as a utility rather than an application. This will probably provide plenty of fuel for Emacs versus vi discussions.

Readers used a montage of operating systems—Linux was the most popular, of course, with other flavors of Unix (SunOs, Solaris, Ultrix, Dynix, etc.) and DOS also showing up frequently. OS/2, VMS, Mac/OS, Windows NT, and NeXT were also listed. Twenty-six different operating systems, including all the Unix variants, were listed.

Linux Journal readers program in a variety of languages. C was most popular, followed by C++, Tk/Tcl, perl, and FORTRAN. FORTRAN? Twenty-five different languages were listed.

If nothing else, these results show that *Linux Journal* readers, and by extension Linux users in general, are a diverse bunch.



Casting the Net: From ARPANET to Internet and Beyond

Danny Yee

Issue #17, September 1995

In a couple of places Salus pretends he's writing a book for the masses—at one point he devotes a couple of pages to explaining the difference between datagram- and circuit-based networks, for example—but this pretence is not maintained—while Casting the Net doesn't assume a great deal of technical knowledge, it is still very much a technical history, written for those who work with networks and networking protocols.

- Author: Peter H. Salus
- Publisher: Addison-Wesley
- ISBN: 0-201-87674-4
- Reviewer: Danny Yee

In *A Quarter Century of Unix* Peter Salus explored the history of Unix; in *Casting the Net* he turns to the history of the Internet. After a brief look at the "prehistory" of networking, he covers the development of the ARPANET in some detail. He then discusses a variety of material, organised thematically and roughly chronologically: early networks in Europe and Japan (but nothing about Australia); the development of new protocols (particularly for mail); the switch to TCP/IP; the OSI protocol wars; UUCP and Usenet; BITNET and Fidonet (and a bit on IBM's VNET); the NSFnet; the NREN and the NII; the most recent commercialisation and explosion of the Internet; and so forth. Information is current up to December 1994, so *Casting the Net* is not **too** badly out of date.

In a couple of places Salus pretends he's writing a book for the masses---at one point he devotes a couple of pages to explaining the difference between datagram- and circuit-based networks, for example---but this pretence is not maintained---while *Casting the Net* doesn't assume a great deal of technical knowledge, it is still very much a technical history, written for those who work with networks and networking protocols. For example, the book includes all the

April Fools' Day RFCs, material that can hardly be appreciated by anyone who's never read an RFC or tried to understand a networking protocol.

Whereas *A Quarter Century of Unix* was built out of quotes, more of *Casting the Net* is taken up by diagrams, time lines, and digressions. Most of these are reprinted from easily accessible sources (like the digressions, many of the quotes are from RFCs), so there is a lot less original material than in the earlier book, and I don't think it is as impressive an achievement. It includes a lot of good material, however, and it's a good read; once again, I finished it within a day of receiving my copy. As a compact technical history of the Net, it doesn't have much competition.

Archive Index Issue Table of Contents

Advanced search

The Unix Philosophy

Belinda Frazier

Issue #17, September 1995

This book satisfactorily explains how and why the Unix operating system developed as it has.

- Author: Mike Gancarz
- Publisher: Digital Press
- ISBN: 1-55558-123-4
- Price: $19.95
- Reviewer: Belinda Frazier, info@linuxjournal.com

I was first drawn to this book by the title and by the following in the introduction: “The creators of the Unix operating system started with a radical concept: They assumed that the users of their software would be computer literate from the start.” Of course, it is just this assumption that helped keep Unix from being used very much outside the technical, university, computer-literate environment for many years. This book satisfactorily explains how and why the Unix operating system developed as it has; compelling arguments explain why the Unix philosophy allows for good software design and why, in the author's opinion, Unix will become the world's operating system.

The book is geared mostly toward readers who haven't yet used Unix, but the author also intended that experienced Unix programmers would reread it several times so as not to forget the nine main tenets of the Unix Philosophy, as well as the less stringently “enforced” minor tenets. (I think enforcement is accomplished by “true” Unix programmers looking down their noses at those who go astray from the true Unix philosophy.)

The book credits Stephen Bourne, William Joy, Brian Kernighan, and Dennis Ritchie, among the many people who contributed to the Unix Philosophy, each by giving some major contribution to Unix in the early days. For example, William Joy brought text editing and C-like command language (he developed vi and the C-shell); Dennis Ritchie brought us the C programming language.

The main tenets (each of which have sub-tenets) of the Unix philosophy are as follows:

1. Small is beautiful.
2. Make each program do one thing well.
3. Build a prototype as soon as possible.
4. Choose portability over efficiency.
5. Store numerical data in flat ASCII files.
6. Use software leverage to your advantage.
7. Use shell scripts to increase leverage and portability.
8. Avoid captive user interfaces.
9. Make every program a filter.

The author introduces each tenet with a simple, real-world example (or “case study”) , then further explains why the tenet is important by including non-technical computer-world examples.

