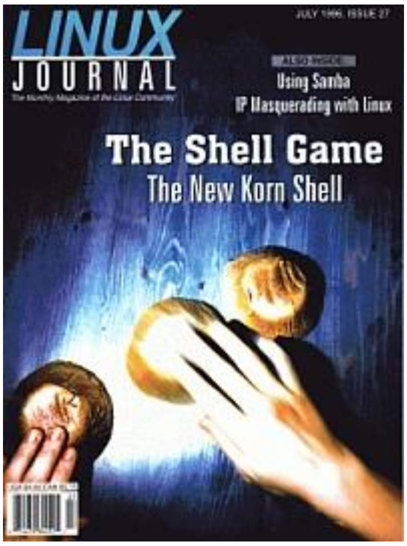
LINUX JOURNAL

Linux Journal Issue #27/July 1996



Features

News and Articles

Columns

Directories & References

Archive Index

Advanced search

LINUX JOURNAL

IP Masquerading with Linux

Chris Kostick

Issue #27, July 1996

A few months ago, Chris concluded an article on building a Linux firewall with an allusion to Linux's ability to hide an entire network behind a single IP address—called IP masquerading. This month, he explains how to enable and configure IP masquerading, also known as Network Address Translation, for Linux.

It seems everyone wants on the Internet nowadays, and for good reason. There is plenty of information to obtain, people to send e-mail to, web pages to look at and software to download. Besides that, businesses are finding acceptable means of advertising, and in some cases, selling merchandise. But with all the rush to get on the Internet, people are finding Internet addresses are not as readily available as they once were. Some network administrators are experiencing that in many environments; they don't have enough network addresses to meet the demand.

Instead of going through the motions of obtaining another block or two of class C addresses, some administrators hide a set of unregistered addresses behind a network address translation (NAT) device. The Internet is prepared for these “private” addresses, and blocks of addresses are reserved for this purpose. RFC 1597 specifies the addresses 10.0.0.0 through 10.255.255.255, 172.16.0.0 through 172.31.255.255, and 192.168.0.0 through 192.168.255.255 to be used in these instances.

The RFC strongly recommends that if you, as a network administrator, are going to use a private address, you should select addresses from the ranges given. One notably important reason is that if a packet happens to pass through the NAT with its original IP address intact, the backbone routers on the Internet will not forward the packet. If, instead, you were using someone else's valid IP address, confusion could occur.

Many firewalls, especially those based on application proxy gateways, naturally hide addresses because of how they function. It is no surprise that Linux can also support address hiding through what is called “IP masquerading”. Setting up masquerading under Linux is not terribly difficult, but there are some subtleties to point out.

Getting Ready

If you are running kernel version 1.2.x, you need to obtain the kernel patch to support masquerading. The patch is available from ftp://ftp.eves.com/pub/masq, or you can download everything you need from www.indyramp.com/masq/. IP masquerading is supported with 1.3.x kernel versions. For this article, I was running version 1.3.56, and all examples are based on this version. For FTP support (mentioned later), you need to have at least kernel version 1.3.37. There is a patch for 1.2.*x* (where *x* >= 4) kernels to support FTP, but I haven't tested it yet. The masqplus-0.4 “jumbo” patch that is available from Indyramp fixes a few bugs and adds support for FTP, RealAudio, and fragmentation for 1.2.13 kernels.

When configuring the kernel to support masquerading, it is important to also say yes to firewall and forwarding support. Here are the parameters I used for configuring my kernel:

```
Network firewalls (CONFIG_FIREWALL) [Y/n/?] y
Network aliasing (CONFIG_NET_ALIAS) [Y/n/?] y
TCP/IP networking (CONFIG_INET) [Y/n/?] y
IP: forwarding/gatewaying (CONFIG_IP_FORWARD) [Y/n/?] y
IP: multicasting (CONFIG_IP_MULTICAST) [Y/n/?] y
IP: firewalling (CONFIG_IP_FIREWALL) [Y/n/?] y
IP: accounting (CONFIG_IP_ACCT) [Y/n/?] y
IP: tunneling (CONFIG_NET_IPIP) [Y/m/n/?] y
eP: firewall packet logging (CONFIG_IP_FIREWALL_VERBOSE) [Y/n/?] y
IP: masquerading (ALPHA) (CONFIG_IP_MASQUERADE) [Y/n/?] y
```

I chose other items not directly related to masquerading such as multicast and tunneling, but I like to have fun.

Notice the IP masquerading software is still considered to be Alpha-quality. This means there are probably still some bugs. The base functionality is there, but not all of the nuances of TCP, UDP, and IP, nor the application protocols, have been thoroughly tested. In addition, the interface may still change as development proceeds.

In order to manipulate the masquerading ruleset, you will need the **ipfw** software version 1.3.6-BETA3, or you can obtain a precompiled binary from ftp.eves.com. Those who use Linux as a filtering firewall and also use **ipfwadm** should note that software does not yet support IP masquerading, so ipfw is necessary. [New: ipfwadm 2.0beta2, now available for Linux 1.3.66 and newer from ftp://ftp.xos.nl/pub/linux/ipfwadm/, **does** support masquerading. Also, it is necessary to use recent versions of ipfwadm with the most recent versions of the kernel due to interface changes—ED]

Applying the Rules

Let's first define what we're trying to accomplish and see how IP masquerading is useful in the environment. Figure 1 shows the networks on which the examples are based. deathstar is the Linux machine employing masquerading in order to hide the network 192.168.1.0.

Masquerading is useful in our architecture because it saves us a little administrative hassle. A number of people in my department have home LANs, and through their PPP connection they can use their other machines to connect to the department lab. We could easily run a routing protocol, like RIP, to make the machines on the lab network aware of the home LANs, but that would take some coordination about who has what network address. It is easier (for us) to use masquerading.

To hide the network, we can issue the command:

```
# ipfw a m all from 192.168.1.0/24 to 0.0.0.0/0
```

This rule indicates that we want to **a**dd a **m**asquerading rule for **all** protocols (which in this case means TCP and UDP). The network we are hiding is 192.168.1.0, and we are hiding connections going to any network (0.0.0.0/0). The /24 indicates we are applying a 24-bit netmask, or 255.255.255.0. Since we specified the network as 192.168.1.0, deathstar will masquerade for all hosts on the network. That's all we need to do.

If I had only wanted deathstar to masquerade for enterprise, then I would have typed in:

```
# ipfw a m all from 192.168.1.2/32 to 0.0.0.0/0
```

But what does it really mean “to masquerade for”? Well, let's examine the affected files and kernel tables for a typical masqueraded connection. We'll use telnet for our example.

Let's verify the rule has been set. We need to look at the ip_forward file in the /proc/net directory. We can use ipfw to do this:

```
# ipfw -n list forward
Type    Proto       From           To      Ports
(masqueradeall  192.168.1.0/24  anywhere
```

This is good. Some administrators mistakenly look in the /proc/net/ip_masquerade file for the rule and when they don't see it, confusion sets in.

For our example, I've started a telnet session from warbird to enterprise. Also, on mccoy, I'm using the tcpdump program to monitor the traffic on 20.2.51.0 and sparcbook to monitor the traffic on 192.168.1.0. We can now look at the ip_masquerade file to examine what is happening (see Listing 1).

Let's decode this stuff. First, the earliest packet is at the bottom. It is a DNS request (therefore UDP) from 192.168.1.2 to 20.2.51.2. mccoy is warbird's DNS server in this case. The Masq column shows us the port on deathstar that is used for the masquerading. For the first DNS request, it is port 60000 (EA60). After the DNS resolution, the TCP connection is established on the next available port over 60000, 60001. Figure 2 illustrates the protocol time-line for the sequence of events up to the TCP open.

Even though the protocol time-line shows how the packets really traverse, the sending and receiving nodes are unaware of this. Hence, the reason they call it masquerading. From warbird's point of view, the traffic will look exactly as expected. That is, packets from enterprise are repackaged by deathstar to look as if they came from enterprise. Listing 2 shows the tcpdump output of the traffic on the 192.168.1.0 network for the telnet session.

Listing 3 shows the protocol traffic on the 20.2.51.0 network during the telnet session. Notice that information originates from deathstar, not warbird. (Another thing you might notice is I don't keep the clocks synchronized very well.)

Another important aspect is maintaining the TCP synchronization numbers. For masquerading to work properly, deathstar must keep the synchronization correct. The TCP sequence number generated by warbird is forwarded by deathstar rather than a new sequence number being generated.

Some final observations about the contents of the /proc/net/ip_masquerade file pertain to the last four fields. The **Init-seq**, **Delta**, and **PDelta** fields deal with the TCP synchronization numbers when ftp data transfers (more in a minute) occur, and the last field is the expiration timer on the masquerade entry. The time is kept in hundredths of seconds; TCP is given 90000 or 15 minutes, and UDP is given 300000 or 5 minutes. As long as traffic is being passed between the two communicating hosts for the masked port, the timer will remain updated. A minor detail about the expiration timer has to do with FTP transfers. FTP uses two connections: a control connection for commands and a data connection for a file transfer. While the data connection is in use for data movement, the control connection will sit idle. If the transfer takes longer than 15 minutes, the masquerading host will close the control connection. The data connection will go to completion, but you will have to reconnect if you want to get more files. This is controlled by the definitions:

```
#define MASQUERADE_EXPIRE_TCP     15*60*HZ
#define MASQUERADE_EXPIRE_TCP_FIN  2*60*HZ
#define MASQUERADE_EXPIRE_UDP      5*60*HZ
```

in the file /usr/include/linux/ip_fw.h. Six hours (360 minutes) seems to be a relatively acceptable timeout value, but change it as you see fit.

Problems

Not all protocols work with IP masquerading. ICMP messages (such as those used by ping) will not be passed through the masquerading host. Also, application protocols that pass their address to the receiving host will not work. The **talk** program is an example of this.

A major exception to the applications that don't work is **ftp**. The IP masquerading software has been written to handle file transfers as of kernel version 1.3.37. FTP clients, under normal operation, will send the server the address and port number to which the server should connect for a transfer. This shouldn't work with masquerading for the same reasons that talk fails. However, the IP masquerading software will intercept the FTP PORT command and masquerade as the client host awaiting for the server to connect to it.

The biggest problem is the most subtle one: IP fragmentation. Fragmentation occurs automatically within the Internet Protocol. IP always wants to fit a datagram in the frame size of the network link it is transmitting over. Most data links define a Maximum Transmission Unit (MTU) of information that will fit within one frame. If the IP datagram to be sent out can't all fit into the MTU size of the frame, it will be fragmented.

An IP datagram carrying a TCP segment is structured like the “Original Datagram” illustration in Figure 3. After fragmentation, the new datagrams appear (also shown in Figure 3). The most important aspect to notice is the placement of the TCP header. With fragmentation, it only appears in the first fragment and not in succeeding ones. Without the header, the host doing the masquerading has no way of determining whether the fragment should be forwarded. The same applies for fragmented UDP packets.

With TCP, this problem is mostly avoided because of TCP's MSS (Maximum Segment Size) negotiation. That's not to say it won't happen, but it doesn't occur most of the time. UDP, however, is much more susceptible to this type of behavior. Your only solution as an administrator is to be careful about controlling MTU sizes on SLIP or PPP networks.

Other problems also exist for X applications (connections back to the X server); RealAudio (patches available, however); and rlogin (rlogind requires a privileged port).

Real World Problem

Actual troubleshooting of masquerading problems is not always as easy as getting the rules straight. One subscriber to the IP masquerading mailing list (see Sidebar) presented an interesting problem. It was solved with simple analysis, code knowledge, and a good hex editor.

The Problem

Greg Priem sent a message to the IP masquerading mailing list describing a problem in which his telnet sessions would freeze. He isolated a sequence of events that reproduced the problem—he would log into his service provider's main host from a machine behind his Linux box and type in **ls -l**.

Analysis

Greg did some initial analysis and posted what he found. The network he was using is illustrated in Figure 4. The telnets were from the Mac to the ISP and other hosts on the Internet. He noticed telnets from the Mac to the Linux Box worked fine, as well as telnets from the Linux Box to the ISP.

Output from tcpdump revealed fragmentation was taking place. I followed up with a message indicating a possible problem and asked Greg to check the MTU sizes on each interface of the Linux Box.

I thought it strange that fragmentation was occurring on a telnet session since telnet uses TCP. As mentioned before, when TCP opens a connection, the MSS negotiation is supposed to eliminate fragmentation.

Further debugging with tcpdump (a handy program) showed the MTU assigned by the ISP was 212. To try to eliminate fragmentation, the SLIP link was also assigned an MTU of 212 by Greg. When looking at the MSS negotiation of the connections, Greg found that from the Linux box to the ISP, the MSS was set to 172, and from the Mac to the Linux box it was the same. However, a connection from the Mac to the ISP showed an MSS of 536.

The Solution

Given that information, I was able to deduce the problem and respond with an appropriate solution.

The connection scenarios are given in Figure 5.

One thing to note was the MSS advertisement of 536 from the Mac when it had an immediate link with an MTU smaller than that. BSD-experienced people will remember this number from the networking code that chose an MSS value for

TCP's negotiation by seeing if the destination was on the local LAN or a remote LAN. The code roughly looked like:

```
if dest_net == local_net
then
        mss = (link MTU) - 40
else
        mss = 536
        /* determined by 576 - 40 */
fi
```

If the destination was on a remote network, it would set the MSS automatically to 536. This was a good number because the RFC for IP stated that the default datagram size for internetworking is 576, meaning every device should be able to handle it without further fragmentation. Forty is subtracted to allow for IP and TCP headers.

A second thing to notice was the Linux box *forwarding* the MSS advertisement. One might think that since a connection is being made from the Linux box as a consequence of masquerading, the MSS value would be based on the network link from the Linux box and not the original value from the sending host.

As an aside, there was the one unexplainable instance of connections made to the ISP host and the ISP sending back an MSS of 1460, as shown at the bottom of Figure 5. It's strange because it was also connected to the PPP link with an MTU of 212. This may be attributed to a lack of knowledge on the ISP's side of the network.

Since both sides were using an MSS value greater than the MTU of either link, there was bound to be fragmentation, even for a TCP connection. Under normal circumstances, this wouldn't matter, but it does confuse masquerading.

The simple solution was to have the ISP support an MTU of at least 576 and for Greg to set the SLIP link with an MTU of 576 or greater. Therefore, no fragmentation would occur.

Greg e-mailed his ISP and waited for an answer. When none arrived he became impatient. Since he didn't have the source to the TCP code on the Mac, the only way to look at it was with a hex editor. He started poking around to see if he could find the BSD-like code where it made the decision for the MSS, and sure enough, he found it. He changed the hard coded values of 536 to 172 (i.e. 212-40), restarted his Mac, and lo and behold, it worked—no more fragmentation! (By the way, the ISP did change the MTU size later.) His approach was a little more daring than what I would have done, but it seems to be the nature of Linux users to patch an existing binary if they can't recompile something.

Conclusions

IP masquerading is an interesting technology, but more importantly, it serves a very useful function for many Internet environments. It works well for common services such as telnet, http, and ftp, but it does not support everything. ICMP messages, talk, remote X applications, and rlogin do not work with masquerading. Fortunately, the software is still in its Alpha versions, and more development is being pursued.

Chris Kostick (ckostick@csc.com) is a Senior Computer Scientist at Computer Sciences Corporation's Network Security Department. He enjoys working with Linux but considers himself a latecomer because he started out at kernel version 1.1.18. As far as computers go, he's not sure if he has more fun debugging TCP/IP problems or shooting DOS machines

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Understanding Red Hat Run Levels

Mark F. Komarinski

Issue #27, July 1996

How to easily add to or modify the existing subsystems of Red Hat distributions of Linux.

If you're one of those who took a chance and got one of the Caldera Previews or got a Red Hat distribution on your system, one of your original thoughts may have been the same as mine: What happened to /etc/rc.local? Where am I supposed to put my custom commands? [One answer: /etc/rc.d/rc.local is available on Red Hat systems—ED] What if I don't want the HTTP server to start?

For those of you out there who administer Sun Solaris machines, this looks quite familiar. But I was just scratching my head for a while until I wound up administering a system, and it all became clear. Time to share the knowledge.

The idea behind the setup is to make everything script-based. For each run level, scripts are run to start each individual service, instead of having a few large files to edit by hand. These scripts are located in /etc/rc.d/init.d, and most take as an option **start** or **stop**. This is to allow the specific programs to start (on bootup) or stop (on shutdown).

This setup involves a bunch of directories under /etc/rc.d/. These are:

rc0.d Contains scripts to run when the system shuts down. Technically, halt or shutdown bring the system to runlevel 0. This directory is mostly made up of kill commands.

rc1.d through rc3.d Scripts to run when the system changes runlevels. Runlevel 1 is usually single-user mode, runlevel 2 is for multi-user setup without NFS, and runlevel 3 is full multi-user and networking.

Runlevel 4 is typically unused.

rc5.d Scripts to start the system in X11 mode. This is the same as runlevel 3, with the exception that the xdm program starts, which provides a graphical login screen.

rc6.d Scripts to run when the system reboots. These scripts are called by a reboot command.

init.d Actually contains all of the scripts. The files in the rc*?*.d directories are really links to the scripts in the init.d directory.

The Boot Sequence

Now that we know where files are located, let's look at what happens in a normal Red Hat boot sequence.

