Linux Journal Issue #11/March 1995

Features

News & Articles

Reviews

Columns

The Humble Beginnings of Linux

Randolph Bentson

Issue #11, March 1995

Randy Bentson reflects on the early days of Linux.

The histories of many programming projects are maintained by oral tradition. After all, what real programmer would take the time to write down what has happened? Because much of Linux was developed by way of e-mail conversations on the net, a slightly more firm record exists. The following is gleaned from those records.

How Many of Us Missed the Boat?

I first worked with Minix in Fall 1989. Dr. Tanenbaum's system was a perfect vehicle for working with operating systems for those who couldn't afford a VAX. It ran on an 8086 with 640 Kbytes and a floppy drive. You could run a few programs in a multi-tasking environment and, since you had the source, you could change the system to your heart's content.

“But wait,” you say, “Minix isn't Linux. What are you talking about?”

“I'm just setting the stage, bear with me a moment.”

The intended target for Minix was students of operating systems in a computer science curriculum. I used it in teaching an upper division class where the term projects were to “enhance the system in some meaningful way.” The projects varied from a serial port driver, to virtual terminals, to simple memory management. No one took the giant step that Linus Torvalds took at the University of Helsinki. (I wish I could say one of my students was changing the course of personal computing!)

The Foundation

As you may know, the memory model of the 8086 is very limiting. It had easy access to only 640 Kbytes of non-virtual memory. Ugh! But that was the target system for Minix because it was the most common and cheapest system available.

Linus rejected that argument and decided that one *needed* virtual memory to be able to do anything interesting. Thus, he reckoned that an 80386 was the minimum processor for his system.

His project was to build a kernel for a virtual memory, pre-emptive, multi-user system. It would have much the same user interface as Minix (in fact it used the same file system as Minix for some time) and that of Unix.

From the beginning, Linus made reference to the GNU portable kernel, Hurd, and made it clear that he wasn't planning to supplant Hurd. Since Hurd was expected to be available in late 1992, Linux was clearly just a hackers' delight.

Growing Use

By the time Linus conceived of his project in April 1991, Minix had changed to support the improved Intel processors, but there was still room for extension. Initially Linux was cast in terms of a Minix project, but by late summer the divergence was starting to show.

Early versions were labeled 0.01 (Sept. 91), 0.02 (Oct. 91), 0.03 (Nov. 91), etc., as a hint that they weren't really releases so much as snapshots of work in progress.

Linus gathered a few supporters who would exercise and enhance his work and who appreciated receiving (and contributing) fixes as quickly as they were developed. The kernel soon came to support all the system calls expected of a Unix kernel as more restrictions were removed.

Linus ported gcc and bash, so there was a basic compiler and command interpreter in place. (Although, to be precise, the compilations were done under Minix up to version 0.12.) There was some discussion in comp.os.minix about the wisdom of going off and starting another OS, but Linus had his dream, or, some would say, he was stubborn and he persisted.

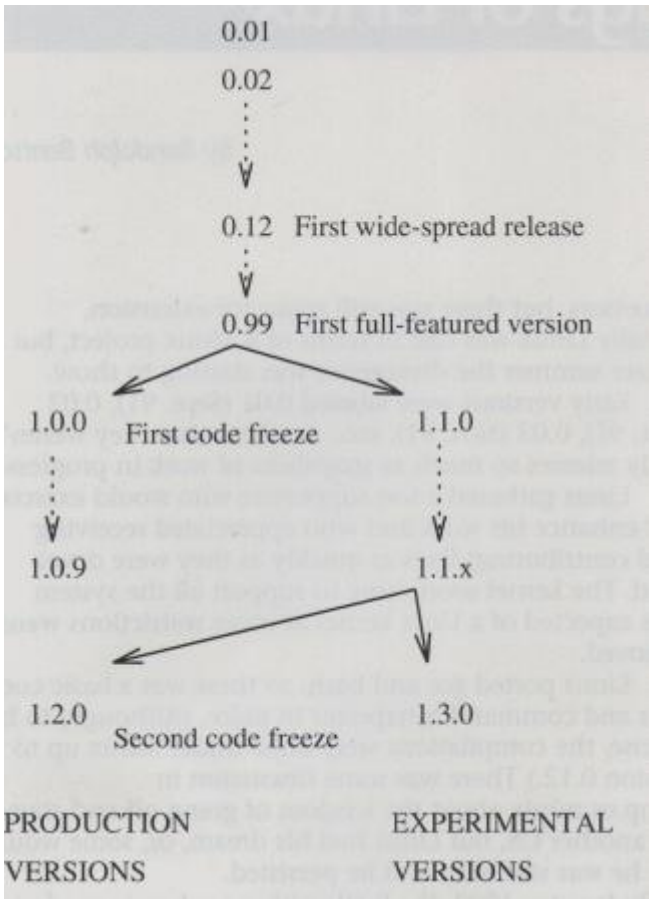
By January 1992, the 0.12 version took only modest care to build and operate and, thus, contributed a lot towards popularizing Linux.

Other Projects Enter…

It should be noted that this was not the only free Unix system for home computers. 386BSD was being developed in California and was a derivative of the Berkeley Unix that had been widely distributed on university campuses around the world. To some extent, 386BSD was a benchmark against which Linux was compared.

At the same time, the various GNU tools were becoming well established in the Unix domain. The standard C compiler, gcc, was regularly found to be better than most vendors' compilers, and the other tools were generally more robust and feature-full than the vendor versions. The fitting of the GNU applications to the Linux kernel was natural and necessary to the success of Linux.

The growing community of Linux users were not afraid to build up a system from sources around the world. A second outside product, the X Window System, provided a GUI interface for Linux users with high-end displays. A third product, NetBSD, provided a springboard to get full Internet support for Linux.



Production Releases

The initial numbering scheme had some limitations, but questions such as “Does 0.11 come before or after 0.2?” were safely avoided and the numbering quickly arrived at a limiting value of 0.99. That version was widely distributed,

and it was regarded as the first full-featured version of the Linux kernel. There was, by then, a sizeable community of users who depended on a stable version of the kernel. Although there were many patches and sub-patches to this version—often arriving daily—the basic version 0.99 was suitable for release.

The Great Release took place at the start of 1994, when Linus identified a stable patch level (0.99pl14r), cleaned up a few last problems, and called it good. This operation was called a “code freeze” and resulted in version 0.99pl15, which held steady long enough for bug fixes, but no enhancements, to arrive.

Part of the code freeze and the Great Release was the recognition that Linux had become a suitable foundation for production systems—systems devoted to doing useful work, instead of being the object of a programmer's machinations. This posed a dilemma: how could Linux continue to evolve and yet be stable?

The solution was simple: have two development paths starting from the same point. The even-numbered releases (1.0.0, 1.0.1, 1.0.2, etc.) followed a slow, careful evolution of a production release system and the odd-numbered releases (1.1.0, 1.1.1, 1.1.2, etc.) were to be the fast-changing, experimental system. Version 0.99pl15, with a few fixes, was the basis of these two systems. Some important fixes moved 1.0.0 to 1.0.9 in the early months of 1994, but that system development path has been unchanged since mid-year. By contrast, 1.1.0 underwent over 50 changes in the first 10 months.

Plans are now afoot for the next major release. Again, the stable and well-tested features of the experimental versions (up past 1.1.60) will be incorporated in a production release called 1.2.0. Its twin, version 1.3.0, will be the basis of yet more experimental work on the kernel.

One thing that also happened with the Great Release was the release itself no longer catalogued its changes. This shortcoming was alleviated when Russell Nelson, nelson@crynwr.com volunteered to distribute a change summary shortly after each patch was distributed.

Distribution Kits

All during the development of the kernel, concurrent development was being done on the tools I've mentioned, as well as others. One of the topics of discussion by users was what they collected for their system. Since new users didn't want to hunt the net for the critical pieces, the idea of a “standard distribution” was established.

One common medium of exchange has been the floppy disk, so the distribution kits have generally been cast in terms of images of MS-DOS-readable disks. One can copy a friend's disk set and then bootstrap Linux. If you're anywhere near a

large community, chances are there is a Linux or Unix users group nearby. If you're lucky, you'll find a set of floppies to borrow. If that fails, it's almost certain you'll find someone who will copy their distribution to your floppies.

Distribution kits include: Debian, MCC, Slackware, Software Landing Systems (SLS), SUSE, TAMU, Yggdrasil.

Publishers

These distribution kits are generally deposited or maintained on an ftp site and mirrored to other ftp sites. Many bulletin boards maintain copies of these distributions. This gives you a second path to acquire Linux: all you have to do is download 50 Mbyte via modem.

The third and, I think, most significant path to acquiring Linux is CD-ROM. A number of companies publish one or more (I've seen as many as four) distributions on a single CD-ROM. The companies add lots of other material, such as X-Windows, the GNU sources and snapshots of archive sites (which contain other, non-distribution kit software), to their packages and sell them for $20 to $40. Since you can easily spend $20 in floppy disks for a distribution kit alone, this is quite a bargain! When one can now buy a single-speed CD-ROM drive for less than $100, getting a distribution by way of CD-ROM is very attractive.

Some of the current Linux CD-ROM publishers include: InfoMagic, Morse Telecommunication, Nascent, Red Hat Software, Trans-Ameritech, Walnut Creek and Yggdrasil Computing, Inc.

It should be noted that distribution kits have different numbering than the kernel itself, and CD-ROMs may have yet another way of identifying versions. This can lead to confusion when someone refers to “the Fall 1993 release” or “the 2.0 release”. If you look at /usr/src/linux/Makefile, you'll find the version, patch level, and sub-level in the first few lines. Look at the README-type files in the root of the distribution to determine the kit's version.

It's Deja Vu All Over Again

My first experience with Unix was in 1980, when I was handed three 2400' reels of half-inch magnetic tape and a two-foot high stack of xerographed manual pages. I was pointed to the VAX and wished the best of luck.

Those were heady times, living on the edge, working without a safety net. One's phone list (of other system administrators) was critical to one's survival. Everyone (the system administrators and select students) had the source code, and one was expected to dive into the kernel and fix things.

But things got boring for a while in the late 1980s: vendors distributed only object files for their Unix systems and there were commercially-available support groups to call. One was expected to manage configuration files and submit bug reports—and then wait for a correction.

In a conversation just last week, I pointed out that those golden days are with us again, only better. First, the number of sites and kernel programmers has grown ten-fold or a hundred-fold, so there are more folks contributing fixes and improvements. Second, since we're running on personal computers, the effects of our changes are localized, and we're even more free to explore. Finally, with widespread Internet service, we're so much better connected to one another.

These are such interesting times!

Randolph Bentson (bentson@grieg.seaslug.org) has been programming for money since 1969-writing more tasking kernels in assembly code than he wants to admit. His first high-level language operating system was the UCSD P-system. For nearly 14 years he has been working with Unix and for the last year he's been enjoying Linux. Randy is the author of the Linux driver for the Cyclades serial I/O card.

Archive Index Issue Table of Contents

Advanced search

Review of Scilab

Robert A. Dalrymple

Issue #11, March 1995

Want to do serious equations solving? Need graphical output? Not into programming? Scilab offers a powerful solution.

When I first installed Linux, I was delighted to find that f2c (with the shell programs f77 or fort77) made FORTRAN coding possible and almost transparent, despite the lack of a FORTRAN compiler. But, after trying a number of publicly available graphics programs, I was unsatisfied with plots of my FORTRAN results. Also, I missed the mathematical power of such programs as MATLAB, which I also (and primarily) use for its great graphics. Then I tried scilab, which was released recently for Unix and Linux by INRIA (*Institut National de Recherche de Informatique et en Automatique*) of France. Although the graphics aren't as good as MATLAB's, scilab did solve most of my problems; so much so, that I installed it on my workstation at work as well.

Scilab is a fully featured scientific package, with hundreds of built-in functions for matrix manipulation, signal processing (complete with its own toolbox), Fourier transforms, plotting, etc. It is based on the use of matrices, which means that, with proper planning, you don't need to use subscripted variables in your programs. Scilab has voluminous help files, documentation and demo programs. Here, I will just outline some of what it can do. There is just too much to cover in one article.

The documentation you will need is found in directories under the main directory scilab-2.0. In doc/intro, the compressed PostScript file intro.ps contains the user's manual, *Introduction to Scilab*. This you will need for sure. In man/LaTeX-doc is Docu.ps, which contains a list of all the scilab functions. This is not really necessary as all of it is available on line via the help command or the help button on the scilab front end. The Signal Processing Toolbox manual in doc/signal shows examples of IIR and FIR filters, spectral analysis, and Kalman filtering.

As a direct descendent of MATLAB, its syntax is similar. For example, to define a vector x, we can type at scilab's **-->** prompt:

```
-->x=[ 1 3 5 8]
```

which is echoed back as:

```
x  =
   !  1.   3.   5.   8. !
```

(exclamation points denote a vector or matrix).

Matrix multiplications are trivial:

```
--> z=[ 7 8 9 10];
--> x * z'
 ans  =
   156.
-->x.*z
 ans  =
!   7.   24.   45.   80.  !
```

The first multiplication was the row vector **x** multiplied by the column vector **z'** (created by a transpose of the row vector **z** using the prime operator). This yields the single number 156. The second multiplication with the **.*** operator is an element by element multiply, resulting in a vector.

Solutions of matrix equations of the form **a x = b** are also simple. For example, let's define the 3x3 coefficient matrix **a**:

```
--> a=[2 1 3; 5 -3 1; 4 4 2]
 a =
!  2.   1.    3. !
!  5.  -3.   1. !
!  4.   4.    2. !
```

For **b=[1 29 -14]'**, a column vector (using the prime operator), we can find **x** several ways:

```
--> x=a\b
 x  =
!   2. !
!  -6. !
!   1. !
```

or alternatively **x=inv(a)*b**, where **inv(a)** produces the inverse of the matrix **a**.

To find and plot the **sin(2Pl x/25)** for **1<x<7lt;100**, first generate the sequence of numbers—**x** ranging from 1 to 100, counting by 1,

```
-->x=[1:100];
```

where the semi-colon is used to stop scilab from echoing back the numbers. To find the sine of all the numbers at once:

```
-->y=sin(2*%pi*x/25);
```

where **%pi** is the scilab intrinsic value for **PI**, and the vector y is the vector of the sines, with the first value **y(1)=sin(2*%pi*1/25)**, the second as **y(2)=sin(2*%pi*2/25)**, and so on. To see these values, we can plot them with **plot(y)**, as shown in Figure 1, which shows the scilab front-end along with the separate x-window plot that was generated automatically by scilab superimposed.

Figure 1

Help is available several different ways. Typing **help** at the scilab prompt, followed by a function name, will produce a window with the help text for that function. Or use the help button in the main window (shown in Figure 1) to create the Scilab Help Panel (shown in Figure 2). This method allows a search of the help files with the apropos command, shown here with a search for the keyword plot. There are 10 entries shown in the figure of the 28 available for plot. With a single click on fplot3d, an xless window pops open with detailed information on the use and the arguments of the function (shown in Figure 3).

One problem I solve often is $s=x \tanh(x)$ for x when s is given. It comes up in the problem of determining the length of a water wave of a given frequency in a known water depth. Since x appears in the argument of the hyperbolic tangent function, this is not an easy problem to solve, requiring an iterative solution method. Instead of writing a Newton-Raphson scheme as I do in FORTRAN, I use the **fsolve** function, which finds the zero of a system of nonlinear functions. First, let's find x given a single value of s.

```
-->s=.5
```

I then define the remainder, **r = s-x*tanh(x)**, which should be zero for the correct value of **x**.

```
-->deff(`[r]=g(x)','r=s-x.*tanh(x)')
```

The **deff** function defines **g(x)**, with single quotes about each part. Now to **fsolve**:

```
-->x=fsolve(.3,g)
  x   =
     .7717023
```

The value 0.3 is my initial guess at the answer. Let's check the answer by substituting it back into **g(x)**:

```
-->r=g(x)
 r  =
   - 1.110E-16
```

Our solution is good. If we had defined **s** as **[0.1 0.2 0.3]** and used **x=fsolve(s,g)**, we would get three solutions at once. (That's why I used **.*** instead of ***** in the definition of **r(x)**.)

Figure 2

Rather than typing the definition of g directly into to scilab, we can define a file —**wvnum**, for example, as the definition of the function **g(x)**. The file would look just like the deff argument given above, with two separate lines, but without the single quotes. We then call the definition into scilab with **getf(`wvnum')**. This can also be done through the File Operations button.

Figure 3

We can assure ourselves that there is only one positive solution for **x** by plotting **g(x)**, say, in the range from 0 to 5 by steps of 0.1:

```
-->fplot2d((0:.1:5),g)
```

fplot2d has the advantage over the plot function of specifying the range of the abscissa and plotting a function instead of a list of numbers. (Note there is a minor error in the help file example—reversing the arguments of fplot2d. This is one of the mistakes I have found. The worst was an error in scilab's attempt to convert a function definition into FORTRAN code. Be careful.)