Tenet 1. Small is beautiful.The book offers an example of how Volkswagen ran an ad campaign with the phrase “small is beautiful” in the US to promote the VW bug, but the idea was generally ignored in the US until the price of oil went up and Americans learned the advantages of small cars. The author draws an analogy to these nouveau small-car-appreciators to programmers at AT&T Bell Labs discovering that small programs were also easier to handle, maintain, and adapt than large programs.

In a non-Unix environment, a program to copy one file to another file might include, as in an example given in the book, twelve steps which do more than perform a file copy. The twelve steps perform extra tasks, some of which are considered “safety features” by some. The steps might include checking to see if the file exists, if the output files are empty, and prompting users to see if they know what they're doing (for example, “Are you really really sure you want to do this, and does your mother know you're doing this?”), etc. Just one step of the sequence might be the actual copy command. A Unix program (or command) would only include the one copy command step. Other small programs would each do the other 11 steps and could be used together if the Unix user wanted to use these extra steps. Although the author purposefully steers away from giving Unix examples until near the end of the book, I would have liked to see several Unix commands strung together to accomplish all the tasks described by the twelve steps.

Tenet 4. Choose portability over efficiency.The example given here is of the Atari 2600 which was the first successful home video game. Most of the code for the game cartridges was very efficient but nonportable. With the advent of new hardware (the “5200”), the code had to be rewritten to run on the 5200 which took time and money. The author proposes that Atari would have been the largest supplier of software in the world if its code had been portable.

There is a three-page analogy of selling Tupperware to the “use software leverage to your advantage” tenet. Who would have realized a multilevel marketing scheme is a good way to write software?

A sub-tenet of the leverage tenet is allow other people to use your code to leverage their own work. Many programmers hoard their source code. The author states that “Unix owes much of its success to the fact that its developers saw no particular need to retain strong control of its source code.” Unix source code was originally fairly inexpensive compared to the cost of developing a new operating system, and companies started choosing Unix as the platform to build their software on. Companies who chose Unix spent their effort and money on developing their applications, rather than on maintaining and developing an operating system.

There were a few too many pages in this book attempting to reach the stubborn Unix-haters, for example trying to soothe the ego of the programmer who measures him/herself by the number of pages to their large programs. I was slightly put off the book by the psychological explanations of how programmers might behave until I realized these were gentle nudges—and Unix-aficionados are not known for gentleness in talking about other operating systems—that might actually get through to even a lifetime-Brand X operating system user.

I'd strongly recommend this book for people from all operating system environments.

Letters to the Editor

Various

Issue #17, September 1995

Readers sound off.

Power Failures?

I like *LJ*, especially the articles on “real world” applications of Linux, as well as those on how some of the software works (e.g., majordomo, Tcl/Tk, RCS) and on how to do things (connect to the Internet).

I don't actually have Linux yet. Truth is, I don't have a computer yet, but I am putting a plan for getting one before the Household Steering Committee soon. Wish me luck.

I do have a major question about Linux. I know that, with Unix, and unlike DOS, you just don't unplug your computer or switch it off. I believe the procedure is to do a “sync” before shutting it down. I presume Linux is the same. What happens when we have a power out(r)age? What kind of a mess will I have to clean up, what data could I lose, and how can I protect myself? Is tape backup the only way? How about a power-failure detection system? A daemon to do periodic syncs? Perhaps *LJ* could run an article on cost- and time-effective ways to ensure maximum up-time and minimum disaster potential.

—Charles L. Hethcoat III

We do wish you luck in your hearings before the Household Steering Committee.

What is missing from your question is what happens to MS-DOS machines if they are in the middle of disk access during a power outage. It turns out that the standard “ext2fs” filesystem under Linux is more stable in the face of disaster than the DOS filesystem. Given the same usage patterns, a Linux machine is much less likely to loose files and data than a DOS machine.

If properly configured, a Linux machine automatically checks when it boots to see if it was cleanly shut down. If it was not, it automatically checks the filesystems to make sure that they are not damaged, and fixes them if they are damaged.

Those who use DOS's SMARTDRV write-caching also have to do the equivalent of a sync before shutting off their computers.

Unix has always had a daemon, called **update**, to do periodic syncs. Linux also has it.

The one thing (besides backing up, which is always the first point of defense) that will almost entirely prevent any filesystem damage is a quality uninteruptable power supply (UPS). In the 9 months that I have had my computer connected to a UPS, I have not lost a single file due to any of the frequent power failures in my area. To be fair to the Linux filesystem, I had only ever lost one file to a power loss—and even that may have been user error.

UPSs range in price from around US$100 to US$1000 for reasonable choices for home and small business use, and will save you from almost anything save a direct lightning strike, from which only off-site backups will protect you. Most UPSs have a connector (usually at extra cost; some connectors you can make yourself if you are technically inclined) that can report power loss to a program running on the computer. We do intend to have an article on that, written by someone who has significantly enhanced the Linux **powerd** daemon.