Once the system boots, /etc/rc.d/rc.sysinit is run first. The starting runlevel (specified in /etc/inittab) is found, and the /etc/rc.d/rc script is run, with the sole option being the runlevel we want to go to. For most startups, this is runlevel 3.

The rc program looks in the /etc/rc.d/rc3.d directory, executing any **K*** scripts (of which there are none in the rc3.d directory) with an option of **stop**. Then, all the **S*** scripts are started with an option of **start**. Scripts are started in numerical order—thus, the S10network script is started before the S85httpd script. This allows you to choose exactly when your script starts without having to edit files. The same is true of the **K*** scripts.

Let's look at what happens when we switch runlevels—say from runlevel 3 (full networking and multi-user mode) to runlevel 1 (single-user mode).

First, all the **K*** scripts in the level to which the system is changing are executed. My Caldera Preview II (Red Hat 2.0) setup has seven K scripts and one S script in the /etc/rc.d/rc.1/ directory. The K scripts shut down nfs, sendmail, lpd, inet, cron, and syslog. The S script then kills off any remaining programs and executes **init -t1 S**, which tells the system to really go into single-user mode.

Once in single-user mode, you can switch back to full multi-user mode by typing **init 3**.

Side-stepping init

There are two additional points I can make here.

First, you can selectively start and stop scripts, even those not native to your runlevel. Executing scripts in /etc/rc.d/init.d/ with an option of **start** or **stop** will start up or stop the programs or services which the script controls. This allows you to turn off NFS from runlevel 3 while keeping all other systems active, for example. Similarly, you can start NFS back up when you are ready.

Stopping NFS in this case would require stopping two systems—nfsfs and nfs. The nfsfs script will mount or ummount any of the NFS-mounted file-systems listed in your /etc/fstab. The nfs script would then shut down the processes associated with NFS, in this case mountd and nfsd.

So the proper procedure for shutting down NFS would be:

```
# /etc/rc.d/init.d/nfsfs stop
Unmounting remote filesystems.
# /etc/rc.d/init.d/nfs stop
Shutting down NFS services: rpc.mountd rpc.nfsd
#
```

And starting NFS would be:

```
# /etc/rc.d/init.d/nfs start
Starting NFS services: rpc.mountd rpc.nfsd
# /etc/rc.d/init.d/nfsfs start
Mounting remote filesystems.
#
```

Managing init Files

Do you want to **not** start the HTTP daemon, without removing the file from the rc3.d directory? Easy. Rename /etc/rc.d/rc3.d/S85httpd to something that does not start with a capital S or a capital K. Your best bet would be to rename files using a lowercase “s” or a lowercase “k”. This way, not only will the scripts not be started, but they'll appear later in an ls file listing, since entries starting with capital letters are shown separately from those beginning with lower case letters. So you'd wind up with a file now called s85httpd, which is somewhat separated from the rest of the entries an an ls -l listing.

An important note here, though: make sure you know what the scripts are doing when you disable them. If you disable something like the S10network script, none of your networking software will work. This is why S10network has such a low number: because other scripts are dependent on the network and must be executed after the network software is in place.

You want to make your own init processes to start and stop? That's easy enough to do. Make a script that accepts the word **start** as an option. Not all scripts need to be able to stop, but if the script starts a process in the background, you should almost certainly include a **stop** option as well. For example, a script that polls the time over the network every time you enter runlevel 3 does not need a **stop**. A script that starts a program to query the network time every 15 minutes would need a start and a stop script, since the program the script started is continuously running. A program for the second example is better suited from a crontab, but being able to do things your own way is at the heart of Unix, isn't it?

Once it's written (*and* tested), put it in the /etc/rc.d/init.d/ directory. Let's say it's the program to check the time on a network machine every 15 minutes, so we'll call the script “netdate”. Once it is in the /etc/rc.d/init.d/ directory, you can make links in the directories you want to start it in. If you want your program to run in runlevel 3, make a link to your script from /etc/rc.d/rc3.d/S*??*netdate. Make *??* a number that will fit in the rest of the directory, such as S55netdate. You'll want it to be above S10 so that the network is started, and S55 isn't taken, so it seems a good enough location. It's not required that there be only one script with each number, but it is good form, since you know exactly what order the scripts will be started in.

If you want to stop the process gracefully during a shutdown, make sure your script accepts **stop**, then make a link to /etc/rc.d/init.d/netdate from /etc/rc.d/rc0.d/K55netdate. Again, you should make sure the number you use is not being used by another subsystem to avoid confusion.

You can test your new setup by using **init 3**. Since the other subsystems are already running, the only one that will start is the one you just added. If the **init 3** command hangs, your script didn't exit; you probably need to put an ampersand at the end of a line to put the problem process in the background. You can also run your script manually from the /etc/rc.d/init.d/ directory.

Now that you know how the subsystems work, you can easily add or modify the existing subsystems for your particular Linux setup.

Mark Komarinski (markk@auratek.com) has been using Linux since 1993 when he first purchased his 386/40. He just completed a book on Linux to be published by Prentice Hall. Mark now works for Aurora Technologies doing internal PC support and manning the customer service phones. He lives in eastern MA with his wife, Brenda.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Advanced search

Filters: Doing It Your Way

Malcolm Murphy

Issue #27, July 1996

A look at several of the more flexible filters, probrams that read some input, perform some operation on it, and write the altered data as output.

One of the basic philosophies of Linux (as with all flavours of Unix) is that each program does one particular task, and does it well. Often you combine several programs to achieve something, either at the shell prompt or in a script, by piping the output of one program into the next. I'm talking about things like

```
ls -l | more
```

and

```
ps -auxw | \
  grep netscape >> people.who.should.be.working
```

But what if the output of one program isn't in the format needed for the next? We need some way of processing the output of one program so that it is ready for the next.

Fortunately, there are many Linux programs that do this job: read some input, perform some operations on it, and write the altered data as the output. These programs are called filters. Some filters do quite limited tasks, such as head, grep and sort, whereas others are more flexible, such as sed and awk. In this article, we're going to look at several of these more flexible filters, and give several examples of what can be done with them.

The name “sed” is a contraction of *stream editor*; sed applies editing commands to a stream of data. A common use for sed is to replace one text pattern with another, as in

```
sed 's/Fred/Barney/g' foo
```

This command takes the file foo, changes every occurrence of **Fred** to **Barney**, and writes the modified version to standard output.

Note that in this example we have placed the actual sed commands inside single quotes. Sed doesn't require that commands be quoted this way, but you will need to use quotes if the sed command includes characters that are special to the shell, such as **$** or *****. This example doesn't have any special characters, so we could just as easily have left out the quotes. Try it and see.

Without the input file foo, sed reads from standard input, so we could achieve the same result with the command

```
   sed 's/Fred/Barney/g' < foo
```

or

```
   cat foo | sed 's/Fred/Barney/g'
```

Note that the first two versions are generally preferred to the third. Using cat just to send input into a pipe creates an extra process which can often be avoided.

We also have to consider the output. By default, the results appear on standard output, but this isn't always what we want. One option is to pipe the output through a pager, for example

```
   sed 's/Fred/Barney/g' foo | more
```

or to redirect it to a file

```
   sed 's/Fred/Barney/g' foo > bar
```

While it is often tempting to write

```
   sed 's/Fred/Barney/g' foo > foo
```

the only thing this achieves is to delete contents of the file foo! Why? Because the first thing the shell does with this command is to open the file foo for output, destroying what was there already. When it tries to read from foo, there is nothing there to read. The result is an empty file. This is an easy mistake to make when redirecting output in this way, so do be careful.

Awk is a bit more flexible than sed; it is a full-fledged programming language in its own right. However, don't let that put you off. Writing simple programs in awk is surprisingly easy, and it often doesn't feel like a programming language [See page 46 of *Linux Journal* issue 25, May 1996—ED]. For example, the command

```
awk '{print NR, $0}' foo
```

prints the file foo, numbering each line as it goes. Awk can also read its input from a pipe or from standard input, exactly like sed, and also writes on standard output, unless you redirect it. The bit between the quotes (which are necessary, since the **{}** characters are also special characters to the shell) is the awk program. I said they can be simple, didn't I? An awk program is simply a sequence of one or more pattern-action statements, in the form

```
pattern { action }
```

Each input line is tested against each pattern in turn. When an input line matches a pattern, the corresponding action is performed. Either the pattern may be empty, in which case every line matches, or the action may be empty, in which case the default action is to print the line.

In the example above, the pattern was empty, so every line matched. The action was to print **NR**, which is a built-in awk variable containing the number of lines read so far, and then print **$0**, which is the current line.

Going On

Now that we've seen the basic idea behind sed and awk, we're going to look at some examples. The best way to learn something is to actually do it, and I recommend that you try out some of these examples yourself as you go along, possibly even with one eye on the man pages. We certainly aren't going to cover everything that sed and awk can do, but you will, it is hoped, have more confidence to try things out yourself once you've finished reading this article.

Our first example is to remove all the spaces from a document. This is easily achieved using sed:

```
sed 's/ *//g' foo
```

This is like the earlier example with Fred and Barney, only here we have used a regular expression: **' *'** (the quotes are included so that you can see the space that is part of the regular expression). sed's **s** (for substitute) command using regular expressions just like grep. The regexp **' *'** matches one or more spaces, which are replaced with *nothing*—they are deleted. This command doesn't deal with tabs, as it stands, but you could modify it to match one or more occurences of either a tab or a space:

```
sed 's/[ {tab}][ {tab}]*//g' foo
```

Double Spacing

Next, we'll think about doublespacing a text file. We can do this using sed's substitute command by replacing **$** (the regexp for the end of a line) with a newline character (which we have to quote with a backslash)

```
sed 's/$/\
/' foo
```

Note that in this example, there isn't a **g** before the second quote, unlike all the earlier examples. The **g** is used to tell sed that the substitution applies to all matches on each line, not just the first match on each line, which is the default behaviour. In this case, since each line only has one end, we don't need the **g**.

Another way of doing this in sed would be:

```
sed G foo
```

If you look at the man page for sed, it says that **G** “appends a newline character followed by the contents of the hold space to the pattern space”. The pattern space is the sed term for the line currently being read, and we don't need to worry about the hold space for now (trust me, it will be empty), so this command does exactly what we want.

It's quite easy to doublespace in awk, using the print statement we saw earlier:

```
awk '{print $0; print ""}' foo
```

Here, the pattern is empty again, matching every line, and the action is to print the entire line, **$0**, then to print nothing, **""**. Each **print** statement starts a new line, so the combined effect of the two commands is to doublespace the file.

Awk actions can (and often do) involve more than one command in this way, but it isn't strictly necessary here. Awk provides a formatted print statement that gives more control over the output than the basic **print** statement. So we could get the same result with:

```
awk '{printf("%s\n\n",$0)}' foo
```

The first argument to the **printf** statement is the *format*, a description of how the output should appear. The format can contain characters to be printed literally (none in this example), escape sequences (such as **\n** for a newline), and *specifications*. A specification is a sequence of characters beginning with a % that controls how the rest of the arguments are printed. For each of the second and subsequent arguments, there must be a specification. In this example, there is one specification, **%s**, which prints a character string. The value associated with that specification is **$0**; the entire line. Unlike **print**, **printf** doesn't automatically start a new line, so two **\n**'s are needed: one to end the original line and one to insert a blank line.

For this seemingly simple example—doublespacing a file—we came up with four different solutions. There is always more than one way of solving a problem, and it normally doesn't matter which one you take. The point is that you usually write an awk or sed program to do a particular task as the need arises, then discard it. You don't necessarily want the “best” solution (whatever that means), you just want something that works, and you want it quickly.

Being Selective

Another quite common task is to select just part of the input. Suppose we want the fifth line of the file foo. In awk, this would be

```
awk 'NR==5' foo
```

which prints the line when **NR**, the number of lines read so far, equals 5. The sed equivalent is

```
sed -n 5p foo
```

By default, sed prints every line of input after all commands have been applied. The **-n** option suppresses this behaviour, so we only get the line we specifically ask for with the **p** command. In this case, we asked for the fifth line, but we could just as easily specified a range of lines, say the third to the fifth, with:

```
sed -n 3,5p foo
```

or, in awk

```
awk 'NR>=3 && NR<=5' foo
```

In the awk version, the **&&** means “and”, so we want the lines where **NR>=3** *and* **NR<=5**, that is, the third through the fifth lines.

Yet another approach would be to combine head and tail

```
head -5 foo | tail -3
```

which uses the head program to get the first 5 lines of the file, and the tail program to only pass the last three lines through.

Yet another common problem is removing only the first line. Remember how the **$** character means the end of the line when it is used in a regular expression? Well, when you use it to specify a line number, it means the last line:

```
sed -n '2,$p' foo
```

In awk, you can use **!=** or **>** to get the same result from either of these commands:

```
awk 'NR>1' foo
awk 'NR!=1' foo
```

When Line Numbers Are Not Enough

Selecting part of a file using line numbers is easy enough to do, but often you don't know the line numbers you want. Instead, you want to select lines based on their contents. In awk, we can easily select a line matching a pattern, with

```
awk '/regexp/' foo
```

Which causes all lines containing ***regexp*** to be printed. There is a direct sed equivalent of this:

```
sed -n '/regexp/p' foo
```

Of course, we can also use grep to do this kind of thing:

```
grep 'regexp' foo
```

but sed can also handle ranges easily. For example, to get all lines of a file up to and including the first line matching a regexp, you would type:

```
sed -n '1,/regexp/p' foo
```

or to get all lines including and after the first line matching regexp:

```
sed -n '/regexp/,$p' foo
```

Remember that $ means the last line in a file. You can also specify a range based on two regexps. Try

```
sed -n '/regexp1/,/regexp2/p' foo
```

Note that this prints all blocks starting with lines containing ***regexp1*** through lines containing ***regexp2***, not just the first one. If there isn't a matching ***regexp2*** for a line containing ***regexp1***, then we get all lines through to the end of the file.

Now we can select some part of the input, based on a regular expression.

We might want to delete some lines that contain a certain pattern. The **d** command does just that:

```
sed '/regexp/d' foo
```

deletes all lines that match the regexp. Or, we might want to delete a block of text:

```
sed '/regexp1/,/regexp2/d' foo
```

deletes everything from a line that contains ***regexp1***, up to and including a line that matches ***regexp2***. Again, sed will select all blocks of text delimited by ***regexp1*** and ***regexp2***, so there is a danger we could delete more than we want to.

Inserting text at a given point is possible, too. The command

```
sed '/regexp/r bar' foo
```

inserts the contents of the file bar after any line that matches the ***regexp*** in the file foo.

Now, we can combine these last two commands to replace a block of text in a file with the contents of another file. We do it like this:

```
sed -e '/START/r bar' -e '/START/,/END/d' foo
```

This finds a line containing **START**, deletes through to a line containing **END**, then reads in the contents of the file bar. Because the **r** command doesn't read in the file until the next input line is read, the **d** command is executed before the new text is read in, so the **d** command doesn't delete the new text, as one might expect, looking at this command. The **-e** option tells sed that the next argument is a command, rather than an input file. Although it is optional when there is only one command, if we have multiple commands, they must each be preceded with **-e**.

Columns

These examples have mostly been line oriented, but we are just as likely to want to deal with columns of data. The filter **cut** can select columns of data. For example, to list the real names of all the users on your system, you could type

```
cut -f5 -d: /etc/passwd
The 5 argument after -f tells cut to list the
fifth column (where real names are stored), and the -d
flag is used to tell cut which character delimits the
field—in the case of the password file, it's a colon. To get
both the username (which is in the first column) and the real
name, we could use
```

```
cut -f1,5 -d: /etc/passwd
```

Awk is also good at getting at columns of data, we could do these tasks with the following awk commands:

```
awk -F: '{print $5}' /etc/passwd
```

and

```
awk -F: '{print $1,$5}' /etc/passwd
```

where the **-F** flag tells awk what character the fields are delimited by. (Do you see the difference between using cut and using awk for printing more than one field? If not, try running the commands again and looking more closely.)

One advantage of using awk is that we can perform operations on the columns.

For example, if we want to find out how much disk space the files in the current directory take up, we could total up the fifth column of the output of **ls -l**:

```
ls -l | grep -v '^d' | \
  awk '{s += $5} END {print s}'
```

In this command, we use grep to remove any lines that begin with **d**, so we don't count directories. We chose grep, but we could just as easily have used awk or sed to do this. One pure awk solution could be:

```
ls -l | awk '! /^d/ {s += $5} END {print s}'
```

where the awk program only totals the fifth column of lines that don't begin with a **d**—the exclamation mark before the pattern tells awk to select lines which *don't* match the regular expression **/^d/**.

