LINUX JOURNAL

Advanced search

Linux Journal Issue #2/April-May 1994

Features

Formation of the XFree86 Project, Inc.
: The developers of XFree86, a free-software package developed and distributed via the world-wide Internet, announce the formation of The XFree86 Project, Inc.

Optimizing Linux Disk Usage *by Jeff Tranter*
: This article describes some simple techniques for making the best use of the disk storage you have now.

Interview with Patrick Volkerding *by Phil Hughes*
: In this issue we interview Patrick Volkerding, the author of “Slackware”, one of the most popular Linux distributions.

News & Articles

Linux Survey Results *by Phil Hughes*
Introduction to the GNU C Library *by Michael K. Johnson*
File System Standard (FSSTD) *by Daniel Quinlan*
What's GNU *by Arnold Robbins*
Linux Around the World
RFQ&A
Linux Counter
Linux Distributions
ICMAKE Part 2 *by Frank B. Brokken and K. Kubat*
Cooking with Linux Linux Leadership *by Matt Welsh*

Columns

XFree86 Announces Formation of the XFree86 Project, Inc.

LJ Staff

Issue #2, April-May 1994

BOSTON, Massachusetts, January 24, 1994—The developers of XFree86, a free-software package developed and distributed via the world-wide Internet, announce the formation of The XFree86 Project, Inc., a not-for-profit corporation. Also announced was the filing by the new corporation for membership in X Consortium, Inc.

BOSTON, Massachusetts, January 24, 1994—The developers of XFree86, a free-software package developed and distributed via the world-wide Internet, announce the formation of The XFree86 Project, Inc., a not-for-profit corporation. Also announced was the filing by the new corporation for membership in X Consortium, Inc.

“This is an exciting day for all of us,” said David E. Wexelblat, President of the XFree86 Project, Inc. “When we started this free software project two years ago, we never imagined that it would grow to this point. Our establishing this corporation and joining X Consortium, Inc., helps give a voice to the entire free software field, in a fast-growing area of largely-commercial software development.”

XFree86 is a package of enhancements to the X Window System, Version 11, Release 5 (X11R5), for use on Intel[r]-based personal computers running Unix and Unix-like operating systems. The X Window System is a vendor-neutral, system-architecture neutral, network-transparent windowing and user interface standard developed by the X Consortium, Inc. XFree86 was initiated in April, 1992, by David Wexelblat, David Dawes, Glenn Lai and Jim Tsillas, to enhance the performance and reliability of X11R5 on the Unix-based personal computers they were using at the time.

Since that time, there have been four major releases of XFree86. The development team has grown to well over 100 developers and testers, and the user community numbers in the 10's or 100's of thousands. XFree86 is the sole implementation for several free- software operating systems (such as Linux, Free BSD, and NetBSD) currently gaining world-wide popularity. XFree86 is also shipped by several commercial operating system vendors in place of, or alongside of, a commercial implementation of The X Window System.

"...there is a choice," says Evan Leibovitch of Sound Software Ltd., "of either utilizing the X server offered with each Unix (sometimes as an option), or springing for the high-performance third-party servers for a little extra cash (except, of course, for XFree86, which is certainly ready to take its place with Kermit, TeX, gcc, Cnews, Linux and GNU Emacs as one of the most significant freeware products of all time)."

XFree86 supports over a dozen operating systems on Intel-based hardware, including SVR4, UnixWare, SVR3.2, Linux, FreeBSD, NetBSD, Mach, and OSF/1. More than 20 common SuperVGA chipsets are supported as well as 6 of the most common video accelerator chipsets including those from S3 and ATI. XFree86 is available free of charge from free software repositories around the world, via the world-wide Internet.

For more information about The XFree86 Project, Inc., or XFree86 itself, contact David E. Wexelblat, President, at AIB Software Corporation, 46030 Manekin Plaza, Suite 160, Dulles, VA 20166, 703-430-9247; Fax 703-450-4560.

Archive Index Issue Table of Contents

Advanced search

Hints & Tips for Optimizing Linux Disk Usage

Jeff Tranter

Issue #2, April-May 1994

It seems that no matter how much hard disk storage you have, it quickly becomes full. Because Linux is free, many users are on a limited budget and can't afford to simply buy more hardware. This article describes some simple techniques for making the best use of the disk storage you have now.

Half Full or Half Empty?

The first question to be answered is, how full is your disk? The best way to do this is with the “df” command. This is what my system shows:

```
% df
Filesystem  1024-blocks   Used   Available   Capacity   Mounted on
/dev/hda1   4247          73427    16108       82%              /
```

In this case I have slightly more that 16 megabytes available on my single 94 megabyte partition, making it 82% full. If you have multiple disk partitions and/or drives, you will see a entry in the listing for each mounted partition.

You may think that as long as you are under 100%, everything is fine. This is not the case, for several reasons.

First, you need some space in reserve for temporary files. Compiling, for example, requires that various intermediate files be stored in a temporary directory. If you don't have enough space for these files, the command will fail, possibly in strange ways. A normal user could even bring the system down by filling up the root disk.

Second, once your disk approaches full capacity, performance starts to degrade. The file system has to spend more time looking for free blocks on the disk to use. The amount of performance degradation depends on how full the disk is, and the type of file system. A good rule of thumb is to keep the disk below 80% full if possible, and definitely below 90%.

Load Balancing

You may have multiple disk drives, and you may have divided one or more drives into separate partitions. This is one way, for example, to avoid having the root partition filled by user data. The disadvantage is that you may find that one partition is full, even though others have space available.

If this is the case, one option is to repartition the drives to better balance the load. This will require backing up all your data first, since repartitioning is destructive.

A second choice is to move some of the files from one partition or disk to another. Using symbolic links, you may be able to do this without affecting other applications.

You likely have at least one swap partition. Choosing its size is a tradeoff between having adequate swap space and losing valuable filesystem space. If you find you have more swap space than needed (the “free” command is useful here), the n you can consider repartitioning to make the swap partition smaller. If you only rarely require a lot of swap space (e.g. during large compiles) an option is to use a smaller swap partition, and to add a swap file on those occasions when extra swap is needed.

If your drives are load balanced and disk space is still a problem, or if you have a single partition, then you are ready for the techniques described in the rest of this article.

Finding the Big Files

A useful concept is the Pareto or 80/20 principle. Applied here, it says that 80% of the disk space is contained in 20% of the files (more or less). How do you find this 20%? One tool is the “find” command. The following command line, for example, will find all of the files on the system that are larger than 1 million bytes (you may have to run this as root so you have access to all the directories; expect it to take a while to run, depending on disk size):

```
% find / -size+1000000c -ls
```

This command tells find to locate any file larger than 1 million characters and printing out its directory information. If you are not familiar with the find command, read up on i t in the man page. It can be a useful tool for identifying areas to attack.

Removing Files

The obvious solution to freeing up disk space is to delete files. With some detective work, you may find a lot of baggage on your system that you don't need. First though, two important warnings:

- Make sure you have backups of everything you are deleting, in case you find out later you really needed it.
- Have an emergency floppy that you can boot from (e.g. the SLS A1 disk) in case you delete something (like /vmlinuz?) that puts your system in an unbootable state.

On an SLS or Slackware system, it is easy to identify and remove software packages. Review the packages that you have installed, see how much disk space they require, and delete the ones you don't need.

Some of the things that can save a lot of space if they are not needed are:

- networking, including NFS and uucp
- newsreaders (tin, trn, cnews, etc...)
- language tools: p2c, f2c, lisp, smalltalk
- TeX and LaTeX
- source code

You can remove the nroff source for the man pages (in /usr/man/man?) and keep only the compressed, formatted pages (in /usr/man/cat?). Make sure all man pages have been formatted first.

The full distribution of emacs lisp source files (*.el) is quite large. You can just keep the byte compiled (*.elc) files.

If you have compiled the Linux kernel, remove the object files (e.g. cd /usr/src/linux ; make clean).

You may have accumulated some old libraries in /lib and /usr/lib. Some binaries may still require them though. You may be able to determine whether they are used by running ldd on your binaries. This is one area I suggest you avoid unless you know what you are doing, because you can easily cause your system to break if you delete needed libraries.

If you run the X window system, the font files take a lot of disk space (they are usually in /usr/lib/X11). Make sure the .pcf font files are compressed. You may be able to delete some of the more obscure fonts as well. If you compress or delete any fonts, be sure to run the mkfontdir command afterward.

I found on my system that there were a number of duplicate files in /bin and /usr/bin. Eliminating duplicates can save space, but watch that you are not just deleting links to the same file. The diff command is useful here for comparing director of files (e.g. diff /bin /usr/bin).

Certain files grow without limit and should be periodically purged. These include:

```
/etc/wtmp
/etc/ftmp
/var/log/notice
```

In general you should truncate these files to zero length (e.g., cp /dev/null /var/log/notice) rather than deleting the files, or the programs that use them may not create new files.

Finally, there may be files that you need periodically but do not need on-line all the time (e.g. the Linux kernel source). Consider backing these up to floppy or tape and restoring them on an as-needed basis.

Compressing Files

If you cannot delete files, there are several options available for making them smaller.

Compression programs can make files significantly smaller. The GNU gzip program provides excellent compression. It is ideal for compressing data files such as documents and source code that do not need to be on-line at all times. The best option to zip will provide the highest level of compression.

Emacs can also be configured to automatically uncompress files when editing; see the lisp file uncompress.el included with UGN emacs version 19. Another space savings with emacs is to compress info pages; GNU emacs version 19 automatically handles reading compressed info pages.

If you maintain multiple revisions of files, particularly source code, then a revision control system such as RCS can save space. Only the differences between file revisions are stored.

Compressing man pages was mentioned previous.ly Traditionally this is done using the standard compress program Some man programs can handle gzip compressed man page as well (you may need to continue using the .Z file extension).

Existing binary files sometimes contain debug information. They can be made smaller using the strip command. This was the case, for example, for the GNU emacs binary I uploaded from an archive site.

For new compiles, you can minimize the size of the binaries (once they are debugged) by compiling without debug and with full optimization (e.g. -O2 for gcc). The linker flags **-s** and **-N** can also be useful; see the gcc and ld man pages for more details.

Advanced Techniques

More advanced users can try some of the more sophisticated techniques described in this section.

The Linux second extended filesystem reserves a portion of the disk for use only by root. This is useful for preventing users from filling the disk, but may not be desirable in some cases (e.g. a disk containing user files only). By default 5% of the disk is reserved. You can change this when formatting a partition; see the mke2fs man page for more details.

Transparently Compressed Executables (TCX), is a utility for compressing binary files in a way that still allows them to be executed. The author is Stewart Forster. It saves disk space at the expense of a minor delay in startup time. It is most useful for large programs that are infrequently executed. I found this package to be quite effective and reliable; it can be found on your local Linux archive site.

For compressing data files, the zlibc package can be used. It works by patching the Linux system calls such as open that read data files, and uncompresses the files on the fly. I have not used this package, but the author claims that compression ratios of 1:3 can easily be achieved. The package can be found on Linux archive sites in both source and binary form; the author is Alain Knaff.

There have been discussions about implementing a compressed filesystem for Linux, similar in concept to what exists for MS-DOS. The ext2 filesystem already has some hooks to allow this to be added. This may be the ultimate solution for optimizing disk usage.

For More Information:

More information on the ideas discussed in this article can be found in the following sources. All are available on the major Linux archive sites.

Linux Kernel Hacker's Guide, Michael K. Johnson

Linux System Administrator's Guide, Lars Wirzenius.

Linux man pages

See also the documentation included with the packages mentioned in this article (zlibc, tcx, etc...).

By day, Jeff Tranter is a software designer for a large telecommunications manufacturer in Kanata, Ontario (Canada's Silicon Valley North). For him, Linux is the realization of a dream to have an affordable Unix compatible system at home that rivals commercial workstations and is just plan fun to hack around with.

Jeff Tranter by day is a software designer for a large telecommunications manufacturer in Kanata, Ontario (Canada's Silicon Valley North). For him, Linux is the realization of a dream to have an affordable Unix compatible system at home that rivals commercial workstations and is just plan fun to hack around with.

Archive Index Issue Table of Contents

Advanced search

Interview with Patrick Volkerding

Phil Hughes

Issue #2, April-May 1994

In the first issue of *Linux Journal* we interviewed Linux Torvarlds. In this issue we interview Patrick Vokerding, the author of “Slackware”, one of the most popular Linux distributions.

Linux Journal: First I would like to find out a little about you. How old are you?

Pat: I'm 27.

Linux Journal: What you do for work, school, etc?

Pat: For school, well.... I finally got my BS in computer science from Moorhead State University last Spring. On the 8 year plan, actually. I started out in computer engineering at Boston University in `85, did that for 2 years, and then took a year off before transferring into the CS program at MSU.

I just recently got a job for a San Francisco-based company making medical archiving systems controlled by Linux boxes. It's been pretty interesting work.

Linux Journal: I am looking for articles on the commercial use of Linux. I would be real interested in an article on what you are doing there.

What you do for fun?

