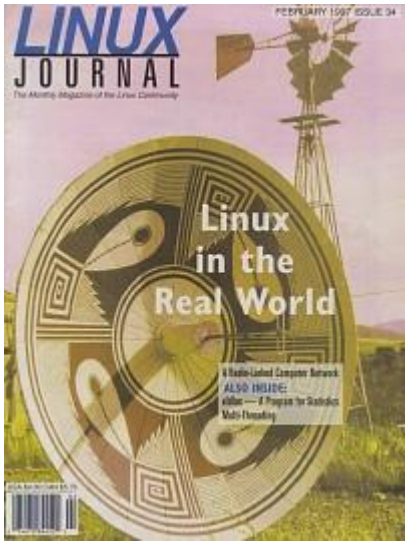
Linux Journal Issue #34/February 1997



Features

NF/Observatory Networking with Linux *by Fred Treasure*
An observatory in New Mexico uses Linux to network the computers providing remote control of its optical telescope.
xldlas - A Program for Statistics *by Thor Sigvaldason*
Unable to find a program for doing simple statistical choresthat worked on Linux, the author decided to write one.
What is Multi-Threading? *by Martin McCarthy*
A primer on multi-threading: the process whereby linux manages several tasks simultaneously.

News & Articles

A Comparison of Xemacs and GNU emacs *by Larry Ayers*
Introducing Real-Time Linux *by Michael Barabanov and Victor Yodaiken*
At Last, An X-Based vi *by Dan Wilder*
XBanner: Making XDM More Attractive *by Amit Margalit*
Graphing with lout *by Michael Hall*
WATCHDOG: The Linux Software Daemon *by Michael Meskes*

Reviews

Product Review System Commander *by Terrence Miller*
Book Review Practical UNIX and Internet Security, 2nd ed. *by Dan Wilder*

WWWsmith

At the Forge CGI Programming *by Reuven Lerner*
Writing CGI Scripts in Python *by Michel Vanaken*
CGI: Safety First *by Hans de Vreught*

Columns

Letters to the Editor
Stop the Presses DECUS and OSW *by Gary Moore and Phil Hughes*
Linux Means Business Practical Linux: A Bosnian Experience *by John Gorkos*
New Products
Linux Gazette Tips from the Graphic Muse *by Michael J Hammel*
Best of Tech Support *by Gena Shurtleff*

Archive Index

Advanced search

NF/ Observatory Networking with Linux

Fred Treasure

Issue #34, February 1997

How one observatory is using Linux to network the computers that provide remote control of its optical telescope.

This article describes the network that NF/ Observatory (NFO) uses to remotely control an automatic optical telescope. More information about the observatory can be found at Western New Mexico University's web site at http://www.wnmu.edu/nfo.

Each of the computers in the NFO network uses the Linux operating system. The four primary computers use version 1.2.13 while some of the R & D computers use version 2.0.0. Linux was chosen because of its reliability, versatility, low cost and native support for the large variety of networking types that we use. NFO uses Ethernet, Spread Spectrum wireless, ham radio and telephone modems at various points in the network.

The NF/ Ranch House

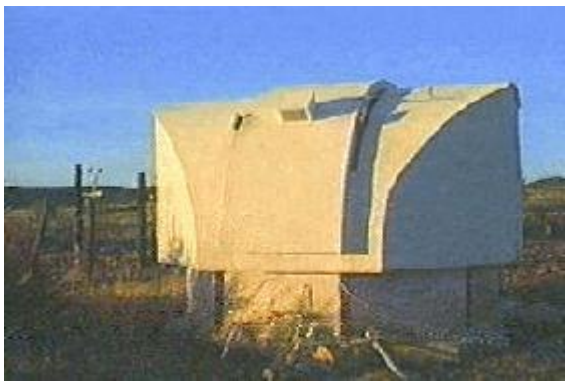
NF/ Ranch Node (scope.wnmu.edu)

This node is located at the NF/ ranch and is the one that uses the antenna shown in Figure 1. When we built the telescope in 1986 there was no telephone service to the ranch. In fact, there wasn't even commercial power to the ranch

until 1984! The three media types used here are ham radio, coaxial Ethernet and telephone modem. Ham radio has been in use for 10 years and predates Linux. The radio link receives information used to program the observing list for the telescope. It also transmits the pictures and telemetry from the telescope back to the data reduction computer and the Internet connection in Silver City.


The Digital Relay Dish at NFO

The Computer Network


The Automatic Radio Linked Telescope

The ham radio equipment consists of a Terminal Node Controller (MFJ 1270), a TAPR 9600 bps modem (http://www.tapr.org/) and a Motorola MOCOM 70 commercial FM transceiver which we modified to send and receive data. Linux communicates with the TNC via an RS-232 serial link. The relevant portion of scope's /etc/rc.d/rc.local files is:

```
#!/bin/sh
/bin/echo "Setting TNC RS232 speed to 9600"
# Also setting clocal to ignore modem control
# lines.
/bin/stty 9600 clocal < /dev/cua2
/bin/stty -a < /dev/cua2
/bin/echo "Sending commands to TNC"
sleep 1
# txdelay
/bin/echo -ne "\300\001\020\300" > /dev/cua2
# persist
/bin/echo -ne "\300\002\377\300" > /dev/cua2
```

```
# slot time
/bin/echo -ne "\300\003\004\300" > /dev/cua2
# tail
/bin/echo -ne "\300\004\004\300" > /dev/cua2
/bin/echo "Commands to TNC done."
/bin/echo "Setting port to AX25 mode."
sleep 1
/usr/local/bin/axattach -s 9600 /dev/ttyS2 KC5ZG-2
sleep 1
/usr/local/ax25/etc/axaddarp 44.30.2.130 WY5G-4
/sbin/ifconfig sl0 198.59.153.205 mtu 512
/sbin/route add 44.30.2.130 sl0
/sbin/route add default gw 44.30.2.130 sl0
```

The Ethernet hardware is an NE2000 clone card that communicates with the dedicated telescope control computer via about 100 feet of RG058 coaxial cable. The two computers communicate using the FTP protocol to move data back and forth. The telescope control computer doesn't use Linux, since it is involved in the real-time control of the CCD camera.

The telephone modem provides a backup link to town. It is rarely used, since it is a long distance call from Silver City to the ranch, and the ham radio link has been quite reliable.

Pinos Altos Mountain Node (pa.wnmu.edu)

This node is located at 8000 feet near the Continental Divide and can be reached by a jeep trail, if it hasn't snowed lately; otherwise, it is a strenuous but beautiful backpacking trip. The reliability of Linux is important here! In addition to the radio that communicates with the ranch, this site also boasts a 2 Mbs Spread Spectrum link and another ham radio link using a PI2 card instead of a TNC. The /etc/rc.d/rc.local file looks like this:

```
#! /bin/sh
# Attach link to NM2 Node Stack
/sbin/axattach -s 9600 /dev/ttyS0 WY5G-8
sleep 1
echo "Ifconfig sl0 to 198.59.153.200"
/sbin/ifconfig sl0 198.59.153.200
/sbin/ifconfig sl0 mtu 512
# configure Wavlan Spread Spectrum link.
/sbin/ifconfig eth0 198.59.153.200
echo "Adding routes"
/sbin/route -n add 44.30.2.130 sl0
/sbin/route -n add 198.59.153.205 sl0
/sbin/route -n add 192.136.110.150 gw 192.136.110.153\
        eth0
/sbin/route -n add 192.136.110.153 eth0\
/sbin/route -n add default gw 192.136.110.153 eth0
echo "Configuring PI2 Card Port A"
/sbin/ifconfig pi0a 198.59.153.200
/sbin/ifconfig pi0a hw ax25 WY5G-8
/sbin/ifconfig pi0a broadcast 198.59.153.255
/sbin/ifconfig pi0a netmask 255.255.255.0
/sbin/ifconfig pi0a arp mtu 512 up
/pi2/piconfig pi0a speed 9600 txdelay 250\
        persist 255 squelch 10 slot 1
echo "Configuring PI2 Card Port B"
/sbin/ifconfig pi0b 44.30.2.137
/sbin/ifconfig pi0b hw ax25 WY5G7
/sbin/ifconfig pi0b broadcast 44.30.2.255
/sbin/ifconfig pi0b netmask 255.255.255.0
/sbin/ifconfig pi0b arp mtu 512 up
```

```
/sbin/ifconfig pi0b 44.30.2.137 hw ax25 WY5G-7 up
/pi2/piconfig pi0b speed 1200
/sbin/route -/
- add 192.136.110.151 sl0
# /sbin/route -n add 192.136.110.151 gw 44.30.2.136 sl0
/bin/axaddarp 198.59.153.205 kc5zg-2
```

One point of interest in the above file is that the commands normally sent to configure the TNC are missing. In this case, the TNC is configured by its internal X1J Node software, which also sets the TNC's serial port to communicate in ax25 mode instead of the more usual nrs mode. This configuration allows the Linux computer and the X1J Node to share one transceiver, a Motorola MITREK modified for data transmission.

The Wavelan interface looks like an Ethernet card to Linux. It is configured by an append line in the /etc/lilo.conf, like this:

```
# LILO configuration file
# generated by "liloconfig"
#
# Start LILO global section
boot = /dev/hda
# compact and faster, but won't work on all
# systems.
delay = 50
vga = normal  # force sane state
ramdisk = 0   # paranoia setting
# End LILO global section
# Linux bootable partition config ends
image = /zImage.wav
root = /dev/hda2
label = wavelan
append = "ether=0,0x390,0x5280,eth0"
read-only
```

The Wavelan interface is experimental, and doesn't work well enough over the seven mile path to the WNMU Node to be used as our primary link. With improved antennas we expect it to be an excellent high speed link.

The other experimental interface uses a PI2 card to control a radio link on another frequency. So far this link hasn't been used, since it interferes with a nearby ham radio voice repeater.

WNMU Node (www.wnmu.edu)

The WNMU Node is NFO's connection to the Internet. The eth0 interface is an NE2000 card connected to Western New Mexico University's system. The eth1 interface is the Wavelan card that talks to both the Pinos Altos Node and to the David's Basement Node. Wavelan works very well over the one mile path to David's Basement. The WNMU Node is unique in that it must act as the proxy server for the computers not directly connected to WNMU's system. Check the arp settings in the following rc.local file:

```
#! /bin/sh
echo "setting ttyS2 to irq 5"
/bin/setserial /dev/ttyS2 irq 5
```

```
echo "sending commands to TNC"
sleep 1
/bin/stty 9600 clocal < /dev/cua2
/bin/echo -ne "\300\001\025\300" > /dev/cua2
/bin/echo -ne "\300\002\377\300" > /dev/cua2
/bin/echo -ne "\300\003\004\300" > /dev/cua2
/bin/echo -ne "\300\004\004\300" %gt; /dev/cua2
/bin/echo "Commands to TNC done..."
#echo "setting /dev/ttyS3 to irq 11"
#/bin/setserial /dev/ttyS3 irq 11
/sbin/ifconfig eth0 192.136.110.153
/sbin/ifconfig eth1 192.136.110.153
echo "Starting WWW Server"
/etc/httpd
echo "Attaching AX25 link to Radio Port"
/sbin/axattach -s 9600 /dev/ttyS2 KC5ZG-1
sleep 1
/sbin/ifconfig sl0 192.136.110.153
/sbin/ifconfig sl0 mtu 512
/sbin/route -n add 192.136.110.150 eth1
/sbin/route -n add 192.136.110.158\
        gw 192.136.110.150 eth1
/sbin/route -n add 192.136.110.159\
        gw 192.136.110.150 eth1
/sbin/route -n add 192.136.110.170\
        gw 192.136.110.150 eth1
/sbin/route -n add 192.136.110.152\
        gw 192.136.110.150 eth1
/sbin/route -n add 192.136.110.128 eth0
/sbin/route -n add default gw 192.136.110.128 eth0
/sbin/route -n add 192.136.110.3 eth0
/sbin/route -n add 44.30.2.130 sl0
/sbin/route -n add 44.30.2.136 sl0
/sbin/route -n add 44.30.2.151 gw 44.30.2.130\
        sl0
/sbin/route -n add 192.136.110.151 gw 44.30.2.130\
        sl0
/sbin/route -n add 192.136.110.154 gw 44.30.2.130\
        sl0
/sbin/route -n add 198.59.153.200 eth1
/sbin/route -n add 198.59.153.205 gw 44.30.2.130\
        sl0
/sbin/route -n add 192.136.110.152\
        gw 192.136.110.150 eth1
/sbin/route -n add 192.136.110.150 eth1
/sbin/route -n add 192.136.110.155 gw 44.30.2.130\
        sl0
/sbin/route -n add 192.136.110.156 gw 44.30.2.130\
        sl0
/sbin/route -n add 44.30.2.145 gw 44.30.2.130 sl0
echo "Clearing stale file locks"
rm /etc/mtab~
rm /nos/spool/mail/*.lck
rm /nos/spool/mqueue/*.lck
rm /nos/spool/*.lck
echo "Publishing wnmu arp entries"
/sbin/arp -s 198.59.153.200 00:c0:df:46:b1:b6 pub
/sbin/arp -s 198.59.153.205 00:c0:df:46:b1:b6 pub
/sbin/arp -s 192.136.110.150 00:c0:df:46:b1:b6 pub
```

The ham radio equipment for this node is similar to that used at the NF/ Ranch Node with the exception the the transceiver which is a TEKK data radio. In its spare time www.wnmu.edu also acts as the web server for the University and NFO.

David's Basement Node (astro.wnmu.edu)

This node is located in the basement of a Victorian mansion in downtown Silver City. It is the mail server for the observatory and connects the Wavlan part of the network to a coaxial cable Ethernet that is the LAN for the computers we

use for teaching an advanced astronomy class. These computers are located near the astro mansion. The data reduction computer is also on the Ethernet LAN and is in a house around the corner from the astro mansion. rc.local looks like this:

```
#! /bin/sh
/sbin/ifconfig eth1 192.136.110.150
echo "Adding routes"
#/sbin/route -n add 44.30.2.147 sl0
#/sbin/route -n add 44.30.2.146 gw 44.30.2.147 sl0
/sbin/route -n add 192.136.110.153 eth1
/sbin/route -n add default gw 192.136.110.153 eth1
/sbin/route -n add 192.136.110.152 eth0
/sbin/route -n add 192.136.110.158 eth0
/sbin/route -n add 192.136.110.159 eth0
/sbin/arp -s 192.136.110.152 00:40:95:26:76:fb
/sbin/arp -s 192.136.110.158 00:40:95:26:77:ab
echo "Publishing wnmu arp entries"
/sbin/arp -s 192.136.110.1 00:40:95:14:ea:41 pub
/sbin/arp -s 192.136.110.3 00:40:95:14:ea:41 pub
/sbin/arp -s 192.136.110.4 00:40:95:14:ea:41 pub
/sbin/arp -s 192.136.110.5 00:40:95:14:ea:41 pub
/sbin/arp -s 192.136.110.6 00:40:95:14:ea:41 pub
/sbin/arp -s 192.136.110.150 00:40:95:14:ea:41 pub
/sbin/arp -s 192.136.110.156 00:40:95:14:ea:41 pub
/sbin/arp -s 192.136.110.151 00:40:95:14:ea:41 pub
/sbin/arp -s 192.136.110.154 00:40:95:14:ea:41 pub
/sbin/arp -s 192.136.110.153 00:40:95:14:ea:41 pub
/sbin/arp -s 192.136.110.7 00:40:95:14:ea:41 pub
/sbin/route add 198.59.153.200 gw 192.136.110.153\
        eth1
/sbin/route add 198.59.153.205 gw 192.136.110.153\
        eth1
```

How the NF/Observatory Got Its Name

I Hope I've provided enough detail in this article to help others set up their own wide area network. Some of the ham radio information will be useful only to licensed amateur radio operators, but the Spread Spectrum devices are available to everyone.

Fred Treasure is an escaped Physicist. He used to work for Johns Hopkins University / Applied Physics Laboratory but now enjoys living in Silver City, New Mexico with the former Barbara Hobbs and their two sons. In his spare time he likes to build computer networks.

Archive Index Issue Table of Contents

Advanced search

xldlas—A Program for Statistics

Thor Sigvaldason

Issue #34, February 1997

xldlas offers a straightforward way to summarize data, plot it and perform regressions on it—and it was written for Linux.

Linux is a virtually unparalleled platform for using freely distributable software. The kernel source is free, the standard utilities are free and so is the X Window System. The whole concept of using free software is incredibly appealing, and many users are tempted to try running their systems without any commercial products whatsoever. Yet this desire is often thwarted by a single missing application; desktop publishing and presentation software are commonly cited as current “holes” in the Linux arsenal.

I was faced with such a problem when I decided to abandon the MS-DOS partition on my hard drive and go all Linux. Since I work with a fair amount of statistical information, I needed a straightforward way to summarize data, plot it and perform regression as needed. gnuplot is great for plotting, but that's all it does. Octave and MuPad have powerful numerical features, but they are overkill for simple statistical chores. Unable to find a program that fit this niche, I decided to write one. The result is xldlas, a program for statistics. In the grand Unix tradition, its name is a pseudo-acronym which stands for “x **l**ies, **d**amned **l**ies, **a**nd **s**tatistics.” The first public release in October 1996 met with quite positive feedback from users, and one of those beta testers (Hans Zoebelein) suggested an article in *Linux Journal* might be a good way to introduce xldlas to a wider audience. The people at *LJ* agreed, and asked me to write this overview. The program runs under the X Window System, and is built using the XForms library. You'll find information on how to download xldlas and associated software at the end of this article.

Using xldlas

The philosophy behind xldlas is to offer standard statistical tools via an easy-to-use point and click interface. To facilitate this approach, common commands

are grouped together into a set of menus. In addition, frequently used commands are available via buttons (See Figure 1).

Figure 1: The xldlas User Interface

Like most statistics packages, xldlas handles a random variable as a vector of values. So a single variable name can refer to dozens, hundreds or thousands of observations.* By grouping data points together under variable names, it is easy to perform relatively complex operations by selecting a few variables and clicking on the relevant command.

*By default, xldlas has a limit of 100 variables of 10,000 observations each. These constraints can easily be adjusted by changing the values for MAX_VARS and MAX_OBS in the source code file xldlas.h.

Of course, before you can perform any kind of statistical operations, you have to get data into xldlas. Since ASCII is the de facto standard for exchanging information under Linux, xldlas allows you to read in space-delimited data from a text file by using the Import command. You supply a file name, and tell xldlas whether the data is in column or row format. The import routine automatically figures out how many variables and observations there are, and reads in the data. To take a concrete example, suppose you have a file which contains space-delimited data on rainfall, temperature and barometric pressure for a single location. After importing this file, xldlas will have three variables in memory, which will be called unknown0, unknown1 and unknown2. You can change these names to anything you like using the Rename command, which is accessible from the Data menu. In addition to this simple ASCII format, xldlas can read and write sets of data in its own proprietary file format. By convention, these files have an .lda extension. Since variable names, descriptions and other useful information are stored in these files, it's generally a good idea to save all your data this way if you plan on using xldlas frequently. The Load, Save and Import commands can all be found in the File menu. To input data by hand, erase variables or perform any kind of editing, there are a number of related commands grouped together in the Data menu. Of these, the most frequently used is probably the Describe command, which generates a table in the main xldlas window showing you the name, number of observations, and a description of every variable currently in memory. In addition to changing observation values, the Edit command can also be used to enter a description for a variable.

Another frequently used item in the Data menu is the Generate command. This routine allows you to perform mathematical transformations on existing data. To continue with the weather example from above, suppose we want to convert our rainfall variable from millimeters to centimeters. With a few clicks

of the mouse, we can easily accomplish this task. We could also add some random noise, find the log of the data, or what have you. It's a far cry from Mathematica, but for simple operations the Generate command is quick and easy to use.

Once you have your data loaded, edited and transformed, the next logical step is to perform some kind of statistical work on it. To get a tabular summary of a single variable, including mean, variance, skewness and kurtosis, there's the Summarize command. If you want to check multiple variables for linear relationships, the Correlation command will produce a table of Pearson coefficients. Similarly, the ANOVA command lets you perform one-way and two-way analyses of variance by simply selecting variable names with your mouse and clicking the Go button.

The workhorse of statistical techniques, ordinary least squares regression, is available via the Regress command. Just select a single variable from the dependent browser, any number from the independent browser, and press Go. If you want to store fitted values, then you can enter a new variable name in the regression window. The output of the regression command is a set of three tables, which summarize the fit of the regression, break down the sum of squares deviations and list coefficient estimates. Relevant t-statistics and their associated probabilities are automatically included, as is the F coefficient and confidence level for a joint test of all the estimates.

xldlas also offers two experimental data fitting routines that use connectionist artificial intelligence techniques. The first, GA Fit, uses genetic algorithms to build a fit equation that minimizes the sum of squares between fitted values and actual observations of a given dependent variable. The second, NN Fit, creates a back-propagation neural network using selected independent variables for the input layer, and a single dependent variable for the output layer. In both cases, the fitted values from these techniques can be stored under a supplied variable name. These routines are sometimes useful for exploring non-linear relationships in data that are generally difficult to examine using standard OLS regression.*

*Although not part of the “standard” statistical toolkit, these sorts of AI techniques are becoming increasingly common in various contexts and are great for data mining. Although their implementations in xldlas are fairly rudimentary, more sophisticated modifications are likely if users request them.

In addition to manipulating data and performing analysis, xldlas allows you to graph variables. All of xldlas's graphical output is actually performed by gnuplot, an application which is included in all major Linux distributions. Two graphing commands are implemented: Plot and Histogram. The former lets you

create line and scatter plots, while the latter generates a histogram describing a variable's distribution. Both sorts of graphs can be titled and labeled, and they can be saved in any format supported by whatever version of gnuplot is installed on your system. In addition, you can set point and line styles, and the Histogram routine includes an optional feature which will superimpose a normal distribution with the same mean and variance as the data being graphed.

xldlas also provides fairly powerful logging facilities. The Log command allows you to echo all of xldlas's output to an ASCII file. A more powerful tool is the TeXLog command, which allows you to create a PlainTeX format log file with a user-supplied name. All subsequent output, such as regression tables, is written to this file in TeX format. Under xldlas's default configuration, all saved graphs are also included as Encapsulated PostScript insertions. This makes writing statistical papers (such as homework assignments) quite fast and efficient, since much of the time-consuming TeX markup is done automatically.

