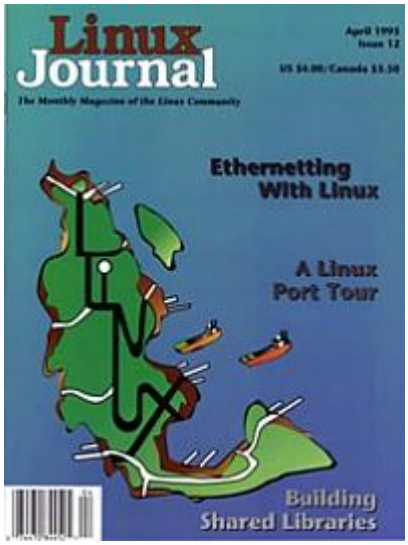
LINUX JOURNAL

Advanced search

Linux Journal Issue #12/April 1995



Features

Linux: It's Not Just for Intel Anymore *by Joseph L. Brothers*
A Linux Port Tour: Taking Linux Beyond its Intel-Processor-Based Beginnings
Leviathon *by Paul M. Sittler*
Accessing On-Line Information through Linux.
Ethernetting Linux *by Terry Dawson*
Connecting Linux to an Ethernet Network
Building Shared Libraries *by Eric Kasten*
Understanding and Building a Linux Shared Library System.

News & Articles

What's GNU? Plan 9 Part II *by Arnold Robbins*
Cooking with Linux *by Matt Welsh*
Kernel Korner : The ELF Object File Format: Introduction *by Eric Youngdale*
Mr. Torvalds Goes to Washington *by Kurt Reisler*

Reviews

Product Review InfoMagic *by Caleb Epstein*
Product Review Xfig *by Robert A. Dalrymple*
Book Review A Quarter Century of Unix *by Peter H. Salus*
Book Review The Mosaic Handbook for the X Window System *by Morgan Hall*

Columns

Letters to the Editor
Novice to Novice Linux Installation and X-Windows *by Dean Oisboid*
New Products
System Administration Setting Up Services *by Mark F. Komarinski*

Archive Index

Advanced search

Linux: It's Not Just for Intel Anymore

Joseph L. Brothers

Issue #12, April 1995

A Linux port tour.

Linux isn't just for breakfast, er, Intel, anymore. Everybody loves it and wants it on their favorite processor: 680x0, Alpha, MIPS, Sparc, PowerPC. That's good, because it makes Linus happy, Linux better, life easier for Linux users, keeps commercial OS vendors on their toes, and sells a lot of hardware. It's good, too, because Linux ports to newer processor technologies help ensure the continuing viability of our favorite operating system. On the other hand, the ports could split development and lead to bugs and confusion from too many code streams. The changes necessary for portability could mean slower Linuxes for everyone. To head off these problems, Linus and the other porters are working together to take Linux beyond itsIntel-processor-based beginnings.

For those who are new to the Linux community, we should start by explaining that there is a long-standing joke about “virtual beer” in the Linux community. The “Oxford Beer Trolls” were credited for having sent “virtual beer” (money with which to buy beer, presumably) to Linus, and soon “virtual beer” meant any sort of thanks or praise. Because the phrase has become common, puns on (virtual) beer are commonplace among Linux users. Let's take a flying tour of the virtual breweries, their plumbing, hydraulic engineers and brew masters, and end with a quick tasting of the product. Let's start with the recipe. Table I lists the ingredients for each port and some notes on the process. You can see how far along each batch is.

- cross-tools consist of at least a compiler and binary utilities (as, ar, ld) that produce executables for the new machine + operating system combination.
- merged source refers to integration of the port with Linus' kernel source.
- simulator is a program that pretends to be the new hardware so new executables can be run and debugged.

- boot is the few hundred words of native assemble code that checks, and may set up the hardware before beginning to load the kernel. The port is done once that code stops changing.
- kernel refers to the minimum operating software needed to start a user shell. It includes memory management, process scheduling, rudimentary device drivers and at least one file system.
- runs shell includes the capability of running the basic Linux command line utilities.
- native tools are the result of using the cross-tools to cross themselves to the new machine + OS.
- SDK means the newly ported Linux can compile a working copy of itself from scratch, completely stand-alone.
- user apps consist of text processing, e-mail, alternate shells and file systems, more device drivers, really a complete character-oriented Linux. In short, everything except X-Windows.
- X-Windows adds a standard graphical user interface.

As you examine the birthplace of each port, you may enjoy keeping track of its relative ability to intoxicate. For virtual brews, this is calibrated in BogoMips (Bogus Misleading Indication of Processor Speed). Remember, this cannot be used to compare different processors.

Note that the “Mips” along the left side of Table II refers to a RISC processor family, not a measurement of speed. Now we'll check in at each brewery to look around, meet the makers, and take the temperature of the batches. We won't spend any time with the ix86 Linux you are all familiar with. Besides, it's not a port; it's the original. On with the tour.

Name: Linux/68k

Linux/68k is a port of Linux to Amiga and Atari 680x0 platforms having hardware memory management and floating point support.

- Status:beta
- FAQ Access:pfah.informatik.unikl.de:8000/pers/jmayer/linux68k-faq ftp://tsx-11.mit.edu/pub/linux/680x0/FAQ (or any tsx-11 mirror)
- FAQ Maintainer:Joerg Mayer, jmayer@informatik.uni-kl.de
- Mailing Lists:Linux-activists channel 680X0 at linux-activists-request@niksula.hut.fi linux-680x0@vger.rutgers.edu to subscribe, send mail to: majordomo@vger.rutgers.edu
- Source Access: www-users.informatik.rwth-aachen.de/~hn/linux68k.html http://src.doc.ic.ac.uk/packages/Linux/tsx-11-mirror/680x0/ ftp://tsx-11.mit.edu/pub/linux/680x0ftp://ftp.germany.eu.net/pub/os/Linux/

Mirror.SunSITE/ftp://src.doc.ic.ac.uk/computing/operating-systems/Linux/tsx-11-mirror/680x0/

- Release Coordinator:Amiga-Hamish Macdonald, hamish@border.ocunix.on.ca Atari - Roman Hodek, rnhodek@cip.informatik.uni-erlangen.de
- Supported Platforms:Amiga-A3000, A3000T, and A4000/40 (but not the A4000/30)Atari-Falcon (plus FPU), TTMac—no information available
- Help Wanted:
  - More hardware-dependent device drivers are needed. The Kernel Hackers Guide needs to be updated for the 680x0 with special emphasis on memory management.
  - Linux/68k runs a beta-quality 680x0 Linux kernel on two makers' platforms, a number of file systems, shells, and some utilities. There is no X-windowing yet, though work on it is progressing. Full native development is possible using the specially contributed tools.
  - The Amiga and the Atari ports were merged so successfully that the same kernel image (the vmlinux from tsx-11) runs on both machines. Another advantage of this is that all user program binaries should work on any machine running Linux/68k if only hardware-independent devices are used. These devices include ramdisk, mem, pty, tty, vt, slip, net/inet, and general SCSI stuff. These file systems have been ported: minix, ext2, msdos, proc, isofs, nfs.

Hamish Macdonald, describing the state of things on January 4:

> I've currently got a private source tree at the v1.1.61 level, I'm tracking Linus' portability changes, and have been submitting comments to him on portability-related abstractions. As time permits, I'll probably be submitting more changes to this end.

Name: Linux/Alpha

Linux/Alpha is a port of Linux V1.0 to the Digital Equipment Corp. Alpha RISC microprocessor.

- Status:Kernel SDK
- Supported Platforms:Linux/Alpha is primarily aimed at PC-class Alpha platforms that support ISA, VLB, and PCI devices.
- FAQ Access:watch comp.os.linux.announce
- FAQ Maintainer:Jim Paradis (paradis@amt.tay1.dec.com)
- Mailing Lists:
  - linux-alpha@vger.rutgers.edu To subscribe, send mail to: majordomo@vger.rutgers.edu

    - linux-axp@amt.tay1.dec.com To subscribe, send mail to: linux-axp-request@amt.tay1.dec.com
- Source Access:gatekeeper.dec.com:/pub/DEC/Linux-Alpha
- Release Coordinator:Jim Paradis (paradis@amt.tay1.dec.com)
- Help Wanted:
    - If you have an Alpha-based PC-class system running OSF/1 (e.g. DEC 2000) you can use the same system for development and test bed. Otherwise, you will need two systems.
    - The development system can be any system that can support the Linux/Alpha cross-development tools. The cross tools have been successfully built and tested on the following systems:
        - Linux 1.1.x 386/486 (natch!)
        - DEC OSF/1 Alpha 2.0
        - DEC RISC/ULTRIX 4.2 (MIPS)
        - SunOS 4.1 (Sparc)
        - Jim Paradis, announcing the Linux/Alpha
- Developers' Kit 20 January `95:
    - The Linux/Alpha Software Developers' Kit is the first public release of Linux operating system components for Digital's Alpha family of microprocessors.
    - The SDK is available via anonymous FTP [see above]. I STRONGLY suggest that you first download the files README and SDK_CONTENTS and read them before downloading anything else (hint: you do NOT need to download all 55Mb in that directory!)
    - The Linux/Alpha SDK is NOT a fully-functional Linux distribution. The documentation is EXTREMELY sketchy and is mainly in the form of back-of-the-envelope notes. Linux/Alpha is not self-hosting; one must cross-compile the kernel and utilities on another system using the available cross-development tools. The kernel is extremely fragile, and several important code paths have not been tested yet. Very few utilities are available; you can bring the system up to a shell prompt, but you can't do much of anything else yet. Device driver support is minimal; currently, we support console-callback device drivers, but these are EXTREMELY slow (console-callback drivers are the Alpha equivalent of BIOS drivers on Intel systems). We have ported three sample drivers so far for the DEC 2000 AXP system: keyboard, text-mode VGA, and Adaptec 1742 SCSI.
    - In other words, Linux/Alpha is currently in a state that only a kernel hacker could love. If that describes you, then by all means download the SDK and give it a try!

Name: Linux/MIPS

Linux/MIPS is a Linux port for computers equipped with Mips R4x00 processors.

- Status:tools alpha; kernel pre-alpha
- Supported Platforms:Deskstations Tyne and Acer PICA with R4400PC andR4600 processors. The Deskstations support the ISA bus.
- FAQ Access:www.waldorf-gmbh.de:/linux-mips-faq.html ftp.waldorf-gmbh.de:/pub/linux/mips/linux-mips-FAQ
- FAQ Maintainer: linux@waldorf-gmbh.de
- Mailing Lists:
    - linux-mips@vger.rutgers.edu to subscribe, send mail to majordomo@vger.rutgers.edu
    - linux-activists channel “mips”. To subscribe, e-mail linux-activists-request@niksula.hut.fi with “X-Mn-Admin: join mips” as the first and only line.
- Source Access:sunsite.unc.edu:/pub/Linux/ALPHA/mipsftp.uni-mainz.de:/pub/Linux/MIPSftp.waldorf-gmbh.de:/pub/linux/mips
- Release Coordinator:Andreas Busse (andy@waldorf-gmbh.de)
- Help Wanted:Sure, you can help! If you have a Mips box, please let us know.
    - From the FAQ:
        - We have a half-way working kernel for the Deskstation boards. Console, floppy, serial and parallel I/O seem to be OK. The C library is nearly complete. We expect the first user process running soon. The kernel will be released when a user can issue shell commands, probably early in 1995.
        - Support/development tools available include cross compilers, assemblers and linkers ready to use for Linux/ix86, SunOs 4.1.3 and Solaris2.3. A Mips R2000/R3000 simulator (SPIM) for Linux/ix86 is also ready to download.
    - Andy Busse comments:
        - My part of the project is kind of project management. And, of course, it was my idea to port Linux to Mipses. From my point of view, different native endianesses is probably the most complicated part of Linux/MIPS. Most systems come up little-endian while some run big-endian only. However, I still hope that it will be possible to have user code compatibility on all supported Mips boxes.
        - Ralf Baechle (ralf@waldorf-gmbh.de) is currently working on the Deskstation port:As you might have seen, the 68k port is

about to be merged into Linus' kernel distribution. Since the 68k port is the most advanced of the ports, I have high hopes for the integration of Linux/68k. It will make porting for all others a lot easier.

Name: Linux/Sparc

Linux/Sparc is a port of Linux to the sun4c, based on Version 7 of the Sparc architecture.

- Status:just starting
- Supported Platforms:Sun 4/20 is typical. A more complete list will be available soon.
- FAQ Access:see mailing list
- FAQ Maintainer: David S. Miller, davem@nadzieja.rutgers.edu
- Mailing Lists:linux-sparc@vger.rutgers.edu To subscribe, send mail to: majordomo@vger.rutgers.edu
- Source Access:ftp://tsx-11.mit.edu/pub/linux/sources/system
- Release Coordinator:David S. Miller, davem@nadzieja.rutgers.edu
- Help Wanted:Contact David S. Miller if you have a Sparc to boot on. In David Miller's words,
    - Right now, I have my test box do the following: 1) Print boot-up messages, 2) Determine the machine type (sun4c, sun4m, sun4d, etc.), 3) Determine the available physical memory on the machine and other types of information, 4) Probe the OpenBoot PROM for devices that are on the machine. The PROM is a real win here.5) BogoMIPS, the most important part of the port! This SUN 4/20 gives 17.94 BogoMIPS. 6) Completely map the kernel's virtual pages. 7) Enable and flush the Virtual Address Cache.
    - I have a lot of the architecture-dependent include/asm-sparc files written and am able to `make config; make dep; make clean' on the tree. A lot of the file system code can be compiled. Getting it to work is another story.
    - The current work on the Sparc port of Linux is aimed at the sun4c machines which are based on Version 7 of the Sparc architecture. The main difference (between machine types) is that the MMU's are accessed in a different fashion in V8 and onward. Fortunately, Version 8 memory management (for sun4m) is defined by the V8 manual “The Sparc Reference MMU”. I am attempting to make sun4m support easy to just plug in later. Yes, this means multi-processor support and all that entails. Although no such machines will exist before mid `95, I am doing some of my code with the Version 9 Sparc in mind: better prepared than not.

- I have been trying to coordinate my code with Linus such that we don't buck heads in the kernel tree, so to speak. Eric Youngdale and Linus have been extremely helpful in deciding how best to integrate my memory-management code into the current tree.

Name: Linux/PowerPC

Linux/PowerPC is a port of Linux to PowerPC processors, initially the 601 and 603.

- Status:resuming
- Supported Platforms:Apple PowerMac, Motorola PowerStack, IBM Power Personal PC. The PowerStack uses both ISA and PCI buses.
- FAQ Access:see mailing list
- FAQ Maintainer: brothers@halcyon.com
- Mailing Lists:linux-ppc@vger.rutgers.edu To subscribe, send mail to: majordomo@vger.rutgers.edu watch the mailing list for announcements
- Source Access:ftp://tsx-11.mit.edu/pub/linux/sources/system/
- Release Coordinator: brothers@halcyon.com
- Help Wanted:
  - A documentation specialist is needed. Knowledge of the Linux Documentation Project, SGML, HTML, TeX, LaTeX, and desire to learn literate programming with “noweb” are required.
  - Volunteers having PC-class RS6000 machines or IBM PowerPersonal PCs are needed for boot and kernel testing and to write or port device drivers.
  - The Apple PowerMac porters mostly have a cross-development environment (not freeware). Access to the Mac's ADB internal bus specifications appears imminent, as Apple now seems willing to release the information under certain conditions.
  - With the addition to the project of some Motorola PowerStacks (on order) and their soon-to-be owners at year end, `94, the PowerStack part of the Linux/PowerPC port is beginning to come together. A GNU cross-development tool set, targeted at the PPC, has been started.
  - Many thanks go to Northwest Nexus (info@halcyon.com) for supporting the Linux/PowerPC Project by providing the author's net access. Thanks also to MicroApl Ltd. (London, UK (MicroAPL@microapl.demon.co.uk)), makers of PortAsm assembler source translators, for their contribution.

Wrapping Up The Virtual Brewery Tour

Jim Paradis sums it up well:

> Implementing ANY operating system on a new platform, is a major undertaking. It has taken dozens, if not hundreds, of programmer-years to bring the Intel version of Linux to the point where it is today. Too, while a kernel port is a significant piece of engineering, it is only a small part of porting an operating system. It is not surprising that the non-Intel versions of Linux are taking some time to appear.

You can all help yourselves to the samples we have provided. What? The glasses are empty? Of course they are, that's one of the risks in breweries like these. Good brews take time. We do hope you've enjoyed your tour of the new virtual breweries, though. And remember, when sampling RISC ports, don't hold your breath!

Joseph L. Brothers, CCP, CDP, is a Senior Software Engineer in Motorola's Wireless Data Group where he writes distributed engineering productivity software using the noweb literate programming tool. On his own time, he volunteers as the Linux/PowerPC Project's PowerStack task coordinator.

Archive Index Issue Table of Contents

Advanced search

Linux in the Real World

Paul M. Sittler

Issue #12, April 1995

Leviathan: A Linux-Based Internet Information Server

Texas Agricultural Extension Service (TAEX) “helps people improve their lives through an educational process which uses scientific knowledge focused on issues and needs”. Much of TAEX's mission involves information transfer to the people of the State of Texas. Like many government agencies tasked with providing better service with a shrinking resource base, TAEX has long been interested in innovative approaches to better serving the public without the barriers of time and distance.

TAEX began experimenting with electronic information distribution in 1984 with dial-in bulletin boards. In 1992, we rescued an aging Compaq DeskPro 286 from the auction block. We transformed it into an experimental Gopher/PopMail/FTP (File Transfer Protocol) server. The surprising response to this soon filled the 40MB drive. We began an exhaustive search for a better platform.

