Advanced search

Linux Journal Issue #4/August 1994

Archive Index

Advanced search

EZ as a Word Processor

Terry Gliedt

Issue #4, August 1994

In this article, Terry Gliedt introduces us to the Andrew project, and gives us a taste of one aspect of it: the ez editor, which can edit text and graphics, and can be used as a word processor.

AUIS had its roots in 1982 when Carnegie Mellon University and the IBM Corporation decided to jointly develop a campus computing facility based on personal computers to replace the time-sharing system then on campus. IBM provided not only generous funding, but also some talented individuals and access to IBM development programs.

The result was a graphical user interface we know as the Andrew User Interface System and a file system, the Andrew File System. The file system formed the basis of Transarc Corporation's Distributed File System (DFS) and is offered as part of the Open System Foundation software.

The Andrew Consortium, composed of a number of corporations and universities, funds the current development of AUIS. AUIS is available on a wide variety of platforms including Linux, AIX, Solaris, Ultrix and HP UX, as well as others.

In June, version 6.3 of the Andrew User Interface System (AUIS) was released by the Andrew Consortium. This release provided support for Linux and shortly thereafter, the package auis63L0-wp.tgz was made available at **sunsite.unc.edu** in **/pub/Linux/X11/andrew**. This particular package contains just a small portion of AUIS that is suitable as a word processor. This article will describe some of the word processing aspects of AUIS. Future articles in *Linux Journal* will describe other pieces of AUIS.

EZ is Easy

ez is the simplest, most general AUIS application possible. It loads a document (or creates a new one) and displays it in a window. That's all. Everything else that happens is controlled by the document itself. For example, if you are editing a text document, you get all the text-editing commands. If you're editing a picture, you get picture-editing commands. If you're editing a text document with pictures, you get both, depending on which piece you are working on at the time. The application, ez, doesn't care (as if any software really cares).

Figure 1. Sample ez Window

WYSLRN

At an initial glance, the document you see displayed by ez appears to be What You See Is What You Get (WYSIWYG). Upon further investigation, however, you'll see that it really is not. While the document appears with various fonts and pictures as you might see on paper, the application does not enforce any concept of a “page”. If you make the screen extra wide or narrow, the text will re-flow to fit the window, not some notation of a page. Documents are printed in PostScript™ (well, not precisely, but that is what comes out in the end), and of course the X fonts do not quite match those used when printing. So ez might better be said to be a WYSIWYG-almost editor. One pundit calls this “WYSLRN” (What You See Looks Really Neat).

The Basic Layout

In Figure 1 you can see a sample of a tiny document. The application and name of the file appear in the title bar. A small message area appears at the bottom of the document. Notice there is a menubar for pull-down menus at the top of the document. You can select these menus as you'd normally expect. As a shortcut AUIS applications will pop-up the menus when you press the middle mouse button (or both buttons simultaneously for a two-button mouse).

The scroll bar looks rather Motif-like, but behaves differently. You can scroll forward by pressing the left mouse button, as usual, but unlike some scrollbars, you can scroll back by pressing the right mouse button. In larger documents you can “grab” the “box” in the scroll bar to quickly scroll through the document.

Most commonly used functions have been assigned to a sequence of keys. Many of the common Emacs keybindings apply to ez (e.g. ^x^c). Several function keys have been assigned useful bindings. F1-3 will invoke the Copy, Cut and Paste functions. AUIS applications will remember the past few Copy/ Cut buffers and F4 will allow you to Paste them in your document and then cycle through them. This allows you to Paste two or three objects by simply pressing 4 a few times without needing to repeat the select-copy-paste sequences. I use this feature all the time and feel crippled when I work with other systems that do not have this capability.

F5 will italicize the selected text and F6 will make it bold. F7 (Plainer) will remove the last style applied against an area. For instance, if you press F5, F6 and then F7, your text will be left in italics because the Plainer function removed the bold style. F8 (Plainest) will remove all nested styles applied to the area.

AUIS applications are highly tailorable. Almost everything described thus far can be made to behave differently. The look and feel I describe here reflects the auis63L0-wp distribution. A future article will describe the various configuration files and settings and how you can control these.

This distribution of AUIS has over 800KB of help text (and this is about one-third of the total). There is extensive help information on how to use all the aspects of the system, including its configuration and personalization files. After installing AUIS, probably the first command a novice would enter is auishelp.

Figure 2 shows a second window of the same file. You can, of course, edit more than one file at a time. You can also have two windows on the same data. If you enter data or make changes in either window, the other is immediately updated (if the changes are in a part of the document that is visible).

Figure 2. Sample Split Window

Checkpointing

If you work for a few minutes without saving, ez will automatically save for you. You can tell this is happening when the message line at the bottom suddenly shows “*Checkpointing...*” and then “*Checkpointed*”. The file is written to a separate file named like the original filename except “.CKP” is appended. Thus if you edit **test.d**, the checkpoint file will be **test.d.CKP**. When you use the “Save” command, your original file is replaced with the new version and the checkpoint file is removed.

EZ Magic

It is important to understand that the word processing files created by ez have their own distinctive format, just like your favorite commercial word processors. This format was conveniently designed from the start to allow ez documents to be sent via conventional electronic mail systems.

This format is the “magic” that allows ez to be so simple. The data quite literally defines what can be done to it. As the data is read, programs that match the data are automatically loaded for you. ez is not a single program, but a group of programs that can cooperate with each other, each of which knows how to manipulate a certain type of object. If ez reads a document with text in it, the text-editing program edits the text *object*. If ez reads a document with a bitmap (picture), the bitmap-editing program edits the bitmap *object*. If your document has several different *objects*, all the different programs cooperate to edit the objects.

However, you need to do more than edit your documents. You need to print them and send them to other people, both electronically and on paper. Therefore, filters have been written to convert AUIS documents to several different formats. To print your documents, you can convert them to PostScript. To exchange documents with people running other word processing systems, you can convert them to RTF format, the most common document interchange format. You can even convert your documents to straight ASCII, though of course all your fonts and pictures will be lost.

Styles

One of the simplest things you can do with ez is to add styles to a document. Styles appear as menu items. If you look at the menus, you will find quite a few styles can be chosen. In Figure 1 you can see some of the more common styles —italics, bold, centered, or left/right justified, and various font sizes to name a few.

Most styles can co-exist peacefully—italics and bold make bold-italics. Others cancel each other out—like center and right-justify text. Some styles you can apply more than once—changing font sizes can be done by successive applications of bigger. To remove a style, select and area and chose the Plainer (F7) menu item. To remove all styles, choose the Plainest (F8) menu item. You can see what styles have been applied by placing the cursor at some point in your document and then entering “Esc-s” (press the escape key and then press the “s” key). In the message area at the bottom of the screen you will see a description of the styles that apply at that point.

Insets

In Figure 1 you can see our document has two types of data—conventional text and a small picture. Tech-nically each of these are insets to ez. An inset is just a piece of data (an object) that has been inserted within some other data (other object). An inset behaves exactly as if it were a separate document. The only difference is that it may not have scrollbars.

Figure 3. Sample Footnote and Header Insets

ez can be configured so that when a file is to be created, a certain inset will be the default. For instance, if you invoke ez on a file called test.d or test.doc, the menus will contain the text inset with menu items you need for writing a paper like this one. Of course, once the file has been created, ez will automatically use the insets indicated by the data itself. Similarly, editing a file called test.ras results in the raster inset being created. Editing test.html results in an html inset being created.

To create a new inset in a text document, move the cursor to where you want the inset to exist and select an item from the "Media" menu (or press Esc-Tab). In Figure 1, I selected the "raster" inset. The causes a raster object to be created, initially appearing as an empty box. Many insets (but not all) allow other insets to be embedded in them. After adding a raster inset, you are editing two documents—each with different attributes. Which "editor" (object) has control is based on where the cursor is positioned. In this example if you click in the raster box, the menu cards will change to those of the raster program. Click in the text area outside the box and the text menus return.

Page Breaks, Footnotes and Headers

Some insets have special meaning for the formatting of the text document. They generally have their own menus. If you select the "Insert Pagebreak" menu item, a thin horizontal line appears on the screen. This causes a new page to begin when the document is printed or previewed.

Footnotes can be added to your document with the "Insert Footnote" menu item. The footnote will appear as a small square with a star in it. Footnotes can be open or closed by simply clicking in the box. They are displayed in-line as shown in Figure 3, but print at the bottom of the page, as you would normally expect.

A page header or footer can be specified for your document by inserting the Header/Footer inset on the Media menu card. As shown in Figure 3, you can specify various fields. There are special $-keywords you can use for variable data. Select the AUIS Help menu and then select Show Help On... When prompted, enter headers to get more detail on headers. Alternatively, you may enter auishelp headers in an xterm window.

Having It Your Way with Styles

The list of styles that come with ez is usually sufficient for simple documents. However, you may need better control of tab stops, paragraph indenting or outdenting, or double spaced lines. Perhaps you want your own combination style that sets the margins, font, point size all from a single menu item. All this and more can be done by selecting the “Edit Styles” menu option. This brings up a second window which is divided into several panels, as shown in Figure 4.

Figure 4. Edit Styles Window

The style editor window has its own menus. The top part of the window, above the double line, is used to select a style to edit. The left-hand panel is a list of all the menu cards that have style options. If you select a menu card name, it is highlighted and its list of styles appears in the right-hand panel. Once you select a menu card and style name, the rest of the panel becomes active. It displays the attributes that apply to the selected style.

To change the attributes of a style, highlight the different options in the attribute panels. The document will not change until the document is redrawn. You can select “Update Document” to force the redraw.

Choose the menu item “Add Style” to create your own style. When you do this you are prompted in the message area at the bottom of the screen for a menu name. You should enter a pair of names like “**Region,MyWay**”. This will add a menu item “MyWay” to the “Region” menu. You should then select the attributes you want for this new MyWay style.

When you edit or add styles with the style editor, the changes only affect the document you are currently editing. They are saved in the data and will exist when you edit the document later. If you copy text containing the new style and paste it in a new document, the style will be transferred to the new document, but the attributes of that style will not be transferred. This is because the attributes of the style are saved at the top of the file in an area you cannot see. So when you copy the selected area, you only get the name of the style, but not the definition of the style (which you cannot see). You will need to either add a new MyWay style to the second document or learn about templates.

Templates

A template is a set of ready-made formatting information or instructions that you can apply to a text document. Templates describe two types of information:

- Style specifications, which determine what formatting styles (such as boldface and centering) can be applied to a document, and exactly how they change the appearance of the text to which you apply them. Thus templates lend a standard appearance to documents by making a style always look the same.

- Set text, which is text that you want to include over and over in many documents. A template saves you time in this case because you only have to type that text once. ez gives you an extensive set of templates. Most of these include style specifications only, but some also include set text (e.g., the template used for creating memos). Creating a template is described in detail in the on-line help (enter the command auishelp templates in an xterm window).

Printing and Previewing

The entire AUIS system is designed to print using PostScript. This was a decision made many years ago and is still in transition. In this version AUIS objects all generate troff output—along with copious amounts of embedded PostScript. The troff is then processed to generate the necessary PostScript. In the auis63L0-wp distribution the default print command will invoke a shell, /usr/andrew/etc/atkprint. The default preview command calls the shell /usr/andrew/etc/ atkpreview. Each of these shells will invoke the groff formatter to generate the PostScript output. In atkpreview the groff output is piped into ghostview.

If you do not have a PostScript printer, you can modify atkprint to pipe the PostScript output through ghostscript to generate the correct stream for your printer. Andreas Klemm andreas@knobel.knirsch.de has written a filter, apsfilter (available from ftp. germany.eu.net in /pub/comp/i386/Linux/Local.EUnet/People/akl/apsfilter*), which works with your /etc/printcap entry and will automatically convert PostScript into the correct DeskJet stream. I have a DeskJet 500 printer which works very well in this mode.

There are other ways to control printing/previewing by AUIS. These will be discussed in a future article about tailoring your AUIS applications.

For More Information

A mailing list is available at info-andrew@andrew.cmu.edu (e-mail to info-andrew-request@andrew.cmu.edu for subscriptions). The newsgroup comp.soft-sys.andrew is dedicated to the discussion of AUIS. A World Wide Web home page can be found at www.cs.cmu. edu:8001:/afs/cs.cmu.edu/project/atk-ftp/ web/andrew-home.html. A book, Multimedia Application Development with the Andrew Toolkit, has been published by Prentice-Hall (ISBN 0-13-036633-1). An excellent tutorial is available from the Consortium by sending e-mail to info-andrew-request@andrew.cmu.edu about the manual, A User's Guide to AUIS.

After spending over twenty years with IBM, Terry left Big Blue last year. Although he has worked with Un*x and AUIS for over six years, he is a relative newcomer to Linux. Terry does contract programming, teaches classes in C/C++

and Unix and writes the occasional technical document. You can reach him at Software Toolsmiths, (507) 356-4710 or by e-mail at tpg@mr.net.

Disaster Recovery

Mark F. Komarinski

Issue #4, August 1994

It's happened to me more than once. It will probably happen to you at one time or another. You turn on your PC, expecting yet another fun session of pure Unix power, when something goes wrong. It won't boot; hard drive not found; it just hangs. Now what? This article will help you figure out what is wrong and get started with fixing it. Read this article before something goes wrong, and it will be easier to fix it when it happens.

Something has gone wrong. That's all you know. Staring at your blank or garbage-ridden screen, the only thing you can think is “Now what do I do?” Even if you have not had this happen yet, there is probably a good chance you will face this. With all of Linux's power, it is still rather easy for a new—or even experienced—user to make a mistake and mess up something.

With some advance preparation, this kind of situation won't leave you stranded. Make sure you know how to track down a problem, have a bootable disk, and have a set of rescue disks, configured for your particular setup.

Your first step is tracking down the problem. Do you get to the `Uncompressing Linux...' message? If not, your problem is with the boot disk or LILO. Having a spare boot disk should allow you to boot your system, and then you can reconfigure LILO or make a new boot disk.

While Linux is booting, do you get past the partition check? If so, your hard drives are probably fine with Linux. I had a hard drive once that made Linux hang when it tried to find the partitions. The drive didn't work in any other system I tested, so the drive was bad.

Also, if you get past the partition check, then the kernel is not your problem. After the partition checks are done, root is mounted and then /etc/inittab is read. As you may or may not recall, /etc/inittab is used by the init program to start login processes and begins reading your /etc/rc files to mount your partitions, start your network among other things. Once the inittab is read, it goes to the corresponding file for mounting additional filesystems, starting network services, and other startup services. If you see your filesystems being mounted, that means that some of your rc files are being started.

Once the inittab is read, it goes to the corresponding startup file (“rc file”) for mounting additional filesystems, starting network services, and other startup services. If you see your filesystems being mounted, that means that some of your rc files are being started.

Finally, make sure that your network services are starting if you want them started on your system. This is one of the final parts to the startup sequence.

Now, what do you do if you know you have a problem? Before you get into a jam, make sure you have backups. If things get too bad you can always re-initialize your partition and restore from an old backup. Also make sure to have backups handy of your /etc directory.

One good idea is to get a copy of the rescue disks available through FTP. These disks will allow you to boot linux from a pair of floppies and access most of your partitions. This way, even if you can't boot because of a bad /etc/inittab file, you can still boot linux and get access to the bad file, then fix it.

Some of these rescue disks come completely ready-made, so that you can use the rescue disks very easily. The disadvantage to these sets is that they may use an older kernel, may not have some pieces that you need (SCSI support, for example), and may not have the set of programs that you want to see in a rescue disk.

There are other sets of rescue disks where you specify which programs you want to include. They also use the current version of the kernel that you are using. The drawbacks to these are that you need to know what you are doing and they take a bit more work than simply getting a pre-built rescue disk. Two such packages are SAR (Search and Rescue) and rescue. Each of these packages is small, as they both use programs that are already on your system.

If you have two floppy drives, you can go through the rescue disk(s) and find out what programs that you'd like to add, such as your favorite editor. Usually one disk can contain all the programs you'd need in the event of a disaster, but having two disks chock full of utilities will be even better. Here's how:

First, put a floppy in your second drive. I have a 5.25 HD drive as my second floppy, so I'll use that in my examples.

The **fdformat** program is used to low-level format a floppy. Its syntax is:

```
fdformat <device>
```

where <device> is the name and type of drive you're using. For example, I have a high density 5.25" drive as drive 2, so my <device> would be /dev/fd1h1200. A high density 3.25" would be /dev/fd1H1440.

Now you put a filesystem on it. Use the same filesystem that you are using on the root partition of your system. In my case, that would be the Second Extended Filesystem (ext2). So, let's put a filesystem on my floppy:

```
mke2fs -c /dev/fd1h1200
```

Replace the /dev/hd1h1200 with /dev/fd1H1440 if your second drive is a 3.5" high density drive.

Now you should have a filesystem on a disk. Mount it on an unused directory. The /mnt directory is usually used for this. If /mnt does not exist on your system, do

```
mkdir /mnt then do mount -t ext2 /dev/fd1 /mnt
```

Your disk will now be mounted on /mnt. At this point, start copying over whatever programs you want. Make sure of two things:

1. Make sure that the shared libraries on the rescue disk will work with the programs that you put on the disk.
2. Make sure that you copy over all the files you need. Some editors have configuration files or help files you may need.

If you are using a rescue disk such as SAR or rescue, you won't need to worry about libraries and you can skip ahead a few paragraphs. Or you can read it and get a better hint about how the shared libraries work.

