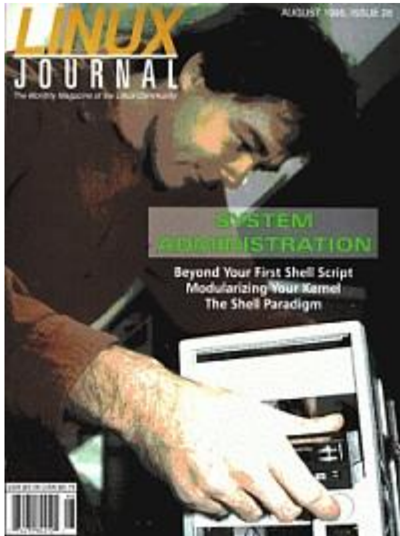
LINUX JOURNAL

Advanced search

Linux Journal Issue #28/August 1996



Features

Beyond Your First Shell Script *by Brian Rice*
: How to write versatile, robust Bourne shell scripts that will run flawlessly under other shells as well.

Diff, Patch, and Friends *by Michael K Johnson*
: De-mystifying patches and the tools used to create and apply them.

Auto-loading Kernel Modules *by Preston F. Crow*
: Make your system leaner by modularizing the kernel.

The Cold, Thin Edge *by Todd Graham Lewis*
: Taking the Shell Paradigm to its Brutal Limits. Whether you use Tcl, shells, Perl, or C, there is usually an option whereby tools from one programming environment can be imported into another. Here's how to “push the envelope”.

Basic fvwm Configuration, Part 2 *by John M Fisk*
: Customizing the Desktop. Organize and customize those pop-up menu entries.

News and Articles

Mobile-IP *by Ben Lancki, Abhijit Dixit, and Vipul Gupta*
: Transparent Host Migration on the Internet

Graphing with Gnuplot and Xmgr *by Andy Vaught*
: Two graphing packages available under Linux

Certifying Linux
: by Heiko Eissfeldt

Columns

Letters to the Editor
Stop the Presses
Kernel Korner Device Drivers Concluded
Book Review Bandits on the Information Superhighway
Book Review World Wide Web Journal, Issue One
Book Review Civilizing Cyberspace
New Products

Directories & References

Consultants Directory

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

LINUX JOURNAL

Beyond Your First Shell Script

Brian Rice

Issue #28, August 1996

As your shell scripts get more complex, you'll need to put a directive at the beginning to tell the operating system what sort of shell script this is.

So here it is—your first shell script:

```
lpr weekly.report
Mail boss < weekly.report
cp weekly.report /floppy/report.txt
rm weekly.report
```

You found yourself repeating the same few commands over and over: print out your weekly report, mail a copy to the boss, copy the report onto a floppy disc, and delete the original. So it was a big time saver when someone showed you that you can place those commands into a text file ("dealwithit", for instance), mark the file as executable with **chmod +x dealwithit**, and then run it just by typing its filename.

But you'd like to know more. This script you've written is not very robust; if you run it in the wrong directory, you get a cascade of ugly error messages. And the script is not very flexible either—if you'd like to print, mail, backup, and delete some other file, you'll have to create another version of the script. Finally, if someone asks you what kind of shell script you've written—Bourne? Korn? C shell?—you can't say. Then read on.

Last question first: what kind of shell script is this? Actually, the script above is quite generic. It uses only features common to all the shells. Lucky you. As your shell scripts get more complex, you'll need to put a directive at the beginning to tell the operating system what sort of shell script this is.

```
#!/bin/bash
```

The **#!** should be the first two characters of the file, and the rest should be the complete pathname of the shell program you intend this script to be run by.

Astute readers will note that this line looks like a comment, and, since it begins with a **#** character, it is one, syntactically. It's also magic.

When the operating system tries to run a file as a program, it reads the first few bytes of the file (its "magic number") to learn what kind of file it is. The byte pattern **#!** means that this is a shell script, and that the next several bytes, up to a newline character, make up the name of the binary that the OS should really run, feeding it this script file.

Paranoid programmers will make sure that no spaces are placed after the executable name on the **#!** line. You are paranoid, aren't you? Good. Also, notice that running a shell script requires that it be read first; this is why you must have both read and execute permission to run a shell script file, while you need only execute permission to run a binary file.

In this article, we will focus on writing programs for the Bourne shell and its descendants. A Bourne shell script will run flawlessly not only under the Bourne shell, but also under the Korn shell, which adds a variety of features for efficiency and ease of use. The Korn shell itself has two descendents of its own: the POSIX standard shell, which is virtually identical to the Korn shell; and a big Linux favorite, the Bourne Again shell. The Bourne Again shell ("bash") adds mostly interactive features from the descendants of the C shell, Bill Joy's attempt to introduce a shell that would use C-like control structures. What a great idea! Due to a few good reasons, most shell programming has followed the Bourne shell side of the genealogical tree. But people love the C shell's interactive features, which is why they too were incorporated into the Bourne Again shell.

Let me rephrase: do not write C shell scripts. Continue to use the C shell, or its descendant tcsh, interactively if you care to; your author does. But learn and use the Bourne/Korn/bash shells for scripting.

So here is our shell script now:

```
#!/bin/bash
lpr weekly.report
Mail boss < weekly.report
cp weekly.report /floppy/report.txt
rm weekly.report
```

If we run this script in the wrong directory, or if we accidentally name our file something other than "weekly.report", here's what happens:

```
lpr: weekly.report: No such file or directory
./dealwithit: weekly.report: No such file or directory
cp: weekly.report: No such file or directory
rm: weekly.report: No such file or directory
```

and we get a bunch of “Permission denied” messages if we run the script when the permissions on the file are wrong. Bleah. Couldn't we do a check at the beginning of the program, so that if something is wrong we can avert all these ugly messages? Indeed we can, using (surprise!) an **if**.

It occurs to us that if **cat weekly.report** works, so will most of the things our script wants to do. The shell's if statement works just as this thought suggests: you give the if statement a command to try, and if the command runs successfully, it will run the other commands for you too. You also can specify some commands to run if the first command—called the “control command”-- fails. Let's give it a try:

```
#!/bin/bash
if
    cat weekly.report
then
    lpr weekly.report
    Mail boss < weekly.report
    cp weekly.report /floppy/report.txt
    rm weekly.report
else
  echo I see no weekly.report file here.
fi
```

The indentation is not mandatory, but does make your shell scripts easier to read. You can put the control command on the same line as the if keyword itself.

This new version works great when there's an error. We get only one “No such file or directory” message, an improvement over four, and then our helpful personalized error message appears. But the script isn't so hot when it works: now we get the contents of weekly.report dumped to the screen as a preliminary. This is, after all, what cat does. Couldn't it just shut up?

You may know something about redirecting input and output in the world of Unix: the **>** character sends a command's output to a file, and the **<** character arranges for a command to get its input from a file, as in our Mail command. So if only we could send the cat command's output to the trash can instead of to a file... Wait! Maybe there's a trash can file somewhere. There is: /dev/null. Any output sent to /dev/null dribbles out the back of the computer. So let's change our cat command to:

```
cat weekly.report >/dev/null
```

Because you are paranoid, you may be wondering whether sending output to the trash can will affect whether this command succeeds or fails. Since /dev/null always exists and is writable by anyone, it will not fail.

Now our script is much quieter. But when cat fails, we still see the

```
cat: weekly.report: No such file or directory
```

error message. Why didn't this go into the trash can too? Because error messages flow separately from output, even though they usually share a common destination: the screen. We redirected standard output, but said nothing about errors. To redirect the errors, we can:

```
cat weekly.report >/dev/null 2>/dev/null
```

Just as **>** means “Send output here,” **2>** means “Send errors here.” In fact, **>** is really just a synonym for **1>**. Another, terser way to say the above command is this:

```
cat weekly.report >/dev/null 2>&1
```

The incantation **2>&1** means “Send errors (output stream number 2) to the same place ordinary output (output stream number 1) is going to.” By the way, this **2>** jazz only works in the Bourne shell and its descendants. The C shell makes it annoying to separate errors from output, which is one of the reasons people avoid programming in it.

You may be saying to yourself: “This cat trick is fun, but isn't there some way I can just give a true-or-false expression? Like, either the file exists and is readable, or not?” Yes, you can. There is a command whose whole job is to succeed or fail depending on whether the expression you give is true or not: test. This is why your test programs called “test” never work, by the way. Here is our program, rewritten to use test:

```
#!/bin/bash
if
    test -r weekly.report
then
    lpr weekly.report
    Mail boss < weekly.report
    cp weekly.report /floppy/report.txt
    rm weekly.report
else
    echo I see no weekly.report file here.
fi
```

The test command's **-r** operator means, “Does this file exist, and can I read it?” test is quiet regardless of whether it succeeds or fails, so there's no need for anything to get sent to /dev/null.

Test also has an alternative syntax: you can use a **[** character instead of the word test, so long as you have a **]** at the end of the line. Be sure to put a space between any other characters and the **[** and the **]** characters! We can make our if look like this now:

```
if [ -r weekly.report ]
```

Hey, now *that* looks like a program! Even though we're using brackets, this is still the test command. There are lots of other things test can do for you; see its man page for the complete list. For example, we seem to recall that what lets you delete a file is not whether you can read it, but whether the directory it sits in gives you write permission. So we can re-write our script like this:

```
#!/bin/bash
if [ ! -r weekly.report ]
then
    echo I see no weekly.report file here.
    exit 1
fi
    if [ ! -w . ]
then
    echo I will not be able to delete
    echo weekly.report for you, so I give up.
    exit 2
fi
# Real work omitted...
```

Each test now has a **!** character in it, which means “not”. So the first test succeeds if the weekly.report is not readable, and the second succeeds if the current directory (“.”) is not writable. In each case, the script prints an error message and exits. Notice that there's a different number fed to exit each time. This is how Unix commands (including **if** itself!) tell whether other commands succeed: if they exit with any exit code other than 0, they didn't. What each non-zero number (up to 255) means, other than “Something bad happened,” is up to you. But 0 always means success.

If this seems backwards to you, give yourself a cookie. It *is* backwards. But there's a good design reason for it, and it's a universal Unix-command convention, so get used to it.

Notice also that our real work no longer has an if wrapped around it. Our script will only get that far if none of our error conditions are detected. So we can just assume that all those error conditions are not in fact present! Real shell scripts exploit this property ruthlessly, often beginning with screenfuls of tests before any real work is done.

Now that we've made our script more robust, let's work on making it more general. Most Unix commands can take an argument from their command lines that tells them what to do; why can't our script? Because it has “weekly.report” littered all through it, that's why. We need to replace weekly.report with something that means “the thing on the command line.” Meet **$1**.

```
#!/bin/bash
if [ ! -r $1 ]
then
    echo I see no $1 file here.
    exit 1
fi
if [ ! -w . ]
then
```

```
        echo I will not be able to delete $1 for you.
        echo So I give up.
        exit 2
fi
lpr $1
Mail boss < $1
# and so forth...
exit 0
```

$1 means the first argument on the command line. Yes, **$2** means the second, **$3** means the third, and so on. What's **$0**? The name of the command itself. So we can change our error messages so that they look like this:

```
echo $0: I see no $1 file here.
```

Ever noticed that Unix error messages introduce themselves? That's how.

Unfortunately, now there's a new threat to our program: what if the user forgets to put an argument on the command line? Then the right thing for $1 to have in it would be nothing at all. We might be back to our cascade of error messages, since a lot of commands, such as rm, complain at you if you put nothing at all on their command lines. In this program's case, it's even worse, since the first time $1 is used is as an argument to test -r, and test will give you a syntax error if you ask it to test -r nothing at all. And what does lpr do if you put nothing at all on its command line? Try it! But be prepared; you could end up with a mess.

Fortunately, test can help. Let's put this as the *very first* test in our program, right after the **#!/bin/bash**:

```
if [ -z "$1" ]
then
        echo $0: usage: $0 filename
        exit 3
fi
```

Now if the user puts nothing on the command line, we print a usage message and quit. The **-z** operator means “is this an empty string?”. Notice the double quotes around the **$1**: they are mandatory here. If you leave them out, test will give an error message in just the situation we are trying to detect. The quotes protect the nothing-at-all stored in **$1** from causing a syntax error.

This if clause appears at the very top of many, many shell scripts. Among its other benefits, it relieves us from having to wrap **$1** in quote marks later in our program, since if **$1** were empty we would have exited at the start. In fact, the only time quotes would still be necessary would be if **$1** could contain characters with a special meaning to the shell, such as a space or a question mark. Filenames don't, usually.

What if we want our script to be able to take a variable number of arguments? Most Unix commands can, after all. One way is clear: we could just cut and paste all the stuff in our shell script, so we'd have a bunch of commands that dealt with **$1**, then a bunch of commands that dealt with **$2**, and so forth. Sound like a good idea? No? Good for you; it's a terrible idea.

First of all, there would be some fixed upper limit on the number of arguments we could handle, determined by when we got tired of cutting, pasting, and editing our script. Second, any time you have many copies of the same code, you have a quality problem waiting to happen. You'll forget to make a change, or fix a bug, all of the many places necessary. Third, we often hand wildcards, like *, to Unix commands on their command lines. These wildcards are expanded into a list of filenames before the command runs! So it's very easy to get a command line with more arguments than some arbitrary, low limit.

Maybe we could use some kind of arithmetic trick to count through our arguments, like **$i** or something. This won't work either. The expression **$i** means “the contents of the variable called i”, not “the i'th thing on the command line.” Furthermore, not all shells let you refer to command-line words after **$9** at all, and those that do make you use **${10}**, **${11}**, and so forth.

So what do we do? This:

```
while [ ! -z "$1" ]
do
    # do stuff to $1
    shift
done
```

Here's how we read that script: “While there's something in **$1**, we mess with it. Immediately after we finish messing with it, we do the **shift** command, which moves the contents of **$2** into **$1**, the contents of **$3** into **$2**, and so forth, regardless of how many of these command-line arguments there are. Then we go back and do it all again. We know we've finished when there's nothing at all in $1.”

This technique allows us to write a script that can handle any number of arguments, while only dealing with **$1** at a time. So now our script looks like this:

```
#!/bin/bash
while [ ! -z "$1" ]
do
    # do stuff to $1
    if [ ! -r $1 ]
    then
        echo $0: I see no $1 file here.
        exit 1
    fi
        # omitted test...
    lpr $1
    Mail boss < $1
```

```
    # and so forth...
    shift
done
exit 0
```

Notice that we nested **if** inside **while**. We can do that all we like. Also notice that this program quits the instant it finds something wrong. If you would like it to continue on to the next argument instead of bombing out, just replace an **exit** with:

```
shift
continue
```

The **continue** command just means “Go back up to the top of the loop right now, and try the control command again.” Thought question: why did we have to put a shift right before the continue?

Here's a potential problem: we've made it easy for someone to use this program on files that live in different directories. But we're only testing the current directory for writability. Instead, we should do this:

```
if [ ! -w `dirname $1` ]
then
    echo $0: I will not be able to delete $1 for you.
    # ...
```

The dirname command prints out what directory a file is in, judging from its pathname. If you give dirname a filename that doesn't start with a directory, it will print “.”--the current directory. And those backquotes? Unlike all other kinds of quotation marks, they don't mean “this is really all one piece ignore spaces.” Instead, backquotes—also called “grave accents”--mean “Run the command inside the backquotes before you run the whole command line. Capture all of the backquoted command's output, and pretend that was what appeared on the larger command line instead of the junk in backquotes.” In other words, we are substituting a command's output into another command line.

So here is the final version of our shell script:

```
#!/bin/bash
while [ ! -z "$1" ]
do
    if [ ! -r $1 ]
    then
        echo $0: I see no $1 file here.
        shift
        continue
    fi
    if [ ! -w `dirname $1` ]
    then
        echo $0: I will not be able to delete $1 for you.
        shift
        continue
    fi
    lpr $1
    Mail boss < $1
    cp $1 /floppy/`basename $1`
    rm $1
```

```
    shift
done
exit 0
```

An exercise for the reader: what does **`basename $1`** do?

Now there are only two other techniques you need to know to meet the vast majority of your scripting needs. First, suppose you really do need to count. How do we do the equivalent of a C for loop? Here's the traditional Bourne shell way:

```
i=0
upperlim=10
while [ $i -lt $upperlim ]
do
    # mess with $i
    i=`expr $i + 1`
done
```

Notice that we did not use the **for** keyword. **for** is for something else entirely. Instead, here we initialize a variable **i** to 0, then we enter and remain in the loop as long as the value in **i** is less than 10. (Fortran programmers will recognize **-lt** as the less-than operator; guess why **>** is not used in this context.) The rather mysterious line

```
i=`expr $i + 1`
```

calls the expr command, which evaluates arithmetic expression. We stuff expr's output back into **i** using backquotes.

Ugly, isn't it? And not especially fast either, since we are running a command every time we want to add 1 to **i**. Can't the shell just do the arithmetic itself? If the shell is the Bourne shell, no, it can't. But the Korn shell can:

```
((i=i+1))
```

Use that syntax if it works, and if you don't need portability. The bash shell uses something similar:

```
i=$(($i+1))
```

which is a bit more portable (it even works in the Korn shell), since it is specified by POSIX, but still won't work for some non-POSIX bourne shells.