The best server software ran on Unix platforms, but they were prohibitively expensive. We were already using some Unix platforms for mail and networking applications. We tried to use one as a gopher server, but the response time was inadequate. We realized that additional duty as an information server would require significant system upgrades. We would need to increase main memory, add more mass storage, upgrade the operating system, and obtain the separately marketed “software development system” (basically a C compiler, awk, and yacc). Unfortunately, any one of these items cost more than an entire high-end Intel-based PC! When we tried to get price quotes, we discovered that the TCP/IP networking package for the upgraded SVR4 operating system was “not yet available”. Coincidentally, our agency also received another decrease in funding levels.

About that same time, a Finnish college student added networking to his free “Linux” operating system. This free Unix clone ran on readily available Intel

80386/486 processors with inexpensive drives. Initial experimentation on an 80386-based box showed that it actually worked quite well! TCP/IP networking was thoughtfully included, as was a superb C compiler system. Linux was like no other Unix system we had ever used, as it combined some BSD features with some SVR4 features, while maintaining some POSIX compliance. Our initial confusion changed to the delightful perception that Linux provided a most sensible mix of desirable features.

We obtained a low-end 80486 machine, fitted with a network card, and installed Linux. Both UMNs gopherd server and NWU's GN combination Gopher/WWW server compiled easily, and “Leviathan” was born. From a user's viewpoint, Leviathan seemed to operate faster, and we hoped that it would be able to handle a bigger user load than the Compaq. We transplanted the information tree from the Compaq to Leviathan, and both machines ran side by side, tangible proof that simple (and obsolete) computers could still be useful.

The initial information served via gopher included the “Master Gardener” files, the TAEX personnel directory, and abstracts of all Extension bulletins and leaflets. Wherever possible, we also provided the full text of these publications. We put the TAEX Agricultural Software Catalog on line. The 200MB drive soon filled, and a 300MB drive was added. Leviathan began performing as a bootp and PopMail server as “extra duty”. We scanned several clip-art collections and made them accessible. Usage grew steadily. Soon, users from all over the world were logging in round the clock. The March 1994 access logs showed that 2,245 sites obtained almost 100MB of mostly clip-art at a rate of more than 500 accesses daily. Then, Mosaic happened.

We had been experimenting with the Mosaic WWW browser since December 1993, but it was under Linux and X-Windows that I first experienced an implementation of Mosaic that really impressed me. Mosaic under Linux was stable, flashy, and very useful. This caused me to review Mosaic for DOS/ Windows platforms. While not as stable and full-featured as the Unix versions, we evaluated it as tolerable. The World Wide Web concept seemed tremendously important, so we began demonstrating Mosaic throughout the Agency in February 1994. Newer versions of GN had added the ability to serve a new protocol, Hyper Text Transfer Protocol (http), which was used to tie together a “World Wide Web” (WWW) of networked information servers. In May 1994, text pages were marked up for Hyper Text Markup Language (HTML), and GN began responding to http requests as well as gopher users. Leviathan broke the 1,000 accesses/10MB per day mark that month with 31,427 requests, 3,406 of which were from http (Mosaic) clients.

We added anonymous FTP access in August 1994 at the request of many users. Leviathan began distributing the National 4-H Enrollment Management

software via gopher, http, and ftp access. In September, Leviathan also became TAEX's departmental CCSO qi/ph maintenance client. The 2,000 accesses per day watermark occurred in October 1994, when GN served 28,345 files (510MB) in 63,000 accesses, of which 60% (36,000) went to gopher clients. Access rates increased by approximately 10,000 per month in both November (72,300) and December (83,801). December 15th's 4,570 accesses broke the previous record of 4,370 accesses established the day before. December's average access rates were about 2,700 per day, with 20.6MB of files retrieved daily. Gopher type accesses still accounted for 51% of the accesses (42,371), but http accesses increased more than gopher accesses from the previous month.

Cumulatively, between February and the end of December, Leviathan served 188,672 files (3.71 gigabytes) to 33,542 unique machines in 434,025 separate transactions, of which 295,330 went to gopher users. During the same time, another 4,844 files (151MB) were retrieved via anonymous ftp. Leviathan was still serving as a bootp and PopMail server in its spare time, while accepting logins for maintenance of our part of the University-wide CCSO “ph” directory services database system. All this activity took place on an inexpensive 80486/33 computer sitting under a table, with neither monitor nor keyboard attached.

Information Server Content

Leviathan is a niche server, publishing information of interest mostly to adult distance learners and educators. We have the obligatory information describing the organization, and searchable abstracts and texts of selected agency publications. The clip-art collection has grown to over 1,600 individual images, from three states, available in multiple formats, with another 999 nearly ready. We added a small experimental slide image collection that may be accessed through a “Contact Sheet Image Selection” imagemap using a pointing device. The TAEX Computer Technology Group “OnLine” computer user newsletter is available online, and the “Master Gardener Problem Solver” has been extremely popular. The TAEX Software Catalog is available interactively and may be downloaded in PostScript, text, HTML, and WordPerfect formats.

Leviathan also serves documents from sources external to Extension. The Texas Telecommunications Strategic Plan and its Executive Summary are accessible. The National Performance Review documents are available. A large collection of Internet information, including all RFCs (Requests For Comment), FYIs (For Your Information), and STDs (STandards Documents) to date may be browsed. Leviathan includes convenient links to all WWW and gopher servers and enables easy access to other Cooperative Extension information servers. To aid others who wish to establish WWW sites, Leviathan provides a collection of icons (some developed locally) and links to pertinent tutorial and technical documents.

Future Directions

TAEX is preparing a newer, faster server with more storage capacity to better serve Leviathan's users. The immense popularity of the clip art collections has stimulated us to prepare even more. We are preparing an extensive collection of slide images that may be distributed freely for use in demonstrations and publications. We are updating our online personnel directory with plans to provide color photos and sound clips of everyone in the agency for easy access by the traditional media. The online personnel directory may someday be linked to a "home page" for every person that includes a description of individual expertise and experience. We are considering making several large topical databases accessible online. The popular Master Gardener series begs to be updated with color images, sound clips and animation sequences. We plan to add more on-line newsletters replete with color graphics. We are considering distributing some free software packages by gopher and http access.

Reaching Leviathan sidebar

TAEX is working with several other departments and agencies to stimulate the proliferation of more information servers. The South West Agricultural Meteorological Information Service (SWAMI) has been supplying pertinent weather information from a Sun workstation for several months. More recently, "Monarch", the TAEX Planning, Performance, and Accountability Server, came on-line on a DOS/Windows platform to keep the public and state legislators appraised of agency activities. Three of the Texas Research and Extension Centers plan to establish information servers. Two more experimental servers using alternative free software are currently "under construction" as TAEX explores this evolving technology.

The combination of telecommunications and computer innovations will together produce a technological imperative for change. This may require a major paradigm shift from information distribution toward providing information access. TAEX is preparing for the day when all of the information produced and distributed by the agency may also be made available digitally and online. With luck, it may actually be ready when the public is.

Impact of The TAEX Information Server

Leviathan has achieved some degree of national and international recognition. The NCSU list of "Top Ten Home Pages by Cooperative Extension Workers" points to us, as does their list of "Top Ten Extension-Related WWW Pages". Other Cooperative Extension services across the US and in Australia, Mexico, India, Israel, and Czechoslovakia have downloaded clipart. A cookbook, published in the UK by an author from Bangladesh, incorporated some of the

clip art as illustrations. An Australian server began mirroring Leviathan down under. Leviathan has been mentioned in several Canadian Agricultural publications. Leviathan was designated as a “Gopher Jewel” by several sites, and America OnLine lists it as their first agricultural information site.

The on-line software and image distribution has cut distribution costs considerably and made TAEX products more easily accessible. For example, Leviathan distributed 2.6GB of clip art on-line, saving taxpayers almost $17,000, compared to traditional floppy-based distribution methods. Each download of the Software catalog saves $5.00 in direct printing and mailing costs and has increased software distribution activity substantially. The National 4-H Enrollment Management software, distributed across the Internet has saved taxpayers more than $1,200 so far and provides more timely dissemination of updated versions. The 500MB of “Master Gardener” files would have filled 100,000 pages if printed, but being available electronically has saved $4,000 to date in printing costs alone. Most importantly, Leviathan has created an awareness of information server technology as a viable adjunct to traditional information distribution techniques.

Usage Analysis

While TAEX has received many favorable comments via e-mail, a more accurate progress assessment can be made by analyzing user access patterns.

Between February 13 and December 31, 1994, certain trends emerged. Daily usage patterns were high between 9:00 AM and 5:00 PM, but there were no “dead” hours. Weekend usage was about half that of weekday usage. Mosaic (http) access grew rapidly, but represented only 49% of Leviathan's total accesses. Gopher access was still important in 1994. The majority (62%) of the users were in the United States. Of these, most (67%) were from educational institutions, 16% were from the commercial (.com) domain, and 6% were from the government (.gov) domain. Leviathan's users were mostly (80%) from outside the TAMU system. Some 69,476 accesses (16%) were from Texas, which means that 84% of the requests were from out of state. Texas County Extension Agents visited 1062 times, while 215 Texas Extension Specialists connected 38,561 times.

Leviathan has been visited by people in 62 identifiable countries outside the United States on six continents. There have been contacts from 793 different educational domains in all 50 states. Delphi users accessed Leviathan 3,365 times, and America OnLine subscribers called 2,172 times. Compuserve users finally connected 48 times in December. The server has been contacted by 1,270 individual host machines more than 50 times; 561 of these visited more than 100 times; and 22 dropped in more than 1000 times in that period.

Lessons Learned from the Leviathan Experience

- If you build it, they will come....The demand for on-line information is staggering.
- If you provide useful information, they will come back, repeatedly, for more.
- Providing networked access to information is often cheaper than traditional methods of distribution.
- Networked access to information is only useful as to those who are networked. It should be viewed an adjunct to existing distribution methods.
- Simple text-based information is valuable. FTP and gopher are not dead. While the sizzle may sell the steak, content is more important than presentation.
- Consider your audience. Ensure that documents make sense when viewed in a text-only mode.
- Use graphics sparingly. Keep in-lined graphics small. Many users have slow dial-up network connections.
- The Internet does not end at the state line. We now serve people in other states and nations of the global village at no incremental cost.
- Servers must serve 24 hours a day. It is 3 PM some where on the Internet all the time.
- Servers should not be shut down for update and maintenance. This implies a multitasking operation.
- Graphics and audio files should be clearly identified through use of icons.
- Keep menu pages small and simple, with 5 ± 2 selections.
- Almost any old obsolete computer can be used as a server. Storage capacity is more important than powerful CPUs.
- Current WYSIWYG editors are often harder to use than simple text editors for producing HTML.
- Storage space and network bandwidth are both finite.
- Time costs more than equipment.
- Administering a server takes more time than expected.
- Organizing information is sometimes harder than producing it.
- Writing HTML documents is far easier than it first appears.
- We often overestimate what we can do in a week and underestimate what we can do in a year.
- Linux is not merely a hobbyist's toy; it is a solid, stable, professionally-implemented Unix clone that performs superbly as a production information server platform.

Summary

Evolving telecommunications and computer technology are combining to produce a technological imperative for change, requiring information access as well as traditional information distribution techniques. Government agencies will continue to be asked to provide more and better services with a shrinking resource base. Demand for information access will likely continue to increase for the foreseeable future, causing a corresponding demand for more network bandwidth. The TAEX combination FTP/Gopher/WWW server demonstrates that a low cost (> $1,500 US) computer running only free software can supply a moderate-to-large amount of information. Usage is growing at a rate of 10,000 accesses monthly. Demand for gopher http access. FTP is still a viable and desired method of file transfer. Information servers provide information to inhabitants of the global village without barriers of time and distance. TAEX's Gopher/WWW/FTP information server is a promising alternative medium for outreach and distance learning for our changing clientele. It is a digital extension of the agency's motto: People Helping People.

Paul M. Sittler (p-sittler@tamu.edu) is a Computer Systems Engineer for the Texas Agricultural Extension Service. He enjoys playing with technology and making it useful to others of his species. He received a BS and MS in Vocational/ Industrial Technical Education from Texas A&M University.

Archive Index Issue Table of Contents

Advanced search

Ethernetting Linux

Terry Dawson

Issue #12, April 1995

Linux comes with the networking tools. Terry shows you how to get your Linux system talking over Ethernet.

Many people who install Linux on a machine at their workplace or university also wish to connect their machine to the local area network, so that they can make use of an Internet connection or other machines on the network. If you are one of these people and the only thing that has been stopping you is not knowing where to start, I hope this article will demonstrate that it is easy to do so-as long as you are properly prepared and know what things to watch out for.

The following are a few key areas to consider when connecting your Linux machine to a network:

- The type of network you are connecting to.
- The type of network interface card (NIC) you will need.
- Configuring the kernel to support the network interface card.
- Addresses-Host, network, broadcast and router.
- Configuring Linux for your network connection.
- Routing.
- Names and nameservers.

The type of network you are connecting to is very important for a number of reasons. Most important is the many different network types. Currently Linux provides good support for Ethernet networks, but not much support for other network types such as ARCNet, Token Ring, FDDI and wireless LANs. If you intend to use an Ethernet network, keep reading. If not, don't despair—some very promising development efforts are under way to provide support for the other types of networks.

Ethernet

If you are fortunate enough to have an Ethernet network to connect to, then you still have to determine what type of cabling has been installed. Two popular types of cabling are employed, and you will need to find out which has been installed so you can choose an appropriate type of network interface card. The most popular cabling types are 10base2, commonly called “ThinNet”, which uses 5mm diameter coaxial cable and BNC (bayonet type) connectors, and 10baseT, commonly called “twisted pair” or “unshielded twisted pair”, which uses a cable with four conductors and an RJ-45 (telephone type) connector. The type of cabling will help determine what type of network interface card to choose for your Linux machine.

Ethernet Card

Your network interface card needs to have a connector that suits the cabling type you have installed. A number of modern cards come with both types of connector, and these are generally called “combo” cards. You should also think about whether you want a card suitable for an 8 bit slot or a 16 bit slot. The 16 bit cards perform better but are generally slightly more expensive. Another factor to consider is the bus type. If you are using a PCI machine, naturally you will need a NIC that suits a PCI bus. Be careful: you must also ensure that the type of card you choose is supported by a Linux kernel driver. It is best to avoid “clone” cards. While Linux supports some clones of popular cards, not all clones are the same. To be sure there are no problems, obtain the genuine article or try the card before you buy it. The Ethernet-HOWTO lists the types of cards supported by Linux and contains descriptions of each of them. You should refer to it before spending any money on a card.

The HOWTO recommends you opt for a 16 bit card such as the 3Com 3c503/16 or the SMC Elite 16/WD8013. Other cards which you might consider if you have a Vesa Local Bus or PCI motherboard are the BOCA Research cards. You might also choose an NE2000, but be careful, since some cards that claim 100% compatibility are not 100% compatible. (Their claims are based on being “100% software compatible”, meaning that they provide drivers for DOS that allow, for example, NetWare access. These drivers are useless with Linux.)

Cards to *avoid* are the 3Com 3c501 card (it performs badly and is broken by design) and Cabletron and Xircom cards, since free Linux drivers are unlikely ever to be available for their products, because these companies have chosen to require a non-disclosure agreement before releasing programming information, which would make it illegal to write a freely distributable Linux driver.

When installing the NIC you must make sure that the card's configuration does not clash with any other installed hardware. Some cards come with a DOS program to configure them. They use a programmable interface, and you should run this to “strap” the card with the configuration you want. You should be particularly careful of the IRQ, control port address and shared memory address settings. Each of these must be free for your NIC to use and be unused by any other hardware in the computer. I use a WD8003 strapped for control port 0x280, IRQ 7 and Shared Memory 0xD0000. Be careful if you use specialized hardware such as SCSI controllers or Multiport Serial cards, as they often use IRQ or Control Port settings in similar ranges, and may conflict. After you have physically installed the NIC, your next step is to check if your kernel already has support for your card. If it doesn't, recompile it so that it does. The easiest way to check if your kernel already supports your card is to reboot your machine. Check that the card is properly detected by the kernel by reading the messages the kernel prints when it is booting. If your card is properly detected, the kernel will print a message something like:

```
eth0: WD80x3 at 0x280, 00 00 C0 AD 37 1C WD8003,
        IRQ 7, shared memory at 0xd0000-0xd1fff.
        wd.c:v1.10 9/23/94 Donald Becker
        (becker@cesdis.gsfc.nasa.gov)
```

The settings listed should match those that you configured your card for. If your card has not been properly detected, rebuild your kernel to make sure the kernel has support. This is pretty straightforward and you have likely done it before. You simply change to the **/usr/src/linux** directory and run **make config**. You will be prompted as to whether to include various drivers. The most important sections for you to answer Yes to are:

- Networking Support?
- TCP/IP networking?
- Network device support?
- The driver for your card.

After you have configured the kernel to support all of the hardware you have installed, you do a **make dep; make** to build the kernel. Don't forget to do a **make zlilo** so that **lilo** will run your new kernel when you reboot. If you are happy that all has gone well, then you can reboot your machine and check that your card is properly detected as described earlier. If it isn't, double check that you have done everything correctly and that you have no hardware conflict. If you still have problems, refer to the Ethernet-HOWTO again, as it has lots of information to help guide you through determining what might be the problem.