The idea behind shared libraries is that many common C functions get included in one file in a common location. This saves a lot of space as those common functions no longer need to be duplicated in each program binary. The drawback is that it is a tiny bit slower because now two files have to be loaded instead of one. For the toss-up between speed and size, I'll take the size, especially on a floppy with very limited space.

Another small problem with shared libraries is that programs compiled to use a new library won't work if the only library that is available is an older one. For example, a program compiled to use version 4.4 of the libraries won't work if the only set of libraries available is version 4.3. You'll wind up getting an error message about incompatible libraries. If this happens, get a new copy of the libraries or recompile the program to use an older library.

[Ed. Note: this is not strictly true. With modern libraries, the user will get a message, but the program will still try to run if all the necessary symbols are there. For instance, I'm running some binaries compiled under libc 4.5.8 which run fine with my libc 4.4.4, other than giving an error message. I don't know if you want to deal with this or not; probably not.]

To check what versions of libraries the programs are looking for, use the **ldd** command:

```
ldd <program>
```

This will return the version of libraries that the program was compiled under. **ldd /bin/write** for me returns:

```
libc.so.4 (DLL Jump 4.4pl1)
```

If the files in the /lib directory are libc.so.4.4.1 or above, it will be fine to put the `write' command on your disk. If the library needed is newer than the library on the rescue disk, then you would need to find an older version of the program and put that on the floppy. For example, if the library on the rescue disk was libc.so.4.3.1, I'd need to find an older version of write to put on the disk, or else put libc.so.4.4.1 on the disk.

You don't need to put just executables on this disk. A copy of gzip and a bunch of HOWTO files can come in quite handy as well. Here's a list of suggested files, all available through FTP or on many BBSs. Some of these files may be on the rescue disk you have. Make sure.

Take any of these editors. I find that ed is small and compact, but not much fun with heavy editing or large files. For you, joe may be worth the extra 98k it takes up. If you are unfamiliar with joe or ed, you can use vi, which is a standard program on just about all UNIX systems:joe editor 133kvi editor 101ked editor 35k

General Everyday Utilities:diff 61k (finds changes in big files)grep 61kgzip 46klilo 40kMAKEDEV 9kmknod 3k

Backup utilities:This will vary depending on how you did your backup. You may want a copy of tar, afio and ftape. Get some utilities for the filesystems you run:e2fsck 35kmke2fs 20k

Get some HOWTO files (compress with gzip for real space savings!):Installation-HOWTO 48kSCSI-HOWTO 41kFtape-HOWTO 18k

One more thing you'll want on-hand is a list of all of the cards that are in your machine, the IRQs that they use, and whether they are used by Linux or not. Sometimes a problem can be an incorrectly configured kernel or card.

If you keep these disks set aside and updated often, you'll be ready for anything that might happen.

Tip of the month: When you hit the backspace, do you see /'s followed by the character you just backspaced over? Don't you hate it, too? It reminds me of reading *The Unix Programming Environment*. Get a new copy of agetty and this should cure the problem. A copy distributed with some Slackware releases had this problem.

Wine

Bob Amstadt

Issue #4, August 1994

Wine is not an Emulator, or WINdows Emulator; whichever you call it, Wine allows you to run programs compiled for MS Windows under Linux, FreeBSD, and NetBSD. In this article, Bob Amstadt (the founder of the Wine project) and Michael Johnson explain what Wine is and how it works.

In the first few years of Linux, many people asked whether it would be possible to run MS Windows under Linux. They were tired of rebooting over and over to run applications such as word processors that are available under Windows, but not under Linux.

Since Windows (in some incarnation) runs under SCO, iRTX, QNX, and SVR4 of various flavors, why couldn't it run under Linux as well? The immediate answer was that all the other OS's that run Windows 3.1 have access to some Windows source code, and are working under “non-disclosure agreements” from Microsoft, and that Windows, as it comes out-of-the-box, takes over the entire machine. It is possible to run Windows 3.0 in “real mode”, where it doesn't take over the machine, but very few applications will run in that mode.

Others pointed out that most of the other implementations crash just as often as Windows itself does, and that Linux is a 32-bit operating system, while Windows is a 16-bit system, and that standard Windows binaries are 16-bit binaries.

Some people suggested rewriting Windows for Linux and X11, and then laughed at their own joke.

Attention turned to the DOS emulator that became widely useable about a year and a half after Linux's debut. Many were excited when, with some rather ugly hacks, the DOS emulator was able to run Windows 3.0 in “real mode”, at least for a few developers who knew what they were doing. It became apparent that there are very few useful Windows applications which still run in real mode, that it would be very difficult to get Windows to run in standard mode under Linux, and that if Windows could be made to run under Linux, it would make the system unstable.

Maybe it's Not a Joke, After All

A year ago or so (in June 1993), Bob Amstadt started writing the beginnings of Wine. He started by writing the code necessary to load Windows binaries into memory (Windows binaries are not in a format Linux understands, and need to be “fixed up” when they are run) and some code for Linux to allow modifying some of the memory management structures (the LDT) in such a way that Windows binaries could run.

He set up a structure for rewriting the rest of Windows, and a few programmers joined him in writing these functions. A year later, over 35% of the functions have been written, and many small applications run, more or less. These include the standard Windows applets Solitaire and Winmine, as well as a commercial security system application. Each week, more programs are added to the success list.

What Wine Does

Wine translates the Windows API (Application Programming Interface, how Windows applications call Windows) into equivalent functionality available through the standard Unix and X interfaces. When a Windows program creates a window, Wine converts that into a call to create a window through the standard Xlib library. When DOS interrupts are called, for example to read a file, Wine translates them into Unix system calls.

Wine also implements other API's that are available for Windows. For instance, the WinSock API is becoming the standard way of accessing TCP/IP networking under Windows, and Wine provides the functionality of WinSock, mapping it to the standard Unix socket calls.

Wine should run applications at approximately the same speed as Windows, because the application code is running on the native CPU-Wine does not emulate a 386 or 486 processor. Because Wine itself is a 32-bit application, it has a chance to be faster than MS Windows in some areas. Windows, by contrast, runs in 16-bit mode and has to do a lot of “segment arithmetic” that is not necessary in 32-bit mode.

Wine loads Windows binaries; it also loads the closely related DLLs (dynamically linked libraries) which most applications require. Wine itself uses a DLL called sysres.dll, in which are stored all the bitmaps for the standard buttons on the title bar.

Better Than Better…

OS/2 2.x was marketed as “A better Windows than Windows”, and in some ways this was true. OS/2 provides “Crash Protection”, whereby one misbehaving Windows application cannot crash another application. Windows, on the other hand, does not adequately protect applications from each other. With Wine, each program is run as a separate process, which utilizes the protection already provided by the Unix process model, where each process is separate.

Not only will you not crash your computer, you will not crash X either. X is designed to be robust, and to check the data that is passed to it by applications, so that random bad data will not cause the X server to crash.

There are additional benefits beyond “crash protection” to running each Window binary as a separate process. One is that each program has more resources. Instead of all the Windows programs running at the same time having to share GDI heap (a memory resource), each program has access to its own entire heap.

Under Windows, when an application is busy and changes the cursor to the dreaded hourglass, the user has to wait to use all the programs currently running on the computer. With Wine, when one application is busy, the user can let that application be busy while switching to another application and getting some work done.

In Unix, filesystems are “multithreaded”, which means that multiple processes may be reading and writing files simultaneously. DOS does not allow this, so Wine, by using the Unix functionality for reading and writing files, allows much faster and smoother access to files, both on the hard drive and on floppies. The user can back up to any medium and work at the same time, just as you would expect under Unix.

Running MS Windows programs on remote computers that are also running MS Windows is not easy or fun. It requires that special remote-access programs be running on both computers, and none of the many remote access programs is compatible with any of the other programs, because they all use closed, proprietary protocols. Furthermore, they usually require that the remote user be the sole user of the machine.

X, by contrast, is designed to run remotely, and to run on multi-user systems. Because Wine translates Windows calls into X calls, Wine allows the same easy remote access that X does. It is just as simple (although a little slower) for a user in Moscow to run the Windows application on your computer as it is for you, if you are on the Internet, and so is she, and you give her an account on your machine.

How Wine Does It

As mentioned before, Wine is a single process which translates Windows calls, including undocumented calls that applications need (and a few DOS int21 calls as well) to X and Unix calls, respectively. It is also responsible for properly loading Windows applications. Wine reads the executable file, and correctly loads the code, data, and resources into memory.

Wine is a single process which uses only one non-standard system call, which is required to be able to run 16-bit code instead of 32-bit code. Therefore, it is relatively simple to port Wine to operating systems which conform to POSIX (more or less), have X, and for which the source is free, or which provide an appropriate alternate non-standard system call for setting up the LDT so that 16-bit code can run. Wine was originally developed for Linux, but the port to FreeBSD and NetBSD took less than a week.

The most basic window-handling functions in Wine are written as an interface to the appropriate Xlib functions for manipulating X windows. However, wherever possible, other internal functions in Wine call the basic window-handling functions in Wine instead of Xlib. This has several benefits: it makes Wine more modular, it makes the basic window-handling functions be better tested, and it makes it less necessary for every Wine developer to have X programming expertise.

Running Wine

Wine can be run like any other X program: you can run it from the command line:

```
 % wine sol
```

or you can put it in a menu, or launch it from a file manager. When launched, programs run and act as if they were running under Windows. By default, applications come up in a special kind of window that looks very much like those provided by Windows instead of looking like other X windows. It looks as if a normal Windows window popped up in the middle of all your X windows, because your window manager doesn't put a standard “frame” on the window. Unfortunately, those windows do not interact well with virtual window managers like fvwm (they don't go away when you switch to another virtual screen), so there is an option to bring up an X window that contains the application window, and this is well-behaved with virtual window managers.

In the Future

Wine is still in ALPHA testing. It only runs a few applications (other than the test applications that come with it) at this point, although more are being added rapidly to the list. At some point, when Wine supports several major applications, it will be released as BETA software for anyone to play with. However, the BETA release will most likely be missing several features. DDE and OLE are not likely to be supported in the BETA release, and because X has no standard printing mechanism, printing will probably also be unsupported.

Development will not stop with the BETA release. If anything, it will speed up as more programmers become familiar with the project. New BETA releases will be released periodically as Wine progresses.

In the Present

To continue to develop Wine, we need your help. All the work on the project is currently being done by volunteers with Internet access, so anyone with Internet access may join the project. If you are interested, but are not skilled at Windows programming, start by reading the FAQ, available from tsx-11.mit.edu or aris.com in /pub/linux/ALPHA/Wine/Wine.FAQ.

There are several projects that can be done by newcomers to the project who are not yet skilled in Windows programming, and there are also reading recommendations for learning the Windows programming skills you need to be of more help to the project.

If you have more money than time, please consider a donation to the Wine project. Donations will be used to hire programmers to accelerate the development. If you are interested in making a donation of any size, please contact bob@amscons.com.

Bob Amstadt graduated from Rose-Hulman Institute of Technology in 1986 with a BS in both Electrical Engineering and Computer Science. For the past five years he has worked as an independent engineering consultant specializing in embedded control and communications systems. His first exposure to Linux was in December 1992 when he installed it on his e-mail server. He began work on Wine as a result of discussions on comp.os.linux in May and June of 1993.

Eagles BBS

Ray Rocker

Issue #4, August 1994

Ray Rocker describes how Linux was used to facilitate the development of one popular bulletin board system.

The Birth of Eagles BBS

It all started as a port of Pirates Bulletin Board System (PBBS) version 1.6 to ESIX/System V. Pirates BBS was written by Ed Luke at Mississippi State University in the summer of 1989, and earned its place in Internet legend through the now-defunct Mars Hotel BBS. Two friends from high school, then students at the University of Southern Mississippi (which is my alma mater as well), generated enough interest in an Internet BBS to get space on a machine in the Computer Science Department. Together we ported Pirates, which was strictly BSD code at the time, to ESIX. The Eagle's Nest BBS was born April 14, 1992.

It wasn't easy or pretty, as any early user of the Eagle's Nest will tell you, but it eventually worked. Along the way I added a lot of features from other BBSes that Pirates 1.6 didn't have, brought in things from later Pirates revisions, and even had a few original ideas. It wasn't meant to be anything but a simple PBBS port, but people started taking an interest in our "enhanced Pirates", so we gave it our own name—Eagles BBS—in honor of the Eagle's Nest, and of our school's mascot from which that name is derived.

We released Eagles BBS 1.0 in August 1992, and I've updated it four times since then. My partners have since dropped out of the project. Thanks to a lot of gracious folks across the Net, I received ports of it to quite a few other operating systems. None of these received more than casual attention, until we discovered Linux. Before I get into that, I should give a little technical information about the software itself.

What is Eagles BBS?

Eagles BBS is a self-contained, multifunction bulletin board package that is accessed by logging in to a special account, either through telnet, dial-in, or a terminal. Three executables comprise the EBBS system: bbsrf, bbs, and bbs.chatd.

bbsrf is a special setuid root login shell. A user “bbs” is placed in /etc/passwd with bbsrf as its login shell. bbsrf optionally erases the login from /etc/utmp, making it invisible to the “finger” and “who” commands. It then does a chroot to the bbs home directory, does a setuid back to “bbs”, and execs the bbs program.

The chroot is done for security purposes. With it, anyone who might break out of the bbs program into a shell will only have access to the bbs home directory, not the rest of the host system. Because of the chroot, the BBS is completely self-contained underneath the bbs account's home directory, except for the record in /etc/passwd. All files needed by the BBS at run time are copied under ~bbs; for example the /etc/termcap file is copied to ~bbs/etc, the shared C library is copied to ~bbs/lib, and so on. This is a major portability headache. More on this later.

The bbs program does most of the work. First it prompts the user for a userid and password, which are validated by a lookup in the ~bbs/.PASSWDS file. This is much like how the standard system login program works, except ~bbs/.PASSWDS is a binary file. Once logged in, the user navigates through a series of menus accessing the features of the BBS.

The menu system is full-screen, and uses the terminal capability database (~bbs/etc/termcap) for efficient manipulation of the terminal. EBBS does not use

libcurses but a stripped-down, simplified curses look-alike dating from the Pirates days. Because the user is usually coming in through telnet, the BBS cannot always tell what terminal type the user has; there is a menu option for the user to specify this, however. The menu system is hierarchical; from the top level Main Menu one can access submenus, such as the Talk Menu, Xyz Menu, File Menu, and Mail Menu. The functions available from the menus can be roughly divided into seven categories, which I'll briefly describe:

1. Sending/reading mail: Mail may be sent to other users of the BBS. In addition, a facility for forwarding BBS mail (and posts) to a remote Internet mailbox is included. Receiving mail from Internet is not supported; since users do not have real accounts insofar as the operating system is concerned, this would be a little tricky.

2. Posting/reading messages: Up to 80 boards may be set up by the operator on which users may post public messages. The operator may restrict boards to certain groups of users with permission masks. The BBS boards do not interface with outside message services like Usenet or Fido.
3. File upload/download: The BBS has a facility for file uploading and downloading with serial-line protocols like Zmodem and Kermit, allowing files to be transferred straight from home computers to the BBS. Using these protocols across a slow Internet link usually does not work very well, however. Most sites also offer ftp access to their file bases since it's more suited to long-distance network transfers.
4. Talk: An emulation of the Unix “talk” utility is available, allowing one-on-one talk sessions with other BBS users. The talk facility does not allow connections to remote systems like Unix talk does.
5. Chat: A local chat system is built into the BBS. Up to four separate rooms may be configured. A chat daemon (bbs.chatd, the third program making up the BBS) process is spawned when the first user enters a room, and serves to relay messages to all users in the room. A number of IRC-like features have been hacked into the chat facility, like private messages, actions, and commands to show who's logged on.
6. User utilities: For setting things like your nickname (not login userid), terminal type, and address for mail/post forwarding. These are on the Xyz Menu.
7. Administrator utilities: These are only accessible by privileged users. This includes things like account and board creation and removal, setting permission flags on accounts, mail cleans, and welcome screen editing.

As you can see, EBBS, like its Pirates predecessor, stands on its own for the most part—hooks to other services are few and far between. Much of this is necessary because of the chroot. Some sites have integrated Internet Relay Chat clients into their BBSes, but even then the built-in chat system is quite popular. EBBS seems to be popular among computing neophytes because of the simple menu system and builtin editor. On the same token, power users sometimes get frustrated by its limitations.

The isolated nature of a PBBS or EBBS system encourages some interesting relationships among its regular users, especially after a system has been around for a while. Some of the things that go on would surprise even the most ardent soap opera fan! I'll leave that topic open for another article, though.

Enter Linux

Back at the Eagle's Nest, in late 1993, the decision was made to switch its host machine (a 1990 vintage 25 MHz Cheetah Gold 486 PC) from ESIX to Linux, because Linux simply had more to offer. So, I was called upon to do another port. This was the kick in the pants I needed to finally buy a new hard disk and put a real operating system on my own PC. Armed with the 0.99.12 kernel and SLS 1.03 distribution, I tackled the task before me, and found it to be quite easy to go from my System V Release 4 ported code to Linux. Here is a list of the issues I ran up against, which should make an initial “gotcha” list for anyone porting a network application to Linux. You'll notice many of the same things mentioned in Michael K. Johnson's excellent “Porting To Linux” article in *Linux Journal*, Vol. 1 Edition 1.