Finally, all xldlas commands are documented on-line in the Help menu. There are also a number of on-line tutorials, which many users of xldlas have found to be a very useful introduction.

Coming Soon

There are currently plans afoot to add a number of features to xldlas. Regressions using Probit and Logit models are high on the to-do list. A set of statistical filters is also likely to be available before long, making it possible to easily detrend data, remove outliers, and so on. HTML format log files will soon be supported. Another important task is to expand the documentation in the source code so it can be more easily modified by people other than the original author.

Like almost all freely distributable software, xldlas development is driven by user feedback. If there are features you want to see, send me some e-mail, preferably with a reference to the algorithm you would like to see implemented.

Resources

The best way to get a copy of xldlas is from its homepage at www.a42.com/~thor/xldlas. If you only have FTP access to the Internet, you can get it from ftp://sunsite.unc.edu/ (in pub/Linux/X11/xapps/math/). Both full source and a Linux ELF executable are included in the distribution, which is named xldlas-X.Y-srcbin.tgz, where X.Y is the version number (0.40 at the time of writing).

To run the included executable or compile from the source code, you'll need to have the XForms library installed on your system. For more information about XForms, visit the XForms homepage at bragg.phys.uwm.edu/xforms. Although designed to run under Linux, xldlas will apparently compile under almost all flavours of Unix for which the Xforms library exists, although a little tinkering with the Makefile is sometimes necessary. Make sure you look at the README file included in every distribution to get the latest news on compiling and running xldlas.

gnuplot is available at sunsite.unc.edu/pub/Linux/apps/math/gplotbin.tgz. It is also included in most Linux distributions.

For full functionality, xldlas also requires you to have a fairly complete TeX package installed on your machine. NTeX and teTeX are commonly used under Linux, and are available at http://sunsite.unc.edu/pub/Linux/apps/tex/.

Thor Sigvaldason has completed most of a PhD on the use of connectionist AI techniques in economic modeling. By the time this article appears, he'll either be a visiting pre-doc at the Santa Fe Institute or working in New York City. He can be reached by e-mail at thor@netcom.ca.

Archive Index Issue Table of Contents

Advanced search

What Is Multi-Threading?

Martin McCarthy

Issue #34, February 1997

With the 2.0 kernel, most Linux users now have the capability of using multi-threaded processes—well, what does that mean?

Perhaps one of the reasons you use Linux is because it is a multi-tasking operating system. In which case you are probably well aware of the utility of being able to do more than one thing at a time—perhaps compiling the utility which will bring you fame and fortune whilst editing a letter of resignation to your boss. So you might recognise the utility of an individual process that can do more than one thing at once.

When might this be a good idea? One application for multi-threading is a program which relies on a large number of very similar and independent mathematical operations—the oft-quoted example of this is matrix multiplication. We look at a simple example of a multi-threaded matrix multiplier later in this article.

Another type of application which can benefit from multi-threading is a server application. Whenever a client attempts to connect to the server, a new thread can be created to look after that client whilst the “watcher” thread continues to wait for more clients to connect.

“But wait!” I hear you cry. “Lots of server applications already work like that, but they simply fork another process rather than starting another thread.”

“You're right...” I reply.

“Don't interrupt,” you continue. “This sounds like another way of doing the same thing, but for no good reason other than to say it's a `multi-threaded application' and so you can bump up the price to those who like to be blinded by science.”

At this point I decide to ignore any more of your interjections and just explain.

Yes, creating a new thread is very similar to forking a new process, but there are differences. When a new process is forked, it shares relatively little data with the parent process which created it; when a new thread is created, it shares much more information (such as all the global variables and static local variables, the open files, and the process ID). The overhead of creating separate copies of everything makes the creation of a new process slower than the creation of a new thread. And the time it takes to pass control from one process to another (a *context switch*) is longer than the time required to pass control from one thread to another. Also, since the threads share their data space, passing information from one thread to another is much more straightforward than passing information from one process to another—while this does have its own problems, it need not be difficult if a little care is taken. A simple example of a multi-threaded server is given below.

A third type of application which could benefit from multi-threading is a Doom-like game where each of the computer-controlled baddies has its own "intelligence" and can act independently. The main game-play part of the program could be in its own thread (or multiple threads), there could be another thread handling each of the characters in the game who are controlled by real people, and yet more threads for each of the characters controlled by the computer.

POSIX Threads

There is a POSIX standard for multi-threading: 1003.1c. If you want to write portable programs (which doesn't seem like a bad idea to me), it would be wise to stick to this standard rather than use non-POSIX conforming libraries or using the raw system calls which Linus so kindly provided in the Linux kernel (about which I say just a little more later).

The examples in this article use POSIX multi-threading functions called from C. However, the concepts in some non-POSIX libraries and systems are often very similar. Solaris threads are easily understood by someone familiar with POSIX threads, and while Java threads and the multi-threading in the Win32 and OS/2 APIs are a little different, there should be little in these which would leave you quaking in your boots.

Linux Threading: Linus Gave Us clone()

Multi-threading capability is included in the version 2.0 Linux kernel (and many version 1.3 kernels). The *clone()* system call creates a new *context of execution*, or "COE" (to use Linus' term), which can be a new process, a new thread, or one of a range of possibilities which doesn't fit into either of these categories. The

fork() system call is actually a call to clone() with particular values as parameters, and the **pthread_create()** function call could be a call to clone() with a different set of values as parameters. clone() is used by some implementations of multi-threading libraries to provide *kernel threads*, but further discussion of clone() is beyond the scope of this article.

User-Level Threads Versus Kernel-Level Threads

Threading libraries can be implemented using kernel features for creation and scheduling (such as clone()), or entirely in user space where code in the library handles the creation of threads and the scheduling between threads within a process. In general, user-level thread libraries will be faster than kernel-thread libraries. On the other hand, kernel-thread libraries are likely to make better use of kernel facilities—if the kernel makes better use of multiple processors in the next release, so might all your multi-threaded programs. However, the programmer shouldn't need to worry about whether the library is implemented as a user-level library, a kernel-level library, or a combination of the two; the operation of the program should be essentially the same.

Available Libraries

There are a many POSIX threads libraries available for Linux, and some make no attempt to be POSIX-compliant. Also, some of the POSIX threads libraries are compliant with an early draft of the standard or just part of the standard. Anything which isn't compliant with the final POSIX standard may show different behaviour in some circumstances or have slightly different function calls. That is not to say these aren't good multi-threading libraries, but you should be aware of what you're using.

All examples in this article were tested with Xavier Leroy's release 0.3 (ALPHA) of LinuxThreads, covered by the GNU Library General Public License. This library is still being developed and aims to be fully POSIX-compliant sometime in the future. It is clone()-based, and so it has the advantages and disadvantages of a kernel-level implementation.

Don't Use Global Data

As previously noted, multi-threaded programming isn't really very difficult. However, there are ways to make life difficult for yourself.

Guides to good programming often say to avoid using global data. Maybe you've never seen the point of this—particularly if you're a careful programmer and you know to look after your global data. With multi-threading, there is another reason to avoid using global data. Consider the trivial code fragments in Listing 2.

When this function is called you would expect to see the numbers “0 1 2 3” printed. And indeed, this is what would happen. If you were to call this function from two different threads you might expect to see two sets of the numbers printed, one from each thread. If the two threads were to call this function at about the same moment you might expect to see the two sets of numbers interlaced, perhaps something like:

```
0 1 0 1 2 2 3
3
```

Here, thread A calls fn(), which starts to print the numbers. When thread A is only as far as “1”, thread B starts up and calls fn(), which manages to get as far as “2” before thread A gets control again. The call to fn() completes for thread A and thread B gets control again and finishes off.

However, this is not what would happen. Let me stress this is **NOT** what would happen. Instead, the output might be:

```
0 1 0 1 2 3
4 5 6 7 8 9 10 11 12 13 14...
```

What is happening is thread A calls fn() which prints the numbers up to “1”. Then thread B starts up and calls fn(). As the **for** loop is entered, the value of **i** is set to 0. Remember **i** is global, so it is shared by both threads—when thread B changes the value of **i**, thread A will see the same change. The counter reaches the value “2”, at which point thread A is given control again. When it attempts to increment **i**, it now reaches “3” and then “4”, at which point thread A does not print the value of **i** and is ready to exit the function. When thread B gets control again, it prints the current value of **i** (which is “4”, courtesy of thread A), increments it to “5”, then does the comparison **i!=4**. The comparison doesn't fail (and will not fail for a **long** time—not until **i** has reached the maximum **int** value and looped around again), so the loop continues printing numbers.

This sounds horrible. It is horrible! And it can be avoided by not using global data. If **i** was declared local to fn(), each thread would have its own copy and you would get the expected output.

How To Use Global Data

Sometimes you may find you need to use global data, either because you are adding multi-threading features to an existing program which cannot be rewritten, or because you believe global data is the correct thing to use. Notice, in this context, I'm saying “global data” when what I really mean is *shared data*--the data may not be global to the program, but it is global to (or shared between) one or more threads. In a conventional, single-threaded program **static** local variables can be a useful convenience; however, these are effectively

global variables and if you don't take care with them, they will cause you problems with bugs that can be very hard to track down. So what do you do with global data? All you have to do is make sure no other thread can change the value of your global data between the time you set a value to that data and the time you use the value. POSIX threads provides a number of ways of performing this *synchronisation* between threads. The simplest are mutual exclusion locks, or mutexes. A thread locks a mutex at the start of a section of code, and unlocks it at the end of that section of code. A thread which holds a lock on a mutex is said to own that mutex. While one thread owns that mutex, no other thread will be able to execute that same section of code.

Consider our previous example with the runaway loop. We can rewrite this so that it still uses global data, but with mutexes to prevent two threads modifying the same loop counter at the same time.

In Listing 3, **loopLock** is a mutex variable. Mutex variables should always be initialised before use, either by a call to the initialisation function **pthread_mutex_init()**, which can be used to set values for attributes which are particular to that mutex variable, or by using the standard macro **PTHREAD_MUTEX_INITIALIZER** which uses default values. These attributes can affect the priority and scheduling of the thread which owns the mutex, or dictate which threads can operate on the mutex (either those within the same process or any threads which can access the memory where the mutex resides). We make use of only default mutex attributes here.

When a thread makes a call to **pthread_mutex_lock()**, the mutex variable it tries to own will be either free or owned by another thread. (If a thread makes an attempt to lock a mutex which it already owns, the result is undefined. Don't do it!) If the mutex is free, the thread will take ownership of that mutex and **pthread_mutex_lock()** will return. If another thread owns that mutex, the call will wait until the mutex comes free and can claim ownership for its own calling thread. The thread then owns that mutex until it makes a call to **pthread_mutex_unlock()**. (Again, if a thread makes an attempt to unlock a mutex which it doesn't already own, the result is undefined.)

As a result, if more than one thread tries to execute the code in the function at the same time, then only the first will own the lock and enter the loop. Any other threads will sit at the **pthread_mutex_lock()** call until the first thread has exited the loop and unlocked the mutex. Then ownership will be given to just one of the waiting threads which will be able to enter the loop safely.

So if there are two threads which call this function at the same time, the output will be:

```
0 1 2 3
0 1 2 3
```

As expected, each pass through the loop completes before the next starts and the output is not interlaced.

Remember… Don't Use Global Data

Of course, this problem has a much more elegant solution: avoid using the global data in the first place. In Listing 4, the loop counter is local to the function and there is no conflict; each call to the function has its own version of **i**, and so each thread which calls that function will have its own version of **i**.

However, two (or more) different threads can be inside the loop at the same time and so the output can be interlaced, as we originally expected. For example:

```
0 1 0 1 2 2 3
3
```

If it is not appropriate for your application to have more than one thread within a certain piece of code at the same time (this piece of code is commonly referred to as a *critical section*), you may still wish to use mutexes, even though you do not have shared data.

Remember, though, critical sections can negate some of the advantages of multi-threading by forcing threads to wait for others. It is a good idea to keep any necessary critical sections as short as possible.

Thread-Safe libc

What the programmer may need to be aware of is whether the other libraries being used are *thread-safe* or not. Otherwise, problems such as those described above may (or, more likely, will) occur. A good example is with every C and C++ programmer's friend, **errno**. When a system call fails for some reason, it is common for the function called to return a generic failure value (such as *NULL* or -1) and an indication of why it failed in **errno**--which is, of course, a global variable. Just think of the confusion if one thread calls **putenv()**, which fails and sets **errno** to **ENOMEM**. Then there is a context switch and another thread calls **open()** and fails, setting **errno** to **ELOOP**. Then there is another context switch; the return value from **putenv()** is checked, the failure is spotted, and confusion breaks out because it looks like **putenv()** failed with **ELOOP**, which is not possible.

Or consider the standard function **strtok()**. This breaks a string into tokens and works in two ways. On the first call, you pass in a string which is to be

tokenised. It stores this string in a static area. On subsequent calls you pass in a NULL string and **strtok()** works its way through the already stored string. If two threads call **strtok()** concurrently, the first one will have its stored string overwritten by the string that the second tells **strtok()** to store.

Fortunately, thread-safe versions of **errno** and thread-safe versions of standard functions within libraries are now available. Take a look at **/usr/include/errno.h**, for example. If you have a recent set of include files you may find the following bit of code:

```
#if defined(_POSIX_THREAD_SAFE_FUNCTIONS) || \
defined(_REENTRANT)
extern int*     __errno_location  __P((void));
#define errno   (*__errno_location ())
#else
extern int errno;
#endif
```

errno is redefined to be an **int** returned by a function, rather than the usual global variable, when either **_POSIX_THREAD_SAFE_FUNCTIONS** or **_REENTRANT** has been defined.

When compiling code for multi-threaded programs, always use:

```
#define _REENTRANT
```

in your code (or use the compiler option **-D_REENTRANT** to make use of this thread-safe macro for **errno**).

Similarly, recent versions of libc contain thread-safe versions of standard unsafe functions. So there is now the usual unsafe **strtok()**, but also a thread-safe function **strtok_r()**--see your man pages for the details.

Creating a Thread

All this, and we still haven't created a thread! Okay, let's put that right. Listing 5 is a very trivial, and complete, example of a multi-threaded program.

When I compile (with **cc helloworld.c -o helloworld -lpthread**) and run this program, I get the output: **Hello world world Hello Hello world world Hello Hello world world Hello Hello world world Hello Hello world world Hello**

Let's take a look at this program. There is a mutex variable **printfLock**; this may not be strictly necessary, but it is included just in case we are not using a thread-safe version of libc—just to be safe, mutex locking is put around the call to **printf()**.

There are two new function calls in **main()**: **pthread_create()** and **pthread_join()**. The first of these, not surprisingly, creates a new thread. The first parameter points to a variable of type **pthread_t**; this can be used to identify the newly created thread, not unlike the file-handle returned from the **fopen()** call. The second parameter allows you to set various attributes for the new thread. If the value is **NULL**, default attributes are used, and this is fine for now. The third parameter is the function which the thread is to execute; this is always a function which takes a **void*** as a parameter and has a **void*** return value. The fourth parameter is the argument which is to be passed as a parameter to the thread function defined in parameter 3.

As soon as **pthread_create()** has successfully created a new thread, the function which is passed as the third parameter will be running.

In this example, the thread function takes a string as an argument and prints that string ten times before exiting. As you can see from the output, both threads execute at the same time and their output is interleaved.

The **pthread_join()** function waits for the specified thread (identified using the **pthread_t** variable returned in parameter one of the thread creation) to exit. A thread exits when the thread function returns, or when the thread makes an explicit call to the function **pthread_exit()**.

Matrix Multiplication

Listing 6 shows a simple program which will perform matrix multiplication on a pair of fixed-size square matrices, with a separate thread performing the calculations for each column of the resulting matrix.

As a quick refresher, if it has been a while since you did matrix mathematics, the result of multiplying two matrices is given by the program in Listing 1. Running the program produces the output seen in Figure 1.

Figure 1. Matrix Manipulation Results

```
|  9  8  7  6  |   |  1   2   3   4  |   |  150  180  210  240 |
|  6  6  4  3  |   |  4   5   6   7  |   |  84  102  120  138  |
|  3  2  1  0  | x |  7   8   9  10  | = |  18   24   30   36  |
|  0 -1 -2 -3  |   | 10  11  12  13  |   |  -48 -54  -60  -66  |
```

There are a number of things worth noting about this program.

First of all, you will probably get a speed increase using a multi-threaded program like this only if you have a system with more than one processor. Why? Because with a single-processor system the one processor must perform all the calculations *and* spend extra time scheduling between the different threads. With a multi-processor system, the scheduler may be able to run the

same process over multiple CPUs by simultaneously running a different thread on each.

Secondly, it uses global data. And after I went to all that trouble to say it was a dangerous thing to do. And I don't even use mutexes. You might well be offended! However, I wanted to illustrate a case when it *is* safe to use global data. (Whether it is a good idea to use global data is another issue, and that line of philosophical debate is not the purpose of this article—I am simply showing that global data *can* be used.) Why is it safe here? Well, the two matrices **mat1** and **mat2** are read-only; the **result** matrix is written to, but each of the threads only writes to a specific part of the array which is used to store the resulting column which that thread is working on. There is never any conflict.

A Simple Server

Listing 7 shows one way in which a simple multi-threaded server could be constructed.

Please note, as in all the examples, error checking has been kept to a minimum in order to keep the code simple and readable. “Real” code needs to be much more robust than this. Always check the man pages for possible return-codes and error-values.

The server works as follows: it listens on port 12345 to service requests from clients, and it also performs an unrelated task without regard to client activity (that is, it prints “server is running” onto standard output every second).

In thread terms, it works as follows: the main thread creates a thread to listen for clients connecting to the server port, then it goes into an infinite loop, printing the “running” message, then sleeping for one second. The “client listening” thread listens to a socket on port 12345. Whenever a client connects to that socket (e.g., someone types “telnet hostname 12345”) the connection is accepted and another thread is started to handle just that one client. The “client listening” thread then continues to listen for more clients connecting. The “client handling” thread prints a useful message to the client (“type `X' to quit”), then waits for the client to send a message back. If the message begins with “X”, the socket is closed and the thread exits; otherwise, the message is printed again and the thread waits once more for data from the client.

In a server written in this way, there is a thread running for each client that is simultaneously connected. This gives the advantages of a server which forks a new process for each client that connects, but without the extra overheads involved in forking a new process or in switching between processes, and with the ability to communicate easily within threads.

Summary

Writing multi-threaded programs need not be difficult (certainly not as difficult as some people would have you believe), although it is certainly possible to make life difficult for yourself. It is more often useful on multi-processor systems than uni-processor systems. But there are many cases where performance can be improved on uni-processor systems by multi-threading, such as when handling requests from multiple simultaneous connections in a client/server application, when overlapping multiple I/O requests, or when writing programs which make use of graphical user interfaces.

Most of the topics discussed here should be applicable to non-Linux systems and (somewhat more loosely) to non-POSIX systems.

Martin has been a software-engineer for 10 years and has been using Linux since version 0.12. But he has more fun playing in a couple of bands and decorating his flat with Celtic knot-work. He can be reached at marty@ehabitat.demon.co.uk.

Archive Index Issue Table of Contents

Advanced search

A Comparison of Xemacs and Gnu Emacs

Larry Ayers

Issue #34, February 1997

Emacs aficionados will point out that Emacs is intended to be left running all of the time; in some ways the editor doubles as an operating environment or shell.

Introduction

Most Linux users have probably used the Gnu Emacs text editor at one time or another, if only out of curiosity. Originally intended as a programmer's editor, Emacs has grown over the years, accreting to itself pieces of LISP code from a variety of contributors. Now one of the most common complaints about the editor is its sheer bulk and concomitant slowness to load. Emacs aficionados will point out that Emacs is intended to be left running all of the time; in some ways the editor doubles as an operating environment or shell. As high-speed CPUs, faster hard disks and larger amounts of RAM have become more affordable, this complaint is no longer as much of a concern as it once was.

Emacs makes up for this minor drawback by being very configurable and extensible. Name a function related to text and there is probably an Emacs mode that will facilitate it. Many people use Emacs as their sole mail and Usenet news client, while programmers will find code-editing, compiling, and debugging well supported for a variety of programming languages. You can syntax-highlight any sort of text or code. The shell mode allows input and output from your preferred shell to be the contents of a buffer, from which text can be cut and pasted into other buffers. The various modes such as mail, news and the programming-specific modules are loaded and unloaded as needed.

Though many people over the years have contributed modes and enhancements to Emacs, the program is still firmly controlled by Richard Stallman of the Free Software Foundation, who wrote the first version back in the mid-seventies.

Two Editors Diverged in a Yellow Wood...

Before Gnu Emacs became well integrated with the X Window System there arose a group of Emacs-users, both private and corporate, who were impatient with the pace of the progress Stallman and the FSF were making on the program. They initiated a project to improve the user interface of Emacs (especially the X Window System support) with the intention of eventually merging the two code lines together.

One of the companies involved, the now-defunct Lucid, Inc., gave its name to the first versions of this divergent Emacs; you may have seen archives of Lucid Emacs files on the Linux FTP sites or on archive CDs.

Lucid Emacs acquired a new name, Xemacs, when the Lucid company folded. The University of Illinois, Sun Microsystems and the Amdahl Corporation emerged as the primary institutional and corporate supporters of the program.

Over the next few years conflicts arose between the FSF and the Xemacs team. Richard Stallman agreed to incorporate the Xemacs code into Gnu Emacs, but the conditions he set were unacceptable to the Xemacs developers. A stalemate has resulted. It looks as if for the foreseeable future the two versions of Emacs will develop in parallel. On the one hand, this may seem like a lot of wasted effort, but on the other, an element of competition has been introduced which in some ways benefits the end-user.

The Free Software Foundation might have been tempted to rest on its laurels if not for Xemacs. After all, the Gnu Emacs of even a couple of years ago was quite a unique and respectable piece of software. Improvements in recent years are tending towards refinements and user-interface improvements. The basic editing functionality really doesn't need much work.