Pat: I do all kinds of things for fun. Linux is my big fun project right now - gets pretty crazy sometimes trying to keep up with all of the development going on; for instance, last week the new C libraries, GCC, and kernel were all released within a couple days of each other. Luckily, I like keeping my machine current. Judging from the mail I get when things fall a bit behind, so does everyone else.

Hmmmmm.... other things? Well, I like to brew my own beer. That was my big hobby until Linux began to demand more of my time. It's a fun process. My

favorite part is firing up my 140K BTU burner and boiling it up for an hour or so. The aroma it releases when you throw the hops into the rolling boil is magnificent! The Homebrew Digest and rec.crafts.brewing were my big Internet hangouts before I came across Linux.

I also love music, especially the Grateful Dead. I've gone to, oh... 75 or so of their shows. The summers of `87 and `88 I followed the band all over the US in my `67 Firebird convertible. I've also got lots of portable recording equipment and tape the shows whenever I can. The Dead let their fans do this, which is pretty unique in the music world.

I play guitar, too. I'm still waiting for Jerry [Garcia] to invite me up on stage. :^)

***Linux Journal*:** Another DeadHead? I've only gone to 30 or 40 shows. My first show was in 1974 and I have around 100 tapes from shows.

Now, on to the real stuff.

When did you first start working with computers?

Pat: Way back in 1973 when I was just a kid, I went on a field trip with my class to the computer department at North Dakota State University. The room where they kept the machines totally amazed me - lots of big whirring machines with flashing lights all over the place and rows of those big drives with disk platters. One of the sysops showed me how to play Star Trek on a DecWriter teletype-style terminal. It was an instantaneous addiction.

At the time, though, there was no way I could get a home computer. I don't even think such things existed, so I started getting interested in electronics, which was more accessible. I'd build logic gates out of relays and things like that. When the first personal computers like the TRS-80, Apple][, and Atari 400/800 came out, I became a fixture in many of the stores that sold them. I couldn't afford one, but the store owners would let me hang out and use their machines. I taught myself BASIC and would write little store demo programs to ensure I'd stay welcome there.

I got an Apple][Plus with a 300 baud AppleCat modem right around 1980, when it was the hot machine. I used Apple as my only computer right up until 1990. I even had a C compiler and Unix-like operating system for it. It was nothing even resembling Linux, though.

***Linux Journal*:** When did you first start working with Linux? Pat: I first heard about Linux in late 1992 from a friend named Wes at a party in Fargo, North Dakota. I didn't download it right away, but when I needed to find a LISP

interpreter for a project at school, I remembered seeing people mention clisp ran on Linux. So, I ended up downloading one of the versions of Peter MacDonald's SLS distribution. *Linux Journal*: Describe what Slackware is?

Pat: Well, I guess I can assume we all know what Linux is. :^) Slackware consists of a basic Linux system (the kernel, shared libraries, and basic utilities), and a number of optional software packages such as the GNU C and C++ compilers, networking and mail handling software, and the X window system.

***Linux Journal*:** Why did you decide to do a distribution?

Pat: That's a good one. I never really did decide to do a distribution. What happened was that my AI professor wanted me to show him how to install Linux so that he could use it on his machine at home, and share it with some graduate students who were also doing a lot of work in LISP. So, we went into the PC lab and installed the SLS version of Linux.

Having dealt with Linux for a few weeks, I'd put together a pile of notes describing all the little things that needed to be fixed after the main installation was complete. After spending nearly as much time going through the list and reconfiguring whatever needed it as we had putting the software on the machine in the first place, my professor looked at me and said, “Is there some way we can fix the install disks so that new machines will have these fixes right away?”. That was the start of the project. I changed parts of the original SLS installation scripts, fixing some bugs and adding a feature that installed important packages like the shared libraries and the kernel image automatically.

I also edited the description files on the installation disks to make them more informative. Most importantly, I went through the software packages, fixing any problems I found. Most of the packages worked perfectly well, but some needed help. The mail, networking, and uucp software had a number of incorrect file permissions that prevented it from functioning out of the box. Some applications would coredump without any explanation—for those I'd go out looking for source code on the net. SLS only came with source code for a small amount of the distribution, but often there would be new versions out anyway, so I'd grab the source for those and port them over. When I started on the task, I think the Linux kernel was at around 0.98pl4 (someone else may remember that better than I do...), and I put together improved SLS releases for my professor through version 0.99pl9. By this time I'd gotten ahead of SLS on maybe half of the packages in the distribution, and had done some reconfiguration on most of the remaining half. I'd done some coding myself to fix long-standing problems like a finger bug that would say users had `Never

logged in' whenever they weren't online. The difference between SLS and Slackware was starting to be more than just cosmetic.

In May, or maybe as late as June of `93, I'd brought my own distribution up to the 4.4.1 C libraries and Linux kernel 0.99pl11A. This brought significant improvements to the networking and really seemed to stabilize the system. My friends at MSU thought it was great and urged me to put it up for FTP. I thought for sure SLS would be putting out a new version that included these things soon enough, so I held off for a few weeks. During this time I saw a lot of people asking on the net when there would be a release that included some of these new things, so I made a post entitled “Anyone want an SLS-like 0.99pl11A system?” I got a tremendous response to the post.

After talking with the local sysadmin at MSU, I got permission to open an anonymous FTP server on one of the machines - an old 3b2. I made an announcement and watched with horror as multitudes of FTP connections crashed the 3b2 over, and over, and over. Those who did get copies of the 1.00 Slackware release did say some nice things about it on the net. My archive space problems didn't last long, either. Some people associated with Walnut Creek CDROM (and ironically enough, members of the 386BSD core group) offered me the current archive space on ftp.cdrom.com.

***Linux Journal*:** Why did you call it Slackware?

Pat: My friend J.R. “Bob” Dobbs suggested it. ;^) Although I've seen people say that it carries negative connotations, I've grown to like the name. It's what I started calling it back when it was really just a hacked version of SLS and I had no intention of putting it up for public retrieval. When I finally did put it up for FTP, I kept the name. I think I named it “Slackware” because I didn't want people to take it all that seriously at first.

It's a big responsibility setting up software for possibly thousands of people to use (and find bugs in). Besides, I think it sounds better than “Microsoft”, don't you?

***Linux Journal*:** Some of the people out here in Seattle call them MicroSquish. :-) I admit that I initially avoided going from SLS to Slackware because I didn't take the name seriously. But the feedback I heard on the Internet pointed out why I should take it seriously. What did you expect to happen with the distribution?

Pat: I never planned for it to last as long as it has. I thought Peter MacDonald (of SLS) would take a look at what I was doing and would fix the problems with SLS. Instead, he claimed distribution rights on the Slackware install scripts since they were derived from ones included in SLS. I was allowed to keep what I had up

for FTP, but told Peter I wouldn't make other changes to Slackware until I'd written new installation scripts to replace the ones that came from SLS. I wrote the new scripts, and after putting that much work into things I wasn't going to give up. I did everything I could to make Slackware the distribution of choice, integrating new software and upgrades into the release as fast as they came out. It's a lot of work, and sometimes I wonder how long I can go on for.

***Linux Journal*:** What sort of help have you received?

Pat: Most recently, Savio Lam wrote the dialog program used to create the color installation menus on the latest release of Slackware (1.1.2). Ian Kluft put the smail package together for me. Vince Shakan's newspak collection of configuration scripts were very useful for compiling applications like elm and Taylor UUCP. Louis LaBash just contributed a kit to compile a version of perl with working IPC. Early on, Allen Gwinn sent me the lpd package. All the users that send me bug reports helped a lot, too. I especially like when they don't flame me too severely in the process.

Other than a few key packages from development teams that I trust, like GNU GCC from H.J. Lu, or the XFree86 2.0 package, I compile nearly all of the software myself. There are still a few remaining bits of SLS in the package - you can spot them by looking for files with 1992 time stamps.

***Linux Journal*:** Do you have any idea of the number of copies of Slackware that might be in use today?

Pat: Thousands, but it would be hard to estimate an exact count. I have no idea how many CDs have been pressed with Slackware on them, but I can think of four companies that have produced them. Harald T. Alvestrand, who is attempting to count Linux users, posted this estimate: I don't claim that these numbers are in any way unbiased. They are just the 240 machine descriptions that have arrived at the counter. Obviously, they give a slight bias towards net-oriented releases.

See table

Of the seven “unknown” ones, two were misspellings of slackware. There are biases in the summing too; one who listed “slackware, sls” got only the “slackware” part counted. At least, it is a datum.

(Now for some wild speculation: If this is average, and 8000 LGX CDs have been sold, and 9 of them turn up here, that would mean that I have a return rate of 0.11 percent on LGX users. A similar return rate on the others would mean

218,000 machines in the Linux installed base. My statistics professor would flunk me for extrapolation.....)

***Linux Journal*:** Does Slackware have a future?

Pat: I would like to think so. I really enjoy working with Linux, and have had a blast making a complete package like Slackware available and easy enough for beginners to install. Ian Murdock (of the Debian distribution) and I have tossed around the idea of a merger since last fall. It's possible that this could eventually happen.

At this point, I've got some nice scripts that create packages, including the installation scripts that create the symbolic links. Other than answering my mail, it's not that hard to keep Slackware up to date. If I'm going to bother to keep my own machine current, I might as well be updating the packages on the FTP site at the same time.

***Linux Journal*:** Do you want/expect any commercial benefit from Slackware?

Pat: I haven't accepted any so far. It would be nice to make money as a result of it, but not from selling the actual package. I'm not interested in going into the mail-order CD-ROM business or anything like that, but my experience with Linux has taught me a lot of valuable skills. It looks like the project has saved me from a life of COBOL. What more could I ask for than that?

***Linux Journal*:** Thanks for doing this. I think the readers are going to be real interested in this monthly interview column and I really want to get people from all walks of the Linux world interviewed.

Pat: Sure!

Patrick

Linux Journal Survey Results

Phil Hughes

Issue #2, April-May 1994

When we were first kicking around the idea of *Linux Journal*, we decided that a survey posted to the comp.os.linux news group on Usenet would be the best way to find areas of interest. In April, 1993 we posted that survey to the Internet and tabulated the answers.

The initial answers told us two things:

- *Linux Journal* was needed
- People knew what they wanted to see in *Linux Journal*

In order to make sure we were still on track, we posted the same survey to Ascent in December, 1993. The letters column comes from answers to question two of the survey, “Are there any other topics you would like to see covered?” The table below shows average responses to question one of the survey.

Survey question 1 answers:

Rate your interest in the following features:

(1=not interested, 2=possible interest, 3=interested, 4=very interested, 5=that's my subject!)

Based on the results of this survey, we have tailored the content of *LJ* and will continue to tailor it based on reader feedback.

With regard to advertising (question 3), about 90% of those who responded supported the policy. People who felt we should let anyone advertise outnumbered those who felt that having advertising was a mistake. Many of the readers felt that advertising would help them locate new products and saw that as an advantage.

Virtually everyone who returned the survey said they would like to subscribe and about 10% offered to write articles. Interest in advertising was lower (about 3%), but that was to be expected. We thank you all for your support.

When we first started getting subscribers, about 85% were from the United States. Survey results were running more like 65% from the US. We then made an offer of free copies of *LJ* for user groups to give out to their members and, the response was about 50% non-US.

Our conclusion was that our higher non-US subscription rate was the problem. So, we negotiated with international distributors and now offer the same subscription rate ($19/year) anywhere on the planet. We expect this change will get our subscriber base more in line with the survey results.

Linux Journal Survey Results

Distribution of *Linux Journal* Subscribers March 1994

Archive Index Issue Table of Contents

Advanced search

Linux Programming Hints

Michael K. Johnson

Issue #2, April-May 1994

In this column, I'll explore the GNU C Library. The Free Software Foundation (FSF) has written an excellent reference manual, available in an electronic form that can be printed or read on-line, but I think that an introduction will help some people get started.

Introduction to the GNU C Library

In this column, I'll explore the GNU C Library. The Free Software Foundation (FSF) has written an excellent reference manual, available in an electronic form that can be printed or read on-line, but I think that an introduction will help some people get started.

The GNU C Library is more than a re-implementation of the Standard C Library; while it has all the features of the Standard C Library, it has far more interesting and useful features as well. Unfortunately, it is not necessarily a good idea to use all those features in your programs.

GNU vs. Standard

One method that the FSF has used to avoid copyright infringement lawsuits from unhappy commercial vendors has been to remove restrictions and arbitrary limits from the GNU versions of programs. For example, where the standard version of a program might be limited to handling lines less than 4096 characters long, the GNU version is likely to handle lines of any length that memory can hold.

They have followed the same philosophy in their version of the C Library: why not make improvements, so long as the library is still compatible? So where most standard C libraries contain a **printf()** which causes a segmentation violation when something like **printf("%s", NULL)** is called, the GNU C library prints (**null**). This is not a feature used to print (**null**), but a debugging aid which

allows the programmer to find and correct buggy code more easily, without having to inspect core files caused by segmentation violations.

While maintaining POSIX compatibility, the FSF has significantly extended the C library, making it far more useful in the process. Unfortunately, when you use these extensions, your program becomes less portable to other platforms. To make a program generally useful, the GNU C library should be ported to any platform where your program might be useful.