Scilab has a variety of other plotting functions available. Histograms, contour plots, 3d plots, and plots of vectors (useful for flow fields) are all available. I sometimes use scilab to plot data from another program. By saving the data in an ASCII file, with known numbers of rows and columns, the data is read into scilab by a read command: **z=read(`datafile',m,n)**, where **(m,n)** are the numbers of rows and columns. Then the data can be contoured, for example, by **contour(1:m,1:n,z,10)** for 10 contour levels.

Plotting data in three dimensions is also straightforward. Using the **z** data from above, a similar call to **plot3d(1:m,1:n,z,45,45)** produces a 3D plot with a view point associated with the spherical coordinates 45 and 45 (in degrees). By setting the program to use color, **xset("use color",1)**, then **plot3d1** with the same arguments gives a color shaded plot. Looking through the demo program sources will show you how to animate this type of plot.

Printing figures is easy. One way is to simply use the Print button in the scilab graphic window. This sends the figure directly to your PostScript printer, if you

have one. The same thing is accomplished with the command **xbasimp(0,'foo.ps')**, which outputs the contents of plotting window 0 to a PostScript printer (despite what the documentation says). Using **xbasimp(0,'foo.ps',0)** will instead make a file named **foo.ps.0**, which can be printed with an external scilab program called Blpr. The file foo.ps.0 is not quite a PostScript file, as a preamble translating scilab abbreviations is missing. Blpr adds that preamble producing a PostScript file that can be redirected into a true PostScript file or printed directly. Scilab also comes with external programs to include PostScript figures in LaTeX documents, if you're not already using epsf.sty with LaTeX.

Long programs can be written in a file for use with scilab. This means you don't have to do everything interactively. These files are easily pulled into the scilab program by using the File Operations button, clicking the program file name and then the Exec button. Debugging a large program is relatively simple; just write the file and then run it repeatedly, making corrections as you go along.

There are lots of other things that I haven't mentioned, such as integration functions, ordinary differential equation solvers, nonlinear optimization tools, symbolic manipulation of polynomials and linear systems, interfaces to Maple, C, and FORTRAN, and many others. These you'll just have to try out on your own. But I think that you will agree the effort is worth it and that scilab does bring mathematical clout to the Linux environment.

Getting Scilab

Robert A. Dalrymple teaches coastal engineering at the University of Delaware and directs the Center for Applied Coastal Research. He uses Linux at home and work and has more fun with it than he should, as he has other things he is supposed to do!

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Introduction to LINCKS

Martin Sjolin

Issue #11, March 1995

A lot of noise has been made lately about “Computer Supported Cooperative Work” (CSCW) and “groupware”. This is an introduction to using an interesting CSCW product, which is licensed under the GNU GPL and is available for Linux. Another article will cover building a small CSCW application with LINCKS.

LINCKS is a multi-user, client-server, object-centered database system built on top of a storage server called NODE. The storage server provides LINCKS with an append-only, object-centered database and object development history (“from which object was this object created?”), temporal history (relationship between objects), and optional command history (which is not enabled by default). An object consists of an image section, where anything can be stored (text, images, sounds, etc.); a set of attributes which, in a sense, types the object; and a set of links to other objects.

The current main user interface, **xlincks**, is an X11 application that allows you to edit and browse a LINCKS database in a hypertext fashion, using Emacs-like commands. **xlincks** provides a basic environment for document editing in a work-group setting, but provides only limited mark-up facilities at this time. The hypertext nature of LINCKS, together with the advanced view support, allows easy sharing of information and re-use of parts of documents (section, subsection, or paragraph) simply by creating links to the parts to be used.

A view is a definition of how to present the data to the user. You can define your own views, which are stored and edited like any other objects in the database. You define the logical structure (parts and their order), the location in the database from which to retrieve the different parts and the formatting attributes for the objects. By applying different views to the same objects, you are able to look at a document, for example, as a “full document”, as a “document overview”, or as only the highest level, the table of contents. Any editing of an object performed in one view is automatically propagated to all views built from (containing) the same object.

Also, when several users edit the same documents in a database, LINCKS issues warnings when parallel editing of the same object (a section, subsection, paragraph, etc.) occurs. Thus, you can coordinate your work with other people working with you using a secondary channel (email, **talk**, telephone, etc.) for synchronization.

LINCKS has its origin as a research database system, developed at the Intelligent Information System Laboratory at the University of Linköping, Sweden. We released LINCKS under the GNU GPL in the Autumn 1993, to gain feedback from outside researchers and users. Since then, we have ported LINCKS to SunOS 4.1.x, SunOS 5.x, Ultrix, AIX, Dynix, Linux, HP-UX, IRIX, SCO, 4.3BSD Reno, and DEC OSF/1 Alpha. It is running under X11R5, X11R6, and OpenWindows (and even X11R4, but not with full functionality). We ported LINCKS to Linux to enable us to run demonstrations on a portable unix box.

Installing and Running LINCKS

To run LINCKS, you must be running the RPC port-mapper (normally started from /etc/rc.d/rc.inet2), a kernel with TCP/IP, and X11 for **xlincks**. Also, you will need at least 8MB of RAM to run the database servers and the X11 application interface **xlincks** on the same machine.

To install the LINCKS binary distribution (lincks2x.ybin.tgz), you first should create a new user who will own the database, perhaps called “lincks” (It is not a good idea to install LINCKS as “root”).

- As root, **cd** / and un-tar the zipped tar archive.
- If you created a special LINCKS user, change the owner of all the files in / usr/local/lincks/ directory by typing **chown -R lincks /usr/local/lincks**.
- Then make /usr/local/bin/dbpasswd set-user-id “lincks” by typing **chown lincks /usr/local/bin/dbpasswd** and **chmod +s /usr/local/bin/dbpasswd** to enable users to modify their own passwords, even though the files are owned by “lincks”
- Finally, running as user “lincks” start the database server by typing **lincks -s /usr/local/lincks/DB**

The next step is either to continue reading this article or run the interactive tutorial. To start the on-line tutorial, you invoke **xlincks** with **xlincks/usr/local/lincks/DB**. You will be prompted for your LINCKS login and password; give the user name “demo” and the password “demo”.

The following is the LINCKS directory hierarchy:

- usr/local/man/{manl,catl} for formatted and preformatted manual pages

- usr/local/bin/ for the LINCKS programs
- usr/local/lincks/ is the main LINCKS directory with a “ready-to-run” database in the DB subdirectory, postscript documentation, FAQ, README, etc. The bitmaps subdirectory contains X11 bitmaps which might be missing on few systems.

System Description

LINCKS consists of eleven different programs which can divided into three different classes: the database server (**monitor**, **netserv**, and **dbs**); general utility programs (**dbpasswd**, **dbdump**, **dbroot**, **t2lincks** and **cutoff** which are normal clients), and the main application program (**xlincks**).

Database Directory

Both the application program and the utility programs take the last argument to be a path to a directory with the LINCKS database files. If you are only running one database server, or if you get tired of typing the path to the database directory, simply set the environment variable **LINCKSDBDIR** to point to this directory and omit the path. A typical database directory might look like:

```
-r--r--r--   1 lincks   iislab      33554 Aug 23 01:27 .fonttrans
 -rw-r--r--   1 lincks   iislab         18 Sep 18 19:30 .indexfile.lock
 -rw-r--r--   1 lincks   iislab        158 Aug 23 01:28 .lincksrc
 -rw-r--r--   1 lincks   iislab     592883 Aug 23 02:07 1.dat
 -rw-r--r--   1 lincks   iislab      82368 Aug 23 02:07 1.mol
 -rw-r--r--   1 lincks   iislab          6 Aug 23 01:56 data.names
 -rw-------   1 lincks   iislab        494 Aug 23 01:27 groups
 -rw-r--r--   1 lincks   iislab      61944 Aug 23 02:07 index
 -rw-r--r--   1 lincks   iislab          6 Aug 23 01:56 molecule.names
 -rw-r--r--   1 lincks   iislab        305 Jun 28 17:08 passwd
```

The .lincksrc is the configuration file which tells the LINCKS software where to find the database directory, log file directory, executable, TCP/IP port, etc. The passwd file contains a user name, a user id, and an encrypted password for each user. Access protection is defined by the groups file and by an optional wrgroups file. The **dbs** processes, stores, and retrieves objects and their contents in the 1.dat and 1.mol files. The **monitor** process uses the **index** file for storing the object identifications for each object. The *.names files contain the names of all the *.dat and *.mol files, which hold the actual data.

Whenever a LINCKS program accesses the index file, it must create a .indexfile.lock file, which contains the program name, its process id, and current host name. For more information, please see the *LINCKS System Administration Manual*.

The Database Server

The **monitor** process serializes object creation, imposes access control, and provides for parallel editing notifications. The monitor allows you to define which objects a particular user has *read* and *write* access via the group file and the optional wrgroups file.

The **netserv** process handles connections from all clients. After validating the user name and password against the **passwd** file (which is changed using **dbpasswd**), **netserv** forks and creates the **dbs** process for each connected client.

Each **dbs** process interacts with one client. It retrieves and stores objects in the *.dat and *.mol files using the monitor to ensure unique object identifiers and to synchronize access to the common database files. The **dbs** process dies when the client process closes the connection.

Utility Programs

The **lincks** program is used to start a LINCKS database server. If the database server is interrupted by a system crash, an .indexfile.lock file exists, but no LINCKS software is running. You will receive a warning message. Just remove the lock file (first check for a running process, please!) and re-execute the lincks command.

The **dbstat** utility is used to check for a running database server (**monitor** and **netserv**). **dbstat** connects to the netserv and monitor processes and returns some status information, as shown in Figure 1, **dbstat**

You use **dbpasswd** to add or to change a user's password. It is installed suid to enable every user to modify his or her own password, which is encrypted by default. When you add a new user, you must edit the **passwd**, groups and, if you use it, wrgroups files by hand. When you are finished editing, restart the database server to enable the monitor to re-read the protection files.

To export or dump a LINCKS database, use the dbdump program, which exports a whole LINCKS database in text format. If you have stored any binary object in the database (images, sounds, object code, etc) **dbdump** will create a file for each of the binary objects. You use **t2lincks** to import an exported database. Of course, **t2lincks** can also be used to import an object or any information into a database—registers or articles, for example—as long as it is in the proper format (see the **t2lincks** manual pages).

Figure 1. dbstat

The two last utilities are not used very often—**dbroot** creates a completely empty database (it destroys any existing objects in the database if any already exist). You only run **dbroot** when setting up a new database. To garbage collect a LINCKS database run cutoff, which removes all objects which are not referenced by any other objects.

Application Programs—xlincks

To edit and browse a LINCKS database, you have to use the X11 interface application program, **xlincks**. If you are using a small screen, you would probably like to change the .Xdefaults as described in LINCKS.FAQ. Also, you might install lincks2x.ydb.tgz, which contains a database using smaller fonts. If you are installing LINCKS from the source distributions, see the script DB/scalefonts.

When started up, **xlincks** will create a series of windows. For each one, use the mouse to place it where you want it, then click the left button. (In these directions, if the mouse button is not otherwise specified, use the left button.) One of the windows should be the **xlincks** command menu (see Figure 2, "Logged in as **emptyuser4**").

Much of the information in a database is organized hierarchically. When you start up **xlincks**, you should get one or more windows which are your entry points into the database. In Figure 2, we have logged in as user emptyuser4. To see more of the database, "expand" (see "expanding" below) some item in a window, thereby using it as the root of a new window.

Figure 2. Logged in as emptyuser4

The following convention is often used: <<xxx>> usually means "xxx" is a placeholder. A placeholder indicates that the view calls for something of type "xxx" to be in this place. Editing the placeholder will then create something of type "xxx" in the database. A box (frame) around an item often means that it's something that can be expanded. This is, however, simply a convention used by some views.

On-Line Help

There are two kinds of on-line help: the help text associated with a particular button in the command menu and on-line documentation. You get the button help text by clicking on the button, about which you wish to know more, using the right mouse button.

When you log in, you should get a window called "Help!" All of the on-line documentation is organized under this window. You can "expand" the

appropriate items to see the documentation. This is not true for the tutorial account, but there is a button in the command menu called “Help Window ...” which will bring up the “Help!” window if you click on it.

Expanding or Following a Link

To expand something, click with the left mouse button on the thing you want to expand, while at the same time holding down the “Control” button. This will open a new window. The key command **meta-l meta-e** has the same effect. Or you click with the middle mouse button on the thing you want to expand while at the same time holding down the Control button; this will expand the item and reuse the window. The key command **meta-l meta-E** (note the capital E) has the same effect.

The previous paragraph described how to expand something using the default view, or GPD (General Presentation Descriptor) in LINCKS parlance. To expand something using a view other than the default, click on the “Expand ...” command menu button, which will bring up a menu of GPDs. Next, click on the item you wish to expand and then click (in the menu) on the GPD you wish to use. For example, if you wish to see something in “node view” (the entire database object), click on the item you want to see, bring up the menu, and click on the menu item “node”.

A Very Small Example

Using **xlincks**, log in using user name **emptyuser4** and password emptyuser4. You should get the windows, as in Figure 2.

First, we'll add a link in the “Empty User 4” window to the “Help!” window:

Figure 3. After adding a link

- Click on the item to which you wish to have a link (the source), in this case, the line “Things You Can Do In **xlincks** in the ”Help!“ window”, then
- Click on the item after which we wish to put the link (the destination), that is the **<<item>>** in the “Empty User 4”.
- Finally, click on the command menu “Add Link” button in the command menu.

Now the “Empty User 4” windows should look as in Figure 3, “After adding a link”. For fun, try to expand the “Simon Says” and see if you really get the “Help!” window.

Figure 4. After inserting a plural item

Now we'll start writing our upcoming article for *Linux Journal* about “Linux on the Road”. We prefer to store the article in our home directory (the “Empty User 4” window). First, move the cursor to the “Simon Says.” line in “Empty User 4” window. Then, insert a new item in folder view by giving the command meta-l meta-i (insert closest plural), which will create a sibling of the current item. (See “The xlincks User” manual or the next part of this series for more information.)

Now, replace the “item” as displayed in Figure 4, “After inserting a plural item”, with ”Linux on the Road“.

Then, move to “Linux on the Road” and expand it using **meta-l meta-e** (or control-left-button if you like), yielding the result in Figure 5.

Figure 5. Specify the expansion view

Now, replace the “empty: GPD name” with “full document” which is the name of the view (GPD) we'll use. Expand on the “full document” line, which gives us a full view of the document as shown in Figure 6. (You might need to move out of the line to make the change take effect, before you expand it.)

Figure 6. Full document view

Now we can start writing using the regular Emacs-like command for editing and the **meta-l** prefix commands for inserting and deleting structure items.

When the document is getting long, we tend to lose the overview of the different parts—we are no long able to see the forest for all trees. To get a top-level table of contents of our document, “Linux on the Road”, simply:

- Select the title of the “Linux on the Road” document.
- Press the “Expand ...” button on the command menu.
- Select/press the “document contents” in expand menu.

and we should get something like Figure 7.

Figure 7. Document Overview

Now, change the title or author field in one of the windows. Then move out of the field and watch what happens. The change propagates to all other views built from the same content object.

Lastly, we remove the “Help!” item from our home directory. Move to the “Help!” item in “Empty User 4” type **meta-l meta-r** (for remove closest plural) to remove it.

We must save to the database server to make our changes permanent, since all our editing so far has been done in the client. To save the contents of a window, we only have to press the “Store” button or type **control-x**, **control-s**.

Last words

In the next article, we will cover in more detail how to define your own views (GPDs) and their different parts (structure, access, expand, format, and auto-links) by building a very small application.

For more information, please send e-mail either to lincks@ida.liu.se or to the author. Bug reports are welcome to lincks-bugs@ida.liu.se. To subscribe to the LINCKS User mailing list, lincks-users@ida.liu.se, send e-mail to lincks-users-request@ida.liu.se, which is handled by a human.

Martin Sjolin is currently pursuing a PhD in the Department of Computer and Information Science, University of Linköping, Sweden, in the field of Artificial Intelligence. He is responsible for support and development of LINCKS, when he is not browsing the net. He enjoys cooking, backpacking, skiing, wind surfing, canoeing and reading, whenever he is not hacking on LINCKS or Linux/Mac!

Archive Index Issue Table of Contents

Advanced search

Languages

Robert Sanders

Issue #11, March 1995

If you like parentheses, Scheme may be the utility language you are looking for. Robert explains why.

The great thing about Unix is that you can write programs to do just about anything you want. The down side is that you may have to write a program every time you want to get anything done. A Unix system administrator—and the average Linux user is his own administrator—is faced with a seemingly never-ending stream of small jobs which are too tedious for human hands but too infrequent for a large programming effort.