Where Credit is Due

I wish to thank you for the coverage you gave to our Linux sessions in DECUS at Washington D.C.

About the only change that I would make to the article is to publicly recognize the efforts of Kurt Reisler, Chairman of the Unix Special Interest Group of the U.S. Chapter of DECUS.

Kurt first pointed out the interest in Linux to me and suggested that Linus come to New Orleans DECUS. Kurt's efforts “inspired” me to fund Linus' first trip to DECUS, which consequently drove the funding of the Alpha system for Linus to start his port. Kurt directed the schedule for DECUS in D.C. as well as hosting most of the Linux activities. Kurt has been a tireless advocate of Unix for as long as I have known him, which is over ten years.

I ask that you print this letter, as I believe very strongly in credit where credit is due.

Jon “maddog” Hall, Senior LeaderUNIX Software Group, Digital Equipment Corporation

Mea Culpa! Kurt's efforts certainly deserve recognition and respect, and I was amazed to re-read my article and find that I had left out any mention of him. Thank you very much for bring this mistake to my attention. I apologize profusely.

Three Dimensions

I just received the July issue of *LJ*. My only complaint is that it's so good, I rip through it as fast as I can! The reviews of X servers are well-timed. Also, (I work in a virtual reality lab) I think you need some reviews of the OpenGL ports available. Perhaps an explanation of them, and some of the other (free) 3-D renderers out there (Mesa, etc). Maybe even an article on writing 3-D applications under one of the packages...

Also, Greg's article on X setup is beneficial to say the least! Every time I help set up someone's X windows, they ask me how I learned all that; I reply, “The hard way!” The interview with Mr. Zborowski is good; not too many people know the roots and beginnings of the software packages they're using.

Good work!

Trent Tuggle, tuggle@vsl.ist.ucf.edu

I'm glad you find *LJ* useful. We do plan to have reviews of the OpenGL ports, and we have a standing request for a Mesa developer to write an article on Mesa, which already looks very good and shows a lot of potential for further development.

You Say Potato, I Say Potahto

I just do not understand what your magazine is all about. Perhaps it's me, but I have good knowledge of Unix and some of its derivatives, and Linux is still a mystery to me. I'm not even sure of the name; is it pronounced “Lynux” or “Linux”? Anyone I've asked is not sure, but I have received both pronunciations.

Then there's the content of your magazine. Someone must be interested in the esoteric content since there does not seem to be a shortage of these articles in your publication, especially those using acronyms that I suppose make sense to someone.

I hoped some articles would cater to the “uninformed” like me who are just getting into Linux and would prefer not to decipher every article just to find it does not matter.

I want system administration tips and tricks; comparisons to Unix functions; commercial applications that have been tried and found useful; and evaluations of the various flavors of Linux. What about an article about Caldera? Speak in regular English using terms that are recognizable to us Unix aficionados.

Irwin Luchs, San Diego, CA

Everyone who's been confused about the pronunciation of Linux will be happy to know that there is a sound file of Linus pronouncing Linux, in English and Swedish, uranus.it.swin.edu.au/~jn/linux/saylinux.htm. We pronounce it with a short i, as in “lint” and a u as in “under”. As with most things related to Unix and Linux, there's more than one right way.

We strive to include articles that appeal to our diverse readership, but we realize that not every reader will be interested in every article in every issue. We try to not leave acronyms unexplained, but we don't catch every one.

We do have a semi-regular column on system administration, as well as reviews of products and Linux distributions. An article about Caldera is in the works. We always welcome article ideas, and many of our articles result from reader suggestions.

Stop the Presses

Phil Hughes

Issue #17, September 1995

Bad news and good news.

First, the Bad News

Regrettably, we had to increase our subscription price as of September 1. While paper costs have risen steadily for some time, we have until now kept our original subscription price. But our last printing bill increased 20 percent because of higher paper prices, and we are no longer able to absorb the extra cost. The price for a one-year subscription in the US is now US$22; outside North America it's US$32. See the new insert card for other subscription rates.

Digital Releases BLADE

Digital has released a preliminary "end-user" release of Linux/Alpha, called BLADE—short for "Basic Linux/Alpha Distribution Eyesore". As you can probably tell from the name, the "end-users" in mind are developers. It is designed to install on a "NoName" AXPpci33 motherboard with a SCSI drive. Networking is not completely working as of this writing, although some capability, including telnet, ftp, and rlogin are now working. X-Windows functionality is not yet ready to be included in the distribution, but work on both networking and X-Windows is progressing.