So what does **for** do? It allows you to wade through a list of items, assigning a variable to each element of the list in turn. Here's a trivial example:

```
for a in Larry Moe Curly
do
    echo $a
done
```

which would print

```
Larry
Moe
Curly
```

Less trivially, we can use this to handle the case where we want to do something for each word in a variable:

```
mylist="apple banana cheese rutabaga"
for w in $mylist
do
    # mess with $w
done
```

or for each file matched by a shell wildcard pattern:

```
for f in /docs/reports/*.txt
do
    pr -h $f $f | lpr
done
```

or for each word in the output of a command:

```
for a in `cat people.txt`
do
    banner $a
done
```

Here's how you can use for to simulate the C for you know and love:

```
for i in 0 1 2 3 4 5 6 7 8 9 10 11
do
    # mess with $i
done
```

Of course, it'd be very difficult to have a variable upper limit with this syntax, which is why we usually use the while loop shown above.

Congratulations! You've now seen what's at work in the vast bulk of practical shell scripts. Go forth and save time!

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

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

LINUX JOURNAL

Diff, Patch, and Friends

Michael K. Johnson

Issue #28, August 1996

"Kernel patches" may sound like magic, but the two tools used to create and apply patches are simple and easy to use—if they weren't, some Linux developers would be too lazy to use them...

Diff is designed to show you the **diff**erences between files, line by line. It is fundamentally simple to use, but takes a little practice. Don't let the length of this article scare you; you can get some use out of diff by reading only the first page or two. The rest of the article is for those who aren't satisfied with very basic uses.

While diff is often used by developers to show differences between different versions of a file of source code, it is useful for far more than source code. For example, diff comes in handy when editing a document which is passed back and forth between multiple people, perhaps via e-mail. At *Linux Journal*, we have experience with this. Often both the editor and an author are working on an article at the same time, and we need to make sure that each (correct) change made by each person makes its way into the final version of the article being edited. The changes can be found by looking at the differences between two files.

However, it is hard to show off how helpful diff can be in finding these kinds of differences. To demonstrate with files large enough to really show off diff's capabilities would require that we devote the entire magazine to this one article. Instead, because few of our readers are likely to be fluent in Latin, at least compared to those fluent in English, we will give a Latin example from *Winnie Ille Pu*, a translation by Alexander Leonard of A. A. Milne's *Winnie The Pooh* (ISBN 0-525-48335-7). This will make it harder for the average reader to see differences at a glance and show how useful these tools can be in finding changes in much larger documents.

Quickly now, find the differences between these two passages:

```
Ecce Eduardus Ursus scalis nunc tump-tump-tump
occipite gradus pulsante post Christophorum
Robinum descendens. Est quod sciat unus et solus
modus gradibus desendendi, non nunquam autem
sentit, etiam alterum modum exstare, dummodo
pulsationibus desinere et de no modo meditari
possit. Deinde censet alios modos non esse. En,
nunc ipse in imo est, vobis ostentari paratus.
Winnie ille Pu.
```

```
Ecce Eduardus Ursus scalis nunc tump-tump-tump
occipite gradus pulsante post Christophorum
Robinum descendens. Est quod sciat unus et solus
modus gradibus descendendi, nonnunquam autem
sentit, etiam alterum modum exstare, dummodo
pulsationibus desinere et de eo modo meditari
possit. Deinde censet alios modos non esse. En,
nunc ipse in imo est, vobis ostentari paratus.
Winnie ille Pu.
```

You may be able to find one or two changes after some careful comparison, but are you sure you have found *every* change? Probably not: tedious, character-by-character comparison of two files should be the computer's job, not yours.

Use the diff program to avoid eyestrain and insanity:

```
diff -u 1 2
--- 1   Sat Apr 20 22:11:53 1996
+++ 2   Sat Apr 20 22:12:01 1996
 -1,9 +1,9
 Ecce Eduardus Ursus scalis nunc tump-tump-tump
 occipite gradus pulsante post Christophorum
 Robinum descendens. Est quod sciat unus et solus
-modus gradibus desendendi, non nunquam autem
+modus gradibus descendendi, nonnunquam autem
 sentit, etiam alterum modum exstare, dummodo
-pulsationibus desinere et de no modo meditari
+pulsationibus desinere et de eo modo meditari
 possit. Deinde censet alios modos non esse. En,
 nunc ipse in imo est, vobis ostentari paratus.
 Winnie ille Pu.
```

There are several things to notice here:

- The file names and last dates of modification are shown in a “header” at the top. The dates may not mean anything if you are comparing files that have been passed back and forth by e-mail, but they become very useful in other circumstances.
- The file names (in this case, 1 and 2—are preceded by **---** and **+++**.
- After the header comes a line that includes numbers. We will discuss that line later.
- The lines that did not change between files are shown preceded by spaces; those that are different in the different files are shown preceded by a character which shows **which** file they came from. Lines which exist only in a file whose name is preceded by **---** in the header are preceded by a **-** character, and vice-versa for lines preceded by a **+** character. Another way to remember this is to see that the lines preceded by a **-** character

were *removed* from the first (**---**) file, and those preceded by a **+** character were *added* to the second (**+++**) file.
- Three spelling changes have been made: “desendendi” has been corrected to “descendendi”, “non nunquam” has been corrected to “nonnunquam”, and “no” has been corrected to “eo”.

Perhaps the main thing to notice is that you didn't need this description of how to interpret diff's output in order to find the differences. It is rather easy to compare two adjacent lines and *see* the differences.

It's not always this easy

Unfortunately, if too many adjacent lines have been changed, interpretation isn't as immediately obvious; but by knowing that each marked line has been changed in some way, you can figure it out. For instance, in this comparison, where the file 3 contains the damaged contents, and file 4 (identical to file 2 in the previous example) contains the correct contents, three lines in a row are changed, and now each line with a difference is not shown directly above the corrected line:

```
diff -u 3 4
--- 3   Sun Apr 21 18:57:08 1996
+++ 4   Sun Apr 21 18:56:45 1996
 -1,9 +1,9
 Ecce Eduardus Ursus scalis nunc tump-tump-tump
 occipite gradus pulsante post Christophorum
 Robinum descendens. Est quod sciat unus et solus
-modus gradibus desendendi, non nunquam autem
-sentit, etiam alterum nodum exitare, dummodo
-pulsationibus desinere et de no modo meditari
+modus gradibus descendendi, nonnunquam autem
+sentit, etiam alterum modum exstare, dummodo
+pulsationibus desinere et de eo modo meditari
 possit. Deinde censet alios modos non esse. En,
 nunc ipse in imo est, vobis ostentari paratus.
 Winnie ille Pu.
```

It takes a little more work to find the added mistakes; “nodum” for “modum” and “exitare” for “exstare”. Imagine if 50 lines in a row had each had a one-character change, though. This begins to resemble the old job of going through the whole file, character-by-character, looking for changes. All we've done is (potentially) shrink the amount of comparison you have to do.

Fortunately, there are several tools for finding these kinds of differences more easily. GNU Emacs has “word diff” functionality. There is also a GNU “wdiff” program which helps you find these kinds of differences without using Emacs.

Let's look first at GNU Emacs. For this example, files 5 and 6 are exactly the same, respectively, as files 3 and 4 before. I bring up emacs under X (which provides me with colored text), and type:

```
M-x ediff-files RET
5 RET
6 RET
```

In the new window which pops up, I press the space bar, which tells Emacs to highlight the differences. Look at Figure 1 and see how easy it is to find each changed word.

Figure 1. ediff-files 5 6

GNU wdiff is also very useful, especially if you aren't running X. A pager (such as less) is all that is required—and that is only required for large differences. The exact same set of files (5 and 6), compared with the command **wdiff -t 5 6**, is shown in Figure 2.

Figure 2. wdiff -t 5 6

If you are getting extra character sequences like **ESC[24** instead of getting underline and reverse video, it's probably because you are using **less**, which by default doesn't pass through all escape characters. Use **less -r** instead, or use the more pager. Either should work.

wdiff uses the *termcap* database (that's what the **-t** option is for) to find out how to enable underline and reverse video, and not all termcap entries are correct. In some instances, I've found that the **linux** termcap entry works well for other terminals, since the codes for turning underline and reverse video on and off don't differ very much across terminals. To use the linux termcap entry, you can do this:

```
TERM=linux wdiff -t 5 6 | less -r
```

This will work only with bourne shell derivatives such as bash, not with csh or tesh. But since you need to do this only to correct for a broken termcap database, this limitation shouldn't be too much of a problem.

wdiff isn't always built with the termcap support needed to underline and reverse video, and it's not always what you want even if you have a working termcap database, so there's an alternate output format that is just as easy to understand. We'll kill two birds with one stone by also showing off wdiff's ability to deal with re-wrapped paragraphs while showing off its ability to work without underline and reverse video. File 8 is the same as the correct file 2, shown at the beginning of this article, but file 7 (the corrupted one) now has much shorter lines, which makes them even harder to compare “by eye”:

```
Ecce Eduardus Ursus scalis
nunc tump-tump-tump occipite
gradus pulsante post
Christophorum Robinum
```

```
descendens. Est quod sciat
unus et solus modus gradibus
desendendi, non nunquam autem
sentit, etiam alterum nodum
exitare, dummodo pulsationibus
desinere et de no modo
meditari possit. Deinde censet
alios modos non esse. En, nunc
ipse in imo est, vobis
ostentari paratus.
Winnie ille Pu.
```

wdiff is not confused by the differently-wrapped lines. The command **wdiff 7 8** produces this output:

```
Ecce Eduardus Ursus scalis nunc tump-tump-tump
occipite gradus pulsante post Christophorum
Robinum descendens. Est quod sciat unus et solus
modus gradibus
[-desendendi, non nunquam-]
{+descendendi, nonnunquam+} autem
sentit, etiam alterum [-nodum
exitare,-] {+modum exstare,+} dummodo
pulsationibus desinere et de [-no-] {+eo+}
modo meditari
possit. Deinde censet alios modos non esse. En,
nunc ipse in imo est, vobis ostentari paratus.
Winnie ille Pu.
```

Remember the **+** and **-** characters? They mean the same thing with wdiff as they mean with diff. (Consistent user interfaces are wonderful.)

Chunks

Near the beginning of this article, I promised to explain this line:

```
 -1,9 +1,9
```

that describes the **chunk** that diff found differences in. In each file, the chunk starts on line 1 and extends for 9 lines beyond the first line. However, with this small example, the chunk shown in the example contains the *whole* file. With larger files, only the lines around the changes, called the **context**, are shown.

In files 9 and 10, I've inserted a lot of blank lines in the middle of the paragraph, in order to show what multiple chunks look like. File 9 is damaged, file 10 is correct (except for the blank lines in the middle of the paragraph):

```
<h3>diff -u 9 10</h3>
--- 9   Mon Apr 22 15:46:37 1996
+++ 10  Mon Apr 22 15:46:14 1996
 -1,7 +1,7
 Ecce Eduardus Ursus scalis nunc tump-tump-tump
 occipite gradus pulsante post Christophorum
 Robinum descendens. Est quod sciat unus et solus
-modus gradibus desendendi, non nunquam autem
+modus gradibus descendendi, nonnunquam autem
 -33,7 +33,7
 sentit, etiam alterum modum exstare, dummodo
-pulsationibus desinere et de no modo meditari
+pulsationibus desinere et de eo modo meditari
 possit. Deinde censet alios modos non esse. En,
```

```
    nunc ipse in imo est, vobis ostentari paratus.
    Winnie ille Pu.
```

So you see that we have one seven-line chunk starting at line 1 and one seven-line chunk starting at line 33 are shown here.

You should notice several things here:

- There is one header at the top of each chunk.
- Blank lines are included as part of a chunk's context.
- Lines that are not changed and that are not within three lines of a changed line are not included in any chunk.

"Patches" (or "diffs") are the output of the diff program. They include all the chunks of changes between the two files.

Other formats

This only brushes the surface of diff. For one thing, the three lines of unchanged context is configurable. Instead of using the **-u** option, you can use the **-U *lines*** option to specify any reasonable number of lines of context. You can even specify **-U 0** if you don't want to use any context at all, though that is rarely useful.

What does the **-u** (or **-U *lines***) argument mean? It specifies the *unified diff format*, which is the particular format covered here. Other formats include:

- "context diffs" which have the same information as unified diffs, but are less compact and less readable
- "ed script diffs" or "normal diffs" which are in a format that can be easily converted into a form that can be used to cause the (nearly obsolete) editor ed to automatically change another copy of the old file to match the new file. This format has no context and could easily be replaced by **-U 0**, except for compatibility with older software and the POSIX standard.

You will almost never want to create context or normal diffs, but it may be useful to recognize them from time to time. Context diffs are marked by the use of the character **!** to mark changes, and normal diffs are marked by the use of the characters **<** and **>** to mark changes.

Here are examples:

```
diff -c 1 2
*** 1   Sat Apr 20 22:11:53 1996
--- 2   Sat Apr 20 22:12:01 1996
***************
*** 1,9 ****
  Ecce Eduardus Ursus scalis nunc tump-tump-tump
```

```
  occipite gradus pulsante post Christophorum
  Robinum descendens. Est quod sciat unus et solus
! modus gradibus desendendi, non nunquam autem
  sentit, etiam alterum modum exstare, dummodo
! pulsationibus desinere et de no modo meditari
  possit. Deinde censet alios modos non esse. En,
  nunc ipse in imo est, vobis ostentari paratus.
  Winnie ille Pu.
--- 1,9 ----
  Ecce Eduardus Ursus scalis nunc tump-tump-tump
  occipite gradus pulsante post Christophorum
  Robinum descendens. Est quod sciat unus et solus
! modus gradibus descendendi, nonnunquam autem
  sentit, etiam alterum modum exstare, dummodo
! pulsationibus desinere et de eo modo meditari
  possit. Deinde censet alios modos non esse. En,
  nunc ipse in imo est, vobis ostentari paratus.
  Winnie ille Pu.
diff 1 2
4c4
< modus gradibus desendendi, non nunquam autem
---
> modus gradibus descendendi, nonnunquam autem
6c6
< pulsationibus desinere et de no modo meditari
---
< pulsationibus desinere et de eo modo meditari
```

There are a few other important things to note here:

- In context diffs, the ***** character is used in place of the unified diff's **-** character, and the **-** character is used in place of the **+** character. The context diff format was designed before the unified diff format, but the unified diff format's choice of characters is mnemonic and therefore preferable.
- Context diffs repeat all context twice for each chunk. This is a waste of space in files, but far more importantly, it separates the changes too widely, making patches less human-readable.
- Normal, old-style diffs are very contracted and use very little space. They are useful in situations where you don't normally expect a human to read them, where saving space makes a lot of sense, and where they will never be applied to files which have changed. For example, RCS (covered in the May 1996 issue of *LJ*) uses a format almost identical to old-style diffs to store changes between versions of files. This saves space and time in a situation where any context at all would be a waste of space.

Using Patches

When someone changes a file that other people have copies of (source code, documentation, or just about any other text file), they often send patches instead of (or in addition to) making the entire new file available. If you have the old file and the patches, you might wish that you could have a program apply the patches. You might think that normal diff format, which was made to look like input to the ed program, would be the best way to accomplish this.

As it turns out, this is not true.

A program called **patch** has been written which is specifically designed to apply patches to files (change the files as specified in the patch). It correctly recognizes all the formats of patches and applies them. With unified and context diffs, patch can usually apply patches, *even if lines have been added or removed from the file*, by looking for unchanged context lines. Only if the context lines have themselves been changed is patch likely to fail.

To apply patches with patch, you normally have a file containing the patch (we'll call it *patchfile*), and then run patch:

```
patch < patchfile
```

Patch is very verbose. If it gets confused by anything, it stops and asks you in English (it was written by a linguist, not a computer scientist) what you want to do. If you want to learn more about patch, the man page is unusually readable.

Other Related Tools

If you read the RCS article in the May issue (*Take Command: Keeping Track of Change*, *LJ* #25, May 1996), you may have noticed that the article talked a bit about a program called rcsdiff. rcsdiff is really just a front end to diff. That is, it looks for arguments that it understands (such as revision numbers and the filename) and prepares two files representing the two versions of the file you are examining. It then calls diff with the remaining options. The RCS article used **-u** to get the unified format without explaining what it meant, but you can use **-c** to get context diffs, or use **-U *lines*** to choose the amount of context you get in a unified diff, or use any other diff options you like.

You may notice that rcsdiff produces more verbose output than normal diff. From the RCS article:

```
rcsdiff -u -r1.3 -r1.6 foo
===============================================
RCS file: foo,v
retrieving revision 1.3
retrieving revision 1.6
diff -u -r1.3 -r1.6
--- foo 1996/02/01 00:34:15     1.3
+++ foo 1996/02/01 01:05:28     1.6
 -1,2 +1,6
 This is a test of the emergency
-RCS system.  This is only a test.
+RCS version control system.
+This is only a test.
+
+I'm now adding a few lines for
+the next version.
```

It looks just like a normal unified diff except for the first 5 lines.

This doesn't prevent you from sending patches to people. The patch program is extremely good about ignoring extraneous information. It can even ignore

news or mail headers, extra comments written in a file outside a patch, and people's signatures following patches. Patch tells you when it is determining whether text is part of a patch or not by saying “**Hmm...**”

If you don't care *how* two files differ, but just want to know *whether* they differ, the cmp program will tell you. It works not only for text files, but also for binary files. In this example, the files 5 and 6 are different; 2 and 4 are the same:

```
cmp 5 6
5 6 differ: char 159, line 4
cmp 2 4
```

Notice that when two files are the same, cmp doesn't say anything at all. It only tells you explicitly if the files have been changed. For use in writing shell scripts, cmp also returns true if the files are the same and false if they don't, as shown by this shell session:

```
if cmp 5 6 ; then
  echo "same"
else
  echo "different"
fi
5 6 differ: char 159, line 4
different
if cmp 2 4 ; then
  echo "same"
else
  echo "different"
fi
same
```

There are several other programs with related functionality. In particular, diff3 can be used to merge together two different files that have both been edited from a common ancestor file. That common ancestor must exist in order for diff3 to work correctly.