Software configuration

If you are still with me, you are nearly ready to run. All you need to run are a few commands to start testing your network connection. After you have configured your kernel, you have to configure your Linux machine to suit your network. At this point you need worry about IP addresses. If you are lucky, you will have a network administrator who will have assigned you an IP address and told you the network and broadcast addresses to use. If not, you will have to find out another way. A good way is often to check the configuration of another machine that is already working. The network address is an address that refers to the whole network you are connected to. It is advertised so that people on other networks know how to get to you. Your host IP address is one address that belongs to that network. This must be yours and only yours, or else you will face lots of strange problems, so make sure you don't use a host address someone else is already using. The broadcast address is a special address that allows anyone to send data to everyone on your network. Some special services use this, and it is very important that it be configured to the appropriate value. Another important number you will need is your “netmask”. This is a mechanism that allows your machine to determine which host addresses are local to you (on the same network) and which ones are remote. The following example would be typical of what you would expect to find:

```
IP address:             202.105.54.56
Network address:        202.105.54.0
Broadcast address:      202.105.54.255
Netmask:                255.255.255.0
```

Once you have this information, be sure you have the correct software on your Linux machine. You must be particularly careful to ensure that the network tools you have (**ifconfig, route**) match the version of kernel you use. The NET-2-HOWTO describes where to get these tools and how to install them. If you run the **ifconfig** program with no command-line arguments, you will see that it lists the device mentioned in the kernel boot messages: “eth0”. This is your Ethernet device. It needs to be configured with the information above, and the **ifconfig** program is designed to do just that. Use a command line such as:

```
ifconfig eth0 HOST netmask NETMASK\
     broadcast BROADCAST up
```

So for the above example use the command line:

```
ifconfig eth0 202.105.54.56\
        netmask 255.255.255.0\
        broadcast 202.205.54.255 up
```

If you again run the **ifconfig** command with no command line arguments, you should see it now has the appropriate values configured.

More configuration info

Once you have your Ethernet device configured, you have one step remaining. As described earlier, the netmask tells your machine which addresses are local and which are remote. If the address is local, your Linux machine can route any datagrams directly to the Ethernet device. If they are remote, datagrams should be sent to the route which supports the link to the rest of the Internet. The router also has an address, so you will need to obtain this from your network administrator. Linux keeps a special table in memory to look up where to send datagrams. This table, called the routing table, is manipulated with the **route** command. In a simple installation, as you will most likely have, you will need to configure two routes for your Ethernet: one for your local network, and another that tells your Linux machine what to do with datagrams for any remote host. This latter route is called the "default" route.

The route commands are:

```
route add NETWORK dev eth0
route add default gw ROUTER dev eth0
```

and for the example listed earlier (assuming the router address is as shown):

```
route add 202.105.54.0 dev eth0
route add default gw 202.105.54.1 dev eth0
```

You can use the **route -n** command to display the contents of the routing table. The **-n** argument says to show the addresses as numbers and not try to look up their names, because you don't yet have your name resolver configured. To configure your name resolver, you will need to find out the address of the "NameServer" or "DNS" from your network administrator and put this address in your **/etc/resolv.conf** file in a line that looks like **nameserver NNN.NNN.NNN.NNN**, where **NNN.NNN.NNN.NNN** is the IP address of your nameserver.

Now you should be able to **telnet** to other IP hosts, both local and remote. If you have configured the name resolver of your Linux machine, then you can use their names, otherwise you should use their addresses.

In addition, the Linux *Network Administrator's Guide* is available from **sunsite.unc.edu** in the directory **/pub/Linux/docs/LDP/**, and can be ordered on paper from SSC (the publishers of *Linux Journal*) and O'Reilly & Associates.

Terry Dawson (terryd@extro.ucc.su.oz.au) has nearly 10 years experience in packet switched data communications, and maintains the NET-2-HOWTO and HAM-HOWTO documents for fun. Terry is keen to see Linux used widely in Amateur Radio applications.

Linux Programming Hints

Eric Kasten

Issue #12, April 1995

Building shared libraries for Linux is often considered a black art. In this article, Eric explains five simple steps to producting a standard Linux shared library and tells the cuious where to find more information

Shared libraries are probably most often used because they allow for the creation of shared executables, which take less disk space. They also allow the compression of multiply defined global variables into a single instance of the variable that all program modules share. Also possible is the creation of a compatible, drop-in replacement for an existing shared library. Improvements or fixes in the replacement library are then immediately available to executables the library is linked with. This last possibility is beyond the scope of this article.

Dynamically linked libraries (DLLs) have become an important part of the Linux system. Even though ELF (the executable and linking format designed for Unix SVR4), which makes creating shared libraries trivial, is just over the horizon, the current **a.out** DLL shared libraries will probably need to be supported for some time. In many cases, older versions of Linux will still need support, and commercial **a.out** libraries may require that an executable be built using **a.out** DLLs, because **a.out** libraries and ELF libraries cannot be mixed in one executable. Until ELF makes its way from the alpha releases of Linux into the more stable releases required for a production environment-and probably even after that-**a.out** shared libraries will continue to be built and used.

Provided with the source code for a static library, a shared version of the library can be created by completing five well defined steps. This article will explain how to apply these steps to create a simple shared library. Its aim is to help you understand shared libraries and how they are built, so you can successfully create more complicated shared libraries in the future.

Background

This article assumes the use of **gcc** 2.6.2 and DLL **tools** 2.16 with libc 4.6.27. Other versions may have slightly different syntax or may operate differently. All these items may be obtained by anonymous **ftp** from **tsx-11.mit.edu** in **/pub/linux/packages/GCC/** (**tools-2.16.tar.gz** is in the **src** directory). Follow closely all the installation instructions in the release notes, or unnecessary problems may result.

Shared libraries consist of two basic parts: the **stub** and the **image**. The stub library has an extension of **.sa**. The stub is the library an executable will be linked to. It provides redirection of shared functions and variables to the location where the real shared functions and variables reside in memory. The library image has an extension of **.so**, followed by a version number.

The library image contains the actual executable functions used by binary programs. The image also contains two tables of particular note: the jump table and the global offset table (GOT). The jump table contains eight-byte entries which redirect a call to a shared function from the jump table to the real function. The jump table exists to provide a method for creating compatible replacement libraries. Since each function has an entry of fixed size in the jump table, the jump table can provide an entry point for these functions at a location that remains constant between revisions of a library. This allows previously linked executables to continue to function without recompilation. The global offset table functions for global variables as the jump table does for library functions.

Each shared library is loaded at a fixed address between **0x60000000** and **0xc0000000**. If an executable is linked to two or more shared libraries, the libraries must not occupy the same address range. If two libraries should overlap, the location an executable is redirected to may not contain the expected function or variable. A list of registered shared libraries can be found in the **tools** 2.16 distribution in the directory **doc/table_description**. Examine this file when defining the load address for a new shared library to ensure that it doesn't conflict with the address for an existing library. In addition, you should probably register the address space used by a new shared library so that future libraries will not conflict with it. Registration is particularly important if the library is to be distributed.

Before Beginning

As mentioned earlier, this procedure is directed at the creation of a *simple* shared library. Although the steps for building a more complex library are the same, the process of modifying multiple or complex makefiles can become somewhat confusing. For your first attempt it is a good idea to select a library

which has all the library source in a *single directory*. A good choice may be the JPEG library, which can be retrieved by anonymous FTP from **ftp.funet.fi** with file name **/pub/gnu/ghostscript3/jpegsrc.v5.tar.gzi**. Or you could create several simple source code modules and a makefile to compile and build a static library. This test library need not do anything useful, since it is only for educational purposes. However, since you will already understand the inner workings of the build process, you can avoid the effort of attempting to understand another program's makefile logic. Also, be sure that a static version of the library can be successfully compiled before approaching the construction of a shared one.

Step One: Setup

The method presented here is not the only way to create a shared library, but it has often proved successful. It provides, in the form of a file to include in the makefile, a simple record of the parameters and the method used to build a particular library. First, create the file that will be included in the makefile; call it **Shared.inc**. The file should look something like:

```
SL_NAME=libxyz
SL_PATH=/usr/local/lib
SL_VERSION=1.0.0
SL_LOAD_ADDRESS=0x6a380000
SL_JUMP_TABLE_SIZE=1024
SL_GOT_SIZE=1024
SL_IMPORT=/usr/lib/libc.sa
SL_EXTRA_LIBS=/usr/lib/gcc-lib/i486-linux\
        /2.6.2/libgcc.a -lc
SHPARMS=-l$(SL_PATH)/$(SL_NAME)\
        -v$(SL_VERSION) \
        -a$(SL_LOAD_ADDRESS) \
        -j$(SL_JUMP_TABLE_SIZE) \
        -g$(SL_GOT_SIZE)
VERIFYPARMS=-l$(SL_NAME).so.$(SL_VERSION) -- \
        $(SL_NAME).sa
CC=gcc -B/usr/bin/jump
pre-shlib: $(LIBOBJECTS)
shlib-import:
        buildimport $(SL_IMPORT)
shlib:  $(LIBOBJECTS)
        mkimage $(SHPARMS) -- $(LIBOBJECTS)
$(SL_EXTRA_LIBS)
        mkstubs $(SHPARMS) -- $(SL_NAME)
        verify-shlib $(VERIFYPARMS)
```

The first section consists of a series of variable definitions. These variables have the following meanings:

SL_NAME
: The name of the library which is being built.

SL_PATH
: The location where the shared library will live.

SL_VERSION

The library version.

SL_LOAD_ADDRESS
: The absolute address in memory where the library will be loaded. (Examine the **table_description** file provided with the DLL tools to make sure this address doesn't overlap with another library).

SL_JUMP_TABLE_SIZE
: The size of the jump table. (Give this any value for the moment; an appropriate value will be determined later).

SL_GOT_SIZE
: The size of the global offset table. (Give this any value for the moment; an appropriate value will be determined later).

SL_EXTRA_LIBS
: Other libraries which are required to build the shared image.

SL_IMPORT indicates other shared libraries to import symbols from. These imported symbols are used to help direct global variable references to their proper locations in other shared libraries. The libraries specified here should be any shared libraries which are required to build the target library. The target **shlib-import** makes use of a **/bin/sh** script called **buildimport**, which is invoked with **SL_IMPORT** as a parameter. The **build import** script should contain the following commands:

```
#!/bin/sh
echo -n > $JUMP_DIR/jump.import
for lib in $*;
  do nm --no-cplus -o $lib | \
     grep '__GOT__' | sed 's/__GOT__/_/'\
      > $JUMP_DIR/jump.import
done
```

This script uses nm, grep and sed to extract the symbols from the global offset tables of each of the stub libraries specified on the command line to create a file called jump.import (the nm command sequence is excerpted from “Using DLL Tools With Linux”). Be sure to **chmod u+x buildimport**. **SL_EXTRA_LIBS** are libraries which will be required to successfully build the library. Usually most of these libraries can be determined by examining a makefile which builds an executable using this library (often there are test programs included with the source for the library). libgcc.a is required with gcc 2.6.2; if it is left out, there will be an unresolved reference for **_main**. It is usually necessary to explicitly specify libc with **-lc**. If there should be unresolved references when the library image is made, chances are that a required library was omitted.

The definition of **CC** as **gcc -B/usr/bin/jump** is telling the compiler to use an assembler called **/usr/bin/jumpas** instead of the default assembler. Be sure to check what other parameters are specified in the original makefile (and whether **CC** was defined as the compiler variable) and make additions and changes as necessary. **CC** is nearly always defined, and thus has been used in this example. If you use a version of DLL tools earlier than version 2.16, it may be necessary to specify **CC** as **gcc -B/usr/dll/jump/**.

The targets **pre-shlib** and **shlib** both have **LIBOBJECTS** as dependencies. You will probably find a list or a variable containing a list of the library dependencies in the target for the static library in the original makefile. You should define **LIBOBJECTS** as this list of dependencies, or you should replace all instances in Shared.inc with the dependencies specified for the static library. Take care when constructing a dependency list for a shared library; it is not uncommon for source code modules to be compiled even though they are not part of the *final library*. The only objects that should be compiled during the building of a shared library are those that will eventually become part of the library. If other objects are compiled, the symbols and globals used in those modules will end up in the jump configuration files for the library, and possibly in the library itself. These undesirable functions and variables may result in troublesome behavior or failure of the library build process.

In general, make sure you understand how the library object files are built. Also, make certain that the shared library objects are built using the same flags and options that were present for the original library. Now edit the library makefile (make a backup first), and add the following statement to the end of the list of makefile targets:

```
include Shared.inc
```

Finally, from the source directory of the library, do the following:

```
mkdir jump
JUMP_LIB=libxyz
export JUMP_LIB
JUMP_DIR=`pwd`/jump
export JUMP_DIR
```

These commands create a work directory for the DLL tools and assembler, and set the necessary environment variables which are required to successfully build a shared library. It will be necessary to use setenv if a csh variant is in use. Remember to replace libxyz with the name of the target library (as specified in **SL_NAME**).

Step Two: The First Compile

Before each compile remove the old **.o** files to ensure that the object code is rebuilt. Executing a **make clean** may be sufficient; however, be careful-many makefiles will remove more than the **.o** files and you may need to reconfigure the source code. Often an **rm *.o** will work more dependably.

If everything has been set up properly, it should now be possible to begin the first compile by entering:

```
make pre-shlib
```

This step compiles the library using the assembler prefixed by the **-B** switch. This will extract the necessary symbols from the library source into a file called **jump.log**. From **jump.log**, the global variables and functions will be extracted into the necessary configuration files where the DLL tools will find them. Once all the source has been compiled, change to the directory that was specified in **JUMP_DIR. Jump.log** should be in this directory. Now execute the following:

```
getvars
getfuncs
rm -f jump.log
```

These commands will create the files **jump.vars** and **jump.funcs**. **jump.vars** contains a list of the global variables found during the compile, while **jump.funcs** contains a list of functions. If, for some reason, you don't want to export a symbol found in **jump.funcs** or **jump.vars**, move the entry to a file called **jump.ignore** in the **JUMP_DIR** directory. Be sure to remove any entries added to **jump.ignore** from the original file. Now return to the compile directory.

Step Three: Importing Symbols

Now you should create the **jump.imports** file. Since a target was previously defined in **Shared.inc**, simply enter:

```
make shlib-import
```

There now should be a file called **jump.imports** in the **JUMP_DIR** directory. Nothing needs to be done with this file; it will be used to determine which global variables should be located in one of the imported libraries.

Step Four: The Second Compile

The second compile is necessary to determine the sizes of the global variables. The sizes of the globals must be known so that the GOT pointers can be set

properly. Remove the **.o** files from the previous compile and then do the following:

```
make pre-shlib
```

Now change to the **JUMP_DIR** directory and execute:

```
getsize > jump.vars-new
mv jump.vars jump.vars-old
mv jump.vars-new jump.vars
```

Step Five: Building The Library

Before actually building the shared image and stub libraries, the jump table and GOT must be allocated enough storage for all the existing functions and global variables as well as for functions or globals that may be added in revisions to the library. To determine the required number of bytes for the jump table and the GOT, execute the following:

```
wc -l $JUMP_DIR/jump.funcs
wc -l $JUMP_DIR/jump.vars
```

Multiply the resultant line counts by 8 to calculate a lower bound for the number of bytes required for existing functions and global variables, respectively. These values should be padded significantly to allow for future library expansion. Now edit **Shared.inc** and replace the settings of **SL_JUMP_TABLE_SIZE** and **SL_GOT_SIZE** with the values just determined. If you receive an overflow message while building the image, increase these values. Keep in mind that these sizes should be multiples of 8, and that the values calculated are minimums, and will probably not be sufficient to build the library image.

Now everything should be ready to actually build the shared image and stub. Without removing the **.o** files, execute:

```
make shlib
```

This will first build the image, and then the stub library. Then the stub and image will be verified to check that the libraries were built properly. If all goes well, the last message should be something like:

```
Used address range 0x6a37f020-0x6a395020 be aware! must be unique! The
stub library and the sharable libraries have identical symbols.
```

The address range indicated in the first line is somewhat misleading, since a load address of **0x6a380000**, not **0x6a37f020**, was specified. This is normal. However, make note of the last address since it indicates the last address used by the library. This address is usually padded somewhat to make sure that

room is left for expansion. The address range might be recorded as **0x6a380000-0x6a395fff** or **0x6a380000-0x6a39ffff**, depending on how much space might be required in the future.

The second line indicates that the image and stub libraries were built correctly. If the verification process should indicate that the stub and image differ, an error has occurred. Possibly one of the most common errors is when the **JUMP_LIB** environment variable and **SL_NAME** don't match. Double check that these two variables match if there should be a problem. If everything has gone correctly there should now be a stub and image library. The image should be copied to the directory specified by **SL_PATH** and the stub should be placed where it can be found by the compiler and linker. Once these files have been copied to their final directories, enter:

```
    ldconfig -v
```

There should be output similar to the following, indicating that **ldconfig** has created a symbolic link for the new library in which the name only contains the major version number. This is done because a look-up on the library is done using only the major version number.

```
    libxyz.so.1 => libxyz.so.1.0.0 (changed)
```

If **ldconfig** doesn't find the library, make sure that the directory in which the library is located is included in the list in **/etc/ld.so.conf**. It should now be possible to make use of the new library. **Shared.inc, jump.vars, jump.funcs, jump.import** and **jump.ignore** should be saved. These files will be useful if you need to rebuild the library or create a compatible replacement.