Following are the minimum hardware requirements:

- Digital AXPpci33 motherboard with SRM console firmware
- 8MB or more main memory
- 1.44MB floppy drive
- 100MB or larger IDE or SCSI hard drive (340MB or larger suggested if you're going to do any kind of serious software development)
- VGA video board and monitor
- Keyboard

WYSIWYG for Linux

One of the most requested products to be ported to Linux is a WYSIWYG word processor (see “Reader Survey Results”, this issue). Caldera has announced that it has contracted with Novell to port and develop **WordPerfect 6.0 for Linux**. According to Caldera, their native port for Linux will be available some time during 4Q95 and will include HTML authoring tools to allow users to prepare documents for the World Wide Web.

Caldera is also porting the OpenDoc engine to Linux, and will be providing it and their own ORB (“Object Request Broker”, an important facility upon with OpenDoc is built) as part of the Caldera Network Desktop.

OpenDoc provides a vendor-independent way for applications to work together. For those familiar with Microsoft's proprietary OLE (“Object Linking and Embedding”), OpenDoc provides all the services provided by OLE, and more. It is developed and endorsed by a large consortium of companies, including Novell, IBM, Apple, and now Caldera, and it runs on Unix and Unix-like operating systems as well as MacOS, MS Windows, and OS/2.

Columnist on Leave

Mark Komarinski, author of *Linux Journal*'s “Linux System Administration” column, is writing a book on Linux, and so has suspended his column for a few months. We expect to welcome Mark back at the end of this year.

Linux at Open Systems World

Open Systems World (OSW) is hosting its Second Annual Linux Conference at OSW '95 in Washington, DC. As we did last year, *Linux Journal* will be sponsoring and organizing the event, which will be held on November 13 and 14. OSW will continue through Friday, November 17.

Like last year, the two-day conference will include one day of sessions and tutorials and a one-day class for novice and intermediate Linux users. This year, the schedule is more streamlined, with more time allotted for questions and answers than last year, as so many attendees requested.

The sessions on Monday will include a panel of several companies which are commercially involved with Linux in different ways. They will present what they do with Linux and then participate in a panel discussion. Linux International, a group which promotes Linux for both personal and business use, will give a presentation detailing its activities and its plans for future activities. Author Matt Welsh will give a short class on porting Unix applications to Linux, and there will be several other presentations as well.

Monday night, a BOF (Birds of a Feather) session will be held. Those intending to attend the conference who wish to also attend the BOF session are encouraged to send e-mail to info@linuxjournal.com so that we can schedule an appropriate and convenient meeting space.

Tuesday, there will be an all-day tutorial entitled “Linux for the New User”. Topics will range from “What and Why Linux?” through choosing a distribution, installing networking, installing and configuring the X Windows System, and finding the Linux applications you need.

On both days, the format will be open, and questions from the audience will be gladly accepted. Time has been set aside for Q&A sessions, as well.

Details are available on the WWW from www.mcsp.com/OSW-FedUNIX.html, or you can send e-mail to oswinfo@mcsp.com. Otherwise, you can send mail to Open Systems World, Inc., 10440 Shaker Drive, Suite 103, Columbia, MD 21046, fax 301-596-8803, or phone 301-596-8800.

Archive Index Issue Table of Contents

Advanced search

Databases

Dean Oisboid

Issue #17, September 1995

This month Dean gives us a tour of databases available for Linux, from the novice perspective.

This article will continue a mini-expedition of sorts through the abundance of offerings found in Linux. Last month I gave a rough accounting of spreadsheets and text editors available. The listing was nowhere near complete and was almost certainly out-of-date by printing time. The same caveat will apply here as well.

My new fountain for material is Infomagic's four-CD Linux Developer's Resource—Live Slackware, mirrors of Tsx-11, Sunsite, GNU archives, and a variety of other topics.

It's easy to get lost in the CD maze of Linux programs and my glasses are already thick from too many hours in front of a computer. The following is not meant to be a definitive listing, just a gentle survey of some of what is available.

As I did in last month's column, I want to stress that the following products are not commercial, although be commercial versions may be available. Also, let me state my expectations to make you are aware of my viewpoint. As a DOS database enthusiast, I expect many of the Linux database offerings to be on par with, say, a dBase II or dBase III, and perhaps compatible with them. Kind of like what I found with the spreadsheets being on par with early versions of Lotus-123 or Excel. Also, I hope for an interface nicer than the infamous dBase dot-prompt.

The term “relational databases” refers to a way of “joining” databases based on common (and usually indexed) fields. For example, a physician database might link to a patient database with the common field of physician ID appearing in both. When you choose a physician (thereby also setting an ID) all of the patients with the same physician ID will be also be selected. Not the best example, but it conveys the idea.