Most of the programs I write are run once and thrown away. A significantly smaller number see action as often as once a week. I can count the number of programs run every day on one hand. Obviously, I can't afford to spend much time on any one program. I need a language in which it's easy to develop, easy to debug, and easy to extend. C, the traditional Unix programming language, doesn't offer any of these. In this article I'll introduce a language which does.

Scheme is closely related to Lisp, a language whose name once stood for “LISt Processing”. Lisp first saw the light of day in 1958; unfortunately, it has not become a stellar commercial success since then. In fact, common knowledge says that Lisp and its sister language Scheme are bloated and slow. While that may have been true in the bad old days when every programmer wrote in assembler and toggled the program into the computer's front panel with his teeth, today it's more important to maximize programmer productivity than to minimize machine cost. Scheme advances this goal by providing the programmer with a flexible but safe language which allows him to operate at a higher level than he would with C.

Scheme's Features

How exactly does it do that, you ask? First, and most importantly, Scheme provides automatic memory management. A C programmer must explicitly allocate and de-allocate every object he uses. If the program allocates more than he de-allocates, memory will leak and be wasted. If the program de-allocates too often, the program will behave incorrectly or even crash. Thanks to a process known as “garbage collection”, a Scheme programmer need only concern himself with allocation. He allocates an object when he needs it, and the Scheme runtime system frees it when the object is no longer needed.

Scheme provides a richer selection of data types than C does. While the C programmer has only numbers, characters, arrays, and pointers to choose from, the Scheme programmer has at his disposal numbers, characters, arrays, strings, lists, association lists, functions, closures, ports, and booleans. In addition, Scheme and C disagree on how to handle typing information: C assigns a type to each variable, and each variable may hold only values of that type. Scheme assigns no type to variables, but identifies each value with a type. One of the benefits of this approach is that it allows “polymorphism”, which means that one function can take arguments of many types. For example, you might have one function that could search for a word or a list of words.

An arguably peripheral issue plays an important part in making Scheme such a wonderful language for development: Scheme is available in both interpreted and compiled implementations. My experience with interpreted languages shows that development with an interpreted language leads to faster prototyping and debugging of the finished product. After two years programming in interpreted languages (mostly Perl and Scheme/Lisp), I cannot tolerate the edit-compile-run cycle that plagues the C programmer. With most Scheme interpreters, you can simply reload the particular function definition that you have changed. You have the full capabilities of the language at your disposal from the debugger, and you can even modify a running program!

Finally, a Scheme program is usually safer and more robust than its C counterpart. The Scheme primitives are type-safe—unlike the C primitives which will let you add a string, a character, and an integer, or cast any number to a pointer and then dereference it—and the Scheme environment provides significantly better error-checking than any C compiler could.

The Bottom Line

Why does C allow these deficiencies to exist? Do Scheme's conveniences come for free? Of course not.

As with most things in computer science, you are given three options: fast, cheap, and correct (you may pick two). The most common Scheme implementation, an interpreter, suffers from slowness and slightly inflated memory usage. Scheme compilers produce much faster code at the expense of larger executables and decreased (or totally removed) error checking. Which two of the attributes you pick depends on which two you need. Most programs don't need to be blindingly fast, but you can make Scheme fast if you need it.

Another drawback stems directly from the overwhelming popularity of C. Most external libraries and system interfaces are available as C-linkable libraries. Scheme users have little or no access to such libraries. My favorite Scheme implementation's solution to this is its Foreign Function Interface (FFI). An FFI allows a Scheme program to access variables and functions written in another language. Here's a short Scheme program that uses the FFI provided by Bigloo, a version of Scheme, to access the C function “printf” and the global system error variable “errno”:

```
(module ffi-example
        (foreign (int errno "errno")
                 (int printf
                      (string . foreign)
                      "printf"))
        (main show-errno))
 (define (show-errno argv)
   (printf "The value of errno is %d" errno)
   (newline))
```

Scheme can allow C the use of its functions through similar directives. With a decent FFI, Scheme and C programs can share data and interfaces as freely as two C programs.

Even the slowest Scheme interpreter is adequately fast for most of my day-to-day programs. Most Unix programs spend most of their time waiting for I/O to complete, and mine are no exception. The few programs that must be as computationally efficient as possible gain respectable increases in speed from using a Scheme compiler. In some cases, a program compiled by the Bigloo compiler at maximum optimization ran exactly as fast as the C equivalent compiled with “gcc -O2”. A trivial example is provided below. (See the table and two program listings).

```
time       language
0.72    gcc -O2
0.72    Bigloo -unsafe
1.03    Bigloo
2.92    SCM/compiled
3.00    Scheme->C
79.04   Scheme48
90.30   SCM
91.76   Perl5
109.04  GNU awk
174.36  Perl4
```

Bigloo Version

```
(module optest
       (main main))
(define (main argv)
  (let ((b (string->integer (cadr argv)))
        (j 0))
    (do ((i 1 (+ i 1)))
        ((> i b))
 (if (even? i)
          (set! j (+ j 1))
          (set! j (- j 1))))
    (display j)
    (newline)))
```

C Version

```
int
main(int argc, char *argv[])
{
  int i = 0, j = 0, b;
  b = atoi(argv[1]);
  while (i++ < b) {
    i % 2 ? j++ : j-;
  }
  printf("%d\n", j);
  return(j);
}
```

Scheme Idioms

Now that I've convinced you that Scheme isn't too expensive, I'd like to introduce you to programming in Scheme. The best reference for this is the "Revised Revised Revised Revised Report on Scheme" (R4RS), the great-great-grandson of the original Scheme language definition. The 1990 IEEE standard for Scheme is number 1178. However, I'll attempt to show the joy of Scheme programming with a few examples.

First, let me explain that Scheme departs from the esthetic norm for computer languages. Quite unlike a C or Fortran program—which is organized as a series of statements, usually one per line—a Scheme program consists of a series of parenthesized lists, "S-expressions" Each S-expression may define a new function or variable, invoke a function, or may simply be a literal data list. That's part of the genius of Scheme: program code and data are virtually indistinguishable. S-expressions can contain other S-expressions (which may contain other S-expressions, etc.). That means that Scheme's equivalent of statements may contain other statements, and that Scheme lists may contain other lists. To truly understand Scheme, you must be comfortable with recursive relationships like that.

Unlike most statement-oriented languages, Scheme has no operators. All actions are handled by functions or special forms, both of which exist superficially as S-expressions. C's "+" operator appears in Scheme as the "+" function. Functions are invoked by placing their names at the beginning of a code S-expression. For example, (+ 2 2) produces the number 4. The S-expression (display (+ 2 2)) prints the number 4 on the standard output. Some other S-expressions:

```
(define some-variable 12)
(define (some-function argument)
   (display argument))
(- some-variable 1)
(display (* some-variable 2))
```

Most people new to Scheme dislike the parentheses at first, but grow to enjoy them. Scheme's syntax is more regular than that of state-based languages, more conducive to language-sensitive editing, and easier to manipulate with macros. (Macros are beyond the scope of this article.)

Conditional Expressions

Scheme provides the familiar “if” expression for conditional execution of code. One difference between C's if statement and Scheme's if statement is that the Scheme “if” statement returns a value. In fact, all Scheme statements return a value. This property of Lisp, coupled with the purported inefficiency of Lisp systems, caused some wit to comment, “Lisp programmers know the value of everything and the cost of nothing.”

This function prints an “s” if the number passed to it is not 1.

```
(define (plural-print-s num)
  (display (if (eq? num 1) "" "s")))
```

Another conditional statement is the “cond” form; rather than a simple either-or choice, “cond” evaluates each of several tests and executes the corresponding expression of the first one to evaluate to true.

```
(define (print-type object)
  (cond ((number? object)
         (display "number"))
        ((string? object)
         (display "string"))
        ((list? object)
         (display "list"))
        (else
         (display "I don't know that type"))))
```

Recursion and Iteration

Most C programmers will be familiar with the “for” and “while” iterative loops for repetition. Recursion is a little less-known in the C world. Scheme provides several very powerful methods of repetition in both iterative and recursive forms.

This recursive function prints a list of numbers with each number incremented by one.

```
(define (print-list+1 arglist)
  (if (pair? arglist)
      (begin
      (display (+ 1 (car arglist)))
```

```
        (newline)
        (print-list+1 (cdr arglist)))))
```

(car and cdr are functions that return the first element of a list and a list minus its first element, respectively.)

Recognizing that applying an operation to each element of a list is a common operation, Scheme provides the “map” function. Here is the same program written using “map”:

```
  (define (print-list+1 arglist)
    (map (lambda (arg) (display (+ 1 arg)) (newline))
        arglist))
```

The “lambda” form is similar to a function definition but the resulting function has no name. Although this function may not be called by name, it may be passed to other functions that take functions as arguments (confused yet?). In this case, “map” takes as its first argument a function. It then applies that function to each element of its second argument, which must be a list.

Scheme also provides the “do” loop, which functions almost identically to C's “for” loop.

Scheme Implementations

Because of its simplicity and simple syntax, Scheme has become a favorite of language implementors, and as a result, dozens of Scheme implementations are available for free. These may be divided into two categories: interpreters and compilers. (Well, more correctly, the compilers usually include both compilers and interpreters.)

My favorite compiler is Bigloo. Written by Manuel Serrano of France's INRIA, Bigloo compiles Scheme code to efficient C, which is then compiled by gcc into executable code. Because of this method, Bigloo is both portable and open to improvements in the system C compiler. As mentioned earlier, programs compiled by Bigloo usually run within a few percent of the C equivalent program. Manuel, currently in the throes of his doctoral dissertation, may be reached at Manuel.Serrano@inria.fr.

Other compilers include the widely used Scheme->C, produced by Joel Bartlett (who may still be reachable as bartlett@decwrl.dec.com) at DEC's Western Research Lab. Programs compiled by Scheme->C aren't quite as fast at simple tasks as those compiled by Bigloo, but Scheme->C sports a much more advanced garbage collector. For programs with large data sets, Scheme->C may be a wiser choice.

Chez Scheme is also available for Linux. Chez is a venerable compiler written by R. Kent Dybvig (dybvig@cs.indiana.edu), one of the authors of the de facto Scheme standard (R4RS). Chez Scheme is not free.

Several Scheme interpreters have gained popularity in recent months. One of the hottest recent products is STk, an interpreter which makes John Ousterhout's Tk easy-to-use widget set available from an object-oriented dialect of Scheme. STk isn't the best performer among the interpreter crowd, but it easily bests Ousterhout's Tcl.

Aubrey Jaffer's SCM, one of the most mature Scheme interpreters, offers small size, high speed, and a growing library of extensions. These modules include POSIX interfaces, socket I/O, and a curses screen management library. SCM's author also maintains a library of helpful Scheme functions in a package called SLIB. I use SLIB in several of my code examples. The author, Aubrey Jaffer, may be reached as jaffer@ai.mit.edui.

The Free Software Foundation recently began an effort to provide a standard scripting and application extension language for their products. This language, called GUILE for some acronymic reason that escapes me now, will be based upon the SCM interpreter. To find out more about GUILE, send mail to gel-request@cygnus.com (that's not a misspelling!).

Finally, a group of Scheme enthusiasts at MIT (Scheme's birthplace) have undertaken an ambitious project to make Scheme as practical at Unix scripting as the Bourne and Korn shells. Their Scheme shell (scsh) combines the superior Scheme language with the powerful process/pipe-based data flow mechanism of the Unix shells.

Program Examples

Here are some program examples to give you a feel for programming in Scheme.

Program 1

Program 2

Listing 3

Program 1 is the standard fibonacci algorithm, and Program 2 is the standard recursive factorial. These should work under any Scheme implementation. Listing 3 shows an implementation of the tried-and-true Unix->DOS file conversion utility for the SCM interpreter and SLIB, its library package. It reads a file of linefeed-terminated lines and outputs a file whose lines are terminated

by carriage returns and newlines. Pay special attention to the definition of the function chomp; this function splits a string into a list of characters, filters out all the unwanted characters, and collapses the list back into a string. Listing 4 shows a definition of chomp that corresponds more closely to what a C programmer would write. Note the striking difference in complexity.

Listing 4

Listing 5

In Listing 5 is a program which takes a list of files on the command line and arranges them into disk-sized groups. This process will be familiar to those of you who installed Linux from floppy disks. The program illustrates the power of Scheme's list-manipulation procedures. map and for-each both execute a function for every member of a list. find-if returns the first element of a list that matches specified condition. These and other list techniques are usually implemented in C with cumbersome, error-prone loops.

Parting Words

Every language has its place in the programmer's toolkit. I don't use Scheme for every task—most of the time I use Perl, and sometimes I even use C. However, Scheme lets me write error-free programs in less time, with less effort and less pain, than almost any other language would. It provides me with many facilities that I would have to write for myself if using some other language, and unlike some other very high level languages, can be compiled to blindingly fast native code.

Obtaining Scheme for your Linux Box

I maintain an archive of Scheme interpreters and compilers pre-compiled for Linux. To retrieve one of these, ftp to ftp.mindspring.com and look in the directory named /users/rsanders/lang. You will find these files:

- bigloo-bin.tar.gz The Bigloo compiler version
- bigloo-elf-bin.tar.gz Bigloo with ELF shared libraries
- scheme2c-bin.tar.gz The Scheme->C compiler from DEC
- scheme2c-elf-bin.tar.gz Scheme->C with ELF shared libraries
- scm-bin.tar.gz Aubrey Jaffer's SCM interpeter
- slib.tar.gz Jaffer's SLIB library
- stk-bin.tar.gz The Tk-compatible Scheme interpreter

If your system is capable of compiling and running ELF binaries, then I suggest you use the packages that contain ELF shared libraries. The use of these shared libraries can reduce application startup time and size by up to 180 KB.

Bibliography

Revised Revised Revised Revised Report on the Algorithmic Language Scheme (R4RS) - William Clinger, Jonathan Rees et al. Postscript and DVI versions available via anonymous FTP from swiss-ftp.ai.mit.edu in /archive/scheme-reports. Also available in HTML form from swiss-ftp.ai.mit.edu as /archive/scm/HTML/r4rs*. The Usenet newsgroups comp.lang.scheme and comp.lang.lisp. Introductory books (from the comp.lang.scheme FAQ):

1. Daniel P. Friedman and M. Felleisen.*The Little LISPer*MIT Press (Cambridge, MA), 3rd printing, 1989.ISBN 0-262-56038-0. Science Research Associates (Chicago),3rd ed, 1989. 206 pages.

   Good for a quick introduction. Uses Scheme instead of Common Lisp. (The book uses a dialect of Scheme with footnotes about translating to Scheme or Common Lisp. The footnotes won't allow a non-expert to use Common Lisp for the advanced chapters because of the complexity.)
2. Brian Harvey and Matthew Wright*Simply Scheme: Introducing Computer Science*MIT Press, Cambridge,MA, 1994. 583 pages.ISBN 0-262-08226-8. $49.95.

   This book is ideal for students with little or no previous exposure to programming. The book is designed to be used before SICP (the authors call it a SICP “prequel”), and makes Scheme fun by sheltering the students from potentially confusing technical details. Unlike Pascal or C, the emphasis is on ideas, not obscure matters of syntax and arbitrary rules of style. High schools who have shied away from using Scheme because they found SICP to be too challenging should consider using this book instead.

   The text gradually and gently introduces students to some of the key concepts of programming in Scheme. It starts off with functions and function composition and continues with the notion of functions as data (first-class functions) and programs that write programs (higher-order functions). Since the complexity of the language is hidden, students can get involved in some of the more interesting and fun aspects of the language earlier than in other texts. Then the book progresses through the more complicated concepts of lambda, recursion, data abstraction and procedural abstraction, and concludes with sequential techniques, but with careful attention to topics students often find difficult. There are five chapters on recursion alone! There's also a pitfalls section at the end of most chapters to help students recognize and avoid common errors. The book uses several programs as examples, including a tic-tac-toe program, a pattern matcher, a miniature spreadsheet, and a simple

database program. Source code for the programs is available by anonymous ftp from anarres.cs.berkeley.edu:/pub/scheme/, or for $10 on IBM or Macintosh diskettes from the publisher.

3. Harold Abelson and Gerald Jay Sussman, with Julie Sussman.*Structure and Interpretation of Computer Programs*MIT Press (Cambridge, MA) and McGraw-Hill (New York), 1985.542 pages. ISBN 0-262-01077-1 $55.

   The teacher's manual, which is also available from MIT Press (ISBN 0-262-51046-4 $20), does NOT contain solutions to the exercises, but does contain hints on teaching with the book.

   Starts off introductory, but rapidly gets into powerful Lisp-particular constructs, such as using closures and engines, building interpreters, compilers and object-oriented systems. Often referred to by its acronym, SICP, which is pronounced “Sick-Pee”. This is the classical text for teaching program design using Scheme, and everybody should read it at least once. MIT problem sets are available from the repositories, and materials from Gustavus Adolphus College are available from ftp.gac.edu:/pub/SICP/.