Major Differences

Though both Gnu Emacs and Xemacs are programmed in the LISP programming language, the underlying structures are different enough that LISP add-on packages which work with one may not work with the other. Luckily many of the package maintainers are attempting to make their modules work with both Xemacs and Gnu Emacs, though the functionality may not be identical.

As an example, Bill Perry's W3 package, which turns Emacs into the world's only LISP-based web browser, works with both Emacsen (an odd term which is often used as a plural form of Emacs), but the Xemacs version will display inline images and backgrounds whereas Gnu Emacs won't.

Xemacs features an iconic toolbar, and several modes, e.g., the aforementioned W3 and the VM mail mode, have their own toolbars which only appear in Xemacs, though otherwise the modes function well in Gnu Emacs.

Gnu Emacs is limited to fixed-width fonts, while Xemacs can use variable-width proportional fonts. This feature will be of little interest to programmers, as source code looks ragged and is harder to read with proportional fonts. I confess I've never felt the need for anything but fixed-width fonts in a text editor, but tastes and needs differ.

I've found a few packages which work only with Gnu Emacs. One is NC, which replaces the native Emacs directory browser, Dired, with a Norton Commander-like interface.

Installing Xemacs

Precompiled binaries for Linux and many other operating systems are available from ftp://ftp.xEmacs.org and its mirrors. Generally there is a common file which everyone needs; this contains the platform-independent LISP files and miscellaneous documentation, a mix of the GNU Emacs documents and Xemacs-specific stuff. A smaller file contains the binaries for your platform. For Linux there is a choice between statically-linked Motif executables, dynamically-linked Motif executables (for those who own a copy of the the Motif libraries), and dynamically-linked Athena executables; the latter is probably the best choice if you don't have Motif.

If you want to compile your own executables, be prepared to make lots of disk space available. 70MB is a bare minimum (including the unpacked source code) that will work if you “compile in place”, i.e., the source-code directory will be where you leave Xemacs. This way the new executable refers to LISP files in the source directory. Otherwise you need another 50MB momentarily. The structure of the Xemacs directory tree is hard-coded into the executable; if you'd like to customise it the Makefile has paths that can be edited.

There are various reasons for compiling your own executable—perhaps you have no need of the icon toolbars. I've noticed that I rarely use them now that I'm more familiar with the equivalent keystrokes, and they do use memory you might not have to spare. The toolbar support (as well as the pop-up menu support) can be disabled at compile time, helping to create a leaner executable.

Another possible change is the inclusion of a frequently used package in the executable, rather than loading it separately from the ~/.Emacs initialization file.

Once you've unpacked the massive Xemacs archive files you may very well be dismayed by the amount of disk space they occupy. Luckily you can delete quite a bit of them and still have a great editor. If you look at the subdirectories under the LISP directory you will find many bulky packages for which you may never have a need, such as the Object-Oriented Browser and Hyperbole. If you have no desire to use Xemacs as a web browser (W3 is well-implemented but rather slow), the W3 and URL subdirectories can go. I recommend spending some time browsing through the LISP subdirectories; the un-byte-compiled *.el files are human-readable and most contain informative notes which will help you decide if you want to keep them or not. Another substantial amount of disk-space can be reclaimed by simply gzipping all of the *.el files; Xemacs doesn't need them at all. The reason that they are included with Xemacs is to give users the opportunity to edit them in order to modify their behaviour. The byte-compiled *.elc files are what Xemacs actually uses.

Configuration

When either version of Emacs is started the program immediately checks your home directory for an .Emacs file. This LISP file tells Emacs which packages you would like loaded and can set a vast array of user choices. Xemacs comes with a sample.Emacs file (in the /etc subdirectory) that is well commented and includes several default functions which many have found useful. Even more useful is the version-detection code, which determines whether you are running Gnu Emacs or Xemacs and the version number. Separate sections of the file are read by corresponding versions of Emacs, allowing the use of the same .Emacs file for any version of Xemacs or Gnu Emacs.

This permits the user to have both Xemacs and Gnu Emacs installed and usable at the same time, allowing for comparison. Xemacs 19.14 refines this even further, putting its configuration in a separate file called ~/.xEmacs-options that is called from within the ~/.Emacs file.

Though the prospect of writing LISP code in order to introduce your own functions and modifications into your .Emacs file may seem daunting, learning by imitating what others have done can be effective as well. Another help is the detailed documentation supplied with both versions in the form of hierarchical Info files. Several skilled Emacs users have posted their personal ~.Emacs files on web pages; a search conducted with Lycos or Alta Vista for the keywords Emacs, Gnu Emacs, or Xemacs will easily find such pages.

Advantages of Gnu Emacs

Although at first exposure it may seem that Xemacs offers more to users, Gnu Emacs has certain features that will tip the scales in its favor for some people. If you have the GPM console mouse server installed (this is included in most

distributions of Linux) Gnu Emacs interacts well with it, giving full mouse support in console mode. Xemacs has limited mouse functionality in a console session, but has the advantage of full face support in a console, i.e., syntax highlighting will work in a console session, although without as many colors as in an X session.

Gnu Emacs has a convenient editing feature: text copied or cut from a file can be accessed from a pull-down menu on the menu bar, with each piece of text identified by its first few words. While there is a way to implement a similar function in Xemacs, it's not documented in the distribution.

If memory usage is a factor for you, Gnu Emacs typically uses about 2MB less than Xemacs, and its startup time is slightly shorter. Of course, if you would like a lean, low-memory version of either Emacs, you can compile without X Window System support.

Safety

Both versions of Emacs protect you from losing files or unsaved text. As you type, the current buffer is periodically saved (at user-configurable intervals) to a file called “#filename#”, which can be restored in a later editing session. As soon as a file is successfully saved, this temporary file is automatically deleted. The normal type of backup file (*filename*~) is also created when files are saved. It would be difficult to lose very much text with these safeguards in effect.

Interesting Packages

Although both Gnu Emacs and Xemacs come with HTML editing modes, another possibility is the excellent HTML-Helper-Mode by Nelson Minar. This mode, available from http://www.santafe.edu/~nelson/, is quick, has good syntax highlighting, and supports Netscape tags and tables.

Ben Wing, one of the main developers of Xemacs, has written an elaborate SGML editing mode, which could be useful to anyone writing in the Linuxdoc SGML format, as used by the Linux Documentation Project. This package is included with Xemacs.

The VM mail system is included with Xemacs, and can be obtained separately for use with Gnu Emacs. Though Rmail (the original Emacs mail client) comes with both Emacs versions, it's not as full-featured as VM and uses a proprietary message format, which is a nuisance if you wish to access mail folders with other mail programs.

And then there is William Perry's W3, an ongoing project (consisting of a package of LISP files) which allows Emacs to function as a web browser. In its

latest incarnation W3 supports style-sheets, inline images, background colors and bitmaps, and even some of the Netscape tags. It's written in LISP, though, and tends to be rather slow. With graphics turned off, running it is like running an improved Lynx as part of Emacs. W3 is definitely worth checking up on from time to time, as development is active and newer versions of Xemacs are likely to be optimized for running W3 as well. The current stable and beta versions of W3 can be obtained from ftp://ftp.cs.indiana.edu in the /pub/eLISP/w3 directory.

Conclusion

Either one of these two editors contains more features and obscure functions than most of us will ever use. Xemacs is characterised by its bells and whistles, and its developers maintain a strong presence on the Internet. Gnu Emacs may have more users, many of whom are also willing to help newcomers, but if you are interested in influencing future development of either editor, you will probably have more luck with the Xemacs team. Luckily the basic editing commands in each version are nearly identical, so if you learn one it doesn't take long to come up to speed in the other.

Larry Ayers (layers@vax2.rain.gen.mo.us), lives on a small farm in northern Missouri, where he is currently engaged in building a timber-frame house for his family. He operates a portable band-saw mill, does general woodworking, plays the fiddle and searches for rare prairie plants, as well as growing shiitake mushrooms. He is also struggling with configuring a Usenet news server for his local ISP. His e-mail address is layers@vax2.rain.gen.mo.us.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Introducing Real-Time Linux

Michael Barabanov

Victor Yodaiken

Issue #34, February 1997

While Linux seems a natural solution for many applications, when milliseconds become critical, a robust multitasking environment may be too busy. RT-Linux gets the system under control to meet real-time computing needs.

If you wanted to control a camera or a robot or a scientific instrument from a PC, it would be natural to think of using Linux so you could take advantage of the development environment, the X Window System, and all the networking support. However, Linux cannot reliably run these kinds of hard real-time applications. A simple experiment can illustrate the problem. Take a speaker and hook it up to one of the pins from the parallel port; then run a program that toggles the pin. If your program is the only one running, the speaker will produce a nice, somewhat steady tone—not completely steady, but not bad. When Linux updates the file system every couple of seconds, you might notice a small change in tone. If you move the mouse over a couple of windows the tone becomes irregular. If you start Netscape, you hear intervals of silence as your program waits for higher priority processes to run.

The problem is that Linux, like most general purpose operating systems, is designed to optimize *average* performance and to try to give every process a fair share of compute time. This is great for general purpose computing, but for real-time programming, precise timing and predictable performance are more important than average performance. For example, if a camera fills a buffer every millisecond, a momentary delay in the process reading that buffer can cause data loss. If a stepper motor in a lithography machine must be turned on and off at precise intervals in order to minimize vibration and to move a wafer into position at the correct time, a momentary delay may cause an unrecoverable failure. Consider what might happen if the task that causes an emergency shutdown of a chemistry experiment must wait to run until Netscape redraws the window.

It turns out redesigning Linux to provide guaranteed performance would take an enormous amount of work. And taking on such a job would defeat our original purpose. Instead of having an off the shelf general purpose OS, we would have a custom-made special purpose OS that would not be riding the wave of the main Linux development effort. So we slipped a small, simple, real-time operating system *underneath* Linux. Linux becomes a task run only when there is no real-time task to run, and we pre-empt Linux whenever a real-time task needs the processor. The changes needed in Linux itself are minimal. Linux is mostly unaware of the real-time operating system as it goes about its business of running processes, catching interrupts, and controlling devices. Real-time tasks can run at quite a high level of precision. In our test P120 system, we can schedule tasks to run within a precision of about 20 microseconds.

Real-time Linux is a research project with two goals. First, we want a practical, non-proprietary tool we can use to control scientific instruments and robots. Our other goal is to use RT-Linux for research in real and non-real-time OS design. We'd like to be able to learn something about how to make operating systems efficient and reliable. For example, even a non-real-time operating system should be able to determine whether it can guarantee timing needed for its I/O devices. We're also interested in what types of scheduling disciplines actually turn out to be the most useful for real-time applications. Following this dual purpose, in this paper we discuss both how to use RT-Linux and how it works.

Using RT-Linux 2.0.RT.1

Let us consider an example. Suppose we want to write an application that polls a device for data in real-time and stores this data in a file. The main design philosophy behind RT-Linux is the following:

Real-time programs should be split into small, simple parts, with hard real-time constraints, and larger parts that do more sophisticated processing.

Following this principle, we split our application into two parts. The hard-real-time part will execute as a real-time task and copy data from the device into a special I/O interface called *real-time fifo*. The main part of the program will execute as an ordinary Linux process. This part will read data from the other end of the real-time fifo and display and store the data in a file.

The real-time component will be written as a kernel module. Linux allows us to compile and load kernel modules without rebooting the system. Code for a module always starts with a define of MODULE and an include of the module.h

file. After that, we include the real-time header files **rt_sched.h** and **rt_fifo.h** and declare an **RT_TASK** structure.

```
#define MODULE
#include <linux/module.h<
/* always needed for real-time tasks */
#include <linux/rt_sched.h>
#include <linux/rt_fifo.h>
RT_TASK mytask;
```

The real-time task structure will contain pointers to code, data, and scheduling information for this task. The task structure is defined in the first include file. Currently, RT-Linux has only one fairly simple scheduler. In the future, the schedulers will also be loadable modules. Currently, the only way for real-time tasks to communicate with Linux processes is through special queues called real-time fifos. Real-time fifos have been designed so that the real-time task will never be blocked when it reads or writes data. Figure 1 illustrates real-time fifos.

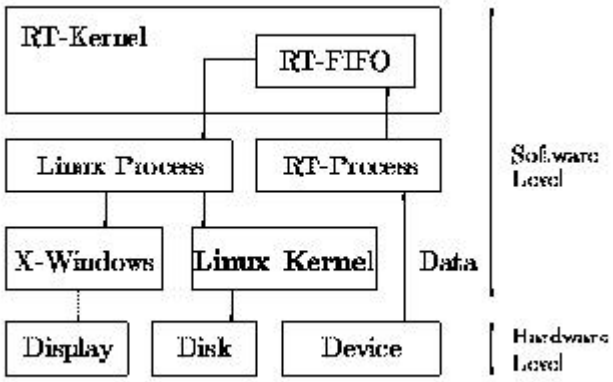


Figure 1. Real-time fifos

The example program will simply loop, reading data from the device, writing the data to an RT-fifo, and waiting for a fixed amount of time.

```
/* this is the main program */
void mainloop(int fifodesc) {
        int data;
        /* in this loop we obtain data from */
        /* the device and put it into the */
        /* fifo number 1 */
        while (1) {
                data = get_data();
                rt_fifo_put(fifodesc, (char *)
                  &data, sizeof(data));
                  /* give up the CPU till the */
                  /* next period              */
                rt_task_wait();
        }
}
```

All modules must contain an initialization routine. The initialization routine for the example real-time task will:

- record the current time,
- initialize the real-time task structure,
- and put the task on the periodic schedule.

The **rt_task_init** routine initializes the task structure and arranges for an argument to be passed to the task. In this case, the argument is a fixed descriptor for a real-time fifo. The **rt_make_periodic** routine puts the new task on the periodic scheduling queue. Periodic scheduling means the task is scheduled to run at certain intervals in time units. The alternative is to make the task run only when an interrupt causes it to become active.

```
This function is needed in any module. It */
/* will be invoked when the module is loaded. */
int init_module(void)
{
        #define RTfifoDESC  1
        /* get the current time */
        RTIME now = rt_get_time();
        /* `rt_task_init' associates a function */
        /* with the RT_TASK structure and sets  */
        /* several execution parameters:        */
        /* priority level = 4,                  */
        /* stack size = 3000 bytes,             */
        /* pass 1 to `mainloop' as an argument  */
        rt_task_init(&mytask, mainloop,
            RTfifoDESC, 3000, 4);
        /* Mark `mytask' as periodic. */
        /* It could be interrupt-driven as well */
        /* mytask will have period of 25000 */
        /* time units. The first run is in  */
        /* 1000 time units from now         */
        rt_task_make_periodic(&mytask, now +
            1000, 25000);
        return 0;
}`
```

Linux also requires that every module have a cleanup routine. For a real-time task, we want to make sure a dead task is no longer scheduled.

```
/* cleanup routine. It is invoked when the */
/* module is unloaded.                     */
void cleanup_module(void)
{
        /* kill the realxi--time task */
        rt_task_delete(&mytask);
}
```

That's the end of the module. We also need a program which runs as an ordinary Linux process. In this example, the process will just read data from the fifo and write (copy) the data to **stdout**.

```
#include <rt_fifo.h>
#include <stdio.h>
#define RTfifoDESC  1
#define BUFSIZE 10
int buf[BUFSIZE];
int main()
```

```
{
        int i;
        int n;
        /* create fifo number 1 with size of */
        /* 1000 bytes                        */
        rt_fifo_create(1, 1000);
        for (n=0; n < 100; n++) {
                /* read the data from the fifo */
                /* and print it                */
                rt_fifo_read(1, (char *) buf,
                        BUFSIZE * sizeof(int));
                for (i = 0; i < BUFSIZE; i++) {
                        printf("%d ", buf[i]);
                }
                printf("\n");
}
        /* destroy fifo number 1 */
        rt_fifo_destroy(1);
        return 0;
}
```

The main program could also display data on the screen, send it over the network, etc. All these activities are assumed to be non-real-time. The fifo size must be big enough to avoid overflows. Overflows can be detected, and another fifo can be used to inform the main program about them.

Why Linux Can't Do Real-Time and Why Simple Fixes Don't Work

Although Linux has system calls for suspending a process for a given time interval, it does not guarantee the process will be resumed as soon as this interval has passed. Depending on system load, the process might be scheduled more than a second later. Furthermore, a user process can be preempted at an unpredictable moment and forced to wait for its share of the CPU time. Assigning the highest priorities to critical tasks does not help, partly because of the Linux “fair” time-sharing scheduling algorithm. This algorithm tries to make sure every user program gets its fair share of computer time. Of course, if we have a real-time task, we want it to get the CPU whenever it needs it, no matter how unfair that may be. Linux virtual memory also contributes to unpredictability. Pages belonging to any user process can be swapped out to disk at any time. Bringing the requested page back to RAM takes an unpredictable amount of time in Linux.

Some of these problems are easily, or somewhat easily, fixed. It's possible to create a new class of special Linux processes which are more real-time. We could change the scheduling algorithm, so that real-time processes are scheduled round-robin or periodically. We could *lock* a real-time process into memory, so that its pages will never be swapped out. In fact, both ideas are part of the POSIX.1b-1993 specification which defines standards for “real-time” processes. And POSIX.1b-1993 is being incorporated into Linux. In newer versions of Linux, system calls are already provided for locking user pages in memory, making the scheduler policy priority-based and even for more predictable handling of signals.

POSIX.1b-1993 does not solve all our problems. It's not intended to solve the kinds of problems we discussed at the beginning of this article. The standard is aimed at so-called *soft* real-time programs. A program which displays video in a window is a perfect example of a soft real-time task. We want this task to run quickly and quite often in order to get a good quality display, but a few milliseconds here or there won't make much difference. For hard real-time problems, the POSIX standard has several drawbacks:

- Linux processes are *heavyweight* processes associated with significant overhead from process switching. Although Linux is relatively fast in switching processes, it can take several hundred microseconds on a fast machine. This would make it impossible to schedule a task to poll a sensor every 200 microseconds.
- Linux follows the standard Unix technique of making kernel processes non-preemptive. That is, when a process is making a system call (and running in kernel mode) it cannot be forced to give up the processor to another task, no matter how high the priority of the other task. For people who write operating systems, this is wonderful, because it makes a lot of very complicated synchronization problems disappear. For people who want to run real-time programs it is not so wonderful, since important processes cannot be scheduled while the kernel works on behalf of even the least important process. In kernel mode, it cannot be rescheduled. For example, if Netscape calls **fork**, the fork will complete before any other process can run.
- Linux disables interrupts in *critical sections* of kernel code. This disabling of interrupts means a real-time interrupt can be delayed until the current process, no matter how low its priority, finishes its critical section. Consider this piece of code:

```
line1: temp = qhead;
line2: qhead = temp->next;
```

Suppose that before the kernel gets to line 1, **qhead** contains the address of a data structure that is the only data structure on the queue and that **qhead->next** contains 0. Now suppose the kernel routine finishes line 1 and computes the value **temp->next** (which is 0), and then is halted by an interrupt that causes a new element to be added the the queue. When the interrupt routine finishes, **qhead->next** will not be equal to 0 any more, but when the kernel routine continues it will assign the 0 value to **qhead** and so will lose the new element. To prevent these types of errors, the Linux kernel makes extensive use of the **cli** command to *clear (disable) interrupts* during these critical sections. The kernel routine in this example would disable interrupts before it began changing the queue and re-enable interrupts only when the operation was complete; thus, interrupts would sometimes be delayed. It's hard to calculate the worst possible delay that can be caused by a critical section. You'd have to carefully

examine the code for every driver (and much of the rest of the OS as well) to even make a good estimate. We've measured delays of as long as 1/2 millisecond. Consider what such a delay would mean to our camera routine.

Changing the Linux kernel to be a preemptable real-time kernel with low interrupt processing latency would require substantial rewriting of the Linux kernel code—almost writing a new one. Real-time Linux uses a simpler and more efficient solution.

How RT-Linux Works

The basic idea is to make Linux run under the control of a real-time kernel (See Figure 2). When there is real-time work to be done, the RT operating system runs one of its tasks. When there is no real-time work to be done, the real-time kernel schedules Linux to run. So Linux is the lowest priority task of the RT-kernel.

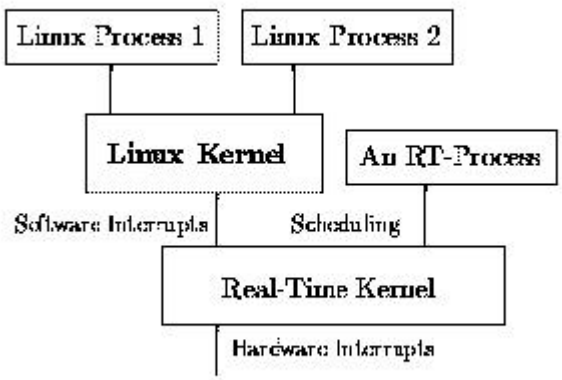


Figure 2

The problem with Linux disabling interrupts is solved by simulating the Linux interrupt-related routines in the real-time kernel. For example, whenever the Linux kernel invokes the **cli() -** routine that is supposed to disable interrupts, a *software* interrupt flag is reset instead. All interrupts get caught by the RT-kernel and passed to the Linux kernel according to the state of this flag and the interrupt mask. Therefore, the interrupts are always available for the RT-kernel, while still allowing Linux to “disable” them. In the example above, the Linux kernel routine would call **cli()** to clear the soft interrupt flag. If an interrupt occurred, the real-time executive would catch it and decide what to do. If the interrupt caused a real-time task to be run, the executive would save the state of Linux and start the real-time task immediately. If the interrupt just needed to be passed along to Linux, the real-time executive would set a flag showing a

pending interrupt, then resume Linux execution without running the Linux interrupt handler. When Linux re-enables interrupts, the real-time executive will process all pending interrupts and cause the corresponding Linux handlers to execute.

The real-time kernel is itself non-preemptable, but since its routines are very small and fast, this does not cause big delays. Testing on a Pentium 120 shows the maximum scheduling delay to be less than 20ms.