Take a deep breath, and onward.

mbase (v5.0) is a relational database system originally written for the Amiga and ported to other platforms. It uses a C-like format to do the database programming and, unfortunately, that's as far as I could go with it. To compile required **ncurses** which I installed but apparently not into the directories expected by the mbase Makefile. Editing the Makefile didn't help much either. mbase is beyond this novice, at least setting it up is. I don't think mbase will meet my expectation for a friendly interface if it uses a C format. As a final punch, when I removed ncurses Linux started to behave weirdly. Strange, since I used the installpkg and removepkg utilities included in Slackware and they're pretty reliable (assuming they are the problem). I reinstalled ncurses and now Linux is happy. [ncurses is becoming the standard for Linux, which should make some of these problems go away. For more information on ncurses, see page [??]--Ed]

lincks (v2.2 pl 1) is an OODBMS or object oriented database management system. It's covered in the March, 1995 issue of *Linux Journal*, but I made the mistake of *not* reading the article before installing. Consequently I gave up in frustration after a very short time, confused over trying to get the RPC portmapper to portmap. I'll stop here and refer people interested in lincks to the afore-mentioned article. However a thought flashed through my head: I'm looking for a single-user type database system and many of these are multi-user, client-server Leviathans. I have an inkling that many more won't work because they expect some sort of network.

onyx (v2.24), a database prototyping program, is also based on C-like language. It was reviewed by the program author, Michael Kraeh, in the very first issue of *Linux Journal*, March, 1994. Single users can use it as a database program, I guess, but I wasn't able to use it. The documentation didn't explicitly say how to compile or how to get started, quite an annoyance for this novice. I did figure out that **make config** would start the process, and a nice structured series of questions popped up. These questions would configure onyx but nowhere is there help as to what they refer to. I wasn't surprised when it bombed. With the swap file active I tried a **make all install** which ground for awhile but accomplished nothing. I took a final glance through the various files using **mc** as my snooping instrument but discovered nothing to help. Since onyx is still a work in progress like many Linux programs, I continued on to the next offering.

postgres (v4.0.1a) and **Ingres** (v8.9) I'm lumping together. Both are apparently high-end, multi-user, client-server systems. Both have installations that stumped me because I'm a single user and didn't install any network programs. Ingres, to make things tough, recommends looking in setup.doc for installing instructions. Guess which file didn't exist? While neither of these of packages will meet my xBase expectations for ease (I'm guessing on this), I do hope that an expert with some free minutes will write a couple of in-depth articles on how to use both of these programs. I heard of postgres and Ingres even before fiddling with Unix so I'm curious about them. [There is a relatively new project to create “Postgres 95”, which is intended to be a more modern, bug-fixed version of postgres. It should also be easier to build than the old “University postgres”—Ed]

pfl (0.2 alpha) untars nicely but the doinst.sh script didn't work for me. According to the documentation, pfl is similar to Lisp which tells me that this may not be suitable for impressionable novices and certainly not what I'm looking for in an easy to use program. That is, I doubt it's xBase compatible.

qddb (1.41.3 beta) stands for “Quick and Dirty Database”. It's a database development system that uses but doesn't require Tcl. However it does require bison 1.22 and flex 2.4.1 to compile. After so many packages my heart is beginning to fade. I admit I looked at what it takes to configure this package and didn't even give it a try. It looks *that* scary and something tells me qddb won't be what I have been expecting anyway. Yes, I'm being an outright wimp.

DBF (1.5) calls itself the x-base manipulation package and surprisingly (after so many novice failures) it actually worked for me. It's a small collection of programs that manipulate dbf files. For example, dbfadd adds a record or layer of information to a database. dbflst lists the records in a database. dbftst lists the structure. And there are other such utilities. I wouldn't want to manage a complicated database system with this package but for quick and easy fiddling with a dbf file, it works, and the programs don't take up much hard drive space.

typhoon (1.06) is another relational database system. One of the text files states that it was a rush job meaning incomplete documentation to make poor novices sweat. I liked the honesty and hated the reality. Like many of the previous database packages, it just wouldn't compile correctly. The process takes two steps. The first, **configure**, runs cleanly. The second, **make**, crashes with some sort of menu. I tried **make all** and **make install**, both under /usr and /usr/local, but to no avail. I may go back for another try after fiddling with some of the other packages because typhoon was inspired by Raima's db_VISTA, a system I'd like to try. I assume the one major difference between this and db_VISTA is the C structure for all processes. typhoon, according to its literature, can handle a variety of unions, relation keys, and variable types. Referential integrity is also part of a good list of features. The author is currently working on, among other things, import and export functions. It all *sounds* impressive! If it would only compile for me.

