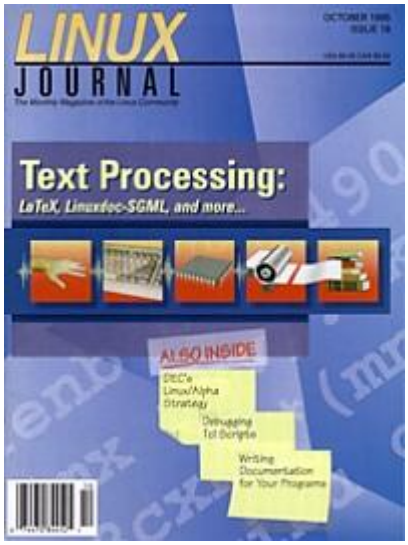
Linux Journal Issue #18/October 1995



Features

Flexible Formatting with Linuxdoc-SGML *by Christian Schwarz*
An introduction on how to produce multiple outputs from a single source file.
Using grep *by Eric Goebelbecker*
Moving from DOS? Discover the power of this Linux utility.
Writing man Pages Using groff *by Matt Welsh*
Learn how to document your programs just like real programmers do.
LaTex for the Timid *by Kim Johnson*
Don't be afraid. It's not as hard as you think to create beautiful output.

News & Articles

Linux on Alpha: A Strategic Choice *by Jon "maddog" Hall*
Debugging Tcl Scripts *by Stephen Uhler*
Indexing with Glimpse *by Michael K. Johnson*

Reviews

Book Review WWW Books *by Brian Rice*

Columns

Letters to the Editor
Stop the Presses *by Phil Hughes*

Flexible Formatting with Linuxdoc-SGML

Christian Schwarz

Issue #18, October 1995

Have your cake and eat it too with this simple but powerful text processing facility assembled by a well-known Linux guru.

As Linux becomes more and more popular, a lot of documentation is required, not only for newcomers, but for all users. Just think of all the FAQs, HOWTOs, manual pages, and books everyone needs for their daily work. Some people want to read these documents as plain ASCII text, while others want to read them over the World Wide Web or print them on their PostScript printer. It is possible to make an HTML version of an ASCII document for the Web and a nicely-formated PostScript version for people to print, but all the different formats have to be maintained separately. This is theoretically possible, but doesn't happen in real life.

We need a documentation system that can produce different formats from a single source. The Linux Documentation Project faced this exact dilemma when the HOWTO project was started, so Matt Welsh wrote the Linuxdoc-SGML package to solve it. With this package, all documentation is formatted in a similar way. But SGML is very flexible, so you can use the system to write many different kinds of documentation; as an example, the XFree86 project uses Linuxdoc-SGML for all of its documentation.

A Linuxdoc-SGML Example

```
<!doctype linuxdoc system>
<article>
<title>The Very Short Story
<author>A. Author
<date>1 Jan 1970
<p>
Once upon a time, they lived happily ever after.
<article>
```

As you can see here, the Linuxdoc-SGML syntax is very simple. Commands are written in angle brackets: **<command>**. When they apply to a block of the text

they appear in a pair surrounding that block, so **</article>** before the block is balanced by **</article>** after the block. There is also an abbreviation for the latter case if the block is short: **<tt/typewriter font/**.

The first line of the document specifies the document type. Here you will always specify **linuxdoc system**, since this refers to the main macros of the Linuxdoc-SGML package. Then you start your document with the **<article>** command and close it at the end with the corresponding "article off" command **</article>**. The article itself starts with the title, the author, and the date (which is optional). After that you can start writing the body text. The **<p>** command indicates the beginning of the first paragraph. You don't have to worry about spaces or line breaks when writing the text, since multiple spaces between words are ignored and line breaks are automatically inserted at the appropriate positions. To begin a new paragraph, insert a blank line, which is a "synonym" for **<p>**.

Running Linuxdoc-SGML

Linuxdoc-SGML is actually a collection of programs that work together to provide the final output. You need to know how to use each of them; several examples will help. The **format** program creates files designed for LaTeX, groff, or makeinfo, and is part of the process of creating HTML files, which is explained more fully below. The **-T** argument tells **format** which program it is writing files for.

There is one utility for running each of the formatting programs (groff, etc). Each has a name starting with "q", like "qtex".

To get a PostScript file via LaTeX, just type

```
format -T latex example.sgml > example.tex
qtex example
```

and Linuxdoc-SGML will create a LaTeX-format file, use LaTeX to process that file, then use **dvips** to turn that into the PostScript file example.ps. Note that you need to have LaTeX and dvips installed, along with Linuxdoc-SGML, for this to work.

If you prefer a DVI file, you may use a **-d** switch with **qtex**:

```
format -T latex example.sgml | qtex -d > example.dvi
```

The plain ASCII output is created with a similar procedure. Just run:

```
format -T nroff example.sgml | qroff > example.txt
```

To get texinfo output that can be read with the GNU info program, use:

```
format -T info example.sgml
```

This will create the necessary files in the current directory automatically. Of course, you need the GNU texinfo package installed on your system to make texinfo files.

The HTML output needs a little bit more care, since two compilation stages are necessary to get all cross references built. First, you have to have the **LINUXDOC** environment variable set up correctly; you will want to put a line such as:

```
export LINUXDOC=~/linuxdoc-sgml-1.2
```

in your bash startup file, or:

```
setenv LINUXDOC=~/linuxdoc-sgml-1.2
```

in your csh or tcsh startup file.

Once that is working, you have to run several commands to get finished HTML:

```
format -T html example.sgml | prehtml | \
   fixref > tmp.html
format -T html example.sgml | prehtml >> tmp.html
cat tmp.html | html2html example > example.html
rm tmp.html
```

It's a good idea to put these commands in a shell script since you will call these commands often. Here's a simple version you can use:

```
bin/bash
bin/bash
# sgml2html
[ -z -$1- ] && { echo -What file?-; exit 1 }
BASE=`basename $1 .sgml'
[ ! -f $BASE.sgml ] && { echo -No file $BASE.sgml-; exit 2 }
TMP=$$tmp.html
format -T html $1 | prehtml | fixref > $TMP
format -T html $1 | prehtml >> $TMP
cat $TMP | html2html $BASE > $BASE.html
rm $TMP
```

This script requires that your input file has the extension **.sgml**.

This script must be given the full file name, and it requires that the file have the extension **.sgml** to work correctly.

All of this is documented more completely in the excellent, short manual provided with Linuxdoc-SGML.

The Details

Now that we have explained the necessary “overhead”, we can start writing larger documents. Therefore, you will probably want to create sections and subsections, start the document with an abstract, and insert a table of contents. You can just add the following lines after the **date** line:

```
<abstract>
Here is the abstract.
<abstract>
<toc>
<sect>The First Section
<sect1>The First Subsection
```

Depending on the output format you want to create, the table of contents will have different styles: in a LaTeX document you get a standard table of contents, and in an HTML file you will get a list of cross references for the different sections and subsections.

As you can see from our example, the **<sect>** command creates a new section and the **<sect1>** command a new subsection. You can access five different levels simply by increasing the number in the command name. Note that you will still have to insert the **<p>** command to start the first paragraph after an sectioning command. For technical documentation it is often necessary to include “verbatim” text—text that is passed through without interpretation by the SGML parser. You can do this with the **<verb>** command:

```
<verb>This text is not interpreted <verb>
```

However, some very special characters do need some additional handling even when they appear inside a verbatim environment. For example, the sequence of an opening angle bracket and a slash (**</**) has to be written as **&ero;etago;**. The manual that comes with the Linuxdoc-SGML package includes a list of all special characters and the commands used to enter them as literal characters.

The Linuxdoc-SGML package supports three different font styles beside the normal font: boldface, italics, and typewriter. To select a font insert the command **bf**, **em**, or **tt**, respectively. To switch back to the default font just use the corresponding “slashed” command, as in the following example:

```
This <em>text</em> is written in italics!
```

An abbreviated version of each of these is sometimes useful:

```
This <em/text/ is written in italics!
```

Now let's take a look at different list types that are supported:

1. **itemize** for bulleted lists

2. **enum** for numbered lists, and
3. **descrip** for description lists, as demonstrated in this list.

You just start the list "environment" with the **<list>** command and close it with **</list>**. To produce a new item (either a bullet or a number) you can use the **<item>** command. In the description environment, you have to use the **<tag>** command with an argument that contains the "keyword", as in the following example:

```
<descrip>
<tag/First./ This is the first point.
<tag>Second.</tag> And this is the second.
</descrip>
```

If you want to create WWW pages with this package, you may want to create cross references to other WWW pages. This is done with the **<url>** command, as in the following example:

```
<url url=-http://gnus.com/pub/text.html- name=-Text
document->
```

This creates a link to the WWW page **text.html** on the specified host and displays **Text document** as the title of the link. When you create non-HTML documents, both the name and the URL are shown, so that the reader can still find the document.

Christian Schwarz studies mathematics in Munich and has worked with Linux for two years. He contributed the texinfo support for the Linuxdoc-SGML package. You may reach him at the address schwarz@monet.m.isar.de.

Archive Index Issue Table of Contents

Advanced search

Using grep

Eric Goebelbecker

Issue #18, October 1995

New Linux users unfamiliar with this standard Unix tool may not realize how useful it is. In this tutorial for the novice user, Eric demonstrates grep techniques.

When I first started working in systems integration, I was primarily a PC support person. I spent a lot of time installing and supporting Windows applications in various PC LAN configurations, running various versions (and vendors) of TCP/IP transports. Since then, I have successfully ditched DOS and moved on. Now, after working on various versions of Unix for a few years, I porting some of our networking and data manipulation libraries to other platforms and environments, such as the AS/400 minicomputer and the Macintosh. This ongoing experience has given me a chance to appreciate just how powerful the tools we take for granted with Linux really are.

Searching for a word (or any other value) in a group of files is a very common task. Whether it's searching for a function in a group of source code modules, trying to find a parameter in a set of configuration files, or simply looking for a misplaced e-mail message, text searching and matching operations are common in all environments.

Unfortunately, this common task doesn't have an easy solution on all platforms. On most, the best solution available is to use the search function in an editor. But when it comes to Linux (and other Unix descendants), you have many solutions. One of them is **grep**.

grep is an acronym for “global regular expression print,” a reference to the command in the old **ed** line editor that prints all of the lines in a file containing a specified sequence of characters. grep does exactly that: it prints out lines in a file that contain a match for a *regular expression*. We'll gradually delve into what a regular expression is as we go on.

Starting Out

First, let's look at a quick example. We will search for a word in the Configure script provided with Linux for setting up the Linux kernel source, which is usually installed in the /usr/src/linux directory. Change to that directory and type (the **$** character is the prompt, don't type it):

```
$ grep glob Configure
```

You should see:

```
# Disable filename globbing once and for all.
```

glob is in **bold** to illustrate what grep matched. grep does not actually print matches in bold.

grep looked for the sequence of characters **glob** and printed the line of the Configure file with that sequence. It did not look for the **word glob**. It looked for **g** followed by **l** followed by **o** followed by **b**. This points out one important aspect of regular expressions: they match sequences of characters, not words.

Before we dig any deeper into the specifics of pattern matching, let's look at grep's “user interface” with a few examples. Try the following two commands:

```
$ grep glob < Configure
$ cat Configure | grep glob
```

both of these two commands should print

```
# Disable filename globbing once and for all.
```

which probably looks familiar.

In all of these commands, we have specified the regular expression as the first argument to grep. With the exception of any command line switches, grep always expects the regular expression as the first argument.

However, we presented grep with three different situations and received the same response. In the first exercise, we provided grep with the name of a file, and it opened that file and searched it. grep can also take a list of filenames to search.

In the other two exercises we illustrated a feature that grep shares with many other utilities. If no files are specified on the command line, grep reads *standard input*. To further illustrate standard input let's try one more example:

```
$ grep foo
```

When you run that, grep appears to “hang” waiting for something. It is. It's waiting for input. Type:

```
tttt
```

and press **return**. Nothing happens. Now type:

```
foobar
```

and press enter. This time, grep sees the string **foo** in **foobar** and echos the line **foobar** back at you, which is why **foobar** appears twice. Now type **ctrl**-d, the “end-of-file” character, to tell grep that it has reached the end of the file, whereupon it exits.

You just gave grep an input file that consisted of **tttt**, a newline character, **foobar**, a newline character, and the end-of-file character.

Piping input into grep from standard input also has another frequent use: filtering the output of other commands. Sometimes cutting out the unnecessary lines with grep is more convenient than reading output page by page with **more** or **less**:

```
$ ps ax | grep cron
```

efficiently gives you the process information for crond.

Special Characters

Many Unix utilities use regular expressions to specify patterns. Before we go into actual examples of regular expressions, let's define a few terms and explain a few conventions that I will use in the exercises.

- **Character** any printable symbol, such as a letter, number, or punctuation mark.
- **String** a sequence of characters, such as **cat** or **segment** (sometimes referred to as a **literal**).
- **Expression** also a sequence of characters. The difference between a string and an expression is that while strings are to be taken literally, expressions must be evaluated before their actual value can be determined. (The manual page for GNU grep compares regular expressions to mathematical expressions.) An expression usually can stand for more than one thing, for example the regular expression **th[ae]n** can stand for **then** or **than**. Also, the shell has its own type of expression, called *globbing*, which is usually used to specify file names. For example, ***.c** matches any file ending in the characters **.c**.

- **Metacharacters** the characters whose presence turns a string into an expression. Metacharacters can be thought of as the operators that determine how expressions are evaluated. This will become more clear as we work through the examples below.

Interference

You have probably entered a shell command like

```
$ ls -l *.c
```

at some time. The shell "knows" that it is supposed to replace ***.c** with a list of all the files in the current directory whose names end in the characters **.c**.

This gets in the way if we want to pass a literal ***** (or **?**, **|**, **$**, etc.) character to grep. Enclosing the regular expression in `single quotes' will prevent the shell from evaluating any of the shell's metacharacters. When in doubt, enclose your regular expression in single quotes.

Basic Searches

The most basic regular expression is simply a string. Therefore a string such as **foo** is a regular expression that has only one match: **foo**.

We'll continue our examples with another file in the same directory, so make sure you are still in the /usr/src/linux directory:

```
$ grep Linus CREDITS
```

```
Linus
N: Linus Torvalds
E: Linus.Torvalds@Helsinki.FI
D: Personal information about Linus
```

This quite naturally gives the four lines that have Linus Torvalds' name in them.