4. George Springer and Daniel P. Friedman*Scheme and the Art of Programming*MIT Press and McGraw Hill,1990, 596 pages. ISBN 0-262-19288-8, $50.

   Introduces basic concepts of programming in Scheme. Also deals with object oriented programming, co-routining, and continuations. Gives numerous examples. Has more of an emphasis on teaching Scheme than SICP, and can be seen as an alternative to SICP. Source code from the chapters is available from ftp.cs.indiana.edu:/pub/scheme-repository/lit/sap/

Robert Sanders (rsanders@mindspring.com) works as the senior engineer at MindSpring Enterprises, an Atlanta-based Internet Service Provider. Ever since he escaped his role as dosemu's author, he's been hard at work studying computer languages and their implementations. He likes Linux.

Archive Index Issue Table of Contents

Advanced search

Linux Events

LJ Staff

Issue #11, March 1995

Awards, UniForum and more.

Software Development `95-Excellence In Programming Award

Dr. Dobb's Journal is presenting Linus Torvalds with a "*Dr. Dobb's Journal* Excellence In Programming" Award in a special presentation on February 16, 1995, 11:00 a.m. at the Moscone Center in San Francisco.

UniForum and Usenix, March 1995

UniForum, March 14-16, Dallas, Texas-UniForum will have a Linux Showcase, a clustering of Linux Exhibitors. Linux topics will be presented to conference attendees and there will be various Linux BOFS. (Birds of a Feather Sessions).

Usenix (Technical Conference):E-mail: office@usenix.org WWW URL: www.uniforum.orgPhone: 510-528-8649

UniForum (Exhibits) Registration Phone (The Interface Group): (617) 449-6600, FAX: (617) 449-2674.

University of Washington Computer Fair

March 16, HUB Auditorium, UW, Seattle, Washington, Phil Hughes speaks on "Linux-THE Unix for PCs".

DECUS `95 in Washington DC

DECUS (Digital Equipment Computer Users Society) Conference, May 6-11 at the Washington DC Convention Center. Linus Torvalds will be speaking. Two Day Linux Track on May 10 and 11. Contact: DECUS Customer Service 1-800-DECUS55 (US or Canada) or Phone: 508-841-7800. FAX: 508-841-3357. E-mail: information@decus.org.

Pentiums and Non-Pentiums

Phil Hughes

Issue #11, March 1995

Scared of Pentiums? Looking for an alternative while you are waiting for the Alpha port? There are choices.

We just went through the “Pentiums don't always divide correctly” fiasco. This got a lot of people talking about Linux ports to other hardware such as the PowerPC and the DEC Alpha. While these ports are a good idea and are progressing, there are also some alternatives to Intel's Pentium chip that will run the same code.

The first of these *Non-Pentiums* is the K5 from Advanced Micro Devices (AMD). This chip promises pin-for-pin compatibility with the Pentium and promises greater internal speed for the same clock rate. To accomplish this, the K5 decodes X86 instructions into internal, long-word instructions that have fixed fields and word lengths—a technique pioneered with the RISC architecture. Converting the variable-length, variable-format X86 instructions to this consistent format makes it possible for the hardware to schedule up to four instructions in one clock cycle. AMD claims this will make it possible for the K5 to offer sustained performance 30% greater than an Intel Pentium running at the same clock rate.

Another maker of a *Non-Pentium* is NexGen Microproducts with their Nx586 chip. The Nx586 uses a similar internal architecture to the AMD chip and claims a 10% speed improvement on integer arithmetic. The Nx586 lacks an on-board floating point unit, but the level-2 cache controller is on the CPU.

The third choice is the M1 from Cyrix Corporation. There are currently no details available on this chip but it is another chip designed to match the capabilities of the Pentium.

While all this is happening, Intel is also working on more models of the Pentium chip. For example, they have produced a power-managed 75MHz Pentium designed for notebooks.

It seems competition is good incentive for companies to produce better, more inexpensive chips. Maybe this time next year you will have to pick between a Pentium-based, K5 based, Alpha-based or PowerPC-based laptop running Linux, each for under $1000.

[Editorial note: I think he is dreaming, but I wouldn't complain, either.]

Phil Hughes is the publisher of *Linux Journal*.

Archive Index Issue Table of Contents

Advanced search

What's GNU?

Arnold Robbins

Issue #11, March 1995

In this two-part article, Arnold shows us how small is beautiful when it comes to user interfaces.

This column briefly describes *Plan 9 From Bell Labs*, an operating system done by the original group at Bell Labs that did Unix. We will be focusing on the user interface part of Plan 9. It is interesting, since the major components are either freely available from AT&T, or have been cloned in freely available software. The article will be concluded next month.

In the late 1980's, the research group at Bell Labs started to feel that Unix had reached the end of its useful life as a research vehicle. They decided that it was time to start over, taking the useful lessons learned from Unix, and going on from there. A brand new operating system was developed, named *Plan 9 From Bell Labs*.

The result is documented in two sets of papers. The early papers discuss the overall design of Plan 9, its shell, compiler, and window system. The later set contains additional papers about the system and the entire reference manual for the system. What is really neat is that PostScript for all of this is available via anonymous ftp (see the sidebar). The reference manual is huge, over 650 pages; it helps to print it on a duplexing printer, if you have one available. A mailing list of Plan 9 licensees and other folks who are interested in Plan 9 is also available.

Plan 9 is a distributed system. It consists of three components: File servers, where all the user files live; CPU servers, where computing intensive tasks are done; and terminals, which handle the user interface. The compute and file servers are large machines that live in a machine room. At Bell Labs, they are connected by a high-speed fiber network, although the software does not require this. The terminals are small computers with mice, keyboards, bitmapped displays, and network connections to the file and compute servers.

Terminals may have local disk drives for performance reasons, but they are used for caching files and are not strictly necessary.

Plan 9 is also a heterogeneous system. The operating system has been ported to the MIPS, Motorola 680x0, Intel 80386/486, and Sun SPARC architectures. At Bell Labs, they tend to use the MIPS systems for their servers and the other systems for the terminals, but again, that is not built in to the software.

Plan 9 also has a number of nice innovations in the software architecture seen by the programmer. As a simple example of this, in Unix, there are multiple system calls that affect the meta-information about a file (owner, mode, etc) such as **chown**, **chmod**, and **utime**. In Plan 9, there is only one, **wstat**, which writes the **stat** information about a file. As another example, all user and group names are returned by the system as strings, the programmer never has to manage the conversion between numeric user ids and strings. There are many other very elegant improvements upon the Unix design in Plan 9.

Plan 9 is also one of the first systems to use Unicode, a 16 bit character set. The **sam** and **9term** programs discussed below also support Unicode, making it possible, for example, to type a real smiley character, instead of the usual three-character ASCII glyph.

(A parenthetical note on my soapbox. In many ways, Plan 9 is a considerably superior design over Unix. It would be worthwhile for those interested in a free version of Plan 9 to consider starting from the Linux code base, using the device drivers, memory management, and whatnot. Linux itself is and will remain a Unix clone, and Unix is not Plan 9. Starting from Linux will be particularly easy when Linux 2.0 comes out, as it will be multi-platform, like Plan 9, or so I'm told.)

This should whet your appetite. Both the early and the current Plan 9 papers are well worth reading. The manual is also fun to browse.

Plan 9 is not (unfortunately) generally available. Universities may license it from AT&T for no cost (other than time spent by the lawyer to review the license). Upon signing a license, AT&T sends one hard-copy of the manual and a CD-ROM. The current (as of December 1994) release is almost two years old, and the system has evolved somewhat. A new release, using PC based hardware as the porting base, is in preparation, but no release date is known yet. The AT&T researchers are working towards a way to release it more generally, but it will still require some kind of license; it will not be freely available the way Linux, NetBSD, or FreeBSD are.

In this article, we will take a look at the Plan 9 editor, windowing system, and shell. They are important, because the editor is freely available, and there are freely available clones of the others.

The sam Editor

The Plan 9 editor is named **sam**. (Some history here. The original Unix editor was **ed**. It was command-driven. Rob Pike wrote a mouse-driven editor for the Blit terminal named **jim**. The successor to **jim** was **sam**, also written by Rob Pike. Basically, they're all a bunch of friendly, down to earth sort of programs.... :-) (I'm told that **sam** is short for “samantha”, and female.)

sam is a multi-file, multi-window editor that elegantly combines extended regular expressions (**egrep**-style) and the powerful **ed** command set with mouse driven text selection, cutting, and pasting. In particular, all operations act upon the selected text, which can include *multiple* lines. Replacement text can include newline characters as well.

sam also provides an infinite “undo” capability, so you don't have to worry about making mistakes.

One of the windows that sam provides you is the command window, where you type in commands. What is nice is that, just like the text in any other window, you can edit the text in the command window, then select the edited line with the mouse, and send it again as input. In other words, you can edit previous commands and submit them for execution again. If a substitution didn't work or do quite what you wanted it to do, undo the change, edit the command, and try again. Do this as often as you like. Or, if you used a series of commands on a chunk of text once, and need to do that series again, select all the command lines, and send them all at once. (The command window is similar to the mini-buffer in Emacs.)

As an example, when replying to email, I'll often include the original letter, preceded with > signs. Sometimes I end up with text that has only part of a line, like this:

```
> So what
> is your opinion about the future life of
> systems like MVS, VMS, VM, and Solaris?
```

I can select these lines as a single group, and then reformat it with the following commands:

```
s/^> //g
|fmt
s/^/> /g
```

This removes the > signs, runs the text through **fmt** to make it look nice, and then adds the > signs back in. The result might be:

```
> So what is your opinion about the future life of systems
>like MVS, VMS, VM, and Solaris?
```

(In fact, I was able to snarf the commands out of my article text, paste them into the command window, edit them a bit, and then submit them to make the new text above.)

The command language is particularly powerful, using a notation called "structural regular expressions". Essentially, regular expressions can be cascaded together to select increasingly more specific chunks of text upon which to operate. Here is an example from the **sam** paper. Suppose you wish to change all occurrences of a variable n to now be called num. You could use the following command:

```
, x/[A-Za-z_][A-Za-z_0-9]*/ g/n/ v/../ c/num/
```

The comma selects all lines (an abbreviation for 0 through **$**, the last line). The **x** command extracts text to operate upon. It is an iterator, meaning that the command following it will be executed for each match of the text. The sam paper explains the rest of the command: "The pattern **[A-Za-z_][A-Za-z_0-9]*** matches C identifiers. Next, **g/n/** selects those containing an n. Then **v/../** rejects those containing two (or more) characters, and finally **c/num/** changes the remainder (identifiers **n**) to **num**." The **g** and **v** commands are conditionals. **g** says execute the command only if the pattern matches; **v** is the opposite—execute the command only if the pattern does not match.

Simple changes are often made with the mouse. But for complex, sweeping changes, a command language such as the one in **sam** is essential. Indeed, this is why **vi** includes the **ed** command set as a subset.

As mentioned, sam is a multi-file editor. You can have several files open in windows at once, and several windows on the same file. This is particularly useful for cut and paste operations when going from one file to the next. The command language also provides commands for doing operations on all files that contain, or do not contain, a particular regular expression.

To summarize why I find sam attractive:

1. It is multi-file and multi-window.
2. It has a powerful command language that makes many editing operations easy.
3. It is possible to edit your commands.

This last is particularly useful; it is one of those things that once you have it, you can't believe you ever lived without it.

sam is implemented on top of two libraries. The **libframe** library provides windows (frames) of text. This library is implemented in turn on top of libg, a graphics library. For Unix, the Plan 9 **sam** and **libframe** code is used, essentially unchanged, on top of **libXg**, an implementation of **libg** for X windows using the Xt toolkit. All of this software supports Unicode. It is possible, for example, to enter the 1/2 symbol by typing **ALT-1-2**.

See the sidebar for the **ftp** location of **sam**; AT&T has graciously made it available free of any licensing worries. There is also a mailing list for **sam** users.

The mailing list archive includes a **sam** emulator for Emacs, written by Rick Sladkey (jrs@world.std.com).

The 9term Terminal Emulator

The Plan 9 windowing system is called 81/2, since it was the Eight and a half'th windowing system that Rob Pike had written. To create a new window, you select **New** from the right button menu, and sweep out the window you want. That window then runs a shell (**rc**, discussed below).

What is unusual about the windows in 81/2 is that you can edit the text directly in the window. Thus, if you make a typing error in a command, you fix the error, select the entire line, and resubmit it. This obsoletes the need for built-in command history as found in current Unix shells, such as **ksh**, **bash** and **tcsh**.

Furthermore, windows provide more complicated text editing capabilities. By pressing the ESCAPE key, the window goes into “hold mode”. All text is kept in the window. It is not sent to the program running in the window until the ESCAPE key is hit again, leaving hold mode. Hold mode is indicated by an extra white border inside the window, and a larger, white arrow for the mouse cursor.

You might use this, for example, when sending mail to someone. Run the **Mail** command, and then go into hold mode. Type in, edit, and rearrange your letter to your hearts content. Then leave hold mode, and out goes your mail to the **Mail** program.

Finally, by default, 81/2 windows do *not* scroll. Instead, text just buffers up inside them until you are ready to look at it. The down-arrow key on the keyboard allows you to quickly move through the buffered text. You can use the button 2 menu to change the behavior of the window so that it does scroll.

The non-scrolling mode is a feature; it obviates the need for pager programs like **more**, **pg**, and **less**.

The Unix program that emulates 81/2 windows is called **9term**. **9term** was written by Matty Farrow, at the University of Sydney, in Australia. It adds an additional library, **libtext**, on top of **libframe** and **libXg**. **9term** is both small and fast. On a Sun SPARCstation LX, **9term** starts almost instantly, while an xterm can take several seconds to start up, noticeably longer.

Besides hold mode described above, **9term** provides cut and paste editing and the ability to search backwards or forwards in the window for a particular piece of text (whatever is currently selected). **9term** also uses the up-arrow key to allow you to go backwards in the window, something that must be done with the mouse in 81/2.

The **9term** interface is consciously similar to that of **sam**. Button 1 in both programs selects text, button 2 supplies the editing menu, and button 3 provides the control menu. Double-clicking button 1 at the end of a line selects the whole line, and double-clicking in the middle of a word selects the word. Finally, in both programs, menus “remember” the last command issued. Thus, the next time you call up a menu, the default action is to do what you did last time, which is often what you want to do.

Having the identical interface makes using your system much easier; you don't have to mentally “switch gears” when moving from one window to the next: your mouse and keyboard work the same way, no matter what.

9term is fast because it is simple. Unlike **xterm**, it is not emulating a real terminal (or two or three), trying to interpret and process escape sequences. This means, for example, that you can't run **vi** or **emacs** inside a **9term** window. On the other hand, why would you want to? **sam** is considerably more powerful than **vi**, and much easier to learn than **Emacs**. My intent is not to start Yet Another Religious Editor War. Rather, the philosophy is that **9term** doesn't have to be complicated to support screen editors, since a powerful editor, **sam**, is already available.

In the same directory as the **9term** distribution there is a tar file with a large set of Unicode fonts for use with X. I particularly like the **pelm.latin1.9** font.

The sidebar describes where to get 9term and the Unicode fonts.

In next month's conclusion, we'll discuss the shells to run inside **9term**, and the **9wm** window manager that completes the 81/2 emulation.

Arnold Robbins (arnold@gnu.ai.mit.edu) is a professional programmer and semi-professional author. He has been doing volunteer work for the GNU project since 1987 and working with Unix and Unix-like systems since 1981.

Archive Index Issue Table of Contents

Advanced search

Installing Linux via NFS

Greg Hankins

Issue #11, March 1995

Greg Hankins describes how to install Slackware Linux over a network.

One of the easiest and quickest ways to install Linux is over a network, using NFS (Network File System). All you need is a machine that has the full Linux distribution available for NFS mounting. In this article, I'll describe in detail how to install Linux via NFS using the Slackware 2.1.0 distribution, although the concept of installing operating systems over the network is by no means specific to Linux or Slackware.

The first step in the installation process is to make sure that the filesystem containing the distribution is available for mounting on your soon-to-be Linux box. Talk to your sysadmins and ask them nicely to export the filesystem for you to mount. You'll also need to get an IP address for your Linux machine. Your sysadmin can help you with this, too. While you're gathering information, find out your netmask, gateway address, the IP address of the machine containing the distribution, and what the path to the distribution directory is. Write these down; you'll need all this information later.

Next, you'll need to make a boot disk and a root-install disk to boot Linux on your PC. Since we want to install via NFS, it's rather important that we choose a boot disk with networking drivers. The boot disks can be found in the **bootdsks.144** or **bootdsks.12** directory in the slackware distribution. Choose the directory according to the size floppy disk drive you have: the “144” directories are for 3.5" floppies, and the “12” directories are for 5.25" floppies. If you are using IDE disk drives, use the **net** boot disk. If you are using SCSI disk drives, use the **scsinet** boot disk. Root-install disks can be found in the **rootdsks** directory. The **color** disk uses a full-screen color install program and is very nice. This is the one I used when I wrote this article. Alternatively, you can use the **tty** disk. It uses text based install scripts. Installation using the **umsds** disk should also be possible, in the unlikely event you need to keep DOS around.