FlagShip is a good-for-10-days demo and reading some of the docs gave me great hopes and familiar ground to explore. It offered CA/Clipper and dBase compatibility, an ability to include C code and with included utilities can even access FoxPro programs and databases! Could this be my Grail achieved? It started out easy enough. Copy the multi-meg fs4lix.gz1 and .gz2 to /tmp/FS and concatenate them together with:

```
  cat fs4lix.gz* > fsdemo.tar.gz
```

After that I ran FSinstall which walked me through a menu. And then my luck failed. The program did something to my screen so that I couldn't see what I was typing; rebooting cleared the problem. Going to /usr/FSsrc, I compiled a sample program successfully but the result was a file called a.out which again killed my screen when I tried to run it. Fine, maybe FlagShip is expecting to run under X-Windows. Delete a.out, set the swapfile active, into X-Windows, compile another test program and, yup!, another a.out which weirds up the system. Luckily, exiting X-Windows restored things. Perhaps it may be the ncurses that I previously installed for mbase. FlagShip seems to want a different version of curses and unfortunately I don't have the know-how to test this idea.

Farewell to FlagShip. At least the package includes a useful though tedious FSuninstall. It asks you for every single FlagShip-related file whether you want to delete it. 2.9 billion prompts for deletion or it felt that way after awhile. But of all the programs I looked at, FlagShip held the most interest for me as a visitor from DOS and probably would be what I would purchase if I needed a Unix database system. Of course, I would have someone else install it.

My expectations for something familiar weren't met. None of the packages looked like a dBase clone except for FlagShip which I never got to any sort of visual state. Yet where I was expecting an immature field it looks or *sounds* like these programs are fairly sophisticated (although quite a chore to compile and install).

But unmet expectations weren't the frustration. It was the general lack of success. I make an effort for this column to end with some sort of achievement. Sure, I may end up re-installing Linux twelve times, but always I've ended positively as a sort of morale boost for us intrepid novices. Not this time. Except for the DBF set of xbase utilities, this Casey struck out, overwhelmed by intense installation instructions.

What also bothered me is that many of the packages—and I am not just referring to databases—lack decent documentation. For a novice, two crucial items of information are *where* to unzip, uncompress, or untar the packages and *how* to compile, particularly if the Makefile requires or expects things in a certain way. I applaud the ".lsm" files which accompany many zipped and compressed files on the CDs and on the Net. They give some help in figuring out what goes where before you actually unzip.

I can hear some of the software authors saying "So what? The hackers, the experts can figure things out," and this may be true. But Linux has been getting great reviews and mentions in many popular and diverse magazines, and with that you're going to get people other than experts snooping around and wondering, "Why the uproar?" These people represent the future audience of Linux and like it or not, these Linux novices will have to be attended to in terms of easier installation routines and clearer documentation. It's part of the price of success!

Next month: more mischief and a happy ending.

Dean Oisboid, owner of Garlic Software, is a database consultant, Unix beginner, and avowed Sandman addict.

New Products

LJ Staff

Issue #17, September 1995

Metro Link OpenGL for Linux, E&S OpenGL For Linux and more.

Metro Link OpenGL for Linux

Metro Link released a software-accelerated version of SGI's OpenGL for Linux. Metro OpenGL (MOGL) is an environment for developing 2-D and 3-D graphics applications. OpenGL runs on a variety of platforms without the need to rewrite applications for each system's graphics driver. Rotating, bouncing blobs can be displayed and interactively controlled on screen via 250 OpenGL routines. Objects can be rendered in wireframe, shaded-solid, and transparent modes–with user-definable textures, surfaces, and other attributes. Metro OpenGL for Linux is US$199.

Contact Metro Link Incorporated, 4711 North Powerline Road, Fort Lauderdale, FL 33309. Phone: 305-938-0283. Fax: 305-938-1982. E-mail: holly@metrolink.com. WWW: www.metrolink.com/.

E&S OpenGL For Linux

Evans & Sutherland Computer Corporation (E&S) and Portable Graphics, Inc., a wholly-owned subsidiary of E&S, announced the availability of OpenGL for Linux from E&S. OpenGL for Linux is a software implementation of the OpenGL Sample Implementation from SGI that runs as an extension to the standard X-Windows package on Linux. It can be used to write, compile and run OpenGL applications. OpenGL for Linux passes the OpenGL conformance test suites for all currently shipping servers. A LinkKit is provided to allow users to configure additional extensions and video drivers as needed. E&S Open GL costs US$79.00 plus shipping and handling.

Contact Portable Graphics P. O. Box 161002, Austin, TX 78716. Phone: 800-580-1160 (512-306-0460). Fax: 800-580-0616 (512-306-0016). E-mail: linuxogl@portable.com.

Interface Object Server System (DIOSS) Version 1.0

DIOSS Corp. has released Distributed Interface Object Server System (DIOSS) Version 1.0, a development and delivery system for X/Motif applications. DIOSS lets you create X/Motif based applications and does not require linking in X/Motif object code. DIOSS separates the application from the windowing system overhead by providing a Remote Procedure Call (RPC) based interface object background daemon which services all interface creation requests and events (call it a Motif Server).

Contact Tom O'Neil. Phone: 703-671-0706. E-mail: dioss@in-tech.com. WWW: http://www.in-tech.com/ . The Personal Edition of DIOSS Linux is US$99. The Corporate Linux version is US$1995.