Trail's End

This article has outlined a method for creating a simple shared library from scratch. This basic method provides a starting point for understanding and constructing a shared library. Many other topics are covered and more depth is presented in “Using DLL Tools With Linux” by David Engel and Eric Youngdale. This document can be found in the **doc** directory provided with the **tools** 2.16 distribution. Information on both DLLs and ELF can also be found in the GCC FAQ, which can be retrieved via anonymous **ftp** from **www.mrc-apu.cam.ac.uk** as file **/pub/linux/GCC-FAQ**.

Eric Kasten (tigger@petroglyph.cl.msu.edu) has been a systems programmer since 1989. Presently, he is pursuing his masters in computer science at Michigan State University, where his research focuses on networking and distributed systems. Well thought out comments and questions may be emailed to him.

What's GNU

Arnold Robbins

Issue #12, April 1995

This month's column concludes the article on Plan 9 From Bell Labs, and those parts of it that have been re-implemented in freely available software.

Last month we described the origins of Plan 9, the **sam** editor, and the **9term** terminal emulator. Well, what about the shell to run inside the window? Here too, the Plan 9 authors took the opportunity to rethink the issue of just how should a shell work. The Plan 9 shell is called **rc**, because it “runs commands”.

The rc Shell

Although in many ways the Bourne shell is a simple, elegant, high-level programming language, it has a serious flaw, in that it was designed to be much like a macro-processing language. Input text is scanned, rescanned, and rescanned again, as each stage of processing is performed. (This is carried to an almost absurd length in the Korn shell, with something like eleven different processing stages.) This leads to rather complicated and baroque quoting rules, with the need for nested escape sequences.

In **rc**, the input text is scanned and parsed exactly once. The language has a real **yacc**-based grammar, making it clear what everything means. The quoting rules are very simple. Quoted text must be enclosed between single quotes. To get a single quote inside a quoted string, double it (as in FORTRAN). An explicit operator is used to provide string concatenation, and variables can be lists of strings, not just single strings.

The syntax is closer to that of C or **awk**, instead of Bourne's Algol 68. This leads to less clutter, avoiding unnecessary keywords and semi-colons. It is much more like C than the fabled **csh** is.

rc provides shell functions, and signal handlers are written as functions with special names (**sighup**, **sigterm**, etc.), instead of using strings. I/O redirection is

also more powerful, with a notation for hooking up file descriptors besides 0 and 1 to the input and output ends of a pipe.

A freely distributable clone of **rc** is available. It was written by Byron Rakitzis, and implements the language described in the **rc** paper, with some extensions. The beauty of **rc** is that it is small and fast, and shell programs can be quite elegant. It also runs on just about any kind of Unix system.

When using **rc** with **9term**, it is conventional to set the primary prompt to be just a single semi-colon, and the secondary prompt to be empty. This allows you to snarf entire commands, including the prompt, and resend them. The semi-colon is treated as a simple null statement. The use of double-clicking to select the whole line, and the default saved action of the menus make sending and resending the same line over again extremely simple; most of the work can be done with just the mouse.

The Resources sidebar lists the **ftp** location of the **rc** shell. There is also a mailing list of people who use **rc**.

The es Shell

es is the “extensible shell”. Paul Haahr and Byron Rakitzis thought it would be interesting to try and combine some of the capabilities of functional languages with those of Unix shells. Many internal capabilities of the shell (such as I/O redirection and setting up pipelines) are available as built-in functions in the language, and program fragments can be passed around as arguments to functions.

es provides first class functions, lexical scope, an exception system, and rich return values (i.e. functions can return values other than just numbers). Most of this is beyond the scope of this article to explain. **es** is described in a paper in the Winter 1993 Usenix Conference Proceedings. It helps to read this paper, and also to go through the archives of the mailing list to see how the language evolved. For the full details on **es**, you'll need to read the paper, the man page, and the file initial.es in the **es** distribution. It is a good idea to also look at the sample .esrc file, too.

Basically, the idea behind **es** is to take the primitive operations that a shell does, such as forking processes, creating pipes, and setting up I/O redirections, and make them available as functions that a user program can call directly. In turn, traditional shell syntax is built on top of these primitive operations.

Lexical scoping allows you to save the definition of an operation, and then replace it with your own operation on top of the previous one. Here is an example from the paper on **es**. This code implements a pipeline profiler. It

saves the definition of **%pipe**, which creates pipes, and provides a new one that times each component of the pipeline, using the old **%pipe** to actually create the pipeline. (**es** is the prompt from **es** used for examples in the paper. The default prompt is a semi-colon.)

```
es > let (pipe = $fn-%pipe) {
        {
                fn %pipe first out in rest {
                {
                        if (~ $#out 0) {
                                time first
                        } {
                                $pipe { time $first } $out
$in { %pipe $rest }
                        }
                }
es> cat paper9  tr -cs a-zA-Z0-9 '\012' | sort |
 uniq -c | sort | -nr  sed 6q
213 the
150 a
120 to
115 of
109 is
 96 and
 2r    0.3u   0.2s      cat paper9
 2r    0.3u   0.2s      tr -cs a-zA-Z0-9 \012
 2r    0.5u   0.2s      sort
 2r    0.4u   0.2s      uniq -c
 3r    0.2u   0.1s      sed 6q
 3r    0.6u   0.2s      sort -nr
```

This is a simple example, yet it illustrates some of the power available in **es**. **es** really deserves a column on its own. For more information, see the above sources and the mailing list archive.

The sidebar lists the **ftp** location for **es**, and a mailing list is also available.

The 9wm Window Manager

The tools we've seen so far, notably **sam** and **9term**, are built on top of X Windows, and work with any window manager. For some time, I ran them using **mwm**.

In the fall of 1993, I obtained a version of **gwm**, the Generic Window Manager, with WOOL (Windows Object Oriented Lisp) code that implemented an interface very similar to that of the original Bell Labs Blit terminal. This provides a simple, clean interface, similar to that used on Plan 9 (8½ can be considered a further evolutionary step from the Blit). This code was written by John Mackin at the University of Sydney. The resources sidebar shows where you can get this code, if you're interested. This code works, but it is large and slow.

However, a new window manager recently became available, **9wm**. **9wm** implements the window management policies of 8½, under X windows. Written by David Hogan at the University of Sydney, it uses raw Xlib (not a pretty sight),

and is completely ICCCM compliant. **9wm** is also small, and very fast. To quote from the README file:

> **9wm** is an X window manager which attempts to emulate the Plan 9 window manager 8½ as far as possible within the constraints imposed by X. It provides a simple yet comfortable user interface, without garish decorations or title-bars. Or icons. And it's click-to-type. This will not appeal to everybody, but if you're not put off yet then read on. (And don't knock it until you've tried it).

9wm is “click to type”. This means you have to move the mouse into a particular window and then click button one. That window becomes the current window, indicated by a thick black border. Other windows have a thin black border. This behavior is identical to **sam**'s.

The **9wm** menu (accessed through button 3 on the root window) consists of five items:

- **New** - open a new window (**9term** or **xterm** if no **9term**)
- **Reshape** - change the shape of a window on the screen,
- **Move** - move a window,
- **Delete** - blow away a window,
- **Hide** - “iconify” a window.

What is perhaps most noticable about **9wm** (and 8½) is that there are no icons. Instead, to remove a window from the screen, you select **Hide** from the main menu. The cursor becomes a target. You move the target to the window to be hidden, and then click button 3. Clicking any other button cancels the operation.

When a window is hidden, it disappears from the screen completely, not even leaving an icon. Instead, a new item appears at the bottom of the button 3 **9wm** menu, with the name of the window. To open the window again, you simply select the window's name from the menu.

As with the other programs, the **9wm** menu remembers what you did last time, so that the next time you pop up the menu, the previous selection is already highlighted

The 9menu Command Line Menu Program

And now, my own small contribution to the picture. The GWM Blit emulation, which I used for quite awhile, understood that it was built on top of X, and when you selected **New**, it gave you a menu of hosts (that you defined in a

configuration file) on which to start remote **xterm**s. This was nice, and I found it missing under **9wm**.

(In Plan 9, this is not an issue; the multiple hosts in the network are very tightly integrated, but in X with Unix, it is a problem.)

What I wanted was a simple program to which you could give menu items and associated commands, and this program would pop up a window that was nothing but a menu. Selecting an item would run a command. The program would be long lived, leaving the menu up permanently. A program close to this exists, **xmenu**. Unfortunately, **xmenu** goes away after executing the command, and is not well behaved when interacting with **9wm**.

Inspired by **9wm**, starting with its menu code, and with help from David Hogan, I wrote **9menu**. **9menu** pops up a window containing the list of items, and executes the corresponding command when a button is pressed.

9menu allows you to write your own menus and customize the behavior to suit you, without the headaches of a **.twmrc** or **.mwmrc** file. It is real easy to have one item spawn another **9menu**, giving a similar effect to pull-right menus.

Here are two I use it: one for remote systems, the other for programs I may want to run. Being lazy, I have **xterm** in both. I use a shell script named **rxterm** that knows about the remote hosts I will want to open windows on, and whether they can start a **9term** or an **xterm**. (This is left over from the GWM Blit code, and is mostly for convenience.) These examples are from my **.xinitrc**. The **-geometry** strings are to get **9wm** to place the windows even though they start out iconified.

```
9menu -geometry 67x136-4+477 -iconic -popdown -label Remotes \
        'solaria:rxterm solaria.cc.gatech.edu' \
        'burdell:rxterm burdell.cc.gatech.edu' \
        'chrome:rxterm chrome.cc.gatech.edu' \
        'xterm:rxterm xterm' \
        exit &
9menu -geometry 103x102-3+624 -iconic -popdown -label 'X programs' \
        'xterm:rxterm xterm' \
        xtetris xlock '9wm restart' '9wm exit' exit &
```

I start the programs using **-iconic** so that they'll be automatically hidden and part of the **9wm** menu. The **-popdown** option causes the menu to automatically iconify itself after an item is selected, since I find this to be the most convenient way for me to work: pop up the menu, select an item, and then go on with what I want to do without the menu hanging around. Although not nearly as large scale a program as **sam**, **9term**, or **9wm**, I find that **9menu** completes the package for me.

Experiences

I have been using this environment for almost two years, and find it to be clean, elegant, and easy to use. Initially, I started by using **rc**, and then **sam** when it became available in early 1993. Shortly after that, I started beta-testing **9term**, in particular getting it to work correctly under SunOS. In the fall of 1993, the GWM Blit code became available, and I switched to that, using it for almost a year. In the spring of 1994, I started beta-testing **9wm**, which was finally released at the end of 1994. I switched to **es** in January of 1993 after reading about it and hearing the presentation at the winter Usenix.

The research group at Bell Labs is well known for applying the “small is beautiful” principle to software design. This was initially true of Unix, and has been re-applied to distributed systems, shells, and user interfaces with Plan 9.

The interface is simple, consistent, easy to use, and very clean. All the programs described in this column behave the same way, in terms of what the buttons do, which window is current, and how the menus remember the previous operation.

An important point that I have not emphasized so far, is that all the programs use pop-up menus. I find this to be an enormous convenience, particularly compared to systems like Windows or the Macintosh, where you must move the mouse to the menu bar to pull down a particular menu. Pop-up menus save an incredible amount of otherwise useless mouse motion, leading to a system that is much easier to use.

My first exposure to window systems was long ago, on a Blit terminal. The interface was simple, clean and elegant. Ever since then, I had been searching for an X windows environment that matched the Blit's elegance. Now, with the combination of **sam**, **9term**, **9wm** and **rc** or **es**, I feel that I have finally found that environment, and I'm very happy. What's even nicer is that all of these programs are fast, and I have the broad range of X applications available to me also (**xoj**, anyone?). This latter point is unfortunately not true of the only other alternative, **mgr** (which I used until **9term** became available.

Using These Programs Under Linux

All the programs described here can be made to compile under Linux. I don't have a Linux system of my own (believe or not!), but for a while I borrowed one, and was able to bring up all of these programs. Unfortunately, the system was a laptop, with too small a screen to make using X worthwhile. **sam** comes up fine, using the **Make.solaris** makefile as a starting point. **9wm** also compiled just fine. **9term** took a little bit of work, but it did compile and run. After asking on the mailing lists, I learned that **9term** does not (yet) work quite correctly under

Linux. This may be fixed by the time you read this column, though. Two people to contact for information about porting **9term** to Linux are Pete Fenelon (pete@minster.york.ac.uk), and Markus Friedl (msfriedl@faui01.informatik.uni-erlangen.de). **rc** and **es**, both compile and run under Linux, but with some work. For **rc**, you have to generate the **sigmsgs.c** file by hand, based on **/usr/src/linux/include/sys/signal.h**. There is one other bug, reported by Jeremy Fitzhardinge, which is that **rc** uses **int**s for the array of additional groups, while Linux uses **gid_t**s, which are **short**s. **es** requires similar changes for the signal handling, but these are actually documented in the Makefile.

Summary

The combination of **9term** and **9wm** provides a very close emulation of the elegant Plan 9 user interface. **sam** is a powerful, easy to use editor. **rc** is a simple, clean shell, and **es** is a nifty shell with lots of promise. It is worth reading the papers describing each of these components. The complete combination proves once again that "small is beautiful."

Acknowledgements

Thanks to Chris Siebenmann and Daniel Ehrlich, maintainers of the various mailing lists, for their help, as well as to the members of the lists who responded to my questions about Linux. Thanks to Bob Flandrena, Paul Haahr, and Miriam Robbins for their comments.

References

- Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, "Plan 9 from Bell Labs", *Proc. of the Summer 1990 UKUUG Conf.*, London, July, 1990, pp. 1-9.
- Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, "Plan 9, A Distributed System", *Proc. of the Spring 1991 EurOpen Conf.*, Troms, May, 1991, pp. 43-50.
- Rob Pike, "The Text Editor **sam**", *Software—Practice and Experience*, November 1987, Vol. 17, #11, pp. 133-153.
- Rob Pike, "8½, the Plan 9 Window System", *Proc. of the Summer 1991* Usenix Conf., Nashville, June 1991, pp. 257-265.
- Tom Duff, "Rc—A Shell for Plan 9 and UNIX Systems", *Proc. of the Summer 1990 UKUUG Conf.*, London, July, 1990, pp. 21-33.

These papers are all available in Postscript as part of the Plan 9 documentation.

- Paul Haahr and Byron Rakitzis, "Es: A shell with higher-order functions", *Proceedings of the Winter 1993 Usenix Conf.*, January 1993, pp. 53-62.

This paper is available for **ftp** along with the **es** source code.

Arnold Robbins (arnold@gnu.ai.mit.edu) is a professional programmer and semi-professional author. He has been doing volunteer work for the GNU project since 1987 and working with Unix and Unix-like systems since 1981. Questions and/or comments about this column can be addressed to the author via postal mail c/o *Linux Journal*, or via e-mail.

Archive Index Issue Table of Contents

Advanced search

Cooking with Linux: Amsterdam on Fifty Guilders a Day

Matt Welsh

Issue #12, April 1995

If you happen to be visiting Amsterdam for business, leisure, or, say, and International Linux Symposium, these important travel tips might come in handy.

Ah, Europe. There's no place like it. Except for, maybe, certain neighborhoods in Chicago, but there it's easier to get through customs. At any rate, not long ago I had the privilege to speak at the International Linux Symposium in Amsterdam, which is a small town in upstate New York. Ha ha! Just kidding. I'm talking about the version of Amsterdam found in the Netherlands, of course, which is a large city with a lot of canals and people who speak Dutch. When I first heard about the conference, I was certainly interested but perhaps not very optimistic about my ability to find a few thousand dollars worth of change between the sofa cushions. Then something dawned on me: would I *ever* have another legitimate excuse to visit Amsterdam? Probably not, so I managed to pull off the travel expenses and hop on a plane to the fair city of canals. En route I even managed to pull together a talk about applications porting, but I had a lot of help from the flight crew.

Well, needless to say, I had such a wonderful time in Amsterdam that I feel it's my moral duty to share with you the many varied pieces of travelling wisdom that I accumulated during the week-long adventure. Otherwise you might run into the same problems that I did, while I was there, and let me tell you up front that it's not much fun to be yelled at (in Dutch, no less) by a very angry waiter wielding a fork and knife. With the handy tidbits contained in this article, you should be able to handle the situation with ease. (Hint: whenever possible, use the Dutch phrase *Hoe laat opent het zwembad?* I have no idea what it means, but it seems to calm people down.)

Amsterdam is a nice place to go if you enjoy being run over. Michael K. Johnson, *LJ*'s illustrious editor, and I learned this the hard way as soon as we stepped out of the RAI train station from the airport. There we were, trying to get our

bearings, taking in the countryside, observing the locals, etc., standing in what *appeared* to be an innocent walkway. Walkway? Fat chance. This was none other than the *bicycle lane*, which is widely regarded as one of the most dangerous places on the face of the Earth to stand with a map and five large pieces of luggage. Later, we discovered that standing in the bike lane is a capital crime in the Netherlands—that is, if you don't get killed in the act.

So, there we were, looking like your canonical tourists, when a moped screamed by doing 80 KPH (that's 3 million miles per hour, for those of you in the US), nearly knocking Michael into the road, which is a lot like the bike lane except that the vehicles there are much more deadly. In fact, the crosswalks in Amsterdam have lights marked with the international symbols for “Don't Walk” and “Good Luck”. Crossing the street is always an adventure, and well worth the price of the trip alone, in case you're into near-death experiences.

All cab drivers in Amsterdam are certified maniacs. After discovering the dangers of being a pedestrian (the Dutch word for which is, literally, “bumper fodder”) I wanted to see what it was like on the other side of the wheel and opted for travel by taxicab whenever possible. The first thing to realize here is that you can't hail a cab in Amsterdam, but you can call for one. Another big surprise is that the Dutch have perfected teleportation technology, demonstrated by the fact that no later than thirty seconds after you hang up the phone, a cab will materialize in front of you. This is certainly convenient, especially when you happen to call from your hotel room.