- Signal handling. BSD, and SVR4 via the signal(2) call, have restartable signals. In Linux you must explicitly ask for BSD signal handling or use the somewhat different sigaction interface. [Editor's Note: See the last section of Linux Programming Hints on page 33.] For me, it was just easier to change the code to assume signal handlers need to be reset.
- select(2) and the timeout parameter. Like so many apps, EBBS assumed that select(2) leaves the contents of the timeval struct alone. You've got to reset the timeout if you're calling select in a loop.
- the FD_SET macro and its kin live in <sys/time.h>. Not that this should bog down anyone's porting, but it's worth mentioning since most Unixes put these macros in <sys/types.h> or <sys/select.h>.
- Terminal I/O. A lot of BSD source relies on gtty(2) and stty(2) and obscure ioctl flags. When I realized they weren't in the base C library, I changed it all over to termio. Probably a good idea anyway, even if I did have to put the BSD source from PBBS back for the NeXT port.

The Eagle's Nest went online with Linux in October 1993 and we've been faithfully keeping step with the latest revisions ever since. Compared to our experience running on ESIX, Linux has been both good and bad.

The good: the ext2 filesystem and kernel proper. Some BBS operations requiring lots of file access, notably the scan for new messages and the “visit” function (to mark all messages as read) were excruciatingly slow under ESIX, but lightning fast under Linux - with the exact same hardware, of course. A new message scan that might take ten seconds before will get done in one or two now. All the buffering Linux does really made a difference for us.

The bad: the TCP/IP networking. The Eagle's Nest sometimes gets a couple of hundred logins a day, and it shares bandwidth with anonymous ftp service and a WWW server. Hence, the kernel gets battered pretty badly network-wise. Through the releases from 0.99 pl13 to the present 1.1.8, we've seen lots of strange things like dying inetds, frequent reboots to revive the network, and connections hanging both on connect and close. Luckily, it keeps getting better, and as of 1.1.8 the only annoying problem left is occasional hanging at connect time, and only when it has to deal with long delays or lost packets. Unfortunately such conditions are not at all uncommon on Internet.

That aside, we at the Eagle's Nest are pretty happy with Linux, and I now use it almost exclusively myself. I have another operating system on my disk, but it's just a faint memory. Its name starts with D I think. Doom! Yeah, that's it!

Looking to the Future

As I mentioned already, no port of EBBS has generated anywhere close to the interest the Linux port has received. The worldwide scope of Linux is especially evident. Interest in EBBS for Linux has sprouted on every continent except Antarctica, all with very little promotion on my part! See sidebar for a partial list of Linux EBBS systems available on Internet.

Not too long after the Linux port was done, I decided that the current code was getting too messy for any more nontrivial additions or enhancements. Here are a few of the things holding advancement of the present incarnation of EBBS back:

- The chroot. It's a security feature but also a portability nightmare. Every flavor of Unix requires a different set of system files under ~bbs to allow the system to work, and indeed some haven't been figured out yet. It also makes the install script a big mess. Consider shared libraries. On some systems they're in /lib, some in /usr/lib, some in /usr/shlib. On some systems all you need is libc.so.*, on others (like Solaris) you need a half dozen or more shared libraries. It's a huge mess. In the latest version (2.2) I have added a compile time option to turn off the chroot if desired. To get rid of it safely would be even better.
- Because of the chroot, it's nearly impossible to add hooks to the outside system into the BBS. Many BBS operators have expressed the need for two-way Internet mail and Usenet, and with the chroot this is impractical if not impossible to do with EBBS. A few hooks outside the BBS proper have been successfully done, for example the outgoing mail/post forwarding, and IRC, 4m, and Gopher clients.
- Keeping track of read posts, and the .DIR files. The Pirates BBS way of keeping track of which user has read which post involves each post having a record in a .DIR file associated with each board. These records have things like the owner and title of the post, and one byte for each possible account on the system. When the user reads the post, their byte in the post's record in the .DIR is marked as such. The race conditions involved with simultaneous updates of these files are rampant and tricky to guard against. Also, since each .DIR record must be a known size, the numbers of BBS accounts is fixed at compile time, not a desirable feature.

So last November, I embarked on a complete rewrite, with the goal of keeping all or most of the existing EBBS features but a completely new foundation. Five months later I'm almost done with the first cut. The fact that this is a spare-time venture, plus a couple of rewrites from scratch when I got frustrated, have made it slow going.

The development of the new version is being done 100% on Linux, and I'm making a point to stay as POSIX-compliant as possible so ports can be done without too much hassle. These are some of the features under development:

1. No more chroot. See above. I am going to get rid of the Shell Escape menu function for this reason, as it's risky, and disallow the system(3) C library call anywhere in my code, since it spawns a shell as well. In short, I'm trying to make it impossible to break out.
2. Run time configuration, to allow binary distribution (the Linux way!). Right now, all EBBS optional features and access permissions must be configured at compile time. A set of initialization files that the BBS reads at run time will be a major improvement. In particular I am trying to make configurable anything that may vary between distributions, such as the mail program to use for forwarding. In addition, I plan to make all files used by the BBS ASCII rather than binary, so they may be easily patched with an editor if something goes awry. This includes the passwds file and boards file.
3. No more .DIR files. I will be storing the bits showing which posts/mail a user has read in a per-user config file, eliminating the need for the .DIR files. And, it eliminates the restriction on the number of accounts.
4. Separation of the user interface and BBS core, only interacting via a carefully defined API set. This will allow anyone who wants to build their own user interface, for example an X-Windows GUI, to do so easily, with only knowledge of the APIs for accessing the BBS core functions. I plan to write a network API set which will allow clients to connect to remote BBS servers via a yet-undefined TCP/IP-based protocol. This way, things like menu movements and editing may be done on the client, with none of the lag that has become all too common on the increasingly crowded “information highway”.
5. And a few more features which aren't present now: group mailing, replies by mail to post authors, per-board managers, added chat features like private rooms, and more.

At the time of this writing, this rewrite is very nearly ready for initial testing, with only a few features yet to be added. If all goes well, something will be publicly available by the time you read this. I'm tentatively calling it Eagles BBS 3.0 and plan to distribute under the GNU copyleft or something similar; everything is subject to change at this point of course.

I hope this work culminates in a useful addition to the already-impressive array of software for Linux. The demand for Linux bulletin board software is there, it seems. It's been an enlightening and educational experience for me, however time consuming. The development environment offered by gcc, gdb, and the myriad of other tools is as good as any, I think. At first the documentation left a bit to be desired but that has improved markedly. I'd like to encourage other programmers to do the same with their favorite spare-time projects-the Linux community is very responsive and helpful with everything from suggestions to development to distribution. I'm proud to be involved with it.

Where to Get Eagle BBS

Internet BBSes running Linux

Ray Rocker (rrrocker@ingr.com) (rock@seabass.st.usm.edu) currently lives in Huntsville, Alabama, USA, where he is employed as a software analyst. On those rare occasions when he isn't in front of an SVGA monitor, he enjoys good beer, weightlifting, jogging, college football and basketball, and can usually be found on the amateur radio bands during contest weekends as WQ5L.

Linux Does Comics

Robert Suckling

Issue #4, August 1994

In this article, Robert Suckling describes a “real-world” application of Linux: running a subscription service. He also describes some possibilities that he has not yet deployed, but has run tests on, and which are feasible.

A few years ago, some friends and I opened a small store selling mostly role-playing games. We stocked many other games and as a sideline some of the partners wanted to sell comics.

A little bit later, after Linux version 0.11 hit the streets, I bought my first home PC. It was a 386/33dx with 120 MB of disk and 20 MB of RAM, and for a while it was just my toy.

Back at the store, comic sales were picking up, and when you sell comics you also take subscriptions. So Tony, one of the partners, started to take lists of comic subscriptions written on paper. When the comics orders came in he would spread out all the new comics on a table and scan each list for titles on the table, and when he saw one he would put it in the subscriptions folder.

Tony started to stress out when there were about 50 people with subscriptions. The time to scan the 50 lists on paper with the 100 comic titles spread out on the table was getting overwhelming. Tony would try to hurry, but then he started to miss comics and make mistakes, and then people would ask why we did not save their comics for them.

Linux to the Rescue

At this point one of our less computer-oriented partners, Greg, tried to put the subscriptions on a spreadsheet. He tried to list each customer as a row and each comic title as a column. It did not work, because the spread sheet was too wide to print. At this point I brought in my computer from home, and Greg helped me type in the comic data. I stored each subscription in a separate file.

Now the big transformation that was needed was simply to use cat and sort. This turned lists of subscriptions (of titles) into lists of titles (of subscriptions). “The Guys” like hard copy, so they print the list sorted by titles, take each title in turn, find the list of subscriptions for that title on the printout, and put a copy into each comic slot.

My first incarnation of this was written under DOS mostly with DOS-perl. But being a Unix hacker, I missed all the Unix system calls and stuff. After playing a bit more with Linux, and after I installed the first MCC-interim Linux distribution on my PC, I switched the comic system over to Linux.

Now the comics were going to the right customers. We still used the paper pages for users to make changes to their subscriptions, though. And one of us still had to take each page each week and enter the changes into a file and print out the new subscriptions.

After a few months of editing subscription pages it became tiresome and error-prone to make all the customer changes. Customers would come in and ask us where the X-Men titles they signed up for were, and we would check their current list on the computer, and it would not be there, but the paper page where they requested it would be in the book.

I wrote a set of menus with perl (what else) that would let users delete and change their subscriptions online. So far, so good, but we could not allow a customer to add simply anything to their subscription. We could have a customer try to subscribe to something new that we could not get for them, so we had a problem with customers adding things.

My solution was that a customer could add any title that another customer was already getting. At least most of the time, if one customer was getting a copy of some item, getting another copy was not a problem. For this I took the current list of titles of things being asked for and made a file that the user could search through to get to a single item that was desired. This item then could be added to their subscription.

Something old, something new…

So now customers were able to add something old to their subscription. Now for the new stuff. The comic book distributors we use have a computerized ordering system, so they supply us with a computerized listing of new titles at the beginning of each month. This data is in the form of a DOS disk with a file of one-line entries for each title. Yet another perl script pulls out the new titles and saves them in another file like the current items listing I just talked about. Now the users can search through this file and add new titles to their subscription. The distributers also supply a 300- to 400-page catalog that we give out free to our subscription holders.

There is a two week window each month from when we get the data and the catalogs to the time our customers can add anything from the catalogs to their subscription. After two weeks we make our order to the distributor, so I delete the new titles file, and the menus detect this and no longer let customers add new stuff, so only the current old titles are available after the 15th.

I set up a dumb serial terminal in the front of the store and wrote some menus that would let customers edit their subscription file. Customers can add titles from the distributors' listings or from a list of popular titles. This way a customer cannot add a title we can not get, but he can change the number of each title or the starting and ending issue numbers, as well as delete a title. We have been using this system for nearly two years, and we now have over 200 subscribers managing over 1800 subscriptions on-line with Linux.

The competition

Last spring I attended a major comic book trade show for comic book store owners. One of the speakers sold software that could be used to maintain comic book subscriptions. I asked him if he could support customers making changes to their own subscription. He said, “No way”-he could not have some kid walking up to a DOS box, because the kid could type CTRL-ALT-DEL or something and destroy the database.

That cannot happen on my dumb terminal running from my Linux box.

In the Future

Another thing I have wanted to do at the store was run an inventory system on the gaming stock. The gaming stock, unlike the comic stock, can be reordered from the distributors on demand. My partners do not understand how easy it would be to implement an inventory system, but I have all the tools waiting for them, should they some day want one.

I have learned how to print barcodes from scratch. I plan to print product numbers on the labels with my own program so that a barcode reader can be used at the checkout counter. Given:

- a last night current inventory,
- a model inventory (what we think we should stock) and,
- a POS(point of sale)-terminal,

I can maintain a current inventory listing, and a “diff” (list of differences) of the current inventory with the model inventory as a reorder listing. This can be filtered with product listings from game distributors to make the actual reorder lists for each game distributor automatically.

Post

I wanted to know if Linux and perl could keep up with the most important store application, the Point Of Sale Terminal, so I set up a test database using perl's ndbm arrays.

One day I plan to barcode with “code 3 of 9”, a popular barcode format, which would contain the product numbers. Then a light-pen or a laser scanner can read the barcode just like you see at the grocery store. With that item code, an item's price can be looked up and an itemized receipt can be produced. More importantly, it also lets me make a record of what was sold, so we can reorder it.

I have a DOS disk from one of the game distributors that contains their full listing of products - about one Megabyte of data: about 50 characters per record, with about 20,000 records. I set up an ndbm file that mapped product numbers into item description and pricing information.

I stored each key (product number) in an array and picked a random index and looked up that key. I then put that in a loop and ran a few time trials.

With my 386/33dx system, I could access over 300 stock items in my test database per second. If you are really good, you can scan in about 1-2 items per second, and most people are much slower than that. In other words, there is plenty of time to make few database references per item. From that, I can see that there is plenty of CPU power to run a few cash registers, as well as time to edit comic subscriptions and even do some reordering, all at the same time. Remember, this is on an old 386/33dx.

Back to the Present

Every now and then I hear what a pain it is to do a reorder, but my partners are not yet ready to take the big step to automate. Some day the store will need an inventory system badly and Linux will be there, waiting for them. The store broke $1.5 million in gross sales this spring, after 3 years in business, so we may need to take this step soon.

If this interests you, here is our address and phone number:

The Streagic Castle

(Games, Comics and Miniatures) 114 North Toll Gate Road Bel Air Maryland 21014 +1 410 638 2400

Sorry, we do not do mail order.

Public Service Announcement

If there is anyone in the Bel Air, Maryland area that wants a copy of Linux, bring a big pile of floppies by the store (please call first) and I will see what I can do.

Archive Index Issue Table of Contents

Advanced search

Linus Torvalds at DECUS `94

Bob Tadlock

Issue #4, August 1994

The 1994 DECUS conference offered a large assortment of hardware and software for everyone, from the novice to the professional. DECUS is the Digital Equipment Computer Users Society and its semi-annual conference was held in New Orleans May 7-12, 1994.

The Primary Highlight for the Linux/Unix Community was the special appearance of the “Father of Linux”, Linus Torvalds. There were two sessions which featured Linus and both were very informative and entertaining. The sessions were called “An Introduction to Linux” and “Implementation Issues in Linux”. During the sessions, I learned a great deal about the Linux Operating System as well as the promising future of Linux.

According to Linus, there are over 176,000 lines of code in the kernel of which 88,000 lines of code are for device drivers. While Linus is the primary author of the kernel, most of the device driver code has been contributed by others.

The history of Linux may be short compared to the Unix time scale: three years versus twenty-five years. However, much has happened since the beginning of Linux. A few milestones for Linux follow:

From 1991 to the present, Linus has demonstrated tremendous dedication to Linux. For example, the virtual memory code was written in just three days-and it was the three days before Christmas! In three short years, Linux has transformed itself from a very simple terminal emulator to a powerful enhanced Unix operating system. A great deal of work is now being done to further enhance the existing kernel and implement new features. Some of the work in progress involves kernel threads, extended memory management, file system optimizations, and ports to other architectures such as the 68K and the Power PC and maybe even the DEC Alpha.

I had the pleasure of spending an afternoon with Linus in New Orleans, Louisiana and was able to learn a lot about Linus and what his interests are other than the Linux operating system. This was my first chance to meet Linus in person and I was shocked to find that he was not at all the “computer hacker geek” that I dared to imagine. Linus is one of the nicest and most personable people I have ever had the pleasure of spending time with. He has a wonderful sense of humor and is very funny. I found myself laughing non-stop during our chats. Linus lives at home with his family and cats and likes to spend time with his girlfriend. He teaches at the University of Helsinki.

Lately, he has been traveling around the world talking about Linux. Linus jokes that

“if you want to travel around the world and be invited to speak at a lot of different places, just write a Unix operating system.” Upon meeting Linus, one would never dream that he was the author of a Unix operating system; he never brags, boasts or even seems to want very much recognition for his massive effort.

There never seemed to be any selfish attitude at all from Linus. I wish all software developers had the attitude and determination to produce quality code such as Linux and want little, if anything, in return. Someone asked him at DECUS how he felt about other people and companies using part or all the Linux kernel for profit and not giving him any money. Linus simply replied, “I wrote Linux for public use, not for the money.”

More info

Releases

Archive Index Issue Table of Contents

Advanced search

Unix and Computer Science

Ronda Hauben

Issue #4, August 1994

Although the word Linux does not appear in this article, it does offer a lot of interesting background on Unix. That background tells us how we got to the point where Linx is a reality.

This year is the 25th anniversary of the invention of the Unix kernel in 1969 at Bell Labs. The following Work In Progress was presented at the Usenix Summer 1993 Conference in Cincinnati, Ohio. A longer paper based on this research has been proposed for the June 1994 Usenix Conference as a contribution to a 25th year commemorative discussion about the significance of the Unix breakthrough and lessons to be learned from it for making the next step forward.

The Multics (1965-68) project had been created to “show that general-purpose, multiuser, timesharing systems were viable”1. Based on the results of research gained at MIT using the MIT Compatible Time-Sharing System (CTSS), AT&T and G.E. agreed to work with MIT to build a “new hardware, a new operating system, a new file system, and a new user interface”. Though the project proceeded slowly and it took years to develop Multics, Doug Comer, a Professor of Computer Science at Purdue University, explains that “fundamental issues were uncovered” in the process of the research on Multics, “new approaches were explored and new mechanisms were invented”. The most important, he explains, was that “participants and observers alike became devoted to a new form of computing (the interactive, multiuser, timesharing system.). As a result, the Multics project dominated computer systems research for many years, and many of its results are still considered seminal.”

By 1969, however, AT&T made a decision to withdraw from the project. Describing that period, Dennis Ritchie, one of the inventors of Unix at Bell Labs writes, “By 1969, Bell Labs management, and even the researchers came to believe that the promises of Multics could be fulfilled only too late and too expensively.”2

"Even before the GE-645 Multics machine was removed from the premises," Ritchie explains, "an informal group led primarily by Ken Thompson, had begun investigating alternatives."