As I said earlier, the Unix shells have different metacharacters, and use different kinds of expressions. The metacharacters **.** and ***** cause the most confusion for people learning regular expression syntax after they have been using shells (and DOS, for that matter).

In regular expressions, the character **.** acts very much like the **?** at the shell prompt: it matches any single character. The *****, by contrast, has quite a different meaning: it matches *zero* or more instances of the *previous* character.

If we type

```
$ grep tha. CREDITS
```

we get this (partial listing only):

```
S: Northampton
E: Hein@Informatik.TU-Clausthal.de
```

As you can see, grep printed every instance of **tha** followed by any character. Now try

```
$ grep 'tha*' CREDITS
S: Northampton
D: Author of serial driver
D: Author of the new e2fsck
D: Author of loopback device driver
```

We received a much larger response with "*****". Since "*****" matches *zero* or more instances of the previous character (in this case the letter "a"), we greatly increase our possibility of a match because we made **th** a legal match!

Character Classes

One of the most powerful constructs available in regular expression syntax is the *character class*. A character class specifies a range or set of characters to be matched. The characters in a class are delineated by the **[** and **]** symbols. The class **[a-z]** matches the lowercase letters **a** through **z**, the class **[a-zA-Z]** matches all letters, uppercase or lowercase, and **[Lh]** would match upper case **L** or lower case **h**.

```
$ grep 'sm[ai]' CREDITS
E: csmith@convex.com
D: Author of several small utilities
```

since our expression matches **sma** or **smi**. The command

```
$ grep '[a-z]' CREDITS
```

gives us most of the file. If you look at the file closely, you'll see that a few lines have no lowercase letters; these are the only lines that grep does not print.

Now since we can match a set of characters, why not exclude them instead? The circumflex, **^**, when included as the *first* member of a character class, matches any character *except* the characters specified in the class.

```
$ grep Sm CREDITS
```

gives us three lines:

```
D: Small patches for kernel, libc
D: Smail binary packages for Slackware and Debian
N: Chris Smith
$ grep 'Sm[^i]' CREDITS
```

gives us two

```
D: Small patches for kernel, libc
D: Smail binary packages for Slackware and Debian
```

because we excluded **i** as a possible letter to follow **Sm**.

To search for a class of characters including a literal **^** character, don't place it first in the class. To search for a class including a literal **-**, place it the very last character of the class. To search for a class including the literal character **]**, place it the first character of the class.

Often it is convenient to base searches on the position of the characters on a line. The **^** character matches the beginning of a line (outside of a character class, of course) and the **$** matches the end. (Users of vi may recognize these metacharacters as commands.) Earlier, searching for **Linus** gave us four lines. Let's change that to:

```
grep 'Linus$' CREDITS
```

which gives us

```
Linus
D: Personal information about Linus
```

two lines, since we specified that **Linus** must be the last five characters of the line. Similarly,

```
grep - CREDITS
```

produces 99 lines, while

```
grep '^-' CREDITS
```

produces only one line:

```
----------
```

In some circumstances you may need to match a metacharacter. Inside a character class set all characters are taken as literals (except **^**, **-**, and **]**, as shown above). However, outside of classes we need a way to turn a metacharacter into a literal character to match.

Matching Metacharacters

For this purpose the special metacharacter **** is used to *escape* metacharacters. Escaped metacharacters are interpreted literally, not as a component of an expression. Therefore **\[** would match any sequence with a **[** in it:

```
$ grep '[' CREDITS
```

produces an error message:

```
grep: Unmatched [ or [^
```

but

```
$ grep '\[' CREDITS
```

produces two lines:

```
E: hennus@sky.ow.nl [My uucp-fed Linux box at home]
D: The XFree86[tm] Project
```

If you need to search for a **** character, escape it just like any other metacharacter: ****

Options

As you can see, with just its support of regular expression syntax, grep provides us with some very powerful capabilities. Its command line options add even more power.

Sometimes you are looking for a string, but don't know whether it is upper, lower, or mixed case. For this situation grep offers the **-i** switch. With this option, case is completely ignored:

```
$ grep -i lINuS CREDITS
                        Linus
N: Linus Torvalds
E: Linus.Torvalds@Helsinki.FI
D: Personal information about Linus
```

The **-v** option causes grep to print all lines that do **not** contain the specified regular expression:

```
$ grep -v '^#" /etc/syslog.conf | grep -v '^$'
```

prints all the lines from /etc/syslog.com that are neither commented (starting with **#**) nor empty (**^$**). This prints six lines on my system, although my syslog.conf file really has 21 lines.

If you need to know how many lines match, pass grep the **-c** option. This will output the number of matching lines (not the number of matches; two matches in one line count as one) without printing the lines that match:

```
$ grep -c Linux CREDITS
33
```

If you are searching for filenames that contain a given string, instead of the actual lines that contain it, use grep's **-l** switch:

```
$ grep -l Linux *
CREDITS
README
README.modules
```

grep also notifies us, for each subdirectory, that it can't search through a directory. This is normal and will happen whenever you use a wildcard that happens to include directory names as well as file names.

The opposite of **-l** is **-L**. This option will cause grep to return the names of files that *do not* contain the specified pattern.

If you are searching for a word and want to suppress matches that are partial words use the **-w** option. Without the **-w** option,

```
$ grep -c a README
```

tells us that it matched 146 lines, but

```
$ grep -wc a README
```

returns only 35 since we matched only the word **a**, not every line with the character **a**.

Two more useful options:

```
$ grep -b Linus CREDITS
301:                    Linus
17446:N: Linus Torvalds
17464:E: Linus.Torvalds@Helsinki.FI
20561:D: Personal information about Linus
$ grep -n Linus CREDITS
7:                      Linus
793:N: Linus Torvalds
794:E: Linus.Torvalds@Helsinki.FI
924:D: Personal information about Linus
```

The **-b** option causes grep to print the byte offset (how many bytes the match is from the beginning of the file) of each match before the corresponding line of output. The **-n** switch gives the line number.

Another grep

GNU also provides **egrep** (enhanced grep). The regular expression syntax supported by GNU egrep adds a few other metacharacters:

- ? Like *, except that it matches zero or one instances instead of zero or more.

- + the preceding character is matched one or more times.
- | separates regular expressions by ORing them together.

```
$ egrep -i 'linux|linus' CREDITS
```

outputs any line that contains **linus** or **linux**.

To allow for legibility, parentheses "**(**" and "**)**" can be used in conjunction with "**|**" to separate and group expressions.

More Than Just grep

This covers many of the features provided by grep. If you look at the manual page, which I strongly recommend, you will see that I did leave out some command-line options, such as different ways to format grep's output and a method for searching for strings without employing regular expressions.

Learning how to use these powerful tools provides Linux users with two very valuable advantages. The first (and most immediate) of these is a time-saving way to process files and output from other commands.

The second is familiarity with regular expressions. Regular expressions are used throughout the Unix world in tools such as **find** and **sed** and languages such as **awk**, **perl** and **Tcl**. Learning this syntax prepares you to use some of the most powerful computing tools available.

Eric Goedelbecker is a systems analyst for Reuters America, Inc. He supports clients (mostly financial institutions) who use market data retrieval and manipulation APIs in trading rooms and back office operations. In his spare time (about 15 minutes a week...), he reads about philosophy and hacks around with Linux. He can be reached via e-mail at eric@nymt.reuter.com.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Writing man Pages Using groff

Matt Welsh

Issue #18, October 1995

groff is the GNU version of the popular nroff/troff text-formatting tools provided on most Unix systems. Its most common use is writing manual pages —online documentation for commands, programming interfaqces, and so forth. In this article, we show you the ropes of writing your own man pages with groff.

Two of the original text processing systems found on Unix systems are troff and nroff, developed at Bell Labs for the original implementation of Unix (in fact, the development of Unix itself was spurred, in part, to support such a text-processing system). The first version of this text processor was called **roff** (for "runoff"); later came troff, which generated output for a particular typesetter in use at the time. nroff was a later version that became the standard text processor on Unix systems everywhere. groff is GNU's implementation of nroff and troff that is used on Linux systems. It includes several extended features and drivers for a number of printing devices.

groff is capable of producing documents, articles, and books, much in the same vein as other text-formatting systems, such as TeX. However, groff (as well as the original nroff) has one intrinsic feature that is absent from TeX and variants: the ability to produce plain-ASCII output. While other systems are great for producing documents to be printed, groff is able to produce plain ASCII to be viewed online (or printed directly as plain text on even the simplest of printers). If you're going to be producing documentation to be viewed online, as well as in printed form, groff may be the way to go (although there are alternatives, such as Texinfo, Lametex, and other tools).

groff also has the benefit of being much smaller than TeX; it requires fewer support files and executables than even a minimal TeX distribution.

One special application of groff is to format Unix man pages. If you're a Unix programmer, you'll eventually need to write and produce man pages of some

kind. In this article, we'll introduce the use of groff through the writing of a short man page.

As with TeX, groff uses a particular text-formatting language to describe how to process the text. This language is slightly more cryptic than systems such as TeX, but also less verbose. In addition, groff provides several macro packages that are used on top of the basic formatter; these macro packages are tailored to a particular type of document. For example, the **mgs** macros are an ideal choice for writing articles and papers, while the **man** macros are used for man pages.

Writing a man Page

Writing man pages with groff is actually quite simple. For your man page to look like others, you need to follow several conventions in the source, which are presented below. In this example, we'll write a man page for a mythical command **coffee** that controls your networked coffee machine in various ways.

Using any text editor, enter the source from Listing 1 and save the result as **coffee.man**. Do not enter the line numbers at the beginning of each line; those are used only for reference later in the article.

```
 1:  .TH COFFEE 1 "23 March 94"
 2:  .SH NAME
 3:  coffee /- Control remote coffee machine
 4:  .SH SYNOPSIS
 5:  /fBcoffee/fP [ -h | -b ] [ -t /fItype/fP ]
 6:  /fIamount/fP
 7:  .SH DESCRIPTION
 8:  /fBcoffee/fP queues a request to the remote
 9:  coffee machine at the device /fB/dev/cf0/fR.
10:  The required /fIamount/fP argument specifies
11:  the number of cups, generally between 0 and
12:  12 on ISO standard coffee machines.
13: .SS Options
14: .TP
15: /fB-h/fP
16: Brew hot coffee. Cold is the default.
17: .TP
18: /fB-b/fP
19: Burn coffee. Especially useful when executing
20: /fBcoffee/fP on behalf of your boss.
21: .TP
22: /fB-t /fItype/fR
23: Specify the type of coffee to brew, where
24: /fItype/fP is one of /fBcolumbian/fP,
25: /fBregular/fP, or /fBdecaf/fP.
26: .SH FILES
27: .TP
28: /fC/dev/cf0/fR
29: The remote coffee machine device
30: .SH "SEE ALSO"
31: milk(5), sugar(5)
32: .SH BUGS
33: May require human intervention if coffee
34: supply is exhausted.
```

Listing 1. Example man Page Source File

Don't let the amount of obscurity in this source file frighten you. It helps to know that the character sequences **\fB**, **\fI**, and **\fR** are used to change the font to boldface, italics, and roman type, respectively. **\fP** sets the font to the one previously selected.

Other groff requests appear on lines beginning with a dot (**.**). On line 1, we see that the **.TH** request is used to set the title of the man page to **COFFEE**, the man section to **1**, and the date of the last man page revision. (Recall that man section 1 is used for user commands, section 2 is for system calls, and so forth. The **man man** command details each section number.) On line 2, the **.SH** request is used to start a section, entitled **NAME**. Note that almost all Unix man pages use the section progression **NAME**, **SYNOPSIS**, **DESCRIPTION**, **FILES**, **SEE ALSO**, **NOTES**, **AUTHOR**, and **BUGS**, with extra, optional sections as needed. This is just a convention used when writing man pages and isn't enforced by the software at all.

Line 3 gives the name of the command and a short description, after a dash (**[mi]**). You should use this format for the **NAME** section so that your man page can be added to the **whatis** database used by the **man -k** and **apropos** commands.

On lines 4—6 we give the synopsis of the command syntax for coffee. Note that italic type **\fI**...**\fP** is used to denote parameters on the command line, and that optional arguments are enclosed in square brackets.

Lines 7—12 give a brief description of the command. Boldface type is generally used to denote program and file names. On line 13, a subsection named **Options** is started with the **.SS** request. Following this on lines 14—25 is a list of options, presented using a tagged list. Each item in the tagged list is marked with the **.TP** request; the line *after* **.TP** is the tag, after which follows the item text itself. For example, the source on lines 14—16:

```
.TP
\fB-h\P
Brew hot coffee. Cold is the default.
```

will appear as the following in the output:

```
-h      Brew hot coffee. Cold is the default.
```

You should document each command-line option for your program in this way.

Lines 26—29 make up the **FILES** section of the man page, which describes any files that the command might use to do its work. A tagged list using the **.TP** request is used for this as well.

On lines 30—31, the **SEE ALSO** section is given, which provides cross-references to other man pages of note. Notice that the string **<\#34>SEE ALSO<\#34>** following the **.SH** request on line 30 is in quotes; this is because **.SH** uses the first whitespace-delimited argument as the section title. Therefore any section titles that are more than one word need to be enclosed in quotes to make up a single argument. Finally, on lines 32—34, the **BUGS** section is presented.

Formatting and Installing the man Page

In order to format this man page and view it on your screen, you can use the command:

```
$ groff -Tascii -man coffee.man | more
```

The **-Tascii** option tells groff to produce plain-ASCII output; **-man** tells groff to use the man page macro set. If all goes well, the man page should be displayed as shown in Figure 1.

```
COFFEE(1)                                                COFFEE(1)
NAME
       coffee - Control remote coffee machine
SYNOPSIS
       coffee [ -h | -b ] [ -t type ] amount
DESCRIPTION
       coffee  queues  a  request to the remote coffee machine at
       the device /dev/cf0. The required amount  argument  speci-
       fies the number of cups, generally between 0 and 12 on ISO
       standard coffee machines.
   Options
       -h     Brew hot coffee. Cold is the default.
       -b     Burn coffee. Especially useful when executing  cof-
              fee on behalf of your boss.
       -t type
              Specify  the  type of coffee to brew, where type is
              one of columbian, regular, or decaf.
FILES
       /dev/cf0
              The remote coffee machine device
SEE ALSO
       milk(5), sugar(5)
BUGS
       May  require  human  intervention  if  coffee  supply   is
       exhausted.
```

Figure 1. Formatted man Page

As mentioned before, groff is capable of producing other types of output. Using the **-Tps** option in place of **-Tascii** will produce PostScript output that you can save to a file, view with GhostView, or print on a PostScript printer. **-Tdvi** will produce device-independent .dvi output similar to that produced by TeX.