I was sure that I'd feel safer taking the cab, somehow lulled into a false sense of security by the fact that I'd be riding in the comfort of a six-ton shuddering hunk of metal with an alert, safety-conscious driver at the wheel. Wrong again. Taxicab drivers in Amsterdam are required, by law, to scare the hell out of their passengers at least twice during the trip. Tipping the driver didn't seem to help, either. But what a thrill! Humming along at 120 KPH, weaving in and out of heavy traffic, surviving at least a dozen near-misses with unwary pedestrians (ha!) and other vehicles (including streetcars), with a driver who's more involved in finishing his cigarette and staring out of the side window while changing the radio than, say, holding onto the steering wheel. I don't know about you, but this is what I call excitement. I was so impressed that I was alive and in one piece after the trip that I paid the driver twice the amount of the actual fare. So taking the taxi in Amsterdam gets high marks in my book.

Be very, very careful when ordering food. Although nearly everyone I met in Amsterdam spoke at least seven languages, including English, this fact didn't seem to help at the local pubs and restaurants. I would routinely enter such an eating establishment and be presented with a menu listing such enticing items as *hutspot met klapstuk*, *uitsmijter*, and *pijptabak*. Being completely ignorant of

the local language and custom, I stuck to the more obvious choices such as *koffie*, *broodje*, *pannekoek*, and *bier*-the last of which turned out to be a reliable default. But I do recommend being brave and ordering entrees at random. You might end up with a wonderful culinary delight such as a piece of white toast, as I did. Before you know it you'll learn what not to order.

Another thing: don't let waitresses at Indonesian restaurants constantly serve you their fun 'n' fruity mixed drinks during the course of the meal, because they'll turn out to be ten guilders a pop. One caveat with this approach: after several of aforementioned drinks, you may not notice them being served. Be on your guard at all times!

The red light district isn't for the faint of heart. Unless, of course, you're into places with names such as the Banana Bar, but I need not go into detail here. On the other hand, the red light district was the home of the most wonderful Spanish restaurant, which we, the teeming mob of mad Linuxers, took over and held all of the chefs for ransom. Just kidding! We were actually served a delightful five-course meal without the necessity of holding the head waiter at gunpoint. At least I think it was five courses-I stopped counting after three. Just imagine the scene: a restaurant full of nearly a hundred Linux users, including such shining figureheads as Remy Card, Fred van Kempen, and Linus Torvalds himself, chowing down on enough Spanish food to feed the armada, having a riotous good time discussing the pros and cons of, say, multiple filesystem block sizes. The amount of geekiness in that room was so thick you could cut it with a knife. I had to duck out routinely for a breath of fresh air-but alas, I was trapped in the back alleys of Amsterdam's most visceral neighborhood. Bon appetit!

You can find the finest coffee in the world in Amsterdam. Really! Scattered all over town are many gourmet coffee shops, wherein the dark brew is served in large quantities. If you think that's interesting, you'll really enjoy the lavish style of these bistros, with their foil-covered ceilings, subdued lighting, and pictures of Bob Marley adorning the walls. Very bizarre. While I won't pretend to understand the Dutch taste in coffee shop decor, I will vouch for the quality of coffee served within-something you won't want to miss while in Amsterdam.

Well, there you have it. I hope that the above advice will help you to get around Amsterdam in better shape than I did. Oh, and one last bit of guidance for your trip: No matter what anybody tells you, it really is fun to get lost amongst narrow, winding streets with names like Leiuweijkstraat and Geifen von vijkeslaan at three o'clock in the morning. Try it some time. You'll thank me later.

Matt Welsh (mdw@sunsite.unc.edu) is a writer and programmer at Cornell University, working with the Linux Documentation Project.

Archive Index Issue Table of Contents

Advanced search

The ELF Object File Format: Introduction

Eric Youngdale

Issue #12, April 1995

The Executable and Linking Format (ELF) has been a popular topic lately. People wonder why the kernel configuations script asks whether or not to donfigure loading ELF executables. As ELF will eventually be the common object file format for Linux binaries, it is appropriate to document it a bit. This month, Eric introduces us to ELF, and next month he will give us a guided tour of real ELF files.

Now that we are on the verge of a public release of ELF file format compilers and utilities, it is a logical time to explain the differences between **a.out** and ELF, and discuss how they will be visible to the user. As long as I am at it, I will also guide you on a tour of the internals of the ELF file format and show you how it works. I realize that Linux users range from people brand new to Unix to people who have used Unix systems for years—for this reason I will start with a fairly basic explanation which may be of little use to the more experienced users, because I would like this article to be useful in some way to as many people as possible.

People often ask why we are bothering with a new file format. A couple reasons come to mind—first, the current shared libraries can be somewhat cumbersome to build, especially for large packages such as the X Window System that span multiple directories. Second, the current **a.out** shared library scheme does not support the **dlopen()** function, which allows you to tell the dynamic loader to load additional shared libraries. Why ELF? The Unix community seems to be standardizing this file format; various implementations of SVr4 such as MIPS, Solaris, Unixware currently use ELF; SCO will reportedly switch to ELF in the near future; and there are rumors of other vendors switching to ELF. One interesting sidenote—Windows NT uses a file format based upon the COFF file format, the SVr3 file format that the Unix community is abandoning in favor of ELF.

Let us start at the beginning. Users will generally encounter three types of ELF files—**.o** files, regular executables, and shared libraries. While all of these files serve different purposes, their internal structure files are quite similar. Thus we can begin with a general description, and proceed to a discussion of the specifics of the three file types. Next month, I will demonstrate the use of the readelf program, which can be used to display and interpret various portions of ELF files.

One universal concept among all different ELF file types (and also **a.out** and many other executable file formats) is the notion of a section. This concept is important enough to spend some time explaining. Simply put, a section is a collection of information of a similar type. Each section represents a portion of the file. For example, executable code is always placed in a section known as **.text**; all data variables initialized by the user are placed in a section known as **.data**; and uninitialized data is placed in a section known as **.bss**

In principle, one could devise an executable file format where everything is jumbled together—MS-DOS binaries come to mind. But dividing executables into sections has important advantages. For example, once you have loaded the executable portions of an executable into memory, these memory locations need not change. (In principle, program executable code could modify itself, but this is considered to be extremely poor programming practice.) On modern machine architectures, the memory manager can mark portions of memory read-only, such that any attempt to modify a read-only memory location results in the program dying and dumping core. Thus, instead of merely saying that we do not expect a particular memory location to change, we can specify that any attempt to modify a read-only memory location is a fatal error indicating a bug in the application. That being said, typically you cannot individually set the read-only status for each byte of memory—instead you can individually set the protections of regions of memory known as pages. On the i386 architecture the page size is 4096 bytes—thus you could indicate that addresses **0-4095** are read-only, and bytes **4096** and up are writable, for example.

Given that we want all executable portions of an executable in read-only memory and all modifiable locations of memory (such as variables) in writable memory, it turns out to be most efficient to group all of the executable portions of an executable into one section of memory (the **.text** section), and all modifiable data areas together into another area of memory (henceforth known as the **.data** section).

A further distinction is made between data variables the user has initialized and data variables the user has not initialized. If the user has not specified the initial value of a variable, there is no sense wasting space in the executable file to store the value. Thus, initialized variables are grouped into the **.data** section,

and uninitialized variables are grouped into the **.bss** section, which is special because it doesn't take up space in the file—it only tells how much space is needed for uninitialized variables.

When you ask the kernel to load and run an executable, it starts by looking at the image header for clues about how to load the image. It locates the **.text** section within the executable, loads it into the appropriate portions of memory, and marks these pages as read-only. It then locates the **.data** section in the executable and loads it into the user's address space, this time in read-write memory. Finally, it finds the location and size of the **.bss** section from the image header, and adds the appropriate pages of memory to the user's address space. Even though the user has not specified the initial values of variables placed in **.bss**, by convention the kernel will initialize all of this memory to zero.

Typically each **a.out** or ELF file also includes a symbol table. This contains a list of all of the symbols (program entry points, addresses of variables, etc.) that are defined or referenced within the file, the address associated with the symbol, and some kind of tag indicating the type of the symbol. In an **a.out** file, this is more or less the extent of the information present; as we shall see later, ELF files have considerably more information. In some cases, the symbol tables can be removed with the strip utility. The advantage is that the executable is smaller once stripped, but you lose the ability to debug the stripped binary. With **a.out** it is always possible to remove the symbol table from a file, but with ELF you typically need some symbolic information in the file for the program to load and run. Thus, in the case of ELF, the strip program will remove a portion of the symbol table, but it will never remove all of the symbol table.

Finally, we need to discuss the concept of relocations. Let us say you compile a simple “hello world” program:

```
main( )
{
       printf("Hello World\n");
}
```

The compiler generates an object file which contains a reference to the function **printf** . Since we have not defined this symbol, it is an external reference. The executable code for this function will contain an instruction to call **printf**, but in the object code we do not yet know the actual location to call to perform this function. The assembler notices that the function **printf** is external, and it generates a relocation, which contains several components. First, it contains an index into the symbol table—this way, we know which symbol is being referenced. Second, it contains an offset into the **.text** section, which refers to the address of the operand of the call instructions. Finally, it contains a tag which indicates what type of relocation is actually present. When you link this file, the linker walks through the relocations, looks up the final address of the

external function **printf**, then patches this address back into the operand of the call instruction so the call instruction now points to the actual function **print**.

a.out executables have no relocations. The kernel loader cannot resolve any symbols and will reject any attempt to run such a binary. An **a.out** object file will of course have relocations, but the linker must be able to fully resolve these to generate a usable executable.

So far everything I have described applies to both **a.out** and ELF. Now I will enumerate the shortcomings of **a.out** so that it is more clear why we would want to switch to ELF.

First, the header of an **a.out** file (struct exec, defined in **/usr/include/linux/a.out.h**) contains limited information. It only allows the above-described sections to exist and does not directly support any additional sections. Second, it contains only the sizes of the various sections, but does not directly specify the offsets within the file where the section starts. Thus the linker and the kernel loader have some unwritten understanding about where the various sections start within a file. Finally, there is no built-in shared library support—**a.out** was developed before shared library technology was developed, so implementations of shared libraries based on **a.out** must abuse and misuse some of the existing sections in order to accomplish the tasks required.

About 6 months ago, the default file format switched from ZMAGIC to QMAGIC files. Both of these are **a.out** formats, and the only real difference is the different set of unwritten understandings between the linker and kernel. Both formats of executable have a 32 byte header at the start of the file, but with ZMAGIC the **.text** section starts at byte offset 1024, while with QMAGIC the **.text** section starts at the beginning of the file and includes the header. Thus ZMAGIC wastes disk space, but, more importantly, the 1024 byte offset used with ZMAGIC makes efficient buffer cache management within the kernel more difficult. With a QMAGIC binary, the mapping from the file offset to the block representing a given page of memory is more natural, and should allow for some performance enhancements in the kernel. ELF binaries are also formatted in a natural way that is compatible with possible future changes to the buffer cache.

I have said that shared library support in **a.out** is lacking—while this is true, it is not impossible to design shared library implementations that work with **a.out**. The current Linux shared libraries are certainly one example; another example is SunOS-style shared libraries which are currently used by BSD-*du-jour*. SunOS-style shared libraries contain a lot of the same concepts as ELF shared libraries, but ELF allows us to discard some of the really silly hacks that were required to piggyback a shared library implementation onto **a.out**.

Before we go into our hands-on description of how ELF works, it would be worthwhile to spend a little time discussing some general concepts related to shared libraries. Then when we start to pick apart an ELF file, it will be easier to see what is going on.

First, I should explain a little bit about what a shared library is; a surprising number of people look at shared libraries as sort of black boxes without a good understanding of what goes on inside. Most users are at least aware of the fact that if they mess up their shared libraries, the system can become nearly unusable. This leads most people to treat them with a certain reverence.

If we step back a little bit, we recall that non-shared libraries (also known as static libraries) contain useful procedures that programs might wish to make use of. Thus the programmer does not need to do everything from scratch, but can use a set of standard well-defined functions. This allows the programmer to be more productive. Unfortunately, when you link against a static library, the linker must extract all library functions you require and make them part of your executable, which can make it rather large.

The idea behind a shared library is that you would somehow take the contents of the static library (not literally the contents, but usually something generated from the same source tree), and pre-link it into some kind of special executable. When you link your program against the shared library, the linker merely makes note of the fact that you are calling a function in a shared library, so it does not extract any executable code from the shared library. Instead, the linker adds instructions to the executable which tell the startup code in your executable that some shared libraries are also required, so when you run your program, the kernel starts by inserting the executable into your address space, but once your program starts up, all of these shared libraries are also added to your address space. Obviously some mechanism must be present for making sure that when your program calls a function in the shared library, it actually branches to the correct location within the shared library. I will be discussing the mechanics of this for ELF in a little bit.

More info about ELF

Now that we have explained shared libraries, we can start to discuss some of the general concepts related to how shared libraries are implemented under ELF. To begin with, ELF shared libraries are position independent. This means that you can load them more or less anywhere in memory, and they will work. The current **a.out** shared libraries are known as fixed address libraries: each library has one specific address where it must be loaded to work, and it would be foolish to try to load it anywhere else. ELF shared libraries achieve their position independence in a couple of ways. The main difference is that you

should compile everything you want to insert into the shared library with the compiler switch **-fPIC**. This tells the compiler to generate code that is designed to be position independent, and it avoids referencing data by absolute address as much as possible.

Position independence does not come without a cost, however. When you compile something to be PIC, the compiler reserves one machine register (**%ebx** on the i386) to point to the start of a special table known as the global offset table (or GOT for short). That this register is reserved means that the compiler will have less flexibility in optimizing code, and this means that it will take longer to do the same job. Fortunately, our benchmark indicates that for most normal programs the drop in performance is less than 3% for a worst case, and in many cases much less than this.

Another ELF feature is that its shared libraries resolve symbols and externals at run time. This is done using a symbol table and a list of relocations which must be performed before the image can start to execute. While this sounds like it could be slow, a number of optimizations built into ELF make it fairly fast. I should mention that when you compile PIC code into a shared library, there are generally very few relocations, one more reason why the performance impact is not of great concern. Technically, it is possible to generate a shared library from code that was not compiled with **-fPIC**, but an incredible number of relocations would need to be performed before the shared library was usable, another reason why **-fPIC** is important.

When you reference global data within a shared library, the assembly code cannot simply load the value from memory the way you would do with non-PIC code. If you tried this, the code would not be position independent and a relocation would be associated with the instruction where you were attempting to load the value from the variable. Instead, the compiler/assembler/linker create the GOT, which is nothing more than a table of pointers, one pointer for each global variable defined or referenced in the shared library. Each time the library needs to reference a given variable, it first loads the address of the variable from the GOT (remember that the address of the GOT is always stored in **%ebx** so we only need an offset into the GOT). Once we have this, we can dereference it to obtain the actual value. The advantage of doing it this way is that to establish the address of a global variable, we need to store the address in only one place, and hence we need only one relocation per global variable.

We must do something similar for functions. It is critical that we allow the user to redefine functions which might be in the shared library, and if the user does, we want to force the shared library to always use the version the user defined and never use the version of the function in the shared library. Since the function could conceivably be used lots of times within a shared library, we use

something known as the procedure linkage table (or PLT) to reference all functions. In a sense this is nothing more than a fancy name for a jumptable, an array of jump instructions, one for each function that you might need to go to. Thus if a particular function is called from thousands of locations within the shared library, control will always pass through one jump instruction. This way, you need only one relocation to determine which version of a given function is actually called, and from the standpoint of performance this is about as good as you are going to get.

Next month, we will use this information to dissect real ELF files, explaining specifics about the ELF file format.

Eric Youngdale Eric Youngdale has worked with Linux for over two years, and has been active in kernel development. He also developed the current Linux shared libraries.

Archive Index Issue Table of Contents

Advanced search

Mr. Torvalds Goes to Washington

Kurt Reisler

Issue #12, April 1995

Linux Torvalds will be speaking at eh Spring US DECUS Symposium in Washington D.C. this May.

Once again, the US Chapter of DECUS, the DEC Users Group, is bringing Linus Torvalds, famous for his work with Linux (and for feeding Australian penguins) to the Spring US DECUS Symposium. This event is scheduled for the week of May 6-11 at the convention center in Washington DC.

Unlike what we did at the Spring `94 Symposium in New Orleans (where Linus gave a couple of technical talks), we are planning an entire day of Linux-related sessions on Wednesday, and a half-day seminar by Linus himself on Thursday. The Linux stream and seminar are going to be specially priced to make them easier to attend. In addition, we are going to have several Linux systems available in the UNIX SIG Campground, as well exhibits from a variety of Linux vendors.

Why should you be interested in this conference? Well, Linux does run on a large number of the PC platforms that Digital sells and supports. In addition, Digital (and Linus) are working on a port of Linux to the Alpha AXP architecture (imagine your Linux system running at 300+ MIPS). Digital has announced that an advance developer's kit (ADK) for the Digital Alpha PC is available on gatekeeper.pa.dec.com. One of the scheduled sessions deals with that porting effort.

Why else? Well, this is a rare opportunity to see and listen to Linus Torvalds, outside of Europe and Down-Under. Combine that with the rest of what is being offered at the Spring 95 US DECUS Symposium, and you have a conference well worth attending.