All boot and root disks are described in more detail in **README** files in each of the disk directories, if you'd like to know more about them. Once you've decided on your boot and root-install disk, you need to format two floppies and copy one image onto each of them. Please read the **INSTALL.TXT** file, section 3.4.2.1, in the top-level slackware directory, for exact instructions on how to do this. In fact, you should probably print this file out, since you'll need it to help you repartition your hard drive later on. While you're making floppies, go ahead and format a spare one. This will be needed later on to make a bootable floppy for emergencies.

Grab the three floppies and go over to your PC. Make sure that you are actually plugged into a network. Then, insert the boot floppy and power up your machine. You should see the letters **LILO** appear on your screen. This means that the **LI**nux **LO**ader is loading Linux. After some whirring, you will be prompted for extra parameters to boot your system with at the **boot:** prompt. You probably don't need to type anything here since you are installing a new system. Just hit <return>. You'll see Linux announce that it's loading the ramdisk, and various device drivers will initialize. This may take a couple of minutes. When the system asks you to switch disks, remove your boot floppy, and insert your root-install floppy. Don't forget to hit <return>. Linux will then load a root filesystem into memory. When the system has finished loading, you'll see a message containing important information (which you should read), and then a **slackware login:** prompt. At this point, you can login as the superuser root; no password is needed.

Once the system is running, the first thing to do is repartition your hard drive. Using the **fdisk** program, create some Linux partitions and a swap partition. See section 4.1 in **INSTALL.TXT** which you printed out earlier, if you need help doing this. Write down the partitions you created! You'll need to know the partition names later on. If you have less than 4MB of RAM, you'll need to activate your swap partition now. Follow the instructions in **/etc/issue** if you don't know how to do this. If you have more than 4MB of RAM, you can activate your swap partition from within the setup program.

Now, run the **setup** program. Unless you have already activated your swap partition, choose the ***A*DDSWAP** option from the menu. setup will detect your swap partition, and ask you if you want to activate it. Answer “yes” and also answer “yes” when asked if you want to use **mkswap** and if you want to activate the swap partition. Then, setup will setup your swap partition for you.

We'll continue the installation by creating our ***T*ARGET** partitions. You can continue on to this step from the **ADDSWAP** option, or choose it from the top-level install menu. Again, setup will identify appropriate partitions for you to use as Linux filesystems. Looking at your list of partitions you created, figure

out which partition you are going to use for your root Linux partition. Enter this partition at the prompt. You'll be asked to choose a type of Linux filesystem to use. Most people use the **ext2** (Linux Second Extended Filesystem). If you are a beginning Linuxer, this is a good choice for you. Having decided which filesystem you want to use, you need to format the partition. Choose the **Check** option from the menu to do this. If you have more than one partition setup for Linux, repeat the process for all your partitions. When you're done, type a **q** to finish the **TARGET** section.

So far so good. You can now continue to the **SOURCE** option, or choose it from the top-level menu if you're not going in sequence. Select option 3 “Install via NFS”. The warning message you'll see next sounds intimidating, but don't worry—it's not that bad. Gather your courage and your list of networking information and continue.

At the prompt, enter the IP address assigned for your PC. Next, enter your netmask. If you have a gateway (i.e., the NFS server is not on the same subnet as your PC), enter it; if not, answer “no”. Enter the IP address of the machine that has the Slackware distribution. Last, enter the pathname to the distribution. The setup program will put you into text mode and try to configure the networking. You should see something similar to this:

```
Configuring ethernet card...
Configuring our gateway...
Mounting NFS...
Current mount table:
/dev/fd0 on / type minix (rw)
none on /proc type proc (rw)
/dev/hda1 on /mnt type ext2 (rw)
111.112.113.114:/slackware on /var/adm/mount type nfs
(rw,addr=111.112.113.114)
```

If you don't see an error occur in the network setup, you can continue. If there is an error, go back and try again. Make sure you enter the correct network information.

That was the hard part. Now, continue on to the **DISK SETS** section, or choose it from the top-level menu. Section 3.1 of **INSTALL.TXT** has good instructions on how to choose disk sets. Aren't you glad you printed it out? After selecting which software you want to install, continue on to the **INSTALL** section, or choose it from the top-level menu. Go through the installation process—this is lengthy, but not nearly as lengthy as installing using floppies. We know this from experience!

Linux is now installed on your PC. Take a break and rejoice. You're nearly done. The only thing left to do, before you go tearing into your new system, is **CONFIGURE** a few more things on your Linux system. Continue on, or choose this option from the top-level menu. The first thing to do, is make a backup

bootable floppy disk. Insert the other disk you formatted earlier and hit “OK”. The next step in the configuration makes links for your modem and mouse devices. I do not recommend making the modem link, because it can mess up serial communications programs later on. But if using **/dev/modem** is easier for you, then let **setup** make the link. The same goes for the mouse link. If you have a CD-ROM, you can configure the CD-ROM device at the next prompt. The last installation step is to configure LILO, the Linux Loader. Choose the appropriate options for the type of operating systems you have on your hard disk. LILO can load several other operating systems (term used loosely here) in addition to Linux. The LILO configuration depends on your setup, but the menus are very self-explanatory.

That's it. You're really done now. Exit setup. You will be dropped back to the **#** prompt. All you have to do now, is eject the floppy disk from your disk drive and type reboot. If you did not install LILO, put your boot floppy in the floppy drive. If you're not an experienced Unix hacker, a good thing to read next is Installation and Getting Started by Matt Welsh. It can be found on **sunsite.unc.edu:/pub/Linux/docs/LDP** and in many bookstores around the country.

Happy Linuxing.

Greg Hankins (greg.hankins@cc.gatech.edu) is an aspiring young sysadmin at Georgia Tech's College of Computing. In his spare time, he also maintains the Linux Serial-HOWTO. He's been on the Linux bandwagon for nearly two years.

Archive Index Issue Table of Contents

Advanced search

Questions

Kim Johnson

Issue #11, March 1995

Kim provides insightful answers to some of the most frequently asked questions.

Linux Journal had double-duty in Washington, D.C. this December at Open Systems World—in addition to running a two-day "Linux Conference", we also had a booth in the exhibit hall where we handed out magazines, sold books and distributions, and answered questions about Linux. This created a problem on Thursday, when both the Linux Conference and the booth were in operation. If all the gurus were speaking and sitting on panels, not to mention listening, who would answer questions about Linux at the *LJ* booth? Although the people who knew about Linux did stop by to help when they could, for most of the day the booth was staffed by Linda Lacy and myself, both relative Linux novices. I use Linux at home, but don't play around with it, and Linda knows people who know Linux and is fairly computer literate, but we are both mainly Linuxers by association. As it turned out, that was enough.

Here are some of the questions most often asked at the booth. They are fairly basic. Some of them I knew the answers to simply by having been around Linux people, others could be answered by reading *LJ*, and still others have answers in the wealth of documentation written for Linux, if you just know where to look! I think these questions shed light on what the outside community has learned about Linux and what it still needs to learn.

"What is, um, Li. . ."

This is my favorite question to answer—"It's pronounced LIH-NUCKS. There is a good explanation on page 4 of this *Linux Journal*...." If the questioner smiles and nods as though he needs more explanation, I go on to explain that it is a free Unix-like operating system that can be run on the 386, 486, and Pentium chips. (At this point, the questioner is usually not interested in the ports of

Linux to the Alpha, PowerPC, or other platforms.) If the questioner is interested, he might ask the next question:

"What is it useful for?"

The answer to this question depends on whether the person asking knows much about Unix. If he is totally satisfied with Windows or DOS and the software for those platforms, feels no need to multitask, and never wants to network his computers or connect to the Internet, then it is possible that Linux would not be useful for him. On the other hand, if the person is interested in one or all of these things, and is interested in working a little to learn how to use his computer better, Linux could be for him.

In addition, Linux is useful because the source code is "free", a quality not appreciated by everyone, but interesting to the people at the conference who had just stopped at the Free Software Foundation booth next door to ours. booth.

"Can I run Linux on my home computer?"

Yes, and many people (including myself) do. It really does look just like other Unix systems, and it really does run on as little as a 386-SX16 with 8 megs (or less) of RAM; I've seen it done. On larger computers it runs as fast or faster than the workstations one would see in business.

"Can I run DOS and Linux at the same time?"

In other words, "My wife and kids use Windows for word processing and games, but I'd like to be able to do some of my Unix work at home." This is made very simple for the DOS users by LILO, the LInux boot LOader, so that when you turn on the computer, if you press shift or some other key of your choice after the "beep" sound, you can go right into DOS and use whatever you need. Plus, you may be able to convince the other members of your family to use Linux—they may even like it. A "DOS emulator" (called "dosemu") is available that runs most versions of DOS and most DOS programs from within Linux, as well.

"Will I have to repartition my hard drive?"

I always answered this question "yes" and encouraged people to page through Matt Welsh's Installation and Getting Started to get some idea of what actually installing Linux involves. However, there is a way of installing Linux "over" a DOS filesystem without repartitioning, but for maximum performance, repartitioning is recommended. All the distributions come with a program called "FIPS", which can split a DOS partition into a DOS partition and a Linux partition without destroying the data on the DOS partition.

"What is a Linux distribution, and what is the best distribution?"

A Linux distribution is the Linux kernel, together with essential utilities and whatever the person or people putting it together thought would be useful. The main difference between distributions is in installation and in what comes along for the ride. There is no best distribution in itself—everything depends on what the person installing it is trying to do.

"Are you affiliated with the Linux Company?"

Well.... the fact that there is no "Linux Company" really disturbs some people and amazes others. This question shows how many people misunderstand the Linux development process and how people expect all products (except for maybe shareware) to be developed inside companies and then sold. The people who make up the Linux Community (as close to an all-inclusive organization as Linux has come) showcase the power of the Internet to bring people together to produce something useful and are the antithesis of the crackers who use the Internet for vandalism and destruction.

"Are there any applications for Linux?"

The answer to this is "sort of". Most free software for Unix is available for Linux. In addition, many software companies are selling their products for use under Linux—for example, the advertisers in this magazine. Finally, Linux has SCO, SVR3, and SVR4 emulation, so it is possible to run SCO, SVR3, and SVR4 binaries under Linux.

"Is it really stable enough to use in a business situation?"

See the articles "The Roger Maris Cancer Center—Depending on Linux" (Issue 5, Sept. 1994), "Linux in Antarctica" (Issue 7, Nov. 1994), and "Virginia Power—Linux Hard at Work" (Issue 9, Jan. 1995) for real life examples of how people are using Linux. (These articles—and more—are available in The Linux Sampler published by SSC.) This is not to say that they grab the newest patches off the net as soon as they arrive and install them willy-nilly on their systems without testing, but that with ordinary caution (all systems are breakable, no matter what the operating system) Linux is viable.

"How many people actually use Linux?"

No one really knows, since no one is required to register their copies. However, the CD distributors are shipping approximately 30,000-40,000 copies a month, which does not include the people who download Linux from the Internet or who borrow their friend's distribution. Some have estimated that around a million people currently use Linux; whatever the number is, it is growing every day.

And finally,

“My (computerese) won't (more technical terms). What's wrong?”

Answer: “Um, I think that I see one of the speakers from the Linux conference coming this way. I'm sure he'll be able to answer your question.”

Kim Johnson is a graduate student in mathematics at the University of North Carolina at Chapel Hill. She spends her spare time keeping her husband from spending more money than they have on excess computer equipment.

Archive Index Issue Table of Contents

Advanced search

BRU—Backup & Restore Utility

Jon Freivald

Issue #11, March 1995

A commercial product for Linux survives trial by fire.

In many ways, I am a “typical” user. Backing up is a pain. Necessary, but still a pain. I'm also used to getting burned by bad tapes and utilities that just don't seem to be very robust (such as tar, or a few other commercial items that will remain unnamed).

I saw the ad for BRU on page 39 of the January `95 *Linux Journal*. I had seen it in other magazines, and heard it once highly recommended by a former business associate. With this as a background, and having a spare $97, I decided it was worth a try. Besides, they offer a 60 day risk-free guarantee. I faxed them an order, not realizing how soon I was going to need BRU.

Ted Cook called me the next day (my fax went out late in the evening). He asked what my kernel version was, if I was running Slackware and had **pkgtool**, if I wanted the pkgtool version or the tar version, and what disk size I needed. I opted for the **pkgtool** version on 3-1/2" disks. BRU, along with a nifty mug (which you can keep even if you decide not to keep BRU) arrived 2 days later.

The package comes on a single 1.44MB floppy with a nicely done spiral-bound manual, plus an addendum sheet outlining the install process for Linux. Installation using pkgtool is quick and painless.

Once installed, you must edit /etc/brutab to define your backup devices to BRU. The file is well commented, and the process is outlined in detail in the manual. I did this, defining my Tandberg 3600 drive. There is also a file /etc/bruxpat that contains patterns of files to be excluded from backups, such as /tmp/*, /proc/*, etc., as well as what files should not be compressed if you are using BRU's built in compression, such as .Z or .gz files. The use of this file is optional.

Here in /etc/brutab I found what I consider to be a flaw with the way BRU is shipped. There is an entry for **OVERWRITE PROTECT**, which is turned on, but it relies on the value of **RECYCLEDAYS**, which is set to zero, effectively disabling the protection afforded. As I will relate, this turned into a painful “gotcha” for me. Having plenty of tapes and a fairly regular backup schedule, I set **RECYCLEDAYS** to 7. There are many other options that can be set in /etc/brutab, most of which can be left alone, or omitted for default values.

I suppose the best way to test a backup product is to backup a system, and then wipe it clean. This is not what I intended to do, but it is effectively what I ended up doing. I ran a backup using BRU the day I received it. Three days later, I ran another backup, went to work, and came home to a failed hard drive. Ugh! Thanks to a good friend, I was able to get a loaner drive the same evening. I booted with my Slackware disks (1.1.2—old, I know, but that's what I'm running...), partitioned and formatted the new drive, and installed only the required packages from the A disk set. I then installed BRU, edited /etc/brutab to define my tape drive, loaded up my tape, and started the restore—or so I thought. What actually happened is that my fingers got dyslexic on me, and instead of telling BRU to extract from the tape, I told it to backup to the tape... This is where the default setting of **RECYCLEDAYS=0** got me. Had it been anything else, or had I remembered to change it back to 7, I would not have overwritten my latest backup tape. (This should no longer be an issue, since EST, Inc. has changed the installation script to update these variables automatically during the installation, as it automatically creates /etc/brutab according to the installer's preferences.)

After thoroughly cussing myself out, kicking the wall, and muttering into thin air for a while, I changed

RECYCLEDAYS to 7, write protected the first tape I had made three days prior, and did the restore. Once complete, I did a reboot, and the system came up perfectly.

I then decided to test BRU's claims of reliability. Sitting back in the corner I have this tape with BAD written all over it. It first failed during a server backup at work (a whole different story about commercial software that doesn't work), so I brought it home, where it worked for a while. Very soon, this tape started always giving me errors. It would almost always appear to write properly, but would fail very shortly into any read operation with media I/O errors. I popped this tape into the drive and changed to /usr/bin and did a backup (BRU stores absolute pathnames only if you explicitly tell it to, otherwise stores everything relative to ./).

BRU complained.

BRU complained again during the “AUTOSCAN” pass.

I created a junk directory, changed to there, and did a restore.

BRU complained.

BRU warned me about my junk media.

BRU restored every single file on the tape.

I don't recommend using bad media for backups, but BRU did prove to me that it really does have the “GUTS” it talks about in the advertisements.

Since then, I've installed an Exabyte 8200 8mm tape drive, and do almost all of my backups there. With “out of the box” tuning as far as buffers go, I get about 240Kbs throughput writing to the tape. The AUTOSCAN feature is very nice, because it will warn you about media errors before you put your tape on the shelf thinking your data is secure. BRU also includes scripts for doing full and incremental (with up to 9 levels) backups. There are no menus—everything is driven from the command line. Hey—I'm not running Windoze here... My backup regime now consists of

```
cd /;bru -cvvvXf /dev/rmt1
```

Twenty minutes or so later, I come back and check, confident that AUTOSCAN will warn me of any problems encountered.

BRU has many, many options, most of which I have not even begun to look at. I like it. It's reliable. It fills a definite need. If you'd like more information, call Ted Cook at Enhanced Software Technologies, Inc., (800) 998-8649 or (602) 820-0042. Tell him I sent you.