Working with Filenames

Often, many files have the same basic name, but different extensions. For example, suppose we have a TeX file foo.tex. Then we could very well have associated files foo.aux, foo.bib, foo.dvi, foo.ps, foo.idx, foo.log, etc. You might want a script to be able to process these files, given the name of the file foo.tex. The basename utility:

```
basename foo.tex .tex
```

will give you the basic name **foo**. If we have a shell variable containing the name of the TeX file, we might use

```
basename ${TEXFILE} .tex
```

Again, there is more than way of getting the basename of a file: you could do this in sed using:

```
echo ${TEXFILE} | sed 's/.tex$//'
```

Whichever approach we take, we can construct the name of the other files once we know the basic name. For example, we can get the name of the log file by:

```
LOGFILE=`basename ${TEXFILE} .tex`.log
```

This is very useful: I use vi for most of my editing, and it allows you to get at the name of the file currently being edited in a macro; **%** is replaced with the filename. If I'm editing a TeX file foo.tex, and I want to preview the dvi file using xdvi, I can transform **%** (let's call it foo.tex) into **foo.dvi** automatically in a macro

```
:!xdvi `basename % .tex`.dvi &
```

I can bind this to a function key and never worry about the name of the dvi file when I want to view it, by adding this line to my .exrc file:

```
map ^R :!xdvi `basename % .tex`.dvi &^M^M
```

The **^R** and **^M** characters are added by typing Control-V Control-R and Control-V Control-M, respectively, assuming you are editing your .exrc file with vi.

Conclusion

In this article, we have looked at the some of the tools available in Linux for filtering text. We have seen how using these filters we can manipulate the output of one command so it is in a more convenient form to be used as the input for another program or to be read by a human. This kind of task arises naturally in a lot of shell-based work, both in scripts and on the command line, so it is a handy skill to have. Although the man pages for sed and awk can be a little cryptic, solutions to problems can often be very simple. With a little practice, you can do quite a lot.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Advanced search

The New KornShell—ksh93

David G. Korn

Charles J. Northrup

Jeffery Korn

Issue #27, July 1996

The KornShell, written by David Korn at Bell Telephone Laboratories, combined the best features of both of these shells, and added the ability to edit and reenter the current and previous commands using the same keystrokes as either the vi or the Emacs editor as the user desired.

The Unix system was one of the first systems that didn't make the command interpreter a part of the operating system or a privileged task. It was written as an ordinary user process with no special permissions or calls to unadvertised functions. This has led to a succession of better and better shells. The early generations of Unix came with a command shell written by Ken Thompson, one of the inventors of the Unix system. By the late 1970s, two vastly improved shells emerged. The Bourne shell, created by Steve Bourne at Bell Telephone Laboratories, was a big improvement as a language. The C shell, created by Bill Joy at the University of California at Berkeley, was a much improved command interpreter but a poor language.

The KornShell, written by David Korn at Bell Telephone Laboratories, combined the best features of both of these shells, and added the ability to edit and reenter the current and previous commands using the same keystrokes as either the vi or the Emacs editor as the user desired. This shell became very popular, but its distribution was restricted. As a result, several freely available imitations such as pdksh and bash were created. An enhanced version of C shell, tcsh, was created to provide visual editing to C shell users.

While the Bourne shell provided a good basis for programming, and this was improved upon by earlier versions of KornShell, it was not adequate for general purpose scripting without combining it with other languages such as the awk programming language. While in most instances the two languages work well together, the performance penalty of using two languages with separate processes is often prohibitive. The Perl language was created to provide a single language with the combined functionality of the shell and awk. However, Perl has a syntax that many find difficult to understand.

ksh93, the latest major revision of the KornShell language provides an alternative to Tcl and Perl. As a programming language, it has comparable speed and functionality to each of these languages, yet is arguably the best interactive shell. It is a superset of the POSIX 1003.2 shell standard. Like Tcl, it is extensible and embeddable with a C language application programming interface. In fact, two graphical shells have been created using ksh93. One of these, dTksh, is a Motif-based language developed by Novell. The other, Tksh, written by Jeff Korn at Princeton University, uses the Tk library, and is briefly discussed here.

The best way to describe the new features found in ksh93 is to illustrate them through an example. We will create a shell script named **lsc**, shown in Listing 1, to provide an **ls** output with subdirectory names printed in bold. We will need to maintain the multi-column output associated with the standard ls.

The lsc script will produce the ls output for each directory name provided as a command line argument. The default action is to produce the ls output for the current directory. Several modifications can be made to the lsc script for enhanced performance. We leave them as an exercise for the reader. We perform the following high level actions for each directory name to be processed.

for each directory do

```
 load directory entries into array entries
 load entries
calculate number of columns in multi-column output
calculate maximum number of rows
print the current directory name
determine output layout
add entries to row[] array
add entries to col[] array
calculate the column widths
display the output
```

done

Arrays

ksh93 provides one-dimensional indexed and associative arrays. An array element is referenced as ***`varName`*****`[`*****`subscript`*****`]`**. Indexed arrays use arithmetic expressions for subscripts. This permits computation within the subscript expression. The statement **varName[3+8]** for example, references the

11th element of the indexed array. (Arithmetic expressions are described more fully below).

The elements of an indexed array can be initialized from a list using the ***varName*=(....)** command. This provides a convenient notation for initializing an array to contain the names of files in a given directory. The number of entries in the array describes the number of files found. As an example, consider the following statement to initialize the entries indexed array with the names of files found in the current directory: **entries=(*)**

An associative array uses arbitrary strings for subscripts. We could, for example, create a state tax associative array and reference elements by the state name. This works even for space separated tokens within the string, such as New Jersey.

```
typeset -A StateTax
StateTax[New Jersey]=0.06
print ${StateTax[New Jersey]}
```

Several special positional parameter expansions are provided for array processing. Using **${*varName*[@]}** refers to all elements of the array. The subscripts of an array can be referenced with **${!*varName*[@]}**. The notation **${#*varName*[@]}** provides the number of elements within the array. Elements within a numeric subscript range can be referenced using **${*varName*[@]:*offset*:*length*}**. This special notation works with both indexed and associative arrays.

Arrays are used throughout the example lsc script. We define **video** as an associative array with capability names from the *terminfo* database as subscripts. The definition of **video** is provided as a compound assignment for an associative array.

```
video=(
   [bold]=$(tput bold)
   [reset]=$(tput reset)
   [reverse]=$(tput reverse)
)
```

Each element is assigned a value from the standard output of a **tput** execution for the capability name. For example, **video[bold]** is the terminfo sequence for bold lettering. Similarly, **video[reverse]** will provide reverse video output.

Using the notation **$(*command*)** will cause ***command*** to execute in a subshell of the current ksh. In many instances, ksh will not actually fork/exec a subshell when command is a built-in or a shell function. (Built-in functions are described below).

Expanded Name Space

In ksh93 a variable is defined by a ***`name=value`*** pair. The variable name space is hierarchical with ***`.`*** (dot) delimiters. The expanded name space permits an aggregate definition for a variable.

The lsc script will produce multi-column output. We visualize the output as a table consisting of rows and columns. A common definition for row and column is provided by the definition of a compound variable named **cell**.

```
cell=(
  # maximum number of cells
  integer maximum=0
  # maximum width based on entries
  integer width=0
  # current index within the cell
  integer index=0
  # content of the cell
  typeset entries
)
```

This defines the variable **cell**, with aggregate members **maximum**, **width**, **index**, and **entries**. A reference of **${cell.index}** provides the value associated with the index aggregate. Using the **eval** command we can create additional variables with the same aggregates. We can, for example, define variables **row** and **col** to have the same definition as cell:

```
eval row="$cell"
eval col="$cell"
```

Internationalization Support

ksh93 provides support for internationalization. Double-quoted strings preceded by a **$** are checked for message substitution. If the string appears in the message catalog, then ksh93 will substitute the string with the corresponding string from the message catalog. Otherwise, the string is unchanged.

In the lsc example, we display an error message of **"not found"** for any command line arguments that are not readable directories. The error message we provide is defined with internationalization support (see line 33 of Listing 1). If the shell variable **LANG** is defined to some locale other than **POSIX**, ksh will attempt to replace the error message using internationalization support. Otherwise, the message remains unchanged.

Executing **ksh -D** on a shell script will output all messages identified for internationalization. In the lsc script, for example, **ksh -D** will output the following message.

```
"${video[reverse]} not found ${video[reset]}"
```

KornShell Development Kit (KDK)

ksh93 is extensible through the KornShell Development Kit (KDK). You can write your own built-in functions in C and load them into the current shell environment through the **builtin** command. This feature is available on operating systems with the ability to load and link code into the current process at run time.

A built-in command is executed without creating a separate process. Instead, the command is invoked as a C function by ksh. If this function has no side effects in the shell process, then the behavior of this built-in is identical to that of the equivalent stand-alone command. The primary difference in this case is performance: the overhead of process creation is eliminated. For commands of short duration, the effect can be dramatic. For example, on SUN OS 4.1 wc on a small file of about 1000 bytes runs about 50 times faster as a built-in command than as a separate process.

In addition, built-in commands that have side effects on the shell environment can be written. Using the API, available through the KornShell Development Kit, you can extend the application domain for shell programming. For example, an X-Windows extension that makes heavy use of the shell variable namespace was added as a group of built-in commands. The result is a windowing shell that can be used to write X-Windows applications.

While there are definite advantages to adding built-in commands, there are some disadvantages as well. Since the built-in command and ksh share the same address space, a coding error in the built-in program may affect the behavior of ksh, perhaps causing it to core dump or hang. Debugging is also more complex since the built-in's code is now a part of a larger entity. The isolation provided by a separate process guarantees that all resources used by the command will be freed when the command completes; this guarantee does not apply to built-ins. Also, since the address space of ksh will be larger, this may increase the time it takes ksh to **fork()** and **exec()** a non-builtin command [though not on more advanced operating systems like Linux, which conserve memory and time by doing “copy-on-write” when they fork—ED]. It makes no sense to add a built-in command that takes a long time to run or that is run only once, since the performance benefits will be negligible. Built-ins that have side effects in the current shell environment have the disadvantage of increasing the coupling between the built-in and ksh making the overall system less modular and more monolithic.

Despite these drawbacks, in many cases extending ksh by adding built-in commands makes sense and allows reuse of the shell scripting capability in an application-specific domain.

In the lsc example, we need to determine the maximum string size within a list of strings. This is required to determine the initial number of columns in the multi-column display. We will also use this to determine the maximum width for a column of entries. A typical shell implementation would be given as:

```
(( max_stringSize = 0 ))
for fileName in *
do
if (( max_stringSize < ${#fileName} ))
then
   (( max_stringSize = ${#fileName} ))
fi
done
```

(See **Arithmetic Expressions**, below, for an explanation of **((** and **))**.)

To improve performance, we can re-write this function in C. In a simple example, the shell equivalent function required 0.58 seconds of CPU. The C built-in function provided 0.08 seconds of CPU for the same task. The function name is preceded with “b_” to indicate that it is a built-in function. When compiled, the strlenList.o object is then archived into a shared library. To reference the **strlenList** function, we must load it into the current ksh environment through the **builtin** command (see line 29 of Listing 1).

```
#pragma prototyped
#include "shell.h"
#include "stdio.h"
int b_strlenList(int argc, char **argv,
                 void *extra)
{
    register int max, n = 0
    char **cp = NULL;
    cp=argv;
    while ( *(++cp) ) {
        n = strlen(*cp);
        max = max < n ? n : max;
    }
    fprintf(stdout,"%d\n", max);
    return(0);
}
```

Functions:

ksh93 provides two methods for function definitions. The formats are given as:

```
function name
{
    body
}
name()
{
    body
}
```

The second function format is provided for compatibility with POSIX standards. The primary distinction is that of variable name scope. In a POSIX function, a variable definition has global scope. In the following POSIX function **bar**, variable **foo** is redefined to a value of 6.

```
typeset foo=5
bar()
{
    typeset foo=6
    echo $foo
}
bar
6
echo $foo
6
```

Variable definitions in ksh93 functions have local scope. In the following ksh93 function **bar**, a local variable **foo** is defined and has precedence over the global variable **foo**.

```
typeset foo=5
function bar
{
    typeset foo=6
    echo $foo
}
bar
6
echo $foo
5
```

Discipline Functions

ksh93 provides active variables through a series of *discipline* functions. From the shell level, you can write *get*, *set*, and *unset* disciplines. Through the KornShell Development Kit, you can also add disciplines unique to your environment.

When a variable is referenced, as in **$foo**, ksh will invoke the get discipline associated with **foo**. The default discipline is to simply return the current value associated with **foo**. From the shell level, you can define a **foo.get** discipline function.

The set discipline is called when a value is assigned to a variable. Within the set discipline, the special variable **.sh.name** is the name of the variable whose value is being set.

On line 31 of lsc, we define a **max_stringSize.get** discipline function. Every reference to **${max_stringSize}** will result in this function being executed. The value of the special **.sh.value** variable is the value returned from the discipline.

printf Statement

In ksh93, a **printf** statement is available following the ANSI C **printf** definition. This permits formatting specifications to be applied to each argument. To appreciate the differences between the standard **print** and **printf** statements, consider how you would output the contents of the **entries** array (from the lsc example), one per line. The standard print statement would display the file names as space-separated tokens on a single line. Using the **printf** statement with a **"%s\n"** format, however, would produce the desired results.

Arithmetic Commands

ksh93 statements of the form **((** ***expression*** **))** are called *arithmetic* commands. Arithmetic commands return True when the value of the enclosed expression is non-zero, and False when the expression evaluates to zero. The construct **$((*expression*))** can be used as a word or part of a word. It is replaced by the value of *expression*.

In the lsc example, line 38, we evaluate the value of the discipline function using:

```
(( .sh.value = $(strlenList ${entries[@]}) + 3 ))
```

ksh93 will evaluate the expression, which includes an assignment to the **.sh.value** variable. Note that the:

```
$(strlenList ${entries[@]})
```

will invoke the **strlenList** built-in function and return the maximum width of the strings (given as element values) in the **entries[]** array. We add 3 to this value for formatting purposes.

ANSI C Strings

An ANSI C string is defined by preceding the *single-quoted* string with a **$**. For example, **$'*'** is the literal asterisk, *****. With ANSI C strings, all characters between the single quotes retain their literal meaning, except for escape sequences. An escape sequence is introduced by the escape character \.

ANSI C string support provides an essential feature for shell programmers. Consider, for example, having to process variables with embedded tabs in their values. Without ANSI C string support, we would not be able to effectively test the value of the variable for embedded tabs. As an example, consider the following script:

```
print "foo\tbar" > /tmp/foobar
read aline < /tmp/foobar
if [[ "${aline}" == "foo\tbar" ]]
then print TRUE
fi
```

The comparison (see **Conditional Commands**, below) will fail. We can replace the conditional with ANSI C strings and ensure proper functionality. The example above should be rewritten as:

```
print "foo\tbar" > /tmp/foobar
read aline < /tmp/foobar
if [[ "${aline}" == $'foo       bar' ]]
then    print TRUE
fi
```

On line 45 of Listing 1, we must test to see if the directory is empty. The preceding **entries=(*)** in an empty directory will set the entries variable to the literal asterisk if no files are found.

Conditional Commands

A conditional command in ksh93 evaluates a test-expression and returns either True or False. Conditional commands can be used as part of an “Or list” (**||**), “And list” (**&&**), or as part of an if-elif-else command. Conditional commands have the format:

```
[[ test-expression ]]
```

When used in conjunction with an “And list”, ksh93 evaluates the test-expression and will execute the “And component” only if the test-expression evaluates to True. We use a conditional command as part of an “And list” such that the return statement will be executed only if the test-expression is True.

```
[[ ${entries[0]} == $'*' ]] && return 2
```

Iteration Control

The **for** command has two formats. The traditional format is provided to iterate on each word in a list. The format is:

```
for variableName [ in word-list ]
do   compound-list
done
```

An arithmetic **for** command has been provided that is very similar to the C programming language **for** statement. The format is:

```
for (( initExpr ; condition ; loopExpr ))
do   compound-list
done
```

The ***initExpr***ession is evaluated by ksh prior to executing the **for** command. The ***condition*** is then evaluated prior to each iteration of ***compound-list***. If the ***condition*** is non-zero, then ksh executes the ***compound-list***. The ***loopExpr***ession is evaluated at the end of each iteration.

Name Referencing

A new typeset option has been added for name referencing. Using **typeset -n** ***nameReference=variableName*** will associate ***nameReference*** with ***variableName***. A special alias, **nameref**, is provided as the equivalent for **typeset -n**. A shell script may use the reference name to refer to the variable name. Name referencing provides a convenient mechanism to pass the name of compound variables, or arrays, to ksh functions. This is more efficient than passing the variable's content.

In the lsc example, function **setOutput** must add the directory entries to the appropriate row and column. We could have defined separate functions named **addToRow** and **addToColumn** for this purpose. The main body of the functions, however, would be equivalent. Instead, we opted to write a single function **addToCell** that uses a **nameref** to the cell type passed as a parameter.

The **addToCell** function accepts three arguments, of which the first two are required. The first argument is the cell type and must be either **row** or **col**. We create a nameref using the local variable **cell** to be equivalent to the cell type specified. A reference to **${cell.index}** would therefore be equivalent to **${row.index}** or **${col.index}**.

FPATH

ksh functions are not inherited across invocations of ksh. A child shell process, for example, does not have access to the functions defined within the parent ksh invocation. This has historically limited the re-usability of ksh functions. As a solution, ksh93 will search the colon-separated list of directories given by the **FPATH** variable value, for an executable file with the same name as the function. In the lsc example, we can eliminate the last statement:

```
   lsc "${@}"
```

The **FPATH** can then be set to the directory containing the lsc file. From the shell level, we can now call lsc. ksh93 will load the lsc script and will call the lsc function with the command line arguments specified. Note that the supporting functions defined in the lsc script are available to the lsc function.