Real-time tasks run at the kernel privilege level in order to provide direct access to the computer hardware. They have fixed allocations of memory for code and data—otherwise, we would have to allow for unpredictable delays when a task requests new memory or pages in a code page. Real-time tasks cannot use Linux system calls or directly call routines or access ordinary data structures in the Linux kernel, as this would introduce the possibility of inconsistencies. In our example above, the kernel routine changing the queue would invoke **cli**, but this would not prevent a real-time task from starting. So we cannot allow the real-time task to directly access the queue. We do, however, need a way for real-time tasks to exchange data with the kernel and with user tasks. In a data collection application, for example, we might need to send the data collected by an RT-task over the network, or write it locally to a file, while displaying it on the screen.

Real-time fifos are used to pass information between real-time processes and ordinary Linux processes. RT-fifos, like real-time tasks, are never paged out. This eliminates the problem of unpredictable delays due to paging. And real-time fifos are designed to never block the real-time task.

Finally, the question—how the real-time kernel keeps track of the real-time—arises. When implementing schedulers for real-time systems, there is usually a tradeoff between the rate of clock interrupts and *task release jitter*. Typically, sleeping tasks are resumed during the execution of the periodic clock interrupt handler. A comparatively low clock interrupt rate does not impose much overhead, but at the same time causes tasks to be resumed either prematurely or too late. In real-time Linux, this problem is obviated by using a high-granularity, one-shot timer in addition to standard periodic clock interrupts. Tasks are resumed in the timer interrupt handler precisely when needed.

What's Next for RT-Linux

The current version of RT-Linux is available by anonymous ftp from luz.cs.nmt.edu. Information on RT-Linux can be found on the web at luz.cs.nmt.edu/~rtlinux. The system is in active development, so it's not at a production level of stability, but it's pretty reliable. We are developing some applications as well, and these will also be on the web site. We are asking

people who use the system to make their applications available on the web site as well.

Michael Barabanov

Victor Yodaiken is a professor of computer science at New Mexico Tech. His research is on operating systems, real-time, and automata theory.

Archive Index Issue Table of Contents

Advanced search

At Last, An X-Based vi

Dan Wilder

Issue #34, February 1997

One reader's quest has come to a successful conclusion.

The **vi** editor and kin are used (if maybe not always loved) by many who value a small, nimble, no-frills programmer's editor. The keystrokes are somewhat cryptic, but mostly just terse. With a small but sufficient command set, a rudimentary set of modes, and no scripting language to speak of, vi is extensible only by modifying the source code. But it starts up in the blink of an eye. Its keyboard response speed is just short of dazzling. Sophisticated global commands are available. The learning curve is steep at the beginning, but once the muscles have learned the commands and the brain has forgotten them, vi is both fast and effortless.

Unfortunately most vi clones have little in the way of X Windows support. When running in an xterm, some will resize nicely, and some will let you click the mouse to position the cursor. But that's about it for mouse support. No scrollbars. X selection at your own peril. Not much other use for the mouse at all.

Enter vile-5.6, compiled for X11 as **xvile**.

While not perfect, the basis is solid, and this editor delivers:

- A vertical scrollbar
- Good integration of primary X selection
- Mouse-based cut and paste
- Mouse selection of regions for other commands
- Mouse commands to open, resize and close subwindows (panes)

vi also allows commands prefaced by a colon—the so-called “colon commands”. In xvile colon commands I found a few jarring things; the release notes and help files warn about these.

While I'm no vi zealot, I've used vi off and on for seven years, so I'm pretty familiar with the command set. With xvile I found few surprises in the standard vi commands. Mostly, it just did what it should. Paul Fox (not the Paul D. Fox of Crisp fame), Tom Dickey, Kevin Buettner, and a host of others must have put in many long evenings on this project. MicroEMACS was the starting point, but this doesn't look much like Emacs. No mere Emacs vi-mode, this is a vi editor down to its code. Weighing in at about 280K stripped, it is larger than conventional vi, but small enough to be nearly as fast.

Building It

After downloading the source from a sunsite.unc.edu mirror site, I unpacked it and followed the instructions in README.CFG. Configuration and build were uneventful, and xvile ran the first time I tried it.

When I typed:

```
xvile
```

a new window popped up. At a guess, I typed:

```
:h
```

and the main help screen appeared.

Features, Good and Bad

The help screens are ordinary read-only buffers, so you can use standard search and paging commands to look through them. This is an improvement over the help I've seen with other vi clones. There is, however, no introductory information, and no hypertext capability. This won't bother you if you already use vi. If you don't and would like to try this editor, a good book might help (see Resources, page 13).

There are extensions. Among these are commands to manipulate panes, rebind keys, justify text, and so on. There is a sophisticated macro capability, which perhaps I'll use some day, but the basic operation of the editor is fast partly because it does not at all depend on macros.

As with Emacs, functions are not inextricably bound to keys. Some Emacs commands have found their way into extensions to the colon command mode. For instance:

```
:bind-key undo-changes-backward u
:bind-key format-til M-w
```

can provide my preferred undo binding, and adds ragged-right text justification on ALT-w. These commands (without colons) also work in the .vilerc file, as do the other colon commands, so if you have some standard setup commands, you don't have to type them every time.

If you are thinking “bind-key looks like Emacs” you are partly right. “vile” stands for “vi Like Emacs.” As the README file says, the joke is old, but somehow the name has stuck.

I wasn't delighted with a few things. Chief among these were the weaknesses in the colon commands. For example, I'm used to typing:

```
:.w tmp
```

to write the current line to a file. xvile insists on the longer equivalent form:

```
:.,.w tmp
```

which my fingers forget to type.

Many of the colon commands originate with the ancient ed, via ex, which gives you all its formidable power for the things it does very well, such as global substitutions using regular expressions. For example, to duplicate the entire contents of lines 50 to 100 in a file, with at-signs (**@**) for delimiters, you might type:

```
:50,100s/..*/@&@&@/
```

and a line that looked like:

```
framos.bin
```

would now look like:

```
@framos.bin@framos.bin@
```

The most common ex colon commands are there, and do just about what you'd expect, though adding some dialogue might help the novice user. An experienced vi user just starting to use xvile should keep an eye on this dialog at first, as sometimes it asks questions that won't be anticipated, or might indicate when a colon command is not having its intended effect. Once you learn the quirks, you can ignore the dialogue. But many less common ex colon commands are missing, or have incompatible implementations. I use these commands a lot, and it took a while for me to get used to the differences.

Keystrokes can be rebound, but mouse-clicks cannot. I'd sure like to bind some things like **CTL-ALT-BUTTON3-DOWN** to useful actions.

The copying policy is clouded. vile is derived from MicroEMACS, which is under a license that limits commercial use. The extensions are under GPL. Paul Fox informs me he's tried without success to contact the MicroEMACS authors to clarify this situation. The README suggests you buy the original authors a beer if you ever meet them.

Some xvile differences are big improvements over standard vi. For instance, if you use the simplest form of the yank command to yank ten lines of text, thus:

```
10yy
```

you may then change to an alternate file:

```
:e
```

and paste that same text with just a p. This differs from standard vi, where the text goes away on a file change.

Another annoyance of standard vi is gone. You don't have to write your buffer to a file whenever you change windows. Don't abuse this unless you *really* trust your local electric power company, and your kids don't ever kick the power switch, or unless you don't mind enabling automatic save to disk. But it is a nice touch, when all you want is a quick look at another file or two or three or ten.

Unlike some vi clones, when you write a file you don't lose your place in the alternate file. In fact, xvile keeps your place in as many files as you have open.

The vertical scrollbar is handy, both for moving through the file and for telling where you are. In addition, mouse clicks on the scrollbar let you split or unsplit the window, and resize the panes.

The keypad HOME, END, PAGE UP and PAGE DOWN keys all do what you'd hope they do. So do the arrow keys. This again is different from some the vi clones.

Example xvile Window

An example xvile window is shown in Figure 1. The shaded text in the lower window is a mouse selection, to which any of those cursor-movement “operator” commands can be applied using **^s** (which I've rebound to **g**). So in traditional vi, to delete a word starting at the cursor, you would type:

```
dw
```

where **d** is the delete operator and **w** is the one-word cursor movement command. In xvile, you could also select a word (or three, or a few lines or whatever), using the mouse, then type:

```
d^s
```

Alternately, you can left-click at each end of the deletion with the **d** command intervening, thus: and so on with all the other operator commands, such as **c** (change), **j** (join), and so on. A complete list of these is elicited in xvile with the command:

```
:list-operators
```

An X selection can also be pasted by moving the mouse cursor where you want to paste, clicking the left button to set the cursor there, and then clicking the middle button to paste at the cursor. Rectangular selections are accomplished by holding down the Control key while doing the selection. Unfortunately, they are bound on the right by the shortest line included in the selection.

Summary

My search for an X-based vi is over. For me, xvile fills the bill.

If you aren't a vi user, but you like to try a new editor once in a while, give this one a try. Don't expect it to be self- teaching, however: you'll need the book.

If you are already a vi user seeking to use your mouse for more than a paperweight, and if you won't be seriously upset at having to relearn a few of your keystroke sequences, take a look at xvile.

Resources

xvile source can be retrieved via anonymous ftp from sunsite.unc.edu in the file /pub/Linux/apps/editors/vi/vile-5.6.tar.gz Use a mirror of sunsite where possible.

A good book for learning vi is *Learning the vi Editor*, by Linda Lamb, published by O'Reilly & Associates, ISBN: 0-937175-67-6, 192 pages, $21.95

The authors of vile are:Paul Fox, pgf@foxharp.boston.ma.usKevin Buettner, kev@primenet.comTom Dickey, dickey@clark.net

Dan Wilder is a programmer and system administrator in Seattle. He can be reached as dan@gasboy.com.

Archive Index Issue Table of Contents

Advanced search

XBanner: Making XDM More Attractive

Amit Margalit

Issue #34, February 1997

XBanner turns your boring, staid XDM login screen into one of those cool things you'll want to show all your friends (nonchalantly, of course).

XBanner was invented and designed from the beginning to serve one purpose —to beautify the login screen XDM usually generates. This beautification is accomplished by drawing a piece of text in a very large font, then rendering some graphic effect on the text and/or the screen background.

Figure 1. Standard XDM Login Screen

Figure 1 shows a plain login screen as displayed by XDM “straight out of the box”. Linux, which uses plain XDM, has a similar if not identical login screen. Commercial companies, like HP and DEC, supply custom XDMs that generate a nice login screen with the company logo, a background, etc. Figure 2 shows how my home system greets me when I turn it on. Using XBanner, any system can look as special as mine with minimal work on your part.

Figure 2. Author's Login Screen

Summary of Features

The text graphic rendering effects available in XBanner include many interesting things. In this article, I describe a few:

- **StandOut**: This effect, combined with a proper selection of colors, gives the text a look resembling that of Motif or MS Windows buttons. Motif adds a few pixels on the left and top of the button in a lighter color than the button and a darker shade to the right and bottom of the button to create a 3D effect. StandOut uses a similar technique on the entire text. Each letter and symbol appears to be three-dimensional like the Motif buttons. The thickness of the letters can be selected.

- **Backlight**: This effect draws 1-pixel thick lines around the text, giving it many outlines. If you choose a dark foreground for the text, then have XBanner generate a color gradient for the outlines going from a bright color near the letters to the same color as the background of the screen, the text looks as if it is being illuminated from behind.
- **FatText**: With the proper selection of parameters this effect makes your text look fat rather than flat. Some color combinations can actually make the text look as if it has round edges rather than sharp. Also, the FatText color gradient can be instructed to use more than two colors to get other interesting results.
- **FgPlasma**: XBanner draws a field of Plasma Clouds, cuts from it a template in the shape of selected text and places it on the screen. In Figure 2, the title “Good Guys Free!” is an example of the FgPlasma rendering effect.
- **FgGrad**: This draws a color gradient on the text itself. Instead of a plain vertical bar with serifs, the letter “I” would be striped in different colors.

Drawing the text is only the first of the two prime jobs XBanner was designed to do; it can also render a nice background. Apart from the simple fill-style background, XBanner can generate color gradients on the background using many different patterns. Here are some examples:

- **Fan**: This effect draws a nice fan, centered at the middle of the bottom line of the screen. The selected color gradient determines the colors of the fan. See the background of Figure 2 for a sample of the fan effect.
- **TopDown / LeftRight**: These create a color gradient going from one side of the screen to the other.
- **BgPix**: XBanner has the ability to tile the whole screen with a pixmap (.XPM file) before it draws the selected text.
- **Plasma**: PlasmaClouds fill the entire screen. Selecting such colors as White and DeepSkyBlue produces a wonderful sky-like image with genuine-looking white clouds.
- **Ripples**: This background style has the appearance of ripples in a pond. This background style, a result of color-cycling (explained below), produces beautiful effects.

Almost any object involving a color-gradient in XBanner can be color-cycled, generating a sense of motion. If you cycle the colors of a Fan background style, the entire fan appears to rotate in one direction. The color gradient of the **FgGrad** effect can also be cycled, giving a sense of motion to the text. Cycling the PlasmaCloud effects is also a nice idea. You can cycle the background Plasma, the foreground Plasma, or even both! I use color-cycling for my home system login screen, shown in Figure 2. Unfortunately, color-cycling is not available for hard-copy.

The Ripples background style was specifically designed for color-cycling. The ripples' wave-fronts move at different speeds, producing a more realistic look.

Another neat feature of XBanner is the ability to draw a star on the corners of the text, creating a “glinting” effect that appears at random locations at random intervals. You can also place a pixmap or set of pixmaps on the screen, underline any text or effect, and select many other options.

Why Write XBanner?

After working with X on Linux for a while, I realized I always start X, and I usually do it shortly after logging in. Therefore, I decided to run XDM, which gives me a login box already in X. XDM installed and ran out-of-the-box with no problems, no hassle.

After just a few days, I found I hated the way it looked. XDM's default is a dull, boring gray screen with a simple text box and nothing else. Compared with some commercial versions of Unix, it was definitely lacking. Since I wanted Linux to look cool, too, I decided to write XBanner.

Tested Platforms

XBanner was written primarily for Linux, but it is not Linux-specific. It can be compiled and run in any environment supporting X11 release 4 and up. I have tested XBanner on Linux, Ultrix, Digital Unix, Solaris, SGI IRIX, AIX and even VMS.

I have had a report that XBanner does not work with X Inside's Accelerated X server. I tried to contact X Inside about this issue, twice. I have been completely ignored—pity.

Getting XBanner

The XBanner home page, sponsored by the Physics department of California State University, Fullerton can be found at:

http://physics.fullerton.edu/XBanner/

And the FTP locations are:

ftp://physics.fullerton.edu/pub/Linux/XBanner/<\n> ftp://sunsite.unc.edu/pub/Linux/X11/xutils/ftp://vvtp.tn.tudelft.nl/pub/XBanner/

I expect new mirrors will become available by publication time. At the time of writing, the latest version is in XBanner1.3.tar.gz.

Getting and Setting Up XDM

Most Linux distributions (Debian, Slackware, et al.) include a package that sets up XDM and its configuration files. In case your distribution doesn't include XDM, it can be found at sunsite.unc.edu, the “home” of Linux software on the Internet—ftp://sunsite.unc.edu/pub/Linux/X11/xutils/xdm.tar.gz.

Compilation and installation is easy. XDM comes with an Imakefile. If your system has X11 installed properly, type **xmkmf -a** in the directory of the XDM source, then type **make**. If you have problems, consult the FAQs. Setting up the X11 environment and using imake are beyond the scope of this article.

After the installation is complete, create a directory /usr/lib/X11/xdm/ and copy the files from the config/ directory in the XDM source tree into it. Note that /usr/lib/X11/ should be synonymous to /usr/X11R6/lib/ on Linux.

How XDM Works

XDM reads the file /usr/lib/X11/xdm/xdm-config (see Listing 1 for the default contents), and extracts from it the location of the rest of the configuration files. Nearly all XDM configuration files are defined in xdm-config.

Listing 1. xdm-config Notice the files with the **_0** suffix refer to **:0**--the local display. XDM runs **Xsetup_0** to initialize the display, then pops up its login box and asks the user for a username and password. When the user has finished typing this information, XDM checks the password database. If the user is authenticated, XDM runs a few things including the Xsession script that sets up the user's environment and loads the window-manager.

These two files, **Xsession** and **Xsetup_0** are the files to which lines are added in order to run XBanner and Freetemp. This is discussed more completely later in this article.

Compiling and Installing XBanner

After downloading the XBanner source archive, unpack it using:

```
gzip -dc XBanner1.3.tar.gz | tar xvf-
```

Change directories to XBanner1.3/ and enter **make**. On Linux systems, this should proceed with no problems at all. After compilation is done you will have four executables:

- **xbanner** - the main XBanner program
- **freetemp** - utility to free X11 resources taken by XBanner

- **xb_check** - checks resource files for validity
- **random_effect** - executes the xbanner binary with a random resource file

Typing **make install** will install the executables to the directory /usr/local/bin/X11/ (not including the **random_effect** utility), and set proper permissions. You can change the destination directory by editing the Makefile.

If your system does not have the XPM library, the compiler might complain that it cannot find **libXpm.a** or **-lXpm**. In this case, edit the Makefile; it contains instructions on disabling XPM support.

Now, to set up a good default resources file, go to the XBanner1.3/ directory and issue the command:

```
cp samples/XBanner.ad \
/usr/lib/X11/app-defaults/XBanner
```

Making the Link Between XDM and XBanner

Next, tell XDM to run XBanner by adding a line to XDM's **Xsetup_0** script. This script runs before the login box pops up. Adding this will ensure that XBanner is run each time XDM wants to display the login box. The Xsetup_0 script is straightforward. Just add to the bottom of the file a line like this one:

```
/usr/local/bin/X11/xbanner -file \
/usr/local/etc/login_screen.res
```

Where **login_screen.res** (see Listing 2) contains the resources that generate the desired screen layout.

Listing 2. login_screen.res After XBanner draws its graphics, some server-side resources are left allocated (colormap entries) in order to ensure that programs running after XBanner can't alter the colors, and thereby make XBanner's graphics look a mess. The **freetemp** utility is the program that tells the X server to release the colormap entries. Also, it should be noted that some of the nicer features of XBanner, such as color-cycling and glittering stars, are done by leaving a process running, which must then be stopped before logging in. The freetemp utility does all the work of stopping lingering processes and releasing the colormap entries.

The best place to run freetemp is from the Xsession file. This file is run for **each** logged-in user. Place the freetemp line as close to the top of the file as possible, and make sure it is not within the context of an **if** statement or some other script element which might prevent it from running. As a safety measure, you can add a call to freetemp right before the call to xbanner in the Xsetup_0

script. This call is done just as a precautionary measure; it might save you in case something unexpected happened when the previous user logged out.

Creating a Resource File

The simplest way to start is to check out the many examples included in the installation. I have included samples of everything I could think of: color-cycling, multiple lines, glinting stars, different backgrounds, pixmaps, etc. The files in the samples/Demo/ directory are provided to serve as a template. Simply pick one and start modifying it. Experiment with the options until you find a look that pleases you.

The XBanner documentation is made up of two main files and some supplementary files. The file XBanner_Users_Guide.html contains, among other things, a hands-on structured guide to building a resource file from scratch. Follow the steps, and each click will bring you to the related resources and the available possibilities.

The file Resource_Reference.html contains a complete reference of all the resources XBanner recognizes. Each resource is given with its class name, an extensive explanation of what this resource does, what values it accepts and the command-line equivalent. The file also contains a description of the different *types* of resources XBanner uses, and some ideas.

This should be all you need to create a resource file. But of course, you can send me e-mail for suggestions and ideas. Many of the features of XBanner were implemented as a result of such questions.

Running the Demo

In order to run the demo properly—and in fact in order to use XBanner—you must have scalable fonts. These are fonts that can be scaled to any size without losing quality. XFree86 comes with a package of scalable fonts, and the XBanner documentation contains a special chapter about finding scalable fonts and installing them. You do not need XDM at all to be able to see the demo.

It is advisable that you close all applications before running the demo, to allow XBanner to allocate the necessary colors for the demo. Also, the best way to view the demo is with an empty screen with only a single small xterm located close to the bottom of the screen from which to run the demo.

After compilation, go to the directory samples/Demo/ and enter **./Demo** or **./Demo.bash** (if your system does not have csh).

Credits

I'd like to thank Oren Tirosh (orenti@hiker.org.il) for many hours of help, support, ideas, algorithms and more. Thanks also to Rich McClelan of Fullerton University for all the help he's given me in setting up the web pages, and getting me the space.

Thanks to Ben Spade (spade@spade.com) and Chris Yeo (Chris.Yeo@usask.ca) for their help with the documentation of XBanner 1.3, and other stuff.

Thanks also to the hundreds of people who have notified me about bugs, offered ideas, requested features, or sent me a pat on the back by e-mail.

Amit Margalit works for Digital in Israel in X11/Motif programming support. He started using Linux back in the happy days of Linux 0.99.12 and is preaching for the migration to Linux at every opportunity. He may be reached at amitm@doronx.iso.dec.com and amitm@netvision.net.il.

Archive Index Issue Table of Contents

Advanced search

Graphing with lout

Michael Hall

Issue #34, February 1997

A document layout language, lout can also be used to generate graphs in PostScript.

Most documentation tools can also make graphs. This article will show you how to use lout, a document layout language, to create graphs. At the end we will have a script which creates a history graph of system activity.

First of all, lout is a document layout program which generates PostScript. It was written by Jeffrey Kingston at the Basser Department of Computer Science, the University of Sydney, Australia. It is available on Red Hat distributions such as Caldera's, and many Linux ftp sites. Also, the original sources are available at ftp.cs.su.oz.au under the “jeff” directory.

In many ways, lout is a preprocessor for PostScript, since many lout statements turn into PostScript instructions. In other ways, lout is a language for writing structured documents—we describe lists, sections, chapters, books and reports with lout commands, and our text becomes the arguments to those commands. In this article, we will focus on the commands in lout's graph package, and ignore its other capabilities for documents.

In general, lout commands start with a commercial-at sign, **@**. Arguments to a command appear to its right, enclosed in braces, and option settings can appear between the command and its argument:

```
@Command options  {  argument }
```

Usually the argument can contain lout commands also, so we'll get nested groups of arguments with their braces helping to sort everything out.