The info pages which are shipped with diff are probably installed on your system. If you want to learn more about diff, try the command **info diff** or use info mode from within emacs or jed.

diff, wdiff, patch, and emacs are available via ftp from the canonical GNU ftp archive, prep.ai.mit.edu, in the directory /pub/gnu/

Michael K. Johnson His wife Kim likes A. A. Milne and briefly studied Latin (unlike Michael, whose experience with Latin was limited to singing in choir), which is why she owns Winnie Ille Pu as well as *Tela Charlottae* (Charlotte's Web).

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Auto-loading Kernel Modules

Preston F. Crow

Issue #28, August 1996

Removing code from the kernel that provides unneeded support is one option long associated with Linux. Now, you can remove code that is not constantly required, putting it in modules loaded on command.

Like many operating systems, Linux offers support for numerous devices, file systems, and network protocols. Unfortunately, this growing support increases the memory requirements of the kernel. Linux partially solves this problem by allowing selection of only the features you need when compiling the kernel. This is further improved by allowing some features to be compiled as modules, so they can be loaded only when they are needed. The loading and unloading of modules can be automated with the use of kerneld, making the use of features compiled as modules just as easy as using those included in the basic kernel.

To use kerneld, you should start by installing the most recent version of the modules package, found at www.pi.se/blox/modules/. I'm using modules-1.3.69f, but there's probably a newer version out by the time you read this article. Also, you'll need a kernel at least as recent as 1.3.57.

Fortunately, kerneld automatically knows about most modules. All you must do is run it in your startup script. For Slackware-based systems, you'll need to edit /etc/rc.d/rc.local. You should include the following:

```
# Update kernel-module dependencies file
[ -x /sbin/depmod ] && {
        /sbin/depmod -a
}
# Start kerneld
[ -x /sbin/kerneld ] && {
        /sbin/kerneld
}
```

For Red Hat systems, you can install the contributed modules RPM on ftp.redhat.com in /pub/contrib/RPMS/ called modules-1.3.57-3.i386.rpm which

provides support for kerneld. A newer version will probably be released by the time you read this, so look for a later version of the modules utility if you can't find it there. Alternately, read the article “Understanding Red Hat Runlevels” in *LJ* issue 27 (July 1996) and create a kerneld boot script in /etc/init.d with appropriate links in /etc/rc2.d, /etc/rc3.d, /etc/rc4.d, and /etc/rc5.d.

In either case, this runs depmod, which updates dependency information used by kerneld and then starts kerneld, which forks and hides in the background until the kernel needs it.

Now, all you need to do is reconfigure your kernel to use modules for the features you're not always using, and build and install the kernel and the modules. If you've never built modules before, simply add two steps to the kernel compilation process: **make modules** and **make modules_install**.

When you boot the new kernel, you should have all your modules loading automatically whenever you try to use them. The command **lsmod** will tell you which modules are loaded. Of course, it is a good idea to keep your old kernel bootable in case something doesn't work as expected.

Unfortunately, kerneld doesn't know about every module you might want to install—particularly those not part of the kernel distribution. To install these modules, you'll need to tell kerneld about them in /etc/conf.modules. Kerneld needs to know both where to find the module and what event triggers loading it.

I strongly recommend you use the default directories for your modules. Otherwise, you'll have to add not only the new path to /etc/conf.modules, but all the default paths as well. To see the default paths, use **modprobe -c | more**.

Telling kerneld what triggers the loading of a module requires adding **alias** entries in /etc/conf.modules. For device drivers, such as zftape.o or joystick.o, the format is based on the device type (character or block) and major number. For example, I use **alias char-major-15 joystick** for the joystick driver. You can get a bunch of examples by running **modprobe -c** to see the defaults. You can have multiple entries for the same module if there are multiple events that should trigger loading it.

You may also need to set an alias if you want to load an optional module, like BSD compression with PPP. The simplest alias to use for BSD compression is **alias ppp bsd_comp**. This will tell kerneld to load bsd_comp instead of ppp, but since bsd_comp requires the real ppp module (which requires slhc), it will load slhc and ppp first. Of course, if you have trouble with this, you can always load

the modules explicitly in your dialing script and unload them in /etc/ppp/ip-down.

You can also use kerneld to set up a dial-on-demand network connection. When the kernel receives a request to send a packet to an IP address for which there is no routing information, it asks kerneld if it can establish a route to that address. When kerneld receives such a request, it runs /sbin/request-route, which should, generally, be a script to start PPP or SLIP, thereby establishing a route.

So, all you have to do is replace /sbin/request-route with your dialing script. Well... not exactly. If you rely completely on an outside nameserver, you can probably get away with that. In general, however, you need to be careful, as kerneld may call request-route several times, once for each IP address the kernel needs to resolve. This can be solved by using a lock file for the modem device, which is an option for chat and pppd. [You should be using that option anyway! —ED]

What Should Be Compiled as a Module?

When configuring your system, at first you may think it would be best to compile everything as a module. This isn't always a good idea, as it won't always save memory. Each module uses memory in 4K pages, so the last page will generally have some space wasted. Therefore, if you'll almost always have the module in use, you might as well compile it into the kernel. Also, keep in mind that kerneld itself consumes some memory (in my experience, at least 12 pages), so if you only have a few small modules to worry about, it would be better to compile them into the kernel or load them explicitly in the startup script.

Modules for file systems must be loaded as long as the file system is mounted, even if you're not using it. So if you keep /dos mounted all the time, don't bother to compile support for FAT as a module. If you don't like that option, you could look into using an automount daemon instead of keeping the file system mounted.

Be careful with modules that include information that may be changed when run. For example, the sound driver keeps track of the volume, and if you compile it as a module, the volume will be reset to the default each time it is loaded.

Finally, be careful not to compile something as a module if it will be used at boot time before kerneld is started. This includes the root file system, of course. For many systems, you'll find that you need both ELF and a.out support before kerneld starts. You may be able to overcome some problems by

installing kerneld as one of the first programs executed by the startup scripts, but be careful if you're also doing dial-on-demand, as you may have something like sendmail in your startup scripts that will trigger it. As long as you have your old kernel around as a safety net, though, feel free to experiment.

Preston Crow Preston Crow is a graduate student in computer science at Dartmouth College. He became a happy Linux user in the summer of 1995, shortly before becoming happily married.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

The Cold, Thin Edge

Todd Graham Lewis

Issue #28, August 1996

Open up your Unix toolbox and you will see a complete set of tools ready to be used. The ability to differentiate separate, simultaneous processes and direct their input and output at your discretion and the will to use this ability, constitute the shell paradigm.

The Shell Paradigm is described (by me at least) as taking some of a true operating system's most beautiful characteristics and bending, twisting, folding, spindling, and mutilating them into obscenely obtuse and imperfect tools. That these characteristics can be bent, twisted, etc., and still work is, of course, what gives them their beauty.

Open up your Unix toolbox (/usr/bin for you gnubies), and you will see a complete set of tools, ready for use. Much as the discovery of a basic technology distinguishes one epoch of human history from another, redirection and job control under Unix create a golden age of computing in contrast to the iron-age toils of MS-DOS. Because of the simple ability to differentiate separate, simultaneous processes and direct their input and output at your discretion, there are few limits to the ways in which you can use these tools to assemble simple Unix processes. This ability, and the will to use it, constitute the shell paradigm.

But where power resides lies danger. How much **|** **&** and **popen()** can a single process take before it disintegrates into a heap of intractable spaghetti code? How many different programming contexts can we use before our simple program hurtles out of control towards the nether-regions of “Kernel Panic: Out of memory”? [A lot—ED]

This article will describe to you how to mix and match I/O streams to and from executables in different environments. If you are hacking a Perl script and want to throw a little **grep** in for good measure, go right ahead; it's possible. Finally, we will discuss the limits to and wisdom of these techniques.

The Shell

The capability to have processes communicate easily among themselves is inherent in the design of Unix systems, so the appellation “shell paradigm” is somewhat of a misnomer. Nonetheless, the shell is the context in which most people are familiar with I/O redirections, so we will start there. As we will later see, all these facilities can be easily recreated in places other than at the shell prompt.

There are several ways to use process redirection within the shell. You can take the output of a process and direct it to a file, for example:

```
cd ~; ls > /tmp/ls.file
```

Alternatively, you can append output to existing files:

```
cd ~/bin; ls >> /tmp/ls.file
```

You can also take the output of a process and redirect it as the input of another process:

```
cd ~; ls | grep lj.article
```

Within most shells, including the Bourne-compatible bash and zsh, you can integrate the output of your command within other commands. For example, if you wanted to generate a file with yesterday's time appended to the end, you could do the following:

```
touch /usr/acct/atlanta/data.`
  date --date '1 day ago' +"%Y%m%d"
 `
```

which just generated a file named data.19960503 for me. What you get depends on how quickly you read your *Linux Journal*. It also depends on which free OS you are running; FreeBSD's version of date does not offer the **1 day ago** facility, so you will have to get and compile gnu-date if you are silly enough not to run Linux (or if your employer uses FreeBSD.)

C

External-command inclusion is nice in C when you need a function already implemented as a Unix tool which you don't want to recode. For example, if you need to sort a stream of data or compress an output file, using **sort** or **gzip** rather than coding it natively is an efficient way to accomplish the task. There are two ways to use external programs under C: **system()** and **popen()**.

If you have a large amount of data in strings that you want to sort using the **sort** program, you can use **popen()** to call the sort program, sort the data and read the result back from the program. If you just want to compress a file, you can use the simpler **system()** function. Neither function is unfamiliar to a C programmer, but if either is unfamiliar to *you*, Look in the Linux man pages, where they are documented. If you want more explanation, you can read *Advanced Programming in the Unix Environment*, by W. Richard Stevens.

However, if you need to *interact* with the program you call, it is possible to do this with a C library that comes with a tool called “Expect”, which is described later in the Tcl section.

Mother of Perl

Whereas there are a number of different ways to manipulate process I/O within the shell, there is really only one within Perl: as a filehandle. This is actually a testimony to the beauty of Perl's design; kudos to Larry Wall for making it so simple.

You can include other processes from within Perl in several different manners, all with the **open ()** command. For example, if you wanted to open a process **bottle** to which the output of your Perl script should be sent, you would use

```
open (BOTTLE, "| ~<bin/bottle"
```

to direct the output. Similarly, if you wanted to read the input of bottle, you would do much the same thing, adding the pipe symbol (**|**) at the end:

```
open (BOTTLE, "~<bin/bottle |")
```

In the first case, you could only write to filehandle bottle, whereas in the second case, you could do nothing but read.

Commands opened in this manner can also get fancy. Everything within the quotation marks is executed from within a subshell, so commands like either of the following will work:

```
open (BOTTLE, "cd ~; /bin/bottle |")
```

```
open (FIND, "cd /home/tlewis; find . -name $string -print |")
```

At this point many people ask, “What if I want to do both reading and writing?” You can't do this with the **open ()** command, so Perl is broken, right? No, not really. The fact that you can't easily open a two-way pipe is a design decision. As explained in the Unix FAQ:

The problem with trying to pipe both input and output to an arbitrary slave process is that deadlock can occur, if both processes are waiting for not-yet-generated input at the same time.

Again, it is possible to do this with Expect, as we'll see later.

A short example:

```
#!/usr/bin/Perl
open (ACCT, "(cd /usr/acct/;".
  "for i in `ls | grep -v admin`; do; ".
  "cat $i/date.19960503; done) | sort |");
while (<ACCT>) {
     chop;
     ($A,$B,$C) = split;
     print "$C $A $B\n";
}
```

This would take the data in a limited subset of the /usr/acct/ directory, sort it based on the first entry in each line of each file, reformat the data and print it to standard output. By mixing Perl and shell tools, this job becomes a lot easier.

Tcl/Tk

Tcl is a simple scripting language designed as a command language which could easily be applied to various C programs for smooth configuration and user interaction. Tk is a language which grew out of Tcl in which graphical user interfaces can be constructed. One usually refers to them together as Tcl/Tk.

Tk has gained much popularity recently as an extremely easy way to construct graphical interfaces under X-Windows. If you have used **make xconfig** when compiling any of the recent (since 1.3.60) development kernels, you have used Tk. The program Tkined, a network management tool for Linux, uses Tk; it is based on Scotty, a Tcl extension offering various network functions such as access to SNMP data.

In accordance with its original design goals, Tcl allows you to interact with external processes in a fairly intuitive manner. Simple commands may be executed under Tcl with a simple **exec** command. For example:

```
exec ls | grep -v admin
```

returns exactly the same result as it did in the previous Perl example, but prints it to standard output, much like the **system()** command in C.

If you wish to interact with the output of a process or direct information to its input, you need to associate it with a filehandle, much as in Perl. This is done via the **open** command, as in:

```
set g0 [open |sort r+]
```

This opens the command **sort** for input. You would send data to the handle **g0** elsewhere in the program using **puts** and then read from the output using **gets**. The **r+** switch means that you can both write data to the process (data to be sorted) and read data from the process (sorted data). If you just wanted the data to be sent to standard output, you would use:

```
set g0 [open |sort w]
```

giving you write access to the process.

Wait, you say, this means that I can both read and write from a process? Yes, it does. Doesn't the Unix FAQ say this is a bad thing? Yes, it does. If you use this functionality to construct webs of interlocking, self-feeding processes, then you are *really* asking for trouble. Keep it simple if you are going to do this at all.

Expect

While it is potentially dangerous, people went so wild over this feature of Tcl that an extension to Tcl called Expect, a programming environment in its own right, was invented and has soared to new heights of popularity among certain users.

For example, ftp is a fairly simple program. You interact via a command line with a local program which then executes your commands. Because this uses the simple Unix STDIN/STDOUT method of interaction, you can write shell scripts to ftp files; I use such a script to retreive RFCs from the Internet automatically. However, a program like telnet is virtually impossible to script because you are not sending data to the program itself—you are sending data through a network connection to be interpreted on a remote machine. So, if you need to maintain a large number of routers, and if the only way to configure or check on these routers is via telnet, you are in trouble.

Expect solves this problem by using Unix's pseudo-tty mechanism. With Expect, you can script dialogues between your program and another one in which your program responds intelligently to the other. Think of a dialer program like dip or chat, except you can script dialogues with other programs instead of modems.

Expect is the height of inter-program communication, short of socket-based or sysV-ipc. (If you don't know, don't ask.) While it originally started as a Tcl extension, it has also been rendered into a C library; you can access its features from within C programs or from other environments which can use C libs, such as Perl.

Smooth Sailing, But Rocks Ahead

In the introduction to his book *Tcl and the Tk Toolkit*, John Ousterhout mentions that even though Tcl was originally designed to be a simple scripting language where all programs would have at least “some new C code”, the simplicity of the environment which they gave the programmer proved too enticing. “Most Tcl/Tk users never write any C code at all,” Ousterhout writes, “and most Tcl/Tk applications consist solely of Tcl scripts.”

This is either a good or a bad thing, depending on whether your criteria are ease-of-use or efficiency/power. Responding to the rise of Tcl in his typically understated manner, GNU Luminary and urban legend Richard Stallman posted a USENET article entitled “Why you should not use Tcl”:,

Tcl was not designed to be a serious programming language. It was designed to be a “scripting language”, on the assumption that a “scripting language” need not try to be a real programming language. So Tcl doesn't have the capabilities of one.

The ability to interact with other programs in new, unorthodox and some would say dangerous ways is what makes Tcl so appealing to some and so appalling to others. This is typical of the dilemma in using Unix tools from within non-shell programs.

Conclusion

It usually comes down to a matter of time. If you're trying to enter your code in the country fair, these techniques aren't going to win you a blue ribbon. If, however, you want to get it done by 7 PM so you can go to the fair, these might do the trick.

In an age of near-gigaflops-speed chips in home computers, a few wasted cycles here and there aren't going to kill anyone, especially for a program that will be run once or twice and then thrown away. Extending the shell philosophy to development work is also an attractive option—the speed with which you can hack together workable programs makes these techniques alluring to programmers on a tight deadline. Tcl/Tk is a perfect example of extending the shell philosophy to speed up development cycles. Of course, the inefficiencies of this approach are the cause of nearly all of the intense debate over the merits of Tcl/Tk.

Whether it be Tcl, shell, Perl, or C, no matter what your programming technique of choice might be, there is usually an option whereby tools from other programming environments can be imported for your use. If Richard Stallman

writes you a nasty letter criticizing you for it, though, don't say you weren't warned.

Todd Graham Lewis (tlewis@mindspring.com) has moved on to bigger and much better things with Mindspring Enterprises, the largest Internet Service Provider in the Southeastern US. There, he is learning a lot from his fellow engineers who have fancy “Computer Science” degrees. He wonders why everyone doesn't learn computing the same way he did—by playing with his Linux box.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Basic FVWM Configuration

John M. Fisk

Issue #28, August 1996

If you've recently set up fvwm and are using the default system.fvwmrc, you'll find that clicking the left mouse button anywhere in the root window brings up a pop-up menu. Not all of those entries will be valid for your system. Here's how to change them.