A function autoload feature is provided, in which an auto-loaded function definition is loaded and retained within the ksh93 environment upon the first reference to the function name. This provides better performance since the search and load steps are eliminated for subsequent references.

Summary

ksh93, the latest major revision of the KornShell language, provides an alternative to Tcl and Perl. As a programming language, it has comparable speed and functionality to each of these languages. Like Tcl, it is extensible and embeddable with a C language application programming interface. The New KornShell, ksh93, and the Tksh products are available from Global Technologies, Ltd., Inc., 5 West Ave, Old Bridge, NJ 908-251-2840.

David G. Korn AT&T Research, Technical Manager

Charles J. Northrup Global Technologies Ltd., Inc., CIO

Jeffery Korn Princeton University, Computer Science Department

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Samba in the Home and Office

Peter Kelly

Issue #27, July 1996

A Linux computer can be a great server, not only for other Linux computers, but also for computers running other operating systems. Peter gives an example of how to do this effectively.

Linux users consistently experiment, finding uses for Linux far beyond what was even thought of five years ago, when Linux itself was an experiment. For many users, Linux has been far more than just another Unix clone; people want something extra. The fact that Linux comes with networking built in, including all the tools needed for connecting to the Internet, makes it easy to pick Linux. The decision is not based on the pure cost of Linux (negligible), the decision is based on the vast amount Linux enables you to do with your PC.

With a small investment (or perhaps just rearranging hardware), you can have a complete home network with two computers, even if only one of the computers runs Linux. Linux is also a good network server for an office, and setting up a home network can give you the experience you want to set up an office network.

I didn't know Samba even existed until a friend showed me his Linux drives on his WFWG 3.11 file manager. He showed me that he could copy files back and forth just as if the Linux drives were local. What he built was a small home LAN that consists of 2 computers. He wanted Samba installed on his main Linux server so that his kids could run large programs on the server, without having to take up limited hard drive space on their machine.

Home Networks

The home LAN installation was easy. The sources compiled “out of the box,” and the default settings for the installation were used:

/usr/local/samba/bin for binaries /usr/local/samba/lib for configuration files /usr/local/samba/locks for samba locks /usr/local/samba/man for samba manpages /usr/local/samba/var for samba logs

Some Linux distributions come with Samba included, in which case /usr/bin/, /var/samba/, and /usr/man are more likely places for files, and the configuration file is usually kept in /etc/smb.conf.

A basic configuration file (which defaults to /usr/local/samba/lib/smb.conf) would consist of some basic entries like **[global]**, **[printers]**, and **[homes]**. These specify global configuration details that describe the environment, printers that are available to clients, and user-specific home directories. You also create your own sections for other directories which you wish to export to other machines; see the **[dos]** entry.

The general structure of a configuration file is like that of a Windows .INI file, with sections of statements, and comments on lines of their own that start with **;** characters.

```
;-----------------------------------------------
; Service(s): [globals] [homes] [printers]
;-----------------------------------------------
[globals]
    status=yes
    printing = bsd
    guest account = dos
    browseable = yes
    lock directory = /usr/local/samba/locks
    domain master = yes
    os level=33
[homes]
    comment = Home Directories
    guest ok = no
    read only = no
    browseable = yes
[printers]
    comment = All Printers
    path = /usr/spool/public
    printcap name = /etc/printcap
    printable = yes
    public = yes
    writable = no
    create mode = 0700
    browseable = yes
    load printers = yes
[dos]
    comment = Dos public directory
    path = /g/dos
    public = yes
    writable = yes
    printable = no
    guest ok = yes
```

This configuration file allows Samba to serve a printer, a shared directory, and home directories to network clients. There must be a user named “dos” in the /etc/passwd file with no password or a known password (and preferably /bin/false for a shell, at least if there is no password) in order to have a “world shared” directory, as shown in the **[dos]** section. Make sure the directory (in this case, /g/dos/) is owned by this user “dos”.

If you want more explanation of this file, use **man smb.conf**.

At boot time, /usr/local/samba/lib/rc.samba starts the smbd and nmbd daemons and has them wait for client connections. I run them as daemons because I want some extra speed when I issue a request from Samba.

A normal /usr/local/samba/lib/rc.samba file looks like this:

```
#!/bin/sh
PATH=/usr/local/samba/bin:$PATH
smbd -D -d1
nmbd -D -d1 -G MY_WORKGROUP \
 -n THIS_MACHINE_NAME
```

For both smbd (the file and printer server daemon) and nmbd (the nameserver daemon), the **-D** option says to act as a daemon, working in the background. The **-d1** says to be a little more verbose than usual with debugging messages; you will probably want to remove that once your network is stable. The **-G** option specifies the netbios group (or lanmanager domain) that the computer should be part of, and the **-n** option can be used to specify the name of the server on the network; if it is ommitted, the server's normal hostname is used instead.

Some people disagree with running the Samba daemons as daemons that are always running in the background, and prefer to run them from inetd. This gives slower network response, but if demand is light it reduces load on the server. The basic entry in the /etc/inetd.conf file that comes with most distributions is:

```
netbios-ssn stream tcp nowait root /usr/sbin/smbd smbd
netbios-ns  dgram  udp wait   root /usr/sbin/nmbd nmbd
```

You obviously have to provide the correct path to the binaries to have them called via inetd.

On the client end, whether it is running WFWG 3.11 or Win95, TCP/IP should be the default protocol. Each machine that has services should show up in browse list. To connect to the **dos** shared service on the Samba server, you could just use the basic **net** command.

```
c:\> net use d: \\MACHINE_NAME\dos
password : XXXXXX
```

This would be sufficient for a small LAN that needs to share a couple of home dir's and a printer. Make sure there is a valid /etc/printcap file with proper entries for your printer; setting up standard Linux printing is beyond the scope of this article. You can do **man printcap** for additional information on the syntax this file requires, or read the Linux Printing HOWTO which provides much more detailed information on printing setup.

Office Networks

What I have done with Samba at my office with about 20 computers on the LAN is more complex, but was not difficult to set up and is very stable. If you are only going to set up a home network, you can probably skip to the end of this article. If you already administer a Windows 95 network, the Windows-specific information presented here probably isn't new to you.

The Samba server is running Linux 1.2.13 (elf) with samba-1.9.15p8 running on a 100MHZ Pentium with 16M ram, 4G SCSI disk, and a 4G DAT.

The Samba clients are running WFWG 3.11 and Win95 on various i486's with 8-16M of ram.

Win95 Features

1. **Policies via the new Win95 Registry:** The registry is a new format that stores all settings for users and system specific settings. There is a Registry Editor that is needed to modify settings held within the registry.

Policies are what a user can and can't do on the system and what they can and can't do on the network. There is also a Policy Editor to edit user and computer policies from.

2. **Remote logon authentication:** where all Win95 client machines have **all** logins to the network and client machine be authenticated via Linux accounts.

This is where you would set up Samba to be the Domain Controller.

1. Log in to the Win95 workstation with domain password if in the smb.conf file you have **user = security** and you set up the Win95 registry to “require domain authentication before access to windows”.

2. Samba has a function to read the /etc/passwd, look up the uid and verify the password the user entered is correct.

3. If the password is correct, a result code is sent to the Win95 machine for “access granted”.

4. If there is a **[netlogon]** entry in the smb.conf file, this directory is checked for a config.pol file that the Win95 machine wants to read for the policies for the machine and user. This must be set up in the Win95 setup in the registry with “remote update” and “automatic path” in the Network settings of the registry.

5. If you have **logon script = %U.bat** in the smb.conf file, the specified batch file will be executed on the client for each user. (%U is replaced by the user name, so %U.bat becomes username.bat—you can have a separate batch file for each user). Make sure the logon scripts (which will be kept in the directory specified in the **[netlogon]** section) use DOS-style line endings; a good way to ensure that is to use a DOS editor on a DOS system to create the files.

The logon scripts are good if you use them. Only simple DOS commands are required:

```
net time /set /yes
```

would match up the time on the server to the workstation. It is nice to have to maintain time on only one system. Having policies stored on the server is another good idea. You can update the policy file from another workstation and the next time a user logs in, the policy file is read and the client registry is updated—automatically!

All the necessary information about these Win95 specifics is found in the Windows 95 Resource Kit. Other discussions of these topics can be found at:

- comp.os.ms-windows.networking.tcp-ip for Windows and TCP/IP networking.
- comp.os.ms-windows.setup.win95 for setup, hardware, and driver issues in Win95.
- comp.os.ms-windows.networking.win95 for Win95 to Novell, TCP/IP, other nets.

For the larger LAN, the smb.conf file looks like this:

```
; ---------------------------------------------
; Service(s): [globals] [homes] [printers]
; ---------------------------------------------
;
[globals]
    status = yes
;    This enables or disables logging of
;    connections to a status file that
;    smbstatus can read. Yes by default.
    printing = bsd
;    See manpage for your system.  This
;    one is Linux and requires BSD
;    printing entries.
    guest account = dos
;    for printing to work
    invalid users = root, @wheel
;    don't let super-users access from
;    the network
    browseable = yes
;    By default, everything is browsable
;    unless specified elsewhere in
;    services sections
    hosts allow = 10.10.1.
;    you can specify who is allowed in
;    10.10.1. is a class C network that
;    never sees the internet
    lock directory = /var/lock/samba/locks
;    Locks for sessions
    log file = /var/log/samba/log.%m
;    Individual logfile for each client
;    machine
    syslog = 2
;    Anything level 2 and below will also
;    be sent to syslogd
    message command = /bin/mail -s  \
        'message from %f on %m' \
         pkelly < %s; rm %s
;    If someone sends a "win-popup"
;    message - mail it to sys admin
    socket options = TCP_NODELAY
    dead time = 30
;    Close any unused sessions after
;    30 minutes - good for big network.
    read prediction = yes
;    Speeds up reads from disk
    share modes = yes
;    For a 'dos share' type of use
    max xmit = 8192
;    This  option controls the maximum
;    packet size that will be negotiated
;    by Samba.
    os level = 33
;    This integer value controls what level
;    Samba advertises itself as for  browse
;    elections. See BROWSING.txt for details.
    security = user
;    For /etc/passwd to be used
;    for Domain Logons to work
    domain master = yes
;    Master browser
    domain logons = yes
;    For network authentication
    logon script = scripts/login.bat
;    Single batch file to be executed
;    when users logon to the network
;    These are simple dos Batch files
;    logon script = scripts/%U.bat
;    individual batch files - where %U
;    is the person's logon name
[netlogon]
    comment = Network Logon Services
    path = /u/netlogon
;    This is the default setting for
;    the Win95 machines to look for
;    the config.pol file and and .bat
;    scripts to run for the client.
    writable = yes
;    I make this writable so I can add
;    or delete items in the config.pol
;    file and update the .bat scripts
    guest ok = no
;    guests not allowed on our network
[homes]
    comment = Secure Home Directory for : %u
    path = /u/users/%u
;    This will match up the user's name
;    to their home directory.
    guest ok = no
;    guests not allowed on our network
    read only = no
;    Let people write to their own
;    home directory.
    create mode = 640
;    This is handy!  I can set this for
;    each service differently.  So users
;    can create files people can't
;    delete in their home dir.
    writable = yes
;    The above "read only = no" does
;    this, but I like to be safe :)
    browseable = no
;    Don't let people know who's home
;    directories are there.
[printers]
    comment = HP4L in BSC Office
    path = /usr/spool/public
    printcap name = /etc/printcap
;    "man printcap" for details on the
;    syntax for your printer.
    printable = yes
    public = yes
;    Everyone connected can print!
    writable = no
;    Default
    create mode = 0700
;    Default
    browseable = no
;    Default
    load printers = yes
;-------------------------------------------
; fcp Services
;-------------------------------------------
[programs]
    comment = Shared Programs
    path = /u/programs
;    This is where I store the shared programs
;    and have only read access for people.
    public = yes
;    Public - but not writable for all.
    writable = yes
;    Writable for the sys admin to install
;    new programs.
    create mode = 644
;    What the ownerships are to be
[data]
    comment =  Data Directories
    path = /u/data
    public = no
;    You have to be a member of this group
;    who owns these files to be able to
;    work on the files
    create mode = 770
;    This is for all the database files that
;    need to be shared and group writable.
;    The 770 is needed because dir-'s are
;    sometimes created and need to be
;    executable in order to work right.
    writeable = yes
;    Allow people to write and delete files
    volume = "Data on Fileserver"
```

I totally replaced a LANtastic network with Win95 and Windows for Workgroups as the clients and Linux Samba servers for the servers with that configuration. TCP/IP is the only protocol used, and the peer-to-peer networking people were used to with LANtastic is still available with the client network software.

I have totally eliminated all network-related errors I was getting from a multi-user C-Tree database written by Angus Systems Group Ltd. All disk accesses from the Samba server have dropped to about half the time they used to take, and the system as a whole performs much better than on the previous MS-DOS fileserver. The MS-DOS .EXE's load three times faster over the network.

Peter Kelly (pkelly@ets.net) is a Network Administrator for JDP Computer Systems and Systems Software. He also does database and network functions for O & Y Properties Inc.'s 1 First Canadian Place. Sometimes he does leave his Linux X-Workstation to go outside to eat or to attend part-time classes at the University of Toronto's Computer Science Facilty.

Archive Index Issue Table of Contents

Advanced search

Maceater

Jonathan Gross

Issue #27, July 1996

Linux provides a very inexpensive alternative to high-cost commercial servers and routers.

Many small businesses are becoming interested in Internet connectivity, but they are unwilling (or unable) to fork over the cash for the necessary hardware. With routers costing close to $2000 and bandwidth as expensive as it is, there doesn't seem to be a viable method of putting small networks on the Internet. However, that depends on what you need—high bandwidth is not required for basic services, like e-mail, and Linux provides a *very* inexpensive alternative to high-cost commercial servers and routers.

The Birth of “maceater”

About six months ago, I was sitting around drinking beer with Rob, a graphic artist friend of mine. He was complaining about the lack of connectivity at his office and trying to figure out how to get at least e-mail at work. His network consisted of Apple PowerMacs, Quadras, and Performas connected via Ethernet and speaking Appletalk, Apple's networking protocol.

A bell in my head went off, and I set about building a Linux box that would solve his problems. I went out the next weekend and bought a used 386DX25 with 4MB of RAM, a 340MB hard drive, and an NE2000 ethernet card for $600. Add to that a 28.8 modem for another $200, and the frame for “maceater” was born. I screwed everything together, and fired it up with DOS to make sure the hardware would actually function. Finding no problems, I repartitioned the drive and built a lean, mean, 1.2.13 a.out Slackware-based system with **IP_FORWARD** turned on.

Everything that wasn't needed for networking, system administration, and basic user functions I left out. The resulting system used very little space, leaving plenty of space for user directories, swap, and building new programs.

After some hacking with the PBX, I got the phone line hooked up and a PPP link established with our ISP (from whom we had purchased a dedicated phone line). Our domain had been registered, and we had a full class C to start assigning IPs. I gave Rob a crash course in Unix, DNS, and pico, and we were off. Several users had requested dial-in PPP, so we started assigning office workstations one end of the IP range and off-site workstations to the other.

We had to install MacTCP on all the machines and reconfigure them to speak TCP to each other (and to “maceater”, the Linux box). Setting up the Macs was fairly easy, and despite the tendency of MacTCP (and its newer sibling “Open Transport”) to puke all over System 7.5.3 at random intervals, we had everything routing internally in about four hours. We could ping all the Macs from maceater, and all the Macs could telnet into the Linux box. A word of warning: make sure you apply all the patches from Apple for MacTCP and Open Transport, as they have a number of potentially nasty bugs in them.

Linux and Appletalk

As we were fighting with the Macs, the topic of disk space (and the lack thereof on the Macs) came up. Another bell went off in my head, and I grabbed the source for Netatalk, a package for Unix boxes that allows them to speak Appletalk and perform a number of services, including printing from a Unix machine to an Appletalk-connected printer, printing from a Mac to a Unix printer, and accessing Unix file systems from a Mac. (Netatalk is available at www.umich.edu/~rsug/netatalk.)

Netatalk works best with a newer kernel, so I built a 1.3.74 kernel (the latest kernel available at the time of the installation) with Appletalk enabled and IP forwarding on. I started to compile Netatalk and left for dinner. Three hours later (it's a 386, remember), I installed the binaries, and fired up afsd, the apple file system daemon. After reading some of the docs and setting up a mountable volume, I re-opened the Chooser on one of the Macs and presto! There was an entry for Linux sitting amongst the other Macs. Clicking on “Linux” opened up a folder that contained /usr/local/bin, the volume that I had mounted, looking like any regular Mac folder. I copied some files back and forth, and since nothing was corrupted, declared it a success—and much easier than using something like Fetch to move files around.

It took about a full weekend of work, mostly because compiling anything on a 386 is painfully slow. We did as much remote compiling on my workstation at work (a DX4-100) as we could, transferring the resulting binaries over to the maceater.

Thus far, maceater performs the following major functions:

1. Runs sendmail for hlm.com (version 8.7.5)

2. Functions as a pop-mail server for ~20 workstations

3. Provides the primary nameservice for hlm.com (bind, version 4.9.3)

4. Runs an experimental web server (Apache 1.0.0)

5. Provides one line of dial-in PPP or shell access for employees (PPP 2.0.0e)

6. Routes packets for the entire network

7. Serves as an FTP site

8. Acts as an native Appletalk fileserver (Netatalk version 1.3.3b2)

As of this writing, maceater has been up for 82 days, during which we have compiled and upgraded sendmail, bind, and pppd. Load averages about 0.5, depending on how many people are running shells.