Let's jump right in and take a look at a simple lout document for a graph:

```
@SysInclude {graph}
@SysInclude {doc}
@Doc @Text @Begin
@Graph { @Data pairs {solid}  { 1 1  2 10  3 5 } }
@End @Text
```

Put this into a file called graph.l, and run lout like this:

```
lout -s -o ps  graph.l
```

lout reads the input file, and generates PostScript, writing it to the output file, here called **ps**. The **-s** tells lout that we have a stand-alone document, and we don't want it to make its cross-reference files.

When you look at the resulting **ps** file with Ghostview (or print it to a printer), you'll see a nice little graph at the top of the page. lout has automatically generated a box, figured out some appropriate scales, plotted the lines and displayed the graph. The graph package in lout has many options to modify the characteristics of this graph; we'll see some of them later.

For now, the important part is the **@Graph** command:

```
@Graph  {  @Data  options  { data-values } }
```

This tells lout to make a graph from a data set in its argument list. We gave the **@Data** command the option “pairs {solid}”, which tells lout that the data set has pairs of x-y coordinate values, and that we want the data points connected with solid lines.

As for the rest of the example, the **@SysInclude** commands tell lout which packages we'll be using in the document. The **@Doc** command says that the body of the document follows, which is delimited by the **@Text @Begin** and **@End @Text** statements. And that's about as difficult as lout gets.

System Activity Graph

Suppose we want to make a system activity graph. We'll want three curves, one each for the user, system and idle percentages. The user percentage curve will be a solid line, with filled-in circles to indicate each data point. The system and idle percentage curves won't have symbols at each data point, but the system percent curve will be dashed, and the idle percent curve will be dotted.

Moving from the data sets to the graph itself, we want the graph to fill the page, in a landscape orientation. There will be a title at the top, a brief description below the graph, and a y-axis label to the left. Instead of a legend box to identify the curves, we want an appropriate label to the right of the last point in each curve on the graph.

Implementation

So that's what we want. How do we get there? We will use awk as an interface between vmstat and lout. The overall flow will look like:

```
vmstat | awk | lout
```

vmstat generates numbers for the system activity over a period of time, and passes them on to awk. awk's role is to take those numbers, collect the ones it needs, and generate **@Graph** and **@Data** statements for lout. Finally, lout takes its document and converts it to PostScript. We can take the PostScript output and view it, print it, or embed it elsewhere. If the system is up all the time, we can even use a crontab entry to automatically print out daily activity reports at the end of each day.

Listing 1. SysAct

The complete script is shown in Listing 1. Let's step through it together. The echo command sends the lout setup code; we include the graph and doc packages, set the default font for the document, and turn off the page headers to avoid getting page numbers.

After the **@Text @Begin** statements, we show the rotated graph, or, as expressed in lout:

```
@Display 90d @Rotate @Graph { ... }
```

Here the **@Rotate** command takes two arguments: on its left, the number of degrees to rotate, and on its right, the lout object to rotate. In this case, the object we rotate is our entire graph.

The **@Graph** statement is followed by several graph options. We set the graph size, and we set some captions. We add a caption above the graph with **abovecaption { title text }**. Similarly, we add a comment below the graph, and a y-axis label to the left of the graph. Note that we rotated the **leftcaption** so that it would be parallel to the y-axis of the graph. After the captions, we specify the tick marks on the x-axis. By default, lout generated ticks only for every other minute. We can specify where we want ticks and what to label each tick with the **xticks** and **yticks** options. In this case, the y-axis ticks are fine, but we want a tick and label on the horizontal axis for each minute. The last two graph options tell lout to not leave any free space between the graph curves and the axes of the graph. This way, the bottom line of the graph box becomes the 0% line of the chart.

Now that all the setup is done, we can start collecting data. We call vmstat, and tell it that we want 60 samples taken 10 seconds apart, to collect data over a 10-minute period. We send these measurements to awk.

Inside the awk script, we collect the user, system, and idle percentages in separate arrays, taking care to skip over the header lines. When vmstat exits, we'll use these saved values to generate data sets for lout.

Inside the **END** routine of the **awk** script, we first generate the lout code to put the curve identifiers at the right side of the graph. In the code, these appear in an **objects { ... }** option to the **@Graph** command. This is a powerful feature which lets us put nearly anything anywhere on the graph. After the objects are written, we can generate the data sets. An awk function generates the lout statements for us—we pass the array name, the point-style, and the line-style to the routine called **gen_data**. It takes care of generating a well-formatted data set for lout.

Finally, the awk script finishes the lout document by declaring the end of the text with **@End @Text**.

The combined output of echo and awk are piped into lout, which sends the PostScript for the system activity graph to its standard output. This script could serve as the basis for many other graphs:

- We could pass in the options to vmstat to get better control over the sampling interval and duration.
- Instead of rotating the graph to take up the full page, we could leave it on a portrait-aligned page, and put additional graphs below it (for swap, free and buffer memory, perhaps).
- We could instruct the awk script to highlight severe system conditions by changing the line-style and point-style in separate data sets when the data values become too high or too low.
- We could generate annotations for the graph automatically, showing the time and percentage values when the system percentage crosses some threshold.

The principle is the same for any other data. Need a disk-usage graph? Run **du** through an interface script and generate some lout code to graphically show the disk-hogs on the system. Modify the script a little to get a directory usage graph for a single user.

I hope this brief introduction has piqued your interest in the graphing abilities of lout. For more information about lout, see the excellent user and reference

manuals that come with it. They cover everything you'll need to know to use lout.

Availability

Scripts for the system activity graph and a disk-usage graph can be found at: http://www.balr.com.

Mike Hall is a senior consultant at BALR Corporation, and can be reached at mghall@balr.com.

Archive Index Issue Table of Contents

Advanced search

Watchdog—The Linux Software Daemon

Michael Meskes

Issue #34, February 1997

Here's a program that will watch your computer for those nasty system hang ups and reboot it as needed in your absence.

Starting with version 1.3.51 the Linux kernel provides a watchdog driver. Not only does it take care of hardware watchdogs, but it also implements a software watchdog. It was created by Alan Cox, alan@lxorguk.ukuu.org.uk, and modularized by Angelo Haritsis, ah@doc.ic.ac.uk.

What is a Software Watchdog?

The Linux software watchdog is a timer that continuously counts down from a specified start value. Once it reaches zero a function named “watchdog_fire” is called that logs the event and initiates a hard reset. A watchdog daemon is therefore needed to refresh the timer. As long as the daemon is present the timer will never fire, but if the daemon stops working for some reason, the timer will fire and reboot the machine. However, with no additional hardware, the watchdog is not able to recover from every possible hang up. For instance, if the interrupt system itself hangs up, the software watchdog is no cure at all.

What WATCHDOG Does and How It Came To Be

At start-up, the daemon opens the watchdog device (a character device with major number 10 and minor number 130) starting the watchdog process—an infinite loop in which the daemon alternately writes to the watchdog device to refresh the timer, then sleeps for 10 seconds. If the daemon is killed, the device file is closed, and the timer is disabled again.

Initially, I compiled the example given by Alan Cox in the source tree to be able to start, refresh and stop the software watchdog daemon. Since I always run the latest kernel and libc release and rely on the machine to be up and running for e-mail, even this simple version proved very useful for me.

As time went by, more features were added. First, WATCHDOG became a real daemon in that it automatically forks as a background process, tucking its process ID away. Then, since WATCHDOG is performing a critical job, logging to the standard syslog facility was implemented, as well as some user-friendly options:

- Logging is done in verbose mode each time the process awakes.
- The sleep interval can be set from the command line.
- The file system is synchronized every time the process updates the timer.
- The watchdog device name can be chosen on the command line.

This rudimentary version was then released as a Debian package to give more people the chance to use, and thereby, test it. The next step was to implement some enhancements, namely the ability to test the accessibility of a user-named file every time the process is awake. This ability allows WATCHDOG to catch problems with machines mounting most or all of their directory tree via NFS or similar means of remote file systems—this connection can hang despite the local machine functioning correctly. Since this hangup can cause huge problems, the WATCHDOG daemon tries to access one file on this mount every time it awakes. If the connection is down, it will not be able to refresh the timer, and the system will be rebooted. WATCHDOG can be instructed to initiate a soft-boot instead of the normal hard reset when any call returns an error message.

At this point in the development of WATCHDOG, I released the daemon as version 1.0 to all Debian archives and to tsx-11.mit.edu.

What Has Changed Since Version 1.0?

With more people using WATCHDOG some problems have been encountered and fixed, and I began working on the one problem that still worried me. From time to time the process table got so full that no more processes could be started, so I changed WATCHDOG to check the process table every time it is awake. If the table is full, the system is rebooted. Doing a hard reset in this case seemed a bad idea, so instead, WATCHDOG initiates a complete shutdown. This action also posed a problem, since the process table might stay full despite WATCHDOG's attempt to kill all processes. For example, it is not possible to start the shutdown binary when the process table is not empty. To take care of this situation WATCHDOG contains code for the complete shutdown process, including the following actions:

- idles init, so nothing new can be started,
- kills all processes,
- writes a reboot record to wtmp,

- turns off accounting,
- turns off quota,
- turns off swap,
- unmounts all mounted partitions,
- calls reboot.

With this feature working correctly, version 2.0 was released on July 30, 1996 to tsx-11.mit.edu and every Debian mirror.

Of course, there are more ideas that need implementation. If anyone is in need of a check different from those described in this article, please write me at meskes@debian.org.

Michael Meskes made first contact with Linux while working on his doctoral thesis. He has been involved in testing and bug fixing the kernel and the C library since 1994. Since 1995 he has maintained several packages for Debian GNU/Linux. He likes professional football, in particular, the San Francisco 49ers. He welcomes your comments sent to meskes@debian.org.

Archive Index Issue Table of Contents

Advanced search

System Commander

Terrence Miller

Issue #34, February 1997

After reviewing System Commander for a week or so I can say that this program does exactly what it says it does—somewhat.

- Manufacturer: V Communications
- Price: $99.95 per license; 10 licenses for $649
- Reviewer: Terrence Miller

If you run more than one operating system, System Commander can provide a no hassle method for alternating between them.

After reviewing System Commander for a week or so I can say that this program does exactly what it says it does—somewhat. Before I get into details, let me say a few things about my experiences with the program. All testing was done on a generic laptop using a 486DX-33 with 4MB RAM.

To begin with, I was anxious to get started when I received my review copy, and the first thing I did was break into the package and begin to scan the owner's manual. Perhaps it was just the copy I received, or perhaps I was overly rough in my eagerness, but the manual did not seem to hold up well to handling—within minutes the spine was beginning to separate from the pages. My first thought was that if this was any indication of the quality assurance program at V Communications, I was going to be in trouble. Fortunately, my fears were unfounded. While the manual was the weakest link physically, it turned out I didn't need to bend the pages often enough for it to matter. It has got to be one of the best laid out manuals I've run a cross in a long time. Got a question on the installation? If the manual doesn't cover it, the on-line help probably will.

As to the System Commander program, you will need about 1/2MB uncompressed hard disk space for the installation. Each copy is licensed for a single user only. The installation program has a basic Master Boot Record virus

checker and prompts you several times to read chapter 3 of the manual regardless of how secure you feel in the installation process. Personally, as easy as the program is to use, I **HIGHLY** recommend you read the manual thoroughly. There are some quirks specific to certain operating systems that can really cause headaches.

Something to consider before using System Commander is exactly which OS you will be using. Some preliminary planning and decision making prior to beginning can save you many hours of frustration, as I found out the hard way. Unless you really enjoy backing up your hard drive and repartitioning two or three times you might consider getting a copy of Partition Magic or a similar program to save time. If you have already partitioned your drive and have several different OSes already, you are ready to roll—unless you are masochistic enough to want to install Windows 95 later. Suffice it to say, the manual devotes **MANY** pages to the installation "quirks" of Win95. Once you have your hard drive partitioned the way you want/need it though, the rest is a snap.

System Commander lives up to its promise of being easy to use and fairly intuitive. It also delivers on the promise to remove itself completely from RAM after your OS selection is made. While the specifics of how it does this are beyond the scope of this review, it works.

There are a few things though that disturbed me. The manual says nothing about backing up your original disk and using the backup for the actual installation. This should be obvious to most seasoned users, but it is particularly important in this instance. The installation process places System Commander on your hard drive and *saves the original boot record to the floppy for later restoration should you decide to uninstall*. I don't know about you, but in my opinion, anything that writes back to the disk needs to be run from a copy.

Another inadequate feature of System Commander is something called "user ID". This feature is supposed to let you leave personal information for later identification should the computer be stolen. The problem is that it is too easy to remove. The same problem occurs in the "password" protection routine. Supposedly, if this option is chosen, the system cannot be used without the password. Unfortunately, if you don't (or can't) set the CMOS to boot from the hard drive before the floppy, the entire point is moot—any old system disk will boot you up. Even worse, if you set the password and forget it, it is going to cost you to get access to your system again. V Communications has a "special" service for getting you back into the system. You best have your dialing finger and credit card ready though before you call. The upshot is, most people will

figure it's not worth it to turn on the password function, which totally defeats its purpose.

Another tiny glitch I ran across involves warm boots. While I know most people don't do some of the things I do with my system (I **REALLY** like to test things out), if you start the system and then give it the CTRL+ALT+DEL salute while the selection menu is on the screen, the system locks totally. Not a problem for most people, but it leaves a little tiny nagging doubt in the back of my head: “If this locks the system, can anything else do it too?” Fortunately, it didn't happen in any other situation.

Now, back to my earlier statement. System Commander really does what it says, somewhat. The advertising expounds the ability to run up to 100 OSes on the same computer. However, this is true only if you count MS-DOS 3.0, MS-DOS 3.1, MS-DOS 4.0, 5.0, 6.0, PC-DOS 3.0, 4.0, 5.0, DR-DOS 5.0, 6.0, etc., etc., etc., as separate operating systems. If I were to be generous, I might say that MS-DOS, DR-DOS and PC-DOS are different operating systems, but not the different versions of each. You also have to count Win95 (English version), Win95 (Spanish version), etc., as separate operating systems to make the count reach 100. But then, who would want two, (or, shudder at the thought, THREE) copies of Win95 on the same computer? I guess if you want to get technical about it, it does run 100 OSes, but most of us will probably not have more than two or three.

So, what are my overall thoughts on this program? Well, as for what it claims to do, I give it a big thumbs up. As to its practical uses, I give it a conditional thumbs up. If you regularly use two widely different operating systems, you may want to consider this system. It would probably be very useful to someone who uses Win95 and another OS. I would also declare that it might prove useful to those who do a lot of cross-platform programming. However, I am a bit turned off by the price tag. If you use only Linux and DOS for instance, you probably would be satisfied with LILO, and would consider System Commander to be LILO with eye-candy added. If you use OS/2 with DOS 6.x, you might find it interesting. If you use Win95 with older DOS versions, you would probably find it useful. I just have a hard time with the suggested list price of $99. I'm sorry, but I'm a cheapskate at heart. If the list were more like $30 or even $40, I wouldn't hesitate. Perhaps those who use operating systems such as Solaris and Xenix or QNX wouldn't mind the price, if it saves them some trouble. Being unfamiliar with those systems, I can't say for sure. The bottom line is, how much is it worth to you? It is easy to use. I'd check my other options first, and if they were just too much trouble, then I'd consider System Commander.

Terrence Miller is a part-time hacker/breaker of computers who drives a truck in his real life. (Yes, some truckers do have cognitive powers!) He is the ultimate cheapskate when it comes to buying things except when he really wants them!

Archive Index Issue Table of Contents

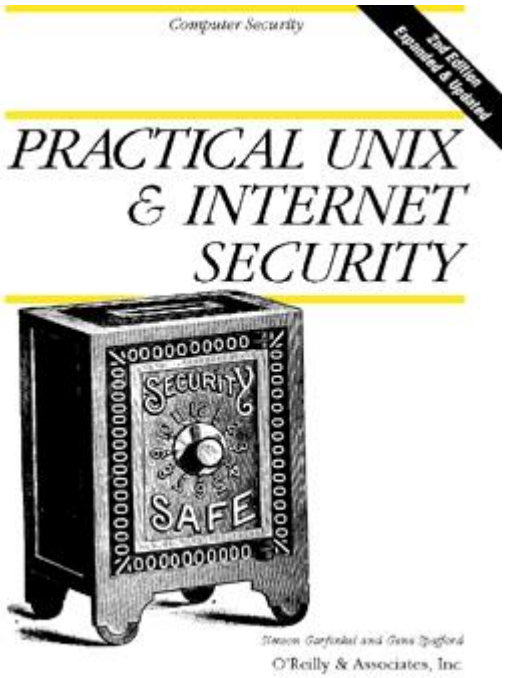
Advanced search

Practical Unix and Internet Security, Second Edition

Dan Wilder

Issue #34, February 1997

This book amounts to a survey course that covers everything from the basics to the advanced topics, not always to equal depth, but more than well enough to get you started in the right direction when you need to tackle a new topic or one you are a little rusty on.



- Title: Practical Unix & Internet Security, Second Edition
- Authors: Simson Garfinkel and Gene Spafford
- Publisher: O'Reilly & Associates, Inc. 1996
- ISBN: 1-56592-148-8
- Pages: 971
- Price: US $39.95, CAN $56.95
- Reviewer: Dan Wilder

Practical Unix & Internet Security is the much revised and enhanced second edition of O'Reilly's familiar *Practical Unix Security.* This book amounts to a survey course that covers everything from the basics to the advanced topics, not always to equal depth, but more than well enough to get you started in the right direction when you need to tackle a new topic or one you are a little rusty on. You may need additional resources. The 47 pages of well-annotated bibliography and resource lists found at the end should help greatly with further reading, contacts and so on.

The emphasis is on understanding principles. To the reader is left the exercise of implementing an appropriate and well-designed security strategy. This is good, for there are not likely to be many cut-and-dried approaches of any generality worth their salt. A useful strategy may vary a lot from site to site, and from time to time. No one book could even hope to cover in great detail all situations and all varieties of Unix. This book sets out to furnish the broad background and the perspective necessary to such an undertaking. I believe the authors have risen well to their task. Notwithstanding the overall very general approach, many detailed examples of procedures are given, to illustrate and anchor the discussion, and to give the Unix novice a place to start on this rather complex topic. Too many books about computing never descend to a concrete plane, and their lessons may be lost on just those who could benefit most. This book avoids that pitfall. Indeed, the experienced reader will likely skip many of the examples, while appreciating the insights and points of interest found between them. The beginning of the book talks about Unix basics: the file system, permissions, devices, users and passwords. Much of these several chapters is spent talking about organizational issues as well. This is appropriate, as Unix security is not merely a technical issue, but has a substantial social dimension. Accordingly, policy, history and risk assessment are treated briefly here. The common sense approach taken is exemplified by this passage (page 44):

The key to successful risk assessment is to identify all of the possible threats to your system, but only to defend against those risks which you think are realistic threats. Chapters on advanced topics include:

- Cryptography
- Telephone Security
- UUCP
- TCP/IP Networks
- TCP/IP Services
- WWW Security
- RPC, NIS, NIS+ and Kerberos
- NFS

- Firewalls
- Wrappers and Proxies
- Secure SUID and Network Programs

as well as a whole section on handling security incidents. The appendices have some very nice security checklists.

I found most of the information presented accurate and up-to-date, though of course not always complete. There were some exceptions. For example, in the chapter on UUCP (page 421):

UUCP was designed and optimized for low-speed connections. When used with modems capable of transmitting at 14.4 Kbps or a faster rate, the protocols become increasingly inefficient.

The authors are perhaps correct about historical UUCP, which unfortunately represents what is currently shipping from many vendors. A modern UUCP, such as the Taylor UUCP present in most Linux distributions, will give even the fastest competing file transfer methods a run for their money, over the same connection. If it were not for the negative tone about UUCP the authors take in this chapter, I'd chalk this one up as information that just didn't make the cut. In a later chapter, while offering alternatives to the expense of installing and maintaining a firewall, the authors finally touch on the tip of the UUCP iceberg (page 668):

Use a hard-wired UUCP connection to transfer email between your internal network and the Internet. This connection will allow your employees to exchange email with other sites for work-related purposes, but will not expose your network to IP-based attacks.

Bingo! This is one of several reasons why many businesses and individuals in the Seattle area, and no doubt elsewhere, use UUCP for some or all of their e-mail service. Too bad it didn't rate mention back where they were discouraging us from even considering UUCP. An amusing comparison of Linux and the GNU utilities with some others is found on page 704. The authors cite a study using a program called “fuzz” in which Unix utilities crashed when presented with random inputs. Over a quarter of standard Unix utilities crashed, while less than a tenth of Linux (mostly GNU) utilities tested did so. Though all the commercial vendors were presented with the results of these tests, a re-test some years later gave similar results. While as much as one utility in ten is still pretty high, it is a testimony to free software and GNU in particular that the levels attained are significantly lower than those in commercial systems. The implication drawn, however, is less amusing. The authors point out, correctly, that many of the same problems that will make a program crash on random

input, will allow a skilled attacker who is adept at exploiting the mechanisms of the crashes, such as buffer overflows and array bounds violations, to obtain behavior from a program not anticipated by its authors or installers. Woven through this work are discussions of software quality, an issue dear to my heart. Garfinkel and Spafford touch on these in the introduction (pages 17-18): @quote:[...] software designers are not learning from past mistakes. For instance, buffer overruns ... have been recognized as a major Unix problem for some time, yet software continues to be discovered containing such bugs, and new software is written without consideration of these past problems [...]

A more serious problem than any particular flaw is the fact that few, if any, vendors are performing an organized program of testing on the software they provide ... few apparently test their software to see what it does when presented with unexpected data or conditions.

In the chapter “Writing Secure SUID and Network Programs” they spend much more time on this theme. Lists of good, basic, common-sense rules are found there, such as “Don't use routines that fail to check buffer boundaries when manipulating strings of arbitrary length.” Violations of this rule alone have resulted in several CERT advisories, including, I suspect, a very recent advisory concerning a popular e-mail transfer program. Many other guidelines found in this chapter could in the past have prevented a number of serious breaches of security. For example, “check all return codes from system calls” and “Using the access() function followed by an open() is a race condition, and almost always a bug.” Perhaps one of these guidelines will help me, some day soon. I think I'll put this book on my night stand, for evenings of enjoyable late-night study as the rainy season moves in.