If you've recently set up fvwm and are using the default system.fvwmrc, you'll find that clicking the left mouse button anywhere in the root window brings up a pop-up menu. What you may also quickly discover, to your dismay, is that many of these program items don't *do* anything, either because the program doesn't exist or is incorrectly set up.

Because of this, you'll probably want to remove these menu entries. Also, sooner or later, you'll install programs that you'd like to add to the pop-up menu. Or you may decide that you want to reorganize the menu into categories, such as Editors, Graphics, Viewers, Network Apps, and so forth. Whatever the reason, configuring the pop-up menus is easy and a huge amount of fun. So let's look at how it's done.

Suppose you do a lot of text-editing or programming and have several editors you enjoy using. You now want to organize the pop-up menu by program category and want to put all your favorite editors under one pop-up sub-menu, called Editors.

For the sake of simplicity, we'll leave out a discussion of using command line options for things such as geometry, foreground and background colors, fonts, and so forth. We'll use fairly simple examples and assume that you can go back later and customize the command line options.

Be sure you've made a backup copy of your current *working* version of .fvwmrc. After that, load up your favorite editor and open the file used to define the pop-up menus, .fvwmrc. This will include entries such as:

```
Popup "Applications"
  Title "Applications"
  Exec  "Wingz"     exec Wingz &
  Exec  "Xmgr Plot" exec xmgr -g 780x730+362+3 &
  Exec  "Ghostview" exec ghostview &
  Exec  "Seyon"     exec seyon -modem /dev/modem &
  Exec  "SciLab"    exec scilab &
  Exec  "X3270"     exec x3270 &
  Exec  "Xfilemanager" exec xfilemanager &
  Exec  "Xfm"       exec xfm &
  Exec  "Xgrab"     exec xgrab &
  Exec  "Xxgdb"     exec xxgdb &
EndPopup
```

Your .fvwmrc will probably look a bit different than this, so find the section of the file that defines **Popup "*name*" ... EndPopup** stanzas. Without fully understanding how things work, you could probably use an entry such as the above as a template and modify the entries to include the programs you want. What you'll discover, however, is that this isn't hard to understand.

An important point to keep in mind is that you must define sub-menus first, before you define the main menu. The reason for this is actually quite simple: when fvwm starts it reads the .fvwmrc configuration file from beginning to end. If you define the main menu first, it encounters references to menu items (your sub-menus) that haven't been defined yet, and so it is unable to correctly set up the menus. Also, sub-menus can be nested to any depth. Once you have a list of editors to add to a sub-menu, and you've worked out the command line options you intend to use, you're ready to start.

The basic entry for a menu takes the form:

```
Popup "name"
  Title  "title"
  Exec  "program"  exec command &
  Exec  "program"  exec command &
  Exec  "program"  exec command &
  Nop   ""
  Exec  "program"  exec command &
  Exec  "program"  exec command &
EndPopup
```

Let's briefly look at each part of the entry. The entry begins with the word **Popup** and a **"*name*"** which is used to refer to this menu itself (we'll see how this is used in a minute when we talk about adding a sub-menu to the main menu). The next line, beginning with the word **Title**, specifies the title that appears at the top of the menu. Notice that the title is enclosed in double quotes. Next are a series of familiar **Exec** stanzas, each of which is used to launch a program. This time, however, the word following Exec and enclosed in double quotes is the name that will appear on the menu. A stanza for the xedit editor might look something like:

```
  Exec "XEdit" exec xedit -font 9x15 -g +5+20 &
```

The menu would then include an entry with the name XEdit: clicking on this would launch the xedit program with the font and geometry options that are specified on the command line. Don't forget the ampersand (**&**) at the end of the entry.

You'll also notice a line which begins with the word **Nop**, which, as its name suggests (to some people, anyway), performs “no operation”. It does, however, allow you to create separator lines between menu items. **Nop** followed by a pair of double quotes with **no spaces between them** (**""**) creates a separator line. This is very useful for visually separating a list of items in the menu. However, if **Nop** is followed by double quotes **with a space between them** (**" "**), an empty entry is created between menu items instead of a line. Try both and see the difference.

Finally, the reserved word **EndPopup** is used to indicate that the menu has been defined. Pretty simple, eh? Once you understand how menus are defined, you can easily use an existing menu definition as a “template” for creating one of your own.

One more quick point to mention: it is possible to launch fvwm modules from a pop-up menu. As with the **InitFunction** entries mentioned last month, these are really quite simple and use the same form:

```
Module  "name"  module
```

For example, to start up the FvwmPager, you would add something like the following:

```
Module  "Pager"  FvwmPager
```

Notice two things: the menu item name for the module can be anything you want—it doesn't have to be the same as the module name. Second, you do not put an ampersand (**&**) at the end of the command line.

Putting It All Together

So, now that we've touched on the basics, let's put all of this together and create a sub-menu for our editors. Supposing that we wanted the menu title to be **Editors** and the Popup itself to be referred to as **editors**, then we could create something similar to the example given below:

```
Popup "editors"
  Title "Editors"
  Exec  "XE&dit"      exec xedit &
  Exec  "X&Coral"     exec xcoral &
  Exec  "GNU &Emacs"  exec emacs -g 84x47 &
  Exec  "&XEmacs"     exec xemacs &
  Exec  "XW&PE"       exec xwpe -font 9x15 &
  Exec  "X&WE"        exec xwe -font 9x15 &
```

```
    Exec  "&aXe"        exec axe -noserver &
    Exec  "&NEdit"      exec nedit &
    Exec  "E&Z Editor"  exec ez &
  EndPopup
```

So far so good, eh. What's that? What are those ampersands doing in the menu item name entry? Fvwm allows you to define keyboard shortcuts to use with menus. Placing an ampersand in an item name causes the letter following the ampersand to be underlined. Then, when the menu is displayed, hitting that underlined letter causes the program item to be executed.

In the menu defined above, the letter “d” in XEdit would be underlined and would appear as “XEdit”. Once the menu has been displayed, hitting a “d” launches xedit. It goes without saying that you should avoid defining the same hot-key for two items in the name menu.

Ok, we're almost done. Now that we've created a sub-menu, let's add this to the main menu. An entry for a sub-menu takes the form:

```
    Popup  "Editors"  editors
```

Pretty easy, huh? The syntax should start to look pretty familiar to you by now. The line begins with the word “Popup” indicating the the item is a sub-menu of some kind. Following this, and enclosed in double quotes, is the item name that will appear on the menu. Finally, the last argument is the *name of the pop-up menu itself*. Remember that we decided to call the pop-up **editors**? This is the name by which the sub-menu is called. Be careful not to mix up the name of the pop-up menu with the title (e.g., “Editors”) that the menu uses.

Well, congratulations! You should now be well on your way to customizing and configuring. There are many more things that can be included on a pop-up menu although programs, modules, and sub-menus are probably the the ones you'll use the most. Once you get comfortable creating menu entries, skim over the fvwm manual page and take a look at the sample fvwmrc file that comes with the fvwm distribution to get ideas about what else can be done.

As a final word of exhortation let me suggest that “moderation in all things” is probably sage advice. It is unnecessary and unwise to create an entry for every program on your system. Add programs that you frequently use and make them accessible. Nesting sub-menus beyond one or two deep is likely to make getting at them more of a chore than it is worth. Also, more than 15 to 20 items on a single menu will probably make it a bit unwieldy. [Actually, experts in human-computer interaction suggest that the human mind is less efficient when dealing with more than 7 items (or groups of items) together. —ED] Use your discretion and divide things up if you need to. Most of all, though, have fun!

Color Customization

By now you should start feeling pretty good about fvwm. You've learned the basics of creating a start up desktop and you've re-organized and customized your pop-up menus. This is pretty good, eh? One of the next items on the customization to-do list is invariably colors. Like most everything associated with fvwm, colors are extensively customizable. Doing this, however, can be just a bit tricky, not because it's all that difficult, but because several entries govern how colors are applied to various programs and windows. Once you track all of these down, and understand a few simple concepts about how colors are defined, the rest is play. At the outset, however, it is helpful to know something about how fvwm views windows (no, not **that** Windows...).

Fvwm recognizes and makes a distinction between a couple of different “types” of windows. These include the “selected” window—that which has the input focus—“unselected” windows—those which do not have the input focus—and “sticky” windows—those which “stick to the glass” as it were. It is possible to customize the color scheme for each type of window. Parenthetically, let me also point out that it is easy to change the color of the root window or to use a bitmap or pixmap image in the root window as the “wallpaper”. In Part 1 of this series we saw that the xsetroot program allows you to change the color of the root window (see the sample .xinitrc file). There are much more fun and entertaining ways to change the root window, but I'll leave that up to you for the moment (hint: **man xpmroot** and **man xv** should give you some ideas...).

So, back to customizing the various windows. Fvwm allows you to individually customize selected, unselected, and sticky windows as well as menus and the pager. These are set using the following reserved words:

```
StdForeColor
StdBackColor
StickyForeColor
StickyBackColor
HiForeColor
HiBackColor
MenuForeColor
MenuBackColor
MenuStippleColor
PagerForeColor
PagerBackColor
```

ForeColor stands for the foreground color and BackColor stands for the background color. This is quite typical of how colors are designated under X—using a foreground/background combination to set the color scheme. A brief explanation of each of these is as follows:

StdForeColor foreground color for menus and *non-selected* window titles

StdBackColor background color for menus and *non-selected* window frames

StickyForeColor foreground color for *non-selected sticky* window titles

StickyBackColor background color for *non-selected sticky* window frames

HiForeColor foreground color for *selected* window's title

HiBackColor background color for *selected* window frame

MenuForeColor foreground color for menus

MenuBackColor background color for menus

MenuStippleColor color for *shaded-out entries* in a menu

PagerForeColor foreground color for pager

PagerBackColor background color for pager

Setting up the color scheme you want is a matter of adding an entry such as:

```
StdForeColor     black
StdBackColor     wheat
```

This would set the foreground (text) color to black and the background color to wheat.

Armed with this new knowledge, you head off to customize colors, and find an entry that looks like:

```
StdBackColor     #8a4510
```

If you're having a little trouble closing your eyes and visualizing just what the color #8a4510 might look like... ...keep reading.

Xcolorsel to the Rescue!

The color designated by the entry #8a4510 (which, for the curious, happens to be SaddleBrown) is in hexadecimal notation. As of X11 Release 5, there are several means for specifying color: two commonly used formats are RGB color names (such as SaddleBrown) and RGB hexadecimal values (such as #8a4510). The acronym RGB stands for “Red, Green, Blue” and has to do with how colors are generated.

Recall from your school days that all colors can be produced by a combination of primary colors—cyan, magenta, and yellow. Technically speaking, these are the “subtractive” primary colors of paint; when you put them all together, they

subtract **all** the light and make “black” (it really turns out brown). It is also possible to create colors using a combination of red, green, and blue “additive” primary colors of light—when you put them all together, they add up to make white. Hence, the RGB designation indicates the amount each of red, green, and blue light which make up a color. To see what colors are available to you under X, you can view the file /usr/lib/X11/rgb.txt. This file contains a listing of all of the named colors on your system. This might contain a listing such as:

```
...
139  69  19             saddle brown
139  69  19             SaddleBrown
160  82  45             sienna
205 133  63             peru
222 184 135             burlywood
245 245 220             beige
245 222 179             wheat
...
```

Each line contains the color name and three columns of numbers which represent the relative contribution of red, green, and blue values based on a scale from 0-255—the range of numbers that can be stored in 8 bits, or one byte. Pretty clever, eh? For reference sake, white contains the maximum value of red, green, and blue and has a value of 255, 255, 255. Black is defined as 0, 0, 0. This still doesn't answer the question of what #8a4510 looks like, until you know a bit about hexadecimal.

The hexadecimal system uses a base 16 place order system with “digits” including the numbers 0-9 and the letters a-f (representing decimal values 10-16). Knowing that an RGB designation must have an entry for each of the base colors, you can quickly surmise that by breaking 8a4510 into three hexadecimal numbers and converting them to decimal, that you could find their value in the rgb.txt file above (which uses decimal values for the red, green, and blue values). Converting from hex to decimal by hand isn't very difficult, but here's an even easier way to do it:

Enter **bc** at the command line to start the bc online calculator, and enter:

```
ibase=16
```

This sets the input to base 16 (hexadecimal). Now, enter the numbers that you want to convert to decimal separated by a semicolon:

```
8A;45;10
```

You **must** use capital letters for hexadecimal input. The output should look something like:

```
138
69
16
```

Do ctrl-D or type **quit** to exit bc.

Thus, the red value is 138, green is 69, and blue is 16. Going back to the sample rgb.txt entry above, we can see that this is very close to the entry for SaddleBrown (the blue value in rgb.txt is 19 instead of 16) and this is, in fact, what our color turns out to be.

Now, if all of this seems needlessly complex, rest assured that there are easier ways of viewing and handling colors under X. There are a few must-have configuration utilities that make using and customizing X-Windows a lot easier, and a color viewer such as xcolorsel is one of these. You should be able to find a copy of this very useful program at sunsite.unc.edu (or preferably one of the mirror sites) in the /pub/Linux/X11/xutils/colors/ directory. This very handy program:

1. Shows a color patch and the rgb.txt entry for each color.

2. Displays the color entry in any of 16 different formats, including the hexadecimal notation we've just looked at.

3. Lets the user “grab” a color off the desktop and displays the rgb entry that most closely matches it.

4. Lets the user “preview” what a certain foreground/background color combination might look like using its **Set foreground** and **Set background** features.

When you start up xcolorsel, you are initially presented with a display window containing color patches, their corresponding rgb values in decimal notation, and the color name—just as they appear in the rgb.txt file. Clicking on the **Display format** button presents you with a menu of different formats—choosing **8 bit truncated rgb** from this menu lets you view SaddleBrown with its rgb hex value of #8a4510! So, now you can easily see what each color name looks like and, if you're curious, what the hexadecimal notation would be. Thus, the following entries are equivalent:

```
StdForeColor    SaddleBrown
StdForeColor    saddle brown
StdForeColor    #8a4510
```

Keep in mind that hexadecimal notation requires the pound, or hash sign (**#**) prefix.

Use the **Grab color** feature to quickly find out what a color's rgb entry is. To do this click on the **Grab color** button—the cursor will change to a small magnifying glass. Then, position the cursor over any color on your desktop and

click once; xcolorsel will display along the bottom status line how many matches or near-matches there are and will highlight the closest entry in the color display window.

Finally, if you want to see what a particular foreground/background color combination might look like, try this: using the mouse pointer, highlight a color you want to use as either the foreground or background color and then hit either the **Set foreground** or **Set background** buttons at the bottom. The foreground or background colors of the color display window will then be changed to this value. Hitting the **Default colors** button reverts the window to its original color scheme.

Now that you've got a bit of a feel for how colors are defined, you can easily create your own customized “window treatments”. However, if you've made changes, started up fvwm, and things still aren't *quite* the way you'd expected, there still a couple more things you need to know about...

A Word about Styles

One powerful feature of fvwm is that it allows the user to define Styles for any or all applications. The idea is actually a fairly simple one: you can designate how an application window appears and several of its behaviors by setting up a style for it. This can include such things as whether it has a title bar, the size of the window border, whether it has resize handles, what icon it is associated with, and so forth. One such style option, as you might imagine, is color.

The syntax for a Style entry is actually quite simple and might look like:

```
Style "xterm"  Title, Handles, HandleWidth 7, Icon rxterm.xpm
```

That is, it begins with the word Style and is followed by the name of the program enclosed in double quotes—in this example, the xterm program. What follows is a comma-separated list of the various style options that you may wish to apply to the program.

Let's suppose you wanted to change the color of an application window to a simple black text on gray background. Simple enough, although it's important to make two points: first, the Styles color entry only sets the colors of the decorative window frames that fvwm puts around the program window—it doesn't change the colors of the application itself. Second, the colors are used when the window is *non-selected* (that is, it doesn't have the input focus). When the window is selected, the HiForeColor / HiBackColor combination set the color scheme. That said, to change the color scheme when the application window is non-selected you could add an entry such as:

```
Style "xterm"  Color black/gray, Title, Handles, Icon rxterm.xpm
```

The syntax is simply the reserved word Color followed by the *foreground* color name or hex number, a forward slash, and the *background* color name or hex number. You could also designate each color using the reserved words ForeColor and BackColor:

```
Style "xterm"  ForeColor black, BackColor gray, Icon rxterm.xpm
```

Either method will work.

One more quick point about modules and we're done! As previously mentioned, fvwm allows additional functionality to be added using modules such as FvwmPager or the GoodStuff modules. The foreground and background colors of the modules themselves (and not just the decorative window frames as we've been discussing up until this point) can be set using an entry such as:

```
*GoodStuffFore    black
*GoodStuffBack    turquoise
```

Configuration lines for modules must begin with the asterisk (*****) character, as seen in the example above. To specify the foreground color the module name is given with the **Fore** suffix. The background color designation uses the **Back** suffix. In the example above you can see we've changed the color combination to black text on a turquoise background. Again, you can use either the color name or the hexadecimal notation for specifying the color to use.