About system: 80486DX/33, 20MB RAM, 1.2GB SCSI Disk, Tandberg 3600 and external Exabyte 8200 tape drives, and Adaptec 1542B SCSI Host adapter. Linux: Slackware 1.1.2 (highly modified) with kernel 1.1.45

Jon Freivald (jaf@jaflrn.liii.com) is a Small Computer System Specialist for the US Marine Corps, currently stationed in Garden City, New York. He manages a Wide Area Network running Banyan VINES covering the NorthEastern eight states. He has been running Linux at home for over two years.

Archive Index Issue Table of Contents

Advanced search

Tcl and the Tk Toolkit

Phil Hughes

Issue #11, March 1995

In this book, Osterhout offers an introduction and overview of Tcl and Tk to get you started and continues on to the serious stuff.

- **Author**: John K. Osterhout
- **Publisher**: Addison-Wesley
- **ISBN**: 0-201-3337-X
- **Price**: $36.75
- **Reviewer**: Phil Hughes

Tcl and Tk are receiving a lot of press. They offer a quick way to develop X applications or add a graphical front-end to existing code (see *Linux Journal* #8). Plus John Osterhout, the author of Tcl, is now working for Sun—and Sun is talking about turning Tcl into a serious product. In this book, Osterhout offers an introduction and overview of Tcl and Tk to get you started and continues on to the serious stuff.

The first part of the book following the introduction covers the Tcl language itself—chapters include syntax, variables, expressions, lists, flow of control, procedures, string manipulation, file access, processes and error handling—and ends with chapters on tcl internals and the history mechanism.

Next, the book covers the Tk toolkit for writing X-Windows applications. Again, there are a lot of chapters covering widgets, geometry managers, bindings, focus, window managers, communications between applications and invocation of procedures by real-world events. The final chapters discuss configuration, assorted commands and concepts that don't fit elsewhere and also give two applications examples.

The end of the book covers working with Tcl and C, and working with Tk and C. The Tcl chapters first tell you why you may want to combine Tcl with C and then

explain how the interface works and how to develop code using it. There are a liberal number of examples. The Tk and C chapters thoroughly cover the nitty-gritty of the Tk/X interface. If you are not familiar with X-Windows programming, this looks a little scary, but it isn't the fault of Tcl/Tk.

The book ends with an appendix that points you at ftp sites where you can get software and a brief description of how to get it running. There are 450 pages of information contained in the book.

I am not a Tcl/Tk programmer and got the book to start from scratch. When I first looked at the book it reminded me a lot of The C Programming Language written by Kernighan and Ritchie (K&R), who also wrote the C language. Osterhout's book, like K&R's, begins with an overview and then continues by showing the reader all the details. A “hello world” program is used early on as a programming example and this illustrates that both Tcl and C are relatively simple languages. Also of note, is that Brian Kernighan is Addison-Wesley's consulting editor for a series of books.

I remember reading K&R, writing some C code, being a little frustrated, going back, learning more and eventually learning C. When I was learning C (1980) K&R was the only book out there. Today, even though there are hundreds of books on C, I still use K&R for reference and don't look in other C texts for information.

I have played a little with Tcl/Tk using this book and have felt more than a little frustrated. After looking over the book again, my conclusion is that it covers the language in detail as a tutorial, but it is not a good reference book. In the case of C, I use SSC's C Library Reference (a pocket-sized guide that I wrote) as a quick reference. Then I use K&R for additional reference, if needed, because it offers information in an easy-to-find format.

Osterhout's book isn't going to be the book you use when you are writing code and need to look something up. I see it as one of the tools you utilize to get started with Tcl/Tk, but a second, smaller book or pocket guide is necessary to help you with the nuts and bolts as you actually do development. Is it worth buying? Yes. But life will be easier when a reference exists that will complement this book.

Phil Hughes is the publisher of *Linux Journal*.

Archive Index Issue Table of Contents

Advanced search

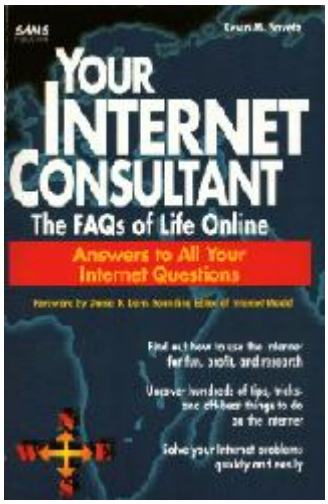
Your Internet Consultant

Phil Hughes

Issue #11, March 1995

The book gets off to a good start by presenting the introduction in a FAQ format.

- Author: Kevin M. Savetz
- Publisher: SAMS
- ISBN: 0-672-30520-8
- Price: $25.00
- Reviewer: Phil Hughes



On the Internet there are what are called FAQs. This stands for Frequently Asked Questions, and they are posted regularly to help decrease the traffic. This book, subtitled The FAQs of Life Online, is a compilation of these FAQs put into a 550 page book.

The book gets off to a good start by presenting the introduction in a FAQ format. First question: What does FAQ mean? And the second is: Does the world need another Internet book? The answer to this second question will help you understand why Kevin wrote the book. He explains how there are a lot of new Internet books and many, if not most, try to be a 1000 page guide that

tells you everything you need to know about the Internet. Then he tells you the world doesn't need another one of those books. I have to agree if, for no other reason than, you really need to learn the basics; then start exploring, rather than reading another book.

But this book is different. It really doesn't offer anything you can't get on the Internet, but it does offer it on paper. FAQs exist because this is information people need to know. Some of it is on the mechanics of using the Internet but much of it is on how to find information and what information can be found.

Some questions are answered with a specific answer, others with the illustrated use of an Internet tool to find the answer. For example, the question, “Where can I find software for my Atari computer?” is answered with the name of two Usenet newsgroups, the name of two archive sites, a gopher site and a mail server.

Chapters include “Can I Do Business on the Internet?”, “Is There Government Information Online?”, “What Do I Need to Know About Internet Culture and Lore?”, “Where Are All the Fun and Games?”, “Is This Book Worth Buying?” and “How Can I Find and Use Software (and Other Stuff)?”. The appendices include a list of “Internet Access Providers”, “Information About the Internet” and “The Internet Offline”.

Besides the normal table of contents, there is a Question Reference and an Index. This combination makes it extremely easy to locate the questions for which you are looking.

Is This Book Worth Purchasing?

It depends on your situation. By that I mean, if you are always connected to the net and enjoy looking for everything online, you may not need this book. On the other hand, if you pay by the hour for your Internet connection or you occasionally like to find something on paper first, this book is a great resource.

For example, I read the book so I could write the review on an airplane between Washington State and Washington, DC. Clearly the $25 cover price of the book was a lot lower than the cost of plugging my laptop computer into the AirPhone, at multi-dollars per minute, in order to look up information on the Internet.

The biggest downside of the book is the fact that information continually changes on the Internet. This means that some of the information is out of date the day the book is printed. However, there is enough stable information that the book will offer lots of answers a year or two after it is printed.

Phil Hughes is the publisher of *Linux Journal*.

Archive Index Issue Table of Contents

Advanced search

Letters to the Editor

Various

Issue #11, March 1995

Readers sound off.

Hardware vs. Software

In “An introduction to block device drivers” (Issue 9, January) you write:

> “... you will not be able to mount the tape, even though it contains the same information as the disk”.

I hope you know that the half-inch tape drives have all the features allowing half-inch tapes, written in a standard way, to be mounted read-only and (if the tape quality permits) to use the tape as a full-featured random-access read/write media after formatting.

I believe the sentence I've quoted has been written to simplify things and not because of lack of knowledge.

Also, I have another question close to the topic. Why are there no raw disk devices in Linux, unlike, for example, BSD and SVR3 (I don't know for sure about SVR4)? Is it because the buffering is implemented in a more correct way so that one is able to do fsck on a block device and then mount it immediately or because of something else?Sincerely,Leo Broukhis leo@zycad.com

LJ Responds:

I was perhaps not explicit enough. My statement was not intended to say anything about the hardware capabilities. The Linux kernel does not have block device drivers for any form of tapes and, therefore, is unable to mount them as filesystems, even if the hardware supports it.

The reason that there are no raw devices is that the VFS is implemented such that an open of a block device, while it does buffer, is immediate. See

block_read() and **block_write()** and notice that they aren't used for reading and writing blocks on mounted devices, but only opened devices.

rm Your Way to Fun and Disaster!

I found the remove article interesting (in January 1995). The remove alias needs to be more general. It only handles one file: you need to do something like:

```
alias rm='mv $* ~/.rm'
```

Actually, it's better to use the **-b** (backup) option in GNU fileutils and do:

```
alias rm='mv -b $* ~/.mv'
```

and default to numbered backups. Then, if you remove foo.c in two different directories, you'll get entries like:

```
foo.c      foo.c.~1~
```

so you can recover each one, with some work.

I also find the **/bin/rm -r ~/.rm** in your .profile incredibly bad.

You often find you want to check something after you log out, so if you log in, you automatically erase your backup right away. It is much better to use cron.

Also, I haven't used it, but I want to try the MIT Athena delete/undelete system (posted to comp.source.misc, v17). It may be worth a try. You should also review this.Marty Leisner leisner@sdsp.mc.xerox.com

LJ Responds:

The MIT Athena delete/undelete system sounds interesting; we hope to review it. Mark answers another letter about that column at the end of his column for this month.

Archive Index Issue Table of Contents

Advanced search

Time Again for Reader Input

Phil Hughes

Issue #11, March 1995

We want to find out more about our current readers.

One year ago we started *Linux Journal*. Answers to questionnaires we posted on Usenet helped direct *LJ*, and your letters and e-mail have continued to help set our course.

Partly because our initial questions were posted in comp.os.linux.misc on Usenet and partly because of where Linux was a year ago, many of the comments pointed us toward monitoring Linux development and following the more technical path of Linux-related work.

Today, in addition to our 10,000 subscribers, *Linux Journal* appears on hundreds of newsstands worldwide. Many of these readers have not been involved in the on-line development effort on the Internet. Because of this, their needs are different from those who initially responded to our questions back in 1993.

We have grown up along with Linux. We see one of our big jobs as helping get the word out on Linux and helping newcomers to Linux get up to speed and involved.

We want to find out more about our current readers. We ask you to fill out the information below and either fax or mail it back to us.

Thanks for your time. Your answers will help direct *LJ* for the next year.

Fax back to +1 206-782-7191 or mail to *Linux Journal*, P.O. Box 85867, Seattle, WA 98145-1867 USA.

Questionaire

Phil Hughes is the publisher of *Linux Journal*.

Archive Index Issue Table of Contents

Advanced search

Linux in Amsterdam

Michael K. Johnson

Issue #11, March 1995

“Linux does endless loops in six seconds” and other amazing comments were overheard at the First Dutch International Symposium on Linux.

In December, when I would otherwise have been preparing for this issue of *Linux Journal*, I spent a week in Amsterdam attending the First Dutch International Symposium on Linux. Linus, Remy Card, Matt Welsh, and several others spoke on a large variety of topics. I will mention here only a sampling of the presentations that were made.

By any measure, the Symposium was a success. Originally arranged to be held in an 80-person room, the symposium was moved to a 120-person room, and the organizers were still forced to turn people away. While many of the participants were Dutch, I believe that the majority were foreign. With participants from all over Europe and North America, and at least one participant from Japan, it was a truly international symposium.

Many of the participants in the symposium had never before met in person, but knew each other well through e-mail and usenet news exchanges. It is an interesting phenomenon to need to be introduced to a person you feel like you know well.

The Dutch are precise people. Their trains run on time, their museums close on time, and their conferences run strictly according to schedule. Each talk started on time. Five minutes from the scheduled end of each speaker's allotted time slot, a kitchen timer rang. Then at the end, five minutes later, it would ring again, and the speaker was allowed (in practice if not in theory) to finish his sentence. This second ring was called **kill -9** by the conference organizers.

Matt Welsh gave the best exposition I have heard yet on how to port to Linux. Not satisfied with the standard answer that most (portable) programs written for POSIX, SVR4, SVR3, or BSD just compile (given the right compiler flags for

BSD programs), he explained what kinds of non-portable code can and will cause problems and suggested ways for the listener to either get around the problem or let the listener know that the problem would take significant effort to code around.

Matt Welsh, Schilpol Airport in Amsterdam

Remy Card gave a technical introduction to the Second Extended Filesystem, showing that there is room for further development of the filesystem built into the filesystem, so that as extra capabilities are added, people do not have to reformat their hard drives. He also explained some of the features that make it less susceptible to corruption than other filesystems.

I gave a talk on how I see *Linux Journal* affecting the Linux market, both in terms of available software and available jobs, and spent some time discussing *Linux Journal* in a question and answer session that was sort of like letters to the editor, only faster.

Michael Kraehe gave a presentation on Onyx, his Copylefted 4GL which he developed (and sells support for) under Linux. It can tie together many different kinds of databases (Informix, Ingres, Postgres, Yard, gawk, Shql, and others) with one data presentation. The 4GL language is more like a shell than other 4GLs, because system commands can be used; a report to print your bills could send the output of a command through a “gawk | groff | lpr” pipeline. It uses the standard Unix tools instead of replacing them.

Peter Braam, of Oxford, gave an explanation of how Linux is being distributed all around the very distributed Oxford campus without seriously compromising security, and without requiring students to become their own systems administrators on their own Linux machines, by using Kerberos for network authorization and SUP (Software Update Protocol, from CMU) for updating packages on students' drives.

A team designing a product called Viper, which is a spin-off of Linux which will provide full kernel threads in a package conforming to POSIX 1003.4a, presented the work that they have done so far and explained what is left to do.

Linus Torvalds (the originator, main guru, and “dictator” of the Linux project, for those new to Linux) gave two talks. The first, which opened the conference, was an introduction to Linux. The second closed the conference, was much more technical, and generated more questions. Linus kindly asked to be interrupted whenever anyone had a question, and the technical talk generated more interrupts than the overview.

He gave an interesting example of how freedom to re-think some assumptions that were made when the original Unix was written allowed a simpler design in Linux. In other versions of Unix, “sleeping” creates a race condition that can cause processes to go to sleep forever, unless interrupts are carefully disabled while going to sleep. Linux uses a slightly different paradigm than most unices (sleeping is done in two steps, first saying, “I'm asleep now” and then later actually going to sleep) which removes the possibility of the race condition without disabling interrupts, improving both performance and robustness.

Linus told us about several efforts underway, including his ongoing 64-bit port to the DEC Alpha. That port is still in its infancy, but is important in helping make the entire kernel more portable, because it uses 64-bit pointers (and 64-bit integers by default), which exposes non-portable code very effectively. Work is being done now to integrate the work being done on several ports into the mainline source tree, towards the goal of making Linux a true multi-platform operating system. “I'll make a [version] 2.0 someday, and the decision when to go to 2.0 is essentially when I have a stable release which is multi-platform,” he said.

Other quotes from Linus were both interesting and (usually) informative. I have included a sampling both of non-technical and technical quotes:

“While in Australia, Dennis Ritchie said that the one feature of Unix he was most proud of was that it was portable. When it comes to Linux, the one decision I am most proud of is not the actual physical design of the system, but the decision to make it free.”

“The most important design issue...is the fact that Linux is supposed to be fun....”

“Linux is a Unix clone for PC clones. The `PC clones' part I'm trying to fix, and others are trying to fix it as well.”

“[Linux] does endless loops in six seconds.”

“I'd [really like to] see Word for Windows under Linux....”

“....So, I obviously decided to make my own Unix. What do I need? I need lots of ignorance, because if you know too much, you know that it's obviously impossible. Then you need the arrogance—`yes, I can do it, and I can do it better than anybody else'. Those two you need to just get started... you need to be a bit crazy and actually go on with it...”

“Somebody from Novell commented that they had an alpha-testing lab of a hundred machines or so, and he said, `No wonder Linux works so well. He has an alpha-testing lab of ten thousand people!'—and that's true.”

“Ext2 is much better than the Minix filesystem.”

“There is a general move [to] getting kernel threads eventually, which could be the Viper threads, or could not.”

“Within, maybe, half a year, a year, I expect to see three reasonably supported different platforms for Linux. The 386 version will be the normal one, though, simply because everybody has one.”

Speaking of an improvement that may debut with 1.3.x: “The improved data cache essentially means that the data cache right now is indexed according to the device and block number on the device... In the long run, we want to make this indexed according to the inode and the offset of the inode, which makes it much easier for the VFS layer to look up a buffer, without the need to go through the low-level filesystem to find out the block number. It also makes it possible to do NFS caching (read caching, because write caching [NFS] is a no-no).”

Linus Torvalds, reading *Linux Journal*

Michael K. Johnson is the editor of *Linux Journal*, and is also the author of the Linux Kernel Hackers' Guide (the KHG).

Archive Index Issue Table of Contents

Advanced search

The rm Command

Phil Hughes

Issue #11, March 1995

In this column we take a look at a Linux command. This month we discuss the features (and dangers) of the rm command.