On the other hand, writing good software that requires the GNU C library may encourage the further spread of the GNU C library. It may also make your programs work better, since the better the library the program is built on, the better the program may be; and some of the higher-level functions may allow you to write simpler, more maintainable code. You can spend less effort getting around library limitations. Buggy libraries can waste a lot of a programmer's time, as veteran programmers know. Since the GNU C library has a reputation as a good implementation of the Standard C library, with useful extensions, you may be doing all your fellow programmers a favor by encouraging the spread of the GNU C library.

Another reason to encourage the spread of the GNU C Library is the very fact that it is free software. It can be a tremendous help to be able to read library source when you don't understand what a library function call is doing.

The linux C library is based almost completely on the GNU C library and will probably be merged with the GNU C library eventually. This does not imply that writing programs under linux requires or encourages writing non-portable programs. The **-ansi** switch for GCC enforces fairly strict ANSI compliance(1), and by default masks references to all the GNU extensions from the header files, so that you can be sure your program is completely portable. Section 1.3.4, Feature Test Macros, in the GNU C Library Reference Manual, explains how to choose which features you want included while using the GNU C Library.

If you write programs based on books like W. Richard Stevens'—Advanced Programming in the Unix Environment, Kernighan and Ritchie's—The C Programming Language, Donald Lewine's—POSIX Programmer's Guide and other such standard references, your code should be portable to many operating systems as well as to linux. However, with linux, you have the choice of using GNU-specific library routines, and of promoting the use of the GNU C library on other platforms as well.

The Meat and Potatoes

For the rest of the column, we will leave such philosophical ramblings behind and assume that you have chosen to use the GNU C library in all its glory, above and beyond the ANSI standard and that you want an introduction to its extensions so that you know what features are there to be used. I will go through the reference manual, pointing out and briefly explaining many of the useful enhancements of the GNU C library. This is not a coherent discussion of the GNU C library, but a list of extensions that people intending to use the GNU C library for serious programming should know about. This way they can decide whether or not to use the features, rather than being condemned by ignorance to ignore them....

If you find these functions worth using, please look them up in the—*GNU C Library Reference Manual*. Don't try to use them just from my descriptions here - these descriptions are just to catch your interest. Follow the references instead.

Error Reporting

argv is often checked within **main()** to find out what name was used to invoke the program. However, for error reporting mechanisms to work, a variable pointing to **argv[0]** has to either be global within at least some part of your program or be passed around a lot from function to function and used as an argument to your error handling functions—both of which can get rather messy.

The GNU C library provides two variables, which are automatically initialized before **main()** is called, which solve this problem. **char *program_invocation_name** contains an exact copy of the name found in **argv[0]**, and **char *program_invocation_short_name** contains a copy with all the leading directory names stripped off. So if **program_invocation_name** contains **/usr/bin/foo**, **program_invocation_short_name** contains **foo**.

With these two variables, error handling functions become a lot simpler and more generic. It is possible to make clean error handling functions without these pre-provided variables, but it requires that you initialize your error handling functions, probably from **main()**, during program initialization. If you assume that the GNU C library is available, you can simply access these variables directly, cutting down on the possibility of programmer error.

Memory Allocation

The GNU C library contains built-in heap consistency checking, meaning that it can check to see if a program has violated some of the rules for accessing

dynamically allocated memory. By calling the **mcheck()** function before any memory allocation functions are called, you can ask that some consistency checks be occasionally made and an error function be called if there are any inconsistencies.

You can also define functions that are called directly before **malloc()**, **realloc()**, and **free()** are called, to check for errors. **mcheck()** is implemented by using these hooks, but it is still possible for you to use the hooks even if you are using **mcheck()** because the functions are "chained"—you just need to follow the rules and the example given in the reference manual to get this to work correctly.

An **mstats()** function is provided, which gets memory allocation statistics including:

- The total number of bytes being managed by **malloc()** (etc.), including memory that has been allocated from the operating system but not allocated to your program by **malloc()**.
- The number of bytes actually allocated to your program.
- The number of "chunks" that have been allocated from the operating system, but which are not in use.
- The number of "chunks" that are actually in use.
- The number of free bytes which have been allocated by **malloc()** from the operating system, but which are not currently allocated to your program.

A dynamic stack allocation facility called **obstacks** is available, and this can be more efficient for some things than malloc. Obstacks have some limitations, but they are implemented as macros and are very quick for small, repeated allocations. They also have a lower space overhead for each small block than **malloc()** does.

Obstacks are built on **malloc()** in much the same way that **malloc()** is built on the system call **brk()**.

A relocating allocator is also provided. This is a memory allocator which provides blocks of memory which may be moved around at any time behind the scenes, and which are therefore referenced through a "handle" which is updated whenever the memory is moved.

It can be a little more work to program with relocating memory because you have to work with, for example, a **char **** instead of a **char ***, but if your program regularly allocates and de-allocates memory in a more-or-less random way, the relocating allocator can provide significant memory savings.

Input and Output

Because there are no really good functions in the standard C library for reading lines, the GNU C library provides some extra functions which are not completely compatible but which work much better. **getline()** can safely read a string as long as memory can hold. **getdlim()** is a generalized version of **getline()**, which gets text until some delimiting character is reached again, without arbitrary limits on how long the line can be. In these functions memory is allocated from within the function, instead of the function requiring you to pass it memory. You are required to free this memory when you are done with it.

Safe formatted string I/O is provided by **snprintf()**, **asprintf()**, and **obstack_printf()** the first of these is a version of **sprintf()** which knows how long a string it has to write into; and the other two dynamically allocate whatever space they need, like **getline()** and **getdlim()**.

The GNU C library provides functions for customizing **printf()**. You can define a **%q** format for the standard **printf()**, for example, and make it do whatever you want. If you would like to be able to easily print out structures in your application, simply make **printf()** conversions for them, and pass pointers to structures into **printf()**. If **%q** is your generic structure-printing conversion, and struct foo has been designated as structure number 1, you could make it possible to write: **printf("%1q\n", &foo)**; and have the contents of foo printed out for you.

scanf() is compatibly extended so that it can optionally allocate string storage itself, so, for instance, you don't have to have a maximum string size.

It is also possible to do standard I/O on memory, using functions like **fmemopen()** to get a **FILE *** which references memory instead of a file. Now all your standard I/O functions can be used to write into memory. It is even possible to define your own types of streams, so you could, for example, write a set of procedures which allow you to use **fprintf()** to “print” to something via SYSV IPC messages.

References

The GNU C Library Reference Manual is an amazingly large and comprehensive work. While it's not perfect and is still being written, it contains a lot of information. I do not know if it is being published on paper, but it's available via ftp from all gnu mirror sites and can easily be printed or formatted for on-line reading from within emacs or the standalone info reader.

I'll take some space here to plug, as usual, some of the books that I have found most helpful, books which I think that my readers should not be without.

When you are programming for modern variants of Unix, you ought not to be without W. Richard Stevens' *Advanced Programming in the Unix Environment*, which has most of the information you need to write real applications under most variants of Unix. Both the principles and the details are covered. ISBN: 0-201-56317-7

For learning how to write POSIX compatible programs which can run on more than just Unix platforms (rather the opposite of this month's column, I'll admit), I recommend Donald Lewine's *POSIX Programmer's Guide*. It's hard to go wrong if you follow this book. ISBN: 0-937175-73-0

What is the GNU Library Public License?

Obtaining FSF Code

Help!

I'm open to suggestions on what programming hints people would like to see. Please send email to johnsonm@sunsite.unc.edu or send paper mail to Programming Tips, *Linux Journal*, P.O. Box 84867, Seattle, WA 98145-1867, and I'll consider your suggestions. If you have any books which you really like and which you would like to see me recommend in this column, please recommend them to me. I'd appreciate a detailed description of any book which you find indispensable as a Unix programmer.

1. American National Standards Institute: American National Standard X3.159-1989-“ANSI C”.

Archive Index Issue Table of Contents

Advanced search

The Linux FSSTD

Daniel Quinlan

Issue #2, April-May 1994

The recent release of the Linux FSSTND (short for “filesystem standard”) promises to bring Linux developers together into a more cohesive group.

The purpose of this standard is to document an improved and consistent filesystem (directory and file) structure for Linux systems.

The first round of discussion on creating a filesystem standard started in August 1993 on the Linux Mail-Net (based at niksula.hut.fi), and from that we were gifted with “FSSTND” (an odd attempt at an acronym), the Mail-Net channel where most of our early discussion took place. From that period came a mountain of debate, many unanswered questions, and uncertainty about how a standard could have any meaning or effect in the loose-knit Linux community.

By the summer of 1993, it was clear that the Linux community was lagging behind the rest of the UNIX world in its organization of directories and files. Not only that, but every developer in the Linux community had implemented the filesystem differently. Finding any kind of concordance between the various Linux distributions was a sobering task. The task of assembling the parts of a system from various sources that would work together was difficult.

FSSTND is based on ideas from SVR4, 4.3BSD, 4.4BSD, SunOS 4, HP-UX, and many other UNIX systems, some of which had been used by various Linux developers. However, it does not follow any single operating system layout in entirety. Instead the filesystem standard attempts to take the best of each filesystem layout and combine them into a homogeneous whole that is well suited to the needs of Linux users everywhere.

After a preliminary draft had been written, many of our uncertainties were dispelled. It turned that some Linux developers were starting to become aware of issues we were trying to solve. In September, the Debian distribution's

developers began to wonder about the same things which we had wondered about months before. We offered them our solution and they began to implement our results. Other distributions soon followed suit (although perhaps not so quickly). Today, Debian, Slackware, TAMU, Linux/PRO, LILO, Rik Faith's util-linux package, and future versions of other major distributions are all trying to follow the first version of the FSSTND.

What does this mean to you as a Linux user? Nobody expects or even wants you to recompile every binary on your system just to conform to FSSTND. The brunt of any changeover work is already being done by Linux developers. This standard should benefit users by making conforming distributions more similar where they need to be: files. Simple as it sounds, remember that UNIX systems are primarily based on files. When the entire hierarchy of files does not work together, the entire system can suffer.

If you are a developer and you want to know what the FSSTND means for you, get a copy of the current draft. The only way we can hope to gain momentum in this cooperative Linux effort is through developers. Besides the benefits I have already outlines, others include making general documentation less difficult, system administration more consistent on different systems, and the development of second and third party packages easier. If distribution A and distribution B both closely follow FSSTND and your package also conforms, it is a good bet that your application will integrate easily into a system based on either distribution.

Look for a major update to the first version sometime in March or April. Some important issues have only been resolved in the days since the public release. The draft is available through anonymous FTP at tsx-11.mit.edu in the directory/pub/linux/docs/linux-standards/fsstnd.

Daniel Quinlan (quinlan@bucknell.edu) FSSTND CoordinatorFebruary 28, 1994

Archive Index Issue Table of Contents

Advanced search

What's GNU?

Arnold Robbins

Issue #2, April-May 1994

This month's column is only peripherally related to the GNU Project, in that it describes a number of the GNU tools on you Linux system and how they might be used. What it's really about is the “Software Tools” philosophy of program development and usage.

The Software Toolbox

The Software Tools philosophy was an important and integral concept in the initial design and development of Unix (of which Linux and GNU are essentially clones). Unfortunately, in the modern day press of Internetworking and flashy GUIs, it seems to have fallen by the wayside. This is a shame, since it provides a powerful mental model for solving many kinds of problems.

Many people carry a Swiss Army knife around in their pants pockets (or purse). A Swiss Army knife is a handy tool to have: it has several knife blades, a screwdriver, tweezers, toothpick, nail file, corkscrew, and perhaps a number of other things on it. For the everyday, small miscellaneous jobs where you need a simple, general purpose tool, it's just the thing.

On the other hand, an experienced carpenter doesn't build a house using a Swiss Army knife. Instead, he has a toolbox chock full of specialized tools—a saw, a hammer, a screwdriver, a plane, and so on. And he knows exactly when and where to use each tool; you won't catch him hammering nails with the handle of his screwdriver.

The Unix developers at Bell Labs were all professional programmers and trained computer scientists. They had found that while a one-size-fits-all program might appeal to a user because there's only one program to use, in practice such programs are: a) difficult to write, b) difficult to maintain and debug, and c) difficult to extend to meet new situations.

Instead, they felt that programs should be specialized tools. In short, each program “should do one thing well.” No more and no less. Such programs are simpler to design, write, and get right—they only do one thing.

Furthermore, they found that with the right machinery for hooking programs together, that the whole was greater than the sum of the parts. By combining several special purpose programs, you could accomplish a specific task that none of the programs was designed for, and accomplish it much more quickly and easily than if you had to write a special purpose program. We will see some (classic) examples of this further on in the column. (An important additional point was that, if necessary, take a detour and build any software tools you may need first, if you don't already have something appropriate in the toolbox.)

Hopefully, you are familiar with the basics of I/O redirection in the shell, in particular the concepts of “standard input,” “standard output,” and “standard error”. Briefly, “standard input” is a data source, where data comes from. A program should not need to either know or care if the data source is a disk file, a keyboard, a magnetic tape, or even a punched card reader. Similarly, “standard output” is a data sink, where data goes to. The program should neither know nor care where this might be. Programs that only read their standard input, do something to the data, and then send it on, are called “filters”, by analogy to filters in a water pipeline.