Thompson and Ritchie presented Bell Labs with proposals to buy them a computer so they could build their own interactive, time sharing operating system. Their proposals weren't acted on. Eventually, Ken Thompson found a little-used and obsolete PDP-7 computer, a tiny machine in the class of a Commodore 64 computer.

The environment Thompson was attempting, explains Ritchie, included "many of the innovative aspects of Multics", such as "an explicit notion of a process as a locus of control, a tree-structured file system, a command interpreter as a user-level program, simple representation of text files, and generalized access to devices"3. Describing the primitive conditions that Thompson faced when attempting to create his desired programming environment, Ritchie writes:

At the start, Thompson did not even program on the PDP itself, but instead used a set of macros for the GEMAP assembler on a GE-635 machine. A postprocesser generated a paper tape readable by the PDP-7. These tapes were carried from the GE machine to the PDP-7 for testing until a primitive Unix kernel, an editor, an assembler, a simple shell (command interpreter), and a few utilities (like the Unix rm, cat, cp commands) were completed. At this point, the operating system was self-supporting; programs could be written and tested without resort to paper tape, and development continued on the PDP-7 itself.4

The result, Ritchie explains, was that,

Thompson's PDP-7 assembler outdid even DEC's in simplicity; it evaluated expressions and emitted the corresponding bits. There were no libraries, no loader or link editor: the entire source of a program was presented to the assembler, and the output file --with a fixed name—that emerged was directly executable.5

The operating system was named Unix, to distinguish it from the complexity that burdened Multics.

As work continued on the Bell Labs operating system, the researchers developed a set of principles to guide their work. Among these principles were:

1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.

2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.6

By 1970, Ritchie writes, the Unix researchers were "able to acquire a new DEC PDP-11. The processor", he remembers, "was among the first of its line delivered by DEC, and three months passed before its disk arrived." Soon after the machine's arrival and while "still waiting for the disk," Ritchie recalls, "Thompson recoded the Unix kernel and some basic commands in PDP assembly language. Of the 24K bytes of memory on the machine, the earliest PDP-11 Unix system used 12K bytes for the operating system, a tiny space for user programs, and the remainder as a RAM disk."8 Ritchie explains, "By early 1973, the essentials of modern C were complete. The language and compiler were strong enough to permit us to rewrite the kernel for the PDP-11 in C during the summer of that year." 9 Each program they built developed some simple capability and they called that program a tool. They wanted the programs to be fun to use and to be helpful to programmers. Describing the achievements of the lab, Doug McIlroy, one of the researchers and Thompson's Department Head when he created the Unix kernel, describes the atmosphere at the lab:

Constant discussions honed the system.... Should tools usually accept output file names? How to handle demountable media? How to manipulate addresses in a higher level language? How to minimize the information deducible from a rejected login? Peer pressure and simple pride in workmanship caused gobs of code to be rewritten or discarded as better or more basic ideas emerged. Professional rivalry and protection of turf were practically unknown: so many good things were happening that nobody needed to be proprietary about innovations.10

The research done at the Labs was concerned with using the computer to automate programming tasks. By a scientific approach to their work and careful attention to detail, Bell Labs researchers determined the essential elements in a design and then created a program to do as simple a job as possible. These simple computer automation tools would then be available to build programs to do more complicated tasks. They created a Unix kernel accompanied by a toolbox of programs that could be used by others at Bell Labs. The kernel consisted of about 11,000 lines of code. Eventually, 10,000 lines of the code were rewritten in C and thus could be transported to other computer systems. “The kernel,” Ken Thompson writes, “is the only Unix code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible.”11 Thompson describes creating the kernel:

What is or is not implemented in the kernel represents both a great responsibility and a great power. It is a soap-box platform on `the way things should be done.' Even so, if `the way' is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized. (12)

The kernel was conceived as what was essential and other features were left to be developed as part of the tools or software that would be available. Thompson explains:

The Unix kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features found in most other operating systems are missing from the Unix kernel. For example, the Unix kernel does not support file access methods, file disposition, file formats, file maximum sizes, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language. Maintenance of such code is as easy as maintaining user code. The idea of implementing `system' code and general user primitives comes directly from Multics.13

During the same period that Bell Labs researchers were doing their early work on Unix, the Bell System was faced with the problem of automating their telephone operations using minicomputers.

“The discovery that we had the need—or actually, the opportunity—in the early `70s to use these minis to support telephone company operations encouraged us to work with the Unix system,” writes Berkley Tague.14 “We knew we could do a better job with maintenance, traffic control, repair, and accounting applications. The existing systems were made up of people and paper,” he relates. “The phone business was in danger of being overwhelmed in the early `70s with the boom of the `60s. There was a big interest then in using computers to help manage that part of the business. We wanted to get rid of all of those Rolodex files and help those guys who had to pack instruments and parts back and forth just to keep things going”.

He goes on to describe the kind of operations that Bell Systems needed to automate. Just as Operating Systems people in the Bell system had come to recognize the need for portability in a computer operating system, Ritchie and Thompson and the other programming researchers at Bell Labs had created the computer language C and rewritten the majority of the Unix kernel in C and thus had made the important breakthrough in creating a computer operating system that was not machine dependent. Describing their breakthrough with Unix, Thompson and Ritchie presented their first paper on Unix at the Symposium on Operating Systems Principles at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973.15

With the research breakthrough of a portable computer operating system, “the first Unix applications”, writes Mohr, in an article in Unix Review, “were installed in 1973 on a system involved in updating directory information and intercepting calls to numbers that had been changed. The automatic intercept system was delivered for use on early PDP-11s. This was essentially the first time Unix was used to support an actual, ongoing operating business.”16

The labs made the software available to academic institutions at a very small charge. For example, John Lions, a faculty member in the Department of Computer Science at the University of New South Wales, in Australia, reported that his school was able to acquire a copy of research Unix Edition 5 for $150 ($110 Australian) in December, 1974, including tape and manuals.17

By 1979, the automation at AT&T had the benefit of research done not only at Bell Labs, but also by researchers in the academic community. Early in its development, word of the Unix operating system and its advantages spread outside of Bell Labs. (Several sources attribute this to the paper that Ritchie and Thompson presented on Unix at the Symposium on Operating Principles in 1973.18)

Unix was attractive to the academic Computer Science community for several reasons. After describing the more obvious advantages like its price, that it could be shaped to the installation, that it was written in C which was attractive when compared with assembly language, that it was sufficiently small that an individual could study and understand it, John Stoneback writes:

Unix had another appealing virtue that many may have recognized only after the fact—its faithfulness to the prevailing mid-'70s philosophy of software design and development. Not only was Unix proof that real software could be built the way many said it could, but it lent credibility to a science that was struggling to establish itself as a science. Faculty could use Unix and teach about it at the same time. In most respects, the system exemplified good computer science. It provided a clean and powerful user interface and tools that promoted and encouraged the development of software. The fact that it was written in C allowed actual code to be presented and discussed, and made it possible to lift textbook examples into the real world. Obviously, Unix was destined to grow in the academic community.19

In trying to teach his students the essentials of a good operating system, John Lions of the University of New South Wales in Australia describes how he prepared a booklet containing the source files for a version of Edition 6 of research Unix in 1976 and the following year completed a set of explanatory notes to introduce students to the code. “Writing these,” he recounts, “was a real learning exercise for me. By slowly and methodically surveying the whole kernel, I came to understand things that others had overlooked.”20

This ability to present his students with a real example of an operating system kernel was a breakthrough. Lions writes:

Before I wrote my notes on Unix, most people thought of operating systems as huge and inaccessible. Because I had been at Burroughs, I knew that people could get to learn a whole program if they spent some time working at it. I knew it would be possible for one person to effectively become an expert on the whole system. The Edition 6 Unix code contained less than 10,000 lines, which positioned it nicely to become the first really accessible operating system.21

In keeping true to the Unix community spirit of helping each other, Lions wrote a letter to Mel Ferentz, Lou Katz and others from Usenix and offered to make copies of his notes available to others. After some negotiation with Western Electric over the patent licensing, he distributed the notes titled “A Commentary on the Unix Operating System” to others with Unix licenses on the conditions that Western Electric had set out.22

Describing how research Unix and its adoption at academic institutions has served to develop computer science, Doug Comer writes:

The use of Unix as a basis for operating systems research has produced three highly desirable consequences. First, the availability of a common system allowed researchers to reproduce and verify each other's experiments. Such verification is the essence of science. Second, having a solid base of systems software made it possible for experimenters to build on the work of others and to tackle significant ideas without wasting time developing all the pieces from scratch. Such a basis is prerequisite to productive research. Third, the use of a single system as both a research vehicle and a conventional source of computing allowed researchers to move results from the laboratory to the production environment quickly. Such quick transition is mandatory of state-of-the-art computing.23

Not only did research Unix serve the academic community, but the contributions of the academic community were incorporated into research Unix. An example is the work at UC Berkeley of designing a virtual memory version of Unix for the VAX computer which was later optimized and incorporated into a release of Unix. “A tide of ideas,” explains Comer, “had started a new cycle, flowing from academia to an industrial laboratory, back to academia, and finally moving on to a growing number of commercial sites.”24

Summarizing the relationship between Bell Labs and the academic community in developing Unix, Comer concludes:

Unix was not invented by hackers who were fooling around, nor did it take shape in a vacuum. It grew from strong academic roots and it has both nurtured and taken nourishment from academia throughout its development. The primary contributors to Unix were highly educated mathematicians and computer scientists employed by what many people feel is the world's premier industrial research center, Bell Laboratories. Although they were knowledgeable and experienced in their own right, these developers maintained professional contacts with researchers in academia, leading to an exchange of ideas that proved beneficial for both sides. Understanding the symbiotic relationship between Unix and the academic community means understanding the background of the system's inventors and the history of interactions between universities and Bell Laboratories.25

John Lions, reviewing his experience as part of the Unix community, concludes, “We have made a large number of contacts and exchanged a great deal of information around the world through this Unix connection. Possibly that is the nicest thing about Unix: it is not so much that the system itself is friendly but that the people who use it are.”26

It is a rare and wonderful event in the development of human society when a scientific and technological breakthrough is made which will certainly affect the future course of social development and which becomes known when its midwives are still alive to tell us about it. Unix, the product of researcher at Bell Labs, the then regulated AT&T system, and academic computer science, and a valuable invention for computer science, for computer education and for the education of the next generation of computer scientists and engineers, is such an event.

Unix and Computer Science© by Ronda Hauben, is adapted from an article to appear in the Spring 1994 issue of The Amateur Computerist newsletter and is being reproduced by permission of the editors of The Amateur Computerist. Electronic copies of the Amateur Computerist are available free from ronda@umcc.umich.edu or au329@cleveland.freenet.edu. Printed copies are available by subscription at $5.00 for a 4 issue subscription (add $2.50 for foreign postage). Checks are to be made payable to R. Hauben. For printed copies, write: The Amateur Computerist, c/o R. Hauben, P.O. Box 4344, Dearborn, MI 48126.

Also, comments are welcome on this work in progress, as a longer work is in process and there will be an effort to respond to comments and suggestions in the longer work.

References

Archive Index Issue Table of Contents

Advanced search

Linux Sound Support

Jeff Tranter

Issue #4, August 1994

Multimedia has received a lot of attention in the computer industry recently. Unix systems traditionally have not provided much support for multimedia in general, and sound in particular, except for some expensive professional systems. Workstation vendors are now scrambling to introduce multimedia-ready systems.

With the availability of Linux and low-cost sound hardware for Intel-based PCs, a sound-capable Unix system is within the reach of most computer hobbyists.

Possibly because sound support was lacking in Unix systems, many new users are confused by the technical jargon specific to sound and electronic music, and the many sound cards available. This article will to explain what can be done with sound under Linux, unravel some of the technical terms, and point the reader to sources of more information.

What can a Sound Card do?

The typical sound card hardware provides the capability for one or more of the following functions:

- playing and recording digitized sound samples
- playing music from an audio Compact Disc in a CD-ROM drive
- generating sounds using an internal FM synthesizer
- controlling external MIDI (Musical Instrument Digital Interface) devices
- miscellaneous functions, such as providing a joystick interface, SCSI disk interface, volume and tone controls, and facilities for mixing of inputs

Different Types of Sound Cards

For digitized sounds, there are two basic parameters that determine the sound quality: sampling rate and sample size.

The sampling rate is the speed at which the analog waveform is converted to digital “samples”. This is expressed in samples per second, or more often (and less accurately), Hertz. The sample size indicates the number of data bits which are stored for each sample; the more bits, the more accurately the sample represents the original waveform. Sounds can also be recorded with one channel (mono) or two channels (stereo). Various coding schemes are used to represent the sample as a numerical value.

As an example, a low-cost sound card can produce single channel, 8-bit samples at 8000 samples per second. This provides sound quality comparable to the telephone network. A 16-bit sound card producing stereo sound at 44100 samples/second is equivalent to Compact Disc audio quality (ignoring issues such as noise and distortion).

Some sound cards also provide hardware for producing sounds using FM synthesis. This technique is based on modifying sine waves. The advantage of this scheme is that the hardware is reasonably simple and not much computing power is required. The disadvantage is that it is difficult to determine the parameters needed to produce specific sounds (e.g., a piano).

Sound cards also typically provide other miscellaneous features, including joystick ports, CD-ROM interface, SCSI interface, MIDI port, facilities for sound input and output, and volume and tone controls.

Supported Sound Cards

The Linux kernel currently supports the following sound cards:

- Roland MPU-401 MIDI interface
- AdLib
- SoundBlaster and compatibles (including ThunderBoard and Ati Stereo F/X)
- SoundBlaster Pro
- SoundBlaster 16
- ProAudioSpectrum 16
- Gravis UltraSound

The Linux kernel also supports the SCSI port provided on some sound cards (e.g., ProAudioSpectrum 16) and the CD-ROM interface provided on the Soundblaster Pro and SoundBlaster 16.

For those who do not (yet) have sound hardware, there are a couple of other options. With a little hardware, a sound interface can be built using the parallel printer port. For a zero-cost solution, there is even a sound driver for the internal speaker of your PC. The driver is compatible with the sound card driver, but the quality may leave something to be desired.

Configuring Linux for Sound

Setting up Linux to support a sound card involves the following steps:

1. installing the sound card
2. configuring and building the kernel with the sound drivers
3. creating the sound device files
4. testing the installation

The first requirement, if you have not already done so, is to install the sound card. Follow the instructions provided by the manufacturer. Be sure to note down the jumper settings for IRQ, DMA channel, and so on; if you are unsure, use the factory defaults. Try to avoid conflicts with other devices (e.g., Ethernet cards) if possible. You will also need speakers, and a microphone if you want to do any recording. A math co-processor is also useful for some sound applications (e.g., changing file formats, adding effects or speech synthesis), but not necessary.

The next step is to configure the Linux kernel. If you are using a recent version (0.99 patch level 14 or later), the sound drivers are included with the kernel release. Follow your usual procedure for building the kernel. When you configure the kernel, enable the sound driver, and answer the questions about sound card settings when prompted by the configure program.

Once the kernel is configured, you need to create the sound device files. The easiest way to do this is to cut the short shell script from the end of the file /usr/src/linux/drivers/sound/Readme.linux, and run it as root. These are the files that will be created:

- /dev/audio- Sun workstation compatible audio device (read/write)
- /dev/dsp- digital sampling device (read/write)
- /dev/mixer- sound mixer
- /dev/sequencer- MIDI, FM, and GUS synthesizer access
- /dev/midi- MIDI device (not yet implemented in current sound driver)
- /dev/sndstat- displays sound driver status when read
- /dev/audio1- for second sound card
- /dev/dsp1- for second sound card

If you are using the PC speaker sound driver, then it will use the following devices:

- /dev/pcaudio- equivalent to /dev/audio
- /dev/pcsp- equivalent to /dev/dsp
- /dev/pcmixer- equivalent to /dev/mixer

Now that the kernel is configured and the device files created, you can verify the sound hardware and software. Follow your usual procedure for installing and rebooting the new kernel. (Keep the old kernel around in case of problems, of course.) Verify that sound card is recognized during kernel initialization. You should see a message such as the following on powerup:

```
snd2 <SoundBlaster Pro 3.2> at 0x220 irq 5 drq 1
```

```
snd1 <Yamaha OPL-3 FM> at 0x388 irq 0 drq 0
```

This should match your sound card type and jumper settings. The driver may also display some error messages and warnings during boot up. Watch for these when booting the first time after configuring the sound driver.

If no sound card is detected when booting, there are a couple of possible reasons. The configuration of the driver could be incorrect and the driver was not able to detect your card in the given I/O address. Another common error is not having the sound driver in the kernel, because you booted with an old kernel instead of the one that was just compiled.

Reading the sound driver status device file provides additional information on whether the sound card driver initialized properly. Sample output should look something like this:

```
% cat /dev/sndstat
Sound Driver:2.4 (Sun Feb 13 14:49:20 EST 1994 root@fizzbin.mitel.com)
Config options: 1aa2
HW config:
Type 2: SoundBlaster at 0x220 irq 5 drq 1
Type 1: AdLib at 0x388 irq 0 drq 0
PCM devices:
0: SoundBlaster Pro 3.2
Synth devices:
0: Yamaha OPL-3
Midi devices:
0: SoundBlaster
Mixer(s) installed
```

If the cat command displays “No such device”, then the sound driver is not active in the kernel. If the printout contains no devices (PCM, Synth or Midi), then your sound card was not detected. Verify that you entered the correct information when configuring the sound driver.