If you wish to make the man page available for others to view on your system, you need to install the groff source in a directory that is present in other users' **MANPATH**. The location for standard man pages is /usr/man. The source for section 1 man pages should therefore go in /usr/man/man1. Therefore, the command:

```
$ cp coffee.man /usr/man/man1/coffee.1
```

will install this man page in /usr/man for all to use (note the use of the **.1** file name extension, instead of **.man**). When **man coffee** is subsequently invoked, the man page will be automatically reformatted, and the viewable text saved in /usr/man/cat1/coffee.1.Z.

If you can't copy man page sources directly to /usr/man (say, because you're not the system administrator), you can create your own man page directory tree and add it to your **MANPATH**. The **MANPATH** environment variable is of the same format as **PATH**; for example, to add the directory /home/mdw/man to **MANPATH** just use:

```
$ export MANPATH=/home/mdw/man:$MANPATH
```

There are many other options and formatting commands available for groff and the man page macros. The best way to find out about these is to look at the files in /usr/lib/groff; the tmac directory contains the macro files themselves, which often contain some documentation on the commands they provide. To use a particular macro set with groff, just use the **-m*macro*** option. For example, to use the **mgs** macros, use:

```
groff -Tascii -mgs files...
```

The man pages for groff describe this option in more detail.

Unfortunately, the macro sets provided with groff are not well-documented. There are section 7 man pages for some of them; for example, **man 7 groff_mm** will tell you about the **mm** macro set. However, this documentation usually only covers the differences and new features in the groff implementation, which assumes you have access to the man pages for the original nroff/troff macro sets (known as DWB—the Documentor's Work Bench). The best source of information may be a book on using nroff/troff which covers these classic macro sets in detail. For more about writing man pages, you can always look at the man page sources (in /usr/man) and determine what they do by comparing the formatted output with the source.

This article is adapted from Running Linux, by Matt Welsh and Lar Kaufman, published by O'Reilly and Associates (ISBN 1-56592-100-3). Among other things, this book includes tutorials of various text-formatting systems used under Linux. Information in this issue of *Linux Journal* as well as *Running Linux* should provide a good head-start on using the many text tools available for the system.

Good luck, and happy documenting!

Matt Welsh (mdw@cs.cornell.edu) is a student and systems programmer at Cornell University, working with the Robotics and Vision Laboratory on projects dealing with real-time machine vision.

Archive Index Issue Table of Contents

Advanced search

LaTeX for the Slightly Timid

Kim Johnson

Issue #18, October 1995

Are you used to the simplicity of your old DOS word processor, and afraid of the complexity of LaTeX? This article tells you how to write a beautiful letter—and gets you started using LaTeX.

I admit it. LaTeX was not my first choice for a text formatter. I was pressured into using LaTeX by my (then) fiancé, and the fact that at the end of the semester, all the IBM PCs and the Macs were being used 24 hours a day, while the Unix terminals were almost totally free. Also, I had to reboot from Linux into DOS every time I wanted to use one of my favorite word processing programs on my fiancé's computer, which was no fun. So, armed with a topology assignment I wanted to type, I started to learn LaTeX.

Even after the first week or so of using LaTeX, it was not my favorite program. It wasn't a word processor or an editor—it was something new to me. I didn't see the italics or the underlining or my math equations on the screen as I entered or edited my papers. After a while, though, I got used to the rules, and my papers looked so beautiful that I fell in love with LaTeX. I am not a LaTeX or TeX (LaTeX is based on TeX) guru—it is all I can do to remember how to start a document. But the freedom and power of LaTeX make it well worth the initial time investment to learn it, and if you are doing ordinary everyday writing, LaTeX is not terribly difficult to use.

It Takes Some Work

There are a few major differences between LaTeX and your favorite word processor. It is not WYSIWYG (what you see is what you get) or WYSLRN (what you see looks really neat—bold is **bold**, italic is *italic*, but new lines and page breaks aren't necessarily where they will be when the document is printed). With LaTeX, what you see is what LaTeX sees, and you type in the exact commands that you want LaTeX to use. This means that WYSLRU (what you see looks really ugly) while you are entering your text. [There are so-called front

ends to LaTeX that are almost WYSIWYG. *Linux Journal* will probably contain articles on those in later issues, because they can make it much easier for you to use LaTeX—ED] However, because what you see is exactly what LaTeX sees, if something in your document output doesn't look right, you can perhaps find it and fix it more easily than you could with a word processor.

Another difference is that when it comes to formatting decisions, LaTeX will handle the details if you handle the general idea. For example, if you want your footnotes labeled with letters instead of numbers, or the numbering of footnotes to begin at 1 on every page, or some other labeling scheme, you need only instruct LaTeX once at the beginning of your document. If you change your mind, you don't need to change all your footnotes—just change the instructions at the beginning, and LaTeX will handle the rest.

The last difference is that there is no standard user interface for LaTeX; you can use whatever text editor you want. I use Emacs, but any Emacs clone, or vi or another editor which uses plain text, will work. Because you can use any editor, starting an article or paper is a little more complex than in a word processor. You need to tell LaTeX the type of document you are creating, normally a “book”, “article”, “report”, or “letter”. Any book on LaTeX will explain the specific differences between the document types; their names should give a good idea of what they are for.

Writing a Letter

Having told LaTeX which type of document you are doing, it figures out the page formatting and the special options needed throughout the document. I will use the “letter” document style to demonstrate the basics of typing documents in LaTeX, as well as the power of LaTeX to make documents easy to format and beautiful.

To begin any document, after starting your editor you need to place a preamble at the start of the file, of the form:

```
documentstyle[...]{...}
```

A few things need to be noted about this preamble. First, the backslash. LaTeX uses the backslash before most commands, so you should get familiar with its location on your keyboard. The brackets contain options that change something less drastic than the document style. For instance, you can change the type size from 10 point to 11 or 12 point, set the entire document in two columns, or use other more advanced options. The braces hold the name of the document type: usually article or letter. The braces and their contents are required.

Since we are writing a letter, we will start out with:

```
documentstyle{letter}
```

All documents need **\begin{document}** at the beginning (preceded, of course, by the **documentstyle** command and, often, by other more advanced commands) and **end{document}** at the end.

Letters are unique among the styles. You may want to write several letters in the same file, reusing the same return address and signature. LaTeX allows you to do this by stating the address and signature before the **\begin{document}** statement. Then, each letter is begun with **\begin{letter}** and ended with **end{letter}**. After all your letters, put **end{document}** to end the document.

The format of the **address** and **signature** commands are similar and something you will become very familiar with in LaTeX. Type the command, followed by the argument (for these commands, your address and your name and title) in braces:

```
address{Linux Journal\\P.O. Box 85867\\
Seattle, WA 98145—1867}
signature{Kim Johnson\\Chief Bottle Washer}
```

Notice the double backslashes; these are added to force a new line—simply pressing **return** will not work, as I explain later.

Now, begin your first letter:

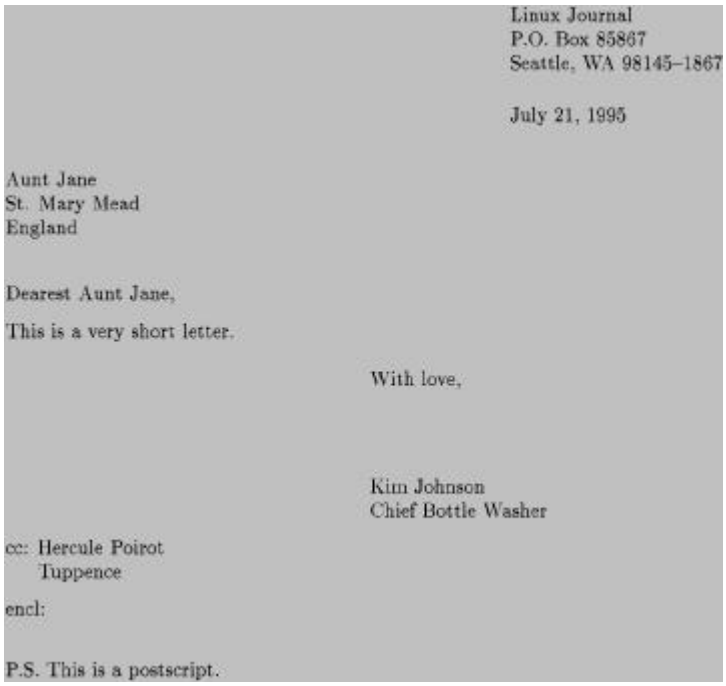
```
\begin{document}
\begin{letter}{Aunt Jane\\
St. Mary Mead\\England}
```

The argument in the second set of braces is the address of the recipient of the letter. Note the recurring double backslashes and the tendency to put arguments in braces to set them apart. You may want to add an **end{letter}** at this point and then type the rest of the letter between the **\begin** and **end**—it saves a lot of time debugging later.

The other options are mostly self-explanatory from the example letter.

```
opening{Dearest Aunt Jane,}
This is a very short letter.
closing{With love,}
cc{Hercule Poirot\\Tuppence}
encl
ps{P.S. This is a postscript.}
end{letter}
end{document}
```

Your name is added underneath the closing of the letter from the **signature** command that was used before the **\begin{letter}**, as shown in Figure 1.



Linux Journal
P.O. Box 85867
Seattle, WA 98145–1867

July 21, 1995

Aunt Jane
St. Mary Mead
England

Dearest Aunt Jane,

This is a very short letter.

With love,

Kim Johnson
Chief Bottle Washer

cc: Hercule Poirot
Tuppence

encl:

P.S. This is a postscript.

Figure 1. Output from LaTeX

It would be possible to arrange this letter on the page “by hand” on most word processors, and even on LaTeX, but the point is LaTeX does the formatting and you do the writing. The other document styles do not format quite so aggressively, but they do allow you to add your title and dedications without having to worry about getting everything centered and on the correct page.

Outputting Your Document

Now, suppose that you want to see what your letter really looks like—you certainly don't want to send it as it appears in your letter.tex file. So, after saving the file and exiting the editor (or just popping over to another virtual console), from within the directory where letter.tex is, type latex **letter.tex**. LaTeX will format your file it in a way that can be displayed and printed. The formatted file will be called letter.dvi; DVI stands for “device independent”, which means that your file can be displayed or printed using a number of programs. If you are using X-Windows, type **xdvi letter.dvi** to see what you have written. If you are not running X-Windows, you may be able to use **dvgt**, which comes with some distributions and is also available from sunsite.unc.edu in the /pub/Linux/apps/tex/dvi/ directory. However, its user interface is not the best possible for a novice, and **xdvi** is certainly easier to use.

Special Formatting Commands

“The text looks great!” you say, “But I didn't just want to type plain text in, I wanted to add italics and underlining and mathematical equations and footnotes and...” (If you didn't get past the command **latex letter.tex**, I'll give

some hints for debugging your letter later.) Most of these commands are easy enough and very similar to the commands used so far.

For example, to create a footnote, type **footnote{This is the text of the footnote.}** wherever you want the footnote to appear. Note that you will have to be careful with spacing—put the **footnote** command after any punctuation, but before any trailing space. LaTeX will make a complete list of the footnotes and number them correctly. However, if you want to number them yourself, use **footnote[*num*]{text}** instead. Remember that brackets contain optional things, so you don't need to number the footnotes yourself unless automatic numbering won't do what you want.

Creating italic and bold text is very similar, except the bold and italic commands are actually inside the braces instead of outside. This is so that more than one command can apply to a region. Here are some examples: I can make some words **{\bf *bold*}**, **{em *italic*}**, **{em\bf *bold-italic*}**, or **{em *have* a {\bf *bold*} word in a group of *italic words*}**. (em stands for emphasized text; it, which stands for italic, works as well.) LaTeX has many other type styles which work the same as bold and italic; they are documented in all LaTeX reference books.

Another method used in LaTeX for text formatting is similar to the one you used for the letter environment with the **\begin** and **end** commands. For example, here is how to center text:

```
\begin{center}
Here is my centered text,\\
here are two\\
more lines of centered text.
end{center}
```

Again, as in the letter, the double backslashes signal a new line. The **\begin** and **end** commands, along with **{table}**, **{quotation}** and many other options, are documented in the reference manuals.

To sum up, when dealing with some text which is to be put in a separate place on the page, as in **footnote** and **opening**, the text goes after the command in braces. When dealing with a few words in the flow of the main text, the command (like **em** and **\bf**) goes within the brackets along with the text, to set it off from the text around it. And finally, when dealing with larger blocks of text within the document which need to be displayed specially, such as centered text or a table, commands such as **\begin{center}** and **end{center}** are used around the text.

A note about line breaks and spacing: When LaTeX sees a line break in your typed-in text, it just assumes that your line got too long and you went to the

next line to keep entering text for the same paragraph. LaTeX knows better than you do how many words fit on a line, so one line break just doesn't register with LaTeX. Two or more line breaks (one or more blank lines), on the other hand, are interpreted as a "new paragraph", so LaTeX will skip a line and/or indent, whatever is appropriate in your document style, and will not put that extra new line in the output. If you want to force LaTeX to break a line without a new paragraph, you must use a ****, a double backslash.

LaTeX also interprets extra spaces and extra empty lines just as it would one space or one empty line. You probably won't know exactly how many lines to leave blank; just leave it up to LaTeX. As in the letter above, LaTeX knows where on the page to put everything so it looks good, so let it do the work. On the other hand, if LaTeX does not know how to make things look right, you do have some control. As above, use **** (or equivalently, **linebreak**) to specify a line break, and **pagebreak** to begin a new page. If this doesn't work, see one of the books about it, or else ask your local "TeXnician" (TeX-speak for "TeX guru").

Finally, you can add horizontal and vertical space with **hspace{*width*}** and **vspace{*height*}** respectively, where ***width*** and ***height*** refer to the amount of space you want added. For example, **hspace{.25in}** would make the current line be at least a quarter inch high; it's the equivalent of an infinitely thin letter a quarter of an inch high. This may not work at the beginnings of paragraphs, the end of lines, the end of pages, and various other spots. Again, see the book if you can't get it to work the way you want!