All in all, the hardware for maceater cost us about $800 and a weekend to get it running smoothly, although much of that was fighting with MacTCP and ironing out problems with our ISP. Hardly anyone in the office knows the Internet gateway/fileserver for their beloved Macs is an old clunky-looking PC sitting on the shelf in their supplies closet and was built in a weekend from spare parts. If only they knew...

Jonathan Gross is Editor of *WEBsmith* magazine and likes to infiltrate Windows and Macintosh networks with Linux boxes in his “spare” time.

Archive Index Issue Table of Contents

Advanced search

Object Databases: Not Just for CAD/CAM Anymore

Gregory A. Meinke

Issue #27, July 1996

As Esther Dyson put it, “Using tables to store objects is like driving your car home and then disassembling it to put it in the garage. It can be assembled again in the morning, but one eventually asks whether this is the most efficient way to park a car.”

Applications are getting more complex and dependent on larger quantities of persistent data. Most applications rely on relational databases to manage this abundance of data. However, object databases have become another attractive option for a variety of applications. As Esther Dyson put it, “Using tables to store objects is like driving your car home and then disassembling it to put it in the garage. It can be assembled again in the morning, but one eventually asks whether this is the most efficient way to park a car.” [ORF96]

Object databases got their start in the CAD/CAM world. Object databases support the programmer-defined data types and complex relationships that CAD/CAM applications demand. To manage the additional complexity, object-oriented programming languages are becoming the standard for developing today's mainstream applications. Using an object database is a natural extension to this language choice. Object databases provide better performance, faster development, and more robust programs. This article examines these claims and looks at a public domain object database, the Texas Persistent Store.

Faster Development and More Robust Programs

Relational databases use a separate programming language, called “Structured Query Language” (SQL). Occasionally, a similar, but non-standard, query language is used to define the layout of the tables and interaction with the database. One shortcoming of relational databases is they can store only a limited set of data types; in order to store objects of more complex types they must somehow be mapped into the primitive types supported by SQL. In contrast, object databases use an object-oriented programming language for data definition and manipulation of the objects within the database. This eliminates the “impedance mismatch” of trying to map your complex objects and relationships into the limited data types and tables of the relational world. The reduction of error-prone translation code lets the programmer concentrate on the semantics of the object's behavior instead of the syntax of storing and retrieving the object. Without embedded SQL, runtime storage errors are eliminated.

While relational databases must use SQL to recreate these relationships at runtime, object databases capture the inter-object relationships directly in the database. This makes development easier by reducing the lines of codes written and the lines of code executed at runtime. A positive side effect of this is that you will not have to make any design compromises to accommodate join tables or add foreign key identifiers to your classes.

Object databases work on the principle of starting from a named object and navigating to other objects within the class hierarchy. These named objects can be singular objects or containers of objects. Navigation to the contained objects allows an object database to immediately load objects without needing to query. This adds up to less code for the programmer to write and test, making for more robust programs and shorter development cycles.

Increased Performance

If the faster development and more robust programs were not enough to convince you, let's try increased performance. The goal of many vendors is to make access to persistent objects as fast as access to transient objects. This is an impossible goal because loading a stored object requires accessing a disk and possibly a network. Sophisticated client caching and memory management techniques provide very low overhead once the object is loaded into memory. Some implementations, like the Texas Persistent Store and ObjectStore, have no overhead once the object is swapped into memory. Most relational systems do not cache the results on the client system, thereby incurring unnecessary network transmission and additional queries on the next access.

Unfortunately, there are few current benchmarks that compare relational and object databases to back up these performance claims. There are two common object database benchmarks: the Engineering Database Benchmark—also known as the 001, the Sun Benchmark or the Cattell Benchmark—developed at Sun Microsystems, and the 007 Benchmark, developed at the University of Wisconsin. The 001 Benchmark was intended to prove that object databases out-perform relational databases in engineering applications. The results showed that the measured object databases were 30 or more times faster than the benchmarked relational databases [CAT92]. The 007 tries to provide a broader mix of measurements, including multi-user access. Implementations of the 007 benchmark are audited by the University of Wisconsin and should be available from participating database vendors [LOO95].

Some advanced object database features include clustering and configurable object-fetching policies. Clustering allows programmers to indicate a collection of objects will be used together. All the objects in a cluster are loaded into the client cache when any one of them is requested. This reduces the number of disk and network transfers to load the client cache. Some vendors allow configurable object fetching policies that allows customization of the volume of extra data the server sends along. These performance gains usually come at the expense of increased lines of code and extra performance analysis.

To Swizzle or Not to Swizzle

Object databases come in two models. One requires you to inherit from a vendor-supplied persistent base class, a la the Object Database Management Group (ODMG) standard [CAT96]. The persistent base class provides the interface for making database requests of the objects. The other model is a pointer swizzling technique that allows you to use the pointers to persistent objects as if they never left memory. I believe the pointer swizzling technique is superior in programming model and flexibility, and I will cover this technique in further detail.

Pointer swizzling is the changing or mapping of the on-disk format pointer to the in-memory format pointer. Swizzling of pointers takes place transparently to the client program. When the program uses a pointer to an unloaded object, a segmentation violation occurs. The vendor library traps that violation and fetches the object from the database. It then sets the pointer to the newly loaded object and returns control to the client program. The client program is totally unaware that a database access occurred.

The use of standard C++ memory management techniques allow the same application code to work on both transient and persistent objects. Objects are constructed using the C++ placement **new** operator. Allocation in persistent memory implicitly stores the object in the database. Removing objects from the database is as simple as calling the C++ **delete** operator.

The Texas Persistent Store

The Texas Persistent Store is a public domain pointer swizzling object database for C++. Texas was created and is maintained by the University of Texas at Austin. The current 0.4 beta release supports the Linux 1.2.9, Solaris 2.4, SunOS 4.1.3, and DEC Ultrix 4.2 platforms, all using the GNU g++ 2.5.8 or g++ 2.6.3 compiler. It also supports OS/2 2.1 using the IBM CSet compiler and the Sun

3.0.1 C++ compiler. White papers and the source are available via anonymous ftp from cs.utexas.edu, in the directory /pub/garbage, or from the OOPS Research Group's home page at www.cs.utexas.edu/users/oops.

My setup consists of Slackware 1.2.8 running on a 486/100 with 16Mb of memory. Texas installed and ran on my Linux machine with minimal hassle. Due to a compiler template bug in g++ 2.6.3, you must patch the compiler or modify the makefiles to use the **-fexternal_templates** compiler switch. The documentation describes both the bug and the fixes, making the library installation fairly painless. Texas comes with a few test programs and examples to ensure the system is performing correctly.

Texas Features

To start coding using the Texas library, you have to understand only four easy features: the initialization macros, opening and closing the persistent stores, finding and creating named roots, and allocating objects into persistent memory. Here, I discuss each of these features briefly and then jump in and look at some code.

Initialization of the Texas library takes place by invoking the **TEXAS_MAIN_INIT()** macro. This macro sets up the signal handler, reading in the schema information and virtual function tables. The **TEXAS_MAIN_UNINIT()** macro removes signal handlers and resets the system to its previous state.

Use the **open_pstore()** function to open a database. If the file does not exist, the database is created and opened. Opening a database starts a transaction. You can manipulate the transactions during the lifetime of the program by calling **commit_transaction()** or **abort_transaction()**. **commit_transaction()** will save all of the current persistent objects to disk and start a new transaction, while **abort_transaction()** throws away all of the dirty pages and starts a new transaction. To close the database use the **close_pstore()** function. This implicitly calls **commit_transaction()** and closes the file database. If you do not want to commit the current work you can call the **close_pstore_without_commit()** function.

Named roots are your entry points for retrieving the persistent objects from the database. They provide the mechanism by which a program can directly navigate to objects or search containers for objects. You create a named root by using the **add_root()** function. A named root is retrieved with the **get_root()** call and the database is queried for the existence of a named root with the **is_root()** function.

The Texas memory allocation macros, **pnew()** and **pnew_array()** hide the C++ placement operator **new**. The allocation macros also hide the instantiation of the **TexasWrapper** template classes. The TexasWrapper class handles the creation and registration of schema information with the database. The schema information holds the layout of the class attributes while in the database.

Hello Persistent World

Let's take a look at an example of how easy it is to make things persistent in Texas. Sticking with tradition, we write the familiar “hello world” program, but with a persistent twist: we record how many times the program has been executed. Listing 1 shows the code for this task.

First we initialize the Texas library, passing it the argc and argv arguments from main. The program then opens up a persistent store named “hello.pstore” in the current working directory.

The persistent store is queried to see if a named root **"COUNT"** exists using the **is_root()** function. If the named root does not exist, allocation of a new integer takes place. The new integer is initialized to zero and named **"COUNT"** using the **add_root()** function. Otherwise, we retrieve the integer from the database. The counter is incremented and the results printed to standard output. All the dirty objects are committed and the database is closed. The library is uninitialized and the program exits.

With each successive run of the program, the integer named **"COUNT"** will be retrieved, incremented and rewritten to the database. You will notice this is all quite seamless: there are no explicit calls to queries, inserts, loads, or saves.

Pointer Swizzling Examined in Texas

Next, we briefly explore how Texas swizzles pointers at page fault time and handles memory management. This is by no means a complete discussion of these topics. Readers interested in learning more about the Texas system should download the white papers and source code.

Texas uses conventional virtual memory access mechanisms to ensure the first access of any persistent page is intercepted by the Texas library. This page is loaded from the database and scanned for persistent pointers. Swizzling to in-memory addresses occurs on all persistent pointers on that page. All new pages are reserved and access protected. This faulting and reserving process repeats as the program traverses the object hierarchy of unloaded pages. The pages of virtual memory are reserved one step ahead of the actual referencing page. This implies the program can never see a swizzled pointer, only access protected pointers to unloaded objects. The Linux implementation uses the **mprotect()** system call to set up the access protection on the pages. An in-depth discussion of this topic can be found in the Texas white paper presented at the Fifth International Workshop on Persistent Object Systems [SIN92].

Texas allows you to access multiple databases, each with its own persistent heap. The standard transient heap and stack are also available for non-persistent memory allocation. Texas does not partition its address space into regions, allowing pages from different heaps to be interleaved within memory. Each page must belong only to a single heap, so Texas maintains separate free lists for each heap. A new page is created when the free list is empty or no free memory chunk available is large enough. New pages are partitioned into uniformly sized memory chunks large enough to hold the object being allocated. All of the other chunks are linked onto the free list. This uniform chunking of a page makes for trivial identification of the object headers on the page. Only the first header of a page needs to be examined to determine the size of all memory chunks on that page. The alignment of the other object's headers follows trivially. The object's header stores the schema information for the object so it can be identified and correctly swizzled.

A More Complex Example

While the Hello Persistent World program is not very exciting, it shows the minimal effort needed to make an object (an integer in this case) persistent. The next example demonstrates the power of object databases to capture the relationships between objects. This contrived example shows several many-to-many relationships. It also exposes some of the current deficiencies in the Texas library. The example is a system to track the many different research papers and books that clutter my office. See Figure 1 for a class diagram using non-unified Booch notation. The design file and the source code for both examples are available on my home page at www.qds.com/people/gmeinke.

The class diagram shows class **PublishedWork**, an abstract base class for all published material. It presents trivial methods for querying the object for its title, price, the number of pages, and a list of authors. The relationship between an **Author** and their **PublishedWork** is an example of a many-to-many relationship. The relationship between a **Publisher** and the **Books** they have published is one-to-many. Expressing these complex relationships in relational databases is awkward due to the foreign keys and the intermediate join table needed for the many-to-many relationships. By contrast, Texas handles these complex relationships with C++ containers and stores them directly in the object database. No compromises are necessary to the object design for foreign key data members.

Current Limitations and Future Work

Current limitations of the Texas library include the lack of multi-user support and the inability to query containers to find certain instances of objects. The query limitation stems from the fact that there are no containers provided with the Texas library. Most commercial object database vendors provide a set of optimized container classes that support queries. These limitations are minor if what you need is a very fast, single user, persistent store of objects. Another limitation is the inability to treat persistent and transient objects transparently. You cannot discover what heap an object is allocated on; this causes problems in objects with pointers to other contained objects. While this is a minor limitation for smaller programs, it does affect development of larger, more complex, multiple-database programs.

The future of Texas looks bright. It is a robust and efficient single user, portable library. A colleague and I are planning to port Texas to Windows NT. This will round out support for the most popular platforms, Solaris, Linux, and NT. We also plan to provide minor enhancements for the transparent treatment of the heaps. STL and a persistent allocator may provide some relief for lack of container and query support, but multi-user support is still off in the future.

Conclusions

Object databases are not the silver bullet of software development, but they do provide a more robust and natural programming environment for people already using an object-oriented programming language. They provide better performance and more performance tuning options than relational databases. For small to medium sized single-user projects, the Texas database is an attractive choice; for larger multi-user projects, you may want to check out ObjectStore from Object Design, Inc. ObjectStore supports a large number of platforms and compilers—unfortunately, not Linux. ObjectStore is a very fast and flexible object database product. For more information on ObjectStore, visit their home page at www.odi.com or subscribe to the ObjectStore development mailing list. (To subscribe, send e-mail to ostore-request@qds.com with no subject and **subscribe** in the message body.)

Special thanks to Rob Murray of Quantitative Data Systems and Craig Heckman of Superconducting Core Technologies for their great comments and help.

[ORF96] Robert Orfali, Dan Harkey, & Jeri Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, Inc. pp. 164, 1996.

[CAT92] R.G.G. Cattell, and J. Skeen. Object Operations Benchmark, ACM Transactions on Database Systems,17(1):1-31, 1992.

[LOO95] Mary E. S. Loomis, *Object Databases: The Essentials*, Addison-Wesley Publishing Company, pp. 197-200, 1995.

[CAT96] R.G.G. Cattell, *The Object Database Standard: ODMG - 93*, Release 1.2, Morgan Kaufmann Publishers, Inc., 1996.

[SIN92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson, Texas: An Efficient, Portable Persistent Store, Fifth International Workshop on Persistent Object Systems, 1992.Object Databases: Not Just for CAD/CAM Anymore.

Greg Meinke (gmeinke@qds.com) works at Quantitative Data Systems, Inc. on distributed business systems using C++, CORBA, and ObjectStore databases.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Advanced search

Serving Two Masters

Michael K. Johnson

Issue #27, July 1996

You installed Windows 95, and now you can't boot Linux. Don't panic. The fix is simple, and doesn't require removing Windows or Linux.

In spite of all the Linux zealots who infect the comp.os.linux.advocacy Usenet newsgroup with cries for the total elimination of MS-Windows—and usually Microsoft, as well—there are many people who want to be able to switch between Linux and Windows 95. Many users who were using **LILO** to choose easily between Windows 3.*x* and Linux installed Windows 95, and then found that Linux would no longer boot. Some who use **LOADLIN** also found that they were in trouble.

LILO

Windows 95 follows a “world domination” strategy when it is installed; it overwrites the *master boot record* on the hard drive with one which, by default, only boots Windows. The master boot record is essentially a very small program that is loaded from a fixed place on the hard drive as the first step of loading an operating system from the hard drive. The boot record installed by old versions of MS-DOS and by Windows 95 is very limited: it is only capable of allowing the boot process to proceed to the single partition which is marked as the “active” partition.

In contrast, the LILO loader that comes with Linux installs a boot record which allows you to choose to continue booting from any partition quite easily. In fact, LILO even allows you to boot from partitions on the second drive, something that the DOS master boot record cannot do. LILO's boot record *is* capable of booting Windows 95.

Notice that I didn't call LILO's boot record a “master” boot record. That's because while it can be installed as the master boot record, it doesn't have to be. On each partition on your hard drive, there is another boot record. When the DOS master boot record boots from the active partition, it does so by loading the boot record on the active partition. LILO's boot record can be installed either as the master boot records on the hard drive or as the boot record for your Linux partition.

See sidebar

In order to use LILO, therefore, you need to either install LILO's boot loader as the master boot loader, or install it as the boot loader on your Linux partition and make your Linux partition active. The advantage to installing it as the boot loader on your Linux partition and making your Linux partition active is that the next time you install Windows, all you will have to do to use LILO to choose your operating system is use the DOS **fdisk** program to make your Linux partition active.

Recovery recipe: Use an emergency boot floppy or boot from your installation boot/root floppy or floppy set. Get a shell prompt, probably either by choosing it from a menu or by pressing Alt-F2 (see the documentation for your Linux distribution if you don't know how to get a shell prompt). Then execute the following commands:

```
mkdir /mnt
mount -t ext2 /dev/rootdevice /mnt
/mnt/sbin/lilo -r /mnt
umount /mnt
```

This assumes that /mnt does not exist; if it already exists, you don't need to create it. ***rootdevice*** is the name of the device that holds your root filesystem, such as hda2 (second partition of your first IDE hard drive) or sda1 (first partition of your first SCSI hard drive). If you don't remember which it is, you may have to use the fdisk program, which should be included with both installation programs and emergency boot disks, to find it. The **-r /mnt** part means to pretend that /mnt is your root directory. If your distribution didn't put lilo into the /sbin directory, you may have to look for it.