Now you should be ready to play a sample sound file, and send it to the sound device as a basic check of sound output, for example,

```
   % cat endoftheworld >/dev/dsp
   % cat crash.au >/dev/audio
```

Some sample sound files can be obtained from the file snd-data-0.1.tar.Z, available on many Linux archive sites.

If you have sound input capability, you can do a quick test of this using commands such as the following:

```
   # record 4 seconds of audio from microphone
   % dd bs=8k count=4 </dev/audio >sample.au
   # play back sound
   % cat sample.au >/dev/audio
```

If these tests pass, you can be reasonably confident that the sound hardware and software are working. If you experience problems, consult one of the references listed at the end of this article.

More on Music

When it comes to playing music, there are several methods that can be used.

First, don't plan on digitizing a 60 minute music CD as a sound file and storing it on your hard disk. Some simple calculations show the size of the data involved:

```
        60 minutes
     x 60 secs/minute
     x 44100 samples/second
     x 2 channels
     x 2 bytes/sample
     = more than 635 million bytes!
```

Small sound samples at low sampling rates are reasonable though:

```
       10 seconds
     x 8000 samples/second
     x 1 channel
     x 1 byte/sample
     = 80 thousand bytes
```

A more compact method of storing music is MOD files. These originated with the Amiga computer, but MOD file editors and players are now available on other systems as well, including Linux. MOD files essentially store a bank of short samples (instruments), and sequencing information (musical notes). These files are of the order of 30K to 300K bytes in size and can represent several minutes of music (i.e. a complete song).

MIDI stands for Musical Instrument Digital Interface, and as the name suggests, is a standard hardware and software protocol for allowing electronic musical instruments to communicate. MIDI files describe songs in terms of keypress events, and can be played using either an internal FM synthesizer on the sound card, or external MIDI devices.

Another, less common, file format is Adagio files. Adagio is a music description language developed at Carnegie Mellon University. Programs to play Adagio files, and convert between Adagio and MIDI files are available.

Games

We can't forget another important application for sound—games! Several graphical games supporting sound run under Linux, including Bdash and Xboing.

Sound Tools and Utilities

To make use of the sound support in the Linux kernel, applications are needed. Several of the important ones are Sox (a utility for converting sound files from one format to another and adding effects), and tracker (a MOD file player). Some graphical programs that run under X are available as well, including xmix (a sound mixer), xplay (a sound player/recorder), and xmp (a MIDI file player).

There are many others, including some interesting applications such as speech synthesis.

The Future

Now it's time for me to get up on my soapbox. Sound support in Linux is quite new, and application support is not as complete as for some other operating systems. I envision seeing a complete set of sound tools, that would provide a consistent, graphical user interface for all of the common sound functions. This would help bring Linux to the forefront of multimedia operating systems. Some of the tools that need to be developed are:

- sound player/recorder tool
- file conversion utility
- audio mixer
- MOD file player
- MOD file editor
- MIDI file player and sequencer
- FM synthesizer patch editor
- text to speech synthesizer (how about support for a /dev/speak device?)
- an X window manager supporting sound

Many of these already exist in various forms, and just need more development to be more reliable and consistent. Any volunteers?

An interesting side note is that the author of the Linux sound drivers, Hannu Savolainen, is porting them to other Intel operating systems as well. The package has been dubbed VoxWare, and should make it easier to write sound applications that are portable across several operating systems.

Sources of more information

The Linux Sound-HOWTO document provides more information on the topics discussed in this article and provides other references. Look for it on your local Linux archive site.

If you have access to the Internet, the following FAQs (Frequently Asked Question documents) are regularly posted to the usenet newsgroup news.announce as well as being archived at the site rtfm.mit.edu in the directory /pub/usenet/news.answers:

PCsoundcards/generic-faq (Generic PC Soundcard FAQ)

PCsoundcards/soundcard-faq (comp.sys.ibm.pc.soundcard FAQ)

PCsoundcards/gravis-ultrasound/faq (Gravis Ultrasound FAQ)

audio-fmts/part1,part2 (Audio file format descriptions)

These FAQs also list several product specific mailing lists and archive sites. The following Usenet news groups discuss sound and/or music related issues:

alt.binaries.sounds.misc (Digitized sounds and software)

alt.binaries.sounds.d (Discussion and follow-up group)

alt.binaries.multimedia (Multimedia sounds and software)

alt.sb.programmer (SoundBlaster programming topics)

comp.multimedia (Multimedia topics)

comp.music (Computer music theory and research)

If you have Internet mail access, then you can subscribe to the SOUND channel of the Linux Activists mailing list. To find out how to join the mailing list, send mail to linux-activists-request@joker.cs.hut.fi.

The Readme files included with the kernel sound driver source code contain useful information about the sound card drivers. These can typically be found in the directory /usr/src/linux/drivers/sound.

The Linux Software Map (LSM) is an invaluable reference for locating Linux software. Searching the LSM for keywords such as “sound” is a good way to identify applications related to sound hardware. The LSM can be found on various anonymous FTP sites, including sunsite.unc.edu in the file /pub/Linux/docs/LSM.gz.

Note that at the time this article was written, the sound driver was at version 2.4, and was included as part of the Linux kernel version 0.99 patch level 15 (and is probably ancient history by the time you read this).

Jeff Tranter (Jeff_Tranter@mitel.com) is a software designer for a high-tech telecommunications company in Kanata, Canada. He is the author of the Linux Sound HOWTO.

ICMAKE Part 4

Frank B. Brokken

K. Kubat

Issue #4, August 1994

In part 1, Brokken and Kubat explained where the ideas for icmake came from, the basics of the program and where you can get a copy. In Parts 2 and 3 we covered the grammar of icmake source files. In this final part of the article we show examples of the use of icmake.

5. Some examples.

Three examples will be given in this final section, completing our discussion of icmake. The first example illustrates a `traditional make script', used with icmake. The example was taken from the `callback utility', developed by Karel (and also available from beatrix.icce.rug.bl). The second example is a simple dos2unix script which may be used to convert DOS textfiles to Unix textfiles: it uses awk to do the hard work. Finally, the attic-move script is presented, implementing a non-destructive remove, by moving files into an `attic.zip'. More examples can be found in the icmake distribution tar.gz file. The examples are annotated by their own comment, and are presented as they are currently used.

5.1. The callback-(ic)make script.

```
#!/usr/local/bin/icmake -qi
#define CC                  "gcc"
#define CFLAGS              "-c -Wall"
#define STRIP               "strip"
#define AR                  "ar"
#define ARREPLACE           "rvs"
#define DEBUG               ""
#define CALLBACKDIR         "/conf/callback"
#define BINDIR              "/usr/local/bin"
#define VER                 "1.05v
int compdir (string dir)
{
    int
        i,
        ret;
    list
        ofiles,
        cfiles;
    string
        hfile,
        curdir,
        cfile,
        ofile,
        libfile;
    curdir = chdir (".");
    libfile = "lib" + dir + ".a";
    hfile = dir + ".h";
    chdir (dir);
    if (hfile younger libfile)
        cfiles = makelist ("*.c");
    else
        cfiles = makelist ("*.c", younger, libfile);
    for (i = 0; i < sizeof (cfiles); i++)
    {
        cfile = element (i, cfiles);
        ofile = change_ext (cfile, ".o");
        if (! exists (ofile) || ofile older cfile)
            exec (CC, DEBUG, CFLAGS, cfile);
    }
    if (ofiles = makelist ("*.o"))
    {
        exec (AR, ARREPLACE, libfile, "*.o");
        exec ("rm", "*.o");
        ret = 1;
    }
    chdir (curdir);
    return (ret);
}
void linkprog (string dir)
{
    chdir (dir);
    exec (CC, DEBUG, "-o", dir, "-l" + dir, "-lrss", "-L. -L../rss");
    chdir ("..");
}
void buildprogs ()
{
    int
        cblogin,
        cbstat,
        rss;
    chdir ("src");
    cblogin = compdir ("cblogin");
    cbstat  = compdir ("cbstat");
    rss     = compdir ("rss");
    if (cblogin || rss)
        linkprog ("cblogin");
    if (cbstat || rss)
        linkprog ("cbstat");
    chdir ("..");
}
void instprog (string prog, string destdir)
{
    chdir ("src/" + prog);
    exec (STRIP, prog);
    exec ("chmod", "700", prog);
    exec ("cp", prog, destdir);
    chdir ("../..");
}
void install ()
{
    buildprogs ();
    instprog ("cblogin", CALLBACKDIR);
    instprog ("cbstat",  BINDIR);
}
void cleandir (string dir)
{
    chdir ("src/" + dir);
    exec ("rm", "-f", "*.o lib*.a", dir);
    chdir ("../..");
}
void clean ()
{
    exec ("rm", "-f", "build.bim");
    cleandir ("cblogin");
   cleandir ("cbstat");
    cleandir ("rss");
```

```
}
void makedist ()
{
    list
        examples;
    int
        i;
    clean ();
    chdir ("examples");
    examples = makelist ("*");
    for (i = 0; i < sizeof (examples); i++)
        if (exists ("/conf/callback/" + element (i, examples)) &&
            "/conf/callback/" + element (i, examples) younger
                element (i, examples))
            exec ("cp", "/conf/callback/" + element (i, examples),
                element (i, examples));
    chdir ("..");
    exec ("rm", "-f", "callback-" + VER + ".tar*");
    exec ("tar", "cvf", "callback-" + VER + ".tar", "*");
    exec ("gzip", "callback-" + VER + ".tar");
    exec ("mv", "callback-" + VER + ".tar.z",
            "callback-" + VER + ".tar.gz");
}
void main (int argc, list argv)
{
    if (element (1, argv) == "progs")
        buildprogs ();
    else if (element (1, argv) == "install")
        install ();
    else if (element (1, argv) == "clean")
        clean ();
    else if (element (1, argv) == "dist")
        makedist ();
    else
    {
        printf ("\n"
        "Usage: build progs   - builds programs\n"
        "       build install - installs program\n"
        "       build clean   - cleanup .o files etc.\n"
        "\n"
        "       build dist    - makes .tar.gz distrib file\n"
        "\n");
        exit (1);
    }
    exit (0);
}
```

5.2. The Dos to Unix script.

```
#!/usr/local/bin/icmake -qi
/*
                            DOS2UNIX
This script is used to change dos textfiles into unix textfiles.
*/
string
    pidfile;
void usage(string prog)
{
    prog = change_ext(get_base(prog), ""); // keep the scriptname
    printf("\n"
        "ICCE ", prog,
        ": Dos to Unix textfile conversion.  Version 1.00\n"
        "Copyright (c) ICCE 1993, 1994. All rights reserved\n"
        "\n",
        prog, " by Frank B. Brokken\n"
        "\n"
        "Usage: ", prog, " file(s)\n"           // give help
        "Where:\n"
        "file(s): MS-DOS textfiles to convert to UNIX textfiles\n"
        "\n");
    exit (1);                                // and exit
}
void dos2unix(string file)
{
    if (!exists(file))
        printf("'", file, "' does not exist: skipped\n");
    else
    {
        printf("converting: ", file, "\n");
        exec("/bin/mv", file, pidfile);
        system("/usr/bin/awk 'BEGIN {FS=\"\\r\"}; {print $1}' " +
               pidfile + " > " + file);
    }
}
void process(list argv)
{
    int
        i;
                               // make general scratchname
    pidfile = "/tmp/dos2unix." + (string)getpid();
    echo(OFF);                 // no echoing of exec-ed progs
    for (i = 1; i < sizeof(argv); i++)
        dos2unix(element(i, argv));      // convert dos 2 unix
    if (exists(pidfile))
        exec("/bin/rm", pidfile);       // remove final junk
}
int main(int argc, list argv)
{
    if (argc == 1)
        usage(element(0, argv));
    process(argv);                  // process all arguments
    return (0);                    // return when ready
}
```

5.3. The Attic Move script.

```
#! /usr/local/bin/icmake -qi
/*
       This script is used to implement a non-destructive rm
*/
#define YEAR    "1993, 1994"
#define VERSION "1.10"
int
    flags_done,
    extract,
    viewmode,
    debug;
string
    home,
    attic,
    cwd,
    progname,
    recurs,
    forced,
    unzipflag;
void kill(string s)
{
    printf(s, "\n\n");
    exit(1);
}
void preamble(list argv, list envp)
{
    int
        index;
    cwd = chdir(".");                       // get cwd
    for (index = 0; home = element(index, envp); index += 2)
    {
        if (home == "HOME")                 // HOME found
        {                                   // get it
            home = element(index + 1, envp);
            break;                          // and done
        }
    }
if (!home)
        kill("$HOME not found");
    progname = change_ext(element(0, argv), "");
    attic = home + "/attic";      // set $HOME/attic, change to
}
void check_attic()
{
    if (!exists(attic))              // attic should exist
    {
        printf(attic, " does not exist. Create it [y/n] ? ");
        if (getch() != "y")         // not a "y" ?
            kill("ok.");
        system("mkdir " + attic);   // make the attic subdir
        exec("chmod", 700, attic);  // private use
    }                              // else attic must be dir
    else if (!((int)element(0, stat(attic)) & S_IFDIR))
        kill("'" + attic + "' is not a directory");
    attic += "/attic";              // append the zip-name
    chdir("/");                    // go to the root
}
void set_flags(string arg)
{
    int
        index;
    string
        flag;
                              // process all arguments
    for (index = 1; flag = element(index, arg); index++)
    {
        if (flag == "r")       // process encountered options
            recurs = "-r";
        if (flag == "d")
            debug++;
        else if (flag == "f")
            forced = "-f";
        else if (flag == "x")
            extract++;
        else if (flag == "v")
        {
            extract++;
            viewmode++;
            unzipflag = "-l ";
        }
        else
            kill("Unrecognized flag '-" +
                 flag +
                 "': " +
                 progname +
                 "aborted");
    }
    if (extract && unzipflag == "")
        unzipflag = "-u ";         // use proper unzip flag
}
list options(int argc, list argv)
{
    int
        index;
    list
        ret;
    string
        arg;
for (index = 0; index < argc; index++)
    {
        arg = element(index, argv); // get next argument
        if (element(0, arg) == "-") // first element is a - ?
            set_flags(arg);        // then set flags
        else
            ret += (list)arg;     // or add to list to return
    }
    return (ret);                   // returned list
}
void usage()
{
    printf
    (
        "\n"
        "ICCE AM (Attic Move) non-destructive remove. Version "
        VERSION "\n"
        "Copyright (c) ICCE " YEAR ". All Rights Reserved\n"
        "\n"
        "AM by Frank B. Brokken\n"
        "\n"
        "Usage: ", progname, " [options] file(s)\n"
        "Where:\n"
        "   options:\n"
        "      -d: Debug mode: no execution but display of commands\n"
        "      -f: Forced processing of indicated files\n"
        "      -r: Recursive removal of directory contents\n"
        "      -v: View current contents of the attic\n"
        "      -x: Extract files from the attic to their original place\n"
        "          (i.e., if you are permitted to do so...\n"
        "\n"
        "   file(s): names of files and directories to move to/from
        the attic\n"
        "\n"
        "   ", home, "/attic/attic.zip is used to store the files.\n"
        "\n"
    );
    exit (1);
}
string prefix_path(string file)
{
    string
        el,
        ret;
    int
        index;
    if (element(0, file) != "/")  // if file isn't an absolute path
        file = cwd + file;       // then make an absolute path
    for (index = 1; el = element(index, file); index++)
        ret += el;         // remove first char from abs path
    return (ret);                  // return modified string
}
void retrieve(string file)
{
    string
        cmd;
    cmd = "unzip "                  // update only
        + unzipflag
        + attic;
if (!viewmode)
        cmd += " "
             + prefix_path(file);  // and the file (+ path prefix)
    if (debug)
        printf("( cd /; ", cmd, " )\n");  // debug: show command
    else
        system(cmd);                     // else exec cmd
}
void remove(string file)
{
    string
        cmd;
    cmd = "zip -my "         // remove, remove links as links
        + forced             // maybe forced
        + " "
        + recurs             // maybe recursive
        + " "
        + attic              // target zip
        + " "
        + prefix_path(file);  // and the file (+ path prefix)
    if (debug)
        printf("( cd /; ", cmd, " )\n");  // debug: show command
    else
        system(cmd);          // else exec cmd
}
void one_file(string file)
{
    if (extract)                 // either retrieve or remove
        retrieve(file);           // the file
    else
        remove(file);
}
void process(int argc, list argv)
{
    int
        index;
    for (index = 1; index < argc; index++)
        one_file(element(index, argv));    // process one file
}
int main(int argc, list argv, list envp)
{
    echo (OFF);
    preamble(argv, envp);       // set progname and attic dir.
    argv = options(argc, argv); // get the options
    argc = sizeof(argv);       // determine remaining arguments
```

```
       if (argc == 1 && !viewmode) // none left and no viewmode?
           usage();               // give usage and exit 1
       check_attic();             // check accessability of attic
       if (viewmode)              // view contents
           retrieve("");
       else                      // or
           process(argc, argv);    // process remaining arguments
       return (0);                // done
   }
```

Slackware 2.0 Released

Phil Hughes

Issue #4, August 1994

Over the past few months, Slackware has become the most popular Linux distribution to be made available on the Internet. This new release will differ in one major way yet continue to offer the availability and reliability that Slackware has become known for.

As *LJ* is just about to go to the printer, a new version of Slackware [see Patrick Volkerding interview, *LJ* #2] is being released. Over the past few months, Slackware has become the most popular Linux distribution to be made available on the Internet. This new release will differ in one major way yet continue to offer the availability and reliability that Slackware has become known for.