With the Unix shell, it's very easy to set up data pipelines:

```
program_to_create_data | filter1 | .... | filterN > final.pretty.data
```

We start out by creating the raw data; each filter applies some successive transformation to the data, until by the time it comes out of the pipeline, it is in the desired form.

This is fine and good for standard input and standard output. Where does the standard error come in to play? Well, think about **filter1** in the pipeline above. What happens if it encounters an error in the data it sees? If it writes an error message to the standard output, it will just disappear down the pipeline into **filter2**'s input, and the user will probably never see it. So programs need a place where they can send error messages so that the user will notice them. This is the standard error, and it is usually connected to your console or window, even if you have redirected the standard output of your program away from your screen.

For filter programs to work together, the format of the data has to be agreed upon. The most straightforward and easiest format to use is simply lines of text. Unix data files are generally just streams of bytes, with lines delimited by the ASCII LF (Line Feed) character, conventionally called a “newline” in the Unix

literature. (This is **'\n'** if you're a C programmer.) This is the format used by all the traditional filtering programs. (Many earlier operating systems had elaborate facilities and special purpose programs for managing binary data. Unix has always shied away from such things, under the philosophy that it's easiest to simply be able to view and edit your data with a text editor.)

OK, enough introduction. Let's take a look at some of the tools, and then we'll see how to hook them together in interesting ways. In the following discussion, we will only present those command line options that interest us. As you should always do, double check your system documentation for the full story.

The first program is the who command. By itself, it generates a list of the users who are currently logged in. Although I'm writing this on a single-user system, we'll pretend that several people are logged in:

```
$ who
arnold  console Jan 22 19:57
miriam  ttyp0   Jan 23 14:19    (:0.0)
bill    ttyp1   Jan 21 09:32    (:0.0)
arnold  ttyp2   Jan 23 20:48    (:0.0)
```

Here, the **$** is the usual shell prompt, at which I typed who. There are three people logged in, and I am logged in twice. On traditional Unix systems, user names are never more than eight characters long. This little bit of trivia will be useful later. The output of who is nice, but the data is not all that particularly exciting.

The next program we'll look at is the cut command. This program cuts out columns or fields of input data. For example, we can tell it to print just the login name and full name from the **/etc/passwd** file. The **/etc/passwd** file has seven fields, separated by colons:

```
arnold:xyzzy:2076:10:Arnold D. Robbins:/home/arnold:/bin/ksh
```

To get the first and fifth fields, we would use cut like this:

```
$ cut -d: -f1,5 /etc/passwd
root:Operator
...
arnold:Arnold D. Robbins
miriam:Miriam A. Robbins
...
```

With the **-c** option, cut will cut out specific characters (i.e. columns) in the input lines. This command looks like it might be useful for data filtering.

Next we'll look at the **sort** command. This is one of the most powerful commands on a Unix-style system; one that you will often find yourself using when setting up fancy data plumbing. The sort command reads and sorts each file named on the command line. It then merges the sorted data and writes it to

standard output. It will read the standard input if no files are given on the command line (thus making it into a filter). The sort is based on the machine collating sequence (ASCII) or based on user-supplied ordering criteria.

Finally (at least for now), we'll look at the uniq program. When sorting data, you will often end up with duplicate lines, lines that are identical. In general, all you need is one instance of each line. This is where uniq comes in. The uniq program reads its standard input, which it expects to be sorted. It only prints out one copy of each duplicated line. It does have several options. Later on, we'll use the **-c** option, which prints each unique line, preceded by a count of the number of times that line occurred in the input.

Now, let's suppose this is a large BBS system with dozens of users logged in. The management wants the SysOp to write a program that will generate a sorted list of logged in users. Furthermore, even if a user is logged in multiple times, his name should only show up in the output one time.

The SysOp could sit down with his system documentation and write a C program that did this. It would take him perhaps a couple of hundred lines of code and about two hours to write it, test it, and debug it. However, knowing his software toolbox, he starts out by generating just a list of logged on users:

```
$ who | cut -c1-8
arnold
miriam
bill
arnold
```

Next, he sorts the list:

```
$ who | cut -c1-8 | sort
arnold
arnold
bill
miriam
```

Finally, he runs the sorted list through uniq, to weed out duplicates:

```
$ who | cut -c1-8 | sort | uniq
arnold
bill
miriam
```

The sort command actually has a -u option that does what uniq does. However, uniq has other uses for which one cannot substitute sort -u.

The SysOp puts this pipeline into a shell script, and makes it available for all the users on the system:

```
# cat > /usr/local/bin/listusers
who | cut -c1-8 | sort | uniq
```

```
^D
# chmod +x /usr/local/bin/listusers
```

There are four major points to note here. First, with just four programs, on one command line, the SysOp was able to save himself about two hours worth of work. Furthermore, the shell pipeline is just about as efficient as the C program would be, and it is much more efficient in terms of programmer time. People time is much more expensive than computer time, and in our modern “there's never enough time to do everything” society, saving two hours of programmer time is no mean feat.

Second, it is also important to emphasize that with the *combination* of the tools, it is possible to do a special purpose job never imagined by the authors of the individual programs.

Third, it is also valuable to build up your pipeline in stages, as we did here. This allows you to view the data at each stage in the pipeline, which helps you acquire the confidence that you are indeed using these tools correctly.

Finally, by bundling the pipeline in a shell script, other users can use your command, without having to remember the fancy plumbing you set up for them. In terms of how you run them, shell scripts and compiled programs are indistinguishable.

After the previous warm-up exercise, we'll look at two additional, more complicated pipelines. For them, we need to introduce two more tools.

The first is the tr command, which stands for “transliterate.” The tr command works on a character-by-character basis, changing characters. Normally it is used for things like mapping upper case to lower case:

```
$ echo ThIs ExAmPlE HaS MIXED case! | tr '[A-Z]' '[a-z]'
this example has mixed case!
```

There are several options of interest:

- **-c** work on the complement of the listed characters, i.e. operations apply to characters not in the given set
- **-d** delete characters in the first set from the output
- **-s** squeeze repeated characters in the output into just one character.

We will be using all three options in a moment.

The other command we'll look at is **comm**. The **comm** command takes two sorted input files as input data, and prints out the files' lines in three columns. The output columns are the data lines unique to the first file, the data lines

unique to the second file, and the data lines that are common to both. The -1, -2, and -3 command line options omit the respective columns. (This is non-intuitive and takes a little getting used to.) For example:

```
$ cat f1
11111
22222
33333
44444
$ cat f2
00000
22222
33333
55555
$ comm f1 f2
        00000
11111
                22222
                33333
44444
        55555
```

A single dash as a file name tells comm to read the standard input instead of a regular file.

Now we're ready to build a fancy pipeline. The first application is a word frequency counter. This helps an author determine if he or she is over-using certain words.

The first step is to change the case of all the letters in our input file to one case. “The” and “the” are the same word when doing counting.

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | ...
```

The next step is to get rid of punctuation. Quoted words and unquoted words should be treated identically; it's easiest to just get the punctuation out of the way.

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' | ...
```

The second **tr** command operates on the complement of the listed characters, which are all the letters, the digits, the underscore, and the blank. The \012 represents the newline character; it has to be left alone. (The ASCII TAB character should also be included for good measure in a production script.)

At this point, we have data consisting of words separated by blank space. The words only contain alphanumeric characters (and the underscore). The next step is break the data apart so that we have one word per line. This makes the counting operation much easier, as we will see shortly.

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
< tr -s '[ ]' '\012' | ...
```

This command turns blanks into newlines. The **-s** option squeezes multiple newline characters in the output into just one. This helps us avoid blank lines. (The **>** is the shell's "secondary prompt." This is what the shell prints when it notices you haven't finished typing in all of a command.)

We now have data consisting of one word per line, no punctuation, all one case. We're ready to count each word:

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | sort | uniq -c | ...
```

At this point, the data might look something like this:

```
 60 a
  2 able
  6 about
  1 above
  2 accomplish
  1 acquire
  1 actually
  2 additional
```

The output is sorted by word, not by count! What we want is the most frequently used words first. Fortunately, this is easy to accomplish. The sort command takes two more options:

- **-n** do a numeric sort, not an ASCII one
- **-r** reverse the order of the sort

The final pipeline looks like this:

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | sort | uniq -c | sort -nr
 156 the
 60 a
 58 to
 51 of
 51 and
 ...
```

Whew! That's a lot to digest. Yet, the same principles apply. With six commands, on two lines (really one long one split for convenience), we've created a program that does something interesting and useful, in much less time than we could have written a C program to do the same thing.

A minor modification to the above pipeline can give us a simple spelling checker! To determine if you've spelled a word correctly, all you have to do is look it up in a dictionary. If it is not there, then chances are that your spelling is incorrect. So, we need a dictionary. If you have the Slackware Linux distribution, you have the file **/usr/lib/ispell/ispell.words**, which is a sorted, 38,400 word dictionary.

Now, how to compare our file with the dictionary? As before, we generate a sorted list of words, one per line:

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | sort -u | ...
```

Now, all we need is a list of words that are *not* in the dictionary. Here is where the comm command comes in.

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | sort -u |
> comm -23 - /usr/lib/ispell/ispell.words
```

The -2 and -3 options eliminate lines that are only in the dictionary (the second file), and lines that are in both files. Lines only in the first file (the standard input, our stream of words), are words that are not in the dictionary. These are likely candidates for spelling errors. This pipeline was the first cut at a production spelling checker on Unix.

There are some other tools that deserve brief mention.

- **grep** search files for text that matches a regular expression
- **egrep** like grep, but with more powerful regular expressions
- **wc** count lines, words, characters
- **tee** a T-fitting for data pipes, copies data to files and to the standard output
- **sed** the stream editor, an advanced tool
- **awk** a data manipulation language, another advanced tool

The Software Tools philosophy also espoused the following bit of advice: “Let someone else do the hard part.” This means, take something that gives you most of what you need, and then massage it the rest of the way until it's in the form that you want.

To summarize:

- 1. Each program should do one thing well. No more, no less.
- 2. Combining programs with appropriate plumbing leads to results where the whole is greater than the sum of the parts. It also leads to novel uses of programs that the authors might never have intended.
- 3. Programs should never print extraneous header or trailer data, since these could get sent on down a pipeline. (A point we didn't mention earlier.)
- 4. Let someone else do the hard part.

- 5. Know your toolbox! Use each program appropriately. If you don't have an appropriate tool, build one.

As of this writing, all the programs we've discussed are in the textutils-1.9 package, available via anonymous ftp from prep.ai.mit.edu in the /pub/gnu directory, file textutils-1.9.tar.gz.

None of what I have presented in this column is new. The Software Tools philosophy was first introduced in the book *Software Tools* by Brian Kernighan and P.J. Plauger (Addison-Wesley, ISBN 0-201-03669-X). This book showed how to write and use software tools. It was written in 1976, using a preprocessor for FORTRAN named ratfor (RATional FORtran). At the time, C was not as ubiquitous as it is now; FORTRAN was. The last chapter presented a ratfor to FORTRAN processor, written in ratfor. Ratfor looks an awful lot like C; if you know C, you won't have any problem following the code.

In 1981, the book was updated and made available as *Software Tools in Pascal* (Addison-Wesley, ISBN 0-201-10342-7). Both books remain in print, and are well worth reading if you're a programmer. They certainly made a major change in how I view programming.

Initially, the programs in both books were available (on 9-track tape) from Addison-Wesley. Unfortunately, this is no longer the case, although you might be able to find copies floating around the Internet. For a number of years, there was an active Software Tools Users Group, whose members had ported the original ratfor programs to essentially every computer system with a FORTRAN compiler. The popularity of the group waned in the middle '80s as Unix began to spread beyond universities.

With the current proliferation of GNU code and other clones of Unix programs, these program now receive little attention; modern C versions are much more efficient and do more than these programs do. Nevertheless, as exposition of good programming style, and evangelism for a still-valuable philosophy, these books are unparalleled, and I recommend them highly.

Acknowledgement: I would like to express my gratitude to Brian Kernighan of Bell Labs, the original Software Toolsmith, for reviewing this column.

—*Endnotes:*

Questions and/or comments about this column can be addressed to the author via postal mail C/O *Linux Journal*, or via e-mail to arnold@gnu.ai.mit.edu.

Arnold Robbins 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

Linux Around The World....

Magnus Y Alvestad

Eugene G. Crosser

Issue #2, April-May 1994

In this column, we will include short articles and news items about Linux.

According to the Linux Counter, Norway is one of the countries with the most Linux users. While this may be biased by the fact that the Linux Counter itself is based in Norway, it's certainly true that Linux is gaining a foothold there. In the latest issue of the magazine `Mellvik Rapporten', a serious (Appx. $600 / 12 issues) report about open systems and Unix, Linux got a very favorable review. Still, there is little commercial or mainstream interest. Most people are connected and get SLS or Slackware from the ftp sites. They are mostly either Unix gurus that hack the kernel for fun, or Unix users that want Unix at home.

One user tells me that he uses Linux to run a picture manipulation package (XITE), and that it runs better on Linux than on a HP 755. Better—as in the user interface works 100% compared to 80% on the HP, not better as in faster.

At the University of Oslo, the Department of Mathematics is planning to use Linux for one of their machine rooms. This will give a lot of students Linux experience.

Magnus Y Alvestad,Oslo NORWAY

The former Soviet Union is definitely not the best computerized part of the world (although the situation changes rapidly now), but there are already 10 to 20 installations of Linux around here. Most of the systems are in personal use, and used primarily as mail and news servers. They usually have a uucp connection to Relcom—the largest xUSSR “professional” computer network. There is a linux directory on one of the Relcom ftp/mail servers in Moscow, containing fresh kernel releases and some selected packages.

Here I would like to mention the importance of free systems like Linux in the country with a long tradition of pirating software. Pirating was a policy approved by the government in the times of communists, and now, in the times of growing market economy, it is hard to withstand the tradition. The main argument of the apologists of pirating is the extremely high cost of the software, as compared to the average income of a person living in this country. Indeed, say, the Borland C compiler is almost beyond the reach of an independent programmer. And now we can tell these people: OK, if you really cannot buy MS/DOS of OS/2 or Windows, you can go for Linux. You will get an excellent system, with development tools and all-what-you-want for free, and no one will accuse you of being a thief. I believe that this fact can play its role in making our society more civilized.

Of Linux-related development, I would mention if mail—a freeware FidoNet support package that I am writing and that is currently in beta-testing. Although it works on various Unix platforms, the development is made on a Linux box.

Eugene G. Crosser,Moscow RUSSIA

WAUUG Discusses Linux

Archive Index Issue Table of Contents

Advanced search

RFQ&A

LJ Staff

Issue #2, April-May 1994

Request for Questions and Answers. Win a free subscription to *Linux Journal*.

Starting with the April issue, *Linux Journal* will have a Linux Question and Answer column. If you would like to submit a question, send it to: info@linuxjournal.com or mail or FAX it to *Linux Journal* at our Seattle office. questions@fylz.com Editorial address:

Linux Journal, P.O. Box 85867Seattle, WA 98145-1867fax: +1 206-782-7191tel: +1 206-782-7733

To be considered for use in *LJ*, questions should be Linux-specific. Questions applicable to any Linux distribution will be considered first. Questions can be related to hardware, system, or programming.

If you have an informative or interesting answer to a question that had caused you or a colleague grief, we would also be grateful if you would send it along without waiting for us to pose the question.

All entries used will qualify for a complimentary subscription to *Linux Journal* that will be awarded each month. If you are already a subscriber, you can use your prize for a friend or colleague.

Archive Index Issue Table of Contents

Advanced search

Linux Counter

LJ Staff

Issue #2, April-May 1994

What is it? The Linux Counter is an attempt to get an idea of how many Linux users there are in the world.

The experiment has been running since October 1993 and has so far registered more than 4,500 Linux users. Data from other sources indicate that this may be somewhere between 0.5% and 5% of all Linux users.

All the reports are available for anonymous FTP from aun.uninett.no, directory pub/misc/Linux-counter, and are updated every hour.

How to register

Send E-mail to Linux-counter@uninett.no with the SUBJECT line

“I use Linux at place” where place is one or more of school, work, or home. You will get back a letter with 3 things:

- An acknowledgement
- A form that you can fill out and send in with more information about yourself, your machine, and your 386Linux-using friends
- A report giving the current status of the counter

You can update your “vote” at any time, by sending an E-mail message from the same account. Duplicates will be weeded out.

Privacy issues

The counter will NOT give out any information about individual persons. The only exception is this:

If the person writes the following:

```
//PERSON Name: (my name is.....) may-publish: yes
```

this is taken to mean that his name, E-mail address and country can be published in the report called “persons”, which might be useful for people

wanting to meet other Linux users.

You can remove permission at any time, by sending in another registration with this field set to “no”.

Harald Tveit Alvestrand lives in Trondheim, Norway. He can be reached via E-mail at Harald.T.Alvestrand@uninett.no

Places Where Linux is Used

Growth in Linux World Wide

Archive Index Issue Table of Contents

Advanced search

Linux Distributions

LJ Staff

Issue #2, April-May 1994

When people first start looking at Linux there are some hurdles. The first one is understanding that Linux is free. Because so much softwre is licensed, the idea that you can get a copy and legally give it to all your friends and use it on all your computers seems to take some getting used to.

When people first start looking at Linux there are some hurdles. The first one is understanding that Linux is free. Because so much software is licensed, the idea that you can get a copy and legally give it to all your friends and use it on all your computers seems to take some getting used to.

Once people get over that hurdle, the next problem seems to be “which distribution should I get?”. This is because there isn't just one Linux—a concept that is familiar to many Unix system users but fairly foreign to the MS-DOS or MS-Windows user. This article addresses that question. Hopefully the third question, where do I get it, can be answered by our advertisers.

Historically, the word Linux had a somewhat different meaning. It was really just the operating system kernel. People would then collect various pieces of software (a compiler from one place, login code from another and so on) and put together their own system. The problem is that this took a lot of time, and as the amount of support software available grew the choices grew as well.

The solution was for someone to make a distribution, a collection of software consisting of the kernel and all the support programs that a user would need. What happened is that more than one “someone” did this because of differing needs. And each of these distributions was called “Linux”.

What's wrong with calling them Linux? Nothing, if we can avoid the confusion that it initially introduces. The confusion comes from two areas:

- Different distributions contain different programs.

- Linux is really the kernel. When we lump all the other programs in and still call the contents Linux we are ignoring the work of others including FSF for the C compiler and most of the utilities, and the University of California at Berkeley for Ingres and Postgres.

I am sure the label Linux will stick, but I do think it is important that we recognize that there's a lot more to what we get in a distribution than code written by those who would be considered Linux developers.

Now, on to distributions. This isn't intended to be a complete list, just a quick look for beginners. I encourage anyone with a different distribution that they consider important to write to us and tell us about it.

The first complete Linux distribution was SLS. Although the most recent release is sorely out of date today, I mention it for historical reasons. Many people who are using Linux started with this distribution, which was produced by Peter MacDonald. It also offered the basis for other distributions like Slackware.

MCC

Developed at the University of Manchester, MCC was designed to be quickly installed by anyone. It has excellent documentation and is both compact and very stable. MCC was designed to be installed on 386 systems in a lab used by computer science students. It includes networking via ethernet and a complete development system but lacks such things as print spooling, uucp and X windows. Because of its excellent documentation and compact size (easily fits in 30MB of hard disk space; distribution is 8 floppies) it is a great place to start-particularly if you have little or no Unix experience.

At the time of this writing (February, 1994), the MCC distribution is based on a rather old kernel. It is, however, very stable and well tested.

Slackware

Slackware evolved from the ideas behind SLS, a complete Linux system with an installation system that makes it possible to pick and choose what you want. It isn't perfect but it is very current, well supported and very reliable. The imperfections are generally in terms of a missing link or wrong permission. Here at *Linux Journal* we have two systems running Slackware, and the only serious problem we encountered was with smail. After talking to other Unix users we determined that the problem was actually an smail bug, not a Linux bug. We replaced smail with sendmail+IDA and all seems to be fine now. Some distributors (such as Trans-Ameritech) are distributing Slackware on CD-ROM.

Yggdrasil

Available only on CD-ROM, this is a very popular distribution. Current and complete, it offers a quick way to get a working system up and running. It includes X-windows (in fact, it requires you to load X to do system configuration) and a pretty amazing set of tools. Like most of the up-to-date distributions, it has a few bugs. It is, however, a great choice for someone who wants to get a working Linux system with X up and running quickly. I also feel it is well worth the $50 for the amount of source code you get in a compact form. We have one Linux system in the office running Yggdrasil.

Debian

This is a new distribution currently in beta test. I have not run this but am on the developer's list. It seems to be progressing rapidly toward a professional-quality distribution. It also appears that Debian will be adopted as the official Linux distribution of the Free Software Foundation. In structure Debian is much like Slackware, but the level of effort going into it is going to make it a very clean product.

Linux/PRO

This is a professional-quality Linux distribution. It is currently being distributed in Holland with U.S. distribution planned in the near future. Again, I have not worked with this distribution, but it is being developed by a new company, ARIS, as a commercial-quality product.

Which one should you get?

That depends on your needs and what equipment you have. If you have a CD-ROM drive, buying Linux on a CD is a good choice. A CD can hold over 600MB of files, and most of the CD distributions have hundreds of megabytes on them. The low price tag (less than $50) makes a CD an inexpensive way to get the information.

If you don't have a CD-ROM drive, but you do have Internet access, downloading the files from one of the ftp sites is an alternative.

If you don't have Internet access, try looking around on local bulletin board systems. Hundreds of them offer Linux distributions. Or contact your local Unix (or Linux) user's group. Many of them know people who will make a copy of one of the distributions for you if you supply the disks.

If all else fails, there are people who copy distributions to floppy disks and sell them. Costs are generally around $2/disk.

There is a manual called Linux Installation and Getting Started, written by Matt Welsh, that I highly recommend (see the review in *LJ* #1, page 10). This runs about 200 pages and offers answers to many of the common questions about getting your Linux system up and running. It is available for ftp access on many of the Internet sites that have Linux distributions. It's also available on paper, comb bound from SSC.

In conclusion, if you have been thinking about Linux, take the plunge. It works. It's a real operating system, useful both to help you learn about Unix-like systems and to use for real projects.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

ICMAKE Part 2

Frank B. Brokken

K. Kubat

Issue #2, April-May 1994

In part 1, Brokken and Kubat explained where the ideas for icmake came from, the basics of the program and where you can get a copy. In Part 2 we cover the grammar of icmake source files. The final part of this article will appear next month and will show examples of the use of icmake.

4. The grammar of icmake source files

Icmake source files are written according to a well-defined syntax, closely resembling the syntax of the C programming language. This is no coincidence. Since the C programming language is so central in the Unix operating system, we assumed that many people using the Unix operating system are familiar with this language. Providing a new tool which is founded on this familiar programming language relieves everybody of the burden of learning yet another dialect, thus simplifying the use of the new system and allowing its new users to concentrate on its possibilities rather than on its grammatical form.

Considering icmake's specific function, we have incorporated a lot of familiar constructs from C into icmake: most C operators were implemented in icmake, as were some of the standard C runtime functions. In this respect icmake's grammar is a subset of the C programming language. However, we have taken the liberty of defining two datatypes not normally found in C. There is a datatype `string' (yes, its variables contain strings) and a datatype `list', containing lists of strings. We believe these extensions to the C programming language are so minor that just this paragraph would probably suffice for their definition. However, they will be described in somewhat greater detail in the following sections. Also, some elements of C++ are found in icmake's grammar: some icmake-functions have been overloaded; they do different but comparable tasks depending on the types of arguments they are called with. Again, we believe this to be a minor departure from the `pure C' grammar, and think this practice is very much in line with C++'s philosophy.

4.1. Comment and preprocessor directives.

One of the tasks of the preprocessor is to strip the makefile of comment. Icmake recognizes two types of comment: standard C-like comment and end-of-line comment, which is also recognized by the Gnu C compiler and by Microsoft's C compiler.

Standard comment must be preceded by /* and must be closed by */. This type of comment may stretch over more than one line. End-of-line comment is preceded by // and ends when a new line starts.

Lines which start with #! are skipped by the preprocessor. This feature is included to allow the use of executable makefiles. Apart from the #! directive, icmake recognizes two more preprocessor directives: #include and #define. All preprocessor directives start with a `#'-character which must be located at the first column of a line in the makefile.

4.1.1 The #include directive.

The **#include** directive must obey the following syntax:

```
#include "filename"
```

or:

```
#include <filename>
```

When the preprocessor icm-pp encounters this directive, `filename' is read. The filename may include a path specification. When the filename is surrounded by double quotes, icm-pp attempts to access this file exactly as stated. When the filename is enclosed by < and >, icm-pp attempts to access this file relative to the directory pointed to by the environment variable IM. Using the **#include** directive, large icmake scripts may be modularized, or a set of standard icmake source scripts may be used to realize a particular icmake script.

4.1.2. The #define directive.

The **#define** directive is a means of incorporating constants in a makefile. The directive follows the following syntax:

```
#define identifier redefinition-of-identifier
```

The defined name (the name of the defined constant) must be an identifier according to the C programming language: the first character must be an

underscore or a character of the alphabet; subsequent characters may be underscores or alphanumerics.

The redefinition part of the *#define* directive consists of spaces, numbers, or whatever is appropriate. The preprocessor simply replaces all occurrences of the defined constant following the *#define* directive by the redefinition part. Note that redefinition's are not further expanded; an already defined name which occurs in the redefinition part is not processed but is left as-is.

Also note that icm-pp considers the redefinition part to be all characters found on a line beyond the defined constant. This would also include comment, if found on the line. Consequently, it is normally not a good idea to use comment-to-end-of-line on lines containing #define directives.

4.2. Constants, types and variables

Constants may be used in the makefile to indicate a number or a string. Int constants are denoted by numeric characters; e.g., 13 is an int constant. A second way to denote an int constant is by enclosing a character in single quotes. The numeric value of the constant is then the ascii number of the character, e.g., the constant `A' has the value 65. The character between quotes may not be `escaped', such as `\n'. Only single characters are allowed in this notation of integer constants.

String constants are denoted by text between double quote marks, e.g., “a string” is a piece of text.

Icmake recognizes four types: int, string, list and void. The types serve the following purposes:

- int: The type `int' is used to represent numerical 16-bit signed values.
- string: The type `string' is used to represent strings, like the strings used in C.
- list: The type `list' is used for variables and return values of functions consisting of lists of strings. There are no list-constants. Instead, lists always have to be built run-time.
- void: The type `void' is used only with functions, to indicate that these functions do not return values.

The types int, string and list are also used for defining variables and arguments. Icmake allows global variables and local variables. The declaration of a variable or an argument must state the type of the variable; a counter variable would be an int, while a variable containing the names of all files having extension `.c' would be a list.

Some of the built-in functions of icmake (see the section about icmake's functions) return a value of one of the types int, string or list. The returned value may be assigned to a variable of the same type or may be passed to another function.

Similarly to built-in functions, user-defined functions are assumed to return a value which is either int, string or list. The int type is the default. Functions may be defined as not returning a value. Such functions have the `void' returntype.

The definition of variables follows a C-like syntax. Arguments are defined as in ansi-C. An illustration of the use of types is found in the following listing. Note the use of the constants 55 and “main.c” (a string constant).