The **rm** command is probably one of the first commands you learned. Here we look at some options that may save you a lot of time. Before we get into the details, some words of warning. In the Unix tradition, Linux does not ask unnecessary questions. If you tell it to remove a file or a set of files, it will do just that. If you want it to ask you for confirmation, you will need to ask it to do that.

The basic syntax of **rm** is:

```
rm [options] filenames
```

The options must start with **a -**. One or more filenames can be specified, and wildcards are permitted (because the shell, not **rm**, expands them).

If you are not familiar with wildcards (* and ? to name the most dangerous), read up on them. Placing a wildcard character in the file name list by accident can make you a very unhappy camper.

rm is the command used, in Linux terminology, to **unlink** a file. What this means is that the directory entry for the file is removed. A side effect (and the effect that we generally expect) is that the file is deleted. But this may not be the case.

The Linux file system makes it possible for a file to have more than one name or directory entry. The **ln** command allows you to create these additional names or links. If these links are *hard links*, links created with the **ln** command without the **-s** option, you have a file that can be accessed by these multiple names.

By using the **rm** command on one of these names, you only delete the name, not the actual file. When the last name pointing to the file is removed, the file is finally removed.

Now that you know about the basics, there are a bunch of options that make it possible to do more than just remove a file. A handy option for the timid is **-i**. The **-i** stands for interactive. When specified, **rm** will prompt you before it deletes each file. If you respond with **y** or **Y** the file will be deleted, otherwise the delete will be skipped.

For example, if you enter:

```
rm -i dog cat pig
```

you will be prompted with:

```
rm: remove `dog'?
```

Pressing **y** or **Y** and <return> will cause the file dog to be deleted. No matter what you pressed, **rm** will then move along to the next file, in this case **cat**, and prompt again.

Normally, if rm encounters a file that you do not have write permission to, but you do have permission to modify the file's directory, it will ask for confirmation. You then enter **y** or **Y** followed by <return> to force the removal of the file. The **-f** option overrides this default behavior. If you specify **-f**, **rm** will do the removal without the prompt. This option also eliminates the error message that **rm** generally produces if a specified file is not found.

Now, the scary option, **-r**. The **-r** stands for recursive. If you specify a directory name and the **-r** option, **rm** will remove the specified directory and all its contents, including any subdirectories contained within it (and the subdirectories' files and subdirectories and so forth). For example, if you had a directory named Joe in your current directory which contained the files name address phone and a directory Other that contained the files ssn and age, you could delete each file individually with the following command:

```
rm Joe\name Joe\address Joe\phone Joe\Other\ssn\ Joe\Other\age
```

You could then use the **rmdir** command to remove the directories Other and Joe:

```
rmdir Joe/Other Joe
```

the command:

```
rm -r Joe
```

Finally, a trick. A common problem people run into is how to delete a file whose name starts with **a -**. For example, if you entered the command

```
rm -garbagefile
```

in an attempt to remove a file named **-garbagefile**, you would get the error message:

```
rm: illegal option -g
```

Try **rm -help** for more information.

This is because rm assumes that if its first argument starts with **a -** it is an option. The solution is to use a name that does not confuse **rm**. For example, you can use either the full pathname of the file or a relative pathname where you explicitly specify the current directory using **./**. Thus, the following command would do the job:

```
rm ./-garbagefile
```

Phil Hughes is the publisher of *Linux Journal*.

Archive Index Issue Table of Contents

Advanced search

Linux in the Real World

Grant Edwards

Issue #11, March 1995

Linux has several convenient strong points when working with non-standard hardware. Freely available source code allowed Grant Edwards to compete a project much more easily than he would have been able to without Linux.

A year ago I installed Linux at home on my 386-20 and fired up X11 on my Hercules monochrome graphics adapter. I edited a source file with GNU emacs and compiled it with gcc. A real “Unix” system at home! I was so thrilled I walked around grinning for days. “This is so cool!” I exclaimed. “That's nice, but what do you do with it?” was the reply. I didn't have much of an answer then—but I do now.

This article describes a real world Linux application. The requirement was for an unattended computer to gather data from three different fluid level measurement devices and relay that data to a remote location.

The site is a coal-fired power plant about 100km away from the computer. The three measurement devices are mounted on an upright cylindrical tank with a height and diameter of approximately 7m. The tank holds water which will be mixed with ash to produce a slurry that is easier to handle than dry ash. There is 120VAC power at the tank, but no telephone line. The environment is benign, other than the constant presence of a powder-fine dust that resembles a cross between Portland cement and cake flour.

The measurement devices all use different serial protocols and physical layers. Two protocols use printable ASCII with each frame terminated by CR/LF. The first of these (ASCII Modbus) resembles Intel hex records on an RS-232 physical layer. The second is a proprietary command interface that utilizes shell-like commands over an RS-485 physical layer.

The third protocol (RTU Modbus) consists of binary data with the end-of-frame marked by a gap larger than 3 byte times. The physical layer is half-duplex FSK.

The interface to the computer is an RS-232 port connected to a proprietary FSK modem.

The hardware selected for the system is a rack-mounted, industrial 486 machine with 16M of RAM and a 500M IDE disk drive. The industrial PC chosen has several features useful for unattended operation:

- Ability to boot without a keyboard.
- Ability to boot without a video board.
- A hardware watchdog timer that can reset the computer in case of system lockup.

Since no phone line was available to provide communications between the data gathering system and the central host, a cellular phone was used.

Additional Hardware

Cellular communication—all you need is money.

For this task I purchased a pair of Microcom DeskPort 14.4K modems that support the MNP-10 “cellular” feature set. Cellular telephone connections vary much more in quality from minute to minute than do land lines, and drop-outs are more frequent and longer. For reliable cellular communications, modems need the ability to re-equalize and adjust baud rates and packet sizes accordingly. With the MNP-10 features disabled, I was unable to maintain a reliable connection even at 300 and 600 baud. With the error correction enabled, the connection was usually maintained at 9.6K or 12K baud.

UUCP was chosen over SLIP due to UUCP's ability to queue work and to automatically redial and restart a transfer after a call is dropped.

The cellular telephone is a Motorola “Bag” style 3-watt cellular which was on hand and available for use. A “cellular connection” box had to be purchased for the phone. The cellular connection is a black box, about the size of a pack of cigarettes, that plugs between the handset and the radio. It provides an RJ-11 jack, generates dial tone, responds to the modem's switch-hook transitions and converts DTMF tones to handset key presses.

Since one of the ASCII interfaces runs on the RS-485 physical layer, a board from Opto-22 was chosen that had a standard 16450 UART with opto-isolated RS-485 drivers and receivers.

System Software

The system required unattended, remote operation and simultaneous communication on four serial ports. While all of this would be possible under MS-DOS it would require a significant amount of effort, while a Unix-type OS would support all of them right “out of the box”.

Copies of Coherent and ISC SVr2 were available for use, but I chose Linux for two reasons. First, I was (and still am) running Linux at home. More importantly, Linux source code was available in case something needed to be customized or fixed.

A borrowed Fall 1993 Yggdrasil CD provided the base system, although uucp and mail didn't work as installed. I downloaded new copies of smail and Taylor UUCP, and they installed and configured with no problems. getty-ps was installed to allow the modem to be used both as a dial-out device by uucp and as a dial-in device for remote logins.

Application Software

The first job was a watchdog timer daemon. The watchdog daemon needed to do an I/O port write periodically to reset the hardware watchdog timer. In order to provide for orderly system shutdown, when the watchdog timer daemon receives a term signal, it disables the timer. Since the port address was above 0x400 the **ioperm()** and **_outb()** system calls wouldn't work.

(The kernel only maintains permission maps for ports below 0x400.) Instead, the daemon does an **open()** on /dev/port and uses the **lseek()** and **write()** system calls to do the port I/O. Since the I/O is small and infrequent, the system call overhead isn't a problem.

The next job was the software that gathers data via the three serial ports. Should it be a single, large program that talks to all three devices? This would be needlessly complex compared to writing three separate programs, each of which talks to a single device. This is especially true, since the three devices all used different protocols and provided different sets of data.

Each of the three programs gathers data (one sample every five seconds) and writes a line of text on stdout for each sample. Each line of output includes time stamp, status and data values. Each program has a command line option that specifies how long to gather data before terminating.

A simple ASCII output format with whitespace-separated columns allows easy manipulation and data reduction using familiar Unix tools, such as awk and gnuplot. A few lines of one of the data files is shown below:

```
94-01-28 18:52:41 OK 0 4.745400 998.4952
94-01-28 18:52:47 OK 0 4.745406 998.4937
```

Half Duplex Communication

It's like a cheap speaker phone—only one end can talk at a time.

The only unusual problem associated with the data gathering programs was the use of a half-duplex FSK modem. RTS must be asserted when the Linux host sending a command and then dropped to allow the device to respond. This can't be done easily from user level software, so the serial port driver was modified. Two lines of code were added to the driver so that it asserts RTS at the beginning of a transmission and drops it at the end. You don't often need source to the OS, but in this case, it saved a large amount of extra effort that would have been required to add custom hardware to control RTS.

Running the Show

Once the individual data acquisition programs were debugged, something was needed to execute the individual programs and coordinate the whole process. On Unix systems, that means a shell script: nothing complicated, just an infinite loop that does the following:

- start each of the three data acquisition programs in the background with a command line switch set to run for six hours and with stdout redirected into a file.
- wait for all three of the above to terminate.
- compress the data files and uucp them to the destination.

This shellscript is started in the background by an entry in /etc/rc.local and runs forever shipping data files off four times a day.

Glitches and Problems

It all sounds quite smooth after the fact, but the project was not without its little hiccups. The most embarrassing problem occurred while attempting to reboot the system remotely. I typed **shutdown -fh** instead of **shutdown -fr** so the system halted rather than rebooting. The system was down for a week before a trip could be made to the site to push the reset button.

The dial-in/dial-out port connected to the cellular modem would occasionally be permanently locked by **getty**. This prevented uucp from dialing out to transfer data. A crontab entry was added to periodically kill the getty on that port. There were two other instances where all communications were lost. After some experimentation, I determined that the cellular telephone had somehow been powered off.

Perusal of the user's manual and a call to the service provider revealed that the cellular phone shuts itself off if not used for eight hours. This happened twice—apparently the cellular connection doesn't always detect the modem off-hook condition, and this resulted in the cellular telephone turning itself off after eight hours of inactivity. UUCP should have retried several times before the eight hour timeout, so the exact sequence of events is still a bit of a mystery. The immediate solution was to configure a uucp crontab entry to make sure it will “phone home” once an hour even if there is not any work to be done. The eight hour timeout can be disabled and this will be done when it is convenient to take the phone in to the shop.

On a more mundane note, I managed to break my shared libraries the first time I attempted to upgrade them in order to run a newer version of the “man” utility. It was a simple task to boot using the bootdisk/CD-ROM and fix the libraries to the point where the system would again boot from the IDE drive. (When you upgrade your shared libraries, read the directions twice before you start and follow them *exactly*.)

Summary

The project was an unqualified success and similar Linux configurations are being considered for other test sites in the future.

Grant Edwards (edwar028@gold.tc.umn.edu) is an electrical engineer for a manufacturer of process control equipment. He has been messing with Unix systems while doing product design work since the early 1980's.

Archive Index Issue Table of Contents

Advanced search

New Products

LJ Staff

Issue #11, March 1995

AT&T Introduces the POSIX Korn Shell, Softfocus Releases Linux Port of BTree/ISAM V3.1 and more.

AT&T Introduces the POSIX Korn Shell

AT&T Software Solutions Division announced the availability of the latest release of David Korn's enhanced shell program for UNIX—POSIX Korn Shell (PKSH), informally called KSH-93 during the early stages of development. PKSH includes many new features and has been designed to comply with the POSIX shell standard. At the same time, PKSH has been enhanced to boost programmer productivity through improved performance and support for re-use and extensions. New features and program improvements include new commands, additional variables, support of associative as well as indexed arrays, and support for compound data objects and discipline functions.

PKSH is available for Linux for $149 for a binary license for one system. Please contact Ed Cartier, AT&T Licensing, (908)580-5719, ecartier@attmail.att.com

XESS Spreadsheet for Linux

Applied Information Systems has released its XESS 3.0 spreadsheet product for Linux. XESS is an X Windows product that provides a full complement of spreadsheet functions and graphs, plus many advanced features. It includes a platform-independent API for developing client/server applications that share data in real-time with spreadsheets. A Tcl interface is also available.

XESS is supported on Linux and most Unix and OpenVMS systems. Demonstration versions are available by ftp at ftp.uu.net in the directory /vendor/ais. AIS may be reached at 100 Europa Drive, Chapel Hill, NC 27514; 1-919-942-7801; fax 1-919-493-7563; e-mail info@ais.com.

Softfocus Releases Linux Port of BTree/ISAM V3.1

Softfocus formally announced the release of a Linux version of their popular BTree/ISAM file manager. BTree/ISAM is a complete random and sequential file management library written entirely in portable C. Available since 1982, it is currently in use worldwide on a wide range of operating systems and compilers. The distribution includes complete source code, excellent documentation, and technical support through email, phone, fax, and BBS.

“Actually, we've been actively supporting Linux for over a year now,” says Jon Simkins, president of Softfocus, “We take pride in the portability of our software to all robust C platforms and Linux certainly qualifies there. But interest in Linux as a serious development platform has increased dramatically over the last several months and we thought we'd better let the world know that BTree/ISAM v3.1 is Linux compatible.” Simkins added that the low cost and high performance of BTree/ISAM make it very popular with Linux developers.

Simkins is effusive in his praise for Linux. “I first installed Linux last year after one of my clients recommended it. I was immediately surprised by how robust and complete a system this was. Linux is now my principle development system and it's no exaggeration to say I'd be lost without it. The quality of the C development tools made the BTree/ISAM port trivial.”

Softfocus may be contacted at: Softfocus, 1343 Stanbury Rd., Oakville, Ontario CANADA L6L 2J5

For product information, e-mail info@softfocus.com or contact Softfocus at (905)825-0903 (voice) or (905)825-1025 (fax).

Archive Index Issue Table of Contents

Advanced search

How to Log Friends and Influence People

Mark Komarinski

Issue #11, March 1995

This month, Mark discusses how to keep track of what is happening to (and on) your Linux system.

The syslog daemon (known as syslogd) is used to receive messages from various parts of the Linux (or other Unix) system and distribute these messages to other locations. This allows you, the administrator, to review system activities by keeping a log of what goes on behind the scenes. It's a great way of tracking down problems and also of reviewing your system security.

For example, you can keep track of serious errors, such as TCP/IP problems or lack of memory (for when ghostscript and X get a bit out of hand). At the same time, you can debug that pesky mail problem that a user is having, or you can keep track of who has used the su program to assume root status, and you can also find out who failed trying to do so.

To get a copy of the syslog program, see the sidebar on obtaining syslog from an ftp site near you.

Syslogd has a setup file, called /etc/syslog.conf, which tells syslogd what messages to log, and what to do with the messages it receives. By default (i.e., the /etc/syslog.conf file is empty), messages are not stored anywhere. Most distributions of Linux have some basic setup of the syslog.conf file included, and the syslogd program is often started in the /etc/rc.d/rc.inet2 file.

The setup of your syslog.conf file is in three parts. Let's look at a sample syslog.conf file:

```
# /etc/syslog.conf
 *.info;*.notice    /usr/adm/messages
 *.debug           /usr/adm/debug
 *.warn            /usr/adm/syslog
```

The first line is a comment. Any lines starting with # or completely blank lines are considered comments and are ignored by syslogd. You can see two columns here, separated by at least one tab. The left side (the one with *.info;*.notice in it) defines the *facility* and the *logging level*. The right side defines the *destination* of the messages.

The facility can be defined as the “kind” of program that is running. For example, “mail” is the facility used whether you are running **sendmail** or **smail**. There are nine pre-defined facilities, plus eight facilities that are locally defined, a “wildcard” facility that means “all facilities” and one facility for issuing timestamps.

Here is what each facility is for:

Facility	Used For
user	user processes
kern	kernel messages
mail	mail
daemon	various daemons (such as ftpd)
auth	authorization (such as login)
lpr	the line printer
news	USENET news (nntp)
uucp	UUCP (Unix to Unix copy)
cron	the cron daemon
mark	a periodic message created for placing timestamps in log files
local0-local7	locally-defined facilities
*	all facilities (except mark)

One special note about the mark facility. This facility is created by syslogd every twenty minutes at the info level. This will allow you to quickly see time change in a log file, along with making sure that syslogd is running (and logging). And as noted above, the special * facility (which means all the available facilities) does not include mark.