The part that is the same is that Slackware continues to grow into a file-system-compliant, easy-to-install Linux package that offers a very large assortment of programs. The main new features of 2.0 are:

- Better package installation/removal tools, including tools to create your own packages.
- A “contrib” directory with over 40 MB of extra packages. Users are encouraged to contribute packages they've put together.
- Many more precompiled kernels to support any of the hardware supported by the standard kernel releases.
- Integration of the UMSDOS filesystem allowing you to run Linux on top of an MS-DOS filesystem.

The major change is that Pat Volkerding made a deal with Morse Communications. Morse, by partially funding the development of Slackware, will get to distribute the “official version” of Slackware. I asked Pat why he cut the deal. He said, “Mostly because they asked me first. I've had other inquiries since then but I'm happy with the decision to let Morse publish it. I think they'll do a nice job with it.”

Pat went on to say “I hadn't planned to join up with a CD manufacturer in an official sense, but when I thought about it I decided it would be better for Slackware. Getting some project funding has allowed me to put more time into it and it will remain free and available for FTP, of course.”

One other important addition is that Morse will offer 90 days of free support with the purchase of their Slackware Pro 2.0 Linux system. It is expected to be available in mid-July.

Linux Programming Hints

Michael K. Johnson

Issue #4, August 1994

In this month's column, I said that I would give a simple screen-locking example that uses the VT, or Virtual Terminal, ioctl()'s that I documented in that column.

In case you can't remember or didn't read last month's column, the VT ioctl()'s allow you to specify from a user program what the kernel should do about the virtual terminals, or virtual consoles. (These are essentially the same. For the rest of the column, I will refer to them as virtual consoles, not virtual terminals, for no particular reason).

A program can request that the kernel give it raw scan codes instead of full keystrokes, can tell the kernel that you are going into graphics mode on that terminal, and do many other low-level things. XFree86 uses these ioctl()'s heavily, as does svgalib. The Linux DOS emulator (which is really a BIOS emulator) uses them, and the loadable keymaps program (kbd) uses them.

If you didn't read last month's column, the main content of the column will be included in future man pages to be released by the Linux man page project.

I have written a program called vlock, which is a screen locker which can lock virtual consoles. I don't have space here to reproduce the entire source code, but I will give enough details for you to easily construct your own similar program. Instructions for obtaining a copy via anonymous ftp on the Internet follow the code in this column.

Why?

My original purpose in writing vlock was to demonstrate a use for the VT ioctl()'s that they had really not been designed for, to show their flexibility.

If you are like many Linux users, you may have one or two sessions of X running, and a few console logins active at the save time, and be switching back and forth between them. Perhaps you have been editing a program you have been working on, and don't want your roommates or children to start playing with your files while you go away from your computer for one reason or another, but you really don't feel like logging out and restarting all your sessions.

xlock could solve your problem if you only have an X session that you want to lock, but anyone can still switch to the console even when xlock is running. You need a program that can lock all the sessions at once. Well, maybe you need a program that can lock all the sessions at once...

How?

My first idea for locking the console was to read raw scancodes from the keyboard instead of reading normal characters, and to ignore anything but the scancodes for alphanumeric keys, the shift key, the caps lock key, and the control key, and write a state machine to get keys from that. This would automatically ignore the ALT-Fn keys that are normally used to switch from virtual console to virtual console, so those keypresses would not make the VC switch. Of course, it would be possible that there would be some problems with some national keyboards, but it would mostly work for mostly anyone.

However, that would involve a lot of work, and a lot of testing, and I'm too lazy to do that much work if there is an easy way to do it. (I later realized that there was a serious security problem with this approach as well. I'll let you try to figure out what the flaw is, and I'll explain at the end of this column.)

I then noticed that there are ioctl()'s specifically for telling the kernel to ask first before switching virtual consoles. It is possible for a program to explicitly refuse to let the kernel switch virtual consoles. These ioctl()'s only work on virtual consoles, so first we need to open one of the virtual consoles to perform the ioctl()'s on. The easiest thing to do is this:

```
  if (vfd = open("/dev/console", O_RDWR) < 0) {
    perror("vlock: could not open /dev/console");
    exit (1);
  }
```

/dev/console stands for the current screen. The assumption is that when vlock is run, it will be run on the current virtual console. (It turns out that this assumption does not create a security hole, although it might look to you like it ought to.)

It would also be possible to switch to an unallocated virtual terminal, like X does, which might be preferable in some circumstances. To do this, we could have used the ioctl VT_OPENQRY to find the number of the first available virtual console, opened the appropriate device (/dev/ttynn, where nn is the number returned by VT_OPENQRY), and used VT_ACTIVATE to switch to that virtual console.

It is a lot easier to just open /dev/console.

```
c = ioctl(vfd, VT_GETMODE, &vtm);
if (c < 0) {
  fprintf(stderr,
        "This tty is not a virtual console.\n");
  is_vt = 0;
} else {
  is_vt = 1;
}
```

We will treat the VT_GETMODE and VT_SETMODE ioctl()'s like the termios interface: first we get the current settings, then we change the local copy, then we set the kernel's copy to look like the changed local copy.

VT_GETMODE fills a vt_mode structure with the current VT settings. If it returns an error, the program must not be running on a virtual console. vlock does not exit on this error, but it does set the is_vt variable to 0, and it does not try to use any more VT ioctl()'s if the is_vt variable is set to 0.

```
/* we set SIGUSR{1,2} to point to *_vt() */
sigemptyset(&(sa.sa_mask));
sa.sa_flags = 0;
sa.sa_handler = release_vt;
sigaction(SIGUSR1, &sa, NULL);
sa.sa_handler = acquire_vt;
sigaction(SIGUSR2, &sa, NULL);
```

We will arrange in a moment for SIGUSR1 to be sent to the process whenever the kernel is requested to change away from the virtual console the program is running on, and for SIGUSR2 to be sent to the process whenever the kernel is requested to change to the virtual console the program is running on. These requests can be caused by the user pressing ALT-Fn keys or by other programs issuing a VT_ACTIVATE ioctl.

When SIGUSR1 is received, release_vt() is called:

```
void release_vt(int signo) {
  if (!o_lock_all)
    /* kernel is allowed to switch */
    ioctl(vfd, VT_RELDISP, 1);
  else
    /* kernel is not allowed to switch */
    ioctl(vfd, VT_RELDISP, 0);
}
```

The variable o_lock_all is set if the user wants to lock all virtual consoles at once. It is not set if the user only wants to lock the current virtual console. VT_RELDISP is used to tell the kernel that the program acknowledges that it has received the signal asking it to relinquish the virtual console, and tells the kernel whether or not it agrees to do so. The third argument is set to 1 to allow the kernel to switch to another virtual console, or set to 0 to prevent the kernel from switching to another virtual console.

When SIGUSR2 is received, acquire_vt() is called:

```
void acquire_vt(int signo) {
  /* This call is not currently required under Linux,
     but it won't hurt, either... */
  ioctl(vfd, VT_RELDISP, VT_ACKACQ);
}
```

Linux does not actually require that this be done; it is included for compatibility with SYSV, which does require that it is called. I included it in vlock mainly so that if someone wanted to port vlock to some version of SYSV, there would be one less stumbling block for him or her.

Now that we have set up these signal handlers, we will tell the virtual console manager about them.

We did not want to tell the virtual console manager to route requests to change virtual consoles through these signals until the signals' handlers had been installed, because to do otherwise could cause a small possibility of a bug on very slow machines which are running too many processes at once.

```
if (is_vt) {
/* Keep a copy around to restore
     at appropriate times */
  ovtm = vtm;
  vtm.mode = VT_PROCESS;
  /* handled by release_vt(): */
  vtm.relsig = SIGUSR1;
  /* handled by acquire_vt(): */
vtm.acqsig = SIGUSR2;
  ioctl(vfd, VT_SETMODE, &vtm);
}
```

ovtm is another vt_mode structure, like vtm. Setting vtm.mode to VT_PROCESS causes the kernel to ask permission to change virtual consoles. Setting **vtm.relsig** to **SIGUSR1** and **vtm.acqsig** to **SIGUSR2** tells the kernel how to ask permission.

At this point, all that needs to be done is to handle all reasonable signals, so that people can't break in by typing **control-c** or **control-** or control-break, and to then ask for the user to type in a password and check it against the real password. There is a library function, **getpass()**, which gets a password from the user without echoing it to the screen.

Unfortunately, this function is broken under at least one shadow password implementation, because signal handlers are not installed correctly, so to make a screen locking program that works with shadow passwords, you either have to fix the shadow password library or write your own version of **getpass()**. With vlock, I chose to tell people that vlock doesn't work right with shadow passwords without fixing their shadow password library, rather than writing my own version of the function.

Once a correct password has been entered, the program can just exit. This is acceptable under Linux, at least. However, in case this doesn't work with some other SYSV implementations of the VT **ioctl()**'s, I have included code in vlock to restore everything, including the VT state, to the original settings. That's why I made the copy of vtm called ovtm a few code fragments ago.

Don't re-invent the wheel

Unless you want to, of course. By the time you read this, I will probably have upgraded vlock several times. The newest version of vlock will always be available from the ftp site tsx-11.mit.edu in the directory /pub/linux/sources/usr.bin in a file called vlock-m.n.tar.gz, where m and n are the major and minor version numbers of the release, respectively.

As of this writing, the current version of vlock is 0.6. If you cannot use ftp, but do have Internet e-mail, you may send e-mail to johnsonm@redhat.com and request a copy, and I can send you a uuencoded gzipped tar file containing both sources and a binary for vlock. Also, the Debian distribution of Linux includes vlock.

The Flaw…

Near the beginning of this article I said I would explain the fundamental flaw with trying to lock the virtual consoles by simply capturing all the keys. The problem is that someone could easily log in from the network or a modem or serial terminal and run a program (they would probably have to write it first) which would issue a request to change the virtual console. This program would be a little trickier than it sounds at first, but it is possible to write it. The kernel would honor the **ioctl()** requesting the change, and the screen-locking program would be defeated.

Loose ends

I am finding out that many Unix programmers are a little confused about signals. This is understandable, because there are at least three standards for using signals. [Purists, please don't tell me that there are actually more; I'm trying to keep things relatively simple here. A much more detailed explanation, which is historically correct to the best of my knowledge, can be found in Advanced Programming in the Unix Environment, by W. Richard Stevens, in chapter 10, Signals.] Though I have mentioned the differences in signals before, in issue one, I will explain more explicitly here.

The original signals were unreliable. The **signal()** function was used to install a signal handler that was good for one invocation of the signal, and once the signal handler had been invoked once, the signal handler would uninstall. So you would install your signal handler like this:

signal(SIGUSR1, signal_handling_function);

and then you would implement your signal handling function like this:

```
void signal_handling_function(int signo) {
  signal(SIGUSR1, signal_handling_function);
  /* Do whatever the signal handling
     function is supposed to do... */
}
```

The problem with taking this approach is that occasionally a second signal would arrive in between the time that the kernel uninstalled the signal handler and the time that the signal handler re-installed itself.

The problem with not taking this approach is that signal handlers need to be reentrant.

Unfortunately, as reliable signals were introduced, BSD revised **signal()** to not get uninstalled when it was called, while SYSV left **signal()** the way it was. There is more to the story, but it only gets more confusing.

Fortunately, there is absolutely no need to be confused. There is no need to use **signal()** at all. Don't use it: to do so (without knowing all the details about the **signal()** function on all different versions of Unix) is to write non-portable code.

POSIX defines an alternate interface that is the same on all POSIX-compliant platforms. This interface is called sigaction, and is more powerful and flexible than either version of **signal()**. [sigaction is derived from the first BSD implementation of reliable signals, so code which uses sigaction will not only be portable to all POSIX platforms, but to pre-POSIX BSD systems as well.] Unfortunately, it is a little more complex, but you can write your own signal management wrapper functions to get exactly the kind of signals you need. Here is an example:

```
typedef void signal_handler(int);
signal_handler *
my_signal(int signo, signal_handler *func, int oneshot) {
 struct sigaction sact, osact;
  sigemptyset(&sact.sa_mask);
  sact.sa_handler = func;
  if (oneshot) {
    sact.sa_flags = SA_ONESHOT;
  } else {
    sact.sa_flags = 0;
  }
  if (sigaction(signo, &sact, &osact) < 0) {
    return (SIG_ERR);
  } else {
    return (oact.sa_handler);
  }
}
```

This is not perfect, but it creates an interface to sigaction that is as convenient as **signal()** but will have the same semantics no matter what system it is compiled on, unlike **signal()**.

It works like **signal()**, except that it takes a third argument. That third argument determines whether the signal handler remains installed when it is called or if it is uninstalled as soon as it is called.

There are two normal reasons to have a signal handler automatically uninstalled. The first is if the signal handler is not reentrant—if it is not safe to run the signal handler again until while it is already being run. The second is for those times when you really only want to catch one instance of a signal, for example **SIGALRM**.

You may have noticed the call to **sigemptyset()** in the code above. It is important for it to be there, but I have not yet mentioned it. It turns out that it is possible for a sigaction signal handler to mask out certain signals while it is being run. Perhaps the most common occurrence of this is in signal handlers that are not reentrant. These signal handlers can set their **sa_mask** to keep from being called again while they are being invoked, by using code like this:

```
  sigemptyset(&sact.sa_mask);
  sigaddset(&sact.sa_mask, SIGFOO);
  sact.sa_handler = signal_handler;
  sact.sa_flags = 0;
  if (sigaction(SIGFOO, &sact, &osact) < 0) {
    do_signal_error(SIGFOO);
  }
```

This will allow you to use a non-reentrant signal handler for **SIGFOO**. Of course, this code will have to be altered slightly to fit into your application. You will at least have to use a real signal name instead of **SIGFOO**...

If you are interested in doing more with signals, look up the **sigaction()** function in a modern Unix programming book or manual, and also read up on “signal sets”, which may be found under the following functions; **sigemptyset()**, **sigfillset()**, **sigaddset()**, **sigdelset()**, **sigismember()**, **sigprocmask()**, **sigpending()**, **sigsetjmp()**, **siglongjmp()** and **sigsuspend()**. These provide very fine-tuneable support for all sorts of fancy signal work, which I will not try to cover this month.

Please send e-mail to johnsonm@redhat.com or send paper mail to Programming Tips, *Linux Journal*, P.O. Box 85867, Seattle, WA 98145-1867, if you have any suggestions or comments about this column. I'd like to know what you have found useful so far.

If there are any undocumented Linux features that you would like to see covered, I'll look at them. I may write a column, if there is enough interest. I'd also like to have guest columnists write for Linux Programming Hints.

Archive Index Issue Table of Contents

Advanced search

Cooking with Linux

Matt Welsh

Issue #4, August 1994

This month, Cooking with Linux jogs your memory with a look at the history of Linux. Or, at least, that's how we remember it...

Linux users these days have it easy. Back when I was starting out with the system (around the 0.12 days), we didn't have the luxuries of networking, documentation, DOSEMU, or XFree86 for that matter. We didn't have “Plug and Play” Linux distributions on media ranging from floppy to tape to CD-ROM. There were two so-called “distributions” to choose from: H.J. Lu's classic “boot/root” diskettes, or the then-popular MCC-Interim release-which contained nearly all of the Linux software available at the time on a whopping seven diskettes.

Today, Linux development has improved to such a point that nothing seems to be challenging anymore. Running gcc without a segmentation fault used to give me thrills; now it's a part of everyday life. (On occasion,

I will fire up my ancient 0.95+ kernel and try to compile Emacs with gcc-1.38, just to bring back the excitement of cursing at the machine for hours as error messages fill the screen.) Linux is just too easy. Anybody can download Slackware and run their own WWW server over a SLIP line. And without having to compile a single line of code! It's sad, I'm telling you.

To fully appreciate the power of Linux, I believe that all newcomers should be forced to relive those days of yore, when the definition of a “real hacker” was someone who would stay up all night in a vain attempt to decipher the semantics of the serial driver. (Possibly in order to get the serial cheese grater to work, but that is another story.) Without going through this kind of painful, teeth-grinding ordeal it is too easy for users to take the many, many hours of hard work by the Linux developers for granted.

So, I think it is appropriate at this time to inflict a reality check on the current generation of Linux users, in the form of a short “Do You Remember?” quiz. Without overdoing the nostalgia, let us say that if you can remember at least, er, half of the items in this list then you are qualified to call yourself a Linux grunt—who has earned the right to sit back and drink coffee while your Linux box formats your thesis with TeX, recompiles libc, and provides anonymous FTP access to the majority of Eastern Europe all at the same time.

Be warned: Some of the items in the following list may bring back painful memories. If you are overly sentimental or have small children, you may want to consider turning the page—now.

Do you remember…

1. ...comp.os.linux? All right, that's an easy one, but since the proliferation of the—what is it now?—five Linux newsgroups, some of us may have forgotten how pleasant it was to spend a Friday evening—scratch that—an entire week, catching up on articles in the single Linux-related Usenet forum. Unless you had a killfile larger than the collected works of Terry Pratchett you simply had no hope of following it all. And woe to the faithful reader that missed “important” postings, such as announcements of new kernel releases, or the weekly “Linux is Obsolete” flamewar.

Eventually, the Linux community wised up and decided to create comp.os.linux.announce, and six months later the current slew of Linux groups. This was at a time when nearly 50% of postings to c.o.l were calls for a newsgroup split, so it was inevitable.

Bonus question: If you remember comp.os.linux, what about alt.os.linux, its predecessor? Or, what about the first Linux threads to emerge on comp.os.minix? (If you can remember this, then you really are a Linux hacker. Either that, or you spend too much time reading Usenet.)