For additional information on the Spring `95 DECUS US Symposium is available in the US DECUS home page at www.decus.org, or request a registration kit by

sending e-mail to information@decus.org, or giving the DECUS office a call at 1-800-DECUS-55.

Kurt Reisler (klr@umbc.edu) is the Chair of the DECUS UNIX SIG, the captain of the UNISIG International Luge Team, and a collector of teddy bears. Although he has been running Linux for only a year, he has been involved with UNIX for the past 18 years.

Archive Index Issue Table of Contents

Advanced search

A review of InfoMagic's December 1994 Release

Caleb Epstein

Issue #12, April 1995

The three CD-ROMs are packed full of up-to-date Linux distributions, documentation, source code and even a “live” file system you can run your system from.

You can find everything you need, whether you're using or installing Linux for the first time or you know the ropes like a veteran. For the proto-Linuxer, the distribution comes complete with a 28-page, CD-sized, *Quick Start Guide* which is based on Matt Welsh's Linux Installation HOWTO. The three CD-ROMs are packed full of up-to-date Linux distributions, documentation, source code and even a “live” file system you can run your system from. The aptly named “Developer's Resource” is a great value.

InfoMagic delivers three CD-ROMs full of Linux and Linux-related software with their latest offering, the December-pressed *Linux Developer's Resource*.

What You Get

The first disc comprises the on-line documentation (HOWTOs), DOS utilities, and InfoMagic's large collection of Linux distributions. They provide JE-0.9.3 (Japanese Extensions to Linux), MCC-1.0+, Slackware-2.1.0, SLS-1.06, TAMU-1.0D, and BOGUS-1.0.1. Some of these distributions aren't as up-to-date as others, particularly the smaller ones such as MCC and TAMU, but that is simply because they haven't been updated recently, not because InfoMagic is shipping stale software. Other directories on this disc contain a DOS-based installation program and a copy of Microsoft's Multimedia Viewer with the HOWTOs compiled just for it.

On the second disc you'll find mirrors of the SunSite Linux FTP archive (ftp://sunsite.unc.edu/pub/Linux), Alan Cox's Linux networking archive (ftp://sunacm.swan.ac.uk/pub/misc/Linux/Networking), and the “live” file system, which is a fully unpacked copy of the Slackware 2.1.0 distribution. In theory,

and with a suitably fast CD-ROM drive, you should be able to run off of the “live” file system on this disc and a small root partition on your hard drive. Since InfoMagic can't know which packages you wish to use from the CD, it is up to you to set up the “link farm” on your root partition which points to directories on the disc (i.e. **ln -s /cdrom/live/usr/local /usr/local**).

The last disc contains copies of the tsx-11 Linux FTP archive (ftp://tsx-11.mit.edu/pub/linux), the official Linux kernel archive up to and including kernel version 1.1.72 (ftp://ftp.cs.helsinki.fi/pub/Software/Linux/Kernel) and the Free Software Foundation FTP archive (ftp://prep.ai.mit.edu/pub/gnu). To avoid keeping redundant copies of the XFree86 X Window System with both the tsx-11 and SunSite directories, InfoMagic has chosen to split this large component out into its own directory. Releases 2.1.0, 2.1.1, and 3.1 are available.

Also on the last disc are the Debian-0.91 distribution; Japanese HOWTO documents; the Wine Microsoft Windows emulator archive; a Scheme interpreter; and the Oberon system, an object oriented programming and operating environment. There are also some demos of commercial products: the Unix Cockpit, an X-based file manager and Executor, a Macintosh emulator which runs under Linux (or doesn't - Executor doesn't seem to work under 1.1.x Linux kernels, which prevented me from testing it). There is also FlagShip, a CA/Clipper-like development system, and a self-described “early demo” of a commercial BBS system for Linux, called Zbbs.

This distribution has everything you need, whether you're installing Linux for the first time or you know the ropes like a veteran and want to update your system. There's plenty of documentation in all sorts of formats, from the easily printed to the easily browsed. The handy *Quick Start Guide* is an excellent primer for the novice Linux user. It covers such important topics as device naming, drive partitioning, and file system creation. When coupled with InfoMagic's wide array of Linux distributions, you've got everything you need to install Linux on your PC.

I find that distributions like this make for excellent emergency backup media. They contain recent copies of all of the major Linux distributions (pick your favorite) as well as more recent stuff which you can cull from the Linux and GNU sources on discs 2 and 3 and compile for yourself. If you're interested at all in Unix or are a code junkie like myself, this is the package for you.

Using the Discs

I've been using Linux for about two years and have it installed on my PC at home. I've got two hard drives on my system, an IDE drive where I keep all of my DOS and Windows stuff, and a larger SCSI drive that I use for Linux. To test

drive the Developer's Resource, I decided that I'd clear out two non-essential partitions from my Linux disk and install a couple of different distributions in their stead. I backed up the old partitions with my tape drive and got down to work.

I had enough room for a single 185 MB partition on which I could install a distribution. With this amount of disk space I knew I could install plenty of software, but I'd need to pick and choose to some degree because the distributions I was looking at are pretty large.

Because I'd tried it once before with an earlier InfoMagic release, I decided to first install Slackware on my newly-unified ext2fs partition. I figured I'd probably be able to get a full system up and running in an hour or so. I wasn't disappointed. I was able to put together a boot/rootdisk combination for my system in about five or ten minutes, most of this time taken up by the writing of the disk images to floppies. I popped in the boot floppy, rebooted, and got down to business. Using the **colortty** rootdisk, I was greeted with nice looking color dialog boxes which make the installation procedure look very professional.

Slackware's **setup** routine found the existing swap and ext2fs file systems on my SCSI hard drive. I told it to ignore the ones I didn't want to touch (the **/, /usr,** and **/usr/local** of my pre-existing system) and to use the new partition I had set up as its root file system. My MS-DOS file systems were also found and I added these to the file system table without a hitch, even though I wanted them mounted as **/dos/c** , **/dos/d** , and **/dos/e** which is a bit out of the ordinary.

I then installed all of the packages I was interested in (just about all of them) and rebooted. My system came up right away and I was able to log in as **root** and add myself as a user to the system. One thing I noticed after the installation was that my new partition was almost completely full. I installed almost every package in the Slackware distribution and was left with something shy of 10 MB free space on my 185 MB partition. A minimal installation - no TeX, games, etc.—would be a good deal smaller, but 185 MB is obviously not quite enough room for the whole of Slackware to fit comfortably.

One minor complaint I have about Slackware is its treatment of manual pages. They are stored in compressed, pre-formatted form (i.e. in **/var/catman** instead of **/usr/man**), which saves disk space but limits your flexibility. I like having the manual page sources around, so I can format the output for viewing on a terminal, or an X display, or turn them into DVI or PostScript files. With only the pre-formatted pages at your disposal, you're stuck with ASCII (or ISO-8859-1) and a baroque system whereby underlining is denoted by a combination of underscores and backspaces interspersed between the characters. Yuck.

Because this installation went so easily and was so trouble-free for me, I thought I should try out some other distributions and compare them to what I felt was an extremely polished and professional Slackware release. I didn't run it for long, since I wanted to check out BOGUS too.

It's Not BOGUS

Having read a bit about the BOGUS Linux distribution when it was announced, I decided to try it out next. The BOGUS distribution is maintained by Rik Faith, Doug Hoffman, and Kevin Martin and is billed as hacker-centric system for experienced Linux users. The BOGUS installation process requires more of a hands-on approach than does the Slackware one, but the extra work involved is not difficult and should come easily to anyone familiar with administering Unix or Linux systems. The handy little QuickStart guide is also a good tutorial for this sort of thing.

The README file for the BOGUS 1.0 distribution (additional files and docs to upgrade from 1.0 to 1.0.1 are also on this disc) is pretty short and doesn't do much hand holding. You're instructed to partition your drive for root, swap, user, and some optional partitions but not told how to do so. This might frighten away the casual user but will have the hacker licking his or her chops. You might also refer to the QuickStart guide as it provides an excellent tutorial on the process. I would try it with my large root partition and my pre-existing swap and see what happen.

After rebooting, I found the design of Kevin Martin's boot floppies intriguing. As with most distributions, the boot disk loads into a RAM disk so that it runs quickly and doesn't wear out your floppy drive. The contents of the second floppy are also loaded into the 4 MB RAM disk by typing **get_files** once the system has finished booting. Once the second disk has been loaded, you've got everything you need to set up and install your new system (or recover from a crash) and you're free to use the floppy drive for other things. This isn't for low-end systems, though, as the documentation says that 12 to 16 MB RAM are required.

BOGUS relies heavily on a utility called **pms** , Rik Faith's excellent Package Management System, to do most of the installation work for the rest of the distribution. A script called **/usr/src/install.all** is used to install the BOGUS packages on your system. This is simply a shell script with a bunch of calls to **pms** in it.

The **pms** program looks for its package files in a directory called **/usr/src/DIST** . Seeing as I was working with limited disk space, I decided to fool BOGUS by pointing a symbolic link at the directory on the CD containing the 1.0 distribution files and let the install script chug away. It worked like a charm. The

process ran to completion but somewhere in one of the last couple of packages I had run out of disk space. It turns out that the 185 MB I had budgeted is not enough for a complete BOGUS distribution. Had I read all of the documentation beforehand, I would have found that it requires about 205 MB in toto.

The packages which hadn't been installed were completely non-essential (some X-based games as it turned out) so I wasn't too worried. I did a bit of snooping around and found that **pms** keeps a log of all of the packages which have been installed in the directory **/var/adm/pms** . For each package there is a time stamp file with information about when that package was installed. For the packages being installed when my disk ran out of space, the timestamp files were empty. When I figured this out, it was a simple matter of using **pms -d** to remove all traces of the semi-installed packages. I made even more room by deleting more stuff I knew I wouldn't be using (TeX-related, mostly).

The 1.0.1 release of BOGUS requires you either to install it on top of an existing BOGUS-1.0 distribution or to overwrite outdated package files in the distribution directory with their newer counterparts before running **install.all** . I realized I'd wasted time installing some of the packages from 1.0, but this was only after the installation was complete and I'd done the cleanup described above.

Changing my symlink to point to the new packages in the **ADDITIONS** directory, I set about upgrading my new BOGUS-1.0 system to 1.0.1. This was also done by means of a **pms** wrapper script and some direction from the README file. I had freed up enough disk space earlier to allow the upgrade to go smoothly without filling up my disk.

I now rebooted and was greeted with the BOGUS boot sequence. For those keeping score at home, BOGUS seems to give a nod to the BSD camp, particularly where the boot procedure is concerned. The system start-up scripts are called **/etc/rc.*** instead of the **/etc/rc.d/*** you'll find most other places. The messages reminded me of Sun's messages.

I found a minor bug: **/bin/passwd** wasn't setuid root. This meant that a normal user couldn't change his or her password. BOGUS also seems to lack any batch commands for adding users. I couldn't find any, but I might not have looked in the right places. The **pms** tool is, in my opinion, excellent. It is used to build the system binaries from the 95 MB of sources and patches, all of which are available on the disc, and to install the resulting packages on your system when it is done. A very impressive piece of work.

Despite the Spartan nature of the installation instructions and the lack of a nifty user interface, BOGUS is an extremely full-featured and well-rounded

distribution. The emphasis is definitely on the software developer who has a powerful system and not the casual user. Inasmuch as InfoMagic calls this a “Developer's Resource”, I think there should be a wide audience for BOGUS.

Caleb Epstein (epstein_caleb@jpmorgan.com) lives in Brooklyn and works at J.P. Morgan in New York City. He has been using Linux at home for over two years, and is looking for other Linux users in the area to join him for a not-so-virtual beer at the local micro.

Archive Index Issue Table of Contents

Advanced search

Xfig

Robert A. Dalrymple

Issue #12, April 1995

Need figures and charts? Robert shows us how to get here with a Linux system. Guess how he did the figures for this article?

Need a schematic diagram for a paper? How about a floorplan of a house? A title page for a report? Or do you need to add text to an existing PostScript figure? Xfig can do all that. Xfig (facility for interactive generation of figures under X11) is a drawing program that provides a powerful (forget Windows Paintbrush!) tool to get the look you want.

Figure 1

Typing **xfig** brings up a window with a variety of panels. You make (and edit) drawings with the tools along the left side of the window as shown in Figure 1. The top half of the column is the drawing modes panel. It includes tools for creating two types of circles (starting at the center - the left choice - or starting with a point on the circle), ellipses (same options), a variety of splines (that go through your points or near control points), arcs, line segments, open and closed figures and text. You can also import PostScript figures to embed in your drawing. Any figures that you make can then be moved, scaled, flipped, copied to other parts of the drawing, or rotated using the editing mode panel.

To make a circle, click your cursor on the left circle tool in the drawing mode panel. Now move to the drawing canvas and click again. This places the center of the circle at that point. Now move the cursor in any direction until the growing circle on the canvas is the size you want. Click the left mouse button again to fix the circle in place. If you don't like it, click the right mouse button and the circle disappears, allowing you to start over. (Or you can select the undo button on the top line of the window.) To pick a different shape, click on another drawing mode tool. Most of the other drawing tools work about the same way.

If you are the least bit unhappy with your figure, edit it using the editing mode panel. Individual objects can be moved by use move mode: click the left mouse button on the object, move the cursor to the new location and click again. (The right mouse button cancels the move.) Clicking on some types of objects brings up an edit panel, which allows for micro-adjustments to the shape and characteristics of the object. Figure 2 shows the edit panel that corresponds to the upper blue box of Figure 1.

Figure 2

The box is blue, as is the lower box, but the intensity is 60%, while the lower box is 95%. Using the edit panel, the color could be changed to seven others (counting black and white) and the intensities can be varied. The point box shows the x and y coordinates. Each value may be changed by editing the boxes. For other shapes, such as a polyline (made of a connected series of line segments), individual points can be moved, subtracted or added to change the shape of the line.

Text fonts, line thicknesses, and colors all can be changed before drawing an object by using the indicator panel at the bottom of the Xfig window. These buttons change according to the mode you have chosen. One neat feature is the smart-links mode used when moving objects. Lines connecting boxes in your figure expand or shrink with the movement of a box, keeping everything connected. This helps when you want to move things in flow and organizational charts.

The man pages for Xfig serve as a complete user's manual, providing much more detail than I have here. You can print them with **man -t,xfig| lpr -Plp** , with the - **t** , providing a formatting appropriate for a PostScript printer named **lp** . Besides describing all the features, the man page provides details about changing the default parameters. I aliased **xfig** to **xfig -P -e ps -startf 16** , so that my default export parameters are portrait rather than landscape on the PostScript formatted page, and the font size starts up at 16, instead of the 12 point default size.

Xfig will export your drawing in a variety of formats, such as PostScript, Latex (and PicTex), X11 bitmap (xpm), PIC and HPGL, for printing or including into a document - in color and with the fonts you want. You can do the exporting from within Xfig or via postprocessing using **fig2dev** , which comes with Xfig.

fig2dev -L ps NAME.fig NAME.ps converts *NAME.fig* to a PostScript file, *NAME.ps.* The other valid graphics language (-L) options are box, epic, eepic, eepicemu, ibmgl, latex, null, pic, pictex, ps, pstex, pstex_t, textyl, and tpic.

To add extra flourishes to your drawing, trying using xpaint along with xfig. Xpaint is a simple-to-use paint program, written by David Koblas, that will import xbm (X11 bitmap) from xfig and save the result in a variety of formats, including PostScript. Figure 3 gives an example of effects that you can add to Xfig-generated drawings with Xpaint.

Figure 3

Xpaint comes up with a toolbox filled with a variety of painting tools. The file button opens a new canvas, retrieves an old canvas, or imports a figure. Once the canvas is open, a palette is presented for colors and patterns at the bottom. Getting around **xpaint** is very simple and a credit to its designer.

Getting Xfig sidebar

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

Archive Index Issue Table of Contents

Advanced search

A Quarter Century of Unix

Danny Yee

Issue #12, April 1995

Salus has chosen and edited his source material well, however, and inserted his own summary and exposition in appropriate places. The result is a great read, with the voices of the various creators providing unique perspectives on the events they participated in (some scores of people are quoted from at length).

- Author: **Peter H. Salus**
- Publisher: **Addison-Wesley 1994**
- ISBN: **0-201-54777-5**
- Reviewer: **Danny Yee**
- Summary: **From Space Travel to Plan 9 and Linux**

A Quarter Century of Unix is a history of Unix, a kind of annotated collection of reminiscences. It begins at the “birth” of Unix, with Ken Thompson looking for a machine to play Space Travel on, then jumps back to provide the context, both in the history of computing in general and in the particular setup at Bell Labs. Part two describes the work done up to 1974, both on Unix and on the tools and language (C) so closely associated with it. Part three tries to pin down some of the things that made Unix unique: its style, the strong contributions by users and user groups, and the key role of some of its more famous tools. Parts four and five trace the expansion of Unix: the development of BSD and the commercial Unixes, the creation of SUN, the ambivalent relationship with DEC, legal issues and attempts at standardization. The final section offers an overview of the current status of Unix in its many different versions and offers some ideas about where it is heading. There is also a very brief glance at some of the systems that it has influenced, including Bell Lab's new Plan 9 system. The finale has Dennis Ritchie, Brian Kernighan and others offering their ideas on what made Unix work. Particularly noteworthy is the solid treatment of legal issues (three chapters altogether) and the coverage of events outside the United States (in Australia, Europe and Japan).

The format of *A Quarter Century of Unix*, with most of the text in the form of extended quotations, runs the risk of discontinuity and lack of focus. Salus has chosen and edited his source material well, however, and inserted his own summary and exposition in appropriate places. The result is a great read, with the voices of the various creators providing unique perspectives on the events they participated in (some scores of people are quoted from at length).