At this point, you should be able to reboot with the same options that you had before installing Windows 95.

Alternative recovery recipe: Again, boot from your emergency boot floppy or installation boot floppy, but add the command-line argument **root=/dev/hda2** or **root=/dev/sda1** or whatever partition is your root partition. This should eventually look as if you just booted off the hard drive normally. Now, simply log in as root and run the **lilo** command. You should now have the same booting options you had before you installed Windows 95.

Recipe to avoid future disaster with LILO: In your /etc/lilo.conf file, change **boot=/dev/hda** or **boot=/dev/sda** to point to the primary partition which holds your boot images. This is important on large hard drives; you may have partitions which use disk space that is not part of the first gigabyte on the disk, and that, as you probably know, is inaccessible to the BIOS which starts the bootstrap process. With your **boot=/dev/hda3** or **boot=/dev/sda4** statement in place in /etc/lilo.conf, run the lilo command. This will install the LILO boot sector on the partition named in the **boot=** statement.

Now, use the fdisk program to make the Linux partition on which you just installed the boot sector the active partition. You can do this either with the Linux fdisk program or with the DOS fdisk program.

Now, the next time you have to re-install Windows 95 because your .INI files are hopelessly messed up, or Windows 95 refuses to run for no reason, you will be able to boot Linux just by running the DOS/Windows fdisk program and making the Linux partition the active partition. Reboot, and LILO will be working again.

LOADLIN

If you are willing to boot DOS in order to boot Linux, you can use the LOADLIN program. In some cases, this is actually the best way to boot Linux. In particular, some sound cards will work under Linux only if they are first initialized under DOS.

Some people who were using LOADLIN to start up Linux from the DOS prompt have discovered that after installing Windows 95, they can't bring up a DOS command prompt window and boot Linux from there—and it is no fun to reboot into DOS in order to finally get into Linux. Perhaps you are one of those people.

As you have discovered, LOADLIN has some limitations. For example, you can't use it to boot Linux while you are running Windows. Even if you aren't running Windows, if you are using an extended memory manager, it must support VCPI in order for LOADLIN to work. However, these constraints don't cause problems if you run it from a CONFIG.SYS menu item. If menu support hasn't been added, your entire CONFIG.SYS file might look something like this:

```
DEVICE=C:\DOS\HIMEM.SYS
DEVICE=C:\DOS\EMM386.EXE
FILES=40
DOS=HIGH,UMB
```

Let's call that your **DOS** section. You will also need a **LINUX** section, and you will need to be able to choose between them. In order to cause DOS to allow you to choose between them while booting, you will need a **MENU** section. The result looks like this:

```
[MENU]
MENUITEM=DOS, Boot DOS
MENUITEM=LINUX, Boot Linux
[DOS]
DEVICE=C:\DOS\HIMEM.SYS
DEVICE=C:\DOS\EMM386.EXE
FILES=40
DOS=HIGH,UMB
[LINUX]
REM Here is where you would load a driver for
REM a sound card that is not completely
REM supported by Linux.
SHELL=c:\LOADLIN\LOADLIN.exe @c:\LOADLIN\params
```

The **@c:\LOADLIN\params** means that the boot arguments for the kernel are kept in the file c:\LOADLIN\params. This file might look like:

```
root=/dev/hda2
ro
```

The documentation that accompanies LOADLIN explains this in much more detail, but you are likely to find this explanation sufficient to start using LOADLIN under most circumstances.

Many distributions include a copy of LOADLIN. You can also ftp a copy of LOADLIN from tsx-11.mit.edu in the directory /pub/linux/dos_utils/ in the file LOADLIN15.tar.gz.

Michael K. Johnson is the editor of *Linux Journal* and has to boot Windows 95 in order to do OCR (Optical Character Recognition) to convert paper books into on-line ones. He entertains hopes that someday soon, he will no longer have the experience necessary to write an article like this...

Archive Index Issue Table of Contents

Advanced search

Basic fvwm Configuration

John M. Fisk

Issue #27, July 1996

This article will attempt to introduce you to one of the most versatile and popular X-Windows managers: fvwm (which, I've been told, originally stood for “Frugal Virtual Window Manager”).

This article is primarily intended for, and dedicated to, all the novices and newcomers who have joined the worldwide community of Linux users. Welcome aboard! This article will attempt to introduce you to one of the most versatile and popular X-Windows managers: **fvwm** (which, I've been told, originally stood for “Frugal Virtual Window Manager”). Its well-deserved popularity is based, among other things, on its relatively parsimonious memory consumption, an extensively customizable 3 Motif-ish appearance, a virtual desktop, and the ability to extend functionality through the use of modules.

Before we begin, let me state a couple of presuppositions:

1. You've installed X-Windows and the fvwm window manager and have them working, and

2. You're willing to tinker a bit.

If this is you, keep reading!

First, some background: the concept of a window manager is, in essence, a rather simple one. X-Windows itself oversees certain rudimentary tasks such as managing the display hardware (monitor, keyboard, and mouse), handling mouse and keyboard events, and creating the windows which appear on the display. Just exactly *how* windows appear and behave is left up to the *window manager*. Window managers, such as fvwm, control how you interact with programs by providing decorative window frames, window controls, menus, virtual desktops, and so forth. Change to a different window manager and you can create a completely different look and feel even though the programs which run under them are identical.

fvwm was developed by Robert Nation as a derivative of one of the original MIT window managers, **twm** (the *Tabbed Window Manager,* which started life as *Tom's Window Manager* in honor of its author, Tom LaStrange). fvwm is currently maintained by Chuck Hines. The latest official release is fvwm-1.24r, although a beta version—pre2.0pl40 (as of mid-December 1995)--is currently available as well. These can be found at: ftp://sunsite.unc.edu/pub/Linux/X11/window-managers/ ftp://ftp.hpc.uh.edu/pub/fvwm/ ftp://ftp.x.org/contrib/

In addition, there are a number of excellent WWW pages dedicated to fvwm which provide a wealth of information, screen shots, and sample .fvwmrc configuration files. These include:

T.J. Kelly's **The fvwm Home Page**: www.cs.hmc.edu/~tkelly/docs/proj/fvwm.html

Todd Postma's **fvwm Info**: http://xenon.chem.uidaho.edu/~fvwm/

Jens Frank's **fvwm Documentation**: namu19.gwdg.de/fvwm/fvwm.html

Erik Kahler's **fvwm Home Page**: mars.superlink.net/user/ekahler/fvwm.html

At this point it's probably worth mentioning that if you're serious about customizing fvwm, it's a good idea to get the sources even if you don't intend to compile your own version. Why? Some Linux distributions do not include full documentation for every program. Getting the sources for fvwm ensures that you'll have the complete documentation, including manual pages for the various fvwm modules. Also, if you do choose to compile fvwm yourself, you can determine which features you wish to include.

So Let's Begin!

Some of the basics we'll try to cover include:

1. Managing configuration files

2. Starting programs at load up

3. Adding items to the fvwm popup menus

4. Color customization

Keep in mind that the information presented here is intended to be used as a primer. I'd recommend skimming through the manual pages for fvwm in addition to reading this article. The definitive source of information is the manual page and documentation that comes with the program source distribution.

Managing Configuration Files

The first thing that you'll need to know about fvwm is where the configuration file is found. When fvwm starts, it first looks for .fvwmrc in your home directory. For example, if your username is johndoe, fvwm looks for the file /home/johndoe/.fvwmrc. If this file is absent it then looks for the system-wide configuration file /usr/lib/X11/fvwm/system.fvwmrc. If neither of these files are present, it simply exits.

One of the first things you'll want to do is make a copy of the default system-wide configuration file and put it in your home directory:

```
  $ cp /usr/lib/X11/fvwm/system.fvwmrc ~/.fvwmrc
```

Be sure to copy this to ~/.fvwmrc and **not** ~/system.fvwmrc. Creating a copy of the file in your home directory is a good idea for a couple reasons:

1. If your .fvwmrc file gets corrupted, fvwm will fall back to using the system-wide configuration file.

2. If ~/.fvwmrc inadvertently gets deleted, you can use the system.fvwmrc file to re-create it.

3. On a multi-user system, it allows each user to customize her/his own .fvwmrc file.

The other issue to think about at the outset is sequential customizations. Chances are, over the course of several weeks or months, you'll make numerous small and large changes to the config file as you get fvwm customized to work the way you want it. What happens when you accidently overwrite or delete system.fvwmrc or your .fvwmrc? Serious bummer.

Managing system configuration files is important for anyone who is running, and likely administering, his own Linux system. It is wise to have a well-thought-out plan for how changes are implemented and changes tracked. Never do what you cannot undo. fvwm uses a single configuration file, making this a fairly simple task. Here are a few suggestions.

Before making any changes in a default configuration file, make a backup of it and give it a distinctive suffix, such as .dist. For example, before you edit the system.fvwmrc file for the first time, you'd make a copy of it by:

```
   # cp system.fvwmrc system.fvwmrc.dist
```

The .dist suffix alerts you to the fact that this is an original file from the distribution. To further safeguard it, you should copy this to a directory owned by root (the superuser) and set the permissions on that directory to read and execute only, except for root. To do this, you could, for example, make a directory in your /etc or /usr/local directory called backups/ and then copy all default config files to this directory.

To do this, log in as root and enter:

```
   # mkdir /etc/backups
   # chmod 755 /etc/backups
```

Setting the *directory* permissions, as root, to 755 allows root full read, write, and execute permissions, while preventing write permissions for everyone else. With read and execute permissions, users may change to the directory and do a file listing but cannot (since they don't have write permission to that directory) delete a file.

After you've created a backup directory and set the appropriate permissions you should, as root, make backup copies of important configuration files with the .dist suffix. You can also set the permissions for each *file* to read-only by:

```
   # chmod 744 /etc/backups/system.fvwmrc.dist
```

This limits permissions on the file to read only for all users except root. Making backups of default configuration files is helpful only if it actually *works*. Obviously, you'll want to make a backup copy of a working configuration file. This, however, still doesn't address the question of what to do about keeping track of each of your changes. Here are a few suggestions.

1. After you have modified a configuration file, and ensured that it works correctly, make a copy of it and number each file consecutively such as:

```
   fvwmrc.01
   fvwmrc.02
   fvwmrc.03 ...
```

Comments can be added by starting a line with the **#** (hash) character and should probably include the date and what modifications were made. If space is a problem, you can compress all the files you are not currently using.

2. Use **RCS**, the “Revision Control System”, to keep track of your changes. Before you make any changes to the file, run the RCS **check-in** program:

```
   $ ci -l .fvwmrc
```

It will ask you for a description of the file; type:

```
fvwm configuration file
.
```

The `.` on a line by itself finishes the description.

Then, every time you make a change to the file, run the same command. It will ask you for a description of the change you just made; type something like:

```
Added Calculator, xjed, and Emacs to the
"Application" pop-up menu.
.
```

so that you can find this change later.

RCS keeps track of all the changes you have made to .fvwmrc in a file called “.fvwmrc,v”. Instead of storing a complete copy of each new version of the file, it stores only the changes you have made, plus the latest version of the file.

How do you find the change later? The command line

```
$ rlog .fvwmrc | less
```

will give you a history of all the changes you have made. Each change has a **revision number**, starting at 1.1.

If you want to compare the current copy of the file with the version you previously checked in, use this command:

```
$ rcsdiff -u .fvwmrc | less
```

You can retrieve any version of the file you want (overwriting the current version) with

```
$ co -rrevision .fvwmrc
```

If you want more information on how to use RCS, it is available with:

```
$ man rcsintro
```

which provides a good overview of the concepts and commands you'll need to know. [*Linux Journal* also published an overview of RCS in issue 10, page 36—ED]

Using the RCS system is only slightly more complex a method of version control than simply making copies of every modified file. However, the basics are easily mastered and can be used for any file—programming project, article, term paper, etc.—that is undergoing sequential revision.

Getting Things Going

Now that you've copied the .fvwmrc file to your home directory and decided how you'll track your changes to it, you're ready to start tinkering! One of the first things you might be interested in is automatically launching programs when fvwm starts. You may want to start an xterm or two, a clock or calendar, xbiff (to warn you when mail arrives), and so forth. fvwm allows you to define an **InitFunction** within .fvwmrc that handles program launching at initialization. Before discussing this, however, we need to briefly mention xinit.

xinit is the program responsible for starting the X Window System and, like fvwm, it uses an .xinitrc file. A user may create a personalized copy of this file in her or his home directory as ~/.xinitrc, or simply use the system wide default located in /usr/lib/X11/xinit/xinitrc. To see how xinit launches programs at startup, let's look at the default xinitrc file that came with Slackware 2.2.0:

```
# start some nice programs
xsetroot -solid SteelBlue &
fvwm
```

In this simple example, xsetroot sets the color of the root window to SteelBlue. Once it is done, fvwm is started. A slightly more complicated example is given in the manual page for xinit:

```
xrdb -load $HOME/.Xresources
xsetroot -solid gray &
xclock -g 50x50-0+0 -bw 0 &
xload -g 50x50-50+0 -bw 0 &
xterm -g 80x24+0+0 &
xterm -g 80x24+0-0 &
twm
```

In this example, several programs are launched before twm is started. The important thing to point out is that the stanzas which launch programs, such as xclock and the xterm, need to end with an ampersand (**&**) so as to have them running in the background. It is also important that the window manager be started last and that it runs in the foreground (i.e., do not add an ampersand at the end of the line).

fvwm provides a similar method for launching programs at startup using the **InitFunction**. To see how, let's look at a sample entry in .fvwmrc:

```
Function "InitFunction"
# Module  "I"  FvwmBanner
# Get the fvwm GoodStuff button bar running
  Module  "I"  GoodStuff
# Module  "I"  FvwmPager 0 3
# Then, the analog clock - Swallowed by GoodStuff
  Exec  "I"  exec xclock -g 140x160-145+138 &
```

```
    Wait  "I"  xclock
  # Start xmailbox - swallowed by GoodStuff
    Exec  "I"  exec xmailbox &
    Wait  "I"  xmailbox
  # Fire up xload - also swallowed by GoodStuff
    Exec  "I"  exec xload -geometry 80x80+100+100 &
    Wait  "I"  xload
  # Now, start up xcalendar
    Exec  "I"  exec xcalendar -g 345x382+793+372 &
    Wait  "I"  xcalendar
  # Finally, fire up a full screen xterm
    Exec  "I"  exec xterm -sb -j -ls -g 84x48+4+4 &
    Wait  "I"  xterm
  EndFunction
```

Since it's easiest to explain by example, let's see what you'd need to do to add a second xterm to the startup. To begin with, **InitFunction** generally uses two-line stanzas for launching programs, and these take the form:

```
    Exec  "I"  exec program_name -options &
    Wait  "I"  program_title
```

For example, the stanzas which start xcalendar look like:

```
  # Now, start up xcalendar
    Exec  "I"  exec xcalendar -g 345x382+793+372 &
    Wait  "I"  xcalendar
```

Lines which begin with the hash (**#**) character are treated as comments and ignored by fvwm. The first line begins with the reserved word **Exec** and has four arguments:

1. An **"I"** (in double quotes).

2. The word **exec**.

3. The command used to start the program.

4. Any command line options followed by an ampersand (**&**).

The next line begins with the word **Wait** which causes an fvwm function to pause while a window with the name given on the command line is drawn to the display. In this case, the **InitFunction** pauses while a window with the title **xcalendar** is drawn to the display. Once the window has been drawn, fvwm continues.

Be aware that this is a bit of a fib, but a useful one for the moment. The **Wait** function is primarily used when programs are launched on more than one desktop—something we won't touch on for the moment. Assuming you use a single desktop, as in this present example, the **Wait** stanza may be safely omitted.

Going back to our assignment to start another xterm, we could launch a second xterm by adding the following:

```
     Exec   "I"   exec xterm &
     Wait   "I"   xterm
```

which would be sufficient to get a second xterm going.

If you added this entry and started fvwm, you'd find that when fvwm got to the point in the initialization process where the second xterm was started, it would draw the outline of the xterm window and wait for you to position it before proceeding. That's not very convenient if you have to do this every time fvwm starts. The way around this is to add a command line option specifying the xterm's geometry, which allows you to control where to place the window from the command line.

A Word About Geometry

A geometry entry for an application might look something like this:

```
    -geometry 420x360+5+20
```

If this looks a bit cryptic, don't worry, it's actually pretty simple. Converting this statement into plain English yields:

“The application window is 420 pixels wide by 320 pixels high, with its left border 5 pixels from the display's left edge and its top border 20 pixels from the display's top edge.”

Pretty simple, eh? There are actually only a couple of rules to keep in mind when specifying a window's geometry. First, dimensions are generally in terms of pixels, although there are times, notably with xterms and some text editors, in which the width and height dimensions will be in terms of characters. You'll notice in the example .xinitrc file above that the width and height dimensions for the xterm were given as 80x24, or 80 characters wide by 24 columns high. If you bear in mind that your entire display screen is probably 640x480 or 800x600 or 1024x768 pixels, depending on your resolution, you can get a feel for how much of the screen is taken up by an application window that is, say, 400x300 pixels.

The second set of numbers specify the horizontal (x-offset) and vertical (y-offset) distances from the edge of the display screen. Again, this is pretty straightforward: think of the screen in terms of graphing paper in which the upper left hand corner is 0,0 and the values increase as you move from left to right and from top to bottom.