Each facility can have a logging (or severity) level to it. This is used to indicate which types of messages you want to record. The high severity levels (such as emerg) require the most attention, as an emerg usually indicates that the program will fail soon. These severity levels decrease in importance all the way down to the debug level, which is used for properly setting up your software. Once your software is configured correctly, you can change the severity level to a higher one. Check the documentation on your software for its interpretation of the various levels. From highest severity to lowest:emergalertcriterrwarningnoticeinfodebugnone

The severity of none means that no messages from that facility are to be recorded. These levels mean that messages of that level and above are recorded. For example, say you have the following two lines:

```
mail.debug      /var/adm/syslog.mail
mail.emerg      /var/adm/syslog.mail.emerg
```

When **sendmail** (or **smail**), or whatever program is logging as the mail facility) sends syslog a message with a level of debug, it gets placed in the syslog.mail file. Any other messages, of any level from info up to emerg, also get placed in syslog.mail. Emergency messages from mail get sent to both syslog.mail and to syslog.mail.emerg. This setup makes it extremely easy to check for emergency conditions for your software, since a glance at the directory listing for syslog.mail.emerg will tell you if the file has changed recently, and you can easily type **tail/var/adm/syslog.mail.emerg** to glance at the ten most recent entries. In addition, you can find the emergency message in syslog.mail and see the other messages surrounding it to determine the events leading up to the emergency message.

In another example, we can see the use of the “none” severity level:

```
*.alert;user.none       /var/adm/syslog.alert
user.alert              /var/adm/syslog.user.alert
```

Here, all messages at alert severity and above are sent to syslog.alert, *except* for the ones in the user facility. Those get sent to the syslog.user.alert file instead, as specified by the second line.

Now you are probably wondering about the right hand side of the lines. By now, you have figured out that this can be the name of a file. But it can also be

the name of a user or of a remote host. If the intended recipient of the message is a file, the name of the file must start with a /, indicating that you have to give a full path for the filename. Note that this can be just about any file, including /dev/console, which will print the message to the console of the machine, or /dev/lp2 to print your message on paper.

If the recipient is a remote host, it will start with an @ followed by the name of the host. The intended host will receive the message via TCP/IP port 514, and from this point, it is just like a message to syslog locally. A message to the remote host must go through syslog and get logged to the appropriate location. This will allow you to monitor a network of machines from one location. You can also include a comma-separated list of users and the messages will be printed on those users' screens if they are logged in. You can also include a *, which means that every user that is logged in gets the message. This is very useful for extreme emergency conditions where you want the users to realize that something is very wrong and they should log out.

```
#Log mail errors to the mail host for
#the postmaster to deal with
mail.*          @mailhost
#Send kernel emergency warnings to all
#users so they know what's up
kern.emerg      *
```

Once you have your /etc/syslog.conf file set up, you have to first set up the files that will be storing the messages, then restart **syslogd**.

Setting your files up for syslog merely means reviewing the permissions of the log files. By default, files are set up owned by root, in the root group, readable by everyone, and writable only by the root user. For a one-user system, this should be fine. But for a multi-user system, with (possibly) logs of who sent and received e-mail, many of these files should only be read by the root user. A good suggestion for file setup is to make it read-write by root and read-only by the wheel group. This allows you to set up some security (by controlling who is in the wheel group) without having to `su' to root every time you want to check some of the files. Re-starting **syslogd** does not require killing the syslogd program. All that is needed is to send a SIGHUP (signal number 1) to the process, and it will re-read its configuration files. To help you out even more, the PID of syslogd is stored in the /etc/syslog.pid file. As root:

```
kill -HUP `cat /etc/syslog.pid`
```

will re-start the syslog daemon and put your changes into effect. Some Linux packages also include a script called /etc/syslogd.restart, which is merely a script that runs the above kill program. If for some reason you want to actually kill the syslogd program, a TERM signal (or 15) will kill the program (**kill -TERM `cat /etc/syslog.pid`**)

If you have trouble with either of these, consider using the killall program. This is a linux-specific program, but is quite useful:

```
killall -HUP syslogd
```

will work on almost any Linux box. The only possible problem is if you are trying to learn how to administer Unix systems in general, you may be spoiled by using the convenient, but Linux-specific killall command.

For more information, look at the man pages for syslogd and syslog.conf. The *Sendmail* book published by O'Reilly & Associates (often referred to as the "Bat book" because it features a picture of a bat on the cover) also devotes a few pages to syslog (aside from being a great book on configuring sendmail).

Corrections

In my January 1995 article *rm your way to fun and adventure* dealing with the problems of un-deleting, I made a few critical errors which made their way into print. Fortunately, Matt Welsh caught some of these and pointed them out to me.

First, my alias for rm was a bit off. It only copied the first file you gave it (which made it not-usable if you gave it a * or list of files as an argument). The best way to replace it is to create a shell function in the bash shell:

```
rm()
{
mv $@ ~/.rm
}
```

Note that you can use /tmp/.rm/$LOGNAME if you preferred storing files in /tmp as originally suggested:

```
mv $@ /tmp/.rm/$LOGNAME
```

/bin/tcsh uses $LOGIN for storing the username. /bin/bash uses $LOGNAME. This often creates fun for writing shell scripts. If you are using a shell other than /bin/bash, test it first!!

So, the /etc/profile should have these lines:

```
alias waste=/bin/rm
rm()
{
mv $@ ~/.rm
}
if [ ! -e ~/.rm ];
then
  mkdir ~/.rm
  chmod og-rwx ~/.rm
fi
```

if you wish to have the deleted files stored in the user's home directory or:

```
alias waste=/bin/rm
rm()
{
mv $@ /tmp/.rm/$LOGNAME
}
if [ ! -e /tmp/.rm/$LOGNAME ];
then
  mkdir /tmp/.rm/$LOGNAME
  chmod og-rwx /tmp/.rm/$LOGNAME
fi
```

if you wish to have deleted files stored under /tmp/.rm.

Using this method will create some problems for users:

1) This method will not preserve the directory structure. If a user does:

```
rm foo
cd ..
rm foo
```

only the second foo file will be available for deletion, as the original foo was overwritten by the second one.

2) Certain switches to **rm** that users will be accustomed to, such as -r, will not be available to them or may do something different.

3) Make sure that /tmp/.rm has full write permissions to it, so that users can create /tmp/.rm/$LOGNAME if it does not already exist.

4) With the second method, if you delete a directory and /tmp and the directory you are deleting are on two different filesystems, mv will complain about moving directories across filesystems. This can happen, but is less likely to occur, with the first method as well.

And as a side note, if you just type **crontab**, many versions of crontab will drop you into the editor as opposed to **crontab -l** that I had listed. Mine does not, but yours might. The dcron22 used by slackware will automatically drop you into the vi editor. The Vixie Cron used by some other installations will check the value of the EDITOR or VISUAL environmental variables if you wish to use another editor.

So what (you might ask) is being done to prevent this from happening again? Good question. I'm setting up a small mailing list which will have as members some good Linux experts, along with some not-so-experienced users so that any potential confusion or problems get ironed out before we go to print. If you would like to join this mailing list, please drop me an e-mail note at komarimf@craft.camp.clarkson.edu and ask.

Mark Komarinski (komarimf@craft.camp.clarkson.edu) graduated from Clarkson University (in very cold Potsdam, New York) with a degree in computer science and technical communication. He now lives in Troy, New York, and spends much of his free time working for the Department of Veterans Affairs where he is a programmer.

Archive Index Issue Table of Contents

Advanced search

Block Device Drivers: Optimization

Michael K. Johnson

Issue #11, March 1995

Last month, we gave an introduction to interrupt-driver block device drivers. This month, we examine common optimizations that can allow an order of magnitude improvement in performance on some hardware, and then cover some of the points that have been left out in our whirlwind tour of block device drivers.

Last month, I ran out of space for this column just as I was about to start a long discussion on optimizations. In the interest of usefulness, I will mention only the most useful optimizations here before going on to discuss initialization.

I won't have any sample code implementations of the optimizations, because these are complex issues that need to be handled in device-specific ways, and just as the code was much more vague when I discussed interrupt-driven drivers than when I introduced the basic, initial device driver, it would either be so large as to take over the entire magazine or be so vague as to be completely useless if I were to try to write some here. I hope that my explanations are useful to you even without code.

Also, I should warn you that the optimizations I talk about, while representative of common optimizations, are not necessarily representative of anything you will find in the Linux source code, except where I explicitly state otherwise. I'm writing on a slightly more theoretical level—about what can be done, rather than what has been done. Things conceptually similar to what I write about have been done, but the details are my own and may not be the best way to go about things. As usual, use this column as an introduction to the kernel source code and read the actual source code for far more insight than I can give you here.

If you are code-starved and don't care about optimizations, jump right to the second part of this article, where I talk about initialization.

Coalescence

One common optimization is coalescing adjacent requests. This means that when the driver is notified or notices that a request has been added to the request queue, it looks through the request list to see if there is a request for the next block (and possibly more blocks beyond that). If so, it sends a request to the hardware to read more than one block with one command, and when the data comes in from the hardware (presumably in an interrupt service routine), it fills both requests before calling **end_request(1)** (actually, some similar function designed especially for that driver) on either request. After satisfying (or failing to satisfy) the requests, the equivalent of **end_request()** is called for each request, but without waking up processes waiting on either request until the interrupt has been satisfied.

This will require that you write your own version of **end_request()**. Although this probably sounds daunting, it isn't as hard as it sounds, because you can use almost all of it as-is. For example, you could copy it verbatim, except instead of doing **wake_up(&wait_for_request)** at the end, you could add **wait_for_request** to a list of events to wake up when you are ready. Then you would call this new **almost_end_request()** function as soon as you have finished processing each request. When you are done handling the entire interrupt and are ready to wake up processes, iterate over the list of events, calling **wake_up()** on each in turn, from first satisfied to last satisfied.

Note that **wake_up()** will not cause a context switch directly. The driver will not give up control while running **wake_up()** to a process being woken up. Instead, **wake_up()** makes all the processes being woken up “runnable”, and sets the **need_resched** flag. This flag says that the scheduler ought to be called at the next convenient time, such as when returning from a “slow” interrupt handling routine (including the clock handling routine) or when returning from a system call. This means that the driver will not be pre-empted by calling **wake_up()**, and so it will be able to wake up all the necessary processes without being pre-empted.

This will likely take several tries to get right. All I can say to help is “Make sure you have backups. Really.”

The only driver in the Linux kernel that I have noticed doing anything like this is the floppy driver; the track buffer works in a similar way, where more than one request may be satisfied by a single read command sent to the hardware. If you are interested in investigating how it works, read drivers/block/floppy.c and search for **floppy_track_buffer** and read the entire function **make_raw_rw_request()**.

Scatter-Gather

Sounds like a “boondoggle”, doesn't it? Scatter-gather is perhaps a little bit similar in concept to coalescing adjacent requests, but is used with more intelligent hardware, and is perhaps a bit easier to implement. The “scatter” part means that when there are multiple blocks to be written all over a disk (for example), one command is sent out to initiate writing to all those different sectors, reducing the overhead involved in negotiation from O(n) to O(1), where *n* is the number of blocks or sectors to write.

Similarly, the “gather” part means that when there are multiple blocks to be read, one command is sent out to initiate reading all the blocks, and as the disk sends in each block, the corresponding request is marked as satisfied with **end_request(1)** or equivalent device-specific code. You will only be able to easily use **end_request()** unmodified with scatter-gather if each block read or written results in a separate interrupt being generated, and perhaps not even then. The SCSI driver does its own, which is probably the best way to go.

If you want to increase throughput, at the slight expense of response time, you could use timers to help: when your **request()** is notified that there is a request, and sees that there is only one request outstanding, it could set a timer to go off soon (one or two tenths of a second, perhaps), assuming that while waiting, more requests will spill in to be dealt with, and that when a certain number of requests have been made, or the timer has gone off, whichever comes first, scatter-gather will be applied to the blocks. If the **request()** routine is called and notices that “enough” (however many that is...) requests have accumulated, it would un-install the timer and process the requests. If the timer were to go off, all requests would be processed.

Note that the timer used should not be the same static timer used for the hardware timeout. Instead, it should be a dynamically allocated timer. See **<linux/timer.h>** for details on the dynamic timer routines.

I will repeat my standard disclaimer: this is simplified (at least, I'm trying to simplify it...) and if you want more detailed and correct information, study a real driver. The SCSI high-level drivers (scsi.c, sd.c, sr.c) are definitely the place to start. (I don't mention st.c and sg.c because they are character, not block, devices.)

Errors

Optimization is almost easy to write about, and sounds glamorous, but error handling is really where the rubber meets the road. I'm not even going to try to really cover error handling in this column. It is a very complex subject, and different for each piece of hardware. If you have read this far, you should have

the basic knowledge you need to start reading the block device drivers in the kernel. By reading all the error handling there, you will begin to understand how to do error handling nicely. Consider reading the drivers in this sequence: ramdisk.c, hd.c, and floppy.c.

As you write your driver, you will probably start with very simple error handling. As you use your driver, you are likely to discover more error cases to handle. Sometimes you will find that you need to redesign some components of your driver to get error handling correct. As an example, until very recently, error handling in the floppy driver was not very good. You could mount a write-protected floppy read-write and cause serious problems when you then tried to write to the floppy. Also, if you tried (for instance) to write a tar archive on a write-protected floppy, you would get a stream of errors as the driver reset the floppy drive and kept assuming the error would go away. It took a significant rewrite of the floppy driver to solve that problem correctly.

Initialization

This isn't really specific to block device drivers, but it is certainly necessary to know. In issue 9, I gave an example initialization function, but it was mostly pseudo-code and didn't cover many of the things that you need to do to work with real devices. For instance, it doesn't explain how to grab a DMA channel, nor does it explain how to grab an IRQ line.

The easiest way to deal with IRQ and DMA is to allocate both the IRQ line and DMA channel while initializing the device. It's not the best way, but it is the easiest way to start out while you are writing your device driver. When you want to figure out exactly when to allocate and free them, you can read other device drivers. The floppy device driver, for instance, has functions **floppy_grab_irq_and_dma()** and **floppy_release_irq_and_dma()** which do exactly what they say, and are used not only in the initialization code, but all through the rest of the driver.

The **floppy_grab_irq_and_dma()** function is a good place to start to learn how to allocate IRQ lines and DMA channels. According to **<asm/dma.h>**, the IRQ line should be allocated first and released last.

We'll look at IRQs first. **request_irq()** takes four arguments. The first argument to **request_irq()** is the number of the IRQ line to allocate. The second is the interrupt service routine to call when an interrupt is received. The third is a flag which is either set to **SA_INTERRUPT** or something else (presumably 0), which determines whether the argument passed to the interrupt service routine is a pointer to a register structure (0) or the number of the interrupt (**SA_INTERRUPT**), and also whether the interrupt is a “fast” interrupt handler (**SA_INTERRUPT**) or a “slow” interrupt handler (0). A “slow” interrupt handler is

one where more processing is done when the interrupt handler returns, including possibly running the scheduler to choose a new process to become the active one. A “fast” interrupt handler does as little as possible. The fourth argument is the name of the device driver.

request_irq() is simpler. It takes two arguments, the first of which is the IRQ channel, and the second of which is the name of the device driver.

The corresponding freeing functions are even simpler. **free_dma()** takes only the number of the DMA channel, and **free_irq()** takes only the number of the IRQ.

Of course, there is far more to using IRQs, and even more so to using DMA, than allocating and freeing lines and channels, but this is a start. The start, to be pedantic. Read **<asm/dma.h>**, **kernel/dma.c**, **<asm/irq.h>**, and **kernel/irq.c** for details; they are very readable, and have many useful comments.

Mail

K. D. Nguyen sent me some e-mail after reading the January issue of *Linux Journal*, echoing a wish I have heard from other people as well.

> I have been reading two books on Unix device drivers, the KHG, and the Kernel Korner articles since last issue. I feel like I can write some device drivers. But unfortunately, there seems to be something missing from all the books and articles about Unix device drivers. It is the lack of a practice environment. We, the device driver beginners, can only read and look at some device driver code under Linux and try to understand how they work. It would be more fun if there were some hardware or device kit that let us really do some exercise on what we just read about writing Unix device drivers (rather than buying a new color printer and then begging the manufacture for the specs to write a new challenging device driver). Of course, for the meantime, I will keep reading.

There is no real practice environment. The easiest way to start learning is to write a ramdisk driver. Beyond that, many real devices are actually fairly simple. Dive in! It's hard to dabble at the water's edge when you are writing kernel code for a monolithic OS, regardless of whether you are writing code for a simple ramdisk or for a toy device kit or for a real device. The learning curve is quite steep, but that means that in a short time of strenuous learning, you really pick up most of what you need to write basic device drivers.

Also, what features would a practice device support, and what would it do? It's a hard question to answer and one I'm not going to attempt—and I think that

manufacturers won't either. Since you are as likely to screw up your entire system writing a driver for a practice device as for a real one, you might as well work on a real device. There **are** real devices as simple as any toy device.

Other Resources

Michael K. Johnson is the editor of *Linux Journal*, and is also the author of the Linux Kernel Hackers' Guide (the KHG). He is using this column to develop and expand on the KHG.

Archive Index Issue Table of Contents

Advanced search