I did spot a few minor inconsistencies in the text—on page 155 we read “It was 32V that became 3BSD in 1979”, though the Unix versions tree on page 61 shows no such influence—and errors—on page 253 we have “It was clear that AT&T hadn't objected to other derivatives: Linux, MINIX, etc. In the autumn of 1988...”, implying that Linux existed in 1988 (and Linus' name is misspelled in the index, too). But these are just quibbles. A more weighty criticism would be that the book sometimes reads more like myth than history, with the participants portrayed like epic heroes. (It's rather obvious that Salus himself is a Unix fan.) This may worry the historians, but in a way it is the legends and myths that are the most influential, so the distinction is perhaps moot.

A Quarter Century of Unix doesn't assume specialized knowledge, but the more you know about Unix (and to a lesser extent, about architectures and operating systems) the more you will get out of it—if you've never used awk, for example, you will probably have little interest in reading about its origins and development. The main audience will be programmers, administrators, and users with extensive Unix experience. Historians and sociologists of the computer industry will find Salus' work an essential source of primary material, and marketing types might well learn a thing or to from it. *A Quarter Century of Unix* should be a great success; it's just unfortunate that it wasn't written years ago!

Archive Index Issue Table of Contents

Advanced search

The Mosaic Handbook for the X Window System

Morgan Hall

Issue #12, April 1995

I suspect that the Mosaic Handbook for the X Window System was written more for completeness than for a clearly defined audience.

- Authors: **Dougherty, Komen, and Fergeson**
- Publishers: **O'Reilly & Associates**
- ISBN: **1-56592-695-3**
- Reviewer: **Morgan Hall**

Why read a book review? Perhaps the most fundamental reason is to judge whether or not to spend time or money, or both. With this in mind, a reviewer's responsibility is to judge whether or not a particular book is worth the time or effort, or to whom a book would be worthwhile. Perhaps this bit of philosophical musing may alert you to the fact that my feelings are mixed about this particular book.

I suspect that the Mosaic Handbook for the X Window System was written more for completeness than for a clearly defined audience. The general tone and approach are more suited to an MS-Windows or Mac user than to the typical Linux user. The lack of Linux software on the packaged CD-ROM further supports this suspicion. We're a strange breed, a mixture of knowledge and naivety, and probably not easy to characterize.

Let's look at what the book contains, then see who would most benefit from it.

The Mosaic Handbook for the X Window System is a “trade paperback”--the familiar soft cover binding we know well. Inside the back cover is a CD-ROM containing software for Digital, Hewlett Packard, IBM AIX, Silicon Graphics, and Sun machines. Notable by its absence are binaries for Linux or any other “PC-Unix”, such as BSD386 or SCO UNIX.

The book itself starts with an explanation of the internet, the services available on the internet, how client-server software works, and a short history of the World Wide Web (WWW from here on). In addition, it explains why O'Reilly and Associates developed the Global Network Navigator and their view of the net and where it will develop.

Chapter two is concerned with the Mosaic program itself. It asserts that only a SLIP or PPP connection can run mosaic over a dialup line (no, the book never mentions TERM). A quick explanation of how to start up Mosaic, and the book sends the reader straight to O'Reilly's Global Network Navigator to learn the basics of Mosaic. The last half of chapter two is where the beginner to Mosaic can really learn how to use the program.

Chapters three and four are mainly concerned with using Mosaic to prowl the net. Chapter three introduces the reader to various parts of the web; chapter four concentrates on other services, such as archie, WAIS, news, FTP, and telnet.

Chapters five and six are concerned with Mosaic, the program. Chapter five covers customizing Mosaic; chapter six deals with Mosaic and multimedia.

Chapter seven is a brief (and quite useful) introduction to creating simple documents with HTML. It explains how hypertext works, the basic structure of simple hypertext documents, and the minimal set of tags that a new HTML author needs to get started. Serious exploration will quickly go beyond the scope of this explanation, but it's a good start for someone who's totally new to the game. Also, in chapter seven is a brief explanation of an HTML editor and syntax checker, HoTMetaL. I haven't yet tried to find a Linux implementation of it, but it looks like a useful tool. Chasing this goes on the To Do list....

Chapter eight looks toward the future. It asks (and tries to answer, in part), “Where is the Web going?” An interview (that originally appeared in GNN NetNews) with MIT's Michael Dertouzos discusses the evolving WWW standards, the W3O project from CERN.

Finally, the book concludes with four appendices: A is the Mosaic Reference Guide, B is the HTML Reference Guide, C is the list of X resources used by Mosaic, and D is concerned with installing Mosaic. Appendix D emphasizes the CD-ROM supplied with the book, but also mentions obtaining copies from the net and building from source code.

Having not read the companion volumes for Microsoft Windows and the Macintosh, I can only speculate that most of the content is the same. However, the general tone and level of detail make it almost certain.

The Mosaic Handbook for the X Window System is a well-written, informative book. However, it is not targeted at the Linux community. In my opinion, the users most likely to get maximum use from this book will be new users who are approaching the net for the first time. Linux users will have to exercise their network skills to get source or binaries (sunsite has both normal and term-aware copies of Mosaic). I'd recommend borrowing a copy to find the nuggets of information that it contains, but can't, in good conscience recommend that you run right out and buy a copy.

Archive Index Issue Table of Contents

Advanced search

Letters to the Editor

Various

Issue #12, April 1995

Readers sound off.

Linked Lists

I have been using Linux for a couple of years now, and when I saw your magazine I immediately subscribed. I love it.

Only ONE downside.

...continued on page 46 ...continued on page 44

What's the story with this format? I've got to find it one of the most annoying techniques that magazines use, and cannot personally figure out a reason for it. Skipping a page because of a full page ad is fine-I even scan the ad if it looks interesting. The Economist, a weekly magazine that calls itself a newspaper and runs over 100 pages, does NOT do this and I find it much easier to follow the content and spirit of what the author is trying to get across. Everyone else is breaking up their articles into linked lists-a big drag.

What motivates you to make reading your excellent magazine a chore?

—Graham R. Leach g_leach@pavo.concordia.ca

LJ Responds:

Only one downside? That's pretty good...

We do it only as much as is strictly necessary. Several things determine the need to do this. The first is that we do not impose exact article length limits on our authors, which means that the articles don't come out anywhere near even pages much of the time, and the numbers of advertisements of different sizes may not fit the empty spaces. The second is that we have decided that all one

page or more articles must start at the top of a page; we tried not doing that and it looked awful. Another consideration is to start “important” articles near the front of the magazine. Readers expect to find them there but that means that something must be continued to the back half of the magazine. The last consideration is that some articles require color, and we have only so many color pages to work with per issue. The magazine is made up of a number of 16-page “forms” or “signatures” and, in some cases, color is only present on one side of the form. This means that we have to link through the color pages for ads and articles that contain color and then print the non-color material on the other pages. Right now, printing in all color is not affordable. As our press run increases the cost of this additional color becomes less significant. While this will not totally eliminate the “linked list” syndrome it will help decrease its frequency.

When we have more subscribers and advertisers, and can afford to expand the magazine, our options for laying out each article will be widened. Our goal is no articles ever split; although this will probably never be completely possible, we hope to come closer over time.

Linux in Costa Rica

I want to tell how Linux has helped supporting Costa Rica's national network.

We've installed some Linux name and mail servers in the main subdomains of our national Internet networks. Linux is now stable enough for doing administrative chores like nameserving, mail serving and the like.

So CRNet (which is the entity coordinating and giving impulse to our national network) likes the idea of using Linux for these tasks.

In fact, places like the Presidential House, the Universidad Latina (Latin University) and the Veterinary School of the Universidad Nacional use Linux both as name and mail servers.

In the University of Costa Rica (UCR), where I'm working, we are setting up a Linux box as a temporary and limited newsserver.

—Mario A. Guerra mguerra@cariari.ucr.ac.cr

(Un)supported

I recently decided to purchase a “real” Linux Distribution. My prime motivation was the fact that I couldn't seem to get my very unsupported Procom CDROM mounted. Don't feel bad, it didn't work under (IN)Coherent, Oh Esse Too, or

even well under Windoze. Procom tech support told me simply “We no longer support those old drives”, *click...buzz*. So I was getting desperate.

I had tried under my old system (Linux 1.0.9), by using the drivers there. I was under the assumption that most of these drives are similar, just VAR-ed under different labels. I thought this might be a Mitsumi. Anyway, I broke down and ordered Release 4 and a supplement from Trans-Ameritech, mostly because they claimed to work under any drive DOS could recognise.

Not only did T-A send the Distribution Release 4 and the July Supplement, they also included a free copy of the November supplement with Debian and Bogus and BSD4.4R2.0. (at no extra cost to me!) Thanks T-A. Unfortunately, the only scheme I could run my disc from was through UMSDOS running from my true DOS partition. I didn't have the room, nor did I want to. I already had a root partition that I wanted. So, I gave up.

Then, on Martin Luther King Day, I was goofing with the system and I decided, on a lark, to do a raw scan of the DOS binary drivers for the Procom drive. I don't know what I was looking for, but I found it-at the tail of the file was the list of some Sony 500-series drives! I knew this latest distribution had some cdu535 stuff on it, so I forced a boot from loadlin (a story in itself) and (TaDa!) I could talk to the disc! But, I went from PL 1.1.18 back down to 1.0.9 . I scanned the distribution again and found a 1.1.18 kernel with cdu535 support and setup installed that nicely! Happy ending! Drop in if you're in the neighborhood and we'll split a Guiness. Have fun, I know I am. Thanks for doing what the big boys couldn't do!

—Mike Allison be381@freenet.hsc.colorado.edu

More Suggestions

I've just gotten my Linux setup working and got a subscription to *LJ*. I was happily overwhelmed by the depth of information your magazine offers. On that note, I have a few comments/suggestions.

1. Since it appears that Linux is achieving some sort of mass acceptance, it may be in *LJ*'s best interest to appeal to many types of users. I know as much as the next programmer about DOS, more than most about OS/2, but very little about Linux. And I turn to your magazine for help. Unfortunately, I find very little information for the beginning Linux'er. Maybe a beginners column would help? How about a series of articles that covers installation considerations, tips, setup help, and a list of the FAQs and where to get them?

2. I believe that most experienced *nix users expect new users to understand how multi-user systems work. They forget that the “I've outgrown Windows” crowd will be coming onboard and will expect to have their hands held and for their installation routines to handle all the crucial details. My marketing background makes me keenly aware of how first impressions make or break a sale. And you can bet your last dime that Microsoft and IBM will be sucking in new users at a record pace in 1995. So, if you were to include some new user information in your magazine, I'm sure you'd capture a few of the wanna-be's.

Thanks for your time.

—Chris Freyer cfreyer@gate.net

_LJ_ Responds:

1. Part of the problem has been finding authors interested in writing beginning material. We now have several authors interested in this, and more beginning articles will start showing up. Keep your eyes peeled.

2. The first impression means a lot. However, we can't beat the MS marketing machine at its own job. Instead, I think that Linux is and will be for those who have become dissatisfied with MS and (to a lesser extent) IBM. I'm not going to bet that Linux will ever blow MS out of the market. Instead, I'd like it to be the best possible thing for those who are frustrated with the alternatives.

Archive Index Issue Table of Contents

Advanced search

Linux Installation and X-Windows

Dean Oisboid

Issue #12, April 1995

If you are a seasoned hacker—someone who can code a device driver with the left hand, install a 200-node network with the right, all while reciting the pinouts of a parallel port-then skip this article.

Welcome all novice Linuxers. If you are a seasoned hacker—someone who can code a device driver with the left hand, install a 200-node network with the right, all while reciting the pinouts of a parallel port-then skip this article. Novice to Novice aims directly at the PC expert who knows nothing about Linux or Unix, yet is curious about the uproar. As I am a complete Unix beginner myself, hopefully my naive, embarrassing mistakes will help many readers bypass much suffering and ridicule. This series will follow as I try to reach goals appropriate, I think, for the PC user who wants to learn Linux, yet doesn't want to modify his/her existing setup. Getting Linux to run on a DOS disk using UMSDOS is the obvious first goal. That, and getting X-Windows to run, are the topics of this article.

Definitions

For the record, my system is a 486DX-66 with SVGA, dual floppies, Logitech rodent, 14,400 baud modem, and a 350 meg hard drive, of which the bulk is filled with games. I have managed to clean off 150 megs for Linux, which the various manuals and ads suggest is more than enough room. Both MS-DOS 6.2 and Windows 3.11 are installed (and frequently yelled at).

Pick a card. . . .

Despite Unix being one of the longest-lived operating systems, it still carries a mystique, one that has kept its popularity from growing amongst the general public. Unix seems to exist only in the darkest realms of academia, engineering, and high-tech graphics. As a MS-DOS user for too many years, I decided to finally learn Unix, and with that decision, I faced some choices: local schools

offered evening courses for a few hundred dollars a pop; I could buy “personal” versions of most of the big Unix products for still only a few hundred dollars; or I could get—what was this?—some free version of Unix called Linux.

Being a miser, I chose Linux. My friend, David Coons, who is some sort of computer guru for Disney's Imagineering, heartily recommended Morse Telecommunication's Slackware Professional 2.1. He had bought a variety of releases and liked this one. He happily explained why, but since the language he used consisted primarily of acronyms, I just pursed my lips knowingly and nodded my head at the right pauses. My criterion was far simpler: I figured anything with a picture of “Bob” on it to be worthwhile. That this version would allow me to run Linux without repartitioning my hard drive sold me. UMSDOS seemed like a godsend.

I called ACC Bookstore, where I ordered the Slackware set as well as the massive DrX Linux book. The order-taker reassured me that Linux-particularly Slackware-was a great way to learn Unix and not that difficult to figure out, at least on the basic level.

Novice Note: When you buy Linux, see what type of information comes in the enclosed manual. Slackware Professional comes with a 600-page book that duplicates all the vital information in DrX Linux and is better organized. If I had known, I wouldn't have ordered DrX Linux, since the non-duplicated material includes things like “The Kernel Hackers Guide” and the “Japanese Language Extension HOW-TO”. It will be a long while before I understand Linux well enough to start hacking the kernel, and by then, the manual will be outdated. For now, you may want to save your money.

After ordering Linux, I went out and bought a CD-ROM player. I had debated for months about whether to buy a quad-speed (since I already have an original 8-bit SoundBlaster) or to buy the Creative Labs Discovery Package which would upgrade my soundcard to 16-bits but give me a less desirable double-speed CD-ROM. Again, the miser won out and I got the Discovery kit. Not a bad deal, I rationalized, since the rumors were that high-density CDs would start to appear in a year or two, and I would rather be obsolete as cheaply as possible. Also, Linux appears to have solid support for Creative Labs' products. While installing the multimedia kit, I switched my floppy drives so that the 3.5" became my **A:** and the 5.25" became **B:**. David had highly recommended doing this.

I got the CD drive, got Linux, got the manuals, and started screaming. All the manuals suggested repartitioning using FIPS and were of no help regarding UMSDOS-just vague references. Even Doom is listed in the Slackware index but nothing about UMSDOS. I didn't want to repartition! A call to David and he

reassured me that the options would be obvious on what to do when I ran **setup**, after first creating the boot and root disks. Not to panic. Okay, thanks!

Ordinarily, I wouldn't have minded repartitioning. The program to do so, FIPS, a non-data-destructive FDISK, is a brilliant and obvious (are you listening, Microsoft?) utility, but I worried that I might FIPS too much or too little disk space and possibly even kill my MS-DOS programs. UMSDOS is what I wanted, a no-commitment option.

Taking a deep breath and chanting the “Doom” mantra, I began. Following the installation instructions (I hate to RTFM), the first step was to prepare boot and root disks. Since I had a SoundBlaster system, I assumed the SBPCD boot image would be the obvious choice. For the root disk, I opted for UMSDS144, which was the mythological UMSDOS system for 1.44 disks.

With these disks created, I booted with the boot disk. It didn't recognize my CD player. Hmmmm. Of course! The Discovery Kit used a Sony CDU-33a drive. I re-made the boot disk this time choosing the CDU31a option. Yes! Rebooting with this new disk showed Linux recognizing my drive.

I ignored the option to set boot parameters, put in the root disk and stalled at the first hurdle. Did I want a swapfile on my hard drive? I have eight megs of RAM and the manuals said with that amount of RAM not too worry, so I just press **<Enter>** and got a Login prompt. A feeling of lordly omnipotence washed over me as I smugly logged in as **root** and ran setup.

A menu came up and the panic started again. Tags, swapspace-what's with all the choices? I finally figured out that the important first step for beginners is (T)arget. This will set up a **C:\LINUX** subdirectory on your hard drive and prompt you through the other necessary procedures. You will need to select a source; in my case a CD-ROM drive and specifically, the Sony CDU-33a.

Next, you get to choose which disk sets to install. I decided on all of them except the F series (FAQs and HOW-TOs). Finally, you select a method of installation, whether everything goes to hard drive (“SLAKWARE”-where you make all the decisions as to which files to install) or three choices of TAG sets which preselect which files to install: SLACKPRO (all files on the hard disk, with upgrade capability); SLAKPRO2 (some files are links to CD but without easy upgrade capability); or SLAKPRO3 (many links, again without easy upgrade capability). Links are references on your hard disk to the actual files on CD; this conserves hard disk space but gives up access speed.

For my first installation attempt, I chose the “slakware” option, so that everything would go to the hard drive. In the middle of set “X” the drive ran out

of space. Rebooting DOS and using X-Tree (which ran out of memory), I deleted the contents of **C:\LINUX** for another try. I debated whether to upgrade my drive to a full gigabyte. The mortgage was due, and as you already know, I'm a miser, so I didn't.