Dan Wilder writes and enjoys the rain in Seattle, Washington. You may reach him via email to dan@gasboy.com.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

CGI Programming

Reuven M. Lerner

Issue #34, February 1997

A quick guide to how CGI works and why it's a nightmare to debug.

How often has this happened to you: You finish a wonderful CGI program, place it on your server, fire it up, and...much to your chagrin, you find your browser telling you that there has been a “server error”, rather than the expected output from the program?

There is no shame in having this sort of thing happen; experienced CGI programmers see this message plenty of times each week. And while experience might reduce the number of errors a programmer makes, more often it teaches him or her how to deal with such problems.

This month, we will look at all the things that need to be in place so that a web server can run your program. These problems tend to cause programmers an awful lot of trouble, in part because they change the programming paradigm we are used to from our other programming experience.

We Don't Run Things Directly

Most programs are run directly, invoked from the command line (or a graphical user interface), and then displayed on your screen. But CGI programs are different; they are almost always run on a different computer from the one at which the user is sitting. Worse, CGI programs are twice-removed from the user —the user tells the browser what he or she wants to do; then the browser has to pass that data to the server, which then passes the request (and any arguments) along to the program. Thus part of the secret of debugging CGI programs simply involves understanding that the *server* is invoking the program.

Just like all of the other programs on your system, CGI programs must be executable. This isn't a big deal to remember when you are coding in C and C+

+, since gcc and other compilers produce an executable file that has the correct execution bits turned on. But for those of us who write most of our programs in Perl (and similar languages, such as Tcl and Python, which make use of the Unix shell's interpreter), we need to make sure that the programs are executable before they will run. This might sound obvious, but believe me—from the e-mail I get, I can assure you that many people writing and installing CGI programs forget to make sure that their program is readable and executable not only to themselves, but also by the user ID under which the web server is running.

So, if you are getting an error message when running your CGI program, go to the directory in which it is located, and use the Unix “chmod” command to make it executable. Since you want to make it executable to all users on the system—a good idea, given that web servers often run under the “nobody” ID—we can issue the following command:

```
chmod a+x progname
```

Thus if my program is called “progname”, we can double-check to make sure that the program is executable by issuing the **ls** command with the **-l** option, indicating that it should provide information in the “long listing format”. This means that we should like to see information about the file's permissions, ownership, and modification dates, as well as its name. If we list our program, we should see something like:

```
-rwxrwxrwx 1 reuven reuven 12779 Sep 5 15:28 progname
```

The above means that “progname” is owned by the user named “reuven” and the group named “reuven” (not unusual on my personal Linux box!), that it was last modified on September 5th at 3:28 p.m., and that it is 12779 bytes long. But most important is the fact that we see the string “rwx” repeated three times—first for the user (“reuven” in this case), then for the user's group (again, “reuven” in this case), and then for everyone else on the system. Thus the three instances of “rwx” mean that anyone on the system can read, write, and execute progname, which is precisely how we want things to be.

Running from the Command Line

If the program is executable and still gives you problems, your next step should be to run the program from the command line. As I said earlier, we always have to remember that the web server (most likely a different machine than ours) is executing the program on our behalf, which means that we usually see not the program's output, but rather the results of what our browser does with that output! It's generally easiest to debug when we know exactly what the program

is doing, and thus it can only help to execute the program on the web server itself.

So if the program is called “progname”, change into its directory (using the **cd** command) and type:

```
./progname
```

Why precede the program's name with the **./**? Because your shell's **PATH** variable usually won't—and probably shouldn't—include your system's CGI directory, which means that were you to simply type the program's name, your **PATH** might not work. Unless you already include **.**, a.k.a the current directory, in your **PATH**, which is extremely convenient but also a potential security risk. Suffice it to say that the above recipe always works, while leaving off the **./** won't always do the trick.

When you execute the program, what do you see? Most important of all is that a MIME header be the first thing the program sends to its output. Browsers depend on MIME headers to know what sort of data the program will be sending its way, and without such a header, you will almost always get an error from your browser. Why? Because if the browser expects to receive a MIME header, but instead receives an HTML-formatted response from your program, it will usually not know what to do. Rather than display data incorrectly, your browser simply says (in its own terse way) that it didn't understand what sort of data it should expect. It's up to you to make sure that under all circumstances, anything sent to standard output is preceded by a MIME header.

While there isn't enough space to go into the intricacies of MIME headers this month, suffice it to say that unless your program is going to output something other than HTML, you should always make sure that the first thing it prints is:

```
Content-type: text/html
```

Also note that this string should be followed by two newline characters (**\n** in C and Perl), rather than the one that we are used to placing after text strings. That is, Perl programmers should write:

```
print "Content-type: text/html\n\n";
```

while those of you (wisely) using the CGI.pm module in Perl should be able to do the following:

```
use CGI;         # Bring in the CGI module
my $query = new CGI; # Create an instance of CGI
print $query->header("text/html"); # Output the header
```

Why two newline characters? Simply put, those two newlines separate your program's headers (i.e., information *about* the response) from the content that it returns (i.e., the response itself). There is at least one major precedent for this system that you probably use all of the time—e-mail! If you ever look at an e-mail message as transmitted via SMTP (the Internet's Simple Mail Transfer Protocol), you will see that the message's headers and contents are separated by a blank line.

What else should you look for when running your program from the command line? If your program is invoked using the **GET** method (i.e., if it is invoked as though it were a document, without any arguments beyond perhaps some on the command line), then invoking it from the command line should produce some output, even if the output will be a bit weird because of the lack of any input arguments. However, if your program looks at the environment variable **QUERY_STRING** as a source of input, then you can easilyset that value from the Unix shell. If you're using bash or some similar sh/ksh variant on your system, then you should be able to type:

```
export QUERY_STRING="hello+there"
  ./program-name
```

Note that while you don't have to write characters in **QUERY_STRING** in the Web's “percent-hex” encoding scheme (in which odd characters are replaced by a percent sign, followed by the hexadecimal number representing their ASCII code), I suggest that you do, if only to test your program in an environment as close as possible to the one in which it will be invoked. Remember that if we want to know why things are going wrong, we need to pretend to be the web server invoking the program—otherwise we might miss out on a subtle bug that has to do with the encoding (or lack thereof).

Testing out programs that use **POST** is a bit trickier, since the name/value pairs are difficult to emulate when you are invoking the software on your own. But luckily for those of us using Perl, the CGI.pm module allows us to run our programs from the command line without too much trouble. If you invoke a program that uses CGI.pm, you'll see:

```
  ./program-name
(offline mode: enter name=value pairs on standard
input)
```

At this point, you can enter all of the names and values of the elements in the HTML form that is supposed to invoke the CGI program, followed by control-D. Of course, you don't necessarily have to enter all of the form elements, particularly if you're testing a large form—after all, debugging is supposed to *save* you time! But it is certainly convenient to be able to type:

```
name=Reuven
email=reuven@NetVision.net.il
address=17+Disraeli+Street
equation=2%2B2%3D4
<control-D>
```

and see the results of my program with that input, rather than having to add lots of print statements to my program—a tried-and-true system, and one which I certainly use, but this is often much easier. Note that in the above examples, my address has been entered with plus signs between the words (since the space character cannot be transmitted as part of a URL), and with the plus sign (+) and equals sign (=) encoded in percent-hex format.

Since I'm already talking about Perl a bit, let me remind all of you Perl programmers to use the **-w** flag when running your programs! The warnings it produces can be invaluable, especially when coupled with the "diagnostics" package that describes problems in more detail. You should also seriously weigh using the "strict" package, which forces you to declare variables a bit more formally than many Perl programmers are used to doing—but the trade-off is that the Perl compiler is then able to identify potential problems as it is looking at the program. Thus most of my CGI programs have the following in their first few lines:

```
/usr/local/bin/perl5 -w
use strict;          # Interpret variable and
                     # subroutine names strictly
use diagnostics;     # Display documentation
                     # regarding each warning
                     # and error.
use CGI;             # Bring in the CGI module
```

Was the Directory Configured for CGI?

By this point, we know that the program is executable, that it sends the correct MIME header before any other output, and that it seems to work in at least a basic way if invoked from the command line.

So assuming that there aren't any logic errors—which are still a significant hurdle to overcome, even when all of these logistical errors are out of the way—we should make sure that the program is running in a directory that has been configured to run CGI programs!

First, let's think about how most programs work: When you request a document from a server, it generally makes sure that the file exists. If the file does exist, the server returns the document to the browser; if it doesn't, the server returns an error message indicating that the file wasn't found.

CGI programs work slightly differently, though—in this case, the server returns any output *returned* by the program, rather than the program itself. Thus the

server attempts to execute the file as if it were a program, and then takes whatever that program returns and passes it on to the user.

Now the question: How does the server know which files it should simply return, and which should be executed first?

On most systems, that distinction is made by assigning several directories to be for CGI programs. These directories, traditionally called “cgi-bin”, contain the binaries (i.e., the executables) for our CGI programs. Anything contained in these directories is considered to be a program, and is executed; anything sitting elsewhere is considered to be a text file. Thus if your program is sitting in a non-CGI directory, its contents will be sent to the user as if they were a text file, displaying approximately what you would see if you were to use the Unix **more** command. By the same token, if you were to put one of your HTML documents in the CGI directory, requesting it would almost certainly cause an error, since the server would attempt to execute it!

Another approach, which is less common, allows users to place CGI programs anywhere on the system, so long as their filenames have certain suffixes—for example, **.cgi** or **.pl**. The good news is that this allows users to have their own CGI programs without forcing the system administrator to configure a new CGI directory each time a user wants to have a counter, or other common CGI program, on his or her home page. However, CGI programs can be a security risk, particularly when any user on the system can write or install whatever programs he or she might want. Most system administrators prefer being bothered with requests to create new CGI directories to the potential security holes that might exist with the other approach.

In either of these cases—the cgi-bin directory approach or the “any directory” approach—users who experience problems with their CGI programs should make sure that their programs are in the right directories, that the CGI directories are set up for CGI programs, and that they have the right suffixes. I often get e-mail from people who wonder why they get the error **cannot perform POST method to the non-script** whenever they try to execute their CGI program. What this admittedly terse and cryptic error message is trying to say is, “You tried to send data from an HTML form to a file that doesn't appear to be a program. If you want to set this up as a program, make sure to define it as such on your system by using the appropriate configuration!”

The solution, then, is to make sure that your program is in one of the directories declared to be for CGI programs. Your system administrator should be able to do this by using the **ScriptAlias** directive (for NCSA httpd and Apache), or the **Exec** directive (for CERN httpd). Once that is done, your program should run—although once again, I should stress that just because all

of the above steps are in order doesn't mean that your program will work! It just means that your server will be able to run it, and that the program will return at least some results to your browser.

Next Time

That's all the space we have for this installment. Next month, we will discuss the httpd error log, to which our programs send all of their error messages and warnings when things go wrong. We will see how to read the error log, and how to understand what it is saying when our programs don't work when we expect them to—as well as how to use it to our advantage to make sure that our programs work correctly.

Reuven M. Lerner has been playing with the Web since early 1993, when it seemed like more of a fun toy than the world's Next Great Medium. He currently works from his apartment in Haifa, Israel as an independent Internet and Web consultant. When not working on the Web or informally volunteering with school-age children, he enjoys reading (on just about any subject, but especially computers, politics, and philosophy—separately and together), cooking, solving crossword puzzles, and hiking. You can reach him at reuven@the-tech.mit.edu or reuven@netvision.net.il.

Archive Index Issue Table of Contents

Advanced search

Writing CGI Scripts in Python

Michel Vanaken

Issue #34, February 1997

This article is neither a Python tutorial nor a CGI tutorial, but a “Python Presentation from a CGI perspective”.

The Python Reference Manual abstract describes Python as:

> A simple yet powerful, interpreted programming language that bridges the gap between C and shell programming, and is thus ideally suited for “throw-away programming” and rapid prototyping. Its syntax is put together from constructs borrowed from a variety of other languages; most prominent are influences from ABC, C Modula-3 and Icon ... Python is available for various operating systems, amongst which are several flavors of Unix (including Linux), the Apple Macintosh O.S., MS-DOS, MS Windows 3.1, Windows NT, and OS/2.

It should also be noted that Python is an object-oriented language. You can write classes like in C++ or Java. I use Python every time the common guy would use Perl [“common guy”? Sheesh!—Ed, who is a die-hard Perl fan]. It has about the same functionality, while being far more readable. But it should not be restricted to a scripting language—a lot of people are using it for complete applications. It's also a perfect glue language, like Tcl, because it's easy to add new modules (written in C) to it. It can also be embedded in C applications.

Listing 1 shows my very first Python script. It's still used on a file server at the office. It deletes ~*.tmp left everywhere by buggy MS Windows applications. It's not really the common Hello World program (we'll see one later), and maybe it's not the most efficient way to do the job, but it demonstrates several features of the language:

- **Variables**: Variables don't have to be declared.
- **Recursivity**: See the **ScanDir()** function.

- **Platform independence**: The os module provides constants for current directory, parent directory, and so on. See os.curdir, os.pardir...
- **for statement and lists**: In Python, for works differently than in C. **os.listdir()** returns a list of files. For example, in Listing 1, at each iteration of:

```
for p in files
```

p will become the value of the next element in the list. So, if **files** is a triplet with values:

```
['lib', 'include', 'src' ]
```

The first time, **p** will be **'lib'**. At the second iteration, it will be **'include'**, and so on, until it has gone through the whole list.

- **Arrays and indices**: An array can be referred with one or two indices. One index is used to get a single element from the array (like in C). Two indices can be used to get a subset of the array. The first index gives the *from* element, while the second one gives the *to* element. For example, if *a* is an array containing the values **'abcdef'**, **a[2 : 4]** will return **'cd'**. There are defaults for both indices: they default to ***from the start*** and to ***to the end*** respectively. Examples: **a[2 :]** will return **'cdef'**. Negative indices can be used to count from the end; **a[-2]** will return **'e'**. See the *Python Tutorial* at the Python web site (http://www.Python.org/) to learn more about arrays.
- **Blocks**: Unlike C or Pascal-derived languages, there are no **Start-Block** or **End-Block** separators. Python works only with indentations (and the ":" character).

One of my colleagues in the firmware department recently had some problems debugging a TCP/IP application he is writing. There is a server application running in an embedded system, and a client application running on a PC. He was stuck for two days with a protocol problem, and didn't even know if the problem came from the client or from the server. Every test version meant recompiling, eventually downloading the code in the embedded system, and so on. In addition, it's not always easy to debug a device that doesn't even have a screen—you get the point.

So, after we discussed his problems, I decided to write little Python test programs to test his applications. In less than a quarter of an hour, I had tested his server application. This included writing a Python script and running it on a console of a Linux box, concurrently to tcpdump. Since the problem didn't come from the server, I wrote another program to test his client application. This script masqueraded the server, and we immediately discovered the

problem. My colleague was very impressed by the short time it took me to write those two scripts, so I gave him a copy of the *Python Tutorial*.

Some simple scripts using sockets can be found in Listing 2a and2b. They are from the *Python Library Reference*.

My company sells time and attendance software in a client/server environment. Supported platforms include Unix and NT. The biggest problem with time and attendance is that, although general functionalities are the same for all our customers, they all have special specific rules. That's why the software department is considering the inclusion of the Python interpreter in their software. It would allow on-site customization, and it is available on all our platforms.

Documentation and Availability

- The Python home page is located at http://www.python.org/. On the site there is a list of mirror sites, and the current distribution of Python.
- A tutorial and other documents including the Language Reference, Library Reference, a guide on how to extend and embed the interpreter and a FAQ can be found in the doc directory of the Python Home Page (http://www.python.org/doc/).
- Two books will soon be available about Python:
  - *Programming Python*, by Mark Lutz, O'Reilly and Associates Publishers.
  - *Internet Programming with Python*, by Aaron Watters, Guido Van Rossum (the author of the language) and James Ahlstrom, from MIS Press/Henry Holt Publishers. See http://www.python.org/python/arwbook.html.
- And finally, there's a newsgroup devoted to Python: comp.lang.python.

CGI Scripts

In the following text, I will assume that you run your own HTTP daemon locally. My preference is Apache, but any server will do the work, if properly configured.

And of course, you should have installed Python on your system. You'll need to configure it to use the gdbm module, since it's used in count.py.

For the examples of scripts which interface with a relational database, I've used PostGres95 (and its contributed Python module, PyGres95). PostGres95 is

available from http://www.ki.net/postgres95/. PyGres95 is available from http://zen.via.ecp.fr/via_dvpt/products/pygres.html.

To understand the following text, you should know how to write an HTML page, have a general idea of how CGI works, and have a little background with C programming.

Common Scripts

Listing 3, helloworld.py, is our first script. It's very simple. Run from the command line, it will print an HTML document. But you should copy it to your cgi-bin directory, then call it from your browser with the URL http://localhost/cgi-bin/script.py.

This script displays a little message and the local time. Here, you need to note only one thing: the script must send a header describing the contents of the document. This is done by the means of the **Content-type** header. Common values include **text/html**, **text/plain**, **image/gif** or **image/jpeg**. The header is terminated by a blank line. It is used by the client browser, and won't appear in the generated page. And, as you'll see, the script is executed, and not just displayed in the browser. Everything printed to sys.stdout by the script will be sent to the client, while error messages will go to an error log (/usr/local/etc/httpd/logs/error_log, if you are using Apache).

Listing 4 is the well-known Count script written in Python. This is used to display a graphical counter of the number of times that a particular page has been accessed.

This script imports a module called cgi, which I'll describe later. It's used to retrieve the URL parameter passed to the script. This script interfaces with gdbm (which must be included in the modules list when Python is configured) to store **{ URL ; access count }** couples.

This is our first introduction to Python dictionaries. A dictionary is generally referred to as an “associative array” in the literature. It means that you can access arrays by keys instead of indices. For example, if you want to handle an e-mail address book, with couples like these:

```
"Michel", "Michel.Vanaken@ping.be"
"Veronique", "Vero@home.sweet.home"
```

Here is how you should retrieve the address of Michel in C and in Python:

```
struct {
        char    *key ;
        char    *addr ;
} email[ MAX ] ;
int     i ;
```

```
for( i=0 ; i<MAX ; i++ ) {
  if( strcmp( email[ i ].key, "Michel" ) = 0 ) {
            printf( "%s\n", email[ i ].addr ) ;
            break ;
  }
}
if( i = MAX ) {
        printf( "Not found\n" ) ;
}
if email.has_key( "Michel" ) :
        print email[ "Michel" ]
else :
        print "Not found"
```

Adding an entry with Python is also very easy :

```
email[ "Homer" ] =  \
"HSimpson@Springfield.power_plant.com"
```

adds an entry if **Homer** is not a valid key, and overwrites the old value if it is already present.

We see that **Content-type** here is **image/x-bitmap** (since the browser is waiting for an **<img src=...>**).

Of course, the bitmaps aren't very pretty (I drew them with a paint package, saved them as xbm files, then used a lot of keyboard macros and M-Kill/Yank rectangles in Emacs). The goal of this script is not to reinvent the wheel, but to allow readers to compare it with other versions widely available on the Net in different languages.

In order to use this script, the gdbm database must be created. Change the current directory to your cgi-bin directory, run Python, and type:

```
import gdbm
gdbm.open( "counters.gdbm", "n", 0666 )
```

and exit Python with Ctrl-D.

It should also be noted that the xbm file created by this script is bad. It contains an extraneous byte (added in the **print_footer()** function), in order to simplify the **print_digit_values()** function (in this version, there are no tests for commas).

Debugging

Before putting your CGI scripts on-line, you should be sure that they're really clean, by testing them carefully, especially in near bounds or out of bounds conditions. A script that crashes in the middle of its job can cause large problems, like data inconsistency in a database application. You can eliminate most of the problems by running your script from the command line; then testing it from your HTTP daemon.

First, you have to remember that Python is an interpreted language. This means that several syntax errors will not be discovered until run time. You must be sure your script has been tested in every part of the control flow. You can do that by generating parameter sets that you will hardcode at the beginning of your script.

Then, be sure that incorrect input cannot lead to an incorrect behaviour of your script. Don't expect that all parameters received by your script will be meaningful. They can be corrupted during communication, or some hacker could try to obtain more data then normally allowed.

Listing 5 shows a different version of our Hello World script and demonstrates the following features:

- **Tuples**: Tuples are arrays consisting of a number of values separated by commas. Ouput tuples are enclosed in parenthesis. The localtime() function returns a tuple which can be assigned in one variable (that becomes a tuple). Or as in this script, individual elements of the tuple can be assigned at one time to several variables.
- **The elif ("else if") statement**: Listing 5 has two syntax errors that are not detected when the interpreter loads the script, but will crash it when executed. It will crash at Christmas, because there is a call to a Christmas() function which has not been defined, and it will crash again at the New Year's Day, because in addition to "Happy New Year!", it tries to print a "Max" variable which doesn't exist (due perhaps to a cut-and-paste from a script intended to wish someone happy birthday?). Here is what you'll find in the error_log file if the script is accessed on Christmas:

```
Traceback (innermost last):
  File "/cgi-bin/buggy.py", line 59, in ?
    Main()
  File "/cgi-bin/buggy.py", line 53, in Main
    Christmas()
NameError: Christmas
```

The fact that the script seems to execute normally (especially on New Year's Day, since everything that should have been printed is actually printed) can be a pitfall. The script has actually crashed!

Of course, in this script, crashing is not a big problem. But in an Intranet application, it could be very harmful. Imagine, for example, a script that displays a message saying it has updated your stock database, but has in fact crashed immediately after giving the message. The user thinks everything is going well, but the data have not been updated.

Let's get back to Listing 4. We've already seen that the generated xbm is not good; but maybe there are other problems. What happens if:

- The script is called with:

```
<img
src="http://localhost/cgi-bin/count.py?\
http://localhost/index.html">
```

  instead of:

```
<img
src="http://localhost/cgi-bin/count.py?
_url=_http://localhost/index.html">?
```

- The database file counters.gdbm does not exist?
- The access count exceeds 9999?