If your screen were 640x480 pixels, your top left corner would be considered 0,0; the bottom left corner would be 0,480 (remember that the vertical position increases as you move from top to bottom); the top right corner would be 640,0; and the bottom right corner would be 640,480. The other thing to keep in mind is that horizontal and vertical positions are generally (but not always, as we'll see in a minute) specified in terms of the left and top sides of the application window.

For example, suppose that you wanted to put an xterm window in the upper left hand side of the screen. You decide that you want it 10 pixels from the left hand side of the display and 50 pixels from the top. You also want the window to be 400 pixels wide by 320 pixels high. Simple enough. You'd use the following geometry option to accomplish this:

```
   -geometry 400x320+10+50
```

Notice the general form this takes:

```
   -geometry WIDTHxHEIGHT+horizPOS+verticalPOS
```

Using a plus **+** sign before the pixel value indicates the position of the window with respect to its *left* hand or *top* edges. However, using a minus **-** sign specifies the opposite meaning: the horizontal position is the distance in pixels between the application window's *right* hand side and the *right* side of the display, and the vertical position is the distance in pixels between the application window's *bottom* edge and the *bottom* edge of the display.

If this seems a bit confusing try playing with it a bit. Start up fvwm and in an xterm enter the following commands:

```
   xterm -g +5+5 &
   xterm -g -5-5 &
   xterm -g -5+5 &
   xterm -g +5-5 &
```

Try these out and see where the xterm gets put. Note that you can generally abbreviate **-geometry** to a simple **-g**.

We've wandered a bit from our discussion about launching programs at startup. In practical terms, figuring out the correct geometry for all of the applications you want to have started is pretty easy. The first step is to get pencil and paper ready because you'll want to jot some notes.

Customizing the start-up desktop usually begins by starting all of the applications that you want present when fvwm begins. Try out various command line options to get the look and feel that you want. Reading a program's manual page often helps you determine what options are available at run time. Once you get an application running, you can generally resize it by clicking the mouse on one of the “L” shaped window corners and dragging it to a larger or smaller size. Clicking and dragging on the titlebar or side borders lets you position the window.

Once you have everything started, positioned, and sized the way you want it, jot down each application and the command line options, if any, that you used. To get each window's geometry, we'll use a great little program called **xwininfo**.

Start it from an xterm by entering:

```
  $ xwininfo
```

at the command prompt. Notice that you don't use an ampersand for this command. Your mouse cursor will change to a cross-hair and the following instructions will be displayed:

```
  xwininfo: Please select the window about which you
            would like information by clicking the
            mouse in that window.
```

Clicking on an application window produces the output like the following:

```
  xwininfo: Window id: 0x2c00007 "ez ~/fvwm_LJ.ez*"
    Absolute upper-left X:  92
    Absolute upper-left Y:  28
    Relative upper-left X:  0
    Relative upper-left Y:  0
    Width: 528
    Height: 724
    Depth: 8
    Visual Class: PseudoColor
    Border width: 0
    Class: InputOutput
    Colormap: 0x21 (installed)
    Bit Gravity State: ForgetGravity
    Window Gravity State: NorthWestGravity
    Backing Store State: WhenMapped
    Save Under State: no
    Map State: IsViewable
    Override Redirect State: no
    Corners:  +92+28  -532+28  -532-148  +92-148
    -geometry 528x724+85+0
```

In this instance, I clicked on the EZ editor's window, which produced a veritable cornucopia of information. Specifically, the geometry setting that you were looking for is in the last line. Do this for all the applications that you want started, and your work is pretty much done. Find the section in .fvwmrc that defines the **InitFunction**, add or modify the entries so as to start the applications that you want—don't forget to put that ampersand at the end of each **Exec** line!—and you should be all set. Once you've gotten things the way you want them, don't forget to make a backup of your newly modified .fvwmrc file.

One more thing before we leave the subject of launching programs at start up. fvwm comes with a number of *modules*, which are separate programs which must be spawned by fvwm—you can't start these from a command line. There are a number of modules which can generally be found in the /usr/lib/X11/ fvwm directory. A couple of the more common modules to launch at start up include FvwmBanner, which places a decorative banner across the root window; FvwmPager, which serves as a virtual desktop manager when you have multiple desktops going; and the GoodStuff button bar. The entry in **InitFunction** to start an fvwm module is a bit different than a regular application in that it is simpler:

```
    Module  "I"  GoodStuff
    Module  "I"  fvwmPager 0 3
```

You'll notice that there's no **Exec** or **Wait** statement needed. Simply use the reserved word **Module**, followed by **"I"** and then the name of the module to launch with any options.

That wasn't too bad, was it? This should give you the basics that let you customize your startup desktop. Next month, I'll cover launching programs once fvwm has started—and more!

John Fisk (fiskjm@ctrvax.vanderbilt.edu) After three years as a General Surgery resident and Research Fellow at the Vanderbilt University Medical Center, he decided to “hang up the stethoscope” and pursue a career in Medical Information Management. He's currently a full-time student at the Middle Tennessee State University and hopes to complete a graduate degree in Computer Science before entering a Medical Informatics Fellowship. In his dwindling free time he and his wife Faith enjoy hiking and camping in Tennessee's beautiful Great Smoky Mountains. An avid Linux fan since his first Slackware 2.0.0 installation a year and a half ago.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Introducing HyperNews

David Alan Black

Issue #27, July 1996

When you visit a HyperNews article and view a response, you can (depending on local access settings) add your own response to it, start a new thread, or navigate among the threads and responses in various ways.

For all the excitement over the Web—some justified, some exaggerated—a case could still be made for pinning the medal on Usenet as the most important and really original Internet contribution to communication. Fortunately, we have never had to choose between them—and now, with the availability of Daniel LaLiberte's HyperNews package, we can have the best of the functionality of both in one place.

A HyperNews “base article” is a WWW page, with a newsgroup-like segment at the end (see Figure 1--in fact, look at it more than once as you read). The HyperNews segment is organized by thread, with cascaded response subject lines. The responses, moreover, may be entered as plain text, “smart text” (does some formatting for you), a URL, or HTML. Thus the body of any response can incorporate another web page, operate as a mini-web page of its own, or simply consist of a Usenet-like typed statement.

When you visit a HyperNews article and view a response, you can (depending on local access settings) add your own response to it, start a new thread, or navigate among the threads and responses in various ways. Any response you add can—in fact, must—be previewed before it is finally posted, so you have a chance to see what your response will look like and, if you included hyperlinks, whether you got them right.

Retrieving a HyperNews base article is just like retrieving any other web page. Initial access generally goes through the “get” script, so a typical URL looks like union.ncsa.uiuc.edu/HyperNews/get/hypernews.html.

This sample URL actually points to the HyperNews home page, which is not only indispensable if you are interested in running the package, but also provides a pretty spectacular example of HyperNews in action. As you will see if you visit, it is possible to have any number of base articles at a given site—after all, they are web pages. This home page also offers a glimpse of the range of roles that a HyperNews extension can play in the life of a web page. The response section can shoulder most of the substance of the page (as in the “Bugs” article, where much of the information sought is likely to reside in the response section); or function as a relatively low-key guestbook-style scratchpad; or anything in between.

Anatomy of a Base Article

What you see when you visit a HyperNews article is basically a web page with a response outline—possibly a very long one—at the bottom. Behind the scenes, the HyperNews base article is a virtual home page, constructed on demand by the suite of Perl scripts which make up the HyperNews distribution. It consists of the following four sections, each of which is configurable in various ways, shown in Figure 2.

The separation of the base article into these several logical components gives you quite a bit of control over its look and feel. Each base article has an associated .html.urc file in which header, footer, and body URLs are specified. This makes for relatively easy maintenance and testing—perhaps no easier than any other non-trivial web installation, but it's nice that the pieces are separate and that the scripts do the bulk of the work in making sense of them.

The body, in most cases, contains whatever the page is really “about”. There's no point giving examples here—it's a web page, with everything that implies. During the process of creating a base article, you can enter the body by hand in HTML, plain text, or “smart text”; or you can specify a URL. If you enter text or HTML, HyperNews creates a name-body.html file for you, which you can later edit. If you specify a single URL for the body, your base article will consist of a retrieval of that URL, positioned among the other elements of the HyperNews page as sketched out in Figure 2.

The header and footer can be specified in the installation-wide configuration file hnrc, and/or separately for one or more base articles in the relevant .html.urc file(s). Installation-wide header and footer specifications in hnrc may consist of text, URL, or HTML. (I'd advise against putting full-blown HTML in the header specification, though—it's unlikely to be necessary, and it doesn't always mesh with the icons it ends up sharing space with.) Per-article header and footer specifications consist of URLs whose sources will be retrieved and placed, respectively, above the body and below the responses. The header and/or footer can also be left empty.

Both header and footer are propagated to the responses; that is, every response to an article (and responses to responses, etc.) includes and displays their base article's header and footer. This means that you should probably keep the size of the header and footer down, and that they should be relevant to every response, or at least unobtrusive (i.e., this is probably not the right place for site-wide greeting messages).

Access and Security

Disclaimer: I know enough to keep my systems as secure as I feel they need to be, but I am not a security guru. If you install HyperNews, you should familiarize yourself with all the available information on security, and make sure that you are satisfied with the robustness of your particular setup.

As a HyperNews administrator, you have control over who gets to do what. The “who” here has three possible values: administrator, member, and anyone. The “what” includes creating new base articles; reading articles and responses; and adding, deleting, and/or moving responses. You can pretty much mix and match restriction among the access levels and the available activities. For instance, you might want to grant unrestricted read access to your article and responses, but limit response-adding to members.

In all likelihood, you won't want anyone other than administrators (that is, you) creating base articles—otherwise you might wake up one morning to find that your machine has become an outpost of the Barry Manilow fan club. When it comes to adding responses, however, you might want to adopt a more open policy. The one thing about completely open response-adding that worries me is that it's possible to sign with someone else's name and e-mail address. That person will get a confirmation notice by e-mail (if that option is configured, which it is by default); but that may be cold comfort if several hundred web surfers have already read an illegitimately attributed posting. Requiring membership, on the other hand, might be a deterrent to posting, at least for the casual visitor, since it involves taking the time to fill out a form.

I'll describe some further aspects of access and security in the following section.

Installation and Setup

The HyperNews distributions is a suite of Perl 5 CGI scripts. I hit a bit of a snag here—I had to recompile Perl to get rid of some weird and disabling behavior having to do with reading configuration files. (It's still a mystery to me why it happened.) Perl 5.001m seems to be the Perl release of choice for running HyperNews.

I've installed HyperNews on two machines, using a pretty by-the-book directory structure. The major step is:

```
cd /usr/local/lib/httpd/cgi-bin
cp HyperNews1.version.tgz .
tar -xvzf HyperNews1.<version>.tgz
```

version will be something like **1.9B5.5**. This creates the subdirectory HyperNews1.*version*. The installation instructions recommend moving this directory to HyperNews:

```
mv HyperNews1.version HyperNews
```

The HyperNews directory tree must be writeable by your httpd server process during installation (though not thereafter, and in fact it's recommended that you remove write permissions once the package is installed). My server runs as user http, group www, so I did:

```
chown -R http:www HyperNews
```

The HyperNews tree is where the scripts and the principal configuration file live. HyperNews documents (base article and response .html files, and the smaller configuration files that go with them) go in a separate subdirectory, generally under /usr/local/lib/httpd/htdocs (or equivalent). Therefore, you will do something like:

```
cd ../htdoc
mkdir hn
chown http:www hn
```

The next step in installation is to run the setup script, which you do with your HTML browser:

```
netscape http://myhost/HyperNews/.scripts/setup-form.pl
```

You can also run a command-line version of the setup script, but you then have to edit the configuration file manually and run it again. It's handy once you know what you're doing, but not very installer-friendly the first time around.

The setup script asks you a number of questions. I won't cover all of them, but a few things you should be aware of are:

- Note that the default path is /usr/local/etc rather than /usr/local/lib, so if your setup is like mine, you'll want to change it.
- Make sure to push the “Rebuild Password and Group Files” button, unless you have some reason to want to create the password files by hand.
- The name you give for the administrator should be an actual user or valid mail alias.

When you submit the setup form, it creates (backing up if needed) the main configuration file, /usr/local/lib/httpd/HyperNews/hnrc. Once that file is in place, you can change it manually. Some manual changes will take effect immediately; others, specifically those concerning access and membership, are really directives which tell the setup script to create or delete certain access files and symbolic links. For those changes to take effect, you have to run the setup script again; otherwise, your configuration file and actual setup will not accord with each other. (Updating the setup after manually changing hnrc is a good time to run the quicker command-line version, setup.pl.)

If you look at the HyperNews directory after running setup-form.pl, you will see that there are a number of symbolic links to various scripts, as well as two new subdirectories: Admin and SECURED. The basic logic of the directory structure meshes with the levels of access:

- All scripts are kept in HyperNews/.scripts.
- Those that can be run by anyone have links in HyperNews.
- Those that can be run by members have links in SECURED.
- Those that can only be run by administrators have links in Admin.

If you change the who-gets-to-do-what settings in hnrc and run the commandline setup.pl script, the relevant links will be added or deleted.

The setup process also creates password and other access-related files. These files are the gatekeepers for the different access policies governing the script directories. This per-directory security is handled through the server; for example, .htaccess files for NCSA httpd. (And that brings us full circle to my security disclaimer!)

Customizing and Configuring

There's a fair amount of configuring you can do by tweaking the hnrc and/or the article-specific .html,urc files. More major changes involve modifying (polite word for hacking) the Perl scripts. I've done a bit of this. For instance, on one of the sites where I've installed HyperNews, its main use will be to add response functionality to the photographic portfolio of one of my colleagues. We decided to eliminate the icons, present by default, which indicate the response type—happy, angry, comment, question, etc. These icons can be informative, but they also play a large role in setting the tone of the page, and they won't be appropriate for every HyperNews article. I've also thinned out the article response form, since we're not planning to use membership and wanted to incorporate our own wording for the form's instructions.

As with all package-hacking, my assaults on the HyperNews code will make it harder to upgrade to new versions at that particular site. In fact, if I hadn't needed no-icon-ness right away, I might very well have hung tight and seen whether it gets incorporated into the package down the road. One of the great things about the HyperNews home page (see URL, above) is that many suggestions for change and improvement can be, and are, posted—and Daniel LaLiberte, the author of HyperNews, is extremely receptive and responsive. If you want to keep pace with HyperNews development and releases, you can subscribe to the “history” page at union.ncsa.uiuc.edu/HyperNews/get/history.html.

Hyper-Remarks

If a package like HyperNews could be described thoroughly in an article of this length, it probably wouldn't be worth writing an article about. There are plenty of features and possibilities, and a few problems, that I haven't covered. If HyperNews intrigues you, have a look at its home page, where you can read and post responses to many base articles, including a couple of test and guestbook-style ones. You'll find a lot of support from the community of users, and you may very well also find one or more uses for HyperNews in the context of your own web development.

David Alan Black (dblack@candle.superlink.net)

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Letters to the Editor

Michael K. Johnson

Issue #27, July 1996

Readers sound off.

Subscribing—With Interest

Hi! I would like to thank you for doing so interesting a journal. It's my second subscription to *LJ*, and it won't be the last. I found the XForms review and tutorial so interesting that I decided to test it and will, perhaps, even use it at work. I enjoy reading tutorials and explanations about the Linux kernel (thanks a lot to Michael K. Johnson and all the staff of *Linux Journal*).

You try to spread the Linux enthusiasm everywhere. It's a success. Thank you.

Eric Bouchut

Typos in The Devil's in the Details

Juergen Schmidt, an attentive reader, reported a few errors in the third Kernel Korner article about device drivers, co-authored by Georg van Zezchwitz and myself. The errors are my fault, due to the limited time I had to revise the article.

The code printed within the article comes from a real driver, and it is known to run, but sometimes, I forgot to substitute the name of a symbol while copying from the real driver to the article's text.

So, **Skel_Board** (the structure) should read as **Skel_Hw**; **hwp** (the pointer) is equivalent to **board** (replace either one with the other); in **skel_select**, **file** (the **struct file** pointer) should read **filp**.

I'm sorry for these inconsistencies, and I hope they didn't cause headaches to the readers.

—Alessandro Rubini rubini@foggy.systemi.it

Corrections to IP Firewall Article

I have a few corrections to my article *Building a Linux Firewall* in *LJ* #24, April 1996, page 49.

1) Figure 3 is a duplicate of Figure 2. This is my fault. I submitted it this way. Obviously, cut-and-paste from one xterm to another can either be your friend or your enemy. I must have copied from the wrong window. The correct contents for the figure are shown here:

```
# ipfw -n list b
Type   Proto From         To             Ports
deny   udp   anywhere     192.168.1.1/32 any -> any
deny   udp   anywhere     20.2.51.105/32 any -> any
accept udp   anywhere     20.2.51.105/32 domain -> any
accept udp   20.2.61.0/24 20.2.51.105/32 any -> snmp
deny   tcp   anywhere     20.2.51.105/32 any -> any
deny   tcp   anywhere     192.168.1.1/32 any -> any
```

Figure 3. New blocking rule for SNMP to only accept from 20.2.61.0.

I've squeezed that down. Please use a condensed courier font to make it fit, or somehow make it a wide inline figure.