Well, that should get you going! Obviously, there is a **lot** more to color customization than the brief overview presented here. For the curious and adventurous, let me refer you to the manual pages for X and fvwm, and the excellent book *X-Windows System Administrator's Guide* (volume 8 in the X-Windows series) by O'Reilly & Associates publishing. Chapter 6 of this fine reference has a fuller discussion of color and the X-Windows system, including the X-Windows Color Management System (Xcms) that was implemented beginning with release 5. Enjoy!

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

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

LINUX JOURNAL

Mobile-IP: Transparent Host Migration on the Internet

Benjamin Lancki

Abhijit Dixit

Vipul Gupta

Issue #28, August 1996

The proliferation of powerful notebook computers and wireless communication promises to provide users with network access at any time and in any location.

Recent advances in hardware and communication technologies have introduced the era of mobile computing. The proliferation of powerful notebook computers and wireless communication promises to provide users with network access at any time and in any location. This continuous connectivity will allow users to be quickly notified of changing events and provide them with the resources necessary to respond to them even when in transit.

Unfortunately, present day Internetworking protocols like TCP/IP, IPX, and Appletalk behave awkwardly when dealing with *host* migration between networks.[footnote:In the Internet jargon, computers are often referred to as hosts.] Current versions of the Internet Protocol (IP) implicitly assume the point at which a computer attaches to the Internet is fixed, and its IP address identifies the network to which it is attached. Datagrams are sent to a computer based on the location information contained in its IP address.

If a mobile computer, or *mobile host*, moves to a new network while keeping its IP address unchanged, its address will not reflect the new point of attachment. Consequently, existing routing protocols will be unable to route datagrams to it correctly. In this situation, the mobile host must be reconfigured with a different IP address representative of its new location.

Not only is this process cumbersome for ordinary users, but it also presents the problem of informing potential correspondents of the new address.

Furthermore, changing the IP address will cause already-established transport layer connections (for example, ftp or telnet sessions) to be lost. Put simply, under the current Internet Protocol, if the mobile host moves without changing its address, it will lose routing; but if it does change its address, it will lose connections.

Mobile-IP is an enhancement to IP which allows a computer to roam freely on the Internet while still maintaining the same IP address. The Internet Engineering Task Force (IETF) is currently developing a Mobile-IP standard which, at the time of this writing, is in its sixteenth revision. The Mobile-IP architecture, as proposed by the IETF, defines special entities called the **Home Agent** (HA) and **Foreign Agent** (FA) which cooperate to allow a **Mobile Host** (MH) to move without changing its IP address. The term **mobility agent** is used to refer to a computer acting as either a Home Agent, Foreign Agent, or both. A network is described as having **mobility support** if it is equipped with a mobility agent.

Each Mobile Host is associated with a unique home network as indicated by its permanent IP address. Normal IP routing always delivers packets meant for the MH to this network. When an MH is away, a specially designated computer on this network—its Home Agent—is responsible for intercepting and forwarding its packets.

The MH uses a special registration protocol to keep its HA informed of its current location. Whenever an MH moves from its home network to a foreign network or from one foreign network to another, it chooses a Foreign Agent on the new network and uses it to forward a registration message to its HA.

After a successful registration, packets arriving for the MH on its home network are *encapsulated* by its HA and sent to its FA. Encapsulation refers to the process of enclosing the original datagram as data inside another datagram with a new IP header. This is similar to the post office affixing a new address label over an older label when forwarding mail for a recipient who has moved. The source and destination address fields in the outer header correspond to the HA and FA, respectively. This mechanism is also called *tunneling*, since intermediate routers remain oblivious of the original inner-IP header. In the absence of this encapsulation, intermediate routers will simply return packets to the home network. On receiving the encapsulated datagram, the FA strips off the outer header and delivers the newly exposed datagram to the appropriate visiting MH on its local network.

Host movements typically cause some datagrams to be lost while routing tables at the HA and FA re-adjust to reflect the move. However, by using retransmissions and acknowledgments, connections maintained by the

transport layer protocol are able to survive these losses in the same way they survive losses due to congestion. Note that even when the MH is away, datagrams meant for it are always sent first to its home network, in many cases resulting in a non-optimal route.

Figures 1 and 2 show a mobility-supporting internetwork which serves as an illustrative example. It shows two mobility- supporting networks, **Network A** and **Network B**, which are equipped with mobility agents **MA1** and **MA2**, respectively. A mobile host, **MH1**, is also shown, whose home network is **Network A**. Whenever **MH1** is away, **MA1** acts as its home agent. When **MH1** visits **Network B**, **MA2** acts as its foreign agent.

It is worth pointing out that changes introduced by Mobile-IP are independent of the communication medium in use. Even though this figure shows mobility support in a wired internetwork, the Mobile-IP works just as effectively in a wireless environment.

Figure 3 further illustrates the main idea behind Mobile-IP. It shows an IP datagram as it flows from computer A (IP address 18.23.0.15) to the mobile host (IP address 128.226.3.30). In this figure, the mobile host is shown to be away from its home network. Hosts MA1 (IP address 128.226.3.28) and MA2 (IP address 128.6.5.1) are acting as its home agent and foreign agent, respectively.

The IP header in the datagram, as it leaves A, indicates 128.226.3.30 as the destination. In Figure 3, this header is shown as the black portion of the datagram. Therefore, this datagram is routed to Network A (steps 1 and 2). Here, the home agent picks up the datagram and inserts an additional IP header before re-injecting it into the network (steps 3 and 4). The new IP header carries 128.6.5.1 as its destination address. This header is shown with cross hatched lines in Figure 3. As this is the header seen by intermediate routers like R1, the datagram is correctly routed to the foreign agent (step 5). By this time, the registration process has already informed the foreign agent of the mobile host's presence on the local net. When the encapsulated datagram arrives at MA2, the outer header is stripped. The newly exposed header reveals the MH as the destination and the datagram is forwarded appropriately (step 6).

The IETF Mobile-IP draft also allows a Mobile Host to do its own decapsulation. In this case, the MH must acquire a temporary IP address on the foreign network (e.g., using DHCP) to be used for forwarding. This allows a mobile host to receive datagrams away from its home network even in the absence of a Foreign Agent. The downside of this approach is the kernel on the MH must now be modified to handle encapsulated datagrams.

The steady increase in the sales of portable computers is indicative of a growing base of mobile users. IETF's proposed Mobile-IP standard will facilitate inter-operation between mobile devices designed by different vendors and further contribute to the popularization of mobile computing. Our research group at the State University of New York at Binghamton has developed a Mobile-IP implementation for Linux. This software and related documentation can be downloaded from the Linux Mobile-IP home page at http://anchor.cs.binghamton.edu/~mobileip/. The page also contains links to other Linux and portable computing resources.

All three authors are affiliated with the Department of Computer Science at the State University of New York, Binghamton. They can be reached at mobileip@anchor.cs.binghamton.edu.

Benjamin Lancki is an undergraduate student completing his senior year of study. His interests include mobile networking, multimedia software design, and pencil sketching.

Abhijit Dixit is a graduate student working towards a Masters degree. His interests include mobile networking and operating systems.

Vipul Gupta is an Assistant Professor whose interests include parallel processing and computer networks.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

LINUX JOURNAL

Graphing with Gnuplot and Xmgr

Andy Vaught

Issue #28, August 1996

If you need to graph data, there are two packages available for Linux under X: Gnuplot and Xmgr.

Graphing data is one of the oldest uses for a computer, dating back to FORTRAN programs producing character graphics on line-printers. Fortunately, things have advanced somewhat, and modern computers are capable of producing much nicer graphs. Several graphing packages are available for Linux under X and SVGALIB. Two of the most prominent packages are gnuplot and xmgr (a.k.a. ACE/gr). Xmgr is oriented towards graphing and manipulation of externally produced data sets, while gnuplot is used more for plotting data and mathematical functions.

Gnuplot's primary authors are Thomas Williams and Colin Kelley, with many others contributing. Although gnuplot was written independently of the Free Software Foundation, the FSF does distribute it. Gnuplot was written with portability in mind, supporting about four dozen output devices and formats under a dozen operating systems. Under Linux, it will run under both X and SVGALIB. Modifying gnuplot to support a new device involves writing a few device-dependent subroutines that are linked in with the main program.

Xmgr, on the other hand, is tied to X. Developed by Paul Turner, it also runs on many platforms besides Linux, but it outputs only PostScript. In the latter stages of development, Linux was the primary development platform. Development has recently been spread around to a loose organization of interested people.

Gnuplot

Gnuplot has a command-line interface with a mixture of emacs and Unix command line editing similar to the bash shell. Gnuplot may be run in batch mode, where the commands are taken from a file. The **plot** command causes a

plot to be sent to the currently selected device. In the case of the Linux svgalib driver, a graphics mode is selected and a graph is drawn in the current virtual console. When a key is hit, the display changes back to text mode for an additional command. Under X, a new window is created for the graph, while commands are entered in the original shell window.

Gnuplot has a comprehensive on-line help facility that can be accessed by typing **help**. The basic help command lists arguments of the help command by topic. Some subjects, like the **set** command, have many sub-topics. The documentation itself is well written and has many valuable examples of working commands.

A datafile containing points to plot is identified by the file name in single or double quotes. Each line has a two or more space-separated numbers that correspond to a point that is to be plotted. For example, suppose we had a file named “hits”:

```
# Monthly hits on our web site
1   13
2   23
3   66
4   75
5   74
6   82
7   377
8   442
9   512
10  756
11  874
12  946
```

The command **plot "hits"** would plot a graph of the data in the file named hits. Lines in a data file beginning with a **#** character are treated as comment lines. Blank lines are not treated as comments. Instead, they indicate where a line should not be drawn between a pair of points.

Although our example has the x data listed in the first column and the y in the second, gnuplot can handle cases where this is not so. The command:

```
plot "hits" using 2:1
```

would cause the x data to be read from the second column and the y data from the first column.

Plots can be embellished in many ways. Each comma-separated file or mathematical expression (shown later) to plot has two attributes that can be specified by the user: a title and a style. A gnuplot “title” is a string that is displayed with an example of the plot style that labels that data; this is usually called a “legend” by other programs. The style of the plot is selected from several possible ones, including “points”, which displays a symbol at each data

point, “lines”, which draws lines between the points, and “linespoints”, which draws both the lines and the symbols. The color of the line and symbol as well as the type of symbol (plus sign, cross, box) are normally assigned in series by gnuplot to make each distinct, but these can be overridden by the user.

For example, the command **plot "hits" title 'Hits on Website' with linespoints 3 4** plots our data file using lines of type 3 and points of type 4. At the top right will be the string **Hits on Website** next to a short example of type 3 lines and type 4 points. What you actually see depends on the output device being used—lines that are colored on a color display can come out dashed and dotted on monochrome devices (like most PostScript printers).

Our plot is looking better, but it is still not perfect. We want to put labels on the x and y axes to further clue the reader in on what we are looking at. Axis labels are setable parameters, as is the graph title:

```
set xlabel "Month"
set ylabel "Hits"
set title "Hits on the Website"
replot
```

Experimentation is easy to do in gnuplot by using the **replot** command, which repeats the previous plot command. Not only does this save keystrokes, but the author has a friend who likes to type **replot** repeatedly to display a file being appended to by another job he is running, which gives a running display of results as they are calculated.

Our graph is almost finished. Gnuplot's default algorithm for deciding where the x tick marks appear is showing only every other x point. We can make it show them all by:

```
set xtics 1, 1
replot
```

The first number causes the tick marks to start at x=1, and the second causes them to be spaced one unit apart. We could have included a third comma-separated parameter to indicate where the last tick mark should be plotted, but it is unnecessary in our example.

We can do better than month numbers:

```
set xtics ("Jan" 1, "Feb" 2, "Mar" 3, "Apr" 4,
    "May" 5, "Jun" 6, "Jul" 7, "Aug" 8, "Sep" 9,
    "Oct" 10, "Nov" 11, "Dec" 12)
replot
```

We have arrived at a graph worthy of being shown to the boss. The result is shown in Figure 1.

All that remains is to print it out. Gnuplot treats printers and plotters as just another output device. Executing the command:

```
set terminal PostScript
```

tells gnuplot to generate PostScript of the graph instead of console graphics. It is not enough to set the type of terminal. Typing **replot** now will cause gnuplot to spew PostScript to the user terminal. The command:

```
set output "graph.ps"
replot
```

will cause PostScript to be sent to the file graph.ps. If the first character of the filename is a vertical bar, gnuplot interprets the rest of the string as a program that will accept gnuplot's output as its standard input. So a command like:

```
set output "|lp"
replot
```

sends the output to the system's default printer.

Plots of mathematical functions are easy to produce: **plot 2*x** will produce a plot of a line with a slope of two on the default range of -10.0 to +10.0. The y-axis is automatically scaled by default so that all points are visible. For a mathematical function, the x range is taken from a default range. Multiple plots can be overlaid, with separate expressions separated by a comma.

A wide variety of common mathematical functions can be used in expressions —trigonometric, exponential and logarithmic as well as less common functions such as bessel functions and error functions. Expressions are based mostly on C-style expressions including the logical AND (**&&**) and OR (**||**) operators with the notable addition of the FORTRAN power operator (******).

Ranges are specified by a pair of numbers separated by a colon enclosed in square braces. Either or both numbers may be omitted to avoid affecting the current default. The first number specifies the range to begin and the second specifies the end. If we wanted to look at several graphs with the same range, the default range can be changed with the command **set xrange [1:2]**. If we wanted to change the range in only one plot, a range can be specified before the first function being plotted.

Advanced Gnuplot

Three dimensional surfaces can be generated with the **splot** command, which has syntax almost identical to plot. An additional range specifies the range of the y variable and the **set view** command lets the user control the orientation of the plot in space. A simple example would be:

```
splot x*x-y*y title "Hyperbolic Paraboloid"
```

Gnuplot also supports hiding lines that are behind other lines with the **hidden3d** parameter: **set hidden3d**.

Gnuplot can plot “parametric” functions. A parametric function is one in which both the x and y coordinates are functions of a third variable, which in Gnuplot is t. For example:

```
set parametric
plot 2*sin(t), 2*cos(t)
```

produces a circle of radius 2. The command:

```
set trange <range>
```

sets the values of t that are evaluated. Parametric plots are also valid while doing a three-dimensional splot. In this case, the independent variables under gnuplot are u and v.

Gnuplot can also take data points from the standard output of a Unix command specified on gnuplot's command line. This allows the display of points generated from almost any source. The command should be specified like a filename, preceded by a **<** character.

Xmgr

Xmgr is oriented more towards plotting data created from an external source, as opposed to plotting a given mathematical function. Xmgr normally reads files, but can also take input piped from its standard input. Once data has been read into an xmgr set, it can be displayed, scaled, and manipulated in many ways.

Xmgr also has an on-line help. When the “help” menu is selected, xmgr runs your favorite HTML browser (Mosaic by default) with the xmgr documentation as input. Several sites on the internet have this page on-line. If you don't have a browser, you end up having to read raw html. Having a program's documentation as a hypertext document is quite nice, as you can jump from subject to subject as well as being able to do text searches. A gallery of graphs produced by xmgr is also included with the xmgr distribution, which gives the user a visual look at the wide range of effects possible with xmgr.

The first step in graphing some data is to read the data into xmgr. The “Read Sets...” option under “File” produces a file browser from which a file can be selected. Several types of data can be read in, but the two column “XY” format is the most common. The format of the data is much the same as in gnuplot—

individual points on lines by themselves separated by spaces or tabs. Lines beginning with **#** are also considered comment lines, and lines without numeric data (like a blank line) separate sets. Lines beginning with the **@** symbol can control the actions of xmgr separately from the user.

Xmgr data sets are somewhat like registers, in that only a fixed number are available (fixed at compile time), and they are referred to by number. Once the data is in a set, it is displayed immediately. The left hand side of the xmgr window contains a number of buttons that provide shortcuts for various operations.

Most of the shortcut buttons let the user change the appearance of the graph interactively. A set of four arrow buttons scrolls the data in all four directions—tick marks and tick labels are automatically updated. The “Z” and “z” buttons allow uniform zooming in and out. Arbitrary zooms in are accomplished by using the magnifying glass button. This prompts the user for a rectangle that becomes the new limits of the graph. A text line at the top of xmgr's window constantly displays the current position of the mouse, in the coordinates of the graph. A crosshair extending the length and breadth of the window may be toggled to help position the mouse within a pixel of the desired point.

The “autoO” button provides an autoscaling feature. The cursor changes to a crosshair, which when clicked at some point selects the set nearest to the clicked point. The graph is rescaled such that all points in this set are visible. The “autoT” button immediately rescales the tick marks that can get cramped while doing a zoom.

Each data set has several attributes that control how the set is displayed—which symbol is used for points, the color of the symbol, whether the symbols are connected by lines or not, the color and style of the lines, the legend associated with the data set, and more. One rather packed menu controls all these options.

The user has a great deal of control over how the graph is displayed. Major and minor tick marks chosen by xmgr can be overridden. Simple box and line graphics as well as text strings can be drawn at arbitrary locations. All strings can be displayed in a variety of fonts and sizes, with subscripts, superscripts and some special characters available.

To repeat the earlier example using gnuplot, Figure 2 shows the xmgr display immediately after loading the hits file. Figure 3 shows the symbols and legends menu used to control the appearance of the set and the set's legend respectively, while Figure 4 shows the results. Figure 5 shows us getting ready to fix the X-axis by replacing the numbers with month names with the result in

Figure 6. Figure 7 is after the final touch-up, adding a title, giving the Y-axis a name, getting rid of the tenths digit in the tick labels and expanding the X-axis to fill the entire bottom of the graph. Figure 8 is the final PostScript output.