I suggest you try these, and try your own solutions. For the last situation in the list—the access count exceeds 9999—there are several solutions; I suggest modifying the DIGITS value if the incremented value in the **inc_counter()** function has a length that exceeds DIGITS. How would you see the generated file if your web browser displays nothing? Maybe you could add the following code, replace the call to **CGImain()** with **TSTmain()** and run the script from the command line:

```
def TSTmain() :
    #######
 url = "http://localhost/test.html"
 counter = get_put_counter( url )
 print_header()
 print_digits_values( counter )
 print_footer()
```

Form Handling

Listing 6 shows the HTML source for a form we are going to discuss for the remainder of this article. It allows the user to enter some values to perform a query on a database. The action parameter of the form should be adapted to your needs. For a real application, you should replace **localhost** by the fully qualified name of your host. The name of the script should also be adapted to call the right thing. Note that the HTML code defines a *hidden* field (**TableName**).

Let's start with a script that just echoes values entered by the user (see Listing 7). You'll see that, even if you leave the form empty, two parameters are displayed. The first one is (**TableName**), a hidden parameter in our form, and the second one is the value of the **Submit** button (which is also a field). Notice that:

- **CGI module** imported by our scripts is used to parse the input sent by an HTML form. It works with **GET** and **POST** methods.

- **cgi.SvFormContentDict()** builds a dictionary with:

```
        { field name ; field value }
```

couples corresponding to the data encoded by the user.

- **cgi.escape()** is used to convert special characters into their HTML escape sequence (for example, **<** becomes **<**).

Database Queries

Now we are going to look at more “real life” scripts that could be used in an Intranet application.

We are going to use PostGres95. It must be installed and configured properly. I won't explain that process here, since it would require a lot of additional text. But two things should be mentioned:

1. The “user” which is used when a CGI script is run on your system must have access to PostGres95, and to the database being queried.
2. The **connect()** function used in the following scripts may need to be adapted to work on your system. Mine doesn't need any parameters, since everything works with the default settings I've configured.

See the PostGres95 manual for more information.

The PyGres95 modules offer the same interface as the LIBPQ API, which is also described in the PostGres95 manual. You should know that there is a **connect()** function used to connect to the database, and a **query()** function that receives an SQL string as a parameter.

Listing 8 shows a script that will handle queries on a customer database which has a structure similar to what the fields might be in a query form. The script will connect to the database, build an SQL command, query the database, and finally, display the results in a table that is built on the fly for each request. Of course the SQL statements here are very simple, but scripts could be written to do anything.

This script is not very practical. We'd have to write specific code for every table we want to use. The script of Listing 9 implements a general query on any single PostGres95 table/view from an HTML form. This means that it will work for any query where you need a subset of a table. It could work for customers (as in our example), providers or articles. The main difference from the former script is the **build_query()** function:

The script now implements the following behaviour: a query made on a numeric field will require an exact match, while a query made on a text field will be considered as ending with a wildcard. This means that numeric fields are considered to be IDs, and that it's not possible, for example, to use it to search articles with a value between $500 and $1,000. But it can be used to search a personal database for all names beginning with “Van”.

Restriction: to determine the type of a field, we'll consider it numeric if its name ends in “num”. This is because all data sent to a CGI script is seen as text. Of course, you could parse the value to see if it's numeric or not. But it's not always a good choice. If you want to search for all telephone numbers beginning with “800”, our script will look for an exact match if it thinks it's a numeric field, and it will find nothing. Of course, you can also completely rewrite the **build_query()** function to fit your needs.

The script needs to know on which table it should perform the query. That's why our form contains an invisible field called **TableName**. It must be set to the name of the desired table.

The form field names must be the same as the table field names, because the script uses them to perform the query. But, of course, the labels displayed on the user input form can be anything.

And finally, the script contains several lines that can be commented or uncommented to enable or disable some debug strings in the resulting page (e.g., as the query string).

Where to Go from Here?

There are several powerful features of Python that weren't discussed in this article. Python supports exception handling, as in C++ or Modula-3. This can be useful to trap errors in CGI scripting. It's even possible to write a script with a function to send a bug report by e-mail to its author when it detects an unexpected error. And of course, you can write your own classes.

For CGI scripting, although we didn't use them in our sample scripts, some additional features are available. On the Python home page, you'll find code to embed the Python interpreter in Apache. And Apache itself comes with optional modules that interact with PostGres95. But PostGres95 is not the only database available—among others, there is a module for Oracle.

Now, if you want to try Python, the first thing to do is read the Python Tutorial (see Documentation and Availability), then print a copy of the Python Library Reference manual. Then, you should try to reach simple goals—like deleting all ~*.tmp files older than one day, for example.

Finding Python on the Web

1. http://www.python.org/
2. http://www.ora.com/
3. http://www.python.org/python/arwbook.html
4. news:comp.lang.python
5. http://www.ki.net/postgres95/
6. http://zen.via.ecp.fr/via_dvpt/products/pygres.html

Michel Vanakan is a 32-year-old software engineer and part-time network administrator. His interests include fantasy and Sci Fi books and games, walks in the wilderness and flights with light aircraft. He can be reached at Michel.Vanaken@ping.be.

Archive Index Issue Table of Contents

Advanced search

CGI: Safety First

Hans de Vreught

Issue #34, February 1997

A practical guide to safe CGI.

With the current Web hype a system administrator's nightmare has come true: every user on his system wants to have a home page and wants to do CGI programming. The urge to be on the Web is too strong to keep them off. If they had some elementary knowledge of security, things would not be so bad, but this is not the case: most of those users grew up with MS-DOS, where insecurity is a de facto standard. They have a dangerous attitude toward your system.

One major problem with CGI programming is writing to a file; the web server doesn't run as the user who wrote the CGI script—when it tries to write something in a file, it gets a permission denied error. Lowering the permission of that file isn't a solution—it is just plain disastrous.

As you might know, a web server (typically called **httpd**) can be run by any user. However, if you want to have the server listening on the HTTP port (port no. 80 according to STD 2, the Internet standard describing “Assigned Numbers”, at this moment, RFC 1700), it must be started up as root. Normally this is done at boot time from one of the rc files located in /etc. Since running the web server as root is a serious breach of security, all web servers can be configured to run as a different user, say user “www”.

You normally would see one server, running as root, which is responsible for opening port 80 and forking off a copy of itself that is running as www. It starts up as many servers as are needed to handle your load (you can often set a minimum and maximum number of servers in the configuration files). The one running as root does no HTTP stuff—that is done by child servers running as www.

There is one inherent security risk in running CGI scripts when everybody can write CGI scripts: the scripts can be read by other users. Even if you remove

read permission of others and make the script readable and executable by the server's group only, another person can write a script reading the source of your script. If it is important that the source of the program remains secure, you shouldn't be writing a script. Compiled programs don't have to be readable to be executable, whereas scripts must be both readable and executable.

Before we proceed with the real issue, just consider a few of the consequences of the server running as www. Consider the following script:

```
#!/bin/sh
cp /bin/sh /home/cracker/bin/sh
chmod 4755 /home/cracker/bin/sh
cat << EOF
Content-type: text/plain
There is no such thing as a secure server!
EOF
```

You now have created a suid www shell that any cracker can use! For this reason it is important that the user www not have a home directory and not own any important files. Another infamous trick is to send a STOP signal to all servers; that is really sneaky (do you know why?). If you find these kinds of scripts unacceptable, you should ban CGI programming for everybody on your system except a few trusted users.

It should be clear that running all your servers as root is a definite no-no. It is like removing the password entry of root from the password file. If all your servers run as root, dash to your system administrator to correct this and don't leave before it is corrected. If he is hard to convince, well, you have root privileges and maybe you should give him a demonstration of how you can correct his mistake!

Bad Things

The trouble starts when user foo wants to write something in a file. Well, just for the sake of argument we will increment an access counter in a file ~foo/www/access. For example, look at the script counter.sh (it is not very customary to use a Bourne script for CGI programming, but it does the job):

```
#!/bin/sh
ACCESS=
ACCESS=
rm /home/foo/www/access
echo $ACCESS > /home/foo/www/access
cat << EOF
Content-type: text/plain
You are number $ACCESS to visit this page!
EOF
```

A nice little script with one problem: it does not work. It would have worked if foo had executed it, but alas, www was the one who ran the script. User foo might then immediately lower his security by entering: **chmod 666 access**. And

since he noticed his default permissions are always wrong, he makes sure he regularly does a **chmod 666 ***--being unaware of the option of setting his UMASK. Don't think this doesn't occur in real life—I've had several colleagues doing exactly that. Be assured, they were all dedicated MS-DOS users.

Some of them learned it is easier if he doesn't change the files individually, but rather the directories (I almost jumped out of my room when I heard this one): **chmod 777 .** He didn't realize the writing succeeded because the daemon was able to remove the file access. But not just www could have done it—anybody could have done it!

Quickly entering **chmod 1777** may seem the right thing to do. Don't you believe it. Indeed, it isn't *that* bad, but still, anybody on your system can remove that file with a CGI script.

Good Things

It's time for something less depressing. Can you do it right? Sure you can—it just takes a little bit of effort. The problem is you don't really want your CGI scripts to be executed by a stranger like www. You want to have your scripts running as yourself! For the execution of programs as another user, suid and sguid were invented.

If you make a program suid (or sgid), you run it as the user (or group) who owns the file. That was exactly what you wanted in the first place. So you make your scripts suid. And still under Linux it doesn't work. Why? Because suid scripts are insecure and the Linux kernel refuses suid scripts altogether (look at the comp.os.unix FAQ for some stunning ways to become root).

But suid programs are safe to use. It only means that you have to write a wrapper, a little C program, counter.c, which you make suid:

```
#include <stdlib.h>
void main(int argc, char *argv[])
{
   exit(system("/home/foo/www/bin/counter.sh"));
}
```

Does this suid stuff always work? No—many system administrators mount /home as nosuid, meaning the suid bit will never be honored. Why does a system administrator want to do that? Well, you noted the first script in the article that made a suid shell? That wouldn't have worked if /tmp was mounted nosuid. Many system administrators mount only those file systems as suid which must contain suid programs, mounting all others nosuid; for instance, /home doesn't need suid programs for the system to run, so mounting nosuid is a pre-cautious action.

But there is another way. It's a bit like shotgun programming, but you can let sendmail and procmail do the dirty work for you. How does it work? Basically you let the daemon send you mail that will instruct you to perform a certain task. The receiving sendmail will look in your ~/.forward and see that you want your mail to be processed by procmail, which will perform the update script for you. Let me first give you the CGI script:

```
#!/bin/sh
ACCESS=
ACCESS=
cat <<EOF
Content-type: text/plain
You are number $ACCESS to visit this page!
EOF
echo "Subject: access.sh 1928397071" | sendmail foo
```

First some remarks about the mail:

- This mail has no body, which is allowed by RFC 822 (the Internet standard describing the format of mail messages). If you want to include a body, you should precede the body by a mandatory empty line.
- In the subject we have a script part access.sh that will hint at what script to run. It has some sort of a magic cookie (1928397071) to prevent you from processing e-mail which is accidently triggered. If you want this to be a really secure cookie, you shouldn't be writing this line as a script.

Next it is time for the ~/.forward:

```
"|IFS=' '&&exec /usr/bin/procmail -f-||exit 75 #bar"
```

The program procmail needs to look in your file ~/.procmailrc (which shouldn't be world readable, if you want to use cookies for security):

```
LOGFILE=$HOME/log/procmail
:0 b
* ^From www
* ^Subject: access.sh 1928397071
|$HOME/www/bin/access.sh
```

So if the daemon sends a mail with a subject **access.sh 1928397071**, procmail will run access.sh for you:

```
#!/bin/sh
ACCESS=
ACCESS=
rm /home/foo/www/access
echo $ACCESS > /home/foo/www/access
```

Mission accomplished: we have written something safely into a file. Are there any catches? Yeah, sure.

First of all, you need to assure that when two people try to access your page, they won't screw up your files. You need some sort of file locking mechanism (check out **flock**, which also is available in Perl).

Secondly, the sendmail in the CGI script is nonblocking: it doesn't wait until the update is actually done. In most cases, this doesn't really matter, as you are only doing a simple update. Otherwise, you can always try to implement the Perl client-server model.

Don't go for the easy insecure option; keep it safe. Your system administrator will appreciate it.

Hans de Vreught (J.P.M.deVreught@cs.tudelft.nl) is a computer science researcher at Delft University of Technology. He has been using Unix since 1982 (Linux since 0.99.13) and is a profound MS hater (all their products are Bad Things). He likes non-virtual Belgian beer, and he is a real globe-trotter (already twice round the world).

Archive Index Issue Table of Contents

Advanced search

Letters to the Editor

Various

Issue #34, February 1997

Readers sound off.

Fixing PGP for Linux

I read your article in *Linux Journal* #32 about using PGP and came across the comment that building PGP from source for Linux was, well, difficult.

I have included a patch which will allow you to build PGP 2.62 on a Linux 2.0 box. (I'm running gcc 2.7.2, and a 2.0.25 kernel, so your mileage may vary with other setups).

Obtain the US distribution of PGP 2.6.2 from http://web.mit.edu/network/pgp.html (this may work for the international version, but I haven't tried it). Untar everything. Following the instructions in setup.doc, do the following:

cd rsaref/install/unixstrip pgp

Then do the tests setup.doc suggests.

There's definitely a cleaner way to do things, but I stopped playing with it once it worked. Sean C. Malloye-Net, Inc., smalloy@zilker.net

See Listing 1 for the code Sean included with his letter.

It was good to see Michael Johnson's introductory PGP article in the December issue (#32). PGP can indeed be a bit of a challenge to set up, as he notes. This is particularly true for people with ELF systems. I put together a tips page for my local Linux user's group when we had a meeting devoted to installing PGP and integrating it with other applications. It's at http://www.rust.net/~strix/pgphand1.htm.

You really don't need to be a cloak and dagger type to appreciate PGP. In a more mundane setting, I like the ability to keep things private (like course gradebooks) and discourage e-mail forgery (“emergency, exam cancelled...”).Regards, strix@rust.net

More FreeBSD?

In the December (#32) issue of *Linux Journal*, a reader asks whether it would be possible to see more FreeBSD-related articles. Your response mentioned an informal survey, with which I must say I'm unfamiliar, citing a lack of interest in the FreeBSD community.

I'm not sure how you were given such an impression, but I can certainly say that such a verdict is rather inconsistent with my own experience—we're always happy to see articles (or, indeed, entire magazine issues) devoted to our operating system and currently invest considerable energies in promoting it through magazine articles, public presentations and trade show appearances. I would be more than pleased to see the *Linux Journal* take a greater interest in FreeBSD and, until reading your response, was not even aware that such a possibility really existed.

Though some of the more overzealous users in both camps might perceive FreeBSD and Linux as operating systems in strong competition, this viewpoint is both short-sighted and foolish. The real dividing line here is free software vs. commercial software and open systems with complete source code vs. “open systems” which are open only in some marketing department's imagination.

Linux and FreeBSD have far more in common than the zealots would care to admit, and far more important than lineage is the fact that both provide users with something none of the commercial operating systems can match—a sense of actual control over the OS environment and the ability to learn from or change any aspect of it as they see fit, no longer tightly constrained by whatever arbitrary restrictions the manufacturer chooses to impose on them.

I hope that *LJ* does someday extend its reach beyond the implied focus of its title, making it clear the real fight lies not between FreeBSD and Linux but in combating market ignorance about the kinds of things of which free software is capable and how truly viable an alternative to the much more well-known commercial alternatives it can so often be.

The world of free software is only the richer for having multiple solutions to choose from, each with its own unique strengths and areas of coverage, and frankly, I wouldn't have it any other way. Jordan Hubbard jkh@FreeBSD.org, http://www.freebsd.org/

Where Is Linux?

Now more than ever, I find Linux to be the number one operating system of choice for several reasons. Not only is its stability far superior to some commercially developed operating systems; the variety of applications available for Linux is exploding like never before. Its practicality for Internet access exceeds that of any other OS for the PC, and more hardware than ever is supported by it.

I'm the president of the Computer Society at Western Connecticut State University. The goals of the Society are to enlighten and inform its members and to create a common ground on which computer students, professionals, and educators can convene. To meet ongoing demand, we hold Linux discussions and demonstrations as frequently as possible. In addition, Linux is now being used as the primary teaching tool for the Operating Systems course. Not only are computer majors, professionals, and hobbyists alike using Linux as a primary operating system; even those less technically inclined are finding Linux suitable for everyday tasks.

My only question has been and remains: Why are we still not finding Linux distributions on the shelves of local software stores? Currently, I see it widely available at trade shows, expos, conventions, the computer sections of local book stores, and several other locations the general public does not go for software. So, while Joe or Jill Average shop around for new software to try out on their computers, they never see Linux. Why hasn't some company put Linux on the shelf where the light shines? Cory Plock coryp@i84.net

More Praise for Debian Linux

Having just attempted to install Debian Linux, I read your product review on Debian 1.1 in the November *Linux Journal* (#31) with interest. My installation was plagued by problems with an inconsistent file system on my CD from iConnect, apparently as a result of “growing pains” in the process of upgrading Debian to kernel version 2.0. The problems gave me ample opportunity to correspond with the Debian “bugs” server and the folks who develop and manage several of the various packages which make up Debian Linux.

This led me to what I consider (in spite of my failed installation) to be one of the real strengths of Debian Linux—a feature not mentioned by the authors of the *LJ* article. Debian is obviously supported by many people who believe strongly in its value and genuinely want it to work for everyone. The Debian bug reporting system is well organized and efficient. All my questions and problems were addressed promptly by folks who not only sent me friendly and astute replies, but continued to correspond with me, and in a couple of cases, asked

for **my** input and observations. Debian, like GNU software and Linux itself, is truly a community effort.

This is in stark contrast to my experience with another popular distribution which I'm also using. After having SCSI host adaptor problems with the distribution boot disk (identical to one of the problems I had with Debian), I submitted a problem report to both the distribution author and the primary distributor. After several months, I have yet to hear from either.

The Debian folks at iConnect are sending me a replacement CD, and I'll certainly give the distribution another shot.Lindsay Haisley fmouse@fmp.com, http://www.fmp.com/

Linux for Secondary Schools

Linux would seem, in many ways, well positioned to satisfy the growing thirst for computer technology in the secondary schools. It's cost effective, it excels at networking, and the open orientation of the Linux community offers an antidote to both the Mac vs. PC turf wars and over-dependence on a few corporate providers. Perhaps more importantly, Linux offers the option of lots of cheap, reliable, quiet, easy-to-maintain, dumb terminals attached to a single server, vastly increasing the workstation-to-pupil ratio schools can achieve within their meager budgets. Such terminals may not meet our expectations for colorful graphics, but are quite adequate for teaching and using communications skills (word processing, e-mail, ftp), office use (records management), etc. Is anyone in the Linux community targeting this area? Jack McGregor jackmack@interserv.com

Graphics Are Important

Hello, *Linux Journal*! I've just read *LJ* issue 31 and was positively surprised that most of the articles were about graphics. “Multimedia” is important for the desktop market and the private user, but it was forgotten in former times. I found the article on GGI an important concept for Linux. It cannot be the future of Linux that all device drivers must be programmed by hackers on the Net—the manufacturers must be more involved in this part of developing. The GGI article mentioned an essential way to achieve this: the manufacturer can link his own optimized driver in object form into the system that provides a well-defined interface to communicate with the special type of device. I think this must be possible for any type of device, so the manufacturer has minimal work to provide a driver for Linux. Olaf Milbredt o.milbredt@gsn.pb.nw.schule.de

Archive Index Issue Table of Contents

Advanced search

Linux at DECUS and OSW/FedUNIX

Gary Moore

Phil Hughes

Issue #34, February 1997

Phil Hughes and Gary Moore attend the trade shows DECUS and Open Systems World/FedUNIX and bring back some notes from the front.

On November 11 through 13, Carlie Fairchild and I attended the DECUS show in Anaheim, California. While DECUS has generally been a good show for SSC, this show was small and we were the only Linux vendor attending. The best guess why is with UseLinux coming up in the same place in January, it was an easy show for people—vendors as well as Linux-heads—to skip.

There was a series of talks on Linux presented by Jon “maddog” Hall and myself. Attendance was between 20 and 50, and I think we managed to make some converts.

Carlie had also arranged for me to speak to the local Linux users' group on Wednesday night. About 25 people attended (including “maddog”). I presented a talk called *Looking at Linux*. Much of this talk focused on the commercial viability of Linux, which was an issue many of the group's members had been attempting to address on their own. I stressed four criteria for commercial viability: reliability, interoperability, support and capabilities.

The talk was well received and the meeting turned into a informal discussion of Linux in general. I look forward to talking with these people again during the UseLinux show.

And More for User Groups

By the time you read this I will have presented this talk again at The Evergreen State College for their Linux Users' Group. I am also being interviewed on a program called *Geek Talk* on their radio station. While I hardly consider myself

a geek, the last time I was on this station was 17 years ago, so I guess it is about time to be a radio personality again.

SSC's emphasis on users' groups is not new. We made LUG/nut, an inexpensive package consisting of a Linux CD and a card good for a sample issue of *Linux Journal*, available to users' groups over a year ago. We are now working on a way to help new users' groups form, get publicity and attract new members. Watch our web site for details.

—Phil Hughes, Publisher

Open Systems World/FedUNIX

The first week of November, I went to Washington, D.C. to attend Open Systems World/FedUNIX. While several dedicated Linux fans came by the booth, most of the people I talked to knew very little about Linux. Some were just cruising the booths, collecting whatever anyone was giving away, but we don't mind—the literature they picked up may spark some real interest later on. (One show attendee, in addition to taking a few of whatever we had also took the neat twirly thing we'd acquired from another exhibitor's booth.)

Linux vendors in attendance were Yggdrasil Computing, InfoMagic, and Red Hat Software, giving me a chance to meet Adam Richter of Yggdrasil, Bob Young and Lisa Sullivan of Red Hat, and Henry Pierce and Greg Deeds of InfoMagic.

Adding credence to Linux's worth in the minds of those with no free software experience was Digital Equipment's display of a DEC Alpha running Linux and maddog's enthusiasm for the operating system. (By the time I got over to actually see the machine, someone was demonstrating Quake on it. I sat down and showed him a couple things I remembered from playing Doom—it was kind of surreal to be sitting amidst all the professional frumpery of the show while virtually running around swinging a very large and lethal axe.)