But It Didn't Work

Now, back to your letter. Suppose that you typed it all in, tried to LaTeX it, and it didn't work. This happens to everyone once in a while. To get back to the prompt after a failed LaTeX run, type **q** or **x** or **ctrl**-d (this is one of my favorite features of TeX—it's very easy to get out of). A number of common problems cause errors. First, did you remember to put an **end{document}** at the end of your file? Or are there any other **\begin** statements without a matching **end** statement? A variation of this is missing or mismatched braces or brackets. Another common problem is misspelled commands—if you tell LaTeX to "**senter**", it doesn't know what to do. My most common problem using forward slashes instead of backslashes, so I routinely do a search-and-replace for forward slashes with backslashes before printing. Also, I use Emacs in latex-mode, so it warns me about mismatched braces.

Finally, LaTeX has some special characters which can't be used in text as they are, because they have special meanings (which aren't covered in this article). These characters are:

```
# $ % & _ { } ~ ^ < >
```

All of these characters can be produced by LaTeX. The first seven can be produced by typing **#**, **$**, **\\%**, **&**, **_**, **{**, and **}**, and the rest can be produced by typing **verb+~+**, **verb+^+**, **verb+<+**, **verb+>+**, and **verb++**.

LaTeX's error messages are usually helpful, giving a line number near where the error may have occurred and the error type. They are not always correct, though, so you must do a bit of work, but since everything is out in the open, it is usually not too onerous. I recommend frequently saving and LaTeXing your work, so you catch errors quickly and not after your 3000 word report is finished (or so you thought). It is also useful to display your work frequently, and if you are using X-Windows, you can iconify the **xdvi** window when you are entering text or LaTeXing. **xdvi** will automatically use the newest version of the dvi file, so you don't need to exit and restart xdvi over and over.

Printing

When you are finished editing and are ready to print, you have a few options. If you have a PostScript printer, the easiest way to print is probably to use the **dvips** program that should have come with your TeX/LaTeX installation. If you have the **lpr** program installed, the command:

```
dvips filename.dvi | lpr
```

will print the file. If you want to create a PostScript file, you can instead type:

```
dvips filename.dvi -o filename.dvi
```

Other programs are available for other printers. A series of drivers for the HP LaserJet printers are commonly used; for a LaserJet 4, you might type:

```
dvilj4 -e- filename.dvi | lpr
```

to print. The documentation for the LaserJet drivers is available by typing **man dvilj**.

If you have any other printer, it is probably supported by the Ghostscript program. Ghostscript is a PostScript language interpreter included in most Linux distributions. Setting it up is beyond the scope of this article, but once it is set up, you can use **dvips** to create a PostScript file and Ghostscript to print it. For instance, if you have a printer which is compatible with the HP DeskJet 500, you can run:

```
gs -q -sDEVICE=djet500 filename.ps quit
```

A full list of devices is available by typing:

```
gs
devicenames ==
quit
```

Don't use the "**/**" preceding the device names.

If your printer supports printing at different resolutions, you may be able to specify the resolution on the **gs** command line as well. For example, for 240 pixels per inch by 72 pixels per inch, add **-r240x72** to the command line before the filename. More information about Ghostscript may be obtained by running **man gs**.

Warning: Do not just type **lpr letter.dvi** to print your letter; it will most likely be a mess. Most Linux distributions do not come set up to automatically print DVI files correctly. It is possible to install "print filters" which will make your life easier. One such program is called **apsfilter** and will be explained in a future article in *Linux Journal*.

Not Just Letters

You will probably want to do more than write letters. Other document styles are available, as mentioned earlier. This article is not the place to go into all of them, but I find "article" a good style to use in general. You can make a generic template document with the preamble and any other options you like consistently. Then, whenever you want to write a new document, copy that file to your new document, and you will be ready to go with no fuss.

Finally, good luck and have fun! LaTeX is not as difficult as some people make it seem, and the results after a little work can be truly dazzling.

> Kim Johnson is working towards her PhD in Mathematics at the University of North Carolina, Chapel Hill. When she isn't studying, she is usually reading a classic British mystery novel.

LaTeX Resources

- A new version of LaTeX, called LaTeX2e, is now available. Most (but not all) Linux distributions today come with LaTeX2e, which is able to understand the older LaTeX version 2.09 format that this article explains ("compatibility mode"), but also has a native mode which is more powerful, flexible, and extensible. This article covers LaTeX 2.09 since both LaTeX 2.09 and LaTeX2e understand the 2.09 format, so no matter what version of LaTeX came with your distribution of Linux, the examples will work.

- However, if LaTeX intrigues you, you will want to buy a book about it, and the best books available now cover the LaTeX2e format. The differences from what you have learned in this article are not great and are explained in these books.
- The original LaTeX documentation is *LaTeX: A Document Preparation System* (ISBN 0--201--15790--X) by Leslie Lamport, the original author of LaTeX. It provides a good basic introduction to LaTeX, but is sometimes frustratingly lacking in details (like this article). The second edition of this book covers LaTeX2e; the first edition covered LaTeX 2.09.
- *The LaTeX Companion* (ISBN 0--201--54199--8) describes LaTeX2e in detail. By contrast with Lamport's book, *The LaTeX Companion* goes into more detail than any one person would be likely to use—and so will usually have the details you want.
- Since LaTeX is built on top of the venerable TeX formatter, Donald Knuth's *The TeXbook* (ISBN 0--201--13447--0) provides the documentation for the underlying system, and may be useful if you wish to become a LaTeX expert.
- The O'Reilly book *Making TeX Work* (ISBN 1--56592--051--1), by Norman Walsh, provides a different kind of information from the other books mentioned here. It may help you discover how to find, use, and install many of the extra pieces that make things work better in your particular situation and that fulfill your particular typesetting needs.
- The Usenet newsgroup comp.text.tex provides a forum for discussing TeX and LaTeX.

Archive Index Issue Table of Contents

Advanced search

Linux on Alpha: A Strategic Choice

Jon "maddog" Hall

Issue #18, October 1995

A true story of love at first sight.

"Leyenooks?," I asked, "What is that?" I must admit that I was skeptical. Although the young man in front of me seemed amicable enough, it was hard to imagine that he headed up an effort to create a freeware Unix-like operating system. However, Kurt Reisler was enthusiastic about him, and after ten years of association with Kurt as the chairman of the Unix Special Interest Group (UNISIG) of the Digital Equipment Corporation User's Society (DECUS), instinct told me to go along with his ideas. That is why I asked my management to fund Linus Torvalds' first trip to DECUS in New Orleans (spring, 1994), and to fund some equipment at the show to demonstrate Linux.

I had my doubts about this funding as Kurt struggled to get Linux installed on that first PC, but after some able assistance from Linus, he did get it working. I had my first look at the operating system running and in less than ten minutes I had convinced myself that "this was Unix enough for me." Instinct told me, "this is good."

Later that week Linus joined a few of us for a ride on the Nachez, a steam boat that plies the Mississippi River. As we rode up and down the river I started thinking about what Linux might mean for the educational community, and what it might mean for Digital.

Twenty-five years ago I was a student at a university in Philadelphia. Although we had a large computer system, it was kept behind glass doors, and batch jobs on computer cards were passed through a narrow opening in the wall, with printouts coming back over a 24-hour period.

Trying to learn operating systems design on such a system meant using an emulator, and the process of using that emulator through punched cards was really painful, about like having a root canal without anesthetic.

Fortunately at the same school was a little minicomputer lab, and in that lab were three small PDP-8 machines from Digital. It was on these machines, with the aid of some free architecture books given to me by the Digital salesman, and some freeware software that came from DECUS, that I really started to learn about how computer systems worked. I always remembered that lab, those machines, and that Digital salesman.

Years later, after working on large IBM mainframes, heading the department at a small two-year technical college (using Digital's equipment once again), and learning Unix at Bell Laboratories (on Digital's VAX machines) I had an offer to work with Digital's Unix group in Nashua, New Hampshire. I took that offer, in hopes of being able to contribute to the same environment that had helped me learn computers in those early years.

Working in the Unix group, I often heard about universities, colleges, and even high schools that wanted to use Unix to teach computer science. Despite the origins of Unix as a research tool, and the vast contributions to Unix made by the University of California Berkeley and other schools, the licensing terms of our product did not make it easy to share source code.

As a commercial Unix system, we license technology from a variety of companies and integrate that technology into our sources. Some licensing agreements required us to keep the source code private unless the requesting customer had a license agreement directly with the supplier of the technology. Over time, this meant that to get all of the sources to our Unix products, fifteen separate licenses were necessary, at a cost of thousands of dollars, and even then the sources were restricted to a “need to know” basis and were not for consumption by curious students.

A second issue was cost. Schools had been moving towards PCs and Macintosh computers over the years, mostly because of the low cost of the hardware, operating system, and applications. While these machines were fine to do reports on, or to do other types of “application” work, the lack of sources to the operating system, networking, and compilers made them less useful for teaching operating system design. Workstations, on the other hand, tended to use more expensive components, larger amounts of main memory, and were generally outside the price-band of most schools trying to teach computer science to large numbers of students.

As I stood on the deck of the Nachez, several thoughts ran through my mind. I knew that Digital was developing some low-cost Alpha single-board computers which used industry standard buses (PCI and ISA). I also knew that the Alpha (with its 64-bit architecture) had unique capabilities for doing computer science research into large address-space utilization. The Alpha processor was a true

RISC system, which would test the portability of the Linux kernel, and the availability of Linux on Alpha would help develop new concepts for better using the Alpha's blazing speed, currently 1 billion instructions per second (BIPS), even in our commercial Unix product. So I asked Linus if he had ever considered doing a port to the Alpha. “Yes,” he said, “but the Helsinki office of Digital has been having problems locating a system for me, so I may have to do the PowerPC instead.”

My fellow employees tell me that I howled like a wounded hound at that point, and (I find this hard to believe) dropped my Hurricane (a fine New Orleans drink). It was then that I knew I had to help get Linux on Alpha.

The next day I flew back to New Hampshire, and that morning I was on the phone to Bill Jackson, a marketing manager in our Personal Worksystems Group. I explained the situation, and why I felt this was a good thing for Digital. Bill and I had known each other for a long time, and just as I had faith in Kurt Reisler, Bill had faith in me. “maddog,” he said, “I only have a Jensen (a code name for an early Alpha workstation, which had an EISA bus), but it has 96MB of main memory, Ethernet, and 2.5GB of SCSI disk space.” “Throw in a CD-ROM drive and you have a deal,” I said (being a tough negotiator), “my cost center will pay the shipping.”

I still had to really formalize the agreement with Linus. Fortunately he was attending the summer USENIX in Boston, so I took the paperwork loaning him the computer system down to Boston. “How long is the loan?” asked Linus (while munching on a hot dog). “As long as you need it,” I replied, “or until we can get you an even faster system.”

The next week the system was shipped to Helsinki, via the Digital office there.

About the same time I heard about a group of engineers inside of Digital who were also working to port Linux to Alpha. I got them a system identical to the one I had obtained for Linus.

Since Digital was now somewhat in the “Linux market”, I thought it was time to formally write up the value of Linux to Digital, and to give some real thought as to why Digital should support a porting effort. I also had to think of how Linux systems on Alpha might affect the sales of Digital's own product, DEC OSF/1 (since renamed Digital Unix to reflect having been branded as “Unix” by X/Open, Inc.).

I quickly concluded that there were markets for Linux on Alpha, and these markets have some of (but not all) the same characteristics:

They need the source code for:

- changing
- observation
- exchange
- They have more time and manpower than money
- They do not need huge numbers of commercial applications (yet)

I also found a market that would probably *not* want Linux (yet), and these are some of their characteristics:

- They resemble my Mom & Pop (computer illiterate, and proud of it)
- They want some entity to guarantee the operating system
- They are mostly dependent on commercial applications

The markets here are not all black and white. For example, one market that is typically thought of as a “Linux market” is the “computer hobbyist” market. To a lot of commercial computer vendors, this market uses older PCs, cast off from other applications, to fuel a “hobby”, much like the Radio Amateurs of ARRL fame. However, if you really look at this market, you see some of these people buying very sophisticated gear, trying to reach “an edge”. This can be an interesting (but relatively small) market.

Another (much larger) market is the computer science education market. Universities, colleges, and even grade schools teach students to interact with computers, and many teach computer science. With Linux as the operating system, and either PCs or low-cost Alpha processors as the platform, these customers can now actively teach computer science, with access to source code for students to modify and try on their individual systems. Research in computer science (particularly with large address spaces, or with RISC instruction sets) can easily be facilitated with Linux, and the copyleft licenses encourage free exchange of the research results.

The fallout of this is that larger systems (funded, perhaps, by research grants) may also be sold. Or the purchaser of 100 Linux desktop systems might appreciate a server machine to hold the student's files, do the printing, handle mail, etc. It makes sense that the server machine be either of the same architecture as the clients, or at least have data compatibility on the binary level. Intel machines and Alpha machines are both “little endian”, and even if the Intel architecture is only 32 bit, it is relatively easy to make them data compatible.

Another market where Linux would shine is turnkey systems—places where a large number of systems would be purchased, mostly for one application, such

as point-of-sale terminals, or for large user-written applications which have to run on a large number of discrete systems. In this case the savings in operating system license royalty payments might pay for a system programmer to do the integration and support of Linux on system boxes. Also, since there tend to be fewer applications to run on these turnkey systems, it might be easier to stay with one version of Linux, versus having to keep upgrading to newer versions.

Finally, there is another side to the freeware operating system market, and that is what the commercial software developers can gleen from what the freeware people develop. By looking at what features the Linux community puts into their systems, commercial systems can improve by emulating the design decisions made in freeware operating systems. Not every design decision will be followed, but certainly some good ideas have already been seen in Linux, and should be studied by developers of commercial code. We no longer have the luxury of re-inventing the wheel.

I am often asked how Digital Unix (Digital's commercial Unix product) would fit with Linux on Alpha. I see no conflict. Some people want such features as certifiable C2 security, “cluster” style systems (multiple systems working together as a single image), SMP scalable to large numbers of processors, soft realtime support (which I keep asking Linus to put into Linux), large log-based file systems, etc. These features (and more) are all things that Digital Unix has today, and which may or may not show up in Linux in the future, depending on what the Linux developers find “interesting”.

On the other hand, I believe that Digital should continue to work to make Digital Unix more and more compatible with Linux and netBSD (which has also been ported to the Alpha processor, and is available across the network), and provide diskless and dataless support for both Intel and Alpha Linux systems. Already there are some interesting possibilities, since Linux Alpha can run binaries statically linked on Digital Unix systems, and I assume that the same may be true of taking statically linked Linux binaries and running them on Digital Unix systems. By working on binary compatibility between the two operating systems, Digital would help facilitate a large number of applications that would run on both operating systems.