2. ...When the major Linux FTP site was banjo.concert.net? That's right—long before SunSITE was merely a desk calculator, there was banjo. Alan Clegg, who ran banjo at the time, was eventually forced to remove the Linux archives from that machine, as they required too much diskspace—an entire 45 megs. To top it all off, there were dozens of FTP logins a day. It was too much for the poor machine, so banjo had to call it quits.

SunSITE's entry into the Linux world was due in part to Jon Magid (do you remember him?) being hired as the systems administrator for a new FTP/Gopher/WAIS/WWW site funded by Sun Microsystems. Sun gave the SunSITE administrators permission to provide any kind of interesting information that they wished from the new machine, so Jon copied banjo's entire Linux archives over.

Since then, Linux FTP usage on SunSITE has grown to amazing proportions. The Linux archives there now require more than 640 megabytes of disk storage. On a typical day, there might be 2,500 Linux FTP logins, and around 43,000 Linux files downloaded in all. If you were to count all of the Linux-related FTP traffic to SunSITE's many mirrors, as well as the other major Linux archive sites, I'm sure you'd have enough statistics to give the NSF a real headache.

3. ...The Minix filesystem? This filesystem was (and perhaps still is) a favorite of Linux hackers for quite some time. Never mind the fact that it was originally the only filesystem type supported by the kernel, and that it limited partitions to 64 megabytes in size. Linus Torvalds began development of the Linux kernel under Minix, an academic Unix clone featured in an operating systems book by Andy Tanenbaum. So it made perfect sense to implement the Minix filesystem for which Linus had the source code. There is an interesting anecdote (or shall we say “legend?”) in which Linus accidentally trashed one of his Minix filesystems—the one which happened to contain the entire Linux kernel source tree on his system. Linus, being the Minix filesystem wizard that he is, managed to repair the damaged superblock by hand and save untold hours of work. The day was saved.

4. ...Ross Biro and the Linux TCP/IP code? Back in the early days of the Linux kernel (a mind-staggering two years ago), there was no networking support in the Linux kernel whatsoever. Many users were forced to boot MS-DOS (or some other operating system) to talk to the network—others resorted to more dated methods such as (gasp) UUCP.

The first generation of the Linux TCP/IP code (now known as “NET-1”, but that is a posthumous title) was developed by many people, Ross Biro being one of the foremost. Although it supported a limited range of hardware (Ethernet only, of course—SLIP was out of the question), and was far from perfect, it was really quite impressive at the time. Well, perhaps I should put that another way. After pulling an all-nighter to hack the alpha TCP/IP code into my, er, “personalized” kernel, you'd better believe that I was impressed. After wrestling with compilation errors and kernel panics all night, bloodshot and weary, I remember running telnet just as the first rays of dawn made their way through the window—and it worked! Of course, the kernel crashed five minutes later, but it was good enough for me. “Login or bust!” was my motto for the evening.

Ross was one of the key figures in the development of the Linux TCP/IP suite, along with Don Becker and others. However, as more and more Linux users attempted to employ their code, more and more problems would emerge—

inevitably causing Ross' mailbox to be flooded with unwarranted complaints and flames. Not long thereafter Ross “threw in the towel”, tired of the inappropriate treatment by the Linux user community. The Linux development effort lost an important player on that day. (The moral of the story? I forget.)

5. ...Shoelace? These were short lengths of string or cord, used to tie shoes, that were eventually driven into obsolescence by Velcro. Wait a minute! Shoelace was actu-ally the predecessor to LILO, which as we all know is the LInux LOader, responsible for booting Linux (and whatever other operating systems you may have installed). Shoelace was originally the only way to boot Linux from the hard drive—and most Linux users at the time used a kernel floppy. What did using Shoelace entail? The exact details are lost in the mists of time, but I do remember having to modify sectors 508 and 509 of the kernel image—by hand —in order to set the root filesystem device. (In fact, this was necessary even if you didn't use Shoelace—but manual editing of kernel images is something that Linux veterans always love to brag about.)

Another caveat associated with Shoelace was that it could only be recompiled under Minix. In fact, the Linux kernel once shared the same fate: One had to have Minix installed in order to compile the Linux kernel—and, at one time, to even install the Linux software. (This period was to Linux what the Dark Ages were to European history. Minus famine and cultural stagnation, of course.)

6. ...The original Linux FAQ? (Perhaps this question should read, “Do you want to remember the original Linux FAQ?”) The original list of Linux Frequently Asked Questions was coordinated by Marc-Michel Corsini (although the first version was posted by Robert Blum). It was a dinosaur of a document. The list of authors and contributors numbered in the dozens, and the number of mistakes and incongruencies topped that. It was a valiant effort, mind you—in fact, I maintained, or tried to maintain, several sections of the FAQ before giving up. Eventually, as the document was nearing critical mass (800K or so) and was posted in no less than 7 parts each month, everyone agreed that it was too long and out-of-date to maintain any further. Ian Jackson rewrote the FAQ from scratch, and we started the HOWTO project to pick up the pieces. Since then things have been relatively peachy.

The last version of the original FAQ that I have archived is from July 1993, which places it just before Ian's rewrite and the comp.os.linux newsgroup split of that summer. Linux users who were around at that time may remember the following classic excerpts, both of which are attributed to Marc-Michel Corsini:

- “The last-change-date of this posting is always `two minutes ago'. :)”
- “The FAQ contains a lot of information sometimes I've put it down in 3 different ways because people seem not to understand what they read (or

> what I wrote, you know I'm just a froggy and English is not my natural language). What I mean is that not all is in the FAQ but many things are there, so please just take time to read it this will spare a lot of the other Linuxers [and if you think I should rephrase some Q/A just drop me a note with the corrections].”

Now, those were the days.

This list could go on and on, but I believe to have made my point by now. As a matter of fact, most Linux users won't remember those important milestones of Linux development such as, oh, the introduction of the VFS layer, the original implementation of shared libraries, and the first version of the Extended Filesystem. But let sleeping dogs lie.

If this brief excursion into the dark annals of Linux history has taught you one thing, it is that you should be grateful for those foolhearty pioneers that worked for peanuts to blaze the trail for the masses to follow. They had to edit their kernel images by hand, and walk five miles in the snow—barefoot—just to upload the newest set of patches, you know.

And the next time you consider complaining that running Lucid Emacs 19.05 via NFS from a remote Linux machine in Paraguay doesn't seem to get the background colors right, you'll know who to thank.

Matt Welsh (mdw@sunsite.unc.edu) is an artificial intelligence which has been programmed to make somewhat offbase observations of the Linux community from time to time. Comments and questions are welcome; the author can be reached via Internet e-mail, or via paper mail c/o *Linux Journal*.

Archive Index Issue Table of Contents

Advanced search

What's GNU: Bash—The GNU Shell

Chet Ramey

Issue #4, August 1994

Conclusion of an article started last month. While originally written by Brian Fox of the Free Software Foundation, bash is now maintained by Chet Ramey. In this article, Chet explains the history of shells and then goes on to explore features specific to bash.

History

Access to the list of commands previously entered (the command history) is provided jointly by bash and the readline library. bash provides variables (**$HISTFILE**, **$HISTSIZE**, and **$HISTCONTROL**) and the history and fc builtins to manipulate the history list. The value of **$HISTFILE** specifies the file where bash writes the command history on exit and reads it on startup. **$HISTSIZE** is used to limit the number of commands saved in the history. **$HISTCONTROL** provides a crude form of control over which commands are saved on the history list: a value of ignorespace means to not save commands which begin with a space; a value of ignoredups means to not save commands identical to the last command saved. **$HISTCONTROL** was named **$history_control** in earlier versions of bash; the old name is still accepted for backward compatibility. The history command can read or write files containing the history list and display the current list contents. The fc builtin, adopted from POSIX.2 and the Korn Shell, allows display and re-execution, with optional editing, of commands from the history list. The readline library offers a set of commands to search the history list for a portion of the current input line or a string typed by the user. Finally, the history library, generally incorporated directly into the readline library, implements a facility for history recall, expansion, and re-execution of previous commands very similar to csh (“bang history”, so called because the exclamation point introduces a history substitution):

```
$ echo a b c d e
a b c d e
$ !! f g h i
echo a b c d e f g h i
a b c d e f g h i
$ !-2
```

```
echo a b c d e
a b c d e
$ echo !-2:1-4
echo a b c d
a b c d
```

The command history is only saved when the shell is interactive, so it is not available for use by shell scripts.

New Shell Variables

There are a number of convenience variables that bash interprets to make life easier. These include **FIGNORE**, which is a set of filename suffixes identifying files to exclude when completing filenames; **HOSTTYPE**, which is automatically set to a string describing the type of hardware on which bash is currently executing; **command_oriented_history**, which directs bash to save all lines of a multiple-line command such as a while or for loop in a single history entry, allowing easy re-editing; and **IGNOREEOF**, whose value indicates the number of consecutive EOF characters that an interactive shell will read before exiting—an easy way to keep yourself from being logged out accidentally. The auto_resume variable alters the way the shell treats simple command names: if job control is active, and this variable is set, single-word simple commands without redirections cause the shell to first look for and restart a suspended job with that name before starting a new process.

Brace Expansion

Since sh offers no convenient way to generate arbitrary strings that share a common prefix or suffix (pathname expansion requires that the filenames exist), bash implements brace expansion, a capability picked up from csh. Brace expansion is similar to pathname expansion, but the strings generated need not correspond to existing files. A brace expression consists of an optional preamble, followed by a pair of braces enclosing a series of comma-separated strings, and an optional postamble. The preamble is prepended to each string within the braces, and the postamble is then appended to each resulting string:

```
$ echo a{d,c,b}e
ade ace abe
```

Process Substitution

On systems that can support it, bash provides a facility known as process substitution. Process substitution is similar to command substitution in that its specification includes a command to execute, but the shell does not collect the command's output and insert it into the command line. Rather, bash opens a pipe to the command, which is run in the background. The shell uses named pipes (FIFOs) or the /dev/fd method of naming open files to expand the process substitution to a filename which connects to the pipe when opened. This filename becomes the result of the expansion. Process substitution can be used to compare the outputs of two different versions of an application as part of a regression test:

```
  $ cmp <\>(old_prog) <(new_prog)
```

Prompt Customization

One of the more popular interactive features that bash provides is the ability to customize the prompt. Both **$PS1** and **$PS2**, the primary and secondary prompts, are expanded before being displayed. Parameter and variable expansion is performed when the prompt string is expanded, so any shell variable can be put into the prompt (e.g., $SHLVL, which indicates how deeply the current shell is nested). bash specially interprets characters in the prompt string preceded by a backslash. Some of these backslash escapes are replaced with the current time, the date, the current working directory, the username, and the command number or history number of the command being entered. There is even a backslash escape to cause the shell to change its prompt when running as root by using the su command. Before printing each primary prompt, bash expands the variable **$PROMPT_COMMAND** and, if it has a value, executes the expanded value as a command, allowing additional prompt customization. For example, this assignment causes the current user, the current host, the time, the last component of the current working directory, the level of shell nesting, and the history number of the current command to be embedded into the primary prompt:

```
  $ PS1='\u@\h [      ] \W($SHLVL:\!)\$ `
  chet@odin [21:03:44] documentation(2:636)$ cd ..
  chet@odin [21:03:54] src(2:637)$
```

The string being assigned is surrounded by single quotes so that if it is exported, the value of **$SHLVL** will be updated by a child shell:

```
  chet@odin [21:17:35] src(2:638)$ export PS1
  chet@odin [21:17:40] src(2:639)$ bash
  chet@odin [21:17:46] src(3:696)$
```

The \$ escape is displayed as “$” when running as a normal user, but as “#” when running as root.

File System Views

Since Berkeley introduced symbolic links in 4.2 BSD, one of their most annoying properties has been the “warping” to a completely different area of the file system when using cd, and the resultant non-intuitive behavior of “cd ..”. The Unix kernel treats symbolic links physically. When the kernel is translating a pathname in which one component is a symbolic link, it replaces all or part of the pathname while processing the link. If the contents of the symbolic link begin with a slash, the kernel replaces the pathname entirely; if not, the link contents replace the current component. In either case, the symbolic link is visible. If the link value is an absolute pathname, the user finds himself in a completely different part of the file system.

bash provides a logical view of the file system. In this default mode, command and filename completion and builtin commands such as cd and pushd which change the current working directory transparently follow symbolic links as if they were directories. The $PWD variable, which holds the shell's idea of the current working directory, depends on the path used to reach the directory rather than its physical location in the local file system hierarchy. For example:

```
$ cd /usr/local/bin
$ echo $PWD
/usr/local/bin
$ pwd
/usr/local/bin
$ /bin/pwd
/net/share/sun4/local/bin
$ cd ..
$ pwd
/usr/local
$ /bin/pwd
/net/share/sun4/local
```

One problem with this, of course, arises when programs that do not understand the shell's logical notion of the file system interpret “..” differently. This generally happens when bash completes filenames containing “..” according to a logical hierarchy which does not correspond to their physical location. For users who find this troublesome, a corresponding physical view of the file system is available:

```
$ cd /usr/local/bin
$ pwd
/usr/local/bin
$ set -o physical
$ pwd
/net/share/sun4/local/bin
```

Internationalization

One of the most significant improvements in version 1.13 of bash was the change to “eight-bit cleanliness”. Previous versions used the eighth bit of characters to mark whether or not they were quoted when performing word expansions. While this did not affect the majority of users, most of whom used only seven-bit ASCII characters, some found it confining. Beginning with version 1.13, bash implemented a different quoting mechanism that did not alter the eighth bit of characters. This allowed bash to manipulate files with “odd” characters in their names, but did nothing to help users enter those names, so version 1.13 introduced changes to readline that made it eight-bit clean as well. Options exist that force readline to attach no special significance to characters with the eighth bit set (the default behavior is to convert these characters to meta-prefixed key sequences) and to output these characters without conversion to meta-prefixed sequences. These changes, along with the expansion of keymaps to a full eight bits, enable readline to work with most of the ISO-8859 family of character sets, used by many European countries.

POSIX Mode

Although bash is intended to be POSIX.2 conformant, there are areas in which the default behavior is not compatible with the standard. For users who wish to operate in a strict POSIX.2 environment, bash implements a POSIX mode. When this mode is active, bash modifies its default operation where it differs from POSIX.2 to match the standard. POSIX mode is entered when bash is started with the "-o posix" option or when set -o posix is executed. For compatibility with other GNU software that attempts to be POSIX.2 compliant, bash also enters POSIX mode if either of the variables $POSIX_PEDANTIC or $POSIXLY_CORRECT is set when bash is started or assigned a value during execution. When bash is started in POSIX mode, for example, it sources the file named by the value of $ENV rather than the "normal" startup files.

Future Plans

There are several features that will be introduced in the next version of bash, version 1.14, and a number under consideration for future releases. This section will briefly detail the new features planned for version 1.14 and describe features that may appear in later versions.

The new features available in bash-1.14 answer several of the most common requests for enhancements. Most notably, there is a mechanism for including non-visible character sequences in prompts, such as those which cause a terminal to print characters in different colors or in standout mode. There was nothing preventing the use of these sequences in earlier versions, but the readline redisplay algorithm assumed each character occupied physical screen space and would wrap lines prematurely.

Readline has a few new variables, several new bindable commands, and some additional emacs mode default key bindings. A new history search mode has been implemented: in this mode, readline searches the history for lines beginning with the characters between the beginning of the current line and the cursor. The existing readline incremental search commands no longer match identical lines more than once. Filename completion now expands variables in directory names. The history expansion facilities are now nearly completely csh-compatible: missing modifiers have been added and history substitution has been extended.

Several of the features described earlier as appearing in future releases, such as set -o posix and $POSIX_PEDANTIC, are present in version 1.14. There is a new shell variable, OSTYPE, to which bash assigns a value that identifies the version of Unix it's running on (great for putting architecture-specific binary directories into the $PATH). Two variables have been renamed: $HISTCONTROL replaces $history_control , and $HOSTFILE replaces $hostname_completion_file. In both cases, the old names are accepted for backward compatibility. The ksh select construct, which allows the generation of simple menus, has been implemented. New capabilities have been added to existing variables: $auto_resume can now take values of exact or substring, and $HISTCONTROL understands the value ignoreboth, which combines the two previously acceptable values. The dirs builtin has acquired options to print out specific members of the directory stack. The $nolinks variable, which forces a physical view of the file system, has been superseded by the -P option to the set builtin (equivalent to set -o physical); the variable is retained for backward compatibility. The version string contained in $BASH_VERSION now includes an indication of the patch level as well as the "build version". Some little-used features have been removed: the bye synonym for exit and the $NO_PROMPT_VARS variable are gone. There is now an organized test suite that can be run as a regression test when building a new version of bash.

The documentation has been thoroughly overhauled: there is a new manual page on the readline library and the info file has been updated to reflect the current version. As always, as many bugs as possible have been fixed, although some surely remain.

There are a few features that I hope to include in later bash releases. Some are based on work already done in other shells.

In addition to simple variables, a future release of bash will include one-dimensional arrays, using the ksh implementation of arrays as a model. Additions to the ksh syntax, such as varname=(...) to assign a list of words directly to an array and a mechanism to allow the read builtin to read a list of values directly into an array, would be desirable. Given those extensions, the ksh

"set -A" syntax may not be worth supporting (the -A option assigns a list of values to an array, but is a rather peculiar special case).

Some shells include a means of programmable word completion, where the user specifies on a per-command basis how the arguments of the command are to be treated when completion is attempted: as filenames, hostnames, executable files, and so on. The other aspects of the current bash implementation could remain as-is; the existing heuristics would still be valid.