Advanced Xmgr

Once it is in an internal set, the data can be manipulated in many ways. Sets may be edited, deleted, and saved. Arbitrary mathematical functions can be typed in to transform one set to another. Regression, sometime referred to as "curve-fitting", can be done on a variety of standard curves—polynomial, logarithmic and exponential. Histograms can be created from sets with user-definable bin widths. Many other mathematical operations are supported. Individual data sets (as well as complete graphs) can be saved and loaded.

Xmgr also allows the user to define "regions" entered as polygons determined by mouse clicks. Data points within a region can be extracted from data sets into other data sets or removed from data sets. The regression options may also be set to operate only on the inside or outside of a particular region.

More Information

Gnuplot has its own Usenet newsgroup, comp.graphics.gnuplot. The current version is 3.5. Gnuplot can be downloaded from Gnu ftp sites like prep.ai.mit.edu and its mirrors.

The gnuplot 3.5 distribution comes with a tutorial written in LaTeX. The regular gnuplot documentation can be compiled into several different formats, one of which is the on-line help file. Other formats are a VMS .hlp file, a TeX document, nroff/troff format and an .rtf rich-text format. A man page is also provided, which talks about invocation options and X-resources that are used.

The current version of xmgr is 3.01. Xmgr has a home page located at www.teleport.com/~pturner/acegr/index.html. FAQs, on-line documentation, source and binaries are there. Other pages still have some dangling pointers to the old xmgr home page at ogi.edu, where the mailing list is still hosted.

Andy Vaught (ayndy@maxwell.la.asu.edu) is currently a graduate student in physics at Arizona State University and works part-time for Motorola. When not logged in, he enjoys bicycling, skiing and golf. He is also active in the civil air patrol.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Certifying Linux

Heiko Eissfeldt

Issue #28, August 1996

Certifying Linux to POSIX 1.1.

Standards

Part of the success of Linux is due to its commission to standards. One of the first standards for Unix-like operating systems was POSIX.1 (IEC/ISO 9945-1:1990 or IEEE Std. 1003.1-1990), which specifies the system services, the interface and system limits. It has been adopted by all major Unix vendors since its introduction. Higher levels like XPG4 from X/Open (a group of computer vendors) are upwardly compatible with POSIX.1. Finally, once an operating system is branded for Single Unix (or Spec 1170) it may be officially named Unix (TM) (a name which is controlled by X/Open).

Fortunately the design of Linux was aimed at POSIX.1, so nearly all necessary functionality had been implemented from the beginning; however, it needed testing.

Goal

Our primary goal at Unifix was a standard called Federal Information Processing Standard (FIPS) 151-2 from the National Institute of Standards and Technology (NIST), a U.S. Government institute. FIPS 151-2 requires some features that are optional in POSIX.1; thus, FIPS 151-2 includes POSIX.1 and more. We intended to get a certification for Linux on Intel platforms.

Where to Start?

Although usually linked to programming languages, ANSI-C (ISO/IEC 9899:1990) is a must for FIPS 151-2, and this was the first standard to meet. Rüdiger Helsch from Unifix began to clean up header files (namespace pollution issues) and fix the math library to ensure full ANSI-C conformance. Testing was done using our own tools.

FIPS 151-2 at Unifix

In Fall 1995 we acquired the test suite for FIPS 151-2 from NIST. The test procedures are defined in IEEE Std 1003.3-1991 and 2003.3-1992. The first differences were found when compiling the test programs. At a later stage the generated reports showed where tests had failed. In the following months we did a lot of kernel, libc and test program recompiles (more than 80 kernel compiles). Don't try that on a 386 SX with 4 megs! Most fixes had to be done in exit.c and in the termios package. After roughly 250 fixes in our system, and two fixes in the test programs, NISTs bin/verify reported no more non-compliant behaviour. We felt some pride at that point but were not finished yet. Rüdiger wrote the mandatory POSIX Conformance Document, where all system limits and characteristics are specified. Hint: there is an easy way to check for POSIX.1 compliance; a system without these docs is never compliant.

FIPS 151-2 in the Independent Testing Laboratory

Unifix is located in Braunschweig, Germany. and our independent testing laboratory is located in the U.S. So we had to transfer our modified Linux along with instructions for setting up a test PC to reproduce our test results. The lab did a completely new testing and is responsible for compliance afterwards. They were not allowed to use any pre-run test results, so everything had to be done from scratch. After some long-distance calls, all configuration mismatches had been ironed out (the very last problem was a suitable serial loopback cable), and the tests ran successfully. We entered the product at that point under the name Linux-FT and our newly founded company as Open Linux Ltd. (an X/Open member).

The Official Acknowledgement

To see that all went well, we e-mailed to POSIX@nist.gov with topic **send 151-2reg**. The mailrobot returned a list with all certified products, one of which was our system.

Was it worth it? It took considerable money and effort to get to this point. Our partner from the UK, Lasermoon, supported us financially and logistically. We are convinced we have gained much more stability and portability through the certification process. Signal handling improved considerably. A lot of small quirks and flaws scattered throughout the sources have been fixed. Most of those ugly #ifdef linux hacks in applications are disappearing. For application developers and porters these advantages are obvious. Linux-FT is now available and contains all source code (as ensured by GPL).

And Beyond?

Yes, we will do more certifications. POSIX.2 and XPG4 Base are the next stages, and finally the Single Unix branding. We are currently working on them and we hope our current product will enable us to reach XPG4 certification this summer. In the long term we intend our POSIX.1 changes to flow back into the mainstream kernels and libs (see the math lib, for example). The Linux 2.0 kernel sources will probably be run through our test suite before release.

Heiko Eifeldt (heiko@unifix.de) works at Unifix GmbH, Braunschweig, Germany.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

LINUX JOURNAL

Letters to the Editor

Michael K. Johnson

Issue #28, August 1996

Readers sound off.

SPARC?

I was a SPARC user until I encountered a hardware problem. I found it difficult to get service for the SPARC hardware; small site end users don't get much support from SUN, it seems.

The Linux/x86 world is just the opposite. There is a very competitive market offering a wide variety of options in all price/performance ranges. It creates a truly affordable Unix computing system.

I wish the Linux community would stay with x86 instead of making Linux just another piece of software on those expensive proprietary boxes.

—Siuki Chan siuki.chan@xilinx.com

Why Not?

When Unix was first written it was for a PDP-7 (and written in assembly language). Subsequent ports to additional hardware helped turn Unix into the machine-independent system it is today. Associated with the 25-year history of Unix and its machine-independence has been additional overhead.

Linux offers a fresh start. Porting Linux to other architectures can bring a new machine-independent operating system into general use. While many people today see the x86 as the right answer I am sure people felt the same way about the PDP-11 (which was the second platform for Unix). Designing portability into Linux now means it will be much easier to get running on the hardware of the future—whether it be based on the SPARC, DEC Alpha, MIPS or something totally new.

—Haykel Ben Jemia haykel@cs.tu-berlin.de

AWKward mistake

I would like to say that I really like *LJ* and that through it I learn something new about Linux every month. In issue 25 (May 1996), I found the article **Introduction to Gawk** very interesting because I often use Gawk. But when trying things out, I noticed something wrong with the output of the **FS** statement. The article stated that

```
{
    FS=":"
    print $5
 }
```

and

```
BEGIN {
    FS=":"
}
{
    print $5
}
```

are functionally identical, except that the first one is slower.

But when I executed the first program, the first line was blank. With the second program, everything was okay. So I asked on the Red Hat mailing list (because I use the Red Hat distribution) if someone could help me. Marc Ewing provided the real answer to the problem:

The line is split into fields before the rule is evaluated, so when the **FS=":"** is evaluated the first time, the line has already been split up, and either no fifth field exists, or in some situations the fields are wrong. So the two awk programs are not functionally identical; the first program is incorrect.

I hope this information will be useful for someone.

Oops

That was tested before being put in the magazine, but the bug was hard to notice (and was missed) because the /etc/passwd file on the machine used to test the script ran the output off the top of the screen. Thanks for bringing this to our attention.

lcc for ELF?

In Issue 25 of *Linux Journal*, the lcc compiler was reviewed [**Introduction to Awk**, by Ian Gordon] and the FTP site for lcc was listed as ftp://ftp.cs.princeton.edu/

pub/lcc/. However, I can only find a.out ports of lcc to Linux. Is anybody working on ELF support in lcc?

—Arthur D. Jerijian

Yes

The file ftp://ftp.cs.princeton.edu/pub/lcc/contrib/linux-elf.tar.gz is dated November 14th, 1995, so Linux ELF support for lcc has been around for a while.

More AWKward mistakes

To the editor:

I would like to comment on the article on gawk {**An Introcution to Awk**] by Ian Gordon in the May *Linux Journal*. Overall it is a nice introduction to the joys of awk programming, but I wish you had let me review it first.

There are a number of minor and not so minor errors in the article. In order of appearance:

1. Brian Kernighan wasn't one of the original designers of C; he “merely” wrote the book on it with Dennis Ritchie, who designed and implemented C. (Not to diminish his stature in any way; Brian is still a very important and seminal figure in the Unix and C world.)

2. The article says, “gawk also defines several special patterns wich do not match any input at all, the most commonly used being BEGIN and END”. This is incorrect. Only BEGIN and END are defined in awk, there are no others.

3. The statement “If you try to refer to fields beyond NF, their value will be NULL”, if read literally, is misleading. The value is the null or empty string, often denoted **""**. Granted, most programmer types would understand the statement at face value, maybe I'm just being pedantic.

4. There is a major error in the part that describes using a colon as the field separator.

```
        {
                FS = ":"
                print $5
        }
```

In gawk, field splitting occurs using the value of FS *at the time the record was read*. Thus, **$5** will already have been determined, based on the previous value of FS (presumably a space, **" "**). Unix versions of awk do this incorrectly, delaying field splitting until a field is needed, but doing so with whatever value of FS is

current. This is incorrect, and the POSIX standard for awk mandates that field splitting happen the way gawk (and mawk, see below) do it. In fact, my book (cited in the RESOURCES sidebar, thanks!) describes this exact issue.

The correct way to get the desired behaviour is to set FS either using the -F option, or using an assignment inside the BEGIN block, as mentioned later.

5. Some typos: “does not contain a seven field” should be “seventh”, and “modifing” should be “modifying”. And a nit. Calling the Info file a “page” is misleading. When printed, the current documentation is over 330 pages...

6. When talking about variable initialization, the article says “... setting it to 0 for an integer or "" for an integer or a string, respectively.” Not quite. Variables are initialized to **0** for their numeric value and **""** for their string value. All numbers in awk are maintained internally as C double's. Numbers that look like integers are still stored as doubles. This can lead to confusion for C programmers:

```
x = 5 / 4       # x is now 1.25, not 1, no integer truncation
```

(I've been bitten by this one, myself!)

7. The discussion of the array “for” loop is incomplete.

```
for (i in theArray) print i
```

prints each *index* in theArray. To get both the indices and the corresponding values, you would need something like 8.

```
for (i in theArray) print i, theArray[i]
```

A word about implementation speed and comparisons to Perl. There are three freely available awk implementations: the Bell Labs version, gawk, and mawk. Gawk is much much faster than the Bell Labs version. Mawk (ftp from oxy.edu), by Michael Brennan, is a very nice implementation that is (generally) even faster than gawk. Although I haven't done any timings, I'm willing to bet that an awk program run with mawk will give a comparable Perl program a really good run for its money, every time. Gawk's advantages over mawk are its additional features, its ports to more systems, and its comprehensive documentation. Mawk's advantages are its speed and rock solidness.

9. In the resource block, the title of the gawk documentation from the FSF is now *The GNU Awk User's Guide*, with just myself listed as the author. Because the manual changed significantly (it's now about double the previous size), we changed the title, and I am listed as the author because of all the new and heavily revised material in the guide. The title page does give credit where

credit is due, saying “Based on *The Gawk Manual*, by Close, Robbins, Rubin and Stallman.”

Finally, I would like to point out that many Linux distributions apparently don't yet have the latest version, 3.0.0; this should be gotten from a GNU mirror. There are a large number of nifty new features and bug fixes over the previous version, as well as the revised manual.

Please accept this note as constructive comments on an otherwise enjoyable article, one that I wish I had had time to write...

Thanks,

—Arnold Robbins gawk maintainer and documenter arnold@gnu.ai.mit.edu

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Linux Version 2.0

Michael K. Johnson

Issue #28, August 1996

The long-awaited Linux 2.0 is about to be released. What's new? Plenty.

As I write, the final touches are being put on Linux version 2.0. By the time you read this, it probably will have been released, and vendors will be working on distributions which include it. Do you want to upgrade?

If you have hardware that was not supported by 1.2.12, or not supported well, you probably have good reason to upgrade to 2.0. But even if your hardware is supported perfectly well by Linux 1.2, you may still want to upgrade. Linux 2.0 is *fast*. Swapping is fast. Under X, windows appear more quickly. X runs more smoothly. Performance under heavy load is improved. After a few minutes of version 1.3.100 (almost 2.0...), I was not interested in running version 1.2.13 any more.

Whether you choose to upgrade by obtaining all the new pieces from the Internet, or by waiting for your distribution vendor to provide you with a new version, you probably do want to upgrade.

Getting Started

There is a small price to pay for upgrading: several system utilities need to be upgraded in order for 2.0 to work. The system will still run with the old versions, but some functions won't work; in particular, the format of the /proc filesystem has changed, and the **ps** utility has to be replaced. Also, in order to use PPP, a new **pppd** daemon needs to be installed. But unlike the upgrade from 1.0.9 to 1.2.13, when we had to publish an article on how to upgrade here in *Linux Journal*, Linux 2.0 comes with a files called Documentation/Changes that tells you exactly what you need to change and where to get the new versions.

That's just a taste of the new focus on documentation; the Documentation directory tree includes documentation on everything from hunting down and reporting bugs to Linux's support for SMP (**s**ymmetric **m**ulti-**p**rocessing: multiple CPUs in one machine, all sharing the same main memory). There is also an improved configuration system: it includes both text-based and X-based menuing systems, with a built-in help system to explain each configuration choice. Instead of **make config**, choose **make menuconfig** or **make xconfig**.

Improvements

I mentioned improved performance. This is due both to significant restructuring of memory management and significant improvements in task scheduling. These work together in several ways to improve both throughput (the total amount of work that can be done in a certain amount of time) and interactive response to s user's input (typing, mouse movements, etc.).

Automatic, on-demand loading and unloading of kernel modules is available, and almost every driver can be compiled as a loadable module, saving precious, unswappable kernel memory. See Auto-loading Kernel Modules elsewhere in this issue.

SCSI support has also been improved; wide SCSI support now includes all 15 possible devices on supported wide SCSI controllers and SCSI error handling has been improved. Improved SCSI drivers are available for the BusLogic MultiMaster series, the Adaptec 2940 series, and the NCR53c8xx series, among others.

Those of you with IDE have not been ignored—support for several new high-end IDE interfaces is available, and bug fixes for the buggy interfaces (RZ1000 and CMD640) have been incorporated. Support for IDE tape drives is now included.

Linux 2.0 includes SMP support for Intel MP-compliant systems. In addition, some SPARC SMP systems are supported by the experimental Linux/SPARC source included, as well as improved file locking, including a full implementation of mandatory file locking. The /proc filesystem has been expanded and far more system information is available in /proc. A “watchdog” has been developed especially for unattended systems. This includes software and optional hardware support for rebooting a hung system.

Networking Improvements

Native AppleTalk networking has been added to Linux's impressive list of supported networking protocols, and the existing protocols (especially TCP) were improved. Filesystem support for NCP (Novell) and SMB (MS Lan Manager,

etc.) network filesystems has been added. In addition, the NFS filesystem has been greatly improved in conjunction with the memory management improvements; it is now as fast or faster than most other implementations. It is also now possible to keep your root file system on an NFS server, which was made practical by those changes.

There is also expanded support for network devices: many ISDN cards, some frame relay cards, and general synchronous networking support to make it easier to add further synchronous network support. Support for several 100Mbps Ethernet cards, including the popular DEC tulip series and the 3COM 3C590 series, has been added. The improved PPP code now supports “BSD compression”.

Experimental Features

There are also (clearly marked!) features which are experimental in nature.

More multiple platforms support than ever is included in 2.0. While Linux/Alpha is now as stable as Linux/i86, experimental support for SPARCs, PowerPCs, Amigas, Ataris, M68K/VME and a few MIPS-based platforms is also included.

Experimental support for running Java applications as “native binaries” is available by making them look almost like shell scripts to Linux, running the proper interpreter automatically when you attempt to run the Java application.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

LINUX JOURNAL

Advanced search

Device Drivers Concluded

George V. Zezschwitz

Issue #28, August 1996

This is the last of five articles about character device drivers. In this final section, Georg deals with memory mapping devices, beginning with an overall descriptoin of Linux memory management concepts.

Though only a few drivers implement the memory mapping technique, it gives an interesting insight into the Linux system. I introduce memory management and its features, enabling us to play with the console, include memory mapping in drivers, and crash systems...

Address Spaces and Other Unreal Things

Since the days of the 80386, the Intel world has supported a technique called virtual addressing. Coming from the Z80 and 68000 world, my first thought about this was: “You can allocate more memory than you have as physical RAM, as some addresses will be associated with portions of your hard disk”.