Digital's present plans do not include shipping a CD-ROM created by Digital with Linux “Alphabits” on it. We feel that the current companies and groups are doing a fine job, and we will work with those groups to have the Alphabits put on their distributions. Likewise, we will continue to put our contributions out on the Internet for the Linux community to use. Digital's goal is simply to have the best, easiest, and fastest hardware to host the Linux operating system, whether it be Intel-based PCs or systems using the Alpha 64-bit processor.

Jon “maddog” Hall is the Senior Manager of the Unix Software Group at Digital Equipment Corporation. He has been with Digital for 12 years, all in the Unix group, and for four years before that he was a senior Unix systems administrator at Bell Labs. He can be reached via e-mail as maddog@zk3.dec.com

Archive Index Issue Table of Contents

Advanced search

PracTcl Programming Tips

Stephen Uhler

Issue #18, October 1995

Stephen gives a thorough description of how to use puts and other techniques for tracking down and exterminating bugs in your Tcl scripts.

On occasion, I write Tcl programs that don't work right the first time, and thus need to be “debugged”. The easiest way to debug a Tcl program is with the **puts** command.

```
puts stderr "Some useful information to print"
```

A few carefully placed **puts** statements can be used to ferret out most bugs. Unfortunately, it often seems the bugs have a habit of returning as soon as the **puts** statements are removed.

The solution to the recurring bug problem is to wrap **puts** in a procedure, called **dputs**, so we can turn debug printing on or off without changing the code:

```
proc dputs {args} {
  global Debug
  if {[info exists Debug]} {
     puts stderr $args
  }
}
```

This first version of **dputs** checks to see if the global variable **Debug** is set (to anything) before printing the arguments passed to **dputs**. As a side benefit, **dputs** lets us specify what to print as multiple arguments. The **args** parameter, which is special in Tcl, automatically gathers all of the arguments of **dputs** into a single string.

Although **dputs** is an improvement over **puts**, it doesn't take too long to discover the limitation of this version. You have the choice of either too little output or too much. What we would like to do is turn debug printing on or off selectively, in different sections of the program.

We can use the introspective capabilities of Tcl to determine which procedure each **dputs** is being called from, and turn debug printing on or off for each procedure. We'll use the **info level** command to look into the current procedure stack and figure out the name of the procedure that **dputs** is being called from. We can set **Debug** to a glob-style pattern that will cause only those **dputs** statements in procedures that match that pattern to print. As a bonus, we'll print the calling procedure name as part of our output, so it doesn't have to be included as an argument to **dputs**.

```
proc dputs {args} {
  global Debug
  if {![info exists Debug]} return
  set current [expr [info level] - 1]
  set caller toplevel
  catch {
    set caller [lindex [info level $current] 0]
  }
  if {[string match $Debug $caller]} {
    puts stderr "$caller: $args"
  }
}
```

In this version of **dputs**, as before, if **Debug** is not set, no debugging output is produced. The **info level** command returns the current nesting level of the procedure call stack, the **dputs** procedure. Subtracting one from **$current** is the stack level of **dputs**'s caller. The **info level $current** command returns a list of information about the procedure stack at level **$current**, whose first element is the name of the procedure. If **dputs** is called at the global scope, the call to **info level** will fail (**current** will be -1), thus the **catch** around **info level**, which will leave **$caller** with the pre-initialized value of **toplevel**.

Now that we have the name of the procedure that **dputs** was called from, it is a simple matter for **string match** to compare the procedure name in **$caller** with the pattern in **Debug**, and only emit debugging output for the desired procedures. The pattern in **Debug** can be changed interactively at the command prompt, or automatically under program control.

Although this version of **dputs** is better, it requires the programmer know *in advance* what information to pass as arguments to **dputs** in order for the debug output to help locate the bug. Typically, half the battle of debugging is determining what information needs to be printed to find the bug, and what **dputs** prints is probably not right.

We can easily overcome this limitation by remembering that Tcl is an interpreted language. Instead of simply printing canned values that are passed as arguments to **dputs**, we can stop the program at any **dputs** call and allow the programmer to enter arbitrary Tcl commands to elicit information about the current execution state of the program.

The next procedure, **breakpoint**, may be inserted anywhere in a Tcl program to cause it to stop and allow interactive execution of commands. For example, the Tcl moral equivalent of the C **assert** command is implemented by calling **breakpoint** any time an invalid condition is detected. Alternately, **breakpoint** can be inserted into **dputs** so breakpoints can be turned on or off selectively using the **Debug** variable.

The **breakpoint** procedure implements four build-in commands: **+**, **-**, **?** and **C**. The **+** and **-** commands allow the user to move up and down the call stack. The **?** commands prints out useful information about the current stack frame, and **C** returns from **breakpoint**, resuming execution of the program. Any other command is passed to **uplevel** to be evaluated at the appropriate stack level.

```
proc breakpoint {} {
  set max [expr [info level] - 2]
  set current $max
  show $current
  while {1} {
    puts -nonewline stderr "#$current: "
    gets stdin line
    while {![info complete $line]} {
      puts -nonewline stderr "? "
      append line \n[gets stdin]
    }
    switch -- $line {
      + {if {$current < $max} {show [incr current]}}
      - {if {$current > 0} {show [incr current -1]}}
      C {puts stderr "Resuming execution";return}
      ? {show $current}
      default {
        catch { uplevel #$current $line } result
        puts stderr $result
      }
    }
  }
}
```

The procedure **breakpoint** demonstrates the use of the Tcl commands **info level** and **uplevel** to examine the state of a running Tcl program, and the **info complete** command to read and evaluate Tcl commands entered interactively.

First, **info level** computes the depth of the procedure call stack (in **$max**). We need to subtract two from **info level**, one for the **breakpoint** procedure, and one for **dputs**. We then loop (**while {1}**) getting Tcl commands and running them. The variable **$current** contains the current stack level, which we'll print as part of the prompt to the user.

Getting a Tcl command from the console is a little tricky, as a single command might span multiple input lines. We'll use **info complete**, and **append** commands in the inner **while** loop to gather up enough lines of input to form a complete Tcl command. Once we have the entire command, the **switch** statement selects either one of the built-in commands, or it calls **uplevel** to run the command at the current stack level, which may have been modified previously by **+** or **-** commands. The **catch** around **uplevel** insures that an errant

command typed by the user doesn't terminate the program with an error. We then print the result of the command (or the error message if it failed), and loop back to get the next command from the user.

The built-in commands **+** and **-** are used to change the stack level that the commands we enter will be evaluated in. They simply change the value of **$current**. The **?** command calls **show**, and **C** returns, resuming execution of the program. The procedure **show**, which we'll write next, displays useful information about the current stack level.

```
proc show {current} {
  if {$current > 0} {
    set info [info level $current]
    set proc [lindex $info 0]
    puts stderr "$current: Procedure $proc \
                {[info args $proc]}"
    set index 0
    foreach arg [info args $proc] {
      puts stderr \
           "\t$arg = [lindex $info [incr index]]"
    }
  } else {
    puts stderr "Top level"
  }
}
```

The procedure **show** is a shortcut for printing application-specific information while debugging, since the user could type in the Tcl commands to achieve the same result. This version of **show**, which gets passed the stack level **$current** as an argument, prints the procedure name, its arguments, and their values at the time the procedure was called. In **dputs** we used the first element of **info level $current** as the name of the procedure in stack frame **$current**. The remaining elements contain the values of the arguments passed to the procedure. The call to **info args** returns the names of the arguments, which we pair with their values in **info level $current**, using the variable **index** to step though the list of argument values. Here is some sample output from **show**, taken from a debugging session of **HMtag_img**, part of a Tcl HTML library package.

```
4: Procedure HMtag_img {win param text}
        win = .clone1.text
        param = src=green_ball.gif
        text = text
#4: info vars
var text param win
#4: set var(font)
font:courier:14:medium:r
#4: -
3: Procedure HMrender {win tag not param text}
        win = .clone1.text
        tag = img
        not =
        param = src=green_ball.gif
        text =  This is a good point
#3: C
Resuming Execution
```

In conclusion, we started with a simple **puts** for program debugging, and in less than 50 lines of Tcl code, created a powerful debugging environment that can be easily tailored to meet the debugging needs of most Tcl applications.

Stephen Uhler is a researcher at Sun Microsystems Laboratories, where he works with John Ousterhout improving Tcl and Tk. Stephen is the author of the MGR window system and of PhoneStation, a Tcl-based personal telephony environment. He may be reached via email at Stephen.Uhler@Eng.Sun.COM.

Archive Index Issue Table of Contents

Advanced search

Indexing with Glimpse

Michael K. Johnson

Issue #18, October 1995

Glimpse, a simple but effective indexing package, can help you find particular mail messages in large mail archives and keep track of files in large directories.

Since nearly my entire livelihood is maintained by exchanging electronic mail, my e-mail archives (not including many messages more than a year old) currently use nearly 100MB of my precious hard drive space—enough that I'm starting to consider buying a separate hard drive just for my personal files. In a desperate, somewhat successful, attempt to keep better track of my e-mail archives, I recently installed **exmh**, a graphical mail program based on MH (a powerful but complex mail reader) and Tcl/Tk.

One optional program that exmh can use to help manage e-mail is glimpse. From some or all of your mail files, this program builds an index, which exmh uses to quickly search for any word you want to look up. I can now search all my e-mail archives for a long-lost letter in less than a minute.

But this article isn't about exmh, as useful as it is for MH users like me. It's about glimpse, an excellent program in its own right.

Preparing Glimpse

Unlike the well-known grep program, glimpse does not usually take an argument telling which files to search. Instead, by default, glimpse looks in every file which it has indexed. This means that glimpse requires an index to work.

Perhaps the simplest way to use glimpse is to index all your files and search them all when you are looking for something. To do that, you need to create the index with:

```
glimpseindex $HOME
```

or:

```
glimpseindex ~
```

which will index all your files, keeping the index in your home directory in files whose names start with .glimpse_. These files will usually take up about 2% to 3% of the total space of the of the files in your directory.

If you want to exclude certain files' names from the index, you can add their complete path names or “wildcard” expressions with ***** and **?** characters to the file .glimpse_exclude. All of the .glimpse_* files are documented in the glimpseindex man page.

Since your files probably change from time to time, you will need to update the index occasionally. You can either do this manually, using the same command you used to create the index, or create a “cron job” to do it for you (but scheduling jobs with cron is beyond the scope of this article).

Using Glimpse

Now that you have created an index, you can search through it. The easiest way to do this is to simply type:

```
glimpse word
```

Glimpse searches through the default index (the one in your home directory) and returns output similar to grep's with the file name prepended to each matching line.

Perhaps your search doesn't turn up the file you are looking for; the word might be misspelled in the file. If you want to allow a one-letter spelling mistake, you can instead use:

```
glimpse -1 word
```

Perhaps your search turns up far too many matches. You can limit the matches to only files with names matching a certain pattern with the **-F** flag. To search only in files ending in **.c**, use:

```
glimpse -F '.c$' word
```

The argument following **-F** is a full regular expression, like the search patterns used by grep.

You don't have to index only files in your home directory. The **-H** option specifies a different directory tree to index. The index files are stored in the

specified directory. If you want to index the /usr/doc directory provided with many Linux distributions, log in as root (or another user that can write in the /usr/doc directory) and run:

```
glimpseindex -H /usr/doc
```

and then any user able to read the /usr/doc/.glimpse_index file will be able to search those documents with:

```
glimpse -H /usr/doc word
```

If your searches aren't fast enough, you can trade disk space for time by running glimpseindex with the **-o** flag, to indicate an index that takes up 7% to 8% of the space of the files being indexed and increases search speed somewhat, or the **-b** flag to indicate an index that takes up 20% to 30% extra space and increases search speed more.

If you search all the time, you can speed up your searches by running the glimpseserver program in the background. That is covered in the glimpseserver man page.

What Else?

Glimpse can do more than I can cover here, so if you don't see what you are looking for, try it—or at least read the documentation—before giving up. In particular, glimpse supports the options used by **agrep** (approximate grep), a popular search program written by the authors of glimpse several years ago. agrep and its man page are included in the glimpse distribution. Its options include boolean searches of different kinds.

Glimpse is also the search engine used in the Harvest system, which “is an integrated set of tools to gather, extract, organize, search, cache, and replicate relevant information across the Internet”, according to the Harvest Web site at harvest.cs.colorado.edu.

Getting Glimpse

Glimpse is available at the ftp site ftp.cs.arizona.edu in the file glimpse/glimpse.2.1.src.tar.Z. Linux binaries and sources pre-configured for compiling under Linux (as well as the original sources) are available from www.ssc.com/lj/issue18/glimpse.html.

If you wish to participate in the glimpse@cs.arizona.edu mailing list, send a message to glimpse-request@cs.arizona.edu and ask to be added.

The glimpse manual pages are on line as glimpse.cs.arizona.edu:1994/glimpsehelp.html, and glimpse has its own home page at glimpse.cs.arizona.edu:1994.

Michael K. Johnson is the editor of *Linux Journal*,and wishes he had spare time to spend pretending to be a Linux guru.

Archive Index Issue Table of Contents

Advanced search

World Wide Web Books

Brian Rice

Issue #18, October 1995

We find many books on the market these days to help us do-it-yourself publishers.

- Title: *HTML for Fun and Profit*
- Author: Mary E.S. Morris
- Publisher: SunSoft Press/Prentice Hall, 1995; 264 pp. plus CD-ROM
- ISBN: 0-13-359290-1
- Price: $35.95
- Reviewer: Brian Rice

- *Build a Web Site*
- Author: net.Genesis and Devra Hall
- Publisher: Prima Publishing, 1995; 715 pp.
- ISBN: 0-7615-0064-2
- Price: $34.95

- Title: *The HTML Sourcebook*
- Author: Ian S. Graham
- Publisher: John Wiley and Sons, 1995; 416 pp.
- ISBN: 0-471-11849-4
- Price: $29.95

The first impression newcomers to the Internet and the World Wide Web get is that much net information feels very do-it-yourself. Is that good or bad? Some argue that all this volunteer information erodes the difference between the work of casual amateurs and the truly authoritative. Others contend that making this judgement has always been the responsibility of the reader, regardless of the medium, and the explosion of content on the Internet just makes more voices available to choose among. The Internet's detractors seem

to be in the minority (although that doesn't diminish their arguments), and they can't stop Web serving from being a ton of fun. So we find many books on the market these days to help us do-it-yourself publishers. I recently had the pleasure of working with three.