```
string myfun (int x, string y, list z)  // a
user-defined function
{       // of type string, having 3
      int       // parameters
          counter,      // local variables: 2 ints, i;    // 1 string
          and 1 list
      string
          name;
      list
          cfiles;
      counter = 55;     // counter is set to 55 name = "main.c";  //
      name is set to string main.c return (name);    // a string is
      returned to the
  }     // caller
```

4.3. Strings and escape sequences

Strings in makefiles are used to represent both filenames and displayed text. Icmake allows a number of special formatting sequences in strings to facilitate the display of text. These sequences are called, in analogy to the C programming language, escape sequences. Icmake recognizes the following escape sequences: Escape sequence Action

```
\a      alert (bell)
\b      backspace character
\f      formfeed character
\n      newline
\r      carriage return character
                tab
\v      vertical tab
\-other-        literal -other-, e.g., \\
```

Escape sequences in strings are identified by a backslash character \ followed by a character which identifies the escape sequence. Like C, Icmake allows string-concatenation. Long strings, extending over several lines of text, can be built by separating string constants by white-space characters (blanks, tabs, newlines).

4.4. The code of a makefile

This section discusses the user-defined functions which may appear in a makefile and also defines other syntactical constructs.

4.4.1. Flow control statements

Icmake recognizes six control statements:

- **if** statements, including **if-else**
- **while** statements
- **for** statements
- **return** statements
- **break** statements
- **exit** statements

The **exit()** statement, though a function in C, is part of the icmake language. The exit statement may be given an expression yielding an int. If an int expression follows, its value is returned as an int to the operating system. Otherwise, the returned value is undefined. The other flow control statements are analogous to the corresponding ones in the C programming language.

4.4.2. User-defined functions.

Icmake allows the construction of user-defined functions in a makefile. The definition of a function must follow an ansi-C-like syntax, however, minor differences exist between an icmake function and a C function. These differences are highlighted in this section.

The definition of a function must follow the syntax:

1. Optionally, the return type of the function is specified. The type is void, int, string or list. The default return type is int.

   When a function explicitly returns using a return statement, the returned value must match the return type. If a function does not use a returns statement, an undefined value is returned. Functions which are defined as void can also use the return statement, albeit without an expression.
2. Following the optional return type, the function name must follow. The name must be an identifier, i.e., the first character must be an underscore or a character of the alphabet, and optional following characters may be underscores or alphanumerics.
3. Following the function name, a (is expected.
4. A parameter list may follow, consisting of parameter specifications separated by , (this is referred to as an ansi-C parameter list). Parameter

specifications consist of the parameter type (int, string or list) and the parameter name (an identifier).

In contrast to C, icmake does not allow user-defined functions to have a variable number of parameters.

5. Following the optional parameter list, a) is expected.
6. Next, the code of the function is expected: statements enclosed by { and }.
7. Following the first { of the code block, local variables may be defined. The definition of local variables consists of the variable type, one or more variable names separated by commas, and a semicolon.

In contrast to C, local variables can only be defined immediately after the outer curly brace of the function code block. Variables cannot be defined within a block of statements.

In contrast to C, icmake initializes all local variables to zero.

Icmake does not allow forward references. This means that a function may be called only after it has been defined. Recursive function calls are accepted. Furthermore, the statement which calls a function must supply the exact number of required arguments and each argument type must match the parameter list of the function. The built-in functions are predefined and may therefore be used anywhere within functions.

4.4.3. The user-defined function main()

The code section of a makefile must contain at least one user-defined function, called **main()**. The execution of a makefile starts at this function. The run-time support system of icmake provides three arguments which the function **main()** may use. The arguments are used to hold the command line parameters of the icmake invocation and the environment setting.

The three arguments are most commonly referred to as argc, argv and envp. Argc is an int argument, holding the number of command line parameters. Argv is a list, holding the command line parameters themselves. Envp is a list holding the environment setting. A definition of the main() function which uses all arguments argc, argv and envp is given below:

```
int main(int argc, list argv, list envp)
        {
            // statement(s)
        }
```

Users may wish to define the **main()** function without arguments when the command line parameters need not be examined. In this case, the **main()** function can be defined as:

```
int main
{
//statements(s)
}
```

It is also possible to define the **main()** function to use only the first or the first two arguments (argc and argv). A sample makefile which prints its command line arguments is given below. The functions **printf()** and **element()** used in this example are discussed in the function-section below:

The arguments passed to **main()** functions as the list argv are:

```
void main (int argc, list argv)
   {
       int
           i;
       for (i = 0; i < argc; i++)
           printf ("Argument ", i, " is ", element (i, argv), "\n");
   }
```

1. The name of the binary makefile which is interpreted by icm-exec. This is always the first argument.
2. Remaining arguments are those arguments which were explicitly supplied on the command line.

For example, to supply the arguments one, two and three to a makefile called try.im, one of the following invocations can be used:

```
         icmake test - one two three
                 or:
         icmake -i test.im one two three
```

In both cases, the first int argument of the function **main()** will have the value four. The first element of the list argv holds the name of the binary makefile (test.bim); the remaining elements of argv hold the arguments one, two and three.

The third argument of **main()**, envp, is a list holding the setting of the environment (the environment variables). An example of such a variable is PATH, specifying where the operating system searches for executable files. The envp list consists of pairs of elements, where each first element of the pair holds the variable name (e.g., the string PATH) and where the second element of each pair holds the value of the variable (e.g., a list of directories where executable files may be found).

An example of a makefile which prints the settings of environment variables is given below:

```
void main (int argc, list argv, list envp)
        {
            int
                i;
```