Archive Index Issue Table of Contents

Advanced search

System Calls

Michael K. Johnson

Issue #17, September 1995

Functions in the Linux kernel can be called by user programs. Howerver, it takes a bit of preparation. In this column, Michael guides you through the process step by step, explaining why as well as what.

Code in the Linux kernel can be executed in two basic ways. One is to be called by an interrupt, and the other is to be called from a user program (that's my required “white lie” for this column). User programs call code in the kernel through a system call, which is essentially an unusual type of function call.

Of course, when user code calls privileged kernel code, the kernel has to very carefully check the validity of its arguments in order to avoid accidentally doing harm of any sort. If the code is not safe for anyone but the superuser to execute, there are routines for checking that, too.

In the Kernel

Creating a system call is more difficult than creating a normal C language function, but not too difficult. There is certainly more to it than declaring a function in a header file—and for system calls, the only change that is needed to a header file is *not* a function declaration.

The first thing that you need to do is either modify an existing file in the kernel, or create a new file to be compiled. If you create a new file, we will assume that you are able to add it to the appropriate Makefile and use the proper **#include** statements for the code you are writing. You *will* want to make sure that **<linux/errno.h>** is included, because system calls need to be able to return error codes, and those error codes are all defined in errno.h.

You will need to create a function called **sys_*name***, where ***name*** is the name of the system call you are creating. The function must have the return specification **asmlinkage int**, and it may have any number of arguments between 0 and 5, inclusive. The arguments must all be the same size as a **long**; they may not be structures. (Or, at least, not structures larger than a **long**. It would not be wise to make structures the same size as a long because integer arithmetic is done on them. What is a “signed” structure? If you don't want to think about that question, do not use small structures. In truth, don't use them at all.)

The function will return errors as **-E*NAME***. Negative numbers are treated as error values on return (we will see how later) and positive numbers are considered normal return values. This means that on systems with 32-bit **long** values, only 31 bits are available for passing back return values. On 64-bit systems like Linux/Alpha, only 63 bits are available. This makes it difficult to pass addresses in the high half of the range back to user programs.

There are two ways around this. One is to make one of the function's arguments be the address of a user-space variable in which to place the return value. The other is to find some other way of returning an error and making a special way of handling the return value. The first way is, to the best of my knowledge, always preferable, so I will not explain the second way.

Possible Errors

Before reading or writing any area in a user program from the kernel, the **verify_area()** function must be called. In normal use on a 486 or Pentium, it is less important for kernel stability than on the 386 (although it helps detect errors much more cleanly and avoids having processes die in kernel mode), but on the 386 it is absolutely essential to system stability, because the 386 does not honor memory protection when it is in “supervisor” mode, which is the mode the kernel runs in. This means, for instance, that the CPU will happily write to read-only user-space memory from the kernel.

The **verify_area()** function takes three variables. First is one of **VERIFY_READ** or **VERIFY_WRITE**. Second is the address in the *current user program* that is to be verified. Third is the length of the memory area you wish to read or write. It returns 0 if the memory area is valid, and **-EFAULT** if the memory is not valid. A common phrase is something like this:

```
int error;
error = verify_area(VERIFY_WRITE, buf, len);
if (error)
        return error;
...
```

Please note that **verify_area** only verifies addresses in user memory space, not kernel memory space. Memory in kernel space is never swapped out, and is always readable and writable. On the i86 family, the fs segment register is used in the kernel to select the user-space memory of the current process. Other architectures do this differently. This functionality is abstracted out into a few useful functions, explained below.

Your work when writing your system call will be much easier if you do as much testing as possible before committing any resources to the task at hand. As a general rule, tests are done in this order:

Run all necessary **verify_area** tests.

Do (almost) all other tests in an appropriate order, including normal permission testing.

Do **suser()** or **fsuser()** tests if appropriate. These should only be called after other tests have succeeded, because BSD-style root-privilege accounting may be added to the kernel at some point. See the comments in include/linux/kernel.h.

The **suser()** function is used to determine if the process has root permissions for most activities. However, the **fsuser()** function must be used for all filesystem-related permissions. This difference allows servers to assume the file permissions of a user without “becoming” the user, even briefly. This is important because if the server exchanges uid's such that it “becomes” the user for even a moment, the user can disturb the process in various ways, potentially breaching security in many ways. By simply using the fsuid and fsgid functions instead, the server avoids this security nightmare. For this to work, all kernel filesystem permissions testing must use the **fsuser()** function to test for superuser status, and must look at **current->fsuid** and **current->fsgid** for normal permissions on filesystem objects. (For more details on the **current** pointer, see the definition of **task_struct** in include/linux/sched.h.)

A good example of a program that needs this ability is the nfs server. Early versions of the nfs server were not able to use this functionality (because it didn't yet exist), and there were several security holes. The most common nuisance was users noticing that they could kill the server.