Of these books, Morris's *HTML for Fun and Profit* is the only one that gives evidence of having been rushed to press (and it still missed its intended publication date by several weeks). The book has many distracting typos as well as a few truly embarrassing errors—for example, it's regrettable that Garrison Keillor has not read *HTML for Fun and Profit*, because it alleges that gopher originated at the University of Michigan. He should be able to get a good monologue out of that, on the topic of "Yet More Indignities Minnesota Suffers —Stoically, of Course."

HTML for Fun and Profit has the feel of a reference manual, with all the pros and cons that might suggest. It has many tables, and you probably won't come away from the book with the feeling that you're missing raw data.... Most of what Web authors and managers use most of the time is covered, with the exception of post-HTML 2.0 extensions. On the other hand, the prose is wooden, which makes for slow reading, and the onslaught of detail after detail creates a trees-obscuring-the-forest effect. Morris's book is the only one of the three that comes with a CD-ROM, which in this case offers 500MB of the same stuff over and over again. The disc's top-level directory structure offers "mac", "WINNT", "sol1", "sol2", and disappointingly slim "docs" and "src" directories. The directories named after architectures contain binaries for the platforms and all the examples from the book, mostly identical among the various architectures. ("sol1" and "sol2" are SunOS and Solaris 2, respectively.) Can ISO 9660 CD-ROMs have hard links? This one doesn't, and they might have made some room for the things we'd like to see on a Web CD-ROM, like maybe giftrans, which the text itself spends more than one page documenting. So what's here for Linux users? Well, all the text examples are here, though most are pretty trivial; source code for the CERN and NCSA servers is here; and there's source for Perl 4. No selection of Web browsers, no pbmplus, no xv, no giftrans. I was disappointed with the CD-ROM, as well as the book as a whole. Neither really connects the reader with the day-to-day experience of running a Web site.

By contrast, *Build a Web Site* is pervaded by experience. You sense on every page that you are being addressed by people who have worked long and hard with what they are discussing. For example, all the books mention that it is a *faux pas* to use "click here" as a hyperlink, but only this book bothers to give credible reasons why. Only in *Build a Web Site* will you find the magic incantation **telnet *hostname* 80**, the canonical way to test whether your Web server is happy. And only *Build a Web Site* explains why you should care about your server logs.

Build a Web Site includes the specifications for HTTP, HTML, and URLs, which the cynical might assume were included so that the book would take up more shelf space than its competitors at Bookstar. But the authors have taken pains to insert cross-references into the specs, a nice gesture, and I found myself using the specs regularly.

One odd omission from *Build a Web Site* is server-parsed HTML, a topic covered very nicely in *The HTML Sourcebook*. This book's focus is narrower than the others'; its intent is primarily to describe the world of Web authoring. It contains a section on Web servers, many of which are non-Unix, but the presentation is broad rather than deep. Interestingly enough, the *Sourcebook* displays a simple C program that pretends to be a Web server and displays what a client is sending. The program feels out of place, but it's useful nevertheless.

The *Sourcebook* also contains voluminous lists of tools, all with URLs for getting them. Since any paper guide to the Internet is going to be out of date the moment it hits the stands, these locations must be taken with a grain of salt, but you can at least use the locations as hints for archie use. Perhaps knowing this, the author included a section on how to use archie—although he assumes that you will be willing and able to install an archie client rather than telnetting to an archie server directly.

The HTML Sourcebook is the only book of the three to emphasize that the Web's native character set is ISO Latin-1, not ASCII. *Build a Web Site* ignores the issue (except in the included HTML spec), and *HTML for Fun and Profit* makes a muddle of the whole issue. *The HTML Sourcebook* is also the only one that mentions Linux.

Another strength of the *Sourcebook* is that it demonstrates by example the importance of previewing your Web content with a range of browsers, and the book includes the archive locations of many. Web authors often forget to do this test before releasing their material, so it's nice to see the word getting out; I wish Graham had been even more explicit. In fact, my favorite single page in any of these books is in *HTML for Fun and Profit*: it shows a Web home page whose authors neglected the possibility that visitors might have chosen to delay the loading of inline images. It's a train wreck, of course, and all the more satisfying because the page belongs to Sun.

Of these books, I recommend *Build a Web Site* for Linux audiences, possibly supplemented by *The HTML Sourcebook* for finding resources. Perhaps *HTML for Fun and Profit* will be improved in future editions. I also recommend that

anyone contemplating Web work learn perl, perhaps before reading these books.

Brian Rice (rice@kcomputing.com) is Member of Technical Staff with K Computing, a nationwide Unix and Internet training firm.

Archive Index Issue Table of Contents

Advanced search

Letters to the Editor

Various

Issue #18, October 1995

Readers sound off.

Power Failures?

Having worked with Linux for several months now, I thought the information below might be of interest. I rarely use my notebook for working with C code, due to the swap partition slowing things down. A 500-line code fragment will take 30 seconds to become an executable on my Fosa 486DX2[hy]66 notebook with 4MB of RAM; it takes six seconds on my 386DX33 desktop with 8MB.

Following are results I obtained testing battery life: stopwatch timed, average of many runs each operating system. The machine was the notebook mentioned above, which has dual scan colour. Batteries were fully charged at start in each case. I included both continuous and periodic runs.

```
OpSys.  Notes                                          Duration
                                                       (min.)
Windows 3.1     Sparse hd use (solitaire, Majong)      81
MS[hy]DOS 6.22  B&W use only, infrequent hd use            82
OS/2 Warp       Misc programs, moderate hd use         115
Linux   B&W & colour C programming, heavy hd (swap) use     121
```

Have other notebook users observed similar results? The nearly 50% increase using OS/2 or Linux is puzzling and requires further investigation.

R. H. Armstrong, hankarm@compumedia.com

Yes—and there is a perfectly good reason. Linux and OS/2 use the “halt” instruction to stop the CPU when they are idle; DOS and Windows 3.1 do not. This also has an effect on temperature; CPUs running Linux and OS/2 are much, much cooler than those running other operating systems. A few months ago, someone connected a thermocouple to his CPU and ran a bunch of tests and reported on the results on the newsgroups, his tests showed that DOS and Windows ran the CPU very hot, and Linux and OS/2 ran the CPU much cooler.

More Feedback

Although there have been some improvements with regard to the LLS (Linked List Syndrome), I think there is still room for improvement (beside never splitting articles at all).

Personally I find footers and headers like “continued on next/from previous page” irritating. What kind of audience is *LJ* aiming at? Brainless Windoze 96 users? I have enough brain capacities to turn the page without instruction. I can even skip pages that contains ads without instruction; it is my default behavior. May I suggest you only use those “continued” lines when you skip over parts of other articles?

Another disadvantage of the LLS is the need to backtrack the origin of the current article when you want to continue with the next article. When backtracking is necessary to find the next article, could you print the page number of the next article at the end of articles?

Hans de Vreught, J.P.M.deVreught@cp.tn.tudelft.nl

We have taken this suggestion and discontinued the continuation lines when an article jumps over ads. Alert readers will note that you can always tell when an article ends by looking for the “*LJ*” symbol.

Although we'd like to believe that all our readers read *Linux Journal* straight through from cover to cover, we suspect that it's not true. So we leave it up to our readers to use the table of contents to find the articles they want to read.

There is a thing that I don't understand with the *LJ*. Why do you split so many of the articles into pieces? In *LJ*16, “perl” was continued on page 51 and 52. Why not on page 44 and 45? For my feeling, this is rather confusing and I wouldn't like the books at my library sorted like this.

Otherwise I'm glad that there is the *LJ*.

Micha Jung, jungm@sip.medizin.uni-ulm.de

When laying out *Linux Journal*, first we put color items (ads and illustrations) on the color pages, then we fit everything else around that. We try to avoid jumps whenever possible, but sometimes, because of ad placement and so forth, articles have to jump. We try to keep code examples together and we generally don't have more than one jump per article, not counting pages with just ads. Every month it's like working out a puzzle to fit everything. Hopefully, this clears up the reason for jumps.

Linux *is* Real!

Many projects at work have screamed for a Linux solution. Whenever I mentioned it, I would get some interesting looks, and then the subject would change. Most projects would get done by coercing Novell to work, or worse, use a dedicated PC to do certain jobs.

Finally, I presented a Linux solution. Part of my presentation was to have a few *Linux Journal*s with me. Suddenly my audience realized how "real" Linux was. It really helped to have a professional publication to drive the point that this is not just a bunch of people playing around on the Internet.

So, now I have a project on my hands! Thank you for your help in doing a project the right way for a change. I look forward to future issues.

Chris Sullivan, slowhand@usa1.com

Subscription Price and Paper

After the announcement of our recent subscription price increase, the following e[hy]mail dialog took place between a reader and *Linux Journal* publisher Phil Hughes. We'd like to hear other readers' thoughts on this issue.

Phil (in c.o.l.announce): We want to continue to make *LJ* affordable but it has to be affordable for us to print as well. We settled on a $3 increase for 1-year subscriptions and a $5 increase on 2-year subscriptions. If paper prices and postage prices remain fairly stable, we can live with that and hope that you can too. Oh, if you happen to own a forest suitable for making into paper and want to support the Linux community, please contact me.

Charles A. Stickelman (stick@richnet.net): Have you looked into stock made from things other than trees? There appear to be several alternatives to wood pulp; straw and hemp both make a great substitute. Hemp does not need the harsh chemicals to process, and is therefore much less acidic than wood pulp. One benefit of this reduced acidity is that hemp[hy]based paper is much more stable than wood[hy]based; it doesn't turn yellow and deteriorate. This is good for reference materials (like *LJ*) that may need to be around awhile. There are also other economic/ecologic concerns that hemp solves. This has nothing to do with the recreational uses/abuses of hemp by-products.

Phil: Yes, we have. We have used hemp/straw copier paper but had a problem with curl. But hemp[hy]based paper is just not available for the type of press magazines are printed on, and is even more expensive than tree[hy]based paper for other types of printing. To me this is a political issue and I am on the side of getting hemp recognized as a good alternative, which should

substantially reduce the price. *Linux Journal* spends tens of thousands of dollars on printing each month, which sounds like big money to me, but it isn't to the paper industry.

I am more socially conscious than most in this business (or any business). The problem is that it is hard to tell potential subscribers that our products cost more than the competition because we use tree[hy]free, acid[hy]free paper and soy inks. We do what we can and hopefully the environmentally responsible solution will become the low[hy]cost solution. For example, we now use copier paper made from old telephone books that (finally) costs less than non[hy]recycled. [We have just been notified that this paper has been discontinued, sadly—ED]

Charles: I've been a subscriber of *LJ* since issue #1, and I love it! It's very well done; easy to read and quite educational. For the record, I'd be willing to pay an extra $5/yr (on top of you $3/$5 increase) to have *LJ* use soy-based ink and hemp/straw based paper.

Readers, what do you think? Would you be willing to pay more to get *Linux Journal* and other publications on an environmentally-friendly, more durable paper stock (if such paper becomes available)?

Archive Index Issue Table of Contents

Advanced search

Stop the Presses

Phil Hughes

Issue #18, October 1995

In a lot of ways Windows 95 is like Linux; in a lot of ways, it isn't.

Looks like I volunteered (read: was volunteered) to write the Windows 95 article for *Linux Journal*. Seems like we have to do it—every other computer magazine did. I suggested to our Editor that it could be humorous or serious. He told me it didn't matter, I would get flamed either way. So, here goes—you decide if it is humorous or serious.

I am writing this column in early August, just a couple of weeks before the scheduled release of Windows 95. In case you haven't been on this planet for the last year or so, Windows 95 is a product of a small company called Microsoft located about 10 miles east of the *Linux Journal* offices. In a lot of ways Windows 95 is like Linux; in a lot of ways, it isn't. Let's look at a few of the similarities.

Win95 and Linux both run on Intel-based PC hardware. (Linux also runs on other hardware such at the Alpha and Amiga, but that is beside the point.)

Win95 and Linux are both 32-bit operating systems. (Linux on the Alpha is 64-bit, but that isn't important. And when *WINDOWS Magazine* Editor-at-Large John Ruley interviewed Georg Moore, a Microsoft program manager, he was told that Win95 is really a hybrid, with lots of 16-bit components remaining.)

Win95 and Linux both run user applications in protected mode. (Except that Win95 maps memory from 64KB to 4MB into the address space of all applications, with write access to all data areas between 64KB and 4MB. Again, from John Ruley's interview.)

Win95 and Linux both include networking. (If the U.S. government allows this in Win95. And, of course, Linux includes NFS, NIS, uucp, ... In other words, a lot

more networking with an open architecture, which makes it easy to network Linux with other operating systems.)

Win95 and Linux both include a GUI. (Of course, the Win95 GUI only runs on whatever Win95 runs on. The Linux GUI, X-Windows, is available on many platforms.)

Win95 and Linux are both written in C. (Well, that's what I have heard. While Linux source code is freely available, Win95 source is not.)

Win95 and Linux are both licensed software products. (The licenses do differ: the Win95 license says you can't share, the Linux license says you must share. Also, if you didn't see it before, take a look at Linus Torvald's "Linux '95 Final Release", a spoof on Microsoft's license that he posted to Usenet in March (on our Web site at www.ssc.com/lj/issue18/final.html).

Enough for the similarities.

Win95 includes the "Registration Wizard" which allows automatic, on-line registration of the software. Why, even consumer advocate Ralph Nader has noticed this. In fact, in a letter to US President Bill Clinton he said "Another objectional feature of Windows 95 is the Microsoft online `Registration Wizard'. This part of the program is designed to scan automatically a user's hard disk, dial-up Microsoft, and download information to Microsoft about the files on the user's hard disk, including the titles and versions of software applications. Critics of this practice, including the Department of Defense, have questioned the impact of this practice on data security and privacy." While this registration is optional—you have to click OK to enable this—one wrong mouse click can compromise the privacy of your system.

Enough of the rumors.

Let me look at the serious part of what Win95 means for Linux users.

The inclusion of networking in Win95 means that the Internet will grow faster than ever before. The World Wide Web, already experiencing fantastic growth, will grow even faster. (I guess this is why we are starting a magazine called *WEBsmith* for Web developers.) CompuServe figured this out and bought Spry, a company that sells Web sites.

While Win95 shows that Microsoft is finally getting serious about connectivity, Linux was born on the Internet and has a lot of experience working with diverse systems. *PC Week* started its life with Web servers on Linux boxes and today even Spry uses Linux for its Web servers. If 100,000,000 people eventually get

on the Internet with Win95 boxes, the Internet probably will need another 1,000,000 Web sites to handle the traffic and offer new services. Many of these systems could be and will be Linux systems.