```
            for (i = 0; i < sizeof (envp); i += 2)
                printf ("variable ", element (i, envp), " has value ",
                        element (i + 1, envp));
        }
```

4.4.4. Expressions and operators

Icmake allows a large number of operators to form or combine expressions. Binary operators may be used with the following operand-types:

operand-types

1. Each binary operator must be used with two variables or constants of the same type, e.g., the addition of an int and a string is not allowed; icmake performs no default type casting.
2. Some operators may not be used with some types, e.g., string subtraction is not allowed, but string addition is.
3. The operators have a certain priority; some operators are evaluated before others. The priority of operators is identical to the priority used by C.

The binary operators recognized by icmake are summarized in the following table:

binary operators

All binary operators with the exception of the assignment operators are left-associative. The assignment operators are right-associative. The operators at the top of the table have the lowest priority; those at the bottom have the highest priority. Operators with different priority are separated by lines.

The unary operators are summarized in the following table. The unary operators have higher priority than binary operators and are right-associative. The exception is the expression-nesting operator, which surrounds an expression and does not associate.

unary operators

4.4.4.1. Logical operators

Icmake recognizes three logical operators: the logical and (&&), the logical or (||) and the logical not (!). These operators can be used to combine or reverse logical expressions.

The logical not operator reverses the logical outcome of an expression. The logical and operator and the logical or operator group conditions. Icmake

evaluates a combined condition using these operators until the outcome of the condition is determined, in analogy to C:

1. In the condition (c1 && c2), c2 is not evaluated if c1 yields zero, since when c1 yields zero the combined condition can only fail. Therefore, c2 is evaluated only if c1 yields not zero.
2. In the condition (c1 || c2), c2 is not evaluated if c1 yields not zero, since when c1 yields not zero the combined condition can only succeed. Therefore, c2 is only evaluated if c1 yields zero.

Logical operators may be used with any type of expression. An int constant or variable yields its integer representation. A string constant or variable yields not zero when the length of the string is non-zero; e.g., string “a” yields not zero.

A list or variable yields not zero when the number of strings in the list is not zero, e.g., in the following code fragment the making process is stopped when no files with extension “.c” are found:

list

to be continued...

Look for part 3 of 4 of the IC Make Article in the next issue of *Linux Journal*.

Archive Index Issue Table of Contents

Advanced search

Cooking with Linux: Linux Leadership

Matt Welsh

Issue #2, April-May 1994

Despite all evidence to the contrary, people just happen to love rules. They love being told what to do—“leadership” is seen as a virtue. Whatever happened to personal liberty? How can we reconcile freedom with control?

All right, maybe this isn't the best way to expound my crackpot political views, but this ethical question is a good way to present the specific case of Linux.

I'm no political scientist, but it seems fairly obvious that almost any group of people working together needs leadership of some kind. More often than not, that leadership comes from a single person—that person makes the “final” decisions, no matter how intertwined the chain of command appears to be. Sometimes you have organizations that are relatively flat, where everyone is on equal ground—yet, invariably, there is a leader somewhere in the pack. More frequently you find conglomerates that are arranged in a thick mesh of hierarchy, more complex than any ISO standard. In such cases there are many leaders, each at different levels of the proverbial ladder. (Those readers who tuned in last month will remember this being discussed in some detail.)

But let's not repeat ourselves. Compared to most other organizations, the Linux community is as flat as a cracker (or nuttier than a fruitcake, depending on how you look at it). How could a system as complex, muddled, and technically fragile as a Unix kernel emerge from this hoi polloi? Good question.

The funny thing is, a Unix kernel did emerge, like it or not. Unfortunately, more and more people are starting to realize that Linux got where it is today without a central organization, without any kind of development hierarchy, without any rules or defined structure --without any thousand-page-long standards documents. And -this is the kicker—without a single penny of profit to back it all up. It's enough to gray a capitalist's hair.

Am I contradicting myself? Didn't I say that Linux did have some kind of organizational hierarchy? Well, that point is still in dispute. There are certainly no rules or defining factors which force the community to organize in any way. The software is freely distributable. If John Doe wanted, John Doe could start his own development effort, completely detached from, but based on, Linux. What's stopping people from doing just this?

The fact is that the Linux development hierarchy, if there is one, is a contribution-based organization, as discussed last month. A technocracy, if you will. But the primary drive is not control, or fame, and it sure isn't money. The thrust behind the whole thing is the desire to hack. Yes, there are many people in the “Linux community” who don't share these motivations—specifically, those who are aiding the project by selling the software commercially. But that has its place as well—would Linux be where it is today without some kind of commercial support? Probably not.

Of course, that statement depends on how you define “where Linux is today”. Many members of the development community would not consider commercial success to be one of the goals of Linux. Although it has the nice side-effect of increasing the distribution and the installed user base, that's not what Linux is about, fundamentally. Creating a free Unix system shouldn't require commercial success in order to be widely distributed; that's a kind of oxymoron.

Nevertheless, people are beginning to observe that Linux is successful (however you define the term) without any kind of conventional development structure. So, they think, “Shouldn't Linux development be organized centrally?” The lack of central support is seen as some kind of misfeature of the system: something that Linux should have, but doesn't. It is automatically assumed that applying a robust and well-tested hierarchy—beefing up the chain of command, as it were—will make things work better. Hey, it works for Microsoft, why can't it work for us?

In recent months, a number of proposals have come forward in an attempt to centralize the Linux development community. In fact, I supported one of these proposals—it does seem that Linux would benefit from the formation of an organization, somewhat similar to the Free Software Foundation, that would act as an entity to take credit for the software. This organization could own the copyrights, sell the T-shirts, and take the blame—making the legal issues surrounding the marketing and development of the system slightly less messy.

However, it has become quite clear that this kind of structure would be completely foreign to the concept of Linux, as it stands today, as well as to the developers who have brought it this far. Linus and others don't want to deal

with logistics—they just want to hack. It takes enough time and energy to program and debug Linux as it is; the last thing that they need to deal with is more administrivia. Some fear that the avoidance of hierarchy may limit Linux in some way—that in order to really hit a home run in the software world, we will need to pay more attention to meta development. If that's the case, so be it. Linux doesn't need to debunk Microsoft or IBM in order to win with hackers. We're not here to obsolesce Windows NT. Although it is a fun thing to think about.

If nothing else, Linux is about doing things in a new way. (All right, so why are we implementing Unix? Another good question.) Capitalism and commercialism are alive and well, and honestly don't need any help from the Linux community. The approach that the Linux development is taking is a new—or perhaps a very old-fashioned—one, in which technical excellence is valued over market return. Now, everyone admits that Linux has various technical problems, but who says that Linux should berated by the same standards as commercial Unix implementations? I'd rather work with a system that is constantly undergoing development than a large commercial Unix package that I don't even have the source code for. The point of Linux is to develop it—there is no specific “final goal” where development will come to a standstill.

Linux is hard proof that there are alternatives to the corporate and economic systems that make the world go `round. Even so, who's to say? In five years, or maybe a month, the Linux community could fall apart, providing a textbook example of “what not to do” for future generations of software developers to heed.

Many of the primary Linux developers aren't very interested in regrouping themselves into a more structured clan. Unfortunately, this leaves open the possibility of some third party stepping in, forming such a group, and claiming to be the official sponsor of Linux. “Official” by whose mandate? I claim that the only group with the power to claim any kind of officialdom when it comes to Linux is the developers themselves -the ones responsible for bringing us the software in the first place. Happily, nothing of the sort has happened so far. Yet, the clock ticks ever onward: because the developers have been too shy to step forward and claim their title as the official arbiters of operating-systems heraldry, some impostor may attempt to seize the prize instead.

The bottom line is that the Linux development cabal should do what's necessary to protect their own right to the software which they have produced. The GNU General Public License does this by requiring the software to be explicitly copyrighted by the author. However, it seems that much of the credit for Linux has been implicitly defined—it's not immediately clear who's responsible for what. (When asked, I imagine, most of the developers would

point in either direction and say, “It's their fault!”) In order to clear up the copyright fog, a compendium of credits for the development effort is currently being produced. Another way of achieving the same goal would be to form some kind of abstract entity calling itself “The Linux Development Organization”.

This organization may not do anything differently than what is done today. In fact, it could be virtually substance-free: “The set of all Linux hackers” without a rigid definition of the set's members (excepting, of course, the man who started it all—for Linus Torvalds, there's no backing out now.) Either that, or someone could sell inexpensive memberships to the foundation which would go to support Linux and free software in general.

Such a group would give the Linux development effort the appearance of congelation—as if it really had its act together. Such a group would give the rest of the world someone to point to—either in admiration or in scorn.

But some of us just like the sound of “The Linux Foundation”.

Archive Index Issue Table of Contents

Advanced search

Letters to the Editor

Various

Issue #2, April-May 1994

Readers sound off.

Dear Editor,

Thank you for e-mailing the *Linux Journal* advertising rate card. I am not interested in advertising in the *Linux Journal*, for the following two reasons.

1. I believe the free software “formula” (give away the software, charge for services) only works if the software being given away generates a demand for services - i.e., if it is buggy, poorly documented, difficult to understand, hard to port, hard to modify or extend, etc., etc. In my experience, Linux is at least the first three of those things (details on request), but I believe my own software (which is given away free with the Linux distribution) is less so: In the last two years, since Ghost script reached commercial quality, I have received almost no requests for such services from the thousands of users on and off the Internet who might have been expected to want them. A lot of people do post questions and bug reports in the gnu.ghostscript.bug newsgroup, but I don't particularly care to have my time nibbled to death answering the same simple questions over and over again, even if I'm being paid for it, so I usually respond to them by including the answers in the documentation being prepared for the following release. (Once a FAQ gets established, I expect the volume to drop by about 80%.) And the news group also contains regular postings from people who have written new drivers or have ported Ghostscript to new environments with no help from me.

2. The people who embrace Linux are people who (a) don't have much money, and (b) are willing to live with a system that, outside its core functionality, is FAR below commercial quality. These are not the people who are going to become substantial customers of my business, which consists of licensing Ghostscript for commercial use at market-rate prices.

“I have gotten so frustrated I am now willing to switch to a commercial system”. I just don't want to put up with the loss of time and productivity any longer. My one experience with pay-for-service was a waste. It may, of course, turn out that the commercial systems are no better; if so, I'll be coming back to Linux.

I am sure the *Linux Journal* will be valuable to people for whom the savings in up-front cost is more important than having a reliable, supported, productive system, but I don't count myself among them.

Sincerely,L. Peter Deutsch,Aladdin Enterprises

Reply from the Editor:

I wrote an article on Unix on a PC that appeared in the March, 1986, issue of Unix World and have used PC-based Unix systems for almost ten years. Until last year when I decided to “upgrade” some systems to Linux, my work was with products from major commercial vendors. I have found Linux to generally be a better operating system than those commercial products-better in terms of reliability and support.

I don't want to go through a blow by blow description of the problems I have had with commercial products, but let me say that newer commercial products have had more bugs than the older ones. In our office (where we run a network of Linux machines) we have encountered one operating system problem that causes us to have to reboot occasionally. This problem (a serial driver bug that causes ports to hang) is fixed in the current Linux release but we haven't upgraded yet. Note that this bug is virtually identical to a bug that I have been dealing with for four years in a commercial Unix system with a well-known intelligent communications board.

What's the difference? In the commercial system the vendors (the operating system vendor and the communications board vendor) both feel it is in their best interest to deny that a problem exists. The result for me, the user, is that four years later I still have to reboot the system about once a week. Because it is clearly in the best interest of the Linux community to fix all bugs, the existence of this problem was openly discussed and it was fixed very quickly.

This is not to say that Linux is always better than an alternative. But I don't think your experiences are typical of those of most people in the Linux community. Many commercial applications are getting ported to Linux with little or no difficulty-substantially much less difficulty than people were used to in porting to a system such as Xenix and generally much less difficulty than porting to any commercial Unix system.

I also think you are off base with who the average Linux user is. We see more and more people moving to Linux because it does the job for them. Many of these people are coming from MS-DOS because they need the multi-tasking capabilities and ability to seamlessly address memory. Others are using Linux-based computers to replace workstations and X-terminals.

But I feel that once you get more experience with the commercial alternatives to Linux and once Linux gets more experience with commercial users, you will find it is a product that has matured very quickly.Editor

Archive Index Issue Table of Contents

Advanced search

Linux 2000

Phil Hughes

Issue #2, April-May 1994

I see Linux as a progressive movement as well as an operating system. And, with any movement you need to chart your direction. To help with that charting I decided that rather than write the April, 1994 editorial I would just go ahead and write the one for January, 2000. So, here it is...

I see Linux as a progressive movement as well as an operating system. And, with any movement you need to chart your direction. To help with that charting I decided that rather than write the April, 1994 editorial I would just go ahead and write the one for January, 2000. So, here it is...

In the past 7 years we have seen Linux go from an idea for a small Unix-like system into a movement to bring affordable, reliable multi-tasking software to anyone who could buy a rather minimal computer. In fact, we have seen that in some parts of the world people are more likely to have a Linux system than be connected to the electric power grid.

Now we see Linux and an Internet connection in over 100 million homes worldwide. How did this happen? Cost is the best answer. Some of you probably remember an old program loader called MS-DOS. Back in the 1980s it was being marketed as an operating system and it managed to establish a user base close to that of Linux today. But it had three fatal flaws:

- it only ran on one type of computer system and could be expanded to support the full capabilities of new microprocessors
- it cost money
- it didn't support multi-tasking

We can excuse the first flaw as it was originally written for a project at a computer company and was never intended to be marketed to the general public. Although a more visionary company might have made a better decision we can only say that hindsight is 20-20.

The fact that people actually had to pay for a copy of MS-DOS (or, more properly stated, were supposed to pay for it) could also be considered as a very short-sighted decision on the part of Microsoft. As we all know now it is the added value-training, customization and, of course, the user-specific applications-that make the money. Giving away operating systems helps to sell these services along with hardware.

The final flaw, however, is what resulted in the demise of MS-DOS. I remember that back in 1986 I gave a talk at a personal computer user's group meeting in Seattle. I had brought along an IBM-AT (remember those-it had an Intel 80286 processor in it and people ran MS-DOS on them) and a couple of H19 terminals (Heath kit? Another flash from the past for some). I had a version of the Unix system running on this hardware.

I talked about Unix systems pointing out multi-tasking as being a primary benefit. I was amazed when these allegedly computer-literate people didn't understand why multi-tasking was absolutely necessary. In fact, one of the group members actually said “I don't like Unix because it accesses the disk when I'm not doing anything”. Today, 99% of the computer system users don't even know what a disk is, much less a disk access.

With the advent if ISDN in the early 1990s and personal satellite stations in the late 1990s, connectivity became the big issue. People quickly realized that they didn't want to know what their computer was doing, they just wanted to see the results. Could you, for example, imagine manually instructing your computer to call up another computer? Well, Unix systems pretty much pioneered the initial ideas behind these sorts of computer links with the advent of the uucp program suite back in the early 1970s.

When the average user, without using these words, asked for a multi-tasking computer system, Linux was there and waiting. We have to give credit to early Linux activists (and *Linux Journal* itself) for going out to companies that intended to market personal Internet stations and point out that Linux was a more capable and less expensive base to use for their products. The result, as you can see today, is that most personal Internet stations are based on the Linux operating system.

But there is more to the success of Linux.

People recognized they would rather pay for service than “things”. Linux, much like my first car, a `55 Chevy, offers a choice for the consumer. They can either fix it themselves or then can hire someone to fix it. That someone can be a representative of the manufacturer or the kid down the street. This was

certainly not the case with proprietary operating systems or vehicles of the 1990s.

How is *Linux Journal* Doing?

We are pleased to announce that as of November, 1999 90% of our subscribers are now via the Internet rather than on paper. We do, however, see that other 10% as so important because that is where those who are new to computing (yes, there still are some) find out about Linux and how easy it is to get their Linux system on the Internet. Over the years, most of our subscribers have moved from paper copies of *LJ* to an Internet subscription once we got them up to speed.

To make this electronic version possible we had to upgrade our offices to a complete Linux network. Even though all of our editorial and advertising work was done on Linux systems from our humble beginnings in 1993, our production, subscription and accounting systems ran on other computers. Today our seamless ISDN connections (and the satellite link to my office outside of Yaak, Montana) make this startup seem like a nightmare rather than the reality of seven years ago.

What Else Make Linux a Success?

These two events were a significant help in getting Linux on its way.

- formation of MoAmI Semiconductor from Motorola, AMD and Intel engineers in 1994. Linux was first operating system to run on their 32-bit and 64-bit chips in native mode in 1996.
- Linux became the most popular operating system used in computer science classes in 1995. This meant that the pool of available talent in the Linux market was huge.

But we also need to consider the how events in the Unix community helped Linux. When Novell decided to buy USL in 1993 it was seen as a move to get their own product out there instead of following each Microsoft move. As we now know, this worked and NT (remember, it was going to take over the world) became the niche operating system. Likewise, the decision of many fence-sitting vendors to go with Linux gave it the needed push that caused it to become a mainstream system.

This could be considered a political decision. Unix, although open if you had an extra $100,000 for the purchase of source code meant that its openness was restricted to existing companies. Linux, with its $0 to get started characteristics made it possible for creative talent to get into the computer business-much like

in the days of the Altair and the Apple II. The big difference is that the hardware was generic and inexpensive in 1994 so the creative work went into software.

This creativity made the larger vendors realize that they needed to stick with hardware and support as profit centers. Going to Linux as their operating system both reduced their software development costs and made it easier for them to find pre-trained systems programmers for the software work they still needed to perform.

Is there one specific application that I see as making everyone want a Linux-based computer system? Yes. I would say it is the availability of telephone directory information on-line. The fact that it is free and much easier to use than a traditional phone book has caused more people to elect for an Internet connection and Linux which offers a free reader for this information that a 5-year-old can easily use.

What happened to those old Linux activists and developers?

Strange as it may seem, most are still writing code or books. We don't see any who are CEOs or multi-billion dollar corporations. There are successes and many who were involved in the early days of Linux are making a very comfortable living but it seems that these people chose a career path based on their interests rather than attempting to become rich, famous or powerful.

Many of those who were minor players in Linux development or just got involved because Linux was so popular in colleges are now independent consultants. And they credit this happening with the Linux philosophy. The fact that they could get source code and learn about real software while in college gave them the necessary skills to more directly into the work of their choice.

What's Left to do?

We need to offer Internet connectivity to everyone. In the 1930s we started a rural electrification project that was to bring on-grid electricity to everyone in the United States. In the 1950s television broadcasting was seen as a way to get information to everyone. Both of these efforts had benefits but they also had an associated cost. They encouraged people to go to their own separate spaces, interact less with other people and passively consumei—consume both products and consume information.

In 1990 people were much less likely to know the names of their neighbors or world leaders than the names of fictitious characters on TV shows. Although Internet connectivity may not help people get to know their physical neighbors it does help them build a community of electronic neighbors. Using the Internet is active, not passive. Whether people elect to do research or electronically talk

to another person they are now making real choices and possibly talking to real people.

Because Linux was so significant in getting tens of millions of people connected to the Internet in the past five years and Linux machines make up the majority of the machines connected to the Internet today, I see this as a project that the Linux community should take on. In 1993 and 1994 we were all out there telling people about linux. If today we all walked next door, introduced ourselves to those neighbors that have lived there since 1990 and then offered to help them get connected to the Internet we could claim another huge victory for the Linux progressive movement before year's end.

As always, send me e-mail at info@linuxjournal.com and tell me what you think. And mention *LJ* to your neighbor while your at it.

Archive Index Issue Table of Contents

Advanced search

New Products

LJ Staff

Issue #2, April-May 1994

Motif for Linux, MetaCard for Linux and more.

In this column we will publish press releases and announcements about new products that we feel will be of interest to our readers. If you have a product that you feel might be of interest, send your press releases to the editorial offices of *Linux Journal* or e-mail to info@linuxjournal.com.

Motif for Linux

Sequoia International, Inc., has released a complete OSF/Motif 1.2.3 runtime and development environment for Linux, Coherent, FreeBSD, NetBSD and BSD/386. The entire package includes: Shared and static libraries (operating system dependent), header and include files, complete on-line manual pages, sample source code to demo programs and the OSF/Motif Users Guide.Only $149.95. For more information: (305) 480-6118 or info@seq.com.

Expect to see a review of Sequoia's product in a future issue of *Linux Journal* - Editor

MetaCard for Linux

MetaCard Corporation has developed a whole new approach to developing GUI applications and hypermedia documents on Unix/X11 workstations. Rather than expensive and complex applications development environments which require extensive programming with a third generation language, MetaCard offers the ability to create and modify applications using interactive tools and a simple scripting language.

MetaCard can also be used to prepare on-line help and training packages. The multiple-card metaphor and hypertext-linking capability makes it a natural for producing on-line reference manuals. As an example, the complete

documentation for the MetaCard environment itself is available on-line in MetaCard stacks.

MetaCard 1.3 is available on 14 different platforms including Linux. Single user licenses (for any platform) are $495 and come with unlimited e-mail technical support. To receive a free save-disabled but licensable copy of MetaCard, e-mail to info@metacard.com or call (303) 447-3936. You can also download the current engines, documentation and an unlicensed Home stack from ftp.metacard.com, directory MetaCard or from ftp.uu.net, directory vendor/MetaCard on the Internet.

Book: Linux: From PC to Workstation

by Stefan Strobel and Thomas Uhl

Linux, a relatively new free Unix system for PCs, has emerged as a viable alternative to commercial Unix systems. It turns a 386/486-PC into a Unix workstation with performance characteristics comparable to a RISC workstation. The book by Thomas Uhl and Stefan Strobel introduces the concepts and features of Linux. Moreover, it describes the features and services of the Internet which have been instrumental in the rapid development and wide distribution of Linux. Finally, the book gives an overview of the wide range of applications that are available for Linux.

With Linux, a system has become available to the computing community that stands in the tradition and spirit of Unix (from the forward by J. Gulbins).

ISBN 3-540-57383-6 Springer Heidelberg

Note that this book is written in German. Anyone want to review it for *Linux Journal*? -Editor

Linux Installation and Getting Started, by Matt Welsh, Version 2.0, 14th January 1994, is available on paper from the following sources:

For European distribution:

ISBN 3-926671-12-2, 188 pages, DIN A5 (148 x 210 mm) Costs DM 13.80 plus DM 2.70 for postage (includes VAT) and can be mail-ordered from Extent Verlag, Postfach 12 66 48, D-10594 Berlin, GERMANY.

Payment is in advance by check. E-mail andreasb@cs.tu-berlin.de for more information.

For U.S. distribution:

ISBN 0-916151-69-7, 188 pages, letter size (8.5" x 11") Costs $15.00 plus $3.00 shipping. It can be ordered by mail, phone or FAX from SSC, P.O. Box 55549, Seattle, WA 98155. Phone (206) 782-7733 or FAX (206) 782-7191. Payment by check, Visa, MasterCard or American Express. E-mail info@linuxjournal.com for more information.

Commercial Verilog Environment for Linux

Fintronic USA, Inc., is pleased to announce the availability of FinSim for the Linux platform.

FinSim: A high performance, compiled and interpreted simulation environment that supports both UDL/I and Verilog HDL. The FinSim UDL/I simulator features full language implementation. The FinSim Verilog simulator features full Verilog HDL implementation including gate, switch-level, user defined primitives, behavior, specify blocks, path delays, system tasks and functions, PLI, and VCD. SDF capabilities will be released March 1, 1994.

FinSim is part of Intergraph's recently announced Veribest Design System. The FinSim simulator is tightly integrated with Data I/O's ECS schematic capture system, allowing fast schematic simulation and back annotation of simulation results. The FinSim Verilog simulator is compatible with all VCD-based waveform displays including Veritools' Undertow IV, Design Acceleration's Signalscan and Systems Science's Magellan.

With the FinSim Verilog simulator, behavior level designs currently run up to 50 times faster than Verilog-XL v1.6. FinSim's performance at gate and switch level is fast and runs existing benchmarks.

Currently FinSim runs on Sun Sparc, Digital MIPS, IBM RS/6000 and SGI MIPS workstations. It is also available on Intel x86 platforms running Windows NT, SVR4.2, interactive Unix 3.2 or Linux. It will soon be available on Digital Alpha, Pentium and Hewlett-Packard platforms.

FinSim prices for a Unix workstation range from $15,000 to $25,000, including free upgrades to SDF. FinSim's list price for a PC platform is two thirds of the price of FinSim for a Unix workstation.

Fintronic USA, Inc., 1360 Willow Rd., Suite 205, Menlo Park, CA 94025.

Phone +1.415.325.4474, fax +1.415.325.4908 or e-mail info@fintronic.com for more information.

Usenet news via Satellite Link

PageSat, Inc., has redesigned its Usenet News Feed System by integrating the modem and receiver into a single desk-top data terminal. Renamed the PCSAT 100 Wireless Usenet Data Terminal, the system can burst data at speeds up to 57.6 kbps, supports hardware or software flow control and operates with a single external power supply.

Says Duane J. Dubay, National Sales Manager of PageSat: “We do have customers that are running this under Linux. You can ftp to `pagesat.net' and there is a Linux platform in the directory `pub/satellite'.”

For more information, e-mail info@pagesat.net or call +1 415 424-0405.§

Trans-Ameritech presents “The best Linux plus FreeBSD CDROM ever”

Available NOW for same day shipping!

Great news for the Linux community!

SLACKWARE - the critically acclaimed best Linux distribution - is now available on a CDROM from a company known for fastest delivery and best customer service. The new release is a result of a combined effort by Patrick Volkerding (creator of the Slackware distribution) and Trans-Ameritech and is based on Slackware 1.1.1 with kernel p14. In addition to the binary and compressed source distributions, the CDROM contains an uncompressed file-system that can be directly mounted and used. This file-system has many man-pages, info files, games, applications, etc.

To help the first time Linux users, many documentation files are provided. These files are readable from DOS even before installing Linux. For hacker's reference, an uncompressed FreeBSD source tree also is provided. If you have questions or problems Trans-Ameritech provides free support via e-mail within 24 hours.

The new CDROM is available for $30 plus shipping/handling. If you are a current customer, it is only $20. Shipping and handling for overseas (as in UK and Germany) is $8. In it's $5. Pay by check or credit card.

You can order by phone (408) 727-3883 or by FAX (408) 727-3882. Or, by sending e-mail to: roman@trans-ameritech.com Trans-Ameritech Enterprises, Inc., 2342A Walsh Avenue, Santa Clara, CA 95051.

Sealand Systems

Sealevel Systems, Inc., announces the availability of the COMM+8, eight-port RS-232 Serial I/O Board. This board uses 16550 buffered UARTs and supports interrupt sharing and full modem control. For more information, contact Sealevel Systems at +1 803 843 4343.

Archive Index Issue Table of Contents

Advanced search

Linux System Administration

Mark Komarinski

Issue #2, April-May 1994

Now that you have your Linux operating system running, how do you keep it running effectively? That is the job of the system administrator, also known as root, or sysadmin the person who does everything in the background that makes sure the machine you are working on does not come to a screeching halt in the middle of a program.

You as System Administrator

When you booted up your Linux machine, you instantly became the System Administrator. You may have fewer users, or you may not be connected to a network, but many of the functions are similar. Perhaps the only difference is that you know whom to complain to when the system is not configured properly. As this series continues, you should be able to get a good grasp on what you have to do to make your system efficient, secure and user-friendly. And you'll know how to do it.

Secure Passwords: The Key to a Secure System

When you have your machine set up, log in as root. By default, the root user has no password. So when you are asked for a password, just hit the ENTER key. This makes it easy to log in, as you don't have to go hunting around for a password after you set your system up. The disadvantage to this is that anyone can log in as root and have free access to your system. The first thing you must do is to change the password for root. To do this, use the passwd command. The root's password must be impossible to guess by normal users. Also, user accounts should be protected with just as much security. Here are some hints for good passwords:

1. Do not include anything significant to you, such as a nickname, phone number, your last name, etc.

2. Change your password often. (especially if you are on a network).

3. Do not use a word that can be found in a dictionary.

4. Make your password at least 6 characters long, with a mix of letters, digits, and/or punctuation characters.

5. Do not tell anyone your password and don't write it down anywhere.

6. Change your password often (just in case you forgot).

To change your password, type: **passwd**

Type in your password; then you will be prompted to type it in again just to make sure you spelled it right the first time. If the two passwords you entered do not match, your password is not changed. When you type in your password, it will not be shown on the screen. This is to prevent anyone around you from seeing your password.

A sample password changing run looks like this:

```
# passwd
Changing password for root
Enter the new password (minimum of 5 characters)
Please use a combination of upper and lowercase letters and numbers.
New password:
Re-enter new password:
/#
```

Making your system User-Friendly

Now that your system is more secure, make your system more user-friendly. One way of doing this is by changing two files, /etc/issue and /etc/motd. The /etc/issue file contains the message that you see above the login: prompt. This messag e sometimes contains the name of the site and the operating system. For example, my /etc/issue file reads:

```
linux ver. 0.99.14 (enry)
```

The next file you'll want to change is /etc/motd. This file is the one that gets seen every time you log in. Motd stands for Message Of The Day. This file usually contains a welcome screen and messages of general importance for all users. For example, my /etc/motd file has:

```
** Welcome to enry **
Good News!  SLIP will be available Real Soon Now.
Send mail to root for more information.
```

These are just examples. Your setups may look similar to this, but they don't have to be. There are other ways of making the system easier for other users (and yourself). For starters, you may want to look at the man pages for bash, X,

and any other kinds of utility programs you may be using. The man pages are a good source of information.

Multiple drives vs. the single drive concepts

The idea of a drive in UNIX is much different than it is under DOS. The Linux (and UNIX) way of looking at drives is as follows: Each physical drive can be split into partitions (or filesystem), and each partition can be mounted into a directory.

Because of this, there are no drive letters as in DOS; there is no drive A:, B:, C:, and so on. The users will be dealing with only one drive, and everything is shown in terms of directories to the user. You could have one filesystem for the root directory and other files (the / directory), another filesystem for user commands (the /usr directory) and another filesystem for user space (the /usr/users, /home, or /user directory). For example, I have three partitions that I use: one is for the / directory, one is for the /usr directory, and the third one is for the /home directory.

When I cd into the /usr or /home or / directories or any of their subdirectories, it is just like using cd from DOS to change to a subdirectory. I do not have to switch drive letters; I just have to use cd and I automatically go to that filesystem.

There are a few advantages to using this system. First, you can add more total space to the system. If you notice that the /usr/X386 subdirectory will take up more space than /usr has, you can create a new filesystem from the unused portion of your drive and mount that partition to /usr/X386. Second, all file operations are invisible to both the user and the user's programs. DOS needs a drive letter and a directory. Linux just needs a directory.

Linux refers to physical drives, such as the 3.5 inch floppy drive, as files in the /dev directory. For example, the first drive (called A: in dos) is /dev/fd0 in Linux. The fd refers to the fact that it is a floppy drive, and the 0 means that it's the first of that kind of device. Hard drives are referred to in two different ways. One way is the physical drive itself, the first or second (or more) drive. The other way is by partition on that physical drive. Hard drive entries in /dev are prefixes with hd to signify hard drive, followed by an an “a” for the first physical drive “b” for the second physical drive, etc. After that, a number specifies the partition within the physical drive. The first partition is 1, the second partition is 2, etc. SCSI devices follow the same format, only their prefix is **s**. You can remember that as**SCSI** device.

In order to help you think in Linux, here are a few DOS to Linux ways of thinking of the same drive:

```
DOS                     LINUX
A:                      /dev/fd0
B:                      /dev/fd1
C: (assuming first      /dev/hda1
   partition on the
   first hard drive)
E: (assuming second     /dev/hdb2
   partition on the
   second hard drive)
```

In my next article I'll discuss making our own filesystem (in ext2 and xiafs formats), and I'll go over some of the entries in the inittab file. If you have any questions or comments about my article, I can be reached through e-mail at: henry@acca.nmsu.edu or via mail through *Linux Journal*.

(henry@acca.nmsu.edu)

Archive Index Issue Table of Contents

Advanced search