After you check permissions and any other possible error conditions, you probably want to actually get something done. Unless you simply want to return a value that can fit in a 31-bit (or 63-bit for Linux/Alpha) return value, you will need to write to the user memory that you checked with the **verify_area** function at the beginning of function. You can't just use the pointer to user-space memory as a normal pointer. Instead, you have to use a set of special functions to access it. And if you want to read any user-space memory in order to do your system call, you will need to use a similar set of functions to do so.

In older versions of Linux (through 1.2.x), you had to specify what kind of memory access you were making. There were 6 functions for single memory access: **get_fs_byte**, **get_fs_word**, **get_fs_long**, **put_fs_byte**, **put_fs_word**, and **put_fs_long**. These names (and names with the **fs** replaced with **user**) are still supported in newer kernels, but starting with Linux 1.3, they are deprecated. The **get_user** and **put_user** functions are to be used instead. They are easier to read and for the most part easier to use, but because they depend on the type of the pointer being passed to them, they are not tolerant of sloppy pointer use. (This is probably a good thing, since Linux now runs both on little- and big-endian computers, and big-endian computers are not tolerant of sloppy pointer use either.)

The memory block access routines have stayed the same since the earliest versions, even though their names still contain the letters “fs”; **memcpy_tofs** is used to copy a block of memory to user space, and **memcpy_fromfs** is used to copy a block of user memory to memory in kernel space.

All of the memory access routines are defined in include/asm/segment.h—even on architectures without segmentation. On all of the non-Intel architectures, these functions are essentially null functions, since they do not implement segmentation.

Up to this point, you have simply implemented a new function in the kernel. Simply prepending the name with **sys_** will not make it possible to call the function from user code.

You need to make two additions within the kernel. The first is in include/linux/unistd.h, right near the end. You need to look for the last line that starts with **#define __NR** and add your own:

```
   #define __NR_name      ###
```

where *###* is the number one greater than the previous last system call number. In version 1.2.9, that would be 141.

The second change will have to be made in multiple files, one for each architecture that Linux runs on. Each file arch/*/kernel/entry.S will need an additional entry in its system call table. The system call table is kept at the end of the file, and you will simply need to add an entry at the end of the table before the **.space** line and change the **.space** formula at the very end to reflect the new number of system calls.

Invoking Your System Call

Now you *can* call your new function from user code, but how? You can't simply declare **extern int sys_*name*(int arg);** and link. Instead, you have to **#include <unistd.h>** and use the appropriate **syscall*X*()** macro, where ***X*** is the number of arguments the system call takes. The **syscall*X*()** macros are actually defined in include/asm/unistd.h, which gets included by **<unistd.h>** automatically.

If your system call is declared as

```
asmlinkage int sys_name(void);
```

the **syscall0()** invocation is quite easy:

```
_syscall0(int, name)
```

(notice the leading underscore). This gets converted by the C preprocessor into

```
int name(void)
{
long __res;
__asm__ volatile ("int $0x80"
        : "=a" (__res)
        : "0" (__NR_name));
if (__res >= 0)
        return (int) __res;
errno = -__res;
return -1;
}
```

on Linux/i86. Because it uses assembly, it will be different on other architectures. Fortunately, it doesn't really matter. The important point is that it creates a function called ***name*** which generates an interrupt (remember the "white lie" about interrupts? System calls are interrupts, too) which calls the system call, and then returns the result if the answer is positive, and returns -1 if the answer is negative (has the high-order bit set), setting **errno** to the non-negative error number.

If your function has two arguments:

```
asmlinkage int sys_name(int num, struct foo *bar);
```

you would instead use this:

```
_syscall2(int, name, int, num, struct foo *, bar)
```

which would expand to:

```
int name(int num, struct foo * bar)
{
long __res;
__asm__ volatile ("int $0x80"
        : "=a" (__res)
        : "0" (__NR_name),
          "b" ((long)(num)), "c" ((long)(bar)));
if (__res >= 0)
        return (int) __res;
errno = -__res;
return -1;
}
```

Notice the unusual way of specifying the arguments to the macro, where the return type and the name of the function are followed by separate arguments for the type and name of each of the system call's arguments. Figuring out how to specify system calls with 1, 3, 4, or 5 arguments is left as an exercise for the reader.

For the curious: there is one other way that system calls may be called on Linux/i86. iBCS2-based programs call system calls with an **lcall 7,0** instruction instead of an **int $0x80** instruction. The **lcall** instruction takes slightly longer than the **int** instruction, which is why it is the default system call mechanism on Linux, but both are supported. The **lcall** instruction isn't exactly an interrupt, although it acts much like one; technically it is a “call gate”. So my “white lie” isn't really a lie after all.

Michael K. Johnson is the Editor of *Linux Journal*, and pretends to be a Linux guru in his spare time. He can be reached via e-mail as info@linuxjournal.com.