Microsoft will sell tens of millions of copies of Win95 to people who want a desktop workstation. Companies such as Caldera expect to sell Linux with their desktop software to a similar market. The “features” of Win95 I listed above will help many companies and individuals choose Linux instead of Win95. If my 100 million estimate is right and 1% pick Linux instead, that's another 1 million Linux systems.

Another use for Linux systems is vertical applications. A dentist doesn't care what kind of operating system her computer runs. She wants a reliable, cost-effective system that takes care of appointments and billing. With Linux's low cost (free software and low system requirements), it is a fierce, though quiet, competitor in this market.

While Microsoft will sell tens of millions (9 million sales are projected for the remainder of 1995 alone), and maybe eventually hundreds of millions, of copies of Win95, the number of uses for computers is growing. Rather than try to capture the whole Win95 market (which we can't afford to do), Linux can fill plenty of niches far better than Win95.

Finally, a tip for Linux users who wish to try Win95. A bug in the Win95 installer code causes it to destroy the master boot record of your hard disk. This has been reported to Microsoft, but there has been no response. Linux users who install Win95 must then reinstall LILO or whatever other boot manager you are using. While I haven't tried it, I have been told that LILO works fine and will boot Win95.

Readers' Choice Award

Be sure to vote for *Linux Journal*'s Readers' Choice Award. Don't wait. The deadline is October 6, 1995.

Archive Index Issue Table of Contents

Advanced search

Serendipity

Dean Oisbiod

Issue #18, October 1995

This month Dean shares some of the fortuitous discoveries he's made in his journey through the Infomagic Developer's kit, including ports and clones of DOS programs and some favorite utilities.

The four-CD Infomagic Developer's Kit, which, as you know if you've been following my column, has become my choice for getting Linux, holds a lot of code. In browsing through the disks, I've looked at spreadsheets, text editors, games, and all sorts of interesting programs and packages. This month I'll focus on software that should look very familiar to those of you visiting from the MS-DOS world.

In a previous article I mentioned a WordPerfect demo which demonstrated that some very powerful and popular packages do make their way over from DOS. Though I had problems getting the demo to work, that shouldn't reflect poorly on the commercial product. It shows that Linux is being seriously considered as a valid alternative to the DOS/Windows OS mindset. [That demo is the SCO version. Easier to use will be the native Linux version that is currently being ported by Caldera and is expected to be released sometime 4Q95—ED]

The usual caveat holds. Many of the programs undergo frequent revisions. Some are shareware or freeware versions of commercial programs. Others are just as-is.

Unlike their DOS counterparts, many of these programs include the source code. Some have originated in DOS; others I mention because of their tremendous utility or interest, hence the title of “Serendipity” for this month.

Also, I should mention a change in my computer system since my previous column. I still have a 486/66, but it's now loaded with 16MB of RAM. I don't use a swap file as much, but when it's needed I'll mention it; with 16MB and no swap file, X-Windows runs happier and much faster.

Diving into the CDs...

Utilities

mc, or Midnight Commander, which I have written about repeatedly, is a clone of Norton Commander and an absolute godsend. This program removes many of the hassles of copying and unarchiving. It also reduces your chances of making bonehead mistakes by not allowing you to quickly delete things recursively (something, I have been told by experienced hackers, particularly gut wrenching). To delete recursively, that is, to delete a directory and the files and subdirectories under it, mc requires you to wade through two menus and then type “yes” to a prompt for deletion.

Unlike the DOS or Windows Norton Commander, mc offers some truly unusual choices such as those options found via the **f2** key; your choices change depending on whether the cursor is on a directory or a file. If you're on a directory when you press **f2**, you gain the option of creating a tar file of the contents. If you're on a file, the option changes to include dumping the contents or displaying the file with **roff -man**. You also have options to edit a bug report and mail it to root (which didn't work for me, probably because (1) I am root, (2) I don't have a network and haven't set up mail, and (3) I don't know any better.) Another option which I both like and dislike is the information hypertext browser. I like it because it provides a lot of useful information; I dislike it because it uses Emacs (or something close to it), and I still haven't mastered many of the commands. mc also offers a useful search capability that will look for most anything anywhere. mc is a must have.

pkgtool, **installpkg**, **explodepkg**, **removepkg**, and **makepkg** are part of the Slackware setup and are a great set of utilities for file handling. Normally, to install a .gz file I would use mc but I always hated that if I wanted to uninstall the files I'd have to list the tar, remember or write down the files, manually search them out and delete them. installpkg takes care of that by building a script that records where the files went to. When it's time to delete, removepkg reads that script and does the dirty work of deletion. Even better, it won't delete a file if it's in use by some other program. This is a great system for novices. pkgtool is sort of a shell, but I prefer to work directly with the sub-utilities; it temporarily raises my “hacker” factor. makepkg does like it sounds—it makes packages. explodepkg is very similar to installpkg but it doesn't create or affect scripts. It and installpkg work on Slackware compatible and (tar+gzip) packages.

minicom should look quite familiar to you if you have used the DOS versions of the Procomm, Telix, or QModem communications programs. The **alt** commands are mostly here but where, for example, in Telix you'd choose **alt**-o for the options screen, in minicom the sequence is **alt**-a-o. **alt**-a precedes most

of your choices. It took me a very short time to get used to this arrangement. minicom doesn't have as many options as the DOS counterparts. For example, you only get two terminal emulations—VT102 and ANSI—and four protocols for sending/receiving files: Zmodem, Ymodem, Xmodem, and kermit, but the options are enough. This program works well, initiating Zmodem for downloading without hitch, allowing a variety of configuration options, and even including my favorite two options: text capturing (**alt**-a-l) and screen scroll back(**alt**-a-b)---but still could use some improvements, such as adding more options to the somewhat limited dialing directory. I hope that the authors continue to support and add to this program; I like it and greatly prefer it to the X-Windows program **seyon**.

Two small utilities that are neither ports from DOS nor clones but are still quite useful are **dos2unix** and **unix2dos**. These programs just convert text files to and from Unix and DOS formats, which appears consist of removing or adding carriage return characters as needed. Simple and neat.

Miscellaneous

xfractint---this is a port from MS-DOS of what I consider the greatest fractal exploration program ever written. It's free, fast, and fun. Note the “x” starting the name. You'll need X-Windows to run it; it may already be a menu option if you have Slackware—look under “Applications”. I found the program to be essentially the same as the MS-DOS version, with minor differences. Pressing **del** does not choose the video mode as it does in DOS. Also, the color cycling commands (**+**, **-**) worked but the other cycling option, **c**, would occasionally freeze the program. Yet these are minor nitpicks—the fun is playing with the variety of fractal types and the parameters that go with each. My personal favorites are color cycling “plasma” and “dynamic”, which give me flashbacks to many of the Grateful Dead concerts that I missed. As an additional fractal exploration tool, you can develop your own fractal formulas—but I'll let you figure out how.

pov-ray stands for “Persistence of Vision—ray caster”, and, like fractint, it's another great program, with an enthusiastic following. To get pov-ray started requires untarring a few files, which for me meant everything but the source code. My usual complaints about unclear documentation hold for pov-ray; the directions did not quite match reality. The instructions suggest creating a pov-ray directory but the files automatically create /pov when untarred. The instructions suggest you add:

```
setenv POVRAYOPT -l$HOME/povray/include
```

to .cshrc except that, I figure, /povray should read /pov to match the untarred setup. Also, I couldn't find a .cshrc but I did find /etc/csh.cshrc, which I copied

and modified accordingly. From what I can figure, csh.cshrc is the configuration file for the C shell. I also noticed three binaries of pov-ray. Two, povray.s and povray.v, looked exactly the same, or at least were the same size. (I did a file compare and they are different). povray.x was slightly larger. I figured I'd use povray.s.

pov-ray creates Targa graphics files. You'll need to view them and a viewer is not included. The manual suggested either **xv** or **xli**. I installed both. xv was on Slackware's XAP disk (note that this program requires libgr, also on the XAP disk, also be installed) and I finally found xli on the Sunsite archive CD-ROM buried in something like /X11/apps/graphics/viewers.

Not realizing that to activate changes to login scripts all I'd have to do is logout and login again, I completely shutdown the system and rebooted.

Returned to DOS to play a round of Terminal Velocity. After again having vital parts of my ship blown to pixel dust, I booted back to Linux to try pov-ray. The viewers sat waiting. csh.cshrc was now activated. I copied a sample file into /pov/bin and typed:

```
povray.s -w320 -h200 -ichess.pov -otest.tga \
  +ft -a -dG -v
```

Nothing happened. pov-ray couldn't find the include files supposedly pointed to by .cshrc. The familiar stomach cramps started. I didn't want to RTFM, not 250KB worth of text. Instead I browsed through the various Linux HOW-TO files on the main InfoMagic CD looking for information on shells scripts and login initialization. For the heck of it I modified /etc/csh.login but pov-ray again couldn't find the include subdirectory. Of course: I'm not using the C shell! I use bash. Modifying /etc/profile yielded a similar lack of success. What is going on? Finally I tried the obvious: copying the include files into /pov/bin. Yes, that worked! Sure, it's not as elegant as fiddling with the various configuration files —but this solution worked, and worked easily.

And yet pov-ray wouldn't work completely. It could find the include files but it couldn't start the graphics display. Oh yeah, I had to run it through X-Windows. (I may not have *had* to do it in X-Windows, but the command **-dG** tells it to display the picture as it raycasts. When it does so it expects X-Windows.) For about 10 minutes I saw a picture of a chess set develop line by line. While waiting for the picture to finish I read the pov-ray info that it had belched before casting and noted that the binary was compiled with 386s in mind. Later, in reading the instructions, I found the suggestion for recompiling for 486s to speed things up. Now to see the Targa output. I called up xv from my X-Windows applications menu, loaded the file, and saw a chess set. Beautiful! And xv had some options—blur, sharpen, oil paint, spread, and de-speckle, to name

a few—that induced more flashbacks. But I wanted to try the other viewer as well. Via a shell I called up xli. I expected a graphics interface; I got a command line reprimand to specify options. Typing **xli -help** showed that the program offers a tremendous number of options and can manage darn near any type of image, including many I had never heard of, as well as many very familiar in the DOS world. But my lack of familiarity with many of the terms, combined with the command line interface, soured me on xli. I did get the chess set image loaded, but I got no farther, and went back to xv to play some more.

Dean Oisboid, owner of Garlic Software, is a database consultant, Unix beginner, and avowed chocolate addict.

Archive Index Issue Table of Contents

Advanced search

New Products

LJ Staff

Issue #18, October 1995

SmartWare Plus, WM_MOTIF Lite and more.

SmartWare Plus

ANGOSS Software International released SmartWare Plus, an application and systems management tool set and integrated business automation suite for Linux. The SmartWare Plus business suite contains a relational database, spreadsheet, and word processor, and can run concurrently in multi-platform environments in either graphics or text mode. The application development tools include development tracking, from prototype through development, application access security, documentation, prototyping, and auditing. Price: US$50.00.

Contact in North America: ANGOSS Software International 430 King St. West, Suite 201 Toronto Canada, M5V 1L5, phone: 416-593-1122, fax: 416-593-5077, e-mail: sales@angoss.com, Web: www.angoss.com.

Contact in Europe: ANGOSS Software UK The Technology Centre, The Surrey Research Park, Guildford Surrey, GU2 5YH, phone: +44-1483-452-303, fax: +44-1483-453-303.

WM_MOTIF Lite

Software UNO announced the release of an entry-level version of its WM-MOTIF library for porting MS-Windows applications to Linux. WM-MOTIF Lite supports the user interface functions of the MS-Windows API on Linux, and includes a full set of Windows compatible controls and a resource compiler to compile Windows scripts with minimal changes. Available via FTP from ftp.uu.net in /vendor/uno/wmmlite/linux.zip.

Contact: Software UNO, Ltd., phone: 800-840-8649, e-mail: info@uno.com, fax: 809-722-6242.

SuperScheduler

SuperSolutions Corp introduced two tools providing job scheduling and spooling for Linux. SuperScheduler allows system administrators and programmers to create jobs for running any number of programs or reports at predetermined times via event scheduling. SuperSpooler is a set of programs that allow system administrators to set up queues to run commands on local or remote systems, allowing effective distribution of network load. Command line access is available from remote sites.

Contact: SuperSolutions Corp. 722 North First Street Suite 143, Minneapolis MN, 55401 USA, phone: 612-340-9212, fax: 612-332-8411, e-mail: steve.mcdermott@supersolution.com, Web: www.supersolution.com.

CYCLOM-Ye/PCI

Cyclades Corp released its new PCI RISC-based multiserial board, the Cyclom-Ye/PCI. Using the Cirrus Logic CD1400 RISC multiprocessors, the board will support serial speeds up to 115.2KBps on all channels, as well as full modem signals. Available in 8, 16, 24, or 32 ports per PCI slot with DB25 or RJ45 connectors. Price: (16 port) US$1079.99; limited time offer US$499.99.

Contact: Cyclades Corp. 44140 Old Warm Springs Blvd., Fremont, CA 94538 USA, phone: 800-347-6601, fax: 510-770-0355, e-mail: cyclades@netcom.com.

Abraxas PCYACC/HYPERTEXT 2000 Language Toolkit and PCYACC/MIL-LANG 6.0

Abraxas is shipping the PCYACC/HYPERTEXT 2000 package. It includes language engines for ASN.1, HTML, RTF, PostScript, HyperTalk, and SGML, and provides significant error processing features for building hypertext systems.

They have also developed a set of tools to assist in developing military language systems. PCYACC/MIL-LANG 6.0 provides mechanisms to quickly integrate ADA, VHDL, and FORTRAN-90 language systems into defense products, and allow quick changes to the behavior of military equipment that is currently hardwired for specific behavior.

Contact: Abraxas Software, Inc. 5530 S.W. Kelly Ave., Portland OR 97201 USA, phone: 503-244-5253, fax: 503-244-8375, e-mail: abraxas@ortel.org.

XForms V0.75

Xforms is a graphical user interface toolkit and builder based on Xlib for the X Window System. XForms is a portable C library, and comes with a collection of objects (buttons, sliders, and menus). Xforms comes with (among other things), source code for more than 50 demo programs, 200 pages of documentation,

precompiled library and header files, and fdesign, an interactive GUI builder that can be used to design dialogues in a WYSIWYG way and output the corrosponding source code. Available from bragg.phys.uwm.edu/xforms, ftp://bloch.phys.uwm.edu/pub/xforms and others. Price: Free for non-commercial use.