To be more academic: Every address used by the program to access memory (no matter whether data or program code) will be *translated*--either into a physical address in the physical RAM or an exception, which is dealt with by the OS in order to give you the memory you required. Sometimes, however, the access to that location in virtual memory reveals that the program is out of order—in this case, the OS should cause a “real” exception (usually **SIGSEGV**, signal 11).

The smallest unit of address translation is the *page*, which is 4 kB on Intel architectures and 8 kB on Alpha (defined in **asm/page.h**).

When trying to understand the process of address resolution, you will enter a whole zoo of page table descriptors, segment descriptors, page table flags and different address spaces. For now, let's just say the *linear address* is what the program uses; using a *segment descriptor*, it is turned into a *logical address*,

which is then resolved to a *physical address* (or a fault) using the paging mechanism. The *Linux Kernel Hacker's Guide* spends 20 pages on a rather short description of all these beasties, and I see no chance of making a more succinct explanation.

For any understanding of the building, administration, and scope of pages when using Linux, and how the underlying technique—especially of the Intel family—works, you *have* to read the *Linux Kernel Hacker's Guide*. It is freely available by ftp from tsx-11.mit.edu in the /pub/linux/docs/LDP/ directory. Though the book is slightly old [that's a gentle understatement—ED], nothing has changed in the internals of the i386, and other processors looks similar (in particular, the Pentium is exactly like a 386).

Pages—More Than Just a Sheet of Memory

If you want to learn about page management, either you start reading the nice guide *now*, or you believe this short and abstract overview:

- Every process has a virtual address space implemented by several CPU registers which are changed during context switches (this is the zoo of selectors and page description pointers). By these registers, the CPU accesses all the memory segments it needs.
- Multiple levels of translation tables are used to translate the linear address given by the process to the physical address in RAM. The translation tables all reside in memory. They are automatically looked up by the CPU hardware, but they are built and updated by the OS. They are called *page descriptor tables*. In these tables there is one entry (i.e., a “page descriptor”) for every page in the process's address space—we're talking of the logical addresses, also called virtual addresses.

We concentrate now on a few main aspects of pages as seen *by the CPU*:

- A page may be “present” or not—depending on whether it is present in physical memory or not (if it has been swapped-out, or it is a page which has not yet been loaded). A flag in the page descriptor is used to indicate this status. Access to a non-present page is called a “major” page fault. The fault is handled in Linux by the function **do_no_page()**, in **mm/memory.c**. Linux counts page faults for each process in the field **maj_flt** in **struct task_struct**.
- A page may be write-protected—any attempt to write on the page will cause a fault (called “minor page fault”, handled in **do_wp_page()** and counted in the **min_flt** field of **struct task_struct**).

- A page belongs to the address space of one task or several of them; each such task holds a descriptor for the page. "Task" is what microprocessor technicians call a process.

Other important features of pages, as seen *by the OS*, are:

- If multiple processes use the same page of physical memory, they are said to "share" it. The processes hold separate page descriptors for shared page, and the entries can differ—for example, one process can have write permission on the page and another process may not.
- A page may be marked as *copy on write* (grep for **COW** in kernel sources). If, for example, a process forks, the child will share the data segments with the parent, but both will be write-protected: the pages are shared for reading. As soon as one program writes onto a page, the page is doubled and the writing program gets a new page; the other keeps its old one, with a decremented "share count". If the share count is already one, no copy is performed when a minor fault happens, and the page is just marked as writable. Copy-on-write minimizes memory usage.
- A page may be *locked* against being swapped out. All kernel modules and the kernel itself reside in locked pages. As you might remember from the last installment, pages which are used for DMA-transfers have to be protected against being swapped out.
- Page descriptors may also point to addresses not located in physical RAM, but rather the ROM of certain peripherals, RAM buffers for video cards etc., or PCI buffers. Traditionally, on Intel architectures, the range for the first two groups is from 640 kB to 1024 kB, and the range for the PCI buffers is above **high_memory** (the top of physical RAM, defined in **asm/pgtable.h**). The range from 640 KB to 1024 kB not used by Linux, and is tagged as *reserved* in the **mem_map** structure. They are the "384k reserved", appearing in the first kernel message after BogoMips calculation.

Virtual memory allows quite beautiful things like:

- **Demand-loading** a program instead of loading it totally into memory at startup: whenever you start a program, it gets its own virtual address space, which is just associated with some blocks on your filesystem and some space for variables, but the memory is allocated and loading is performed only when you really *access* the different portions of the program.
- **Swapping**, in case your memory gets tight. This means whenever Linux needs memory for itself or a program and unused memory gets tight, it will try to shrink the buffers for the file systems, try to "forget" pages already allocated for program code that is executed (they can be reloaded

from disk at any time anyway), or *swap* some pages containing user data to the swap partition of the hard disk.

- **Memory Protection**. Each process has its own address space and can't look at memory belonging to other processes.
- **Memory Mapping**: Just declare a portion or the whole of a file you have opened as a part of your memory, by means of a simple function call.

Memory Mapping Example

Here we are. The first assumption you should be able to make when thinking about mmaping (**M**emory **Mapping**; usually pronounced em-mapping) a character device driver is you have something like a numbered position and length of that device. Of course, you could count the nth byte in the stream of characters coming from your serial line, but the mmap paradigm applies much more easily to devices that have a well-defined beginning and end.

One character “device” that is used whenever you use svgalib or the server is /dev/mem: a device representing your *physical* memory. The server and svgalib use it to map the video buffer of your graphics adaptor to the user space of the server or the user process.

Once upon a time (am I that old?) people wrote games like Tetris to act on text consoles using BASIC. They tended to write directly into the video RAM rather than using the bloody slow means of BASIC commands. That was exactly like using mmapping.

Looking for a small example to play with **mmap()**, I wrote a small program called **nasty**. As you might know, Arabian writing is right to left. Though I suppose nobody will prefer this style with Latin letters, the following program gives you an idea of this style. Note that nasty *only* runs on Intel architectures with VGA.

If you ever run this program, run it as root (because you otherwise won't get access to /dev/mem), run it in text-mode (because you won't see anything when using X) and run it with a VGA or EGA (because the program uses addresses specific of such boards). You might see nothing. If so, try to scroll back a few lines (Ctrl-PageUp) to the beginning of your screen buffer.

```
/* nasty.c - flips right and left on the
 * VGA console --- "Arabic" display */
# include <stdio.h>
# include <string.h>
# include <sys/mman.h>
int main (int argc, char **argv) {
    FILE    *fh;
    short*  vid_addr, temp;
    int     x, y, ofs;
    fh = fopen ("/dev/mem", "r+");
    vid_addr = (short*) mmap (
```

```
        /* where to map to: don't mind */
        NULL,
        /* how many bytes ? */
        0x4000,
        /* want to read and write */
        PROT_READ | PROT_WRITE,
        /* no copy on write */
        MAP_SHARED,
        /* handle to /dev/mem */
        fileno (fh),
        /* hopefully the Text-buffer :-)*/
        0xB8000);
    if (vid_addr)
        for (y = 0; y < 100; y++)
            for (x = 0; x < 40; x++) {
                ofs = y*80;
                temp = vid_addr [ofs+x];
                vid_addr [ofs+x] =
                  vid_addr [ofs+79-x];
                vid_addr [ofs+79-x] = temp;
            }
    munmap ((caddr_t) vid_addr, 0x4000);
    fclose (fh);
    return 0;
}
```

Playing with mmap()

What could you change in the **mmap()** call above?

You might change the rights for the mapped pages by removing one of the **PROT** flags asking for the right to read, write or execute (**PROT_READ**, **PROT_WRITE**, and **PROT_EXEC**) the data range mapped to the user program.

You might decide to replace **MAP_SHARED** by **MAP_PRIVATE**, allowing you to read the page without writing it (The Copy-on-Write Flag will be set: you will be able to write to the text buffer, but the modified content will not be flushed back to the display buffer and will go to your private copy of the pages.)

Changing the offset parameter would allow you to adapt this **nasty** program to Hercules Monochrome Adapters (by using 0xB0000 as text buffer instead of 0xB8000) or to crash a machine (by using another address).

You might decide to apply the **mmap()** call to a disk file instead of system memory, converting the contents of the file to our “Arabia” style (be sure to fit the length you mmap and access to the real file length). Don't worry if your old mmap man page tells you it is a BSD page—currently the question is who documents the features of Linux rather than who implements them...

Instead of passing **NULL** as first parameter, you might specify an address to which you wish to map the pages. Using recent Linux versions, this wish will be ignored, unless you add the **MAP_FIXED** flag. In this case Linux will un-map any previous mapping at that address and replace it with the desired mmap. If you use this (I don't know why you should), make sure your desired address fits a page boundary (**(addr & PAGE_MASK) == addr**).

At last, we have really hit one of the favorite uses of mmap—especially when you deal with small portions of large files like databases. You will find it helpful —and faster—to map the whole file to memory, in order to read and write it like it was real memory, leaving to the buffer algorithms of Linux all the oddities of caching. It will work much faster than **fread()** and **fwrite()**.

VMA and other Cyberspaces

The guy who has to care for this beautiful stuff is your poor device driver writer. While support for **mmap()** on files is done by the kernel (by each file system type, indeed), the mapping method for devices has to be directly supported by the drivers, by providing a suitable entry in the **fops** structure, which we first introduced in the March issue of *LJ*.

First, we have a look at one of the few "real" implementations for such a support, basing the discussion on the /dev/mem driver. Next, we go on with a particular implementation useful for frame grabbers, lab devices with DMA-support and probably other peripherals.

To begin with, whenever the user calls **mmap()**, the call will reach **do_mmap()**, defined in the mm/mmap.c file. **do_mmap()** does two important things:

- It checks the permissions for reading and writing the file handle against what was requested to **mmap()**. Moreover, tests for crossing the 4GB limit on Intel machines and other knock out-criteria are performed.
- If those are well, a **struct vm_area_struct** variable is generated for the new piece of virtual memory. Each task can own several of these structures, "virtual memory areas" (VMAs).

VMAs require some explanation: they represent the addresses, methods, permissions and flags of portions of the user address space. Your mmaped region will keep its own **vm_area_struct** entry in the task head. VMA structures are maintained by the kernel and ordered in balanced tree structures to achieve fast access.

The fields of VMAs are defined in linux/mm.h. The number and content might be explored by looking at /proc/*pid*/maps for any running process, where *pid* is the process ID of the requested process. Let's do so for our small **nasty** program, compiled with gcc-ELF. While the program runs, your /proc/*pid*/maps table will look somewhat like this (without the comments):

```
# /dev/sdb2: nasty css
08000000-08001000 rwxp 00000000 08:12 36890
# /dev/sdb2: nasty dss
08001000-08002000 rw-p 00000000 08:12 36890
# bss for nasty
08002000-08008000 rwxp 00000000 00:00 0
```

```
# /dev/sda2: /lib/ld-linux.so.1.7.3 css
40000000-40005000 r-xp 00000000 08:02 38908
# /dev/sda2: /lib/ld-linux.so.1.7.3 dss
40005000-40006000 rw-p 00004000 08:02 38908
# bss for ld-linux.so
40006000-40007000 rw-p 00000000 00:00 0
# /dev/sda2: /lib/libc.so.5.2.18 css
40009000-4007f000 rwxp 00000000 08:02 38778
# /dev/sda2: /lib/libc.so.5.2.18 dss
4007f000-40084000 rw-p 00075000 08:02 38778
# bss for libc.so
40084000-400b6000 rw-p 00000000 00:00 0
# /dev/sda2: /dev/mem (our mmap)
400b6000-400c6000 rw-s 000b8000 08:02 32767
# the user stack
bfffe000-c0000000 rwxp fffff000 00:00 0
```

The first two fields on each line, separated by a dash, represent the address the data is mmaped to. The next field shows the permissions for those pages (**r** is for read, **w** is for write, **p** is for private, and **s** is for shared). The offset in the file mmaped from is given next, followed by the device and the inode number of the file. The device number represents a mounted (hard) disk (e.g., 03:01 is /dev/hda1, 08:01 is /dev/sda1). The easiest (and slow) way to figure out the file name for the given inode number is:

```
cd /mount/point
find . -inum inode-number -print
```

If you try to understand the lines and their comments, please notice that Linux separates data into “code storage segments” or **css**, sometimes called “text” segments; “data storage segments” or **dss**, containing initialized data structures; and “block storage segments” or **bss**, areas for variables that are allocated at execution time and initialized to zero. As no initial values for the variables in the **bss** have to be loaded from disk, the **bss** items in the list show no file device (“0” as a major number is **NODEV**). This shows another usage of **mmap**: you can pass **MAP_ANONYMOUS** for the file handle to request portions of free memory for your program. (In fact, some versions of malloc get their memory this way.)

Your Turn

When your device driver gets the call from **do_mmap()**, a VMA has already been created for the new mapping, but not yet inserted into the task's memory structure.

The device driver function should comply to this prototype:

```
int skel_mmap (struct inode *inode,
               struct file *file,
               struct vm_area_struct *vma)
```

vma->vm_start will contain the address in user space to be mapped to. **vma->vm_end** contains its end, the difference between these two elements represents the length argument in the original users call to **mmap()**. **vma-**

>vm_offset is the offset on the mmaped file, identical to the offset argument passed to the system call.

Let's explore how the /dev/mem driver performs the mapping. You find the code lines in drivers/char/mem.c in the function **mmap_mem()**. If you look for something complicated, you will be disappointed: it calls only **remap_page_range()**. If you want to understand what happens here, you *really* should read the 20 pages from the *Kernel Hacker's Guide*. In short, the page descriptors for the given process address space are generated and filled with links to the physical memory. Note, the VMA structure is for *Linux* memory management, whereas the page descriptors are directly interpreted by the *CPU* for address resolution.

If **remap_page_range()** returns zero, saying that no error has occurred, your mmap-handler should do the same. In this case, **do_mmap()** will return the address to which the pages were mapped . Any other return value is treated as an error.

A Concrete Driver

It will be difficult to give code lines for all possible applications of the mmap technique in the different character drivers. Our concrete example is of a laboratory device with its own RAM, CPU and, of course, analog to digital converters, digital to analog converters, digital inputs and outputs, and clocks (and bells and whistles).

The lab device we dealt with is able to sample steadily into its memory and report the status of its work when asked via the character channel, which is an ASCII stream-like channel. The command-based interaction is done via the character device driver we implemented and its read and write calls.

The actual mass transfer of data is done independently from that: by sending a command like **TOHOST** ***`interface address`***, ***`length`***, ***`host address`***, the lab device will raise an interrupt and tell the PC it wants to send a certain amount of data to a given address at the host by DMA. But where should we put that data? We decided not to mix up the clear character communication with the mass data transfer. Moreover, as the user could even upload its own commands to the device, we could make no assumptions about the ordering and the meaning of the data.

So we decided to hand full control to the user and allow him to request DMA-able memory portions mapped to the user address space, and check every DMA request coming from the lab device against the list of those areas. In other words, we implemented something like a **skel_malloc** and **skel_free** by

means of **ioctl()** commands and disallowed any transfer to any other region in order to keep the whole thing safe.

You might wonder why we did not use the direct **mmap()**. Mostly, because there is no equivalent **munmap**. Your driver will *not* be notified when the mapping to the open file is destroyed. Linux does it all by itself: it removes the vma structure, destroys the page descriptor tables and decreases the reference count for the shared pages.

As we have to allocate the DMA-able buffer by **kmalloc()**, we have to free it by **kfree()**. Linux won't allows us to do so when automatically unmapping the user reference, but without the user reference, we don't need the buffer any more. Therefore, we implemented a **skel_malloc()** which actually allocates the driver buffer and remaps it to the user space as well, and **skel_free()** which release that space and unmaps it (after checking if a DMA-transfer is running).

We could implement the remapping in the user library released with our device driver by the same means used by the **nasty** program above. But, for good reason, /dev/mem can be read and written only by root, and programs accessing the device driver should be able to run as normal user, too.

Two tricks are used in our driver. First, we modify the **mem_map** array telling Linux about the usage and permissions of our pages of physical memory. **mem_map** is an array of **mem_map_t** structures, and is used to keep information about all the physical memory.

For all allocated pages we set the **reserved** flag. This is a quick and dirty method, but it reaches its aim under all Linux versions (starting at least at 1.2.x): Linux keeps its hands off our pages! It considers them as if they were a video buffer, a ROM, or anything else it can't swap or release into free memory. The **mem_map** array itself uses this trick to protect itself from processes hungry for memory.

The second trick we use is quickly generating a pseudo file which looks something like an opened /dev/mem. We rebuild the **mmap_mem()** call from the /dev/mem driver, especially because it is not exported in the kernel symbol table and simply apply the same small call to **remap_page_range()**.