Only when completing the arguments to a simple command would the programmable completion be in effect.

It would also be nice to give the user finer-grained control over which commands are saved onto the history list. One proposal is for a variable, tentatively named HISTIGNORE, which would contain a colon-separated list of commands. Lines beginning with these commands, after the restrictions of $HISTCONTROL have been applied, would not be placed onto the history list. The shell pattern-matching capabilities could also be available when specifying the contents of $HISTIGNORE.

One thing that newer shells such as wksh (also known as dtksh) provide is a command to dynamically load code implementing additional builtin commands into a running shell. This new builtin would take an object file or shared library implementing the "body" of the builtin (xxx_builtin() for those familiar with bash internals) and a structure containing the name of the new command, the function to call when the new builtin is invoked (presumably defined in the shared object specified as an argument), and the documentation to be printed by the help command (possibly present in the shared object as well). It would manage the details of extending the internal table of builtins.

A few other builtins would also be desirable: two are the POSIX.2 getconf command, which prints the values of system configuration variables defined by POSIX.2, and a disown builtin, which causes a shell running with job control active to "forget about" one or more background jobs in its internal jobs table. Using getconf, for example, a user could retrieve a value for $PATH guaranteed to find all of the POSIX standard utilities, or find out how long filenames may be in the file system containing a specified directory.

There are no implementation timetables for any of these features, nor are there concrete plans to include them. If anyone has comments on these proposals, feel free to send me electronic mail.

Reflections and Lessons Learned

The lesson that has been repeated most often during bash development is that there are dark corners in the Bourne Shell, and people use all of them. In the original description of the Bourne shell, quoting and the shell grammar are both poorly specified and incomplete; subsequent descriptions have not helped much. The grammar presented in Bourne's paper describing the shell distributed with the Seventh Edition of Unix is so far off that it does not allow the command who|wc. In fact, as Tom Duff states:

"Nobody really knows what the Bourne shell's grammar is. Even examination of the source code is little help."1

The POSIX.2 standard includes a yacc grammar that comes close to capturing the Bourne shell's behavior, but it disallows some constructs which sh accepts without complaint-and there are scripts out there that use them. It took a few versions and several bug reports before bash implemented sh-compatible quoting, and there are still some “legal” sh constructs which bash flags as syntax errors. Complete sh compatibility is a tough nut.

The shell is bigger and slower than I would like, though the current version is faster than previously.

The readline library could stand a substantial rewrite.

A hand-written parser to replace the current yacc-gener-ated one would probably result in a speedup, and would solve one glaring problem: the shell could parse commands in “$(...)” constructs as they are entered, rather than reporting errors when the construct is expanded.

As always, there is some chaff to go with the wheat. Areas of duplicated functionality need to be cleaned up. There are several cases where bash treats a variable specially to enable functionality available another way ($notify vs. set -o notify and $nolinks vs.

set -o physical, for instance); the special treatment of the variable name should probably be removed. A few more things could stand removal; the $allow_null_ glob_expansion and $glob_dot_filenames variables are of particularly questionable value. The $[...] arithmetic evaluation syntax is redundant now that the POSIX-mandated $((...)) construct has been implemented, and could be deleted. It would be nice if the text output by the help builtin were external to the shell rather than compiled into it. The behavior enabled by $command_oriented_history, which causes the shell to attempt to save all lines of a multi-line command in a single history entry, should be made the default and the variable removed.

Availability

As with all other GNU software, bash is available for anonymous FTP from prep.ai.mit.edu:/pub/gnu and from other GNU software mirror sites. The current version is in bash-1.13.5.tar.gz in that directory. Use archie to find the nearest archive site. The latest version is always available for FTP from bash.CWRU.Edu:/pub/ dist. bash documentation is available for FTP from bash.CWRU.Edu:/pub/bash.

The Free Software Foundation sells tapes and CD-ROMs containing bash; send electronic mail to gnu@prep.ai.mit.edu or call +1-617-876-3296 for more information. bash is also distributed with several versions of Unix-compatible systems. It is included as /bin/sh and /bin/bash on several Linux distributions and as contributed software in BSDI's BSD/386 and FreeBSD.

Conclusion

bash is a worthy successor to sh. It is sufficiently portable to run on nearly every version of Unix from 4.3 BSD to SVR4.2, and several Unix workalikes. It is robust enough to replace sh on most of those systems, and provides more functionality. It has several thousand regular users, and their feedback has helped to make it as good as it is today-a testament to the benefits of free software.

1 Tom Duff, “Rc-A Shell for Plan 9 and UNIX systems”, Proc. of the Summer 1990 EUUG Conf., London, July, 1990, pp. 21-33

2 BSD/386 is a trademark of Berkeley Software Design, Inc.

Chet Ramey (chet@po.cwru.edu) is a programmer at Case Western Reserve University and volunteer at the Free Software Foundation.

Letters to the Editor and Linux Q&A

Various

Issue #4, August 1994

Since we are short on Letters to the Editor this month, and because we are inaugurating our promised Q&A column with only a few entries, we are combining them this month. If you like the combined format, please tell us.

The article on page 14 of *Linux Journal* #2 [The “What's GNU” article on using the Unix “tools philosophy”] was interesting to me. The final pipeline on page 17 has given me a better idea of what Unix really is!!

I even got it working, too, in my Slackware 1.1.1 from morse.net-jon kitchin, VK6TU, jon@dialix.oz.au

***LJ* replies**:

We're glad you liked it. You will probably enjoy the article this month on the history of Unix. Like the “What's GNU” article, it does not directly relate to Linux, but it tells a story that explains why we can have Linux in the first place.

Your *Linux Journal* is great. I just read the second issue, which was outstanding, and would like to ftp the GNU C Library Reference Manual. I could not find it at sunsite. What is the file name I should be looking for?-Clinton Carr, clint@netcom.com

***LJ* replies**:

All GNU software (that is, software maintained by the Free Software Foundation) and its documentation, including the GNU C library and the GNU C library reference manual, can be found at prep.ai.mit.edu in the directory /pub/gnu, and on mirror sites around the world. Read gnu.announce, where each post announcing a new product is followed by a list of mirror sites all over the world.

The filename you want is glibc-1.07.tar.gz, which contains both the source code and the documentation.

I am looking for some technical details like process management, memory management, process coordination, file system, etc. of Linux OS for my term project. If you could direct me to get those information I would greatly appreciate it.-Raghib Muhammad, raghib@pegasus.montclair.edu

***LJ* replies**:

First, read the source. It's pretty readable, at least to me.

I have written the beginnings of a book about Linux internals called the Linux Kernel Hackers' Guide. It is not complete, but you can ftp it from tsx-11.mit.edu in /pub/linux/docs/LDP/khg* Also, read Bach's The Design of the Unix Operating System which is about AT&T's internals, but they are somewhat similar. Linus read that book while designing Linux. The KHG contains an annotated bibliography that might help you, as well.

I enjoyed reading the April issue of *Linux Journal* and have two questions on Linux that might be of interest to other readers. I would certainly appreciate any help or solutions.

Does Linux currently support or plan to support QIC-80 tape back-up via a NON-SCSI interface? What I specifically have in mind is an external tape unit that interfaces with the PC via the parallel (printer) port.

I have a date/time system problem with all Linux kernels that I have compiled beginning with patch level 14 and including the latest Linux 1.0, when I compile on my home computer, a 386DX with 8 MB and a math coprocessor. When I compile patch levels 12 and 13, I do not have this problem. When I boot with a floppy containing the newly compiled kernel, Linux starts up without the correct date/time. Instead it thinks it is Jan 1, 1970. If I use the same floppy kernel to boot on my PC at work (a 486DX with 8MB) the current date/time of the system is found. So it appears to be something related to compiling and then booting the kernel on my PC at home. However, this is not a problem with patch levels 13 or 12. Something in the kernel code changed beginning with patch level 14 that now causes difficulties in finding the system time on my home computer.-Donald Mugnai, Silver Spring, MD

***LJ* replies**:

As documented in the Ftape-HOWTO, there is currently no support at all for any tape drives that work off the parallel port. All of these drives use proprietary protocols. No one that we are aware of in the Linux community is currently working on reverse engineering those protocols so as to be able to use those drives. As far as your system time problem, have you tried booting an old patchlevel 12 or 13 kernel on your home PC lately? It might simply be a worn-out battery on your CMOS. If any readers have any better solutions, please send them in. This question might be a little too specific, or be in a FAQ. I was running Slackware Linux 0.99pl14 with a SCSI CD-ROM. The CD-ROM was mounted with:

```
    mount -t iso9960 /dev/sr0 /cd-rom
```

I later wanted to access a different CD, so I did:

```
    umount /cd-rom
```

Popped in the new disk, mounted it, and the system thinks the original disk is in the drive. This doesn't happen with floppies, so there must be a switch to mount that says it's removable media. Is there a special way to mount a CD-ROM so the fs will recognize that it's changed?-Joe Wronski, jwronski@mystery.bbn.com

Eric Youngdale, ericy@cais.com, replied:

Sounds like broken firmware on the drive. Any removable SCSI device should report a media change anytime you change discs, and for some reason this is not happening for you. If the drive were reporting the change, then all of the old buffers would be flushed.

One way to force the issue is to simply attempt to use the drive while the drive is empty. The scsi code notices that there is nothing there and does the right thing.

Joe replied:

Thanks for looking into this. Actually, I had been running a version of Linux earlier than 1.0 (.99pl14, I think). After finally building and installing 1.0, the CD-ROM media changes are detected and reported at mount time. All's as it should be now.

Is there a means with Linux to format and use 3.5" disks above 1.44Mb (1.7MB for instance) as easily as with MS-DOS ?-Xavier Cazin, p6ip051@cicrp.jussieu.fr

***LJ* replies**:

Yes. There are a set of patches on tsx-11.mit.edu that accomplish this. There is something in the directory /pub/linux/patches/ called fdpatch2for15.* which does what you want; it supports 3.5" 1.77 MB disks and 5.25" 1.44 MB disks. With that patch applied to use extra-high-formats, the standard fdformat should allow you to format any supported format.

The patches come with a patched version of mtools which will allow you to access DOS-formatted floppies at these formats. The Linux DOS filesystem should access them correctly without any changes, once the fd driver is patched to recognize the high-capacity formats.

All letters to the editor and submissions for Linux Q&A are subject to editing for clarity, length, grammar, and spelling. Send letters and questions to info@linuxjournal.com, and use a subject like LTE or Q&A. Alternately, send paper mail to *Linux Journal*, P.O. Box 85867, Seattle, WA 98145-1867. E-mail is preferred, because we can print the results of a discussion, which can be more useful than just your letter and our response.

Archive Index Issue Table of Contents

Advanced search

Linux Products and Events

LJ Staff

Issue #4, August 1994

MicroSat Ground Station Software for Linux and X-Windows, Toolkit for Linux CD-ROM (May 1994) and more.

This is the first ALPHA release (0.4) of my MicroSat ground station software for Linux and X-Windows.

The software is available by anonymous ftp from ftp.ucsd.edu in the file microsat-0.4.tar.gz. I uploaded it into the /hamradio/packet/tcpip/incoming directory, but hopefully someone will move it to the /hamradio/sat directory. I also uploaded it to ftp.funet.fi in the /pub/ham/incoming but should be moved into /pub/ham/satellite/microsat.

To run this software, you must be running a 1.0 (or greater) kernel with the AX25.012 release of the GW4PTS AX.25 package (available on sunacm.swan.ac.uk in /pub/Linux/Radio).

All the programs are written using the OpenLook toolkit, but should work with other X-Windows window managers (i.e., tm).

This release consists of the following programs:

- xpb - broadcast monitor
- directory - directory list viewer
- downloaded - downloaded file list viewer
- viewtext - uncompressed ASCII text file viewer
- upload - message upload application
- message - message preparation application

Also included are:

- lha - unpack an lhz compressed file
- unzip - unpack a PKZIP file
- xloadimage - an X-Windows image viewer that has been modified to display EIS images (directly from the .dl file, and from the extracted file)

You should note that this is still ALPHA release software (as is the AX.25 package for the kernel), and I am continuing to develop this software. I believe that it is now in a state that others can give me good feedback.

The release includes all the source, and I encourage others to help develop this software package. I would appreciate hearing from anyone who is planning to make fixes/changes/enhancements to the software, so that I can try to coordinate future releases.

Toolkit for Linux CD-ROM (May 1994)

The new Toolkit for Linux CD-ROM from Walnut Creek features the sunsite.unc.edu archive and the ALPHA and BETA directories from the tsx-11.mit.edu archive. Distributions include Slackware 1.2.0 and MCC. Also includes Xfree86 2.1 and 1.3, tcl/tk, gcc2.4.5, libc4.4.4, emacs 18.58 and 19.22, GNU Ada, lisp, Prolog, Fortran, rexx, Eiffel and more. $39.95 from Walnut Creek CD-ROM, 1547 Palos Verdes Mall, Walnut Creek, CA 94596, e-mail info@cdrom.com, phone +1 501 674-0783 or fax +1 510 674-0821.

Linux & Internet Congress Proceedings

The Linux & Internet Congress sold out in May, but the Congress Proceedings, with articles written by Linus Torvalds, Eric Youngdale, Remy Card, Stephen Tweedie, Bob Amstadt, Drew Eckhard, Dirk Hohndel, and others, is now available. Most of the articles are in English, some are in German. Approximately 380 pages, price 66.-DM (approximately $30 US) plus shipping. Contact:

JF Lehmanns BuchhandlungHardenbergstr. 1110623 BerlinPhone: +49 30 31592320, Fax: +49 30 3139177E-mail: bestellung@jf-lehmanns.de

The International Symposium on Linux

Linux users and developers are invited to attend the International Symposium on Linux, December 8-9, 1994, in Amsterdam, the Netherlands. The entry fee is approximately $75, or $50 for students, in an attempt to just cover the costs of the conference. If enough people participate, the entrance fee will be lowered.

For information, use anonymous ftp to beatrix.icce.rug.nl in /pub/symposium. There is a list of visitors, speakers, and cheap hotels in Amsterdam. You may also send e-mail to linux@icce.rug.nl . The Linux Symposium is organized by the ICCE, University of Groningen, specifically by Frank B. Brokken, Karel Kubat and Piet W. Plomp.

For those without e-mail, you may send paper mail to Karel Kubat, ICCE, Westerhaven 16, 9718 Groningen, Netherlands.

Yggdrasil Summer `94 Linux Plug-and-Play

by Phil Hughes

We just received a review copy of this package compliments of Walnut Creek CD-ROM (info@cdrom.com or 800-786-9907 in the US). I don't intend to write a comprehensive review here, just offer a quick look at what you get. Expect a more complete review in a future issue of *Linux Journal*.

Having used the Yggdrasil Fall 93 version of this product as well as both Slackware from Trans-Ameritech and on floppy directly from the archives on ftp.cdrom.com, I have some reasonable experiences for comparison. And the executive summary is that no Linux distribution is perfect (yet) but this one is certainly worth considering.

What Yggdrasil has attempted to do is give you a package that is easy to get up and running and is complete enough to satisfy most any Linux user. To make this possible you get a 95-page manual, CD-ROM and a 3.5" boot disk. The front cover of the manual tells you what is included (76,323 files, X-Windows, Andrew System, Networking, Games, Multimedia, Text editors, Desktop Publishing and Telecommunications) and the back lists the supported hardware. Thus, you know what you are getting before you have to open the package.

The first 1/4 of the manual takes you through the installation. But, even without reading the manual, it is easy. You boot from the floppy and load a live Linux system with the CD-ROM as your main file system. It's slow but it works. You then have options for how much to install on your hard disk with choices from 4MB to 1GB. If you select the custom installation (best choice as there is a bug in the standard installation) about 35MB of files are loaded. They you can use a tck/tk-based installer program to select the packages you want.

The hardest part of the whole installation is waiting for information to be loaded. I loaded about 200MB of files from an old Mitsumi CD-ROM drive with 8-bit controller and it took almost a day. But it worked. And I had a system that booted, recognized my ethernet adapter, configured X pretty much automatically and talked to my network with no hitches.

[Ed: I also ran Plug-and-play, on a machine with no hard drive at all, simply running off the CD-ROM, and was quite pleased with how easy it was to use. It was at work, and my boss was quite impressed.]

But, as I said, no distribution is perfect. Once you have your Plug-and-Play box up you can play a lot. But if you want to configure printers or any of the other mundane configuration jobs you will have to dig in and do it yourself. Also, on the negative side, the package selection menu is at the level of “do you want the Andrew system” and “do you want Emacs”. Reasonable but distributions like Slackware offer you a much finer set of choices—something that could be important if you are a little cramped for disk space.

The other reservation I have about Yggdrasil is that they have ignored the Linux file system standard. The standard isn't perfect (and it is really an evolving standard, not a cast-in-concrete one) but Yggdrasil makes no attempt to follow it. This may not be a problem as long as you stay with Yggdrasil distributions but could cause problems if you add packages from other distributions or archive sites in the Internet.

Are there any bugs? Most certainly. During the installation there were a few unexplained error messages. beach_ball blows up but xgopher works. There is a bug sheet that comes with the system. And I am sure there are more bugs. But Yggdrasil makes this information available. You can get a current bug report list by sending e-mail to query-pr@yggdrasil.com . And there is a built-in mechanism to report bugs via e-mail.

If you have a CD-ROM drive, Linux on CD-ROM is the way to go. The nominal cost is more than recovered in saved time and saved backup media. And “Plug-and-Play” is certainly worth considering.