Contact: T.C Zhao and Mark Overmars, e-mail: zhao@csd.uwm.edu, Web: bragg.phys.uwm.edu/xforms.

XBasic

XBasic is a program development environment that includes an editor, compiler, debugger, function libraries and a GUI designer for 32/64 bit applications. XBasic was developed from a clean slate, and so avoids implementation dependencies that other languages can not avoid. 32/64 bit from the ground up. Price: Standard Edition US$149/$298, Professional Edition US$249/$498, Professional LAN Edition US$999.

Contact: Basmark Corp. PO Box 40450, Cleveland OH 44140 USA, phone: 216-871-8855, fax: 216-871-9011, e-mail: bmark@ios.com.

The Official Red Hat Linux—`95 Edition

Pacific HiTech and Red Hat Software, Inc, announced the release of The Official Red Hat Linux on CD-ROM. The two CD-ROM set includes DOS shell scripts which allow easier installation and configuration. The latest Linux kernels are on the CD, along with older, and for some purposes, more stable kernels. The packaging system, Red Hat's, RPP(R), allows for point-and-click system installation management. The release contains on-line documentation for installing, using, and managing the Linux operating system. US$39.95.

Contact: Pacific HiTech at 800-765-8369, 801-261-1024, fax 801-261-0310, e-mail: info@pht.com Web: www.pht.com/.

Contact: Red Hat Software, Inc. at 203-454-5500, Web: www.redhat.com.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Porting Linux to the DEC Alpha: Infrastructure

Jim Paradis

Issue #18, October 1995

In the first three parts, Jim describes his early work on Digital's “proof of concept” port of Linux to the Alpha, setting up an infrastructure.

Porting an operating system is not trivial. Operating systems are large, complex, asynchronous software systems whose behavior is not always deterministic. In addition, there are numerous development tools, such as compilers, debuggers, and libraries, that programmers generally take for granted but which are not present at the start of the porting project. The porting team must implement these tools and other pieces of infrastructure before the porting work itself can begin.

This article is the first of three describing one such porting effort by a small team of programmers at Digital Equipment Corporation. Our goal was to port the Linux operating system to the Digital Alpha family of microprocessors. These articles concentrate on the initial proof-of-concept port that we did. Although much of our early work has been superseded by Linus Torvalds' own portability work for 1.2, our tale vividly illustrates the type and scale of the tasks involved in an operating system port.

What Got Us into All This?

The article by Jon Hall on page 29 describes many of the business-case justifications for our involvement in the Linux porting effort. I will describe the actual events that led to my starting work on the Linux port.

First, some background: I work for the Alpha Migration Tools Group, which is an engineering development group within Digital Semiconductor. We were initially chartered near the beginning of the Alpha project to develop automated methods for migrating Digital customers' legacy applications to Alpha-based systems. Our first product was VEST, which translated VAX/VMS binary executables into binaries that could be executed on OpenVMS Alpha. This was

soon followed by MX, which translated MIPS Ultrix executables into executables that run on Alpha systems under Digital Unix. Since then, our charter has expanded into other areas of "enabling technology" (technology which enables users to move to Alpha). In addition to producing translators and emulators, we have supplied technology to third-party vendors, and we have participated in the development of compilers and assemblers for Alpha.

Our involvement in Linux began at the end of 1993, when we realized that there was no entry-level operating system for Alpha-based systems. While OpenVMS, Digital Unix, and Windows NT were all solid, powerful operating systems in their own right, they were too resource-hungry to run on bare-bones system configurations. In many cases, the smallest *usable* configuration of a particular system costs at least several thousand US dollars more than the smallest *possible* configuration. We decided that to compete on the low end with PC-clone systems, we needed to make the lowest-priced system configurations usable. After investigating various alternatives, we decided that Linux had the best combination of price (free), performance (excellent), and support (thousands of eager and competent hackers worldwide, with third-party commercial support starting to appear as well).

Project Goals

When putting together the proposal to do the port, I set forth the following goals for the Linux/Alpha project:

- Price: Linux/Alpha would continue to be free software. All code developed by Digital for Linux/Alpha would be distributed free of charge according to the GNU General Public License. In addition, all tools used to build Linux/Alpha would also be free.
- Resource stinginess: Linux/Alpha would be able to run on base configurations of PC-class Alpha systems. My goal was to be able to run in text mode in 8MB of memory and with X-Windows in 16MB. In addition, a completely functional Linux/Alpha system should be able to fit, with room to spare, on a 340MB hard disk.
- Performance: Linux/Alpha's performance should be comparable to Digital Unix.
- Compatibility: Linux/Alpha should be source-code-compatible with existing Linux applications.
- Schedule: We wanted to be able to show a working port as quickly as possible.

Design Decisions

The above criteria drove several of the design decisions we made regarding Linux/Alpha. To meet the schedule criterion, we decided to “freeze” our initial code base at the Linux 1.0 level and work from there, not incorporating later changes unless we needed a bug fix. This would minimize perturbations to the code stream (a necessity when you're reaching in and changing virtually the whole universe), and would eliminate the schedule drain of constantly catching up to the latest release. We reasoned that once we got a working kernel, we could then make use of what we had learned to catch up to the most current version.

The scheduling criterion also drove our decision to make our initial port a 32-bit (as opposed to a 64-bit) implementation. The major difference between the two involves the C programming model used. Intel Linux uses a “32-bit” model where ints, longs, and pointers are all 32 bits. Digital Unix uses a “64-bit” model where ints are still 32 bits while longs and pointers are 64 bits. At Digital, we have encountered a lot of C code that treats ints, longs, and pointers interchangeably. Code like this might fortuitously work in a 32-bit programming model, but it may produce incorrect results in a 64-bit model. We decided to do a 32-bit initial port so as to minimize the number of such problems. We felt that limiting longs and pointers to 32 bits would not unduly hamper any existing code and by the time new applications appeared which would require larger datatypes, a 64-bit Linux implementation would be available.

We also decided, in the interests of expediency, to use the existing PALcode support for Digital Unix rather than write our own. The Digital Unix PALcode was reasonably well-suited to other Unix implementations, it was readily available, and it had already been extremely well-tested. Using the Digital Unix PALcode in turn required that we use the “SRM” console firmware. The SRM firmware contained device drivers that could be used by Linux via callback functions. While these console callback drivers were extremely slow and had to be run with all interrupts turned off, they did allow us to concentrate on other areas of the Linux port and defer the work on device drivers.

Some design decisions were driven by differences in execution environment between Intel and Alpha. On Intel, the kernel virtual memory space is mapped one to one with system physical memory space. Because of the potential collision with user virtual memory, Intel Linux uses segment registers to keep the address spaces separate. In kernel mode, the CS, DS, and SS segments point to kernel virtual memory space, while the FS segment points to user virtual memory space. This is why there are routines in the kernel such as **put_fs_byte()**, **put_fs_word()**, **put_fs_long()**, etc; this is how data is transferred between kernel space and user space on Intel Linux implementations.

Since Alpha does not have segmentation, we needed to use some other mechanism to ensure that user and kernel address spaces did not collide. One way would be to have only one address space mapped at a time. This requires a translation buffer (sometimes called a translation lookaside buffer, or TLB), a special cache on the CPU used to considerably speed up virtual memory address lookups. But this makes data transfer between user and kernel space cumbersome. It can also exact a performance penalty; on systems that do not implement address space identifiers, using the same virtual address range for kernel space and user space requires that the entire translation buffer be invalidated for that range for every transition between user and kernel space. This could conceivably cause multiple translation buffer misses across every system call, timer tick, or device interrupt.

The other way to avoid address space collisions between user and kernel is to *partition* the address space, assigning specified address ranges to specified purposes. This is the approach taken for the 32-bit Linux/Alpha port. It is simple, it does not require wholesale translation buffer invalidation for every entry to kernel mode, and it makes data transfer between user and kernel an utterly trivial copy.

Designing the address space layout required attention to certain other constraints. First, no address could be greater than 0x7fffffff, because of Alpha's treatment of 32-bit quantities in 64-bit registers. When one issues an LDL (Load Long) instruction, the 32-bit quantity that is loaded is sign-extended into the 64-bit register. Therefore, loading the address 0x81234560 into R0 would result in R0 containing 0xffffffff81234560. Attempting to dereference this pointer would result in a memory fault. There are techniques for double-mapping such problematic addresses, but we decided that we did not need the additional complications for a proof-of-concept port. Therefore, we simply limited virtual addresses to 31 bits.

The other consideration was that we needed an area which was mapped one for one with system physical memory. We did not want to simply use the low 256MB (for instance) because we wanted to be able to place user programs in low addresses, so we chose an area of high memory for this purpose and made the physical address equal the virtual address minus a constant. This is referred to below as the “mini-KSEG”.

Once all the constraints were considered, we ended up with a system virtual memory layout as follows:

```
0x00000000--0x3fffffff        User
0x40000000--0x5fffffff        Unused
0x60000000--0x6fffffff        Kernel VM
0x70000000--0x7bffffff        mini-KSEG (1:1 with physical memory)
0x7c000000--0x7fffffff        Kernel code, data, stack
```

Finally, I had to decide how heavily I would modify the code base to accomplish the port. I felt that I did not have the latitude to make wholesale changes and rearrangements of the code the way Linus did for the 1.1.x to 1.2.x transition. To do so would cause my code to diverge further and further from the mainstream code base, which would adversely affect its acceptance among the Linux community.

I decided to keep the original Intel code 100% intact, so one could conceivably still build an Intel kernel from my code base. The Alpha code would be either additions to or replacements for the Intel code base. Areas that needed to be changed would be set off via conditional compilation. Sometimes this required me to swallow my pride and devise a less clean Alpha-specific version of an algorithm to correspond to a less clean Intel-specific version when I really would rather have implemented a clean, generalized algorithm that could accommodate both. Fortunately, Linus implemented clean, generalized algorithms for all of us when he did his portability work for Linux 1.1.x and Linux 1.2.x.

The Compiler Suite

My initial experiments in compiling some of the Linux code on an Alpha system used the Digital Unix compiler and tools. While this was successful and allowed me to do some early prototyping work, the freeware criterion required that we build the kernel using a freeware compiler and tool set. The GNU C compiler the obvious choice; it is used by Intel Linux, and no other freeware compiler comes close to its functionality and sophistication.

Although gcc can be configured to generate Alpha code, the version available from FSF only understands the 64-bit Digital Unix programming model. While I know a few things about compilers, I'm no expert. I attempted to modify the gcc machine description files for Alpha to generate 32-bit pointers and longs, with disastrous results. It turns out that small changes to machine descriptions can have far-reaching consequences, and I had neither the time nor the inclination to stare at the machine description until I achieved enlightenment.

Fortunately, I did not have to. Another project at Digital had paid Cygnus Support to produce a version of gcc that can generate 32-bit Alpha code, and I was able to use this for my Linux work. The first version that I had implemented only a 32-bit programming model; 64-bit quantities were not available for computation. This drove certain early design decisions. Later on, 64-bit “long long” and “double” datatypes were added, which allowed me to revisit and simplify a number of areas where I needed 64-bit quantities for machine and PALcode interfacing.

I built the compiler and tool suite as cross-compilers on both Digital Unix and Intel Linux and tested both extensively. I did quite a bit of development work both at the office on various Digital Unix systems, and at home on my personal 486-based Linux system.

The Simulator

When I began the Linux/Alpha project, I decided to do my initial debugging not on Alpha hardware but on an Alpha instruction simulator. The simulator, called "ISP", provides much greater control over instruction execution than I could get in hardware. It also provided some support functionality that I would otherwise have to add to the kernel (for example, in ISP I could set and catch breakpoints without needing a breakpoint handler in the kernel). In addition, since I had the source code for ISP I could insert custom code to trap strange conditions as needed.

The version of ISP that I was using included its own versions of the SRM console and Digital Unix PALcode, so I was able to debug my interfaces to the console and PALcode with reasonable assurance that the same code should work unmodified on the real hardware.

While ISP is extremely slow compared with real Alpha hardware, its performance was acceptable for initial debug. In fact, on a 486DX2/66 Linux system, ISP was able to boot up to the shell prompt in under three minutes.

The Bootloader

Once the cross-development and execution environment was in place, I started working on a bootstrap loader. The design of the bootstrap loader was dictated in part by the mechanics of booting from disk via the SRM console. When the user issues the **boot** command to the SRM console, the console first reads in the initial sector of the specified boot device. Two fields in this sector specify the block offset and block count of an initial bootstrap program. The console then reads this bootstrap program into memory beginning at virtual address 0x20000000 and jumps to that address.

While it would be theoretically possible to simply read in the entire kernel this way, it is impractical for two reasons. First, 0x20000000 is in the middle of user memory space. The kernel could not remain there; it would have to be relocated to a more convenient address. Second, the kernel is large; since the bootstrap program loaded by the SRM must be contiguous, booting this way would tend to preclude such things as loading the kernel from a file system.

For these reasons, the preferred method of loading an operating system via the SRM console is to have the console load a small "bootloader" program; this, in

turn, can use the console callback functions to load the operating system itself from disk. Conceptually, the bootloader is rather simple; it sets up the kernel virtual memory space, reads in the kernel, and jumps to it.

The bootloader was developed in stages. The first version simply assumed that the kernel image was concatenated to the end of the bootloader image. The bootloader would examine the boot sector to determine where the kernel started, and it would read the COFF header of the kernel to determine how large it was.

The second major update of the bootloader added the ability to read files from an ext2 file system. This way, both the bootloader and the Linux kernel itself were regular files. The bootloader had to be installed by a special program (e2writeboot) which created a contiguous file on the ext2 file system and which wrote the extents of the bootloader file to the boot block. Nevertheless, this approach added greater flexibility as it made updating the bootloader and the Linux kernel much easier.

The final major update of the bootloader was provided by David Mosberger-Tang; it was the ability to unpack a *compressed* Linux kernel image. Not only did this save disk space, it made loading much faster as well.

Next month, we will cover porting the kernel.

Jim Paradis works as a Principal Software Engineer for Digital Equipment Corporation as a member of the Alpha Migration Tools group. Ever since a mainframe system administrator yelled at him in college, he's wanted to have a multiuser, multitasking operating system on his own desktop. To this end, he has tried nearly every Unix variant ever produced for PCs, including PCNX, System V, Minix, BSD, and Linux. Needless to say, he likes Linux best. Jim currently lives in Worcester, Massachusetts, with his wife, eleven cats, and a house forever under renovation.

Archive Index Issue Table of Contents

Advanced search