Jeff Leyland of Wolfram Research, the makers of Mathematica, spoke about Wolfram switching to Linux as their development platform. There were other speakers I should have made time to hear, but I got caught up talking to people coming by our booth and asking about Linux. I knew that after a few talks, the Linux booths would get flooded with people excited to check it out.

I also heard that Ernst & Young—well-known for their accounting services among other things—apparently use Red Hat Linux in-house and have asked IBM, with whom they contract for computer services, to support their Linux machines. (If you're from Ernst & Young, please send me some mail. We'd like to hear about how you're using Linux.)

Adam Richter predicted a new version of Yggdrasil's Plug-and-Play Linux in the first quarter of '97. At OSW they had pressings of their new 8-CD Internet Archives set, which includes several distributions, including a couple I hadn't heard of before.

I would have felt cut off from the world (yes, even in D.C. on election night) if it hadn't been for David Lescher, who set me up with dial-in PPP access for my laptop; and David Niemi, who made some necessary tweaks to my chat script. I'm also grateful to Mark Komarinski, who put together a Linux talk on very short notice when I found I was dangerously close to having no time whatsoever to prepare one myself.

The Santa Cruz Operation was there giving away copies of their Free SCO OpenServer. Someone who'd just acquired one of those gems asked me why she'd be interested in Linux if she had OpenServer; I noted its limitations and handed her a copy of *Linux Journal*, hoping to plant a seed. Some attendees were being less subtle, affixing prominently to their big blue IBM literature boxes the Linux bumper stickers we were giving away.

—Gary Moore, Editor

Archive Index Issue Table of Contents

Advanced search

Practical Linux: A Bosnian Experience

John Gorkos

Issue #34, February 1997

The NES boxes use a firefly algorithm to encrypt unclassified data, pass it over the classified packet network, and decrypt it on the other side, where it gets routed onto the DDN and evetually onto the Internet.

I'm a U.S. Army officer stationed in Darmstadt, Germany and forward deployed to the former Yugoslavia. Soon after coming to Europe in 1994 I discovered the “Linux Revolution” and have looked for innovative uses ever since. My first opportunity was with my unit in Darmstadt: I converted an unused 80386 machine with 8MB of RAM and a 120MB hard drive into a router/mailer/WWW server. Its primary purpose was to route packets between the 10Mb/s Ethernet and a 19.2 kb/s SLIP link into a DDN termserver. In the 20 months since it came on-line, it has never failed to provide reliable routing and dependable e-mail service.

When the US Army deployed to Bosnia as part of the Implementation Force, we brought with us a complex, impractical data lifeline. The backbone of Army tactical communications is a system called Mobile Subscriber Equipment. It provides between 13 and 58 secure, encrypted digital voice trunks and 16-64KB of encrypted data trunks. Every headquarters has a communications shelter outside with a BBN T/20 packet switch that routes TCP/IP traffic onto an X.25 packet switch network, effectively creating a giant, secure Class B network spanning all of Northern Bosnia.

The unsecured, unclassified portion of the network is provided by Motorola Network Encryption System (NES) boxes scattered throughout the theater. The NES boxes use a firefly algorithm to encrypt unclassified data, pass it over the classified packet network, and decrypt it on the other side, where it gets routed onto the DDN and evetually onto the Internet.

The plus side of this arrangement is Internet access to the soldiers in Bosnia. Originally, we intended to use the system to pass logistical and other non-

sensitive data to the supporting elements in the rear, but it provided a way to pass non-sensitive data and morale e-mail as well. The pipelines are small, and the amount of data forced down them is huge, but it has proved to be an effective, if unwieldy system.

As the signal officer for one of the twelve US Brigades in Bosnia, I am responsible for the care and feeding of approximately 100 desktop and laptop computers, all running Windows for Workgroups with the Microsoft TCP/IP stack. The physical layout of the network at our Brigade Headquarters is quite tenuous: two 250m 10Base2 segments with 30 computers on one and 6 on the other, joined by an Ethernet repeater. In the rough and tumble military world, it's almost a full-time job keeping the cable and all connectors together, dry, and passing data.

The Linux solution stems from the need for a centralized SMTP gateway and information warehouse. The machine is a Pentium 166 sitting on an Intel Atlantis motherboard, with 512KB of cache and 64MB of RAM, with just under 2GB of storage. I'm running kernel version 1.3.100, completely ELF and modularized.

Before I came on the unit, Pegasus mail (a free POP3-based mail client for MS Windows) was used to retrieve mail from various e-mail hosts in Germany. Because of the low bandwidth, users were waiting 5-10 minutes to download just a few messages, and users' mail was “trapped” on the computer used to retrieve messages.

My solution involved the Samba package coupled with the security built in to Windows for Workgroups. I used Samba to export the /home directory of the server, and created user accounts for everyone. The Pegasus mail executables and library files were put in the /home directory with 640 permission flags and root:users as the owner:group. Windows users were then instructed to attach (in Windows parlance) the \\MAILSERV\MAIL share as their M: drive. The Pegasus initialization file required that I use a static drive label (M:, for mail) The user then created an icon in Windows pointing to the M:\WINPMAIL.EXE file.

Adding users was tedious until I broke down and wrote a Perl script to create the entry in /etc/passwd and personalize a generic PMAIL.INI file that was copied over from the /etc/skel directory.

The result is a secure, seamless e-mail server that allows users to log in using the familiar WfW login prompt and run a common e-mail client from any PC on the network. Pegasus uses POP3 to download mail from the Linux box, then stores the mail on the Linux box in the user's home directory. Outgoing mail is sent to the Linux machine for delivery; unreliable links and slow network

response made it too easy for a user to leave mail stuck in the outbound queue when he shut down the e-mail client.

One additional advantage of the Linux box on my network is that I can run my own name server. I began with Nicolai Langfeldt's caching named mini-howto and set all my Windows clients to point to the Linux machine as a primary DNS. I discovered that a significant chunk of my bandwidth was disapearing due to net surfers. I tried to limit the number of browsers my users had, but I found it was easier to send the browsers bogus IP addresses for certain “unsavory” sites that shouldn't be visited with a government computer. I told my named to return the IP of the Linux machine, and created a web page to scold my serious offenders. Everyone got a chuckle out of it, and I got my bandwidth back to a usable level.

I have also been able to provide a whois server for a sizable chunk of the deployed soldiers with e-mail accounts, and an intranet web site with photos and info pertinent to the command.

The Samba programming team has put together an excellent package that allows me to cater to the users' demand for a Windows operating system without tying me to the propriatary e-mail and server programs of Windows NT. I look forward to future releases of Samba that will include domain services and validation, and I'm always on the lookout for new ideas and applications for Linux.

1LT **John Gorkos** is the Communications officer for the 16th Corps Support Group. He hacks Linux when he's not making radio checks or splicing telephone cable. Since there's not much else to do in Bosnia right now, he divides his time between work, lifting weights, and sleeping. He can be reached by e-mail at gorkos@e-mail-bosnia.hanau.army.mil.

Archive Index Issue Table of Contents

Advanced search

New Products

Margie Richardson

Issue #34, February 1997

WebThreads 1.0.1, QUERYFLEX Report Writer, Linux Pro Desktop 1.0 and more.

WebThreads 1.0.1

WebThreads announced its WebThreads 1.0.1 web site interactivity and visitor tracking solution. WebThreads allows you to create truly interactive web sites that respond and change in real time to individual visitors based on their actions. It also provides the webmaster with a tremendous amount of information about what the visitors are doing on your site. WebThreads is a lightweight and easy to install set of CGI scripts written entirely in C that greatly enhances the flexibility, interactivity and visitor traffic reporting accuracy of standard WWW servers. It is available for Linux at a price of $895.

Contact: WebThreads, LLC, 1919 Gallows Road, Tenth Floor, Vienna, VA 22182, Phone: 888-847-3348, E-mail: info@webthreads.com, URL: http://www.webthreads.com/.

QUERYFLEX Report Writer

Infoflex announced the release of its QUERYFLEX report writer for Informix and Infoflex databases. It provides control over report formats, sort order, table join relationships, totaling and data selection. Prices start at $695 for the developer's version. QUERYFLEX is available for Linux.

Contact: Infoflex, 840 Hinckley Road, Suite 107, Burlingame, CA 94010, Phone: 415-340-0220, Fax: 415-697-7696.

Linux Pro Desktop 1.0

WorkGroup Solutions announced the shipping of Linux Pro Desktop 1.0 for an introductory price of $299. Linux Pro Desktop includes Linux Pro Plus, Common

User Environment (desktop applications), Accelerated X-server (version 2.1), Motif and other development tools, and more.

Contact: WorkGroup Solutions, Inc., Department WEX017, P.O. Box 460190, Aurora CO 80046-0190, Phone: 800-234-7813, Fax: 303-699-2703, E-mail: sales@wgs.com, URL: http://www.wgs.com/.

NDP Fortran for Linux

Microway announced the release of NDP Fortran for Linux, which runs on the Digital Alpha processor and all 21164-based motherboards. NDP compilers are known for fast numeric processing. NDP Fortran-Alpha for Linux is available for $1,200.

Contact: Microway, Research Park Box 79, Kingston, MA. 02364, Phone: 508-746-7341, Fax: 508-746-4678, E-mail: nina@microway.com.

Craftworks Linux/AXP 2.2

Craftwork Solutions has released Craftworks Linux/AXP 2.2 for DEC's Alpha chip, offering a solid operating environment for desktop workstation or web server solutions. It provides optimum processing performance and highly reliable, built-in security implementations. Linux/AXP 2.2 is available for $99.95.

Contact: Craftwork Solutions, Inc., 4320 Stevens Creek Blvd. #170, San Jose, CA 95129, Phone: 408-985-1878, Fax: 408-985-1880, E-mail: info@craftwork.com, URL: http://www.craftwork.com/.

Numerics and Visualization for Java

Visual Numerics, Inc. announced new products for assisting web developers in building network-centric applications with visual and numerical requirements. The first product is a set of Visual and Numeric Classes that build upon the Internet Foundation Classes (IFC) in the Netscape ONE open network environment. The second product is a numeric API for Java, called the Java Numeric Library (JNL), to support fundamental numerical methods on basic numeric objects. Pricing is available directly from Visual Numerics.

Contact: Visual Numerics, Inc., Web Products Division, 6230 Lookout Road, Boulder, CO 80301, Phone: 800-983-3947, E-mail: web_products@boulder.vni.com, URL: http://www.vni.com/.

InfoDock Linux Software Development Toolset

InfoDock Associates, a commercial support and development house for Emacs-related software, is pleased to announce InfoDock, a freely redistributable,

integrated information management, productivity and software development toolset. It is built atop XEmacs, but with an easier to use and more comprehensive menu-based user interface. Available in source and Linux ELF binary form from: ftp://xemacs.org/pub/infodock/, or on CD-ROM from Prime Time Freeware at info@ptf.com, or Yggdrasil Computing, info@yggdrasil.com.

Contact: InfoDock Associates, 4880 Stevens Creek Blvd., Suite 205, San Jose, CA 95129-1034, Phone: 408-243-3300, E-mail: info@infodock.com, http://www.infodock.com/.

Wabi 2.2 for Linux

Caldera has announced the availability of Wabi 2.2 for Linux. Wabi is the port of Sunsoft's Wabi technology to enable end users to run the most popular Windows 3.1 applications on Linux-based system software. Wabi 2.2 for Linux is available for $199 US. It requires a 386 or higher Intel-based processor, 16MB RAM, VGA-quality video and 10MB disk space.

Contact: Caldera, 633 South 550 East, Provo, Utah 84606, Phone: 801-229-1675, Fax: 801-229-1579, E-mail: info@caldera.com, URL: http//www.caldera.com/.

Archive Index Issue Table of Contents

Advanced search

Linux Gazette Tips and Tricks

Michael J. Hammel

Issue #34, February 1997

Tips from the Graphics Muse: web page fundamentals.

My first bit of musings is about the use of images in web pages. I get a lot of e-mail from people who've seen my web pages (or possibly my ramblings on various newsgroups or mailing lists) asking how to do *blah* or where can I get *blah* to do *blah* for their web pages. The “where” parts are in my HOWTO pages that can be found at: http://www.csn.net/~mjhammel/linux-graphics-howto.html and http://www.csn.net/~mjhammel/povray/povray.html.

The “how” part is a broad question. I'll summarize. When creating graphics for your pages, keep the following things in mind:

1. Consider your target audience—home users:

- Most home users have slow links. Even 28.8 modems don't load big graphics all that fast. Keep your images small.
- Animation done in the same manner as cell-animation for cartoons (sequences of individual images with slight variations to simulate movement) requires that each cell be loaded across the Net. This is tantamount to one big image taking forever to load.
- Most home users are still limited to 256 colors on their displays. Lots of users have upgraded to better graphics cards, but how many people do you want to alienate with an image requiring 10,000 colors?
    1. If you want the average person to visit your page, you have to provide two things: content and flash. The flash has to be done using as little download time as possible, with as much color as you can squeeze in without overloading the browser (causing it to dither images). The content, not the flash, must be the reason for your pages.

2. Background images should be just that—in the background. Don't make the background so gaudy that it distracts from your content.
3. Use common color maps—this reduces the number of colors the browser has to allocate, leaving some space for other applications. X-based systems can allocate colors into private color maps, but this causes that annoying “flashing” you see (try running Netscape with the **install** command line option, and you'll see what I mean).
4. Flash can be added easily with a simple background over which you add some in-line transparent GIFs.
5. Never use an “Under Construction” image. It's the Web. Of course it's under construction.
6. Don't put those silly graphic dots in place of HTML list bullets. First, they waste the user's time downloading (each requires another connection to the server), and second, they break the formatting rules provided with HTML. It's just not a good practice, and they don't add any real value to your pages.

Now that you know some basic guidelines for your images, how do you go about creating the images? It depends on what kind of images you want to make. If you want a simple, cartoon-like image you can get a drawing program like xpaint. This tool is good for drawing circles and boxes and filling them in, using a window like a canvas to paint on the screen. However, it is limited in what you can do to the image once you've drawn it. One highly popular tool for a number of platforms that provides post-processing capabilities is Adobe Photoshop. Using a tool like Adobe takes a bit of practice, but once you've mastered it, you can do some rather amazing things. A Linux alternative to Photoshop is The GIMP, which can be found at http://www.XCF.Berkeley.EDU/~gimp/. The logo on the Graphics Muse page in LG (Figure 1) was created with The GIMP; so was the background (Figure 2).

Figure 1. Graphics Muse Logo

Figure 2. Background, Graphics Muse Page

Three dimensional images are another matter. There are actually more well-known tools for doing 3D work than there are for doing image manipulation (which is handled by tools like The GIMP). Probably the best known of these is POV-Ray. This tool reads in a text file that uses a “scene description language” to describe how objects in the scene should be positioned and textured. The drawback to these tools is that they lack a point-and-click interface. There are separate tools, known as modelers, available that allow the creation of the scene files without actually rendering the image. In order to create a 3D image you need to either learn the scene description language or learn how to use a modeler that will create it for you.

Next month: How do you create the textures you apply to 3D images? Beyond that, I'm considering writing about how to use Type 1 fonts in your images: how to install them, how you can manipulate them with The GIMP to make interesting logos, etc. I'd also like to provide some tips for using POV-Ray and BMRT (although I have a lot to learn about the latter). And I might also cover how to do animation. Things are pretty open right now. Let me know what you'd like to hear about! (mjhammel@csn.net)

Archive Index Issue Table of Contents

Advanced search

Best of Technical Support

Various

Issue #34, February 1997

Our experts answer your technical questions.

Decreasing Partition Size

I recently got a new PC with Windows 95 installed. With my old PC I used FIPS to decrease the size of the DOS partition and then used Linux FDISK to create Linux partitions. Windows 95 uses VFAT. Do you know of a utility or product like FIPS I could use to decrease the size of the VFAT partition without deleting and adding new partitions? —Lanny Lampl

Partition Magic

There is a fairly inexpensive commercial product called Partition Magic by PowerQuest (**http://www.powerquest.com**). It is a very nice and easy to use program which will change the size of your partition without destroying the data. Of course, you must have sufficient disk space on the drive you want to resize but it is very painless. Another nice feature is that it can “see” Linux partitions as well (it can't manipulate these partitions but will at least let you see the sizes).

One problem with the current version is that you must do the actual manipulation from DOS (version 7.0 if you have Windows 95). You can see your partition information from Windows 95 but it will not allow you to make changes. —Douglas Stoun doug@igor.cmr.fsu.eduFlorida State University

Setting Up tin NNTP

Regarding tin NNTP: I'm a newbie and I finally have IP masquerading and my return e-mail address is properly emorr@fast.net. After setting up tin for NNTP I can read articles without trouble but I cannot post. I get a returncode of **441, post rejected**. Didn't my setting up of sendmail for the correct masquerade

work for NNTP, or are my posts “sent differently” than mail? —Edward W. Morris, Jr

Post Rejected?

News posts are indeed “sent differently”. Sendmail runs on port 25, NNTP on port 119. You will need to masquerade both ports, which you must have done if you can read news from masqueraded machines.

If you are getting **post rejected**, it could be because the news server is not set up to allow you to post. Can you post news from your gateway machine? If not, you will need to talk with whoever administers the news server. —Bob Hauck, boh@wasatch.comWasatch Communications Group

Slow Mailserver

We have a 486 AMDX4-100 Mhz/ 32MB RAM system running Linux Kernel 1.2.13 as our Mailserver. We are using WinPmail as our mail clients in Novell Netware mode and all users have direct POP3/SMTP connection to the Linux machine to retrieve and send Internet mail. The problem is: When the POP3 access is made from the workstations, the Linux machine responds very slowly, meaning it takes more than 3 minutes to establish the connection. What causes this problem and how can it be solved? —MadavaneShuttle Technology

Three Possible Causes

There are a couple of possibilities. One is that you are using TCP wrappers in “paranoid mode” and the reverse lookup for the workstations is failing because there is no **in-addr.arpa** entry for them in your DNS. If this is the case, you should be seeing error messages in your syslog (usually **/var/log/messages**).

You can turn off reverse lookups by removing **paranoid** from your **/etc/hosts.allow** and **/etc/hosts.deny**. That will cause most recent versions of TCP wrapper to still log the access, but with only the IP address and not the hostname. Many popular Linux distributions come with TCP wrappers installed and reverse lookup enabled.

Another possibility is that you have enough clients that 32MB is not sufficent. Some POP servers buffer the mailboxes in RAM while downloading, which uses a lot of RAM if your users are keen on MIME attachments. You should be able to diagnose this with **free** and **top** to see if you are swapping heavily when the slowdown occurs. The fix would be more RAM or a different POP server.

Yet another possibility is that your mail transport (sendmail, smail, etc) and your POP server are not agreeing on locking protocols. This can be fixed by

recompiling one or the other. This one is unlikely if you are using the default servers from one of the major distributions. —Bob Hauck, bobh@wasatch.comWasatch Communications Group

Need ELF Support

I have been running Linux for a while and just decided to upgrade to ELF support with 1.2.13 kernel. When I compiled the kernel with that option set it still wouldn't execute the ELF binary. What should I do? —Patrick Temple

ELF Upgrade Tips

Chances are that you did not install the ELF libraries that you need in order to execute ELF programs. Your library files must match those of your executables.

Many Linux distributions include both sets of libraries to allow users to use either set of executables, especially since many existing programs have not been recompiled as ELF binaries yet. This can be convenient to use, but difficult to duplicate, especially without knowing your exact system setup at this point.

You could be headed for trouble if you try to update your system libraries to support ELF binaries. Since your system is currently completely a.out, some of your binaries will stop working if you don't install the new libraries correctly, and you end up removing your old ones. If you really want to do this, there is a file called **move_to_elf** in the **GCC** directory of your favorite Sunsite mirror. This file describes what you need and what you must do to upgrade to ELF from an a.out system.

If you feel nervous about changing your system library files, you might consider simply installing the newest version of Slackware and restoring your original configuration from it. You can generally get a CD of Slackware 3.x for next to nothing, and you will get not only one of the most recent stable kernels (2.0.x) but your system will be ELF-based. Also, the new Slackware distributions come with a working ELF/a.out mixed set of libraries, which will eliminate your worries if you want support for both. —Chad Robinson, chadr@brttech.comBRT Technical Services Corporation

Dropping Performance with 2.0 Kernel

I have just upgraded my kernel from 1.2.12 to 2.0.18. With this my performance and available memory have dropped significantly. I have only 8MB of RAM and a 16MB swap partition. Everything but the essential modules is handled by kerneld. Why has my memory useage gone up so drastically? Is there anything to fix this? —Mr. Shannon Spurling

Streamlining To Save RAM

In general, performance will be much better with the 2.0.x kernels than with the 1.2.x kernels. One possible answer to your problem may be your use of kerneld. That does add some overhead. Also, loading drivers as modules does use up some additional RAM. You are better off compiling drivers into the kernel if you use them most of the time. You are also better off (especially in a low memory situation) if you can avoid using kerneld entirely. —Steven Pritchard, steve@silug.orgSouthern Illinois Linux Users Group

More Memory Needed

The versions 2.0 or higher need more memory than 1.2.x. The only way to fix the problem is to install more memory. With 16MB or more you will have a better performance than with 1.2.x! —Klaus FrankenS.u.S.E GmbH

X-Terms on Strike

Recently my xterms stopped working. I get the message **cannot load libncurses.so.3.0**. So I downloaded the only package I could find that had "ncurses" in the name—ncurses1.9.9e. This didn't fix my problem. Can you please tell me how to fix this? —Samuel Greeley

Establishing a Symbolic Link

Here are some possible answers. First of all, **libncurses.so.3.0** should be a symbolic link to **libncurses.so.1.9.9e**, so make sure the symbolic link is there. You may need to do an **ldconfig** if you change anything with your libraries. Another possible problem might be if your xterm is a.out and your libncurses.so is ELF or vice versa. You can check that by running file on the files (i.e. **file /usr/X11/bin/xterm**), which should tell you whether your xterm binary and your libncurses.so is a.out or ELF. They have to match or xterm won't run. —Steven Pritchard, steve@silug.orgSouthern Illinois Linux Users Group

Archive Index Issue Table of Contents

Advanced search