2) Several of the ipfwadm commands on page 58 have an additional character within the command line. The character is a right angle bracket, and this could cause some undesirable side effects if typed in that way.

3) The sentence on page 53 “ipfw only supports the deny and accept policies, not reject.” should be corrected to, “ipfw only supports the deny and accept policies for its output. A rule set to reject will still show up as deny.”

—Chris Kostick cykostick@csc.com

Linux in Space

During the Space Shuttle mission STS-75, an astronaut was heard talking about the fact that Linux was installed on a computer on board the spacecraft. A few weeks later, the computer's function was disclosed. The software in use was X-based software developed under Digital Unix and ported to Linux so that it could be used on board the shuttle. Astronaut Ron Parise said in an e-mail message to fellow amateur radio operators:

Pat, et al.:

Linux was installed on one of the IBM Thinkpads that are usually flown on the shuttle. This was in support of the tether experiments. Since the ground-based applications to control those experiments ran on a DEC Alpha it was easy to just port them to a Linux system for on-board use.

73's, Ron WA4SIR

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Stop the Presses

Readers regularly request that we give updates on the progress being made porting Linux to other platforms. We are happy to comply; it's a wonderful opportunity to show off Linux's impressive progress...

by Michael K. Johnson

SPARC!

In the last few months, SPARCLinux has gone from starting to run a few binaries to being stable on a supported single-processor and multiple-processor SPARC machines. Not only that, but supported SPARC machines include a SPARC-based supercomputer, the Fujitsu AP1000+.

David Miller, at Rutgers University, started the port about a year ago. Within a few months, other developers joined the project, and now the team includes several talented Linux Hackers from around the world, such as Miguel de Icaza from Mexico, Peter Zaitcev from Rumania, and Paul Mackerras, David Walsh, David Sitsky, and Andrew Tridgell from Australia. In addition, Ross Technology, a manufacturer and vendor of SPARC hardware, sent high-end hyperSPARC processors to David in order to help the port along.

The SPARCLinux kernel is remarkably stable; David now requires that every kernel pass a “crashme” test (see **Crashme**, below) for about 24 hours before releasing the source code for it. The kernel also performs well, benchmarking better than both SunOS and Solaris in almost all tests that have been done so far. It also provides strong binary compatibility with SunOS, and binary compatibility with Solaris is in the works.

Unfortunately, a SPARCLinux distribution is not ready yet, and if you want to actually use SPARCLinux, it is like a trip into Linux history, gathering binaries from different sites to get enough programs to actually use it. Even when those binaries are collected, the resulting systems is not particularly stable or complete—not because the kernel itself is unstable, but because the user-level programs aren't yet up to snuff. And the bootstrapping process of getting a Linux system installed is not yet simple.

Fortunately, Red Hat Software is working with the SPARCLinux developers to create a Red Hat Commercial Linux for SPARC, and they make their ongoing work available from their web site (www.redhat.com) and their ftp site (ftp.redhat.com). As of this writing, about 70 RPM packages are available for SPARCLinux. While Red Hat works on providing a set of stable user-level programs, Miguel is working on providing a stable set of shared libraries for them to base those programs on.

SPARCLinux currently supports most SUN4c (SPARC 1, 1+, 2, IPC, IPX, SLC, ELC) and Sun4m (SPARC Classic, LX, 5, 10, 20) machines. HyperSPARC machines are supported in single-processor and multi-processor modes.

Others, including sun4d (SPARCCenter 1000/2000) and the old sun4 architecture aren't supported yet. Contact David if you can help with these or provide hardware, **especially sun4d machines** (donations and loans are both accepted...). The sun4u work is underway but not finished yet.

Crashme

The crashme program tries to do what its name implies: it tries to crash the computer on which it is run. It does this by creating "random" data and then trying to run that data as executable code. This causes it to do things that the designers of the operating system never thought about, and finds security holes and operating systems bugs that don't seem to show up reliably in any other way—except while running buggy programs. The difference is that crashme logs exactly what it does and creates its "random" data in exactly the same way each time it runs, making it possible for developers to duplicate—and therefor fix—the bugs that crashme finds.

Few commercial versions of Unix survive crashme for more than a few minutes. Linux, including SPARCLinux, survives running many instances of crashme at once for days on end.

Vger

As many of you know, most of the Linux mailing lists are maintained on a machine called vger.rutgers.edu. Vger is a SPARC currently running SunOS. David's personal goal is to run Linux on Vger, as this will provide another stress test besides crashme to ensure that SPARCLinux is very stable. Besides, it seems more appropriate to run the Linux mailing lists on a Linux machine.

Learning More

One of the mailing lists on vger is sparclinux@vger.rutgers.edu, which is where the SPARCLinux developers hang out. The list is currently fairly quiet, with only a few messages most days, but this may change by the time you read this article. In addition, there are a few invaluable URLs for more information on SPARCLinux, including:

- http://www.geog.ubc.ca/sparclinux.html
- http://www.redhat.com/sparc
- ftp://vger.rutgers.edu/pub/linux/SPARC/
- ftp://ftp/redhat.com/pub/devel/sparc/RedHat/RPMS
- http://cap.anu.edu.au/cap/projects/linux/
- http://amelia.experiment.db.erau.edu.sparclinux/

More Details!

David has promised to write a series of Kernel Korner articles detailing the technical issues encountered while doing the port. He gave an interesting talk at the NCSU Linux Expo 96 in Raleigh, North Carolina, in early April, and we expect even more interesting details from him in his articles. We can expect everything from technical challenges encountered in the porting process to interesting technical details about SPARC architecture.

The Expo

I've mentioned the Linux Expo—but it was about more than the SPARC port, not surprisingly. Alan Cox came over from Wales to give a talk on SMP (Symmetric Multi-Processing, where multiple processors in the same machine share memory address space) on the Intel platform. David's talk also covered SMP, but on the SPARC platform. There were other “nerd” talks as well, and one very good, but less “nerdy”, talk by Jon “maddog” Hall of Digital. His main point was that there are two ways vendors can use standards:

- To create volume in the marketplace
- As a weapon against other vendors

He urged the Linux community to stick together and avoid using standards as weapons, as has happened in the Unix community, to its detriment. As the senior leader of the Digital Unix marketing group, he knows what he is talking about.

Archive Index Issue Table of Contents

Advanced search

New Products

Marjorie Richardson

Issue #27, July 1996

BRU Personal Edition Available, Linux History and Internals Book and more.

BRU Personal Edition Available

Enhanced Software Technologies, Inc., has announced the release of BRU Personal Edition (BRU-PE), offering the same high data reliability as the commercial version of BRU with lower system overhead. BRU-PE is now shipping for Linux, BSD/OS and Free/BSD, SCO Unix and UnixWare, and SunSoft's Solaris x86. Price: $69.95.

Contact: Enhanced Software Technologies, Inc., 5016 S. Ash Avenue, Suite 109, Tempe, AZ. Phone: 1-602-820-0042. Fax: 1-602-491-0865. E-mail ted@estinc.com. URL: www.estinc.com/.

Linux History and Internals Book

Specialized Systems Consultants, Inc., (SSC) just published *Inside Linux: A Look at Operating System Development* by Randolph Bentson. *Inside Linux* looks at the history of operating systems, how they are used, and the details of one operating system—Linux. The contents interweave discussions of history, theory, and practice, allowing the reader to see what happens inside the system.

ISBN: 0-916151-89-1. Price: $22. Contact: Specialized Systems Consultants, PO Box 55549, Seattle, WA 98155-0549. Phone: 1-206-782-7733. Fax: 1-206-782-7191. E-mail: info@linuxjournal.com . URL: http://www.ssc.com/.

SPATCH Version 3.x Released

The Hyde Company has released version 3.x of its SPATCH Alphanumeric Paging Software. This version of SPATCH offers two new SPATCH options, SPATCH On Call Scheduling and SPATCH PagePage. SPATCH is software supports sending text messages to alphanumeric pagers from a user, an application program, the operating system or E-mail. SPATCH is supported on many Unix and Unix-like operating systems including SCO, AIX, HP/UX, SUN/OS, Solaris, Linux, Dec Unix, Irix, Dynix, System V, and DG/UX.

Contact: The Hyde Company, URL: www.spatch.com. Phone: 1-770-495-0718. E-mail: spatch@cy.com.

New Book on the AWK Language

Specialized Systems Consultants, Inc., (SSC) has published *Effective AWK Programming* by Arnold D. Robbins. *Effective AWK Programming* is a comprehensive user's guide for AWK, a programming language defined in the POSIX Command Language and Utilities standard. Information is presented in a tutorial format with practical tips.

ISBN: 0-916151-88-3 Price: $27. Contact: Specialized Systems Consultants, PO Box 55549, Seattle, WA 98155-0549. Phone: 1-206-782-7733. Fax: 1-206-782-7191. E-mail: info@linuxjournal.com . URL: www.ssc.com/.

Archive Index Issue Table of Contents

Advanced search

Consultants Directory

> This is a collection of all the consultant listings printed in *LJ* 1996. For listings which changed during that period, we used the version most recently printed. The contact information is left as it was printed, and may be out of date.

ACAY Network Computing Pty Ltd
Australian-based consulting firm specializing in: Turnkey Internet solutions, firewall configuration and administration, Internet connectivity, installation and support for CISCO routers and Linux.

Address:
Suite 4/77 Albert Avenue, Chatswood, NSW, 2067, Australia
+61-2-411-7340, FAX: +61-2-411-7325
sales@acay.com.au
http://www.acay.com.au

Aegis Information Systems, Inc.
Specializing in: System Integration, Installation, Administration, Programming, and Networking on multiple Operating System platforms.

Address:
PO Box 730, Hicksville, New York 11802-0730
800-AEGIS-00, FAX: 800-AIS-1216
info@aegisinfosys.com
http://www.aegisinfosys.com/

American Group Workflow Automation
Certified Microsoft Professional, LanServer, Netware and UnixWare Engineer on staff. Caldera Business Partner, firewalls, pre-configured systems, world-wide travel and/or consulting. MS-Windows with Linux.

Address:
West Coast: PO Box 77551, Seattle, WA 98177-0551
206-363-0459
East Coast: 3422 Old Capitol Trail, Suite 1068, Wilmington, DE 19808-6192
302-996-3204
amergrp@amer-grp.com
http://www.amer-grp.com

Bitbybit Information Systems
Development, consulting, installation, scheduling systems, database interoperability.

Address:
Radex Complex, Kluyverweg 2A, 2629 HT Delft, The Netherlands
+31-(0)-15-2682569, FAX: +31-(0)-15-2682530
info@bitbybit-is.nl

Celestial Systems Design
General Unix consulting, Internet connectivity, Linux, and Caldera Network Desktop sales, installation and support.

Address:
60 Pine Ave W #407, Montréal, Quebec, Canada H2W 1R2
514-282-1218, FAX 514-282-1218
cdsi@consultan.com

CIBER*NET
General Unix/Linux consulting, network connectivity, support, porting and web development.

Address:
Derqui 47, 5501 Godoy Cruz, Mendoza, Argentina
22-2492
afernand@planet.losandes.com.ar

Cosmos Engineering
Linux consulting, installation and system administration. Internet connectivity and WWW programming. Netware and Windows NT integration.

Address:
213-930-2540, FAX: 213-930-1393
76244.2406@compuserv.com

Ian T. Zimmerman
Linux consulting.

Address:
PO Box 13445, Berkeley, CA 94712
510-528-0800-x19
itz@rahul.net

InfoMagic, Inc.
Technical Support; Installation & Setup; Network Configuration; Remote System Administration; Internet Connectivity.

Address:
PO Box 30370, Flagstaff, AZ 86003-0370

602-526-9852, FAX: 602-526-9573
support@infomagic.com

Insync Design
Software engineering in C/C++, project management, scientific programming, virtual teamwork.

Address:
10131 S East Torch Lake Dr, Alden MI 49612
616-331-6688, FAX: 616-331-6608
insync@ix.netcom.com

Internet Systems and Services, Inc.
Linux/Unix large system integration & design, TCP/IP network management, global routing & Internet information services.

Address:
Washington, DC-NY area,
703-222-4243
bass@silkroad.com
http://www.silkroad.com/

Kimbrell Consulting
Product/Project Manager specializing in Unix/Linux/SunOS/Solaris/AIX/HPUX installation, management, porting/software development including: graphics adaptor device drivers, web server configuration, web page development.

Address:
321 Regatta Ct, Austin, TX 78734
kimbrell@bga.com

Linux Consulting / Lu & Lu
Linux installation, administration, programming, and networking with IBM RS/6000, HP-UX, SunOS, and Linux.

Address:
Houston, TX and Baltimore, MD
713-466-3696, FAX: 713-466-3654
fanlu@informix.com
plu@condor.cs.jhu.edu

Linux Consulting / Scott Barker
Linux installation, system administration, network administration, internet connectivity and technical support.

Address:
Calgary, AB, Canada
403-285-0696, 403-285-1399
sbarker@galileo.cuug.ab.ca

LOD Communications, Inc
Linux, SunOS, Solaris technical support/troubleshooting. System installation, configuration. Internet consulting: installation, configuration for networking hardware/software. WWW server, virtual domain configuration. Unix Security consulting.

Address:
1095 Ocala Road, Tallahassee, FL 32304
800-446-7420
support@lod.com
http://www.lod.com/

Media Consultores
Linux Intranet and Internet solutions, including Web page design and database integration.

Address:
Rua Jose Regio 176-Mindelo, 4480 Cila do Conde, Portugal
351-52-671-591, FAX: 351-52-672-431
http://www.clubenet.com/media/index.html/

Perlin & Associates
General Unix consulting, Internet connectivity, Linux installation, support, porting.

Address:
1902 N 44th St, Seattle, WA 98103
206-634-0186
davep@nanosoft.com

R.J. Matter & Associates
Barcode printing solutions for Linux/UNIX. Royalty-free C source code and binaries for Epson and HP Series II compatible printers.

Address:
PO Box 9042, Highland, IN 46322-9042
219-845-5247
71021.2654@compuserve.com

RTX Services/William Wallace
Tcl/Tk GUI development, real-time, C/C++ software development.

Address:
101 Longmeadow Dr, Coppell, TX 75109
214-462-7237
rtxserv@metronet.com
http://www.metronet.com/~rtserv/

Spano Net Solutions
Network solutions including configuration, WWW, security, remote

system administration, upkeep, planning and general Unix consulting. Reasonable rates, high quality customer service. Free estimates.

Address:
846 E Walnut #268, Grapevine, TX 76051
817-421-4649
jeff@dfw.net

Systems Enhancements Consulting
Free technical support on most Operating Systems; Linux installation; system administration, network administration, remote system administration, internet connectivity, web server configuration and integration solutions.

Address:
PO Box 298, 3128 Walton Blvd, Rochester Hills, MI 48309
810-373-7518, FAX: 818-617-9818
mlhendri@oakland.edu

tummy.com, ltd.
Linux consulting and software development.

Address:
Suite 807, 300 South 16th Street, Omaha NE 68102
402-344-4426, FAX: 402-341-7119
xvscan@tummy.com
http://www.tummy.com/

VirtuMall, Inc.
Full-service interactive and WWW Programming, Consulting, and Development firm. Develops high-end CGI Scripting, Graphic Design, and Interactive features for WWW sites of all needs.

Address:
930 Massachusetts Ave, Cambridge, MA 02139
800-862-5596, 617-497-8006, FAX: 617-492-0486
comments@virtumall.com

William F. Rousseau
Unix/Linux and TCP/IP network consulting, C/C++ programming, web pages, and CGI scripts.

Address:
San Francisco Bay Area
510-455-8008, FAX: 510-455-8008
rousseau@aimnet.com

Zei Software
Experienced senior project managers. Linux/Unix/Critical business software development; C, C++, Motif, Sybase, Internet connectivity.

Address:
2713 Route 23, Newfoundland, NJ 07435
201-208-8800, FAX: 201-208-1888
art@zei.com

Archive Index Issue Table of Contents

Advanced search

UseLinux Announcement

by Jon “maddog” Hall

USENIX is the Unix and advanced computing systems technical and professional association. Since 1975, the USENIX Association has brought together the community of engineers, system administrators, scientists, and technicians working on the cutting edge of the computing world.

Linux International is an international, not-for-profit, volunteer-run organization dedicated to promoting Linux.

So it is not surprising to find these two organizations have banded together to hold a joint **Linux Application Development and Deployment Conference**, which has been dubbed ”“**USELINUX**”“.

Recognizing that two of the main issues around Linux are the lack of commercial applications and the misunderstanding of the Linux market, these two organizations are presenting three additional areas of study to the USENIX conference scheduled for January 6-10 in Anaheim California:

- Technical sessions for developers of the Linux operating system
- Technical sessions for application developers and systems administrators
- Business sessions for people wishing to increase their market revenue by utilizing the Linux operating system

Yes, suits! Suits will be coming to a USENIX and Linux conference to learn how our favorite operating system can be used to increase their business, for it is in this way that Linux will move out of the closet and into its rightful place in the computer marketplace. However, the technical and business sessions will be well-defined, and no “techies” will have to be exposed to drivel about how to make money with Linux—unless they **want** to be exposed!
A technical program committee and a business program committee have been formed, with a call for papers and ideas submitted to USENIX. You can find the call for papers on the USENIX web site at http://www.usenix.org/opsys/index.html.

We urge you to support this effort with your input and by helping us publicize the conference to your favorite application vendors.