My second installation was “slakpro3”, which put the fewest files directly on the hard drive. This option would make later upgrading difficult but is a good exploratory choice. It used only 15 meg, produced about 3000 files, and didn't do much. Commands like **adduser** didn't work. I'm not even sure the the shell was active. Back to DOS for Linux deletion again.

The third installation I tried was to have all the files on the hard drive (“slackpro”), because I didn't know that this was larger than my previous “slakware” installation. After an hour, sixteen **thousand** files, and 150 meg, my drive again ran out of space. Okay, maybe I don't need 2 million fonts for TeX, some of the programming tools, or network stuff. Back to DOS, again. My guess is that 200 meg would handle this type of installation.

Novice Note: The “A” and “Q” disk sets both deal with installing the kernel, whether you want IDE without SCSI, IDE with SCSI, etc. You may want to do two rounds of installations: first, you would just deal with sets “A” and “Q” to find *exactly* which kernel you want. My choices narrowed down to CDU31ao (without SCSI support) or CDU31a (with SCSI support); I installed CDU31a. Your second round of installation would then cover all the other disk sets.

Fourth time, again using the “slackpro” option and ONLY installing the A, AP, D, Q (for the correct kernel), and X data sets, used 50 meg and created about 5000 files, but the instructions regarding LOADLIN didn't work. Apparently LOADLIN doesn't get copied over to the hard drive at any point. I found the LOADLIN.ZIP in the KERNELS subdirectory on the Slackware disk, unzipped it into **C:\LINUX**, and modified the given LINUX.BAT to launch it:

```
rem C:\LINUX.BAT
echo off
cls
echo Put the Slackware CD in the drive!
pause
rem  First, ensure any unwritten disk buffers are flushed:
smartdrv /C
rem  Start the LOADLIN process:
c:\linux\loadlin c:\linux\vmlinuz root=/dev/hda rw
```

IT WORKS!!! IT WORKS!!! IT WORKS!!! No more boot and root disks! The lack of LOADLIN was likely the problem with “slakpro3” not working correctly, but I won't try that out now that this setup works. I “**adduser**”ed an account for myself with no problems, the procedure being very easy, and logged in on that account via Alt-F2. This is “way cool”, having two active accounts going simultaneously.

Novice Note: Capitalization counts! I went nearly crazy trying to run a configuration program. The subdirectories were all **spelled** correctly but some of the letters had to be in capitals. I recommend installing the Mouseless Commander. (I believe it is in the AP dataset. It is now called the Midnight Commander, since it can now be used with a mouse, but Slackware still refers to it as the Mouseless Commander in some places.) It's a great Norton Commander clone and, for this X-tree user, a comfort and an easy way to view files.

Slackware

Well, it took four tries and at least as many hours, but the Linux base is on the MS-DOS partition and appears to run smoothly. I had no problems with either DOS or Windows after installing Linux; it appeared as just another subdirectory, albeit with a ton of files.

X-Windows

Now to install X-Windows. I called David for just a couple of clues. “You're on your own. I didn't install X.” I politely hung up as he explained how he was writing an amazing device driver with his left hand and installing a 200-node network with his right.

With the benefit of hindsight, I highly recommend that before you attempt to install X, you have on hand information about your monitor-specifically bandwidth, horizontal synchronization, and vertical refresh rate. The data should be in your monitor manual. If not, the Linux guides suggest looking in the files called “modeDB.txt” or “Monitors” located in **/usr/X11R6/lib/X11/doc**. Best to have the actual monitor guide. Also you should know what type of video card you have. If you have MS-DOS 6.0 or later, the MSD program will give you that information. In Linux, a program called SuperProbe will also tell you.

Installation is relatively easy, but is only half the job. Again, all this applies to Slackware Professional.

Change to the **/usr/X11R6/lib/ConfigXF** subdirectory (watch for capitalization!) and run ConfigXF. The program will first ask for information about your mouse, if you have one. I have a Logitech mouse so I selected the Microsoft option. The guides suggest this, saying that only if you have an older Logitech mouse should you choose the Logitech option. After this, I agreed with the given defaults and having **/dev/mouse** as the path.

Onward to video cards. From the massive list, I chose the Cirrus GD-5426. Next came a monitor list, and I opted for generic VESA SVGA. After this, it asks you questions regarding virtual desktop size and other things. Since I didn't

understand half of the questions, I just accepted the defaults. Eventually, you get to a screen where you can save the set up, tune the set up, quit, and other choices. For me the option to tune the set up just didn't work, producing a variety of errors.

What did work is this: saving the set up to the default choice. Edit **/usr/X11R6/lib/X11/XF86Config**, looking for the section called “Monitor”. You'll note that Bandwidth, HorizSync, and VertRefresh are marked with “EDIT THIS!!!” It took me too many times to realize that when the X-installation program said you should edit the XF86Config file that you *really* had to edit it. Now the crucial part: replace the information in the file with the information from your monitor manual.

Novice Note: Take a few minutes and learn to use the text editor **vi**. Not only is this editor ubiquitous and small, but many other programs use similar commands. For instance, the **:q** that will quit vi will also get you out of the **man** program.

For example, I entered the following for my CTX CMS-1561 Multiscan monitor:

```
Bandwidth 100
HorizSync 30-60
VertRefresh 50-90
```

In addition to these changes, I also added lines that appeared in the Linux manuals but not in the XF86Config file. They probably aren't needed, but what the heck. Under “Keyboard” I added:

```
AutoRepeat 500 5
```

Under section “Screen”, subsection “Display” I added:

```
Depth 8
```

Save the file. Start X-Windows with startx (or xstart) and, with luck, it will run. If it doesn't, get the Linux manual, skip the automatic installation altogether, and carefully do it yourself, checking that the information in the Config files match your set-up.

Whew! Linux works. X-Windows works. What next? Well, I could install the XAP disk set to give me programs to use when I am in X. Doom could finally come out of hiding. Or I could install TeX and see what those Klingon fonts look like. Or I could tackle SLIP and see if I can get working access to the Internet. Or I could even reassert my latent geekdom and write a “Hello world!” program in GNU C/C++. Linux has so much to explore-but then, that's the fun. Watch out David and all you other Unix gods-we're coming up the ladder!

Dean Oisboid (73717.2343@compuserve.com), owner of Garlic Software, is a database consultant, Unix beginner, and avowed Doom addict.

Archive Index Issue Table of Contents

Advanced search

New Products

LJ Staff

Issue #12, April 1995

ImageCraft releases HC11 C Compiler, version 1.1, Seamless Object-Oriented Software Architecture and more.

ImageCraft releases HC11 C Compiler, version 1.1

ICC11 is a high quality, low cost compiler package that runs on the Linux, OS/2 2.x, and DOS environments. Features include: interspersed C and assembly output, ability to assign different names to text and data sections, allowing better memory utilization, a comprehensive 90 page manual, a fast almost-ANSI C conformant compiler with built-in peephole optimizer, assembler, linker, and librarian, standard C header files and library functions, HC11 specific support such as embedded assembly, pragma for declaring interrupt functions, etc, and calling conventions compatible with other compilers. Technical support over the Internet is available. REXIS, an add-on multitasking executive with subsumption architecture semantics for controlling mobile robots, is also available.

Price: $45.00, plus shipping and handling. Contact: ImageCraft, P.O. Box 64226, Sunnyvale, CA 94088-4226, (408)749-0702. E-mail: imagecft@netcom.com.

Seamless Object-Oriented Software Architecture

Interactive Software Engineering, Inc. (ISE) and Prentice Hall announced the publication of *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*, by Kim Walden and Jean-Marc Nerson (ISBN 0-13-031303-3). This book covers fundamentals and advanced concepts of O-O technology. The authors draw from their extensive experience of consulting on large scale O-O projects worldwide and managing the development of reusable component libraries for large corporations.

Advanced search

LINUX JOURNAL

Setting Up Services

Mark Komarinski

Issue #12, April 1995

Over the past year, this column has focused on configuring and administering several standard Unix tools. This month, I'd like to switch to a bit of networking. Specifically, I introduce how to configure TCP/IP network services for Linux.

To start, imagine the TCP/IP connection you have (anything from an Ethernet connection to a SLIP/PPP connection from your friendly neighborhood network provider) as really 65,536 lines (each of which can handle many different *conversations* at once) coming into your machine. Some of these lines are dedicated to serving a single purpose, but a majority of them are open for use.

Each of these separate lines coming into your machine is known as a port, and each Linux machine, connected to another via TCP/IP, has 65,536 ports available on it. Each of these ports allows a connection between the two machines, assuming there is a program on one end that is listening to that port in question and a program on the other end attempting to connect to it. If a connection with the remote side is requested and a program on the remote side is *listening* to that port, a connection is established. This can be just about any kind of connection, as telnet, FTP, HTTP (World Wide Web), and SMTP (mail) all use sockets to get and receive data. Sockets are the entire combination of local-machine, local port, remote machine, and remote port that defines one of the communications channels available. The number of these channels is really only limited by available memory.

Now, there are two ways to make sure the remote side will be able to pick up on the connection. The first is to start a program running in the background all the time, waiting for a connection. This can be the easiest way to do it, but requires some programming skills and requires the program to be in memory all the time. If you choose to run a program as a daemon, and it is not used often, it will wind up being just wasted memory and get swapped to disk.

The second option is to have one program listen to ALL the ports and then, if a connection is requested, start the associated program. This is good because programs are only run when they are needed, but for programs that require loading a lot of files on startup or may be called very often, it might be too slow and waste CPU power as well.

In Linux, as in most versions of Unix, both of these options exist. Some programs (like the http daemon or sendmail/smail or the NFS daemon) run in the background and make and remove copies of themselves in memory as necessary. These programs usually can have large data files that require up to 30 seconds to load. By loading the programs once, then spawning themselves as needed, these programs can cut their startup time to only a few seconds. Other programs, such as **in.telnetd**, which handles incoming login connections, do not need as much time to load into memory and can be left out of memory until needed.

In the case of **in.telnetd** and other small programs, **inetd** (for Internet Daemon; also called the "superserver") service comes in. **inetd** watches all the ports it can, and if it sees a connection request, it checks its list to see if there is something that wants to watch that port. If there is an entry in its list, it starts the program up with input and output, both directed through the socket. Otherwise, it refuses a connection, and you see the familiar "Connection refused" on your terminal.

The way to set up the inetd program is to edit a file called /etc/inetd.conf. In this file, you may find some lines that look like figure 1.

Now let's decrypt some of this. Any line starting with a # is treated as a comment by inetd. Any other line is broken up into 6 pieces:

1) Service-This defines the port that is being watched by inetd. The name for this is in the **/etc/services** file. If you look there, you'll see lines like:

```
ftp             21/tcp
telnet          23/tcp
smtp            25/tcp          mail
```

Here are three of our favorite services defined. **ftp**, **telnet** and **smtp**. ftp uses port 21, telnet uses port 23, and smtp uses port 25. These connections are in **/etc/services** so that you don't have to remember that ftp is port 21. You just tell inetd **ftp** and it figures out the rest.

2) Socket type-This can be **stream** or **dgram** (for datagram). A stream is usually for a connection that opens for a long time, and (for every case you are likely to see) uses the tcp protocol. Telnet or FTP are great examples of this. A datagram is a small packet of data where there is no real connection and (again, for every

case you are likely to see) will use the udp protocol. Also available are raw, rdm, and seqpacket.

3) Protocol-**tcp** for streams, **udp** for dgram socket types. These types are defined in **/etc/protocols**.

4) Flags-Wait and Nowait. This is applicable only to dgram socket types. Anything else should be defined **nowait**. If a datagram socket connects to another socket and frees the socket for inetd to open another port, it is defined **nowait**. Otherwise, inetd should wait for the connection to close.

5) User[.group]-This defines which user (and group optionally) to run the following program under. It's usually root, but some programs you may want security on and run as a lower user.

6) Command line-Command (including any command line parameters) to run when inetd finds activity on that port. Almost all programs that are intended to be run from **inetd** have names starting with "**in**" to make this obvious.

You may note that you see a **/usr/bin/tcpd** in front of the programs. The **tcpd** program performs a few functions that **inetd** doesn't. For example, **tcpd** can log the connection through the syslog(3) facility (see *Linux Journal* issue 11, for a discussion of syslog), verify a hostname, find the name of the remote user that is connecting, and deny or allow services to hosts that you can specify Some of these options require re-compiling the tcpd program, but can greatly increase the security of your system. One thing you can do, without re-compiling, is limit the services available to sites known for causing you trouble. To deny telnet (and other) access to a site, create a file called **/etc/hosts.deny**. In it, you can list first the access you want to deny, a colon, then the hosts to deny that access to.

First, list the name of the program that you want to deny. This can be **in.fingerd**, **in.telnetd**, **in.ftpd**, etc. You can also use the keyword ALL to signify all services.

Next, list the hosts you want to deny access to by the following methods:

1) Network names starting with a "." will deny access to all hosts that have it as its last network name. **.clarkson.edu** will deny access from any host from Clarkson, such as **craft.camp.clarkson.edu**. A **.edu** will deny anyone in the **.edu** domain.

2) Network names that end in a "." will deny access to all hosts that have the matching string as the front portion of the network name. For example, **128.153**. will deny all of the Clarkson domain, while **128.153.16**. will deny a portion of the Clarkson domain.

3) ALL which denies access to everyone, and LOCAL which matches hosts whose resolved name does not contain a period (.). Many domain name servers will resolve a name on the local subnet to just the hostname instead of host.subnet.net. For example, craft.capm.clarkson.edu could appear to another host on the same subnet as just craft. The man pages for host_access (5) will explain more wildcards.

4) The keyword EXCEPT will exempt specific hosts who would be denied under other rules from being denied.

So a sample **/etc/hosts.deny** could look like this:

```
ALL: .clarkson.edu
EXCEPT: craft.camp.clarkson.edu
```

Which would deny all access to anyone in the Clarkson domain except for users on the machine craft.camp.clarkson.edu.

You can also set up an **/etc/hosts.allow**, following the same methods as the **/etc/hosts.deny**, except that the **hosts.allow** specifies who to specifically allow access to. In the case of a conflict between a host being denied and allowed, the entry in **/etc/hosts.allow** takes precedence, and access is allowed. To make a site more secure, you could put ALL: **ALL** in your **/etc/hosts.deny** (to deny access to everyone), then list in the **/etc/hosts.allow** all the hosts you want to allow in. This way, only the hosts you specify have access to the services that **tcpd** runs. Also, if you have *only* a hosts.allow file, and no **hosts.deny** file, *only* hosts listed in the **hosts.allow** will be allowed any access at all.

See the man pages for tcpd(8) and hosts_allow(5) for more information about how to use **tcpd** at your site.

Now, how does this all work? Let's add something to our **/etc/inetd.conf**. Something simple and easy, say a “fortune” port. Many Linux installations contain the **/usr/games/fortune** command, and a **qotd** (quote of the day) port exists at port 17. So we'll set **inetd** up so that if you **telnet** to port 17, you get the output of the fortune command. So, log into your machine as root and make sure that inetd is running. If it is not, you will want to set up TCP/IP for your machine. Even if you're not connected to anything, you can still set up the loopback device and connect to yourself.

First, make sure that qotd is defined in your /etc/services:

```
qotd    17/tcp
```

Next, we'll add the line in the **/etc/inetd.conf** to make **inetd** start fortune. This can be added anywhere in the **/etc/inetd.conf**:

```
qotd stream tcp nowait root /usr/sbin/tcpd \
        /usr/games/fortune
```

You'll have to restart **inetd** to make it re-read the **inetd.conf** file. An easy way that only works under Linux, but should *always* work under Linux, is:

```
linux:/# killall -HUP inetd
```

On some systems, the PID of the **inetd** process may be kept in a file, such as **/var/run/inetd.pid**, and on non-Linux systems without the **inetd.pid** file, you will have to use the **ps** command to find the PID of the **inetd** process.

Now if you telnet to localhost port 17, you'll find something like this:

```
linux:/# telnet localhost 17
Trying 127.0.0.1
Connected to localhost
Escape character is '^]'.
Money is the root of all wealth
Connection closed by foreign host.
linux:/#
```

There are only a few programs that you can use for this. Things that use curses, like **joe**, or anything that uses the SVGAlib, won't work, as it won't be able to open your tty (remember: to Linux, you're telnetting in from somewhere else).

Any programs you do put in your **inetd.conf** file should have good security. This means:

1) Verify (and modify if necessary) the user that the process is running under. Many need root privileges, but some don't.

2) Verify the security of the program that is being connected to a TCP/IP socket. Something like **/usr/games/fortune** is not interactive, but a program like the old **sendmail** allowed the Internet worm to wind its way through machines a few years ago. (Note that the **sendmail** bug was fixed.)

3) Add extra security to **inetd** by adding something like tcpd, which will allow you to deny or allow various hosts from connecting to your machine. Check the tcpd man pages for more information about **tcpd**.

Now that you have your services set up, you can hook in your own services and use them for whatever you want. If you have questions or comments about his article, or have some topic you would like to see in a future issue of the *Linux Journal*, please send me an e-mail note at komarimf@craft.camp.clarkson.edu.

Mark Komarinski (komarimf@craft.camp.clarkson.edu) graduated from Clarkson University (in very cold Potsdam, New York) with a degree in computer science and technical communication. He now lives in Troy, New York, and

spends much of his free time working for the Department of Veterans Affairs where he is a programmer.

Archive Index Issue Table of Contents

Advanced search