Moreover, DMA-buffers allocated by our **skel_malloc()** call are registered in lists in order to check whether a request for a DMA transfer is going to a valid memory area. The lists are also used to free the allocated buffers when the program closes the device without calling **skel_free()** beforehand. **dma_desc** is the type of those lists in the following lines, which show the code for the **ioctl**-wrapped **skel_malloc()** and **skel_free()**:

```
/* ============================================
 *
 * SKEL_MALLOC
 *
 * The user desires a(nother) dma-buffer, that
 * is allocated by kmalloc (GFP_DMA) (continuous
 * and in lower 16 MB).
 * The allocated buffer is mapped into
 * user-space by
 *  a) a pseudo-file as you would get it when
 *     opening /dev/mem
 *  b) the buffer-pages tagged as "reserved"
 *     in memmap
 *  c) calling the normal entry point for
 *     mmap-calls "do_mmap" with our pseudo-file
 *
 * 0 or <0 means an error occurred, otherwise
 * the user space address is returned.
 * This is the main basis of the Skel_Malloc
 * Library-Call
 */
 * ------------------------------
 * Ma's little helper replaces the mmap
 * file_operation for /dev/mem which is declared
 * static in Linux and has to be rebuilt by us.
 * But ain't that much work; we better drop more
 * comments before they exceed the code in length.
*/
static int skel_mmap_mem (struct inode * inode,
                   struct file * file,
                   struct vm_area_struct *vma) {
    if (remap_page_range(vma->vm_start,
                   vma->vm_offset,
                   vma->vm_end - vma->vm_start,
                   vma->vm_page_prot))
        return -EAGAIN;
    vma->vm_inode = NULL;
    return 0;
}
static unsigned long skel_malloc (struct file *file,
                            unsigned long size) {
    unsigned long    pAdr, uAdr;
    dma_desc         *dpi;
    skel_file_info   *fip;
    struct file_operations  fops;
    struct file      memfile;
    /* Our helpful pseudo-file only ... */
    fops.mmap = skel_mmap_mem;
    /* ... support mmap */
    memfile.f_op = &fops;
    /* and is read'n write */
    memfile.f_mode = 0x3;
    fip = (skel_file_info*)(file->private_data);
    if (!fip) return 0;
    dpi = kmalloc (sizeof(dma_desc), GFP_KERNEL);
    if (!dpi) return 0;
    PDEBUG ("skel: Size requested: %ld\n", size);
    if (size <= PAGE_SIZE/2)
        size = PAGE_SIZE-0x10;
    if (size > 0x1FFF0) return 0;
    pAdr = (unsigned long) kmalloc (size,
                           GFP_DMA | GFP_BUFFER);
    if (!pAdr) {
        printk ("skel: Trying to get %ld bytes"
                "buffer failed - no mem\n", size);
        kfree_s (dpi, sizeof (dma_desc));
        return 0;
    }
    for (uAdr = pAdr & PAGE_MASK;
         uAdr < pAdr+size;
         uAdr += PAGE_SIZE)
#if LINUX_VERSION_CODE < 0x01031D
        /* before 1.3.29 */
        mem_map [MAP_NR (uAdr)].reserved |=
                             MAP_PAGE_RESERVED;
#elseif LINUX_VERSION_CODE < 0x01033A
        /* before 1.3.58 */
```

```
        mem_map [MAP_NR (uAdr)].reserved = 1;
#else
        /* most recent versions */
        mem_map_reserve (MAP_NR (uAdr));
#endif
    uAdr = do_mmap (&memfile, 0,
            (size + ~PAGE_MASK) & PAGE_MASK,
            PROT_READ | PROT_WRITE | PROT_EXEC,
            MAP_SHARED, pAdr & PAGE_MASK);
    if ((signed long) uAdr <= 0) {
        printk ("skel: A pity - "
                "do_mmap returned %lX\n", uAdr);
        kfree_s (dpi, sizeof (dma_desc));
        kfree_s ((void*)pAdr, size);
        return uAdr;
    }
    PDEBUG ("skel: Mapped physical %lX to %lX\n",
            pAdr, uAdr);
    uAdr |= pAdr & ~PAGE_MASK;
    dpi->dma_adr = pAdr;
    dpi->user_adr = uAdr;
    dpi->dma_size= size;
    dpi->next = fip->dmabuf_info;
    fip->dmabuf_info = dpi;
    return uAdr;
}
/* =============================================
 *
 * SKEL_FREE
 *
 * Releases memory previously allocated by
 * skel_malloc
 */
static int skel_free (struct file *file,
                      unsigned long ptr) {
    dma_desc    *dpi, **dpil;
    skel_file_info  *fip;
    fip = (skel_file_info*)(file->private_data);
    if (!fip) return 0;
    dpil = &(fip-).>dmabuf_info);
    for (dpi = fip->dmabuf_info;
         dpi; dpi=dpi->next) {
        if (dpi->user_adr==ptr) break;
        dpil = &(dpi->next);
    }
    if (!dpi) return -EINVAL;
    PDEBUG ("skel: Releasing %lX bytes at %lX\n",
            dpi->dma_size, dpi->dma_adr);
    do_munmap (ptr & PAGE_MASK,
        (dpi->dma_size+(~PAGE_MASK)) & PAGE_MASK);
    ptr = dpi->dma_adr;
    do {
#if LINUX_VERSION_CODE < 0x01031D
        /* before 1.3.29 */
        mem_map [MAP_NR(ptr)] &= ~MAP_PAGE_RESERVED;
#elseif LINUX_VERSION_CODE < 0x01033A
        /* before 1.3.58 */
        mem_map [MAP_NR(ptr)].reserved = 0;
#else
        mem_map_unreserve (MAP_NR  (ptr));
#endif
        ptr += PAGE_SIZE;
    } while (ptr < dpi->dma_adr+dpi->dma_size);
    *dpil = dpi->next;
    kfree_s ((void*)dpi->dma_adr, dpi->dma_size);
    kfree_s (dpi, sizeof (dma_desc));
    return 0;
}
```

Some Final Words on PCI

Technology develops, but the ideas often remain the same. In the old ISA world, peripherals located their buffers at the “very high end of address space”--above

640 KB. Many PCI-cards now do the same, but nowadays, this is something more like the end of a 32-bit address space (like 0xF0100000).

If you want to access a buffer at these addresses, you have to use **vremap()** as defined in linux/mm.h to remap the same pages of this physical memory into your own virtual address space.

vremap() works a little bit like the **mmap()** user call in nasty, but it's much easier:

```
void * vremap (unsigned long offset,
               unsigned long size);
```

You just pass the start address of your buffer and its length. Remember, we always map *pages*; therefore **offset** and **size** have to be page length-aligned. If your buffer is smaller or does not start on a page boundary, map the whole page and try to avoid accessing invalid addresses.

I personally have not tried this, and I'm not sure if the tricks I described above on how to map buffers to user space work with PCI high memory buffers. If you want to give it a try, you definitely have to remove the “brute force” manipulation of the **mem_map** array, as **mem_map** is *only* for physical RAM. Try to replace the **kmalloc()** and **kfree()** stuff with the analogous **vremap()** calls and then perform a second remapping with **do_mmap()** to user space.

But as you might realize, we've come to an end of this series, and now it is up to you to boldly go where no Linuxer has gone before...

Good Luck!

George V. Zezschwitz is a 27-year old Linuxer who enjoys late-night hacking and hates deadlines.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Bandits on the Information Superhighway

Russell King

Issue #28, August 1996

Bandits is not just for new (and not so new) users frightened by the uncertainties of being on-line; it is also for those who are not yet connected, who have put off getting net access through fear of the unknown.

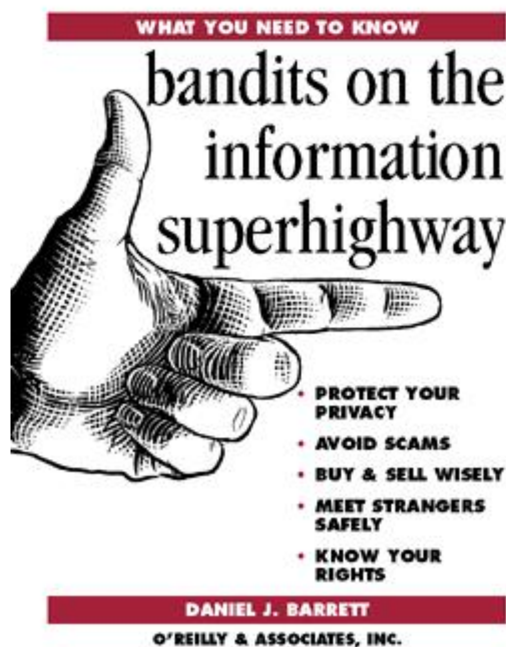
Author: Daniel J. Barrett

Publisher: O'Reilly & Associates

ISBN: 1-56592-156-9

Price: $17.95

Reviewer: Danny Yee



Bandits on the Information Superhighway is the best book yet in O'Reilly's "What You Need to Know" series, and perhaps an even more valuable contribution to making the Internet accessible than their classic *The Whole Internet*. It is a survey of all the bad things that can happen to you on-line: it explains what the dangers are and what you can do to minimise them. *Bandits* is not just for new (and not so new) users frightened by the uncertainties of being on-line; it is also for those who are not yet connected, who have put off getting net access through fear of the unknown.

The eleven chapters of *Bandits* (not counting the introduction) can be read independently of one another. The topics covered are privacy (mostly dealing with e-mail and news rather than computer security in general); get rich pyramid schemes; other common scams (advertisements dressed up as ordinary posts, students trying to get others to write their assignments, etc.); how to avoid paying money for free information; how to buy and sell safely; Usenet spams, April Fools' day jokes, urban legends, and junk e-mail; net relationships (particularly romances); looking after children (including some much needed deflation of media pornography myths); legal issues (what are your rights?); what to do if you *are* ripped off (where you can turn for help and when there isn't anything you can do); and what the future holds for the Internet.

The format of *Bandits*, like that of the other "What You Need to Know" books, is designed to be as friendly as possible: it has short personal anecdotes (including some from ordinary users) in the margin, separate boxes dealing with more specialised subjects, and only as much technical material as is absolutely necessary. But Barrett knows his stuff and the contributors include such respected Usenetters as Joel Furr and Brad Templeton: not once did I stop and think "Hey, that's not right" or "That's not the right way of putting that".

I do think a few improvements could be made to *Bandits*. It assumes in several places that users are connecting to a Unix server over an Ethernet (lots of concern about packet sniffers, and discussion of "finger" and "talk") rather than to an ISP using a modem. (Not only are people in the latter class now a majority of Internet users, but they are the ones who most need *Bandits*, since they are less likely to have a system administrator to turn to for advice or reassurance.) Though lots of URLs are provided as sources for further information, the focus is heavily on e-mail and Usenet and there is little discussion of the Web itself. (It would have been useful to explain, for example, that http://www.univ.edu/admin/ is more likely to be an "official" page than www.univ.edu/~bloggs/me.html.) And finally, there is nothing on purely intellectual banditry. (Serdar Argic, for example, was more than just a spammer: it was his complete reversal of the truth and his creative use of references which really made him dangerous.) Admittedly the ability to distinguish the respectable and objective

from propaganda and the lunatic fringe is hardly something one can hope to teach in a chapter, but it would have been nice to see a few guidelines.

Disclaimer: I requested and received a review copy of *Bandits on the Information Superhighway* from O'Reilly & Associates, but I have no stake, financial or otherwise, in its success.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

World Wide Web Journal, Issue One

Danny Yee

Issue #28, August 1996

The range of topics covered is immense.

Author: The World Wide Web Consortium

Publisher: O'Reilly & Associates

ISBN: 1-56592-169-0

Price: $39.95

Reviewer: Danny Yee

I usually only review periodicals after reading a year's worth of issues, but the first issue of *World Wide Web Journal* looks more like a book than a journal (and it has both an ISBN and an ISSN). Also, the journal is a quarterly, but the first issue consists of the proceedings of an annual conference (the Fourth International World Wide Web Conference, held in Boston in December 1995), so the next three issues may be rather different.

Issue one of the *World Wide Web Journal* contains fifty-nine papers, fifty-seven from the conference mentioned and two from regional conferences. The range of topics covered is immense. To list just a few (in no particular order): why the GIF and JPEG formats aren't good enough for really high quality graphics; low level security in Java; the results from the third WWW Survey; an analysis of Metacrawler use; caching systems; a filtering system to provide restricted access to the Web; a PGP/CCI system for Web security; the Millicent system for financial transactions involving small sums; smart tokens; and better support for real-time video and audio. There are also several papers on the use of the Web in education, on cooperative authoring tools, on Web interfaces to various database and software systems, and a whole pile of other things.

Though none of them assume specialized knowledge, the papers are mostly technical presentations of new ideas for systems and protocols: not everyone who runs a Web server or authors HTML will find them of interest. But anyone interested in the future of Web technology—either because they are involved in its development or out of curiosity—should find enough in the *World Wide Web Journal* to make it worth seeking out a copy.

Disclaimer: I requested and received a review copy of Issue One of the *World Wide Web Journal* from O'Reilly & Associates, but I have no stake, financial or otherwise, in its success.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Civilizing Cyberspace

Danny Yee

Issue #28, August 1996

Do we face a future where privacy is severely limited, where the divide between rich and poor has been widened by massive inequalities in access to information, and where free speech is just a memory?

Author: by Steven E. Miller

Publisher: Addison-Wesley

Reviewer: Danny Yee

Will the United States National Information Infrastructure (NII) be controlled by a few massive companies, vertically integrated to control production and distribution of information at all levels? Or will it resemble the current Internet, where individuals can provide as well as consume information? Do we face a future where privacy is severely limited, where the divide between rich and poor has been widened by massive inequalities in access to information, and where free speech is just a memory? Can we hope for increased accountability of governments and corporations, a more politically active population, and educational and economic benefits for everyone? These are the kind of questions that Steven Miller addresses in *Civilizing Cyberspace*.

Miller begins with a brief look at what the NII is and some of the different visions of what it should be. He then surveys the major players in the policy stakes—state, federal, and local governments, regulatory bodies, cable TV operators, local and long-distance telephone companies, the mass media, the computer industry—and the complex relationships between them. This includes such things as an overview of the various technologies that are likely to have a role in the NII and a brief history of United States telecommunications regulation.

After laying this groundwork, Miller presents the central issues in four chapters. The first looks at the idea of universal service, explaining what this means in the context of the NII and what the options for achieving it are. The second is about the implications of networking for politics, in particular the potential of universal access to public information, experiments with community free-nets and electronic democracy, and the importance of free speech. The third tackles the complex of issues centered around privacy, encryption and civil liberties. The fourth is about communities: the Internet community, virtual communities, and the use of networks in building communities. *Civilizing Cyberspace* closes with a brief look at the economic implications and possibilities of the NII (including intellectual property rights) and a summary of the practical actions, at all levels, that Miller sees as crucial.

The most impressive thing about Miller's book is that it avoids hype, overstatement, and polemic. Miller is neither a neo-Luddite doomsayer prophesying disaster, an optimist proffering a technological utopia, or someone so blinded by their political prejudices that they can't communicate with those who don't share them. He holds passionate beliefs about his subject—he is a strong supporter of public broadcasting, government regulation to avoid monopoly and other evils, civil liberties, government and corporate accountability, and grassroots democracy, among other things—but he is quite open about this and he understands that many do not agree with him on these issues. (He briefly discusses the differing political stances—libertarian, progressive/radical, liberal, corporate conservative, and state socialist—most commonly brought to bear on networking policy issues.) Even if you disagree with his normative suggestions, his book will still be a valuable source of information.

Civilizing Cyberspace is the best single volume introduction to the policy issues surrounding the Internet I have seen. Miller says he wrote it for information technology professionals and non-technical people “piqued by all the talk about the Information Superhighway”, but I think the most important audience for his work consists of the politicians and lobbyists actually involved in formulating policy. (Given the near-unanimous passing of the lunatic Communications Decency Act since *Civilizing Cyberspace* went to press, many of these obviously *need* to read it.)

While *Civilizing Cyberspace* is very United States specific, it does consider international issues in places and its overall message is very relevant in other countries. It is unlikely that anyone will write such a book specifically about the situation in Australia, for one thing, and if we can take heed of the developments in the United States which Miller describes then being a little behind in the development of legislation about computer networks may not be such a bad thing...

Disclaimer: I requested and received a review copy of *Civilizing Cyberspace* from Addison-Wesley, but I have no stake, financial or otherwise, in its success.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

New Products

Caldera Internet Office Suite

Caldera, Inc. announced Caldera Internet Office Suite, mainstream business applications with added Internet-aware functionality. The suite's native Linux applications include Corel's WordPerfect 6.0 for Unix, NCD Software's Z-mail e-mail package, XESS Software's NExS Spreadsheet, and Metrolink's Executive Motif Libraries. The suite sells for $329 with technical support available via email and telephone for a fee.

Contact: Caldera, Inc., 931 West Center Street, Orem, UT 84057 Phone: 801-229-1675 Fax: 801-229-1579 URL: http://www.caldera.com/.

Cosmos Shipping Linux on a Hard Disk

Cosmos Engineering Company is shipping a hard disk upgrade kit with a Linux and Xfree86 system pre-installed and ready to run. It is designed to be added to the PC's existing hard drive(s), which allows users to easily add a robust Linux system to their computers without disturbing their original work environment. It ships with kernel versions 1.2.13 and 1.3.94. Price: $279.

Contact: Cosmos Engineering Company, 5317 Venice Blvd, Los Angeles, CA 90019 Phone: 213-930-2540

TEAMate Web/BBS Server for Linux

TEAMate is a server product for Linux that combines both a Web and BBS interface. Users access a TEAMate server with the Web browser of their choice, a session-based GUI client for Windows or Mac, a VT100 terminal or by sending a mail message with an included query. Linux version Price: $495.

Contact: MMB Development Corporation, 904 Manhattan Avenue, Manhattan Beach, CA 90266. Phone 800-832-6022 or 310-318-1322. Fax 310-318-2162. E-mail info@mmb.com. URL: http://mmb.com.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Consultants Directory

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Ian T. Zimmerman
Linux consulting.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.